Formation à JAVA

Cours N°3
Fonctions avancées JAVA

Version 2.4 - Mai 2003

CENTRE DE FORMATION INDIVIDUALISE DU GITA
Lycée Technique Régional Diderot
61, rue David d'Angers – 75019 PARIS
tél. : 01.40.40.36.27/28— Fax : 01.40.40.36.30
Email : info@greta.gita.fr
http://www.gita.greta.fr
N° SIRET 197 507 122 00046 – code APE 804

SOMMAIRE

A. Accès aux flux et aux fichiers

A.I - Introduction

Les flux sont en matière de programmation, des objets chargés de contrôler l'envoi et la réception de données à travers un périphérique du système ou en dehors du système. La relation entre les flux et les entrées/sorties est très forte et le modèle objet est très fourni.

Les classes *InputStream* et *OutputStream* sont utilisées pour **la réception et l'envoi d'octets** alors que les classes *Reader* et *Writer* sont utilisées pour **les caractères (*ANSI*[1]).**

Quelque soit le type de flux, étant donné que la disponibilité d'un périphérique ne dépend pas du système exécutant le programme, la gestion des exceptions est indispensable et fait partie intégrante de la gestion des flux. En effet, si un flux vers une imprimante est envoyé, il faut anticiper sur la réaction de vos programmes si l'imprimante n'est pas connectée ou s'il lui manque du papier.

A.II - Les classes InputStream, OutputStream

1) Modèle objet

Ce sont des classes abstraites qui n'ont d'intérêt que si elle sont dérivées par des classes plus puissantes capable d'envoyer et de recevoir vers un périphérique donné. Par exemple, la classe *FileInputStream* sera capable de lire des octets dans un fichier.

Les classes descendantes de *InputStream* et *OutputStream* sont très nombreuses, ce qui permet de faire face à des situations très diverses , telles que :

- La lecture/écriture bufferisée grâce à *BufferedInputStream*
- La lecture/écriture de fichiers compressés zip, gzip grâce à *GZIPInputStream* ou *ZIPInputStream*.
- L'impression grâce à *PrintStream*
- …

Le modèle objet est donné dans la figure 1 de la page suivante.

[1] ANSI : Norme de codage des caractères sur 2 octets

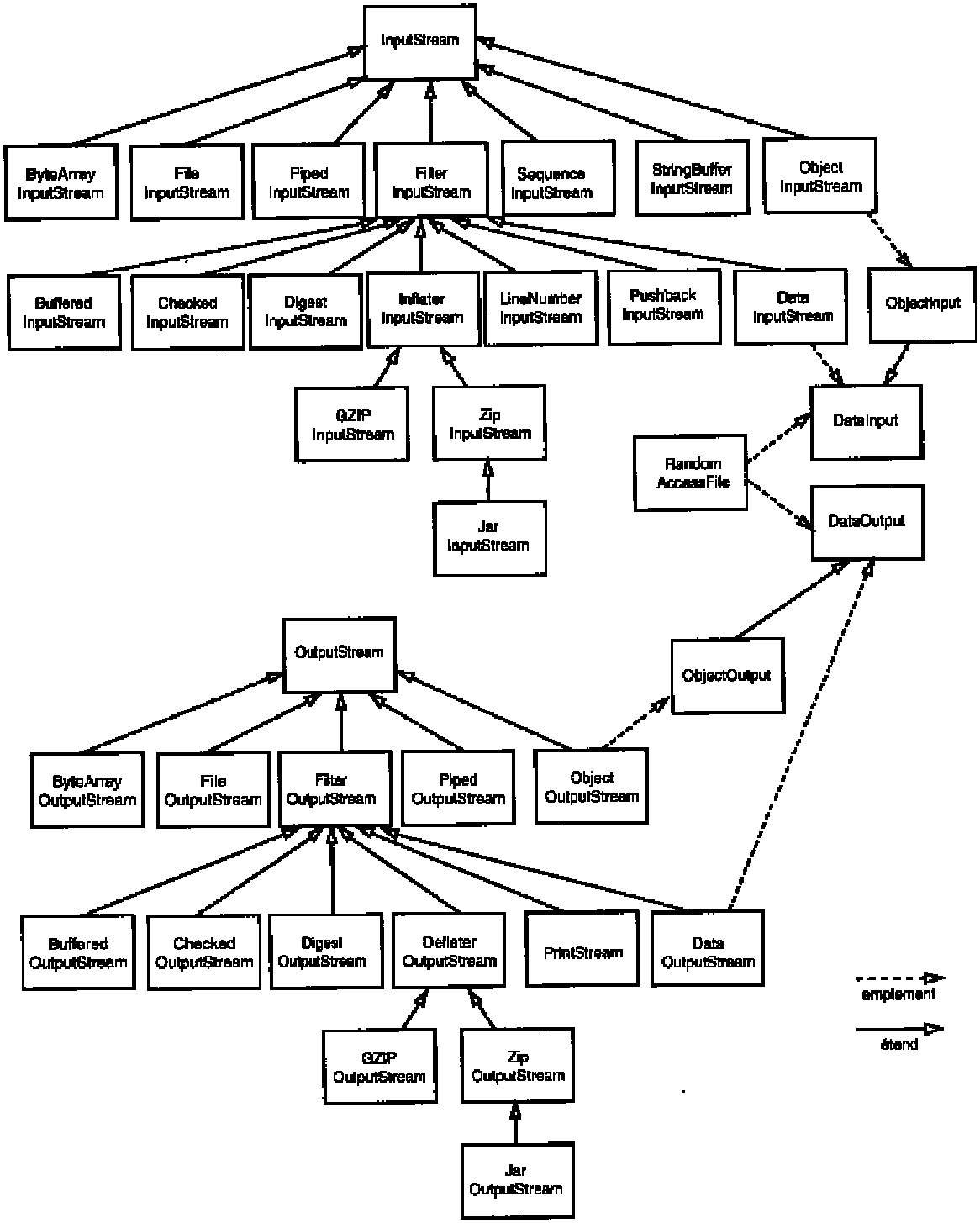


Fig 1 : Hiérarchie des flux d'octets

2) Ecriture et lecture

Les classes *OutputStream* et *InputStream* possèdent des méthodes abstraites qui peuvent être surchargées dans les classes enfant. Ces méthodes renvoient des exceptions qui devront être traitées par l'utilisateur.

Voici par exemple la signature de la méthode *read* de la classe *InputStream* :

```
1  public abstract int read() throws IOException
```

Voici un autre exemple de la méthode *write* de la classe *OutputStream* :

```
2  public abstract void write(int b) throws IOException
```

La lecture ou l'écriture peut également se faire par ensemble d'octets à l'aide des tableaux. Les méthodes *read* et *write* ne sont plus abstraites :

```
1  public void write( byte[] b) throws IOException
```

ou

```
1  public int read( byte [] b ) throws IOException
```

3) Lire/Ecrire dans des fichiers

La classe *FileInputStream* , dérivée de *InputStream* permet d'effectuer une lecture d'octets dans un fichier. Associée à un objet de type *File* , le flux correspond alors à un flux de fichier :

```
1  File f = new File("fiche.dat")
2  FileInputStream fl = new FileInputStream(f);
```

La lecture d'un octet se fait alors avec la méthode *read :*

```
3  byte b=fl.read();
```

4) Lire/Ecrire des données numériques

La classe *DataInputStream* permet de lire des données de type numériques tels que les flottants, les booléens, les octets ou les entiers (type *double, boolean, byte* et ***int***). Voici un aperçu **partiel** des méthodes de la classe *DataInputStream* disponibles :

Méthodes de *DataInputStream* (Non exhaustif)	
int	**read**(byte[] b) Lit l'ensemble des octets du flux et les stocke dans un tableau. Renvoie –1 si la fin du flux a été atte**int**e.
int	**read**(byte[] b, **int** off, **int** len) Lit le flux à partir de *off* sur *len* octets. Renvoie –1 si la fin du flux est atte**int**e
boolean	**readBoolean**() Renvoie un booléen égal à *true* si l'octet lu est non nul et *false* si l'octet est nul
byte	**readByte**() Renvoie l'octet lu.
char	**readChar**() Renvoie le caractère correspondant à l'octet lu
double	**readDouble**() Lit 8 octets et renvoie le double correspondant.

5) Chaînage des flux filtrés

Vous avez remarqué que *DataInputStream* ne sait lire que des types numériques mais que *FileInputStream* ne dispose pas des méthodes nécessaires pour lire des données de type numériques. La question est de savoir comment lire des données de type numériques dans des fichiers ?

La réponse est le chaînage des flux filtrés dont est le principe est d'utiliser plusieurs types de flux en enchaînant leur construction. Voilà un exemple de code permettant de lire des données numériques dans un fichier :

```
1  FileInputStream fis = new FileInputStream("fiche.dat");
2  DataInputStream dis = new DataInputStream(fis);
3
4  double s = dis.readDouble();
```

Cet enchaînement va vous permettre d'associer les différents types de flux de manière à bénéficier de leurs fonctionnalités. En effet, partons du principe que *DataInputStream* sait lire des numériques et *FileInputStream* sait lire dans des fichiers et associons les pour qu'ils nous fassent partager leurs compétences propres.

Dans notre exemple, si nous souhaitons qu'en plus de lire des données numériques dans des fichiers, nous souhaitions que cette lecture soit *bufferisée*[2] , il suffirait d'associer à nos 2 objets précédent, un 3ème issu de la classe *BufferedInputStream* . L'exemple suivant nous montre l'enchaînement de la construction :

```
1  DataInputStream dis = new DataInputStream
2           (new BufferedInputStream
3                ( new FileInputStream("fiche.dat") ));
```

L'objet de type *DataInputStream* est en haut de la chaîne de flux puisque ses méthodes d'accès aux données nous serons utiles pour lire des données de type numériques.

La méthode *read()* de l'objet *dis* va procéder à une lecture *bufferisée* dans le flux.

Cette syntaxe n'est pas très conviviale mais l'empilement des constructeurs est un passage obligé pour arriver à obtenir un flux avec de telles fonctionnalités.

Pour finir, voici un dernier exemple permettant de lire un booléen dans un fichier ZIP:

```
1  ZipInputStream zis = new ZipInputStream(
2                new FileInputStream("fiche.zip") );
3  DataInputStream dis = new DataInputStream(zis);
```

Pour lire un booléen dans ce fichier ZIP, on utilise :

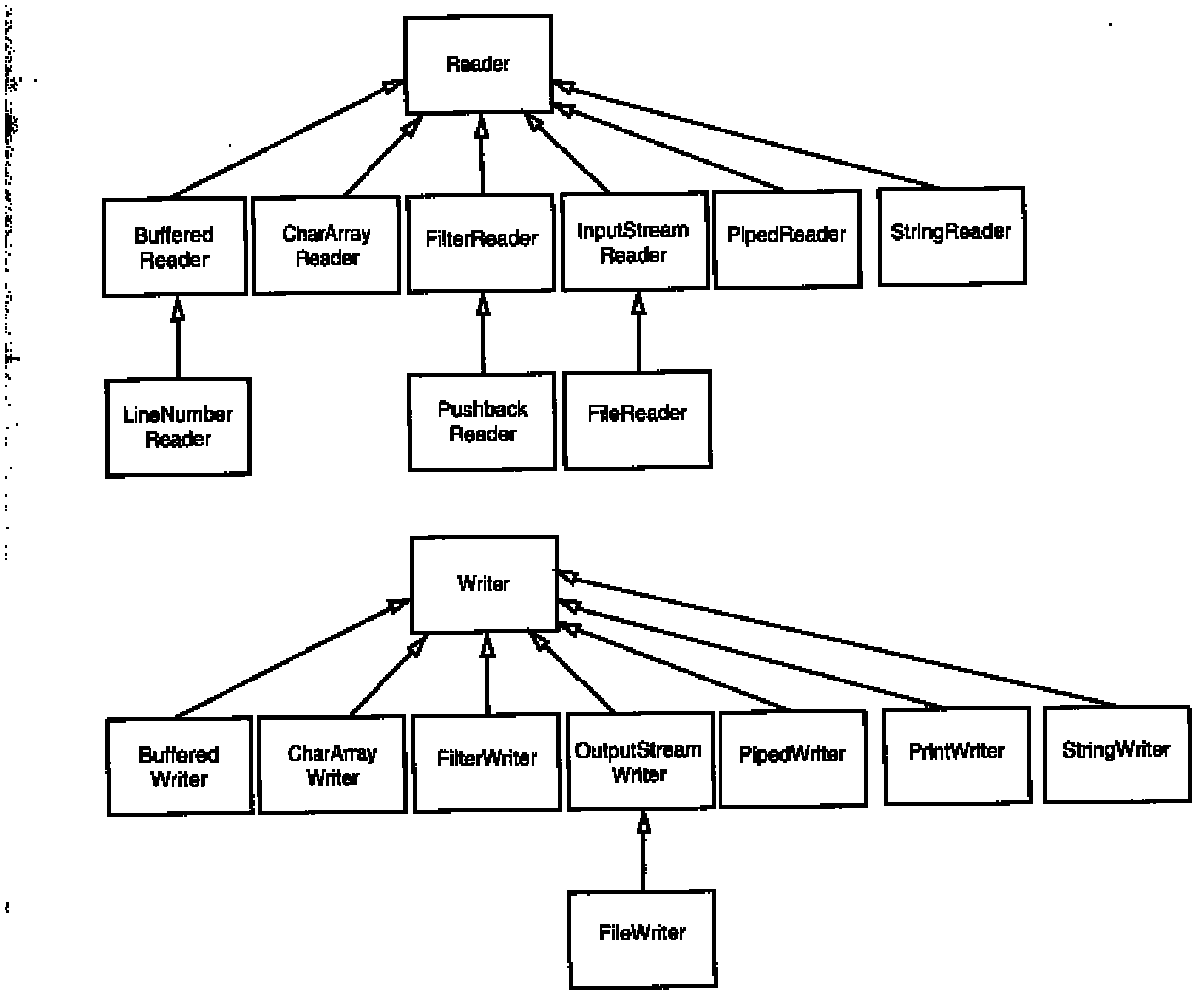
```
4  boolean b = dis.readBoolean();
```

[2] bufferiser : Action de lire un ensemble de données simultanément dans un flux et de les stocker temporairement jusqu'à leur traitement. L'avantage est que l'accession aux données ne s'effectue qu'une seule fois par opposition à une lecture non bufferisée où la lecture de chaque élément nécessite un accès dédié et coûteux en mémoire.

A.III - Les classes *Reader* et *Writer*

Nous avons vu l'écriture de fichiers binaires avec les classes *InputStream* et *OutputStream*. Il est temps de voir comment utiliser les classes *Reader* et *Writer* pour écrire des flux de caractères. Le système de flux filtrés va servir à créer des flux spécifiques.

Voici le modèle objet des classes *Reader* et *Writer* :



Hiérarchie des flux de caractères

1) Flux de caractères

Java propose de coder les caractères selon la norme *UNICODE*[3]. Chaque plate-forme d'exécution ayant une manière différente de coder les caractères , JAVA met tout le monde d'accord en proposant un mécanisme de flux filtrés. Ces classes, issues de *Reader* et de *Writer*, vont s'adapter à l'environnement d'exécution et effectuer un codage à base *d'UNICODE*. La classe *InputStreamReader* , par exemple, transforme un flux d'entrée en octets, en un flux émettant des caractères *UNICODE*.

[3] UNICODE : Système de codage sur 16 bits des caractères opposé à la norme ANSI sur 8 bits

2) Saisie de caractères au clavier

L'API Java propose une classe *System* permettant d'agir avec les propriétés du système hôte. Le champ statique *in* de type *InputStream* , permet de renvoyer des octets correspondant aux caractères tapés au clavier. Il peut être utile de transformer ce flux en un flux de caractères *UNICODE* par le truchement des flux filtrés :

```
1  InputStreamReader fluxUnicode = new InputStreamReader(System.in);
```

3) Types de codage de caractères

Chaque machine hôte peut avoir un jeu de caractères différent selon son pays d'origine. Sans parler pour l'instant d'internationalisation, il est possible de créer des objets de type *InputStreamReader* avec un jeu spécifique :

```
1  fluxUnicode = new InputStreamReader(System.in , "8859_5");
```

La chaîne de caractères passé en argument du constructeur correspond au jeu de caractère *Latin/Cyrillique ISO*. Voici quelques exemples de jeu de caractères :

Jeux de caractères (Non exhaustif)	
8859_1	Latin-1 ISO (jeu par défaut sous Windows)
Cp865	Langues nordiques pour PC
Cp949	Coréen pour PC
MacGreek	Grec pour Macintosh
Cp850	Latin-1 pour PC
Cp280	Italien pour IBM

4) Fichiers

Il existe un type particulier pour lire et écrire dans des fichiers. Il s'agit des classes *FileWriter* et *FileReader*. Ces sont en fait un objet de type *OutputStreamWriter* associé à un objet de type *FileOutputStream*. Les deux programmes suivant sont identiques :

```
1  FileWriter fw = new FileWriter("data.txt");
```

ou

```
1  OutputStreamWriter fw = new OutputStreamWriter(
2                                  new FileOutputStream("data.txt") ) ;
```

Les classes *FileWriter* et *FileReader* sont des classes d'aide permettant d'obtenir un flux de caractères facilement.

5) Flux de sortie de texte

La classe *PrintWriter* est très utile pour envoyer des caractères vers une destination quelconque (Un fichier, une imprimante). Les objets issus de cette classe sont associés avec un objet indiquant la destination :

```
1  PrintWriter pw = new PrintWriter(new FileWriter("data.txt"));
```

L'intérêt du code précédent est de pouvoir disposer de toutes les méthodes telles que *print, println* , tout comme l'objet *System.out*. Ces méthodes permettent d'envoyer tout type de données primitifs tels que *int, char, String* ..

Selon les systèmes hôtes, les caractères de fin de lignes varient (\r\n pour Windows , \n pour UNIX, \r pour Macintosh. Pour résoudre ce problème, JAVA propose un appel à la méthode *System.getProperty("line.separator")* **qui renvoie les caractères correspondant à la plate-forme d'exécution**

6) Flux d'entrée de texte

JAVA ne proposant pas l'équivalent de *PrintWriter* en entrée, il ne nous reste plus qu'à utiliser un objet de type *BufferedReader* associé à l'objet destination :

```
1  BufferedReader br = new BufferedReader(new FileReader("toto.txt"));
```

La méthode *readline* permet de lire ligne par ligne le flux d'entrée. Lorsque la fin du fichier est atteinte, cette méthode renvoie *null* :

```
2  String ligne;
3  while ( ( s= br.readLine()) != null)
4   {
5  … Traitement de lecture
6   }
```

Pour lire autre chose que des caractères, il faut convertir les caractères obtenus grâce aux méthodes de conversion des objets :

```
1  String s = br.readline();
2  double x = Double.parseDouble(s);
```

La méthode statique *parseDouble* permet de convertir une chaîne en un *Double*.

A.IV - Les exceptions

Bien entendu, comme la lecture ou l'écriture dans un flux n'est jamais sûre, les objets permettant l'accès à ces flux respectent le mécanisme des exceptions, au cas où la lecture ou l'écriture échouerait.

1) Lecture dans un fichier

L'exemple suivant montre comment lever une exception pour la lecture caractère par caractère dans fichier :

```
3  String fileString = "";
4  FileReader fr ;
5  try {
6            fr = new FileReader("data.txt");
7            int i=0;
8            while ( (i = fr.read()) != -1 )
9                      fileString += (char) i;
10           fr.close();
11
12           } catch ( FileNotFoundException fe ) {
13                System.out.println("Fichier introuvable");
14
15           } catch (IOException e) {
16                System.out.println("Erreur d'E/S");
17      }
```

La méthode *read* de l'objet *FileReader* renvoie –1 si la fin du fichier est atte**int**e. La chaîne f*ileString* contient alors l'ensemble du fichier. Vous remarquerez que la méthode *read* renvoie un entier qu'il faut convertir en caractère pour le traiter.

Les exceptions levées sont de type *IOException* et *FileNotFoundException*. Elles correspondent aux erreurs courantes se produisant au niveau des fichiers.

2) Ecriture dans un fichier

Voici le même exemple pour l'écriture :

```
18 FileWriter fw ;
19 String chaine = "Chaine a ecrire " ;
20 try {
21          fw = new FileWriter("data.txt");
22
23          for (int i=0 ; i< chaine.length() ; i++)
24               fw.write(chaine.substring(i,i+1));
25               fw.close();
26          } catch ( FileNotFoundException fe ) {
27               System.out.println("Fichier introuvable");
28               throw fe;
29
30          } catch (IOException e) {
31               System.out.println("Erreur d'E/S");
32     }
```

Vous noterez que les mêmes exceptions sont levées pour l'écriture.

Le répertoire utilisé pour lire et écrire dans des fichiers si aucun chemin n'est spécifié, est le répertoire courant correspondant au répertoire d'où le programme a été exécuté. Pour obtenir le chemin complet, vous pouvez faire appel à la méthode statique *getProperty("user.dir")* **de la** classe *System***. Attention cependant à cette méthode qui peut envoyer des exceptions si la propriété n'existe pas ou encore si la sécurité est activée et empêche le programme d'accéder à ces propriétés.**

La plupart du temps, la classe **chargée d'ouvrir les fichiers et la** classe **chargée d'informer l'utilisateur en cas d'échec n'étant pas la même, il devient indispensable de renvoyer les exceptions au code appelant :**

```
1  try {
2           fw = new FileWriter("data.txt");
3           int i=0;
4           for (int i=0 ; i< chaine.length() ; i++)
5                fw.write(chaine.substring(i,i+1));
6                fw.close();
7           } catch ( FileNotFoundException fe ) {
8                System.out.println("Fichier introuvable");
9                throw fe;
10          } catch (IOException e) {
11               System.out.println("Erreur d'E/S");
12               throw e;
13     }
```

A.V - Les fichiers ZIP

Les fichiers compressés sont très utilisés dans le monde de l'internet et JAVA , en tant que langage évolué se devait de couvrir cet aspect. JAVA reconnaît les format de type ZIP et GZIP (Normes RFC 1950 , RFC 1951 et RFC 1952)[4].

Le format ZIP le plus couramment utilisé sera traité ici, sachant que le format GZIP fonctionne de manière semblable.

Les classes utilisées pour traiter les fichier ZIP sont situées dans le package *java.util.zip*. N'oubliez d'ajouter ce package grâce à une instruction *import* appropriée.

1) Principe de lecture

Un fichier ZIP possède un en-tête comprenant des informations sur le nom du fichier et la méthode de compression utilisée. La classe JAVA *ZipInputStream* associée à la classe *FileInputStream* nous permettra d'accéder à chaque entrée individuelle par l'intermédiaire de la classe *ZipEntry*.

La méthode *read* de *ZipInputStream* renvoie –1 dès que la fin de l'entrée est rencontrée. Lorsqu'une entrée à été lue, il est indispensable d'appeler la méthode *closeEntry* avant de passer à l'entrée suivante.

Voici un exemple de lecture dans un fichier zip :

```
1  ZipInputStream zin = new ZipInputStream (
2                                       new FileInputStream(zipname));
3  ZipEntry entry;
4
5  while ((entry=zin.getNextEntry()) != null)
6  {
7  .. Lecture du contenu de l'entrée
8  zin.closeEntry();
9  }
10 zin.Close();
```

Pour lire dans une entrée, vous éviterez de vous servir de la méthode *read* qui ne fourni aucun formatage. Les méthodes spécifiques d'un flux filtré seront plus apropriées. Par exemple, pour lire un fichier texte se trouvant dans un fichier ZIP :

```
1  BufferedReader in = new BufferedReader(
2                                    new InputStreamReader(zin));
3  String s;
4  while (( s= in.readline()) != null )
5   … Traitement de la ligne
```

[4] RFC : Request For Comment , groupes de travail précisant les normes de tous les protocoles et langages concernant l'informatique.

2) Exemple complet

L'exemple complet suivant permet d'ouvrir un fichier ZIP et d'afficher le contenu de chaque entrée dans la fenêtre.

```
1  /**
2   * @version 1.20 17 Aug 1998
3   * @author Cay Horstmann
4   */
5
6  import java.awt.*;
7  import java.awt.event.*;
8  import java.io.*;
9  import java.util.*;
10 import java.util.zip.*;
11 import javax.swing.*;
12 import javax.swing.filechooser.FileFilter;
13
14 public class ZipTest extends JFrame
15    implements ActionListener
16 {  public ZipTest()
17    {  setTitle("ZipTest");
18       setSize(300, 400);
19
20       JMenuBar mbar = new JMenuBar();
21       JMenu m = new JMenu("File");
22       openItem = new JMenuItem("Open");
23       openItem.addActionListener(this);
24       m.add(openItem);
25       exitItem = new JMenuItem("Exit");
26       exitItem.addActionListener(this);
27       m.add(exitItem);
28       mbar.add(m);
29
30       fileList.addActionListener(this);
31
32       Container contentPane = getContentPane();
33       contentPane.add(mbar, "North");
34       contentPane.add(fileList, "South");
35       contentPane.add(fileText, "Center");
36    }
37
38    public void actionPerformed(ActionEvent evt)
39    {  Object source = evt.getSource();
40       if (source == openItem)
41       {  JFileChooser chooser = new JFileChooser();
42          chooser.setCurrentDirectory(new File("."));
43          chooser.setFileFilter(new FileFilter()
44             {  public boolean accept(File f)
45                {  return f.getName().toLowerCase()
46                      .endsWith(".zip")
47                      || f.isDirectory();
48                }
49                public String getDescription()
50                { return "ZIP Files"; }
51             });
52          int r = chooser.showOpenDialog(this);
53          if (r == JFileChooser.APPROVE_OPTION)
54          {  zipname = chooser.getSelectedFile().getPath();
55             scanZipFile();
56          }
57       }
58       else if (source == exitItem) System.exit(0);
59       else if (source == fileList)
60          loadZipFile((String)fileList.getSelectedItem());
61    }
62
63    public void scanZipFile()
64    {  fileList.removeAllItems();
```

```
65        try
66        {  ZipInputStream zin = new ZipInputStream(new
67              FileInputStream(zipname));
68           ZipEntry entry;
69           while ((entry = zin.getNextEntry()) != null)
70           {  fileList.addItem(entry.getName());
71              zin.closeEntry();
72           }
73           zin.close();
74        }
75        catch(IOException e) {}
76     }
77
78     public void loadZipFile(String name)
79     {  try
80        {  ZipInputStream zin = new ZipInputStream(new
81              FileInputStream(zipname));
82           ZipEntry entry;
83           fileText.setText("");
84           while ((entry = zin.getNextEntry()) != null)
85           {  if (entry.getName().equals(name))
86              {  BufferedReader in = new BufferedReader(new
87                    InputStreamReader(zin));
88                 String s;
89                 while ((s = in.readLine()) != null)
90                    fileText.append(s + "\n");
91              }
92              zin.closeEntry();
93           }
94           zin.close();
95        }
96        catch(IOException e) {}
97     }
98
99     public static void main(String[] args)
100          {  Frame f = new ZipTest();
101             f.show();
102          }
103
104          private JComboBox fileList = new JComboBox();
105          private JTextArea fileText = new JTextArea();
106          private JMenuItem openItem;
107          private JMenuItem exitItem;
108          private String zipname;
109       }
```

- **63-76** : Parcours du fichier ZIP (Lecture) à partir d'un flux filtré composé d'un *ZipInputStream* et d'un *FileInputStream*.
- **78-97** : Lecture de l'entrée avec le même flux filtré mais pour chaque entrée, utilisation d'un autre flux filtré composé d'un *BufferedReader* et d'un *InputStreamReader*. La boucle s'arrête lorsque la fin du flux est atteinte et que la méthode *readLine* renvoie *null*.

3) Principe d'écriture

Pour écrire dans un fichier ZIP, l'utilisation d'un flux de type *ZipOutputStream* associé avec un flux de type *FileOutputStream* semble le plus approprié. Un objet de type *ZipEntry* sera crée pour chaque nouvelle entrée dans le fichier. Il suffit de passer au constructeur de *ZipEntry* le nom du fichier , qui déterminera la date de création et le mode de décompression par défaut. Il vous faudra ensuite appeler la méthode *putNextEntry* du flux *ZipOutputStream* pour commencer à écrire :

```
1  FileOutputStream fout = new FileOutputStream ("data.zip");
2  ZipOutputStream zout = new ZipOutputStream(fout);
```

et la boucle à effectuer pour chaque nouvelle entrée :

```
3  {
4  ZipEntry ze = new ZipEntry(filename);
5  zout.putNextEntry(ze);
6  …Envoyer les données à ze
7  zout.closeEntry();
8  }
```

B. Programmation multitâches avec les *Threads*

B.I - Introduction

1) Programmation multitâche

Tous les systèmes d'exploitation actuels ont la possibilité d'effectuer plusieurs tâches en simultané. Toute l'informatique d'aujourd'hui utilise ce principe essentiel à la réalisation de tâches complexes tel que le fenêtrage sous *Windows* par exemple.

En effet, lorsqu'un utilisateur consulte un site *Internet* , l'ordinateur effectue plusieurs tâches en même temps, comme gérer la communication Internet, analyser le contenu reçu, et gérer l'arrivée permanente de données.

Donc, chaque processus exécuté sur la machine est plus ou moins indépendant des autres.

On distingue deux types de programmes multitâches. Le multitâche dit *préemptif* et le multitâche dit *coopératif*. Le premier est un programme dont l'exécution est contrôlé par le système d'exploitation qui peut choisir de le stopper à tout moment sans attendre une autorisation de sa part. Le second peut être interrompu uniquement s'il l'autorise.

Windows NT (Windows 95 et 98 pour les programmes 32bits) un *OS*[5] préemptif ce qui lui confère une plus grande stabilité que les OS coopératifs. En effet, lorsqu'un programme mal conçu se plante pendant son exécution, un appui sur les touches *CTRL-ALT-SUPPR* suffit à récupérer la main pour stopper le programme incriminé.

2) Threads et processus

Il existe une différence entre les *threads* et les processus. Un processus est un programme s'exécutant de manière indépendante des autres processus. Il possède une copie unique de ses propres variables. Le thread, lui, partage les données avec les autres threads. Cela peut paraître dangereux dans un premier temps mais il s'avère que cette possibilité les rend plus rapides et plus faciles à gérer que les processus. En effet, il est bien plus rapide de créer et de détruire les *threads* individuels que de créer des processus.

Dans les G.U.I[6], la gestion des évènements s'exécute dans un *thread* différent de l'application, ce qui permet à ces applications de réagir instantanément ou presque aux actions de l'utilisateur, tout en continuant le traitement des données.

3) Le "*faux multitâche*"

La plupart des ordinateurs n'ayant qu'un seul processeur, le système utilise un mécanisme astucieux faisant *croire* à l'utilisateur qu'il réalise plusieurs choses en même temps. En effet, chaque *thread* dispose d'un *temps de parole* (temps minimal pendant lequel il va s'exécuter) après lequel il doit rendre la main au système pour laisser une chance aux autres *threads* de faire leur travail.

Voilà pourquoi on peut parler de "*faux multitâche*"

[5] OS : Operating System – Système d'exploitation
[6] GUI : Graphic User Interface – Interface graphique utilisateur

B.II - Utiliser les *threads*

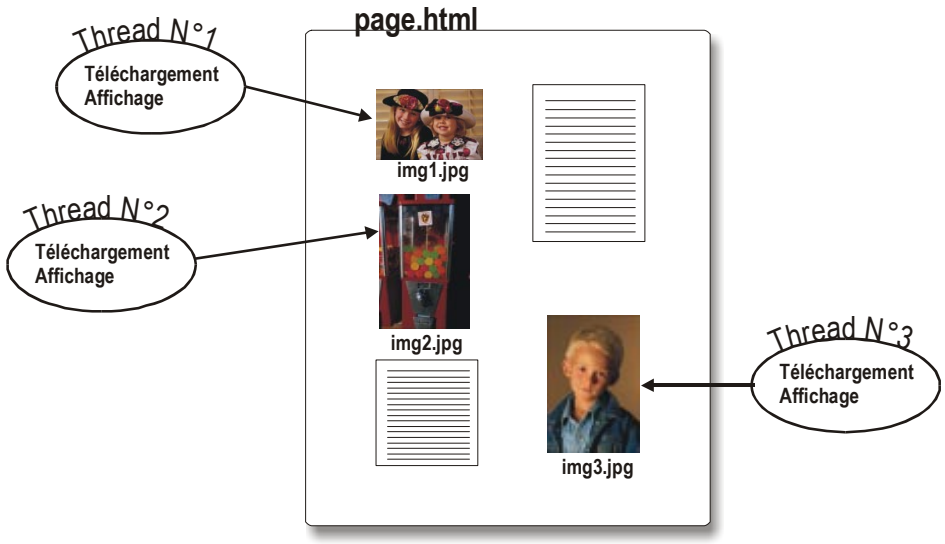
1) Qu'est-ce qu'un *thread*

Il s'agit d'un programme s'exécutant en même temps que d'autres programmes. Il est capable de se mettre en sommeil pour permettre aux autres *threads* d'avoir une chance d'être exécuté. La période de sommeil est fixée par lui et le programmeur aura à charge de donner une valeur raisonnable de ce temps, pour permettre aux autres *threads* de pouvoir s'exécuter.

Dans un environnement multitâche préemptif, même si le *thread* a un temps de sommeil fixé par avance, cela n'empêchera l'OS d'interrompre le programme quand bon lui semblera pour effectuer d'autres tâches. Sur un ordinateur dont la performance est en adéquation avec les exigences minimales du système d'exploitation, le programme multi-thread fonctionnera de manière transparente pour l'utilisateur

2) Un exemple de thread

Le navigateur qui nous sert à consulter des sites Web dispose d'une fonctionnalité intéressante qui consiste à télécharger plusieurs images d'une page en même temps. Le téléchargement et l'affichage d'une image correspond à un *thread* distinct. Le navigateur peut choisir de démarrer plusieurs téléchargements simultanés et cela , en rapport avec le débit maximum de la connexion.



Exemple d'utilisation des Threads

3) L'objet thread

Tout d'abord, lorsqu'on souhaite qu'une classe puisse se comporter comme un *thread* , il faut que :

- Cette classe implémente l'interface *Runnable* , surcharge la méthode *run* et contient un champ de type *Thread*.

ou que :

- Cette classe hérite de la classe *Thread*.

La deuxième solution s'avère irréaliste étant donné que la plupart du temps, vos propres classes hériteront déjà d'une autre classe et par conséquent, ne pourront pas hériter de plusieurs classes. En JAVA, l'héritage multiple n'est pas supporté. Néanmoins, si votre classe n'hérite d'aucune autre, il est préférable de la faire hériter de la classe *Thread*

Voici un diagramme objet représentant le cas de figure le plus courant où l'objet devant s'exécuter comme un *thread* implémente l'interface *Runnable* et hérite de sa classe parent.

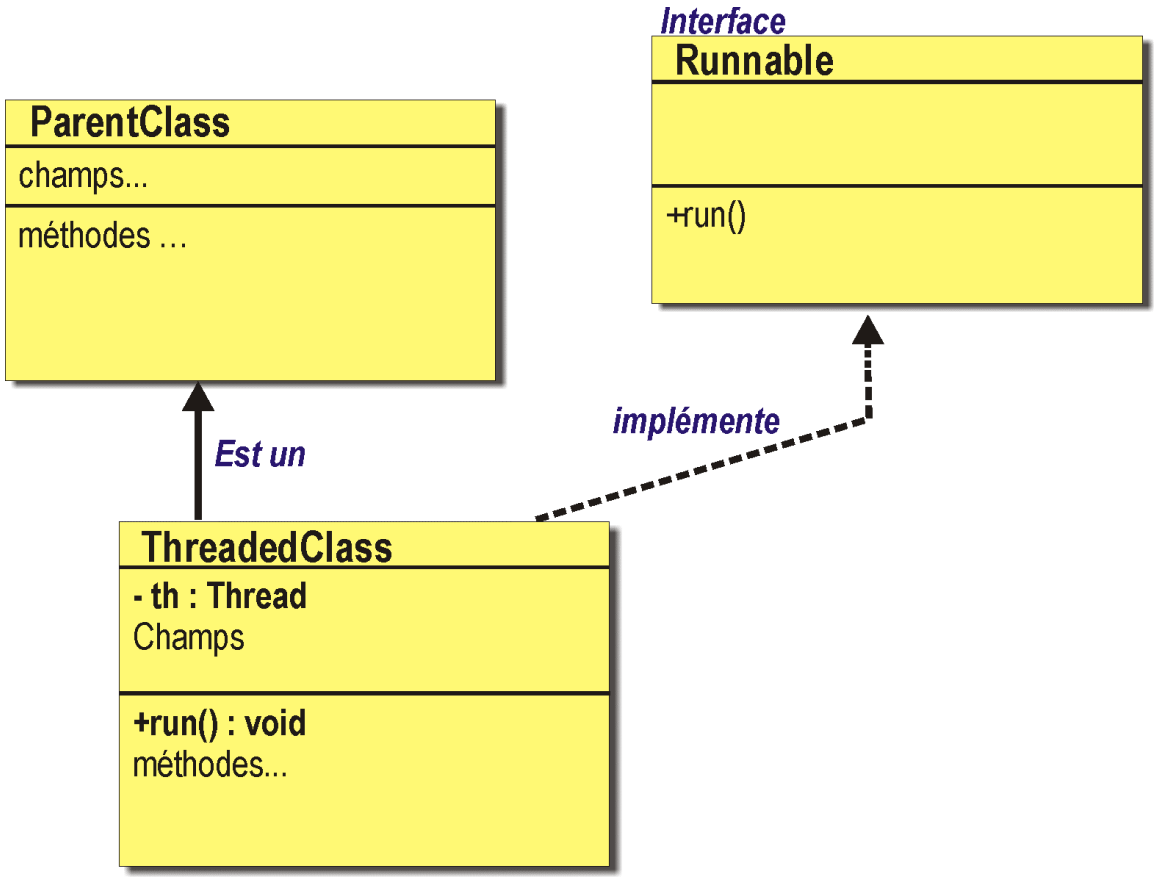


Diagramme objet pour les threads

Le code Java pour ce modèle pourrait être le suivant :

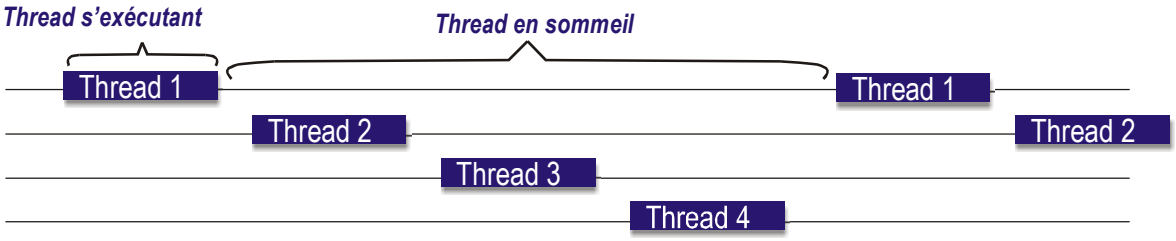
```
1  public class ThreadedClass extends ParentClass implements Runnable {
2
3  // Champs
4  Thread th;
5  …
6
7  // Constructeurs
8  public ThreadedClass() {
9   th = new Thread(this);  // Lie la classe avec le thread qu'elle
10                          // contient
11  th.start();             // Démarre le thread
12 …
13
14 // Méthodes
15 public void run() { // Méthode appelée par start()
16 … Processus d'exécution
17 th.sleep(100);     //Met le thread en sommeil pendant 100ms
18 }
19 }
```

Code partiel de l'utilisation d'un thread

4) Exécution de plusieurs threads

L'objectif du multitâche étant d'exécuter plusieurs *threads* , en simultané, il faut donc une organisation sans de manière à ce que chaque *thread* est une chance de s'exécuter. En tant que développeur, cette responsabilité vous incombe puisque vous allez pouvoir fixer le temps de sommeil de vos *threads*.

Voici un schéma illustrant le processus d'exécution des *threads* sur une machine disposant d'un seul processeur :



Exécution de plusieurs thread sur une machine mono processeur

B.III - Propriétés des *threads*

1) Etats des threads

Les threads peuvent être dans plusieurs états différents :

- **Nouveau** : Le thread a été crée avec *new* mais n'est pas encore activé
- **Exécutable** : La méthode *start* a été appelée mais cela ne signifie pas que le thread est exécuté. Cela va dépendre du système d'exploitation qui doit lui donner une fenêtre d'exécution.
- **Bloqué** : Plusieurs évènements peuvent bloquer un *thread*. La méthode *sleep* lorsqu'elle est appelée, met le *thread* dans cet état. Il y a cependant plusieurs autres causes dont nous parlerons un peu plus loin dans ce chapitre.
- **Mort** : Lorsque la méthode *run* est terminée, le *thread* est considéré comme mort. Il peut y avoir d'autres causes dont nous parlerons plus loin.

2) Thread bloqué

Le thread peut être dans cet état pour les causes suivantes :

- La méthode *sleep* ou *yield* du thread est appelée
- Le *thread* appelle une opération bloquante sur les entrées-sorties. Cette opération ne rendant pas la main tant qu'elle n'est pas terminée.
- Le *thread* appelle la méthode *wait*
- Le *thread* essaie de verrouiller un objet déjà verrouillé par un autre *thread*.
- La méthode *suspend* du *thread* est appelée (Pas conseillée !!!)

Un *thread* bloqué peut uniquement être activé par la même technique que celle qui l'a bloqué. Par exemple, un *thread* bloqué par une opération d'entrée-sortie devra attendre que celle-ci se termine. De même qu'un *thread* bloqué par la méthode *sleep* devra attendre le délai passé en argument.

3) Thread mort (Deadlock)

Voici les différentes raisons qui peuvent faire mourir un *thread* :

- Le *thread* meurt d'une mort naturelle parce que sa méthode *run* s'est terminée
- Le *thread* meurt soudainement parce qu'une exception non récupérée a mis fin à la méthode *run*

Pour connaître l'état d'existence d'un *thread* , il suffit d'appeler sa méthode *isAlive* qui renvoie *false* dans le cas où le *thread* est mort.

4) Interrompre un thread

Un *thread* ne s'arrête que si sa méthode *run* se termine. Il n'existe aucune technique intégrée à Java pour interrompre un *thread* . Celui-ci doit donc vérifier régulièrement, au sein de la méthode *run*, s'il doit s'arrêter :

```
20 public void run {
21 while ( aucune requête de fin && encore du travail à faire)
22  {  Travail à faire
23  }
24 }
```

Néanmoins, lorsque qu'un thread est endormi, il ne peut pas vérifier s'il doit s'arrêter. Voilà tout l'intérêt de la méthode *interrupt()*. Lorsque cette méthode est appelée, l'exception *InterruptException* termine la phase de blocage.

Un thread susceptible d'être interrompu doit donc lever l'exception *InterruptException* :

```
1  public void run {
2
3  try {
4     while ( aucune requête de fin && encore du travail à faire)
5      {  Travail à faire
6      }
7      } catch ( InterruptedException )
8        { // Le thread a été interrompu pendant sleep ou wait
9         }
10 }
```

Si la méthode *interrupt* a été appelée pendant que le thread n'était pas bloqué, aucune exception de type *InterruptException* n'est générée. Le thread doit alors appeler la méthode *interrupted* pour déterminer s'il a été interrompu.

Voici la structure correcte d'un thread :

```
11 public void run {
12
13 try {
14    while ( encore du travail à faire && !interrupted())
15     {  Travail à faire }
16     } catch ( InterruptedException )
17       { // Le thread a été interrompu pendant sleep ou wait }
18 }
```

5) Priorités d'un thread

Il existe 10 niveaux de priorité d'un thread en JAVA. Voici les constantes statiques de la classe *Thread* :

- **MIN_PRIORITY** : Valeur entière de 1
- **NORM_PRIORITY** : Valeur entière de 5
- **MAX_PRIORITY** : Valeur entière de 10

Lorsque le gestionnaire des *threads* doit choisir un nouveau *thread* a exécuter, il choisit la plupart du temps le *thread* ayant la priorité la plus haute.

Un thread peut aussi être interrompu si un thread de priorité supérieure s'est réveillé

Si plusieurs *thread* exécutables ont la même priorité, c'est au système de gestion des *threads* de décider lequel de ces *threads* va être exécuté. Le problème est que, d'un système à l'autre, ce choix peut-être différent. Voici une des faiblesses du langage JAVA qui signifie qu'on ne peut pas être sûr que les programmes multithread s'exécuteront de la même façon sur toutes les plate formes.

6) Threads égoïstes

Un *thread* égoïste est un *thread* qui ne fait pas appel à *sleep* ou *yield*. Par conséquent, il ne laisse pas la chance aux autres *threads* de pouvoir s'exécuter. Néanmoins, dans le cas d'un système multitâche préemptif, c'est le système d'exploitation qui est responsable de l'exécution des *threads*. Le bon fonctionnement du programme dépendra alors de l'OS.

7) Constructeurs et méthodes de la classe *Thread*

Constructeurs de Thread
Thread() Crée un nouveau thread.
Thread(Runnable target) Crée un nouveau Thread relatif à l'objet implémentant l'interface *Runnable*
Thread(Runnable target, String name) Crée un nouveau Thread relatif à l'objet implémentant l'interface *Runnable* avec un nom
Thread(String name) Crée un nouveau *Thread* avec un nom
Thread(ThreadGroup group, Runnable target) Crée un nouveau Thread relatif à l'objet implémentant l'interface *Runnable* , membre du groupe passé en argument.
Thread(ThreadGroup group, Runnable target, String name) Crée un nouveau Thread relatif à l'objet implémentant l'interface *Runnable* , membre du groupe passé en argument avec un nom
Thread(ThreadGroup group, String name) Crée un nouveau *Thread* membre du groupe passé en argument avec un nom

Liste des constructeurs de THREAD

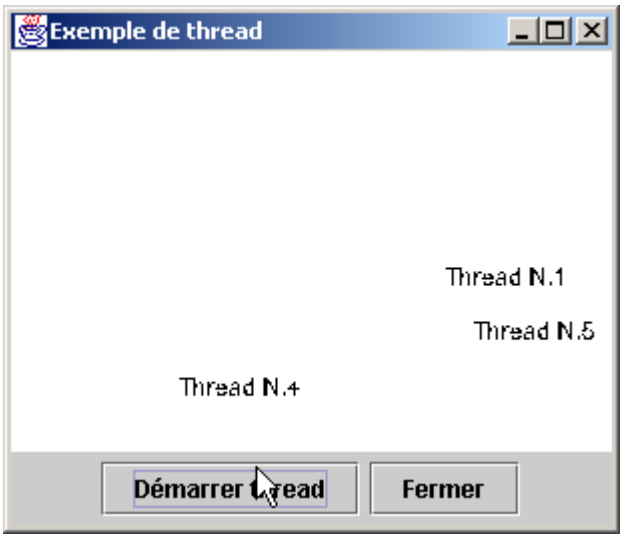
Méthodes de Thread	
static int	**activeCount**() Renvoie le nombre de threads actifs dans le groupe de ce thread
void	**checkAccess**() Détermine si le thread actuellement actif a les permissions de modifier ce thread
static Thread	**currentThread**() Renvoie la reference au thread actuellement exécuté
String	**getName**() Renvoie le nom du thread
int	**getPriority**() Renvoie la priorité de Thread.
ThreadGroup	**getThreadGroup**() Renvoie le groupe de thread auquel le thread appartient
void	**interrupt**() Interrompt le thread (DANGEREUX)
static boolean	**interrupted**() Test si le thread courant a été interrompu
boolean	**isAlive**() Teste si le thread n'est pas mort
boolean	**isInterrupted**() Test si ce thread a été interrompu
void	**join**() Attend que ce thread meurt.
void	**join**(long millis) Attends au moins *millis* millisecondes que ce thread meurt
void	**join**(long millis, int nanos) Même chose mais millis+nanos
void	**run**() méthode appelée par la méthode start pour démarrer le thread
void	**setName**(String name) Change le nom du thread
void	**setPriority**(int newPriority) Change la priorité du thread
static void	**sleep**(long millis) Endort le thread pour une durée *millis* millisecondes
static void	**sleep**(long millis, int nanos) Endort le thread pour une durée *millis* millisecondes + *nanos* nanonsecondes
void	**start**() Démarre le thread et la machine virtuelle appelle la méthode run
String	**toString**() Renvoie une représentation chaîne comportant le nom, la priorité et le groupe auquel appartient le thread
static void	**yield**() Provoque l'arrêt de l'exécution du thread et autorise les autres threads à s'exécuter

Liste non exhaustive des méthodes de la classe Thread

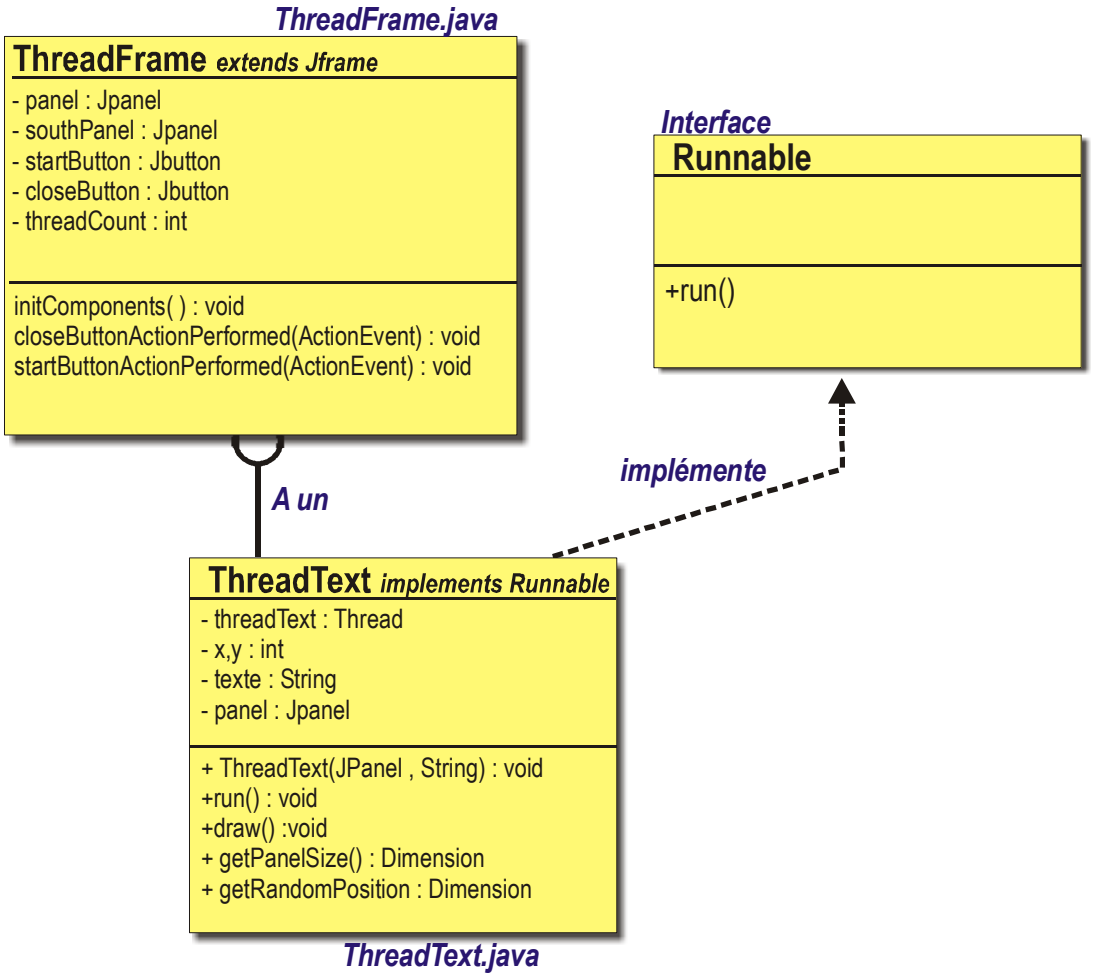
B.IV - Exemple complet

1) Présentation

Voici l'exemple d'une fenêtre affichant un texte clignotant dès qu'on appui sur un bouton. A chaque appui sur ce bouton, un thread différent est exécuté :



Voici le diagramme objet de cet exemple :



Voici le code source des fichiers *ThreadText.java* et *ThreadFrame.java*

2) Code source

```
1  /*
2   * ThreadText.java
3   *
4   * Created on 2 octobre 2001, 17:11
5   */
6  import javax.swing.*;
7  import java.awt.*;
8  /**
9   *
10  * @author  genael
11  * @version
12  */
13 public class ThreadText implements Runnable {
14
15 /** Texte à afficher
16  */
17     private String text;
18
19 /** Position horizontale du texte
20  */
21     private int x;
22
23 /** Position verticale du texte
24  */
25     private int y;
26
27 /** Panneau dans lequel apparait le texte
28  */
29     private JPanel panel;
30
31     private Thread textThread;
32
33     /** Creates new ThreadText */
34     public ThreadText(JPanel conteneur, String text) {
35         this.panel = conteneur;
36         this.text = text;
37         Dimension d = getRandomPosition();
38
39         x=d.width;
40         y=d.height;
41         textThread = new Thread(this);
42         textThread.start();
43      }
44
45 /** Exécute le thread pendant 15s avec une fréquence de clignotement de
   150ms
46  */
47     public void run() {
48
49         try {
50             for (int i=0; i<100;i++)
51             {
52                 draw();
53                 textThread.sleep(150);
54             }
55         } catch ( InterruptedException ie) {}
56
57     }
58
59 /** Dessine le texte ou l'efface s'il est déjà affiché (Mode XOR)
60  */
61     public void draw() {
62         Graphics g = panel.getGraphics();
63         panel.setBackground(Color.white);
64         g.setXORMode(panel.getBackground());
65         g.drawString(text,x,y);
66         g.dispose();
67     }
```

```
68
69 /** Renvoie un objet Dimension correspondant à la taille du panneau
70  */
71 public Dimension getPanelSize() {
72
73     return panel.getSize();
74 }
75
76 /** Renvoie des coordonnées aléatoires dans la limite de la taille du
   panneau
77  */
78 public Dimension getRandomPosition() {
79
80     Dimension d = new Dimension(0,0);
81     Dimension panelSize = getPanelSize();
82     int xmax,ymax;
83     xmax = panelSize.width - 60;
84     ymax = panelSize.height - 20;
85     double nb = Math.random();
86     // Détermine des coordonnées au hasard
87     d.width = (int) Math.rint( xmax * ((1 - Math.random()) ));
88     d.height = (int) Math.rint( ymax * ((1 - Math.random()) ));
89     return d;
90 }
91 }
```

ThreadText.java

- **Ligne 47-57** : Méthode run permettant de lancer le thread

```
1  /*
2   * JFrame.java
3   *
4   * Created on 2 octobre 2001, 16:56
5   */
6  import java.awt.event.*;
7  import javax.swing.*;
8  import *;
9  /**
10  *
11  * @author  genael
12  * @version
13  */
14 public class ThreadFrame extends JFrame {
15
16     /** Creates new form JFrame */
17     public ThreadFrame(String text) {
18         threadCount =1;
19         setTitle(text);
20         initComponents ();
21         pack ();
22     }
23
24     private void initComponents() {
25         panel = new JPanel();
26         southPanel = new JPanel();
27         startButton = new JButton();
28         closeButton = new JButton();
29         addWindowListener(new WindowAdapter() {
30             public void windowClosing(WindowEvent evt) {
31                 exitForm(evt);
32             }
33         }
34         );
35
36         panel.setPreferredSize(new Dimension(300, 200));
37
38         getContentPane().add(panel, BorderLayout.CENTER);
```

```
39          startButton.setText("D\u00e9marrer thread");
40          startButton.addActionListener(new ActionListener()
41              {
42              public void actionPerformed(ActionEvent evt)
43                {
44                    startButtonActionPerformed(evt);
45                }
46              }
47            );
48            southPanel.add(startButton);
49            closeButton.setText("Fermer");
50            closeButton.addActionListener(new ActionListener()
51                {
52                public void actionPerformed(ActionEvent evt) {
53                    closeButtonActionPerformed(evt);
54                }
55            }
56            );
57          southPanel.add(closeButton);
58          getContentPane().add(southPanel, BorderLayout.SOUTH);
59      }
60
61  private void closeButtonActionPerformed(ActionEvent evt) {
62                System.exit(0);
63    }
64
65  private void startButtonActionPerformed(ActionEvent evt) {
66
67  ThreadText thtxt = new ThreadText(panel,"Thread N." + threadCount++);
68  }
69
70      /** Exit the Application */
71      private void exitForm(WindowEvent evt) {
72          System.exit (0);
73      }
74
75      // Variables declaration
76      private JPanel panel;
77      private JPanel southPanel;
78      private JButton startButton;
79      private JButton closeButton;
80      private int threadCount;
81
82  }
```

Code source ThreadFramet.java

- **Ligne 67** : Création d'un objet thread à chaque appui sur le bouton

B.V - Groupes de threads

Dans le cas où votre programme contient une grande quantité de *threads*, il peut être utile de manipuler ces *threads* par groupe. Reprenons l'exemple du navigateur internet téléchargeant des images. Lorsque vous cliquez sur *Arrêter* , le navigateur doit stopper le téléchargement des images. Si chaque image se télécharge par l'intermédiaire d'un *thread* et que tous , font partie d'un groupe, nous disposons d'un moyen efficace de les stopper tous.

1) Construire un groupe de threads

Utilisez le constructeur suivant :

```
1  ThreadGroup g = new ThreadGroup("WebImages0047654");
```

La chaîne passée en argument du constructeur doit être unique

Pour ajouter des threads à ce groupe :

```
2  Thread th = new Thread (g, "Image1");
```

2) Interrompre les threads d'un groupe

Pour interrompre tous les threads d'un groupe, il suffit d'appeler la méthode *interrupt* du groupe :

```
3  g.interrupt();
```

Un groupe peut posséder des groupes enfants. Le fait d'interrompre les threads du groupe parent, interromps également les membres des groupes enfants

3) Connaître les threads actifs

Pour connaître le nombre de threads actifs d'un groupe, il faut appeler la méthode *activecount* :

```
4  int nth = g.activeCount();
```

Il existe un nombre important de methods pour gérer ces groupes. Consultez la doc API Java 2 pour en savoir plus

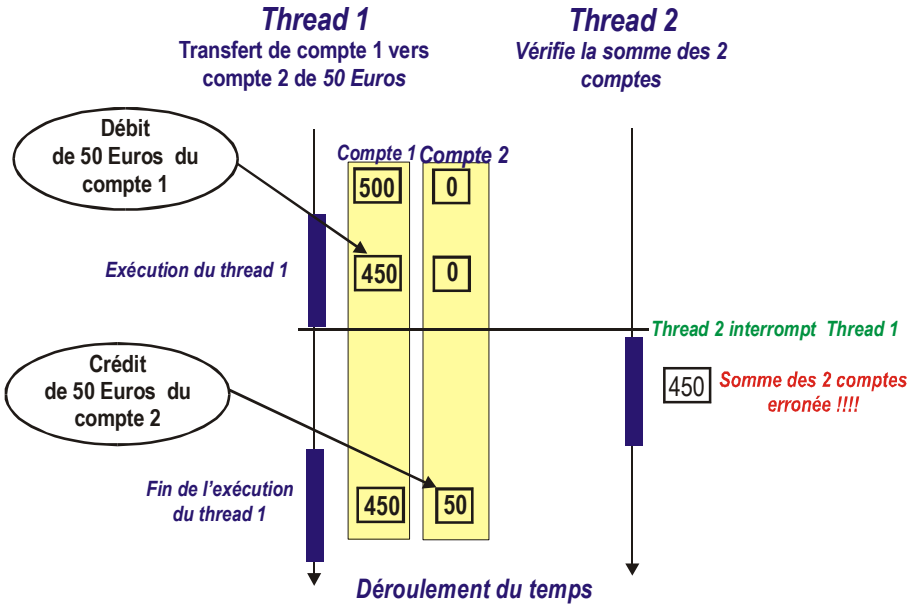
B.VI - Synchronisation

1) Présentation du problème

Dans de nombreux cas, les *threads* partagent les mêmes objets et par conséquent, la modification de ces objets par un *thread* pourrait avoir des conséquences fâcheuses pour les autres *thread* notamment si ceux-ci modifient un objet en même temps.

Java propose un mécanisme de verrouillage des objets par un *thread* de telle sorte qu'aucun autre *thread* ne puisse le modifier aussi. Lorsque le *thread*. Néanmoins cela ne résous pas tous les problèmes puisque les *threads* peuvent également être interrompu pendant leur travail.

Voici une illustration de ce qui pourrait arriver sur un compte bancaire sans verrouillage des données :



Fonctionnement non synchronisé

Pendant un certain laps de temps, la somme des 2 comptes sera inexacte !!!

Il faut donc que le *Thread 1* soit certain de ne pas être interrompu pendant le transfert.

2) mot clé *synchronised*

Pour permettre au thread 1 de ne pas être interrompu, on utilise le mot clé *synchronised* comme modificateur de la méthode qui ne doit pas être interrompue :

```
1  public synchronised void transfert( Compte source,
2                                      Compte dest, int montant) {
3  {
4   if (source.getSolde()<montant)
5           return;
6  source.solde = source.solde - montant;
7  dest.solde = dest.solde + montant;
8  }
```

Dans ce cas, la méthode ne sera pas interrompue entre la ligne 6 et 7.

3) Verrous d'objets

Chaque fois qu'une méthode ou un bloc de code est affecté du mot clé *synchronised* , la méthode est verrouillée et aucun autre objet ne peut appeler cette méthode. Le *Thread* pose un verrou qu'il enlèvera en sortant de la méthode.

Ce système de verrouillage soulève tout de même un problème si jamais le *thread* verrouillé effectue une action qui se prolonge dans le temps, voire infiniment. La conséquence est que tous les autres *threads* n'ont plus aucune chance de s'exécuter (Sauf par découpage du temps au niveau de l'OS dans le cas du multitâche préemptif).

Pour pallier à cet inconvénient, il y a la méthode *wait*.

4) wait et notify

4-a) wait

Lors de l'appel à la méthode *wait* , le *thread* enlève son verrou et vient se mettre en liste d'attente pour l'exécution. L'objet étant déverrouillé, les autres *threads* peuvent alors s'exécuter.

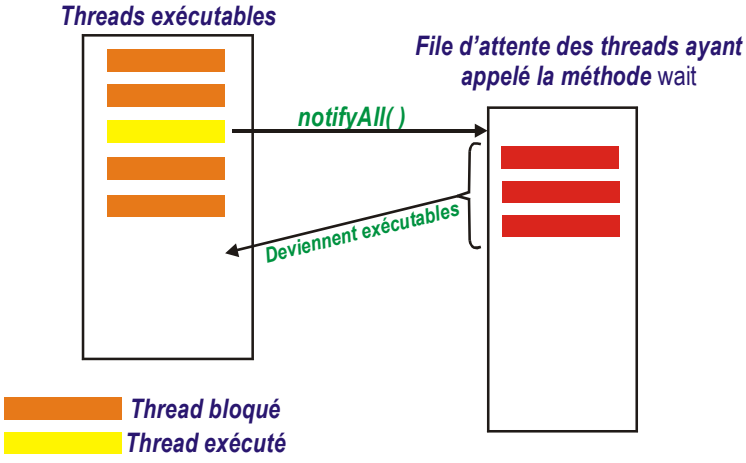
Il existe une différence importante entre un thread en sommeil et un thread ayant appelé la méthode *wait* . En effet, ce dernier est dans une liste d'attente et le gestionnaire de *threads* ne fera rien pour l'exécuter. Il risque donc d'être bloqué indéfiniment à moins qu'un *thread* ne fasse appel à la méthode *notify* ou *notifyAll*.

4-b) notify et notifyAll

Donc , si un *thread* appelle sa méthode *wait* , il n'a aucun moyen de se débloquer lui-même. Il doit donc compter sur les autres *threads* pour appeler la méthode *notifyAll* ou *notify*. Si ce n'est pas le cas, nous sommes en présence d'un verrou mort (*deadlock*).

Pour éviter ce genre de situation, pensez toujours à appeler régulièrement la méthode *notifyAll* ou *notify*.

La méthode *notify* donne l'ordre au gestionnaire des *threads* de débloquer un *thread* au hasard. La méthode *notifyAll* débloque tous les *threads* qui deviennent par voie de conséquence exécutables.



File d'attente des threads

Il peut être dangereux d'appeler la méthode *notify* car vous ne savez pas quel *thread* sera débloqué. Il est plus approprié d'employer la méthode *notifyAll*.

L'endroit le plus approprié pour appeler cette méthode est dans l'objet qui est susceptible de changer la donne pour les *threads* mis dans la file d'attente avec *wait*.

Il faut savoir que le mécanisme de synchronisation ralentit les programmes qui l'utilise. C'est le prix à payer pour un mécanisme assurant une très bonne intégrité des données.

B.VII - Les *threads* et SWING

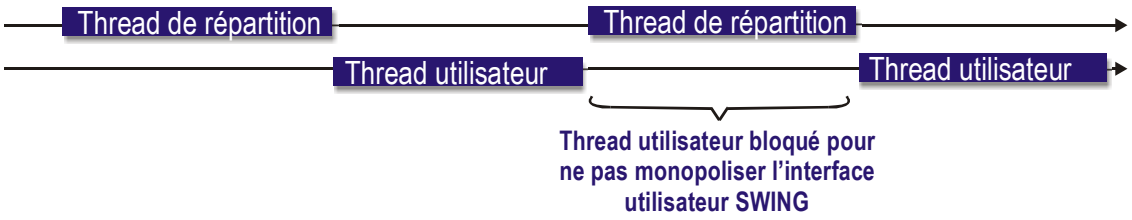
Dans une application graphique, la méthode *main* exécute un *thread* principal. Au moment de l'appel à la méthode *show* ou *setVisible* de la fenêtre principale, un deuxième *thread* est crée. Il est chargé de gérer la répartition des évènements. Pendant ce temps, le *thread* principal s'exécute jusqu'à la fin du *main*.

Dans la plupart des cas, le *thread* principal prend fin rapidement et ceci dès l'affichage de la fenêtre. Il reste donc le *thread* de répartition des évènements qui s'exécute et réagit aux évènements comme l'appui sur un bouton ou l'appui sur une entrée de menu.

Le problème est que *SWING* n'est pas compatible avec les *threads*. En effet, la plupart des méthodes des objets *SWING* ne sont pas synchronisées. La conséquence directe est que **si un des *threads* de l'utilisateur modifie des composants de l'interface utilisateur** (Comme une structure arborescente de type J*Tree* , par exemple) , **les données risquent d'être corrompues si l'interface agit aussi sur ces composants.**

1) Quand utiliser les threads ?

En général, dès qu'une action est susceptible de prendre du temps ou qu'elle n'est pas certaine d'aboutir, il faut employer un *thread* pour ne pas monopoliser l'interface utilisateur. En effet, si vous lancez une connexion réseau et que la connexion n'abouti pas, vous souhaiterez pouvoir annuler pour faire autre chose.



Exemple de fonctionnement des threads et SWING

C. Programmation réseau

C.I - Introduction

Voyons maintenant comment *Java* va nous permettre d'accéder au réseau par l'intermédiaire de classes présentes dans l'API Java et qui répondent partiellement aux exigences d'Internet et des réseaux informatiques.

1) Modèle OSI

Tout d'abord, dans un environnement construit autour du modèle *OSI*[7], il existe plusieurs types de connexion par *paquets*[8] (TCP) ou par *datagrammes*[9] (UDP). Ce modèle basée sur 7 couches dépendantes les unes des autres représente une vue de l'esprit sur les différents niveaux sur lesquels une machine et des utilisateurs peuvent échanger des informations sur le réseau :

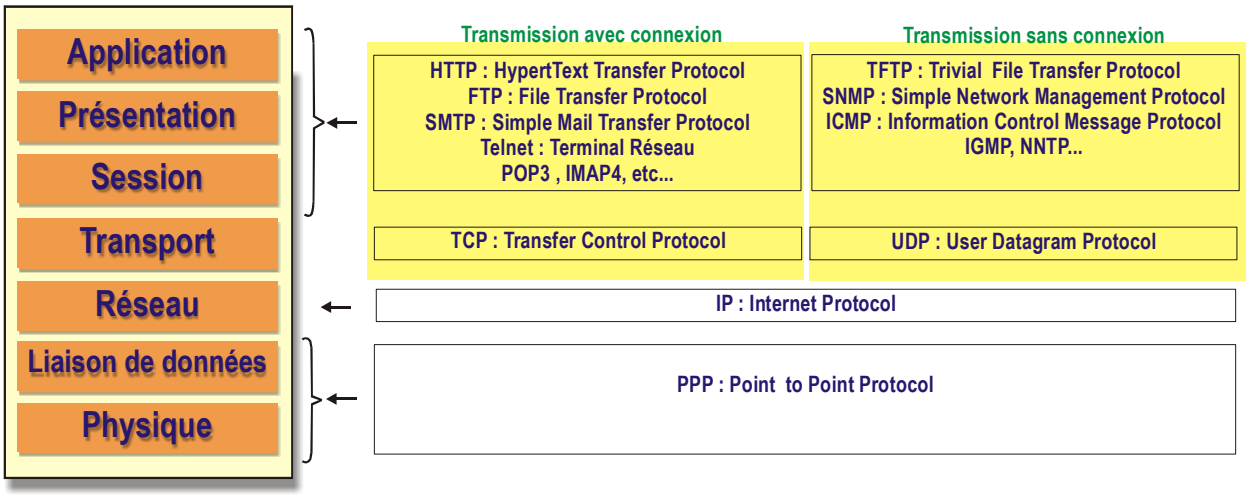
Les 7 couches du modèle OSI

2) Protocoles

Les protocoles de communication réseau peuvent travailler sur des couches différentes et la plupart d'entre eux sont dépendants des protocoles travaillant sur les niveaux inférieurs. Par exemple , le protocole *FTP*[10] qui travaille sur la couche *Application* est dépendant de *TCP*[11] qui lui, travaille sur la couche *Transport*.

Si vous connaissez la plupart des protocoles liés au modèle *OSI*, voici un schéma représentant les protocoles ainsi que les couches sur lesquelles ils travaillent.

[7] OSI : Open System Interconnection : Modèle basé sur une architecture à 7 couches
[8] Paquets : Les données à transmettre sont scindées en plusieurs petites unités appelés Paquets
[9] Datagrammes : Données envoyées sur le réseau sans aucune certitude sur leur réception sur l'hôte de destination
[10] FTP : File Transfer Protocol : Protocole de transfert de fichiers très utilisé sur Internet
[11] TCP : Transport Control Protocol : Protocole de transfert contrôlé



Correspondance modèle OSI et protocoles de la famille TCP/IP

3) Où travaille Java

A priori, les classes *Socket* , *ServerSocket, DatagramPacket* et *DatagramSocket* travaillent sur la couche *Transport* du modèle *OSI*. Mais les classes *INetAdress* , *URLConnection* et *URLEncoder* peuvent travailler sur les couches supérieures de ce même modèle.

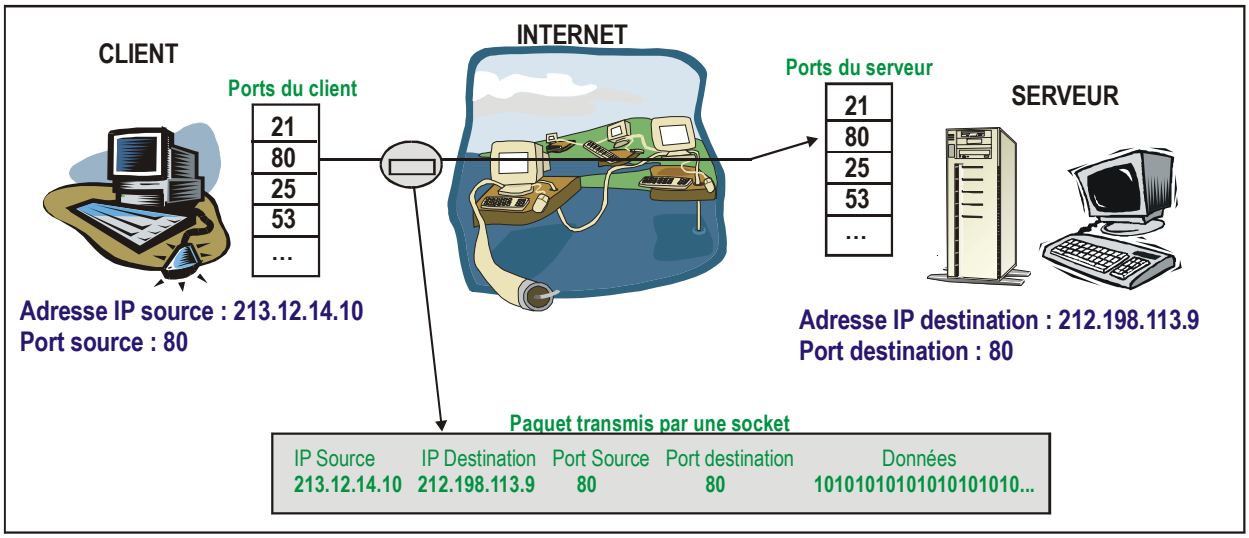
Bien sûr, la plupart des développeurs JAVA ont à leur disposition des packages supplémentaires leur permettant de travailler sur les couches supérieures sans avoir à se soucier de ce qui se passe sur les couches inférieures. Par exemple, *L'API Java-Mail* conçue par *SUN* permet de gérer l'envoi d'emails et prend en charge toutes les subtilités des protocoles *SMTP* ou *POP3*.

C.II - Implémenter des sockets

1) Introduction

Les *sockets* sont des outils permettant d'effectuer une transmission en mode connecté entre deux entités d'un réseau (Un client et un serveur par exemple). Les deux entités possèdent des *ports* qu'ils peuvent utiliser pour envoyer et recevoir des données. Chaque entité possède également une ou plusieurs *adresses IP*[12] du style *192.168.5.12*.

Donc , avec un port source , un port destination, une adresse IP source et une adresse IP destination, une connexion est possible entre ces deux entités :



Exemple de communication avec les sockets

2) Classe Socket

La classe *Socket* se situe dans la librairie *java.net*

2-a) Créer des sockets

La classe *Socket* associée aux flux de données, permet de lire et d'écrire des données sur le réseau. La plupart du temps, les sockets sont utilisées pour se connecter à un serveur. En fait , les serveurs proposent des services (*HTTP , FTP , TELNET...)* que les clients peuvent utiliser.

La classe *Socket* permet à un client de se connecter à un hôte :

```
1  Socket s = new Socket(hote, port );
```

hote représente le nom de l'hôte et *port* représente le port sur lequel la socket va se connecter. Par exemple, si vous vous connectez sur le port 80, vous aurez toutes les chances de créer une connexion vers un serveur web qui pourra renvoyer des pages *HTML*[13].

En créant une socket, vous pouvez également spécifier le port source de la connexion. Si le port source n'est pas spécifié, le programme utilise un port libre choisi parmi tous ceux qui sont libres. Voici la liste des constructeurs disponibles :

Constructeurs de Socket	
protected	Socket()

[12] Adresse IP : Adresse sur 32 bits (Norme IPv4) utilisée pour communiquer sur un réseau Intranet ou Internet
[13] HTML : HyperText Markup Language – Langage utilisé pour les pages internet

	Crée une socket en mode non connecté, avec une version par défaut de *SocketImpl*. Utilisable par les classes enfant de *Socket*.
	Socket(InetAddress address, int port) Crée une connexion vers l'hôte ayant l'adresse et le port donnés en arguments
	Socket(InetAddress address, int port, InetAddress localAddr, int localPort) Crée une connexion vers l'hôte ayant l'adresse et le port donnés en arguments. *localAddr* et *localPort* paramètre les données sources de la socket
protected	**Socket**(SocketImpl impl) Crée une socket en mode non connectée avec une version spécifique de *SocketImpl*
	Socket(String host, int port) Crée une connexion avec le nom de l'hôte spécifié ainsi que le port. L'appel à cette méthode sous entend qu'il y a une résolution de nom en adresse IP.
	Socket(String host, int port, InetAddress localAddr, int localPort) Même chose que la précédente avant en plus , des precisions sur l'adresse locale et le port

Constructeurs de Socket

Il existe d'autres constructeurs non documenté ici car ils ne s'utiliseront plus dans les versions à venir du JDK.

2-b) Lire et écrire des données

A partir de là, il est possible de créer un flux d'entrée ou un flux de sortie qui vous permettrons de communiquer dans les deux sens avec le serveur. Chaque objet *Socket* dispose de deux méthodes intéressantes pour le faire :

- ***getInputStream( )*** : Retourne le flux permettant de lire les octets provenant du serveur pour cette connexion
- ***getOutputStream()*** : Retourne le flux permettant d'écrire des octets à destination du serveur pour cette connexion.

```
1  Socket s = new Socket(hote, port );
2  InputStream in = s.getInputStream();
3  OutputStream out = s.getOutputStream();
```

Il est souvent plus utile, surtout sur Internet, de lire et d'écrire des **caractères** sur le réseau. Pour le faire, il suffit d'utiliser la force de frappe des *flux filtrés*[14] :

```
1  Socket s = new Socket(hote, port );
2  BufferedReader in = new BufferedReader (
3                          new InputStreamReader(s.getInputStream()));
```

Dans notre exemple , nous obtenons un flux de caractères bufférisé provenant de l'hôte. Il nous reste à effectuer une boucle de lecture des caractères :

```
4  String line;
5  while ( (line = in.readLine) != null )
6           System.out.println(line);
```

2-c) Temps d'attente

Sur un réseau, il y a une ribambelle de raisons pour lesquelles la communication peut échouer. Une socket qui tente de communiquer avec une autre machine non disponible

[14] Flux filtrés : Voir chapitre sur les flux et les fichiers

peut bloquer un programme indéfiniment. C'est pourquoi une socket peut attendre un certain temps défini par le programmeur, puis abandonner la connexion si celle-ci ne peut se faire avant le délai imparti. Ce délai se fixe grâce à la méthode *setSoTimeout* :

```
7  s.setSoTimeout(10000);   // Timeout de 10s
```

Après ce délai, la socket est fermée. Toute opération de lecture ou d'écriture déclenche alors une interruption de type *InterruptedIOException*. Par conséquent, il peut être vraiment utile de lever cette exception au cas où :

```
1  try {
2           Socket s = new Socket(hote, port );
3           s.setSoTimeout(10000);
4           BufferedReader in = new BufferedReader (
5                         new InputStreamReader(s.getInputStream()));
6           String line;
7           while ( (line = in.readLine) != null )
8                System.out.println(line);
9      } catch (InterruptedIOException ie) {
10          System.out.println("Délai d'attente dépassé");
11          }
```

L'exemple précédent ne résout pas pour autant le problème puisque l'appel au constructeur peut geler le programme si la connexion n'abouti pas. Il faut déjà posséder un objet de type *Socket* avant d'appeler la méthode *setSoTimeout*.

Pour résoudre ce problème, il vous faudra créer la socket dans un *thread*[15] séparé.

2-d) Socket et Thread

Pour illustrer le point qui vient d'être soulevé, voici une classe nommée *SocketOpener* permettant d'ouvrir une socket dans un thread séparé et de renvoyer la socket lorsqu'elle est ouverte et si elle n'a pas atteint la valeur de *timeout*:

```
1  import java.net.*;
2  import java.io.*;
3
4  class SocketOpener implements Runnable
5  {
6  // Méthode statique ouvrant la socket
7  public static Socket openSocket(String aHost, int aPort,int timeout)
8     {  SocketOpener opener = new SocketOpener(aHost, aPort);
9        Thread t = new Thread(opener);
10       t.start();
11       try
12       {  t.join(timeout);
13       }
14       catch (InterruptedException exception){}
15       return opener.getSocket();
16    }
17
18    public SocketOpener(String aHost, int aPort)
19    {  socket = null;
20       host = aHost;
21       port = aPort;
22    }
23 // Exécution dans un thread séparé
24    public void run()
25    {  try
26       {  socket = new Socket(host, port);
27       }
28       catch (IOException exception){}
```

[15] Thread : Voir le chapitre concernant les threads

```
29      }
30
31      public Socket getSocket()
32      {  return socket;
33      }
34
35      private String host;
36      private int port;
37      private Socket socket;
38  };
```

SocketOpener.java

Pour utiliser cette classe :

```
1   Socket s = SocketOpener.openSocket(host,port,10000);
2
3   if (s==null)
4           System.out.println("La socket n'a pas pu être ouverte");
5   else
6      // Travail avec la socket
```

3) Classe InetAddress

Cette classe est très utile pour utiliser les sockets avec des adresses internet. Elle permet la résolution de nom en adresse IP sous Java et par voie de conséquence, elle fourni à la classe *Socket* la possibilité de travailler directement sur des adresses internet.

Pour créer un objet *InetAddress* :

```
1   InetAddress adr = InetAddress.getByName("www.gita.greta.fr");
```

L'objet *adr* va contenir toutes les informations concernant cette adresse internet et notamment les 4 octets de l'adresse IP récupérable dans un tableau de 4 *bytes* grâce à la méthode *getBytes* :

```
2   byte [] tabl = adr.getBytes();
```

Si vous exécutez le programme qui suit avec la commande suivante, vous obtiendrez, si vous êtes connecté à Internet, l'adresse IP 212.37.208.183 :

```
1   java InetAddressTest www.gita.greta.fr
```

```
1   import java.net.*;
2   import java.io.*;
3
4   public class InetAddressTest {
5
6   public static void main ( String [] args ) {
7
8   try {
9
10       InetAddress adr = InetAddress.getByName(args[0]);
11       System.out.print("Adresse IP : " + adr.toString());
12           } catch (Exception e) {
13                System.out.println(e);
14                }
15   }
16  }
```

InetAddressTest.java

Une fonctionnalité intéressante de cette classe consiste à obtenir toutes les adresses IP correspondant au nom DNS fourni. Il s'agit de la méthode statique *getAllByName* . Voici l'exemple précédent modifié :

```
1  import java.net.*;
2  import java.io.*;
3
4  public class InetAllAddressTest {
5
6  public static void main ( String [] args ) {
7
8  try {
9
10          InetAddress[] adrs = InetAddress.getAllByName(args[0]);
11
12          for (int i=0; i<adrs.length;i++)
13          System.out.println("IP " + i +" : " + adrs[i].toString());
14          } catch (Exception e) {
15               System.out.println(e);
16               }
17 }
18 }
```

En exécutant la commande suivante :

```
1  java InetAllAddressTest www.yahoo.com
```

vous obtiendrez une liste d'adresses IP :

```
1  IP 0 : www.yahoo.com/64.58.76.222
2  IP 1 : www.yahoo.com/64.58.76.228
3  IP 2 : www.yahoo.com/64.58.76.177
4  IP 3 : www.yahoo.com/64.58.76.176
5  IP 4 : www.yahoo.com/64.58.76.229
6  IP 5 : www.yahoo.com/64.58.76.225
7  IP 6 : www.yahoo.com/64.58.76.178
8  IP 7 : www.yahoo.com/64.58.76.227
9  IP 8 : www.yahoo.com/64.58.76.224
10 IP 9 : www.yahoo.com/64.58.76.226
11 IP 10 : www.yahoo.com/64.58.76.223
12 IP 11 : www.yahoo.com/64.58.76.179
```

4) Exemple de connexion à un serveur

Nous allons exécuter le programme suivant qui va réaliser une connexion vers un serveur exécutant un programme qui renvoie systématiquement l'heure du serveur au client :

```
1  import java.net.*;
2  import java.io.*;
3
4  public class ClockClientSocket {
5
6  public static void main ( String [] args ) {
7
8  try {
9
10          Socket s = new Socket("localhost",8189);
11
12          BufferedReader in = new BufferedReader(
13                              new InputStreamReader(s.getInputStream()));
14          PrintWriter out = new PrintWriter( s.getOutputStream(),true);
15
16          String line;
17
18          out.println("quit");
```

```
19
20          while ( !(line=in.readLine()).toUpperCase().equals("STOP") )
21               System.out.println(line);
22
23          System.out.println("Fermeture de la connexion...");
24          s.close();
25
26          } catch (Exception e) {
27               System.out.println(e);
28               }
29  }
30  }
```

ClockClientSocket.java

- **Ligne 10** : Ouvre une connexion avec l'hôte local sur le port 8189
- **Ligne 12** : Crée le flux d'entrée de cette connexion comme un flux de caractères bufferisé.
- **Ligne 14** : Crée le flux de sortie de cette connexion comme un flux de caractères
- **Ligne 18** : Envoie la commande *quit* permettant de mettre fin à la connexion
- **Ligne 20** : Boucle lisant le flux entrant jusqu'à ce que le serveur renvoie la commande *STOP*.

Voici ce que le programme affichera :

```
1  Bienvenue sur ClockServer - Version 1.0 - VALET G ©
2  ***************************************************
3  Voici la date d'aujourd'hui :
4       Wed Oct 17 17:02:51 CEST 2001
5  Echo :quit
6  Fermeture de la connexion...
```

Pour mettre en place, le serveur, il vous faut exécuter la commande suivante :

```
1  java ClockServer
```

puis exécuter le programme client sur la même machine :

```
2  java ClockClientSocket
```

C.III - Implémenter un serveur

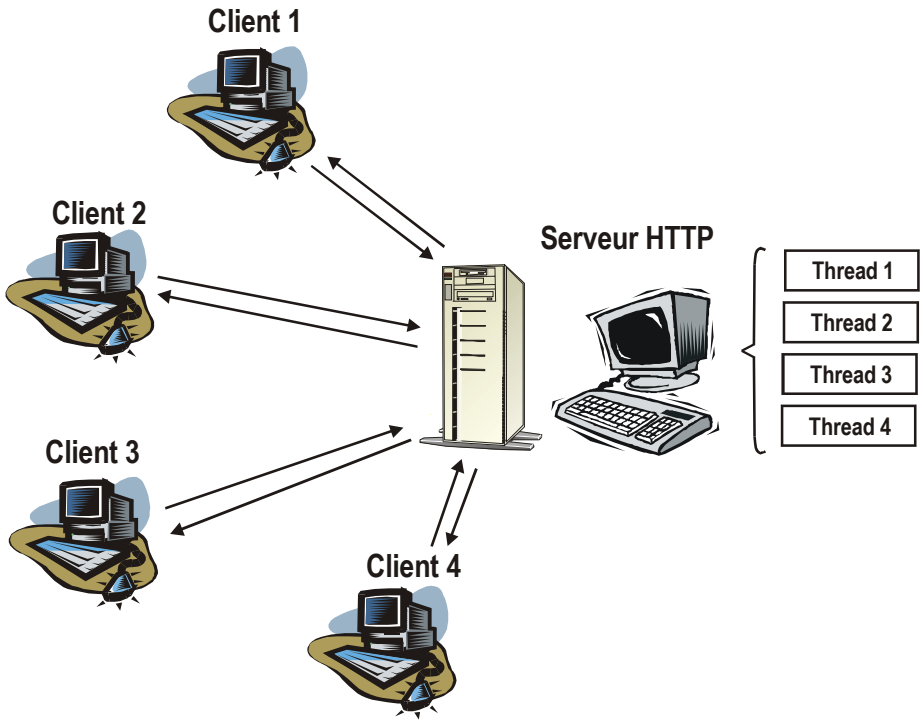
1) Types de serveurs

Maintenant que nous avons vu la manière de se connecter à un serveur, nous allons passer aux choses sérieuses en étudiant ce qui se passe côté serveur. En fait, un serveur est à l'écoute du client qui se connecte et réagit en fonction des données qu'il reçoit.

Il existe 2 types de serveurs : le serveur capable de gérer un seul client et le serveur multi-client.

Par exemple , un serveur *HTTP*[16] est à l'écoute des requêtes de plusieurs clients et leur renvoie les pages au format *HTML* demandées.

[16] http : HyperText Transfer Protocol



Exemple de serveur http servant plusieurs clients

Dans le schéma ci-dessus, chaque connexion vers le client doit s'exécuter dans un threads différents. Les clients peuvent donc effectuer des requêtes simultanément, le serveur saura gérer l'envoi et la réception de données sur ces clients.

2) Classe *ServerSocket*

2-a) Exemple d'un serveur mono-client

Cette classe permet de gérer l'envoi et la réception de données vers un client. Une fois activée, la socket serveur autorise le client à se connecter sur un port déterminé.

Pour créer une telle socket :

```
1  ServerSocket s = new ServerSocket (8189);
```

Ensuite, il faut créer une socket de type *Socket* permettant de gérer les flux vers et depuis le client :

```
2  Socket client = s.accept();
```

l'objet *client* ne sera crée que si un client se présente sur le port 8189.

ATTENTION : La méthode *accept()* est bloquante. Ce qui signifie qu'elle ne se termine que lorsqu'un client se connecte au port concerné. La conséquence directe est que le programme ne se terminera jamais sans l'intervention d'un utilisateur sur le serveur. La plupart du temps, il est indispensable de créer la socket serveur dans un thread séparé, de manière à ne pas trop monopoliser le serveur qui a sûrement d'autres tâches à effectuer. Nous verrons comment faire un peu plus loin dans ce chapitre.

Lorsque le client est connecté, le dialogue peut commencer entre le client et le serveur :

```
3  BufferedReader in = new BufferedReader (
4                          new InputStreamReader(client.getInputStream()));
5  PrintWriter out = new PrintWriter(client.getOutputStream());
```

out et sa méthode *print* ou *println* permettra d'envoyer des caractères au client, alors que la méthode *readLine* de *in* permettra une lecture bufferisée des requêtes du client.

Pour écrire au client :

```
1  out.println("Résultat de la requête");
```

```
2  out.flush();
```

Pour écouter le client :

```
1  String line = in.readLine();
```

Voici un programme d'exemple d'un serveur répondant au client connecté :

```
1  /*
2   * ClockSocket.java
3   *
4   * Created on 10 octobre 2001, 19:19
5   */
6
7  import java.net.*;
8  import java.io.*;
9  import java.util.*;
10 /**
11  *
12  * @author  Génael VALET
13  * @version 1.0 - Oct 2001
14  */
15 public class ServerSocketTest {
16
17     public static void main(String[] args) {
18
19  try  {
20     ServerSocket s = new ServerSocket(8189);
21     while (true) {
22         Socket client = s.accept();
23         BufferedReader in  = new BufferedReader(
24                   new InputStreamReader( client.getInputStream()));
25
26         PrintWriter out = new PrintWriter (
27                           client.getOutputStream() ,true);
28
29         out.println( "Bienvenue sur le serveur" );
30
31         String line;
32         boolean fin=false;
33         while ( (line=in.readLine())!=null && !fin) {
34             out.println("Echo :" + line);
35             if (line.trim().toUpperCase().equals("QUIT")) {
36                 fin = true;
37                 out.println("STOP");
38             }
39         }
40         client.close();
41     }
42   } catch (SocketException se) {
43        System.out.println("Interruption de type SocketException");
44   } catch (Exception e) {
45        System.out.println(e);
46        }
47   }
48 }
```

ServerSocketTest.java

- **Ligne 20 :** Création de l'objet *ServerSocket* sur le port 8189
- **Ligne 22 :** Création d'une connexion de type *Socket* entre le client et le serveur. Cette connexion sera utilisée par le serveur et le client pour lire et écrire.
- **Lignes 23-27** : *in* et *out* sont les flux permettant l'écriture et la lecture sur le client.
- **Ligne 29** : Envoi d'un message de bienvenue sur le client
- **Lignes 33 à 38** : Boucle de lecture de *in* jusqu'à ce que le client envoie la commande *QUIT*. Si le client envoie *QUIT*, le serveur envoie le message d'arrêt et quitte la boucle

- **Ligne 40** : Fermeture de la connexion vers le client

Pour tester ce programme :

compilez-le et exécutez-le puis ouvrez une invite de commande. Ouvrez une session telnet avec la commande

```
1  telnet localhost 8189
```

Le message de bienvenue s'affiche :

```
2  Bienvenue sur le serveur
```

Tapez n'importe quoi puis appuyez sur entrée :

```
3  Echo :Salut
```

Tapez* QUIT *puis Entrée :

```
4  Echo :QUIT
5  STOP
```

Appuyez une dernière fois sur ENTREE :

```
6  Perte de la connexion à l'hôte.
7
8  C:\>
```

La session telnet se termine et les connexions sont perdues. Le serveur est libre et une autre connexion client peut se produire.

Pour fermer le serveur, il suffit de fermer la fenêtre d'exécution

2-b) Exemple d'un serveur multi-clients

Le défaut du programme précédent est que seul un client peut se connecter en même temps. Dans le monde du client/serveur , il est très courant qu'un serveur puisse accepter plusieurs clients. Il faut donc que chaque client puisse être géré par le serveur dans un thread différent.

Voici le code source de la classe *MultiServerSocket* qui démarre une connexion vers le client dans un thread différent pour chaque nouveau client :

```
1  import java.net.*;
2  import java.io.*;
3  import java.util.*;
4
5  /**
6   *
7   * @author  Génael VALET
8   * @version 1.0 - Oct 2001
9   */
10
11 public class MultiServerSocket {
12
13  public static void main(String[] args) {
14
15   try  {
16    ServerSocket s = new ServerSocket(8189);
17    while(true) {
18        ThreadedSocket th = new ThreadedSocket(s.accept());
19        th.start();
20        }
21  } catch (SocketException se) {
```

```
22          System.out.println("Interruption de type SocketException");
23
24      } catch (Exception e) {
25          System.out.println(e);
26          }
27      }
28  }
```

MultiSocketServer.java

- **Ligne 16** : Création de la socket serveur sur le port 8189
- **Ligne 17** : Début de la boucle infinie
- **Ligne 18** : Création d'un thread différent pour chaque nouveau client. La méthode *accept()* attend la connexion d'un client.

Le code source de la classe *ThreadedSocket* est le suivant :

```
1  import java.net.*;
2  import java.io.*;
3  import java.util.*;
4  /**
5   *
6   * @author  Génael VALET
7   * @version 1.0 - Oct 2001
8   */
9  public class ThreadedSocket extends Thread {
10
11     private Socket client;
12
13     public ThreadedSocket(Socket client) {
14         this.client = client;
15     }
16
17     public void run() {
18
19       try  {
20          BufferedReader buffer = new BufferedReader(
21               new InputStreamReader(client.getInputStream()));
22          PrintWriter writer = new PrintWriter(
23                                    client.getOutputStream(),true);
24          writer.println( "Bienvenue sur le serveur" );
25
26          String line;
27          boolean fin=false;
28          while ( (line=buffer.readLine())!=null && !fin) {
29             writer.println("Echo :" + line);
30             if (line.trim().toUpperCase().equals("QUIT")) {
31                fin = true;
32                writer.println("STOP");
33               }
34         }
35         client.close();
36       } catch (SocketException se) {
37             System.out.println("Interruption de type SocketException");
38       } catch (Exception e) {
39             System.out.println(e);
40       }
41  }
42 }
```

ThreadedSocket.java

- **Ligne 17** : Méthode *run* du thread qui sera appelé par la méthode *start()*
- **Lignes 20-23** : Création des flux entrant et sortant de la socket
- **Ligne 24** : Envoi du message de bienvenue au client
- **Ligne 28** : Boucle de lecture des informations en provenance du client
- **Ligne 30** : Si le client renvoie la commande QUIT, la boucle se termine car la variable *fin* sera égale à *true*.

Pour exécuter plusieurs clients, il vous suffit de démarrer plusieurs sessions telnet comme décrit plus haut et de constater que le serveur répond à tous les clients .

C.IV - Connexion à des serveurs http

Sur Internet, un client désireux d'obtenir les pages web d'un site quelconque ne se soucie guère de la plate-forme qui va lui renvoyer ces pages. Ce dont le client peut être sûr, c'est que quelque soit le serveur atteint, il doit répondre aux requêtes de manière

quasi identique. C'est pourquoi les organismes qui contrôlent l'évolution d'internet se sont mis d'accord pour créer un protocole de communication suivant les règles dictées par une RFC[17]

1) Classe URL

1-a) Composition d'une URL

La classe *URL*[18] permet de manipuler des adresses Internet sur différents protocoles et de récupérer des informations sur des parties de l'adresse. En effet une adresse Internet est constituée :

- Du protocole (HTTP, FTP …)
- Du nom DNS du domaine (comme *www.gita.greta.fr*)
- Du chemin de la ressource (comme */formations*)
- Du nom de la ressource (comme *fiche.asp*)
- Des informations sur la requête (comme *id=76)*
- Du nom du signet au sein de la page (comme *dates*)

Une telle adresse ressemblerait à ceci :

```
1  http://www.gita.greta.fr/formations/fiche.asp?id=76#dates
```

1-b) Méthode openConnection

La classe *URL* permet aussi, grâce à sa méthode *openConnection* d'effectuer une connexion vers l'adresse qu'elle représente.

```
2  URL url = new URL ("http://www.gita.greta.fr/index.htm");
3  URLConnection con = url.openConnection();
```

Les constructeurs de la classe *URL* génèrent une exception de type *MalformedURLException* si le protocole spécifié n'existe pas ou si aucun protocole n'est spécifié. Il faut donc lever cette exception systématiquement.

1-c) Méthode openStream

Cette méthode permet d'ouvrir une connexion et de récupérer un flux d'entrée de données depuis le serveur correspondant à cette connexion :

```
4  InputStream in = url.openStream();
```

Cette méthode est équivalente à :

```
1  InputStream in = url.openConnection().getInputStream();
```

2) Classe URLConnection

2-a) Instanciation

Cette classe abstraite sera très utile pour en connaître un peu plus sur le serveur qui fourni les données. En effet, il peut être utile de connaître la version de http supportée par le serveur de manière à lui présenter des requêtes bien formulées.

[17] RFC – Request For Comment – Groupe de travail régissant les normes des différents protocoles informatiques
[18] URL – Uniform Ressource Locator – Adresse Internet

Cette classe peut se connecter à des serveurs sur plusieurs types de protocoles (http, FTP, gopher…). Pour effectuer une connexion vers un serveur http, il vaut mieux utiliser la classe *HttpURLConnection* .

Pour effectuer une connexion vers un serveur, il faut respecter l'ordre des opérations. Tout d'abord, il vous faut créer une connexion avec, si besoin est, la classe *URL* vue précédemment.

```
1  URL url = new URL ("http://www.gita.greta.fr/index.htm");
2  URLConnection con = url.openConnection();
```

2-b) setDoInput et setDoOutput

Ces méthodes sont utilisées pour déterminer si vous allez uniquement recevoir des données ou si vous allez en envoyer. Elles vont fixer les champs booléens *doInput* et *doOutput*.

Par défaut, toute nouvelle connexion fixe le champ *doInput* à la valeur *true* et celle de *doOutput* à *false*.

2-c) *Méthode* setIfModifiedSince

L'appel à cette méthode va déterminer si il est nécessaire de recharger l'objet souhaité selon des critères de temps. En effet, le paramètre à transmettre avec la méthode représente le nombre de millisecondes écoulées depuis le 1er Janvier 1970. Si l'objet n'a pas été modifié depuis la date représentée par ce paramètre, alors l'objet ne sera pas rechargé.

2-d) Méthode setUseCaches

Détermine l'utilisation du cache en fixant le champ booléen *useCaches* à *true* pour activer le cache ou à *false* pour recharger l'objet quoiqu'il arrive. Cette méthode n'est utile que pour les *applets*

2-e) Méthode connect

Cette méthode ouvre une socket vers le serveur et demande à celui-ci les informations d'en-tête de l'objet demandé. Ces informations représentent les propriétés de l'objet comme , la date de dernière modification, le type de contenu (*content type*), la taille de l'objet (*content length*), le type de codage (*content encoding*), ou encore la date d'expiration (*expiration*)

Les méthodes suivantes serviront à obtenir ces informations :

Méthodes d'accès aux informations d'en-tête	
String	**getContentEncoding**()
int	**getContentLength**()
String	**getContentType**()
long	**getDate**()
long	**getExpiration**()

long	getLastModified()

2-f) Méthode getHeaderFieldKey

Voilà comment obtenir les informations d'en-tête du serveur. Voici la signature de cette méthode :

```
1  String getHeaderFieldKey( int n )
```

n représente le nième champ d'en-tête. Il n'est pas possible de savoir combien de champs le serveur contient. Cette méthode renvoie *null* lorsqu'il n'y pas plus de champs.

Voici un programme permettant d'obtenir ces informations auprès d'un serveur :

```
2
3  import java.net.*;
4  import java.io.*;
5
6  public class URLConnectionTest {
7
8          public static void main (String [] args) {
9
10                 try {
11                     URL url = new URL("http://www.gita.greta.fr");
12                     URLConnection con = url.openConnection();
13                     int n = 1;
14                     String key;
15                     con.connect();
16                     while ((key = con.getHeaderFieldKey(n))!= null) {
17                            String valeur = con.getHeaderField(n++);
18                            System.out.println(key + ":" + valeur);
19                         }
20                 } catch (MalformedURLException mie) {
21                    System.out.println("Url incorrecte");
22                 } catch ( IOException ie) {
23                    System.out.println("Problème d'entrées sorties");
24                    System.out.println(ie);
25                    }
26         }
27 }
```

URLConnectionTest.java

Voici la réponse obtenue du serveur :

```
1  Server:Microsoft-IIS/4.0
2  Content-Location:http://www.gita.greta.fr/index.htm
3  Date:Thu, 18 Oct 2001 11:10:56 GMT
4  Content-Type:text/html
5  Accept-Ranges:bytes
6  Last-Modified:Tue, 03 Jul 2001 10:01:04 GMT
7  ETag:"3a837c12a73c11:396c55"
8  Content-Length:21978
```

3) Lire les données provenant de serveurs

Il ne nous reste plus qu'a lire les données envoyées par les serveurs en récupérant un flux d'entrée par la méthode *getInputStream* de la classe *URLConnection* :

```
1  import java.net.*;
2  import java.io.*;
3
```

```
4  public class URLConnectionReadTest {
5
6          public static void main (String [] args) {
7
8                  try {
9                    URL url = new URL("http://homewk");
10                   URLConnection con = url.openConnection();
11                   BufferedReader in = new BufferedReader(
12                        new InputStreamReader(con.getInputStream()));
13
14                   String line;
15                   while ( (line=in.readLine())!=null)
16                          System.out.println(line);
17
18                 } catch (MalformedURLException mie) {
19                     System.out.println("Url incorrecte");
20                 } catch ( IOException ie) {
21                     System.out.println("Problème d'entrées sorties");
22                     System.out.println(ie);
23                        }
24         }
25 }
```

URLConnectionReadTest.java

4) Ecrire des données vers un serveur

Lorsque vous saisissez des données dans un formulaire, les données sont envoyées selon deux méthodes différentes. La méthode *GET* encapsule les données dans l'URL alors que la méthode *POST* envoie les données dans une requête http.

4-a) Méthode GET

Aujourd'hui, la méthode *GET* est de moins en moins utilisée, sauf lorsque la taille des données envoyées et leur confidentialité ne sont pas un problème.

Lorsque vous utilisez le moteur de recherche *Altavista* pour trouver des ressources pour Java et que vous appuyez sur Recherche, le navigateur envoie des informations au serveur. Voici l'URL composée à ce moment :

```
1  http://fr.altavista.com/q?pg=q&q=java&kl=fr&what=fr&mm=1&search.x=26
2                                                        &search.y=1
```

La 1ère partie de l'URL concerne le nom de domaine et donc celui du serveur qui héberge les pages. La 2ème partie , en gras ci-dessus correspond aux données envoyées par la méthode *GET*.

Chaque champ du formulaire est séparé par des &. Par exemple , le champ *kl* vaut *fr* , ce qui signifie que tous les résultats de la recherche devront être de langue française.

4-b) Méthode POST

La méthode *POST* réalise la même chose mais l'URL renvoyée est plus simple puisque les données sont cachées au sein des requêtes *http*.

4-c) Ecrire des données

En Java, nous allons envoyer des données au serveur du GITA par la méthode *POST*. Le but étant de connaître le nombre de visiteurs à l'instant sur le site, nous allons effectuer une requête auprès du serveur suivant :

```
1  http://www.gita.greta.fr/getinfo.asp
```

avec les informations transmises par *POST*

```
2  visit=true
```

Ce qui signifie que les données de retour devront contenir le nombre de visiteurs présents sur le site.

Il faut commencer par créer correctement un objet *URLConnection* puis le paramétrer pour qu'il accepte un flux sortant. Cela passe donc par la création d'un flux de type *PrintWriter* obtenu grâce à la méthode *getOutputStream()* de la classe *URLConnection*:

```
1  URL url = new URL("http://www.gita.greta.fr/getinfo.asp");
2  URLConnection con = url.openConnection();
3
4  con.setDoOutput(true); // Autorise le flux sortant
5
6  PrintWriter out = new PrintWriter(con.getOutputStream());
```

La deuxième étape consiste à écrire les informations devant être transmises par la méthode *POST*

```
7  out.print("visit=true");
8  out.close();
```

Il ne reste plus qu'à lire les données renvoyées par le serveur par le biais d'un flux de caractères bufferisé et d'une boucle s'arrêtant lorsqu'il n'y a plus de données :

```
9  BufferedReader in = new BufferedReader( new InputStreamReader(
10                                          con.getInputStream()));
11
12 String line;
13 while ( (line=in.readLine())!=null)
14         System.out.println(line);
```

Nous obtenons alors le résultat suivant à l'écran :

```
1  Le nombre de visiteurs présents sur le site est de 12
```

Voici le code complet du programme :

```
1  import java.net.*;
2  import java.io.*;
3
4  public class URLConnectionWriteTest {
5
6          public static void main (String [] args) {
7
8           try {
9               URL url = new URL("http://www.gita.greta.fr/getinfo.asp");
10              URLConnection con = url.openConnection();
11              con.setDoOutput(true);
12              PrintWriter out = new PrintWriter(con.getOutputStream());
13              out.print("visit=true");
14              out.close();
15
16              BufferedReader in = new BufferedReader(
17                    new InputStreamReader(con.getInputStream()));
18
19              String line;
20              while ( (line=in.readLine())!=null)
21                    System.out.println(line);
22
23              } catch (MalformedURLException mie) {
24                 System.out.println("Url incorrecte");
25              } catch ( IOException ie) {
26                 System.out.println("Problème d'entrées sorties");
27                 System.out.println(ie);
28                 }
29         }
30 }
```

URLConnectionWriteTest.java

5) Classe URLEncoder

Les caractères ASCII doivent codés différemment dans une URL. En effet, de manière à éviter les caractères qui ne sont pas compris par tous les serveurs. Voici les règles appliquées aux caractères :

- Tous les caractères allant de '*a*' à '*z*', de '*A*' à 'Z', de '*0*' à '*9*', et ".", "-", "*", "_" sont inchangés.
- Le caractère d'espacement ' ' est converti en un signe '+'.
- Tous les autres caractères sont converti en 3 caractères %xy ou xy représente la valeur héxadécimale ASCII du caractère.

L'adresse suivante :

```
1  http://www.gita.greta.fr/fiche.asp?id=12
```

devient

```
1  http%3A%2F%2Fwww.gita.greta.fr%2Ffiche.asp%3Fid%3D1
```

Ce résultat est obtenu en appelant la méthode statique *encode(String s)* de la classe *URLEncoder* :

```
1  String normal = "http://www.gita.greta.fr/fiche.asp?id=12";
2  String coded = URLEncoder.encode(normal);
```

6) Classe URLDecoder

Elle fonctionne de la même façon que *URLEncoder* mais effectue le décodage. La méthode à appeler est *decode*.

D. Bases de données avec JDBC

D.I - Introduction

Incontournable dans le monde de l'informatique, les bases de données sont le moteur de toute application ayant à traiter et sauvegarder beaucoup d'informations. Java se devait d'implémenter une interface digne de ce nom et c'est chose faite depuis l'été 1996 avec le kit JBDC[19]. Depuis, le travail acharné de *Sun* a permis l'intégration complète des bases de données dans les kits fournis par l'éditeur. D'abord avec l'arrivée de JDBC 1 puis JDBC 2.

Architecturé autour du langage SQL[20] , JDBC possède des atouts qui font de Java, le langage idéal pour la connectivité aux bases de données. Voici les principaux avantages de JDBC :

- Les programmes développés en Java avec JDBC sont entièrement portables et peuvent fonctionner sur n'importe quel ordinateur disposant de l'environnement Java
- Quelle que soit le type de base de données, les programmes écrits avec JBDC fonctionneront , quasiment sans modification du code. Un programme écrit pour accéder aux données sur un serveur *SQL Server* de *Microsoft* fonctionnera également sur base *Oracle*
- JDBC propose plusieurs niveaux de *drivers* JDBC (voir plus loin dans ce cours) permettant d'offrir une compatibilité maximum avec des application existantes tout en acceptant la présence de code non Java (C++ par exemple). C'est le cas de l'interface *ODBC*[21]

[19] Java DataBase Connectivity
[20] Structured Query Language
[21] Open DataBase Connectivity

D.II - Présentation

1) Schéma

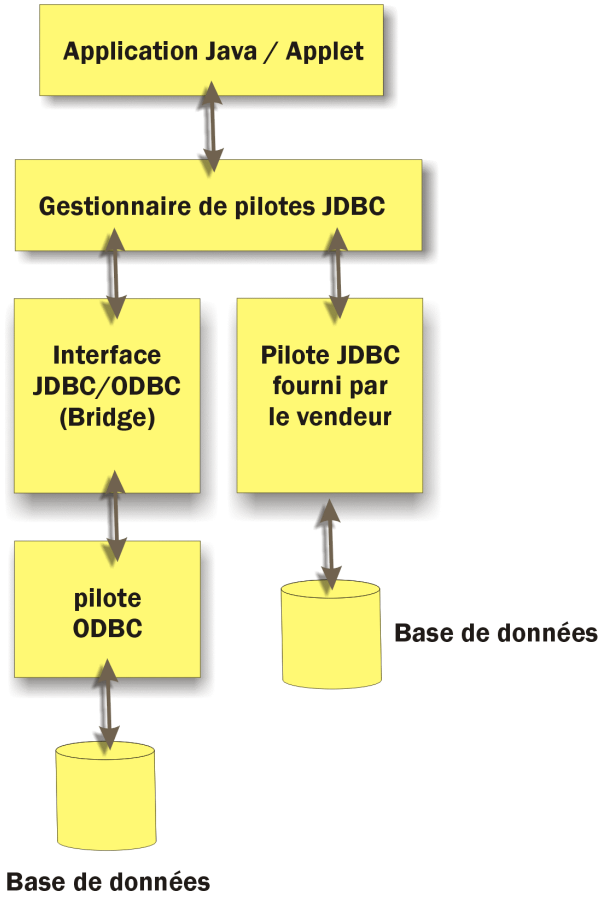


Schéma d'ensemble de l'environnement JDBC

2) Le gestionnaire de pilotes (JDBC Driver Manager)

Il s'agit de l'organe principal de JDBC. Il coordonne les différents pilotes, ce qui permet à un même programme d'accéder à plusieurs bases de marque différente. La classe *DriverManager* fourni dans le package *java.sql* s'exécute dans un environnement statique, ce qui lui permet de gérer efficacement les connections aux bases de données. C'est également lors de l'utilisation de cette classe, que les différentes pilotes sont enregistrés au près du gestionnaire.

3) Les pilotes

Il existe plusieurs types de pilote JDBC :

- Pilote de type I
- Pilote de type II
- Pilote de type III
- Pilote de type IV

Ceci permet, comme nous allons le voir, de s'adapter à tout type d'infrastructure existante.

3-a) Pilote de type I

- Ce type de pilote utilise l'interface *ODBC* qui , dans le monde *Windows* permet de se connecter à tout type de base de données.
- Les appels de JDBC sont convertis en appels ODBC , ce qui en terme de performance, n'est pas toujours l'idéal
- Java utilise une librairie native écrite en C ce qui exclue de le faire fonctionner au sein d'une applet
- Il est fourni avec le kit Java par l'intermédiaire de la classe *sun.jdbc.odbc.JdbcOdbcDriver*

Voici le code nécessaire pour enregistrer ce type de pilote auprès du gestionnaire :

```
1  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

3-b) Pilote de type II

- Une partie de ce pilote est écrit en Java et l'autre partie dans le langage propriétaire natif de la base de données.
- Les appels JDBC sont alors convertis en appels vers les API natives de la base de données
- Cela implique que pour accéder à ce type de pilote, il faut installer sur la machine les API natives
- Les appels au code natif exclue d'utiliser ce type de pilote au sein d'une applet
- Ces pilotes sont généralement payants et écrits par des sociétés tierces ou par l'éditeur de la base de données
- La portabilité du code n'est plus assurée à cause de la partie langage natif

3-c) Pilote de type III

- Pilote 100% Java utilisant une API réseau avec un programme appelé *middleware*[22] , chargé de communiquer avec la base de données.
- Ce pilote est donc totalement indépendant de la base de données, ce qui constitue une bonne solution performante et portable
- Comme aucun appel à un langage natif n'est effectué, le pilote de type III est utilisable avec les applets (à condition que l'applet soit stockée sur le même serveur que la base données)

3-d) Pilote de type IV

- Pilote 100% Java qui se connecte directement à la base de données sans la nécessité d'un programme *middleware*
- Les requêtes sont alors directement traduites et envoyés grâce à l'utilisation des *sockets*[23]
- Le pilote de type IV est utilisable avec les applets (à condition que l'applet soit stockée sur le même serveur que la base données).

Pour trouver un pilote correspondant à votre base de données, utilisez l'adresse :

```
1  http://java.sun.com/products/jdbc
```

[22] Programme intermédiaire agissant comme un traducteur entre une application cliente et une application serveur. C'est un des éléments de l'architecture 3 tiers.
[23] Elément de programmation permettant d'effectuer une connexion à un hôte distant via le réseau

Sachez que la plupart des éditeurs de bases de données fournissent des pilotes de type III ou IV , ce qui constitue un gage de performance et de compatibilité avec le langage *SQL*.

D.III - Mise en œuvre de JDBC

1) Introduction

La mise en œuvre de JDBC suppose que vous ayez déjà des connaissances minimales sur le langage SQL et la structure d'une base de données. Par exemple, si le code SQL suivant vous laisse de marbre, il faut vous procurer immédiatement un ouvrage traitant la question :

```
1  SELECT * FROM Clients WHERE nom='DUPONT' ORDER BY nom
```

2) Se procurer un pilote

En effet, la 1ère tâche à effectuer est de se procurer le pilote Java de la base de données avec laquelle vous souhaitez travailler. Pour cela, vous pouvez vous aider de l'URL suivante :

```
1  http://java.sun.com/products/jdbc
```

Pour travailler avec la passerelle *JDBC/ODBC* , pas besoin de pilote, il est intégré dans le kit Java de Sun.

La plupart du temps, le pilote se présentent sous la forme d'un fichier *JAR*[24]. Ce fichier doit être positionné dans un répertoire liste dans la variable *CLASSPATH*[25]. Le nom du package fourni correspond toujours à un nom faisant partie d'un espace de noms, comme on peut les trouver sur Internet :

```
1  com.mysql.jdbc
```

Ci-dessus, un exemple de nom de package pour le driver de la base de données *MySql* que vous pouvez vous procurer gratuitement à l'adresse :

```
1  http://www.mysql.com/products/connector-j
```

3) Importer le package nécessaire

Votre programme doit comporter l' import nécessaire à l'utilisation de JDBC :

```
1  import java.sql.*;
```

Inutile d'importer le package contenant le pilote. Les bonnes classes seront utilisées lors de l'enregistrement du pilote par le gestionnaire de pilotes.

[24] Java ARchive
[25] Classpath : Variable d'environnement contenant la liste des répertoires concernés par Java susceptible de contenir des packages ou des archives utiles pour la compilation

4) Enregistrer le pilote

L'appel à la méthode statique *forName* de la classe *Class* permet l'enregistrement. Voici un exemple permettant d'enregistrer un pilote *MySql* :

```
1  Class.forName("com.mysql.jdbc.Driver");
```

Voici le même exemple pour travailler avec une base de données *Access* via l'interface *OBDC* :

```
1  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

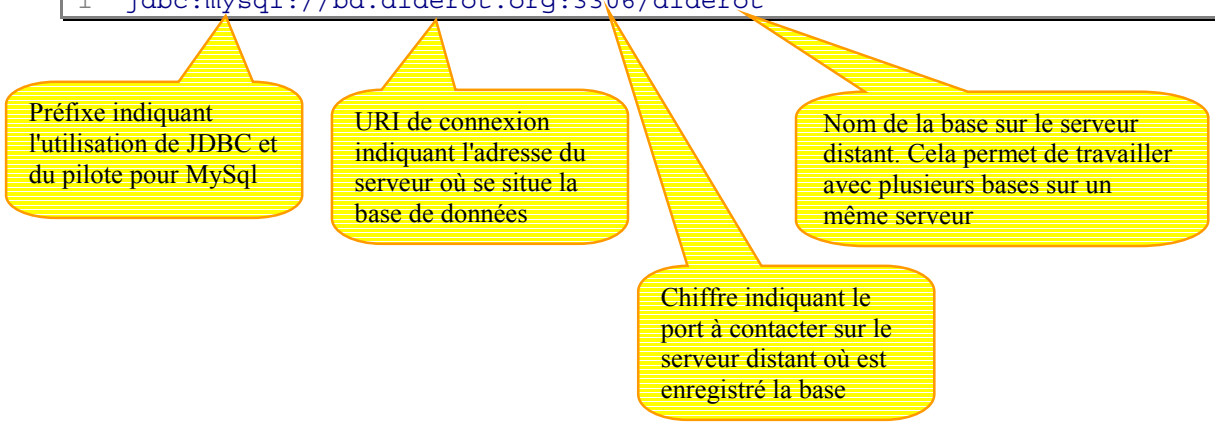
L'exemple ci-dessus permet de travailler avec n'importe qu'elle base de données (Et pas *Access* seulement) à partir du moment où l'on possède le pilote ODBC approprié. L'utilisation du pont JDBC/ODBC ne fait que reporter le problème du pilote vers ODBC.

5) Etablir la connexion à la base de données

5-a) URL de connexion

La base de données peut-être atteinte grâce à une *URL*[26] de connexion. Cette *URL* est composée de plusieurs parties :

```
1  jdbc:mysql://bd.diderot.org:3306/diderot
```



Voici un exemple d'URL de connexion utilisant le pont ODBC/JDBC :

```
1  jdbc:odbc:nom_de_la_base
```

Le pont JDBC/OBDC suppose que la base et ODBC sont installés sur la même machine locale. C'est pourquoi, dans l'URL proposée ci-dessus, ne figure pas le nom d'un serveur distant mais le nom de la source de données locale.

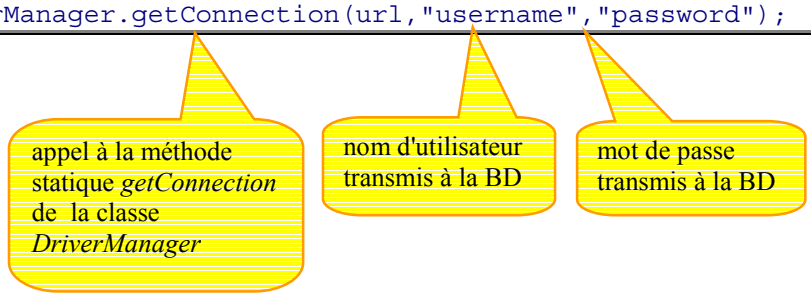
[26] Uniform Ressource Locator

5-b) Méthode getConnection

Rappelons-nous que le gestionnaire de pilote (Driver Manager) est chargé de nous fournir les connexions par l'intermédiaire d'un objet, instance de la classe *Connection*.

Voici un exemple d'ouverture d'une connexion vers une base de données

```
1  String url = "jdbc:mysql://bd.diderot.org:3306/diderot";
2
3  Connection c;
4  c = DriverManager.getConnection(url,"username","password");
```



Il existe plusieurs méthodes *getConnection* dont voici les signatures :

Méthodes getConnection de DriverManager	
`static Connection`	**`getConnection`**`(String url)` Essaie d'effectuer une connection vers la base spécifiée par l'URL
`static Connection`	**`getConnection`**`(String url, Properties info)` Essaie d'effectuer une connection vers la base spécifiée par l'URL. *info* est une instance de la classe *Properties* qui contient les différents paramètres sous forme de couple nom/valeur correspondant aux paramètres d'ouverture de connexion (user, password, ….)
`static Connection`	**`getConnection`**`(String url, String user, String password)` Essaie d'effectuer une connection vers la base spécifiée par l'UR avec le nom d'utilisateur et le mot de passe fournis en paramètres

Lors de l'appel à *getConnection* , le gestionnaire de pilotes essaie tous les pilotes qui ont été enregistrés par l'appel à *Class.forName(…)* jusqu'à ce qu'il obtienne une connexion vers la base désignée par l'URL. Cela signifie que si plusieurs pilotes enregistrés permettent de se connecter à la base, c'est le 1er pilote enregistré qui aura gain de cause.

6) Créer une zone de description de requête (Statement)

La zone de description de requête est implémenté grâce à l'interface *Statement* du package *java.sql*. Il s'agit de créer un espace permettant d'exécuter une ou plusieurs requêtes SQL.

Il existe 3 types de Statement :

- **Statement :** Permet d'effectuer des requêtes simples comme la lecture, la modification ou l'ajout de quelques enregistrements.
- **PreparedStatement :** Permet d'effectuer des requêtes pré compilées , gages de performances plus élevées, surtout lorsque ces requêtes se répètent plusieurs fois. Ces requêtes peuvent également faire passer des paramètres d'exécution

- **CallableStatement :** Permet de faire appel à *des procédures stockées*[27]

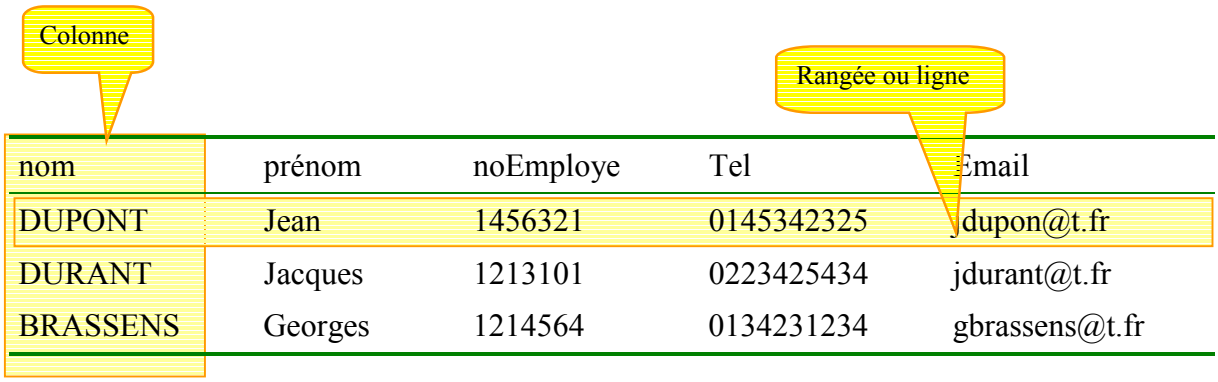
Pour obtenir un objet *Statement* , il faut appeler la méthode *createStatement* de la classe *Connection* :

```
1  String url = "jdbc:mysql://bd.diderot.org:3306/diderot";
2
3  Connection c;
4  c = DriverManager.getConnection(url,"username","password");
5
6  Statement st = c.createStatement();
```

A ce stade, le *Statement* est initialisé mais aucun requête vers la BD n'est effectuée.

7) Exécuter une requête

Voilà le moment tant attendu où nous allons pouvoir obtenir des données. Pour cela, il faut utiliser la classe *ResultSet* , chargée d'organiser les données venant de la base selon des lignes et des colonnes. Une colonne (Column) correspond à toutes les valeurs d'un champ de la table. Une rangée (*Row*) correspond à la valeur de chaque champ de la table:



nom	prénom	noEmploye	Tel	Email
DUPONT	Jean	1456321	0145342325	jdupon@t.fr
DURANT	Jacques	1213101	0223425434	jdurant@t.fr
BRASSENS	Georges	1214564	0134231234	gbrassens@t.fr

On obtient l'objet *ResultSet* en appelant la méthode *executeQuery* de l'objet *Statement* :

```
1  String url = "jdbc:mysql://bd.diderot.org:3306/diderot";
2
3  Connection c;
4  c = DriverManager.getConnection(url,"username","password");
5
6  Statement st = c.createStatement();
7
8  String strSql = "SELECT * FROM clients WHERE ville='paris' ";
9  ResultSet rs = st.executeQuery(strSql);
```

Lance l'exécution de la requête et renvoie l'objet *ResultSet* qui permettra de lire et d'écrire des données dans la base

Prépare une chaîne de caractère contenant la requête qui permet d'obtenir tous les clients résidant à Paris

[27] Procédures complexes programmées par l'administrateur de la base de données pour faciliter la manipulation de données par l'utilisateur. Ces procédures sont comme des fonctions appelées avec , si nécessaire, des paramètres d'exécution définis par l'utilisateur.

8) Lire des données dans un *ResultSet*

Pour exploiter les données contenues dans un *ResultSet* , il n'est pas nécessaire de connaître leur nature. Néanmoins, cela facilité la manipulation des données. En fait, chaque type SQL, est associé avec un type primitif Java, ce qui devrait nous permettre de stocker les données dans des variables Java.

Voici un tableau de correspondance entre les types SQL et les méthodes Java de *ResultSet* permettant de lire ces données :

TYPE SQL	*Méthodes de ResultSet pouvant être utilisée*
TINYINT:	**getByte (usage recommandé)** Peut-être lu aussi par getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getString, getObject
SMALLINT:	**getShort (usage recommandé)** Peut-être lu aussi par getByte, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getString, getObject
INTEGER:	**getInt (usage recommandé)** Peut-être lu aussi par getByte, getShort, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getString, getObject
BIGINT:	**getLong (usage recommandé)** Peut-être lu aussi par getByte, getShort, getInt, getFloat, getDouble, getBigDecimal, getBoolean, getString, getObject d
REAL:	**getFloat (usage recommandé)** Peut-être lu aussi par getByte, getShort, getInt, getLong, getDouble, getBigDecimal, getBoolean, getString, getObject
FLOAT:	**getDouble (usage recommandé)** Peut-être lu aussi par getByte, getShort, getInt, getLong, getFloat, getBigDecimal, getBoolean, getString, getObject
DOUBLE:	**getDouble (usage recommandé)** Peut-être lu aussi par getByte, getShort, getInt, getLong, getFloat, getBigDecimal, getBoolean, getString, getObject
DECIMAL:	**getBigDecimal (usage recommandé)** Peut-être lu aussi par getByte, getShort, getInt, getLong, getFloat, getDouble, getBoolean, getString, getObject
NUMERIC:	**getBigDecimal (usage recommandé)** Peut-être lu aussi par getByte, getShort, getInt, getLong, getFloat, getDouble, getBoolean, getString, getObject
BIT:	**getBoolean (usage recommandé)** Peut-être lu aussi par getByte, getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getString, getObject
CHAR:	**getString (usage recommandé)** Peut-être lu aussi par getByte, getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getDate, getTime, getTimestamp, getAsciiStream, getUnicodeStream, getObject
VARCHAR:	**getString (usage recommandé)** Peut-être lu aussi par getByte, getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getDate, getTime, getTimestamp, getAsciiStream, getUnicodeStream, getObject

LONGVARCHAR:	**getAsciiStream, getUnicodeStream (usage recommandé pour toutes)** Peut-être lu aussi par getByte, getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getString, getDate, getTime, getTimestamp, getObject
BINARY:	**getBytes (usage recommandé)** Peut-être lu aussi par getString, getAsciiStream, getUnicodeStream, getBinaryStream, getObject
VARBINARY:	**getBytes (usage recommandé)** Peut-être lu aussi par getString, getAsciiStream, getUnicodeStream, getBinaryStream, getObject
LONGVARBINARY:	**getBinaryStream (usage recommandé)** Peut-être lu aussi par getString, getBytes, getAsciiStream, getUnicodeStream, getObject
DATE:	**getDate (usage recommandé)** Peut-être lu aussi par getString, getTimestamp, getObject
TIME:	**getTime (usage recommandé)** Peut-être lu aussi par getString, getTimestamp, getObject
TIMESTAMP:	**getTimestamp (usage recommandé)** Peut-être lu aussi par getString, getDate, getTime, getObject

Tableau de correspondance SQL > Type Java

Le code suivant permet de lire un nom et le prénom dans une base de données de clients :

```
1  String url = "jdbc:mysql://bd.diderot.org:3306/diderot";
2
3  Connection c;
4  c = DriverManager.getConnection(url,"username","password");
5
6  Statement st = c.createStatement();
7
8  String strSql = "SELECT * FROM clients WHERE ville='paris' ";
9  ResultSet rs = st.executeQuery(strSql);
10
11 String nom = rs.getString("nom");
12 String prenom = rs.getString(2);
```

L'appel à *getString* suppose que l'on connaît le nom du champ.

Ici, l'appel à *getString* suppose que l'on connaît le numéro de colonne du champ (Ici prenom est la 2ème colonne dans la table

D.IV - Le traitement des exceptions

Evidemment, lorsque vous vous connectez à une base de données, vous n'êtes jamais sûr que :

- La base est accessible
- Vous êtes bien autorisé à accéder à la base
- La requête SQL que vous avec formulé est correcte
- Le pilote est bien enregistré et opérationnel
- …

Donc, pour éviter de bloquer votre programme Java dans le cas où une erreur se produit, vous aurez recours à l'utilisation des exceptions, d'autant plus que cela vous est imposé de la part du compilateur. En effet, toutes les opérations relatives aux bases de données sont susceptibles de renvoyer une exception de type *SQLException*.

Vos programmes devront donc être conçus pour prévoir les actions à mener au cas où les opérations vers les bases de données se déroulent mal.

Voici un exemple complet de programme permettant de se connecter à une base de données avec la gestion des exceptions :

```
13 try {
14          Class.forName("com.mysql.jdbc.Driver");
15
16          } catch (ClassNotFoundException ex) {
17             System.out.println("Pilote JDBC non enregistré");
18          }
19
20 String url = "jdbc:mysql://bd.diderot.org:3306/diderot";
21
22 try {
23          Connection c;
24          c = DriverManager.getConnection(url,"username","password");
25
26          Statement st = c.createStatement();
27
28          String strSql = "SELECT * FROM clients WHERE ville='paris' ";
29          ResultSet rs = st.executeQuery(strSql);
30
31          rs.next() ;
32
33          String nom = rs.getString("nom");
34          String prenom = rs.getString(2);
35
36   } catch (SQLException ex) {
37          System.out.println("Erreur SQL :"+ex.getMessage());
38          }
39
```

Se produit si le pilote désigné n'est pas présent dans le système Java

Se produit si l'une des instructions précédentes s'est mal terminée.

D.V - Caractéristiques d'un ResultSet

Il existe plusieurs types de *ResultSet*. Cette variété de types permet de proposer aux programmeurs des choix en terme de performance et de "fraîcheur" des données. En effet, maintenir un *ResultSet* à jour en rendant visible les modifications effectuées par d'autres utilisateurs est assez coûteux en ressources systèmes. Par contre, un *ResultSet* moins performant permet de moins charger le moteur de la base de données.

Le choix d'un type de *ResultSet* se détermine au moment de la création de l'objet *Statement*. Vous devez fournir à la méthode *createStatement* de la classe *Connection* les arguments nécessaire à ce choix.

1) *ResultSet* de type FORWARD_ONLY

C'est le type par défaut si vous ne spécifiez pas le contraire. Ce type permet de parcourir un *ResultSet* en avant seulement. Ce qui signifie, qu'il ne vous sera pas possible de retourner en arrière pour relire un enregistrement :

Voici un exemple :

```
40 Connection c;
41 c = DriverManager.getConnection(url,"username","password");
42
43 Statement st = c.createStatement(ResultSet.FORWARD_ONLY,
44                                   ResultSet.CONCUR_READ_ONLY);
```

L'argument est un champ statique de la classe *ResultSet*

Spécifie que le *ResultSet* ne peut pas être mis à jour

2) *ResultSet* de type TYPE_SCROLL_INSENSITIVE

Pour ce type, il vous sera possible de parcourir le *ResultSet* dans n'importe quel sens, par contre les modifications effectuées par d'autres utilisateurs de la base de données ne seront pas répercutées.

Voici l'exemple d'un *ResultSet* non modifiable, de type insensible aux modifications faîtes par ailleurs mais que l'on peut parcourir dans tous les sens :

```
1  Connection c;
2  c = DriverManager.getConnection(url,"username","password");
3
4  Statement st = c.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
5                                   ResultSet.CONCUR_READ_ONLY);
```

3) *ResultSet* de type TYPE_SCROLL_SENSITIVE

Ce type permet de voir les modifications effectués par d'autres utilisateurs. Naturellement, ce type de *ResultSet* oblige JDBC à obtenir des informations à jour en permanence. La conséquence est qu'un trop grand nombre de connexion associé à des *ResultSet* de type *TYPE_SCROLL_SENSITIVE* peut engendrer une surcharge du moteur de la base de données.

Voici l'exemple d'un *ResultSet* non modifiable, de type sensible aux modifications faîtes par ailleurs et que l'on peut parcourir dans tous les sens :

```
1  Connection c;
2  c = DriverManager.getConnection(url,"username","password");
```

```
3
4  Statement st = c.createStatement(ResultSet.TYPE SCROLL SENSITIVE,
5                                   ResultSet.CONCUR_READ_ONLY);
```

4) Modifier des enregistrements

Pour pouvoir modifier des enregistrements, il faut le préciser lors de l'appel à la méthode *createStatement* de la classe *Connection*.

Voici l'exemple d'un *ResultSet* modifiable, de type sensible aux modifications faîtes par ailleurs et que l'on peut parcourir dans tous les sens :

```
1  Connection c;
2  c = DriverManager.getConnection(url,"username","password");
3
4  Statement st = c.createStatement(ResultSet.TYPE SCROLL SENSITIVE,
5                                   ResultSet.CONCUR_UPDATABLE);
```

ATTENTION : Certaines bases de données sont incapables de fournir des *ResultSet* avec de telles caractéristiques. En cas de doute, il peut-être très utile de manipuler les méta-données (Voir plus loin dans ce chap.) et notamment d'appeler les méthodes *supportsResultSetType* et *supportsResultSetConcurrency* de la classe *DatabaseMetaData*.

D.VI - Parcourir un *ResultSet*

1) Les méthodes

Il existe plusieurs façons de parcourir un jeu d'enregistrements. Voici un tableau listant les différentes méthodes permettant de naviguer dans un jeu d'enregistrements :

Method Summary	
boolean	**absolute**(int row) Déplace le curseur à la ligne passée en paramètre
void	**afterLast**() Déplace le curseur à la fin de ce ResultSet , juste après la dernière ligne.
void	**beforeFirst**() Déplace le curseur à la fin de ce c avant la première ligne
boolean	**first**() Déplace le curseur sur la première ligne
boolean	**isAfterLast**() Renvoie true si le curseur se situe après la dernière ligne.
boolean	**isBeforeFirst**() Renvoie true si le curseur se situe avant la première ligne.
boolean	**isFirst**() Renvoie true si le curseur se situe sur la première ligne
boolean	**isLast**() Renvoie true si le curseur se situe sur la dernière ligne
boolean	**last**() Déplace le curseur sur la dernière ligne
void	**moveToCurrentRow**() Déplace le curseur à la dernière position mémorisée du curseur. Il s'agit habituellement de la ligne courante
void	**moveToInsertRow**() Déplace le curseur à la position d'insertion d'un nouvel enregistrement
boolean	**next**() Déplace le curseur à la position suivante. Renvoie false si aucune ligne ne suit la ligne courante
boolean	**previous**() Déplace le curseur à la position précédente. Renvoie false si aucune ligne ne précède la ligne courante
boolean	**relative**(int rows) Déplace le curseur d'un nombre de lignes fourni en paramètre. Ce nombre peut-être positif ou négatif
void	**setFetchDirection**(int direction) Donne la direction dans laquelle les lignes doivent être parcourues

Liste Méthodes servant à parcourir un objet ResultSet

2) Exemple

Voici un exemple de programme capable de parcourir un objet *ResultSet* et d'afficher à l'écran le champ *nom* d'une table nommée *Clients* :

```
1  try {
2            Class.forName("com.mysql.jdbc.Driver");
3
4            } catch (ClassNotFoundException ex) {
5               System.out.println("Pilote JDBC non enregistré");
6            }
7
8            String url = "jdbc:mysql://bd.diderot.org:3306/diderot";
9
10 try {
11           Connection c;
12           c = DriverManager.getConnection(url,"username","password");
13
14           Statement st = c.createStatement();
15
16           String strSql = "SELECT nom FROM clients";
17           ResultSet rs = st.executeQuery(strSql);
18
19           while (rs.next())
20                System.out.println(rs.getString("nom"));
21
22    } catch (SQLException ex) {
23           System.out.println("Erreur SQL :"+ex.getMessage());
24           }
25
26
```

Appel à la méthode *next* comme condition de continuation de la boucle

Récupération et affichage du contenu du champ nom

D.VII - Insérer et modifier un ResultSet

Avant de modifier un *ResultSet*, il faut s'assurer que ce dernier est bien du type *CONCUR_UPDATABLE* (Voir *Caractéristiques d'un ResultSet*).

1) Insérer

Avant d'insérer une ligne, il faut faire appel à la méthode *moveToInsertRow()*. Cette opération revient à placer le curseur sur une ligne spéciale utilisée pour l'insertion uniquement.

```
1  rs.moveToInsertRow();
```

Ensuite, il faut mettre à jour les champs de la nouvelle ligne grâce aux méthodes *updateXXX* ou XXX représente le type champ à mettre à jour :

```
2  rs.updateString("nom","VALET"); // Met à jour le champ nom
3  rs.updateString(2,"Génaël"); // Met à jour le 2ème champ
```

A ce stade , la base de données n'a pas encore été modifiée. Elle le sera à l'appel de cette instruction :

```
4  rs.insertRow();
```

L'appel à la méthode *insertRow()* est équivalent à une requête SQL de type *INSERT*. Néanmoins, il est plus judicieux d'utiliser les méthodes de *ResultSet* plutôt que les requêtes SQL. Ceci est valable pour la version 2 de JDBC (Très utilisée aujourd'hui)

La dernière étape facultative consiste à repositionner le curseur à la position qu'il occupait avant l'appel à la méthode *moveToInsertRow()* :

```
5  rs.moveToCurrentRow();
```

2) Modifier

Pour modifier une ligne, il suffit de positionner le curseur sur cette ligne et de faire des appels successifs aux méthodes *updateXXX*.

Pour valider les changements dans la base de données, il convient d'utiliser la méthode *updateRow()*.

```
1  rs.updateString("nom","DUPONT");
2  rs.updateInt("age",35);
3  rs.updateRow();
```

D.VIII - Utilisation de requêtes pré-compilées

1) Introduction

Dans le sous-chapitre précédent, nous décrivions les objets *PreparedStatement* comme des zones de description de requêtes pré-compilées qui offrent de meilleures performances que les objets *Statement* traditionnels.

L'avantage de ces objets est qu'il peuvent être exécutés avec des paramètres différents à chaque exécution. Une requête pourra être executée plusieurs fois avec un seul objet *PreparedStatement*.

2) Comment créer un objet *PreparedStatement* ?

Ce type d'objet est toujours associé à un objet *Connection*. En utilisant un objet *Connection* déjà existant, nous écrirons le code suivant :

```
4  PreparedStatement updatePort = con.prepareStatement(
5                "UPDATE Clients SET port = ? WHERE ville LIKE ?");
```

3) Exécuter avec des paramètres

La base de données est alors prête à exécuter les requêtes consistant à modifier le montant des frais de port pour tous les clients habitant la ville de Paris :

```
6  updatePort.setDouble(1,6.0);     // Frais de port à 6 euros
7  updatePort.setString(2,"Paris"); // Pour les clients habitants Paris
8
9  updatePort.executeUpdate();  // Mise à jour des données
```

Pour changer les frais de port pour tous les clients habitant la ville de Lyon, il suffit de réutiliser la requête pre-compilée :

```
10 updatePort.setDouble(1,7.5);
11 updatePort.setString(2,"Lyon");
12
13 updatePort.executeUpdate();
```

4) Exécuter au sein d'une boucle

Imaginons que l'on souhaite modifier les frais de port pour un certain nombre de villes dont les noms sont stockés dans un tableau. Les frais de port correspondant à chaque ville seront stockées dans un autre tableau :

```
1  PreparedStatement updatePort = con.prepareStatement(
2                "UPDATE Clients SET port = ? WHERE ville LIKE ?");
3
4  String [] ville = {"Paris","Lyon","Bordeaux","Brest","Lille"};
5  double [] port = {6.0 , 7.5, 7.8, 8.0, 7.0 };
6
7  for (int i=0; i<ville.length ; i++) {
8           updatePort.setDouble(1,port[i]);
9           updatePort.setString(2,ville[i]);
10          updatePort.executeUpdate();
11          }
```

Préparation du tableau contenant les villes

Modification des paramètres de la requête

Exécution de la requête à chaque passage dans la boucle

D.IX - Les méta-données

1) Introduction

Dans les pages précédentes , nous sommes partis du constat que nous connaissions parfaitement les types et les noms des champs présents dans la base de données. Parfois, il peut-être utile d'avoir accès à certaines informations sur la structure et les types de données d'une base.

Les méta-données contiennent donc des informations sur la structure de la base de données (Les tables, les champs, …)

Toutes les bases de données n'autorisent pas l'utilisation des méta-données

2) DatabaseMetaData

Il s'agit d'une classe vous permettant d'obtenir un maximum d'informations sur la base de données elle-même. Ces informations explorent toute la complexité des bases de données et selon le type de base, tout ou partie de ces informations peuvent ne pas être disponibles.

Voici un programme qui extrait quelques informations à propos d'une base ACCESS en passant par le pont *JDBC/ODBC*. Le programme utilise la classe *Properties* , qu'il convient de connaître avant de se plonger dans le programme :

```
1  import java.sql.*;
2  import java.util.*;
3
4  public class MetaDonnees {
5           public static void main( String[] args) {
6                try {
7                     // Enregistrement du pilote
8                     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
9
10                    // Etablissement de la connexion
11                    Connection c;
12                    String url = "jdbc:odbc:cours3java";
13
14                    c=DriverManager.getConnection(url);
15
16                    // Obtention des meta-donnees
17                    DatabaseMetaData md = c.getMetaData();
18
19                    // Initialisation d'une liste de proprietes
20                    Properties prop = new Properties();
21
22                    // Acquiert les proprietes
23                    prop.setProperty("Nom du produit",
24                                     md.getDatabaseProductName());
25                    prop.setProperty("Version du produit",
26                                      md.getDatabaseProductVersion() );
27                    prop.setProperty("Nombre max de connexions",
28                                String.valueOf(md.getMaxConnections()) );
29                    prop.setProperty("Mots cles SQL",
30                                          md.getSQLKeywords() );
31                    prop.setProperty("Supporte les procedures stockees",
32                         String.valueOf(md.supportsStoredProcedures()));
33
34                    prop.setProperty("Supporte les procedures stockees",
35                         String.valueOf(md.supportsStoredProcedures()));
36
37                    // Envoie la liste a l'ecran
38                    prop.list(System.out);
39
```

Récupération d'un objet de type *DatabaseMetaData* nommé *md*

Appel à la méthode de *md* informant sur le nom du produit de base de données

Conversion de l'entier représentant le nombre max de connexions en une chaine (String)

Envoie la liste de propriétés directement à l'écran.

```
40
41                } catch ( ClassNotFoundException ex) {
42                    System.out.println("Erreur de config :"
43                                                    +ex.getMessage());
44                } catch ( SQLException ex) {
45                    System.out.println("Erreur SQL :" +ex.getMessage());
46                    }
47                }
48          }
```

Fichier MetaDonnees.java

3) ResultSetMetaData

Voyons maintenant comment obtenir des informations plus détaillées sur les objets *ResutSet* et les lignes et colonnes qu'il contient. En effet, il peut-être très intéressant de connaître la taille ou encore le type de données d'un champ avant d'y insérer des données.

Voici une liste des méthodes pouvant être utilisées pour obtenir des informations

Liste des méthodes de *ResultSetMetaData*	
String	**getCatalogName**(int column) Donne le nom du catalogue auquel appartient la colonne spécifiée
String	**getColumnClassName**(int column) Renvoie le nom qualifié de la classe Java dont les instances seront renvoyées lors de l'appel à la méthode *getObject(...)* de *ResultSet*. La méthode *getObject(..)* peut-être appelée pour obtenir la valeur d'un champ sous forme d'objet Java
int	**getColumnCount**() Renvoie le nombre de champs (colonnes) de cet objet
int	**getColumnDisplaySize**(int column) Renvoie la taille maximale du champ dont le numéro de colonne est donné en paramètres
String	**getColumnLabel**(int column) Renvoie le titre du champ désigné par le numéro de colonne, utilisé pour les impressions ou les affichages.
String	**getColumnName**(int column) Renvoie le nom du champ.
int	**getColumnType**(int column) Renvoie le type SQL du champ spécifié. Les types SQL peuvent être obtenu en utilisant la classe *java.sql.Types*
String	**getColumnTypeName**(int column) Renvoie le type SQL du champ spécifié sous forme de chaîne de caractères.
int	**getPrecision**(int column) Renvoie le nombre de digits décimaux du champ spécifié
int	**getScale**(int column) Renvoie le nombre de digits après la virgule.
String	**getSchemaName**(int column) Renvoie le nom du schéma de table pour le champ spécifié
String	**getTableName**(int column) Renvoie le nom de la table
boolean	**isAutoIncrement**(int column) Indique si le champ spécifié est de type *auto-increment* et donc en lecture seule.
boolean	**isCaseSensitive**(int column) Indique si le champ spécifié est sensible à la casse (majuscules/minuscules)

boolean	**isCurrency**(int column) Indique si la valeur du champ est type monétaire
boolean	**isDefinitelyWritable**(int column) Indique si une écriture sur la valeur de champ sera définitive
int	**isNullable**(int column) Indique la possibilité de trouver des valeurs nulles pour le champ spécifié
boolean	**isReadOnly**(int column) Indique si la valeur du champ est en lecture seule
boolean	**isSearchable**(int column) Indique si le champ spécifié peut faire partie d'une clause SQL *WHERE*.
boolean	**isSigned**(int column) Indique si le champ spécifié peut contenir un nombre signé
boolean	**isWritable**(int column) Indique si la valeur du champ spécifié peut-être écrite

Methodes de la classe ResultSetMetaData

Voici un extrait de programme permettant de vérifier la taille maximale d'une chaîne dans un champ et de tronquer la chaîne à insérer en cas de dépassement :

```
1  ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM TABLE2");
2
3  ResultSetMetaData rsmd = rs.getMetaData();
4
5  int tailleMax = rsmd.getColumnDisplaySize();
6
7  String maChaine = "Chaine à inserer dans la BD";
8
9  if (tailleMax<maChaine.size())
10         maChaine = machaine.substring(1,tailleMax);
11
12 rs.updateString("champ",maChaine);
13 rs.updateRow();
```

Récupération de la taille maximum

Tronquage de la chaîne si dépassement

Mise à jour du champ

Mise à jour de la base

D.X - Transactions avec JDBC

1) Introduction

Lorsque l'on modifie les données d'une base, il est vital de maintenir une certaine cohérence dans les données. Imaginons une base de données contenant toutes les informations concernant les vols d'une compagnie aérienne. Pour supprimer un passager d'un vol en particulier, il faut supprimer la ligne le concernant grâce à une requête de type *DELETE*. Il faut également mettre à jour le nombre de sièges libres sur ce vol. Nous avons là deux opérations à mener. Que se passera-t-il si la première opération qui consiste à supprimer le passager se déroule bien mais que la mise à jour du nombre de sièges libres ne se fait pas ? Il y aura des données corrompues dans la base.

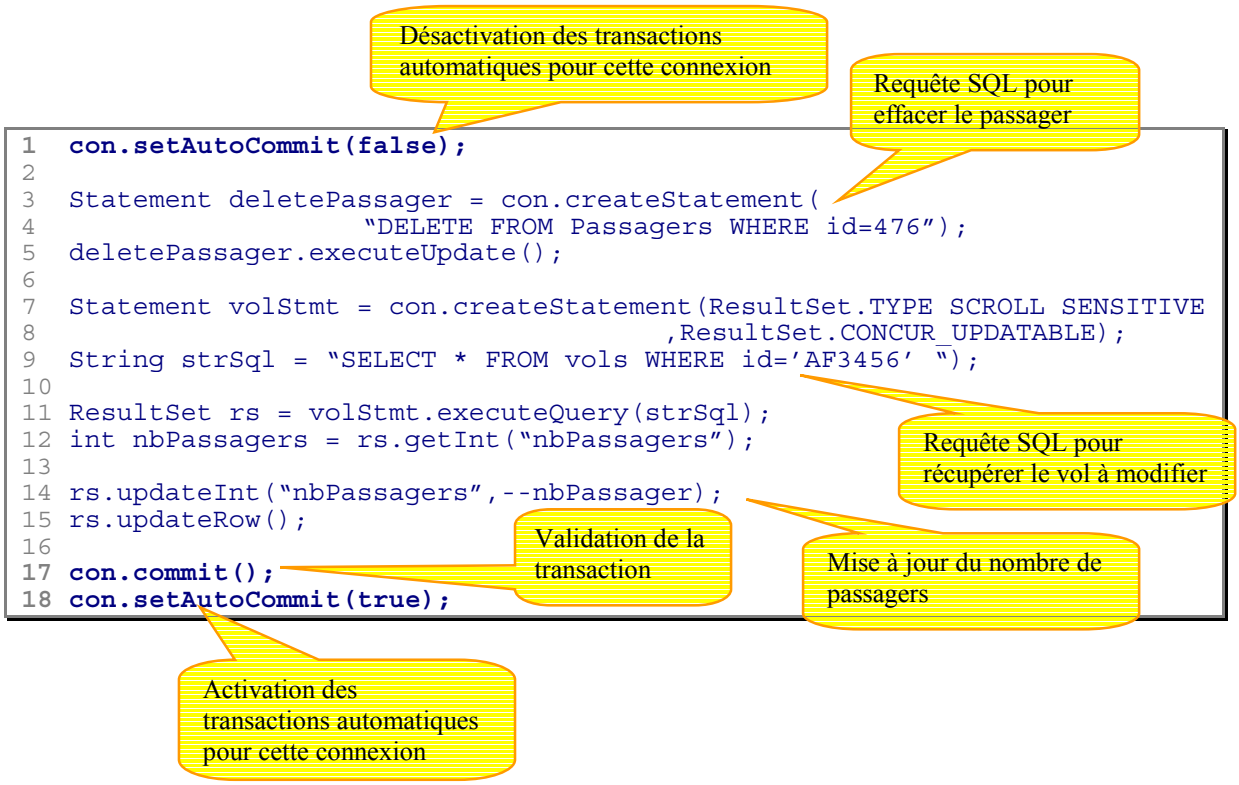
Les transactions peuvent résoudre ce problème. Elles consistent à englober plusieurs opérations dans un même sous-ensemble et de considérer que la transaction a abouti lorsque toutes les opérations ont réussi. Si une des opérations échoue, alors, toutes les autres sont également annulées.

2) Le mode « *auto-commit* »

Par défaut, toute connexion à une base de données est en mode « *auto-commit* ». Chaque commande SQL (SQL Statement), est traité comme une transaction qui sera automatiquement validée juste après avoir été executée.

En désactivant le mode « *auto-commit* » , vous considérez que tous les modifications ou ajouts effectués par objets *Statement* crées à partir d'un objet *Connection* ne seront validés que lors de l'appel à la méthode *commit* de *Connection*.

Voici un exemple illustrant ce qui vient d'être dit. Dans ce programme, il s'agit de supprimer un passager d'un vol et d'incrémenter le nombre de places libres :




```
1  con.setAutoCommit(false);
2
3  Statement deletePassager = con.createStatement(
4                     "DELETE FROM Passagers WHERE id=476");
5  deletePassager.executeUpdate();
6
7  Statement volStmt = con.createStatement(ResultSet.TYPE SCROLL SENSITIVE
8                                          ,ResultSet.CONCUR_UPDATABLE);
9  String strSql = "SELECT * FROM vols WHERE id='AF3456' ");
10
11 ResultSet rs = volStmt.executeQuery(strSql);
12 int nbPassagers = rs.getInt("nbPassagers");
13
14 rs.updateInt("nbPassagers",--nbPassager);
15 rs.updateRow();
16
17 con.commit();
18 con.setAutoCommit(true);
```

3) Annuler une transaction

Afin d'être absolument sûr qu'aucune modifications de données n'a pu être validée, il est préférable d'annuler une transaction lorsqu'une exception de type *SQLException* est générée.

Pour annuler une transaction , il suffit d'appeler la méthode *roollback()* de *Connection*. Il est préférable d'appeler la méthode *rollback()* dans le cas où une des opérations vers la base de données a provoqué une exception de type *SQLException* :

```
1  Connection conn = null;
2  try {
3      conn = DriverManager.getConnection(strUrl);
4      conn.setAutoCommit(false);
5      CallableStatement = /* . . . */
6
7      /* . . . */
8
9      conn.commit();
10 } catch(SQLException e) {
11     conn.rollback();
12 } finally {
13     conn.close();    /* Fermeture de la connexion */
14 }
```

Création de la connexion, du Statement et opérations sous le regîme de la transaction

Validation des opérations précédentes

Annulation de la transaction en cas d'erreur

Notez que les méthodes *rollback()* et *commit()* sont également susceptibles de provoquer une exception. Pour une fiabilité maximum, certains développeurs insèrent l'appel à *rollback()* à l'intérieur d'un bloc *try…catch* . L'imbrication de bloc *try…catch* étant accepté en Java, il devient alors possible d'empêcher une perte des données au cas où la méthode *rollback()* échouerait.

4) Niveaux de transactions

Par défaut, la base de données fixe le niveau de transaction. Avec JDBC, il existe plusieurs niveaux de transactions mais tous ne sont pas forcément pris en charge. Pour savoir quel est le niveau par défaut de votre SGBD, il suffit d'appeler la méthode *getTransactionIsolation()* qui renvoie un entier dont la signification est décrite plus bas.

A l'inverse, la méthode *setTransactionIsolation(int level)* de la classe *Connection* permet de modifier le niveau de transaction. Elle prend comme arguments les valeurs suivantes, champ statique de cette même classe :

- **TRANSACTION_READ_UNCOMMITED** : une opération t1 modifie des données, alors l'opération t2 qui suit pourra lire ces données modifiées.
- **TRANSACTION_READ_COMMITTED** : une opération t1 modifie des données , alors l'opération t2 ne pourra pas lire les données modifiées.
- **TRANSACTION_REPEATABLE_READ** : une opération t1 lit des données, alors que l'opération t2 modifie ces données. Si t1 lit à nouveau ces données, il lit la même valeur que précedemment.
- **TRANSACTION_SERIALIZABLE** : t1 lit un ensemble de données, t2 ajoute des données à cet ensemble, si t1 lit à nouveau l'ensemble, il ne verra pas les données modifiées. Les opérations apparaissent comme si elles avaient été exécutées en séquence.