Introduction au langage JAVA

Tome 2 : cours et exercices

Université de Paris 13

Cours Préparé par A. OSMANI

Sommaire

11 LES SWING 4

11.1 AWT (ABSTRACT WINDOW TOOLKIT) 5
11.2 GRAPHICS2D 7
11.3 ACCESSIBILITE 8
11.4 GLISSER ET DEPLACER 8
11.5 LES SWING 8
11.5.1 Composants 8
11.5.2 Conteneur 9
11.5.3 Gestionnaire de placement 9
11.6 GESTION D'EVENEMENTS 10
11.6.1 Récepteurs d'événements 10
11.6.2 Sources d'événements 11
11.6.3 Les Adapteurs 11

12 LES FLUX ET LES FICHIERS 13

12.1 INTRODUCTION 13
12.2 LA CLASSE INPUTSTREAM 15
12.3 LA CLASSE OUTPUTSTREAM 16
12.4 LES FLUX DE CARACTERES(CLASSES READER ET WRITER) 17
12.4.1 Classe Reader 17
12.4.2 Classe Writer 17
12.5 LES CLASSES DATAINPUTSTREAM ET DATAOUTPUTSTREAM 19
12.6 LES CLASSES FILEINPUTSTREAM ET FILEOUTPUTSTREAM 20
12.7 LA CLASSE JAVA.IO.FILE 22
12.8 ACCES DIRECT AUX FICHIERS 23
12.9 L'INTERFACE SERIALIZABLE 24
12.10 LES FLUX DE FICHIERS ZIP 24
12.11 COMPLEMENT FLUX D'ENTREES/SORTIE 25
12.11.1 Saisir des données envoyées par le clavier 25
12.11.2 Lire ou écrire des caractères dans un fichier 26
12.11.3 Écrire dans un fichier texte 27
12.11.4 Lire dans un fichier texte 27
12.11.5 Écrire dans un fichier binaire 28
12.11.6 Lire dans un fichier binaire 29

TD-TP N° : LES FLUX D'ENTREES ET DE SORTIES 30

13 LES JDBC JAVA DATABASE CONNECTIVITY 33

13.1 INTRODUCTION 33
13.2 JDBC ET LES ARCHITECTURES CLIENT-SERVEUR 34
13.3 TYPOLOGIE DE PILOTES (DRIVERS) JDBC 35
13.4 APPLICATIONS TYPIQUE 36
13.5 STRUCTURE D'UNE APPLICATION JDBC 36
13.6 API JDBC 36
13.6.1 Charger un pilote en mémoire 36
13.6.2 Etablir une connexion 37
13.6.3 Traitement des requêtes SQL simples 38
13.6.4 Récupération des résultats 40
13.6.5 Fermeture d'une connexion 41
13.6.6 Appel à des procédures précompilées 42
13.7 ACCES AUX META-DONNEES 43
13.8 PRISE EN CHARGE DES TRANSACTIONS 43
13.9 CREATION D'UNE SOURCE DE DONNEES DANS ODBC 44
13.10 CREATION D'UNE SOURCE DE DONNEES DANS ODBC 44
13.11 ANNEXES 45

14 PROGRAMMATION RESEAU : SOCKETS ET RMI 47

14.1 LE PACKAGE JAVA.NET 47
14.1.1 La classe java.net.Socket 48
14.1.2 48
14.2 LA CLASSE JAVA.NET.SERVERSOCKET 48
14.3 LA CLASSE JAVA.NET.DATAGRAMSOCKET 48
14.4 JAVA.NET.MULTICASTSOCKET 49
14.5 JAVA.NET.URL 49

11 Les Swing

11 LES SWING 4

11.1 AWT (ABSTRACT WINDOW TOOLKIT) 5
11.2 GRAPHICS2D 7
11.3 ACCESSIBILITE 8
11.4 GLISSER ET DEPLACER 8
11.5 LES SWING 8
11.5.1 Composants 8
11.5.2 Conteneur 9
11.5.3 Gestionnaire de placement 9
11.6 GESTION D'EVENEMENTS 10
11.6.1 Récepteurs d'événements 10
11.6.2 Sources d'événements 11
11.6.3 Les Adapteurs 11

Le JFC (Java Foundation Classes)[1] peut être découpé en cinq groupes : AWT, Swing, API d'accessibilité (Accessibility), dessin 2D et les glisser-déplacer (Drag and Drop). Java 2D (java.awt.Graphics2D) fait partie intégrante de AWT, c'est une extension de la classe java.awt.Graphics.

L'intérêt des Swing et des AWT est de permettre la création d'interfaces utilisateurs. Les éléments graphiques des AWT étaient lourds et limités. Ceci a conduit certains éditeurs à développer leurs propres outils comme les Internet Foundation Classes (IFC) proposées par Netscape plus rapides et moins sensibles aux erreurs que les AWT.
Sun s'est associée avec Netscape pour proposer les Swing en combinant et en améliorant les IFC et les AWT.

Les swing ne remplace pas complément les AWT, elle fournit tout simplement des composant plus performant. La gestion des événements reste la même que celle de java 1.1.
La section suivante donne quelques exemples du package AWT. Les composants AWT sont toujours disponibles dans java 2, mais je vous recommande d'utiliser les swing pour la réalisation de vos interfaces.
Parmi les raisons qui pourrons vous aider à opter pour les Swing :

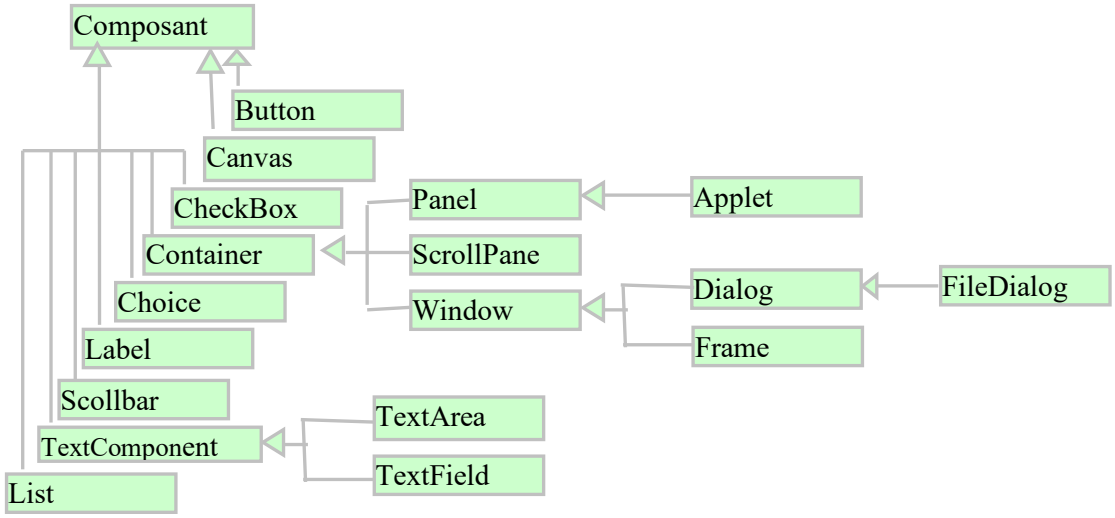
- L'ensemble des composants swing sont plus étendu et plus pratique. La plupart des classes Swing sont dérivées des classes AWT. La classe javax.swing.Jframe, par exemple, dérive de la classe java.awt.Frame. Parmi les classes ajoutées, on peut citer : les structures arborescentes et les registres ;
- Il est possible d'avoir différents types d'affichage des composants swing. Les trois types de « look » standards sont : metal, windows et motif (ces interfaces peuvent être modifiées en exécution) ;

[1] Les JFC sont une collection de classes qui offre des fonctions étendues aux programmeurs.

- Les composants Swing sont moins dépendant des plate-formes d'exécution. D'ou la réduction des erreurs inhérents aux plates-forme.

11.1 AWT (Abstract Window Toolkit)

La plupart des composants de AWT dérivent de la classe java.awt.Component comme le montre la figure suivante :



Il existe au moins de façons différentes de créer des interfaces graphiques : en utilisant les applets (nécessite un navigateur) ou bien en créant des applications autonomes. Dans ce cours nous allons considérer le deuxième cas. Les applications sont essentiellement utilisées en dehors du cadre internet. Elle peuvent tourner sur n'importe quel système.

La classe java.awt.Frame réalise un affichage facile à l'écran. De ce fait, la plupart des applications héritent de cette classe. L'exemple suivant montre comment créer une fenêtre, définir sa taille puis l'afficher à l'écran.

```
import java.awt.*;
class MaFenetre extends Frame
{  MaFenetre(String a) {    super (a);    }
   public static void main (String a[])
    {     Frame fenetre = new MaFenetre("Ma première fenêtre");
          fenetre.setSize(100,200);
          fenetre.setVisible(true);
     }
}
```

Si vous exécutez ce programme vous allez vous rendre compte que c'est difficile d'arrêter le programme. Vous pouvez cacher la fenêtre en cliquant sur l'icône de gauche mais vous ne la fermer pas.

Il existe un moyen de détruire la fenêtre en utilisant la méthode setDefaultCloseOperation. Celle-ci permet de ne rien faire, de cacher le cadre ou de le détruire. Évidemment, ceci ne suffit pas à arrêter le programme.

Le moyen le plus efficace de fermer correctement le programme est d'utiliser l'interface WindowListner. Cette interface contient sept méthodes (voir javadoc), pour l'utiliser il faut implémenter toutes ses méthodes alors que nous n'avons besoin que d'une seule méthode : public void windowClosing(WindowEvent e). La classe WindowAdapter du package AWT donne une implémentation vide de toutes ces méthodes, pour notre cas, il convient de surcharger la méthode qui nous intéresse.
Ainsi, le programme précédant permettant la création d'une fenêtre pourra être arrêté en le complément comme suit :

```
import java.awt.*;
class MaFenetre extends Frame
{  MaFenetre(String a)
   {   setTitle(a);
       addWindowListener( new WindowAdapter()
                                  { public void windowClosing(WindowEvent e){  System.exit(0);
} }    );
    }
  public static void main (String a[])
  {  Frame fenetre = new MaFenetre("Ma première fenêtre");
     fenetre.setSize(100,200);
     fenetre.setVisible(true);
   }
}
```

Quelques exemples d'utilisation des composants awt :
Exemple : boite de sélection de fichiers

```
import java.awt.*;
import java.io.*;
class Editeur extends Frame
{ void loadFile()
  {FileDialog fd = new FileDialog( this,"Ouvrir un fichier",FileDialog.LOAD );
   fd.show() ;
   String fichier = fd.getDirectory() + fd.getFile() ; // repertoire + nom
   if( fd.getFile() == null ) return ;   // si le nom est vide

   try{  FileInputStream fis = new FileInputStream(fichier) ;
         byte [] donnees = new byte[ fis.available() ] ; // tableau à la taille du fichier
         fis.read( donnees ) ; // lit le fichier
         fis.close() ;    // ...
       } catch( IOException e ) {/* …*/       }
  }
  public static void main( String [] argv )
  {   Editeur e= new Editeur(); e.loadFile() ; e.setVisible(true);
  }
}
```

Exemple : création d'un bouton

```
import java.awt.*;
import java.awt.event.*;
class TestBouton extends Frame
{   public TestBouton()
    {  setTitle("exemple de bouton");     setSize(300,200);
       addWindowListener ( new WindowAdapter()
       {  public void windowClosing(WindowEvent e){System.exit(0);}
}          );
```

Exemple : les cases à cocher

```
import java.awt.* ;
public class CasesACocher extends Applet
{   public void init ()
    {  setLayout (new FlowLayout());
       CheckBox c = new CheckBox ("case1");
        add(c);
        add(new CheckBox("case2");
```

Exemple : la liste déroulant

```
import java.awt.* ;
import java.applet.* ;
public class ListeDeroulante extends Applet
{  public void init()
   {  setLayout (new FlowLayout()) ;
      Choice c = new Choice();
      c.addItem("item1");
      c.addItem("item2");
      c.addItem("item3");
      add(c);
      c.select(1);
   }
}
```

Les actions de l'utilisateur sur les composants de l'AWT sont gérés par des événements qui peut être un clic sur la souris, un appui sur une touche, etc.

11.2 Graphics2D

La classe Graphics2D est une extension de la classe java.awt.Graphics. Elle réalise, donc toutes les méthodes de la classe Graphics, parmi lesquelles :
Ainsi qu'un ensemble de fonctions complémentaires :

- Motifs linéaires ;

- Transformation d'objets ;
- Remplissage de motifs et de couleurs ;
- Zone d'affichage variable
- Mélange de couleurs lorsque plusieurs objets se coupent.

11.3 Accessibilité

Ces API permettent d'intégrer des périphériques d'entrées et sorties spéciaux. Le package correspondant est le package : javax.accessibility

11.4 Glisser et déplacer

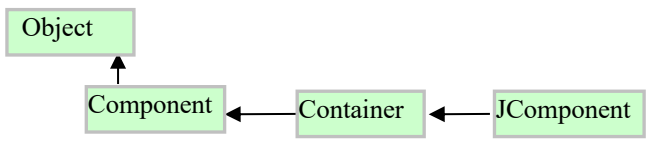
Les fonctionnalités Drag and Drop permettent de transférer des données entre plusieurs applications ou à l'intérieur d 'une même application. Le package java.awt.dnd assure ces possibilités.

11.5 Les swing

La procédure de construction d'une interface en utilisant les Swing est la même que celle utilisant les AWT. Elle consiste à créer des cadre, des composants dans le cadre, de faire une mise en page des composants et de réaliser les méthodes pour la gestion des réponses aux actions des utilisateurs.
La maîtrise de ce package ne peut se faire sans une utilisation effective des classes de ce package dans une application réelle. Cette section présente quelques éléments de base des swing.

11.5.1 Composants

Un *composant* est l'entité fondamentale d'une interface utilisateur. Tout ce que l'on voit sur l'écran d'une application java est un composant. Ceci comprend des fénêtres, des boutons, des cases à cocher, etc. Pour être utilisé, un composant doit généralement être placé dans un *conteneur.*
JComponent est la racine de l'arborescence de composants Swing. Elle descend de la classe *Container* du paquetage AWT. .
Jcomponent héritant de Container, possède à la fois des capacités d'un composant et d'un container.



Les fonctionnalités d'un objet JComponent peuvent être divisées en deux catégories : apparence (taille, emplacement, etc.) et comportement (réaction de l'objet par rapport aux événements contrôlés par l'utilisateur).
Parmi les méthodes de Jcomponent, on peut citer :

- Container getParent() : renvoie le conteneur hébergeant ce composant ;
- String getName(), void setName(String nom) : obtient ou affecte le nom String à ce composant.
- Void setVisible (boolean v) : rend le composant visible ou invisible dans son conteneur.
- Color getForeground(), void setForeground(Color a), Color getBackground()
- Dimension getSize(), void setSize(int largeur, int longueur)
- Cursor GetCursor(), void SetCursor(Cursor a) : obtient ou définit le type de pointeur de la souris au dessus de ce composant

Les composants sont presque les mêmes que ceux des AWT.

- JButton
- JLabel
- JTextArea
- JTextField
- JCheckBox
- JScrollBar
- JList
- JPasswordField

11.5.2 Conteneur

Un conteneur sont des objets dans lesquels sont incorporés des composant. Etant donnée que la classe JComponent hérite de la classe Container, des objets JComponent peuvent donc être des containers.
Exemples de conteneurs :

- JFrame , Jpanel : ils représentent un cadre spécifique à l'environnement ;
- JSrollPane. Il s'agit d'une barre de défilement. Vous pouvez, par exemple, ajouter un JscrollPane puis un Jbouton à un Jframe ;
- JDialog. C'est une boite de dialogue qui apparaît au milieu de l'écran ;

Pour être traités (affichage et positionnement), les composants doivent être intégrés à un conteneur.

- La méthode add(), de la classe Container, permet d'ajouter un composant conteneur
- La méthode remove( )permet d'enlever un composant d'un conteneur.

11.5.3 Gestionnaire de placement

Un gestionnaire de placement est un objet qui contrôle le placement et la taille des composants situés à l'intérieur de la zone d'affichage d'un conteneur.
Java fournit de nombreux gestionnaires de placement et il est tout à fait possible de créer ses propres gestionnaires de placement par la méthode setLayout().
Exemples :

- FlowLayout (Gestionnaire de placement, par défaut, d'un JPanel) : place les objets suivant l'alignement spécifié par les constantes LEFT, CENTER ou RIGHT.
- BorderLayout (Gestionnaire de placement, par défaut, d'un JFrame) : contrairement à FlowLayout qui contrôle complétement la position de chaque composant, BorderLayout place les objets à des emplacements désignés de la fenêtre : au centre, au nord, au sud, à l'est ou à l'ouest. L'instruction, ci-dessus, permet de placer un composant en haut d'un containeur :

  myContainer.add(myComponent, BorderLayout.NORTH);

11.6 Gestion d'événements

Les gestionnaires d'événements permettent d'intercepter les actions des utilisateurs et d'assigner au programme un comportement adapté en réponse. Il existe de nombreuses manières de gérer les événements. Ce chapitre propose une manière classique, c'est celle qui est utilisée par défaut par Jbuilder. Elle se met en œuvre en trois temps : création d'un écouteur d'actions qui attend et capte les actions de l'utilisateur, d'une méthode qui met en œuvre la réponse adaptée, da la classe qui définie cette méthode.
Les objets de Swing communiquent en envoyant des événements. Les événements sont envoyés par un objet source à un ou plusieurs objets listeners. Un listener implémente les méthodes qu'il faut pour gérer les événements qu'il reçoit. Il s'enregiste ensuite auprès d'une source de ce type de message. Parfois, il est nécessaire de définir un adapter entre la source de l'événement et le listener, mais l'inscription d'un listener auprès d'une source se fait toujours avant l'envoi des événements.

Les objets événements héritent de la classe java.util.EventObject. Ils contiennent les informations sur ce qui s'est produit dans la source. La classe EventObject sert à identifier les objet événements, sa seule information est une référence à la source des événements.

Un ActionEvent correspond à une action déterminante décidée sur le composant par l'utilisateur. Il transporte le nom de l'action à effectuer. Par exemple, les MouseEvent sont générés lorsque la souris se déplace dans la zone du composant. Ces événements véhiculent les informations concernant la souris : les coordonnées, les boutons appuis, etc.
Les ActionsEvent fonctionne à un niveau plus élevé que les MousseEvent : il indique qu'un composant a terminé sa tâche.

11.6.1 Récepteurs d'événements

Un événement est transmis sous forme d'argument de la méthode de gestion d'événements de l'objet récepteur. Par exemple, les ActionEvent sont toujours transmis à la méthode actionPerformed du récepteur. public void actionPerformed(ActionEvent e)
A chaque type d'événements correspond une interface listener qui indique comment il sera traité.

Les objets qui recoivent des ActionEvent implémentent l'interface ActionListener qui contient la méthode actionPerformed.

```
public interface ActionListener extends java.util.EventListener
 { public void actionPerformed ( ActionEvent e) ;
  }
```

EventListener est une interface vide. C'est l'interface de base de toutes les interfaces listener

11.6.2 Sources d'événements

Pour recevoir des événements, les listener s'enregistrent auprès d'une source d'événements en appelant une méthode de la source en lui passant sa propre référence.
Exemple : la classe Jbutton est une source d'événements. Un objet de la classe MaClasse va s'inscrire comme suit :

```
class MaClasse implements ActionListener
{  void abonnement (JButton b)
   { b.addActionListener(this) ; // L'objet qui execute cette méthode va recevoir  les
ActionEvent
                                 // en provenance du bouton b. il envoie sa propre référence
    }
   public void actionPerformed (ActionEvent e)   {      }
}
```

Pour gérer les listeners, une source d'ActionEvent, comme JButton de l'exemple précédent, implémente toujours deux méthodes

```
public addActionListener (ActionListener e)  {  }
```

et

```
public void removeActionListener(ActionListener e){  }
```

dans le cas des awt ou des swing les événements sont associés à une unique source. Il peut être distribué à un nombre quelconque de Listeners. Lorsqu'un événement est déclenché, il est distribué individuellement à chaque listener, l'ordre ainsi que l'unicité de la distribution ne sot pas garantis.

11.6.3 Les Adapteurs

En général, il n'est pas souhaitable et parfois impossible que les composants de votre application reçoivent directement des événements. Car l'implémentation d'interface listeners vous oblige à ajouter le traitement des événements. Une solution consiste à placer une classe adapteur entre la source et le récepteur des événements.

Un objet adapteur fait correspondre un événement entrant en une méthode de sortie

Exemple :

```
class Adapteur implements ActionListener
{  MaClasse a ;
   Adapteur (MaClasse x)   {    a = x ;     }
   public void actionPerformed (ActionEvent e)  {   a.laméthode();    }
    public static void main (String [] a)
    {   MaClasse classe = new MaClasse();
            //…
        Jbutton leBouton = …
          // établissement de la liaison
        leBouton.AddActionListener(new Adapteur(classe)) ;
        …
    }
}
```

Exemple d'application :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class DinnerMenu extends JFrame {
  public DinnerMenu( ) {
    super("DinnerMenu v1.0");        setSize(200, 200);         setLocation(200, 200);
    // crée le menu Ustensiles
    JMenu utensils = new JMenu("Ustensiles");       utensils.setMnemonic(KeyEvent.VK_U);
    utensils.add(new JMenuItem("Fourchette"));      utensils.add(new JMenuItem("Couteau"));
    utensils.add(new JMenuItem("Cuiller"));         JMenu hybrid = new JMenu("Hybride");
    hybrid.add(new JMenuItem("Cuichette"));         hybrid.add(new JMenuItem("Cuiteau"));
    hybrid.add(new JMenuItem("Couchette"));         utensils.add(hybrid);
    utensils.addSeparator( );
    // configuration de l'élément Quitter
    JMenuItem quitItem = new JMenuItem("Quitter");       quitItem.setMnemonic(KeyEvent.VK_Q);
    quitItem.setAccelerator( KeyStroke.getKeyStroke(KeyEvent.VK_Q, Event.CTRL_MASK));
    quitItem.addActionListener(new ActionListener( ) {
      public void actionPerformed(ActionEvent e) { System.exit(0); }  });
    utensils.add(quitItem);
    // crée le menu Épices
    JMenu spices = new JMenu("Épices");      spices.setMnemonic(KeyEvent.VK_P);
    spices.add(new JCheckBoxMenuItem("Thym"));     spices.add(new JCheckBoxMenuItem("Romarin"));
    spices.add(new JCheckBoxMenuItem("Origan", true));
    spices.add(new JCheckBoxMenuItem("Fenouil"));
    // crée le menu Fromages
    JMenu cheese = new JMenu("Fromages");        cheese.setMnemonic(KeyEvent.VK_F);
    ButtonGroup group = new ButtonGroup( );      JRadioButtonMenuItem rbmi;
    rbmi = new JRadioButtonMenuItem("Normal", true);    group.add(rbmi);
    cheese.add(rbmi);                            rbmi = new JRadioButtonMenuItem("Extra");
    group.add(rbmi);                             cheese.add(rbmi);
    rbmi = new JRadioButtonMenuItem("Bleu");     group.add(rbmi);
    cheese.add(rbmi);
    // crée une barre de menu et l'utilise dans ce JFrame
    JMenuBar menuBar = new JMenuBar( );      menuBar.add(utensils);
    menuBar.add(spices);               menuBar.add(cheese);
    setJMenuBar(menuBar);
  }
  public static void main(String[] args) {
    JFrame f = new DinnerMenu( );
    f.addWindowListener(new WindowAdapter( ) {
      public void windowClosing(WindowEvent we) { System.exit(0); }
    });
    f.setVisible(true);
  }
}
```

12 Les flux et les fichiers

12 LES FLUX ET LES FICHIERS 13

12.1 INTRODUCTION 13
12.2 LA CLASSE INPUTSTREAM 15
12.3 LA CLASSE OUTPUTSTREAM 16
12.4 LES FLUX DE CARACTERES(CLASSES READER ET WRITER) 17
12.4.1 Classe Reader 17
12.4.2 Classe Writer 17
12.5 LES CLASSES DATAINPUTSTREAM ET DATAOUTPUTSTREAM 19
12.6 LES CLASSES FILEINPUTSTREAM ET FILEOUTPUTSTREAM 20
12.7 LA CLASSE JAVA.IO.FILE 22
12.8 ACCES DIRECT AUX FICHIERS 23
12.9 L'INTERFACE SERIALIZABLE 24
12.10 LES FLUX DE FICHIERS ZIP 24
12.11 COMPLEMENT FLUX D'ENTREES/SORTIE 25
12.11.1 Saisir des données envoyées par le clavier 25
12.11.2 Lire ou écrire des caractères dans un fichier 26
12.11.3 Écrire dans un fichier texte 27
12.11.4 Lire dans un fichier texte 27
12.11.5 Écrire dans un fichier binaire 28
12.11.6 Lire dans un fichier binaire 29

TD-TP N° : LES FLUX D'ENTREES ET DE SORTIES 30

12.1 Introduction

Le développement d'applications nécessite une utilisation abondante des entrées et sorties. Ce cours expliquera comment prendre une information de n'importe quelle source et comment émettre des informations vers toute sortie acceptant une suite d'octets.

Les sources et les destinations des données sont souvent les fichiers. Mais celles-ci peuvent être également des connexions sur un réseau ou des accès à des bases de données.

En définitive, le stockage de toute donnée se fait par une suite d'octets. En java, l'objet qui permet de lire une suite d'octet se nomme flux d'entrée (implémenté par la classe abstraite InputStream) et celui vers lequel on peut écrire se nomme flux de sortie (implémenté par la classe abstraite OutputStream). Tous les autres flux d'octets sont fondés sur ces deux classes. Elles représentent l'interface de plus bas niveau de tous les streams d'octets.

Comme ces classes sont abstraites, java implémente des classes dérivées pour lire et écrire dans les fichiers et communiquer via le réseau.

La classe InputStream dispose d'une seule méthode abstraite, il s'agit de la méthode *public abstract int read() throws IOException.* Cette méthode lit un seul octet et renvoie cet octet ou bien –1 si la fin de la source de donnée est atteinte.

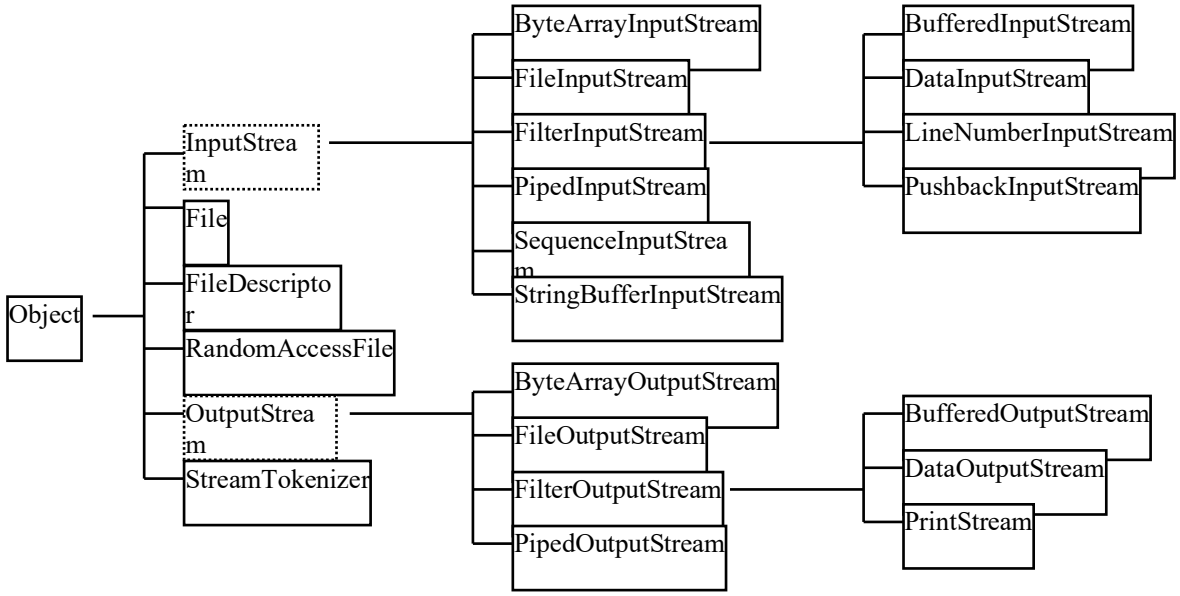
Exemples : la méthode read() est redéfinie dans :

1. l'objet prédéfini System.in (in est un attribut static de la classe java.lang.System) de la sous-classe de InputStream. Elle permet de saisir l'information à partir du clavier.
2. la classe FileInputStream permet de lire un seul octet dans un fichier.

De la même façon, la classe OutputStream définit une seule méthode abstraite : *public abstract void write(int b) throws IOException.* Elle permet d'écrire un octet vers une destination.

Les caractères en java sont écrit dans le format unicode (2 octets). De ce fait les flux d'entrées et sorties conviennent mal au traitement de l'information. Une hiérarchie de classes qui héritent des classes Reader et Writer qui permettent de lire et écrire des caractères unicodes est introduite.

Plus généralement, le langage java possède environ 75 types de flux alors qu'un langage comme le C ne possède qu'un seul : File*.
La figure suivante donne un sous-ensemble des classes permettant la gestion des flux d'entrées et de sorties.



Les 75 classes du package java.io se répartissent en quatre catégories :

- les opérations de niveau octet. Les ancêtres de cette catégorie sont les classes InputStream et OutputStream
- les opérations au niveau caractère. Les classes Reader et Writer sont les classes de base de cette catégorie
- les exceptions spécifiques aux entrées et sorties
- Divers.

Une autre classification permet de donner les catégories suivantes :

- Les flux (streams) d'entrées/sorties ;
- Les flux de caractères et les flux de données binaires ;
- Les flux de communication et de traitement des données ;

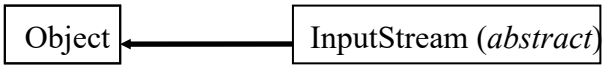
- Les flux d'accès séquentiel et les flux d'accès direct aux fichiers ;
- Les flux avec ou sans les tampons de données.

Pour effectuer une entrée ou une sortie en java , il est toujours nécessaire de passer par les étapes suivantes :

- Ouverture du flux de communication ;
- Ecriture ou lecture des données ;
- Fermeture du flux de communication.

Nous allons étudier quelques classes permettant le traitement des flux.

12.2 La classe InputStream



Toutes les méthodes de classe lèvent les exceptions IOException.
Les méthodes read se bloquent tant que l'entrée n'est pas disponible.

Les méthodes de la classe InputStream sont :

- public int available() throws IOException. Elle renvoie le nombre d'octets actuellement disponible pour la lecture.

```
Exemple : InputStream x = ... ;
            ...
          int compteur =x.available() ;
```

- public void close() throws IOException. Elle ferme le flux d'entrée

```
Exemple : InputStream x = ... ;
          ...
          int entree =x.read() ;
          ...
            x.close() ;
```

- public synchronized void mark (int TailleTamponPourLaLectureAnticipée). Elle marque la position en cours et fixe la taille du tampon pour la lecture anticipée.

```
Exemple : InputStream x = ... ;
          if (x.markSupported())
          {  x.mark (4);
             int tab=new tab[4] ;
              for(int i=0 ;i<tab.length ;i++) tab[i]=x.read() ;
             x.reset() ;
             if (tab[0] =="blabla") System.out.print("type 1") ;
             ...
             else System.out.println("Inconnu") ;
          }
```

- public boolean markSupported(). Elle vérifie si mark() et reset() fonctionnent pour une lecture anticipée.
- read(). Elle lit des octets des flux d'entrée.
  - public abstract int read() throws IOException : lecture d'un seul octet ;

- public int read(byte buffer[]) throws IOException : lecture de buffer.length maximum;
- public int read (byte buffer[], int débutTamponDeLecture, int TailleMaxiALire) throws IOException : lecture de TailleMaxiALire octets maximum vers un tableau commençant à partir de débutTamponDeLecture

- public synchronized void reset() throws IOException. Elle replace la position de lecture sur l'emplacement marqué, si la limite de lecture n'a pas été dépassée.
- public long skip (long NombreOctetsDontDoitAvancerPositionCourante). Elle déplace la position de lecture vers l'avant.

Exemple : InputStream x = ... ;
x.skip(10) ;
int x1 = x.read() ;

Nous avons l'habitude d'utiliser la classe System. Nous avons utilisé, essentiellement, la méthode print() associée à l'objet System.out. Il est également possible d'utiliser la méthode read() associée à l'objet System.in pour la lecture d'un octet. Ceci se fait de la façon suivante :

```
try
{  byte b =(byte)System.in.read();
   int nombreOctetsDisponibles = System.in.available() ;
                    // available permet de vérifier le nombre d'octets disponibles
                    //ensuite un tableau permettant de contenir tous les octets est crée
   if (nombreOctetsDisponibles >0)
   {  byte[] entrée = new byte[nombreOctetsDisponibles] ;
      System.in.read(entrée) ;
   }
} catch (IOException e)
{ System.out.println("L'exception est " + e);
}
```

12.3 La classe OutputStream

La classe OutputStream est une classe abstraite qui hérite de la classe Object. C'est la superclasse pour toutes les opérations de sortie au niveau octet.

public abstract class OutputStream

Les méthodes de cette classe sont :

- public void close() throws IOException : permet de fermer un flux de sortie
- public void flush() throws IOException : garantit que les octets dans le tampon sont bien écrits
- public void write(byte buffer[]) throws IOException : permettent d'écrire des octets sur le flux de sortie.
- public void write(byte buffer[], int offset, int length) throws IOException. Buffer est le tampon à partir du que écrire les octets, length est le nombre d'octets à écrire
- public void write(int leByte) throws IOException . offset est la position de départ du tampon à partir de laquelle écrire.

Exemple :

```
OutputStream os = … ;
os.write(123) ;
os.flush;
os.close();
```

12.4 Les flux de caractères(classes Reader et Writer)

Certaines classes dérivées des classes InputStream et OutputStream permettent la gestion de chaînes, mais elles considèrent des caractères unicode (2octets) équivalent à un octet. Ceci ne fonctionne qu'avec les caractères ISO 8859-1. Les classes Reader et Writer sont introduites dans la version 1.1 pour palier ce manque.

Les deux classes InputStreamReader et OutputStreamWriter permettent de convertir des octets en caractères et vice versa. Un schéma d'encodage peut être indiqué dans les constructeurs de ces classes.
Exemple : Les octets entrants de System.in sont convertit en caractères en les enveloppant dans un InputStreamReader. Le flux InputStreamReader est enveloppé dans BufferedReader qui offre la méthode readLine(). La méthode readLine() permet de convertir une ligne de texte en String. Puis la chaîne est convertit en un entier.

```
try
{ InputStreamReader convert = new InputStreamReader(System.in);
  BufferedReader in = new BufferedReader(convert);
  String laligne = in.readLine();
  int entire = NumberFormat.getInstance().parse(laligne).intValue();
} catch (IOException e) {}
catch (ParseException ee){}
```

12.4.1 Classe Reader

La classe Reader est la superclasse de toutes les opérations d'entrée au niveau caractère.
La classe Reader contient une variable d'instance : protected Object lock. Elle contient l'objet à verrouiller pour les sections de code critique. L'objet lock permet de synchroniser les accès à des blocs de code critique (nous n'allons pas aborder ce point dans ce cours).

L'ensemble des méthodes de cette classe sont :

- protected Reader() et protected Reader(Object lock) automatiquement appelés par des sous-classes pour créer un Reader.
- public abstract void close() throws IOExection. Elle ferme les flux d'entrée.
- public void mark(int TailleTamponPourLaLectureAnticipée). Elle marque la position en cours et fixe la taille du tampon pour la lecture anticipée.
- public boolean markSupported(). Elle vérifie si mark() et reset() fonctionnent pour une lecture anticipée.
- read(). Elle lit des octets des flux d'entrée.
- public int read() throws IOException
- public int read(char buffer[]) throws IOException
- public int read (char TamponOuLireLesCaractères[], int débutTamponDeLecture, int TailleMaxiALire) throws IOException
- public boolean ready() throws IOException. Elle vérifie la disponibilité des données pour empêcher un blocage de la lecture. Elle renvoie true si la prochaine opération de read() est assurée de ne pas avoir à attendre l'entrée des données, false si la garantie ne peut pas être donnée.

12.4.2 Classe Writer

public abstract class Writer

Writer est la superclasse de toutes les classes assurant les opérations au niveau caractères.

Cette classe contient une variable d'instance (protected Object lock) pour le verrouillage des sections critiques. Parmi les méthodes de cette classe on y trouve :

- protected Writer()
- protected Writer(Object o)
- public abstract void close() throws IOException. Elle ferme les flux de sortie
- public abstract void flush() throws IOException. Garantit que tous les caractères placés dans un tampon sont écrits.
- public void write(char [] buffer) throws IOException. Écrit les caractères dans un flux de sortie. buffer : tampon à partir duquel écrire les caractères
- public abstract void write (char[]buffer, int offset , int length) throws IOException. Position de depart du tampon où doit commencer la lecture. length est le nombre de caractères maximum à lire
- public void write(int leChar) throws IOException.
- public void write(String buffer) throws IOException
- public void write(String buffer, int offset,int length) throws IOException

Que font les programmes suivants :

```
class TestIO6
{   public void readStream (InputStream is) throws IOException
    { int b;
      while (true)
      {   b=is.read();
          if (b == -1) break;
          System.out.print ((char)b);
      }
      System.out.flush(); // public void flush()permet de vider le flux de sortie vers la cible
 }
 public static void main (String []ar)
    {  TestIO6 a=new TestIO6();
       try
       {  a.readStream(System.in);
       } catch(IOException e){}
    }
}
```

```
import java.io.*;
class TestIO8
{   public void readReader (Reader r) throws IOException
    {   int c;
        while (true)
        {   c= r.read();
            if (c == -1) break;
            System.out.print ((char)c);
        }
        System.out.flush();
    }
    public static void main (String []ar)
    {   TestIO8 a= new TestIO8();
        StringReader d = new StringReader("java");
        try
        {   a.readReader(d);
        } catch (IOException e) {  System.out.println ("Problème");    }
    }
}
```

12.5 Les classes DataInputStream et DataOutputStream

Elles permettent le lire et d'écrire des types de données de base comme les nombres et les objets String.

La classe DataInputStream :

public class DataInputStream extends FilterInputStream implements DataInput

Les classes DataInputStream et DataOutputStream permettent de lire et d'écrire les chaînes de caractères et les types de base codés sur plusieurs octets. La classe DataOutputStream implémente à son tour l'interface DataOutput. Les interfaces DataOutput et DataInput définissent les méthodes qui lisent les chaînes de caractères, des types numériques de base ainsi que des booléens.
L'exemple suivant permet la lecture d'un double à partir du clavier.

```
DataInputStream dis = new DataInputStream(System.in) ;
double x =dis.readDouble();
```

Que fait le programme suivant

```
import java.io.*;
class TestIO7
{ public static void main (String []ar)
  { DataInputStream clavier = new DataInputStream (System.in);
    try
    {   while (true)
        {   byte b2 = (byte)clavier.readByte();
        System.out.print ((char)b2);
        }
    } catch (IOException e)
    {  System.out.println ("Problème");
    }
  }
}
```

Exemple : Nous avons vu que la class DataInputStream ne permet de lire que des types numériques.

```
DataInputStream x = … ;
float y=x.readFloat() ;
```

d'un autre coté, la classe FileInputStream permet d'accéder à des octets dans les fichiers.

En java, c'est le programmeur qui doit combiner ces deux flux pour pouvoir lire des types numériques, par exemple, dans un fichier. Ceci est obtenu, en passant aux constructeurs des nouveaux objets des flux existant. Ce mécanisme est appelé flux filtrés.

Pour notre exemple utilisant les flux FileInputStream et DataInputStream, la construction du flux filtré se fait comme suit :

```
FileInputStream a = new FileInputStream("f1.txt");
DataInputStream x = new DataInputStream(a);
float y = x.readFloat();
```

La combinaison des flux est utile pour créer les flux dont on a besoin. Un autre exemple intéressant est celui de la bufférisation des données. Par défaut, les flux ne sont pas bufférisés. Ce qui nécessite un appel système pour tout transfert d'un octet dans le cas du flux FileInputStream, par exemple. Pour réduire le nombre d'appels système on peut bufférisier les octets entrant en combinant les flux BufferedInputStream et FileInputStream.

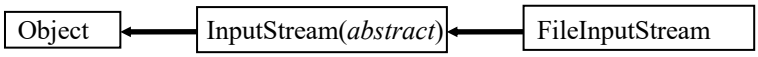
Si l'objectif est de lire des float à partir d'un fichier, en utilisant la bufférisation, voici l'empilement des flux qui permet d'atteindre cet objectif.

```
DataInputStream x =new DataInputStream ( new BufferedInputStream
                                        ( new FileInputStream ("f1.txt") ) );
```

12.6 Les classes FileInputStream et FileOutputStream

La classe java.io.FileInputStream permet de définir un flux d'entrée à partir d'un fichier

```
public class FileInputStream extends InputStream { ... }
```



Les méthodes de cette classe sont :

- public FileInputStream(File f) throws FileNotFoundException
- public FileInputStream(FileDescriptor fd)
- public FileInputStream(String f) throws FileNotFoundException

permettent de construire un FileInputStream à partir d'une source appropriée.

- protected finalize() throws IOException. Elle garantit la fermeture du fichier lorsque le garbage collector a récupéré l'espace alloué au flux
- public final FileDescriptor getFD() throws IOException. Elle donne le descripteur de fichier du flux de fichier

Exemples :

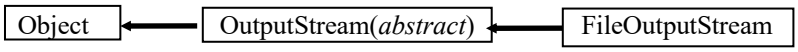
```
try
{   FileInputStream fis = new FileInputStream("leNomDeFichier");
    readStream (fis);
 } catch ( FileNotFoundException e)
{   System.out.print("Fichier introuvable");
}
```

```
import java.io.*;
class TestIO9
{   public static void main (String []ar)
   {   DataInputStream a= new DataInputStream(System.in);
       String t="";
       try
       {    while (true)
            {   byte b = (byte)a.readByte();
                if ((char)b=='0') break;
                t=t + (new Character((char)b)).toString();
            };
            FileInputStream fis = new FileInputStream (t);
       } catch (FileNotFoundException e) {System.out.println ("Fichier introuvable");}
         catch (IOException e){ System.out.println("IO");}
   }
}
```

```
import java.io.*;
class TestIO10
{  public void readStream (InputStream is) throws IOException
   {   int b;
       while (true)
       {   b=is.read();
           if (b == -1) break;      System.out.print ((char)b);
       }
       System.out.flush(); // permet de vider le flux de sortie vers la cible
   }
   public static void main (String []ar)
   { TestIO10 x =new TestIO10();
     System.out.println("donner un nom de fichier");
     DataInputStream clavier = new DataInputStream (System.in);
     String nomFichier="";
     try{  byte b2 =0;
           while (b2 !=10)
           {   b2 = (byte)clavier.readByte();
               if ((b2!=10)&&(b2!=13))nomFichier +=(char)b2;
           }
      } catch (IOException e)  {  System.out.println ("Problème"); }
     try{  FileInputStream fis = new FileInputStream (nomFichier);
           x.readStream (fis);
     } catch (FileNotFoundException e) {  System.out.println ("Fichier introuvable");  }
       catch (IOException e)  {  System.out.print("IO"+e);    }
  }
}
```

La classe FileOutputStream permet de définir un flux de sortie vers un fichier.

public class FileOutputStream extends OutputStream



Les méthodes de cette classe sont :

- public FileOutputStream(File f) throws IOException
- public FileOutputStream(FileDescriptor fd)
- public FileInputStream(String f) throws IOException.
- public final FileDescriptor getFD() throws IOException. Elle donne le descripteur de fichier du flux de fichier

Exemple :

```
import java.io.*;
class TestIO12
{   public void writeStream(OutputStream os) throws IOException
    {  byte b=-1;            System.out.print("taper le texte en terminant par le caractere 0");
       while (b !=0 )
       {   b=(byte)System.in.read();      if (b == -1) break ;
           System.out.print ((char)b);   os.write(b);
       }
       System.out.flush();       os.flush();
    }
    public static void main (String []ar)
    {     TestIO12 x =new TestIO12();
          DataInputStream clavier = new DataInputStream (System.in);
          System.out.print("donner le nom du fichier cible : ");
          String nomFichier2="";
          try
          {   byte b2 =0;
              while (b2 !=10)
              {   b2 = (byte)clavier.readByte();
                  if ((b2!=10)&&(b2!=13))nomFichier2 +=(char)b2;
              }
          } catch (IOException e)  {  System.out.println ("Problème");  }
          try
          {   FileOutputStream fos = new FileOutputStream (nomFichier2);
              x.writeStream (fos);
          } catch (FileNotFoundException e) {  System.out.println ("Fichier introuvable");}
```

12.7 La classe java.io.File

Elle permet d'accéder aux informations concernant un fichier ou un répertoire. Elle permet d'obtenir les attributs d'un fichier, de lister les entrées d'un répertoire, de savoir si un fichier existe, de créer, supprimer ou renommer un fichier. File ne permet pas un accès direct aux données du fichier.

```
import java.io.*;
class EssaiFile
{  public static void main(String[] argv) throws IOException
   {   File  repertoire;              File fichier=null;          File nouveauFichier;
       String[] listeFichiers;
       PrintWriter ecrivain;
       repertoire=new File(argv[0]);
       if (!repertoire.isDirectory()) System.exit(0);
       fichier=new File("fichier.essai");
       System.out.println("le fichier "+fichier.getName()+ (fichier.exists()?" existe":" n'existe pas"));
                          //en sortie : le fichier fichier.essai n'existe pas
       ecrivain=new PrintWriter(new FileOutputStream("fichier.essai"));
       ecrivain.println("bonjour");
       ecrivain.close();
       System.out.println("le fichier "+fichier.getName()+   (fichier.exists()?" existe":" n'existe pas"));
                          //en sortie : le fichier fichier.essai existe
       System.out.println("Sa longueur est "+fichier.length());    //en sortie : Sa longueur est 8
       System.out.println("Son chemin complet est  \n  "+fichier.getAbsolutePath());;
                //   en sortie :  Son chemin complet est
                 //  /inf/aquilon/infmd/charon/public_html/coursJava/fichiersEtSaisies/fichier.essai
       System.out.println();
       listeFichiers=repertoire.list();
       for (int i=0;i < listeFichiers.length;i++)          System.out.println(listeFichiers[i]);
       System.out.println();
       nouveauFichier=new File("autre.essai");
       fichier.renameTo(nouveauFichier);
       System.out.println("le fichier "+fichier.getName()+   (fichier.exists()?" existe":" n'existe plus"));
                //  en sortie : le fichier fichier.essai n'existe plus
       System.out.println("le fichier "+nouveauFichier.getName()+
                          (nouveauFichier.exists()?" existe":" n'existe pas"));
                     //  en sortie : le fichier autre.essai existe
       nouveauFichier.delete();
    }
}
```

L'exécution de ce programme donnera le résultat suivant :

java EssaiFile ../fichiersEtSaisies/ :

le fichier fichier.essai n'existe pas
le fichier fichier.essai existe
Sa longueur est 8
Son chemin complet est
/inf/aquilon/infmd/charon/public_html/coursJava
/fichiersEtSaisies/fichier.essai

fichier.essai
EssaiFile.java
file.html
EssaiFile.class
SaisieClavier.java
SaisieFichier.java
EssaiStream.java
EssaiStreamBis.java
LireFichierBinaire.java
generalites.html
EssaiWriter.java

EssaiFileReader.java
EssaiFileOutputStream.java
lireFichierBinaire.html
ecrireFichierTexte.html
ecrireFichierBinaire.html
saisirAuclavier.html
lireFichierTexte.html
EcrireFichierBinaire.java
EcrireFichierTexte.java

le fichier fichier.essai n'existe plus
le fichier autre.essai existe

12.8 Accès direct aux fichiers

La classe java.io.RandomAccessFile permet d'accéder de façon aléatoire aux enregistrements d'un fichier. Cette classe définit les mêmes métodes pour lire et écrire des données que les classes DataInputStream et DataOutputStream.
Un nouveau fichier peut être créé de la façon suivante :

```
try {
 RandomAccessFile a= new RandomAccessFile ("fiche.txt", "rw");
} catch (IOException e) {}
```

Le principe d'un fichier à accès direct consiste à garder une référence indiquant la position dans le fichier.
L'exemple suivant montre comment faire une copie d'un fichier dans un autre :

```
import java.io.* ;
public class Copie
{   public static void main (String [] a) throws IOException
    {   // on suppose que le nom du fichier source est correctement introduit sur la ligne de commande
        RandomAcessFile source=new RandomAcessFile(new File("a[0]","r");

        BufferedReader b = new BufferedReader(new InputStreamReader(System.in);
        File f =null;
        String s = null;
        while (f==null)
         {   try
             {  s =b.readLine();
             } catch (IOException e) {System.out.print("erreur de lecture"); System.exit(0);}
             try
             {  if (s.length==0) throw new IOException ;
                f= new File(s);
                if (f.exists())
                 { System.out.println("le fichier existe, voulez-vous l'ecraser ?");
                   try
                   {  s =b.readLine();
                   } catch (IOException e) { System.out.print("Erreur de lecture"); System.exit(0);}
                 }
                if (!s.equalsIgnoreCase("o"))
                {   f =null;
                   System.out.print("donner un autre nom");
                }
              }catch (NullPointerException e){System.out.print("probleme"); System.exit(0);}
               catch (IOException e) { System.out.print("nom incorrect, entrer un autre nom :");
            }
            int el;
            RandomAccessFile cible=new RandomAcessFile(f,"rw");
            while ((el=source.read-)!=-1)  cible.write(el);
            source.close();     cible.close();
    }
}
```

12.9 L'interface serializable

Les objet de type DataOutputStream permettent d'écrire des objets quelconque sous forme de type simple. La sérialisation est un mécanisme plus puissant et plus élégant permettant d'enregistrer des graphes d'objets sur disque. Elle permet de concerver l'état des objets entre deux exécutions d'un programme ou d'échanger des objets entre applications.

La sérialisation est un moyen automatique permettant de charger et de sauver l'état d'un objet.

Un objet quelconque implémentant l'interface serializable peut être chargé et sauvé à partir d'un flux. Le mécanisme de sérialisation par défaut sauvegarde les valeurs des variables d'instances et non les variables statiques et les transients[2].

Lorsqu'un objet est sérialisé toutes les références d'objets qu'il contient le sont également. Ceci lui donne la possibilité de sauver des graphes entiers. Il faut donc s'assurer que les objets référencés par des objets sérialisés doivent être aussi sérialisés.

Exemple :

```
Public class Personne implements Serializable
{   String nom;
    Dossier dossier;
    transient String temp ;
            //…
    public static void main(String [] a) throws IOException
    {    Dossier d = … ;
         Personne p = new Personne("p1", d);
         p.temp= "nouvelle Valeur" ;
         ObjectOutputStream o =new ObjectOutputStream(new FileOutputStream("fichier1"));
         o.writeObject(p);
         o.close();
         try
          { ObjectInputStream e= new ObjectInputStream(new FileInputStream("fichier1"));
            Personne p1 =(Personne)e.readObject();
            p1.affiche() ; // la valeur de p1.temp est perdue
            e.close();
          } catch (ClassNotFoundException e) {}
    }
 }
class Dossier implements Serializable
{ int numero
    //       …
}
```

12.10 Les flux de fichiers ZIP

Les fichiers ZIP sont des archives contenant un ou plusieurs fichiers dans un format compressé. Java 2 accepte les format GZIP et ZIP.

L'entête d'un fichier Zip contient les informations concernant le nom du fichier, la méthode de compression utilisée, etc.

[2] Le modificateur transient indique que la variable n'a pas de signification en dehors du contexte courant et qu'il est inutile de la sauvegarder.

Java permet de lire des fichiers ZIP à l'aide de la classe ZipInputStream en empilant le constructeur ZipInputStream sur un FileInputStream. De cette façon on accède individuellement à chaque entrée de l'archive.

La méthode getNextEntry permet de renvoyer une entrée de type ZipEntry. La méthode ZipInputStream permet de renvoyer –1 à la fin de l'entrée courante. Ce qui impose de faire closeEntry sur l'entrée courante pour pouvoir accéder à l'entrée suivante.
Exemple :

```
ZipInputStream zin = new ZipInputStream (new FileInputStream("nom"));
ZipEntry e;
While (e =zin.getNextEntry())!=null)
{  // traitement de e
   zin.closeEntry() ;
}
```

12.11 Complément flux d'entrées/sortie

12.11.1 Saisir des données envoyées par le clavier

Les deux exemples suivants montrent comment lire des données envoyées tout simplement par le clavier.
Les deux programmes d'illustration ont pour objectif de lire des entiers envoyés par l'intermédiaire du clavier, et d'en faire la somme. Leurs cahiers des charges diffèrent uniquement sur la façon d'indiquer la fin de la saisie.

Première méthode

La classe InputStreamReader admet un constructeur InputStreamReader(InputStream), c'est-à-dire un constructeur qui admet en paramètre un flot d'entrée. System.in est une instance de la classe InputStream. Avec une instance de InputStreamReader, on ne peut grosso modo que lire des caractères un à un.
La classe BufferedReader a un constructeur qui prend en argument une instance de Reader dont hérite la classe InputStreamReader. Cette classe permet en particulier de lire une ligne d'un texte, mais en revanche, on ne peut pas lui demander de lire un entier, un double etc...
On lit ici ce qui est envoyé par le clavier ligne par ligne et on découpe le contenu de chaque ligne avec un StringTokenizer pour récupérer les entiers attendus.
Vous devez taper return deux fois de suite pour interrompre la saisie.
Voici le programme que nous proposons.

```
import java.io.*;
import java.util.*;
class SaisieClavier
{   public static void main (String[] argv) throws IOException, NumberFormatException
    {   int somme = 0;
        String ligne;
        StringTokenizer st;
        BufferedReader entree = new BufferedReader(new InputStreamReader(System.in));
        ligne = entree.readLine();
        while(ligne.length() > 0)
        {     st = new StringTokenizer(ligne);
              while(st.hasMoreTokens())
                somme += Integer.parseInt(st.nextToken());
              ligne = entree.readLine();
         }
        System.out.println("La somme vaut : "+somme);
     }
}
```

Deuxième méthode

On utilise ici une instance de StreamTokenizer qui est un analyseur syntaxique plutôt rudimentaire.
Un constructeur de la classe StreamTokenizer prend en paramètre une instance de Reader.
La méthode nextToken() de la classe StreamTokenizer retourne le type de l'unité lexicale suivante, type qui est caractérisé par une constante entière. Cela peut être :
TT_NUMBER si l'unité lexicale représente un nombre. Ce nombre se trouve alors dans le champ nval de l'instance de StreamTokenizer. Ce champ est de type **double**.
TT_WORD si l'unité lexicale représente une chaîne de caractères. Cette chaîne se trouve alors dans le champ sval de l'instance du StreamTokenizer.
TT_EOL si si l'unité lexicale représente une fin de ligne. La fin de ligne n'est reconnue comme unité lexicale que si on a utilisé l'instruction
nomDuStreamTokenizer.eolIsSignificant(**true**);
TT_EOF s'il s'agit du signe de fin de fichier.

```
import java.io.*;
class EssaiStream
{   public static void main (String[] argv) throws IOException
    {    int  type, somme = 0;
         StreamTokenizer entree= new StreamTokenizer(new InputStreamReader(System.in));
         String fin=new String("fin");
         System.out.println("Donnez vos entiers, terminez avc fin");
         while(true)
         { type=entree.nextToken();
           if (type==StreamTokenizer.TT_NUMBER) somme += (int)entree.nval;
           else if ((type == StreamTokenizer.TT_WORD)&& (fin.equals(entree.sval)))
             break;
         }
         System.out.println("La somme vaut : " + somme);
    }
}
```

Voici une exécution :
Donnez vos entiers, terminez avec fin
1 -2 bonjour 4
3 fin 2
La somme vaut : 6

12.11.2 Lire ou écrire des caractères dans un fichier

Le programme ci-dessous écrit une chaîne de caractères dans un fichier nommé copie_essai.txt puis copie le fichier essai.txt caractère par caractère dans le fichier copie_essai.txt.

```
import java.io.*;
class LireEcrireTexte
{  public static void main(String[] argv) throws IOException
   {    FileReader lecteur;
        FileWriter ecrivain;
        int c;
        lecteur = new FileReader("essai.txt");
        ecrivain = new FileWriter("copie_essai.txt");
        ecrivain.write("copie de essai.txt\n");
        while((c = lecteur.read()) != -1)   ecrivain.write(c);
        lecteur.close();        ecrivain.close();
   }
}
```

12.11.3 Écrire dans un fichier texte

On utilise ici une instance de PrintWriter, dont un constructeur prend en argument un Writer dont la classe BufferedWriter hérite.
Vous pouvez utiliser avec une instance de PrintWriter les méthodes print et println de la même façon qu'avec System.out (qui est de la classe PrintStream). La classe PrintWriter (qui hérite de la classe Writer) ne fait qu'améliorer la classe PrintStream (qui hérite de OutStream).
On aurait pu plus simplement initialiser ecrivain par :

```
ecrivain = new PrintWriter(new FileWriter(argv[0]));
```

mais alors les écritures n'utiliseraient pas de mémoire-tampon.

```
import java.io.*;
class EcrireFichierTexte
{  public static void main(String[] argv) throws IOException
   {   PrintWriter ecrivain;
       int n = 5;
       ecrivain =  new PrintWriter(new BufferedWriter(new FileWriter(argv[0])));
       ecrivain.println("bonjour, comment cela va-t-il ?");
       ecrivain.println("un peu difficile ?");
       ecrivain.print("On peut mettre des entiers : ");
       ecrivain.println(n);
       ecrivain.print("On peut mettre des instances de Object : ");
       ecrivain.println(new Integer(36));
       ecrivain.close();
    }
}
```

Apres exécution, le fichier indiqué contient :

```
bonjour, comment cela va-t-il ?
un peu difficile ?
On peut mettre des entiers 5
On peut mettre des instances de Object : 36
```

12.11.4 Lire dans un fichier texte

Premier exemple

Il s'agit ici de lire un fichier de type texte ligne par ligne et de reproduire ce qui est lu directement à l'écran. On compose pour cela un BufferdReader avec un FileReader.
Cet exemple est particulièrement simple.

```
import java.io.*;
class LireLigne
{   public static void main(String[] argv) throws IOException
    {   BufferedReader lecteurAvecBuffer = null;
        String ligne;
        try
        {  lecteurAvecBuffer = new BufferedReader (new FileReader(argv[0]));
         }  catch(FileNotFoundException exc)  { System.out.println("Erreur d'ouverture");  }
        while ((ligne = lecteurAvecBuffer.readLine()) != null)  System.out.println(ligne);
        lecteurAvecBuffer.close();
     }
}
```

A l'exécution, on obtient en sortie exactement le contenu du fichier dont le nom est indiqué sur la ligne de commande.

Deuxième exemple

Il s'agit maintenant de lire des entiers dans un fichier ne contenant que des entiers et d'en faire la somme. Les classes utilisées ici l'on déjà été dans les exemples précédents.

```
import java.io.*;
class LireEntiers
{   public static void main (String[] argv) throws IOException
    {   int somme = 0;
        FileReader fichier = new FileReader(argv[0]);
        StreamTokenizer entree = new StreamTokenizer(fichier);
        while(entree.nextToken() == StreamTokenizer.TT_NUMBER)
           somme += (int)entree.nval;
        System.out.println("La somme vaut : " + somme);
        fichier.close();
    }
}
```

Avec un fichier contenant :

```
5 3 6 2 7
-10 23
```

on obtient : La somme vaut : 36

12.11.5 Écrire dans un fichier binaire

Pour traiter des fichiers binaires, c'est-à-dire qui contienne éventuellement autres choses que des caractères (des **int**(s) écrits en binaire et pas comme des chaînes de caractères...), on peut utiliser la classe DataOutputStream. Celle-ci à un constructeur qui prend en paramètre une instance de la classe OutputStream. On aurait pu initialiser notre variable ecrivain par : DataOutputStream(**new** FileOutputStream(argv[0]));

Cela aurait été identique en apparence, mais si on compose comme il est fait ici avec une instance de BufferedOutputStream, les écritures dans le fichier utilisent une mémoire-tampon, ce qui gagne du temps.

La classe OutputStream dispose de beaucoup de méthodes ; nous donnons des exemples ci-dessous des principales méthodes.

Voici le programme que nous proposons.

```
import java.io.*;
class EcrireFichierBinaire
{   public static void main(String[] argv) throws IOException
    {  DataOutputStream ecrivain;
       ecrivain =  new DataOutputStream(new BufferedOutputStream
                                        (new FileOutputStream(argv[0])));
       ecrivain.writeUTF("bonjour");          ecrivain.writeInt(3);
       ecrivain.writeLong(100000);            ecrivain.writeFloat((float)2.0);
       ecrivain.writeDouble(3.5);             ecrivain.writeChar('a');
       ecrivain.writeBoolean(false);          ecrivain.writeUTF("au revoir");
       System.out.println(ecrivain.size());   ecrivain.close();
    }
}
```

12.11.6 Lire dans un fichier binaire

La classe DataInputStream nous permet de lire un fichier binaire. D'autre méthodes de cette classe pourront être consultées.
Si on ne désire pas utiliser de mémoire-tampon, on peut initialiser la variable lecteur plus simplement par : lecteur= **new** DataInputStream(**new** FileInputStream(argv[0]));

```
import java.io.*;
class LireFichierBinaire
{   public static void main(String[] argv) throws IOException
    {   DataInputStream lecteur;
        lecteur=  new DataInputStream(new BufferedInputStream (new FileInputStream(argv[0])));
        System.out.println(lecteur.readUTF());
        System.out.println(lecteur.readInt());
        System.out.println(lecteur.readLong());
        System.out.println(lecteur.readFloat());
        System.out.println(lecteur.readDouble());
        System.out.println(lecteur.readChar());
        System.out.println(lecteur.readBoolean());
        System.out.println(lecteur.readUTF());
        lecteur.close();
    }
}
```

TD-TP n° : Les flux d'entrées et de sorties

Exercice 1 :

L'attribut in de la classe System utilise une version de la méthode read() qui permet de lire des octets à partir du clavier.
Ecrire un programme qui permet de lire 10 caractères introduit au clavier et les affiche à l'écran. (pour convertir un tableau de bytes en String penser aux constructeurs de la classe String) (Test30)

Exercice 2 :

La méthode read() renvoie un int. Une valeur de retour –1 indique que la fin du flux est atteinte. Ecrire une méthode qui permet de lire les valeurs au clavier et de remplir un tableau de byte de taille 10. Puis de les afficher à l'écran. (Test31)

Exercice 3 :

La méthode available() de la classe InputStream détermine le nombre d'octets accessible en lecture à un moment donné. Ce qui évite au programme de rester en attente. Ecrire le programme qui permet de créer un tableau de byte qui va contenir exactement le nombre d'octets en attente dans le stream d'entrée. (Test32)

Exercice 4 :

Ecrire un programme (TestIO11.java) qui permet de lire simultanément deux fichiers et d'afficher en alternance les lignes de ces fichiers.

Exercice 5 :

Ecrire un programme (TestIO13.java) qui permet de créer un fichier, puis de copier le contenu d'un fichier existant dans le fichier crée.

Exercice 6 :

Créer un fichier permettant un accès aléatoire en lecture/écriture. Le fichier doit contenir les cases d'une matrice d'entiers

a11	…	a1i	…	a1n
…	…	…	…	…
aj1	…	aji	...	ajn
...	...	...	...	...
am1	…	ami	...	amn

- Ecrire la méthode qui permet de sauver cette matrice dans un fichier.
- Ecrire la méthode qui permet de lire aléatoirement une case quelconque de la matrice à partir du fichier.

Exercice 7 :

Ecrire le programme qui permet de lire des entiers à partir d'un fichier et de les ajouter à la fin d'un fichier existant (indication revoir les constructeurs de la class FileOutputStream).

Exercice 8 :

Ecrire un programme qui permet d'afficher les fichiers qui se trouve dans un fichier ZIP.

Eléments de Réponses

Solution exercice 1 :

```
import java.io.*;
class Test30
{   public static void main (String arg[])
    {   byte []b= new byte[2];
        try
        {   System.in.read(b);
        }catch(IOException e){}
       System.out.println(new String(b));
    }
}
```

Solution exercice 4 :

```
import java.io.*;
class TestIO11
{  public void readStream(InputStream is1,InputStream is2) throws IOException
   { int b;
     InputStream is = is1;
     boolean v= false;
     while (true)
     {   b=is.read();
         if (b == -1)  { if (v==true) break ;   else {  v=true;   if (is==is1) is=is2; else is =is1;} }
         if (b==10)   { if ( v==false ) { if (is==is1) is=is2; else is =is1; }  }
         System.out.print ((char)b);
      }
     System.out.flush(); // public void flush()permet de vider le flux de sortie vers la cible
   }
   public static void main (String []ar)
   {  TestIO11 x =new TestIO11();
      System.out.println("donner un nom du premier fichier");
      DataInputStream clavier = new DataInputStream (System.in);
      String nomFichier1="";
      try
      {   byte b2 =0;
          while (b2 !=10)
          {   b2 = (byte)clavier.readByte();
              if ((b2!=10)&&(b2!=13))nomFichier1 +=(char)b2;
           }
       } catch (IOException e) {  System.out.println ("Problème");         }
       System.out.println("donner un nom du deuxieme fichier");
       String nomFichier2="";
       try
       {   byte b2 =0;
           while (b2 !=10)
           {  b2 = (byte)clavier.readByte();
              if ((b2!=10)&&(b2!=13))nomFichier2 +=(char)b2;
           }
        } catch (IOException e) {  System.out.println ("Problème"); }
        try
        {   FileInputStream fis1 = new FileInputStream (nomFichier1);
            FileInputStream fis2 = new FileInputStream (nomFichier2);
            x.readStream (fis1,fis2);
         } catch (FileNotFoundException e) {  System.out.println ("Fichier introuvable");
         } catch (IOException e){ System.out.print("IO"+e); }
    }
}
```

Solution exercice 2 :

```
import java.io.*;
class Test31
{  public static void main (String arg[])
   {  byte []d = new byte[10];
      int t=0;
      try
      {  do
         {  int val = System.in.read();
            if (val!=-1) d[t++]=(byte)val;
            else
            {  System.out.println("Fin du Stream");
               break;}
         } while(t!=10);
         System.out.println(new String(d));
      }catch(IOException e){}
   }
}
```

Solution exercice 3 :

```
import java.io.*;
class Test32
{  public static void main (String arg[])
   {  try
      {  int val = System.in.available();
         if (val>0)
         {  byte []d = new byte[val];
            System.in.read(d);
            System.out.println(new String(d));
         }
         else System.out.println("val="+val);
      }catch(IOException e){}
   }
}
```

Solution exercice 5 :

```
import java.io.*;
class TestIO13
{  public void readwriteStream(InputStream is,OutputStream os) throws IOException
   {  int b;
      while (true)
      {  b=is.read();                    if (b == -1) break ;
         System.out.print ((char)b);     os.write(b);
      }
      System.out.flush();          os.flush();
   }
   public static void main (String []ar)
   {  TestIO11 x =new TestIO11();
      System.out.print("donner le nom du fichier source : ");
      DataInputStream clavier = new DataInputStream (System.in);
      String nomFichier1="";
      try
      {  byte b2 =0;
         while (b2 !=10)
         {  b2 = (byte)clavier.readByte();
            if ((b2!=10)&&(b2!=13))nomFichier1 +=(char)b2;
         }
      } catch (IOException e)  {  System.out.println ("Problème"); }
      System.out.print("donner le nom du fichier cible : ");
      String nomFichier2="";
      try
      {  byte b2 =0;
         while (b2 !=10)
         {  b2 = (byte)clavier.readByte();
            if ((b2!=10)&&(b2!=13))nomFichier2 +=(char)b2;
         }
      } catch (IOException e) {  System.out.println ("Problème"); }
      try
      {  FileInputStream fis = new FileInputStream (nomFichier1);
         FileOutputStream fos = new FileOutputStream (nomFichier2);
         x.readStream (fis,fos);
      } catch (FileNotFoundException e) {  System.out.println ("Fichier introuvable");}
         catch (IOException e){ System.out.print("IO"+e);     }
   }
}
```

13 Les JDBC Java DataBase Connectivity [3]

13 LES JDBC JAVA DATABASE CONNECTIVITY 33

13.1 INTRODUCTION 33
13.2 JDBC ET LES ARCHITECTURES CLIENT-SERVEUR 34
13.3 TYPOLOGIE DE PILOTES (DRIVERS) JDBC 35
13.4 APPLICATIONS TYPIQUE 36
13.5 STRUCTURE D'UNE APPLICATION JDBC 36
13.6 API JDBC 36
13.6.1 Charger un pilote en mémoire 36
13.6.2 Etablir une connexion 37
13.6.3 Traitement des requêtes SQL simples 38
13.6.4 Récupération des résultats 40
13.6.5 Fermeture d'une connexion 41
13.6.6 Appel à des procédures précompilées 42
13.7 ACCES AUX META-DONNEES 43
13.8 PRISE EN CHARGE DES TRANSACTIONS 43
13.9 CREATION D'UNE SOURCE DE DONNEES DANS ODBC 44
13.10 CREATION D'UNE SOURCE DE DONNEES DANS ODBC 44
13.11 ANNEXES 45

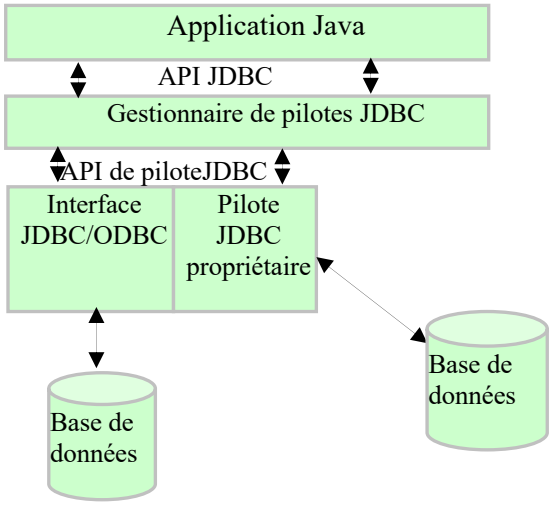
13.1 Introduction

La première version du kit JDBC utilisant SQL pour java a été proposée en 1996. Le principal avantage des programmes développés en java et JDBC est leur caractère universel. En effet, une application écrite dans ce langage peut s'exécutée indifféremment sous Windows, Solaris au un autre environnement.
Cependant, le kit JDK n'offre actuellement aucun outil de programmation de base de données, d'ou la nécessité d'utiliser d'autre logiciels pour la création de formulaire et la gestion des requêtes.

Sun fournit une API pour les accès SQL avec un gestionnaire de pilotes. Pour permettre aux autres pilotes de se connecter aux différentes bases de données. Cette API est l'API JDBC, les règles d'écriture de pilotes sont définies dans le pilote JDBC. Ce protocole reprend le modèle d'ODBC de Microsoft.
Le JDBC est basée sur la même idée que l'ODBC. Le JDBC comprend deux niveaux :

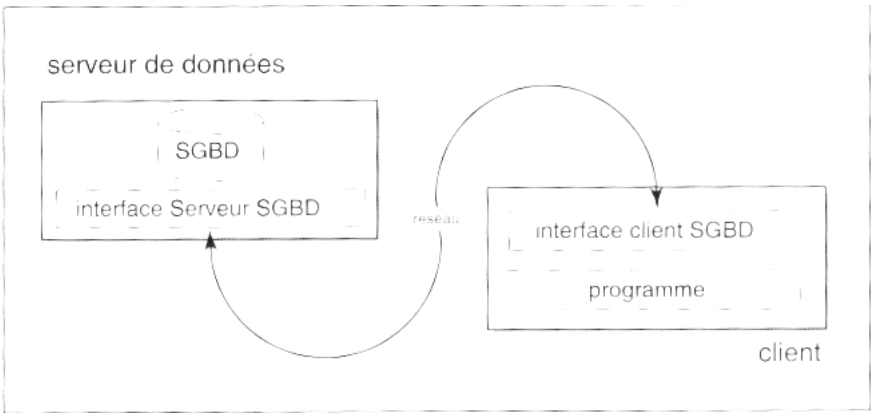
- API JDBC : elle communique avec les API des gestionnaires de pilotes JDBC en leur envoyant des requêtes SQL.
- Le gestionnaire communique avec les autres pilotes qui sont effectivement connectés aux bases de données et renvoie les réponses de façon transparente aux programmeurs.

La couche JDBC représente la partie utilisable par les programmeurs comme le montre figure suivante :



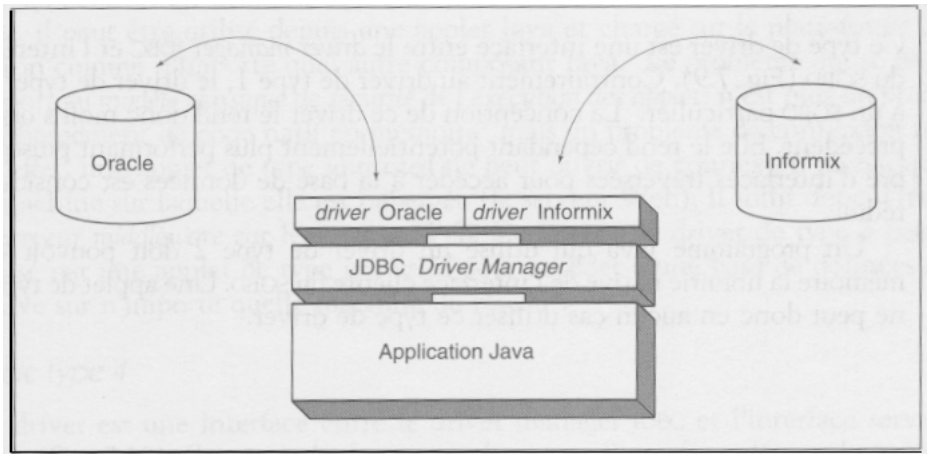
13.2 JDBC et les architectures client-serveur

En mode client/serveur, un programme client s'adresse à un programme qui s 'exécute sur une machine distante (le serveur) pour échanger des informations et des services.
pour un SGBD, le client communique avec une *interface client* du SGBD



Une application qui utilise JDBC est structurée en 2 couches

- la première est orientée vers le programme Java. Elle est composée du *driver manager* qui est un objet Java capable de communiquer avec les autres objets java.
- la seconde est orientée vers le SGBD. Elle utilise des pilotes spécifiques à chaque SGBD.

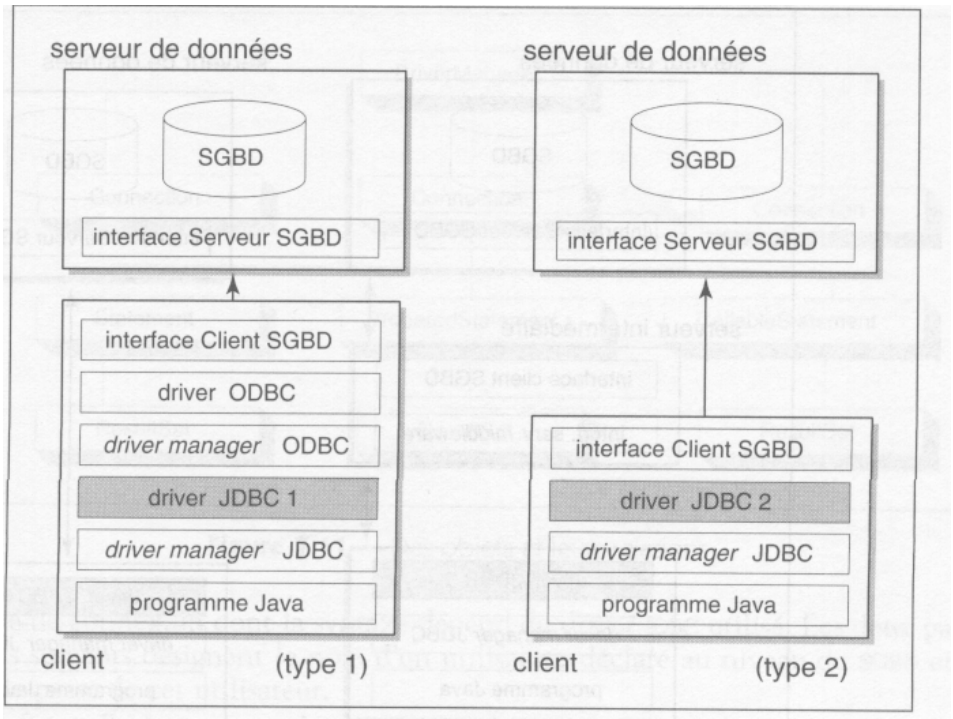


13.3 Typologie de pilotes (Drivers) JDBC

Les pilotes JDBC sont classés en plusieurs types :

- un pilote de type 1 (pont JDBC-ODBC). Il traduit JDBC en ODBC et se sert d'un pilote ODBC pour communiquer avec les bases de données. Cette passerelle est fournie avec Java 2. Cette interface ne supporte pas JDBC2 et nécessite une configuration particulière d'un pilote ODBC.
  Le pilote convertit un appel JDBC standard en un appel ODBC correspondant et l'envoie dans une source de données ODBC, via les bibliothèques ODBC du système d'exploitation. Les bases de données, pa exemple, ACCESS utilisent se type.
  Dans cette architecture, le driver JDBC doit charger en mémoire des librairies dynamiques (dll).

- le pilote de type 2 est partiellement écrit en java et partiellement dans un langage natif, qui communique avec l'API cliente d'une base de données.
  Un pilote de ce type utilise un pilote Java pour communiquer avec une API propre à un fournisseur. Il est identique au pilote de type 1 à ceci près qu 'il doit traverser une couche de moins (conversion ODBC).
  La base de données traite la requête et envoie le résultat au pilote JDBC qui convertit les résultats au standard JDBC et les envoie au programme.
  Ce type de pilote charge en mémoire la librairie native de l 'interface cliente du SGBD.



- Un pilote de type 3 est une bibliothèque en java pur qui se sert d'un protocole indépendant de la base de données pour communiquer les requêtes de base de données à un serveur qui se charge de traduire les requêtes dans un protocole propre aux bases de données.

  Le programme Java envoie directement un appel JDBC (via le pilote JDBC) dans le niveau intermédiaire, sans aucune conversion.

  Ce niveau traite alors la requête au moyen d'un autre pilote JDBC.

  L'inconvénient est que le niveau intermédiaire communique peut-être avec la base de données au moyen d'un pilote de type 1 ou 2.

- Un pilote de type 4 est une bibliothèque en java pur qui traduit directement des requêtes JDBC en un protocole spécifique à la base de données

  C'est la méthode la plus performante et la plus simple à déployer, car elle dispense d'installer des bibliothèques ou des couches intermédiaires

La plupart des concepteurs fournissent des pilotes de type 3 ou 4 avec leur base de données.

13.4 Applications typique

Les JDBC peuvent être utilisés dans les applications ou dans les applets.
Le modèle classique d'accès aux bases de données se fait sur le modèle client-serveur (client – JDBC-serveur de DB). Des modèles en trois voir n niveaux tendent à devenir les plus utilisés. Dans un modèle à trois niveaux, le client ne fait aucun accès directe à la base de données : il appelle une couche intermédiaire du serveur qui exécute la requête.

13.5 Structure d'une application JDBC

Pour effectuer un traitement avec une base de données, il faut:

- charger un pilote en mémoire
- établir une connexion
- récupérer les informations relatives à la connexion
- exécuter des requêtes SQL et/ou des procédures stockées
- fermer la connexion

13.6 API JDBC

Il convient d'utiliser le package java.sql
Elle contient une classe et plusieurs interfaces.Les interfaces sont implémentées dans le pilote du SGBD spécifique au type de base de données.

13.6.1 Charger un pilote en mémoire

Pour pouvoir utiliser une base de données, il faut charger en mémoire un pilote JDBC spécifique au SGBD.

- Classe
- DriverManager encapsule le gestionnaire de pilote JDBC.
- Permet de charger en mémoire un pilote JDBC
- Permet de créer des connexions avec le SGBD

- Tient à jour une liste des implantations de drivers

Pour définir le pilote, on peut utiliser soit:
- la propriété système jdbc.drivers ou
- la méthode Class.forName (String drivername) , lors de l'exécution

Exemple

```
try
{ Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
}
catch (ClassNotFoundException e){…….}
```

charge dynamiquement la classe dont le nom est indiqué en paramètre
- une instance du pilote spécifique est créée
- le pilote en question se met dans la liste du DriverManager

13.6.2 Etablir une connexion

Pour établir une connexion entre le programme java et une base de données, il faut:
- Définir la base de données
- Utiliser l'interface Connection

Définir la base de données

Définir le nom de la base de données

JDBC:<sous-protocole>:<nomBase>

Exemples:

String url="jdbc:odbc:TestJava"
String url="jdbc:oracle://www.lipn.univ-paris13.fr/client"

Utilisation de l'interface Connection

1. Représente une connexion avec le SGBD.
2. Offre des services pour récupérer des références sur les objets qui encapsulent des ordres SQL.
3. Permet également de gérer des transactions

Demander au driverManager d'ouvrir une connexion à la base de données. Si le pilote est répertorié dans le fichier des propriétés système ou indiqué lors de l'exécution, la connexion est établie au moyen d'une méthode de la classe DriverManager :
- public static synchronized Connection getConnection(String url,
- java.util.Properties info) throws SQLException
- public static synchronized Connection getConnection(String url,
- String user, String pwd) throws SQLException
- public static synchronized Connection getConnection(String url) throws SQLException

L'une des méthodes getConnection ( ) de la classe DriverManager permet d'envoyer un objet Connection au programme.

Cet objet sert alors à exécuter toutes les opérations sur la base de données.
Si le pilote JDBC est introuvable, le programme lance une exception ClassNotFoundException.
S'il est impossible d'établir la connexion, une exception SQLException est lancée.
Exemple :

```
Connection connex; //objet servant à la connexion
String url="jdbc:odbc:TestJava"

  try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    connex=java.sql.DriverManager.getConnection(url);
    Statement state=connex.createStatement();
}
   catch(ClassNotFoundException e)
    { System.out.println(e );}
   catch(SQLException e){
        System.out.println(e );}
```

13.6.3 Traitement des requêtes SQL simples

Les instructions SQL sont exécutées à l'aide de l'objet Connection créé par la méthode getConnection ( ) de la classe DriverManager.
Il existe 3 méthodes différentes pour envoyer une requête à la base de données:

- exécution d'une requête simple: utilisation de l'objet Statement
- exécution de plusieurs requêtes identiques: utilisation de l'objet PreparedStatement
- exécution d'une procédure stockée dans la base de données: utilisation de l'objet CallableStatement.

- Statement createStatement() throws SQLException
- Statement createStatement(int resultSetType, int resultSetConcurrency)
  throws SQLException
- PreparedStatement prepareStatement(String sql) throws SQLException
- PreparedStatement prepareStaternent(String sql, int resultSetType, int resultSetConcurrency)
  throws SQLException
- CallableStatement prepareCall(String sql) throws SQLException
- CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency)
  throws SQLException

Chaque méthode permet d'indiquer des options. Le type du groupe de résultats détermine s'il est possible de faire défiler le groupe vers l'arrière, et la simultanéité (concurrency) s'il est possible de mettre le groupe à jour.

Les constantes suivantes de l'interface Connection définissent les valeurs susceptibles d'être utilisées pour chaque méthode.

- int TYPE_FORWARD_ONLY = 1003;
- int TYPE_SCROLL_INSENSITIVE = 1004;
- int TYPE_SCROLL_SENSITlVE = 1005;
- int CONCUR_READ_ONLY = 1007 ;
- int CONCUR_UPDATEABLE = 1008 ;

Exécution d'une requête simple

Utilisation de la méthode Statement createStatement() throws SQLException

Elle génère un objet Statement, qui permet d'exécuter une instruction SQL produisant ou non des résultats.

- ResultSet executeQuery(String sql) throws SQLException
- void executeUpdate(String sql) throws SQLException

executeUpdate ( ) sert à exécuter les instructions SQL ne générant aucun résultat (par exemple, INSERT, UPDATE ou DELETE).

executeQuery( ) permet d'exécuter une requête SELECT sur la base de données et d'extraire un groupe d'enregistrements sous forme ResultSet.

Lecture du résultat
Il est possible de se déplacer dans un objet ResultSet et de le mettre à jour. (voir Interface ResultSet).
Quelques méthodes utiles:

- boolean next() throws SQLException
- String getString(int columnIndex) throws SQLException
- String getString(String columnName) throws SQLException
- void updateString() throws SQLException
- void updateRow() throws SQLException
- void deleteRow() throws SQLException
- void refreshRow() throws SQLException
- void cancelRowUpdates() throws SQLException

L'objet ResultSet correspond à un curseur. Le curseur est initialement défini devant la première ligne. La méthode next ( ) de l'objet ResultSet fait avancer le curseur d'une position. A chaque appel de next ( ), la méthode retourne true s'il existe une ligne valide, et false sinon.

Exemple:

```
String texteCreat= "CREATE TABLE produit (idProduit string(5), Pnom string(20), "+ "Couleur
string(10), Poids integer);";
   state.executeUpdate(texteCreat);

    String sql1="select * from produit;"
    ResutlSet resu=state.executeQuery(sql1);
    //affichage
     System.out.println("table signifiant");
     while (resu.next())
     {
       System.out.println(resu.getString("identifier"));
       }
```

Utilisation de l'interface Statement

- encapsule des ordres SQL de type DML(SELECT, INSERT, UPDATE, DELETE) ou de type DDL (CREATE, DROP)

Utilisation de la méthode Statement createStatement() throws SQLException

L'interface Statement possède les méthodes nécessaires pour exécuter des requêtes sur la base.

- Requête de selection : ResultSet executeQuery(String req) throws SQLException
- Requête de mise à jour : int executeUpdate(String req) throws SQLException

Exemple

```
Connection connex;  //objet servant à la connexion
  String url=" jdbc:odbc:TestJava"
  try
 {   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
     connex=java.sql.DriverManager.getConnection(url);
      Statement state=connex.createStatement();
 }  catch(ClassNotFoundException e) { System.out.println(e );}
     catch(SQLException e){ System.out.println(e );}
  String texteCreat= "CREATE TABLE produit (idProduit string(5), Pnom string(20), "
                          + "Couleur string(10), Poids integer);";
  state.executeUpdate(texteCreat);
  String sql1="select * from produit;"
   ResutlSet resu=state.executeQuery(sql1);
```

13.6.4 Récupération des résultats

Utilisation de l'interface ResultSet. Le résultat se trouve dans un objet ResultSet. (*comme une vecteur d'enregistrements, chaque enregistrement correspond à une ligne d'une table relationnelle=tuple*)
Le parcours du ResultSet se fait par la méthode next qui fait avancer le pointeur d 'une position et retourne *true* s 'il existe une ligne valide.
Une fois le curseur positionné sur un enregistrement, on accède à la colonne voulue par la méthode getXXX() (XXX est un nom de type java correspondant au type de la colonne
Remarque: avant le premier appel à next(), le curseur est positionné avant le premier enregistrement.

Exemple

```
 //affichage
   System.out.println("table produit");
   while (resu.next())
   {
    System.out.println(resu.getString("idProduit"));
   }
```

Quelques méthodes utiles:

- boolean next() throws SQLException
- String getString(int columnIndex) throws SQLException
- String getString(String columnName) throws SQLException
- void updateString() throws SQLException
- void updateRow() throws SQLException
- void deleteRow() throws SQLException
- void refreshRow() throws SQLException

- void cancelRowUpdates() throws SQLException

Il est possible de se déplacer dans un objet ResultSet et de le mettre à jour. (voirInterface ResultSet)

Quelques méthodes utiles:

- boolean next() throws SQLException
- String getString(int columnIndex) throws SQLException
- String getString(String columnName) throws SQLException
- void updateString() throws SQLException
- void updateRow() throws SQLException
- void deleteRow() throws SQLException
- void refreshRow() throws SQLException
- void cancelRowUpdates() throws SQLException

L'objet ResultSet correspond à un curseur.
Le curseur est initialement défini devant la première ligne.
La méthode next ( ) de l'objet ResultSet fait avancer le curseur d'une position.
A chaque appel de next ( ), la méthode retourne true s'il existe une ligne valide, et false sinon.

Exemple:

```
String texteCreat= "CREATE TABLE produit (idProduit string(5), Pnom string(20), "+ "Couleur string(10), Poids integer);";
   state.executeUpdate(texteCreat);

    String sql1="select * from produit;"
    ResutlSet resu=state.executeQuery(sql1);
    //affichage
     System.out.println("table signifiant");
     while (resu.next())
     {
       System.out.println(resu.getString("identifier"));
     }
```

13.6.5 Fermeture d'une connexion

Il faut fermer la connexion à une base de données. Cette opération sert à marquer l'objet en vue de son traitement par le ramasse-miette et de son nettoyage.

On envoie le message close à l'objet Statement qui sert à la connexion, aux divers ResultSet et à la connexion :

```
try
{   state.close();
    connex.close();
    resu.close();
}
catch(Exception) {..}
```

Prise en charge d'une transaction

A sa création, un objet Connection est défini de manière à valider (commit) toute transaction. Si une instruction SQL est exécutée on ne peut l'annuler. On peut cependant définir une transaction avec les méthodes suivantes:

void setAutoCommit(boolean autoCommit) throws SQLException
void commit() throws SQLException
void rollback()throws SQLException

Il faut commencer par appeler la méthode setAutoCommit et la positionner à faux.
L'appel de commit() valide la transaction depuis la dernière validation.
L'appel de rollBack() annule la transaction depuis la dernière validation.

13.6.6 Appel à des procédures précompilées

Utilisation d'une sous-interface de Statement : PreparedStatement

Si on exécute plusieurs fois une même instruction SQL, il faut utiliser un objet de type PreparedStatement au lieu de Statement. PreparedStatement est précompilé avant d'être utilisé.

La version précompilée accepte un nombre quelconque de paramètres.
En créant un PreparedStatement (à l'aide de la méthode prepareStatement ( ) de l'objet Connection), vous pouvez insérer un caractère " ? ", pour marquer l'emplacement d'un paramètre à ajouter.

- PreparedStatement propose des méthodes permettant de définir les valeurs des paramètres.

Comment donner une valeur à un paramètre

Pour donner la valeur au paramètre, il faut utiliser les méthodes de la classe PreparedStatement:

void setXXX(int i, XXX val);

qui positionne la ième colonne du tuple qui est du type XXX à la valeur val
pst.setString(int i, "Dupont ");
pst.executeUpdate();
pour mettre à NULL, utiliser setNull()

Exemple

```
public void insertion()
{      //création de l'instruction
    StringBuffer sql= new StringBuffer(512);
    PreparedStatement stmt;
    sql.append("INSERT INTO produit values (?,?,?,?)") ;
    try
    {  stmt=connexion.prepareStatement(sql.toString()) ;
    } catch(SQLException e){ System.err.println("erreur sql "+e); return; }
        //insertion d'éléments
    String rep=null;
    do
    {  try
        {    stmt.clearParameters();  //réinitialisation des paramètres
             String elem=Console.readString("identifiant du produit?").trim();
             if(elem.length()==0) return;
             stmt.setString(1,elem);
             elem=Console.readString("nom du produit?").trim();
             if(elem.length()==0) return;
             stmt.setString(2,elem);
             elem=Console.readString("couleur du produit?").trim();
```

```
            if(elem.length()==0) return;
            stmt.setString(3,elem);
            elem=Console.readString("poids du produit?").trim();
            if(elem.length()==0) return;
            stmt.setInt(4,Integer.parseInt(elem));
            stmt.executeUpdate();
        } catch(SQLException e){System.err.println("erreur sql "+e); }
        rep= Console.readString("Voulez-vous continuer o/n ?").trim();
    }
    while(rep.equals("o"));
}//fin
```

13.7 Accès aux méta-données

Le package java.sql contient 2 interfaces spécialisés dans les accès dynamiques:

- les objets de type DatabaseMetaData indiquent au moment de l'exécution les éléments SQL supportés par la base et la structure des données de la base
- les objets ResultSetMetaData découvrent, à l 'exécution, les propriétés des champs de la table résultat de la requête SQL.

Comment accéder à un objet de type DatabaseMetaData?
Utiliser la méthode getMetaData() définie dans Connection
Exemple:

```
connexion=java.sql.DriverManager.getConnection(urlBD);
DtabaseMetaData dba=connexion.getMetaData();
//acces aux tables
String[] objectCat={"TABLE"};
ResultSet rs= dba.getTables(null,null,null,objectCat);
```

Comment accéder à un objet de type ResultSetMetaData?
Utiliser la méthode getMetaData() définie dans ResultSet
Exemple

```
//acces aux métadonnees
//la table est référencée par rs
ResultSetMetaData rsMD=rs.getMetaData();
//parcours de la table
//nombre de colonnes
int nbCol=rsMD.getColumnCount();
 while(rs.next())
          {
          int i=0;
          while(i<nbCol)
          {
            String nomCol=rsMD.getColumnName(i+1);//les noms de colonnes commencent à 1
              System.out.println(i+" "+rs.getString(nomCol));
            i++;
          }
          System.out.println();
        }
```

13.8 Prise en charge des transactions

JDBC valide automatiquement chaque instruction SQL qu 'il transmet à la base de données (mode commit).

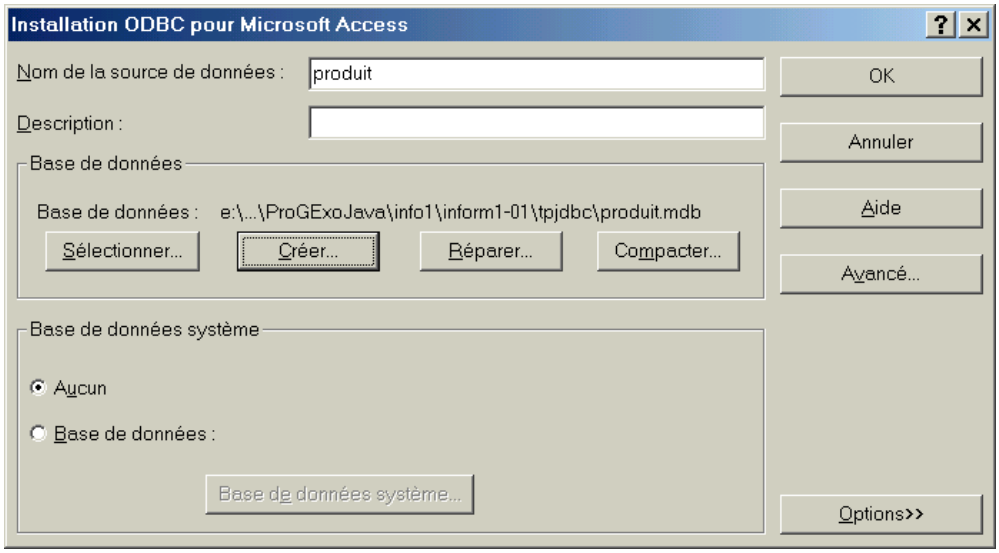
Pour désactiver la validation automatique et définir une transaction, on utilise les méthodes les méthodes suivantes de l'interface Connection:

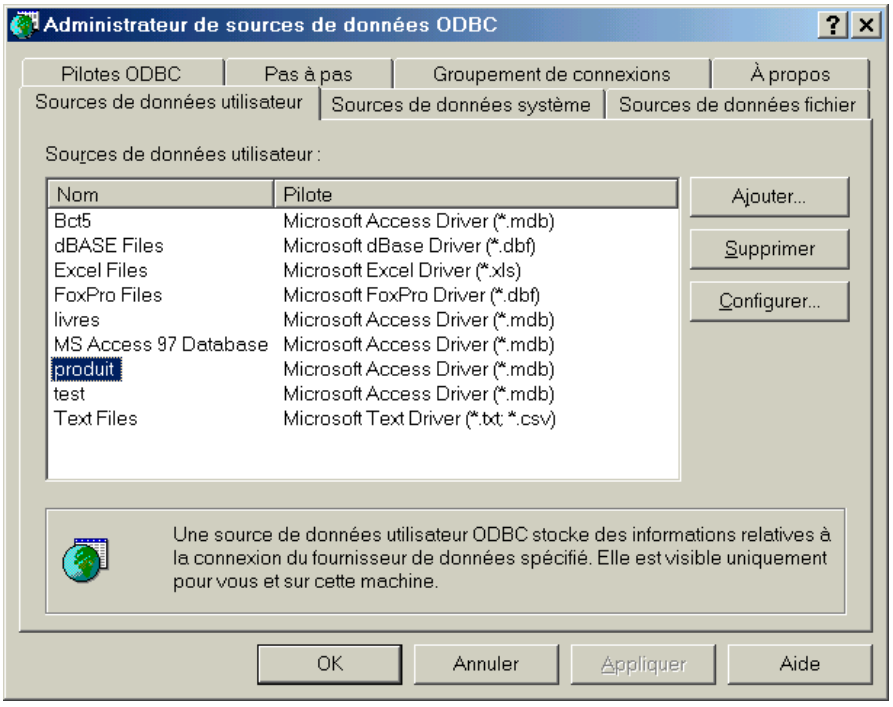
```
void setAutoCommit(boolean autoCommit) throws SQLException
void commit() throws SQLException
void rollback()throws SQLException
```

Il faut commencer par appeler la méthode setAutoCommit et la positionner à faux.
L'appel de commit() valide la transaction depuis la dernière validation.
L'appel de rollBack() annule la transaction depuis la dernière validation.

13.9 Création d'une source de données dans ODBC

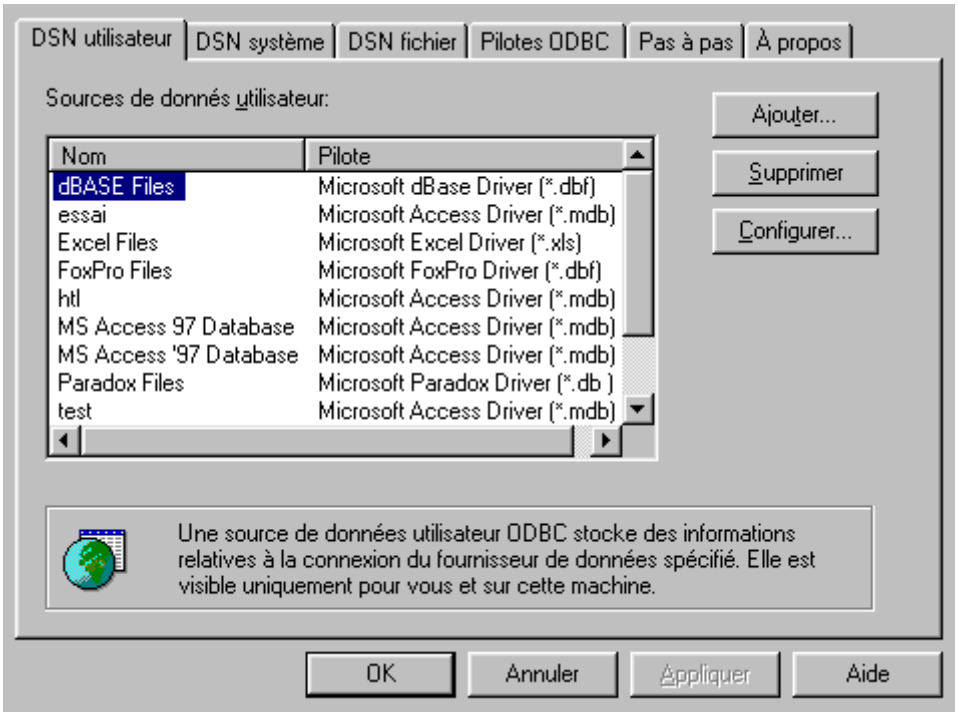
Pour créer une source de données dans ODBC, lancer odbc32





13.10 Création d'une source de données dans ODBC

Pour créer une source de données dans ODBC, lancer odbc32



Cliquer sur l'onglet DSN système de l'administrateur de base de données ODBC et ajouter une base de données: exemple TestJava.mdb

13.11 Annexes

Package java.sql

Provides the JDBC package.

Description

Interface Summary	
Array	JDBC 2.0 The mapping in the Java programming language for the SQL type `ARRAY`.
Blob	JDBC 2.0 The representation (mapping) in the Java™ programming language of an SQL `BLOB`.
CallableStatement	The interface used to execute SQL stored procedures.
Clob	JDBC 2.0 The mapping in the Java™ programming language for the SQL `CLOB` type.
Connection	A connection (session) with a specific database.
DatabaseMetaData	Comprehensive information about the database as a whole.
Driver	The interface that every driver class must implement.
PreparedStatement	An object that represents a precompiled SQL statement.
Ref	JDBC 2.0 A reference to an SQL structured type value in the database.
ResultSet	A ResultSet provides access to a table of data.
ResultSetMetaData	An object that can be used to find out about the types and properties of the columns in a ResultSet.
SQLData	JDBC 2.0 The interface used for the custom mapping of SQL user-defined types.
SQLInput	JDBC 2.0 A input stream that contains a stream of values representing an instance of an SQL structured or distinct type.
SQLOutput	JDBC 2.0 The output stream for writing the attributes of a user-defined type back to the database.
Statement	The object used for executing a static SQL statement and obtaining the results produced by it.
Struct	JDBC 2.0 The standard mapping for an SQL structured type.

Class Summary	
Date	A thin wrapper around a millisecond value that allows JDBC to identify this as a SQL DATE.
DriverManager	The basic service for managing a set of JDBC drivers.
DriverPropertyInfo	Driver properties for making a connection.
Time	A thin wrapper around `java.util.Date` that allows JDBC to identify this as a SQL TIME value.
Timestamp	This class is a thin wrapper around java.util.Date that allows JDBC to identify this as a SQL TIMESTAMP value.
Types	The class that defines constants that are used to identify generic SQL types, called JDBC types.

Exception Summary	
BatchUpdateException	JDBC 2.0 An exception thrown when an error occurs during a batch update operation.
DataTruncation	An exception that reports a DataTruncation warning (on reads) or throws a DataTruncation exception (on writes) when JDBC unexpectedly truncates a data value.
SQLException	An exception that provides information on a database access error.
SQLWarning	An exception that provides information on database access warnings.

14 Programmation Réseau : sockets et RMI

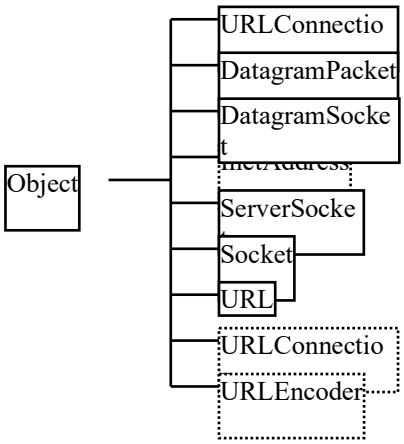
L'un des aspect qui rendent java intéressant concerne son potentiel des applications réseaux. Deux package sont indispensables à connaître pour les accès réseau : java.net et java.rmi. les classes du package net sont spécialisées dans la communication et l'utilisation des ressources de réseau. Les classes du packege rmi proposent des méthodes de haut niveau pour les accès à distance.

Les classes du java.net sont de deux types :

- les classes permettant la manipulation des sockets. Les sockets est une interface de communication réseau de bas niveau. Java fournit plusieurs types de sockets permettant de gérer trois types de protocoles : les protocoles orientés connexion (classe Socket de base java), les protocoles sans connexion (classe DatagramSocket) et les protocoles multicastSocket (une variante de DatagramSocket permettant l'envoi de données n mode multicast).
  Les sockets acceptent plusieurs protocoles sont acceptés, le plus connu et le plus utilisé est le protocole IP. Les applications nécessitent un protocole au dessus des sockets pour interpréter les données. La classe Socket utilise TCP. La classe DatagramSocket utilise le protocole UDP.
- les classes permettant de travailler avec les URL

les RMI généralise la serialisation des objets en java . il permet de travailler de façon transparente avec des objets situés sur des machines distantes, comme sils sont locaux. Plusieurs programmes peuvent travailler sur les mêmes objets à distance.

14.1 Le package java.net



14.1.1 La classe java.net.Socket

Cette classe implémente une socket TCP coté client.

```
try
{  int port = 80;
    Socket s = new Socket("www.lipn.fr", port);
    PrintStream ps = new
    rintStream(s.getOutputStream());
    ps.println("GET /index.html");
    DataInputStream dis = new
    DataInputStream(s.getInputStream());
    String line;
    while((line = dis.readLine()) != null)
       System.out.println(line);
} catch(UnknownHostException e)
{ System.out.println("host introuvable");
} catch (IOException e)
{ System.out.print("probleme de connexion");
}
```

14.2 La classe java.net.ServerSocket

Cette classe implémente une socket TCP coté serveur.

```
int port_d_ecoute = 1234;
ServerSocket serveur = new ServerSocket(port_d_ecoute);
while(true)
{    Socket socket_de_travail = serveur.accept();
     new ClasseQuiFaitLeTraitement(socket_travail);
}
```

14.3 La classe java.net.DatagramSocket

Cette classe implémente une socket UDP

```
// Client
Byte[] data = "un message".getBytes();
InetAddress addr = InetAddress.getByName("falconet.inria.fr");
DatagramPacket packet = new DatagramPacket(data, data.length, addr, 1234);
DatagramSocket ds = new DatagramSocket();
ds.send(packet);
ds.close();
```

```
// Serveur
DatagramSocket ds = new DatagramSocket(1234);
while(true)
{        DatagramPacket packet = new DatagramPacket(new byte[1024], 1024);
         s.receive(packet);
         System.out.println("Message: " + packet.getData());
}
```

14.4 java.net.MulticastSocket

Cette classe implémente une socket multicast (UDP)

```
// Client
Byte[] data = "un message".getBytes();
InetAddress addr = InetAddress.getByName("falconet.inria.fr");
DatagramPacket packet = new DatagramPacket(data, data.length, addr, 1234);
MulticastSocket s = new MulticastSocket();
s.send(packet,(byte)1);
s.close();
```

```
// Serveur
MulticastSocket s = new MulticastSocket(1234);
System.out.println("I listen on port " + s.getLocalPort());
s.joinGroup(InetAddress.getByName("falconet.inria.fr"));
DatagramPacket packet = new DatagramPacket(new byte[1024], 1024);
s.receive(packet);
System.out.println("from: " + packet.getAddress());
System.out.println("Message: " + packet.getData());
s.leaveGroup(InetAddress.getByName("falconet.inria.fr"));
s.close();
```

14.5 java.net.URL

```
URL url = new URL("http://falconet.inria.fr/index.html");
DataInputStream dis = new DataInputStream(url.openStream());
String line;
while ((line = dis.readLine()) != null)
        System.out.println(line);
```

Quelques programmes

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

public class Client
{  public static void main( String argv[] )
   {  try
      { Socket server = new Socket( "www.lipn.fr",40) );
        ObjectOutputStream out = new ObjectOutputStream(server.getOutputStream());
        ObjectInputStream in = new ObjectInputStream( server.getInputStream(  ) );
        out.writeObject( new DateRequest(  ) );
        out.flush(  );
        System.out.println(in.readObject(  ) );
        out.writeObject( new MyCalculation( 2 ) );
        out.flush(  );
        System.out.println( in.readObject(  ) );
        server.close(  );
      } catch ( IOException e )
      {  System.out.println( "Erreur d'E/S " + e ); // Erreur d'E/S
      } catch ( ClassNotFoundException e2 )
      { System.out.println( e2 ); // type d'objet de réponse inconnu
      }
   }
}
```

```
import java.net.*;
import java.io.*;
public class Server
{  public static void main( String argv[] ) throws IOException
   {  ServerSocket ss = new ServerSocket( Integer.parseInt(argv[0]));
      while (true) new ServerConnection( ss.accept() ).start(  );
   }
} // fin de la classe Server
class ServerConnection extends Thread
{ Socket client;
  ServerConnection ( Socket client ) throws SocketException
  { this.client = client;
    setPriority( NORM_PRIORITY - 1 );
  }
  public void run()
  { try
    { ObjectInputStream in =  new ObjectInputStream( client.getInputStream());
      ObjectOutputStream out= new ObjectOutputStream( client.getOutputStream());
      while ( true )
      { out.writeObject( processRequest( in.readObject(  ) ) );
        out.flush(  );
      }
    } catch ( EOFException e3 ) // EOF Normal
    { try
      {  client.close();
      } catch ( IOException e ) { }
    } catch ( IOException e )
    { System.out.println( "Erreur d'E/S " + e ); // Erreur d'E/S
    } catch ( ClassNotFoundException e2 )
    { System.out.println( e2 ); // type d'objet de requête inconnu
    }
  }
  private Object processRequest( Object request )
  {  if ( request instanceof DateRequest )  return new java.util.Date();
     else if (request instanceof WorkRequest)
               return ((WorkRequest)request).execute(  );
    else  return null;
  }
}
```