APPRENTISSAGE DU LANGAGE JAVA

Serge Tahé - ISTIA - Université d'Angers
Septembre 98 - Révision juin 2002

Introduction

Ce document est un support de cours : ce n'est pas un cours complet. Des approfondissements nécessitent l'aide d'un enseignant.

L'étudiant y trouvera cependant une grande quantité d'informations lui permettant la plupart du temps de travailler seul. Ce document comporte probablement des erreurs : toute suggestion constructive est la bienvenue à l'adresse *serge.tahe@istia.univ-angers.fr*.

Il existe d'excellents livres sur Java. Parmi ceux-ci :

1. Programmer en Java de Claude Delannoy aux éditions Eyrolles
2. Java client-serveur de Cédric Nicolas, Christophe Avare, Frédéric Najman chez Eyrolles.

Le premier livre est un excellent ouvrage d'introduction pédagogique au langage Java. Une fois acquis son contenu, on pourra passer au second ouvrage qui présente des aspects plus avancés de Java (Java Beans, JDBC, Corba/Rmi). Il présente une vue industrielle de Java intéressante. Pour approfondir Java dans différents domaines, on pourra se référer à la collection "Java series" chez O'Reilly. Pour une utilisation professionnelle de Java au sein d'une plate-forme J2EE on pourra lire :

3. Programmation j2EE aux éditions Wrox et distribué par Eyrolles.

Septembre 98, juin 2002

Serge Tahé

1. Les bases du langage Java

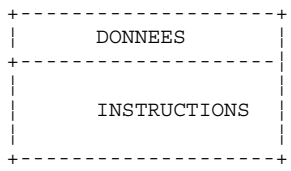
1.1 Introduction

Nous traitons Java d'abord comme un langage de programmation classique. Nous aborderons les objets ultérieurement.

Dans un programme on trouve deux choses

- des données
- les instructions qui les manipulent

On s'efforce généralement de séparer les données des instructions :



1.2 Les données de Java

Java utilise les types de données suivants:

- les nombres entiers
- les nombres réels
- les caractères et chaînes de caractères
- les booléens
- les objets

1.2.1 Les types de données prédéfinis

Type	Codage	Domaine
char	2 octets	caractère Unicode
int	4 octets	$[-2^{31}, 2^{31}-1]$
long	8 octets	$[-2^{63}, 2^{63}-1]$
byte	1 octet	$[-2^{7}, 2^{7}-1]$
short	2 octets	$[-2^{15}, 2^{15}-1]$
float	4 octets	$[3.4\ 10^{-38}, 3.4\ 10^{+38}]$ en valeur absolue
double	8 octets	$[1.7\ 10^{-308}, 1.7\ 10^{+308}]$ en valeur absolue
boolean	1 bit	true, false
String	référence d'objet	chaîne de caractères
Date	référence d'objet	date
Character	référence d'objet	char
Integer	référence d'objet	int
Long	référence d'objet	long
Byte	référence d'objet	byte
Float	référence d'objet	float
Double	référence d'objet	double
Boolean	référence d'objet	boolean

1.2.2 Notation des données littérales

entier	145, -7, 0xFF (hexadécimal)
réel double	134.789, -45E-18 (-45 10^{-18})
réel float	134.789**F**, -45E-18**F** (-45 10^{-18})
caractère	'A', 'b'
chaîne de caractères	"aujourd'hui"
booléen	true, false
date	new Date(13,10,1954) (jour, mois, an)

1.2.3 Déclaration des données

1.2.3.1 Rôle des déclarations

Un programme manipule des données caractérisées par un nom et un type. Ces données sont stockées en mémoire. Au moment de la traduction du programme, le compilateur affecte à chaque donnée un emplacement en mémoire caractérisé par une adresse et une taille. Il le fait en s'aidant des déclarations faites par le programmeur.
Par ailleurs celles-ci permettent au compilateur de détecter des erreurs de programmation. Ainsi l'opération

```
x=x*2;
```

sera déclarée erronée si x est une chaîne de caractères par exemple.

1.2.3.2 Déclaration des constantes

La syntaxe de déclaration d'une constante est la suivante :

```
final type nom=valeur;        //définit constante nom=valeur
```

ex : *final float PI=3.141592F*;

Remarque

Pourquoi déclarer des constantes ?

1. La lecture du programme sera plus aisée si l'on a donné à la constante un nom significatif :

   ex : *final float taux_tva=0.186F*;

2. La modification du programme sera plus aisée si la "constante" vient à changer. Ainsi dans le cas précédent, si le taux de tva passe à 33%, la seule modification à faire sera de modifier l'instruction définissant sa valeur :

   final float taux_tva=0.33F;

Si l'on avait utilisé 0.186 explicitement dans le programme, ce serait alors de nombreuses instructions qu'il faudrait modifier.

1.2.3.3 Déclaration des variables

Une variable est identifiée par un nom et se rapporte à un type de données. Le nom d'une variable Java a n caractères, le premier alphabétique, les autres alphabétiques ou numériques. Java fait la différence entre majuscules et minuscules. Ainsi les variables **FIN** et **fin** sont différentes.

Les variables peuvent être initialisées lors de leur déclaration. La syntaxe de déclaration d'une ou plusieurs variables est :

```
identificateur_de_type variable1,variable2,...,variablen;
```

où *identificateur_de_type* est un type prédéfini ou bien un type objet défini par le programmeur.

1.2.4 Les conversions entre nombres et chaînes de caractères

nombre -> chaîne	*"" + nombre*
chaine -> int	*Integer.parseInt(chaine)*
chaîne -> long	*Long.parseLong(chaine)*
chaîne -> double	*Double.valueOf(chaine).doubleValue()*
chaîne -> float	*Float.valueOf(chaine).floatValue()*

Voici un programme présentant les principales techniques de conversion entre nombres et chaînes de caractères. La conversion d'une chaîne vers un nombre peut échouer si la chaîne ne représente pas un nombre valide. Il y a alors génération d'une erreur fatale appelée **exception** en Java. Cette erreur peut être gérée par la clause *try/catch* suivante :

```
try{
  appel de la fonction susceptible de générer l'exception
} catch (Exception e){
  traiter l'exception e
}
instruction suivante
```

Si la fonction ne génère pas d'exception, on passe alors à **instruction suivante**, sinon on passe dans le corps de la clause *catch* puis à **instruction suivante**. Nous reviendrons ultérieurement sur la gestion des exceptions.

```
import java.io.*;

public class conv1{

  public static void main(String arg[]){
    String S;
    final int i=10;
    final long l=100000;
    final float f=(float)45.78;
    double d=-14.98;

    // nombre --> chaîne
    S=""+i;
    affiche(S);
    S=""+l;
    affiche(S);
    S=""+f;
    affiche(S);
    S=""+d;
    affiche(S);

    //boolean --> chaîne
    final boolean b=false;
    S=""+new Boolean(b);
    affiche(S);

    // chaîne --> int
    int i1;
    i1=Integer.parseInt("10");
    affiche(""+i1);
    try{
      i1=Integer.parseInt("10.67");
      affiche(""+i1);
    } catch (Exception e){
      affiche("Erreur "+e);
    }

    // chaîne --> long
    long l1;
    l1=Long.parseLong("100");
    affiche(""+l1);
    try{
      l1=Long.parseLong("10.675");
      affiche(""+l1);
    } catch (Exception e){
      affiche("Erreur "+e);
    }

    // chaîne --> double
    double d1;
    d1=Double.valueOf("100.87").doubleValue();
    affiche(""+d1);
    try{
      d1=Double.valueOf("abcd").doubleValue();
      affiche(""+d1);
    } catch (Exception e){
      affiche("Erreur "+e);
    }

    // chaîne --> float
    float f1;
    f1=Float.valueOf("100.87").floatValue();
    affiche(""+f1);
    try{
      d1=Float.valueOf("abcd").floatValue();
      affiche(""+f1);
```

```
    } catch (Exception e){
      affiche("Erreur "+e);
    }

}// fin main

  public static void affiche(String S){
    System.out.println("S="+S);
  }
}// fin classe
```

Les résultats obtenus sont les suivants :

```
S=10
S=100000
S=45.78
S=-14.98
S=false
S=10
S=Erreur java.lang.NumberFormatException: 10.67
S=100
S=Erreur java.lang.NumberFormatException: 10.675
S=100.87
S=Erreur java.lang.NumberFormatException: abcd
S=100.87
S=Erreur java.lang.NumberFormatException: abcd
```

1.2.5 Les tableaux de données

Un tableau Java est un objet permettant de rassembler sous un même identificateur des données de même type. Sa déclaration est la suivante :

Type Tableau[]=new Type[n] ou *Type[] Tableau=new Type[n]*

Les deux syntaxes sont légales. **n** est le nombre de données que peut contenir le tableau. La syntaxe *Tableau[i]* désigne la donnée n° *i* où i appartient à l'intervalle *[0,n-1]*. Toute référence à la donnée *Tableau[i]* où *i* n'appartient pas à l'intervalle *[0,n-1]* provoquera une exception.

Un tableau à deux dimensions pourra être déclaré comme suit :

Type Tableau[][]=new Type[n][p] ou *Type[][] Tableau=new Type[n][p]*

La syntaxe *Tableau[i]* désigne la donnée n° *i* de *Tableau* où i appartient à l'intervalle *[0,n-1]*. *Tableau[i]* est lui-même un tableau : *Tableau[i][j]* désigne la donnée n° *j* de *Tableau[i]* où j appartient à l'intervalle *[0,p-1]*. Toute référence à une donnée de *Tableau* avec des index incorrects génère une erreur fatale.

Voici un exemple :

```
public class test1{

  public static void main(String arg[]){
    float[][] taux=new float[2][2];
    taux[1][0]=0.24F;
    taux[1][1]=0.33F;
    System.out.println(taux[1].length);
    System.out.println(taux[1][1]);
  }
}
```

et les résultats de l'exécution :

```
2
0.33
```

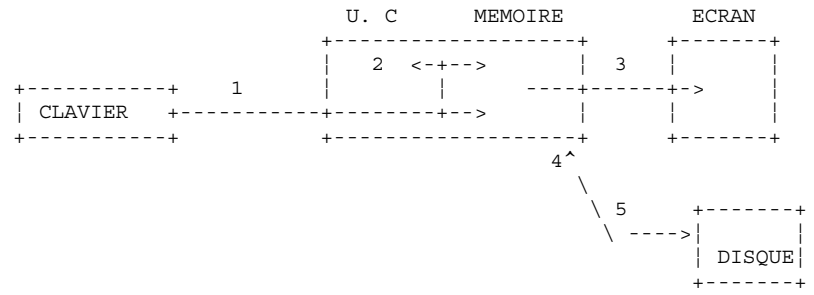
Un tableau est un objet possédant l'attribut *length* : c'est la taille du tableau.

1.3 Les instructions élémentaires de Java

On distingue

1 les instructions élémentaires exécutées par l'ordinateur.
2 les instructions de contrôle du déroulement du programme.

Les instructions élémentaires apparaissent clairement lorsqu'on considère la structure d'un micro-ordinateur et de ses périphériques.



1. lecture d'informations provenant du clavier
2. traitement d'informations
3. écriture d'informations à l'écran
4. lecture d'informations provenant d'un fichier disque
5. écriture d'informations dans un fichier disque

1.3.1 Ecriture sur écran

La syntaxe de l'instruction d'écriture sur l'écran est la suivante :

System.out.println(expression) ou *System.err.println(expression)*

où *expression* est tout type de donnée qui puisse être converti en chaîne de caractères pour être affiché à l'écran. Dans l'exemple précédent, nous avons vu deux instructions d'écriture :

```
System.out.println(taux[1].length);
System.out.println(taux[1][1]);
```

System.out écrit dans un fichier texte qui est par défaut l'écran. Il en est de même pour **System.err**. Ces fichiers portent un numéro (ou *descripteur*) respectivement 1 et 2. Le flux d'entrée du clavier (**System.in**) est également considéré comme un fichier texte, de descripteur 0. Dos comme Unix supportent le tubage (**pipe**) de commandes :

commande1 | *commande2*

Tout ce que *commande1* écrit avec *System.out* est tubé (redirigé) vers l'entrée *System.in* de *commande2*. Dit autrement, *commande2* lit avec *System.in*, les données produites par *commande2* avec *System.out* qui ne sont donc plus affichées à l'écran. Ce système est très utilisé sous Unix. Dans ce tubage, le flux *System.err* n'est lui pas redirigé : il écrit sur l'écran. C'est pourquoi il est utilisé pour écrire les messages d'erreurs (d'où son nom *err*) : on est assuré que lors d'un tubage de commandes, les messages d'erreur continueront à s'afficher à l'écran. On prendra donc l'habitude d'écrire les messages d'erreur à l'écran avec le flux *System.err* plutôt qu'avec le flux *System.out*.

1.3.2 Lecture de données tapées au clavier

Le flux de données provenant du clavier est désigné par l'objet *System.in* de type *InputStream*. Ce type d'objets permet de lire des données caractère par caractère. C'est au programmeur de retrouver ensuite dans ce flux de caractères les informations qui l'intéressent. Le type *InputStream* ne permet pas de lire d'un seul coup une ligne de texte. Le type *BufferedReader* le permet avec la méthode *readLine*.

Afin de pouvoir lire des lignes de texte tapées au clavier, on crée à partir du flux d'entrée *System.in* de type *InputStream*, un autre flux d'entrée de type *BufferedReader* cette fois :

```
BufferedReader IN=new BufferedReader(new InputStreamReader(System.in));
```

Nous n'expliquerons pas ici les détails de cette instruction qui fait intervenir la notion de constructions d'objets. Nous l'utiliserons telle-quelle.

La construction d'un flux peut échouer : une erreur fatale, appelée **exception** en Java, est alors générée. A chaque fois qu'une

méthode est susceptible de générer une exception, le compilateur Java exige qu'elle soit gérée par le programmeur. Aussi, pour créer le flux d'entrée précédent, il faudra en réalité écrire :

```
BufferedReader IN=null;
try{
IN=new BufferedReader(new InputStreamReader(System.in));
} catch (Exception e){
    System.err.println("Erreur " +e);
    System.exit(1);
}
```

De nouveau, on ne cherchera pas à expliquer ici la gestion des exceptions. Une fois le flux *IN* précédent construit, on peut lire une ligne de texte par l'instruction :

```
  String ligne;
  ligne=IN.readLine();
```

La ligne tapée au clavier est rangée dans la variable *ligne* et peut ensuite être exploitée par le programme.

1.3.3 Exemple d'entrées-sorties

Voici un programme d'illustration des opérations d'entrées-sorties clavier/écran :

```
import java.io.*;  // nécessaire pour l'utilisation de flux d'E/S

public class io1{


  public static void main (String[] arg){

    // écriture sur le flux System.out
    Object obj=new Object();
    System.out.println(""+obj);
    System.out.println(obj.getClass().getName());

    // écriture sur le flux System.err
    int i=10;
    System.err.println("i="+i);

    // lecture d'une ligne saisie au clavier
    String ligne;
    BufferedReader IN=null;
    try{
      IN=new BufferedReader(new InputStreamReader(System.in));
    } catch (Exception e){
      affiche(e);
      System.exit(1);
    }
    System.out.print("Tapez une ligne : ");
    try{
      ligne=IN.readLine();
      System.out.println("ligne="+ligne);
    } catch (Exception e){
      affiche(e);
      System.exit(2);
    }
  }//fin main

  public static void affiche(Exception e){
    System.err.println("Erreur : "+e);
  }

}//fin classe
```

et les résultats de l'exécution :

```
C:\Serge\java\bases\iostream>java io1
java.lang.Object@1ee78b
java.lang.Object
i=10
Tapez une ligne : je suis là
ligne=je suis là
```

Les instructions

```
Object obj=new Object();
System.out.println(""+obj);
System.out.println(obj.getClass().getName());
```

ont pour but de montrer que n'importe quel objet peut faire l'objet d'un affichage. Nous ne chercherons pas ici à expliquer la signification de ce qui est affiché. Nous avons également l'affichage de la valeur d'un objet dans le bloc :

```
try{
      IN=new BufferedReader(new InputStreamReader(System.in));
    } catch (Exception e){
      affiche(e);
      System.exit(1);
    }
```

La variable *e* est un objet de type *Exception* qu'on affiche ici avec l'appel *affiche(e)*. Nous avions rencontré, sans en parler, cet affichage de la valeur d'une exception dans le programme de conversion vu plus haut.

1.3.4 Affectation de la valeur d'une expression à une variable

On s'intéresse ici à l'opération *variable=expression;*

L'expression peut être de type : arithmétique, relationnelle, booléenne, caractères

1.3.4.1 Interprétation de l'opération d'affectation

L'opération *variable=expression;* est elle-même une **expression** dont l'évaluation se déroule de la façon suivante :

- La partie droite de l'affectation est évaluée : le résultat est une valeur V.
- la valeur V est affectée à la variable
- la valeur V est aussi la valeur de l'affectation vue cette fois en tant qu'expression.

C'est ainsi que l'opération *V1=V2=expression* est légale. A cause de la priorité, c'est l'opérateur = le plus à droite qui va être évalué. On a donc *V1=(V2=expression)*. L'expression *V2=expression* est évaluée et a pour valeur *V*. L'évaluation de cette expression a provoqué l'affectation de V à V2. L'opérateur = suivant est alors évalué sous la forme *V1=V*. La valeur de cette expression est encore V. Son évaluation provoque l'affectation de V à V1. Ainsi donc, l'opération *V1=V2=expression* est une expression dont l'évaluation

1 provoque l'affectation de la valeur de *expression* aux variables V1 et V2
2 rend comme résultat la valeur de *expression*.

On peut généraliser à une expresion du type : *V1=V2=....=Vn=expression*

1.3.4.2 Expression arithmétique

Les opérateurs des expressions arithmétiques sont les suivants :

+	addition
-	soustraction
*	multiplication
/	division : le résultat est le quotient exact si l'un au moins des opérandes est réel. Si les deux opérandes sont entiers le résultat est **le quotient entier**. Ainsi 5/2 -> 2 et 5.0/2 ->2.5.
%	division : le résultat est le reste quelque soit la nature des opérandes, le quotient étant lui entier. C'est donc l'opération **modulo**.

Il existe diverses fonctions mathématiques :

double sqrt(double x)	racine carrée
double cos(double x)	Cosinus
double sin(double x)	Sinus
double tan(double x)	Tangente
double pow(double x,double y)	x à la puissance y (x>0)
double exp(double x)	Exponentielle
double log(double x)	Logarithme népérien
double abs(double x)	valeur absolue

etc...

Toutes ces fonctions sont définies dans une classe Java appelée **Math**. Lorsqu'on les utilise, il faut les préfixer avec le nom de la classe où elles sont définies. Ainsi on écrira :

```
double x, y=4;
x=Math.sqrt(y);
```

La définition de la classe *Math* est la suivante :

```
public  final  class  java.lang.Math
    extends  java.lang.Object  (I-§1.12)
{
        // Fields
    public final static double E; §1.10.1
    public final static double PI;  §1.10.2

        // Methods
    public static double abs(double  a); §1.10.3
    public static float abs(float  a); §1.10.4
    public static int abs(int  a);  §1.10.5
    public static long abs(long  a);  §1.10.6
    public static double acos(double  a);  §1.10.7
    public static double asin(double  a);  §1.10.8
    public static double atan(double  a);  §1.10.9
    public static double atan2(double  a, double  b);  §1.10.10
    public static double ceil(double  a);  §1.10.11
    public static double cos(double  a); §1.10.12
    public static double exp(double  a); §1.10.13
    public static double floor(double  a); §1.10.14
    public static double                          §1.10.15
        IEEEremainder(double  f1, double  f2);
    public static double log(double  a); §1.10.16
    public static double max(double  a, double  b);§1.10.17
    public static float max(float  a, float  b);  §1.10.18
    public static int max(int  a, int  b); §1.10.19
    public static long max(long  a, long  b); §1.10.20
    public static double min(double  a, double  b);  §1.10.21
    public static float min(float  a, float  b);  §1.10.22
    public static int min(int  a, int  b); §1.10.23
    public static long min(long  a, long  b); §1.10.24
    public static double pow(double  a, double  b);§1.10.25
    public static double random();  §1.10.26
    public static double rint(double  a);  §1.10.27
    public static long round(double  a); §1.10.28
    public static int round(float  a); §1.10.29
    public static double sin(double  a); §1.10.30
    public static double sqrt(double  a);  §1.10.31
    public static double tan(double  a); §1.10.32
}
```

1.3.4.3 Priorités dans l'évaluation des expressions arithmétiques

La priorité des opérateurs lors de l'évaluation d'une expression arithmétique est la suivante (du plus prioritaire au moins prioritaire) :

*[fonctions], [( )],[*, /, %], [+, -]*

Les opérateurs d'un même bloc [ ] ont même priorité.

1.3.4.4 Expressions relationnelles

Les opérateurs sont les suivants : **<, <=, ==, !=, >, >=**

ordre de priorité

>, >=, <, <=
==, !=

Le résultat d'une expression relationnelle est le booléen *false* si expression est fausse*true* sinon.

Exemple :

```
boolean fin;
int x;
fin=x>4;
```

Comparaison de deux caractères

Soient deux caractères C1 et C2. Il est possible de les comparer avec les opérateurs
<, <=, ==, !=, >, >=
Ce sont alors leurs codes ASCII, qui sont des nombres, qui sont alors comparés. On rappelle que selon l'ordre ASCII on a les relations suivantes :

espace < .. < '0' < '1' < .. < '9' < .. < 'A' < 'B' < .. < 'Z' < .. < 'a' < 'b' < .. <'z'

Comparaison de deux chaînes de caractères

Elles sont comparées caractère par caractère. La première inégalité rencontrée entre deux caractères induit une inégalité de même sens sur les chaînes.

Exemples :

Soit à comparer les chaînes "Chat" et "Chien"

```
"Chat" "Chien"
-----------------------
'C' = 'C'
'h' = 'h'
'a' < 'i'
```

Cette dernière inégalité permet de dire que "Chat" < "Chien".

Soit à comparer les chaînes "Chat" et "Chaton". Il y a égalité tout le temps jusqu'à épuisement de la chaîne "Chat". Dans ce cas, la chaîne épuisée est déclarée la plus "petite". On a donc la relation
"Chat" < "Chaton".

Fonctions de comparaisons de deux chaînes

On ne peut utiliser ici les opérateurs relationnels <, <=, ==, !=, >, >= . Il faut utiliser des méthodes de la classe *String* :

```
String chaine1, chaine2;
chaine1=…;
chaine2=…;
int i=chaine1.compareTo(chaine2);
boolean egal=chaine1.equals(chaine2)
```

Ci-dessus, la variable *i* aura la valeur :

0 si les deux chaînes sont égales
1 si chaîne n°1 > chaîne n°2
-1 si chaîne n°1 < chaîne n°2

La variable *egal* aura la valeur *true* si les deux chaînes sont égales.

1.3.4.5 Expressions booléennes

Les opérateurs sont *&& (and)* | | *(or)* et *!* (not). Le résultat d'une expression booléenne est un booléen.

ordre de priorité ! , &&, | |

exemple :

```
int fin;
int x;
fin= x>2 && x<4;
```

Les opérateurs relationnels **ont priorité** sur les opérateurs && et | |.

1.3.4.6 Traitement de bits

Les opérateurs

Soient i et j deux entiers.

i<<n	décale i de n bits sur la gauche. Les bits entrants sont des zéros.
i>>n	décale i de n bits sur la droite. Si i est un entier signé (signed char, int, long) le bit de signe est préservé.
i & j	fait le ET logique de i et j bit à bit.
i | j	fait le OU logique de i et j bit à bit.
~i	complémente i à 1
i^j	fait le OU EXCLUSIF de i et j

Soit

```
int i=0x123F, k=0xF123;
  unsigned j=0xF123;
```

opération	valeur
i<<4	0x23F0
i>>4	0x0123 le bit de signe est préservé.
k>>4	0xFF12 le bit de signe est préservé.
i&j	0x1023
i|j	0xF33F
~i	0xEDC0

1.3.4.7 Combinaison d'opérateurs

a=a+b peut s'écrire a+=b
a=a-b peut s'écrire a-=b

Il en est de même avec les opérateurs */, %,*,<<, >>, &, |,* ^

Ainsi *a=a+2;* peut s'écrire *a+=2;*

1.3.4.8 Opérateurs d'incrémentation et de décrémentation

La notation *variable*++ signifie *variable=variable+1* ou encore *variable+=1*
La notation *variable*-- signifie *variable=variable-1* ou encore *variable-=1*.

1.3.4.9 L'opérateur ?

L'expression *expr_cond ? expr1:expr2* est évaluée de la façon suivante :

1 l'expression *expr_cond* est évaluée. C'est une expression conditionnelle à valeur vrai ou faux
2 Si elle est vraie, la valeur de l'expression est celle de *expr1*. *expr2* n'est pas évaluée.
3 Si elle est fausse, c'est l'inverse qui se produit : la valeur de l'expression est celle de *expr2*. *expr1* n'est pas évaluée.

Exemple

i=(j>4 ? j+1:j-1);

affectera à la variable i :

j+1 si *j>4*, *j-1* sinon

C'est la même chose que d'écrire *if(j>4) i=j+1; else i=j-1;* mais c'est plus concis.

1.3.4.10 Priorité générale des opérateurs

```
() []  fonction                                    gd
! ~ ++ --                                          dg
new (type) opérateurs cast                         dg
*  /  %                                            gd
+  -                                               gd
<<  >>                                             gd
< <=  > >= instanceof                              gd
==    !=                                           gd
&                                                  gd
^                                                  gd
|                                                  gd
&&                                                 gd
||                                                 gd
?   :                                              dg
= += -= etc. .                                     dg
```

gd indique qu'a priorité égale, c'est la priorité gauche-droite qui est observée. Cela signifie que lorsque dans une expression, l'on a des opérateurs de même priorité, c'est l'opérateur le plus à gauche dans l'expression qui est évalué en premier. **dg** indique une priorité droite-gauche.

1.3.4.11 Les changements de type

Il est possible, dans une expression, de changer momentanément le codage d'une valeur. On appelle cela changer le type d'une donnée ou en anglais **type casting**. La syntaxe du changement du type d'une valeur dans une expression est la suivante *(type) valeur*. La valeur prend alors le type indiqué. Cela entraîne un changement de codage de la valeur.

exemple :

```
int i, j;
float isurj;
isurj= (float)i/j;   // priorité de () sur /
```

Ici il est nécessaire de changer le type de *i* ou *j* en réel sinon la division donnera le quotient entier et non réel.

i est une valeur codée de façon exacte sur 2 octets

(float) i est la même valeur codée de façon approchée en réel sur 4 octets

Il y a donc transcodage de la valeur de *i*. Ce transcodage n'a lieu que le temps d'un calcul, la variable *i* conservant toujours son type *int*.

1.4 Les instructions de contrôle du déroulement du programme

1.4.1 Arrêt

La méthode *exit* définie dans la classe *System* permet d'arrêter l'exécution d'un programme.

syntaxe void exit(int status)
action arrête le processus en cours et rend la valeur status au processus père

exit provoque la fin du processus en cours et rend la main au processus appelant. La valeur de *status* peut être utilisée par celui-ci. Sous DOS, cette variable *status* est rendue à DOS dans la variable système **ERRORLEVEL** dont la valeur peut être testée dans un fichier batch. Sous Unix, c'est la variable **$?** qui récupère la valeur de *status* si l'interpréteur de commandes est le Bourne Shell (/bin/sh).

Exemple :

```
  System.exit(0);
```

pour arrêter le programme avec une valeur d'état à 0.

1.4.2 Structure de choix simple

```
 syntaxe :  if (condition) {actions_condition_vraie;} else {actions_condition_fausse;}
```

notes:

- la condition est entourée de parenthèses.
- chaque action est terminée par point-virgule.
- les accolades ne sont pas terminées par point-virgule.
- les accolades ne sont nécessaires que s'il y a plus d'une action.
- la clause else peut être absente.
- Il n'y a pas de then.

L'équivalent algorithmique de cette structure est la structure *si .. alors … sinon* :

```
si condition
  alors actions_condition_vraie
  sinon actions_condition_fausse
finsi
```

exemple

```
if (x>0)  { nx=nx+1;sx=sx+x;} else dx=dx-x;
```

On peut imbriquer les structures de choix :

```
if(condition1)
if (condition2)
    {......}
    else   //condition2
      {......}
else   //condition1
  {.......}
```

Se pose parfois le problème suivant :

```
public static void main(void){
    int n=5;

   if(n>1)
     if(n>6)
       System.out.println(">6");
     else System.out.println("<=6");
}
```

Dans l'exemple précédent, le *else* se rapporte à quel *if* ? La règle est qu'un *else* se rapporte toujours au *if* le plus proche : *if(n>6)* dans l'exemple. Considérons un autre exemple :

```
public static void main(void)
{  int n=0;

   if(n>1)
     if(n>6)   System.out.println(">6");
     else;    // else du if(n>6) : rien à faire
   else System.out.println("<=1");  // else du if(n>1)
}
```

Ici nous voulions mettre un *else* au *if(n>1)* et pas de *else* au *if(n>6)*. A cause de la remarque précédente, nous sommes obligés de mettre un else au *if(n>6)*, dans lequel il n'y a aucune instruction.

1.4.3 Structure de cas

La syntaxe est la suivante :

```
switch(expression) {
  case v1:
     actions1;
     break;
  case v2:
     actions2;
     break;
```

```
    . .. .. .. .. ..
  default:  actions_sinon;
}
```

notes

- La valeur de l'expression de contrôle, ne peut être qu'un entier ou un caractère.
- l'expression de contrôle est entourée de parenthèses.
- la clause *default* peut être absente.
- les valeurs v_i sont des valeurs possibles de l'expression. Si l'expression a pour valeur v_i, les actions derrière la clause **case** v_i sont exécutées.
- l'instruction *break* fait sortir de la structure de cas. Si elle est absente à la fin du bloc d'instructions de la valeur v_i, l'exécution se poursuit alors avec les instructions de la valeur v_{i+1}.

exemple

En algorithmique

```
selon la valeur de choix
  cas 0
    arrêt
  cas 1
    exécuter module M1
  cas 2
    exécuter module M2
  sinon
    erreur<--vrai
findescas
```

En Java

```
int choix, erreur;
switch(choix){
  case 0: System.exit(0);
     case 1: M1();break;
     case 2: M2();break;
     default: erreur=1;
}
```

1.4.4 Structure de répétition

1.4.4.1 Nombre de répétitions connu

Syntaxe

```
for (i=id;i<=if;i=i+ip){
  actions;
  }
```

Notes

- les 3 arguments du *for* sont à l'intérieur d'une parenthèse.
- les 3 arguments du *for* sont séparés par des points-virgules.
- chaque action du *for* est terminée par un point-virgule.
- l'accolade n'est nécessaire que s'il y a plus d'une action.
- l'accolade n'est pas suivie de point-virgule.

L'équivalent algorithmique est la structure *pour* :

```
pour i variant de id à if avec un pas de ip
  actions
finpour
```

qu'on peut traduire par une structure *tantque* :

```
i ← id
tantque i<=if
  actions
  i← i+ip
fintantque
```

1.4.4.2 Nombre de répétitions inconnu

Il existe de nombreuses structures en Java pour ce cas.

Structure tantque (while)

```
while(condition){
    actions;
}
```

On boucle tant que la condition est vérifiée. La boucle peut ne jamais être exécutée.

notes:

- la condition est entourée de parenthèses.
- chaque action est terminée par point-virgule.
- l'accolade n'est nécessaire que s'il y a plus d'une action.
- l'accolade n'est pas suivie de point-virgule.

La structure algorithmique correspondante est la structure tantque :

```
tantque condition
  actions
fintantque
```

Structure répéter jusqu'à (do while)

La syntaxe est la suivante :

```
do{
  instructions;
}while(condition);
```

On boucle jusqu'à ce que la condition devienne fausse ou tant que la condition est vraie. Ici la boucle est faite au moins une fois.

notes

- la condition est entourée de parenthèses.
- chaque action est terminée par point-virgule.
- l'accolade n'est nécessaire que s'il y a plus d'une action.
- l'accolade n'est pas suivie de point-virgule.

La structure algorithmique correspondante est la structure *répéter … jusqu'à* :

```
répéter
 actions
jusqu'à condition
```

Structure pour générale (for)

La syntaxe est la suivante :

```
for(instructions_départ;condition;instructions_fin_boucle){
  instructions;
}
```

On boucle tant que la condition est vraie (évaluée avant chaque tour de boucle). *Instructions_départ* sont effectuées avant d'entrer dans la boucle pour la première fois. *Instructions_fin_boucle* sont exécutées après chaque tour de boucle.

notes

- les 3 arguments du *for* sont à l'intérieur de parenthèses.
- les 3 arguments du *for* sont séparés par des points-virgules.
- chaque action du for est terminée par un point-virgule.
- l'accolade n'est nécessaire que s'il y a plus d'une action.
- l'accolade n'est pas suivie de point-virgule.
- les différentes instructions dans *instructions_depart* et *instructions_fin_boucle* sont séparées par des virgules.

La structure algorithmique correspondante est la suivante :

```
instructions_départ
tantque condition
  actions
  instructions_fin_boucle
fintantque
```

Exemples

Les programmes suivants calculent tous la somme des n premiers nombres entiers.

```
1  for(i=1, somme=0;i<=n;i=i+1)
     somme=somme+a[i];

2   for (i=1, somme=0;i<=n;somme=somme+a[i], i=i+1);

3 i=1;somme=0;
  while(i<=n)
    { somme+=i; i++; }

4 i=1; somme=0;
  do somme+=i++;
     while (i<=n);
```

Instructions de gestion de boucle

break	fait sortir de la boucle for, while, do ... while.
continue	fait passer à l'itération suivante des boucles for, while, do ... while

1.5 La structure d'un programme Java

Un programme Java n'utilisant pas de classe définie par l'utilisateur ni de fonctions autres que la fonction principale *main* pourra avoir la structure suivante :

```
public class test1{
  public static void main(String arg[]){
    … code du programme
  }// main
}// class
```

La fonction *main*, appelée aussi méthode est exécutée la première lors de l'exécution d'un programme Java. Elle doit avoir obligatoirement la signature précédente :

```
  public static void main(String arg[]){
```

ou

```
  public static void main(String[] arg){
```

Le nom de l'argument *arg* peut être quelconque. C'est un tableau de chaînes de caractères représentant les arguments de la ligne de commande. Nous y reviendrons un peu plus loin.

Si on utilise des fonctions susceptibles de générer des exceptions qu'on ne souhaite pas gérer finement, on pourra encadrer le code du programme par une clause **try/catch** :

```
public class test1{
```

```
  public static void main(String arg[]){
    try{
    … code du programme
    } catch (Exception e){
      // gestion de l'erreur
    }// try
  }// main
}// class
```

Au début du code source et avant la définition de la classe, il est usuel de trouver des instructions d'importation de classes. Par exemple :

```
import java.io.*;
public class test1{
  public static void main(String arg[]){
    … code du programme
  }// main
}// class
```

Prenons un exemple. Soit l'instruction d'écriture suivante :

```
System.out.println("java");
```

qui écrit *java* à l'écran. Il y a dans cette simple instruction beaucoup de choses :

- *System* est une classe dont le nom complet est *java.lang.System*
- *out* est une propriété de cette classe de type *java.io.PrintStream*, une autre classe
- *println* est une méthode de la classe *java.io.PrintStream*.

Nous ne compliquerons pas inutilement cette explication qui vient trop tôt puisqu'elle nécessite la compréhension de la notion de classe pas encore abordée. On peut assimiler une classe à une ressource. Ici, le compilateur aura besoin d'avoir accès aux deux classes *java.lang.System* et *java.io.PrintStream*. Les centaines de classes de Java sont réparties dans des archives aussi appelées des paquetages (package). Les instruction *import* placées en début de programme servent à indiquer au compilateur de quelles classes externes le programme a besoin (celles utilisées mais non définies dans le fichier source qui sera compilé). Ainsi dans notre exemple, notre programme a besoin des classes *java.lang.System* et *java.io.PrintStream*. On le dit avec l'instruction *import*. On pourrait écrire en début de programme :

```
import java.lang.System;
import java.io.PrintStream;
```

Un programme Java utilisant couramment plusieurs dizaines de classes externes, il serait pénible d'écrire toutes les fonction *import* nécessaires. Les classes ont été regroupées dans des paquetages et on peut alors importer le paquetage entier. Ainsi pour importer les paquetages *java.lang* et *java.io*, on écrira :

```
import java.lang.*;
import java.io.*;
```

Le paquetage *java.lang* contient toutes les classes de base de Java et il est importé automatiquement par le compilateur. Aussi finalement n'écrira-t-on que :

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

1.6 La gestion des exceptions

De nombreuses fonctions Java sont susceptibles de générer des exceptions, c'est à dire des erreurs. Nous avons déjà rencontré une telle fonction, la fonction *readLine* :

```
String ligne=null;
try{
      ligne=IN.readLine();
      System.out.println("ligne="+ligne);
    } catch (Exception e){
      affiche(e);
      System.exit(2);
}// try
```

Lorsqu'une fonction est susceptible de générer une exception, le compilateur Java oblige le programmeur à gérer celle-ci dans le but d'obtenir des programmes plus résistants aux erreurs : il faut toujours éviter le "plantage" sauvage d'une application. Ici, la fonction

readLine génère une exception s'il n'y a rien à lire parce que par exemple le flux d'entrée a été fermé. La gestion d'une exception se fait selon le schéma suivant :

```
try{
  appel de la fonction susceptible de générer l'exception
} catch (Exception e){
  traiter l'exception e
}
instruction suivante
```

Si la fonction ne génère pas d'exception, on passe alors à **instruction suivante**, sinon on passe dans le corps de la clause *catch* puis à **instruction suivante**. Notons les points suivants :

- **e** est un objet dérivé du type *Exception*. On peut être plus précis en utilisant des types tels que *IOException, SecurityException, ArithmeticException*, etc… : il existe une vingtaine de types d'exceptions. En écrivant *catch (Exception e)*, on indique qu'on veut gérer toutes les types d'exceptions. Si le code de la clause *try* est susceptible de générer plusieurs types d'exceptions, on peut vouloir être plus précis en gérant l'exception avec plusieurs clauses *catch* :

```
try{
  appel de la fonction susceptible de générer l'exception
} catch (IOException e){
  traiter l'exception e
}
} catch (ArrayIndexOutOfBoundsException e){
  traiter l'exception e
}
} catch (RunTimeException e){
  traiter l'exception e
}
instruction suivante
```

- On peut ajouter aux clauses *try/catch*, une clause **finally** :

```
try{
  appel de la fonction susceptible de générer l'exception
} catch (Exception e){
  traiter l'exception e
}
finally{
  code exécuté après try ou catch
}
instruction suivante
```

Ici, qu'il y ait exception ou pas, le code de la clause *finally* sera toujours exécuté.

- La classe *Exception* a une méthode **getMessage()** qui rend un message détaillant l'erreur qui s'est produite. Ainsi si on veut afficher celui-ci, on écrira :

```
catch (Exception ex){
  System.err.println("L'erreur suivante s'est produite : "+ex.getMessage());
  ...
}//catch
```

- La classe *Exception* a une méthode **toString()** qui rend une chaîne de caractères indiquant le type de l'exception ainsi que la valeur de la propriété *Message*. On pourra ainsi écrire :

```
catch (Exception ex){
  System.err.println ("L'erreur suivante s'est produite : "+ex.toString());
  ...
}//catch
```

  On peut écrire aussi :

```
catch (Exception ex){
  System.err.println ("L'erreur suivante s'est produite : "+ex);
  ...
}//catch
```

  Nous avons ici une opération *string* + *Exception* qui va être automatiquement transformée en *string* + *Exception.toString()* par le compilateur afin de faire la concaténation de deux chaînes de caractères.

L'exemple suivant montre une exception générée par l'utilisation d'un élément de tableau inexistant :

```
// tableaux

// imports
import java.io.*;
```

```
public class tab1{
  public static void main(String[] args){
    // déclaration & initialisation d'un tableau
    int[] tab=new int[] {0,1,2,3};
    int i;
    // affichage tableau avec un for
    for (i=0; i<tab.length; i++)
      System.out.println("tab[" + i + "]=" + tab[i]);
    // génération d'une exception
    try{
      tab[100]=6;
    }catch (Exception e){
      System.err.println("L'erreur suivante s'est produite : " + e);
    }//try-catch
  }//main
}//classe
```

L'exécution du programme donne les résultats suivants :

```
tab[0]=0
tab[1]=1
tab[2]=2
tab[3]=3
L'erreur suivante s'est produite : java.lang.ArrayIndexOutOfBoundsException
```

Voici un autre exemple où on gère l'exception provoquée par l'affectation d'une chaîne de caractères à un nombre lorsque la chaîne ne représente pas un nombre :

```
// imports
import java.io.*;

public class console1{
  public static void main(String[] args){
    // création d'un flux d'entrée
    BufferedReader IN=null;
    try{
      IN=new BufferedReader(new InputStreamReader(System.in));
    }catch(Exception ex){}
    // On demande le nom
    System.out.print("Nom : ");
    // lecture réponse
    String nom=null;
    try{
      nom=IN.readLine();
    }catch(Exception ex){}
    // on demande l'âge
    int age=0;
    boolean ageOK=false;
    while ( ! ageOK){
      // question
      System.out.print("âge : ");
      // lecture-vérification réponse
      try{
        age=Integer.parseInt(IN.readLine());
        ageOK=true;
      }catch(Exception ex) {
        System.err.println("Age incorrect, recommencez...");
      }//try-catch
    }//while
    // affichage final
    System.out.println("Vous vous appelez " + nom + " et vous avez " + age + " ans");
  }//Main
}//classe
```

Quelques résultats d'exécution :

```
dos>java console1
Nom : dupont
âge : 23
Vous vous appelez dupont et vous avez 23 ans
```

```
E:\data\serge\MSNET\c#\bases\1>console1
Nom : dupont
âge : xx
Age incorrect, recommencez...
âge : 12
Vous vous appelez dupont et vous avez 12 ans
```

1.7 Compilation et exécution d'un programme Java

Soit à compiler puis exécuter le programme suivant :

```
// importation de classes
import java.io.*;

// classe test
public class coucou{
  // fonction main
  public static void main(String args[]){
    // affichage écran
    System.out.println("coucou");
  }//main
}//classe
```

Le fichier source contenant la classe *coucou* précédente doit obligatoirement s'appeler *coucou.java* :

```
E:\data\serge\JAVA\ESSAIS\intro1>dir
10/06/2002  08:42                 228 coucou.java
```

La compilation et l'exécution d'un programme Java se fait dans une fenêtre DOS. Les exécutables *javac.exe* (compilateur) et *java.exe* (interpréteur) se trouvent dans le répertoire *bin* du répertoire d'installation du JDK :

```
E:\data\serge\JAVA\classes\paquetages\personne>dir "e:\program files\jdk14\bin\java?.exe"
07/02/2002  12:52              24 649 java.exe
07/02/2002  12:52              28 766 javac.exe
```

Le compilateur *javac.exe* va analyser le fichier source .java et produire un fichier compilé .class. Celui-ci n'est pas immédiatement exécutable par le processeur. Il nécessite un interpréteur Java (*java.exe*) qu'on appelle une machine virtuelle ou JVM (Java Virtual Machine). A partir du code intermédiaire présent dans le fichier .class, la machine virtuelle va générer des instructions spécifiques au processeur de la machine sur laquelle elle s'exécute. Il existe des machines virtuelles Java pour différents types de systèmes d'exploitation (Windows, Unix, Mac OS,...). Un fichier .class pourra être exécuté par n'importe laquelle de ces machines virtuelles donc sur n'importe que système d'exploitation. Cette portabilité inter-systèmes est l'un des atouts majeurs de Java.

Compilons le programme précédent :

```
E:\data\serge\JAVA\ESSAIS\intro1>"e:\program files\jdk14\bin\javac" coucou.java

E:\data\serge\JAVA\ESSAIS\intro1>dir
10/06/2002  08:42                 228 coucou.java
10/06/2002  08:48                 403 coucou.class
```

Exécutons le fichier *.class* produit :

```
E:\data\serge\JAVA\ESSAIS\intro1>"e:\program files\jdk14\bin\java" coucou
coucou
```

On notera que dans la demande d'exécution ci-dessus, on n'a pas précisé le suffixe *.class* du fichier *coucou.class* à exécuter. Il est implicite. Si le répertoire *bin* du JDK est dans le PATH de la machine DOS, on pourra ne pas donner le chemin complet des exécutables *javac.exe* et *java.exe*. On écrira alors simplement

```
javac coucou.java
java coucou
```

1.8 Arguments du programme principal

La fonction principale *main* admet comme paramètres un tableau de chaînes : *String[]*. Ce tableau contient les arguments de la ligne de commande utilisée pour lancer l'application. Ainsi si on lance le programme P avec la commande :

```
    java P arg0 arg1 … argn
```

et si la fonction main est déclarée comme suit :

```
public static void main(String[] arg);
```

on aura arg[0]="arg0", arg[1]="arg1" … Voici un exemple :

```
import java.io.*;

public class param1{
  public static void main(String[] arg){
    int i;
    System.out.println("Nombre d'arguments="+arg.length);
    for (i=0;i<arg.length;i++)
      System.out.println("arg["+i+"]="+arg[i]);
  }
}
```

Les résultats obtenus sont les suivants :

```
dos>java param1 a b c
Nombre d'arguments=3
arg[0]=a
arg[1]=b
arg[2]=c
```

1.9 Passage de paramètres à une fonction

Les exemples précédents n'ont montré que des programmes Java n'ayant qu'une fonction, la fonction principale *main*. L'exemple suivant montre comment utiliser des fonctions et comment se font les échanges d'informations entre fonctions. Les paramètres d'une fonction sont toujours passés par valeur : c'est à dire que la valeur du paramètre effectif est recopiée dans le paramètre formel correspondant.

```
import java.io.*;

public class param2{
  public static void main(String[] arg){
    String S="papa";
    changeString(S);
    System.out.println("Paramètre effectif S="+S);
    int age=20;
    changeInt(age);
    System.out.println("Paramètre effectif age="+age);
  }
  private static void changeString(String S){
    S="maman";
    System.out.println("Paramètre formel S="+S);
  }
  private static void changeInt(int a){
    a=30;
    System.out.println("Paramètre formel a="+a);
  }
}
```

Les résultats obtenus sont les suivants :

```
Paramètre formel S=maman
Paramètre effectif S=papa
Paramètre formel a=30
Paramètre effectif age=20
```

Les valeurs des paramètres effectifs "papa" et 20 ont été recopiées dans les paramètres formels S et a. Ceux-ci ont été ensuite modifiés. Les paramètres effectifs ont été eux inchangés. On notera bien ici le type des paramètres effectifs :

- *S* est une référence d'objet c.a.d. l'adresse d'un objet en mémoire
- *age* est une valeur entière

1.10 L'exemple impots

Nous terminerons ce chapitre par un exemple que nous reprendrons à diverses reprises dans ce document. On se propose d'écrire un programme permettant de calculer l'impôt d'un contribuable. On se place dans le cas simplifié d'un contribuable n'ayant que son seul salaire à déclarer :

- on calcule le nombre de parts du salarié **nbParts=nbEnfants/2 +1** s'il n'est pas marié, **nbEnfants/2+2** s'il est marié, où *nbEnfants* est son nombre d'enfants.
- s'il a au moins trois enfants, il a une demi-part de plus
- on calcule son revenu imposable **R=0.72*S** où S est son salaire annuel

- on calcule son coefficient familial **QF=R/nbParts**
- on calcule son impôt **I**. Considérons le tableau suivant :

12620.0	0	0
13190	0.05	631
15640	0.1	1290.5
24740	0.15	2072.5
31810	0.2	3309.5
39970	0.25	4900
48360	0.3	6898.5
55790	0.35	9316.5
92970	0.4	12106
127860	0.45	16754.5
151250	0.50	23147.5
172040	0.55	30710
195000	0.60	39312
0	0.65	49062

Chaque ligne a 3 champs. Pour calculer l'impôt I, on recherche la première ligne où *QF<=champ1*. Par exemple, si QF=23000 on trouvera la ligne

24740 0.15 2072.5

L'impôt I est alors égal à **0.15*R - 2072.5*nbParts**. Si QF est tel que la relation *QF<=champ1* n'est jamais vérifiée, alors ce sont les coefficients de la dernière ligne qui sont utilisés. Ici :

0 0.65 49062

ce qui donne l'impôt I=0.65*R - 49062*nbParts.

Le programme Java correspondant est le suivant :

```
import java.io.*;

public class impots{

  // ------------ main
  public static void main(String arg[]){

    // données

    // limites des tranches d'impôts
    double Limites[]={12620, 13190, 15640, 24740, 31810, 39970, 48360,55790, 92970, 127860, 151250,
172040, 195000, 0};
    // coeff appliqué au nombre de parts
    double Coeffn[]={0, 631, 1290.5, 2072.5, 3309.5, 4900, 6898.5, 9316.5,12106, 16754.5, 23147.5, 30710,
39312, 49062};

    // le programme

    // création du flux d'entrée clavier
    BufferedReader IN=null;
    try{
      IN=new BufferedReader(new InputStreamReader(System.in));
    }
    catch(Exception e){
      erreur("Création du flux d'entrée", e, 1);
    }

      // on récupère le statut marital
    boolean OK=false;
    String reponse=null;
    while(! OK){
      try{
        System.out.print("Etes-vous marié(e) (O/N) ? ");
        reponse=IN.readLine();
        reponse=reponse.trim().toLowerCase();
        if (! reponse.equals("o") && !reponse.equals("n"))
          System.out.println("Réponse incorrecte. Recommencez");
        else OK=true;
      } catch(Exception e){
        erreur("Lecture état marital",e,2);
      }
    }
    boolean Marie = reponse.equals("o");

      // nombre d'enfants
    OK=false;
    int NbEnfants=0;
    while(! OK){
      try{
        System.out.print("Nombre d'enfants : ");
        reponse=IN.readLine();
        try{
          NbEnfants=Integer.parseInt(reponse);
          if(NbEnfants>=0) OK=true;
            else System.err.println("Réponse incorrecte. Recommencez");
        } catch(Exception e){
```

```
          System.err.println("Réponse incorrecte. Recommencez");
        }// try
      } catch(Exception e){
        erreur("Lecture état marital",e,2);
      }// try
    }// while

    // salaire
    OK=false;
    long Salaire=0;
    while(! OK){
      try{
        System.out.print("Salaire annuel : ");
        reponse=IN.readLine();
        try{
          Salaire=Long.parseLong(reponse);
          if(Salaire>=0)  OK=true;
            else System.err.println("Réponse incorrecte. Recommencez");
        } catch(Exception e){
          System.err.println("Réponse incorrecte. Recommencez");
        }// try
      } catch(Exception e){
        erreur("Lecture Salaire",e,4);
      }// try
    }// while

      // calcul du nombre de parts
    double NbParts;
    if(Marie) NbParts=(double)NbEnfants/2+2;
      else NbParts=(double)NbEnfants/2+1;
    if (NbEnfants>=3) NbParts+=0.5;

      // revenu imposable
    double Revenu;
    Revenu=0.72*Salaire;

      // quotient familial
    double QF;
    QF=Revenu/NbParts;

      // recherche de la tranche d'impots correspondant à QF
    int i;
    int NbTranches=Limites.length;
    Limites[NbTranches-1]=QF;
    i=0;
    while(QF>Limites[i]) i++;
      // l'impôt
    long impots=(long)(i*0.05*Revenu-Coeffn[i]*NbParts);

    // on affiche le résultat
    System.out.println("Impôt à payer : " + impots);
  }// main

  // ------------ erreur
  private static void erreur(String msg, Exception e, int exitCode){
    System.err.println(msg+"("+e+")");
    System.exit(exitCode);
  }// erreur

}// class
```

Les résultats obtenus sont les suivants :

```
C:\Serge\java\impots\1>java impots

Etes-vous marié(e) (O/N) ? o
Nombre d'enfants : 3
Salaire annuel : 200000
Impôt à payer : 16400

C:\Serge\java\impots\1>java impots

Etes-vous marié(e) (O/N) ? n
Nombre d'enfants : 2
Salaire annuel : 200000
Impôt à payer : 33388

C:\Serge\java\impots\1>java impots

Etes-vous marié(e) (O/N) ? w
Réponse incorrecte. Recommencez
Etes-vous marié(e) (O/N) ? q
Réponse incorrecte. Recommencez
Etes-vous marié(e) (O/N) ? o
Nombre d'enfants : q
Réponse incorrecte. Recommencez
Nombre d'enfants : 2
```

```
Salaire annuel : q
Réponse incorrecte. Recommencez
Salaire annuel : 1
Impôt à payer : 0
```

2. Classes et interfaces

2.1 L' objet par l'exemple

2.1.1 Généralités

Nous abordons maintenant, par l'exemple, la programmation objet. Un objet est une entité qui contient des données qui définissent son état (on les appelle des **attributs** ou **propriétés**) et des fonctions (on les appelle des **méthodes**). Un objet est créé selon un modèle qu'on appelle une **classe** :

```
public class C1{
  type1 p1;      // propriété p1
  type2 p2;      // propriété p2
  …
  type3 m3(…){   // méthode m3
    …
  }
  type4 m4(…){   // méthode m4
    …
  }
  …
}
```

A partir de la classe *C1* précédente, on peut créer de nombreux objets *O1, O2,…* Tous auront les propriétés *p1, p2,…* et les méthodes *m3, m4, …* Ils auront des valeurs différentes pour leurs propriétés *pi* ayant ainsi chacun un état qui leur est propre.

Par analogie la déclaration

```
int i,j;
```

crée deux objets (le terme est incorrect ici) de type (classe) *int*. Leur seule propriété est leur valeur.

Si *O1* est un objet de type *C1*, *O1.p1* désigne la propriété *p1* de *O1* et *O1.m1* la méthode *m1* de *O1*.

Considérons un premier modèle d'objet : la classe *personne*.

2.1.2 Définition de la classe personne

La définition de la classe *personne* sera la suivante :

```
import java.io.*;

public class personne{
  // attributs
  private String prenom;
  private String nom;
  private int age;

  // méthode
  public void initialise(String P, String N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }

  // méthode
  public void identifie(){
    System.out.println(prenom+","+nom+","+age);
  }
}
```

Nous avons ici la définition d'une classe, donc un type de donnée. Lorsqu'on va créer des variables de ce type, on les appellera des objets. Une classe est donc un moule à partir duquel sont construits des objets.

Les **membres** ou **champs** d'une classe peuvent être des **données** ou des **méthodes** (fonctions). Ces champs peuvent avoir l'un des trois attributs suivants :

privé	Un champ privé (**private**) n'est accessible que par les seules méthodes internes de la classe
public	Un champ public est accessible par toute fonction définie ou non au sein de la classe

protégé Un champ protégé (**protected**) n'est accessible que par les seules méthodes internes de la classe ou d'un objet **dérivé** (voir ultérieurement le concept d'héritage).

En général, les données d'une classe sont déclarées **privées** alors que ses méthodes sont déclarées **publiques**. Cela signifie que l'utilisateur d'un objet (le programmeur)

a n'aura pas accès directement aux données privées de l'objet
b pourra faire appel aux méthodes publiques de l'objet et notamment à celles qui donneront accès à ses données privées.

La syntaxe de déclaration d'un objet est la suivante :

```
public class nomClasse{
  private  donnée ou méthode privée
  public  donnée ou méthode publique
  protected  donnée ou méthode protégée
}
```

Remarques

- L'ordre de déclaration des attributs *private, protected et public* est **quelconque**.

2.1.3 La méthode initialise

Revenons à notre classe **personne** déclarée comme :

```
import java.io.*;

public class personne{
  // attributs
  private String prenom;
  private String nom;
  private int age;

  // méthode
  public void initialise(String P, String N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }

  // méthode
  public void identifie(){
    System.out.println(prenom+","+nom+","+age);
  }
}
```

Quel est le rôle de la méthode *initialise* ? Parce que **nom, prenom** et **age** sont des données **privées** de la classe **personne**, les instructions

```
personne p1;
p1.prenom="Jean";
p1.nom="Dupont";
p1.age=30;
```

sont illégales. Il nous faut initialiser un objet de type *personne* via une méthode publique. C'est le rôle de la méthode *initialise*. On écrira :

```
personne p1;
p1.initialise("Jean","Dupont",30);
```

L'écriture *p1.initialise* est légale car *initialise* est d'accès public.

2.1.4 L'opérateur new

La séquence d'instructions

```
personne p1;
p1.initialise("Jean","Dupont",30);
```

est incorrecte. L'instruction

```
  personne p1;
```

déclare *p1* comme une référence à un objet de type *personne*. Cet objet n'existe pas encore et donc *p1* n'est pas initialisé. C'est comme si on écrivait :

```
personne p1=null;
```

où on indique explicitement avec le mot clé **null** que la variable *p1* ne référence encore aucun objet.

Lorsqu'on écrit ensuite

```
p1.initialise("Jean","Dupont",30);
```

on fait appel à la méthode *initialise* de l'objet référencé par *p1*. Or cet objet n'existe pas encore et le compilateur signalera l'erreur. Pour que *p1* référence un objet, il faut écrire :

```
personne p1=new personne();
```

Cela a pour effet de créer un objet de type *personne* non encore initialisé : les attributs *nom* et *prenom* qui sont des références d'objets de type *String* auront la valeur *null*, et *age* la valeur *0*. Il y a donc une initialisation par défaut. Maintenant que *p1* référence un objet, l'instruction d'initialisation de cet objet

```
p1.initialise("Jean","Dupont",30);
```

est valide.

2.1.5 Le mot clé this

Regardons le code de la méthode *initialise* :

```
public void initialise(String P, String N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }
```

L'instruction *this.prenom=P* signifie que l'attribut *prenom* de l'objet courant (*this*) reçoit la valeur *P*. Le mot clé *this* désigne l'objet courant : celui dans lequel se trouve la méthode exécutée. Comment le connaît-on ? Regardons comment se fait l'initialisation de l'objet référencé par *p1* dans le programme appelant :

```
p1.initialise("Jean","Dupont",30);
```

C'est la méthode *initialise* de l'objet *p1* qui est appelée. Lorsque dans cette méthode, on référence l'objet *this*, on référence en fait l'objet *p1*. La méthode *initialise* aurait aussi pu être écrite comme suit :

```
public void initialise(String P, String N, int age){
   prenom=P;
   nom=N;
   this.age=age;
  }
```

Lorsqu'une méthode d'un objet référence un attribut *A* de cet objet, l'écriture *this.A* est implicite. On doit l'utiliser explicitement lorsqu'il y a conflit d'identificateurs. C'est le cas de l'instruction :

```
this.age=age;
```

où *age* désigne un attribut de l'objet courant ainsi que le paramètre *age* reçu par la méthode. Il faut alors lever l'ambiguïté en désignant l'attribut *age* par *this.age*.

2.1.6 Un programme de test

Voici un programme de test :

```
public class test1{
  public static void main(String arg[]){
    personne p1=new personne();
    p1.initialise("Jean","Dupont",30);
```

```
    p1.identifie();
  }
}
```

La classe *personne* est définie dans le fichier source *personne.java* et est compilée :

```
E:\data\serge\JAVA\BASES\OBJETS\2>javac personne.java

E:\data\serge\JAVA\BASES\OBJETS\2>dir
10/06/2002  09:21                473 personne.java
10/06/2002  09:22                835 personne.class
10/06/2002  09:23                165 test1.java
```

Nous faisons de même pour le programme de test :

```
E:\data\serge\JAVA\BASES\OBJETS\2>javac test1.java

E:\data\serge\JAVA\BASES\OBJETS\2>dir
10/06/2002  09:21                473 personne.java
10/06/2002  09:22                835 personne.class
10/06/2002  09:23                165 test1.java
10/06/2002  09:25                418 test1.class
```

On peut s'étonner que le programme *test1.java* n'importe pas la classe *personne* avec une instruction :

```
import personne;
```

Lorsque le compilateur rencontre dans le code source une référence de classe non définie dans ce même fichier source, il recherche la classe à divers endroits :

- dans les paquetages importés par les instructions *import*
- dans le répertoire à partir duquel le compilateur a été lancé

Dans notre exemple, le compilateur a été lancé depuis le répertoire contenant le fichier *personne.class*, ce qui explique qu'il a trouvé la définition de la classe *personne*. Mettre dans ce cas de figure une instruction *import* provoque une erreur de compilation :

```
E:\data\serge\JAVA\BASES\OBJETS\2>javac test1.java
test1.java:1: '.' expected
import personne;
              ^
1 error
```

Pour éviter cette erreur mais pour rappeler que la classe personne doit être importée, on écrira à l'avenir en début de programme :

```
// classes importées
// import personne;
```

Nous pouvons maintenant exécuter le fichier *test1.class* :

```
E:\data\serge\JAVA\BASES\OBJETS\2>java test1
Jean,Dupont,30
```

Il est possible de rassembler plusieurs classes dans un même fichier source. Rassemblons ainsi les classes *personne* et *test1* dans le fichier source *test2.java*. La classe *test1* est renommée *test2* pour tenir compte du changement du nom du fichier source :

```
// paquetages importés
import java.io.*;
```

```
class personne{
  // attributs
  private String prenom; // prénom de ma personne
  private String nom;    // son nom
  private int age;       // son âge

  // méthode
  public void initialise(String P, String N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }//initialise

  // méthode
  public void identifie(){
    System.out.println(prenom+","+nom+","+age);
  }//identifie
```

```
}//classe
```

```
public class test2{
  public static void main(String arg[]){
    personne p1=new personne();
    p1.initialise("Jean","Dupont",30);
    p1.identifie();
  }
}
```

On notera que la classe *personne* n'a plus l'attribut *public*. En effer, dans un fichier source java, seule une classe peut avoir l'attribut *public*. C'est celle qui a la fonction *main*. Par ailleurs, le fichier source doit porter le nom de cette dernière. Compilons le fichier *test2.java* :

```
E:\data\serge\JAVA\BASES\OBJETS\3>dir
10/06/2002  09:36                633 test2.java

E:\data\serge\JAVA\BASES\OBJETS\3>javac test2.java

E:\data\serge\JAVA\BASES\OBJETS\3>dir
10/06/2002  09:36                633 test2.java
10/06/2002  09:41                832 personne.class
10/06/2002  09:41                418 test2.class
```

On remarquera qu'un fichier .class a été généré pour chacune des classes présentes dans le fichier source. Exécutons maintenant le fichier *test2.class* :

```
E:\data\serge\JAVA\BASES\OBJETS\2>java test2
Jean,Dupont,30
```

Par la suite, on utilisera indifféremment les deux méthodes :

- classes rassemblées dans un unique fichier source
- une classe par fichier source

2.1.7 Une autre méthode initialise

Considérons toujours la classe *personne* et rajoutons-lui la méthode suivante :

```
public void initialise(personne P){
    prenom=P.prenom;
    nom=P.nom;
    this.age=P.age;
  }
```

On a maintenant deux méthodes portant le nom *initialise* : c'est légal tant qu'elles admettent des paramètres différents. C'est le cas ici. Le paramètre est maintenant une référence *P* à une personne. Les attributs de la personne *P* sont alors affectés à l'objet courant (*this*). On remarquera que la méthode *initialise* a un accès direct aux attributs de l'objet *P* bien que ceux-ci soient de type *private*. C'est toujours vrai : les méthodes d'un objet *O1* d'une classe *C* a toujours accès aux attributs privés des autres objets de la même classe *C*.

Voici un test de la nouvelle classe *personne* :

```
// import personne;
import java.io.*;

public class test1{
  public static void main(String arg[]){
    personne p1=new personne();
    p1.initialise("Jean","Dupont",30);
    System.out.print("p1=");
    p1.identifie();
    personne p2=new personne();
    p2.initialise(p1);
    System.out.print("p2=");
    p2.identifie();
  }
}
```

et ses résultats :

```
p1=Jean,Dupont,30
p2=Jean,Dupont,30
```

2.1.8 Constructeurs de la classe personne

Un constructeur est une méthode qui porte le **nom** de la classe et qui est appelée lors de la **création de l'objet**. On s'en sert généralement pour l'initialiser. C'est une méthode qui peut accepter des arguments mais qui ne rend aucun résultat. Son prototype ou sa définition ne sont précédés d'**aucun type** (même pas *void*).

Si une classe a un constructeur acceptant **n** arguments *argi*, la déclaration et l'initialisation d'un objet de cette classe pourra se faire sous la forme :

classe objet =new classe(*arg1,arg2, ... argn*);

ou

classe objet;
…
objet=new classe(*arg1,arg2, ... argn*);

Lorsqu'une classe a un ou plusieurs constructeurs, l'un de ces constructeurs doit être **obligatoirement utilisé** pour créer un objet de cette classe. Si une classe *C* n'a aucun constructeur, elle en a un par défaut qui est le constructeur sans paramètres : *public C()*. Les attributs de l'objet sont alors initialisés avec des valeurs par défaut. C'est ce qui s'est passé lorsque dans les programmes précédents, on avait écrit :

```
  personne p1;
  p1=new personne();
```

Créons deux constructeurs à notre classe *personne* :

```
public class personne{
  // attributs
  private String prenom;
  private String nom;
  private int age;

  // constructeurs
  public personne(String P, String N, int age){
    initialise(P,N,age);
  }
  public personne(personne P){
    initialise(P);
  }

  // méthode
  public void initialise(String P, String N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }
  public void initialise(personne P){
    this.prenom=P.prenom;
    this.nom=P.nom;
    this.age=P.age;
  }

  // méthode
  public void identifie(){
    System.out.println(prenom+","+nom+","+age);
  }
}
```

Nos deux constructeurs se contentent de faire appel aux méthodes *initialise* correspondantes. On rappelle que lorsque dans un constructeur, on trouve la notation *initialise(P)* par exemple, le compilateur traduit par *this.initialise(P)*. Dans le constructeur, la méthode *initialise* est donc appelée pour travailler sur l'objet référencé par *this*, c'est à dire l'objet courant, celui qui est en cours de construction.

Voici un programme de test :

```
// import personne;
import java.io.*;

public class test1{
  public static void main(String arg[]){
    personne p1=new personne("Jean","Dupont",30);
    System.out.print("p1=");
    p1.identifie();
    personne p2=new personne(p1);
    System.out.print("p2=");
    p2.identifie();
```

```
  }
}
```

et les résultats obtenus :

```
p1=Jean,Dupont,30
p2=Jean,Dupont,30
```

2.1.9 Les références d'objets

Nous utilisons toujours la même classe *personne*. Le programme de test devient le suivant :

```
// import personne;
import java.io.*;

public class test1{
  public static void main(String arg[]){
  // p1
    personne p1=new personne("Jean","Dupont",30);
    System.out.print("p1=");  p1.identifie();
  // p2 référence le même objet que p1
    personne p2=p1;
    System.out.print("p2="); p2.identifie();
  // p3 référence un objet qui sera une copie de l'objet référencé par p1
    personne p3=new personne(p1);
    System.out.print("p3=");  p3.identifie();
  // on change l'état de l'objet référencé par p1
    p1.initialise("Micheline","Benoît",67);
    System.out.print("p1=");  p1.identifie();
  // comme p2=p1, l'objet référencé par p2 a du changer d'état
    System.out.print("p2=");  p2.identifie();
  // comme p3 ne référence pas le même objet que p1, l'objet référencé par p3 n'a pas du changer
    System.out.print("p3=");  p3.identifie();
  }
}
```

Les résultats obtenus sont les suivants :

```
p1=Jean,Dupont,30
p2=Jean,Dupont,30
p3=Jean,Dupont,30
p1=Micheline,Benoît,67
p2=Micheline,Benoît,67
p3=Jean,Dupont,30
```

Lorsqu'on déclare la variable *p1* par

```
personne p1=new personne("Jean","Dupont",30);
```

p1 référence l'objet *personne("Jean","Dupont",30)* mais n'est pas l'objet lui-même. En C, on dirait que c'est un pointeur, c.a.d. l'adresse de l'objet créé. Si on écrit ensuite :

```
  p1=null
```

Ce n'est pas l'objet *personne("Jean","Dupont",30)* qui est modifié, c'est la référence *p1* qui change de valeur. L'objet personne("Jean","Dupont",30) sera "perdu" s'il n'est référencé par aucune autre variable.

Lorsqu'on écrit :

```
personne p2=p1;
```

on initialise le pointeur *p2* : il "pointe" sur le même objet (il désigne le même objet) que le pointeur *p1*. Ainsi si on modifie l'objet "pointé" (ou référencé) par *p1*, on modifie celui référencé par *p2*.

Lorsqu'on écrit :

```
personne p3=new personne(p1);
```

il y a création d'un nouvel objet, **copie** de l'objet référencé par *p1*. Ce nouvel objet sera référencé par *p3*. Si on modifie l'objet "pointé" (ou référencé) par *p1*, on ne modifie en rien celui référencé par *p3*. C'est ce que montrent les résultats obtenus.

2.1.10 Les objets temporaires

Dans une expression, on peut faire appel explicitement au constructeur d'un objet : celui-ci est construit, mais nous n'y avons pas accès (pour le modifier par exemple). Cet objet temporaire est construit pour les besoins d'évaluation de l'expression puis abandonné. L'espace mémoire qu'il occupait sera automatiquement récupéré ultérieurement par un programme appelé "ramasse-miettes" dont le rôle est de récupérer l'espace mémoire occupé par des objets qui ne sont plus référencés par des données du programme.

Considérons l'exemple suivant :

```
// import personne;

public class test1{
  public static void main(String arg[]){
    new personne(new personne("Jean","Dupont",30)).identifie();
  }
}
```

et modifions les constructeurs de la classe *personne* afin qu'ils affichent un message :

```
// constructeurs
  public personne(String P, String N, int age){
    System.out.println("Constructeur personne(String, String, int)");
    initialise(P,N,age);
  }
  public personne(personne P){
    System.out.println("Constructeur personne(personne)");
    initialise(P);
  }
```

Nous obtenons les résultats suivants :

```
Constructeur personne(String, String, int)
Constructeur personne(personne)
Jean,Dupont,30
```

montrant la construction successive des deux objets temporaires.

2.1.11 Méthodes de lecture et d'écriture des attributs privés

Nous rajoutons à la classe *personne* les méthodes nécessaires pour lire ou modifier l'état des attributs des objets :

```
public class personne{
  private String prenom;
  private String nom;
  private int age;

  public personne(String P, String N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }

  public personne(personne P){
    this.prenom=P.prenom;
    this.nom=P.nom;
    this.age=P.age;
  }

  public void identifie(){
    System.out.println(prenom+","+nom+","+age);
  }

  // accesseurs
  public String getPrenom(){
    return prenom;
  }
  public String getNom(){
    return nom;
  }
  public int getAge(){
    return age;
  }

  //modifieurs
  public void setPrenom(String P){
    this.prenom=P;
  }
```

```
  public void setNom(String N){
    this.nom=N;
  }
  public void setAge(int age){
    this.age=age;
  }
}
```

Nous testons la nouvelle classe avec le programme suivant :

```
// import personne;

public class test1{
  public static void main(String[] arg){
    personne P=new personne("Jean","Michelin",34);
    System.out.println("P=("+P.getPrenom()+","+P.getNom()+","+P.getAge()+")");
    P.setAge(56);
    System.out.println("P=("+P.getPrenom()+","+P.getNom()+","+P.getAge()+")");
  }
}
```

et nous obtenons les résultats suivants :

```
P=(Jean,Michelin,34)
P=(Jean,Michelin,56)
```

2.1.12 Les méthodes et attributs de classe

Supposons qu'on veuille compter le nombre d'objets *personne* créées dans une application. On peut soi-même gérer un compteur mais on risque d'oublier les objets temporaires qui sont créés ici ou là. Il semblerait plus sûr d'inclure dans les constructeurs de la classe *personne*, une instruction incrémentant un compteur. Le problème est de passer une référence de ce compteur afin que le constructeur puisse l'incrémenter : il faut leur passer un nouveau paramètre. On peut aussi inclure le compteur dans la définition de la classe. Comme c'est un attribut de la classe elle-même et non d'un objet particulier de cette classe, on le déclare différemment avec le mot clé *static* :

```
  private static long nbPersonnes;    // nombre de personnes créées
```

Pour le référencer, on écrit *personne.nbPersonnes* pour montrer que c'est un attribut de la classe *personne* elle-même. Ici, nous avons créé un attribut privé auquel on n'aura pas accès directement en-dehors de la classe. On crée donc une méthode publique pour donner accès à l'attribut de classe *nbPersonnes*. Pour rendre la valeur de *nbPersonnes* la méthode n'a pas besoin d'un objet particulier : en effet *nbPersonnes* n'est pas l'attribut d'un objet particulier, il est l'attribut de toute une classe. Aussi a-t-on besoin d'une méthode de classe déclarée elle aussi *static* :

```
public static long getNbPersonnes(){
  return nbPersonnes;
}
```

qui de l'extérieur sera appelée avec la syntaxe *personne.getNbPersonnes()*. Voici un exemple.

La classe *personne* devient la suivante :

```
public class personne{

  // attribut de classe
  private static long nbPersonnes=0;

  // attributs d'objets
  …

  // constructeurs
  public personne(String P, String N, int age){
    initialise(P,N,age);
    nbPersonnes++;
  }
  public personne(personne P){
    initialise(P);
    nbPersonnes++;
  }

  // méthode
  …

  // méthode de classe
  public static long getNbPersonnes(){
    return nbPersonnes;
  }
```

```
}// class
```

Avec le programme suivant :

```
// import personne;

public class test1{
  public static void main(String arg[]){
    personne p1=new personne("Jean","Dupont",30);
    personne p2=new personne(p1);
    new personne(p1);
    System.out.println("Nombre de personnes créées : "+personne.getNbPersonnes());
  }// main
}//test1
```

on obtient les résultats suivants :

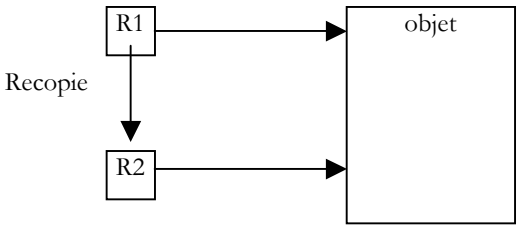
```
   Nombre de personnes créées : 3
```

2.1.13 Passage d'un objet à une fonction

Nous avons déjà dit que Java passait les paramètres effectifs d'une fonction par valeur : les valeurs des paramètres effectifs sont recopiées dans les paramètres formels. Une fonction ne peut donc modifier les paramètres effectifs.

Dans le cas d'un objet, il ne faut pas se laisser tromper par l'abus de langage qui est fait systématiquement en parlant d'objet au lieu de référence d'objet. Un objet n'est manipulé que via une référence (un pointeur) sur lui. Ce qui est donc transmis à une fonction, n'est pas l'objet lui-même mais une référence sur cet objet. C'est donc la valeur de la référence et non la valeur de l'objet lui-même qui est dupliquée dans le paramètre formel : il n'y a pas construction d'un nouvel objet.

Si une référence d'objet R1 est transmise à une fonction, elle sera recopiée dans le paramètre formel correspondant R2. Aussi les références R2 et R1 désignent-elles le même objet. Si la fonction modifie l'objet pointé par R2, elle modifie évidemment celui référencé par R1 puisque c'est le même.



C'est ce que montre l'exemple suivant :

```
// import personne;

public class test1{
  public static void main(String arg[]){
    personne p1=new personne("Jean","Dupont",30);
    System.out.print("Paramètre effectif avant modification : ");
    p1.identifie();
    modifie(p1);
    System.out.print("Paramètre effectif après modification : ");
    p1.identifie();
  }// main

  private static void modifie(personne P){
    System.out.print("Paramètre formel avant modification : ");
    P.identifie();
    P.initialise("Sylvie","Vartan",52);
    System.out.print("Paramètre formel après modification : ");
    P.identifie();
  }// modifie
}// class
```

La méthode *modifie* est déclarée *static* parce que c'est une méthode de classe : on n'a pas à la préfixer par un objet pour l'appeler. Les résultats obtenus sont les suivants :

```
Constructeur personne(String, String, int)
Paramètre effectif avant modification : Jean,Dupont,30
Paramètre formel avant modification : Jean,Dupont,30
Paramètre formel après modification : Sylvie,Vartan,52
```

```
Paramètre effectif après modification : Sylvie,Vartan,52
```

On voit qu'il n'y a construction que d'un objet : celui de la personne *p1* de la fonction *main* et que l'objet a bien été modifié par la fonction *modifie*.

2.1.14 Encapsuler les paramètres de sortie d'une fonction dans un objet

A cause du passage de paramètres par valeur, on ne sait pas écrire en Java une fonction qui aurait des paramètres de sortie de type *int* par exemple car on ne sait pas passer la référence d'un type *int* qui n'est pas un objet. On peut alors créer une classe encapsulant le type *int* :

```
public class entieres{
  private int valeur;

  public entieres(int valeur){
    this.valeur=valeur;
  }

  public void setValue(int valeur){
    this.valeur=valeur;
  }

  public int getValue(){
    return valeur;
  }
}
```

La classe précédente a un constructeur permettant d'initialiser un entier et deux méthodes permettant de lire et modifier la valeur de cet entier. On teste cette classe avec le programme suivant :

```
// import entieres;

public class test2{
  public static void main(String[] arg){
    entieres I=new entieres(12);
    System.out.println("I="+I.getValue());
    change(I);
    System.out.println("I="+I.getValue());
  }
  private static void change(entieres entier){
    entier.setValue(15);
  }
}
```

et on obtient les résultats suivants :

```
I=12
I=15
```

2.1.15 Un tableau de personnes

Un objet est une donnée comme une autre et à ce titre plusieurs objets peuvent être rassemblés dans un tableau :

```
// import personne;

public class test1{
  public static void main(String arg[]){
    personne[] amis=new personne[3];
    System.out.println("----------------");
    amis[0]=new personne("Jean","Dupont",30);
    amis[1]=new personne("Sylvie","Vartan",52);
    amis[2]=new personne("Neil","Armstrong",66);
    int i;
    for(i=0;i<amis.length;i++)
      amis[i].identifie();
  }
}
```

L'instruction *personne[] amis=new personne[3];* crée un tableau de 3 éléments de type *personne*. Ces 3 éléments sont initialisés ici avec la valeur *null*, c.a.d. qu'ils ne référencent aucun objet. De nouveau, par abus de langage, on parle de tableau d'objets alors que ce n'est qu'un tableau de références d'objets. La création du tableau d'objets, tableau qui est un objet lui-même (présence de *new*) ne crée donc en soi aucun objet du type de ses éléments : il faut le faire ensuite.

On obtient les résultats suivants :

```
----------------
```

```
Constructeur personne(String, String, int)
Constructeur personne(String, String, int)
Constructeur personne(String, String, int)
Jean,Dupont,30
Sylvie,Vartan,52
Neil,Armstrong,66
```

2.2 L'héritage par l'exemple

2.2.1 Généralités

Nous abordons ici la notion d'héritage. Le but de l'héritage est de "personnaliser" une classe existante pour qu'elle satisfasse à nos besoins. Supposons qu'on veuille créer une classe *enseignant* : un enseignant est une personne particulière. Il a des attributs qu'une autre personne n'aura pas : la matière qu'il enseigne par exemple. Mais il a aussi les attributs de toute personne : prénom, nom et âge. Un enseignant fait donc pleinement partie de la classe *personne* mais a des attributs supplémentaires. Plutôt que d'écrire une classe *enseignant* en partant de rien, on préfèrerait reprendre l'acquis de la classe *personne* qu'on adapterait au caractère particulier des enseignants. C'est le concept d'héritage qui nous permet cela.

Pour exprimer que la classe *enseignant* hérite des propriétés de la classe *personne*, on écrira :

public class enseignant ***extends*** *personne*

personne est appelée la classe parent (ou mère) et *enseignant* la classe dérivée (ou fille). Un objet *enseignant* a toutes les qualités d'un objet *personne* : il a les mêmes attributs et les mêmes méthodes. Ces attributs et méthodes de la classe parent ne sont pas répétées dans la définition de la classe fille : on se contente d'indiquer les attributs et méthodes rajoutés par la classe fille :

```
class enseignant extends personne{
// attributs
  private int section;

// constructeur
  public enseignant(String P, String N, int age,int section){
    super(P,N,age);
    this.section=section;
  }
}
```

Nous supposons que la classe *personne* est définie comme suit :

```
public class personne{
  private String prenom;
  private String nom;
  private int age;

  public personne(String P, String N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }

  public personne(personne P){
    this.prenom=P.prenom;
    this.nom=P.nom;
    this.age=P.age;
  }

  public String identite(){
    return "personne("+prenom+","+nom+","+age+")";
  }

  // accesseurs
  public String getPrenom(){
    return prenom;
  }
  public String getNom(){
    return nom;
  }
  public int getAge(){
    return age;
  }

  //modifieurs
  public void setPrenom(String P){
    this.prenom=P;
  }
  public void setNom(String N){
    this.nom=N;
  }
```

```
  public void setAge(int age){
    this.age=age;
  }
}
```

La méthode *identifie* a été légèrement modifiée pour rendre une chaîne de caractères identifiant la personne et porte maintenant le nom *identite*. Ici la classe *enseignant* rajoute aux méthodes et attributs de la classe *personne* :

- un attribut *section* qui est le n° de section auquel appartient l'enseignant dans le corps des enseignants (une section par discipline en gros)
- un nouveau constructeur permettant d'initialiser tous les attributs d'un enseignant

2.2.2 Construction d'un objet enseignant

Le constructeur de la classe *enseignant* est le suivant :

```
// constructeur
  public enseignant(String P, String N, int age,int section){
    super(P,N,age);
    this.section=section;
  }
```

L'instruction *super(P,N,age)* est un appel au constructeur de la classe parent, ici la classe *personne*. On sait que ce constructeur initialise les champs *prenom, nom* et *age* de l'objet *personne* contenu à l'intérieur de l'objet *étudiant*. Cela paraît bien compliqué et on pourrait préférer écrire :

```
// constructeur
  public enseignant(String P, String N, int age,int section){
    this.prenom=P;
    this.nom=N
    this.age=age
    this.section=section;
  }
```

C'est impossible. La classe *personne* a déclaré privés (*private*) ses trois champs *prenom*, *nom* et *age*. Seuls des objets de la même classe ont un accès direct à ces champs. Tous les autres objets, y compris des objets fils comme ici, doivent passer par des méthodes publiques pour y avoir accès. Cela aurait été différent si la classe *personne* avait déclaré protégés (*protected*) les trois champs : elle autorisait alors des classes dérivées à avoir un accès direct aux trois champs. Dans notre exemple, utiliser le constructeur de la classe parent était donc la bonne solution et c'est la méthode habituelle : lors de la construction d'un objet fils, on appelle d'abord le constructeur de l'objet parent puis on complète les initialisations propres cette fois à l'objet fils (*section* dans notre exemple).

Tentons un premier programme :

```
// import personne;
// import enseignant;

public class test1{
  public static void main(String arg[]){
    System.out.println(new enseignant("Jean","Dupont",30,27).identite());
  }
}
```

Ce programme ce contente de créer un objet *enseignant* (new) et de l'identifier. La classe *enseignant* n'a pas de méthode *identité* mais sa classe parent en a une qui de plus est publique : elle devient par héritage une méthode publique de la classe *enseignant*.

Les fichiers source des classes sont rassemblés dans un même répertoire puis compilés :

```
E:\data\serge\JAVA\BASES\OBJETS\4>dir
10/06/2002  10:00                765 personne.java
10/06/2002  10:00                212 enseignant.java
10/06/2002  10:01                192 test1.java

E:\data\serge\JAVA\BASES\OBJETS\4>javac *.java

E:\data\serge\JAVA\BASES\OBJETS\4>dir
10/06/2002  10:00                765 personne.java
10/06/2002  10:00                212 enseignant.java
10/06/2002  10:01                192 test1.java
10/06/2002  10:02                316 enseignant.class
10/06/2002  10:02              1 146 personne.class
10/06/2002  10:02                550 test1.class
```

Le fichier *test1.class* est exécuté :

```
E:\data\serge\JAVA\BASES\OBJETS\4>java test1
personne(Jean,Dupont,30)
```

2.2.3 Surcharge d'une méthode

Dans l'exemple précédent, nous avons eu l'identité de la partie *personne* de l'enseignant mais il manque certaines informations propres à la classe *enseignant* (la section). On est donc amené à écrire une méthode permettant d'identifier l'enseignant :

```
class enseignant extends personne{
  int section;

  public enseignant(String P, String N, int age,int section){
    super(P,N,age);
    this.section=section;
  }

  public String identite(){
    return "enseignant("+super.identite()+","+section+")";
  }
}
```

La méthode *identite* de la classe *enseignant* s'appuie sur la méthode *identite* de sa classe mère (*super.identite*) pour afficher sa partie "*personne*" puis complète avec le champ *section* qui est propre à la classe *enseignant*.

La classe *enseignant* dispose maintenant deux méthodes *identite* :

- celle héritée de la classe parent *personne*
- la sienne propre

Si E est un ojet *enseignant*, *E.identite* désigne la méthode *identite* de la classe *enseignant*. On dit que la méthode *identite* de la classe mère est "surchargée" par la méthode *identite* de la classe fille. De façon générale, si *O* est un objet et *M* une méthode, pour exécuter la méthode *O.M*, le système cherche une méthode *M* dans l'ordre suivant :

- dans la classe de l'objet O
- dans sa classe mère s'il en a une
- dans la classe mère de sa classe mère si elle existe
- etc…

L'héritage permet donc de surcharger dans la classe fille des méthodes de même nom dans la classe mère. C'est ce qui permet d'adapter la classe fille à ses propres besoins. Associée au polymorphisme que nous allons voir un peu plus loin, la surcharge de méthodes est le principal intérêt de l'héritage.

Considérons le même exemple que précédemment :

```
// import personne;
// import enseignant;

public class test1{
  public static void main(String arg[]){
    System.out.println(new enseignant("Jean","Dupont",30,27).identite());
  }
}
```

Les résultats obtenus sont cette fois les suivants :

```
   enseignant(personne(Jean,Dupont,30),27)
```

2.2.4 Le polymorphisme

Considérons une lignée de classes : $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n$

où $C_i \rightarrow C_j$ indique que la classe C_j est dérivée de la classe C_i. Cela entraîne que la classe C_j a toutes les caractéristiques de la classe C_i plus d'autres. Soient des objets O_i de type C_i. Il est légal d'écrire :

$O_i=O_j$ avec j>i

En effet, par héritage, la classe C_j a toutes les caractéristiques de la classe C_i plus d'autres. Donc un objet O_j de type C_j contient en lui un objet de type C_i. L'opération

$O_i=O_j$

fait que O_i est une référence à l'objet de type C_i contenu dans l'objet O_j.

Le fait qu'une variable *Oi* de classe *Ci* puisse en fait référencer non seulement un objet de la classe *Ci* mais en fait tout objet dérivé de la classe *Ci* est appelé **polyporphisme** : la faculté pour une variable de référencer différents types d'objets.

Prenons un exemple et considérons la fonction suivante indépendante de toute classe :

```
public static void affiche(Object obj){
   ….
}
```

La classe *Object* est la "mère" de toutes les classes Java. Ainsi lorsqu'on écrit :

```
public class personne
```

on écrit implicitement :

```
public class personne extends Object
```

Ainsi tout objet Java contient en son sein une partie de type *Object*. Ainsi on pourra écrire :

```
enseignant e;
affiche(e);
```

Le paramètre formel de type *Object* de la fonction affiche va recevoir une valeur de type *enseignant*. Comme enseignant dérive de *Object*, c'est légal.

2.2.5 Surcharge et polymorphisme

Complétons notre fonction *affiche* :

```
public static void affiche(Object obj){
   System.out.println(obj.toString());
}
```

La méthode *obj.toString()* rend une chaîne de caractères identifiant l'objet *obj* sous la forme *nom_de_la_classe@adresse_de_l'objet*. Que se passe-t-il dans le cas de notre exemple précédent :

```
enseignant e=new enseignant(...);
affiche(e);
```

Le système devra exécuter l'instruction *System.out.println(e.toString())* où *e* est un objet enseignant. Il va chercher une méthode *toString* dans la hiérarchie des classes menant à la classe *enseignant* en commençant par la dernière :

- dans la classe *enseignant*, il ne trouve pas de méthode *toString()*
- dans la classe mère *personne*, il ne trouve pas de méthode *toString()*
- dans la classe mère *Object*, il trouve la méthode *toString()* et l'exécute

C'est ce que montre le programme suivant :

```
// import personne;
// import enseignant;

public class test1{
  public static void main(String arg[]){
    enseignant e=new enseignant("Lucile","Dumas",56,61);
    affiche(e);
    personne p=new personne("Jean","Dupont",30);
    affiche(p);
  }
  public static void affiche(Object obj){
    System.out.println(obj.toString());
  }
}
```

Les résultats obtenus sont les suivants :

```
enseignant@1ee789
```

```
personne@1ee770
```

C'est à dire *nom_de_la_classe@adresse_de_l'objet*. Comme ce n'est pas très explicite, on est tenté de définir une méthode *toString* pour les classes *personne* et *etudiant* qui surchargeraient la méthode *toString* de la classe mère *Object*. Plutôt que d'écrire des méthodes qui seraient proches des méthodes *identite* déjà existantes dans les classes *personne* et *enseignant*, contentons-nous de renommer *toString* ces méthodes *identite* :

```
public class personne{
  ...

  public String toString(){
    return "personne("+prenom+","+nom+","+age+")";
  }

  ...
}

class enseignant extends personne{
  int section;

  …

  public String toString(){
    return "enseignant("+super.toString()+","+section+")";
  }
}
```

Avec le même programme de test qu'auparavant, les résultats obtenus sont les suivants :

```
enseignant(personne(Lucile,Dumas,56),61)
personne(Jean,Dupont,30)
```

2.3 Classes internes

Une classe peut contenir la définition d'une autre classe. Considérons l'exemple suivant :

```
// classes importées
import java.io.*;

public class test1{

   // classe interne
   private class article{
    // on définit la structure
    private String code;
    private String nom;
    private double prix;
    private int stockActuel;
    private int stockMinimum;

    // constructeur
    public article(String code, String nom, double prix, int stockActuel, int stockMinimum){
      // initialisation des attributs
       this.code=code;
       this.nom=nom;
       this.prix=prix;
       this.stockActuel=stockActuel;
       this.stockMinimum=stockMinimum;
    }//constructeur

    //toString
    public String toString(){
      return "article("+code+","+nom+","+prix+","+stockActuel+","+stockMinimum+")";
    }//toString
  }//classe article

  // données locales
  private article art=null;

  // constructeur
  public test1(String code, String nom, double prix, int stockActuel, int stockMinimum){
    // définition attribut
    art=new article(code, nom, prix, stockActuel,stockMinimum);
  }//test1

  // accesseur
  public article getArticle(){
    return art;
  }//getArticle
```

```
  public static void main(String arg[]){
    // création d'une instance test1
    test1 t1=new test1("a100","velo",1000,10,5);
     // affichage test1.art
     System.out.println("art="+t1.getArticle());
   }//main

}// fin class
```

La classe *test1* contient la définition d'une autre classe, la classe *article*. On dit que *article* est une classe interne à la classe *test1*. Cela peut être utile lorsque la classe interne n'a d'utilité que dans la classe qui la contient. Lors de la compilation du source *test1.java* ci-dessus, on obtient deux fichiers .class :

```
E:\data\serge\JAVA\classes\interne>dir
05/06/2002  17:26                1 362 test1.java
05/06/2002  17:26                  941 test1$article.class
05/06/2002  17:26                1 020 test1.class
```

Un fichier *test1$article.class* a été généré pour la classe *article* interne à la classe *test1*. Si on exécute le programme ci-dessus, on obtient les résultats suivants :

```
E:\data\serge\JAVA\classes\interne>java test1
art=article(a100,velo,1000.0,10,5)
```

2.4 Les interfaces

Une interface est un ensemble de prototypes de méthodes ou de propriétés qui forme un contrat. Une classe qui décide d'implémenter une interface s'engage à fournir une implémentation de toutes les méthodes définies dans l'interface. C'est le compilateur qui vérifie cette implémentation.

Voici par exemple la définition de l'interface *java.util.Enumeration :*

Method Summary

boolean	**hasMoreElements**() Tests if this enumeration contains more elements.
Object	**nextElement**() Returns the next element of this enumeration if this enumeration object has at least one more element to provide.

Toute classe implémentant cette interface sera déclarée comme

```
public class C : Enumeration{
  ...
  boolean hasMoreElements(){....}
  Object nextElement(){...}
}
```

Les méthodes *hasMoreElements()* et *nextElement()* devront être définies dans la classe C.

Considérons le code suivant définissant une classe *élève* définissant le nom d'un élève et sa note dans une matière :

```
  // une classe élève
public class élève{
  // des attributs publics
  public String nom;
  public double note;
  // constructeur
  public élève(String NOM, double NOTE){
    nom=NOM;
    note=NOTE;
  }//constructeur
}//élève
```

Nous définissons une classe *notes* rassemblant les notes de tous les élèves dans une matière :

```
// classes importées
// import élève
```

```
// classe notes
public class notes{

  // attributs
  protected String matière;
  protected élève[] élèves;

  // constructeur
  public notes (String MATIERE, élève[] ELEVES){
    // mémorisation élèves & matière
    matière=MATIERE;
    élèves=ELEVES;
  }//notes

  // toString
  public String toString(){
    String valeur="matière="+matière +", notes=(";
    int i;
    // on concatène toutes les notes
    for (i=0;i<élèves.length-1;i++){
      valeur+="["+élèves[i].nom+","+élèves[i].note+"],";
    };
    //dernière note
    if(élèves.length!=0){ valeur+="["+élèves[i].nom+","+élèves[i].note+"]";}
    valeur+=")";
    // fin
    return valeur;
  }//toString
}//classe
```

Les attributs *matière* et *élèves* sont déclarés *protected* pour être accessibles d'une classe dérivée. Nous décidons de dériver la classe *notes* dans une classe *notesStats* qui aurait deux attributs supplémentaires, la moyenne et l'écart-type des notes :

```
public class notesStats extends notes implements Istats {
  // attributs
  private double _moyenne;
  private double _écartType;
```

La classe *notesStats* dérive de la classe *notes* et implémente l'interface *Istats* suivante :

```
// une interface
public interface Istats{
  double moyenne();
  double écartType();
}//
```

Cela signifie que la classe *notesStats* doit avoir deux méthodes appelées *moyenne* et *écartType* avec la signature indiquée dans l'interface *Istats*. La classe *notesStats* est la suivante :

```
// classes importées
// import notes;
// import Istats;
// import élève;

public class notesStats extends notes implements Istats {
  // attributs
  private double _moyenne;
  private double _écartType;

  // constructeur
  public notesStats (String MATIERE, élève[] ELEVES){
    // construction de la classe parente
    super(MATIERE,ELEVES);
    // calcul moyenne des notes
    double somme=0;
    for (int i=0;i<élèves.length;i++){
      somme+=élèves[i].note;
    }
    if(élèves.length!=0) _moyenne=somme/élèves.length;
    else _moyenne=-1;
    // écart-type
    double carrés=0;
    for (int i=0;i<élèves.length;i++){
      carrés+=Math.pow((élèves[i].note-_moyenne),2);
    }//for
    if(élèves.length!=0) _écartType=Math.sqrt(carrés/élèves.length);
    else _écartType=-1;
  }//constructeur


  // ToString
  public String toString(){
    return super.toString()+",moyenne="+_moyenne+",écart-type="+_écartType;
  }//ToString

  // méthodes de l'interface Istats
  public double moyenne(){
```

```
    // rend la moyenne des notes
    return _moyenne;
  }//moyenne
  public double écartType(){
    // rend l'écart-type
    return _écartType;
  }//écartType
}//classe
```

La moyenne *_moyenne* et l'écart-type *_ecartType* sont calculés dès la construction de l'objet. Aussi les méthodes *moyenne* et *écartType* n'ont-elles qu'à rendre la valeur des attributs *_moyenne* et *_ecartType*. Les deux méthodes rendent -1 si le tableau des élèves est vide.

La classe de test suivante :

```
// classes importées
// import élève;
// import Istats;
// import notes;
// import notesStats;

// classe de test
public class test{
  public static void main(String[] args){
    // qqs élèves & notes
    élève[] ELEVES=new élève[] { new élève("paul",14),new élève("nicole",16), new élève("jacques",18)};
    // qu'on enregistre dans un objet notes
    notes anglais=new notes("anglais",ELEVES);
    // et qu'on affiche
    System.out.println(""+anglais);
    // idem avec moyenne et écart-type
    anglais=new notesStats("anglais",ELEVES);
    System.out.println(""+anglais);
  }//main
}//classe
```

donne les résultats :

```
matière=anglais, notes=([paul,14.0],[nicole,16.0],[jacques,18.0])
matière=anglais, notes=([paul,14.0],[nicole,16.0],[jacques,18.0]),moyenne=16.0,écart-
type=1.632993161855452
```

Les différentes classes de cet exemple font toutes l'objet d'un fichier source différent :

```
E:\data\serge\JAVA\interfaces\notes>dir
06/06/2002  14:06                707 notes.java
06/06/2002  14:06                878 notes.class
06/06/2002  14:07              1 160 notesStats.java
06/06/2002  14:02                101 Istats.java
06/06/2002  14:02                138 Istats.class
06/06/2002  14:05                247 élève.java
06/06/2002  14:05                309 élève.class
06/06/2002  14:07              1 103 notesStats.class
06/06/2002  14:10                597 test.java
06/06/2002  14:10                931 test.class
```

La classe *notesStats* aurait très bien pu implémenter les méthodes *moyenne* et *écartType* pour elle-même sans indiquer qu'elle implémentait l'interface *Istats*. Quel est donc l'intérêt des interfaces ? C'est le suivant : une fonction peut admettre pour paramètre une donnée ayant le type d'une interface I. Tout objet d'une classe C implémentant l'interface I pourra alors être paramètre de cette fonction. Considérons l'interface suivante :

```
// une interface Iexemple
public interface Iexemple{
  int ajouter(int i,int j);
  int soustraire(int i,int j);
}//interface
```

L'interface *Iexemple* définit deux méthodes *ajouter* et *soustraire*. Les classes *classe1* et *classe2* suivantes implémentent cette interface.

```
// classes importées
// import Iexemple;

public class classe1 implements Iexemple{
  public int ajouter(int a, int b){
    return a+b+10;
  }
  public int soustraire(int a, int b){
    return a-b+20;
  }
}//classe
```

```
// classes importées
// import Iexemple;

public class classe2 implements Iexemple{
  public int ajouter(int a, int b){
         return a+b+100;
       }
  public int soustraire(int a, int b){
    return a-b+200;
  }
}//classe
```

Par souci de simplification de l'exemple les classes ne font rien d'autre que d'implémenter l'interface *Iexemple*. Maintenant considérons l'exemple suivant :

```
// classes importées
// import classe1;
// import classe2;

// classe de test
public class test{
  // une fonction statique
  private static void calculer(int i, int j, Iexemple inter){
    System.out.println(inter.ajouter(i,j));
    System.out.println(inter.soustraire(i,j));
  }//calculer

  // la fonction main
  public static void main(String[] arg){
    // création de deux objets classe1 et classe2
    classe1 c1=new classe1();
    classe2 c2=new classe2();
    // appels de la fonction statique calculer
    calculer(4,3,c1);
    calculer(14,13,c2);
  }//main
}//classe test
```

La fonction statique *calculer* admet pour paramètre un élément de type *Iexemple*. Elle pourra donc recevoir pour ce paramètre aussi bien un objet de type *classe1* que de type *classe2*. C'est ce qui est fait dans la fonction *main* avec les résultats suivants :

```
17
21
127
201
```

On voit donc qu'on a là une propriété proche du polymorphisme vu pour les classes. Si donc un ensemble de classes **Ci** non liées entre-elles par héritage (donc on ne peut utiliser le polymorphisme de l'héritage) présentent un ensemble de méthodes de même signature, il peut être intéressant de regrouper ces méthodes dans une interface **I** dont hériteraient toutes les classes concernées. Des instances de ces classes **Ci** peuvent alors être utilisées comme paramètres de fonctions admettant un paramètre de type **I**, c.a.d. des fonctions n'utilisant que les méthodes des objets **Ci** définies dans l'interface **I** et non les attributs et méthodes particuliers des différentes classes **Ci**.

Dans l'exemple précédent, chaque classe ou interface faisait l'objet d'un fichier source séparé :

```
E:\data\serge\JAVA\interfaces\opérations>dir
06/06/2002  14:33                128 Iexemple.java
06/06/2002  14:34                218 classe1.java
06/06/2002  14:32                220 classe2.java
06/06/2002  14:33                144 Iexemple.class
06/06/2002  14:34                325 classe1.class
06/06/2002  14:34                326 classe2.class
06/06/2002  14:36                583 test.java
06/06/2002  14:36                628 test.class
```

Notons enfin que l'héritage d'interfaces peut être multiple, c.a.d. qu'on peut écrire

```
public class classeDérivée extends classeDeBase implements i1,i2,..,in{
...
}
```

où les i_j sont des interfaces.

2.5 Classes anonymes

Dans l'exemple précédent, les classes *classe1* et *classe2* auraient pu ne pas être définies explicitement. Considérons le programme suivant qui fait sensiblement la même chose que le précédent mais sans la définition explicite des classes *classe1* et *classe2* :

```
// classes importées
// import Iexemple;

// classe de test
public class test2{

  // une classe interne
  private static class classe3 implements Iexemple{
      public int ajouter(int a, int b){
        return a+b+1000;
      }
      public int soustraire(int a, int b){
        return a-b+2000;
      }
    };//définition classe3

  // une fonction statique
  private static void calculer(int i, int j, Iexemple inter){
    System.out.println(inter.ajouter(i,j));
    System.out.println(inter.soustraire(i,j));
  }//calculer

  // la fonction main
  public static void main(String[] arg){
    // création de deux objets implémentant l'interface Iexemple
    Iexemple i1=new Iexemple(){
      public int ajouter(int a, int b){
        return a+b+10;
      }
      public int soustraire(int a, int b){
        return a-b+20;
      }
    };//définition i1

    Iexemple i2=new Iexemple(){
      public int ajouter(int a, int b){
        return a+b+100;
      }
      public int soustraire(int a, int b){
        return a-b+200;
      }
    };//définition i2
    // un autre objet Iexemple
    Iexemple i3=new classe3();

    // appels de la fonction statique calculer
    calculer(4,3,i1);
    calculer(14,13,i2);
    calculer(24,23,i3);
  }//main
}//classe test
```

La particularité se trouve dans le code :

```
    // création de deux objets implémentant l'interface Iexemple
    Iexemple i1=new Iexemple(){
      public int ajouter(int a, int b){
        return a+b+10;
      }
      public int soustraire(int a, int b){
        return a-b+20;
      }
    };//définition i1
```

On crée un objet *i1* dont le seul rôle est d'implémenter l'interface *Iexemple*. Cet objet est de type *Iexemple*. On peut donc créer des objets de type interface. De très nombreuses méthodes de classes Java rendent des objets de type interface c.a.d. des objets dont le seul rôle est d'implémenter les méthodes d'une interface. Pour créer l'objet *i1*, on pourrait être tenté d'écrire :

```
    Iexemple i1=new Iexemple()
```

Seulement une interface ne peut être instantiée. Seule une classe implémentant cette interface peut l'être. Ici, on définit une telle classe "à la volée" dans le corps même de la définition de l'objet *i1* :

```
    Iexemple i1=new Iexemple(){
      public int ajouter(int a, int b){
        // définition de ajouter
      }
      public int soustraire(int a, int b){
        // définition de soustraire
      }
    };//définition i1
```

La signification d'une telle instruction est analogue à la séquence :

```
public class test2{
.................
// une classe interne
private static class classe1 implements Iexemple{
      public int ajouter(int a, int b){
        // définition de ajouter
      }
      public int soustraire(int a, int b){
        // définition de soustraire
      }
};//définition classe1
.................
  public static void main(String[] arg){
...........
    Iexemple i1=new classe1();
}//main
}//classe
```

Dans l'exemple ci-dessus, on instantie bien une classe et non pas une interface. Une classe définie "à la volée" est dite une classe **anonyme**. C'est une méthode souvent utilisée pour instantier des objets dont le seul rôle est d'implémenter une interface.

L'exécution du programme précédent donne les résultats suivants :

```
17
21
127
201
1047
2001
```

L'exemple précéent utilisait des classes anonymes pour implémenter une interface. Celles-ci peuvent être utilisées également pour dériver des classes n'ayant pas de constructeurs avec paramètres. Considérons l'exemple suivant :

```
// classes importées
// import Iexemple;

class classe3 implements Iexemple{
  public int ajouter(int a, int b){
    return a+b+1000;
  }
  public int soustraire(int a, int b){
    return a-b+2000;
  }
};//définition classe3

public class test4{

  // une fonction statique
  private static void calculer(int i, int j, Iexemple inter){
    System.out.println(inter.ajouter(i,j));
    System.out.println(inter.soustraire(i,j));
  }//calculer

  // méthode main
  public static void main(String args[]){
    // définition d'une classe anonymé dérivant classe3
     // pour redéfinir soustraire
     classe3 i1=new classe3(){
       public int ajouter(int a, int b){
         return a+b+10000;
       }//soustraire
     };//i1
    // appels de la fonction statique calculer
    calculer(4,3,i1);
  }//main
}//classe
```

Nous y retrouvons une classe *classe3* implémentant l'interface *Iexemple*. Dans la fonction *main*, nous définissons une variable *i1* ayant pour type, une classe dérivée de *classe3*. Cette classe dérivée est définie "à la volée" dans une classe anonyme et redéfinit la méthode *ajouter* de la classe *classe3*. La syntaxe est identique à celle de la classe anonyme implémentant une interface. Seulement ici, le compilateur détecte que *classe3* n'est pas une interface mais une classe. Pour lui, il s'agit alors d'une dérivation de classe. Toutes les méthodes qu'il trouvera dans le corps de la classe anonyme remplaceront les méthodes de même nom de la classe de base.

L'exécution du programme précédent donne les résultats suivants :

```
E:\data\serge\JAVA\classes\anonyme>java test4
10007
```

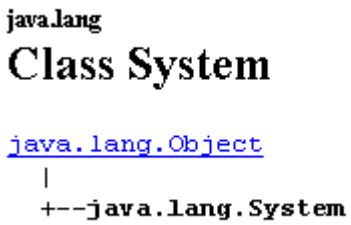
2.6 Les paquetages

2.6.1 Créer des classes dans un paquetage

Pour écrire une ligne à l'écran, nous utilisons l'instruction

```
System.out.println(...)
```

Si nous regardons la définition de la classe *System* nous découvrons qu'elle s'appelle en fait *java.lang.System* :



Vérifions le sur un exemple :

```
public class test1{
  public static void main(String[] args){
    java.lang.System.out.println("Coucou");
  }//main
}//classe
```

Compilons et exécutons ce programme :

```
E:\data\serge\JAVA\classes\paquetages>javac test1.java

E:\data\serge\JAVA\classes\paquetages>dir
06/06/2002  15:40                 127 test1.java
06/06/2002  15:40                 410 test1.class

E:\data\serge\JAVA\classes\paquetages>java test1
Coucou
```

Pourquoi donc pouvons-nous écrire

```
    System.out.println("Coucou");
```

au lieu de

```
    java.lang.System.out.println("Coucou");
```

Parce que de façon implicite, il y a pour tout programme Java, une importation systématique du "paquetage" *java.lang*. Ainsi tout se passe comme si on avait au début de tout programme l'instruction :

```
import java.lang.*;
```

Que signifie cette instruction ? Elle donne accès à toutes les classes du paquetage *java.lang*. Le compilateur y trouvera le fichier *System.class* définissant la classe *System*. On ne sait pas encore où le compilateur trouvera le paquetage *java.lang* ni à quoi un paquetage ressemble. Nous y reviendrons. Pour créer une classe dans un paquetage, on écrit :

```
package paquetage;
// définition de la classe
...
```

Pour l'exemple, créons dans un paquetage notre classe *personne* étudiée précédemment. Nous choisirons *istia.st* comme nom de paquetage. La classe *personne* devient :

```
// nom du paquetage dans lequel sera créé la classe personne
package istia.st;
```

```
// classe personne
public class personne{
  // nom, prénom, âge
  private String prenom;
  private String nom;
  private int age;

  // constructeur 1
  public personne(String P, String N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }

  // toString
  public String toString(){
    return "personne("+prenom+","+nom+","+age+")";
  }

}//classe
```

Cette classe est compilée puis placée dans un répertoire *istia\st* du répertoire courant. Pourquoi *istia\st* ? Parce que le paquetage s'appelle *istia.st*.

```
E:\data\serge\JAVA\classes\paquetages\personne>dir
06/06/2002  16:28                467 personne.java
06/06/2002  16:04       <DIR>          istia

E:\data\serge\JAVA\classes\paquetages\personne>dir istia
06/06/2002  16:04       <DIR>          st

E:\data\serge\JAVA\classes\paquetages\personne>dir istia\st
06/06/2002  16:28                675 personne.class
```

Maintenant utilisons la classe *personne* dans une première classe de test :

```
  public class test{
    public static void main(String[] args){
      istia.st.personne p1=new istia.st.personne("Jean","Dupont",20);
      System.out.println("p1="+p1);
    }//main
  }//classe test
```

On remarquera que la classe *personne* est maintenant préfixée du nom de son paquetage *istia.st*. Où le compilateur trouvera-t-il la classe *istia.st.personne* ? Le compilateur cherche les classes dont il a besoin dans une liste prédéfinie de répertoires et dans une arborescence partant du répertoire courant. Ici, il cherchera la classe *istia.st.personne* dans un fichier *istia\st\personne.class*. C'est pourquoi nous avons mis le fichier *personne.class* dans le répertoire *istia\st*. Compilons puis exécutons le programme de test :

```
E:\data\serge\JAVA\classes\paquetages\personne>dir
06/06/2002  16:28                467 personne.java
06/06/2002  16:06                246 test.java
06/06/2002  16:04       <DIR>          istia
06/06/2002  16:06                738 test.class

E:\data\serge\JAVA\classes\paquetages\personne>java test
p1=personne(Jean,Dupont,20)
```

Pour éviter d'écrire

```
      istia.st.personne p1=new istia.st.personne("Jean","Dupont",20);
```

on peut importer la classe *istia.st.personne* avec une clause *import* :

```
import istia.st.personne;
```

Nous pouvons alors écrire

```
      personne p1=new personne("Jean","Dupont",20);
```

et le compilateur traduira par

```
      istia.st.personne p1=new istia.st.personne("Jean","Dupont",20);
```

Le programme de test devient alors le suivant :

```
// espaces de noms importés
import istia.st.personne;

public class test2{
  public static void main(String[] args){
    personne p1=new personne("Jean","Dupont",20);
    System.out.println("p1="+p1);
  }//main
}//classe test2
```

Compilons et exécutons ce nouveau programme :

```
E:\data\serge\JAVA\classes\paquetages\personne>javac test2.java

E:\data\serge\JAVA\classes\paquetages\personne>dir
06/06/2002  16:28                467 personne.java
06/06/2002  16:06                246 test.java
06/06/2002  16:04       <DIR>          istia
06/06/2002  16:06                738 test.class
06/06/2002  16:47                236 test2.java
06/06/2002  16:50                740 test2.class

E:\data\serge\JAVA\classes\paquetages\personne>java test2
p1=personne(Jean,Dupont,20)
```

Nous avons mis le paquetage *istia.st* dans le répertoire courant. Ce n'est pas obligatoire. Mettons-le dans un dossier appelé *mesClasses* toujours dans le répertoire courant. Rappelons que les classes du paquetage *istia.st* sont placées dans un dossier *istia\st*. L'arborescence du répertoire courant est la suivante :

```
E:\data\serge\JAVA\classes\paquetages\personne>dir
06/06/2002  16:28                467 personne.java
06/06/2002  16:06                246 test.java
06/06/2002  16:06                738 test.class
06/06/2002  16:47                236 test2.java
06/06/2002  16:50                740 test2.class
06/06/2002  16:21       <DIR>          mesClasses

E:\data\serge\JAVA\classes\paquetages\personne>dir mesClasses
06/06/2002  16:22       <DIR>          istia

E:\data\serge\JAVA\classes\paquetages\personne>dir mesClasses\istia
06/06/2002  16:22       <DIR>          st

E:\data\serge\JAVA\classes\paquetages\personne>dir mesClasses\istia\st
06/06/2002  16:01               1 153 personne.class
```

Maintenant compilons de nouveau le programme *test2.java* :

```
E:\data\serge\JAVA\classes\paquetages\personne>javac test2.java
test2.java:2: package istia.st does not exist
import istia.st.personne;
```

Le compilateur ne trouve plus le paquetage *istia.st* depuis qu'on l'a déplacé. Remarquons qu'il le cherche à cause de l'instruction *import*. Par défaut, il le cherche à partir du répertoire courant dans un dossier appelé *istia\st* qui n'existe plus. Examinons les options du compilateur :

```
E:\data\serge\JAVA\classes\paquetages\personne>javac
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                    Generate no debugging info
  -g:{lines,vars,source}     Generate only some debugging info
  -O                         Optimize; may hinder debugging or enlarge class file
  -nowarn                    Generate no warnings
  -verbose                   Output messages about what the compiler is doing
  -deprecation               Output source locations where deprecated APIs are used
  -classpath <path>          Specify where to find user class files
  -sourcepath <path>         Specify where to find input source files
  -bootclasspath <path>      Override location of bootstrap class files
  -extdirs <dirs>            Override location of installed extensions
  -d <directory>             Specify where to place generated class files
  -encoding <encoding>       Specify character encoding used by source files
  -source <release>          Provide source compatibility with specified release
  -target <release>          Generate class files for specific VM version
  -help                      Print a synopsis of standard options
```

Ici l'option *-classpath* peut nous être utile. Elle permet d'indiquer au compilateur où chercher ses classes et paquetages. Essayons. Compilons en disant au compilateur que le paquetage *istia.st* est désormais dans le dossier *mesClasses* :

```
E:\data\serge\JAVA\classes\paquetages\personne>javac -classpath mesClasses test2.java

E:\data\serge\JAVA\classes\paquetages\personne>dir
06/06/2002  16:47                236 test2.java
06/06/2002  17:03                740 test2.class
06/06/2002  16:21       <DIR>          mesClasses
```

La compilation se fait cette fois sans problème. Exécutons le programme *test2.class* :

```
E:\data\serge\JAVA\classes\paquetages\personne>java test2
Exception in thread "main" java.lang.NoClassDefFoundError: istia/st/personne
        at test2.main(test2.java:6)
```

C'est maintenant au tour de la machine virtuelle Java de ne pas trouver la classe *istia/st/personne*. Elle la cherche dans le répertoire courant alors qu'elle est maintenant dans le répertoire *mesClasses*. Regardons les options de la machine virtuelle Java :

```
E:\data\serge\JAVA\classes\paquetages\personne>java
Usage: java [-options] class [args...]
           (to execute a class)
   or  java -jar [-options] jarfile [args...]
           (to execute a jar file)

where options include:
    -client       to select the "client" VM
    -server       to select the "server" VM
    -hotspot      is a synonym for the "client" VM  [deprecated]
                  The default VM is client.

    -cp -classpath <directories and zip/jar files separated by ;>
                  set search path for application classes and resources
    -D<name>=<value>
                  set a system property
    -verbose[:class|gc|jni]
                  enable verbose output
    -version      print product version and exit
    -showversion  print product version and continue
    -? -help      print this help message
    -X            print help on non-standard options
    -ea[:<packagename>...|:<classname>]
    -enableassertions[:<packagename>...|:<classname>]
                  enable assertions
    -da[:<packagename>...|:<classname>]
    -disableassertions[:<packagename>...|:<classname>]
                  disable assertions
    -esa | -enablesystemassertions
                  enable system assertions
    -dsa | -disablesystemassertions
                  disable system assertions
```

On voit que la JVM a également une option *classpath* comme le compilateur. Utilisons-la pour lui dire où se trouve le paquetage *istia.st* :

```
E:\data\serge\JAVA\classes\paquetages\personne>java.bat -classpath mesClasses test2
Exception in thread "main" java.lang.NoClassDefFoundError: test2
```

On n'a pas beaucoup progressé. C'est maintenant la classe *test2* elle-même qui n'est pas trouvée. Pour la raison suivante : en l'absence du mot clé *classpath*, le répertoire courant est systématiquement exploré lors de la recherche de classes mais pas lorsqu'il est présent. Du coup, la classe *test2.class* qui se trouve dans le répertoire courant n'est pas trouvée. La solution ? Ajouter le répertoire courant au *classpath*. Le répertoire courant est représenté par le symbole .

```
E:\data\serge\JAVA\classes\paquetages\personne>java -classpath mesClasses;. test2
p1=personne(Jean,Dupont,20)
```

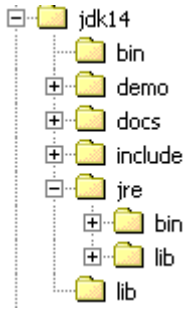
Pourquoi toutes ces complications ? Le but des paquetages est d'éviter les conflits de noms entre classes. Considérons deux entreprises E1 et E2 distribuant des classes empaquetées respectivement dans les paquetages *com.e1* et *com.e2*. Soit un client C qui achète ces deux ensembles de classes dans lesquelles les deux entreprises ont défini toutes deux une classe *personne*. Le client C référencera la classe *personne* de l'entreprise E1 par *com.e1.personne* et celle de l'entreprise E2 par *com.e2.personne* évitant ainsi un conflit de noms.

2.6.2 Recherche des paquetages

Lorsque nous écrivons dans un programme

```
import java.util.*;
```

pour avoir accès à toutes les classes du paquetage *java.util*, où celui-ci est-il trouvé ? Nous avons dit que les paquetages étaient cherchés par défaut dans le répertoire courant ou dans la liste des répertoires déclarés dans l'option *classpath* du compilateur ou de la JVM si cette option est présente. Ils sont également cherchés dans les répertoires *lib* du répertoire d'installation du JDK. Considérons ce répertoire :



Dans cet exemple, les arborescences *jdk14\lib* et *jdk14\jre\lib* seront explorées pour y chercher soit des fichiers **.class**, soit des fichiers **.jar** ou **.zip** qui sont des archives de classes. Faisons par exemple une recherche des fichiers *.jar* se trouvant sous le répertoire *jdk14* précédent :

Nom	Dans le dossier
jaws.jar	E:\Program Files\jdk14\jre\lib
jce.jar	E:\Program Files\jdk14\jre\lib
jsse.jar	E:\Program Files\jdk14\jre\lib
rt.jar	E:\Program Files\jdk14\jre\lib
sunrsasign.jar	E:\Program Files\jdk14\jre\lib
dnsns.jar	E:\Program Files\jdk14\jre\lib\ext
ldapsec.jar	E:\Program Files\jdk14\jre\lib\ext
localedata.jar	E:\Program Files\jdk14\jre\lib\ext
sunjce_provider.jar	E:\Program Files\jdk14\jre\lib\ext
US_export_policy.jar	E:\Program Files\jdk14\jre\lib\security
local_policy.jar	E:\Program Files\jdk14\jre\lib\security
dt.jar	E:\Program Files\jdk14\lib
htmlconverter.jar	E:\Program Files\jdk14\lib
tools.jar	E:\Program Files\jdk14\lib
FileChooserDemo.jar	E:\Program Files\jdk14\demo\plugin\jfc\FileChooserDemo
Font2DTest.jar	E:\Program Files\jdk14\demo\plugin\jfc\Font2DTest
Java2Demo.jar	E:\Program Files\jdk14\demo\plugin\jfc\Java2D
Metalworks.jar	E:\Program Files\jdk14\demo\plugin\jfc\Metalworks
Notepad.jar	E:\Program Files\jdk14\demo\plugin\jfc\Notepad
Stylepad.jar	E:\Program Files\jdk14\demo\plugin\jfc\Stylepad

Il y en a plusieurs dizaines. Un fichier *.jar* peut s'ouvrir avec l'utilitaire *winzip*. Ouvrons le fichier *rt.jar* ci-dessus (rt=RunTime). On y trouve plusieurs centaines de fichiers *.class* dont celles appartenant au paquetage *java.util* :

Nom	Modifié	Taille	Taux	Compr...	Chemin
SimpleTextBoundary.class	07/02/2002 12:05	3 728	0%	3 728	java\text\
SpecialMapping.class	07/02/2002 12:05	415	0%	415	java\text\
StringCharacterIterator.class	07/02/2002 12:04	2 287	0%	2 287	java\text\
TextBoundaryData.class	07/02/2002 12:05	6 038	0%	6 038	java\text\
UnicodeClassMapping.class	07/02/2002 12:05	889	0%	889	java\text\
WordBreakData.class	07/02/2002 12:05	7 010	0%	7 010	java\text\
WordBreakTable.class	07/02/2002 12:05	725	0%	725	java\text\
AbstractList$1.class	07/02/2002 12:04	179	0%	179	java\util\
AbstractMap$1.class	07/02/2002 12:04	830	0%	830	java\util\
AbstractMap$2.class	07/02/2002 12:04	1 015	0%	1 015	java\util\
AbstractMap$3.class	07/02/2002 12:04	839	0%	839	java\util\
AbstractMap$4.class	07/02/2002 12:04	1 017	0%	1 017	java\util\
AbstractMap$SimpleEntry.class	07/02/2002 12:04	1 488	0%	1 488	java\util\
Arrays$ArrayList.class	07/02/2002 12:04	1 227	0%	1 227	java\util\
Collections$1.class	07/02/2002 12:04	976	0%	976	java\util\
Collections$2.class	07/02/2002 12:04	1 429	0%	1 429	java\util\
Collections$3.class	07/02/2002 12:04	1 302	0%	1 302	java\util\
Collections$4.class	07/02/2002 12:04	753	0%	753	java\util\

Une méthode simple pour gérer les paquetages est alors de les placer dans le répertoire *<jdk>\jre\lib* où *<jdk>* est le répertoire d'installation du JDK. En général, un paquetage contient plusieurs classes et il est pratique de rassembler celles-ci dans un unique fichier .jar (JAR=Java ARchive file). L'exécutable *jar.exe* se trouve dans le dossier *<jdk>\bin* :

```
E:\data\serge\JAVA\classes\paquetages\personne>dir "e:\program files\jdk14\bin\jar.exe"
07/02/2002  12:52                28 752 jar.exe
```

Une aide à l'utilisation du programme *jar* peut être obtenue en l'appelant sans paramètres :

```
E:\data\serge\JAVA\classes\paquetages\personne>"e:\program files\jdk14\bin\jar.exe"
Syntaxe : jar {ctxu}[vfm0M] [fichier-jar] [fichier-manifest] [rÛp -C] fichiers ...
Options :
    -c  crÛer un nouveau fichier d''archives
    -t  gÛnÛrer la table des matiÞres du fichier d''archives
    -x  extraire les fichiers nommÛs (ou tous les fichiers) du fichier d''archives
    -u  mettre Ó jour le fichier d''archives existant
    -v  gÛnÛrer des informations verbeuses sur la sortie standard
    -f  spÛcifier le nom du fichier d''archives
    -m  inclure les informations manifest provenant du fichier manifest spÛcifiÛ
    -0  stocker seulement ; ne pas utiliser la compression ZIP
    -M  ne pas crÛer de fichier manifest pour les entrÛes
    -i  gÛnÛrer l''index pour les fichiers jar spÛcifiÛs
    -C  passer au rÛpertoire spÛcifiÛ et inclure le fichier suivant
Si un rÛpertoire est spÛcifiÛ, il est traitÛ rÛcursivement.
Les noms des fichiers manifest et d''archives doivent Ûtre spÛcifiÛs
dans l''ordre des indicateurs ''m'' et ''f''.

Exemple 1 : pour archiver deux fichiers de classe dans le fichier d''archives classes.jar :
       jar cvf classes.jar Foo.class Bar.class
Exemple 2 : utilisez le fichier manifest existant ''monmanifest'' pour archiver tous les fichiers du

           rÛpertoire foo/ dans ''classes.jar'':
       jar cvfm classes.jar monmanifest -C foo/ .
```

Revenons à la classe *personne.class* créée précédemment dans un paquetage *istia.st* :

```
E:\data\serge\JAVA\classes\paquetages\personne>dir
06/06/2002  16:28                467 personne.java
06/06/2002  17:36                195 test.java
06/06/2002  16:04       <DIR>          istia
06/06/2002  16:06                738 test.class
06/06/2002  16:47                236 test2.java
06/06/2002  18:15                740 test2.class

E:\data\serge\JAVA\classes\paquetages\personne>dir istia
06/06/2002  16:04       <DIR>          st

E:\data\serge\JAVA\classes\paquetages\personne>dir istia\st
06/06/2002  16:28                675 personne.class
```

Créons un fichier *istia.st.jar* archivant toutes les classes du paquetage *istia.st* donc toutes les classes de l'arborescence *istia\st* ci-dessus :

```
E:\data\serge\JAVA\classes\paquetages\personne>"e:\program files\jdk14\bin\jar" cvf istia.st.jar
istia\st\*

E:\data\serge\JAVA\classes\paquetages\personne>dir
06/06/2002  16:28                467 personne.java
06/06/2002  17:36                195 test.java
06/06/2002  16:04       <DIR>          istia
06/06/2002  16:06                738 test.class
06/06/2002  16:47                236 test2.java
06/06/2002  18:15                740 test2.class
06/06/2002  18:08                874 istia.st.jar
```

Examinons avec *winzip* le contenu du fichier *istia.st.jar* :

Nom	Modifié	Taille	Taux	Compr...	Chemin
Manifest.mf	06/06/2002 18:08	68	0%	68	meta-inf\
personne.class	06/06/2002 16:28	675	42%	394	istia\st\

Plaçons le fichier *istia.st.jar* dans le répertoire *<jdk>\jre\lib\perso* :

```
E:\data\serge\JAVA\classes\paquetages\personne>dir "e:\program files\jdk14\jre\lib\perso"
06/06/2002  18:08                874 istia.st.jar
```

Maintenant compilons le programme *test2.java* puis exécutons-le :

```
E:\data\serge\JAVA\classes\paquetages\personne>javac -classpath istia.st.jar test2.java

E:\data\serge\JAVA\classes\paquetages\personne>java -classpath istia.st.jar;. test2
p1=personne(Jean,Dupont,20)
```

On remarque qu'on n'a eu qu'à citer le nom de l'archive à explorer sans avoir à dire explicitement où elle se trouvait. Tous les répertoires de l'arborescence *<jdk>\jre\lib* sont explorés pour trouver le fichier *.jar* demandé.

2.7 L'exemple IMPOTS

On reprend le calcul de l'impôt déjà étudié dans le chapitre précédent et on le traite en utilisant une classe. Rappelons le problème :

On se place dans le cas simplifié d'un contribuable n'ayant que son seul salaire à déclarer :

- on calcule le nombre de parts du salarié **nbParts=nbEnfants/2 +1** s'il n'est pas marié, **nbEnfants/2+2** s'il est marié, où *nbEnfants* est son nombre d'enfants.
- s'il a au moins trois enfants, il a une demie part de plus
- on calcule son revenu imposable **R=0.72*S** où S est son salaire annuel
- on calcule son coefficient familial **QF=R/nbParts**
- on calcule son impôt **I**. Considérons le tableau suivant :

12620.0	0	0
13190	0.05	631
15640	0.1	1290.5
24740	0.15	2072.5
31810	0.2	3309.5
39970	0.25	4900
48360	0.3	6898.5
55790	0.35	9316.5
92970	0.4	12106
127860	0.45	16754.5
151250	0.50	23147.5
172040	0.55	30710
195000	0.60	39312
0	0.65	49062

Chaque ligne a 3 champs. Pour calculer l'impôt I, on recherche la première ligne où *QF<=champ1*. Par exemple, si QF=23000 on trouvera la ligne

24740 0.15 2072.5

L'impôt I est alors égal à **0.15*R - 2072.5*nbParts**. Si QF est tel que la relation *QF<=champ1* n'est jamais vérifiée, alors ce sont les coefficients de la dernière ligne qui sont utilisés. Ici :

```
    0         0.65   49062
```

ce qui donne l'impôt I=0.65*R - 49062*nbParts.

La classe **impots** sera définie comme suit :

```
// création d'une classe impots

public class impots{

  // les données nécessaires au calcul de l'impôt
  // proviennent d'une source extérieure

  private double[] limites, coeffR, coeffN;

  // constructeur
  public impots(double[] LIMITES, double[] COEFFR, double[] COEFFN) throws Exception{
    // on vérifie que les 3 tableaux ont la même taille
    boolean OK=LIMITES.length==COEFFR.length && LIMITES.length==COEFFN.length;
    if (! OK) throw new Exception ("Les 3 tableaux fournis n'ont pas la même taille("+
                LIMITES.length+","+COEFFR.length+","+COEFFN.length+")");
    // c'est bon
    this.limites=LIMITES;
    this.coeffR=COEFFR;
    this.coeffN=COEFFN;
  }//constructeur

  // calcul de l'impôt
  public long calculer(boolean marié, int nbEnfants, int salaire){
    // calcul du nombre de parts
    double nbParts;
    if (marié) nbParts=(double)nbEnfants/2+2;
    else nbParts=(double)nbEnfants/2+1;
    if (nbEnfants>=3) nbParts+=0.5;
    // calcul revenu imposable & Quotient familial
    double revenu=0.72*salaire;
    double QF=revenu/nbParts;
    // calcul de l'impôt
    limites[limites.length-1]=QF+1;
    int i=0;
    while(QF>limites[i]) i++;
    // retour résultat
    return (long)(revenu*coeffR[i]-nbParts*coeffN[i]);
  }//calculer
}//classe
```

Un objet **impots** est créé avec les données permettant le calcul de l'impôt d'un contribuable. C'est la partie stable de l'objet. Une fois cet objet créé, on peut appeler de façon répétée sa méthode **calculer** qui calcule l'impôt du contribuable à partir de son statut marital (marié ou non), son nombre d'enfants et son salaire annuel.

Un programme de test pourait être le suivant :

```
//classes importées
// import impots;
import java.io.*;

  public class test
  {
    public static void main(String[] arg) throws IOException
    {
      // programme interactif de calcul d'impôt
      // l'utilisateur tape trois données au clavier : marié nbEnfants salaire
      // le programme affiche alors l'impôt à payer

      final String syntaxe="syntaxe : marié nbEnfants salaire\n"
             +"marié : o pour marié, n pour non marié\n"
             +"nbEnfants : nombre d'enfants\n"
             +"salaire : salaire annuel en F";

      // tableaux de données nécessaires au calcul de l'impôt
      double[] limites=new double[]
{12620,13190,15640,24740,31810,39970,48360,55790,92970,127860,151250,172040,195000,0};
      double[] coeffR=new double[] {0,0.05,0.1,0.15,0.2,0.25,0.3,0.35,0.4,0.45,0.5,0.55,0.6,0.65};
      double[] coeffN=new double[]
{0,631,1290.5,2072.5,3309.5,4900,6898.5,9316.5,12106,16754.5,23147.5,30710,39312,49062};

       // création d'un flux de lecture
       BufferedReader IN=new BufferedReader(new InputStreamReader(System.in));
      // création d'un objet impôt
      impots objImpôt=null;
      try{
        objImpôt=new impots(limites,coeffR,coeffN);
      }catch (Exception ex){
        System.err.println("L'erreur suivante s'est produite : " + ex.getMessage());
        System.exit(1);
```

```
      }//try-catch

      // boucle infinie
      while(true){
        // on demande les paramètres du calcul de l'impôt
        System.out.print("Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour
arrêter :");
        String paramètres=IN.readLine().trim();
        // qq chose à faire ?
        if(paramètres==null || paramètres.equals("")) break;
        // vérification du nombre d'arguments dans la ligne saisie
        String[] args=paramètres.split("\\s+");
        int nbParamètres=args.length;
        if (nbParamètres!=3){
          System.err.println(syntaxe);
          continue;
        }//if
        // vérification de la validité des paramètres
        // marié
        String marié=args[0].toLowerCase();
        if (! marié.equals("o") && ! marié.equals("n")){
          System.err.println(syntaxe+"\nArgument marié incorrect : tapez o ou n");
          continue;
        }//if
        // nbEnfants
        int nbEnfants=0;
        try{
          nbEnfants=Integer.parseInt(args[1]);
          if(nbEnfants<0) throw new Exception();
        }catch (Exception ex){
          System.err.println(syntaxe+"\nArgument nbEnfants incorrect : tapez un entier positif ou nul");
          continue;
        }//if
        // salaire
        int salaire=0;
        try{
          salaire=Integer.parseInt(args[2]);
          if(salaire<0) throw new Exception();
        }catch (Exception ex){
          System.err.println(syntaxe+"\nArgument salaire incorrect : tapez un entier positif ou nul");
          continue;
        }//if
        // les paramètres sont corrects - on calcule l'impôt
        System.out.println("impôt="+objImpôt.calculer(marié.equals("o"),nbEnfants,salaire)+" F");
        // contribuable suivant
      }//while
    }//main
  }//classe
```

Voici un exemple d'exécution du programme précédent :

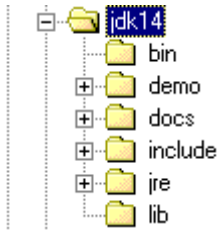
```
E:\data\serge\MSNET\c#\impots\3>java test

Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :q s d
syntaxe : marié nbEnfants salaire
marié : o pour marié, n pour non marié
nbEnfants : nombre d'enfants
salaire : salaire annuel en F
Argument marié incorrect : tapez o ou n

Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :o s d
syntaxe : marié nbEnfants salaire
marié : o pour marié, n pour non marié
nbEnfants : nombre d'enfants
salaire : salaire annuel en F
Argument nbEnfants incorrect : tapez un entier positif ou nul

Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :o 2 d
syntaxe : marié nbEnfants salaire
marié : o pour marié, n pour non marié
nbEnfants : nombre d'enfants
salaire : salaire annuel en F
Argument salaire incorrect : tapez un entier positif ou nul

Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :q s d f
syntaxe : marié nbEnfants salaire
marié : o pour marié, n pour non marié
nbEnfants : nombre d'enfants
salaire : salaire annuel en F

Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :o 2 200000
impôt=22504 F

Paramètres du calcul de l'impôt au format marié nbEnfants salaire ou rien pour arrêter :
```

3. Classes d'usage courant

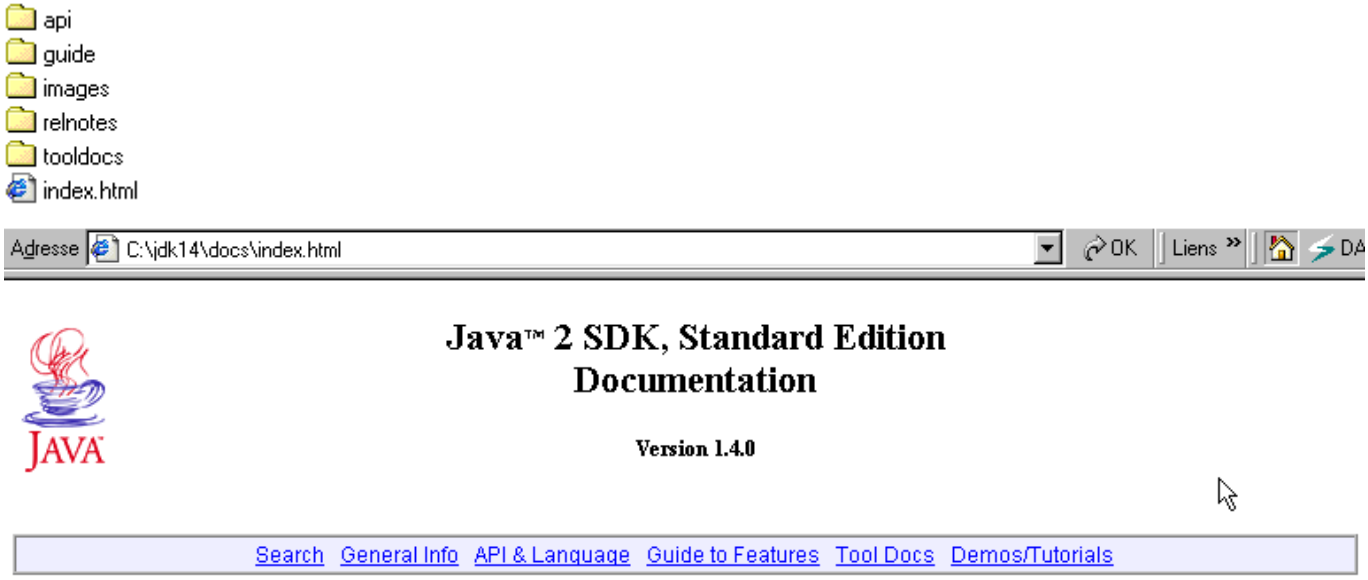
Nous présentons dans ce chapitre un certain nombre de classes Java d'usage courant. Celles-ci ont de nombreux attributs, méthodes et constructeurs. A chaque fois, nous ne présentons qu'une faible partie des classes. Le détail de celles-ci est disponible dans l'aide de Java que nous présentons maintenant.

3.1 La documentation

Si vous avez installé le JDK de Sun dans le dossier <jdk>, la documentation est disponible dans le dossier *<jdk>\docs* :

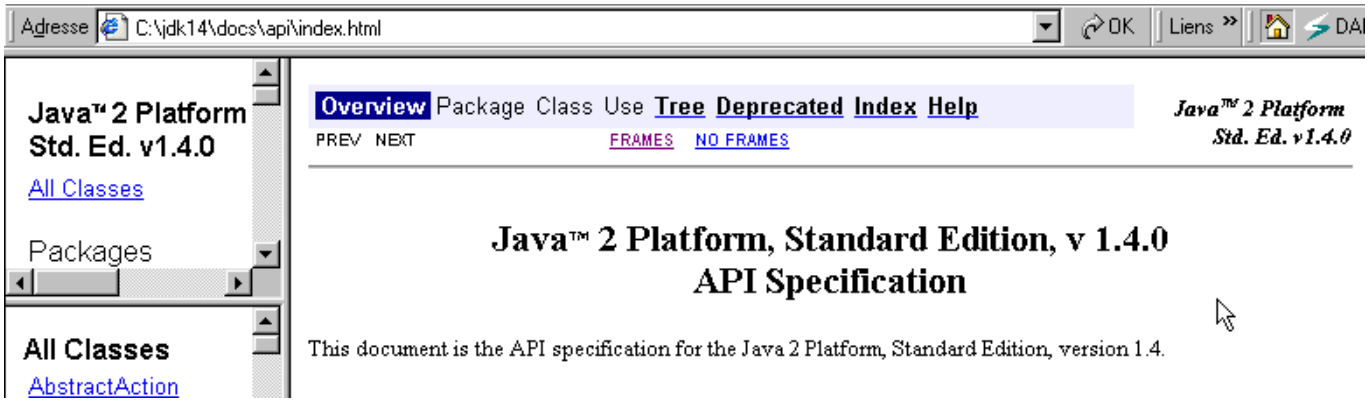


Quelquefois on a un *jdk* mais sans documentation. Celle-ci peut être trouvée sur le site de Sun *http://www.sun.com*. Dans le dossier *docs* on trouve un fichier *index.html* qui est le point de départ de l'aide du JDK :



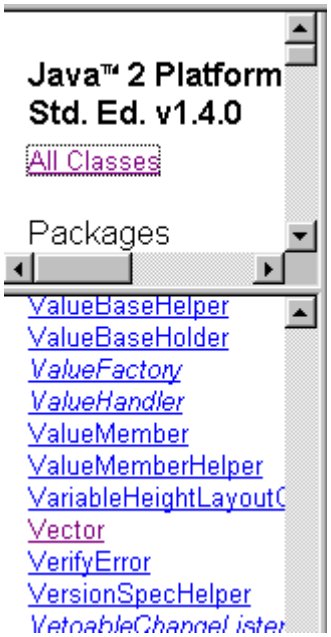
Le lien *API & Language* ci-dessus donne accès aux classes Java. Le lien *Demos/Tutorials* est particulièrement utile pour avoir des exemples de programmes Java. Suivons le lien *API & Language* :

Suivons le lien *Java 2 Platform API* :

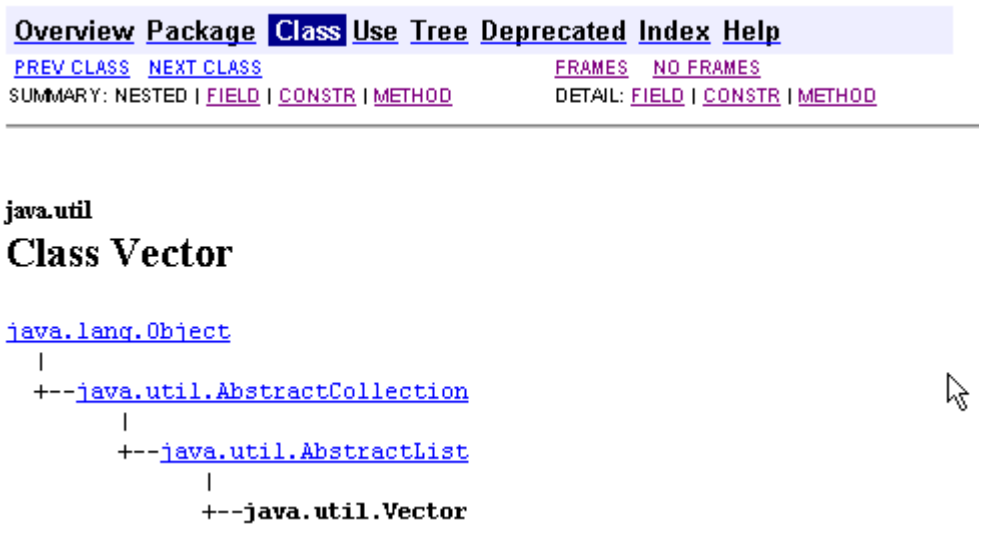


Cette page est le véritable point de départ de la documentation sur les classes. On pourra créer un raccourci dessus pour y avoir un accès rapide. L'URL est *<jdk>\docs\api\index.html*. On y trouve des liens sur les centaines de classes Java du JDK. Lorsqu'on débute, la principale difficulté est de savoir ce que font ces différentes classes. Dans un premier temps, cette aide n'a donc d'intérêt que si on connaît le nom de la classe sur laquelle on veut des informations. On peut aussi se laisser guider par les noms des classes qui indiquent normalement le rôle de la classe.

Prenons un exemple et cherchons des informations sur la classe *Vector* qui implémente un tableau dynamique. Il suffit de chercher dans la liste des classes du cadre de gauche le lien de la classe *Vector* :



et de cliquer sur le lien pour avoir la définition de la classe :



On y trouve

- la hiérarchie dans laquelle se trouve la classe, ici *java.util.Vector*
- la liste des champs (attributs) de la classe
- la liste des constructeurs
- la liste des méthodes

Par la suite, nous présentons diverses classes. Nous invitons le lecteur à systématiquement vérifier la définition complète des classes utilisées.

3.2 Les classes de test

Les exemples qui suivent utilisent parfois les classes *personne*, *enseignant*. Nous rappelons ici leur définition.

```
public class personne{
  // nom, prénom, âge
  private String prenom;
  private String nom;
  private int age;

  // constructeur 1
  public personne(String P, String N, int age){
    this.prenom=P;
    this.nom=N;
    this.age=age;
  }

  // constructeur 2
  public personne(personne P){
    this.prenom=P.prenom;
    this.nom=P.nom;
    this.age=P.age;
  }

  // toString
  public String toString(){
    return "personne("+prenom+","+nom+","+age+")";
  }

  // accesseurs
  public String getPrenom(){
    return prenom;
  }
  public String getNom(){
    return nom;
  }
  public int getAge(){
    return age;
  }

  //modifieurs
  public void setPrenom(String P){
    this.prenom=P;
  }
  public void setNom(String N){
    this.nom=N;
  }
  public void setAge(int age){
    this.age=age;
  }
}
```

La classe *enseignant* est dérivée de la classe *personne* et est définie comme suit :

```
class enseignant extends personne{
// attributs
  private int section;

  // constructeur
  public enseignant(String P, String N, int age,int section){
    super(P,N,age);
    this.section=section;
  }
  // toString
  public String toString(){
    return "etudiant("+super.toString()+","+section+")";
  }
}
```

Nous utiliserons également une classe *etudiant* dérivée de la classe *personne* et définie comme suit :

```
class etudiant extends personne{
  String numero;
```

```
  public etudiant(String P, String N, int age,String numero){
    super(P,N,age);
    this.numero=numero;
  }

  public String toString(){
    return "etudiant("+super.toString()+","+numero+")";
  }
}
```

3.3 La classe String

La classe *String* représente les chaînes de caractères. Soit *nom* une variable chaîne de caractères :

String nom;

nom est un référence d'un objet encore non initialisé. On peut l'initialiser de deux façons :

nom="cheval" ou **nom=new String("cheval")**

Les deux méthodes sont équivalentes. Si on écrit plus tard **nom="poisson",** *nom* référence alors un nouvel objet. L'ancien objet *String("cheval")* est perdu et la place mémoire qu'il occupait sera récupérée.

La classe *String* est riche d'attributs et méthodes. En voici quelques-uns :

public char charAt(int i)	donne le caractère i de la chaîne, le premier caractère ayant l'indice 0. Ainsi *String("cheval").charAt(3)* est égal à 'v'
public int compareTo(chaine2)	*chaine1.compareTo(chaine2)* compare chaine1 à chaine2 et rend 0 si chaine1=chaine2, 1 su chaine1>chaine2, -1 si chaine1<chaine2
public boolean equals(Object anObject)	*chaine1.equals(chaine2)* rend vrai si chaine1=chaine2, faux sinon
public String toLowerCase()	*chaine1.toLowerCase()* rend chaine1 en minuscules
public String toUpperCase()	*chaine1.toUpperCase()* rend chaine1 en majuscules
public String trim()	*chaine1.trim()* rend chaine1 débarassée de ses espaces de début et de fin
public String substring(int beginIndex, int endIndex)	*String("chapeau").subString(2,4)* rend la chaîne "ape"
public char[] toCharArray()	permet de mettre les caractères de la chaîne dans un tableau de caractères
int length()	nombre de caractères de la chaîne
int indexOf(String chaine2)	rend la première position de *chaine2* dans la chaîne courante ou -1 si *chaine2* n'est pas présente
int indexOf(String chaine2, int startIndex)	rend la première position de *chaine2* dans la chaîne courante ou -1 si *chaine2* n'est pas présente. La recherche commence à partir du caractère n° *startIndex*.
int lastIndexOf(String chaine2)	rend la dernière position de *chaine2* dans la chaîne courante ou -1 si *chaine2* n'est pas présente
boolean startsWith(String chaine2)	rend vrai si la chaîne courante commence par *chaine2*
boolean endsWith(String chaine2)	rend vrai si la chaîne courante finit par *chaine2*
boolean matches(String regex)	rend vrai si la chaîne courante correspond à l'expression régulière *regex*.
String[] split(String regex)	La chaîne courante est composée de champs séparés par une chaîne de caractères modélisée par l'expression régulière *regex*. La méthode *split* permet de récupérer les champs dans un tableau.
String replace(char oldChar, char newChar)	remplace dans la chaîne courante le caractère *oldChar* par le caractère *newChar*.

Voici un programme d'exemple :

```
// imports
import java.io.*;

public class string1{
  // une classe de démonstration
  public static void main(String[] args){
    String uneChaine="l'oiseau vole au-dessus des nuages";
    affiche("uneChaine="+uneChaine);
    affiche("uneChaine.Length="+uneChaine.length());
    affiche("chaine[10]="+uneChaine.charAt(10));
    affiche("uneChaine.IndexOf(\"vole\")="+uneChaine.indexOf("vole"));
    affiche("uneChaine.IndexOf(\"x\")="+uneChaine.indexOf("x"));
    affiche("uneChaine.LastIndexOf('a')="+uneChaine.lastIndexOf('a'));
    affiche("uneChaine.LastIndexOf('x')="+uneChaine.lastIndexOf('x'));
    affiche("uneChaine.substring(4,7)="+uneChaine.substring(4,7));
    affiche("uneChaine.ToUpper()="+uneChaine.toUpperCase());
    affiche("uneChaine.ToLower()="+uneChaine.toLowerCase());
    affiche("uneChaine.Replace('a','A')="+uneChaine.replace('a','A'));
    String[] champs=uneChaine.split("\\s+");
    for (int i=0;i<champs.length;i++){
      affiche("champs["+i+"]=["+champs[i]+"]");
    }//for
```

```
    affiche("(\"  abc  \").trim()=["+"  abc  ".trim()+"]");
  }//Main

  // affiche
  public static void affiche(String msg){
    // affiche msg
    System.out.println(msg);
  }//affiche
}//classe
```

et les résultats obtenus :

```
uneChaine=l'oiseau vole au-dessus des nuages
uneChaine.Length=34
chaine[10]=o
uneChaine.IndexOf("vole")=9
uneChaine.IndexOf("x")=-1
uneChaine.LastIndexOf('a')=30
uneChaine.LastIndexOf('x')=-1
uneChaine.substring(4,7)=sea
uneChaine.ToUpper()=L'OISEAU VOLE AU-DESSUS DES NUAGES
uneChaine.ToLower()=l'oiseau vole au-dessus des nuages
uneChaine.Replace('a','A')=l'oiseAu vole Au-dessus des nuAges
champs[0]=[l'oiseau]
champs[1]=[vole]
champs[2]=[au-dessus]
champs[3]=[des]
champs[4]=[nuages]
("  abc  ").trim()=[abc]
```

3.4 La classe Vector

Un vecteur est un tableau dynamique dont les éléments sont des références d'objets. C'est donc un tableau d'objets dont la taille peut varier au fil du temps ce qui n'est pas possible avec les tableaux statiques qu'on a rencontrés jusqu'ici. Voici certains champs, constructeurs ou méthodes de cette classe :

public Vector()	construit un vecteur vide
public final int size()	nombre d'élément du vecteur
public final void addElement(Object obj)	ajoute l'objet référencé par *obj* au vecteur
public final Object elementAt(int index)	référence de l'objet n° *index* du vecteur - les indices commencent à 0
public final Enumeration elements()	l'ensemble des éléments du vecteur sous forme d'énumération
public final Object firstElement()	référence du premier élément du vecteur
public final Object lastElement()	référence du dernier élément du vecteur
public final boolean isEmpty()	rend vrai si le vecteur est vide
public final void removeElementAt(int index)	enlève l'élément d'indice index
public final void removeAllElements()	vide le vecteur de tous ses éléments
public final String toString()	rend une chaîne d'identification du vecteur

Voici un programme de test :

```
// les classes importées
import java.util.*;

public class test1{

// le programme principal main - static - méthode de classe

  public static void main(String arg[]){

// la création des objets instances de classes
    personne p=new personne("Jean","Dupont",30);
    enseignant en=new enseignant("Paula","Hanson",56,27);
    etudiant et=new etudiant("Chris","Garot",22,"19980405");
    System.out.println("p="+p.toString());
    System.out.println("en="+en.toString());
    System.out.println("et="+et.toString());

// le polymorphisme
    personne p2=(personne)en;
    System.out.println("p2="+p2.toString());
    personne p3=(personne)et;
    System.out.println("p3="+p3.toString());

// un vecteur
    Vector V=new Vector();
    V.addElement(p);V.addElement(en);V.addElement(et);
    System.out.println("Taille du vecteur V = "+V.size());
```

```
    for(int i=0;i<V.size();i++){
      p2=(personne) V.elementAt(i);
      System.out.println("V["+i+"]="+p2.toString());
    }
} // fin main
}// fin classe
```

Compilons ce programme :

```
E:\data\serge\JAVA\poly juin 2002\Chapitre 3\vector>dir
10/06/2002  10:41                1 134 personne.class
10/06/2002  10:41                  619 enseignant.class
10/06/2002  10:41                  610 etudiant.class
10/06/2002  10:42                1 035 test1.java
E:\data\serge\JAVA\poly juin 2002\Chapitre 3\vector>javac test1.java

E:\data\serge\JAVA\poly juin 2002\Chapitre 3\vector>dir
10/06/2002  10:41                1 134 personne.class
10/06/2002  10:41                  619 enseignant.class
10/06/2002  10:41                  610 etudiant.class
10/06/2002  10:42                1 035 test1.java
10/06/2002  10:43                1 506 test1.class
```

Exécutons le fichier *test1.class* :

```
E:\data\serge\JAVA\poly juin 2002\Chapitre 3\vector>java test1
p=personne(Jean,Dupont,30)
en=etudiant(personne(Paula,Hanson,56),27)
et=etudiant(personne(Chris,Garot,22),19980405)
p2=etudiant(personne(Paula,Hanson,56),27)
p3=etudiant(personne(Chris,Garot,22),19980405)
Taille du vecteur V = 3
V[0]=personne(Jean,Dupont,30)
V[1]=etudiant(personne(Paula,Hanson,56),27)
V[2]=etudiant(personne(Chris,Garot,22),19980405)
```

Par la suite, nous ne répéterons plus le processus de compilation et d'exécution des programmes de tests. Il suffit de reproduire ce qui a été fait ci-dessus.

3.5 La classe ArrayList

La classe *ArrayList* est analogue à la classe *Vector*. Elle n'en diffère essentiellement que lorsque elle est utilisée simultanément par plusieurs threads d'exécution. Les méthodes de synchronisation des threads pour l'accès à un *Vector* ou un *ArrayList* diffèrent. En-dehors de ce cas, on peut utiliser indifféremment l'un ou l'autre. Voici certains champs, constructeurs ou méthodesde cette classe :

ArrayList()	construit un tableau vide
int size()	nombre d'élément du tableau
void add(Object obj)	ajoute l'objet référencé par *obj* au tableau
void add(int index,Object obj)	ajoute l'objet référencé par *obj* au tableau en position *index*
Object get(int index)	référence de l'objet n° *index* du tableau - les indices commencent à 0
boolean isEmpty()	rend vrai si le tableau est vide
void remove(int index)	enlève l'élément d'indice *index*
void clear()	vide le tableau de tous ses éléments
Object[] toArray()	met le tableau dynamique dans un tableau classique
String toString()	rend une chaîne d'identification du tableau

Voici un programme de test :

```
// les classes importées
import java.util.*;

public class test1{

// le programme principal main - static - méthode de classe

  public static void main(String arg[]){

// la création des objets instances de classes
    personne p=new personne("Jean","Dupont",30);
    enseignant en=new enseignant("Paula","Hanson",56,27);
    etudiant et=new etudiant("Chris","Garot",22,"19980405");
    System.out.println("p="+p);
    System.out.println("en="+en);
    System.out.println("et="+et);
```

```
// le polymorphisme
    personne p2=(personne)en;
    System.out.println("p2="+p2);
    personne p3=(personne)et;
    System.out.println("p3="+p3);

// un vecteur
    ArrayList personnes=new ArrayList();
    personnes.add(p);personnes.add(en);personnes.add(et);
    System.out.println("Nombre de personnes = "+personnes.size());
    for(int i=0;i<personnes.size();i++){
      p2=(personne) personnes.get(i);
      System.out.println("personnes["+i+"]="+p2);
    }
} // fin main
}// fin classe
```

Les résultats obtenus sont les mêmes que précédemment.

3.6 La classe Arrays

La classe *java.util.Arrays* donne accès à des méthodes statiques permettant différentes opérations sur les tableaux en particulier les tris et les recherches d'éléments. En voici quelques méthodes ;

static void sort(tableau)	trie *tableau* en utilisant pour cela l'ordre implicite du type de données du tableau, nombre ou chaînes de caractères.
static void sort (Object[] tableau, Comparator C)	trie trie *tableau* en utilisant pour comparer les éléments la fonction de comparaison C
static int binarySearch(tableau,élément)	rend la position de *élément* dans *tableau* ou une valeur <0 sinon. Le tableau doit être auparavant trié.
static int binarySearch(Object[] tableau,Object élément, Comparator C)	idem mais utilise la fonction de comparaison C pour comparer deux éléments du tableau.

Voici un premier exemple :

```
import java.util.*;

public class sort2 implements Comparator{

  // une classe privée interne
  private class personne{
    private String nom;
    private int age;
    public personne(String nom, int age){
      this.nom=nom; // nom de la personne
      this.age=age; // son âge
    }
    // récupérer l'âge
    public int getAge(){
      return age;
    }
    // identité de la personne
    public String toString(){
      return ("["+nom+","+age+"]");
    }
  }; // classe personne

  // constructeur
  public sort2() {
    // un tableau de personnes
    personne[] amis=new personne[]{new personne("tintin",100),new personne("milou",80),
     new personne("tournesol",40)};
    // tri du tableau de personnes
    Arrays.sort(amis,this);
    // vérification
    for(int i=0;i<3;i++)
      System.out.println(amis[i]);
  }//constructeur

  // la fonction qui compare des personnes
  public int compare(Object o1, Object o2){
    // doit rendre
    // -1 si o1 "plus petit que" o2
    // 0 si o1 "égal à" o2
    // +1 si o1 "plus grand que" o2
    personne p1=(personne)o1;
    personne p2=(personne)o2;
    int age1=p1.getAge();
    int age2=p2.getAge();
    if(age1<age2) return (-1);
      else if (age1==age2) return (0);
        else return +1;
  }//compare
```

```
  // fonction de test
  public static void main(String[] arg){
    new sort2();
  }//main

}//classe
```

Examinons ce programme. La fonction main crée un objet *sort2*. Le constructeur de la classe *sort2* est le suivant :

```
  // constructeur
  public sort2() {
    // un tableau de personnes
    personne[] amis=new personne[]{new personne("tintin",100),new personne("milou",80),
     new personne("tournesol",40)};
    // tri du tableau de personnes
    Arrays.sort(amis,this);
    // vérification
    for(int i=0;i<3;i++)
      System.out.println(amis[i]);
  }//constructeur
```

Le tableau à trier est un tableau d'objets *personne*. La classe *personne* est définie de façon privée (*private*) à l'intérieur de la classe *sort2*. La méthode statique *sort* de la classe *Arrays* ne sait pas comment trier un tableau d'objets *personne*, aussi est-on obligé ici d'utiliser la forme *void sort(Object[] obj, Comparator C)*. *Comparator* est une interface ne définissant qu'une méthode :

```
  int compare(Object o1, Object o2)
```

et qui doit rendre 0 : si o1=o2, -1 : si o1<02, +1 : si o1>o2. Dans le prototype *void sort(Object[] obj, Comparator C) l*e second argument C doit être un objet implémentant l'interface *Comparator*. Dans le constructeur *sort2*, on a choisi l'objet courant *this* :

```
    // tri du tableau de personnes
    Arrays.sort(amis,this);
```

Ceci nous oblige à faire deux choses :

1. indiquer que la classe *sort2* implémente l'interface *Comparator*
   ```
   public class sort2 implements Comparator{
   ```
2. écrire la fonction *compare* dans la classe *sort2*.

Celle-ci est la suivante :

```
  // la fonction qui compare des personnes
  public int compare(Object o1, Object o2){
    // doit rendre
    // -1 si o1 "plus petit que" o2
    // 0 si o1 "égal à" o2
    // +1 si o1 "plus grand que" o2
    personne p1=(personne)o1;
    personne p2=(personne)o2;
    int age1=p1.getAge();
    int age2=p2.getAge();
    if(age1<age2) return (-1);
      else if (age1==age2) return (0);
        else return +1;
  }//compare
```

Pour comparer deux objets *personne*, on utilise ici l'âge (on aurait pu utiliser le nom).

Les résultats de l'exécution sont les suivants :

```
[tournesol,40]
[milou,80]
[tintin,100]
```

On aurait pu procéder différemment pour mettre en œuvre l'interface *Comparator* :

```
import java.util.*;

public class sort2 {

  // une classe privée interne
  private class personne{
…….
  }; // classe personne

  // constructeur
  public sort2() {
    // un tableau de personnes
    personne[] amis=new personne[]{new personne("tintin",100),new personne("milou",80),
     new personne("tournesol",40)};
```

```
    // tri du tableau de personnes
    Arrays.sort(amis,
      new java.util.Comparator(){
        public int compare(Object o1, Object o2){
          return compare1(o1,o2);
        }//compare
      }//classe
    );
    // vérification
    for(int i=0;i<3;i++)
      System.out.println(amis[i]);
  }//constructeur

  // la fonction qui compare des personnes
  public int compare1(Object o1, Object o2){
    // doit rendre
    // -1 si o1 "plus petit que" o2
    // 0 si o1 "égal à" o2
    // +1 si o1 "plus grand que" o2
    personne p1=(personne)o1;
    personne p2=(personne)o2;
    int age1=p1.getAge();
    int age2=p2.getAge();
    if(age1<age2) return (-1);
      else if (age1==age2) return (0);
        else return +1;
  }//compare1

  // main
  public static void main(String[] arg){
    new sort2();
  }//main
}//classe
```

L'instruction de tri est devenue la suivante :

```
    // tri du tableau de personnes
    Arrays.sort(amis,
      new java.util.Comparator(){
        public int compare(Object o1, Object o2){
          return compare1(o1,o2);
        }//compare
      }//classe
    );
```

Le second paramètre de la méthode *sort* doit être un objet implémentant l'interface *Comparator*. Ici nous créons un tel objet par *new java.util.Comparator()* et le texte qui suit {….} définit la classe dont on crée un objet. On appelle cela une **classe anonyme** car elle ne porte pas de nom. Dans cette classe anonyme qui doit implémenter l'interface *Comparator*, on définit la méthode *compare* de cette interface. Celle-ci se contente d'appeler la méthode *compare1* de la classe *sort2*. On est alors ramené au cas précédent.

La classe *sort2* n'implémente plus l'interface *Comparator*. Aussi sa déclaration devient-elle :

```
public class sort2 {
```

Maintenant nous testons la méthode *binarySearch* de la classe *Arrays* sur l'exemple suivant :

```
import java.util.*;

public class sort4 {

  // une classe privée interne
  private class personne{
    // attributs
    private String nom;
    private int age;

    // constructeur
    public personne(String nom, int age){
      this.nom=nom; // nom de la personne
      this.age=age; // son âge
    }

    // récupérer le nom
    public String getNom(){
      return nom;
    }

    // récupérer l'âge
    public int getAge(){
      return age;
    }
    // identité de la personne
    public String toString(){
      return ("["+nom+","+age+"]");
    }
  }; // classe personne
```

```
  // constructeur
  public sort4() {
    // un tableau de personnes
    personne[] amis=new personne[]{new personne("tintin",100),new personne("milou",80),
     new personne("tournesol",40)};

    // des comparateurs
    java.util.Comparator comparateur1=
     new java.util.Comparator(){
       public int compare(Object o1, Object o2){
         return compare1(o1,o2);
        }//compare
      }//classe
      ;
    java.util.Comparator comparateur2=
     new java.util.Comparator(){
       public int compare(Object o1, Object o2){
         return compare2(o1,o2);
        }//compare
      }//classe
      ;

    // tri du tableau de personnes
    Arrays.sort(amis,comparateur1);
    // vérification
    for(int i=0;i<3;i++)
      System.out.println(amis[i]);
    // recherches
    cherche("milou",amis,comparateur2);
    cherche("xx",amis,comparateur2);
  }//constructeur

  // la fonction qui compare des personnes
  public int compare1(Object o1, Object o2){
    // doit rendre
    // -1 si o1 "plus petit que" o2
    // 0 si o1 "égal à" o2
    // +1 si o1 "plus grand que" o2
    personne p1=(personne)o1;
    personne p2=(personne)o2;
    int age1=p1.getAge();
    int age2=p2.getAge();
    if(age1<age2) return (-1);
      else if (age1==age2) return (0);
        else return +1;
  }//compare1

  // la fonction qui compare une personne à un nom
  public int compare2(Object o1, Object o2){
    // o1 est une personne
    // o2 est un String, le nom nom2 d'une personne
    // doit rendre
    // -1 si o1.nom "plus petit que" nom2
    // 0 si o1.nom "égal à" nom2
    // +1 si o1.nom "plus grand que" nom2
    personne p1=(personne)o1;
    String nom1=p1.getNom();
    String nom2=(String)o2;
    return nom1.compareTo(nom2);
  }//compare2

  public void cherche(String ami,personne[] amis, Comparator comparateur){
    // recherche ami dans le tableau amis
    int position=Arrays.binarySearch(amis,ami,comparateur);
    // trouvé ?
    if(position>=0)
      System.out.println(ami + " a " + amis[position].getAge() + " ans");
    else System.out.println(ami + " n'existe pas dans le tableau");
  }//cherche

  // main
  public static void main(String[] arg){
    new sort4();
  }//main
}//classe
```

Ici, nous avons procédé un peu différemment des exemples précédents. Les deux objets *Comparator* nécessaires aux méthodes *sort* et *binarySearch* ont été créés et affectés aux variables *comparateur1* et *comparateur2*.

```
    // des comparateurs
    java.util.Comparator comparateur1=
     new java.util.Comparator(){
       public int compare(Object o1, Object o2){
         return compare1(o1,o2);
        }//compare
      }//classe
      ;
    java.util.Comparator comparateur2=
     new java.util.Comparator(){
```

```
      public int compare(Object o1, Object o2){
        return compare2(o1,o2);
       }//compare
     }//classe
     ;
```

Une recherche dichotomique sur le tableau *amis* est faite deux fois dans le constructeur de *sort4* :

```
    // recherches
    cherche("milou",amis,comparateur2);
    cherche("xx",amis,comparateur2);
```

La méthode *cherche* reçoit tous les paramètres dont elle a besoin pour appeler la méthode *binarySearch* :

```
  public void cherche(String ami,personne[] amis, Comparator comparateur){
    // recherche ami dans le tableau amis
    int position=Arrays.binarySearch(amis,ami,comparateur);
    // trouvé ?
    if(position>=0)
     System.out.println(ami + " a " + amis[position].getAge() + " ans");
    else System.out.println(ami + " n'existe pas dans le tableau");
  }//cherche
```

La méthode *binarySearch* travaille avec le comparateur *comparateur2* qui lui-même fait appel à la méthode *compare2* de la classe *sort4*. La méthode *rend* la position du nom cherché dans le tableau s'il existe ou un nombre <0 sinon. La méthode *compare2* sert à comparer un objet *personne* à un nom de type *String*.

```
  // la fonction qui compare une personne à un nom
  public int compare2(Object o1, Object o2){
    // o1 est une personne
    // o2 est un String, le nom nom2 d'une personne
    // doit rendre
    // -1 si o1.nom "plus petit que" nom2
    // 0 si o1.nom "égal à" nom2
    // +1 si o1.nom "plus grand que" nom2
    personne p1=(personne)o1;
    String nom1=p1.getNom();
    String nom2=(String)o2;
    return nom1.compareTo(nom2);
  }//compare2
```

Contrairement à la méthode *sort*, la méthode *binarySearch* ne reçoit pas deux objets *personne*, mais un objet *personne* et un objet *String* dans cet ordre. Le 1er paramètre est un élément du ableau *amis*, le second le nom de la personne cherchée.

3.7 La classe Enumeration

Enumeration est une interface et non une classe. Elle a les méthodes suivantes :

public abstract boolean hasMoreElements()	rend vrai si l'énumération a encore des éléments
public abstract Object nextElement()	rend la référence de l'élément suivant de l'énumération

Comment exploite-t-on une énumération ? En général comme ceci :

```
  Enumeration e=…      // on récupère un objet enumeration
  while(e.hasMoreElements()){
    // exploiter l'élément e.nextElement()
  }
```

Voici un exemple :

```
// les classes importées
import java.util.*;

public class test1{

// le programme principal main - static - méthode de classe

  public static void main(String arg[]){

// la création des objets instances de classes
    personne p=new personne("Jean","Dupont",30);
    enseignant en=new enseignant("Paula","Hanson",56,27);
    etudiant et=new etudiant("Chris","Garot",22,"19980405");
    System.out.println("p="+p.toString());
    System.out.println("en="+en.toString());
    System.out.println("et="+et.toString());

// le polymorphisme
    personne p2=(personne)en;
```

```
    System.out.println("p2="+p2.toString());
    personne p3=(personne)et;
    System.out.println("p3="+p3.toString());

// un vecteur
    Vector V=new Vector();
    V.addElement(p);V.addElement(en);V.addElement(et);
    System.out.println("Taille du vecteur V = "+V.size());
    int i;
    for(i=0;i<V.size();i++){
      p2=(personne) V.elementAt(i);
      System.out.println("V["+i+"]="+p2.toString());
    }

// une énumération
    Enumeration E=V.elements();
    i=0;
    while(E.hasMoreElements()){
      p2=(personne) E.nextElement();
      System.out.println("V["+i+"]="+p2.toString());
      i++;
    }
   }// fin main
}//fin classe
```

On obtient les résultats suivants :

```
p=personne(Jean,Dupont,30)
en=enseignant(personne(Paula,Hanson,56),27)
et=etudiant(personne(Chris,Garot,22),19980405)
p2=enseignant(personne(Paula,Hanson,56),27)
p3=etudiant(personne(Chris,Garot,22),19980405)
Taille du vecteur V = 3
V[0]=personne(Jean,Dupont,30)
V[1]=enseignant(personne(Paula,Hanson,56),27)
V[2]=etudiant(personne(Chris,Garot,22),19980405)
V[0]=personne(Jean,Dupont,30)
V[1]=enseignant(personne(Paula,Hanson,56),27)
V[2]=etudiant(personne(Chris,Garot,22),19980405)
```

3.8 La classe Hashtable

La classe *Hashtable* permet d'implémenter un dictionnaire. On peut voir un dictionnaire comme un tableau à deux colonnes :

clé	valeur
clé1	valeur1
clé2	valeur2
..	...

Les clés sont uniques, c.a.d. qu'il ne peut y avoir deux clés identiques. Les méthodes et propriétés principales de la classe **Hashtable** sont les suivantes :

public Hashtable()	constructeur - construit un dictionnaire vide
public int size()	nombre d'éléments dans le dictionnaire - un élément étant une paire (clé, valeur)
public Object put(Object key, Object value)	ajoute la paire *(key, value)* au dictionnaire
public Object get(Object key)	récupère l'objet associé à la clé *key* ou *null* si la clé *key* n'existe pas
public boolean containsKey(Object key)	vrai si la clé *key* existe dans le dictionnaire
public boolean contains(Object value)	vrai si la valeur *value* existe dans le dictionnaire
public Enumeration keys()	rend les clés du dictionnaire sous forme d'énumération
public Object remove(Object key)	enlève la paire (clé, valeur) où clé=*key*
public String toString()	identifie le dictionnaire

Voici un exemple :

```
// les classes importées
import java.util.*;

public class test1{

// le programme principal main - static - méthode de classe

  public static void main(String arg[]){

// la création des objets instances de classes
    personne p=new personne("Jean","Dupont",30);
    enseignant en=new enseignant("Paula","Hanson",56,27);
    etudiant et=new etudiant("Chris","Garot",22,"19980405");
```

```
    System.out.println("p="+p.toString());
    System.out.println("en="+en.toString());
    System.out.println("et="+et.toString());

// le polymorphisme
    personne p2=(personne)en;
    System.out.println("p2="+p2.toString());
    personne p3=(personne)et;
    System.out.println("p3="+p3.toString());

// un dictionnaire
    Hashtable H=new Hashtable();
    H.put("personne1",p);
    H.put("personne2",en);
    H.put("personne3",et);
    Enumeration E=H.keys();
    int i=0;
    String cle;
    while(E.hasMoreElements()){
      cle=(String) E.nextElement();
      p2=(personne) H.get(cle);
      System.out.println("clé "+i+"="+cle+" valeur="+p2.toString());
      i++;
    }
  }//fin main
}//fin classe
```

Les résultats obtenus sont les suivants :

```
p=personne(Jean,Dupont,30)
en=enseignant(personne(Paula,Hanson,56),27)
et=etudiant(personne(Chris,Garot,22),19980405)
p2=enseignant(personne(Paula,Hanson,56),27)
p3=etudiant(personne(Chris,Garot,22),19980405)
clé 0=personne3 valeur=etudiant(personne(Chris,Garot,22),19980405)
clé 1=personne2 valeur=enseignant(personne(Paula,Hanson,56),27)
clé 2=personne1 valeur=personne(Jean,Dupont,30)
```

3.9 Les fichiers texte

3.9.1 Ecrire

Pour écrire dans un fichier, il faut disposer d'un flux d'écriture. On peut utiliser pour cela la classe *FileWriter*. Les constructeurs souvent utilisés sont les suivants :

`FileWriter(String fileName)`	crée le fichier de nom *fileName* - on peut ensuite écrire dedans - un éventuel fichier de même nom est écrasé
`FileWriter(String fileName, boolean append)`	idem - un éventuel fichier de même nom peut être utilisé en l'ouvrant en mode ajout (*append*=true)

La classe *FileWriter* offre un certain nombre de méthodes pour écrire dans un fichier, méthodes héritées de la classe *Writer*. Pour écrire dans un fichier texte, il est préférable d'utiliser la classe *PrintWriter* dont les constructeurs souvent utilisés sont les suivants :

PrintWriter(Writer out)	l'argument est de type *Writer*, c.a.d. un flux d'écriture (dans un fichier, sur le réseau, …)
PrintWriter(Writer out, boolean autoflush)	idem. Le second argument gère la bufferisation des lignes. Lorsqu'il est à faux (son défaut), les lignes écrites sur le fichier transitent par un buffer en mémoire. Lorsque celui-ci est plein, il est écrit dans le fichier. Cela améliore les accès disque. Ceci-dit quelquefois, ce comportement est indésirable, notamment lorsqu'on écrit sur le réseau.

Les méthodes utiles de la classe *PrintWriter* sont les suivantes :

void print(Type T)	écrit la donnée T (String, int, ….)
void println(Type T)	idem en terminant par une marque de fin de ligne
void flush()	vide le buffer si on n'est pas en mode autoflush
void close()	ferme le flux d'écriture

Voici un programme qui écrit quelques lignes dans un fichier texte :

```
// imports
import java.io.*;

public class ecrire{
  public static void main(String[] arg){
```

```
    // ouverture du fichier
    PrintWriter fic=null;
    try{
      fic=new PrintWriter(new FileWriter("out"));
    } catch (Exception e){
      Erreur(e,1);
    }
    // écriture dans le fichier
    try{
      fic.println("Jean,Dupont,27");
      fic.println("Pauline,Garcia,24");
      fic.println("Gilles,Dumond,56");
    } catch (Exception e){
      Erreur(e,3);
    }
    // fermeture du fichier
    try{
      fic.close();
    } catch (Exception e){
      Erreur(e,2);
    }
  }// fin main

  private static void Erreur(Exception e, int code){
    System.err.println("Erreur : "+e);
    System.exit(code);
  }//Erreur
}//classe
```

Le fichier *out* obtenu à l'exécution est le suivant :

```
Jean,Dupont,27
Pauline,Garcia,24
Gilles,Dumond,56
```

3.9.2 Lire

Pour lire le contenu d'un fichier, il faut disposer d'un flux de lecture associé au fichier. On peut utiliser pour cela la classe *FileReader* et le constructeur suivant :

FileReader(String nomFichier) ouvre un flux de lecture à partir du fichier indiqué. Lance une exception si l'opération échoue.

La classe *FileReader* possède un certain nombre de méthodes pour lire dans un fichier, méthodes héritées de la classe *Reader*. Pour lire des lignes de texte dans un fichier texte, il est préférable d'utiliser la classe *BufferedReader* avec le constructeur suivant :

BufferedReader(Reader in) ouvre un flux de lecture bufferisé à partir d'un flux d'entrée *in*. Ce flux de type *Reader* peut provenir du clavier, d'un fichier, du réseau, ...

Les méthodes utiles de la classe *BufferedReader* sont les suivantes :

int read()	lit un caractère
String readLine()	lit une ligne de texte
int read(char[] buffer, int offset, int taille)	lit *taille* caractères dans le fichier et les met dans le tableau *buffer* à partir de la position *offset*.
void close()	ferme le flux de lecture

Voici un programme qui lit le contenu du fichier créé précédemment :

```
// classes importées
import java.util.*;
import java.io.*;

public class lire{
  public static void main(String[] arg){
    personne p=null;
    // ouverture du fichier
    BufferedReader IN=null;
    try{
      IN=new BufferedReader(new FileReader("out"));
    } catch (Exception e){
      Erreur(e,1);
    }
    // données
    String ligne=null;
    String[] champs=null;
    String prenom=null;
    String nom=null;
    int age=0;
```

```
    // gestion des éventuelles erreurs
    try{
      while((ligne=IN.readLine())!=null){
        champs=ligne.split(",");
        prenom=champs[0];
        nom=champs[1];
        age=Integer.parseInt(champs[2]);
        System.out.println(""+new personne(prenom,nom,age));
      }// fin while
    } catch (Exception e){
      Erreur(e,2);
    }

    // fermeture fichier
    try{
      IN.close();
    } catch (Exception e){
      Erreur(e,3);
    }
  }// fin main

  // Erreur
  public static void Erreur(Exception e, int code){
    System.err.println("Erreur : "+e);
    System.exit(code);
  }

}// fin classe
```

L'exécution du programme donne les résultats suivants :

```
personne(Jean,Dupont,27)
personne(Pauline,Garcia,24)
personne(Gilles,Dumond,56)
```

3.9.3 Sauvegarde d'un objet personne

Nous appliquons ce que nous venons de voir afin de fournir à la classe *personne* une méthode permettant de sauver dans un fichier les attributs d'une personne. On rajoute la méthode *sauveAttributs* dans la définition de la classe *personne* :

```
  // ------------------------------
  // sauvegarde dans fichier texte
  // ------------------------------
  public void sauveAttributs(PrintWriter P){
    P.println(""+this);
  }
```

Précédant la définition de la classe *personne*, on n'oubliera pas d'importer le paquetage *java.io* :

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

La méthode *sauveAttributs* reçoit en unique paramètre le flux *PrintWriter* dans lequel elle doit écrire. Un programme de test pourrait être le suivant :

```
// imports
import java.io.*;
// import personne;

public class sauver{
  public static void main(String[] arg){
    // ouverture du fichier
    PrintWriter fic=null;
    try{
      fic=new PrintWriter(new FileWriter("out"));
    } catch (Exception e){
      Erreur(e,1);
    }
    // écriture dans le fichier
    try{
      new personne("Jean","Dupont",27).sauveAttributs(fic);
      new personne("Pauline","Garcia",24).sauveAttributs(fic);
      new personne("Gilles","Dumond",56).sauveAttributs(fic);
    } catch (Exception e){
      Erreur(e,3);
    }
    // fermeture du fichier
    try{
      fic.close();
    } catch (Exception e){
      Erreur(e,2);
    }
  }// fin main
```

```
  // Erreur
  private static void Erreur(Exception e, int code){
    System.err.println("Erreur : "+e);
    System.exit(code);
  }//Erreur
}//classe
```

Compilons et exécutons ce programme :

```
E:\data\serge\JAVA\poly juin 2002\Chapitre 3\sauveAttributs>javac sauver.java

E:\data\serge\JAVA\poly juin 2002\Chapitre 3\sauveAttributs>dir
10/06/2002  10:52               1 352 personne.class
10/06/2002  10:53                 842 sauver.java
10/06/2002  10:53               1 258 sauver.class

E:\data\serge\JAVA\poly juin 2002\Chapitre 3\sauveAttributs>java sauver

E:\data\serge\JAVA\poly juin 2002\Chapitre 3\sauveAttributs>dir
10/06/2002  10:52               1 352 personne.class
10/06/2002  10:53                 842 sauver.java
10/06/2002  10:53               1 258 sauver.class
10/06/2002  10:53                  83 out

E:\data\serge\JAVA\poly juin 2002\Chapitre 3\sauveAttributs>more out
personne(Jean,Dupont,27)
personne(Pauline,Garcia,24)
personne(Gilles,Dumond,56)
```

3.10 Les fichiers binaires

3.10.1 La classe RandomAccessFile

La classe *RandomAccessFile* permet de gérer des fichiers binaires notamment ceux à structure fixe comme on en connaît en langage C/C++. Voici quelques méthodes et constructeurs utiles :

RandomAccessFile(String nomFichier, String mode)	constructeur - ouvre le fichier indiqué dans le mode indiqué. Celui-ci prend ses valeurs dans : r : ouverture en lecture rw : ouverture en lecture et écriture
void writeTTT(TTT valeur)	écrit valeur dans le fichier. TTT représente le type de valeur. La représentation mémoire de valeur est écrite telle-quelle dans le fichier. On trouve ainsi writeBoolean, writeByte, writeInt, writeDouble, writeLong, writeFloat,... Pour écrire une chaîne, on utilise *writeBytes(String chaine).*
TTT readTTT()	lit et rend une valeur de type TTT. On trouve ainsi readBoolean, readByte, readInt, readDouble, readLong, readFloat,... La méthode *read()* lit un octet.
long length()	taille du fichier en octets
long getFilePointer()	position courante du pointeur de fichier
void seek(long pos)	positionne le curseur de fichier à l'octet pos

3.10.2 La classe article

Tous les exemples qui vont suivre utiliseront la classe *article* suivante :

```
  // la structure article
  private static class article{
    // on définit la structure
    public String code;
    public String nom;
    public double prix;
    public int stockActuel;
    public int stockMinimum;
  }//classe article
```

La classe java *article* ci-dessus sera l'équivalent de la structure *article* suivante du C

```
struct article{
  char code[4];
  char nom[20];
```

```
  double prix;
  int stockActuel;
  int stockMinimum;
}//structure
```

Ainsi nous limiterons le code à 4 caractères et le nom à 20.

3.10.3 Ecrire un enregistrement

Le programme suivant écrit un article dans un fichier appelé "data" :

```
// classes importées
import java.io.*;

public class test1{

// teste l'écriture d'une structure (au sens du C) dans un fichier binaire

  // la structure article
  private static class article{
    // on définit la structure
    public String code;
    public String nom;
    public double prix;
    public int stockActuel;
    public int stockMinimum;
  }//classe article

  public static void main(String arg[]){

    // on définit le fichier binaire dans lequel seront rangés les articles
    RandomAccessFile fic=null;

    // on définit un article
    article art=new article();
    art.code="a100";
    art.nom="velo";
    art.prix=1000.80;
    art.stockActuel=100;
    art.stockMinimum=10;

    // on définit le fichier
    try{
        fic=new RandomAccessFile("data","rw");
    } catch (Exception E){
        erreur("Impossible d'ouvrir le fichier data",1);
    }//try-catch

    // on écrit
    try{
      ecrire(fic,art);
    } catch (IOException E){
        erreur("Erreur lors de l'écriture de l'enregistrement",2);
    }//try-catch

    // c'est fini
    try{
      fic.close();
    } catch (Exception E){
      erreur("Impossible de fermer le fichier data",2);
    }//try-catch
  }//main

  // méthode d'écriture
  public static void ecrire(RandomAccessFile fic, article art) throws IOException{
      // code
    fic.writeBytes(art.code);
       // le nom limité à 20 caractères
    art.nom=art.nom.trim();
    int l=art.nom.length();
    int nbBlancs=20-l;
    if(nbBlancs>0){
      String blancs="";
      for(int i=0;i<nbBlancs;i++) blancs+=" ";
      art.nom+=blancs;
    } else art.nom=art.nom.substring(0,20);
    fic.writeBytes(art.nom);
       // le prix
    fic.writeDouble(art.prix);
       // les stocks
    fic.writeInt(art.stockActuel);
    fic.writeInt(art.stockMinimum);
  }// fin écrire

  // ------------------------erreur
  public static void erreur(String msg, int exitCode){
      System.err.println(msg);
      System.exit(exitCode);
```

```
  }// fin erreur
}// fin class
```

C'est le programme suivant qui nous permet de vérifier que l'exécution s'est correctement faite.

3.10.4 Lire un enregistrement

```
// classes importées
import java.io.*;

public class test2{

// teste l'écriture d'une structure (au sens du C) dans un fichier binaire

  // la structure article
  private static class article{
    // on définit la structure
    public String code;
    public String nom;
    public double prix;
    public int stockActuel;
    public int stockMinimum;
  }//classe article

  public static void main(String arg[]){

    // on définit le fichier binaire dans lequel seront rangés les articles
    RandomAccessFile fic=null;

    // on ouvre le fichier en lecture
    try{
      fic=new RandomAccessFile("data","r");
    } catch (Exception E){
      erreur("Impossible d'ouvrir le fichier data",1);
    }//try-catch

    // on lit l'article unique du fichier
     article art=new article();
    try{
      lire(fic,art);
    } catch (IOException E){
      erreur("Erreur lors de la lecture de l'enregistrement",2);
    }//try-catch

    // on affiche l'enregistrement lu
    affiche(art);

    // c'est fini
    try{
      fic.close();
    } catch (Exception E){
      erreur("Impossible de fermer le fichier data",2);
    }//try-catch
  }// fin main

  // méthode de lecture
  public static void lire(RandomAccessFile fic, article art) throws IOException{
    // lecture code
    art.code="";
    for(int i=0;i<4;i++) art.code+=(char)fic.readByte();
     // nom
    art.nom="";
    for(int i=0;i<20;i++) art.nom+=(char)fic.readByte();
    art.nom=art.nom.trim();
     // prix
    art.prix=fic.readDouble();
     // stocks
    art.stockActuel=fic.readInt();
    art.stockMinimum=fic.readInt();
  }// fin écrire

  // ---------------------affiche
  public static void affiche(article art){
    System.out.println("code : "+art.code);
    System.out.println("nom : "+art.nom);
    System.out.println("prix : "+art.prix);
    System.out.println("Stock actuel : "+art.stockActuel);
    System.out.println("Stock minimum : "+art.stockMinimum);
  }// fin affiche

  // ------------------------erreur
  public static void erreur(String msg, int exitCode){
        System.err.println(msg);
        System.exit(exitCode);
  }// fin erreur
}// fin class
```

Les résultats d'exécution sont les suivants :

```
E:\data\serge\JAVA\random>java test2
code : a100
nom : velo
prix : 1000.8
Stock actuel : 100
Stock minimum : 10
```

On récupère bien l'enregistrement qui avait été écrit par le programme d'écriture.

3.10.5 Conversion texte --> binaire

Le programme suivant est une extension du programme d'écriture d'un enregistrement. On écrit maintenant plusieurs enregistrements dans un fichier binaire appelé *data.bin*. Les données sont prises dans le fichier *data.txt* suivant :

```
E:\data\serge\JAVA\random>more data.txt
a100:velo:1000:100:10
b100:pompe:65:6:2
c100:arc:867:10:5
d100:fleches - lot de 6:450:12:8
e100:jouet:10:2:3
```

```
// classes importées
import java.io.*;
import java.util.*;

public class test3{

// fichier texte --> fichier binaire

  // la structure article
  private static class article{
    // on définit la structure
    public String code;
    public String nom;
    public double prix;
    public int stockActuel;
    public int stockMinimum;
  }//classe article

  public static void main(String arg[]){

    // on définit le fichier binaire dans lequel seront rangés les articles
    RandomAccessFile dataBin=null;
    try{
        dataBin=new RandomAccessFile("data.bin","rw");
    } catch (Exception E){
       erreur("Impossible d'ouvrir le fichier data.bin",1);
    }

    // les données sont prises dans un fichier texte
    BufferedReader dataTxt=null;
    try{
      dataTxt=new BufferedReader(new FileReader("data.txt"));
    } catch (IOException E){
      erreur("Impossible d'ouvrir le fichier data.txt",2);
    }

    // fichier .txt --> fichier .bin
    String ligne=null;
    String[] champs=null;
    int numLigne=0;
    String champ=null;
    article art=new article();  // article à créer
    try{
      while((ligne=dataTxt.readLine())!=null){
        // une ligne de +
        numLigne++;
         // décomposition en champs
         champs=ligne.split(":");
         // il faut 5 champs
        if(champs.length!=5)
          erreur("Ligne "+numLigne+" erronée dans data.txt",3);
         //code
        art.code=champs[0];
        if(art.code.length()!=4)
          erreur("Code erroné en ligne "+numLigne+" du fichier data.txt",12);
         // nom, prénom
        art.nom=champs[1];
         // prix
        try{
          art.prix=Double.parseDouble(champs[2]);
        } catch (Exception E){
          erreur("Prix erroné en ligne "+numLigne+" du fichier data.txt",4);
```

```
        }
         // stock actuel
        try{
          art.stockActuel=Integer.parseInt(champs[3]);
        } catch (Exception E){
          erreur("Stock actuel erroné en ligne "+ numLigne + " du fichier data.txt",5);
        }
         // stock actuel
        try{
          art.stockActuel=Integer.parseInt(champs[3]);
        } catch (Exception E){
          erreur("Stock actuel erroné en ligne "+ numLigne + " du fichier data.txt",5);
        }
        // on écrit l'enregistrement
        try{
          ecrire(dataBin,art);
        } catch (IOException E){
          erreur("Erreur lors de l'écriture de l'enregistrement "+numLigne,7);
        }
        // on passe à la ligne suivante
      }// fin while
    } catch (IOException E){
      erreur("Erreur lors de la lecture du fichier data.txt après la ligne "+numLigne,8);
    }
    // c'est fini
    try{
         dataBin.close();
    } catch (Exception E){
         erreur("Impossible de fermer le fichier data.bin",10);
    }
    try{
         dataTxt.close();
    } catch (Exception E){
         erreur("Impossible de fermer le fichier data.txt",11);
    }
  }// fin main

  // méthode d'écriture
  public static void ecrire(RandomAccessFile fic, article art) throws IOException{
     // code
    fic.writeBytes(art.code);
       // le nom limité à 20 caractères
    art.nom=art.nom.trim();
    int l=art.nom.length();
    int nbBlancs=20-l;
    if(nbBlancs>0){
      String blancs="";
      for(int i=0;i<nbBlancs;i++) blancs+=" ";
      art.nom+=blancs;
    } else art.nom=art.nom.substring(0,20);
    fic.writeBytes(art.nom);
       // le prix
    fic.writeDouble(art.prix);
       // les stocks
    fic.writeInt(art.stockActuel);
    fic.writeInt(art.stockMinimum);
  }// fin écrire

  // ------------------------erreur
  public static void erreur(String msg, int exitCode){
       System.err.println(msg);
       System.exit(exitCode);
  }// fin erreur

}// fin class
```

C'est le programme suivant qui permet de vérifier que celui-ci a correctement fonctionné.

3.10.6 Conversion binaire --> texte

Le programme suivant lit le contenu du fichier binaire *data.bin* créé précédemment et met son contenu dans le fichier texte *data.text*. Si tout va bien, le fichier *data.text* doit être identique au fichier d'origine *data.txt*.

```
// classes importées
import java.io.*;
import java.util.*;

public class test5{

// fichier texte --> fichier binaire

  // la structure article
  private static class article{
    // on définit la structure
    public String code;
    public String nom;
    public double prix;
```

```
    public int stockActuel;
    public int stockMinimum;
  }//classe article

  // main
  public static void main(String arg[]){

    // on définit le fichier binaire dans lequel seront rangés les articles
    RandomAccessFile dataBin=null;
    try{
        dataBin=new RandomAccessFile("data.bin","r");
    } catch (Exception E){
        erreur("Impossible d'ouvrir le fichier data.bin en lecture",1);
    }

    // les données sont écrites dans un fichier texte
     PrintWriter dataTxt=null;
    try{
      dataTxt=new PrintWriter(new FileWriter("data.text"));
    } catch (IOException E){
      erreur("Impossible d'ouvrir le fichier data.text en écriture",2);
    }

    // fichier .bin --> fichier .text
     article art=new article();  // article à créer

    // on exploite le fichier binaire
    int numRecord=0;
    long l=0;  // taille du fichier
    try{
      l=dataBin.length();
    } catch (IOException e){
      erreur("Erreur lors du calcul de la longueur du fichier data.bin",2);
    }
    long pos=0; // position courante dans le fichier
    try{
      pos=dataBin.getFilePointer();
    } catch (IOException e){
      erreur("Erreur lors de la lecture de la position courante dans data.bin",2);
    }

     // tant qu'on n'a pas dépassé la fin du fichier
    while(pos<l){
      // lire enregistrement courant et l'exploiter
      numRecord++;
      try{
        lire(dataBin,art);
      } catch (Exception e){
        erreur("Erreur lors de la lecture de l'enregistrement "+numRecord,2);
      }
      affiche(art);

      // écriture de la ligne de texte correspondante dans dataTxt

  dataTxt.println(art.code.trim()+":"+art.nom.trim()+":"+art.prix+":"+art.stockActuel+":"+art.stockMinimum
);

      // on continue ?
      try{
        pos=dataBin.getFilePointer();
      } catch (IOException e){
        erreur("Erreur lors de la lecture de la position courante dans data.bin",2);
      }
    }// fin while

    // c'est fini
    try{
        dataBin.close();
    } catch (Exception E){
        erreur("Impossible de fermer le fichier data.bin",2);
    }

    try{
      dataTxt.close();
    } catch (Exception E){
        erreur("Impossible de fermer le fichier data.text",2);
    }

  }// fin main

  // méthode de lecture
  public static void lire(RandomAccessFile fic, article art) throws IOException{
    // lecture code
    art.code="";
    for(int i=0;i<4;i++) art.code+=(char)fic.readByte();
     // nom
    art.nom="";
    for(int i=0;i<20;i++) art.nom+=(char)fic.readByte();
    art.nom=art.nom.trim();
     // prix
    art.prix=fic.readDouble();
     // stocks
```

```
    art.stockActuel=fic.readInt();
    art.stockMinimum=fic.readInt();
  }// fin écrire

  // ---------------------affiche
  public static void affiche(article art){
    System.out.println("code : "+art.code);
    System.out.println("nom : "+art.nom);
    System.out.println("prix : "+art.prix);
    System.out.println("Stock actuel : "+art.stockActuel);
    System.out.println("Stock minimum : "+art.stockMinimum);
  }// fin affiche

  // ------------------------erreur
  public static void erreur(String msg, int exitCode){
        System.err.println(msg);
        System.exit(exitCode);
  }// fin erreur
}// fin class
```

Voici un exemple d'exécution :

```
E:\data\serge\JAVA\random>java test5
code : a100
nom : velo
prix : 1000.0
Stock actuel : 100
Stock minimum : 0
code : b100
nom : pompe
prix : 65.0
Stock actuel : 6
Stock minimum : 0
code : c100
nom : arc
prix : 867.0
Stock actuel : 10
Stock minimum : 0
code : d100
nom : fleches - lot de 6
prix : 450.0
Stock actuel : 12
Stock minimum : 0
code : e100
nom : jouet
prix : 10.0
Stock actuel : 2
Stock minimum : 0

E:\data\serge\JAVA\random>more data.text
a100:velo:1000.0:100:0
b100:pompe:65.0:6:0
c100:arc:867.0:10:0
d100:fleches - lot de 6:450.0:12:0
e100:jouet:10.0:2:0
```

3.10.7 Accès direct aux enregistrements

Ce dernier programme illustre la possibilité d'accéder directement aux enregistrements d'un fichier binaire. Il affiche l'enregistrement du fichier *data.bin* dont on lui passe le n° en paramètre, le 1er enregistrement portant le n° 1.

```
// classes importées
import java.io.*;
import java.util.*;

public class test6{

// fichier texte --> fichier binaire

  // la structure article
  private static class article{
    // on définit la structure
    public String code;
    public String nom;
    public double prix;
    public int stockActuel;
    public int stockMinimum;
  }//classe article

  // main
  public static void main(String[] args){
```

```
    // on vérifie les arguments
    int nbArguments=args.length;
    String syntaxe="syntaxe : pg numéro_de_fiche";
    if(nbArguments!=1)
      erreur(syntaxe,20);
     // vérification n° de fiche
    int numRecord=0;
    try{
      numRecord=Integer.parseInt(args[0]);
    } catch(Exception e){
      erreur(syntaxe+"\nNuméro de fiche incorrect",21);
    }

    // on ouvre le fichier binaire en lecture
     RandomAccessFile dataBin=null;
    try{
      dataBin=new RandomAccessFile("data.bin","r");
    } catch (Exception E){
      erreur("Impossible d'ouvrir le fichier data.bin en lecture",1);
    }

    // on se positionne sur la fiche désirée
    try{
      dataBin.seek((numRecord-1)*40);
    } catch (Exception e){
      erreur("La fiche "+numRecord+" n'existe pas",23);
    }

    // on la lit
     article art=new article();
    try{
      lire(dataBin,art);
    } catch (Exception e){
      erreur("Erreur lors de la lecture de l'enregistrement "+numRecord,2);
    }

    // on l'affiche
    affiche(art);

    // c'est fini
    try{
      dataBin.close();
    } catch (Exception E){
      erreur("Impossible de fermer le fichier data.bin",2);
    }//try-catch

  }// fin main

  // méthode de lecture
  public static void lire(RandomAccessFile fic, article art) throws IOException{
    // lecture code
    art.code="";
    for(int i=0;i<4;i++) art.code+=(char)fic.readByte();
     // nom
    art.nom="";
    for(int i=0;i<20;i++) art.nom+=(char)fic.readByte();
    art.nom=art.nom.trim();
     // prix
    art.prix=fic.readDouble();
     // stocks
    art.stockActuel=fic.readInt();
    art.stockMinimum=fic.readInt();
  }// fin écrire

  // ---------------------affiche
  public static void affiche(article art){
    System.out.println("code : "+art.code);
    System.out.println("nom : "+art.nom);
    System.out.println("prix : "+art.prix);
    System.out.println("Stock actuel : "+art.stockActuel);
    System.out.println("Stock minimum : "+art.stockMinimum);
  }// fin affiche

  // ------------------------erreur
  public static void erreur(String msg, int exitCode){
        System.err.println(msg);
        System.exit(exitCode);
  }// fin erreur
}// fin class
```

Voici des exemples d'exécution :

```
E:\data\serge\JAVA\random>java test6 2
code : b100
nom : pompe
prix : 65.0
Stock actuel : 6
Stock minimum : 0
```

```
E:\data\serge\JAVA\random>java.bat test6 20
Erreur lors de la lecture de l'enregistrement 20
```

3.11 Utiliser les expression régulières

3.11.1 Le paquetage java.util.regex

Le paquetage *java.util.regex* permet l'utilisation d'expression régulières. Celles-ci permettent de tester le format d'une chaîne de caractères. Ainsi on peut vérifier qu'une chaîne représentant une date est bien au format jj/mm/aa. On utilise pour cela un modèle et on compare la chaîne à ce modèle. Ainsi dans cet exemple, j m et a doivent être des chiffres. Le modèle d'un format de date valide est alors **"\d\d/\d\d/\d\d"** où le symbole **\d** désigne un chiffre. Les symboles utilisables dans un modèle sont les suivants (documentation Microsoft) :

Caractère	**Description**
****	Marque le caractère suivant comme caractère spécial ou littéral. Par exemple, "n" correspond au caractère "n". "\n" correspond à un caractère de nouvelle ligne. La séquence "\\" correspond à "\", tandis que "\(" correspond à "(".
^	Correspond au début de la saisie.
$	Correspond à la fin de la saisie.
*****	Correspond au caractère précédent zéro fois ou plusieurs fois. Ainsi, "zo*" correspond à "z" ou à "zoo".
+	Correspond au caractère précédent une ou plusieurs fois. Ainsi, "zo+" correspond à "zoo", mais pas à "z".
?	Correspond au caractère précédent zéro ou une fois. Par exemple, "a?ve?" correspond à "ve" dans "lever".
.	Correspond à tout caractère unique, sauf le caractère de nouvelle ligne.
(modèle**)**	Recherche le *modèle* et mémorise la correspondance. La sous-chaîne correspondante peut être extraite de la collection **Matches** obtenue, à l'aide d'Item **[0]...[n]**. Pour trouver des correspondances avec des caractères entre parenthèses ( ), utilisez "\(" ou "\)".
x|y	Correspond soit à *x* soit à *y*. Par exemple, "z|foot" correspond à "z" ou à "foot". "(z|f)oo" correspond à "zoo" ou à "foo".
{*n*}	*n* est un nombre entier non négatif. Correspond exactement à *n* fois le caractère. Par exemple, "o{2}" ne correspond pas à "o" dans "Bob," mais aux deux premiers "o" dans "fooooot".
{*n*,}	*n* est un entier non négatif. Correspond à au moins *n* fois le caractère. Par exemple, "o{2,}" ne correspond pas à "o" dans "Bob", mais à tous les "o" dans "fooooot". "o{1,}" équivaut à "o+" et "o{0,}" équivaut à "o*".
{*n***,***m***}**	*m* et *n* sont des entiers non négatifs. Correspond à au moins *n* et à au plus *m* fois le caractère. Par exemple, "o{1,3}" correspond aux trois premiers "o" dans "foooooot" et "o{0,1}" équivaut à "o?".
[*xyz***]**	Jeu de caractères. Correspond à l'un des caractères indiqués. Par exemple, "[abc]" correspond à "a" dans "plat".
[^*xyz***]**	Jeu de caractères négatif. Correspond à tout caractère non indiqué. Par exemple, "[^abc]" correspond à "p" dans "plat".
[*a-z***]**	Plage de caractères. Correspond à tout caractère dans la série spécifiée. Par exemple, "[a-z]" correspond à tout caractère alphabétique minuscule compris entre "a" et "z".
[^*m-z***]**	Plage de caractères négative. Correspond à tout caractère ne se trouvant pas dans la série spécifiée. Par exemple, "[^m-z]" correspond à tout caractère ne se trouvant pas entre "m" et "z".
\b	Correspond à une limite représentant un mot, autrement dit, à la position entre un mot et un espace. Par exemple, "er\b" correspond à "er" dans "lever", mais pas à "er" dans "verbe".
\B	Correspond à une limite ne représentant pas un mot. "en*t\B" correspond à "ent" dans "bien entendu".
\d	Correspond à un caractère représentant un chiffre. Équivaut à [0-9].
\D	Correspond à un caractère ne représentant pas un chiffre. Équivaut à [^0-9].
\f	Correspond à un caractère de saut de page.
\n	Correspond à un caractère de nouvelle ligne.
\r	Correspond à un caractère de retour chariot.
\s	Correspond à tout espace blanc, y compris l'espace, la tabulation, le saut de page, etc. Équivaut à

"[\f\n\r\t\v]".

\S	Correspond à tout caractère d'espace non blanc. Équivaut à "[^ \f\n\r\t\v]".
\t	Correspond à un caractère de tabulation.
\v	Correspond à un caractère de tabulation verticale.
\w	Correspond à tout caractère représentant un mot et incluant un trait de soulignement. Équivaut à "[A-Za-z0-9_]".
\W	Correspond à tout caractère ne représentant pas un mot. Équivaut à "[^A-Za-z0-9_]".
\num	Correspond à *num*, où *num* est un entier positif. Fait référence aux correspondances mémorisées. Par exemple, "(.)\1" correspond à deux caractères identiques consécutifs.
n	Correspond à *n*, où *n* est une valeur d'échappement octale. Les valeurs d'échappement octales doivent comprendre 1, 2 ou 3 chiffres. Par exemple, "\11" et "\011" correspondent tous les deux à un caractère de tabulation. "\0011" équivaut à "\001" & "1". Les valeurs d'échappement octales ne doivent pas excéder 256. Si c'était le cas, seuls les deux premiers chiffres seraient pris en compte dans l'expression. Permet d'utiliser les codes ASCII dans des expressions régulières.
\x*n*	Correspond à *n*, où *n* est une valeur d'échappement hexadécimale. Les valeurs d'échappement hexadécimales doivent comprendre deux chiffres obligatoirement. Par exemple, "\x41" correspond à "A". "\x041" équivaut à "\x04" & "1". Permet d'utiliser les codes ASCII dans des expressions régulières.

Un élément dans un modèle peut être présent en 1 ou plusieurs exemplaires. Considérons quelques exemples autour du symbole **\d** qui représente 1 chiffre :

modèle	signification
\d	un chiffre
\d?	0 ou 1 chiffre
\d*	0 ou davantage de chiffres
\d+	1 ou davantage de chiffres
\d{2}	2 chiffres
\d{3,}	au moins 3 chiffres
\d{5,7}	entre 5 et 7 chiffres

Imaginons maintenant le modèle capable de décrire le format attendu pour une chaîne de caractères :

chaîne recherchée	modèle
une date au format jj/mm/aa	\d{2}/\d{2}/\d{2}
une heure au format hh:mm:ss	\d{2}:\d{2}:\d{2}
un nombre entier non signé	\d+
un suite d'espaces éventuellement vide	\s*
un nombre entier non signé qui peut être précédé ou suivi d'espaces	\s*\d+\s*
un nombre entier qui peut être signé et précédé ou suivi d'espaces	\s*[+|-]?\s*\d+\s*
un nombre réel non signé qui peut être précédé ou suivi d'espaces	\s*\d+(.\d*)?\s*
un nombre réel qui peut être signé et précédé ou suivi d'espaces	\s*[+|]?\s*\d+(.\d*)?\s*
une chaîne contenant le mot juste	\bjuste\b

On peut préciser où on recherche le modèle dans la chaîne :

modèle	signification
^modèle	le modèle commence la chaîne
modèle$	le modèle finit la chaîne
^modèle$	le modèle commence et finit la chaîne
modèle	le modèle est cherché partout dans la chaîne en commençant par le début de celle-ci.

chaîne recherchée	modèle
une chaîne se terminant par un point d'exclamation	!$
une chaîne se terminant par un point	\.$
une chaîne commençant par la séquence //	^//
une chaîne ne comportant qu'un mot éventuellement suivi ou précédé d'espaces	^\s*\w+\s*$
une chaîne ne comportant deux mot éventuellement suivis ou précédés d'espaces	^\s*\w+\s*\w+\s*$
une chaîne contenant le mot secret	\bsecret\b

Les sous-ensembles d'un modèle peuvent être "récupérés". Ainsi non seulement, on peut vérifier qu'une chaîne correspond à un modèle particulier mais on peut récupérer dans cette chaîne les éléments correspondant aux sous-ensembles du modèle qui **ont été entourés de parenthèses**. Ainsi si on analyse une chaîne contenant une date jj/mm/aa et si on veut de plus récupérer les éléments jj, mm, aa de cette date on utilisera le modèle (\d\d)/(\d\d)/(\d\d).

3.11.2 Vérifier qu'une chaîne correspond à un modèle donné

La classe *Pattern* permet de vérifier qu'une chaîne correspond à un modèle donné. On utilise pour cela la méthode statique

```
boolean Matches(String modèle, String chaine)
```

avec : *modèle* : le modèle à vérifier, *chaine* : la chaîne à comparer au modèle. Le résultat est le booléen *true* si chaine correspond à modèle, *false* sinon.

Voici un exemple :

```
import java.io.*;
import java.util.regex.*;

// gestion d'expression régulières
public class regex1 {
  public static void main(String[] args){
    // une expression régulière modèle
    String modèle1="^\\s*\\d+\\s*$";
    // comparer un exemplaire au modèle
    String exemplaire1="  123  ";
    if (Pattern.matches(modèle1,exemplaire1)){
      affiche("["+exemplaire1 + "] correspond au modèle ["+modèle1+"]");
    }else{
      affiche("["+exemplaire1 + "] ne correspond pas au modèle ["+modèle1+"]");
    }//if
    String exemplaire2="  123a  ";
    if (Pattern.matches(modèle1,exemplaire2)){
      affiche("["+exemplaire2 + "] correspond au modèle ["+modèle1+"]");
    }else{
      affiche("["+exemplaire2 + "] ne correspond pas au modèle ["+modèle1+"]");
    }//if
  }//main

  public static void affiche(String msg){
    System.out.println(msg);
  }//affiche
}//classe
```

et les résultats d'exécution :

```
[  123  ] correspond au modèle [^\s*\d+\s*$]
[  123a  ] ne correspond pas au modèle [^\s*\d+\s*$]
```

On notera que dans le modèle "^\s*\d+\s*$" le caractère \ doit être doublé à cause de l'interprétation particulière que fait Java de ce caractère. On écrit donc : String modèle1="^\\s*\\d+\\s*$";

3.11.3 Trouver tous les éléments d'une chaîne correspondant à un modèle

Considérons le modèle "\d+" et la chaîne " 123 456 789 ". On retrouve le modèle dans trois endroits différents de la chaîne. Les classes **Pattern** et **Matcher** permettent de récupérer les différentes occurrences d'un modèle dans une chaîne. La classe *Pattern* est la classe gérant les expressions régulières. Une expression régulière utilisée plus d'une fois nécessite d'être "compilée". Cela accélère les recherches du modèle dans les chaînes. La méthode statique **compile** fait ce travail :

```
public static Pattern compile(String regex)
```

Elle prend pour paramètre la chaîne du modèle et rend un objet *Pattern*. Pour comparer le modèle d'un objet *Pattern* à une chaîne de caractères on utilise la classe *Matcher*. Celle-ci permet la comparaison d'un modèle à une chaîne de caractères. A partir d'un objet *Pattern*, il est possible d'obtenir un objet de type *Matcher* avec la méthode *matcher* :

```
public Matcher matcher(CharSequence input)
```

input est la chaîne de caractères qui doit être comparée au modèle.

Ainsi pour comparer le modèle "\d+" à la chaîne " 123 456 789 ", on pourra créer un objet *Matcher* de la façon suivante :

```
Pattern regex=Pattern.compile("\\d+");
Matcher résultats=regex.matcher("  123  456  789  ");
```

A partir de l'objet *résultats* précédent, on va pouvoir récupérer les différentes occurrences du modèle dans la chaîne. Pour cela, on utilise les méthodes *suivantes* de la classe *Matcher* :

```
public boolean find()
public String group()
public int start()
public Matcher reset()
```

La méthode *find* recherche dans la chaîne explorée la première occurence du modèle. Un second appel à *find* recherchera l'occurrence suivante. Et ainsi de suite. La méthode rend *true* si elle trouve le modèle, *false* sinon. La portion de chaine correspondant à la dernière occurence trouvée par *find* est obtenue avec la méthode *group* et sa position avec la méthode *start*. Ainsi, si on poursuit l'exemple précédent et qu'on veuille afficher toutes les occurences du modèle "\d+" dans la chaîne " 123 456 789 " on écrira :

```
while(résultats.find()){
  System.out.println("séquence " + résultats.group() + " trouvée en position " + résultats.start());
    }//while
```

La méthode *reset* permet de réinitialiser l'objet *Matcher* sur le début de la chaîne comparée au modèle. Ainsi la méthode *find* trouvera ensuite de nouveau la première occurence du modèle.

Voici un exemple complet :

```
import java.io.*;
import java.util.regex.*;

// gestion d'expression régulières
public class regex2 {
  public static void main(String[] args){
    // plusieurs occurrences du modèle dans l'exemplaire
    String modèle2="\\d+";
    Pattern regex2=Pattern.compile(modèle2);
    String exemplaire3="  123  456 789";
    // recherche des occurrences du modèle dans l'exemplaire
    Matcher matcher2=regex2.matcher(exemplaire3);
    while(matcher2.find()){
      affiche("séquence " + matcher2.group() + " trouvée en position " + matcher2.start());
    }//while
  }//Main

  public static void affiche(String msg){
    System.out.println(msg);
  }//affiche
}//classe
```

Les résultats de l'exécution :

```
Modèle=[\d+],exemplaire=[  123  456  789 ]
Il y a 3 occurrences du modèle dans l'exemplaire
123 en position 2
456 en position 7
789 en position 12
```

3.11.4 Récupérer des parties d'un modèle

Des sous-ensembles d'un modèle peuvent être "récupérés". Ainsi non seulement, on peut vérifier qu'une chaîne correspond à un modèle particulier mais on peut récupérer dans cette chaîne les éléments correspondant aux sous-ensembles du modèle qui **ont été entourés de parenthèses**. Ainsi si on analyse une chaîne contenant une date jj/mm/aa et si on veut de plus récupérer les éléments jj, mm, aa de cette date on utilisera le modèle (\d\d)/(\d\d)/(\d\d).

Examinons l'exemple suivant :

```
import java.io.*;
import java.util.regex.*;

// gestion d'expression régulières
public class regex3 {
  public static void main(String[] args){
    // capture d'éléments dans le modèle
    String modèle3="(\\d\\d):(\\d\\d):(\\d\\d)";
    Pattern regex3=Pattern.compile(modèle3);
    String exemplaire4="Il est 18:05:49";
    // vérification modèle
```

```
    Matcher résultat=regex3.matcher(exemplaire4);
    if (résultat.find()){
      // l'exemplaire correspond au modèle
      affiche("L'exemplaire ["+exemplaire4+"] correspond au modèle ["+modèle3+"]");
      // on affiche les groupes
      for (int i=0;i<=résultat.groupCount();i++){
        affiche("groupes["+i+"]=["+résultat.group(i)+"] en position "+résultat.start(i));
      }//for
      }else{
        // l'exemplaire ne correspond pas au modèle
        affiche("L'exemplaire["+exemplaire4+" ne correspond pas au modèle ["+modèle3+"]");
      }
    }//Main

    public static void affiche(String msg){
      System.out.println(msg);
    }//affiche
}//classe
```

L'exécution de ce programme produit les résultats suivants :

```
L'exemplaire [Il est 18:05:49] correspond au modèle [(\d\d):(\d\d):(\d\d)]
groupes[0]=[18:05:49] en position 7
groupes[1]=[18] en position 7
groupes[2]=[05] en position 10
groupes[3]=[49] en position 13
```

La nouveauté se trouve dans la partie de code suivante :

```
    // vérification modèle
    Matcher résultat=regex3.matcher(exemplaire4);
    if (résultat.find()){
      // l'exemplaire correspond au modèle
      affiche("L'exemplaire ["+exemplaire4+"] correspond au modèle ["+modèle3+"]");
      // on affiche les groupes
      for (int i=0;i<=résultat.groupCount();i++){
        affiche("groupes["+i+"]=["+résultat.group(i)+"] en position "+résultat.start(i));
      }//for
      }else{
        // l'exemplaire ne correspond pas au modèle
        affiche("L'exemplaire["+exemplaire4+" ne correspond pas au modèle ["+modèle3+"]");
      }
```

La chaîne *exemplaire4* est comparée au modèle *regex3* au travers de la méthode *find*. Une occurence du modèle *regex3* est alors trouvée dans la chaîne *exemplaire4*. Si le modèle comprend des sous-ensembles entourés de parenthèses ceux-ci sont disponibles grâce à diverses méthodes de la classe *Matcher* :

```
public int groupCount()
public String group(int group)
public int start(int group)
```

La méthode *groupCount* donne le nombre de sous-ensembles trouvés dans le modèle et *group(i)* le sous-ensemble n° *i*. Celui est trouvé dans la chaîne à une position donnée par *start(i)*. Ainsi dans l'exemple :

```
L'exemplaire [Il est 18:05:49] correspond au modèle [(\d\d):(\d\d):(\d\d)]
groupes[0]=[18:05:49] en position 7
groupes[1]=[18] en position 7
groupes[2]=[05] en position 10
groupes[3]=[49] en position 13
```

Le premier appel à la méthode *find* va trouver la chaîne 18:05:49 et automatiquement créer les trois sous-ensembles définis par les parenthèses du modèle, respectivement 18, 05 et 49.

3.11.5 Un programme d'apprentissage

Trouver l'expression régulière qui nous permet de vérifier qu'une chaîne correspond à un certain modèle est parfois un véritable défi. Le programme suivant permet de s'entraîner. Il demande un modèle et une chaîne et indique alors si la chaîne correspond ou non au modèle.

```
import java.io.*;
import java.util.regex.*;

// gestion d'expression régulières
public class regex4 {
  public static void main(String[] args){

    // données
    String modèle=null,chaine=null;
```

```
    Pattern regex=null;
    BufferedReader IN=null;
    Matcher résultats=null;
    int nbOccurrences=0;

    // gestion des erreurs
    try{
      // on demande à l'utilisateur les modèles et les exemplaires à comparer à celui-ci
      while(true){
        // flux d'entrée
        IN=new BufferedReader(new InputStreamReader(System.in));
        // on demande le modèle
        System.out.print("Tapez le modèle à tester ou fin pour arrêter :");
        modèle=IN.readLine();
        // fini ?
        if(modèle.trim().toLowerCase().equals("fin")) break;
        // on crée l'expression régulière
        regex=Pattern.compile(modèle);
        // on demande à l'utilisateur les exemplaires à comparer au modèle
        while(true){
          System.out.print("Tapez la chaîne à comparer au modèle ["+modèle+"] ou fin pour arrêter :");
          chaine=IN.readLine();
          // fini ?
          if(chaine.trim().toLowerCase().equals("fin")) break;
          // on crée l'objet matcher
          résultats=regex.matcher(chaine);
          // on recherche les occurrences du modèle
          nbOccurrences=0;
          while(résultats.find()){
            // on a une occurrence
            nbOccurrences++;
            // on l'affiche
            System.out.println("J'ai trouvé la correspondance ["+résultats.group()
            +"] en position "+résultats.start());
            // affichage des sous-éléments
            if(résultats.groupCount()!=1){
              for(int j=1;j<=résultats.groupCount();j++){
                System.out.println("\tsous-élément ["+résultats.group(j)+"] en position "+
                  résultats.start(j));
              }//for j
            }//if
            // chaîne suivante
          }//while(résultats.find())
          // a-t-on trouvé au moins une occurrence ?
          if(nbOccurrences==0){
            System.out.println("Je n'ai pas trouvé de correspondance au modèle ["+modèle+"]");
          }//if
          // modèle suivant
        }//while(true)
      }//while(true)
    }catch(Exception ex){
      // erreur
      System.err.println("Erreur : "+ex.getMessage());
      // fin avec erreur
      System.exit(1);
    }//try-catch
    // fin
    System.exit(0);
  }//Main
}//classe
```

Voici un exemple d'exécution :

```
Tapez le modèle à tester ou fin pour arrêter :\d+
Tapez la chaîne à comparer au modèle [\d+] ou fin pour arrêter :123 456 789
J'ai trouvé la correspondance [123] en position 0
J'ai trouvé la correspondance [456] en position 4
J'ai trouvé la correspondance [789] en position 8
Tapez la chaîne à comparer au modèle [\d+] ou fin pour arrêter :fin

Tapez le modèle à tester ou fin pour arrêter :(\d\d):(\d\d)
Tapez la chaîne à comparer au modèle [(\d\d):(\d\d)] ou fin pour arrêter :14:15
abcd 17:18 xyzt
J'ai trouvé la correspondance [14:15] en position 0
        sous-élément [14] en position 0
        sous-élément [15] en position 3
J'ai trouvé la correspondance [17:18] en position 11
        sous-élément [17] en position 11
        sous-élément [18] en position 14
Tapez la chaîne à comparer au modèle [(\d\d):(\d\d)] ou fin pour arrêter :fin

Tapez le modèle à tester ou fin pour arrêter :^\s*\d+\s*$
Tapez la chaîne à comparer au modèle [^\s*\d+\s*$] ou fin pour arrêter :  1456
J'ai trouvé la correspondance [  1456] en position 0
Tapez la chaîne à comparer au modèle [^\s*\d+\s*$] ou fin pour arrêter :fin

Tapez le modèle à tester ou fin pour arrêter :^\s*(\d+)\s*$
```

```
Tapez la chaîne à comparer au modèle [^\s*(\d+)\s*$] ou fin pour arrêter :1456
J'ai trouvé la correspondance [1456] en position 0
        sous-élément [1456] en position 0
Tapez la chaîne à comparer au modèle [^\s*(\d+)\s*$] ou fin pour arrêter :abcd 1
456
Je n'ai pas trouvé de correspondances
Tapez la chaîne à comparer au modèle [^\s*(\d+)\s*$] ou fin pour arrêter :fin

Tapez le modèle à tester ou fin pour arrêter :fin
```

3.11.6 La méthode split de la classe Pattern

Considérons une chaîne de caractères composée de champs séparés par une chaîne séparatrice s'exprimant à l'aide d'un fonction régulière. Par exemple si les champs sont séparés par le caractère , précédé ou suivi d'un nombre quelconque d'espaces, l'expression régulière modélisant la chaîne séparatrice des champs serait "\s*,\s*". La méthode s*plit* de la classe *Pattern* nous permet de récupérer les champs dans un tableau :

```
public String[] split(CharSequence input)
```

La chaîne *input* est décomposée en champs, ceux-ci étant séparés par un séparateur correspondant au modèle de l'objet *Pattern* courant. Pour récupérer les champs d'une ligne dont le séparateur de champs est la virgule précédée ou suivie d'un nombre quelconque d'espaces, on écrira :

```
    // une ligne
    String ligne="abc  ,, def  , ghi";
    // un modèle
    Pattern modèle=Pattern.compile("\\s*,\\s*");
    // décomposition de ligne en champs
    String[] champs=modèle.split(ligne);
```

On peut obtenir le même résultat avec la méthode *split* de la classe *String* :

```
public String[] split(String regex)
```

Voici un programme test :

```
import java.io.*;
import java.util.regex.*;

// gestion d'expression régulières
public class split1 {
  public static void main(String[] args){
    // une ligne
    String ligne="abc  ,, def  , ghi";
    // un modèle
    Pattern modèle=Pattern.compile("\\s*,\\s*");
    // décomposition de ligne en champs
    String[] champs=modèle.split(ligne);
    // affichage
    for(int i=0;i<champs.length;i++){
      System.out.println("champs["+i+"]=["+champs[i]+"]");
    }//for
    // une autre façon de faire
    champs=ligne.split("\\s*,\\s*");
    // affichage
    for(int i=0;i<champs.length;i++){
      System.out.println("champs["+i+"]=["+champs[i]+"]");
    }//for
  }//Main
}//classe
```

Les résultats d'exécution :

```
champs[0]=[abc]
champs[1]=[]
champs[2]=[def]
champs[3]=[ghi]
champs[0]=[abc]
champs[1]=[]
champs[2]=[def]
champs[3]=[ghi]
```

3.12 Exercices

3.12.1 Exercice 1

Sous Unix, les programmes sont souvent appelés de la façon suivante :

```
$ pg -o1 v1 v2 ... -o2 v3 v4 …
```

où `-oi` représente une option et `vi` une valeur associée à cette option. On désire créer une classe **options** qui permettrait d'analyser la chaîne d'arguments `-o1 v1 v2 ... -o2 v3 v4 …` afin de construire les entités suivantes :

optionsValides	dictionnaire (Hashtable) dont les clés sont les options `oi` valides. La valeur associée à la clé `oi` est un vecteur (Vector) dont les éléments sont les valeurs `v1 v2 …` associées à l'option `-oi`
optionsInvalides	dictionnaire (Hashtable) dont les clés sont les options `oi` invalides. La valeur associée à la clé `oi` est un vecteur (Vector) dont les éléments sont les valeurs `v1 v2 …` associées à l'option `-oi`
optionsSans	chaîne (String) donnant la liste des valeurs `vi` non associées à une option
erreur	entier valant 0 s'il n'y a pas d'erreurs dans la ligne des arguments, autre chose sinon : 1 : il y a des paramètres d'appel invalides 2 : il y a des options invalides 4 : il y a des valeurs non associées à des options S'il y a plusieurs types d'erreurs, ces valeurs se cumulent.

Un objet **options** pourra être construit de 4 façons différentes :

public options (String arguments, String optionsAcceptables)

arguments	la ligne d'arguments `-o1 v1 v2 ... -o2 v3 v4 …` à analyser
optionsAcceptables	la liste des options `oi` acceptables

Exemple d'appel : *options opt=new options("-u u1 u2 u3 -g g1 g2 -x","-u -g");*

Ici, les deux arguments sont des chaînes de caractères. On acceptera les cas où ces chaînes ont été découpées en mots mis dans un tableau de chaînes de caractères. Cela nécessite trois autres constructeurs :

public options (String[] arguments, String optionsAcceptables)
public options (String arguments, String[] optionsAcceptables)
public options (String[] arguments, String[] optionsAcceptables)

La classe options présentera l'interface suivante (accesseurs) :

public Hashtable getOptionsValides()
rend la référence du tableau **optionsValides** construit lors de la création de l'objet **options**
public Hashtable getOptionsInvalides()
rend la référence du tableau **optionsInvalides** construit lors de la création de l'objet **options**
public String getOptionsSans()
rend la référence de la chaîne **optionsSans** construite lors de la création de l'objet **options**
public int getErreur()
rend la valeur de l'attribut **erreur** construite lors de la création de l'objet **options**
public String toString()
s'il n'y a pas d'erreur, affiche les valeurs des attributs **optionsValides, optionsInvalides, optionsSans** ou sinon affiche le numéro de l'erreur.

Voici un programme d'exemple :

```
import java.io.*;
//import options;

public class test1{


  public static void main (String[] arg){
```

```
    // ouverture du flux d'entrée
    String ligne;
    BufferedReader IN=null;
    try{
      IN=new BufferedReader(new InputStreamReader(System.in));
    } catch (Exception e){
      affiche(e);
      System.exit(1);
    }
    // lecture des arguments du constructeur options(String, string)
    String options=null;
    String optionsAcceptables=null;
    while(true){
      System.out.print("Options : ");
      try{
        options=IN.readLine();
      } catch (Exception e){
        affiche(e);
        System.exit(2);
      }
      if(options.length()==0) break;
      System.out.print("Options acceptables: ");
      try{
        optionsAcceptables=IN.readLine();
      } catch (Exception e){
        affiche(e);
        System.exit(2);
      }
      System.out.println(new options(options,optionsAcceptables));
    }// fin while
  }//fin main


public static void affiche(Exception e){
    System.err.println("Erreur : "+e);
  }

}//fin classe
```

Quelques résultats :

```
C:\Serge\java\options>java test1
Options : 1 2 3 -a a1 a2 -b b1 -c c1 c2 c3 -b b2 b3
Options acceptables: -a -b
Erreur 6
Options valides :(-b,b1,b2,b3)  (-a,a1,a2)
Options invalides : (-c,c1,c2,c3)
Sans options : 1 2 3
```

3.12.2 Exercice 2

On désire créer une classe **stringtovector** permettant de transférer le contenu d'un objet *String* dans un objet *Vector*. Cette classe serait dérivée de la classe *Vector* :

```
class stringtovector extends Vector
```

et aurait le constructeur suivant :

```
  private void stringtovector(String S, String separateur, int[] tChampsVoulus,
      boolean strict){

    // crée un vecteur avec les champs de la chaîne S
    // celle-ci est constituée de champs séparés par separateur
    // si séparateur=null, la chaîne ne forme qu'un seul champ
    // seuls les champs dont les index sont dans le tableau tChampsVoulus
    // sont désirés. Les index commencent à 1
    // si tChampsvoulus=null ou de taille nulle, on prend tous les champs
    // si strict=vrai, tous les champs désirés doivent être présents
```

La classe aurait l'attribut privé suivant :

```
private int erreur;
```

Cet attribut est positionné par le constructeur précédent avec les valeurs suivantes :

0 : la construction s'est bien passée
4 : certains champs demandés sont absents alors que strict=true

La classe aura également deux méthodes :

```
  public int getErreur()
```

qui rend la valeur de l'attribut privé *erreur*.

```
  public String identite(){
```

qui affiche la valeur de l'objet sous la forme (erreur, élément 1, élément 2, …) où *éléments i* sont les éléments du vecteur construit à partir de la chaîne.

Un programme de test pourrait être le suivant :

```
import java.io.*;
//import stringtovector;

public class essai2{
  public static void main(String arg[]){
    int[] T1={1,3};
    System.out.println(new stringtovector("a : b : c :d:e",":",T1,true).identite());
    int[] T2={1,3,7};
    System.out.println(new stringtovector("a : b : c :d:e",":",T2,true).identite());
    int [] T3={1,4,7};
    System.out.println(new stringtovector("a : b : c :d:e",":",T3,false).identite());
    System.out.println(new stringtovector("a : b : c :d:e","",T1,false).identite());
    System.out.println(new stringtovector("a : b : c :d:e",null,T1,false).identite());
    int[] T4={1};
    System.out.println(new stringtovector("a : b : c :d:e","!",T4,true).identite());
    int[] T5=null;
    System.out.println(new stringtovector("a : b : c :d:e",":",T5,true).identite());
    System.out.println(new stringtovector("a : b : c :d:e",null,T5,true).identite());
    int[] T6=new int[0];
    System.out.println(new stringtovector("a : b : c :d:e","",T6,true).identite());
    int[] T7={1,3,4};
    System.out.println(new stringtovector("a  b  c d e"," ",T6,true).identite());
  }
}
```

Les résultats :

```
(0,a,c)
(4,a,c)
(0,a,d)
(0,a : b : c :d:e)
(0,a : b : c :d:e)
(0,a : b : c :d:e)
(0,a,b,c,d,e)
(0,a : b : c :d:e)
(0,a : b : c :d:e)
(0,a,b,c,d,e)
```

Quelques conseils :

1. Pour découper la chaîne S en champs, utiliser la méthode *split* de la classe *String*.
2. Mettre les champs de S dans un dictionnaire D indexé par le numéro du champ
3. Récupérer dans le dictionnaire D les seuls champs ayant leur clé (index) dans le tableau *tChampsVoulus*.

3.12.3 Exercice 3

On désire ajouter à la classe **stringtovector** le constructeur suivant :

```
public stringtovector(String S, String separateur, String sChampsVoulus,boolean strict){

    // crée un vecteur avec les champs de la chaîne S
    // celle-ci est constituée de champs séparés par separateur
    // si séparateur=null, la chaîne ne forme qu'un seul champ
    // seuls les champs dont les index sont dans sChampsVoulus sont désirés
    // les index commencent à 1
    // si sChampsvoulus=null ou "", on prend tous les champs
    // si strict=vrai, tous les champs désirés doivent être présents
```

La liste des champs désirés est donc dans une chaîne (String) au lieu d'un tableau d'entiers (int[]). L'attribut privé *erreur* de la classe peut se voir attribuer une nouvelle valeur :

2 : la chaînes des index des champs désirés est incorrecte

Voici un programme exemple :

```
import java.io.*;
//import stringtovector;

public class essai1{
  public static void main(String arg[]){
    String champs=null;
    System.out.println(new stringtovector("a: b :c :d:e ",":","1 3",true).identite());
    System.out.println(new stringtovector("a: b :c :d:e ",":","1 3 7",true).identite());
    System.out.println(new stringtovector("a: b :c :d:e ",":","1 4 7",false).identite());
    System.out.println(new stringtovector("a: b :c :d:e ","","1 3",false).identite());
    System.out.println(new stringtovector("a: b :c :d:e ",null,"1 3",false).identite());
    System.out.println(new stringtovector("a: b :c :d:e ","!","1",true).identite());
    System.out.println(new stringtovector("a: b :c :d:e ",":","",true).identite());
    System.out.println(new stringtovector("a: b :c :d:e ",":",champs,true).identite());
    System.out.println(new stringtovector("a: b :c :d:e ",null,champs,true).identite());
    System.out.println(new stringtovector("a: b :c :d:e ","","",true).identite());
    System.out.println(new stringtovector("a: b :c :d:e ",":","1 !",true).identite());
    System.out.println(new stringtovector("a b c d       e ","  ","1 3",false).identite());
  }
}
```

Quelques résultats :

```
(0,a,c)
(4,a,c)
(0,a,d)
(0,a: b :c :d:e)
(0,a: b :c :d:e)
(0,a: b :c :d:e)
(0,a,b,c,d,e)
(0,a,b,c,d,e)
(0,a: b :c :d:e)
(0,a: b :c :d:e)
(2)
(0,a,c)
```

Quelques conseils :

1. Il faut se ramener au cas du constructeur précédent en transférant les champs de la chaîne *sChampsVoulus* dans un tableau d'entiers. Pour cela, découper *sChampsVoulus* en champs avec un objet *StringTokenizer* dont l'attribut *countTokens* donnera le nombre de champs obtenus. On peut alors créer un tableau d'entiers de la bonne dimension et le remplir avec les champs obtenus.
2. Pour savoir si un champ est entier, utiliser la méthode **Integer.parseInt** pour transformer le champ en entier et gérer l'exception qui sera générée lorsque cette conversion sera impossible.

3.12.4 Exercice 4

On désire créer une classe **filetovector** permettant de transférer le contenu d'un fichier texte dans un objet *Vector*. Cette classe serait dérivée de la classe *Vector* :

```
class filetovector extends Vector
```

et aurait le constructeur suivant :

```
  // ---------------------- constructeur
  public filetovector(String nomFichier, String separateur, int [] tChampsVoulus,boolean strict, String
tagCommentaire){

    // crée un vecteur avec les lignes du fichier texte nomFichier
    // les lignes sont faites de champs séparés par separateur
    // si séparateur=null, la ligne ne forme qu'un seul champ
    // seuls les champs dont les index sont dans tChampsVoulus sont désirés
    // les index commencent à 1
    // si tChampsvoulus=null ou vide, on prend tous les champs
    // si strict=vrai, tous les champs désirés doivent être présents
    // si ce n'est pas le cas, la ligne n'est pas mémorisée et son index
    // est placé dans le vecteur lignesErronees
    // les lignes blanches sont ignorées
    // ainsi que les lignes commençant par tagCommentaire si tagCommentaire != null
```

La classe aurait les attributs privés suivants :

```
    private int erreur=0;
    private Vector lignesErronees=null;
```

L'attribut *erreur* est positionné par le constructeur précédent avec les valeurs suivantes :

0 : la construction s'est bien passée

1 : le fichier à exploiter n'a pas pu être ouvert
4 : certains champs demandés sont absents alors que strict=true
8 : il y a eu une erreur d'E/S lors de l'exploitation du fichier

L'attribut *lignesErronees* est un vecteur dont les éléments sont les numéros des lignes erronées sous forme de chaîne de caractères. Une ligne est erronée si elle ne peut fournir les champs demandés alors que strict=true.

La classe aura également deux méthodes :

```
  public int getErreur()
```

qui rend la valeur de l'attribut privé *erreur*.

```
  public String identite(){
```

qui affiche la valeur de l'objet sous la forme (erreur, élément 1 élément 2 …,(l1,l2,…)) où éléments i sont les éléments du vecteur construit à partir du fichier et li les numéros des lignes erronées.

Voici un exemple de test :

```
import java.io.*;
//import filetovector;

public class test2{
  public static void main(String arg[]){
    int[] T1={1,3};
    System.out.println(new filetovector("data.txt",":",T1,false,"#").identite());
    System.out.println(new filetovector("data.txt",":",T1,true,"#").identite());
    System.out.println(new filetovector("data.txt","",T1,false,"#").identite());
    System.out.println(new filetovector("data.txt",null,T1,false,"#").identite());
    int[] T2=null;
    System.out.println(new filetovector("data.txt",":",T2,false,"#").identite());
    System.out.println(new filetovector("data.txt",":",T2,false,"").identite());
    int[] T3=new int[0];
    System.out.println(new filetovector("data.txt",":",T3,false,null).identite());
  }
}
```

Les résultats d'exécution :

```
[0,(0,a,c) (0,1,3) (0,azerty,cvf) (0,s)]
[4,(0,a,c) (0,1,3) (0,azerty,cvf),[5]]
[0,(0,a:b:c:d:e) (0,1 :2 : 3:  4: 5) (0,azerty : 1 : cvf : fff: qqqq) (0,s)]
[0,(0,a:b:c:d:e) (0,1 :2 : 3:  4: 5) (0,azerty : 1 : cvf : fff: qqqq) (0,s)]
[0,(0,a,b,c,d,e) (0,1,2,3,4,5) (0,azerty,1,cvf,fff,qqqq) (0,s)]
[0,(0,a,b,c,d,e) (0,1,2,3,4,5) (0,# commentaire) (0,azerty,1,cvf,fff,qqqq) (0,s)]
[0,(0,a,b,c,d,e) (0,1,2,3,4,5) (0,# commentaire) (0,azerty,1,cvf,fff,qqqq) (0,s)]
```

Quelques conseils

1. Le fichier texte est traité ligne par ligne. La ligne est découpée en champs grâce à la classe **stringtovector** étudiée précédemment.
2. Les éléments du vecteur consitué à partir du fichier texte sont donc des objets de type **stringtovector**.
3. La méthode **identite** de **filetovector** pourra s'appuyer sur la méthode **stringtovector.identite()** pour afficher ses éléments ainsi que sur la méthode **Vector.toString()** pour afficher les numéros des éventuelles lignes erronées.

3.12.5 Exercice 5

On désire ajouter à la classe **filetovector** le constructeur suivant :

```
  public filetovector(String nomFichier, String separateur, String sChampsVoulus,
      boolean strict, String tagCommentaire){

    // crée un vecteur avec les lignes du fichier texte nomFichier
    // les lignes sont faites de champs séparés par separateur
    // si séparateur=null, la ligne ne forme qu'un seul champ
    // seuls les champs dont les index sont dans tChampsVoulus sont désirés
    // les index commencent   1
    // si sChampsvoulus=null ou vide, on prend tous les champs
    // si strict=vrai, tous les champs désirés doivent être présents
    // si ce n'est pas le cas, la ligne n'est pas mémorisée et son index
    // est placé dans le vecteur lignesErronees
    // les lignes blanches sont ignorées
    // ainsi que les lignes commençant par tagCommentaire si tagCommentaire != null
```

La liste des index des champs désirés est maintenant dans une chaîne de caractères (*String*) au lieu d'être dans un tableau d'entiers.

L'attribut privé *erreur* peut avoir une valeur supplémentaire :

2 : la chaînes des index des champs désirés est incorrecte

Voici un exemple de test :

```
import java.io.*;
//import filetovector;

public class test1{
  public static void main(String arg[]){
    System.out.println(new filetovector("data.txt",":","1 3",false,"#").identite());
    System.out.println(new filetovector("data.txt",":","1 3",true,"#").identite());
    System.out.println(new filetovector("data.txt","","1 3",false,"#").identite());
    System.out.println(new filetovector("data.txt",null," 1 3",false,"#").identite());
    String S2=null;
    System.out.println(new filetovector("data.txt",":",S2,false,"#").identite());
    System.out.println(new filetovector("data.txt",":",S2,false,"").identite());
    String S3="";
    System.out.println(new filetovector("data.txt",":",S3,false,null).identite());
  }
}
```

Les résultats :

```
[0,(0,a,c) (0,1,3) (0,azerty,cvf) (0,s)][4,(0,a,c) (0,1,3) (0,azerty,cvf),[5]]
[0,(0,a:b:c:d:e) (0,1 :2 : 3:  4: 5) (0,azerty : 1 : cvf : fff: qqqq) (0,s)]
[0,(0,a:b:c:d:e) (0,1 :2 : 3:  4: 5) (0,azerty : 1 : cvf : fff: qqqq) (0,s)]
[0,(0,a,b,c,d,e) (0,1,2,3,4,5) (0,azerty,1,cvf,fff,qqqq) (0,s)]
[0,(0,a,b,c,d,e) (0,1,2,3,4,5) (0,# commentaire) (0,azerty,1,cvf,fff,qqqq) (0,s)]
[0,(0,a,b,c,d,e) (0,1,2,3,4,5) (0,# commentaire) (0,azerty,1,cvf,fff,qqqq) (0,s)]
```

Quelques conseils

1. On transformera la chaîne *sChampsVoulus* en tableau d'entiers *tChampVoulus* pour se ramener au cas du constructeur précédent.

4. Interfaces graphiques

Nous nous proposons ici de montrer comment construire des interfaces graphiques avec JAVA. Nous voyons tout d'abord quelles sont les classes de base qui nous permettent de construire une interface graphique. Nous n'utilisons dans un premier temps aucun outil de génération automatique. Puis nous utiliserons JBuilder, un outil de développement de Borland/Inprise facilitant le développement d'applications Java et notamment la construction des interfaces graphiques.

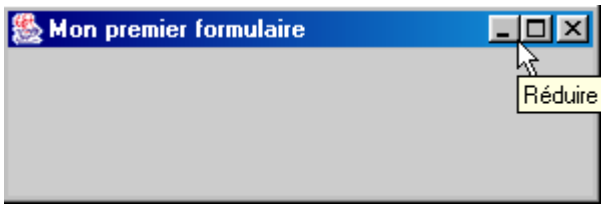
4.1 Les bases des interfaces graphiques

4.1.1 Une fenêtre simple

Considérons le code suivant :

```
// classes importées
import javax.swing.*;
import java.awt.*;

  // la classe formulaire
  public class form1 extends JFrame {

    // le constructeur
       public form1() {
        // titre de la fenêtre
        this.setTitle("Mon premier formulaire");
        // dimensions de la fenêtre
        this.setSize(new Dimension(300,100));
    }//constructeur

    // fonction de test
       public static void main(String[] args) {
        // on affiche le formulaire
          new form1().setVisible(true);
       }
  }//classe
```

L'exécution du code précédent affiche la fenêtre suivante :



Une interface graphique dérive en général de la classe de base *JFrame* :

```
  public class form1 extends JFrame {
```

La classe de base *JFrame* définit une fenêtre de base avec des boutons de fermeture, agrandissement/réduction, une taille ajustable, etc ... et gère les événements sur ces objets graphiques. Ici nous spécialisons la classe de base en lui fixant un titre et ses largeur (300 pixels) et hauteur (100 pixels). Ceci est fait dans son constructeur :

```
    // le constructeur
       public form1() {
        // titre de la fenêtre
        this.setTitle("Mon premier formulaire");
        // dimensions de la fenêtre
        this.setSize(new Dimension(300,100));
    }//constructeur
```

Le titre de la fenêtre est fixée par la méthode *setTitle* et ses dimensions par la méthode *setSize*. Cette méthode accepte pour paramètre un objet *Dimension(largeur,hauteur)* où *largeur* et *hauteur* sont les largeur et hauteur de la fenêtre exprimées en pixels.

La méthode *main* lance l'application graphique de la façon suivante :

```
        new form1().setVisible(true);
```

Un formulaire de type *form1* est alors créé (*new form1()*) et affiché (*setVisible(true)*), puis l'application se met à l'écoute des événements qui se produisent sur le formulaire (clics, déplacements de souris, ...) et fait exécuter ceux que le formulaire gère. Ici, notre formulaire ne gère pas d'autres événements que ceux gérés par la classe de base *JFrame* (clics sur boutons fermeture, agrandissement/réduction, changement de taille de la fenêtre, déplacement de la fenêtre, ...).

Lorsqu'on teste ce programme en le lançant à partir d'une fenêtre Dos par :

```
java form1
```

pour exécuter le fichier *form1.class*, on constate que lorsqu'on ferme la fenêtre qui a été affichée, on ne "récupère pas la main" dans la fenêtre Dos, comme si le programme n'était pas terminé. C'est effectivement le cas. L'exécution du programme se fait de la façon suivante :

- au départ, un premier thread d'exécution est lancé pour exécuter la méthode *main*
- lorsque celle-ci crée le formulaire et l'affiche, un second thread est créé pour gérer spécifiquement les événements liés au formulaire
- après cette création et dans notre exemple, le thread de la méthode *main* se termine et seul reste alors le thread d'exécution de l'interface graphique.
- lorsqu'on ferme la fenêtre, celle-ci disparaît mais n'interrompt pas le thread dans lequel elle s'exécutait
- on est obligé pour l'instant d'arrêter ce thread en faisant Ctrl-C dans la fenêtre Dos d'où a été lancée l'exécution du programme.

Vérifions l'existence de deux threads séparés, l'un dans lequel s'exécute la méthode *main*, l'autre dans lequel s'exécute la fenêtre graphique :

```
// classes importées
import javax.swing.*;
import java.awt.*;
import java.io.*;

// la classe formulaire
 public class form1 extends JFrame {

 // le constructeur
 public form1() {
   // titre de la fenêtre
   this.setTitle("Mon premier formulaire");
   // dimensions de la fenêtre
   this.setSize(new Dimension(300,100));
 }//constructeur

 // fonction de test
 public static void main(String[] args) {
   // suivi
   System.out.println("Début du thread main");
   // on affiche le formulaire
   new form1().setVisible(true);
   // suivi
   System.out.println("Fin du thread main");
 }//main
}//classe
```

L'exécution donne les résultats suivants :

```
E:\data\serge\JAVA\interfaces\intro1>java      form1
Début du thread main
Fin du thread main
```

On peut constater que le thread *main* est terminé alors que la fenêtre est, elle, encore affichée. Le fait de fermer la fenêtre ne termine pas le thread dans laquelle elle s'exécutait. Pour arrêter ce thread, on fait de nouveau Ctrl-C dans la fenêtre Dos.

Pour terminer cet exemple, on notera les paquetages importés :

- *javax.swing* pour la classe *JFrame*
- *java.awt* pour la classe *Dimension*

4.1.2 Gérer un événement

Dans l'exemple précédent, il nous faudrait gérer la fermeture de la fenêtre nous-mêmes pour que lorsqu'elle se produit on arrête l'application, ce qui pour l'instant n'est pas fait. Pour cela il nous faut créer un objet qui "écoute" les événements qui se produisent sur la fenêtre et détecte l'événement "fermeture de la fenêtre". On appelle cet objet un "listener" ou gestionnaire d'événements. Il existe différents types de gestionnaires pour les différents événements qui peuvent se produire sur les composants d'une interface graphique. Pour le composant *JFrame*, le listener s'appelle *WindowListener* et est une interface définissant les méthodes suivantes (cf documentation Java)

Method Summary	
void	**windowActivated**(WindowEvent e) La fenêtre devient la fenêtre active
void	**windowClosed**(WindowEvent e) La fenêtre a été fermée
void	**windowClosing**(WindowEvent e) L'utilisateur ou le programme a demandé la fermeture de la fenêtre
void	**windowDeactivated**(WindowEvent e) La fenêtre n'est plus la fenêtre active
void	**windowDeiconified**(WindowEvent e) La fenêtre passe de l'état réduit à l'état normal
void	**windowIconified**(WindowEvent e) La fenêtre passe de l'état normal à l'état réduit
void	**windowOpened**(WindowEvent e) La fenêtre devient visible pour la première fois

Il y a donc sept événements qui peuvent être gérés. Les gestionnaires reçoivent tous en paramètre un objet de type *WindowEvent* que nous ignorons pour l'instant. L'événement qui nous intéresse ici est la fermeture de la fenêtre, événement qui devra être traité par la méthode *windowClosing*. Pour gérer cet événement, on pourra créer un objet *Windowlistener* à l'aide d'une classe anonyme de la façon suivante :

```
  // création d'un gestionnaire d'événements
  WindowListener win=new WindowListener(){
    public void windowActivated(WindowEvent e){}
    public void windowClosed(WindowEvent e){}
    public void windowClosing(WindowEvent e){System.exit(0);}
    public void windowDeactivated(WindowEvent e){}
    public void windowDeiconified(WindowEvent e){}
    public void windowIconified(WindowEvent e){}
    public void windowOpened(WindowEvent e){}
  };//définition win
```

Notre gestionnaire d'événements implémentant l'interface *WindowListener* doit définir les sept méthodes de cette interface. Comme nous ne voulons gérer que la fermeture de la fenêtre, nous ne définissons du code que pour la méthode *windowClosing*. Lorsque les autres événements se produiront, nous en serons avertis mais nous ne ferons rien. Que ferons-nous lorsqu'on sera averti que la fenêtre est en cours de fermeture (*windowClosing*) ? Nous arrêterons l'application :

```
    public void windowClosing(WindowEvent e){System.exit(0);}
```

Nous avons là un objet capable de gérer les événements d'une fenêtre en général. Comment l'associe-t-on à une fenêtre particulière ? La classe *JFrame* a une méthode *addWindowListener(WindowListener win)* qui permet d'associer un gestionnaire d'événements "fenêtre" à une fenêtre donnée. Ainsi ici, dans le constructeur de la fenêtre nous écrirons :

```
  // création d'un gestionnaire d'événements
  WindowListener win=new WindowListener(){
    public void windowActivated(WindowEvent e){}
    public void windowClosed(WindowEvent e){}
    public void windowClosing(WindowEvent e){System.exit(0);}
    public void windowDeactivated(WindowEvent e){}
    public void windowDeiconified(WindowEvent e){}
    public void windowIconified(WindowEvent e){}
    public void windowOpened(WindowEvent e){}
```

```
   };//définition win
   // ce gestionnaire d'événements va gérer les évts de la fenêtre courante
   this.addWindowListener(win);
```

Le programme complet est le suivant :

```
// classes importées
import javax.swing.*;
import java.awt.*;
import java.io.*;
import java.awt.event.*;

// la classe formulaire
 public class form2 extends JFrame {

 // le constructeur
 public form2() {
   // titre de la fenêtre
   this.setTitle("Mon premier formulaire");
   // dimensions de la fenêtre
   this.setSize(new Dimension(300,100));
   // création d'un gestionnaire d'événements
   WindowListener win=new WindowListener(){
     public void windowActivated(WindowEvent e){}
     public void windowClosed(WindowEvent e){}
     public void windowClosing(WindowEvent e){System.exit(0);}
     public void windowDeactivated(WindowEvent e){}
     public void windowDeiconified(WindowEvent e){}
     public void windowIconified(WindowEvent e){}
     public void windowOpened(WindowEvent e){}
   };//définition win
   // ce gestionnaire d'événements va gérer les évts de la fenêtre courante
   this.addWindowListener(win);
 }//constructeur

 // fonction de test
 public static void main(String[] args) {
   // on affiche le formulaire
   new form2().setVisible(true);
 }//main
}//classe
```

Le paquetage *java.awt.event* contient l'interface *WindowListener*. Lorsqu'on exécute ce programme et qu'on ferme la fenêtre qui s'est afichée on constate dans la fenêtre Dos où a été lancé le programme, la fin d'exécution du programme ce qu'on n'avait pas auparavant.

Dans notre programme, la création de l'objet chargé de gérer les événements de la fenêtre est un peu lourde dans la mesure où on est obligé de définir des méthodes même pour des événements qu'on ne veut pas gérer. Dans ce cas, au lieu d'utiliser l'**interface** *WindowListener* on peut utiliser la **classe** *WindowAdapter*. Celle-ci implémente l'interface *WindowListener*, avec sept méthodes vides. En dérivant la classe *WindowAdapter* et en redéfinissant les seules méthodes qui nous intéressent, nous arrivons au même résultat qu'avec l'interface *WindowListener* mais sans avoir besoin de définir les méthodes qui ne nous intéressent pas. La séquence

- définition du gestionnaire d'événements
- association du gestionnaire à la fenêtre

peut se faire de la façon suivante dans notre exemple :

```
   // création d'un gestionnaire d'événements
   WindowAdapter win=new WindowAdapter(){
     public void windowClosing(WindowEvent e){System.exit(0);}
   };//définition win
   // ce gestionnaire d'événements va gérer les évts de la fenêtre courante
   this.addWindowListener(win);
```

Nous utilisons ici une classe anonyme qui dérive la classe *WindowAdapter* et redéfinit sa méthode *windowClosing*. Le programme devient alors :

```
// classes importées
import javax.swing.*;
import java.awt.*;
import java.io.*;
import java.awt.event.*;

// la classe formulaire
 public class form2 extends JFrame {

 // le constructeur
 public form2() {
   // titre de la fenêtre
   this.setTitle("Mon premier formulaire");
```

```
   // dimensions de la fenêtre
   this.setSize(new Dimension(300,100));
   // création d'un gestionnaire d'événements
   WindowAdapter win=new WindowAdapter(){
     public void windowClosing(WindowEvent e){System.exit(0);}
   };//définition win
   // ce gestionnaire d'événements va gérer les évts de la fenêtre courante
   this.addWindowListener(win); }//constructeur

 // fonction de test
 public static void main(String[] args) {
   // on affiche le formulaire
   new form2().setVisible(true);
 }//main
}//classe
```

Il donne les mêmes résultats que le précédent programme mais est plus simple d'écriture.

4.1.3 Un formulaire avec bouton

Ajoutons maintenant un bouton à notre fenêtre :

```
// classes importées
import javax.swing.*;
import java.awt.*;
import java.io.*;
import java.awt.event.*;

// la classe formulaire
 public class form3 extends JFrame {

  // un bouton
  JButton btnTest=null;
  Container conteneur=null;

 // le constructeur
  public form3() {
    // titre de la fenêtre
    this.setTitle("Formulaire avec bouton");
    // dimensions de la fenêtre
    this.setSize(new Dimension(300,100));
    // création d'un gestionnaire d'événements
    WindowAdapter win=new WindowAdapter(){
     public void windowClosing(WindowEvent e){System.exit(0);}
    };//définition win
    // ce gestionnaire d'événements va gérer les évts de la fenêtre courante
    this.addWindowListener(win);
    // on récupère le conteneur de la fenêtre
    conteneur=this.getContentPane();
    // on choisit un gestionnaire de mise en forme des composants dans ce conteneur
    conteneur.setLayout(new FlowLayout());
    // on crée un bouton
    btnTest=new JButton();
    // on fixe son libellé
    btnTest.setText("Test");
    // on ajoute le bouton au conteneur
    conteneur.add(btnTest);
  }//constructeur

 // fonction de test
 public static void main(String[] args) {
   // on affiche le formulaire
   new form3().setVisible(true);
 }//main
}//classe
```

Une fenêtre *JFrame* a un conteneur dans lequel on peut déposer des composants graphiques (bouton, cases à cocher, listes déroulantes, ...). Ce conteneur est disponible via la méthode *getContentPane* de la classe *JFrame* :

```
  Container conteneur=null;
.........
    // on récupère le conteneur de la fenêtre
    conteneur=this.getContentPane();
```

Tout composant est placé dans le conteneur par la méthode *add* de la classe *Container* . Ainsi pour déposer le composant C dans l'objet *conteneur* précédent, on écrira :

```
conteneur.add(C)
```

Où ce composant est-il placé dans le conteneur ? Il existe divers gestionnaires de disposition de composants portant le nom **XXXLayout**, avec XXX égal par exemple à *Border*, *Flow*, ... Chaque gestionnaire de disposition a ses particularités. Par exemple, le gestionnaire *FlowLayout* dispose les composants en ligne en commençant par le haut du formulaire. Lorsqu'une ligne a été remplie, les composants sont placés sur la ligne suivante. Pour associer un gestionnaire de mise en forme à une fenêtre *JFrame*, on utilise la méthode *setLayout* de la classe *JFrame* sous la forme :

```
  setLayout(objet XXXLayout);
```

Ainsi dans notre exemple, pour associer un gestionnaire de type *FlowLayout* à la fenêtre, on a écrit :

```
    // on choisit un gestionnaire de mise en forme des composants dans ce conteneur
    conteneur.setLayout(new FlowLayout());
```

On peut ne pas utiliser de gestionnaire de disposition et écrire :

```
  setLayout(null);
```

Dans ce cas, on devra donner les coordonnées précises du composant dans le conteneur sous la forme (x,y,largeur,hauteur) où (x,y) sont les coordonnées du coin supérieur gauche du composant dans le conteneur. C'est cette méthode que nous utiliserons le plus souvent par la suite.

On sait maintenant comment sont déposés les composants au conteneur (*add*) et où (*setLayout*). Il nous reste à connaître les composants qu'on peut placer dans un conteneur. Ici nous plaçons un bouton modélisé par la classe *javax.swing.JButton* :

```
  JButton btnTest=null;
.........
    // on crée un bouton
    btnTest=new JButton();
    // on fixe son libellé
    btnTest.setText("Test");
    // on ajoute le bouton au conteneur
    conteneur.add(btnTest);
```

Lorsqu'on teste ce programme, on obtient la fenêtre suivante :

Si on redimensionne le formulaire ci-dessus, le gestionnaire de disposition du conteneur est automatiquement appelé pour replacer les composants :

C'est le principal intérêt des gestionnaires de disposition : celui de maintenir une disposition cohérente des composants au fil des modifications de la taille du conteneur. Utilisons le gestionnaire de disposition *null* pour voir la différence. Le bouton est maintenant placé dans le conteneur par les instructions suivantes :

```
// on choisit un gestionnaire de mise en forme des composants dans ce conteneur
conteneur.setLayout(null);
    // on crée un bouton
btnTest=new JButton();
    // on fixe son libellé
btnTest.setText("Test");
    // on fixe son emplacement et ses dimensions
btnTest.setBounds(10,20,100,20);
    // on ajoute le bouton au conteneur
conteneur.add(btnTest);
```

Ici, on place explicitement le bouton au point (10,20) du formulaire et on fixe ses dimensions à 100 pixels pour la largeur et 20 pixels pour la hauteur. La nouvelle fenêtre devient la suivante :

Si on redimensionne la fenêtre, le bouton reste au même endroit.

Si on clique sur le bouton *Test*, il ne se passe rien. En effet, nous n'avons pas encore associé de gestionnaire d'événements au bouton. Pour connaître le type de gestionnaires d'événements disponibles pour un composant donné, on peut chercher dans la définition de sa classe des méthodes *addXXXListener* qui permettent d'associer au composant un gestionnaire d'événements. La classe *javax.swing.JButton* dérive de la classe *javax.swing.AbstractButton* dans laquelle on trouve les méthodes suivantes :

Method Summary	
void	**addActionListener**(ActionListener l)
void	**addChangeListener**(ChangeListener l)
void	**addItemListener**(ItemListener l)

Ici, il faut lire la documentation pour savoir quel gestionnaire d'événements gère le clic sur le bouton. C'est l'interface *ActionListener*. Celle-ci ne définit qu'une méthode :

Method Summary	
void	**actionPerformed**(ActionEvent e)

La méthode reçoit un paramètre *ActionEvent* que nous ignorerons pour le moment. Pour gérer le clic sur le bouton *btntest* de notre programme on lui associe d'abord un gestionnaire d'événements :

```
      btnTest.addActionListener(new ActionListener()
       {
          public void actionPerformed(ActionEvent evt){
            btnTest_clic(evt);
          }
       }//classe anonyme
    );//gestionnaire d'evt
```

Ici, la méthode *actionPerformed* renvoie à la méthode *btnTest_clic* que nous définissons de la façon suivante :

```
public void btnTest_clic(ActionEvent evt){
  // suivi console
  System.out.println("clic sur bouton");
}//btnTest_click
```

A chaque fois que l'utilisateur clique sur le bouton *Test*, on écrit un message sur la console. C'est ce que montre l'exécution suivante :

```
E:\data\serge\JAVA\interfaces\intro1>java.bat form4
clic sur bouton
clic sur bouton
```

Le programme complet est le suivant :

```
// classes importées
import javax.swing.*;
import java.awt.*;
import java.io.*;
import java.awt.event.*;

// la classe formulaire
 public class form4 extends JFrame {

  // un bouton
  JButton btnTest=null;
  Container conteneur=null;

 // le constructeur
  public form4() {
    // titre de la fenêtre
    this.setTitle("Formulaire avec bouton");
    // dimensions de la fenêtre
    this.setSize(new Dimension(300,100));
    // création d'un gestionnaire d'événements
    WindowAdapter win=new WindowAdapter(){
     public void windowClosing(WindowEvent e){System.exit(0);}
    };//définition win
    // ce gestionnaire d'événements va gérer les évts de la fenêtre courante
    this.addWindowListener(win);
    // on récupère le conteneur de la fenêtre
    conteneur=this.getContentPane();
    // on choisit un gestionnaire de mise en forme des composants dans ce conteneur
    conteneur.setLayout(null);
    // on crée un bouton
    btnTest=new JButton();
    // on fixe son libellé
    btnTest.setText("Test");
    // on fixe son emplacement et ses dimensions
    btnTest.setBounds(10,20,100,20);
    // on lui associe un gestionnaire d'évt
    btnTest.addActionListener(new ActionListener()
     {
       public void actionPerformed(ActionEvent evt){
         btnTest_clic(evt);
       }
     }//classe anonyme
    );//gestinnaire d'evt
    // on ajoute le bouton au conteneur
    conteneur.add(btnTest);
  }//constructeur

  public void btnTest_clic(ActionEvent evt){
    // suivi console
    System.out.println("clic sur bouton");
  }//btnTest_click

 // fonction de test
 public static void main(String[] args) {
   // on affiche le formulaire
   new form4().setVisible(true);
 }//main
}//classe
```

4.1.4 Les gestionnaires d'événements

Les principaux composants swing que nous allons présenter sont les fenêtres (JFrame), les boutons (*JButton*), les cases à cocher (*JCheckBox*), les boutons radio (*JButtonRadio*), les listes déroulantes (*JComboBox*), les listes (*JList*), les variateurs (*JScrollBar*), les étiquettes (*JLabel*), les boîtes de saisie monoligne (*JTextField*) ou multilignes (*JTextArea*), les menus (*JMenuBar*), les éléments de menu (*JMenuItem*).

Les tableaux suivants donnent une liste de quelques gestionnaires d'événements et les événements auxquels ils sont liés.

Gestionnaire	Composant(s)	Méthode d'enregistrement	Evénement
ActionListener	*JButton , JCheckbox, JButtonRadio, JMenuItem*	public void addActionListener(ActionListener)	clic sur le bouton, la case à cocher, le bouton radio, l'élément de menu
	JTextField		l'utilisateur a tapé [Entrée] dans la zone de saisie
ItemListener	*JComboBox, JList*	public void addItemListener(ItemListener)	L'élément sélectionné a changé
InputMethodListener	*JTextField, JTextArea*	public void addMethodInputListener(InputMethodListener)	le texte de la zone de saisie a changé ou le curseur de saisie a changé de position
CaretListener	*JTextField, JTextArea*	public void addcaretListener(CaretListener)	Le curseur de saisie a changé de position
AdjustmentListener	*JScrollBar*	public void addAdjustmentListener(AdjustmentListener)	la valeur du variateur a changé
MouseMotionListener		public void addMouseMotionListener(MouseMotionListener)	la souris a bougé
WindowListener	*JFrame*	public void addWindowlistener(WindowListener)	événement fenêtre
MouseListener		public void addMouselistener(MouseListener)	événements souris (clic, entrée/sortie du domaine d'un composant, bouton pressé, relâche)
FocusListener		public void addFocuslistener(FocusListener)	événement focus (obtenu, perdu)
KeyListener		public void addKeylistener(KeyListener)	événement clavier(touche tapée, pressée, relachée)

Composant	Méthode d'enregistrement des gestionnaires d'évenements
JButton	public void addActionListener(ActionListener)
JCheckbox	public void addItemListener(ItemListener)
JCheckboxMenuItem	public void addItemListener(ItemListener)
JComboBox	public void addItemListener(ItemListener) public void addActionListener(ActionListener)
Container	public void addContainerListener(ContainerListener)
JComponent	public void addComponentListener(ComponentListener) public void addFocusListener(FocusListener) public void addKeyListener(KeyListener) public void addMouseListener(MouseListener) public void addMouseMotionListener(MouseMotionListener)
JFrame	public void addWindowlistener(WindowListener)
JList	public void addItemListener(ItemListener)
JMenuItem	public void addActionListener(ActionListener)
JPanel	comme Container
JScrollPane	comme Container
JScrollBar	public void addAdjustmentListener(AdjustmentListener)
JTextComponent	public void addInputMethodListener(InputMethodListener) public void addCaretListener(CaretListener)
JTextArea	comme JTextComponent
JTextField	comme JTextComponent public void addActionListener(ActionListener)

Tous les composants, sauf ceux du type *TextXXX*, étant dérivés de la classe J*Component,* possédent également les méthodes associées à cette classe.

4.1.5 Les méthodes des gestionnaires d'événements

Le tableau suivant liste les méthodes que doivent implémenter les différents gestionnaires d'événements.

Interface	Méthodes
ActionListener	public void actionPerformed(ActionEvent)
AdjustmentListener	public void adjustmentValueChanged(AdjustmentEvent)
ComponentListener	public void componentHidden(ComponentEvent) public void componentMoved(ComponentEvent) public void componentResized(ComponentEvent) public void componentShown(ComponentEvent)
ContainerListener	public void componentAdded(ContainerEvent) public void componentRemoved(ContainerEvent)

FocusListener	public void focusGained(FocusEvent) public void focusLost(FocusEvent)
ItemListener	public void itemStateChanged(ItemEvent)
KeyListener	public void keyPressed(KeyEvent) public void keyReleased(KeyEvent) public void keyTyped(KeyEvent)
MouseListener	public void mouseClicked(MouseEvent) public void mouseEntered(MouseEvent) public void mouseExited(MouseEvent) public void mousePressed(MouseEvent) public void mouseReleased(MouseEvent)
MouseMotionListener	public void mouseDragged(MouseEvent) public void mouseMoved(MouseEvent)
TextListener	public void textValueChanged(TextEvent)
InputmethodListener	public void InputMethodTextChanged(InputMethodEvent) public void caretPositionChanged(InputMethodEvent)
CaretLisetner	public void caretUpdate(CaretEvent)
WindowListener	public void windowActivated(WindowEvent) public void windowClosed(WindowEvent) public void windowClosing(WindowEvent) public void windowDeactivated(WindowEvent) public void windowDeiconified(WindowEvent) public void windowIconified(WindowEvent) public void windowOpened(WindowEvent)

4.1.6 Les classes adaptateurs

Comme nous l'avons vu pour l'interface *WindowListener*, il existe des classes nommées *XXXAdapter* qui implémentent les interfaces *XXXListener* avec des méthodes vides. Un gestionnaire d'événements dérivé d'une classe *XXXAdapter* peut alors n'implémenter qu'une partie des méthodes de l'interface *XXXListener*, celles dont l'application a besoin.

Supposons qu'on veuille gérer les clics de souris sur un composant *Frame f1*. On pourrait lui associer un gestionnaire d'événements par :

```
  f1.addMouseListener(new gestionnaireSouris());
```

et écrire :

```
  public class gestionnaireSouris implements MouseListener{
    // on écrit les 5 méthodes de l'interface MouseListener
    // mouseClicked, ..., mouseReleased)
  }// fin classe
```

Comme on ne souhaite gérer que les clics de souris, il est cependant préférable d'écrire :

```
public class gestionnaireSouris extends MouseAdapter{
    // on écrit une seule méthode, celle qui gère les clics de souris
    public void mouseClicked(MouseEvent evt){
    …
    }
}// fin classe
```

Le tableau suivant donne les classes adapteurs des différents gestionnaires d'événements :

Gestionnaire d'événements	Adaptateur
ComponentListener	ComponentAdapter
ContainerListener	ContainerAdapter
FocusListener	FocusAdapter
KeyListener	KeyAdapter
MouseListener	MouseAdapter
MouseMotionListener	MouseMotionAdapter
WindowListener	WindowAdapter

4.1.7 Conclusion

Nous venons de présenter les concepts de base de la création d'interfaces graphiques en Java :

- création d'une fenêtre
- création de composants
- association des composants à la fenêtre avec un gestionnaire de disposition
- association de gestionnaires d'événeemnts aux composants

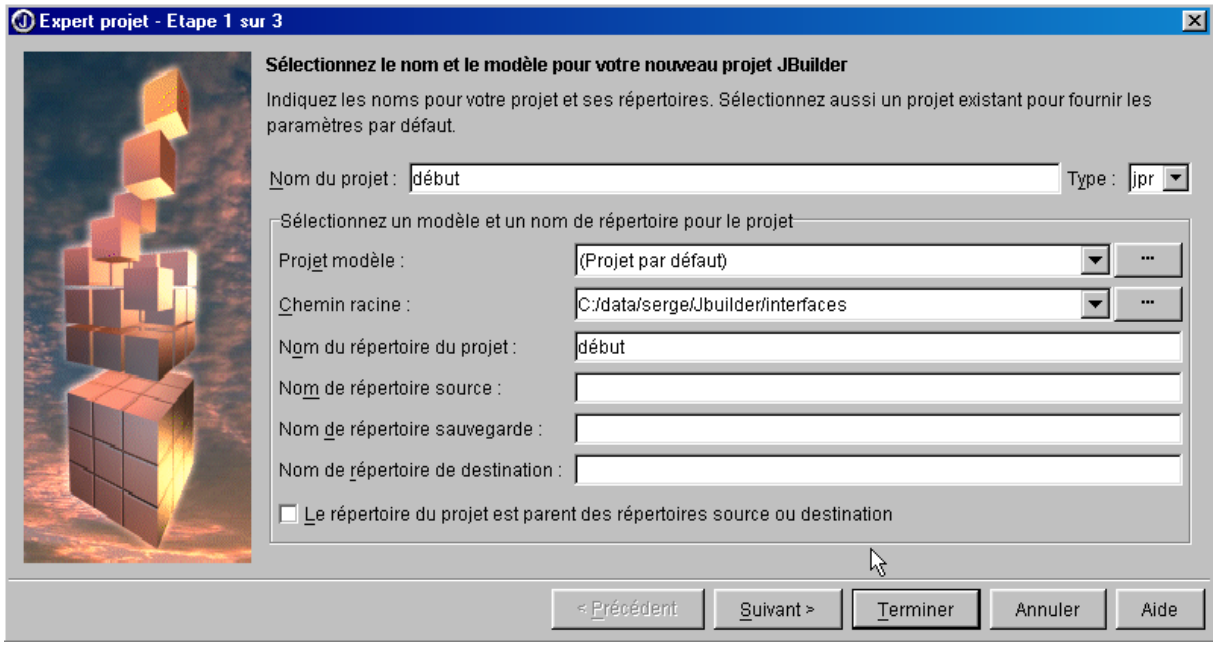
Maintenant, plutôt que de construire des interfaces graphiques "à la main" comme il vient d'être fait nous allons utiliser Jbuilder, un outil de développement Java de Borland/Inprise dans sa version 4 et supérieure.Nous utiliserons les composants de la bibliothèque *java.swing* actuellement préconisée par Sun, créateur de Java.

4.2 Construire une interface graphique avec JBuilder

4.2.1 Notre premier projet Jbuilder

Afin de nous familiariser avec Jbuilder, construisons une application très simple : une fenêtre vide.

1. Lancez Jbuilder et prendre l'option *Fichier/nouveau projet*. La 1ère page d'un assistant s'affiche alors :



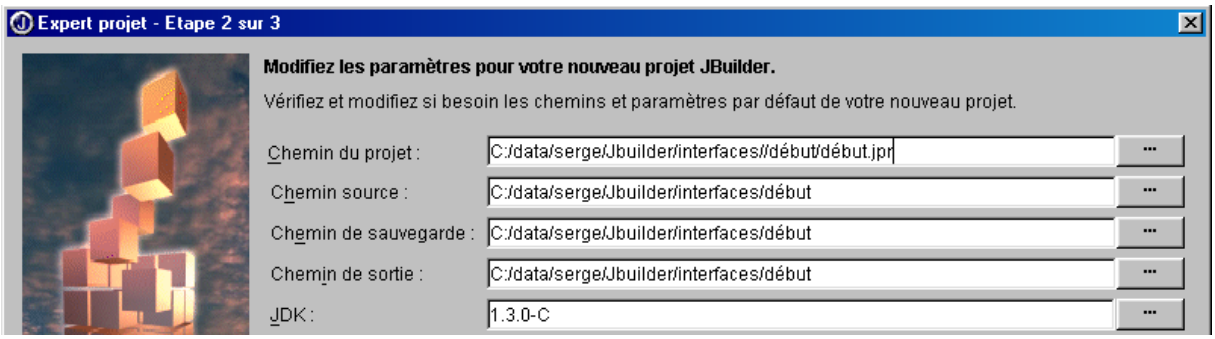
2. Remplir les champs suivants :

Nom du projet	**début** provoquera la création d'un fichier projet *début.jpr* sous le dossier indiqué dans le champ "Nom du répertoire du projet"
Chemin racine	indiquez le dossier sous lequel sera créé le dossier du projet que vous allez créer.
Nom du répertoire projet	indiquez le nom du dossier où seront placés tous les fichiers du projet. Ce dossier sera créé sous le répertoire indiqué dans le champ "Chemin racine"

Les fichiers source (.java, .html, ...), les fichiers destinations (.class,..), les fichiers de sauvegarde pourraient aller dans des répertoires différents. En laissant leurs champs vides, ils seront placés dans le même répertoire que le projet.

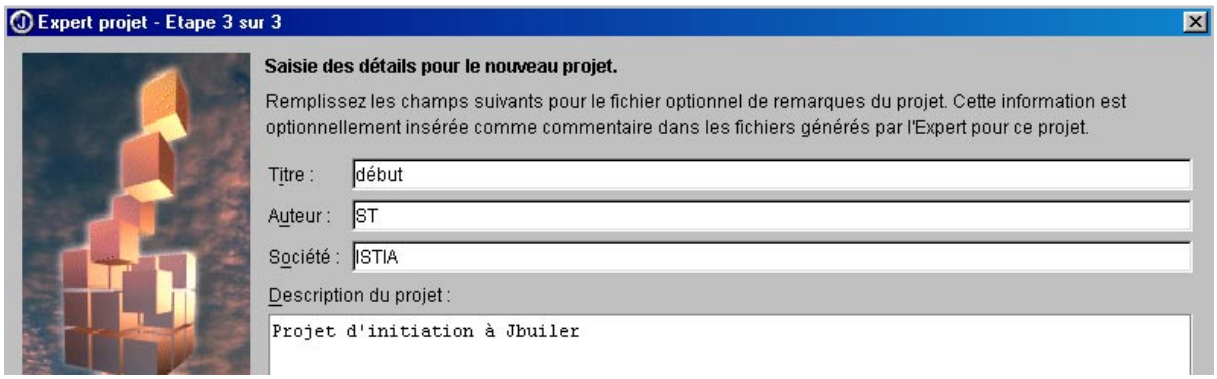
Faites [suivant]

3. Un écran confirme les choix de l'étape précédente



Faites [suivant]

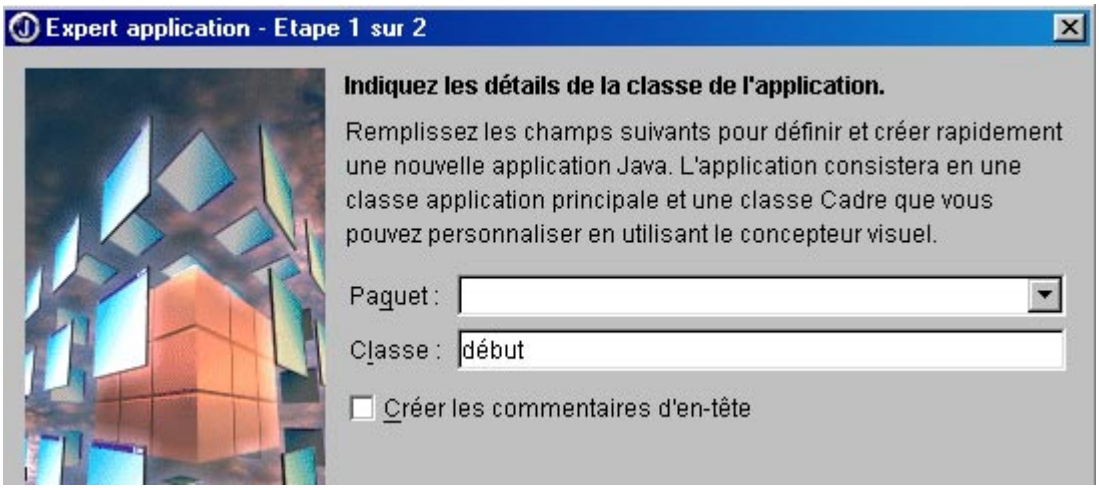
4. Un nouvel écran vous demande de caractériser votre projet :



Faites [terminer]

5. Vérifier que l'option *Voir/projet* est cochée. Vous devriez voir la structure de votre projet dans la fenêtre de gauche.

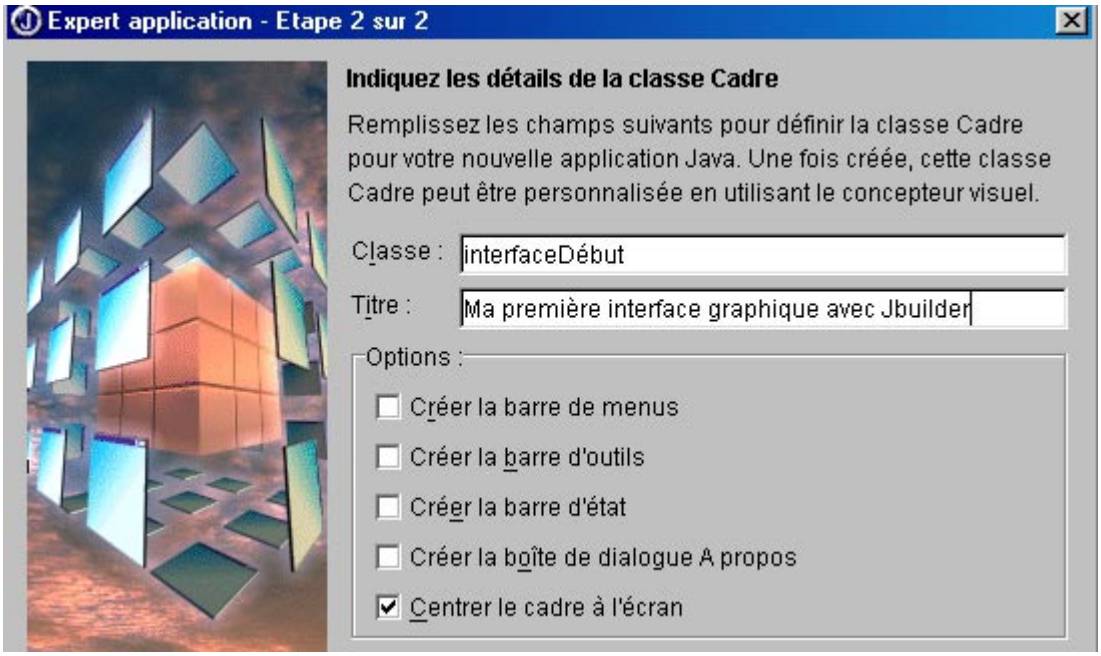
6. Maintenant construisons une interface graphique dans ce projet. Choisissez l'option *Fichier/Nouveau/Application* :



Dans le champ *classe*, on met le nom de la classe qui va être créée. Ici on a repris le même nom que le projet.

Faites [suivant]

7. L'écran suivant apparaît :



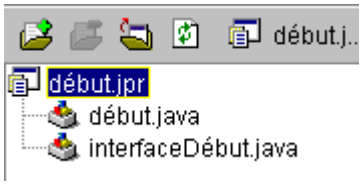
Classe	**interfaceDébut** C'est le nom de la classe correspondant à la fenêtre qui va être construite
Titre	C'est le texte qui apparaîtra dans la barre de titre de la fenêtre

Notez que ci-dessus, nous avons demandé que la fenêtre soit centrée sur l'écran au démarrage de l'application.

Faites [Terminer]

8. C'est maintenant que Jbuilder s'avère utile.

- il génère les fichiers source .java des deux classes dont on a donné les noms : la classe de l'application et celle de l'interface graphique. On voit apparaître ces deux fichiers dans la structure du projet dans la fenêtre de gauche



- pour avoir accès au code généré pour les deux classes, il suffit de double-cliquer sur le fichier .java correspondant. Nous reviendrons ultérieurement sur le code généré.
- Vérifiez que l'option *Voir/Structure* est cochée. Elle permet de voir la structure de la classe actuellement sélectionnée (double clic sur le .java). Voici par exemple la structure de la classe *début* :

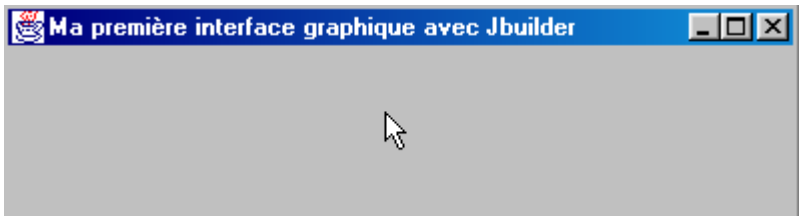
On apprend ici :

- les bibliothèques importées (*imports*)
- qu'il y a un constructeur *début*()
- qu'il y a une méthode statique *main*()
- qu'il y a un attribut *packFrame*

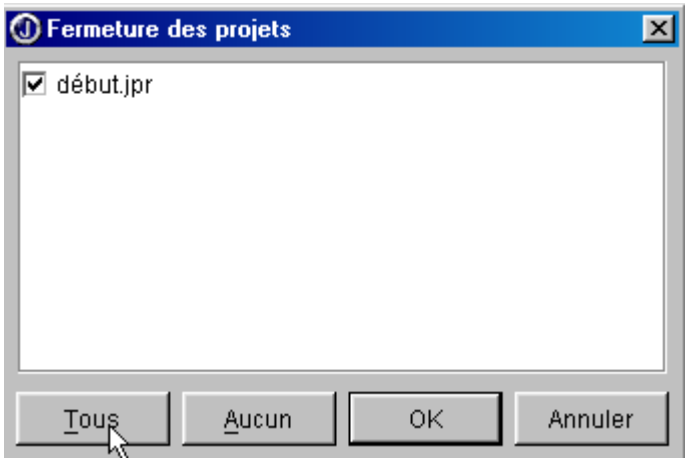
Quel est l'intérêt d'avoir accès à la structure d'une classe ?

- vous avez une vue d'ensemble de celle-ci. C'est utile si votre classe est complexe.
- vous pouvez accéder au code d'une méthode en cliquant dessus dans la fenêtre de structure de la classe. De nouveau c'est utile si votre classe a des centaines de lignes. Vous n'êtes pas obligé de passer toutes les lignes en revue pour trouver le code que vous cherchez.

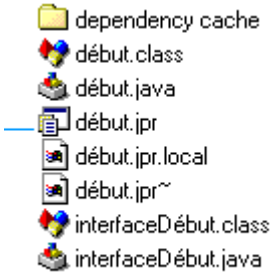
Le code généré par Jbuilder est déjà utilisable. *Faites Exécuter/Exécuter* le projet ou F9 et vous obtiendrez la fenêtre demandée :



De plus, elle se ferme proprement lorsqu'on clique sur le bouton de fermeture de la fenêtre. Nous venons de construire notre première interface graphique. Nous pouvons sauvegarder notre projet par l'option *Fichier/Fermer les projets* :



En appuyant sur *Tous*, tous les projets présents dans la fenêtre ci-dessus seront cochés. *OK* les fermera. Si nous avons la curiosité d'aller, avec l'explorateur windows, dans le dossier de notre projet (celui qui a été indiqué à l'assistant au début de la configuration du projet), nous y trouvons les fichiers suivants :



Dans notre exemple, nous avions demandé à ce que tous les fichiers (source .java, destination .class, sauvegarde .jpr~) soient dans le même dossier que le projet .jpr.

4.2.2 Les fichiers générés par Jbuilder pour une interface graphique

Ayons maintenant la curiosité de regarder ce que Jbuilder a généré comme fichiers source .java. Il est important de savoir lire ce qui est généré puisque la plupart du temps nous sommes amenés à ajouter du code à ce qui a été généré. Commençons par récupérer notre projet : *Fichier/Ouvrir un projet* et sélectionnons le projet *début.jpr*. Nous retrouvons le projet construit précédemment.

4.2.2.1 La classe principale

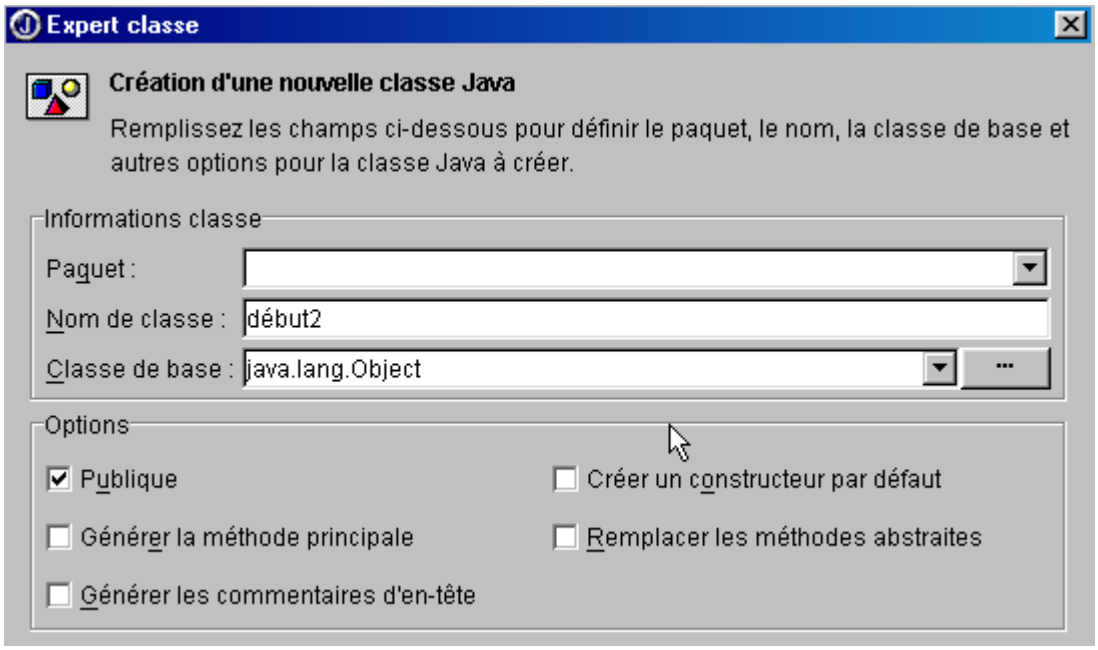
Examinons la classe *début.java* en double-cliquant sur son nom dans la fenêtre présentant la liste des fichiers du projet. Nous avons le code suivant :

```
import javax.swing.UIManager;
import java.awt.*;

public class début {
  boolean packFrame = false;

  /**Construire l'application*/
  public début() {
    interfaceDébut frame = new interfaceDébut();

    //Valider les cadres ayant des tailles prédéfinies
    //Compacter les cadres ayant des infos de taille préférées - ex. depuis leur disposition
    if (packFrame) {
      frame.pack();
    }
    else {
      frame.validate();
    }
    //Centrer la fenêtre
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    Dimension frameSize = frame.getSize();
    if (frameSize.height > screenSize.height) {
      frameSize.height = screenSize.height;
    }
    if (frameSize.width > screenSize.width) {
      frameSize.width = screenSize.width;
    }
    frame.setLocation((screenSize.width - frameSize.width) / 2, (screenSize.height - frameSize.height) /
2);
    frame.setVisible(true);
  }
  /**Méthode principale*/
  public static void main(String[] args) {
    try {
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    new début();
  }
}
```

Commentons le code généré :

1. la fonction *main* fixe l'aspect de la fenêtre (*setLookAndFeel*) et crée une instance de la classe *début*.
2. le constructeur *début()* est alors exécuté. Il crée une instance *frame* de la classe de la fenêtre (*new interfaceDébut()*). Celle-ci est construite mais pas affichée.
3. La fenêtre est alors dimensionnée selon les informations disponibles pour chacun des composants de la fenêtre (*frame.validate*). Elle commence alors son existence séparée en s'affichant et en répondant aux sollicitations de l'utilisateur.
4. La fenêtre est centrée sur l'écran ceci parce qu'on l'avait demandé lors de la configuration de la fenêtre avec l'assistant.

Voyons ce qui serait obtenu si on réduisait le code *début.java* à son strict minimum telle que nous l'avons fait en début de chapitre. Créons une nouvelle classe. Prenez l'option *Fichier/Nouvelle classe* :



Nommez la nouvelle classe *début2* et faites [Terminer]. Un nouveau fichier apparaît dans le projet :

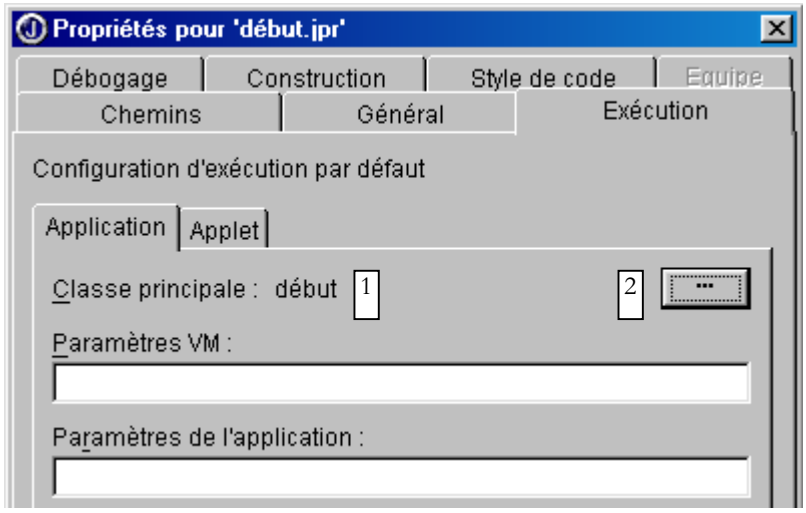
Le fichier *début2.java* est réduit à sa plus simple expression :

```
public class début2 {
}
```

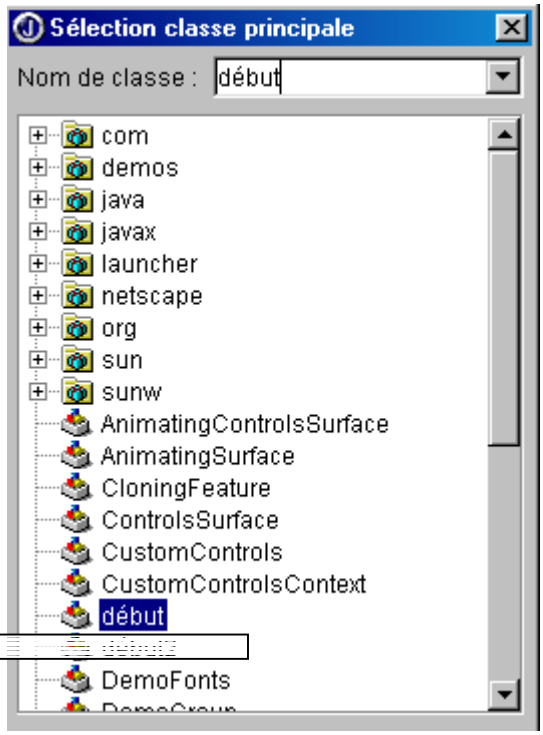
Complétons la classe de la façon suivante :

```
public class début2 {
  // fonction principale
  public static void main(String args[]){
    // crée la fenêtre
    new interfaceDébut().show();  // ou new interfaceDébut.setVisible(true);
  }//main
}//classe début2
```

La fonction *main* crée une instance de la fenêtre *interfaceDébut* et l'affiche (*show*). Avant d'exécuter notre projet, il nous faut signaler que la classe contenant la fonction *main* à exécuter est maintenant la classe *début2*. Cliquez droit sur le projet *début.jpr* et choisissez l'option *propriétés* puis l'onglet *Exécution* :



Ici, il est indiqué que la classe principale est *début* (1). Appuyez sur le bouton (2) pour choisir une autre classe principale :



Choisissez *début2* et faites [OK].

Faites [OK] pour valider ce choix puis demandez l'exécution du projet par *Exécuter/Exécuter projet* ou F9. Vous obtenez la même fenêtre qu'avec la classe *début* si ce n'est qu'elle n'est pas centrée puisqu'ici cela n'a pas été demandé. Par la suite, nous ne

présenterons plus la classe principale générée par Jbuilder car elle fait toujours la même chose : créer une fenêtre. C'est sur la classe de cette dernière que désormais nous nous concentrerons.

4.2.2.2 La classe de la fenêtre

Regardons maintenant le code qui a été généré pour la classe *interfaceDébut* :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class interfaceDébut extends JFrame {
  JPanel contentPane;
  BorderLayout borderLayout1 = new BorderLayout();

  /**Construire le cadre*/
  public interfaceDébut() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  /**Initialiser le composant*/
  private void jbInit() throws Exception  {
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(borderLayout1);
    this.setSize(new Dimension(400, 300));
    this.setTitle("Ma première interface graphique avec Jbuilder");
  }
  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0);
    }
  }
}
```

Les bibliothèques importées

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

Il s'agit des bibliothèques *java.awt*, *java.awt.event* et *javax.swing*. Les deux premières étaient les seules disponibles pour construire des interfaces graphiques avec les premières versions de Java. La bibliothèque *javax.swing* est plus récente. Ici, elle est nécessaire pour la fenêtre de type *JFrame* qui est utilisée ici.

Les attributs

```
  JPanel contentPane;
  BorderLayout borderLayout1 = new BorderLayout();
```

JPanel est un type conteneur dans lequel on peut mettre des composants. *BorderLayout* est l'un des types de gestionnaire de mise en forme disponibles pour placer les composants dans le conteneur. Dans tous nos exemples, nous n'utiliserons pas de gestionnaire de mise en forme et placerons nous-mêmes les composants à un endroit précis du conteneur. Pour cela, nous utiliserons le gestionnaire de mise en forme *null*.

Le constructeur de la fenêtre

```
  /**Construire le cadre*/
  public interfaceDébut() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  /**Initialiser le composant*/
  private void jbInit() throws Exception  {
    contentPane = (JPanel) this.getContentPane();
```

```
    contentPane.setLayout(borderLayout1);
    this.setSize(new Dimension(400, 300));
    this.setTitle("Ma première interface graphique avec Jbuilder");
  }
```

1. Le constructeur commence par dire qu'il va gérer les événements sur la fenêtre (*enableEvents*), puis il lance la méthode *jbInit*.
2. Le conteneur (*JPanel*) de la fenêtre (*JFrame*) est obtenu (*getContentPane*)
3. Le gestionnaire de mise en forme est fixé (*setLayout*)
4. La taille de la fenêtre est fixée (*setSize*)
5. Le titre de la fenêtre est fixé (*setTitle*)

Le gestionnaire d'événements

```
  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0);
    }
```

Le constructeur avait indiqué que la classe traiterait les événements de la fenêtre. C'est la méthode *processWindowEvent* qui fait ce travail. Elle commence par transmettre à sa classe parent (*JFrame*) l'événement *WindowEvent* reçu puis si celui-ci est l'événement *WINDOW_CLOSING* provoqué par le clic sur le bouton de fermeture de la fenêtre, l'application est arrêtée.

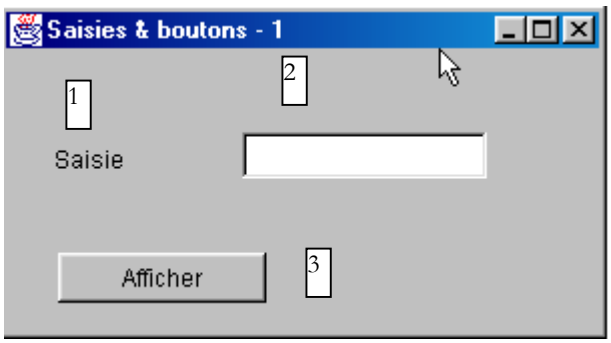
Conclusion

Le code de la classe de la fenêtre est différent de celui présenté dans l'exemple de début de chapitre. Si nous utilisions un autre outil de génération Java que Jbuilder nous aurions sans doute un code encore différent. Dans la pratique, nous accepterons le code produit par Jbuilder pour construire la fenêtre afin de nous concentrer uniquement sur l'écriture des gestionnaires d'événements de l'interface graphique.

4.2.3 Dessiner une interface graphique

4.2.3.1 Un exemple

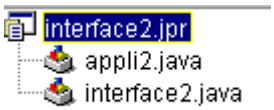
Dans l'exemple précédent, nous n'avions pas mis de composants dans la fenêtre. Construisons maintenant une fenêtre avec un bouton, un libellé et un champ de saisie :



Les champs sont les suivants :

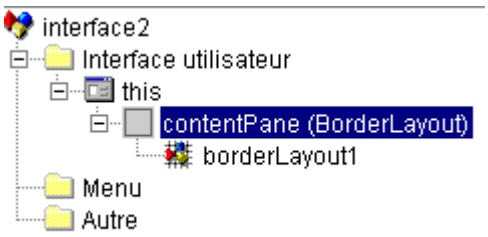
n°	nom	type	rôle
1	lblSaisie	JLabel	un libellé
2	txtSaisie	JTextField	une zone de saisie
3	cmdAfficher	JButton	pour afficher dans une boîte de dialogue le contenu de la zone de saisie txtSaisie

En procédant comme pour le projet précédent, construisez le projet *interface2.jpr* sans mettre pour l'instant de composants dans la fenêtre.



Dans la fenêtre ci-dessus, sélectionnez la classe *interface2.java* de la fenêtre. A droite de cette fenêtre, se trouve un classeur à onglets :

L'onglet *Source* donne accès au code source de la classe *interface2.java*. L'onglet *Conception* permet de construire visuellement la fenêtre. Sélectionnez cet onglet. Vous avez devant vous le conteneur de la fenêtre, qui va recevoir les composants que vous allez y déposer. Il est pour l'instant vide. Dans la fenêtre de gauche est montrée la structure de la classe :



this représente la fenêtre
contentPane son conteneur dans lequel on va déposer des composants ainsi que le mode de mise en forme de ces composants dans le conteneur (BorderLayout par défaut)
borderLayout1 une instance du gestionnaire de mise en forme

Sélectionnez l'objet *this*. Sa fenêtre de propriétés apparaît alors sur la droite :

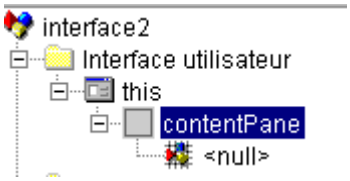
name	this
background	Control
contentPane	contentPane
cursor	
defaultCloseOperation	HIDE_ON_CLOSE
enabled	True
font	
foreground	Black
iconImage	
JMenuBar	
locale	<défaut>
resizable	True
state	NORMAL
title	Saisies & boutons - 1

Certaines de ces propriétés sont à noter :

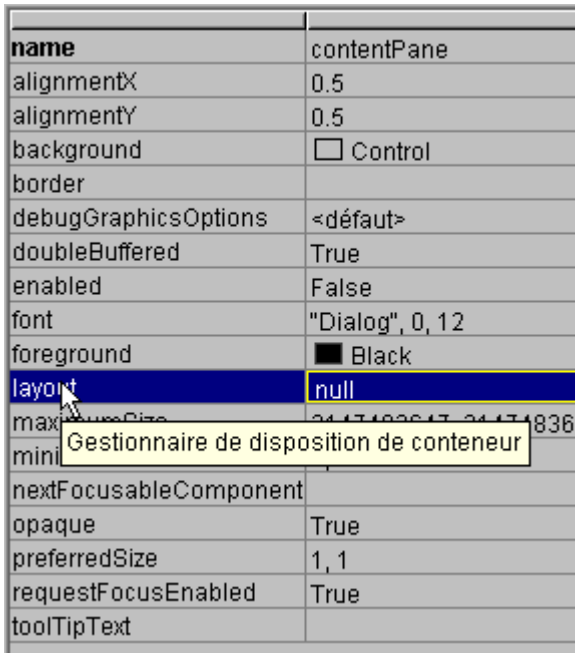
background pour fixer la couleur de fond de la fenêtre
foreground pour fixer la couleur des dessins sur la fenêtre
JMenuBar pour associer un menu à la fenêtre
title pour donner un titre à la fenêtre
resizable pour fixer le type de fenêtre
font pour fixer la police de caractères des écritures dans la fenêtre

L'objet *this* étant toujours sélectionné, on peut redimensionner le conteneur affiché à l'écran en tirant sur les points d'ancrage situés autour du conteneur :

Nous sommes maintenant prêts à déposer des composants dans le conteneur ci-dessus. Auparavant, nous allons changer le gestionnaire de mise en forme. Sélectionnez l'objet *contentPane* dans la fenêtre de structure :

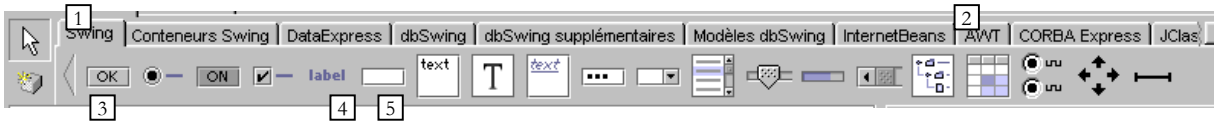


Puis dans la fenêtre de propriétés de cet objet, sélectionnez la propriété *layout* et choisissez parmi les valeurs possibles, la valeur *null* :

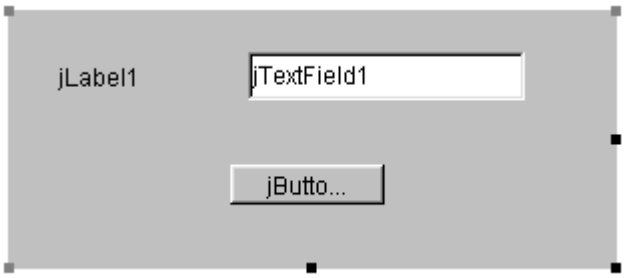


Cette absence de gestionnaire de mise en forme va nous permettre de placer librement les composants dans le conteneur. Il est temps maintenant de choisir ceux-ci.

Lorsque le volet *Conception* est sélectionné, les composants sont disponibles dans un classeur à onglets en haut de la fenêtre de conception :



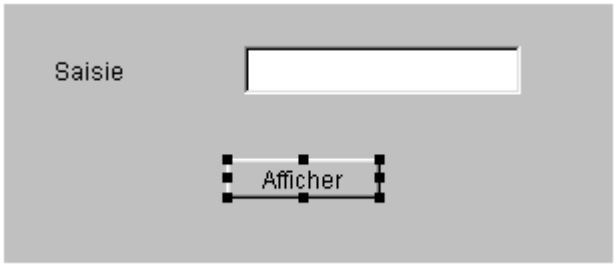
Pour construire l'interface graphique, nous disposons de composants *swing* (1) et de composants *awt* (2). Nous allons utiliser ici les composants *swing*. Dans la barre de composants ci-dessus, choisissez un composant *JLabel* (3), un composant *JTextField* (4) et un composant *JButton* (5) et placez-les dans le conteneur de la fenêtre de conception.



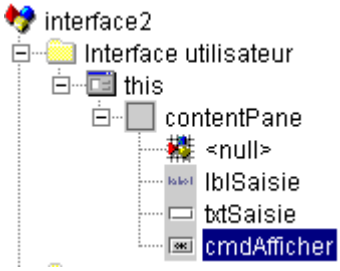
Maintenant personnalisons chacun de ces 3 composants :

- l'étiquette (JLabel) jLabel1
  Sélectionnez le composant pour avoir sa fenêtre de propriétés. Dans celle-ci, modifiez les propriétés suivantes : **name** : lblSaisie, **text** : Saisie
- le champ de saisie (JTextField) jTextfield1
  Sélectionnez le composant pour avoir sa fenêtre de propriétés. Dans celle-ci, modifiez les propriétés suivantes : **name** : txtSaisie, **text** : ne rien mettre
- le bouton (JButton) : **name** : cmdAfficher, **text** : Afficher

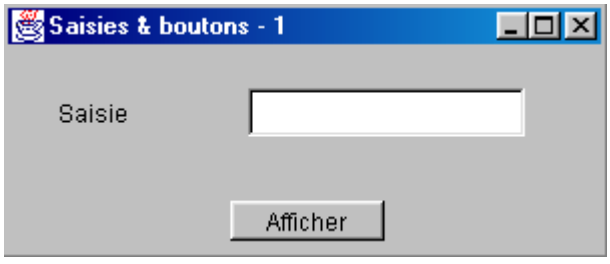
Nous avons maintenant la fenêtre suivante :



et la structure suivante :



Nous pouvons exécuter (F9) notre projet pour avoir un premier aperçu de la fenêtre en action :



Fermez la fenêtre. Il nous reste à écrire la procédure liée à un clic sur le bouton *Afficher*. Sélectionnez le bouton pour avoir accès à sa fenêtre de propriétés. Celle-ci a deux onglets : *propriétés* et *événements*. Choisissez *événements*.

La colonne de gauche de la fenêtre liste les événements possibles sur le bouton. Un clic sur un bouton correspond à l'événement *actionPerformed*. La colonne de droite contient le nom de la procédure appelée lorsque l'événement correspondant se produit. Cliquez sur la cellule à droite de l'événement *actionPerformed* :

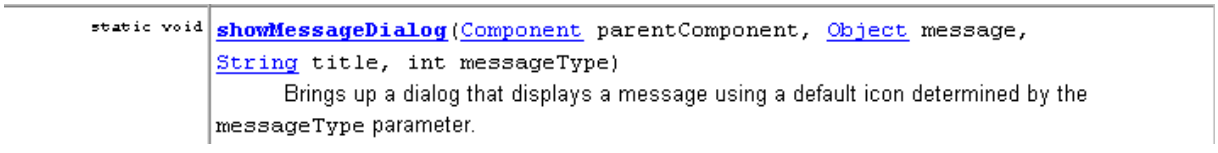
Jbuilder génère un nom par défaut pour chaque gestionnaire d'événement de la forme *nomComposant_nomEvénement* ici *cmdAfficher_actionPerformed*. On pourrait effacer le nom proposé par défaut et en inscrire un autre. Pour avoir accès au code du gestionnaire *cmdAfficher_actionPerformed* il suffit de double-cliquer sur son nom ci-dessus. On passe alors automatiquement au volet source de la classe positionné sur le squelette du code du gestionnaire d'événement :

```
  void cmdAfficher_actionPerformed(ActionEvent e) {
 }
```

Il ne nous reste plus qu'à compléter ce code. Ici, nous voulons présenter une boîte de dialogue avec dedans le contenu du champ *txtSaisie* :

```
  void cmdAfficher_actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(this, "texte saisi="+txtSaisie.getText(),
"Vérification de la saisie",JOptionPane.INFORMATION_MESSAGE);
 }
```

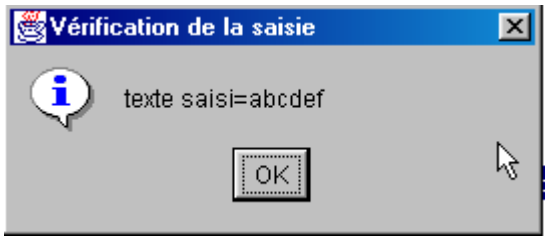
JOptionPane est une classe de la bibliothèque *javax.swing*. Elle permet d'afficher des messages accompagnés d'une icône ou de demander des informations à l'utilisateur. Ici, nous utilisons une méthode statique de la classe :



parentComponent l'objet conteneur "parent" de la boîte de dialogue : ici *this*.
message un objet à afficher. Ici le contenu du champ de saisie
title le titre de la boîte de dialogue
messageType le type du message à afficher. Conditionne l'icône qui sera affichée dans la boîte à côté du message. Les valeurs possibles :

INFORMATION_MESSAGE, QUESTION_MESSAGE, ERROR_MESSAGE,
WARNING_MESSAGE, PLAIN_MESSAGE

Exécutons notre application (F9) :



4.2.3.2 Le code de la classe de la fenêtre

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.beans.*;
import javax.swing.event.*;


public class interface2 extends JFrame {
  JPanel contentPane;
  JLabel lblSaisie = new JLabel();
  JTextField txtSaisie = new JTextField();
  JButton cmdAfficher = new JButton();

  /**Construire le cadre*/
  public interface2() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  /**Initialiser le composant*/
  private void jbInit() throws Exception  {
    lblSaisie.setText("Saisie");
    lblSaisie.setBounds(new Rectangle(25, 23, 71, 21));
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(null);
    this.setSize(new Dimension(304, 129));
    this.setTitle("Saisies & boutons - 1");
    txtSaisie.setBounds(new Rectangle(120, 21, 138, 24));
    cmdAfficher.setText("Afficher");
    cmdAfficher.setBounds(new Rectangle(111, 77, 77, 20));
    cmdAfficher.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        cmdAfficher_actionPerformed(e);
      }
    });
    contentPane.add(lblSaisie, null);
    contentPane.add(txtSaisie, null);
    contentPane.add(cmdAfficher, null);
  }
  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
```

```
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0);
    }
  }
  void cmdAfficher_actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(this, "texte saisi="+txtSaisie.getText(), "Vérification de la
saisie",JOptionPane.INFORMATION_MESSAGE);
  }
}
```

Les attributs

```
  JPanel contentPane;
  JLabel lblSaisie = new JLabel();
  JTextField txtSaisie = new JTextField();
  JButton cmdAfficher = new JButton();
```

On trouve le conteneur des composants de type *JPanel* et les trois composants.

Le constructeur

```
  /**Construire le cadre*/
  public interface2() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  /**Initialiser le composant*/
  private void jbInit() throws Exception  {
    lblSaisie.setText("Saisie");
    lblSaisie.setBounds(new Rectangle(25, 23, 71, 21));
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(null);
    this.setSize(new Dimension(304, 129));
    this.setTitle("Saisies & boutons - 1");
    txtSaisie.setBounds(new Rectangle(120, 21, 138, 24));
    cmdAfficher.setText("Afficher");
    cmdAfficher.setBounds(new Rectangle(111, 77, 77, 20));
    cmdAfficher.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        cmdAfficher_actionPerformed(e);
      }
    });
    contentPane.add(lblSaisie, null);
    contentPane.add(txtSaisie, null);
    contentPane.add(cmdAfficher, null);
  }
```

Le constructeur *interface2* est semblable au constructeur de la précédente interface graphique étudiée. C'est dans la méthode *jbInit* qu'on trouve des différences : le code de construction de la fenêtre dépend des composants qu'on y a placés. On peut reprendre le code de *jbInit* en y mettant nos propres commentaires :

```
  private void jbInit() throws Exception  {
    // la fenêtre elle-même (taille, titre)
    this.setSize(new Dimension(304, 129));
    this.setTitle("Saisies & boutons - 1");
    // le conteneur des composants
    contentPane = (JPanel) this.getContentPane();
    // pas de gestionnaire de mise en forme pour ce conteneur
    contentPane.setLayout(null);
    // l'étiquette lblSaisie (libellé, position, taille)
    lblSaisie.setText("Saisie");
    lblSaisie.setBounds(new Rectangle(25, 23, 71, 21));
    // le champ de saisie (position, taille)
    txtSaisie.setBounds(new Rectangle(120, 21, 138, 24));
    // le bouton Afficher (libellé, position, taille)
    cmdAfficher.setText("Afficher");
    cmdAfficher.setBounds(new Rectangle(111, 77, 77, 20));
    // le gestionnaire d'évt du bouton
    cmdAfficher.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        cmdAfficher_actionPerformed(e);
      }
    });
    // ajout des 3 composants au conteneur
    contentPane.add(lblSaisie, null);
    contentPane.add(txtSaisie, null);
    contentPane.add(cmdAfficher, null);
```

```
  }//jbInit
```

On notera deux points :

- ce code aurait pu être écrit à la main. Cela veut dire qu'on peut se passer de Jbuilder pour construire une interface graphique.
- la façon dont le gestionnaire d'événement du bouton *cmdAfficher* est fixé. Le gestionnaire d'événement du composant *cmdAfficher* aurait pu être déclaré par *cmdAfficher.addActionListener(new gestionnaire())* où *gestionnaire* serait une classe avec une méthode publique *actionPerformed* chargée de gérer le clic sur le bouton *Afficher*. Ici, Jbuilder utilise comme gestionnaire, une instance d'une classe anonyme :

```
 new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        cmdAfficher_actionPerformed(e);
      }
```

Une nouvelle instance de l'interface *ActionListener* est créée avec sa méthode *actionPerformed* définie dans la foulée. Celle-ci se contente d'appeler une méthode de la classe *interface2*. Tout ceci n'est qu'un artifice pour définir dans la même classe que la fenêtre les procédures de traitement des événements des composants de cette fenêtre. On pourrait procéder autrement :

```
cmdAfficher.addActionListener(this)
```

qui fait que la méthode *actionPerformed* sera cherchée dans *this* c'est à dire dans la classe de la fenêtre. Cette seconde méthode semble plus simple mais la première a sur elle un avantage : elle permet d'avoir des gestionnaires différents pour des boutons différents alors que la seconde méthode ne le permet pas. Dans de dernier cas, en effet, l'unique méthode *actionPeformed* doit traiter les clics de différents boutons et donc commencer par identifier lequel est à l'origine de l'événement avant de commencer à travailler.

Les gestionnaires d'événements

On retrouve ceux déjà étudiés :

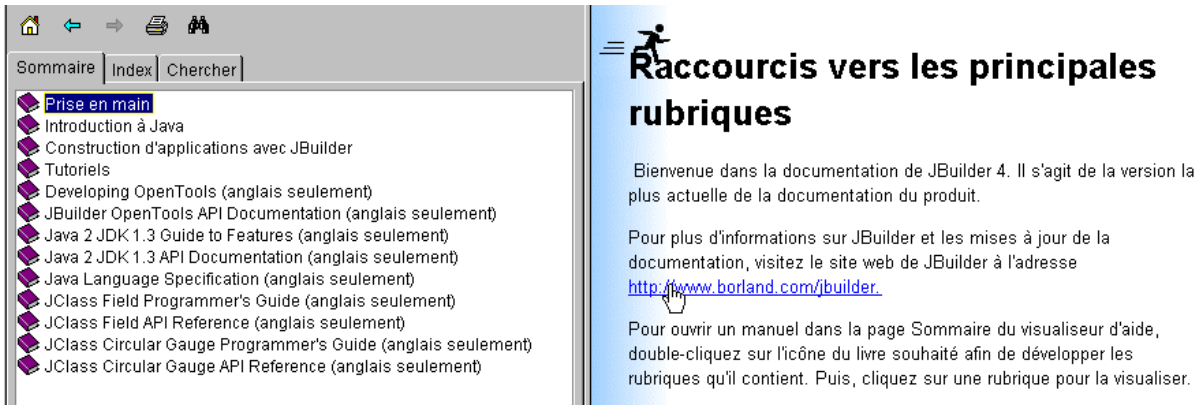
```
  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0);
    }
  }

  // clic sur bouton Afficher
  void cmdAfficher_actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(this, "texte saisi="+txtSaisie.getText(), "Vérification de la
saisie",JOptionPane.INFORMATION_MESSAGE);
  }
```

4.2.3.3 Conclusion

Des deux projets étudiés, nous pouvons conclure qu'une fois l'interface graphique construite avec Jbuilder, le travail du développeur consiste à écrire les gestionnaires des événements qu'il veut gérer pour cette interface graphique.

4.2.4 Chercher de l'aide

Avec Java, on a souvent besoin d'aide à cause notamment du très grand nombre de classes disponibles. Nous donnons ici quelques indications pour trouver de l'aide sur une classe. Prenez l'option *Aide/Rubriques d'aide* du menu.



L'écran d'aide a généralement deux fenêtres :

- celle de gauche où l'on dit ce qu'on cherche. Elle a trois onglets : *Sommaire*, *Index* et *Chercher*.
- celle de droite qui présente le résultat de la recherche

On dispose d'une aide sur la façon d'utiliser l'aide de Jbuilder. Choisissez dans l'aide de Jbuilder, l'option *Aide/Utilisation de l'aide*. On vous explique alors comment utiliser l'aide. On vous indique par exemple les différentes composantes du visualisateur d'aide :

Les parties principales du visualiseur d'aide

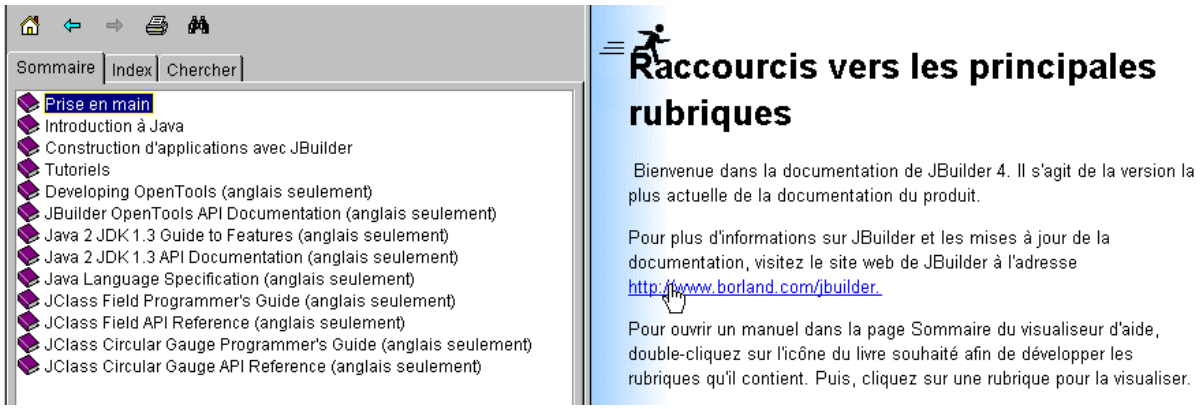
Le visualiseur d'aide de JBuilder comprend les éléments suivants :

- Le menu principal
- La page Sommaire, qui montre la table des matières de tous les manuels
- La page Index, qui montre l'index de tous les manuels
- La page Chercher, qui permet la saisie des mots à rechercher dans tous les manuels
- Le volet Contenu, qui affiche le texte de la rubrique sélectionnée
- Les boutons suivants :

Bouton	Nom	Description
	Début	Va à la première rubrique de l'historique.
	En arrière	Va à la rubrique précédente de l'historique.
	En avant	Va à la rubrique suivante de l'historique.
	Imprimer	Imprime la rubrique en cours.
	Chercher	Cherche du texte dans la rubrique en cours.

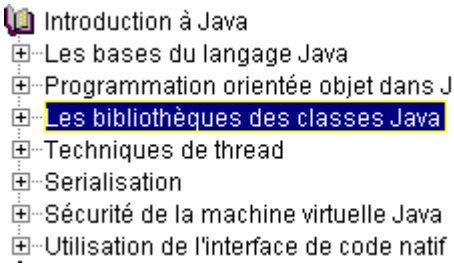
Examinons d'un peu plus près, les pages *Sommaire* et *Index*.

4.2.4.1 Aide : Sommaire



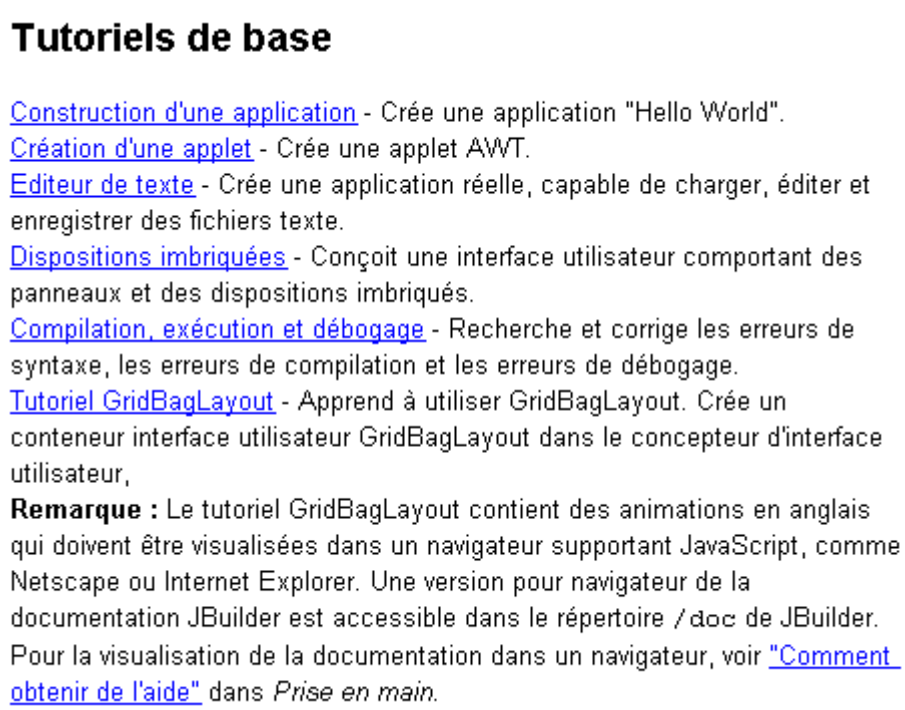
Sommaire : Introduction à Java

On trouvera ici des éléments de base de Java mais pas seulement comme le montre la liste des thèmes évoqués dans cette option :



Sommaire : Tutoriels

Si nous sélectionnons l'option Tutoriels dans le sommaire ci-dessus, la fenêtre de droite nous présente une liste de tutoriels disponibles :

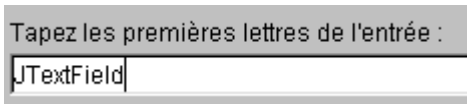


Les tutoriels de base sont particulièrement intéressants pour commencer à prendre en main Jbuilder. Il en existe beaucoup d'autres que ceux présentés ci-dessus et lorsqu'on veut développer une application il peut être utile de vérifier auparavant s'il n'y a pas un tutoriel qui pourrait nous aider.

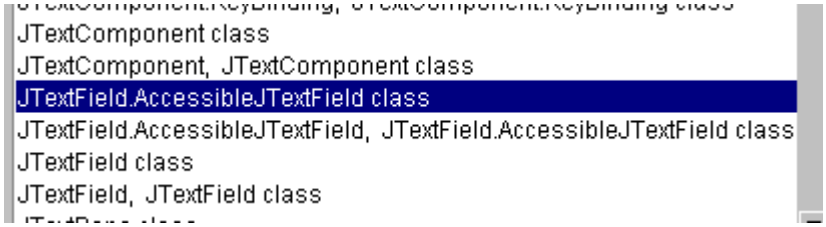
En sélectionnant l'option Java 2 JDK 1.3, on a accès à toutes les bibliothèques du JDK. Généralement ce n'est pas là qu'il faut chercher si on a besoin d'informations sur une classe précise et qu'on ne sait pas dans quelle bibliothèque elle se trouve. Par contre, cette option présente de l'intérêt si on est intéressé par avoir une vue globale des bibliothèques de Java.

4.2.4.2 Aide : Index

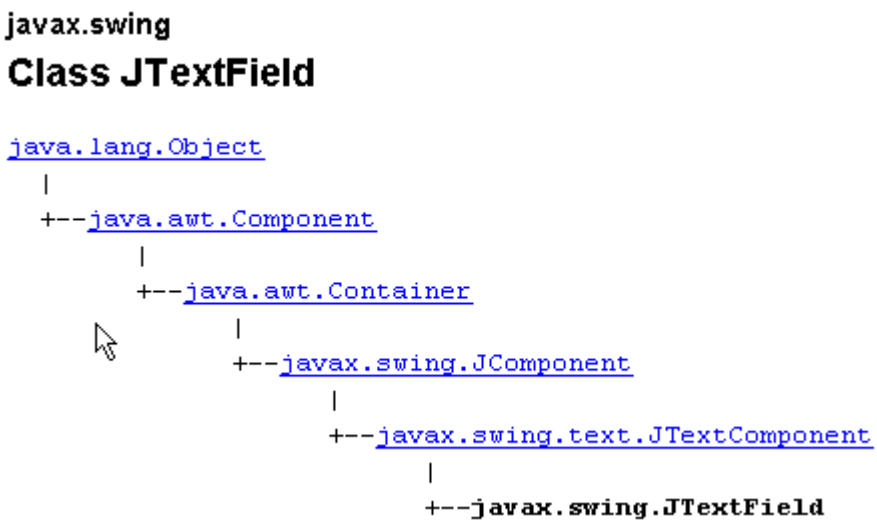
Sélectionnez l'onglet Index de la fenêtre de gauche dans l'aide. Cette option vous permet par exemple de trouver de l'aide sur une classe. Supposons par exemple que vous vouliez connaître les méthodes des champs de saisie *swing JTextField*. Tapez *JTextField* dans le champ de saisie du texte recherché :



L'aide va ramener les entrées d'index commençant par le texte tapé :



Il vous reste à double-cliquer sur l'entrée qui vous intéresse, ici *JTextField class*. L'aide sur cette classe s'affiche alors dans la fenêtre de droite :



Une description complète de la classe vous est alors donnée.

4.2.5 Quelques composants swing

Nous présentons maintenant diverses applications mettant en jeu les composants *swing* les plus courants afin d'en découvrir les principales méthodes et propriétés. Pour chaque application, nous présentons l'interface graphique et le code intéressant notamment celui des gestionnaires d'événements.

4.2.5.1 composants JLabel et JTextField

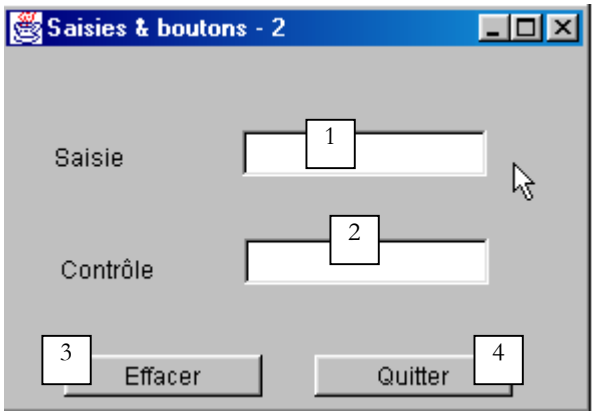
Nous avons déjà rencontré ces deux composants. *JLabel* est un composant texte et *JTextField* un composant champ de saisie. Leurs deux méthodes principales sont

String getText()	pour avoir le contenu du champ de saisie ou le texte du libellé
void setText(String unTexte)	pour mettre unTexte dans le champ ou le libellé

Les événements habituellement utilisés pour *JTextField* sont les suivants :

actionPerformed	signale que l'utilisateur a validé (touche Entrée) le texte saisi
caretUpdate	signale que l'utilisateur à bougé le curseur de saisie
inputMethodChanged	signale que l'utilisateur à modifié le champ de saisie

Voici un exemple qui utilise l'événement *caretUpdate* pour suivre les évolutions d'un champ de saisie :



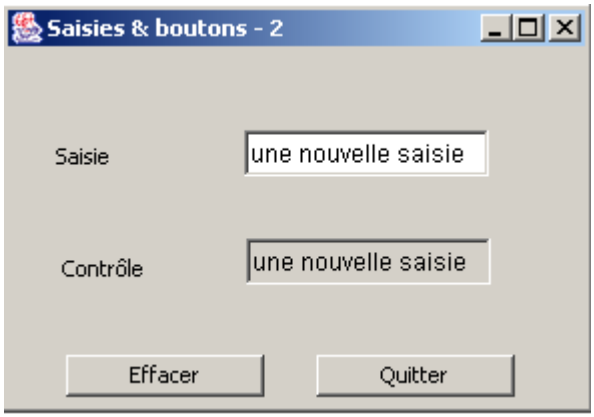
n°	type	nom	rôle
1	JTextField	txtSaisie	champ de saisie
2	JTextField	txtControle	affiche le texte de 1 en temps réel
3	JButton	cmdEffacer	pour effacer les champs 1 et 2
4	JButton	cmdQuitter	pour quitter l'application

Le code pertinent de cette application est le suivant :

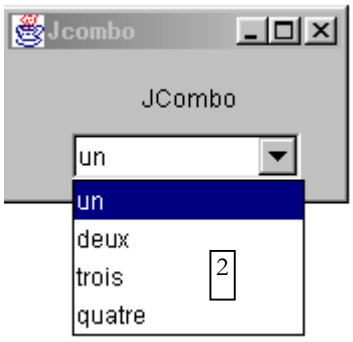
```
import java.awt.*;
....

public class Cadre1 extends JFrame {
  JPanel contentPane;
  JTextField txtSaisie = new JTextField();
  JLabel jLabel1 = new JLabel();
  JLabel jLabel2 = new JLabel();
  JTextField txtControle = new JTextField();
  JButton CmdEffacer = new JButton();
  JButton CmdQuitter = new JButton();

  /**Construire le cadre*/
  public Cadre1() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  /**Initialiser le composant*/
  private void jbInit() throws Exception  {
    ....
    txtSaisie.addCaretListener(new javax.swing.event.CaretListener() {
      public void caretUpdate(CaretEvent e) {
        txtSaisie_caretUpdate(e);
      }
    });
```

```
...
    CmdEffacer.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        CmdEffacer_actionPerformed(e);
      }
    });
...
    CmdQuitter.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        CmdQuitter_actionPerformed(e);
      }
    });
....
}

  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
  protected void processWindowEvent(WindowEvent e) {
...
  }

  void txtSaisie_caretUpdate(CaretEvent e) {
    // le curseur de saisie a bougé
    txtControle.setText(txtSaisie.getText());
  }

  void CmdQuitter_actionPerformed(ActionEvent e) {
    // on quitte l'application
    System.exit(0);
  }

  void CmdEffacer_actionPerformed(ActionEvent e) {
    // on efface le contenu du champ de saisie
    txtSaisie.setText("");
  }
}
```

Voici un exemple d'exécution :



4.2.5.2 composant JComboBox



Un composant *JComboBox* est une liste déroulante doublée d'une zone de saisie : l'utilisateur peut soit choisir un élément dans (2) soit taper du texte dans (1). Par défaut, les *JComboBox* ne sont pas éditables. Il faut appeler explicitement la méthode *setEditable(true)* pour qu'ils le deviennent. Pour découvrir la classe *JComboBox*, tapez *JComboBox* dans l'index de l'aide.

L'objet *JComboBox* peut être construit de différentes façons :

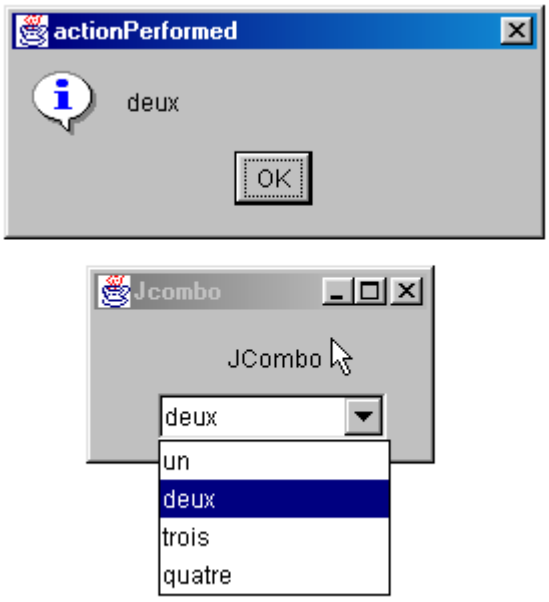
new JComboBox()	crée un combo vide
new JComboBox (Object[] items)	crée un combo contenant un tableau d'objets
new JComboBox(Vector items)	idem avec un vecteur d'objets

On peut s'étonner qu'un combo puisse contenir des objets alors qu'habituellement il contient des chaînes de caractères. Au niveau visuel, ce sera le cas. Si un *JComboBox* contient un objet *obj*, il affiche la chaîne *obj.toString()*. On se rappelle que tout objet a une méthode *toString* héritée de la classe *Object* et qui rend une chaîne de caractères "représentative" de l'objet.

Les méthodes utiles de la classe *JCombobox* sont les suivantes :

void addItem(Object unObjet)	ajoute un objet au combo
int getItemCount()	donne le nombre d'éléments du combo
Object getItemAt(int i)	donne l'objet n° i du combo
void insertItemAt(Object unObjet, int i)	insère unObjet en position i du combo
int getSelectedIndex()	donne le n° de l'élément sélectionné dans le combo
Object getSelectedItem()	donne l'objet sélectionné dans le combo
void setSelectedIndex(int i)	sélectione l'élément i du combo
void setSelectedItem(Object unObjet)	sélectionne l'objet spécifié dans le combo
void removeAllItems()	vide le combo
void removeItemAt(int i)	enlève l'élément n° i du combo
void removeItem(Object unObjet)	enlève l'objet spécifié du combo
void setEditable(boolean val)	rend le combo éditable (val=true) ou non (val=false)

Lors du choix d'un élément dans la liste déroulante se produit l'événement *actionPerformed* qui peut être alors utilisé pour être averti du changement de sélection dans le combo. Dans l'application suivante, nous utilisons cet événement pour afficher l'élément qui a été sélectionné dans la liste.



Nous ne présentons que le code pertinent de la fenêtre.

```
public class Cadre1 extends JFrame {
  JPanel contentPane;
  JComboBox jComboBox1 = new JComboBox();
  JLabel jLabel1 = new JLabel();

  /**Construire le cadre*/
  public Cadre1() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    // traitement - on remplit le combo
    String[] infos={"un","deux","trois","quatre"};
    for(int i=0;i<infos.length;i++)
      jComboBox1.addItem(infos[i]);
  }
  /**Initialiser le composant*/
  private void jbInit() throws Exception  {
    ....

    jComboBox1.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        jComboBox1_actionPerformed(e);
      }
    });
    ....
  }

  void jComboBox1_actionPerformed(ActionEvent e) {
    // un nouvel élément a été sélectionné - on l'affiche
    JOptionPane.showMessageDialog(this,jComboBox1.getSelectedItem(),
"actionPerformed",JOptionPane.INFORMATION_MESSAGE);
  }

}
```

4.2.5.3 composant JList

Le composant swing *JList* est plus complexe que son homologue de la bibliothèque *awt*. Il y a deux différences importantes :

- le contenu de la liste est géré par un objet différent de la liste elle-même. Ici nous prendrons un objet *DefaultListModel* objet qui s'utilise comme un *Vector* mais qui de plus avertit l'objet *JList* dès que son contenu change afin que l'affichage visuel de la liste change aussi.
- la liste n'est pas défilante par défaut. Il faut mettre la liste dans un conteneur *ScrollPane* qui lui, permet ce défilement.

Dans le code source, la définition d'une liste peut se faire de la façon suivante :

```
  // le vecteur des valeurs de la liste
DefaultlistModel valeurs=new DefaultListModel();
// la liste elle-même à laquelle on associe le vecteur de ses valeurs
  JList jList1 = new JList(valeurs);
  // le conteneur défilant dans lequel on place la liste pour avoir une liste défilante
  JScrollPane jScrollPane1 = new JScrollPane(jList1);
```

Pour inclure la liste *jList1* dans le conteneur *jScrollPane1*, le code généré par JBuilder procéde différemment :

- déclaration du conteneur dans les attributs de la fenêtre

```
  JScrollPane jScrollPane1 = new JScrollPane();
```

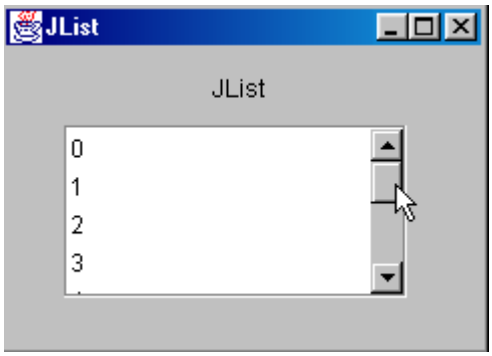
- puis dans le code de *jbInit*, la liste est associée au conteneur

```
    jScrollPane1.getViewport().add(jList1, null);
```

Pour ajouter des valeurs à la liste *JList1* ci-dessus, il suffit de les ajouter à son vecteur de valeurs *valeurs* :

```
    // init liste
    for(int i=0;i<10;i++)
      valeurs.addElement(""+i);
```

et on obtient alors la fenêtre suivante :



Comment cette interface est-elle construite ?

- on choisit un composant *JScrollPane* dans la page "Conteneurs swing" des composants et on le dépose dans la fenêtre en le dimensionnant à la taille désirée
- on choisit un composant *JList* dans la page "swing" des composants et on le dépose dans le conteneur *JScrollPane* dont il occupe alors toute la place.

Le code généré par Jbuilder doit être légèrement remanié pour obtenir le code suivant :

```
public class interfaceAppli extends JFrame {
  JPanel contentPane;
  JLabel jLabel1 = new JLabel();
  DefaultListModel valeurs=new DefaultListModel();
  JList jList1 = new JList(valeurs);
  JScrollPane jScrollPane1 = new JScrollPane();
  JLabel jLabel2 = new JLabel();


  /**Construire le cadre*/
  public interfaceAppli() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    // traitement
    // on inclut la liste dans le scrollPane
    // init liste
    for(int i=0;i<10;i++)
      valeurs.addElement(""+i);
  }

  /**Initialiser le composant*/
  private void jbInit() throws Exception  {
```

```
    ....
    // la liste jList1 est associé au conteneur jcrollPane1
     jScrollPane1.getViewport().add(jList1, null);
  }
  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
  protected void processWindowEvent(WindowEvent e) {
    ....
  }
}
```

Découvrons maintenant les principales méthodes de la classe *JList* en cherchant *JList* dans l'index de l'aide. L'objet *JList* peut être construit de différentes façons :

Constructor Summary

JList() Constructs a JList with an empty model.
JList(ListModel dataModel) Constructs a JList that displays the elements in the specified, non-null model.
JList(Object[] listData) Constructs a JList that displays the elements in the specified array.
JList(Vector listData) Constructs a JList that displays the elements in the specified Vector.

Une méthode simple est celle que nous avons utilisée : créer un *DefaultListModel* V vide puis l'associer à la liste à créer par *new JList(V)*. Le contenu de la liste n'est pas géré par l'objet *JList* mais par l'objet contenant les valeurs de la liste. Si le contenu a été construit à l'aide d'un objet *DefaultListModel* qui repose sur la classe *Vector*, ce sont les méthodes de la classe *Vector* qui pourront être utilisées pour ajouter, insérer et supprimer des éléments de la liste. Une liste peut être à sélection simple ou multiple. Ceci est fixé par la méthode *setSelectionMode* :

setSelectionMode

public void **setSelectionMode**(int selectionMode)

Determines whether single-item or multiple-item selections are allowed. The following selectionMode values are allowed:

- SINGLE_SELECTION Only one list index can be selected at a time. In this mode the setSelectionInterval and addSelectionInterval methods are equivalent, and only the second index argument is used.
- SINGLE_INTERVAL_SELECTION One contiguous index interval can be selected at a time. In this mode setSelectionInterval and addSelectionInterval are equivalent.
- MULTIPLE_INTERVAL_SELECTION In this mode, there's no restriction on what can be selected. This is the default.

On peut connaître le mode de sélection en cours avec *getSelectionMode* :

int	**getSelectionMode**() Returns whether single-item or multiple-item selections are allowed.

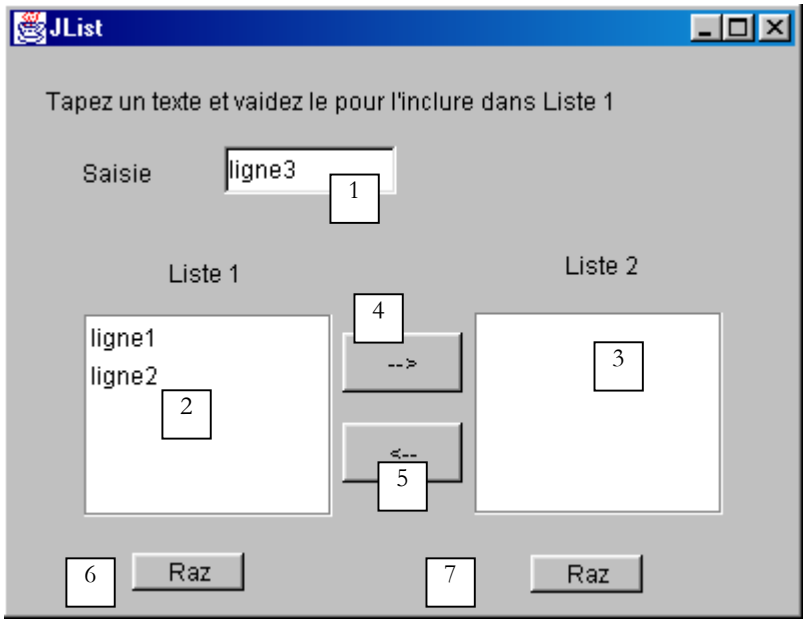
Le ou les éléments sélectionnés peuvent être obtenus via les méthodes suivantes :

int	**getSelectedIndex**() Returns the first selected index; returns -1 if there is no selected item.
int[]	**getSelectedIndices**() Returns an array of all of the selected indices in increasing order.

Nous savons associer un vecteur de valeurs à une liste avec le constructeur *JList(DefaultListModel)*. Inversement nous pouvons obtenir l'objet *DefaultListModel* d'une liste *JList* par :

ListModel	**getModel**() Returns the data model that holds the list of items displayed by the JList component.

Nous en savons assez pour écrire l'application suivante :



Les composants de cette fenêtre sont les suivants :

n°	type	nom	rôle
1	JTextField	txtSaisie	champ de saisie
2	JList	jList1	liste contenue dans un conteneur jScrollPane1
3	JList	jList2	liste contenue dans un conteneur jScrollPane2
4	JButton	cmd1To2	transfère les éléments sélectionnés de liste 1 vers liste 2
5	JButton	cmd2To1	fait l'inverse
6	JButton	cmdRaz1	vide la liste 1
7	JButton	cmdRaz2	vide la liste 2

L'utilisateur tape du texte dans le champ (1) qu'il valide. Se produit alors l'événement *actionPerformed* sur le champ de saisie qui est utilisé pour ajouter le texte saisi dans la liste 1. Voici le code utile pour cette première fonction :

```
public class interfaceAppli extends JFrame {
  JPanel contentPane;
  JLabel jLabel1 = new JLabel();
  JLabel jLabel2 = new JLabel();
  JLabel jLabel3 = new JLabel();
  JTextField txtSaisie = new JTextField();
  JButton cmd1To2 = new JButton();
  JButton cmd2To1 = new JButton();
  DefaultListModel v1=new DefaultListModel();
  DefaultListModel v2=new DefaultListModel();
  JList jList1 = new JList(v1);
  JList jList2 = new JList(v2);
  JScrollPane jScrollPane1 = new JScrollPane();
```

```
  JScrollPane jScrollPane2 = new JScrollPane();
  JButton cmdRaz1 = new JButton();
  JButton cmdRaz2 = new JButton();
  JLabel jLabel4 = new JLabel();

  /**Construire le cadre*/
  public interfaceAppli() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }//interfaceAppli

  /**Initialiser le composant*/
  private void jbInit() throws Exception  {
    ...
    txtSaisie.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        txtSaisie_actionPerformed(e);
      }
    });
    ...
    // Jlist1 est placé dans le conteneur jScrollPane1
    jScrollPane1.getViewport().add(jList1, null);
    // Jlist2 est placé dans le conteneur jScrollPane2
    jScrollPane2.getViewport().add(jList2, null);
    ...
  }
  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
  protected void processWindowEvent(WindowEvent e) {
...
  }

  void txtSaisie_actionPerformed(ActionEvent e) {
    // le texte de la saisie a été validé
    // on le récupère débarrassé de ses espaces de début et fin
    String texte=txtSaisie.getText().trim();
    // s'il est vide, on n'en veut pas
    if(texte.equals("")){
      // msg d'erreur
      JOptionPane.showMessageDialog(this,"Vous devez taper un texte",
        "Erreur",JOptionPane.WARNING_MESSAGE);
      // fin
      return;
    }//if
    // s'il n'est pas vide, on l'ajoute aux valeurs de la liste 1
    v1.addElement(texte);
    // et on vide le champ de saisie
    txtSaisie.setText("");
  }// txtSaisie_actionperformed
}//classe
```

Le code de transfert des éléments sélectionnés d'une liste vers l'autre est le suivant :

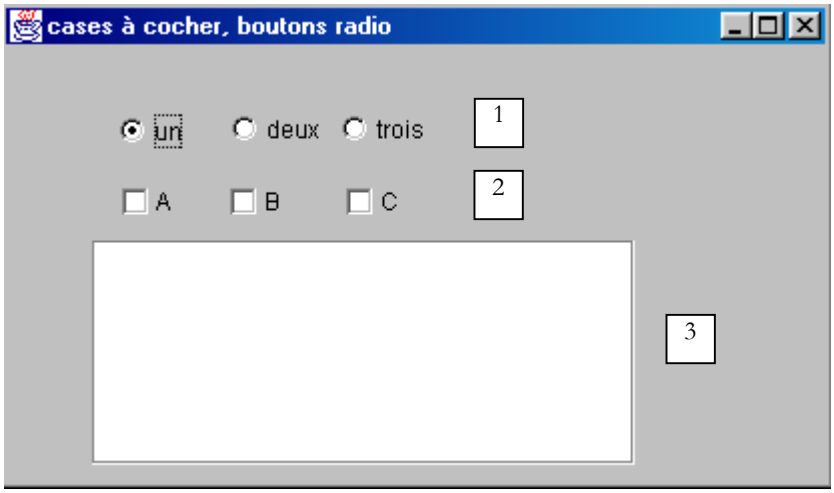
```
  void cmd1To2_actionPerformed(ActionEvent e) {
    // transfert des éléments sélectionnés dans la liste 1 vers la liste 2
    transfert(jList1,jList2);
  }//cmd1To2

  void cmd2To1_actionPerformed(ActionEvent e) {
    // transfert des éléments sélectionnés dans jList2 vers jList1
      transfert(jList2,jList1);
  }//cmd2TO1

  private void transfert(JList L1, JList L2){
      // transfert des éléments sélectionnés dans la liste 1 vers la liste 2
      // on récupère le tableau des indices des éléments sélectionnés dans L1
      int[] indices=L1.getSelectedIndices();
      // qq chose à faire ?
      if (indices.length==0) return;
      // on récupère les valeurs de L1
      DefaultListModel v1=(DefaultListModel)L1.getModel();
      // et celles de L2
      DefaultListModel v2=(DefaultListModel)L2.getModel();
      for(int i=indices.length-1;i>=0;i--){
        // on ajoute à L2 les valeurs sélectionnées dans L1
        v2.addElement(v1.elementAt(indices[i]));
        // les éléments de L1 copiés dans L2 doivent être supprimés de L1
        v1.removeElementAt(indices[i]);
      }//for
  }//transfert
```

Le code associé aux boutons *Raz* est des plus simples :

```
void cmdRaz1_actionPerformed(ActionEvent e) {
  // vide liste 1
  v1.removeAllElements();
}//cmd Raz1

void cmdRaz2_actionPerformed(ActionEvent e) {
  // vide liste 2
  v2.removeAllElements();
}///cmd Raz2
```

4.2.5.4 *Cases à cocher JCheckBox, boutons radio JButtonRadio*

Nous nous proposons d'écrire l'application suivante :

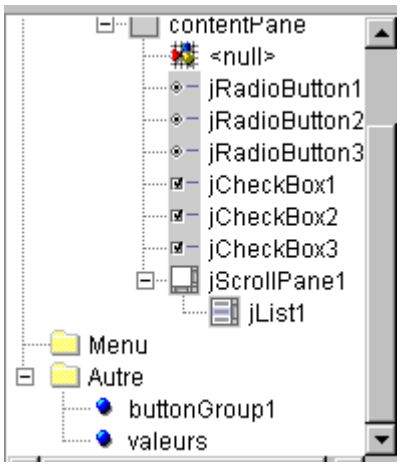


Les composants de la fenêtre sont les suivants :

n°	type	nom	rôle
1	JButtonRadio	jButtonRadio1 jButtonRadio2 jButtonRadio3	3 boutons radio faisant partie du groupe buttonGroup1
2	JCheckBox	jCheckBox1 jCheckBox2 jCheckBox3	3 cases à cocher
3	JList	jList1	une liste dans un conteneur jScrollPane1
4	ButtonGroup	buttonGroup1	composant non visible - sert à regrouper les 3 boutons radio afin que lorsque l'un d'eux s'allume, les autres s'éteignent.

Un groupe de boutons radio peut se construire de la façon suivante :

- on place chacun des boutons radio sans se soucier de les regrouper
- on place dans le conteneur, un composant swing *ButtonGroup*. Ce composant est non visuel. Il n'apparaît donc pas dans le concepteur de la fenêtre. Il apparaît cependant dans sa structure :



On voit ci-dessus dans la branche *Autre* les attributs non visuels de la fenêtre. Une fois un groupe de boutons radio créé, on peut lui associer chacun des boutons radio. Pour ce faire, on sélectionne les propriétés du bouton radio :

name	jRadioButton1
constraints	null
buttonGroup	buttonGroup1
actionCommand	un
alignmentX	0.0
alignmentY	0.5
background	LightGray

et dans la propriété *buttonGroup* du bouton radio, on met le nom du groupe dans lequel on veut mettre le bouton radio, ici *buttonGroup1*. On répète cette opération pour les 3 boutons radio.

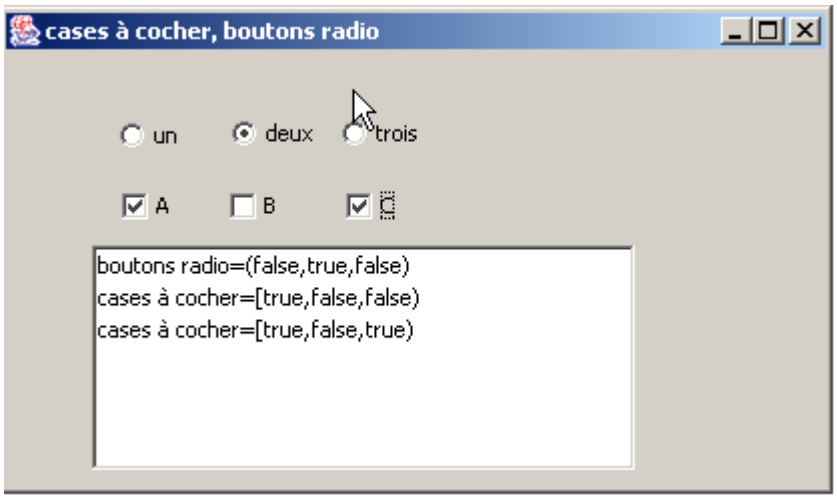
La principale méthode des boutons radio et cases à cocher est la méthode *isSelected()* qui indique si la case ou le bouton est coché. Le texte associé au composant peut être connu avec *getText()* et fixé avec *setText(String unTexte)*. La case/bouton radio peut être coché avec la méthode *setSelected(boolean value)*.

Lors d'un clic sur un bouton radio ou case à cocher, c'est l'événement *actionPerformed* qui est déclenché. Dans le code qui suit, nous utilisons cet événement pour suivre les changements de valeurs des boutons radio et cases à cocher :

```
public class interfaceAppli extends JFrame {
  JPanel contentPane;
  JRadioButton jRadioButton1 = new JRadioButton();
  JRadioButton jRadioButton2 = new JRadioButton();
  JRadioButton jRadioButton3 = new JRadioButton();
  JCheckBox jCheckBox1 = new JCheckBox();
  JCheckBox jCheckBox2 = new JCheckBox();
  JCheckBox jCheckBox3 = new JCheckBox();
  ButtonGroup buttonGroup1 = new ButtonGroup();
  JScrollPane jScrollPane1 = new JScrollPane();
  DefaultListModel valeurs=new DefaultListModel();
  JList jList1 = new JList(valeurs);

  /**Construire le cadre*/
  public interfaceAppli() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  /**Initialiser le composant*/
  private void jbInit() throws Exception  {
    jRadioButton1.setSelected(true);
    jRadioButton1.setText("un");
    jRadioButton1.setBounds(new Rectangle(57, 31, 49, 23));
    jRadioButton1.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        afficheRadioButtons(e);
      }
    });
```

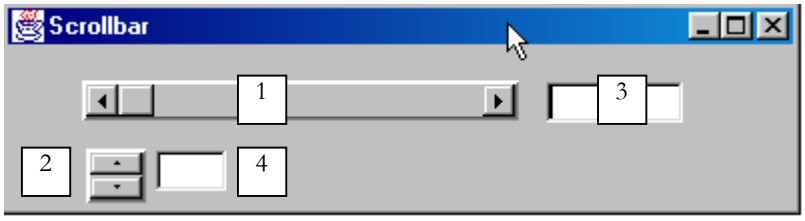
```
    jRadioButton2.setBounds(new Rectangle(113, 30, 49, 23));
    jRadioButton2.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        afficheRadioButtons(e);
      }
    });
    jRadioButton2.setText("deux");

    jRadioButton3.setBounds(new Rectangle(168, 30, 49, 23));
    jRadioButton3.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        afficheRadioButtons(e);
      }
    });
    jRadioButton3.setText("trois");

// les boutons radio sont regroupés
    buttonGroup1.add(jRadioButton1);
    buttonGroup1.add(jRadioButton2);
    buttonGroup1.add(jRadioButton3);

// cases à cocher
    jCheckBox1.setText("A");
    jCheckBox1.setBounds(new Rectangle(58, 69, 32, 17));
    jCheckBox1.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        afficheCases(e);
      }
    });

    jCheckBox2.setBounds(new Rectangle(112, 69, 40, 17));
    jCheckBox2.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        afficheCases(e);
      }
    });
    jCheckBox2.setText("B");

    jCheckBox3.setText("C");
    jCheckBox3.setBounds(new Rectangle(170, 69, 37, 17));
    jCheckBox3.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        afficheCases(e);
      }
    });

  ....
  }
  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
  protected void processWindowEvent(WindowEvent e) {
    ...
  }

  private void afficheRadioButtons(ActionEvent e){
    // affiche les valeurs des 3 boutons radio
    valeurs.addElement("boutons radio=("+jRadioButton1.isSelected()+","+
      jRadioButton2.isSelected()+","+jRadioButton3.isSelected()+")");
  }//afficheRadioButtons

  void afficheCases(ActionEvent e) {
    // affiche les valeurs des cases à cocher
    valeurs.addElement("cases à cocher=["+jCheckBox1.isSelected()+","+
      jCheckBox2.isSelected()+","+jCheckBox3.isSelected()+")");
  }//afficheCases

}//classe
```

Voici un exemple d'exécution :



4.2.5.5 composant JScrollBar

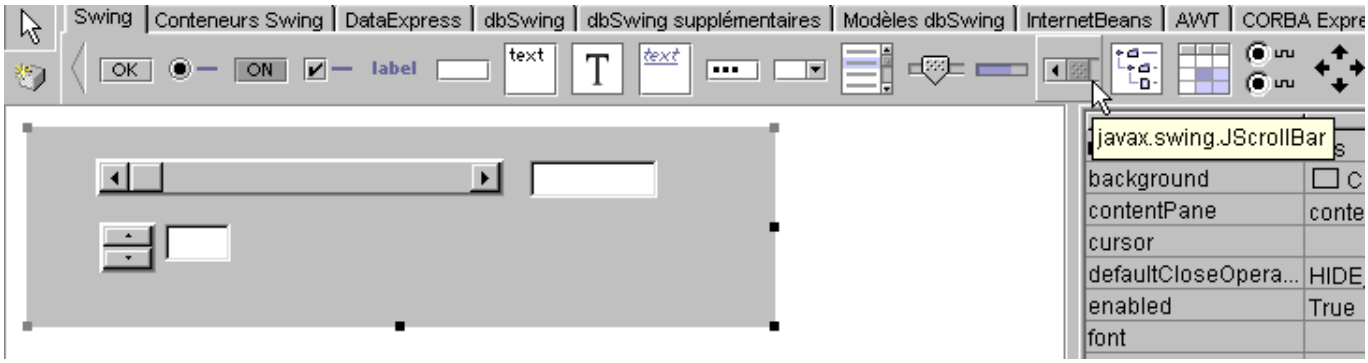
Réalisons l'application suivante :



n°	type	nom	rôle
1	JScrollBar	jScrollBar1	un variateur horizontal
2	JScrollBar	jScrollBar2	un variateur vertical
3	JTextField	txtvaleurHS	affiche la valeur du variateur horizontal 1 - permet aussi de fixer cette valeur
4	JTextField	txtvaleurVS	affiche la valeur du variateur vertical 2 - permet aussi de fixer cette valeur

- Un variateur *JScrollBar* permet à l'utilisateur de choisir une valeur dans une plage de valeurs entières symbolisée par la "bande" du variateur sur laquelle se déplace un curseur.
- Pour un variateur horizontal, l'extrémité gauche représente la valeur minimale de la plage, l'extrémité droite la valeur maximale, le curseur la valeur actuelle choisie. Pour un variateur vertical, le minimum est représenté par l'extrémité haute, le maximum par l'extrémité basse. Le couple (min,max) vaut par défaut (0,100).
- Un clic sur les extrémités du variateur fait varier la valeur d'un incrément (positif ou négatif) selon l'extrémité cliquée appelée *unitIncrement* qui est par défaut 1.
- Un clic de part et d'autre du curseur fait varier la valeur d'un incrément (positif ou négatif) selon l'extrémité cliquée appelée *blockIncrement* qui est par défaut 10.
- Ces cinq valeurs (min, max, valeur, unitIncrement, blockIncrement) peuvent être connues avec les méthodes *getMinimum(), getMaximum(), getValue(), getUnitIncrement(), getBlockIncrement()* qui toutes rendent un entier et peuvent être fixées par les méthodes *setMinimum(int min), setMaximum(int max), setValue(int val), setUnitIncrement(int uInc), setBlockIncrement(int bInc)*

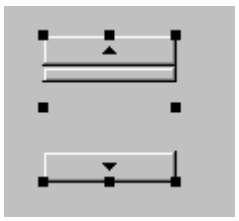
Il y a quelques points à connaître dans l'utilisation des composants *JScrollBar*. Tout d'abord, on le trouve dans la barre des composants swing :



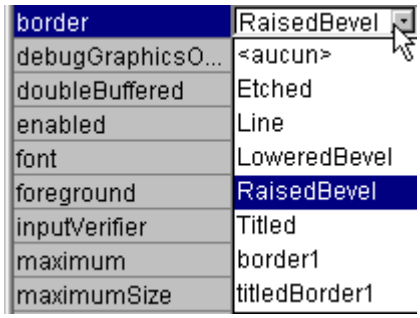
Lorsqu'on le dépose sur le conteneur, il est par défaut vertical. On le rend horizontal avec la propriété *orientation* ci-dessous:

blockIncrement	10
border	border1
debugGraphicsO...	0
doubleBuffered	False
enabled	True
font	"Dialog", 0, 12
foreground	Black
inputVerifier	
maximum	100
maximumSize	32767, 16
minimum	0
minimumSize	5, 16
model	
nextFocusableCo...	
opaque	True
orientation	HORIZONTAL
preferredSize	48, 16
requestFocusEn...	False
toolTipText	
unitIncrement	1
value	0

Sur la feuille de propriétés ci-dessus on voit qu'on a accès aux propriétés *minimum*, *maximum*, *value*, *unitIncrement*, *blockIncrement* du *JScrollbar*. On peut donc les fixer à la conception. Lorsqu'on place un scrollbar sur le conteneur sa "bande de variation" n'apparaît pas :



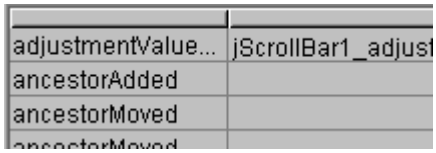
On peut régler ce problème en donnant une bordure au composant. Cela se fait avec sa propriété *border* qui peut avoir différentes valeurs :



Voici ce que donne par exemple *RaisedBevel* :

Lorsqu'on clique sur l'extrémité supérieure d'un variateur vertical, sa valeur diminue. Cela peut surprendre l'utilisateur moyen qui s'attend normalement à voir la valeur "monter". On règle ce problème en donnant une valeur négative à *unitIncrement* et *blockIncrement*.

Comment suivre les évolutions d'un variateur ? Lorsque la valeur de celui-ci change, l'événement *adjustmentValueChanged* se produit. Il suffit d'associer une procédure à cet événement pour être informé de chaque variation de la valeur du scrollbar.



Le code utile de notre application est le suivant :

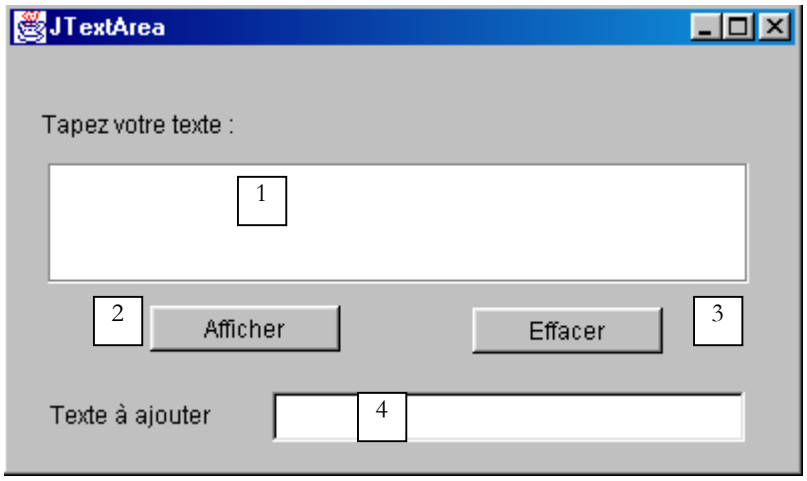
```
....

public class cadreAppli extends JFrame {
  JPanel contentPane;
  JScrollBar jScrollBar1 = new JScrollBar();
  Border border1;
  JTextField txtValeurHS = new JTextField();
  JScrollBar jScrollBar2 = new JScrollBar();
  JTextField txtValeurVS = new JTextField();
  TitledBorder titledBorder1;

  /**Construire le cadre*/
  public cadreAppli() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  /**Initialiser le composant*/
  private void jbInit() throws Exception  {
...

// une bordure pour les scrollbars
    border1 = BorderFactory.createBevelBorder(BevelBorder.RAISED,Color.white,Color.white,new Color(134,
134, 134),new Color(93, 93, 93));
    // pas de titre bour la bordure
    titledBorder1 = new TitledBorder("");

    jScrollBar1.setOrientation(JScrollBar.HORIZONTAL);
    jScrollBar1.setBorder(BorderFactory.createRaisedBevelBorder());
    jScrollBar1.setAutoscrolls(true);
    jScrollBar1.setBounds(new Rectangle(37, 17, 218, 20));
    jScrollBar1.addAdjustmentListener(new java.awt.event.AdjustmentListener() {
      public void adjustmentValueChanged(AdjustmentEvent e) {
        jScrollBar1_adjustmentValueChanged(e);
```

```
      }
    });

    txtValeurHS.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        txtValeurHS_actionPerformed(e);
      }
    });

    jScrollBar2.setBounds(new Rectangle(39, 51, 30, 27));
    jScrollBar2.addAdjustmentListener(new java.awt.event.AdjustmentListener() {
      public void adjustmentValueChanged(AdjustmentEvent e) {
        jScrollBar2_adjustmentValueChanged(e);
      }
    });
    jScrollBar2.setAutoscrolls(true);
    jScrollBar2.setUnitIncrement(-1);
    jScrollBar2.setBorder(BorderFactory.createRaisedBevelBorder());

    txtValeurVS.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        txtValeurVS_actionPerformed(e);
      }
    });

  ......
  }

  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
  protected void processWindowEvent(WindowEvent e) {
...
  }

  void jScrollBar1_adjustmentValueChanged(AdjustmentEvent e) {
    // la valeur du scrollbar 1 a changé
    txtValeurHS.setText(""+jScrollBar1.getValue());
  }

  void jScrollBar2_adjustmentValueChanged(AdjustmentEvent e) {
    // la valeur du scrollbar 2 a changé
    txtValeurVS.setText(""+jScrollBar2.getValue());
  }

  void txtValeurHS_actionPerformed(ActionEvent e) {
    // on fixe la valeur du scrollbar horizontal
    setValeur(jScrollBar1,txtValeurHS);
  }

  void txtValeurVS_actionPerformed(ActionEvent e) {
    // on fixe la valeur du scrollbar vertical
    setValeur(jScrollBar2,txtValeurVS);
  }

  private void setValeur(JScrollBar jS, JTextField jT){
    // fixe la valeur du scrollbar jS avec le texte du champ jT
    int valeur=0;
    try{
      valeur=Integer.parseInt(jT.getText());
      jS.setValue(valeur);
    }
    catch (Exception e){
      // on affiche l'erreur
      afficher(""+e);
    }//try-catch
  }//setValeur

  void afficher(String message){
    // affiche un message dans une boîte
    JOptionPane.showMessageDialog(this,message,"Menus",JOptionPane.INFORMATION_MESSAGE);
  }//afficher

}
```

Voici un exemple d'exécution :

4.2.5.6 composant JTextArea

Le composant *JTextArea* est un composant où l'on peut entrer plusieurs lignes de texte contrairement au composant *JTextField* où l'on ne peut entrer qu'une ligne. Si ce composant est placé dans un conteneur défilant (*JScrollPane*) on a une champ de saisie de texte défilant. Ce type de composant pourrait se rencontrer par exemple dans une application de courrier électronique où le texte du message à envoyer serait tapé dans un composant *JTextArea*. Les méthodes usuelles sont *String getText()* pour connaître le contenu de la zone de texte, *setText(String unTexte)* pour mettre *unTexte* dans la zone de texte, *append(String unTexte)* pour ajouter *unTexte* au texte déjà présent dans la zone de texte. Considérons l'application suivante :



n°	type	nom	rôle
1	JTextArea	txtTexte	une zone de texte multilignes
2	JButton	cmdAfficher	affiche le contenu de 1 dans une boîte de dialogue
3	JButton	cmdEffacer	efface le contenu de 1
4	JTextField	txtAjout	texte ajouté au texte de 1 lorsqu'il est validé par la touche Entrée.
5	JScrollPane	jScrollPane1	conteneur défilant dans lequel on a placé la zone de texte 1 afin d'avoir une zone de texte défilante.

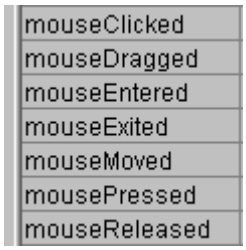
Le code utile est le suivant :

```
.....
public class cadreAppli extends JFrame {
  JPanel contentPane;
  JLabel jLabel1 = new JLabel();
  JButton cmdAfficher = new JButton();
  JScrollPane jScrollPane1 = new JScrollPane();
  JTextArea txtTexte = new JTextArea();
  JLabel jLabel2 = new JLabel();
  JTextField txtAjout = new JTextField();
  JButton cmdEffacer = new JButton();

  /**Construire le cadre*/
  public cadreAppli() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  /**Initialiser le composant*/
  private void jbInit() throws Exception  {

    cmdAfficher.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        cmdAfficher_actionPerformed(e);
      }
    });
    txtAjout.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        txtAjout_actionPerformed(e);
      }
    });
    cmdEffacer.addActionListener(new java.awt.event.ActionListener() {
```

```
      public void actionPerformed(ActionEvent e) {
        cmdEffacer_actionPerformed(e);
      }
    });
  .......
    jScrollPane1.getViewport().add(txtTexte, null);
  }
  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
  protected void processWindowEvent(WindowEvent e) {
    ........
  }

  void cmdAfficher_actionPerformed(ActionEvent e) {
    // affiche le contenu du TextArea
    afficher(txtTexte.getText());
  }

    void afficher(String message){
    // affiche un message dans une boîte
    JOptionPane.showMessageDialog(this,message,"Suivi",JOptionPane.INFORMATION_MESSAGE);
  }// afficher

  void cmdEffacer_actionPerformed(ActionEvent e) {
    txtTexte.setText("");
  }//cmdEffacer_actionPerformed

  void txtAjout_actionPerformed(ActionEvent e) {
    // ajout de texte
    txtTexte.append(txtAjout.getText());
    // raz ajout
    txtAjout.setText("");
  }//
}
```

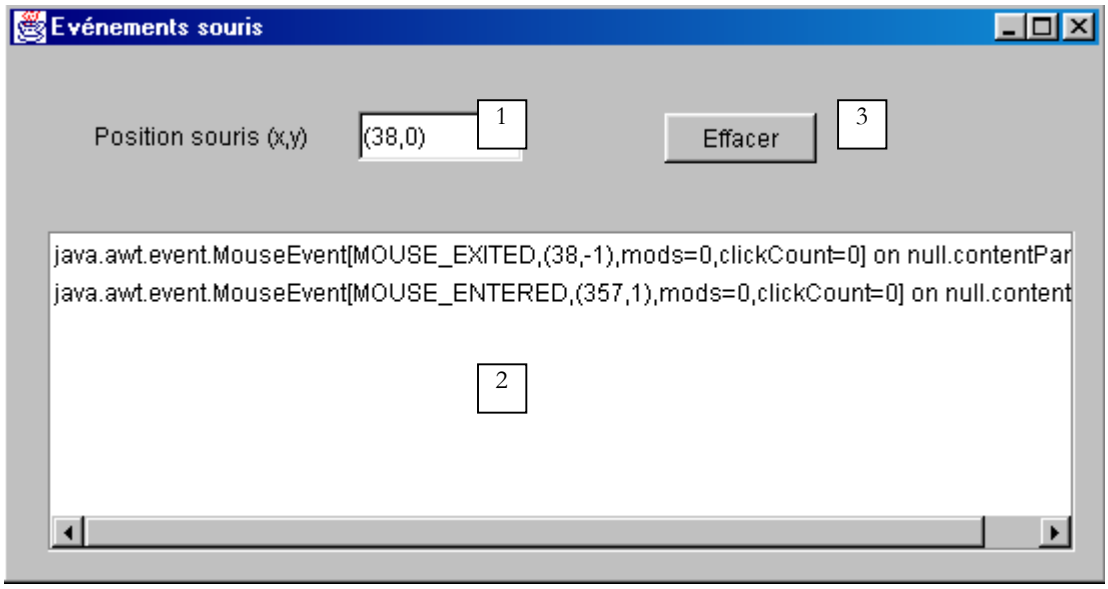
4.2.6 Événements souris

Lorsqu'on dessine dans un conteneur, il est important de connaître la position de la souris pour par exemple afficher un point lors d'un clic. Les déplacements de la souris provoquent des événements dans le conteneur dans lequel elle se déplace. Voici par exemple ceux proposés par JBuilder pour un conteneur *JPanel* :



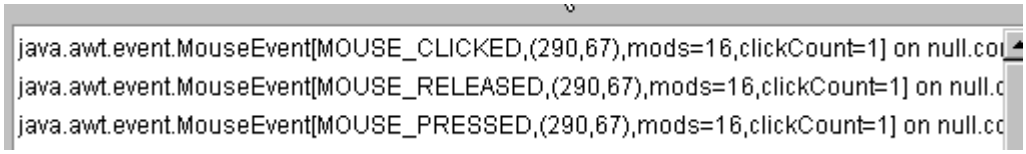
mouseClicked	clic de souris
mouseDragged	la souris se déplace, bouton gauche appuyé
mouseEntered	la souris vient d'entrer dans la surface du conteneur
mouseExited	la souris vient de quitter la surface du conteneur
mouseMoved	la souris bouge
mousePressed	Pression sur le bouton gauche de la souris
mouseReleased	Relâchement du bouton gauche de la souris

Voici un programme permettant de mieux appréhender à quels moments se produisent les différents événements souris :



n°	type	nom	rôle
1	JTextField	txtPosition	pour afficher la position de la souris dans le conteneur (évt MouseMoved)
2	JList	lstAffichage	pour afficher les évts souris autres que MouseMoved
3	JButton	cmdEffacer	pour effacer le contenu de 2

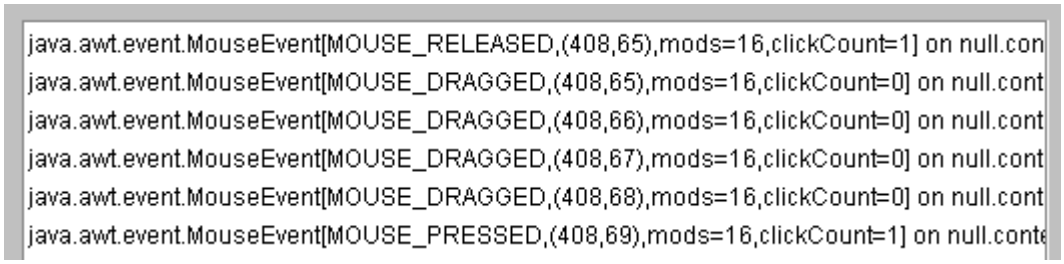
Lorsqu'on exécute ce programme, voici ce qu'on obtient pour un clic :



Les événements sont empilés par le haut dans la liste. Aussi la copie d'écran ci-dessus indique qu'un clic provoque trois événements, dans l'ordre :

1. *MousePressed* lorsqu'on appuie sur le bouton
2. *MouseReleased* lorsqu'on le lâche
3. *MouseClicked* qui indique que la succession des deux événements précédents est considérée comme un clic. Ce pourrait être un double clic. Mais ci-dessus, l'information *clickCount=1* indiqe que c'est un simple clic.

Maintenant si on appuie sur le bouton, déplace la souris et relâche le bouton :



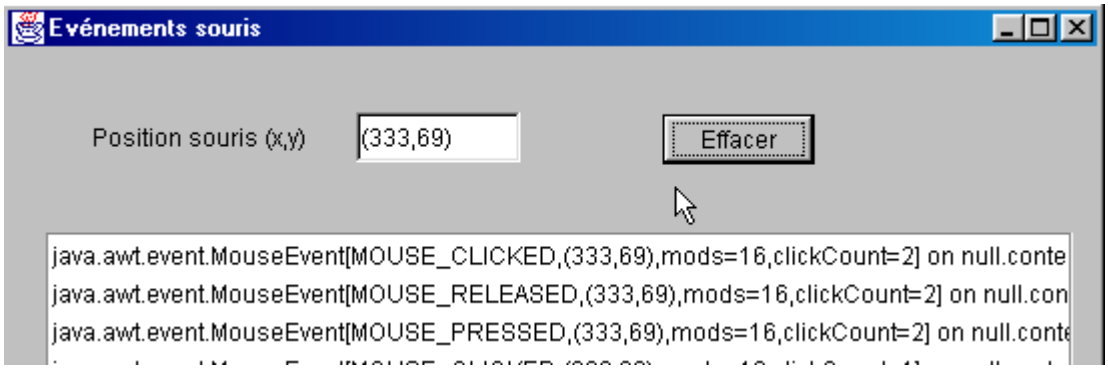
On voit là les trois événements :

1. *MousePressed* lorsqu'on appuie initialement sur le bouton
2. *MouseDragged* lorsqu'on déplace la souris, bouton appuyé
3. *MouseReleased* lorsqu'on lâche le bouton

Dans les deux exemples ci-dessus, on voit qu'un événement souris ramène avec lui diverses informations dont les coordonnées (x,y) de la souris, par exemple (408,65) dans la première ligne ci-dessus.

Si on continue ainsi, on découvre que l'événement *MouseExited* est déclenché dès que la souris quitte le conteneur ou passe sur l'un des composants de celui-ci. Dans ce dernier cas, le conteneur reçoit l'événement *MouseExited* et le composant l'événement *MouseEntered*. L'inverse se produira lorsque la souris quittera le composant pour revenir sur le conteneur.

Que se passe-t-il lors d'un double clic ?



On a exactement les mêmes événements que pour un clic simple. Seulement l'événement rapporte avec lui l'information clickCount=2 (cf ci-dessus) indiquant qu'il y a eu en fait un double clic.

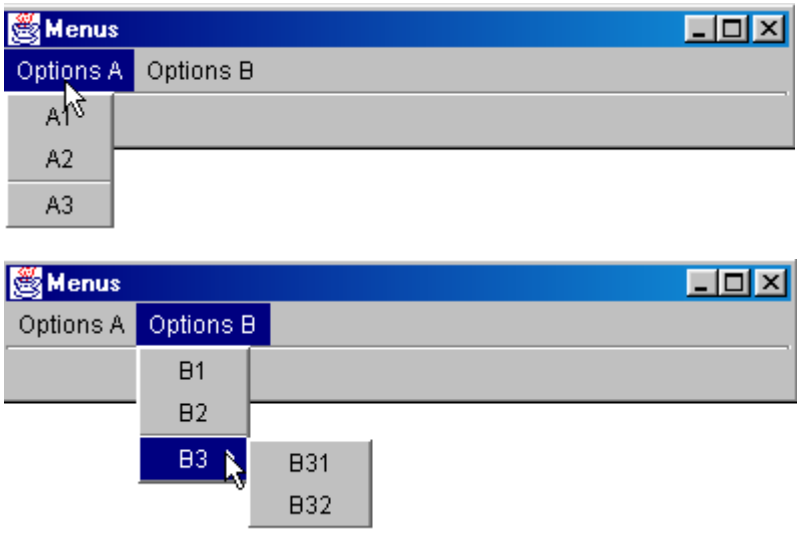
Le code utile de cette application est le suivant :

```
public class Cadre1 extends JFrame {
  JPanel contentPane;
  JLabel jLabel1 = new JLabel();
  JTextField txtPosition = new JTextField();
  JScrollPane jScrollPane1 = new JScrollPane();
  DefaultListModel valeurs=new DefaultListModel();
  JList lstAffichage = new JList(valeurs);
  JButton cmdEffacer = new JButton();

  /**Construire le cadre*/
  public Cadre1() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  /**Initialiser le composant*/
  private void jbInit() throws Exception  {

    contentPane.addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
      public void mouseMoved(MouseEvent e) {
        contentPane_mouseMoved(e);
      }
      public void mouseDragged(MouseEvent e) {
        contentPane_mouseDragged(e);
      }
    });

    contentPane.addMouseListener(new java.awt.event.MouseAdapter() {
      public void mouseEntered(MouseEvent e) {
        contentPane_mouseEntered(e);
      }
      public void mouseExited(MouseEvent e) {
        contentPane_mouseExited(e);
      }
      public void mousePressed(MouseEvent e) {
        contentPane_mousePressed(e);
      }
      public void mouseClicked(MouseEvent e) {
        contentPane_mouseClicked(e);
      }
      public void mouseReleased(MouseEvent e) {
        contentPane_mouseReleased(e);
      }
    });

    cmdEffacer.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        cmdEffacer_actionPerformed(e);
      }
    });
```

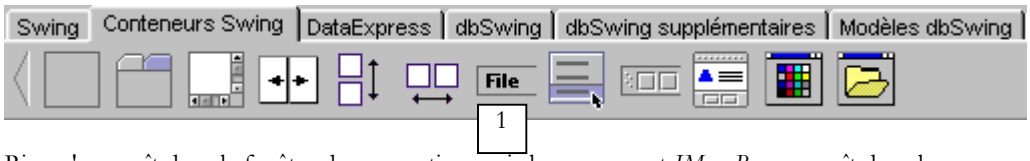
```
    jScrollPane1.getViewport().add(lstAffichage, null);

...............
  }
  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
  protected void processWindowEvent(WindowEvent e) {
  .........
  }

  void contentPane_mouseMoved(MouseEvent e) {
    txtPosition.setText("("+e.getX()+","+e.getY()+")");
  }

  void contentPane_mouseDragged(MouseEvent e) {
    afficher(e);
  }

  void contentPane_mouseEntered(MouseEvent e) {
    afficher(e);
  }

  void afficher(MouseEvent e){
    // affiche l'évt e dans la liste lstAffichage
    valeurs.insertElementAt(e,0);
  }

  void contentPane_mouseExited(MouseEvent e) {
    afficher(e);
  }

  void contentPane_mousePressed(MouseEvent e) {
    afficher(e);
  }

  void contentPane_mouseClicked(MouseEvent e) {
    afficher(e);
  }

  void contentPane_mouseReleased(MouseEvent e) {
    afficher(e);
  }

  void cmdEffacer_actionPerformed(ActionEvent e) {
    // efface la liste
    valeurs.removeAllElements();
  }
}
```

4.2.7 Créer une fenêtre avec menu

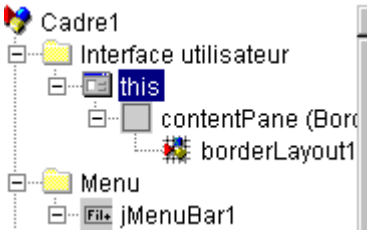
Voyons maintenant comment créer une fenêtre avec menu avec Jbuilder. Nous allons créer la fenêtre suivante :



Créez un nouveau projet avec au départ une fenêtre vide. Choisissez dans la liste des composants "Conteneurs Swing" le composant *JMenuBar* (cf 1 ci-dessous) et déposez-le sur la fenêtre en cours de conception.



Rien n'apparaît dans la fenêtre de conception mais le composant *JMenuBar* apparaît dans la panneau de structure de votre fenêtre :

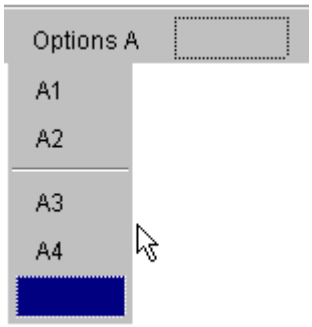


Double-cliquez sur l'élément *jMenuBar1* ci-dessus pour avoir accès au menu en mode conception :

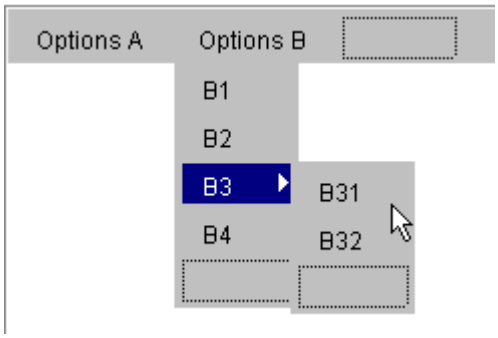
Une barre d'outils est disponible pour construire l'ensemble des options du menu :

1 insérer un élément de menu
2 insérer un séparateur
3 insérer un menu imbriqué
4 supprimer un élément de menu
5 désactiver un élément de menu
6 élément de menu à cocher
7 inverser le bouton radio

Pour créer votre premier élément de menu, tapez "Options A" dans la case A ci-dessus, puis dessous dans l'ordre : A1, A2, séparateur, A3, A4.

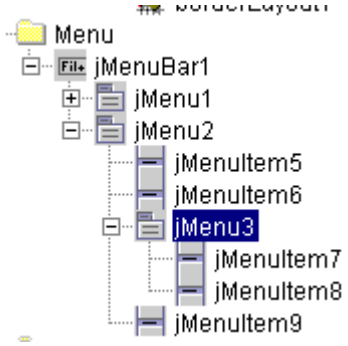


Puis à côté :



Utilisez l'outil 3 pour indiquer que B3 est un sous-menu imbriqué.

Au fur et à mesure de la conception du menu, la structure logique de notre fenêtre évolue :



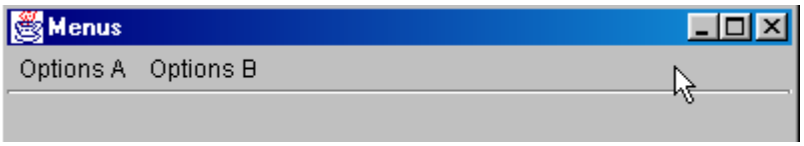
Si nous exécutons notre application maintenant, nous verrons une fenêtre vide sans menu. Il nous faut associer le menu créé à notre fenêtre. Pour ce faire, dans la structure de la fenêtre, sélectionnez l'objet *this* :

Vous avez alors accès aux propriétés de *this* :

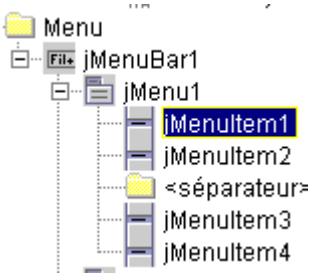
name	this
background	Control
contentPane	contentPane
cursor	
defaultCloseOpera...	HIDE_ON_CLOSE
enabled	True
font	
foreground	Black
iconImage	
JMenuBar	jMenuBar1
locale	<défaut>
resizable	True
state	NORMAL
title	Menus

L'une de celles-ci est *JMenuBar* qui sert à fixer le menu qui sera associé à la fenêtre. Cliquez dans la cellule à droite de *JMenuBar*. Tous les menus créés vous seront alors proposés. Ici, nous n'aurons que *jMenuBar1*. Sélectionnez-le.

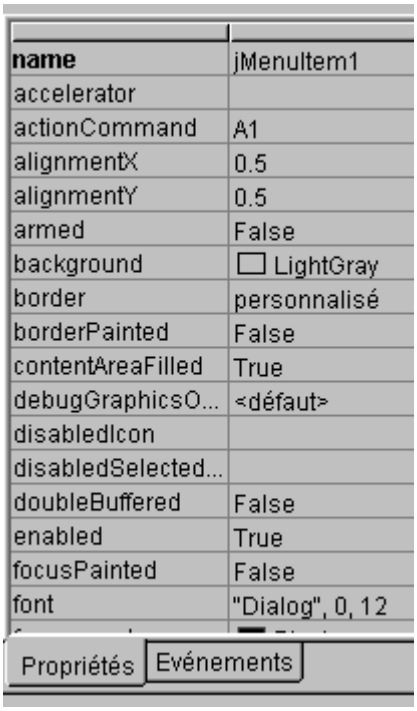
Lancez l'exécution de l'application (F9) :



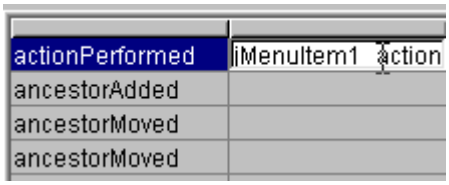
Maintenant, on a un menu mais les options ne font, pour l'instant, rien. Les options de menu sont traitées comme des composants : elles ont des propriétés et des événements. Dans la structure du menu, sélectionnez l'option *jMenuItem1* :



Vous avez alors accès à ses propriétés et événements :



Sélectionnez la page des événements et cliquez sur la cellule à droite de l'événement *actionPerformed* : c'est l'événement qui se produit lorsqu'on clique sur un élément de menu. Une procédure de traitement vous est proposée par défaut. Double-cliquez dessus pour avoir accès à son code :

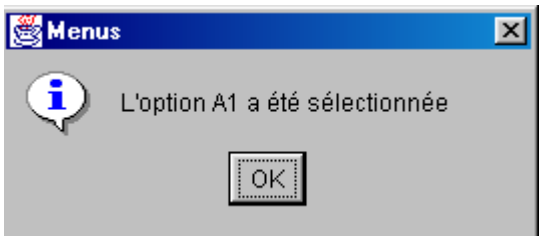


Nous écrirons le simple code suivant :

```
void jMenuItem1_actionPerformed(ActionEvent e) {
  afficher("L'option A1 a été sélectionnée");
}

void afficher(String message){
  // affiche un message dans une boîte
```

```
    JOptionPane.showMessageDialog(this,message,"Menus",JOptionPane.INFORMATION_MESSAGE);
  }//afficher
```

Exécutez l'application et sélectionnez l'option A1 pour obtenir le message suivant :



Le code utile de cette application est le suivant :

```
.....
public class Cadre1 extends JFrame {
  JPanel contentPane;
  BorderLayout borderLayout1 = new BorderLayout();
  JMenuBar jMenuBar1 = new JMenuBar();
  JMenu jMenu1 = new JMenu();
  JMenuItem jMenuItem1 = new JMenuItem();
  JMenuItem jMenuItem2 = new JMenuItem();
  JMenuItem jMenuItem3 = new JMenuItem();
  JMenuItem jMenuItem4 = new JMenuItem();
  JMenu jMenu2 = new JMenu();
  JMenuItem jMenuItem5 = new JMenuItem();
  JMenuItem jMenuItem6 = new JMenuItem();
  JMenu jMenu3 = new JMenu();
  JMenuItem jMenuItem7 = new JMenuItem();
  JMenuItem jMenuItem8 = new JMenuItem();
  JMenuItem jMenuItem9 = new JMenuItem();

  /**Construire le cadre*/
  public Cadre1() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  /**Initialiser le composant*/
  private void jbInit() throws Exception  {
    // la fenêtre est associée à un menu
    this.setJMenuBar(jMenuBar1);

    jMenu1.setText("Options A");
    jMenuItem1.setText("A1");
    jMenuItem1.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        jMenuItem1_actionPerformed(e);
      }
    });
    jMenuItem2.setText("A2");
    jMenuItem3.setText("A3");
    jMenuItem4.setText("A4");
    jMenu2.setText("Options B");
    jMenuItem5.setText("B1");
    jMenuItem6.setText("B2");
    jMenu3.setText("B3");
    jMenuItem7.setText("B31");
    jMenuItem8.setText("B32");
    jMenuItem9.setText("B4");
    jMenuBar1.add(jMenu1);
    jMenuBar1.add(jMenu2);
    jMenu1.add(jMenuItem1);
    jMenu1.add(jMenuItem2);
    jMenu1.addSeparator();
    jMenu1.add(jMenuItem3);
    jMenu1.add(jMenuItem4);
    jMenu2.add(jMenuItem5);
    jMenu2.add(jMenuItem6);
    jMenu2.add(jMenu3);
    jMenu2.add(jMenuItem9);
    jMenu3.add(jMenuItem7);
    jMenu3.add(jMenuItem8);
  }
  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
```

```
  protected void processWindowEvent(WindowEvent e) {
....
  }

  void jMenuItem1_actionPerformed(ActionEvent e) {
    afficher("L'option A1 a été sélectionnée");
  }

  void afficher(String message){
    // affiche un message dans une boîte
    JOptionPane.showMessageDialog(this,message,"Menus",JOptionPane.INFORMATION_MESSAGE);
  }//afficher

}
```

4.3 Boîtes de dialogue

4.3.1 Boîtes de message

Nous avons déjà utilisé la classe *JOptionPane* afin d'afficher des messages. Ainsi le code suivant :

```
import javax.swing.*;

public class dialog1 {
  public static void main(String arg[]){
    JOptionPane.showMessageDialog(null,"Un message","Titre de la boîte",JOptionPane.INFORMATION_MESSAGE);
  }
}
```

produit l'affichage de la boîte suivante :

Lorsqu'on ferme cette fenêtre, elle disparaît mais le thread d'exécution dans laquelle elle s'exécutait lui n'est pas arrêté. Ce phénomène ne se produit normalement pas. Les boîtes de dialogue sont utilisées au sein d'une application qui à un moment ou à un autre utilise une instruction *System.exit(n)* pour arrêter tous les threads. On s'en souviendra dans les exemples à venir tous bâtis sur le même modèle. Sous Dos, l'application peut être interrompue par Ctrl-C. Avec JBuilder, on utilisera l'option *Exécuter/Réinitialiser le programme (Ctrl-F2)*. Par ailleurs, le premier argument de *showMessageDialog* est ici *null*. En général ce n'est pas le cas. C'est plutôt *this* où *this* désigne la fenêtre principale de l'application.

4.3.2 Looks and Feels

L'apparence de la boîte ci-dessus pourrait être différente. Il est possible de paramétrer cette apparence via la classe *javax.swing.UIManager*. Lorsque nous avons commenté le code généré par JBuilder pour notre première fenêtre nous avons rencontré une instruction sur laquelle nous ne nous sommes pas attardés :

```
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

La méthode *setLookAndFeel* de la classe *UIManager* (UI=User Interface) permet de fixer l'apparence des interfaces graphiques. La classe *UIManager* dispose d'une méthode permettant de connaître les "apparences" possibles pour les interfaces :

`static UIManager.LookAndFeelInfo[]`	**`getInstalledLookAndFeels`**`()`

La méthode rend un tableau d'éléments de type *LookAndFeelInfo*. Cette classe a une méthode :

`String`	**`getClassName`**`()`

qui donne le nom de la classe "implémentant" une apparence donnée. Essayons le programme suivant :

```
import javax.swing.*;

public class LookAndFeels {
  // affiche les look and feels disponibles
  public static void main(String[] args) {
    // iste des look and feels installés
    UIManager.LookAndFeelInfo[] lf=UIManager.getInstalledLookAndFeels();
    // affichage
    for(int i=0;i<lf.length;i++){
      System.out.println(lf[i].getClassName());
    }//for
  }//main
}//classe
```

Il donne les résultats suivants :

```
javax.swing.plaf.metal.MetalLookAndFeel
com.sun.java.swing.plaf.motif.MotifLookAndFeel
com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

Il semble donc y avoir trois "apparences "différentes". Reprenons notre programme d'affichage de messages en essayant les différentes apparences possibles :

```
import javax.swing.*;

public class LookAndFeel2 {
  // affiche les look and feels disponibles
  public static void main(String[] args) {
    // liste des look and feels installés
    UIManager.LookAndFeelInfo[] lf=UIManager.getInstalledLookAndFeels();
    // affichage
    for(int i=0;i<lf.length;i++){
      // gestionnaire d'apparence
      try{
        UIManager.setLookAndFeel(lf[i].getClassName());
      }catch(Exception ex){
        System.err.println(ex.getMessage());
      }//catch
      // message
      JOptionPane.showMessageDialog(null,"Un
message",lf[i].getClassName(),JOptionPane.INFORMATION_MESSAGE);
    }//for
  }//main
}//classe
```

L'exécution donne les affichages suivants :

correspondant de droite à gauche aux "looks" Metal, Motif, Windows.

4.3.3 Boîtes de confirmation

La classe *JOptionPane* a une méthode *showConfirmDialog* pour afficher des boîtes de confirmation avec les boutons *Oui*, *Non*, *Annuler*. Il y a plusieurs méthodes *showConfirmDialog* surchargées. Nous en étudions une :

`static int`	**`showConfirmDialog`**`(Component parentComponent, Object message, String title, int optionType)`

parentComponent le composant parent de la boîte de dialogue. Souvent la fenêtre ou la valeur null
message le message à afficher

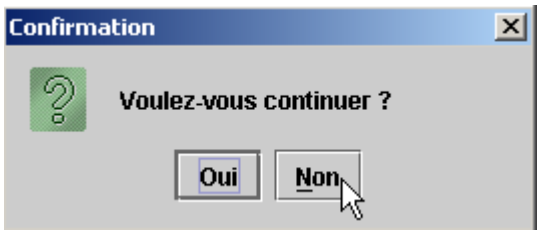
title	le titre de la boîte
optionType	JOptionPane.YES_NO_OPTION : boutons oui, non JOptionPane.YES_NO_CANCEL_OPTION : boutons oui, non, annuler

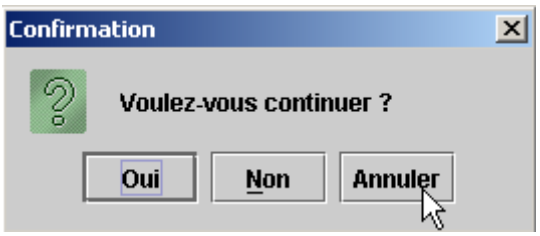
Le résultat rendu par la méthode est :

JOptionPane.YES_OPTION	l'utilisateur a cliqué sur oui
JOptionPane.NO_OPTION	l'utilisateur a cliqué sur non
JOptionPane.CANCEL_OPTION	l'utilisateur a cliqué sur annuler
JOptionPane.CLOSED_OPTION	l'utilisateur a fermé la boîte

Voici un exemple :

```
import javax.swing.*;

public class confirm1 {
  public static void main(String[] args) {
    // affiche des boîtes de confirmation
    int réponse;
    affiche(JOptionPane.showConfirmDialog(null,"Voulez-vous continuer
?","Confirmation",JOptionPane.YES_NO_OPTION));
    affiche(JOptionPane.showConfirmDialog(null,"Voulez-vous continuer
?","Confirmation",JOptionPane.YES_NO_CANCEL_OPTION));
  }//main

  private static void affiche(int réponse){
    // indique quel type de réponse on a eu
    switch(réponse){
      case JOptionPane.YES_OPTION :
        System.out.println("Oui");
        break;
      case JOptionPane.NO_OPTION :
        System.out.println("Non");
        break;
      case JOptionPane.CANCEL_OPTION :
        System.out.println("Annuler");
        break;
      case JOptionPane.CLOSED_OPTION :
        System.out.println("Fermeture");
        break;
    }//switch
  }//affiche
}//classe
```





Sur la console, on obtient l'affichage des messages *Non* et *Annuler*.

4.3.4 Boîte de saisie

La classe *JOptionPane* offre également la possibilité de faire une saisie avec la méthode *showInputDialog*. Là encore, il existe plusieurs méthodes surchargées. Nous présentons l'une d'entre-elles :

`static String`	**`showInputDialog`**`(Component parentComponent, Object message, String title, int messageType)`

Les arguments sont ceux déjà rencontrés plusieurs fois. La méthode rend la chaîne de caractères tapée par l'utilisateur. Voici un exemple :

```
import javax.swing.*;

public class input1 {
  public static void main(String[] args) {
    // saisie
```

```
    System.out.println("Chaîne saisie ["+
      JOptionPane.showInputDialog(null,"Quel est votre nom","Saisie du nom",JOptionPane.QUESTION_MESSAGE
)
      + "]");
  }//main
}//classe
```

L'affichage de la boîte de saisie :

L'affichage console :

```
Chaîne saisie [dupont]
```

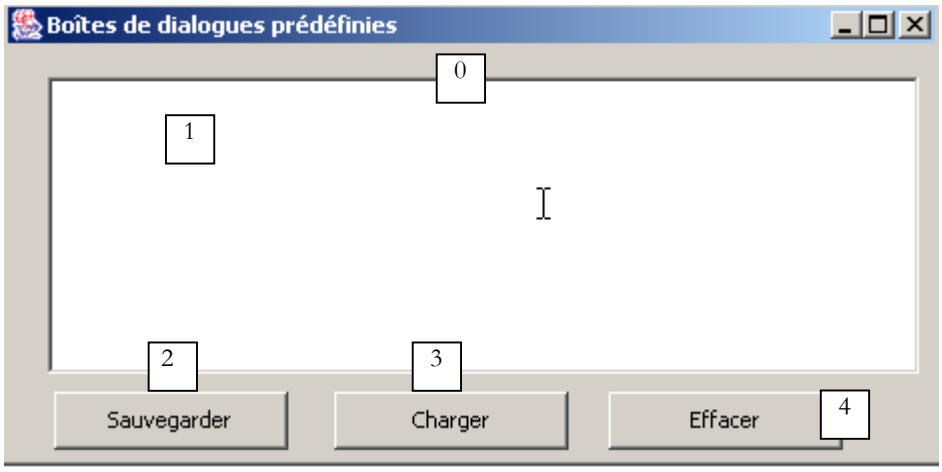
4.4 Boîtes de sélection

Nous nous intéressons maintenant à un certain nombre de boîtes de sélection prédéfinies dans Java 2 :

JFileChooser boîte de sélection permettant de désigner un fichier dans l'arborescence des fichiers
JColorChooser boîte de sélection permettant de choisir une couleur

4.4.1 Boîte de sélection JFileChooser

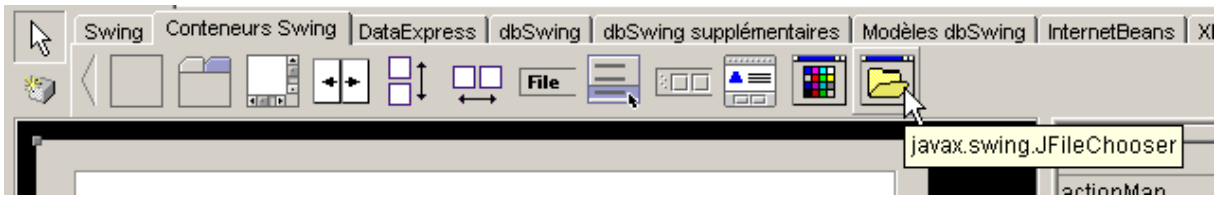
Nous allons construire l'application suivante :



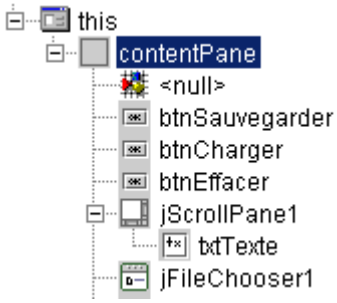
Les contrôles sont les suivants :

N°	type	nom	rôle
0	JScrollPane	jScrollPane1	Conteneur défilant pour la boîte de texte 1
1	JTextArea lui-même dans le JScrollPane	txtTexte	texte tapé par l'utilisateur ou chargé à partir d'un fichier
2	JButton	btnSauvegarder	permet de sauvegarder le texte de 1 dans un fichier texte
3	JButton	btnCharger	permet de charger le contenu d'un fichier texte dans 1
4	JButton	btnEffacer	efface le contenu de 1

Un contrôle non visuel est utilisé : *jFileChooser1*. Celui-ci est pris dans la palette des conteneurs swing de JBuilder :



Nous déposons le composant dans la fenêtre de conception mais en-dehors du formulaire. Il apparaît dans la liste des composants :



Nous donnons maintenant le code utile du programme pour en avoir une vue d'ensemble :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.filechooser.FileFilter;
import java.io.*;

public class dialogues extends JFrame {
  // les composants du cadre
  JPanel contentPane;
  JButton btnSauvegarder = new JButton();
  JButton btnCharger = new JButton();
  JButton btnEffacer = new JButton();
  JScrollPane jScrollPane1 = new JScrollPane();
  JTextArea txtTexte = new JTextArea();
  JFileChooser jFileChooser1 = new JFileChooser();

  // les filtres de fichiers
  javax.swing.filechooser.FileFilter filtreTxt = null;

  //Construire le cadre
  public dialogues() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
      // autres initialisations
      moreInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  // moreInit
  private void moreInit(){
    // initialisations à la construction de la fenêtre
    // filtre *.txt
    filtreTxt = new javax.swing.filechooser.FileFilter(){
      public boolean accept(File f){
        // accepte-t-on f ?
        return f.getName().toLowerCase().endsWith(".txt");
      }//accept
      public String getDescription(){
        // description du filtre
        return "Fichiers Texte (*.txt)";
      }//getDescription
    };
    // on ajoute le filtre
    jFileChooser1.addChoosableFileFilter(filtreTxt);
    // on veut aussi le filtre de tous les fichiers
```

```
    jFileChooser1.setAcceptAllFileFilterUsed(true);
    // on fixe le répertoire de départ de la boîte FileChooser au répertoire courant
    jFileChooser1.setCurrentDirectory(new File("."));
  }
```

```
  //Initialiser le composant
  private void jbInit() throws Exception  {
.....................
  }
  //Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée
  protected void processWindowEvent(WindowEvent e) {
.....................
    }
  }
```

```
  void btnCharger_actionPerformed(ActionEvent e) {
    // choix d'un fichier à l'aide d'un objet JFileChooser

    // on fixe le filtre initial
    jFileChooser1.setFileFilter(filtreTxt);
    // on affiche la boîte de sélection
    int returnVal = jFileChooser1.showOpenDialog(this);
    // l'utilisateur a-t-il choisi qq chose ?
    if(returnVal == JFileChooser.APPROVE_OPTION) {
      // on met le fichier dans le champ texte
       lireFichier(jFileChooser1.getSelectedFile());
    }//if
  }//btnCharger_actionPerformed
```

```
  // lireFichier
  private void lireFichier(File fichier){
    // affiche le contenu du fichier texte fichier dans le champ texte

    // on efface le champ texte
    txtTexte.setText("");

    // qqs données
    BufferedReader IN=null;
    String ligne=null;
    try{
      // ouverture fichier en lecture
      IN=new BufferedReader(new FileReader(fichier));
      // on lit le fichier ligne par ligne
      while((ligne=IN.readLine())!=null){
        txtTexte.append(ligne+"\n");
      }//while
      // fermeture fichier
      IN.close();
    }catch(Exception ex){
      // une erreur s'est produite
      txtTexte.setText(""+ex);
    }//catch
  }

  // effacer
  void btnEffacer_actionPerformed(ActionEvent e) {
    // on efface la boîte de texte
    txtTexte.setText("");
  }
```

```
  // sauvegarder
  void btnSauvegarder_actionPerformed(ActionEvent e) {
    // sauvegarde le contenu de la boîte de texte dans un fichier

    // on fixe le filtre initial
    jFileChooser1.setFileFilter(filtreTxt);
    // on affiche la boîte de sélection de sauvegarde
    int returnVal = jFileChooser1.showSaveDialog(this);
    // l'utilisateur a-t-il choisi qq chose ?
    if(returnVal == JFileChooser.APPROVE_OPTION) {
      // on écrit le contenu de la boîte de texte dans fichier
       écrireFichier(jFileChooser1.getSelectedFile());
    }//if
  }
```

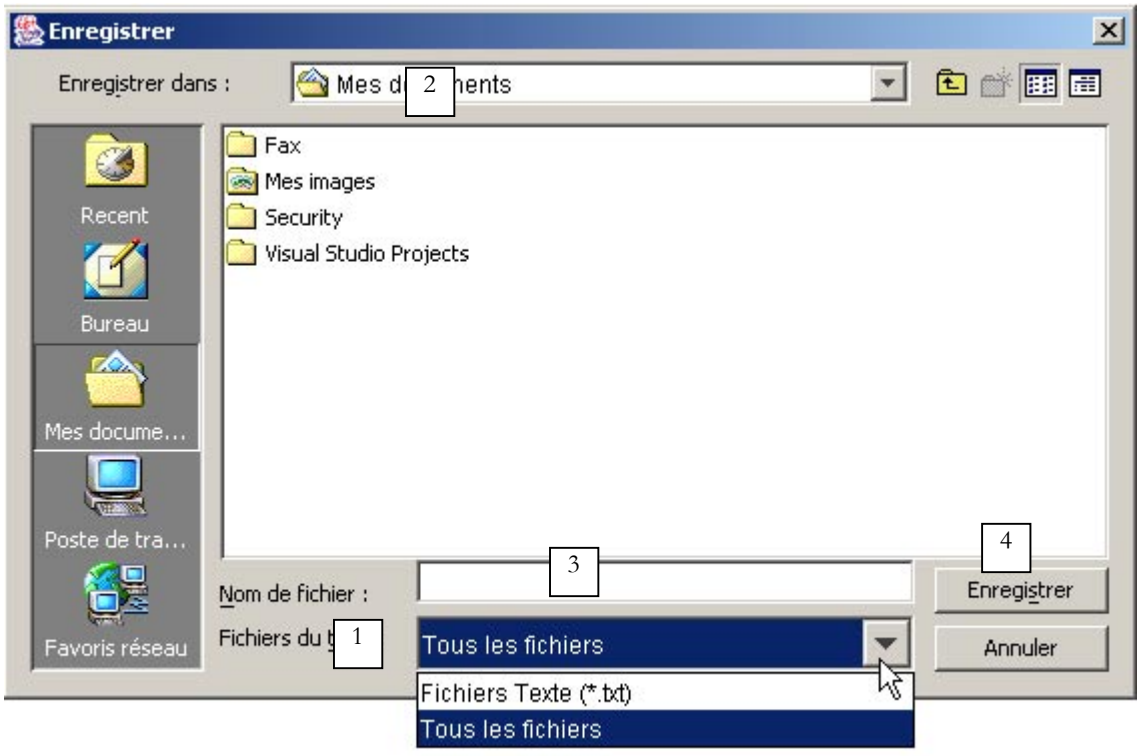
```
  // lireFichier
  private void écrireFichier(File fichier){
    // écrit le contenu de la boîte de texte dans fichier

    // qqs données
    PrintWriter PRN=null;
    try{
```

```
      // ouverture fichier en écriture
      PRN=new PrintWriter(new FileWriter(fichier));
      // on écrit le contenu de la boîte de texte
      PRN.print(txtTexte.getText());
      // fermeture fichier
      PRN.close();
    }catch(Exception ex){
      // une erreur s'est produite
      txtTexte.setText(""+ex);
    }//catch
  }
```

Nous ne commenterons pas le code des méthodes *btnEffacer_Click*, *lireFichier* et *écrireFichier* qui n'amènent rien qui n'a déjà été vu. Nous nous attarderons sur la classe *JFileChooser* et son utilisation. Cette classe est complexe, du genre "usine à gaz". Nous n'utilisons ici que les méthodes suivantes :

addChoosableFilter(FileFilter)	fixe les types de fichiers proposés à la sélection
setAcceptAllFileFilterUsed(boolean)	indique si le type 'Tous les fichiers" doit être proposé à la sélection ou non
File getSelectedFile()	le fichier (File) choisi par l'utilisateur
int showSaveDialog()	méthode qui affiche la boîte de sélection de sauvegarde. Rend un résultat de type *int*. La valeur *jFileChooser.APPROVE_OPTION* indique que l'utilisateur a fait un choix valide. Sinon, il a soit annulé le choix, soit une erreur s'est produite.
setCurrentDirectory	pour fixer le répertoire initial à partir duquel l'utilisateur va commencer à explorer le système de fichiers
setFileFilter(FileFilter)	pour fixer le filtre actif

La méthode *showSaveDialog* affiche une boîte de sélection analogue à la suivante :



1 liste déroulante construite avec la méthode **addChoosableFilter**. Elle contient ce qu'on appelle des filtres de sélection représentés par la classe **FileFilter**. C'est au développeur de définir ces filtres et de les ajouter à la liste 1.

2 dossier courant, fixé par la méthode **setCurrentDirectory** si cette méthode a été utilisée, sinon le répertoire courant sera le dossier "Mes Documents" sous windows ou le home Directory sous Unix.

3 nom du fichier choisi ou tapé directement par l'utilisateur. Sera disponible avec la méthode **getSelectedFile()**

4 boutons **Enregistrer/Annuler**. Si le bouton *Enregistrer* est utilisé, la méthode s*howSaveDialog* rend le résultat **jFileChooser.APPROVE_OPTION**

Comment les filtres de fichiers de la liste déroulante 1 sont-ils construits ? Les filtres sont ajoutés à la liste 1 à l'aide de la méthode :

addChoosableFilter(FileFilter) fixe les types de fichiers proposés à la sélection

de la classe *JFileChooser*. Il nous reste à connaître la classe *FileFilter*. C'est en fait la classe *javax.swing.filechooser.FileFilter* qui est une classe abstraite, c.a.d. une classe qui ne peut être instantiée mais seulement dérivée. Elle est définie comme suit :

FileFilter()	constructeur
boolean accept(File)	indique si le fichier f appartient au filtre ou non
String getDescription()	chaîne de description du filtre

Prenons un exemple. Nous voulons que la liste déroulante 1 offre un filtre pour sélectionner les fichiers **.txt* avec la description *"Fichiers texte (*.txt)"*.

- il nous faut créer une classe dérivée de la classe *FileFilter*
- utiliser la méthode *boolean accept(File f)* pour rendre la valeur *true* si le nom du fichier f se termine par *.txt*
- utiliser la méthode *String getDescription()* pour rendre la description *"Fichiers texte (*.txt)"*

Ce filtre pourrait être défini dans notre application de la façon suivante :

```
javax.swing.filechooser.FileFilter filtreTxt = new javax.swing.filechooser.FileFilter(){
  public boolean accept(File f){
    // accepte-t-on f ?
    return f.getName().toLowerCase().endsWith(".txt");
  }//accept
  public String getDescription(){
    // description du filtre
    return "Fichiers Texte (*.txt)";
  }//getDescription
};
```

Ce filtre serait ajouté à la liste des filtres de l'objet *jFileChooser1* par l'instruction :

```
  // on ajoute le filtre *.txt
  jFileChooser1.addChoosableFileFilter(filtretxt);
```

Tout ceci et quelques autres initialisations sont faites dans la méthode *moreInit* exécutée à la construction de la fenêtre (cf programme complet ci-dessus). Le code du bouton *Sauvegarder* est le suivant :

```
void btnSauvegarder_actionPerformed(ActionEvent e) {
  // sauvegarde le contenu de la boîte de texte dans un fichier

  // on fixe le filtre initial
  jFileChooser1.setFileFilter(filtreTxt);
  // on affiche la boîte de sélection de sauvegarde
  int returnVal = jFileChooser1.showSaveDialog(this);
  // l'utilisateur a-t-il choisi qq chose ?
  if(returnVal == JFileChooser.APPROVE_OPTION) {
    // on écrit le contenu de la boîte de texte dans fichier
     écrireFichier(jFileChooser1.getSelectedFile());
  }//if
}
```

La séquence des opérations est la suivante :

- on fixe le filtre actif au filtre *.txt afin de permettre à l'utilisateur de chercher de préférence ce type de fichiers. Le filtre "Tous les fichiers" est également présent. Il a été ajouté dans la procédure *moreInit*. On a donc deux filtres.
- la boîte de sélection de sauvegarde est affichée. Ici on perd la main, l'utilisateur utilisant la boîte de sélection pour désigner un fichier du système de fichiers.
- lorsqu'il quitte la boîte de sélection, on teste la valeur de retour pour voir si on doit ou non sauvegarder la boîte de texte. Si oui, elle doit l'être dans le fichier obtenu par la méthode *getSelectedFile*.

Le code lié au bouton "Charger" est très analogue au code du bouton "Sauvegarder".

```
void btnCharger_actionPerformed(ActionEvent e) {
  // choix d'un fichier à l'aide d'un objet JFileChooser

  // on fixe le filtre initial
  jFileChooser1.setFileFilter(filtreTxt);
  // on affiche la boîte de sélection
  int returnVal = jFileChooser1.showOpenDialog(this);
  // l'utilisateur a-t-il choisi qq chose ?
  if(returnVal == JFileChooser.APPROVE_OPTION) {
```

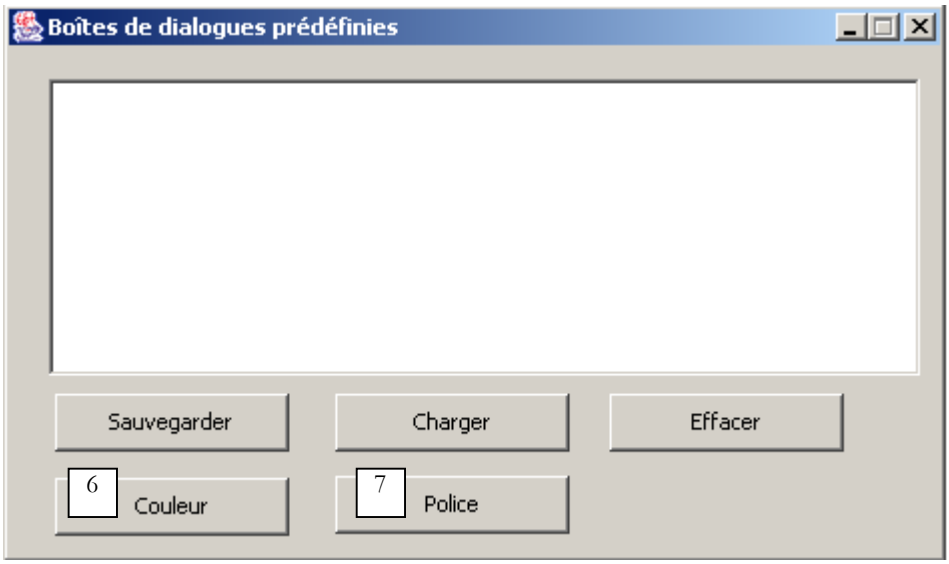
```
      // on met le fichier dans le champ texte
       lireFichier(jFileChooser1.getSelectedFile());
    }//if
  }//btnCharger_actionPerformed
```

Il y a deux différences :

- pour afficher la boîte de sélection de fichiers, on utilise la méthode *showOpenDialog* au lieu de la méthode *showSaveDialog*. La boîte de sélection affichée est analogue à celle affichée par la méthode *showSaveDialog*.
- si l'utilisateur a bien sélectionné un fichier, on appelle la méthode *lireFichier* plutôt que la méthode *écrireFichier*.

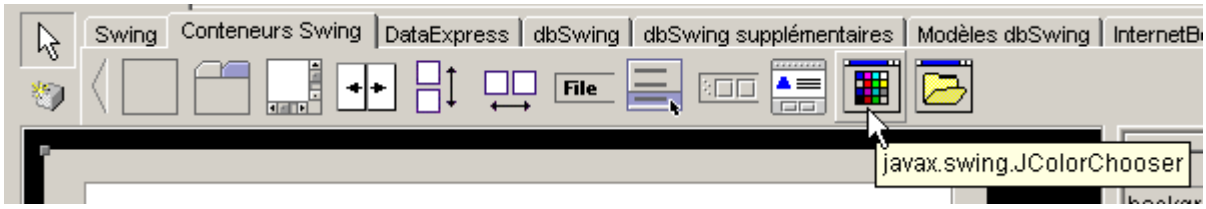
4.4.2 Boîtes de sélection JColorChooser et JFontChooser

Nous continuons l'exemple précédent en ajoutant deux nouveaux boutons :

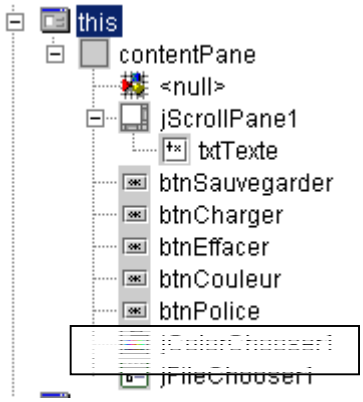


N°	type	nom	rôle
6	JButton	btnCouleur	pour fixer la couleur des caractères du TextBox
7	JButton	btnPolice	pour fixer la police de caractères du TextBox

Le composant *JColorChooser* permettant d'afficher une boîte de sélection de couleur peut être trouvée dans la liste des composants swing de JBuilder :



Nous déposons ce composant dans la fenêtre de conception, mais en-dehors du formulaire. Il est non visible dans le formulaire mais néanmoins présent dans la liste des composants de la fenêtre :



La classe *JColorChooser* est très simple. On affiche la boîte de sélection des couleurs avec la méthode *int showDialog* :

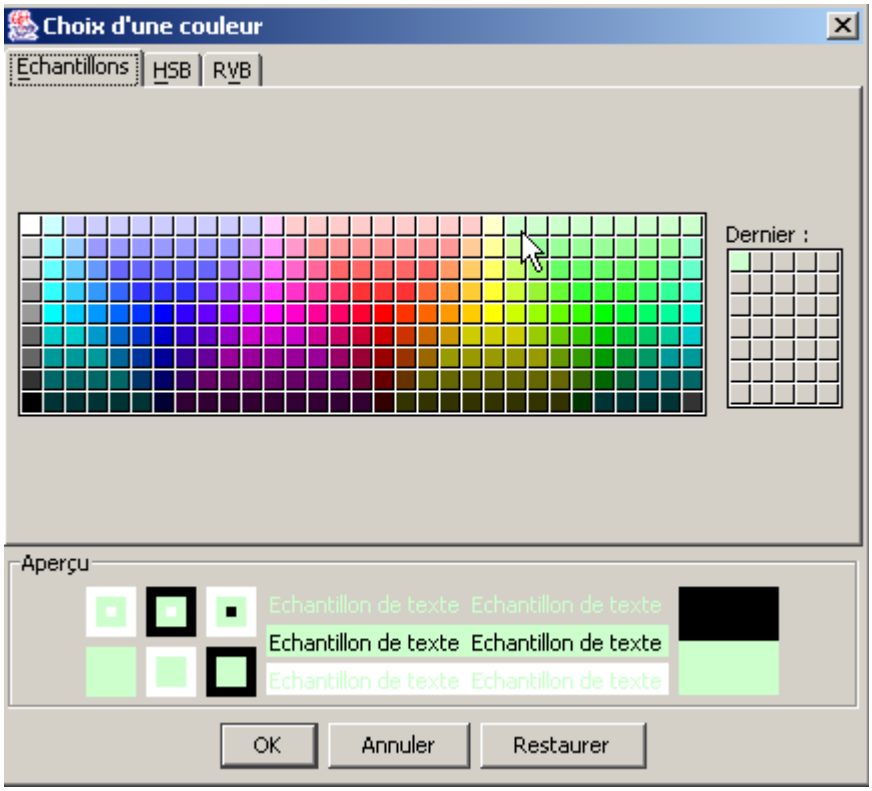
`static Color`	**`showDialog`**`(Component component, String title, Color initialColor)`

Component component	le composant parent de la boîte de sélection, généralement une fenêtre *JFrame*
String title	le titre dans la barre de titre de la boîte de sélection
Color intialColor	couleur initialement sélectionnée dans la boîte de sélection

Si l'utilisateur choisit une couleur, la méthode rend une couleur sinon la valeur *null*. Le code lié au bouton *Couleur* est le suivant :

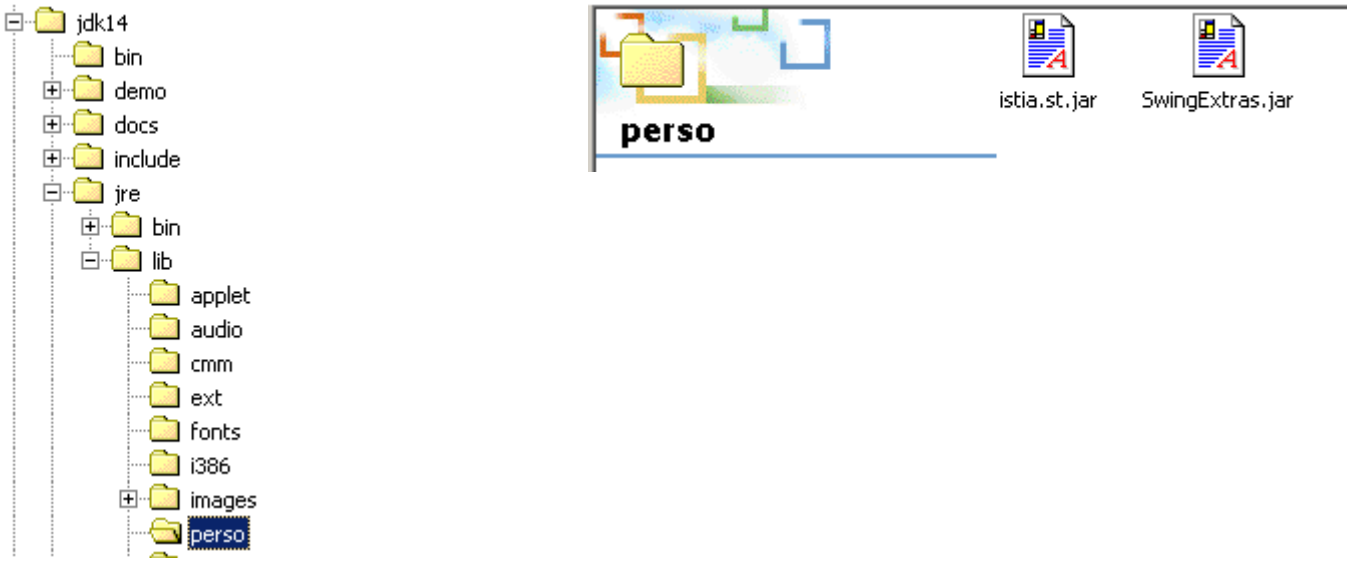
```
void btnCouleur_actionPerformed(ActionEvent e) {
  // choix d'une couleur de texte à l'aide du composant JColorChooser
  Color couleur;
  if((couleur=jColorChooser1.showDialog(this,"Choix d'une couleur",Color.BLACK))!=null){
    // on fixe la couleur des caractères de la boîte de texte
    txtTexte.setForeground(couleur);
  }//if
}
```

La classe *Color* est définie dans *java.awt.Color*. Diverses constantes y sont définies dont Color.BLACK pour la couleur noire. La boîte de sélection affichée est la suivante



Assez curieusement, la bibliothèque swing n'offre pas de classe pour sélectionner une police de caractères. Heureusement, il y a les ressources Java d'internet. En effectuant une recherche sur le mot clé "JFontChooser" qui semble un nom possible pour une telle classe on en trouve plusieurs. L'exemple qui va suivre va nous donner l'occasion de configurer JBuilder pour qu'il utilise des paquetages non prévus dans sa configuration initiale.

Le paquetage récupéré s'appelle *swingextras.jar* et a été placé dans le *dossier <jdk>\jre\lib\perso* où *<jdk>* désigne le répertoire d'installation du jdk utilisé par JBuilder. Il aurait pu être placé n'importe où ailleurs.



Regardons le contenu du paquetage *SwingExtras.jar* :

Nom	Modifié	Taille	Taux	Compr...	Chemin
Manifest.mf	15/02/2001 18...	68	0%	68	meta-inf\
DaySelectionListener.java	08/02/2001 18...	311	38%	193	com\lamatek\swingextras\
IconFactory.java	08/02/2001 18...	12 669	82%	2 308	com\lamatek\swingextras\
ImagePanel.java	08/02/2001 18...	752	52%	363	com\lamatek\swingextras\
IntroDialog.java	08/02/2001 18...	1 378	54%	635	com\lamatek\swingextras\
JDateChooser.java	08/02/2001 18...	10 483	73%	2 853	com\lamatek\swingextras\
JErrorFrame.java	08/02/2001 18...	6 931	74%	1 824	com\lamatek\swingextras\
JFlatButton.java	08/02/2001 18...	1 384	65%	487	com\lamatek\swingextras\
JFontChooser.java	08/02/2001 18...	9 017	70%	2 744	com\lamatek\swingextras\
JFontPreviewPanel.java	08/02/2001 18...	1 673	60%	668	com\lamatek\swingextras\
JInternalToolBar.java	08/02/2001 18...	10 479	74%	2 692	com\lamatek\swingextras\
JNumericField.java	08/02/2001 18...	8 700	77%	2 042	com\lamatek\swingextras\
LAFMenu.java	08/02/2001 18...	2 324	60%	929	com\lamatek\swingextras\
icon.gif	08/02/2001 18...	388	14%	335	com\lamatek\swingextras\
thumb.gif	08/02/2001 18...	106	9%	96	com\lamatek\swingextras\

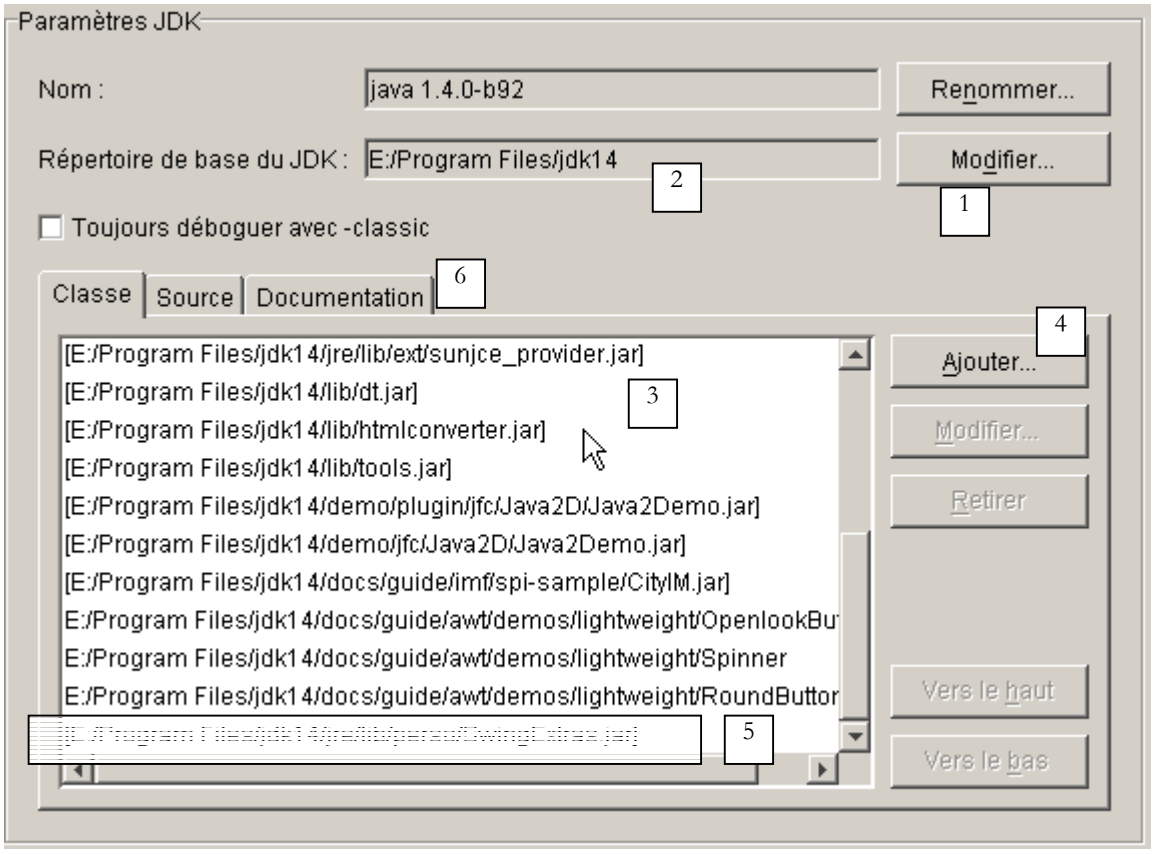
On y trouve le code source java des composants proposés dans le paquetage. On y trouve également les classes elles-mêmes :

JDateChooser.class	11/02/2001 12...	6 317	47%	3 353	com\lamatek\swingextras\
JErrorFrame$1.class	11/02/2001 12...	815	42%	470	com\lamatek\swingextras\
JErrorFrame$2.class	11/02/2001 12...	674	38%	419	com\lamatek\swingextras\
JErrorFrame$3.class	11/02/2001 12...	610	38%	377	com\lamatek\swingextras\
JErrorFrame$4.class	11/02/2001 12...	815	43%	468	com\lamatek\swingextras\
JErrorFrame$5.class	11/02/2001 12...	793	41%	464	com\lamatek\swingextras\
JErrorFrame.class	11/02/2001 12...	5 414	48%	2 794	com\lamatek\swingextras\
JFlatButton$1.class	11/02/2001 12...	632	40%	378	com\lamatek\swingextras\
JFlatButton.class	11/02/2001 12...	997	48%	515	com\lamatek\swingextras\
JFontChooser.class	11/02/2001 12...	6 659	47%	3 547	com\lamatek\swingextras\
JFontPreviewPanel.class	11/02/2001 12...	2 053	47%	1 098	com\lamatek\swingextras\
JInternalToolBar$1.class	11/02/2001 12...	1 440	51%	710	com\lamatek\swingextras\
JInternalToolBar$2.class	11/02/2001 12...	714	45%	390	com\lamatek\swingextras\
JInternalToolBar$3.class	11/02/2001 12...	985	52%	470	com\lamatek\swingextras\

De cette archive, on retiendra que la classe *JFontChooser* s'appelle en réalité *com.lamatek.swingextras.JFontChooser*. Si on ne souhaite pas écrire le nom complet, il nous faudra écrire en début de programme :

```
import com.lamatek.swingextras.*;
```

Comment feront le compilateur et la machine virtuelle Java pour trouver ce nouveau paquetage ? Dans le cas d'une utilisation directe du JDK cela a déjà été expliqué et le lecteur pourra retrouver les explications dans le paragraphe sur les paquetages Java. Nous détaillons ici la méthode à utiliser avec JBuilder. Les paquetages sont configurés dans le menu *Options/Configurer les JDK* :



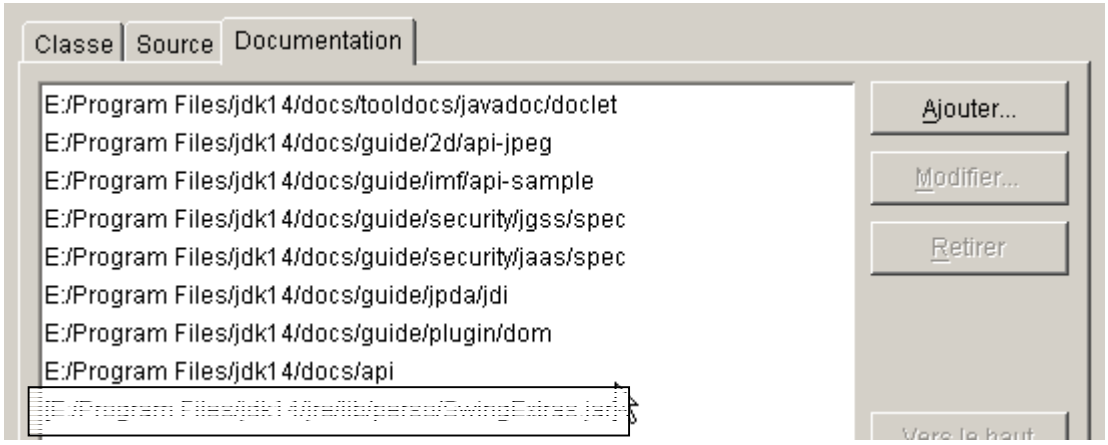
1 Le bouton *Modifier* (1) permet d'indiquer à JBuilder quel JDK utilisé. Dans cet exemple, JBuilder avait amené avec lui le JDK 1.3.1. Lorsque le JDK 1.4 est sorti, il a été installé séparément de JBuilder et on a utilisé le bouton *Modifier* pour indiquer à JBuilder d'utiliser désormais le JDK 1.4 en désignant le répertoire d'installation de ce dernier

2 répertoire de base du JDK actuellement utilisé par JBuilder

3 liste des archives java (.jar) utilisées par JBuilder. On peut en ajouter d'autres avec le bouton *Ajouter* (4)

4 Le bouton *Ajouter* (4) permet d'ajouter de nouveaux paquetages que JBuilder utilisera à la fois pour la compilation et l'exécution des programmes. On l'a utilisé ici pour ajouter le paquetage *SwingExtras.jar* (5)

Maintenant JBuilder est correctement configuré pour pouvoir utiliser la classe *JFontChooser*. Cependant, il nous faudrait avoir accès à la définition de cette classe pour l'utiliser correctement. L'archive *swingextras.jar* contient des fichiers html qu'on pourrait extraire pour les exploiter. C'est inutile. La documentation java incluse dans les fichiers .jar est directement accessible de JBuilder. Il faut pour cela configurer l'onglet *Documentation* (6) ci-dessus. On obtient la nouvelle fenêtre suivante :

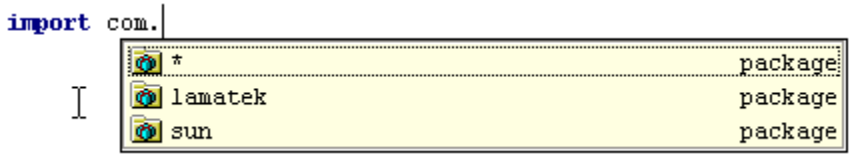


Le bouton *Ajouter* nous permet d'indiquer que le fichier *SwingExtras.jar* doit être exploré pour la documentation. Cette démarche validée, on peut constater qu'on a effectivement accès à la documentation de *SwingExtras.jar*. Cela se traduit par diverses facilités :

- si on commence à écrire l'instruction d'importation

```
import com.lamatek.swingextras.*;
```

on pourra constater que JBuilder nous fournit une aide :

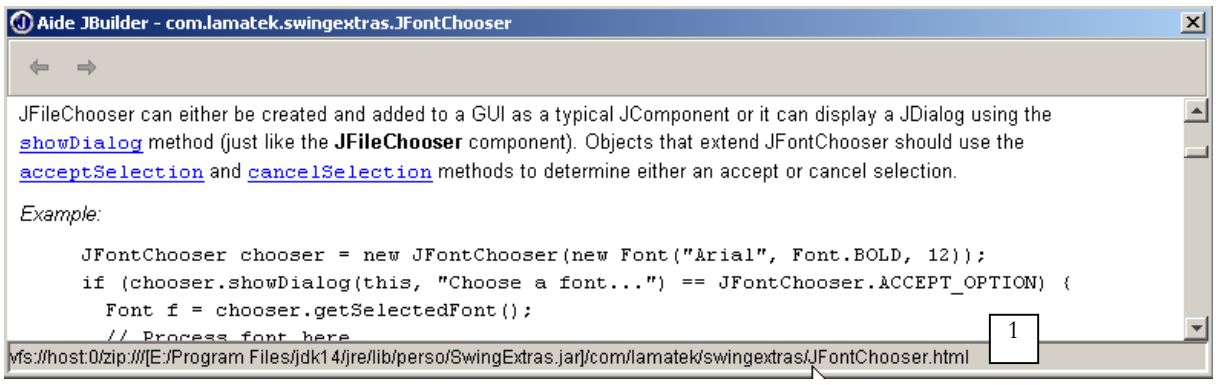


Le paquetage *com.lamatek* est bien trouvé.

- si maintenant sur le programme suivant :

```
import com.lamatek.swingextras.*;

public class test{
  JFontChooser jFontChooser1=null;
}
```

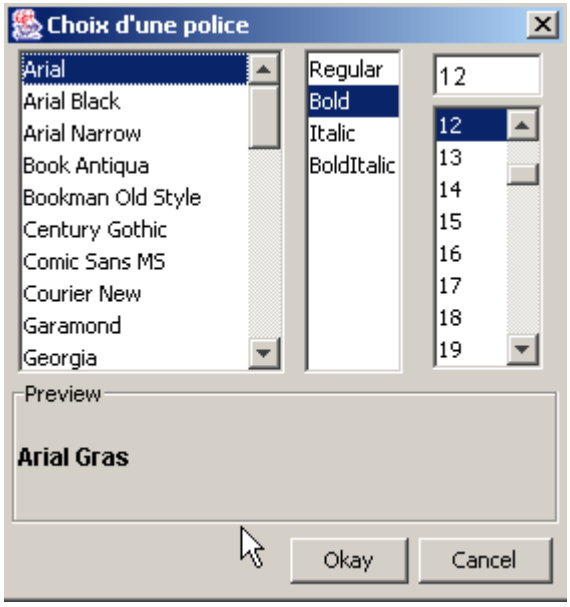
on fait F1 sur le mot clé *JFontChooser*, on obtient une aide sur cette classe :



On voit dans la barre d'état 1 que c'est bien un fichier html du paquetage *swingextras.jar* qui est utilisé. L'exemple présenté ci-dessus est suffisament explicite pour qu'on puisse écrire le code du bouton *Police* de notre application :

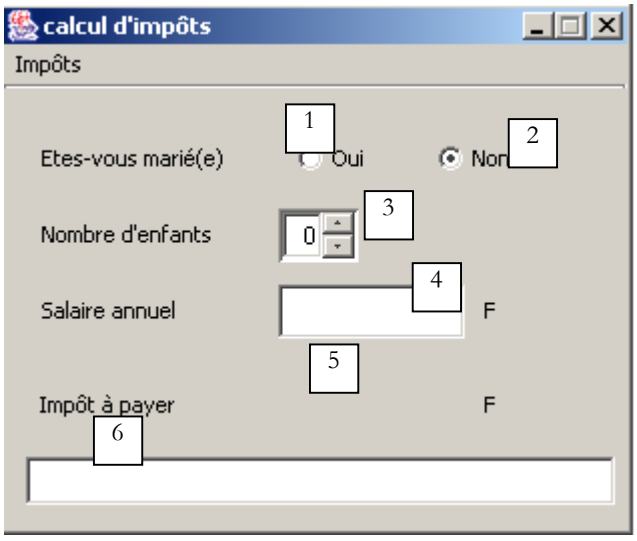
```
void btnPolice_actionPerformed(ActionEvent e) {
  // choix d'une police de texte à l'aide du composant JFontChooser
  jFontChooser1 = new JFontChooser(new Font("Arial", Font.BOLD, 12));
  if (jFontChooser1.showDialog(this, "Choix d'une police") == JFontChooser.ACCEPT_OPTION) {
    // on change la police des caractères de la boîte de texte
    txtTexte.setFont(jFontChooser1.getSelectedFont());
  }//if
}
```

La boîte de sélection affichée par la méthode *showDialog* ci-dessus est la suivante :



4.5 L'application graphique IMPOTS

On reprend l'application IMPOTS déjà traitée deux fois. Nous y ajoutons maintenant une interface graphique :



Les contrôles sont les suivants

n°	type	nom	rôle
1	JRadioButton	rdOui	coché si marié
2	JRadioButton	rdNon	coché si pas marié
3	JSpinner	spinEnfants	nombre d'enfants du contribuable

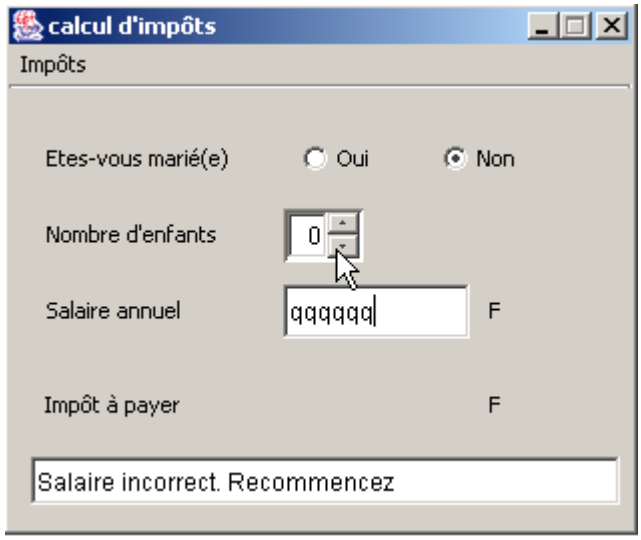
			Minimum=0, Maximum=20, Increment=1
4	JTextField	txtSalaire	salaire annuel du contribuable en F
5	JLabel	lblImpots	montant de l'impôt à payer
6	JTextField	txtStatus	champ de messages d'état - pas modifiable

Le menu est le suivant :

option principale	option secondaire	nom	rôle
Impots			
	Initialiser	mnuInitialiser	charge les données nécessaires au calcul à partir d'un fichier texte
	Calculer	mnuCalculer	calcule l'impôt à payer lorsque toutes les données nécessaires sont présentes et correctes
	Effacer	mnuEffacer	remet le formulaire dans son état initial
	Quitter	mnuQuitter	termine l'application

Règles de fonctionnement

- le menu *Calculer* reste éteint tant qu'il n'y a rien dans le champ du salaire
- si lorsque le calcul est lancé, il s'avère que le salaire est incorrect, l'erreur est signalée :



Le programme est donné ci-dessous. Il utilise la classe *impots* créée dans le chapitre sur les classes. Une partie du code produit automatiquement pas JBuilder n'a pas été ici reproduit.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.filechooser.FileFilter;
import java.io.*;
import java.util.*;
import javax.swing.event.*;

public class frmImpots extends JFrame {
  // les composants de la fenêtre
  JPanel contentPane;
  JMenuBar jMenuBar1 = new JMenuBar();
  JMenu jMenu1 = new JMenu();
  JMenuItem mnuInitialiser = new JMenuItem();
  JMenuItem mnuCalculer = new JMenuItem();
  JMenuItem mnuEffacer = new JMenuItem();
  JMenuItem mnuQuitter = new JMenuItem();
  JLabel jLabel1 = new JLabel();
  JFileChooser jFileChooser1 = new JFileChooser();
  JTextField txtStatus = new JTextField();
  JRadioButton rdOui = new JRadioButton();
  JRadioButton rdNon = new JRadioButton();
  JLabel jLabel2 = new JLabel();
  JLabel jLabel3 = new JLabel();
  JTextField txtSalaire = new JTextField();
  JLabel jLabel4 = new JLabel();
  JLabel jLabel5 = new JLabel();
```

```
  JLabel lblImpots = new JLabel();
  JLabel jLabel7 = new JLabel();
  ButtonGroup buttonGroup1 = new ButtonGroup();
  JSpinner spinEnfants=null;
  FileFilter filtreTxt=null;

  // les attributs de la classe
  double[] limites=null;
  double[] coeffr=null;
  double[] coeffn=null;
  impots objImpots=null;

  //Construire le cadre
  public frmImpots() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    // autres initialisations
    moreInit();
  }

  // initialisation formulaire
  private void moreInit(){
    // menu Calculer inhibé
    mnuCalculer.setEnabled(false);
    // zones de saisie inhibées
    txtSalaire.setEditable(false);
    txtSalaire.setBackground(Color.WHITE);
    // champ d'état
    txtStatus.setBackground(Color.WHITE);
    // spinner Enfants - entre 0 et 20 enfants
    spinEnfants=new JSpinner(new SpinnerNumberModel(0,0,20,1));
    spinEnfants.setBounds(new Rectangle(137,60,40,27));
    contentPane.add(spinEnfants);
    // filtre *.txt pour la boîte de sélection
    filtreTxt = new javax.swing.filechooser.FileFilter(){
      public boolean accept(File f){
        // accepte-t-on f ?
        return f.getName().toLowerCase().endsWith(".txt");
      }//accept
      public String getDescription(){
        // description du filtre
        return "Fichiers Texte (*.txt)";
      }//getDescription
    };
    // on ajoute le filtre *.txt
    jFileChooser1.addChoosableFileFilter(filtreTxt);
    // on veut aussi le filtre de tous les fichiers
    jFileChooser1.setAcceptAllFileFilterUsed(true);
    // on fixe le répertoire de départ de la boîte FileChooser
    jFileChooser1.setCurrentDirectory(new File("."));
  }//moreInit

  //Initialiser le composant
  private void jbInit() throws Exception  {

.................

  }

  //Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée
  protected void processWindowEvent(WindowEvent e) {
......................
  }

  void mnuQuitter_actionPerformed(ActionEvent e) {
    // on quitte l'application
    System.exit(0);
  }

  void mnuInitialiser_actionPerformed(ActionEvent e) {
    // on charge le fichier des données
    // qu'on sélectionne avec la boîte de sélection JFileChooser1
    jFileChooser1.setFileFilter(filtreTxt);
    if (jFileChooser1.showOpenDialog(this)!=JFileChooser.APPROVE_OPTION)
      return;
    // on exploite le fichier sélectionné
```

```
    try{
      // lecture données
      lireFichier(jFileChooser1.getSelectedFile());
      // création de l'objet impots
      objImpots=new impots(limites,coeffr,coeffn);
      // confirmation
      txtStatus.setText("Données chargées");
      // le salaire peut être modifié
      txtSalaire.setEditable(true);
      // plus de chgt possible
      mnuInitialiser.setEnabled(false);
    }catch(Exception ex){
      // problème
      txtStatus.setText("Erreur : " + ex.getMessage());
      // fin
      return;
    }//catch
  }

  private void lireFichier(File fichier) throws Exception {
    // les tableaux de données
    ArrayList aLimites=new ArrayList();
    ArrayList aCoeffR=new ArrayList();
    ArrayList aCoeffN=new ArrayList();
    String[] champs=null;

    // ouverture fichier en lecture
    BufferedReader IN=new BufferedReader(new FileReader(fichier));
    // on lit le fichier ligne par ligne
    // elles sont de la forme limite coeffr coeffn
    String ligne=null;
    int numLigne=0;           // n° de ligne courannte
    while((ligne=IN.readLine())!=null){
      // une ligne de plus
      numLigne++;
      // on décompose la ligne en champs
      champs=ligne.split("\\s+");
      // 3 champs ?
      if(champs.length!=3)
        throw new Exception("ligne " + numLigne + "erronée dans fichier des données");
      // on récupère les trois champs
       aLimites.add(champs[0]);
       aCoeffR.add(champs[1]);
       aCoeffN.add(champs[2]);
    }//while
    // fermeture fichier
    IN.close();
    // transfert des données dans des tableaux bornés
    int n=aLimites.size();
    limites=new double[n];
    coeffr=new double[n];
    coeffn=new double[n];
    for(int i=0;i<n;i++){
      limites[i]=Double.parseDouble((String)aLimites.get(i));
      coeffr[i]=Double.parseDouble((String)aCoeffR.get(i));
      coeffn[i]=Double.parseDouble((String)aCoeffN.get(i));
    }//for
  }

  void mnuCalculer_actionPerformed(ActionEvent e) {
    // calcul de l'impôt
    // vérification salaire
    int salaire=0;
    try{
      salaire=Integer.parseInt(txtSalaire.getText().trim());
      if(salaire<0) throw new Exception();
    }catch (Exception ex){
      // msg d'erreur
      txtStatus.setText("Salaire incorrect. Recommencez");
      // focus sur champ erroné
      txtSalaire.requestFocus();
      // retour à l'interface
      return;
    }
    // nbre d'enfants
    Integer InbEnfants=(Integer)spinEnfants.getValue();
    // calcul de l'impôt
    lblImpots.setText(""+objImpots.calculer(rdOui.isSelected(),InbEnfants.intValue(),salaire));
    // effacement msg d'état
    txtStatus.setText("");
  }

  void txtSalaire_caretUpdate(CaretEvent e) {
    // le salaire a changé - on met à jour le menu calculer
    mnuCalculer.setEnabled(! txtSalaire.getText().trim().equals(""));
  }
```

```
  void mnuEffacer_actionPerformed(ActionEvent e) {
    // raz du formulaire
    rdNon.setSelected(true);
    spinEnfants.getModel().setValue(new Integer(0));
    txtSalaire.setText("");
    mnuCalculer.setEnabled(false);
    lblImpots.setText("");
  }
}
```

Nous avons utilisé ici un composant disponible qu'à partir du JDK 1.4, le *JSpinner*. C'est un incrémenteur, qui permet ici à l'utilisateur de fixer le nombre d'enfants. Ce composant swing n'était pas disponible dans la barre des composants swing de JBuilder 6 utilisé pour le test. Cela n'empêche pas son utilisation même si les choses sont un peu plus compliquées que pour les composants disponibles dans la barre des composants. En effet, on ne peut déposer le composant *JSpinner* sur le formulaire lors de la conception. Il faut le faire à l'exécution. Regardons le code qui le fait :

```
    // spinner Enfants - entre 0 et 20 enfants
    spinEnfants=new JSpinner(new SpinnerNumberModel(0,0,20,1));
    spinEnfants.setBounds(new Rectangle(137,60,40,27));
    contentPane.add(spinEnfants);
```

La première ligne crée le composant *JSpinner*. Ce composant peut servir à diverses choses et pas seulement à un incrémenteur d'entiers comme ici. L'argument du constructeur *JSpinner* est ici un modèle d'incrémenteur de nombres acceptant quatre paramètres (valeur, min, max, incrément). Le composant a la forme suivante :

valeur	valeur initiale affichée dans le composant
min	valeur minimale affichable dans le composant
max	valeur maximale affichable dans le composant
incrément	valeur d'incrément de la valeur affichée lorsqu'on utilise les flèches haut/bas du composant

La valeur du composant est obtenue via sa méthode *getValue* qui rend un type *Object*. D'où quelques transtypages pour avoir l'entier dont on a besoin :

```
    // nbre d'enfants
    Integer InbEnfants=(Integer)spinEnfants.getValue();
    // calcul de l'impôt
    lblImpots.setText(""+objImpots.calculer(rdOui.isSelected(),
      InbEnfants.intValue(),salaire));
```

Une fois le composant *JSpinner* défini, il est placé dans la fenêtre :

```
    spinEnfants.setBounds(new Rectangle(137,60,40,27));
    contentPane.add(spinEnfants);
```

Avant toute chose, l'utilisateur doit utiliser l'option de menu *Initialiser* qui construit un objet *impots* avec le constructeur

```
  public impots(double[] LIMITES, double[] COEFFR, double[] COEFFN) throws Exception
```

de la classe *impots*. On rappelle que celle-ci a été définie en exemple dans le chapitre des classes. On s'y reportera si besoin est. Les trois tableaux nécessaires au constructeur sont remplis à partir du contenu d'un fichier texte ayant la forme suivante :

```
12620.0    0       0
13190      0.05    631
15640      0.1     1290.5
24740      0.15    2072.5
31810      0.2     3309.5
39970      0.25    4900
48360      0.3     6898.5
55790      0.35    9316.5
92970      0.4     12106
127860     0.45    16754.5
151250     0.50    23147.5
172040     0.55    30710
```

195000 0.60 39312
0 0.65 49062

Chaque ligne comprend trois nombres séparés par au moins un espace. Ce fichier texte est sélectionné par l'utilisateur à l'aide d'un composant *JFileChooser*. Une fois l'objet de type *impots* construit, il ne reste plus qu'à laisser l'utilisateur donner les trois informations dont on a besoin : marié ou non, nombre d'enfants, salaire annuel, et appeler la méthode *calculer* de la classe *impots*. L'opération peut être répétée plusieurs fois. L'objet de type *impots* n'est lui construit qu'une fois, lors de l'utilisation de l'option *Initialiser*.

4.6 Ecriture d'applets

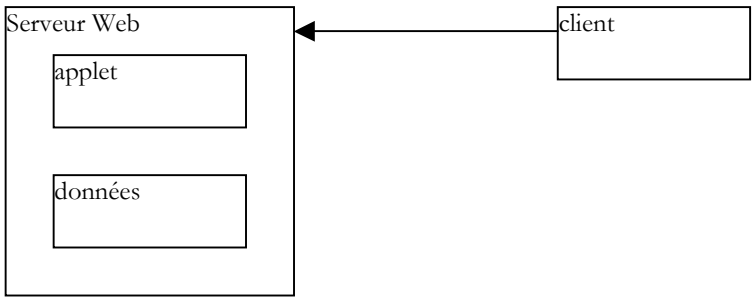
4.6.1 Introduction

Lorsqu'on a écrit une application avec interface graphique il est assez aisé de la transformer en *applet*. Stockée sur une machine A, celle-ci peut être téléchargée par un navigateur Web d'une machine B de l'internet. L'application initiale est ainsi partagée entre de nombreux utilisateurs et c'est là le principal intérêt de transformer une application en applet. Néammoins, toute application ne peut être ainsi transformée : pour ne pas nuire à l'utilisateur qui utilise une applet dans son navigateur, l'environnement de l'applet est réglementé :

- une applet ne peut lire ou écrire sur le disque de l'utilisateur
- elle ne peut communiquer qu'avec la machine à partir de laquelle elle a été téléchargé par le navigateur.

Ce sont des restrictions fortes. Elles impliquent par exemple qu'une application ayant besoin de lire des informations dans un fichier ou une base de données devra les demander par une application relais située sur le serveur d'où elle a été téléchargée.

La structure générale d'une application Web simple est la suivante :



- le client demande un document HTML au serveur Web généralement avec un navigateur. Ce document peut contenir une applet qui fonctionnera comme une application graphique autonome au sein du document HTML affiché par le navigateur du client.
- cette applet pourra avoir accès à des données mais seulement à celles situées sur le serveur web. Elle n'aura pas accès ni aux ressources de la machine cliente qui l'exécute ni à celles d'autres machines du réseau autres que celle à partir de laquelle elle a été téléchargée.

4.6.2 La classe JApplet

4.6.2.1 Définition

Une application peut être téléchargée par un navigateur Web si c'est une instance de la classe **java.applet.Applet** ou de la classe **javax.swing.JApplet**. Cette dernière dérive de la première qui est elle-même dérivée de la classe *Panel* elle-même dérivée de la classe *Container*. Une instance *Applet* ou *JApplet* étant de type *container* pourra donc contenir des composants (*Component*) tels que boutons, cases à cocher, listes, … Donnons quelques indications sur le rôle des différentes méthodes :

Méthode	Rôle
public void destroy();	détruit l'instance d'Applet
public AppletContext getAppletContext();	récupère le contexte d'exécution de l'applet (document HTML dans lequel il se trouve, autres applets du même document, …)
public String getAppletInfo();	rend une chaîne de caractères donnant des informations sur l'applet

public AudioClip getAudioClip(URL url);	charge le fichier audio précisé par URL
public AudioClip getAudioClip(URL url, String name);	charge le fichier audio précisé par URL/name
public URL getCodeBase();	rend l'URL de l'applet
public URL getDocumentBase();	rend l'URL du document HTML contenant l'applet
public Image getImage(URL url);	récupère l'image précisée par URL
public Image getImage(URL url, String name);	récupère l'image précisée par URL/name
public String getParameter(String name);	récupère la valeur du paramètre *name* contenu dans la balise <applet> du document HTML qui contient l'applet.
public void init();	cette méthode est lancée par le navigateur lors du lancement initial de l'applet
public boolean isActive();	état de l'applet
public void play(URL url);	joue le fichier son précisé par URL
public void play(URL url, String name);	joue le fichier son précisé par UR/name
public void resize(Dimension d);	fixe la dimension du cadre de l'applet
public void resize(int width, int height);	idem
public final void setStub(AppletStub stub);	
public void showStatus(String msg);	met un message dans la barre d'état de l'applet
public void start();	cette méthode est lancée par le navigateur à chaque affichage du document contenant l'applet
public void stop();	cette méthode est lancée par le navigateur à chaque fois que le document contenant l'applet est abandonné au profit d'un autre (changement d'URL par l'utilisateur)

La classe *JApplet* a amené quelques améliorations à la classe *Applet*, notamment la capacité à contenir des composants *JMenuBar* c.a.d. des menus, ce qui n'était pas possible avec la classe *Applet*.

4.6.2.2 Exécution d'une applet : les méthodes init, start et stop

Lorsqu'un navigateur charge une applet, il appelle trois méthodes de celle-ci :

init Cette méthode est appelée lors du chargement initial de l'applet. On y mettra donc les initialisations nécessaires à l'application.

start Cette méthode est appelée à chaque fois que le document contenant l'applet devient le document courant du navigateur. Ainsi lorsqu'un utilisateur charge une applet, les méthodes **init** et **start** vont être exécutées dans cet ordre. Lorsqu'il va quitter le document pour en visualiser un autre, la méthode **stop** va être exécutée. Lorsqu'il reviendra dessus plus tard, la méthode **start** sera exécutée.

stop Cette méthode est appelée à chaque fois que l'utilisateur quitte le document contenant l'applet.

Pour beaucoup d'applets, seule la méthode **init** est nécessaire. Les méthodes **start** et **stop** ne sont nécessaires que si l'application lance des tâches (**threads**) qui tournent en parallèle et en continu souvent à l'insu de l'utilisateur. Lorsque celui-ci quitte le document, la partie visible de l'application disparaît, mais ces tâches de fond continuent à travailler. C'est souvent inutile. On profite alors de l'appel du navigateur à la méthode **stop** pour les arrêter. Si l'utilisateur revient sur le document, on profite de l'appel du navigateur à la méthode **start** pour les relancer.

Prenons par exemple une applet qui a une horloge dans son interface graphique. Celle-ci est maintenue par une tâche de fond (**thread**). Lorsque dans le navigateur l'utilisateur quitte la page de l'applet, il est inutile de maintenir l'horloge qui est devenue invisible : dans la méthode **stop** de l'applet, on arrêtera le **thread** qui gère l'horloge. Dans la méthode **start**, on le relancera afin que lorsque l'utilisateur revient sur la page de l'applet, il retrouve une horloge à l'heure.

4.6.3 Transformation d'une application graphique en applet

On suppose ici que cette transformation est possible, c'est à dire qu'elle vérifie les restrictions d'exécution des applets. Une application est lancée par la méthode *main* d'une de ses classes :

```
…
public class application{
  …
  public static main void(String arg[]){
    …
  }
  …
}// fin classe
```

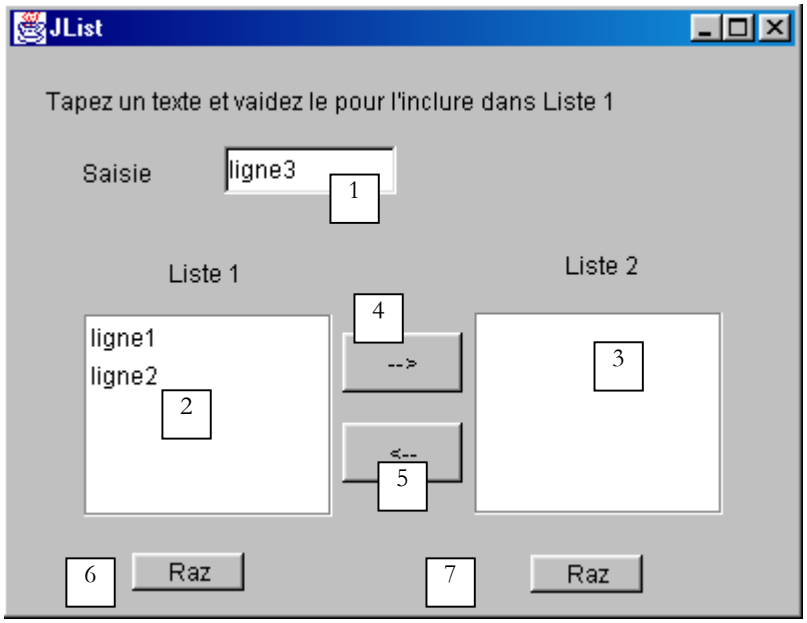
Une telle application est transformée en applet de la façon suivante :

```
import java.applet.JApplet;
…
public class application extends JApplet{
  …
  public void init(){
    …
  }
  …
}// fin classe
```

On notera les points suivants :

1. la classe *application* dérive maintenant de la classe *JApplet*
2. la méthode *main* est remplacée par la méthode *init*.

A titre d'exemple, nous revenons sur une application étudiée : la gestion de listes.



Les composants de cette fenêtre sont les suivants :

n°	type	nom	rôle
1	JTextField	txtSaisie	champ de saisie
2	JList	jList1	liste contenue dans un conteneur jScrollPane1
3	JList	jList2	liste contenue dans un conteneur jScrollPane2
4	JButton	cmd1To2	transfère les éléments sélectionnés de liste 1 vers liste 2
5	JButton	cmd2To1	fait l'inverse
6	JButton	cmdRaz1	vide la liste 1
7	JButton	cmdRaz2	vide la liste 2

Le squelette de l'application était le suivant :

```
public class interfaceAppli extends JFrame {
  JPanel contentPane;
  JLabel jLabel1 = new JLabel();
  JLabel jLabel2 = new JLabel();
  JLabel jLabel3 = new JLabel();
  JTextField txtSaisie = new JTextField();
  JButton cmd1To2 = new JButton();
  JButton cmd2To1 = new JButton();
  DefaultListModel v1=new DefaultListModel();
  DefaultListModel v2=new DefaultListModel();
  JList jList1 = new JList(v1);
  JList jList2 = new JList(v2);
  JScrollPane jScrollPane1 = new JScrollPane();
  JScrollPane jScrollPane2 = new JScrollPane();
```

```
  JButton cmdRaz1 = new JButton();
  JButton cmdRaz2 = new JButton();
  JLabel jLabel4 = new JLabel();

  /**Construire le cadre*/
  public interfaceAppli() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }//interfaceAppli

  /**Initialiser le composant*/
  private void jbInit() throws Exception  {
    ...
    txtSaisie.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        txtSaisie_actionPerformed(e);
      }
    });
    ...
    // Jlist1 est placé dans le conteneur jScrollPane1
    jScrollPane1.getViewport().add(jList1, null);
    // Jlist2 est placé dans le conteneur jScrollPane2
    jScrollPane2.getViewport().add(jList2, null);
    ...
  }
  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
  protected void processWindowEvent(WindowEvent e) {
...
  }

  void txtSaisie_actionPerformed(ActionEvent e) {
.............
}//classe
```

```
  void cmd1To2_actionPerformed(ActionEvent e) {
..........
  }//transfert
```

```
  void cmdRaz1_actionPerformed(ActionEvent e) {
    // vide liste 1
    v1.removeAllElements();
  }//cmd Raz1

  void cmdRaz2_actionPerformed(ActionEvent e) {
    // vide liste 2
    v2.removeAllElements();
  }///cmd Raz2
```

Pour transformer la classe *appli* en *applet*, il faut procéder comme suit :

- modifier la classe *interfaceAppli* précédente pour qu'elle dérive non plus de *JFrame* mais de *JApplet* :

```
public class interfaceAppli extends JApplet {
```

- supprimer l'instruction qui fixe le titre de la fenêtre *JFrame*. Une applet *JApplet* n'a pas de barre de titre

```
    // this.setTitle("JList");
```

- changer le constructeur en méthode *init* et dans cette méthode supprimer la gestion des événements fenêtre (WindowListener, ...). Une applet est un conteneur qui ne peut être redimensionné ou fermé.

```
    /**Construire le cadre*/
  public init() {
    // enableEvents(AWTEvent.WINDOW_EVENT_MASK);
      try {
        jbInit();
      }
      catch(Exception e) {
        e.printStackTrace();
      }
    }//interfaceAppli
```

- La méthode *processWindowEvent* doit être supprimée ou mise en commentaires

```
  /**Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée*/
  //protected void processWindowEvent(WindowEvent e) {
  //  super.processWindowEvent(e);
  //  if (e.getID() == WindowEvent.WINDOW_CLOSING) {
  //    System.exit(0);
  //  }
  // }
```

On donne à titre d'exemple le code complet de l'applet sauf la partie générée automatiquement par JBuilder

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class interfaceAppli extends JApplet {
  JPanel contentPane;
  JLabel jLabel1 = new JLabel();
  JLabel jLabel2 = new JLabel();
  JLabel jLabel3 = new JLabel();
  JTextField txtSaisie = new JTextField();
  JButton cmd1To2 = new JButton();
  JButton cmd2To1 = new JButton();
  DefaultListModel v1=new DefaultListModel();
  DefaultListModel v2=new DefaultListModel();
  JList jList1 = new JList(v1);
  JList jList2 = new JList(v2);
  JScrollPane jScrollPane1 = new JScrollPane();
  JScrollPane jScrollPane2 = new JScrollPane();
  JButton cmdRaz1 = new JButton();
  JButton cmdRaz2 = new JButton();
  JLabel jLabel4 = new JLabel();

  /**Construire le cadre*/
  public void init() {
    // enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }//interfaceAppli

  /**Initialiser le composant*/
  private void jbInit() throws Exception  {
    //setIconImage(Toolkit.getDefaultToolkit().createImage(interfaceAppli.class.getResource("[Votre
icône]")));
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(null);
    this.setSize(new Dimension(400, 308));
    // this.setTitle("JList");
    jScrollPane1.setBounds(new Rectangle(37, 133, 124, 101));
    jScrollPane2.setBounds(new Rectangle(232, 132, 124, 101));
.............
  }

  void txtSaisie_actionPerformed(ActionEvent e) {
    // le texte de la saisie a été validé
    // on le récupère débarrassé de ses espaces de début et fin
    String texte=txtSaisie.getText().trim();
    // s'il est vide, on n'en veut pas
    if(texte.equals("")){
      // msg d'erreur
      JOptionPane.showMessageDialog(this,"Vous devez taper un texte",
        "Erreur",JOptionPane.WARNING_MESSAGE);
      // fin
      return;
    }//if
    // s'il n'est pas vide, on l'ajoute aux valeurs de la liste 1
    v1.addElement(texte);
    // et on vide le champ de saisie
    txtSaisie.setText("");
  }

  void cmd1To2_actionPerformed(ActionEvent e) {
    // transfert des éléments sélectionnés dans la liste 1 vers la liste 2
    transfert(jList1,jList2);
  }//cmd1To2

  private void transfert(JList L1, JList L2){
      // transfert des éléments sélectionnés dans la liste 1 vers la liste 2
      // on récupère le tableau des indices des éléments sélectionnés dans L1
      int[] indices=L1.getSelectedIndices();
      // qq chose à faire ?
```

```
      if (indices.length==0) return;
      // on récupère les valeurs de L1
      DefaultListModel v1=(DefaultListModel)L1.getModel();
      // et celles de L2
      DefaultListModel v2=(DefaultListModel)L2.getModel();
      for(int i=indices.length-1;i>=0;i--){
        // on ajoute à L2 les valeurs sélectionnées dans L1
        v2.addElement(v1.elementAt(indices[i]));
        // les éléments de L1 copiés dans L2 doivent être supprimés de L1
        v1.removeElementAt(indices[i]);
      }//for
  }//transfert

  private void affiche(String message){
    // affiche message
      JOptionPane.showMessageDialog(this,message,
        "Suivi",JOptionPane.INFORMATION_MESSAGE);
  }// affiche

  void cmd2To1_actionPerformed(ActionEvent e) {
    // transfert des éléments sélectionnés dans jList2 vers jList1
      transfert(jList2,jList1);
  }//cmd2TO1

  void cmdRaz1_actionPerformed(ActionEvent e) {
    // vide liste 1
    v1.removeAllElements();
  }//cmd Raz1

  void cmdRaz2_actionPerformed(ActionEvent e) {
    // vide liste 2
    v2.removeAllElements();
  }//cmd Raz2

}//classe
```

On peut compiler le source de cette applet. On le fait ici avec le JDK, sans JBuilder.

```
E:\data\serge\Jbuilder\interfaces\JApplets\1>javac.bat interfaceAppli.java

E:\data\serge\Jbuilder\interfaces\JApplets\1>dir
12/06/2002  16:40               6 148 interfaceAppli.java
12/06/2002  16:41                 527 interfaceAppli$1.class
12/06/2002  16:41                 525 interfaceAppli$2.class
12/06/2002  16:41                 525 interfaceAppli$3.class
12/06/2002  16:41                 525 interfaceAppli$4.class
12/06/2002  16:41                 525 interfaceAppli$5.class
12/06/2002  16:41               4 759 interfaceAppli.class
```

L'application peut être testée avec le programme **AppletViewer** du JDK qui permet d'exécuter des applets ou un navigateur. Pour cela, il faut créer le document HTML **appli.htm** qui aura en son sein l'applet :

```
<html>
  <head>
    <title>listes swing</title>
  </head>
  <body>
    <applet
      code="interfaceAppli.class"
      width="400"
      height="300">
    </applet>
  </body>
</html>
```

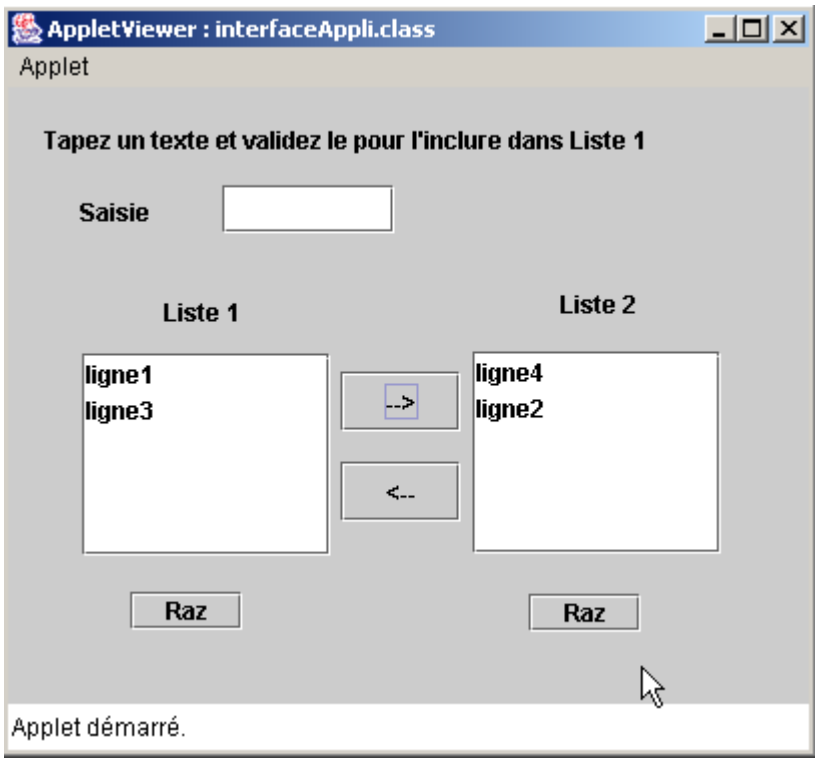
On a là, un document HTML classique si ce n'est la présence du tag **applet.** Celui-ci a été utilisé avec trois paramètres :

code="interfaceAppli.class"	nom de la classe JAVA compilée à charger pour exécuter l'applet
width="400"	largeur du cadre de l'applet dans le document
height="300"	hauteur du cadre de l'applet dans le document

Lorsque le fichier *appli.htm* a été écrit, on peut le charger avec le programme *appletviewer* du JDK ou avec un navigateur. On tape dans une fenêtre Dos la commande suivante dans le dossier du fichier *appli.htm* :

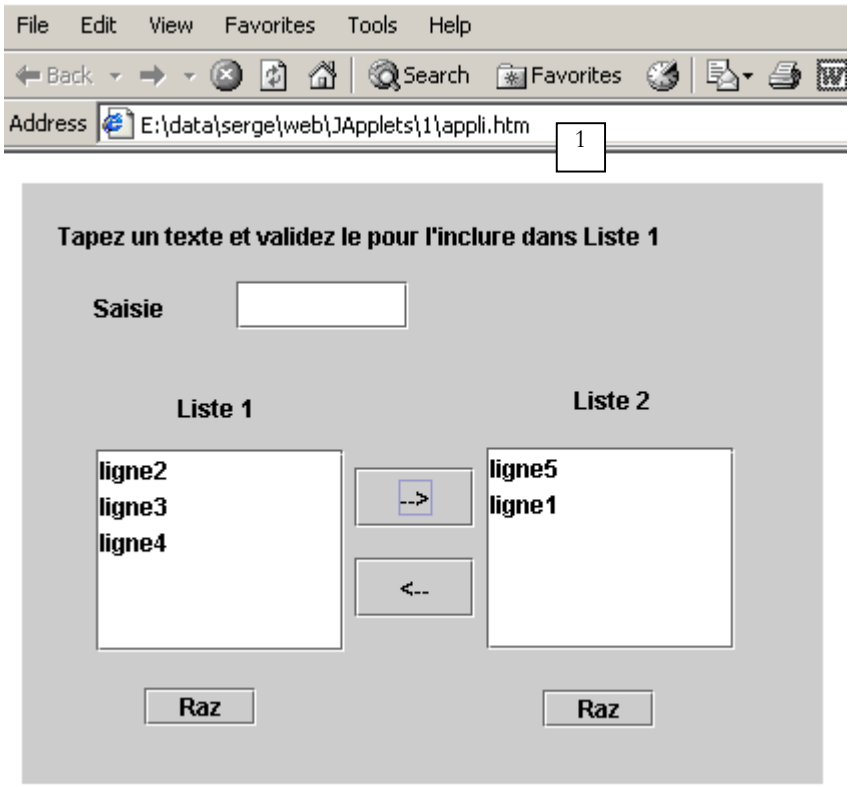
appletviewer appli.htm

On suppose ici que le répertoire de l'exécutable *appletviewer.exe* est dans le PATH de la fenêtre Dos, sinon il faudrait donner le chemin complet de l'exécutable *appletviewer.exe*. On obtient la fenêtre suivante :

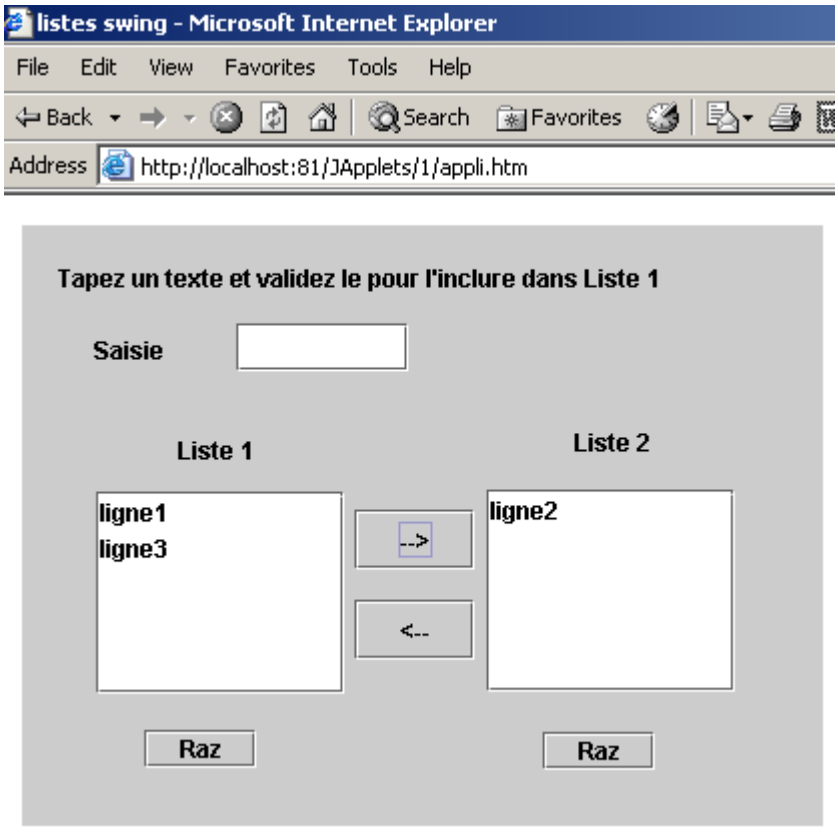


On arrête l'exécution de l'applet avec l'option *Applet/Quitter*. Testons maintenant l'applet avec un navigateur. Celui-ci doit utiliser une machine virtuelle Java 2 pour pouvoir afficher les composants swing. Jusqu'à récemment (2001), cette contrainte posait problème car Sun produisait des JDK que les éditeurs de navigateurs ne suivaient qu'avec beaucoup de retard. Finalement Sun a mis un terme à cet obstacle en mettant à disposition des utilisateurs une application appelée "Java plugin" qui permet aux navigateurs Internet Explorer et Netscape Navigator d'utiliser les tout derniers JRE produits par Sun (JRE=Java Runtime Environment). Les tests qui suivent ont été faits avec IE 5.5 muni du plugin Java 1.4.

Une première façon de tester l'applet *interfaceAppli.class* est de charger le fichier HTML *appli.htm* directement dans le navigateur en double-cliquant dessus. Le navigateur affiche alors la page HTML et son applet sans l'aide d'un serveur Web :



On voit d'après l'URL (1) que le fichier a été chargé dans le navigateur directement. Dans l'exemple suivant, le fichier *appli.htm* a été demandé à un serveur Web Apache travaillant sur le port 81 de la machine locale :



4.6.4 L'option de mise en forme <applet> dans un document HTML

Nous avons vu qu'au sein d'un document HTML, l'applet était référencée par la commande de mise en forme (tag) **<applet>**. Cette commande peut avoir divers paramètres :

```
<applet
 code=
 width=
 height=
 codebase=
 align=
 hspace=
 vspace=
    alt=
>
 <param name1=nom1 value=val1>
 <param name2=nom2 value=val2>
 texte
>
</applet>
```

La signification des paramètres est la suivante :

Paramètre	Signification
code	obligatoire - nom du fichier .class à exécuter
width	obligatoire - largeur de l'applet dans le document
height	obligatoire - hauteur …
codebase	facultatif - URL du répertoire contenant l'applet à exécuter. Si codebase n'est pas précisé, l'applet sera cherché dans le répertoire du document html qui la référence
align	facultatif - positionnement (LEFT, RiGHT, CENTER) de l'applet dans le document
hspace	facultatif - marge horizontale entourant l'applet exprimée en pixels
vspace	facultatif - marge verticale entourant l'applet exprimée en pixels
alt	facultatif - texte écrit en lieu et place de l'applet si le navigateur ne peut le charger
param	facultatif - paramètre passé à l'applet précisant son nom (name) et sa valeur (value). L'applet pourra récupérer la valeur du paramètre nom1 par **val1=getParameter("nom1")** On peut utiliser autant de paramètres que l'on veut
texte	facultatif - sera affiché par tout navigateur incapable d'exécuter une applet, par exemple parce qu'il ne possède pas de machine virtuelle Java.

Prenons un exemple pour illustrer le passage de paramètres à une applet :

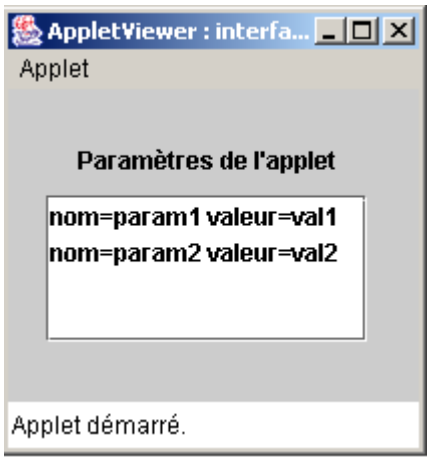
```
<html>
  <head>
    <title>paramètres applet</title>
  </head>
  <body>
    <applet code="interfaceParams.class" width="400" height="300">
       <param name="param1" value="val1">
       <param name="param2" value="val2">
    </applet>
  </body>
</html>
```

Le code Java de l'applet *interfaceParams* a été généré comme indiqué précédemment. On a construit une application JBuilder puis on a fait les quelques transformations mentionnées plus haut :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class interfaceParams extends JApplet {
  JPanel contentPane;
  JScrollPane jScrollPane1 = new JScrollPane();
  JLabel jLabel1 = new JLabel();
  DefaultListModel params=new DefaultListModel();
  JList lstParams = new JList(params);

  //Construire le cadre
  public void init() {
    // enableEvents(AWTEvent.WINDOW_EVENT_MASK);
```

```
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    // autres initialisations
    moreInit();
  }

  // initialisations
  public void moreInit(){
    // affiche les valeurs des paramètres de l'applet
    params.addElement("nom=param1 valeur="+getParameter("param1"));
    params.addElement("nom=param2 valeur="+getParameter("param2"));
  }//more Init

  //Initialiser le composant
  private void jbInit() throws Exception  {
.............
    this.setSize(new Dimension(205, 156));
    //this.setTitle("Paramètres d'une applet");
    jScrollPane1.setBounds(new Rectangle(19, 53, 160, 73));
.............
  }
}
```

L'exécution avec AppletViewer :



4.6.5 Accéder à des ressources distantes depuis une applet

Beaucoup d'applications sont amenées à exploiter des informations présentes dans des fichiers ou des bases de données. Nous avons indiqué qu'une applet n'avait pas accès aux ressources de la machine sur laquelle elle s'exécute. C'est une mesure de bon sens. Sinon il suffirait d'écrire une applet pour "espionner" le disque de ceux qui la chargent. L'applet a néammoins accès aux ressources du serveur à partir duquel elle a été téléchargée, par exemple des fichiers. C'est ce que nous allons voir maintenant.

4.6.5.1 La classe URL

Toute application Java peut lire un fichier présent sur une machine du réseau grâce à la classe *java.net.URL* (URL=Uniform Resource Locator). Une URL identifie une ressource du réseau, la machine sur laquelle il se trouve ainsi que le protocole et le port de communication à utiliser pour la récupérer sous la forme :

protocole:port//machine/fichier

Ainsi l'URL *http://www.ibm.com/index.html* désigne le fichier *index.html* de la machine *www.ibm.com*. Il est accessible par le protocole *http*. Le port n'est pas précisé : le navigateur utilisera le port 80 qui est le port par défaut du service http.

Détaillons quelques-uns des éléments de la classe URL :

public URL(String spec)	crée une instance d'URL à partir d'une chaîne du type "protocol:port//machine/fichier" - lance une exception si la syntaxe de la chaîne ne correspond pas à une URL
public String getFile()	permet d'avoir le champ fichier de la chaîne "protocole:port//machine/**fichier**" de l'URL
public String getHost()	permet d'avoir le champ machine de la chaîne "protocole:port//**machine**/fichier" de l'URL
public String getPort()	permet d'avoir le champ port de la chaîne "protocole:**port**//machine/fichier" de l'URL
public String getProtocol()	permet d'avoir le champ protocole de la chaîne "**protocole**:port//machine/fichier" de l'URL
public URLConnection openConnection()	ouvre la connexion avec la machine distante **getHost()** sur son port **getPort()** selon le protocole **getProtocol()** en vue de lire le fichier **getFile()**. Lance une exception si la connexion n'a pu être ouverte
public final InputStream openStream()	raccourci pour *openConnection().getInputStream()*. Permet d'obtenir un flux d'entrée à partir duquel le contenu du fichier **getFile()** va pouvoir être lu. Lance une exception si le flux n'a pu être obtenu
public String toString()	affiche l'identité de l'instance d'URL

Il y a ici deux méthodes qui nous intéressent :

- **public URL(String spec)** pour spécifier le fichier à exploiter
- **public final InputStream openStream()** pour l'exploiter

4.6.5.2 Un exemple console

On va écrire un programme Java, sans interface graphique, chargé d'afficher à l'écran le contenu d'une URL qu'on lui passe en paramètre. Un exemple d'appel pourrait être le suivant :

DOS> **java urlcontenu http://istia.univ-angers.fr/index.html**

Le programme est relativement simple :

```
import java.net.*;      // pour la classe URL
import java.io.*;       // pour les flux (stream)

public class urlcontenu{

// affiche le contenu de l'URL passée en argument
// ce contenu doit être du texte pour pouvoir ^tre lisible

  public static void main (String arg[]){
    // vérification des arguments
    if(arg.length==0){
      System.err.println("Syntaxe pg url");
      System.exit(0);
    }

    try{
      // création de l'URL
      URL url=new URL(arg[0]);
      // lecture du contenu
      try{
        // création du flux d'entrée
        BufferedReader is=new BufferedReader(new InputStreamReader(url.openStream()));
        try{
          // lecture des lignes de texte dans le flux d'entrée
          String ligne;
          while((ligne=is.readLine())!=null)
            System.out.println(ligne);
        } catch (Exception e){
            System.err.println("Erreur lecture : " +e);
        }
        finally{
          try { is.close(); } catch (Exception e) {}
        }
```

```
      } catch (Exception e){
          System.err.println("Erreur openStream : " +e);
      }
    } catch (Exception e){
      System.err.println("Erreur création URL : " +e);
    }
  }// fin main
}// fin classe
```

Après avoir vérifié qu'il y avait bien un argument, on tente de créer une URL avec celui-ci :

```
// création de l'URL
      URL url=new URL(arg[0]);
```

Cette création peut échouer si l'argument ne respecte pas la syntaxe des URL *protocole:port//machine/fichier*. Si on a une URL syntaxiquement correcte, on essaie de créer un flux d'entrée à partir duquel on pourra lire des lignes de texte. Le flux d'entrée fourni par une URL *URL.openStream()* fournit un flux de type *InputStream* qui est un flux orienté caractères : la lecture se fait caractère par caractère et il faut former les lignes soi-même. Pour pouvoir lire des lignes de texte, il faut utiliser la méthode *readLine* des classes *BufferedReader* ou *DataInputStream*.On a choisi ici *BufferedReader*. Il ne nous reste qu'à transformer le flux *InputStream* de l'URL en flux *BufferedReader*. Cela se fait en créant un flux intermédiaire *InputStreamReader* :

```
BufferedReader is=new BufferedReader(new InputStreamReader(url.openStream()));
```

Cette création de flux peut échouer. On gère l'exception correspondante. Une fois le flux obtenu, il ne nous reste plus qu'à y lire les lignes de texte et à afficher celles-ci à l'écran.

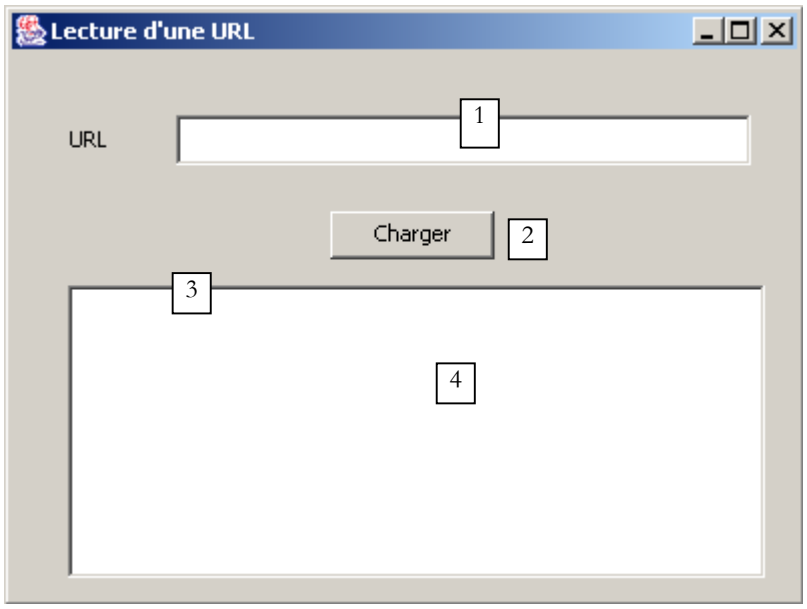
```
     String ligne;
        while((ligne=is.readLine())!=null)
            System.out.println(ligne);
```

Là encore la lecture peut engendrer une exception qui doit être gérée. Voici un exemple d'exécution du programme qui demande une URL locale :

```
E:\data\serge\Jbuilder\interfaces\JApplets\3>java urlcontenu http://localhost
<html>

<head>


<title>Bienvenue dans le Serveur Web personnel</title>
</head>

<body bgcolor="#FFFFFF" link="#0080FF" vlink="#0080FF"
topmargin="0" leftmargin="0">

<table border="0" cellpadding="0" cellspacing="0" width="100%"
bgcolor="#000000">
.............
                </table>
                </center></div></td>
            </tr>
        </table>
        </td>
    </tr>
</table>
</body>
</html>
```

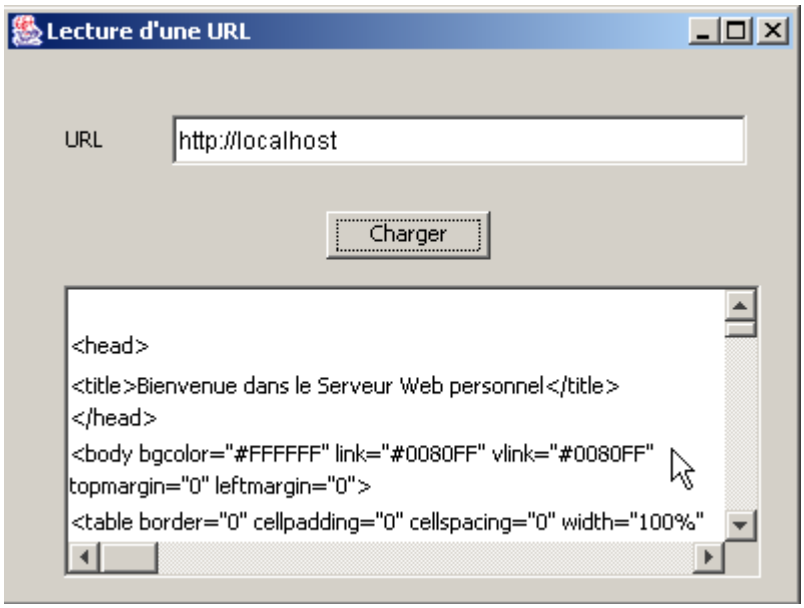
4.6.5.3 Un exemple graphique

L'interface graphique sera la suivante :



Numéro	Nom	Type	Rôle
1	txtURL	JTextField	URL à lire
2	btnCharger	JButton	bouton lançant la lecture de l'URL
3	JScrollPane1	JScrollPane	panneau défilant
4	lstURL	JList	liste affichant le contenu de l'URL demandée

A l'exécution, nous avons un résultat analogue à celui du programme console :



Le code pertinent de l'application est le suivant :

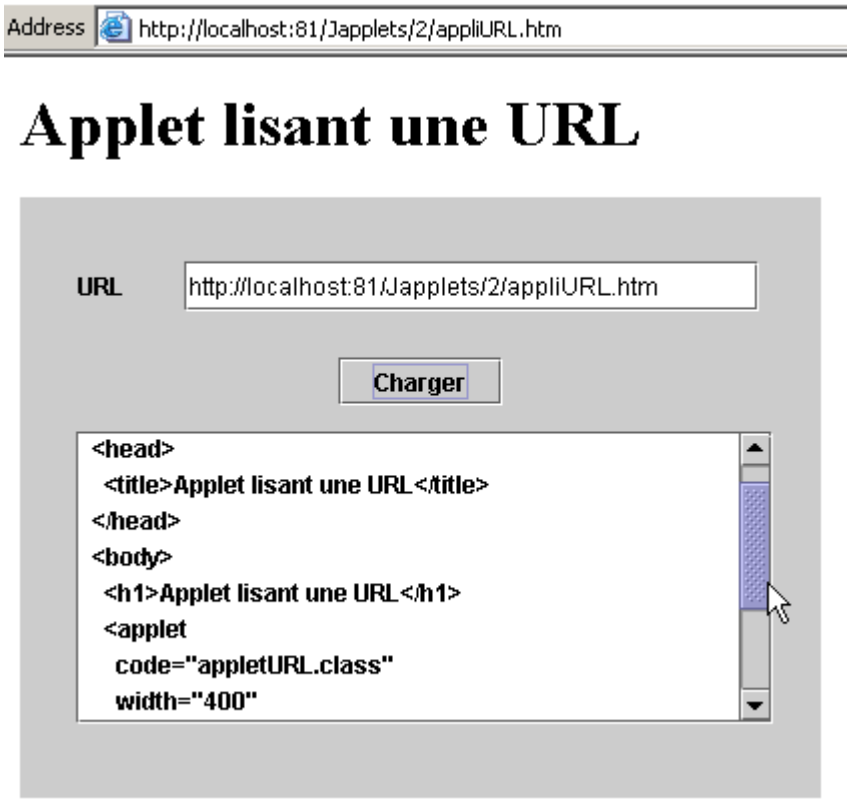
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.net.*;
import java.io.*;

public class interfaceURL extends JFrame {
  JPanel contentPane;
  JLabel jLabel1 = new JLabel();
  JTextField txtURL = new JTextField();
  JButton btnCharger = new JButton();
```

```
  JScrollPane jScrollPane1 = new JScrollPane();
  DefaultListModel lignes=new DefaultListModel();
  JList lstURL = new JList(lignes);

  //Construire le cadre
  public interfaceURL() {
..............
  }

  //Initialiser le composant
  private void jbInit() throws Exception  {
..............
  }

  //Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée
  protected void processWindowEvent(WindowEvent e) {
...................
  }

  void txtURL_caretUpdate(CaretEvent e) {
    // positionne l'état du bouton Charger
    btnCharger.setEnabled(! txtURL.getText().trim().equals(""));
  }

  void btnCharger_actionPerformed(ActionEvent e) {
    // affiche le contenu de l'URL dans la liste
    try{
      afficherURL(txtURL.getText().trim());
    }catch(Exception ex){
      // affichage erreur
      JOptionPane.showMessageDialog(this,"Erreur : " +
ex.getMessage(),"Erreur",JOptionPane.ERROR_MESSAGE);
    }//try-catch
  }

  private void afficherURL(String strURL) throws Exception {
    // affiche le contenu de l'URL strURL dans la liste
    // aucune exception n'est gérée spécifiquement. On se contente de les remonter

    // création de l'URL
    URL url=new URL(strURL);
    // création du flux d'entrée
    BufferedReader IN=new BufferedReader(new InputStreamReader(url.openStream()));
    // lecture des lignes de texte dans le flux d'entrée
    String ligne;
    while((ligne=IN.readLine())!=null)
      lignes.addElement(ligne);
    // fermeture flux de lecture
    IN.close();
  }
}
```

4.6.5.4 Une applet

L'application graphique précédente est transformée en applet comme il a été présenté plusieurs fois. L'applet est insérée dans un document HTML *appliURL.htm* :

```
<html>
  <head>
    <title>Applet lisant une URL</title>
  </head>
  <body>
    <h1>Applet lisant une URL</h1>
    <applet
      code="appletURL.class"
      width="400"
      height="300"
    >
    </applet>
  </body>
</html>
```

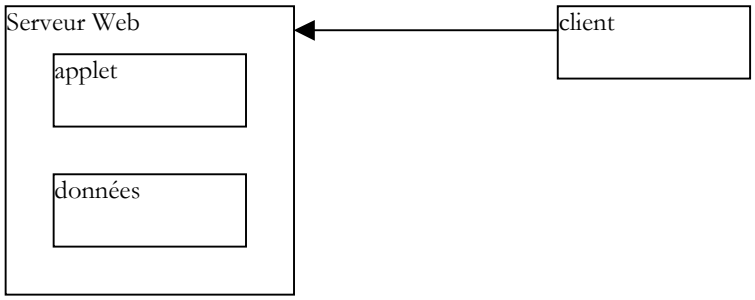


Dans l'exemple ci-dessus, le navigateur a demandé l'URL *http://localhost:81/Japplets/2/appliURL.htm* à un serveur web *apache* fonctionnant sur le port 81. L'applet a été alors affichée dans le navigateur. Dans celle-ci, on a demandé de nouveau l'URL *http://localhost:81/Japplets/2/appliURL.htm* pour vérifier qu'on obtenait bien le fichier *appliURL.htm* qu'on avait construit. Maintenant essayons de charger une URL n'appartenant pas à la machine qui a délivré l'applet (ici *localhost*) :

Le chargement de l'URL a été refusé. On retrouve ici la contrainte liée aux applets : elles ne peuvent accéder qu'aux seules ressources réseau de la machine à partir de laquelle elles ont été téléchargées. Il y un moyen pour l'applet de contourner cette contrainte qui est de déléguer les demandes réseau à un programme serveur situé sur la machine d'où elle a été téléchargée. Ce programme fera les demandes réseau à la place de l'applet et lui renverra les résultats. On appelle cela un programme relais.

4.7 L'applet IMPOTS

Nous allons transformer maintenant l'application graphique IMPOTS en applet. Cela présente un certain intérêt : l'application deviendra disponible à tout internaute disposant d'un navigateur et d'un Java plugin 1.4 (à cause du composant *JSpinner*). Notre application devient ainsi planétaire... Cette modification va nécessiter un peu plus que la simple transformation application graphique --> applet devenue maintenant classique. L'application utilise une option de menu *Initialiser* dont le but est de lire le contenu d'un fichier local. Or si on se représente une application Web, on voit que ce schéma ne tient plus :



Il est évident que les données définissant les barèmes de l'impôt seront sur le serveur web et non pas sur chacune des machines clientes. Le fichier à lire sur le serveur web sera ici placé dans le même dossier que l'applet et l'option *Initialiser* devra aller le chercher là. Les exemples précédents ont montré comment faire cela. Par ailleurs, pour sélectionner le fichier des données localement, on avait utilisé un composant *JFileChooser* qui n'a plus lieu d'être ici.

Le document HTML contenant l'applet sera le suivant :

```
<html>
  <head>
    <title>Applet de calcul d'impôts</title>
  </head>
  <body>
    <h2>Calcul d'impôts</h2>
    <applet
      code="appletImpots.class"
      width="320"
      height="270"
    >
    <param name="data" value="impots.txt">
    </applet>
  </body>
</html>
```

On notera la paramètre *data* dont la valeur est le nom du fichier contenant les données du barème de l'impôt. Le document HTML, la classe *appletImpots*, la classe *impots* et le fichier *impots.txt* sont tous dans le même dossier du serveur Web :

```
E:\data\serge\web\JApplets\impots>dir
12/06/2002  13:24               247 impots.txt
13/06/2002  10:17               654 appletImpots$2.class
13/06/2002  10:17               653 appletImpots$3.class
13/06/2002  10:17               653 appletImpots$4.class
13/06/2002  10:17               651 appletImpots$5.class
13/06/2002  10:06               651 appletImpots$6.class
13/06/2002  10:17             8 655 appletImpots.class
13/06/2002  10:17               657 appletImpots$1.class
13/06/2002  10:24               286 appletImpots.htm
13/06/2002  10:17             1 305 impots.class
```

Le code de l'applet est le suivant (nous n'avons mis en relief que les changements par rapport à l'application graphique) :

```
...........
import java.net.*;
import java.applet.Applet;

public class appletImpots extends JApplet {
  // les composants de la fenêtre
  JPanel contentPane;
.........................

  // les attributs de la classe
  double[] limites=null;
  double[] coeffr=null;
  double[] coeffn=null;
  impots objImpots=null;
```

```
  String urlDATA=null;

  //Construire le cadre
  public void init() {
    //enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    // autres initialisations
    moreInit();
  }

  // initialisation formulaire
  private void moreInit(){
    // menu Calculer inhibé
    mnuCalculer.setEnabled(false);
.................
    // on récupère le nom du fichier du barême des impôts
    String nomFichier=getParameter("data");
    // erreur ?
    if(nomFichier==null){
      // msg d'erreur
      txtStatus.setText("Le paramètre data de l'applet n'a pas été initialisé");
      // on bloque l'option Initialiser
      mnuInitialiser.setEnabled(false);
      // fin
      return;
    }//if
    // on fixe l'URL des données
    urlDATA=getCodeBase()+"/"+nomFichier;
  }//moreInit

  //Initialiser le composant
  private void jbInit() throws Exception  {
...................
  }

  void mnuQuitter_actionPerformed(ActionEvent e) {
    // on quitte l'application
    System.exit(0);
  }

  void mnuInitialiser_actionPerformed(ActionEvent e) {
    // on charge le fichier des données
    try{
      // lecture données
      lireDATA();
      // création de l'objet impots
      objImpots=new impots(limites,coeffr,coeffn);
.................
  }

  private void lireDATA() throws Exception {
    // les tableaux de données
    ArrayList aLimites=new ArrayList();
    ArrayList aCoeffR=new ArrayList();
    ArrayList aCoeffN=new ArrayList();
    String[] champs=null;

    // ouverture fichier en lecture
    BufferedReader IN=new BufferedReader(new InputStreamReader(new URL(urlDATA).openStream()));
    // on lit le fichier ligne par ligne
.....................
  }

  void mnuCalculer_actionPerformed(ActionEvent e) {
    // calcul de l'impôt
.......................
  }

  void txtSalaire_caretUpdate(CaretEvent e) {
...........
  }

  void mnuEffacer_actionPerformed(ActionEvent e) {
...............
  }
}
```

Nous ne commentons ici que les modifications amenées par le fait que le fichier des données à lire est maintenant sur une machine distante plutôt que sur une machine locale :

Le nom de l'URL du fichier des données à lire est obtenu par le code suivant :

```
    // on récupère le nom du fichier du barême des impôts
    String nomFichier=getParameter("data");
    // erreur ?
    if(nomFichier==null){
      // msg d'erreur
      txtStatus.setText("Le paramètre data de l'applet n'a pas été initialisé");
      // on bloque l'option Initialiser
      mnuInitialiser.setEnabled(false);
      // fin
      return;
    }//if
    // on fixe l'URL des données
    urlDATA=getCodeBase()+"/"+nomFichier;
```

Rappelons-nous que le nom du fichier "*impots.txt*" est passé dans le paramètre *data* de l'applet :

```
    <param name="data" value="impots.txt">
```

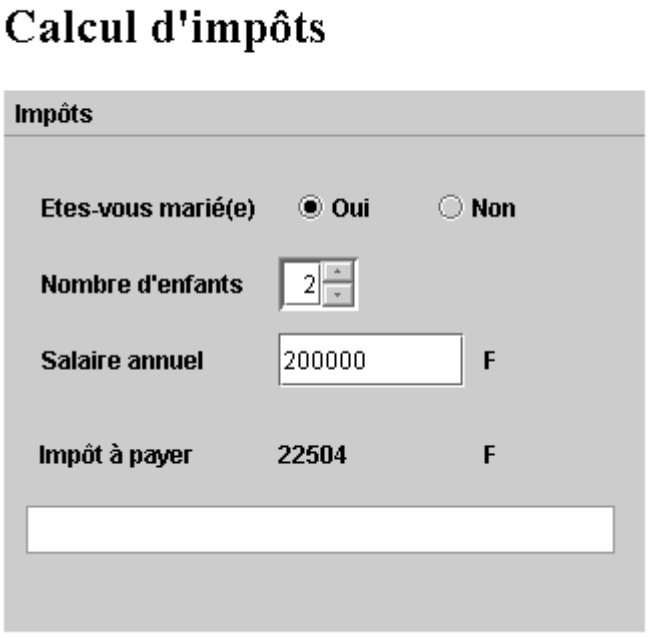
Le code ci-dessus commence donc par récupérer la valeur du paramètre *data* en gérant une éventuelle erreur. Si l'URL du document HTML contenant l'applet est *http://localhost:81/JApplets/impots/appletImpots.htm*, l'URL du fichier *impots.txt* sera *http://localhost:81/JApplets/impots/impots.txt*. Il nous faut construire le nom de cette URL. La méthode *getCodeBase()* de l'applet donne l'URL du dossier où a été récupéré le document HTML contenant l'applet, donc dans notre exemple *http://localhost:81/JApplets/impots*. L'instruction suivante permet donc de construire l'URL du fichier des données :

```
    urlDATA=getCodeBase()+"/"+nomFichier;
```

On trouve dans la méthode *lireFichier()* qui lit le contenu de l'URL *urlData*, la création du flux d'entrée qui va permettre de lire les données :

```
    // ouverture fichier en lecture
    BufferedReader IN=new BufferedReader(new InputStreamReader(new URL(urlDATA).openStream()));
```

A partir de là, on ne peut plus distinguer le fait que les données viennent d'un fichier distant plutôt que local. Voici un exemple d'exécution de l'applet :



4.8 Conclusion

Ce chapitre a présenté

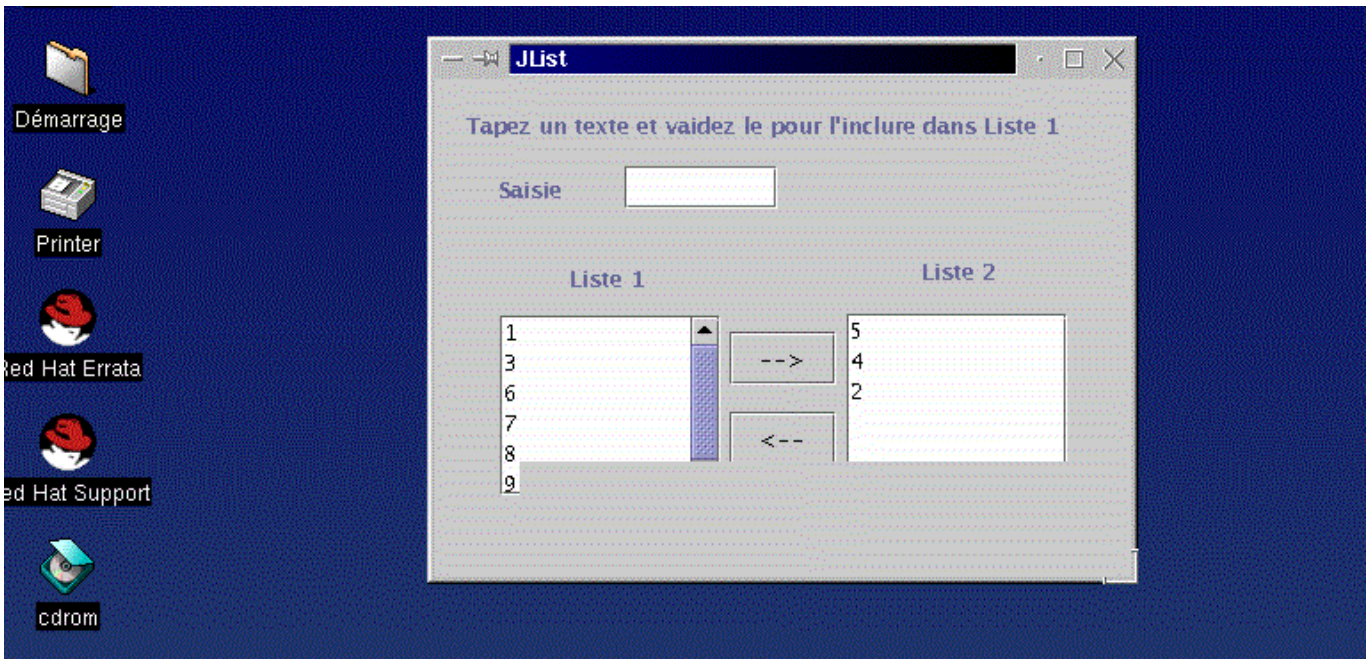
- une introduction à la construction d'interfaces graphiques avec Jbuilder
- les composants swing les plus courants
- la construction d'applets

Nous retiendrons que

- le code généré par JBuilder peut être écrit à la main. Une fois ce code obtenu d'une manière ou d'une autre, un simple JDK suffit pour l'exécuter et JBuilder n'est alors plus indispensable.
- l'utilisation d'un outil tel que JBuilder peut amener des gains de productivité importants :
  - s'il est possible d'écrire à la main le code généré par JBuilder, cela peut prendre beaucoup de temps et ne présente que peu d'intérêt, la logique de l'application étant en général ailleurs.
  - le code généré peut être instructif. Le lire et l'étudier est une bonne façon de découvrir certaines méthodes et propriétés de composants qu'on utilise pour la première fois.
  - JBuilder est multi plate-formes. On conserve donc ses acquis en passant d'une plate-forme à l'autre. Pouvoir écrire des programmes fonctionnant à la fois sous windows et Linux est bien sûr un facteur de productivité très important. Mais il est dû à Java lui-même et non à JBuilder.

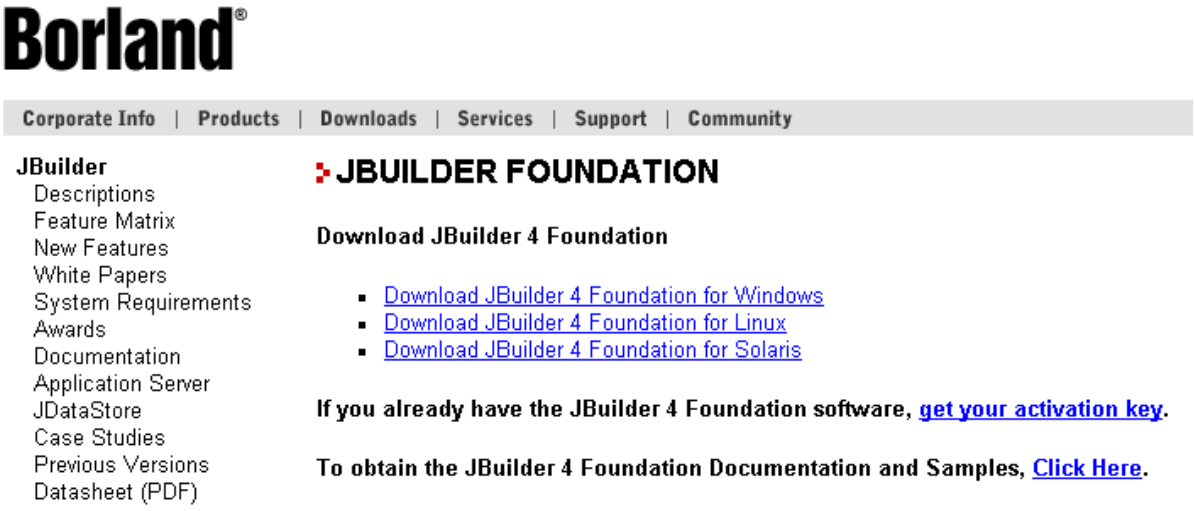
4.9 Jbuilder sous Linux

Tous les exemples précédents ont été testés sous Windows 98. On peut se demander si les programmes écrits sont portables tels quels sous Linux. Moyennant que la machine Linux en question ait les classes utilisées par les différents programmes, ils le sont. Si vous avez par exemple installé JBuilder sous Linux les classes nécessaires sont déjà sur votre machine. Voici par exemple ce que donne l'exécution d'un de nos programmes sur le composant JList avec JBuilder 4 sous Linux :



Nous présentons ci-dessous l'installation de JBuilder 4 Foundation sur une machine Linux. Son installation sur une machine Win9x ne pose aucun problème et est analogue au processus qui va maintenant être décrit. L'installation des version sultérieures est sans doute différente mais ce document peut néanmoins quelques informations utiles.

Jbuilder est disponible sur le site d"Inprise à l'url *http://www.inprise.com/jbuilder*



On trouve sur cette page les liens pour notamment Windows et Linux. Nous décrivons maintenant l'installation de JBuilder sur une machine Linux ayant l'interface graphique KDE.

En suivant le lien Jbuilder 4 for Linux, un formulaire est présenté. On le remplit et au final on récupère deux fichiers :

jb4docs_fr.tar.gz	la documentation et les exemples de Jbuilder4
jb4fndlinux_fr.tar.gz	Jbuilder4 Foundation. C'est une version limitée du Jbuilder4 commercial mais suffisant dans un contexte éducatif.

Une clé d'activation du logiciel vous est envoyée par courrier électronique. Elle vous permettra d'utiliser Jbuilder4 et vous sera demandée dès la 1ère utilisation. Si vous perdez cette clé, vous pouvez revenir à l'url ci-dessus et suivre le lien "*get your activation key*". Nous supposons par la suite que vous êtes *root* et que vous êtes placé dans le répertoire des deux fichiers tar.gz ci-dessus. On décompresse les deux fichiers :

[tar xvzf *jb4fndlinux_fr.tar.gz*]
[tar xvzf *jb4docs_fr.tar.gz*]

```
[ls -l]
drwxr-xr-x    3 nobody   nobody       4096 oct 10  2000 docs
drwxr-xr-x    3 nobody   nobody       4096 déc  5 13:00 foundation
```

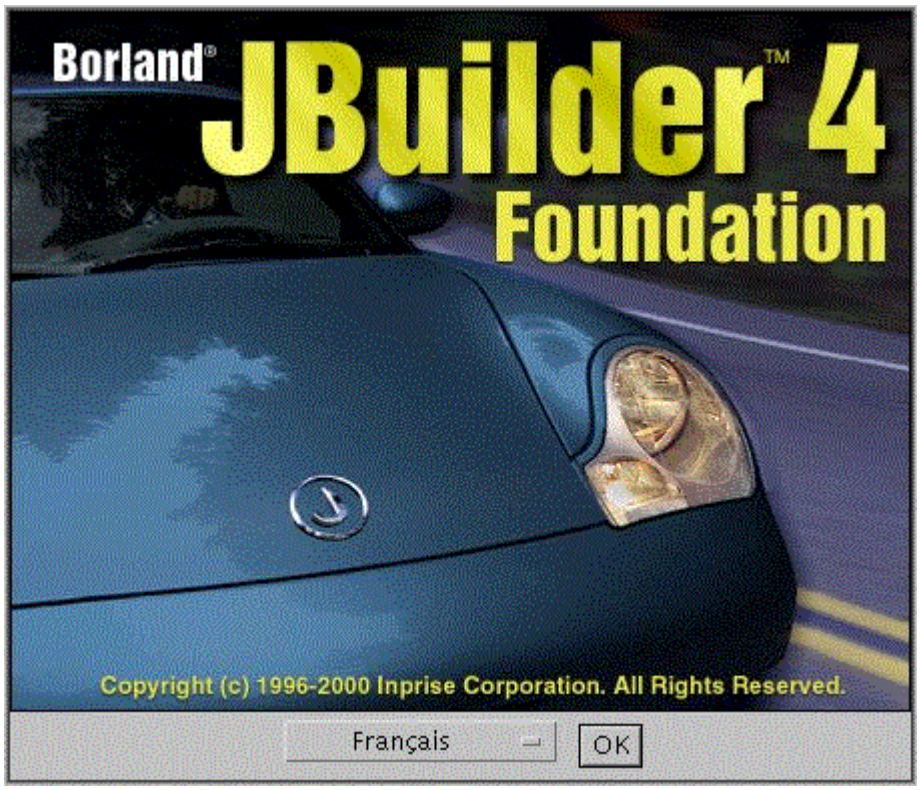
```
[ls -l foundation]
-rw-r--r--    1 nobody   nobody       1128 déc  5 13:00 deploy.txt
-rwxr-xr-x    1 nobody   nobody   69035365 déc  5 13:00 fnd_linux_install.bin
drwxr-xr-x    2 nobody   nobody       4096 déc  5 13:00 images
-rw-r--r--    1 nobody   nobody      15114 déc  5 13:00 index.html
-rw-r--r--    1 nobody   nobody      23779 déc  5 13:00 license.txt
-rw-r--r--    1 nobody   nobody      75739 déc  5 13:00 release_notes.html
-rw-r--r--    1 nobody   nobody      31902 déc  5 13:00 whatsnew.html
```

```
[ls -l docs]
-rw-r--r--    1 nobody   nobody       1128 oct 10  2000 deploy.txt
-rwxr-xr-x    1 nobody   nobody   40497874 oct 10  2000 doc_install.bin
drwxr-xr-x    2 nobody   nobody       4096 oct 10  2000 images
-rw-r--r--    1 nobody   nobody       9210 oct 10  2000 index.html
-rw-r--r--    1 nobody   nobody      23779 oct 10  2000 license.txt
-rw-r--r--    1 nobody   nobody      75739 oct 10  2000 release_notes.html
-rw-r--r--    1 nobody   nobody      31902 oct 10  2000 whatsnew.html
```

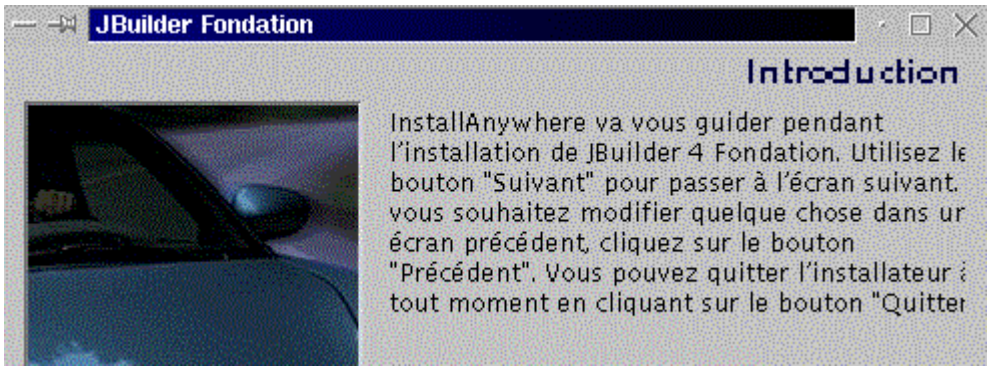
Dans les deux répertoires générés, le fichier *.bin* est le fichier d'installation. Par ailleurs, avec un navigateur affichez le fichier *index.html* de chacun des répertoires. Ils donnent la marche à suivre pour installer les deux produits. Commençons par installer Jbuilder Foundation :

```
[cd foundation]
[./fnd_linux_install.bin]
```

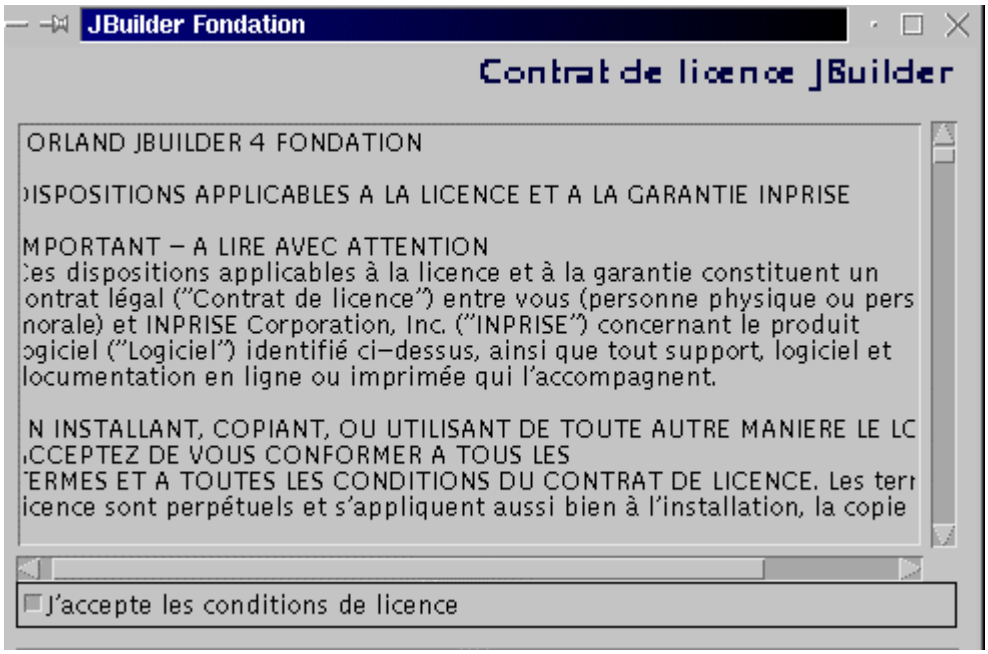
Arrive le 1er écran de l'installation. Vont suivre de nombreux autres :



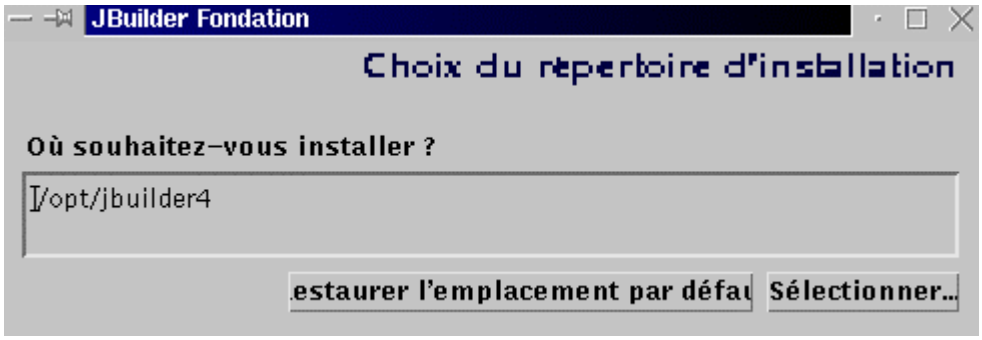
Validez. Suit un écran d'explications :



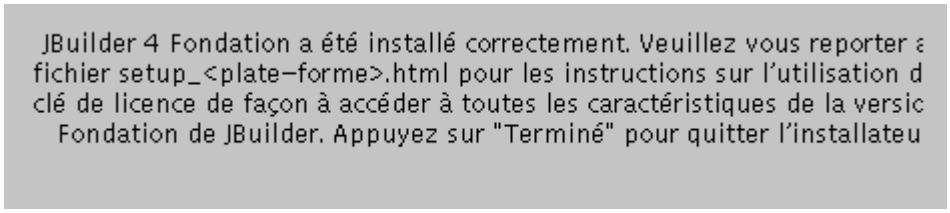
Faites [suivant].



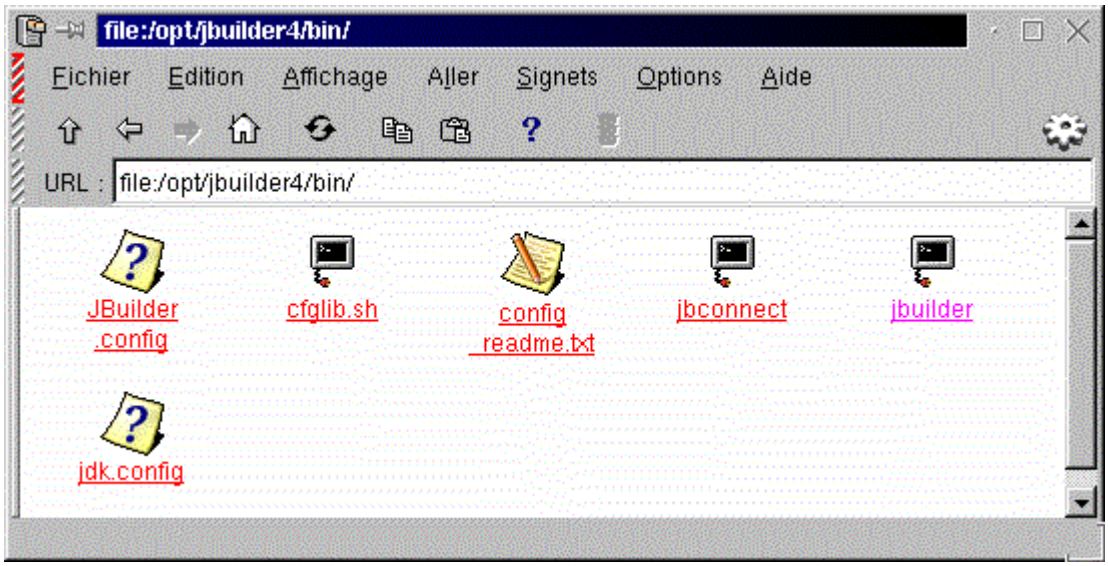
Acceptez le contrat de licence et faites [suivant].



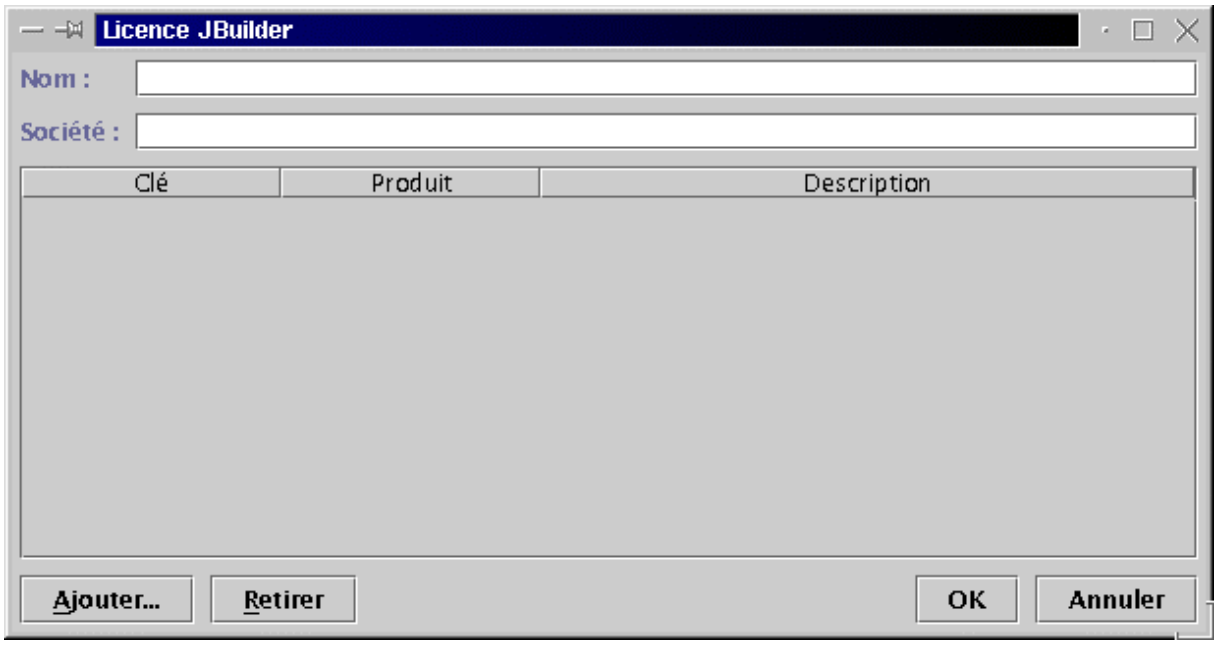
Acceptez l'emplacement proposé pour Jbuilder (notez le, vous en aurez besoin ultérieurement) et faites [suivant]. L'installation se fait rapidement.



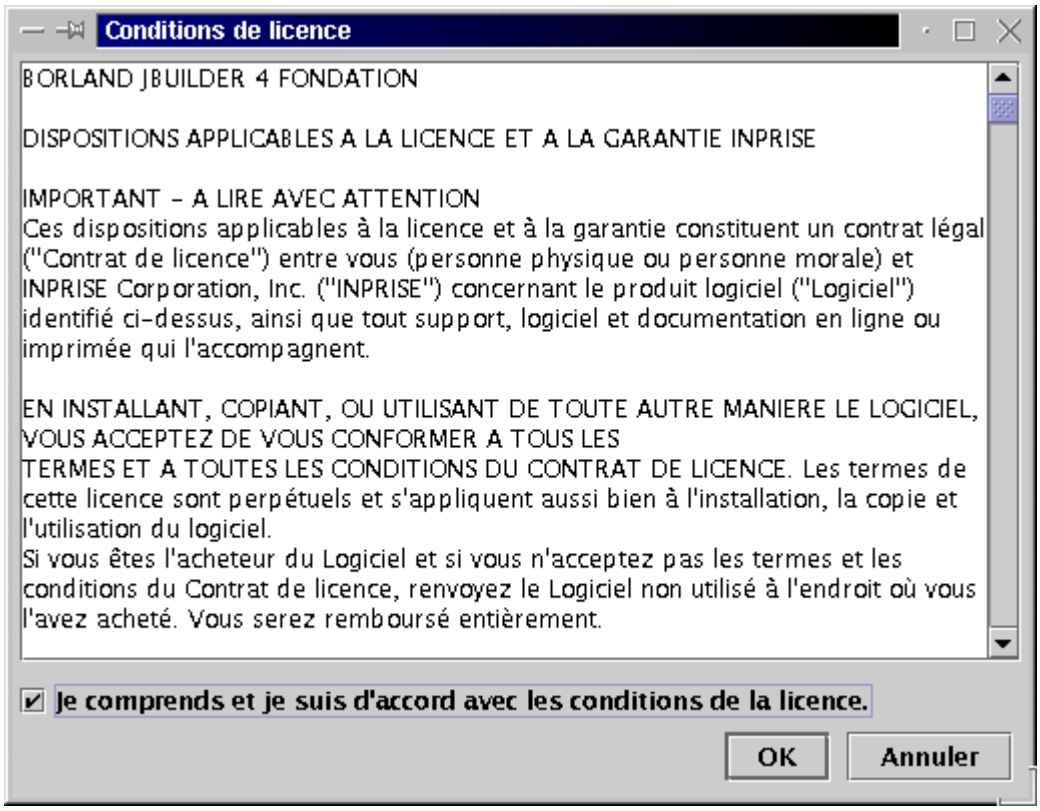
Nous sommes prêts pour un premier test. Dans KDE, lancez le gestionnaire de fichiers pour aller dans le répertoire des exécutables de Jbuilder : */opt/jbuilder4/bin* (si vous avez installé jbuilder dans */opt/jbuilder4*) :



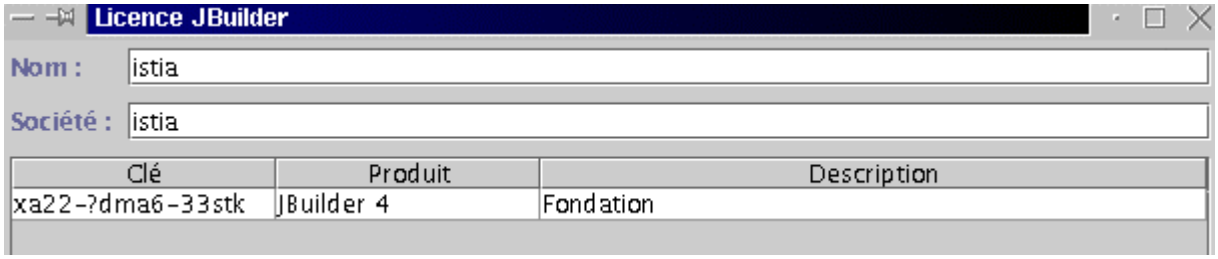
Cliquez sur l'icône *jbuilder* ci-dessus. Comme c'est la 1ère fois que vous utilisez Jbuilder, vous allez devoir entrer votre clé d'activation :



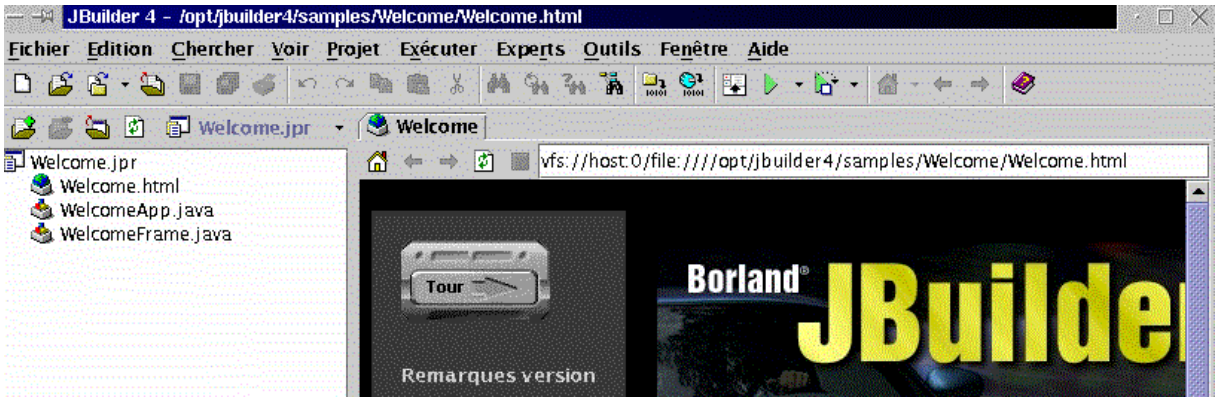
Remplissez les champs *Nom* et *Société*. Faites [Ajouter] pour entrer votre clé d'activation. On rappelle que celle-ci a du vous être envoyée par mail et que si vous l'avez perdue, vous pouvez l'avoir à l'url où vous avez récupéré Jbuilder 4 (cf début de l'installation). Une fois votre clé d'activation acceptée, on vous rappellera les conditions de licence :



En faisant OK, vous revenez à la fenêtre de saisie des informations déjà vue :



Faites OK pour terminer cette phase qui ne s'exécute que lors de la 1ère exécution. Apparaît ensuite, l'environnement de développement de Jbuilder :



Quittez Jbuilder pour installer maintenant la documentation. Celle-ci va installer un certain nombre de fichiers qui seront utilisés dans l'aide de Jbuilder, aide qui pour l'instant est incomplète. Revenez dans le répertoire où vous avez décompressé les fichiers tar.gz de Jbuilder. Le programme d'installation est dans le répertoire *docs*. Placez-vous dedans :

```
[cd docs]
[ls -l]
```

```
-rw-r--r--    1 nobody   nobody       1128 oct 10  2000 deploy.txt
-rwxr-xr-x    1 nobody   nobody   40497874 oct 10  2000 doc_install.bin
drwxr-xr-x    2 nobody   nobody       4096 oct 10  2000 images
-rw-r--r--    1 nobody   nobody       9210 oct 10  2000 index.html
-rw-r--r--    1 nobody   nobody      23779 oct 10  2000 license.txt
-rw-r--r--    1 nobody   nobody      75739 oct 10  2000 release_notes.html
-rw-r--r--    1 nobody   nobody      31902 oct 10  2000 whatsnew.html
```

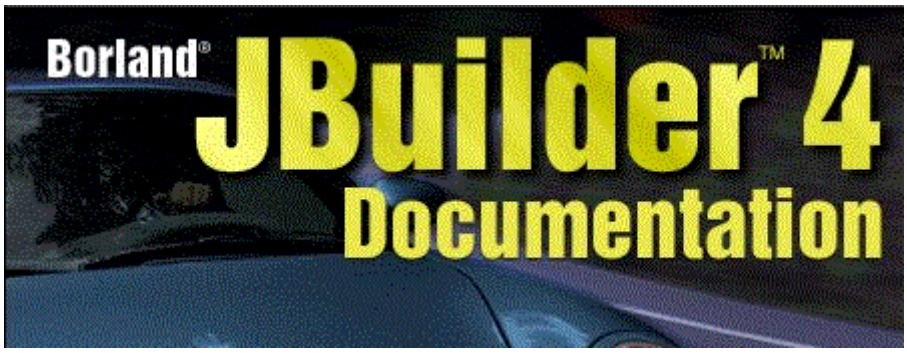
Le fichier d'installation est *doc_install.bin* mais on ne peut le lancer immédiatement. Si on le fait, l'installation échoue sans qu'on comprenne pourquoi. Lisez le fichier *index.html* avec un navigateur pour une description précise du mode d'installation. L'installateur a besoin d'une machine virtuelle java, en fait un programme appelé *java* et celui-ci doit être dans le PATH de votre machine. On rappelle que PATH est une variable Unix dont la valeur de la forme *rep1:rep2:...:repn* indique les répertoires *repi* qui doivent être explorés dans la recherche d'un exécutable. Ici, l'installateur demande l'exécution d'un programme *java*. Vous pouvez avoir plusieurs versions de ce programme selon le nombre de machines virtuelles Java que vous avez installées ici ou là. Assez logiquement, nous utiliserons celui amené par Jbuilder4 et qui se trouve dans */opt/jbuilder4/jdk1.3/bin*. Pour mettre ce répertoire dans la variable PATH :

```
[echo $PATH]
/usr/local/sbin:/usr/sbin:/sbin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/
usr/bin:/usr/X11R6/bin:/root/bin
[export PATH=/opt/jbuilder4/jdk1.3/bin/:$PATH]
[echo $PATH]
/opt/jbuilder4/jdk1.3/bin/:/usr/local/sbin:/usr/sbin:/sbin:/usr/local/sbin:/usr/local
/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
```

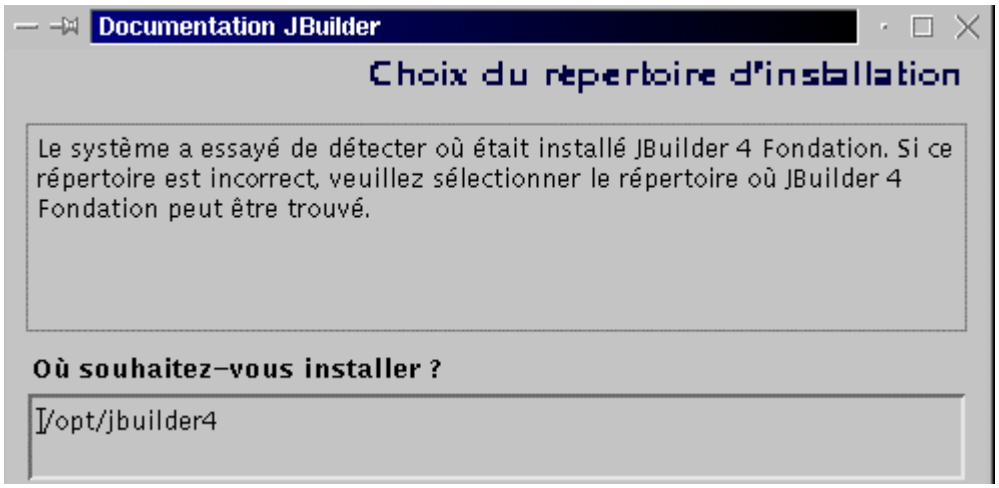
Nous pouvons maintenant lancer l'installation de la documentation :

```
[./doc_install.bin]
Preparing to install...
```

C'est une installation graphique qui va se faire :

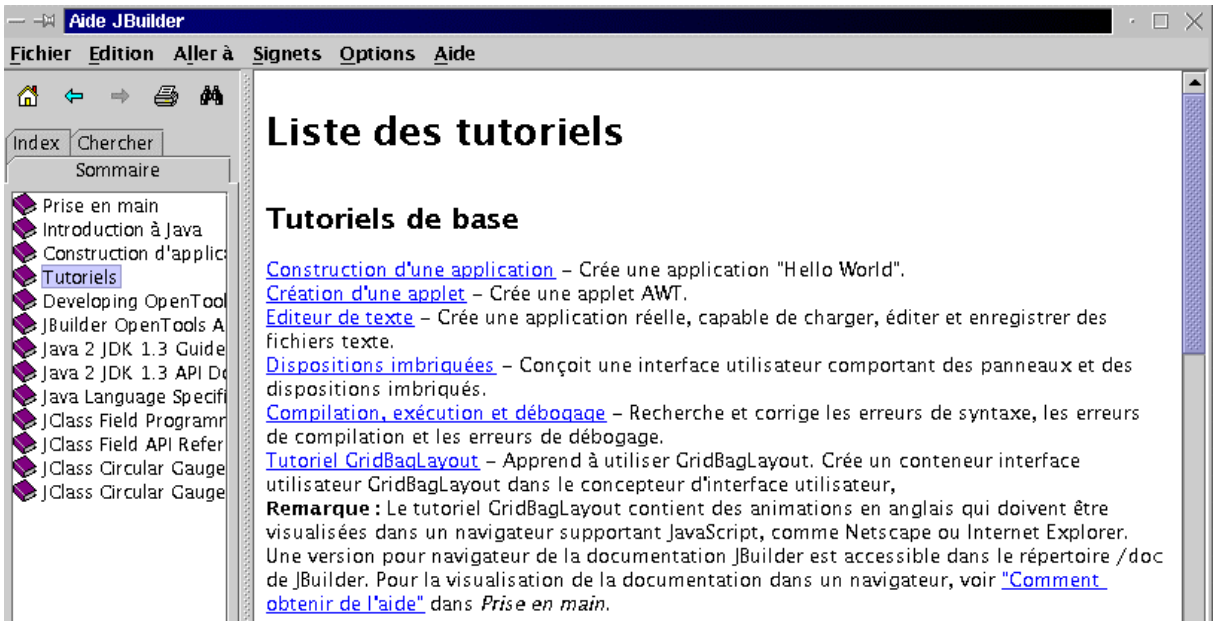


Elle est très analogue à celle de Jbuilder Foundation. Aussi ne la détaillerons-nous pas. Il y a un écran qu'il ne faut pas rater :

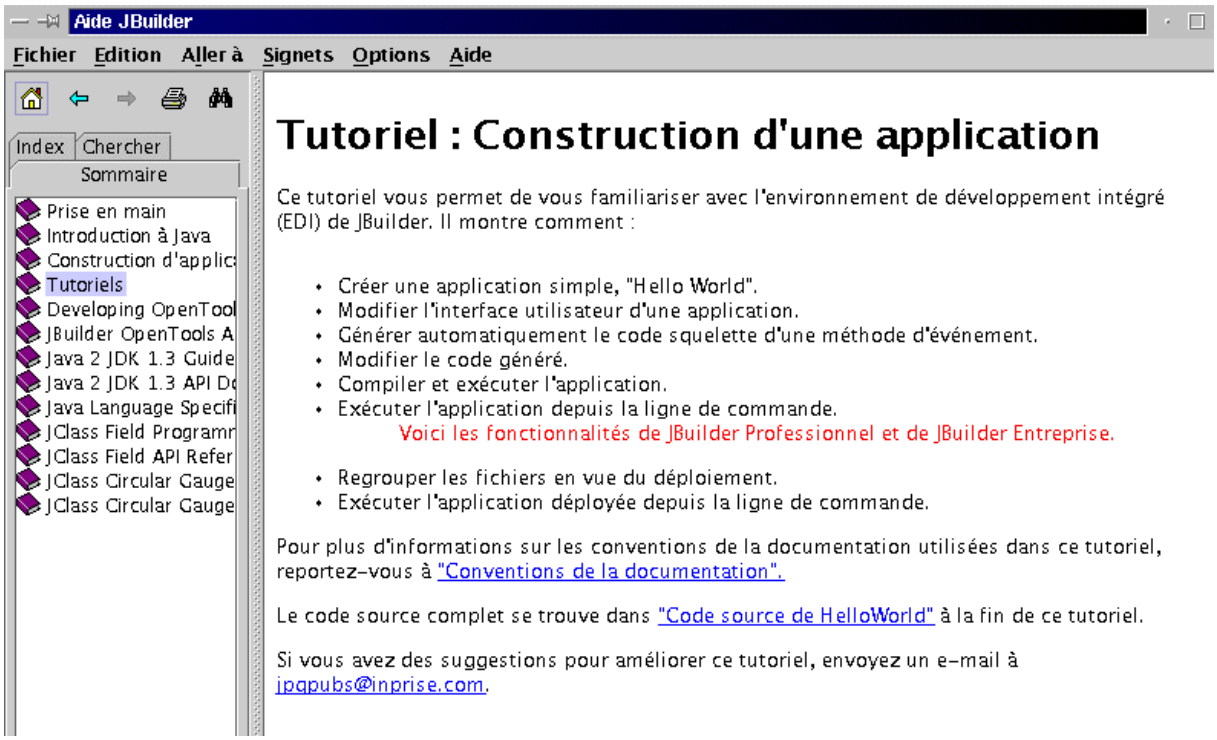


L'installateur doit normalement trouver tout seul où vous avez installé Jbuilder Foundation. Ne changez donc pas ce qui vous est proposé sauf bien sûr si c'était faux.

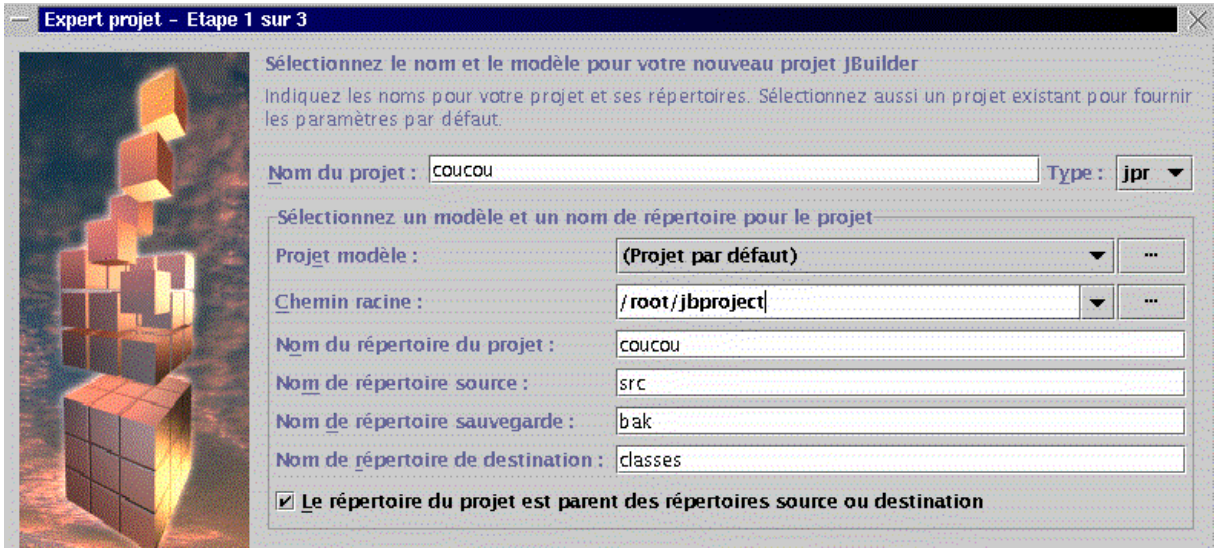
Une fois la documentation installée, nous sommes prêts pour un nouveau test de Jbuilder. Lancez l'application comme il a été montré plus haut. Une fois Jbuilder présent, prenez l'option *Aide/Rubrique d'aide*. Dans le cadre de gauche, cliquez sur le lien *Tutoriels* :



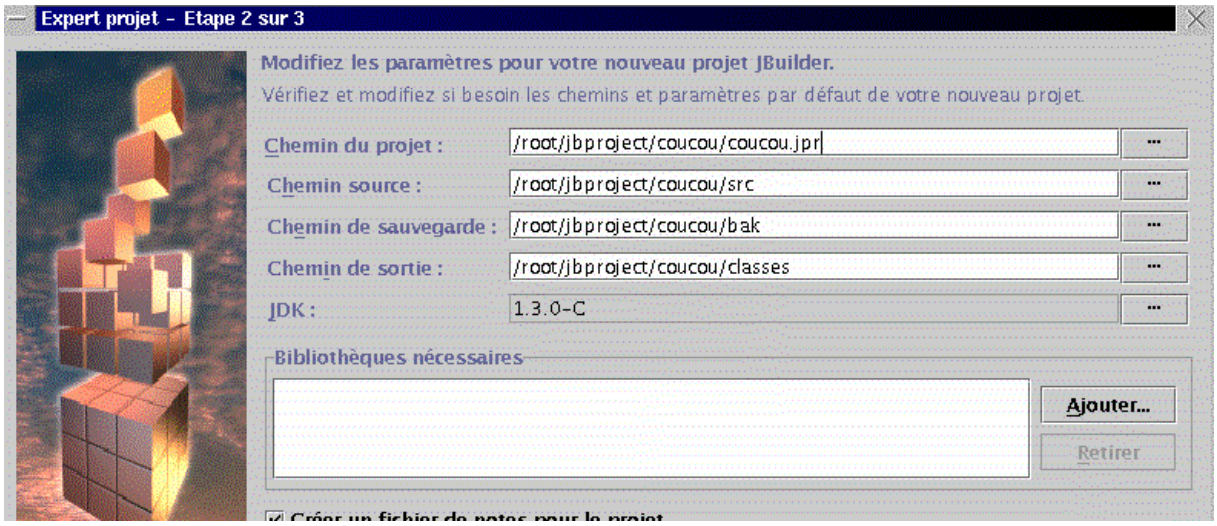
Jbuilder est livré avec un ensemble de tutoriels qui vous permettront de démarrer en programmation Java. Ci-dessous on a suivi le tutoriel "*Construction d'une application*" ci-dessus.



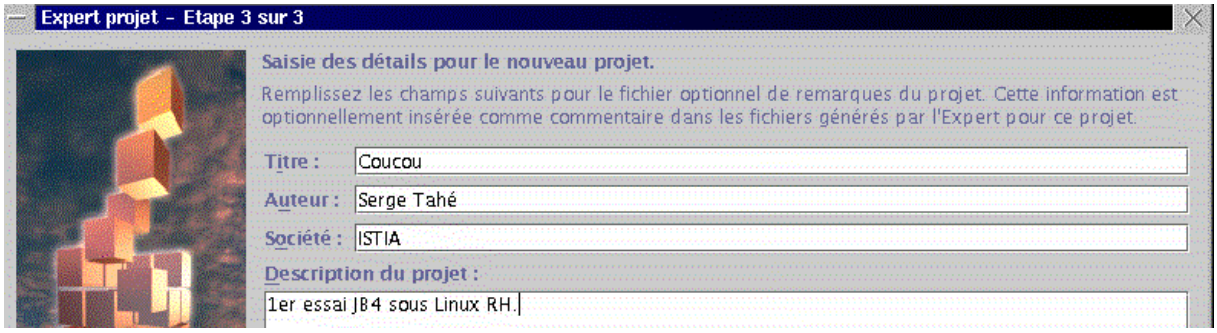
Dans Jbuilder, faites *Fichier/Nouveau projet*. Un assistant va vous présenter 3 écrans :



Nous allons créer une simple fenêtre avec le titre "Bonjour tout le monde". Nous appelons ce projet *coucou*. L'exemple a été exécuté par *root*. Jbuilder propose alors de rassembler tous les projets dans un répertoire *jbproject* du répertoire de connexion de *root*. Le projet *coucou* sera placé dans le répertoire *coucou* (Champ Nom du répertoire du projet) de *jbproject*. Faites [suivant].

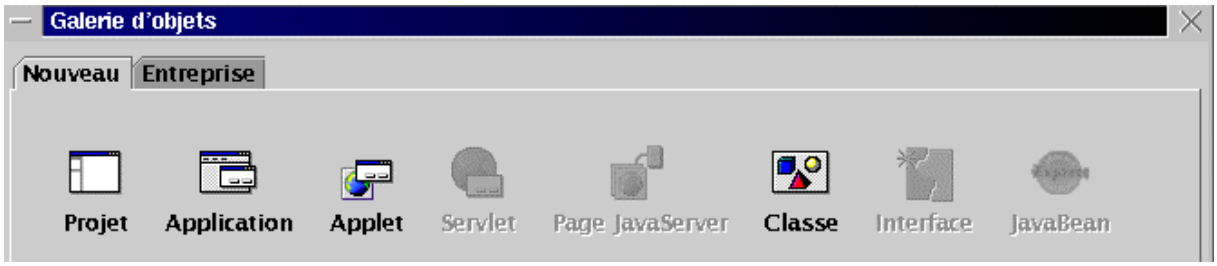


Ce second écran est un résumé des différents chemins de votre projet. Il n'y a rien à modifier. Faites [suivant].

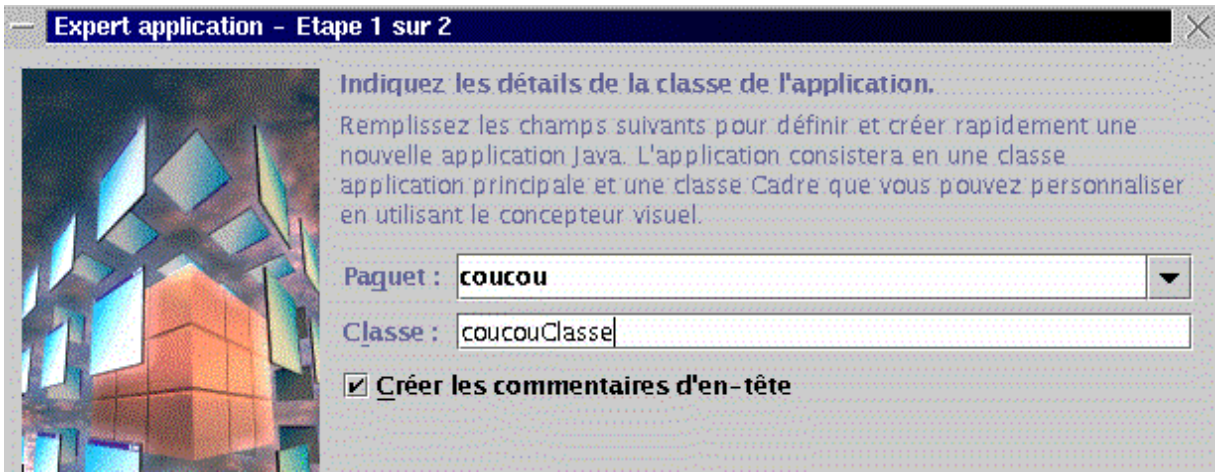


Le 3ième écran vous demande de personnaliser votre projet. Complétez et faites [Terminer].

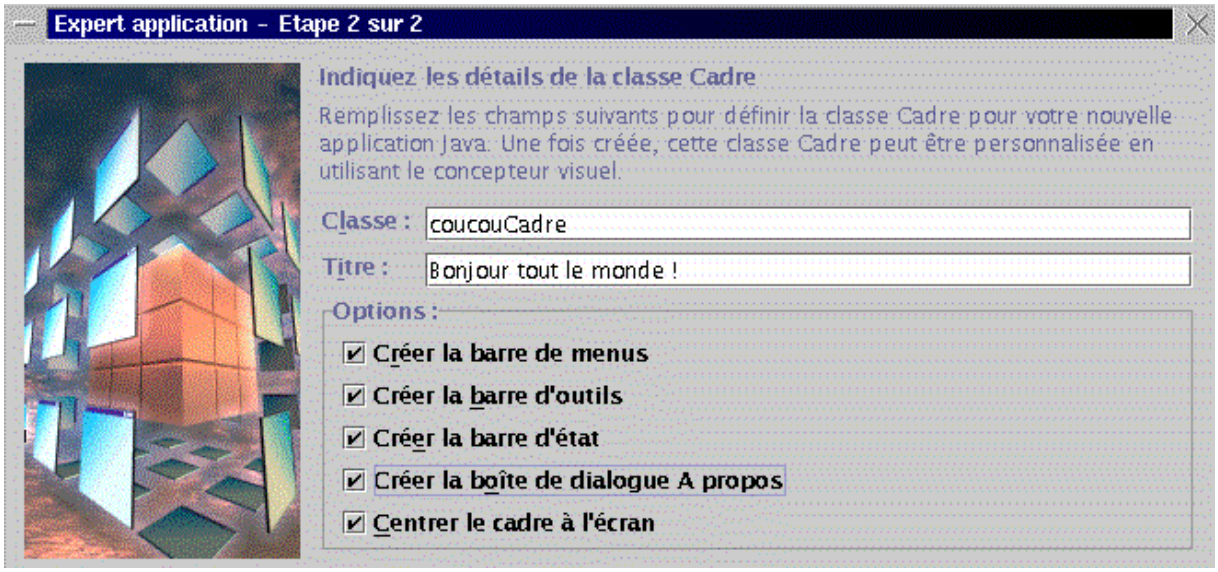
Faites maintenant *Fichier/Nouveau* :



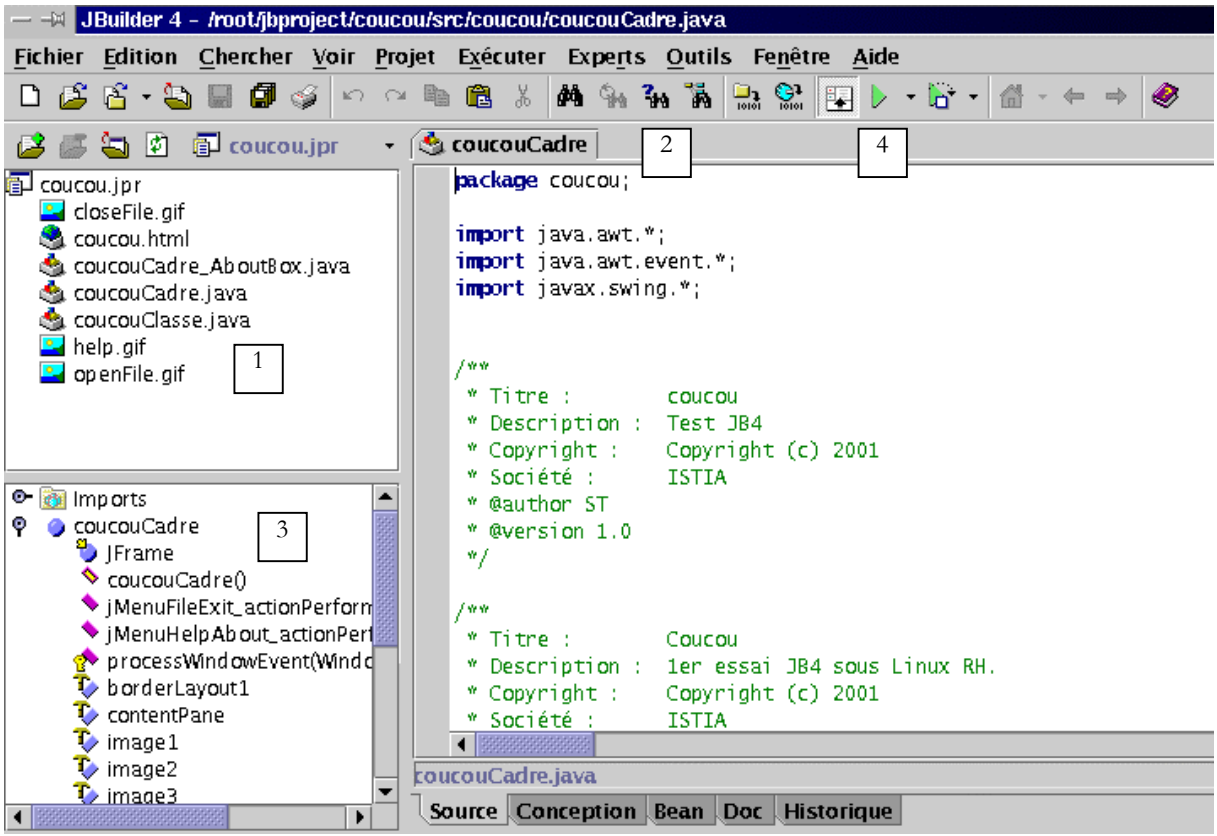
Choisissez *Application* et faites OK. Un nouvel assistant va vous présenter 2 écrans :



Le champ *paquet* reprend le nom du projet. Le champ classe demande le *nom* qui sera donné à la classe principale du projet. Prenez le nom ci-dessus et faites [suivant] :



Notre application comporte une seconde classe Java pour la fenêtre de l'application. Donnez un nom à cette classe. Le champ *Titre* est le titre que vous voulez donner à cette fenêtre. Le cadre *options* vous propose des options pour votre fenêtre. Prenez-les toutes et faites [Terminer]. Vous revenez alors à l'environnement Jbuilder qui s'est enrichi des informations que vous avez données pour votre projet :



Dans la fenêtre de gauche/haut (1), vous avez la liste des fichiers qui composent votre projet. On trouve notamment les deux classes *.java* dont nous venons de donner le nom. Dans la fenêtre de droite (2) vous avez le code Java de la classe *coucouCadre*. Dans la fenêtre de gauche/bas, vous avez la structure (classes, méthodes, attributs) de votre projet. Il est prêt à être exécuté. Appuyez sur le bouton 4 ci-dessus ou faites *Exécuter/Exécuter le projet* ou faites F9. Vous devriez obtenir la fenêtre suivante :

En cliquant sur l'option *Aide*, vous devriez retrouver des informations que vous avez données aux assistants de création. Nous n'irons pas plus loin. Il est temps maintenant de vous plonger dans les tutoriels.

Avant de terminer, montrons simplement que vous pouvez utiliser non pas Jbuilder et son interface graphique mais le jdk qu'il a amené avec lui et qu'il a placé dans **/opt/jbuilder4/jdk1.3**. Construisez le fichier *essai1.java* suivant :

```
import java.io.*;

public class essai1{
  public static void main(String args[]){
    System.out.println("coucou");
    System.exit(0);
  }
}
```

Compilons-le et exécutons-le avec le jdk de Jbuilder :

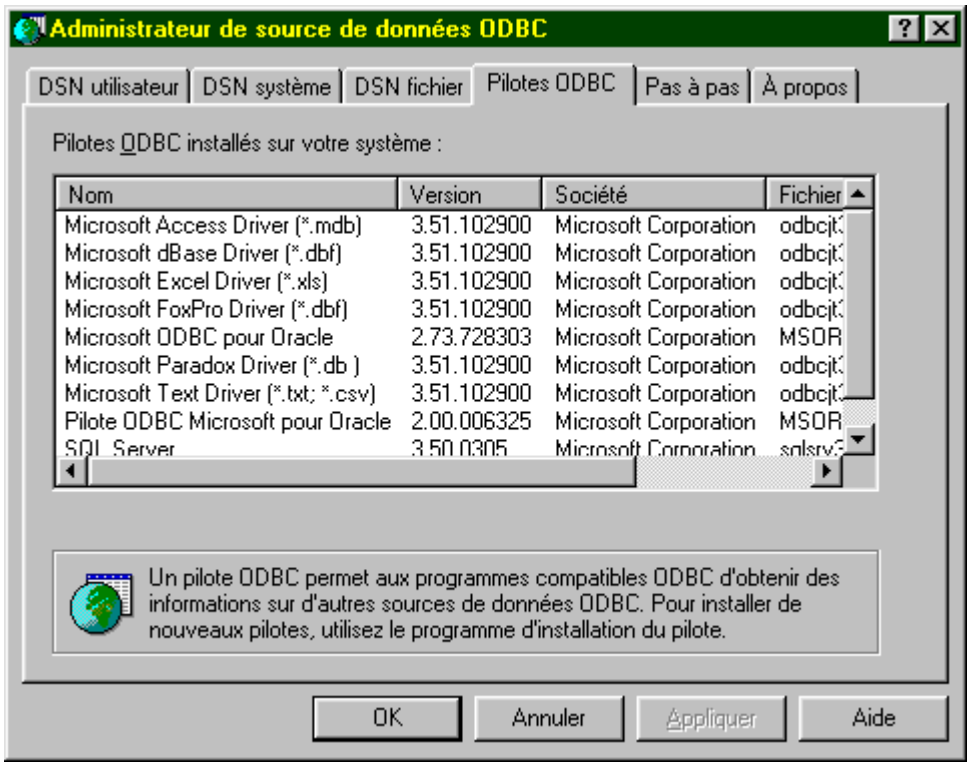
```
[ls -l *.java]
-rw-r--r--    1 root     root          135 mai  9 21:57 essai1.java
[/opt/jbuilder4/jdk1.3/bin/javac essai1.java]
```

```
[/opt/jbuilder4/jdk1.3/bin/java essai1]
coucou
```

5. Gestion des bases de données avec l'API JDBC

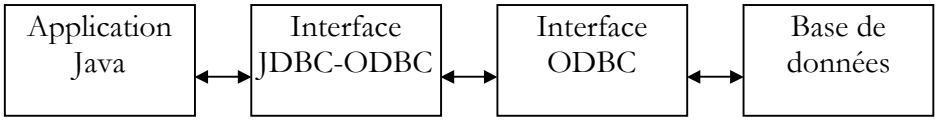
5.1 Généralités

Il existe de nombreuses bases de données sur le marché. Afin d'uniformiser les accès aux bases de données sous MS Windows, Microsoft a développé une interface appelée ODBC (Open DataBase Connectivity). Cette couche cache les particularités de chaque base de données sous une interface standard. Il existe sous MS Windows de nombreux pilotes ODBC facilitant l'accès aux bases de données. Voici par exemple, une liste de pilotes ODBC installés sur une machine Win95 :



Une application s'appuyant sur ces pilotes peut utiliser n'importe quelle bases de données ci-dessus sans ré-écriture.

Afin que les applications Java puissent tirer parti elles-aussi de l'interface ODBC, Sun a créé l'intervace JDBC (Java DataBase Connectivity) qui va s'intercaler entre l'application Java et l'interface ODBC :



5.2 Les étapes importantes dans l'exploitation des bases de données

5.2.1 Introduction

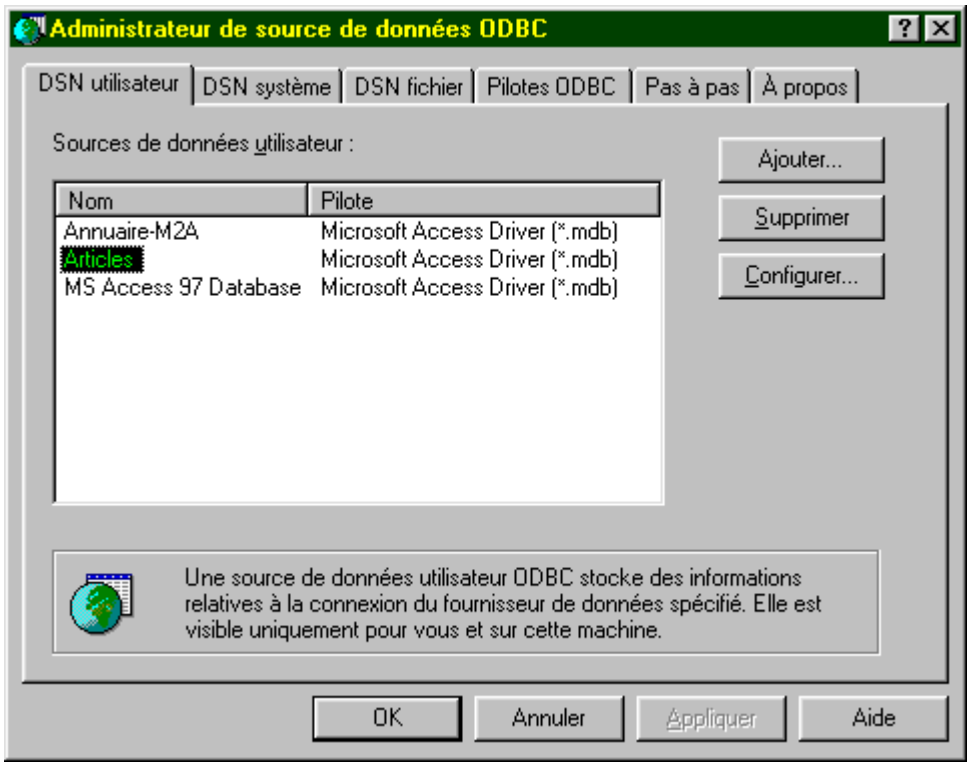
Dans une application JAVA utilisant une base de données avec l'interface JDBC, on trouvera généralement les étapes suivantes :

1. Connexion à la base de données
2. Émission de requêtes SQL vers la base
3. Réception et traitement des résultats de ces requêtes
4. Fermeture de la connexion

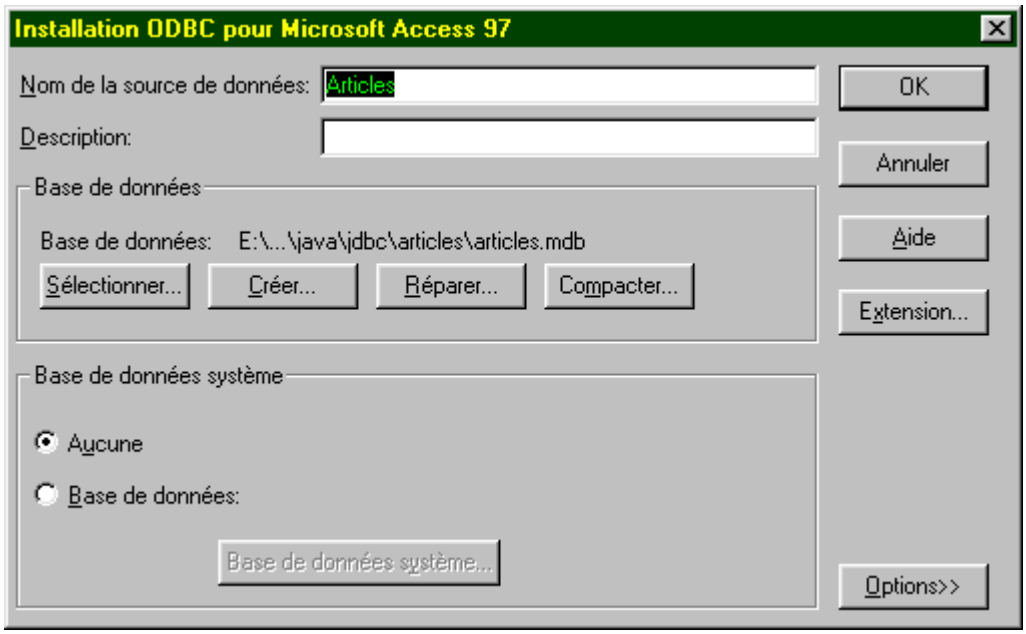
Les étapes 2 et 3 sont réalisées de façon répétée, la fermeture de connexion n'ayant lieu qu'à la fin de l'exploitation de la base. C'est un schéma relativement classique dont vous avez peut-être l'habitude si vous avez exploité une base de données de façon interactive. Nous allons détailler chacune de ces étapes sur un exemple. Nous considérons une base de données ACCESS appelée ARTICLES et ayant la structure suivante :

nom	type
code	code de l'article sur 4 caractères
nom	son nom (chaîne de caractères)
prix	son prix (réel)
stock_actu	son stock actuel (entier)
stock_mini	le stock minimum (entier) en-deça duquel il faut réapprovisionner l'article

Cette base ACCESS est définie comme source de données « utilisateur » dans le gestionnaire des bases ODBC :



Ses caractéristiques sont précisées à l'aide du bouton *Configurer* comme suit :



Cette configuration consiste essentiellement à associer à la base ARTICLES, le fichier Access *articles.mdb* correspondant à cette base. Ceci fait, la base ARTICLES est accessible aux applications utilisant l'interface ODBC.

5.2.2 L'étape de connexion

Pour exploiter une base de données, une application Java doit d'abord opérer une phase de connexion. Celle-ci se fait avec la méthode de classe suivante :

```
Connection DriverManager.getConnection(String URL, String id, String mdp)
```

avec

DriverManager	classe Java détenant la liste des pilotes disponibles pour l'application
Connection	classe Java établissant un lien entre l'application et la base grâce auquel l'application va transmettre des requêtes SQL à la base et recevoir des résultats
URL	nom identifiant la base de données. Ce nom est analogue aux URL de l'Internet. C'est pourquoi il fait partie de la classe URL. Cependant l'Internet n'intervient aucunement ici. L'URL a la forme suivante : **jdbc:nom_du_pilote:nom_de_la_source;param=val1;param2=val2** Dans nos exemples où l'on utilisera exclusivement des pilotes ODBC, le pilote s'appelle **odbc**. La troisième partie de l'URL est constituée du nom de la source avec d'éventuels paramètres. Dans nos exemples, il s'agira de sources ODBC connues du système. Ainsi l'URL de la source de données *Articles* définie précédemment sera **jdbc:odbc:Articles**
id	Identité de l'utilisateur (login)
mdp	mot de passe de l'utilisateur (password)

Pour résumer, le programme se connecte à une base de données :

- identifiée par un nom (URL)
- sous l'identité d'un utilisateur (id, mdp)

Si ces trois paramètres sont corrects, et si le pilote capable d'assurer la connexion de l'application Java à la base de données précisée existe, alors une connexion est réalisée entre l'application Java et la base de données. Celle-ci est matérialisée pour le programme par l'objet de type *Connection* rendu par la classe *DriverManager*. Comme cette connexion peut échouer pour diverses raisons, elle est susceptible de lancer une exception. On écrira donc :

```
Connection connexion=null;
URL base=...;
String id=...;
String mdp=...;
try{
  connexion=DriverManager.getConnection(base,id,mdp);
```

```
  } catch (Exception e){
    // traiter l'exception
  }
```

Pour que la connexion à une base de données soit possible, il faut disposer du pilote adéquat. Dans nos exemples, il s'agira du pilote ODBC capable de gérer la base de données demandée. S'il faut que ce pilote soit disponible dans la liste des pilotes ODBC présents sur la machine, il faut également disposer de la classe JAVA qui fera l'interface avec lui. Pour ce faire, l'application va requérir la classe qui lui est nécessaire de la façon suivante :

```
  Class.forName(String nomClasse)
```

La classe **Class** n'est en rien liée à l'interface JDBC. C'est une classe générale de gestion des classes. Sa méthode statique **forName** permet de charger dynamiquement une classe et donc de bénéficier de ses attributs et méthodes statiques. La classe faisant l'interface avec les pilotes ODBC de MS Windows s'appelle « **sun.jdbc.odbc.JdbcOdbcDriver** ». On écrira donc (la méthode peut générer une exception) :

```
try{
  Class.forName(« sun.jdbc.odbc.JdbcOdbcDriver ») ;
} catch (Exception e){
  // traiter l'exception (classe inexistante)
}
```

Les classes nécessaires à l'interface JDBC se trouvent dans le package **java.sql**. On écrira donc en début de programme :

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

Voici un programme permettant la connexion à une base de données :

```
import java.sql.*;
import java.io.*;

// appel : pg PILOTE URL UID MDP
// se connecte à la base URL grâce à la classe JDBC PILOTE
// l'utilisateur UID est identifié par un mot de passe MDP

public class connexion1{
  static String syntaxe="pg PILOTE URL UID MDP";

  public static void main(String arg[]){
    // vérification du nb d'arguments
    if(arg.length<2 || arg.length>4)
      erreur(syntaxe,1);
    // connexion à la base
    Connection connect=null;
    String uid="";
    String mdp="";
    if(arg.length>=3) uid=arg[2];
    if(arg.length==4) mdp=arg[3];
    try{
      Class.forName(arg[0]);
      connect=DriverManager.getConnection(arg[1],uid,mdp);
      System.out.println("Connexion avec la base " + arg[1] + " établie");
    } catch (Exception e){
      erreur("Erreur " + e,2);
    }
    // fermeture de la base
    try{
      connect.close();
      System.out.println("Base " + arg[1] + " fermée");
    } catch (Exception e){}
  }// main

  public static void erreur(String msg, int exitCode){
    System.err.println(msg);
    System.exit(exitCode);
  }
}// classe
```

Voici un exemple d'exécution :

```
E:\data\java\jdbc\0>java connexion1 sun.jdbc.odbc.JdbcOdbcDriver jdbc:odbc:articles
Connexion avec la base jdbc:odbc:articles établie
Base jdbc:odbc:articles fermée
```

5.2.3 Émission de requêtes vers la base de données

L'interface JDBC permet l'émission de requêtes SQL vers la base de données connectée à l'application Java ainsi que le traitement des résultats de ces requêtes. Le langage SQL (*Structured Query Language*) est un langage de requêtes standardisé pour les bases de données relationnelles. On y distingue plusieurs types de requêtes :

- les requêtes d'interrogation de la base (SELECT)
- les requêtes de mise à jour de la base (INSERT, DELETE, UPDATE)
- les requêtes de création/destruction de tables (CREATE, DELETE)

On suppose ici que le lecteur connaît les bases du langage SQL.

5.2.3.1 La classe Statement

Pour émettre une requête SQL, quelle qu'elle soit, vers une base de données, l'application Java doit disposer d'un objet de type **Statement**. Cet objet stockera, entre autres choses, le texte de la requête. Cet objet est nécessairement lié à la connexion en cours. C'est donc une méthode de la connexion établie qui permet de créer les objets *Statement* nécessaires à l'émission des requêtes SQL. Si *connexion* est l'objet symbolisant la connexion avec la base de données, un objet *Statement* est obtenu de la façon suivante :

```
Statement requete=connexion.CreateStatement();
```

Une fois obtenu un objet *Statement*, on peut émettre des requêtes SQL. Cela se fera différemment selon que la requête est une requête d'interrogation ou de mise à jour de la base.

5.2.3.2 Émettre une requête d'interrogation de la base

Une requête d'interrogation est classiquement une requête du type :

```
select col1, col2,... from table1, table2,...
where condition
order by expression
...
```

Seuls les mots clés de la première ligne sont obligatoires, les autres sont facultatifs. Il existe d'autres mots clés non présentés ici.

1. Une jointure est faite avec toutes les tables qui sont derrière le mot clé **from**
2. Seules les colonnes qui sont derrière le mot clé **select** sont conservées
3. Seules les lignes vérifiant la condition du mot clé **where** sont conservées
4. Les lignes résultantes ordonnées selon l'expression du mot clé **order by** forment le résultat de la requête.

Le résultat d'un *select* est une table. Si on considère la table ARTICLES précédente et qu'on veuille les noms des articles dont le stock actuel est au-dessous du seuil minimal, on écrira : *select nom from articles where stock_actu<stock_mini*. Si on les veut par ordre alphabétique des noms, on écrira : *select nom from articles where stock_actu<stock_mini order by nom*

Pour exécuter ce type de requête, la classe *Statement* offre la méthode **executeQuery** :

```
ResultSet executeQuery(String requête)
```

où *requête* est le texte de la requête SELECT à émettre.

Ainsi, si

- *connexion* est l'objet symbolisant la connexion avec la base de données
- **Statement s=connexion.createStatement()** crée l'objet *Statement* nécessaire pour l'émission des requêtes SQL
- **ResultSet rs=s.executeQuery(« *select nom from articles where stock_actu<stock_mini* »)** exécute une requête *select* et affecte les lignes résultats de la requête à un objet de type *ResultSet*.

5.2.3.3 La classe ResultSet : résultat d'une requête select

Un objet de type *ResultSet* représente une table, c'est à dire un ensemble de lignes et de colonnes. A un moment donné, on n'a accès qu'à une ligne de la table appelée **ligne courante**. Lors de la création initiale du *ResultSet*, la ligne courante est la ligne n° 1 si le *ResultSet* n'est pas vide. Pour passer à la ligne suivante, la classe *ResultSet* dispose de la méthode *next* :

```
  boolean next()
```

Cette méthode tente de passer à la ligne suivante du *ResultSet* et rend *true* si elle réussit, *false* sinon. En cas de réussite, la ligne suivante devient la nouvelle ligne courante. La ligne précédente est perdue et on ne pourra revenir en arrière pour la récupérer. La table du *ResultSet* a des colonnes *col1, col2,....* Pour exploiter les différents champs de la ligne courante, on dispose des méthodes suivantes :

```
Type getType("coli")
```

pour obtenir le champ « *coli* » de la ligne courante. *Type* désigne le type du champ *coli*. On utilise assez souvent la méthode *getString* sur tous les champs, ce qui permet d'obtenir le contenu du champ en tant que chaîne de caractères. On convertit ensuite si nécessaire. Si on ne connaît pas le nom de la colonne, on peut utiliser les méthodes

```
Type getType(i)
```

où i est l'indice de la colonne désirée (i>=1).

5.2.3.4 Un premier exemple

Voici un programme qui affiche le contenu de la base ARTICLES créée précédemment :

```
import java.sql.*;
import java.io.*;

// affiche le contenu d'une base système ARTICLES

public class articles1{

  static final String DB="ARTICLES";   // base de données à exploiter

  public static void main(String arg[]){

    Connection connect=null;   // connexion avec la base
    Statement S=null;      // objet d'émission des requêtes
    ResultSet RS=null;     // table résultat d'une requête
    try{
      // connexion à la base
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      connect=DriverManager.getConnection("jdbc:odbc:"+DB,"","");
      System.out.println("Connexion avec la base " + DB + " établie");
      // création d'un objet Statement
      S=connect.createStatement();
      // exécution d'une requête select
      RS=S.executeQuery("select * from " + DB);
      // exploitation de la table des résultats
      while(RS.next()){      // tant qu'il y a une ligne à exploiter
        // on l'affiche à l'écran
        System.out.println(RS.getString("code")+","+
          RS.getString("nom")+","+
          RS.getString("prix")+","+
          RS.getString("stock_actu")+","+
          RS.getString("stock_mini"));
      }// ligne suivante
    } catch (Exception e){
      erreur("Erreur " + e,2);
    }
    // fermeture de la base
    try{
      connect.close();
      System.out.println("Base " + DB + " fermée");
    } catch (Exception e){}
  }// main

  public static void erreur(String msg, int exitCode){
    System.err.println(msg);
    System.exit(exitCode);
  }
}// classe
```

Les résultats obtenus sont les suivants :

```
Connexion avec la base ARTICLES établie
a300,vélo,1202,30,2
d600,arc,5000,10,2
d800,canoé,1502,12,6
x123,fusil,3000,10,2
s345,skis nautiques,1800,3,2
f450,essai3,3,3,3
f807,cachalot,200000,0,0
z400,léopard,500000,1,1
g457,panthère,800000,1,1
Base ARTICLES fermée
```

5.2.3.5 La classe ResultSetMetadata

Dans l'exemple précédent, on connaît le nom des colonnes du *ResultSet*. Si on ne les connaît pas, on ne peut utiliser la méthode *getType(nom_colonne)*. On utilisera plutôt *getType(n° colonne)*. Cependant, pour obtenir toutes les colonnes, il nous faudrait savoir combien de colonnes a le *ResultSet* obtenu. La classe *ResultSet* ne nous donne pas cette information. C'est la classe *ResultSetMetaData* qui nous la donne. Plus généralement, cette classe nous donne des informations sur la structure de la table, c'est à dire sur la nature de ses colonnes.

On a accès aux informations sur la structure d 'un *ResultSet* en instanciant tout d'abord un objet *ResultSetMetaData*. Si RS est un *ResultSet*, le *ResultSetMetaData* associé est obtenu par :

```
RS.getMetaData()
```

On notera deux méthodes utiles dans la classe *ResultSetMetaData* :

1. **int getColumnCount()** qui donne le nombre de colonnes du *ResultSet*
2. **String getColumnLabel(int i)** qui donne le nom de la colonne i du *ResultSet* (i>=1)

5.2.3.6 Un deuxième exemple

Le programme précédent affichait le contenu de la base ARTICLES. On écrit ici, un programme qui exécute sur la base ARTICLES toute requête SQL *Select* que l'utilisateur tape au clavier.

```
import java.sql.*;
import java.io.*;

// affiche le contenu d'une base système ARTICLES

public class sql1{

  static final String DB="ARTICLES";   // base de données à exploiter

  public static void main(String arg[]){

    Connection connect=null;   // connexion avec la base
    Statement S=null;      // objet d'émission des requêtes
    ResultSet RS=null;     // table résultat d'une requête
    String select;       // texte de la requête SQL select
    int nbColonnes;      // nb de colonnes du ResultSet

    // création d'un flux d'entrée clavier
    BufferedReader in=null;
    try{
      in=new BufferedReader(new InputStreamReader(System.in));
    } catch(Exception e){
      erreur("erreur lors de l'ouverture du flux clavier ("+e+")",3);
    }
    try{
      // connexion à la base
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      connect=DriverManager.getConnection("jdbc:odbc:"+DB,"","");
      System.out.println("Connexion avec la base " + DB + " établie");
      // création d'un objet Statement
      S=connect.createStatement();
      // boucle d'exécution des requêtes SQL tapées au clavier
      System.out.print("Requête : ");
      select=in.readLine();
      while(!select.equals("fin")){
```

```
      // exécution de la requête
      RS=S.executeQuery(select);
      // nombre de colonnes
      nbColonnes=RS.getMetaData().getColumnCount();
      // exploitation de la table des résultats
      System.out.println("Résultats obtenus\n\n");
      while(RS.next()){      // tant qu'il y a une ligne à exploiter
        // on l'affiche à l'écran
        for(int i=1;i<nbColonnes;i++)
          System.out.print(RS.getString(i)+",");
        System.out.println(RS.getString(nbColonnes));
      }// ligne suivante
      // requête suivante
      System.out.print("Requête : ");
      select=in.readLine();
    }// while
   } catch (Exception e){
    erreur("Erreur " + e,2);
   }
   // fermeture de la base et du flux d'entrée
   try{
    connect.close();
    System.out.println("Base " + DB + " fermée");
    in.close();
   } catch (Exception e){}
  }// main

  public static void erreur(String msg, int exitCode){
    System.err.println(msg);
    System.exit(exitCode);
  }
}// classe
```

Voici quelques résultats obtenus :

```
Connexion avec la base ARTICLES établie
Requête : select * from articles order by prix desc
Résultats obtenus


g457,panthère,800000,1,1
z400,léopard,500000,1,1
f807,cachalot,200000,0,0
d600,arc,5000,10,2
x123,fusil,3000,10,2
s345,skis nautiques,1800,3,2
d800,canoé,1502,12,6
a300,vélo,1202,30,2
f450,essai3,3,3,3

Requête : select nom, prix from articles where prix >10000 order by prix desc
Résultats obtenus


panthère,800000
léopard,500000
cachalot,200000
```

5.2.3.7 Émettre une requête de mise à jour de la base de données

Un objet de type *Statement* permet de stocker les requêtes SQL. La méthode que cet objet utilise pour émettre des requêtes SQL de mise à jour (INSERT, UPDATE, DELETE) n'est plus la méthode *executeQuery* étudiée précédemment mais la méthode *executeUpdate* :

int executeUpdate(String requête)

La différence est dans le résultat : alors que *executeQuery* renvoyait la table des résultats (*ResultSet*), *executeUpdate* renvoie le nombre de lignes affectées par l'opération de mise à jour.

5.2.3.8 Un troisième exemple

Nous reprenons le programme précédent que nous modifions légèrement : les requêtes tapées au clavier sont maintenant des requêtes de mise à jour de la base ARTICLES.

```
import java.sql.*;
import java.io.*;

// affiche le contenu d'une base système ARTICLES

public class sql2{

  static final String DB="ARTICLES";  // base de données à exploiter

  public static void main(String arg[]){

    Connection connect=null;   // connexion avec la base
    Statement S=null;          // objet d'émission des requêtes
    ResultSet RS=null;         // table résultat d'une requête
    String sqlUpdate;          // texte de la requête SQL de mise à jour
    int nbLignes;              // nb de lignes affectées par une mise à jour

    // création d'un flux d'entrée clavier
    BufferedReader in=null;
    try{
      in=new BufferedReader(new InputStreamReader(System.in));
    } catch(Exception e){
      erreur("erreur lors de l'ouverture du flux clavier ("+e+")",3);
    }
    try{
      // connexion à la base
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      connect=DriverManager.getConnection("jdbc:odbc:"+DB,"","");
      System.out.println("Connexion avec la base " + DB + " établie");
      // création d'un objet Statement
      S=connect.createStatement();
      // boucle d'exécution des requêtes SQL tapées au clavier
      System.out.print("Requête : ");
      sqlUpdate=in.readLine();
      while(!sqlUpdate.equals("fin")){
        // exécution de la requête
        nbLignes=S.executeUpdate(sqlUpdate);
        // suivi
        System.out.println(nbLignes + " ligne(s) ont été mises à jour");
        // requête suivante
        System.out.print("Requête : ");
        sqlUpdate=in.readLine();
      }// while
    } catch (Exception e){
      erreur("Erreur " + e,2);
    }
    // fermeture de la base et du flux d'entrée
    try{
      // on libère les ressources liées à la base
      RS.close();
      S.close();
      connect.close();
      System.out.println("Base " + DB + " fermée");
      // fermeture flux clavier
      in.close();
    } catch (Exception e){}
  }// main

  public static void erreur(String msg, int exitCode){
    System.err.println(msg);
    System.exit(exitCode);
  }
}// classe
```

Voici le résultat de diverses exécutions des programmes *sql1* et *sql2* :

Liste des lignes de la base ARTICLES :

```
E:\data\java\jdbc\0>java sql1
Connexion avec la base ARTICLES établie
Requête : select nom,stock_actu from articles
Résultats obtenus


vélo,30
```

```
arc,10
canoé,12
fusil,10
skis nautiques,3
essai3,3
cachalot,0
léopard,1
panthère,1
```

On modifie certaines lignes :

```
E:\data\java\jdbc\0>java sql2
Connexion avec la base ARTICLES établie
Requête : update articles set stock_actu=stock_actu+1 where stock_actu>10
2 ligne(s) ont été mises à jour
```

Vérification :

```
E:\data\java\jdbc\0>java sql1
Connexion avec la base ARTICLES établie
Requête : select nom,stock_actu from articles
Résultats obtenus

vélo,31
arc,10
canoé,13
fusil,10
skis nautiques,3
essai3,3
cachalot,0
léopard,1
panthère,1
```

On ajoute une ligne :

```
E:\data\java\jdbc\0>java sql2
Connexion avec la base ARTICLES établie
Requête : insert into articles (code,nom,prix,stock_actu,stock_mini) values ('x400','nouveau',200,20,10)
1 ligne(s) ont été mises à jour
```

Vérification :

```
E:\data\java\jdbc\0>java sql1
Connexion avec la base ARTICLES établie
Requ_te : select nom,stock_actu from articles
Résultats obtenus

vélo,31
arc,10
canoé,13
fusil,10
skis nautiques,3
essai3,3
cachalot,0
léopard,1
panthère,1
nouveau,20
```

On détruit une ligne :

```
E:\data\java\jdbc\0>java sql2
Connexion avec la base ARTICLES établie
Requête : delete from articles where code='x400'
1 ligne(s) ont été mises à jour
Requête : fin
```

Vérification :

```
E:\data\java\jdbc\0>java sql1
Connexion avec la base ARTICLES établie
Requête : select nom,stock_actu from articles
Résultats obtenus
```

```
vélo,31
arc,10
cano_,13
fusil,10
skis nautiques,3
essai3,3
cachalot,0
léopard,1
panthère,1
```

5.2.3.9 Émettre une requête SQL quelconque

L'objet *Statement* nécessaire à l'émission de requêtes SQL dispose d'une méthode *execute* capable d'exécuter tout type de requête SQL :

```
  boolean execute(String requête)
```

Le résultat rendu est le booléen *true* si la requête a rendu un *ResultSet* (*executeQuery*), *false* si elle a rendu un nombre (*executeUpdate*). Le *ResultSet* obtenu peut être récupéré avec la méthode **getResultSet** et le nombre de lignes mises à jour, par la méthode **getUpdateCount**. Ainsi on écrira :

```
Statement S=...;
ResultSet RS=...;
int nbLignes;
String requête=...;
// exécution d'une requête SQL
if (S.execute(requête)){
  // on a un resultset
  RS=S.getResultSet();
  // exploitation du ResultSet
  ...
} else {
  // c'était une requête de mise à jour
  nbLignes=S.getUpdateCount();
  ...
}
```

5.2.3.10 Quatrième exemple

On reprend l'esprit des programmes *sql1* et *sql2* dans un programme *sql3* capable maintenant d'exécuter toute requête SQL tapée au clavier. Pour rendre le programme plus général, les caractéristiques de la base à exploiter sont passées en paramètres au programme.

```
import java.sql.*;
import java.io.*;

// appel : pg PILOTE URL UID MDP
// se connecte à la base URL grâce à la classe JDBC PILOTE
// l'utilisateur UID est identifié par un mot de passe MDP

public class sql3{

  static String syntaxe="pg PILOTE URL UID MDP";

  public static void main(String arg[]){


    // vérification du nb d'arguments
    if(arg.length<2 || arg.length>4)
      erreur(syntaxe,1);

    // init paramètres de la connexion
    Connection connect=null;
    String uid="";
    String mdp="";
    if(arg.length>=3) uid=arg[2];
    if(arg.length==4) mdp=arg[3];

    // autres données
    Statement S=null;          // objet d'émission des requêtes
    ResultSet RS=null;         // table résultat d'une requête d'interrogation
    String sqlText;            // texte de la requête SQL à exécuter
```

```
    int nbLignes;             // nb de lignes affectées par une mise à jour
    int nbColonnes;           // nb de colonnes d'un ResultSet

    // création d'un flux d'entrée clavier
    BufferedReader in=null;
    try{
      in=new BufferedReader(new InputStreamReader(System.in));
    } catch(Exception e){
      erreur("erreur lors de l'ouverture du flux clavier ("+e+")",3);
    }
    try{
      // connexion à la base
      Class.forName(arg[0]);
      connect=DriverManager.getConnection(arg[1],uid,mdp);
      System.out.println("Connexion avec la base " + arg[1] + " établie");
      // création d'un objet Statement
      S=connect.createStatement();
      // boucle d'exécution des requêtes SQL tapées au clavier
      System.out.print("Requête : ");
      sqlText=in.readLine();
      while(!sqlText.equals("fin")){
        // exécution de la requête
        try{
          if(S.execute(sqlText)){
            // on a obtenu un ResultSet - on l'exploite
            RS=S.getResultSet();
            // nombre de colonnes
            nbColonnes=RS.getMetaData().getColumnCount();
            // exploitation de la table des résultats
            System.out.println("\nRésultats obtenus\n----------------\n");
            while(RS.next()){      // tant qu'il y a une ligne à exploiter
              // on l'affiche à l'écran
              for(int i=1;i<nbColonnes;i++)
                System.out.print(RS.getString(i)+",");
              System.out.println(RS.getString(nbColonnes));
            }// ligne suivante du ResultSet
          } else {
            // c'était une requête de mise à jour
            nbLignes=S.getUpdateCount();
            // suivi
            System.out.println(nbLignes + " ligne(s) ont été mises à jour");
          }//if
        } catch (Exception e){
          System.out.println("Erreur " +e);
        }
        // requête suivante
        System.out.print("\nNouvelle Requête : ");
        sqlText=in.readLine();
      }// while
    } catch (Exception e){
      erreur("Erreur " + e,2);
    }
    // fermeture de la base et du flux d'entrée
    try{
      // on libère les ressources liées à la base
      RS.close();
      S.close();
      connect.close();
      System.out.println("Base " + arg[1] + " fermée");
      // fermeture flux clavier
      in.close();
    } catch (Exception e){}
  }// main

  public static void erreur(String msg, int exitCode){
    System.err.println(msg);
    System.exit(exitCode);
  }
}// classe
```

On établit le fichier de requêtes suivant :

```
select * from articles
update articles set stock_mini=stock_mini+5 where stock_mini<5
select nom,stock_mini from articles
insert into articles (code,nom,prix,stock_actu,stock_mini) values ('x400','nouveau',100,20,10)
select * from articles
delete from articles where code='x400'
select * from articles
fin
```

Le programme est lancé de la façon suivante :

```
E:\data\java\jdbc\0>java sql3 sun.jdbc.odbc.JdbcOdbcDriver jdbc:odbc:articles <requetes >results
```

Le programme prend donc ses entrées dans le fichier *requetes* et met ses sorties dans le fichier *results*. Les résultats obtenus sont les suivants :

```
Connexion avec la base jdbc:odbc:articles établie

Requête : (requete 1 du fichier des requetes : select * from articles)
Résultats obtenus
-----------------

a300,vélo,1202,31,3
d600,arc,5000,10,3
d800,canoé,1502,13,7
x123,fusil,3000,10,3
s345,skis nautiques,1800,3,3
f450,essai3,3,3,4
f807,cachalot,200000,0,1
z400,léopard,500000,1,2
g457,panthère,800000,1,2

Nouvelle Requête : (requete 2 du fichier des requetes : update articles set stock_mini=stock_mini+5 where
stock_mini<5)

8 ligne(s) ont été mises à jour

Nouvelle Requête : (requete 3 du fichier des requetes : select nom,stock_mini from articles)

Résultats obtenus
-----------------

vélo,8
arc,8
canoé,7
fusil,8
skis nautiques,8
essai3,9
cachalot,6
léopard,7
panthère,7

Nouvelle Requête : (requete 4 du fichier des requetes : insert into articles
(code,nom,prix,stock_actu,stock_mini) values ('x400','nouveau',100,20,10))

1 ligne(s) ont été mises à jour

Nouvelle Requête : (requete 5 du fichier des requetes : select * from articles)

Résultats obtenus
-----------------

a300,vélo,1202,31,8
d600,arc,5000,10,8
d800,canoé,1502,13,7
x123,fusil,3000,10,8
s345,skis nautiques,1800,3,8
f450,essai3,3,3,9
f807,cachalot,200000,0,6
z400,léopard,500000,1,7
g457,panthère,800000,1,7
x400,nouveau,100,20,10

Nouvelle Requête : (requete 6 du fichier des requêtes : delete from articles where code='x400')


1 ligne(s) ont été mises à jour

Nouvelle Requête : (requete 7 du fichier des requêtes : select * from articles)

Résultats obtenus
-----------------

a300,vélo,1202,31,8
d600,arc,5000,10,8
d800,canoé,1502,13,7
x123,fusil,3000,10,8
s345,skis nautiques,1800,3,8
```

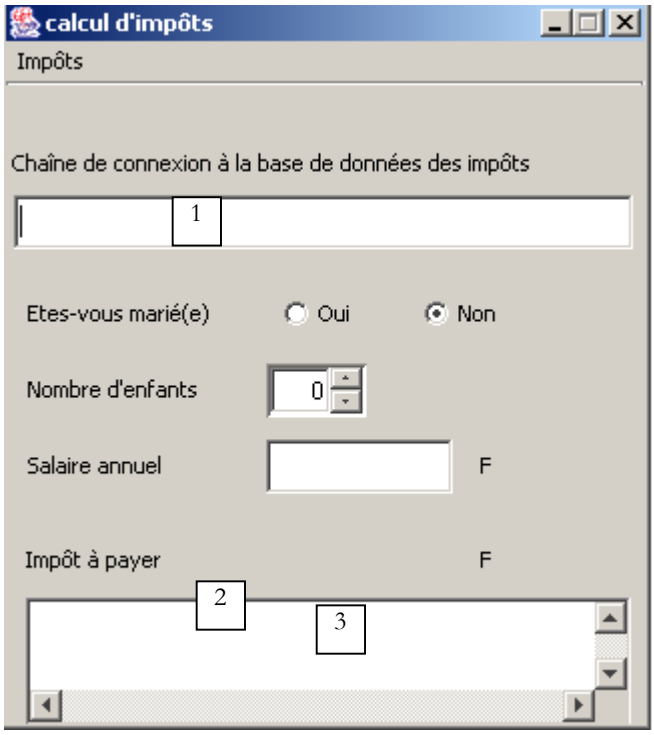
```
f450,essai3,3,3,9
f807,cachalot,200000,0,6
z400,léopard,500000,1,7
g457,panthère,800000,1,7
```

5.3 IMPOTS avec une base de données

La dernière fois que nous avons traité le problème de calcul d'impôts, c'était avec une interface graphique et les données étaient stockées dans un fichier. Nous reprenons cette version en supposant maintenant que les données sont dans une base de données ODBC-MySQL. MySQL est un SGBD du domaine public utilisable sur différentes plate-formes dont Windows et Linux. Avec ce SGBD, une base de données **dbimpots** a été créée avec dedans une unique table appelée **impots**. L'accès à la base est contrôlé par un login/motdepasse ici **admimpots/mdpimpots**. La copie d'écran montre comment utiliser la base **dbimpots** avec MySQL :

```
C:\Program Files\EasyPHP\mysql\bin>mysql -u admimpots -p
Enter password: *********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 18 to server version: 3.23.49-max-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use dbimpots;
Database changed

mysql> show tables;
+--------------------+
| Tables_in_dbimpots |
+--------------------+
| impots             |
+--------------------+
1 row in set (0.00 sec)

mysql> describe impots;
+---------+--------+------+-----+---------+-------+
| Field   | Type   | Null | Key | Default | Extra |
+---------+--------+------+-----+---------+-------+
| limites | double | YES  |     | NULL    |       |
| coeffR  | double | YES  |     | NULL    |       |
| coeffN  | double | YES  |     | NULL    |       |
+---------+--------+------+-----+---------+-------+
3 rows in set (0.02 sec)

mysql> select * from impots;
+---------+--------+---------+
| limites | coeffR | coeffN  |
+---------+--------+---------+
|   12620 |      0 |       0 |
|   13190 |   0.05 |     631 |
|   15640 |    0.1 |  1290.5 |
|   24740 |   0.15 |  2072.5 |
|   31810 |    0.2 |  3309.5 |
|   39970 |   0.25 |    4900 |
|   48360 |    0.3 |    6898 |
|   55790 |   0.35 |  9316.5 |
|   92970 |    0.4 |   12106 |
|  127860 |   0.45 |   16754 |
|  151250 |    0.5 | 23147.5 |
|  172040 |   0.55 |   30710 |
|  195000 |    0.6 |   39312 |
|       0 |   0.65 |   49062 |
+---------+--------+---------+
14 rows in set (0.00 sec)

mysql>quit
```

L'interface graphique de l'application est la suivante :



L'interface graphique a subi quelques modifications :

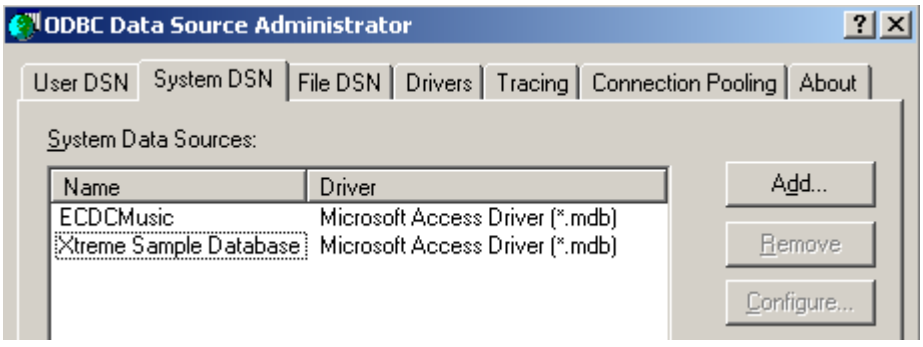
n°	type	nom	rôle
1	JTextField	txtConnexion	chaîne de connexion à la base de données ODBC
2	JScrollPane	JScrollPane1	conteneur pour le Textarea 3
3	JTextArea	txtStatus	affiche des messages d'état notamment des messages d'erreurs

La chaîne de connexion tapée dans (1) a la forme suivante : **DSN;login;motdepasse** avec

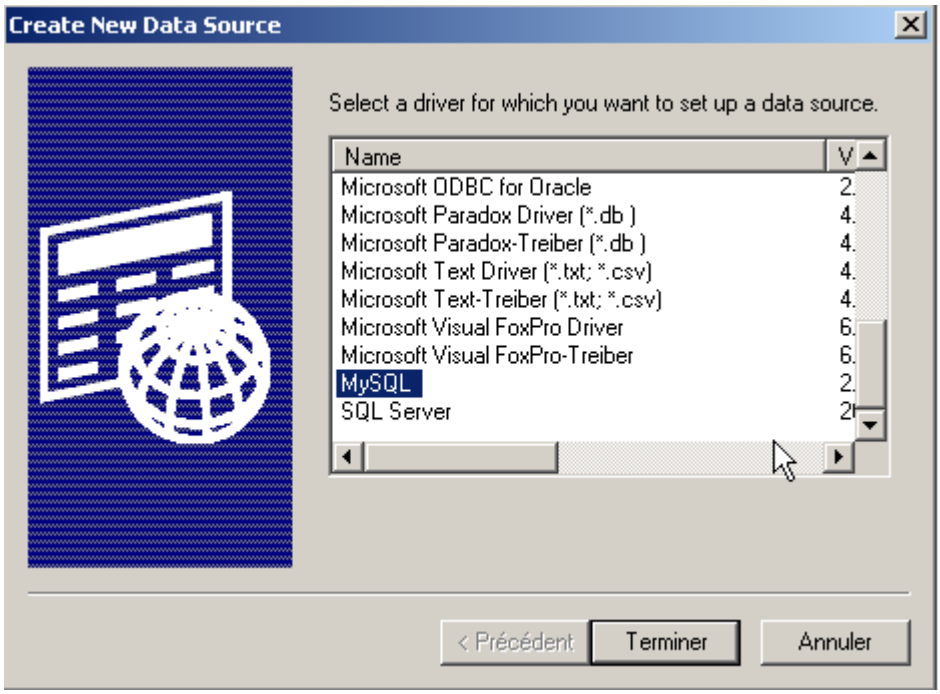
DSN	le nom DSN de la source de données ODBC
login	identité d'un utilisateur ayant un droit de lecture sur la base
motdepasse	son mot de passe

La base de données *dbimpots* a été créée à la main avec MySQL. On la transforme en source de données ODBC de la façon suivante :

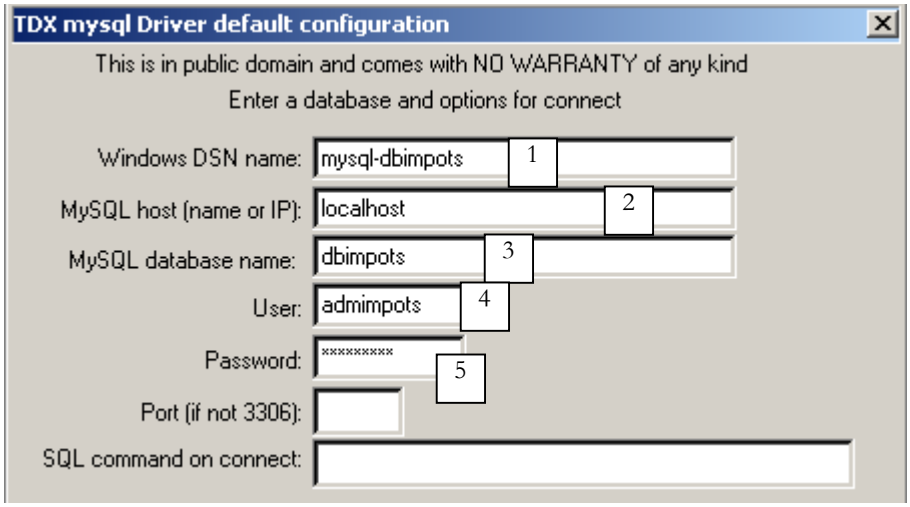
- on lance le gestionnaire des sources de données ODBC 32 bits



- on utilise le bouton [Add] pour ajouter une nouvelle source de données ODBC



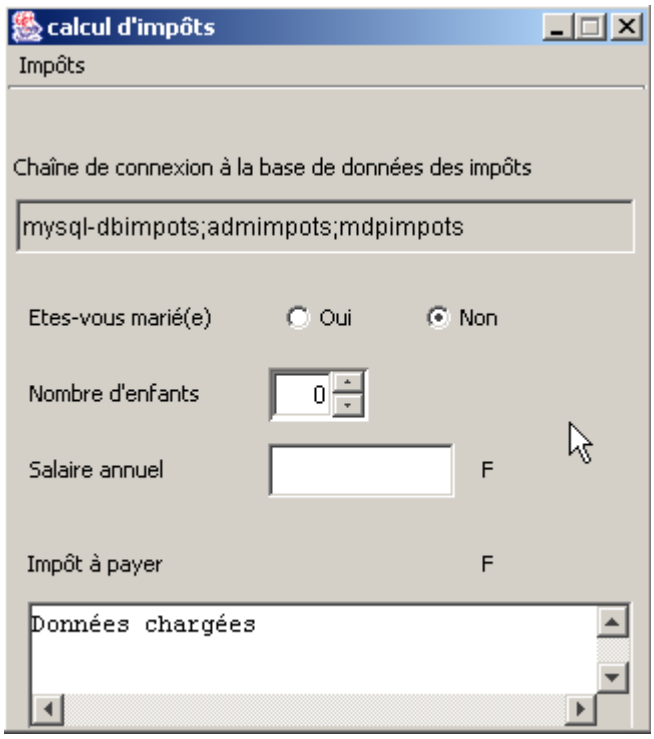
- on désigne le pilote MySQL et on fait [Terminer]

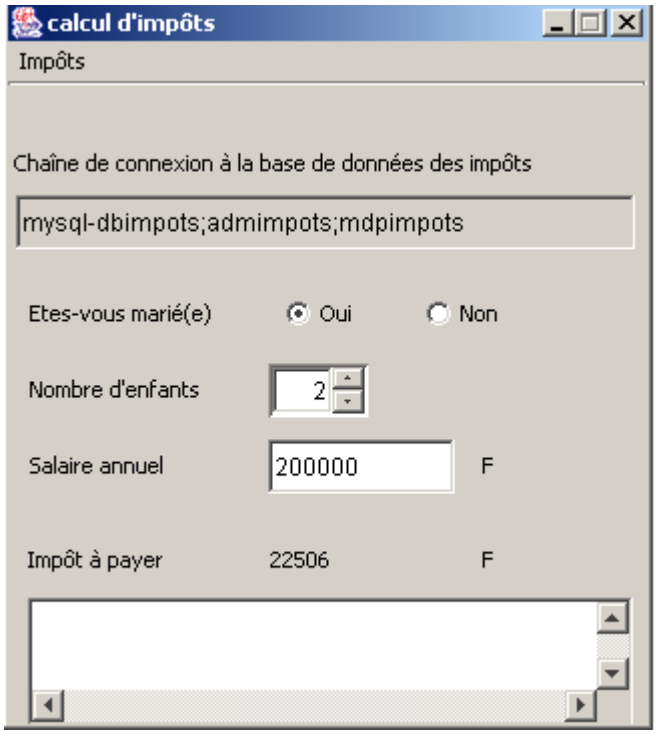


- le pilote MySQL demande un certain nombre de renseignements :

1 le nom DSN à donner à la source de données ODBC - peut-être quelconque
2 la machine sur laquelle s'exécute le SGBD MySQL - ici *localhost*. Il est intéressant de noter que la base de données pourrait être une base de données distante. Les applications locales utilisant la source de données ODBC ne s'en apercevraient pas. Ce serait le cas notamment de notre application Java.
3 la base de données MySQL à utiliser. MySQL est un SGBD qui gère des bases de données relationnelles qui sont des ensembles de tables reliées entre-elles par des relations. Ici, on donne le nom de la base gérée.
4 le nom d'un utilisateur ayant un droit d'accès à cette base
5 son mot de passe

Une fois la source de données ODBC définie, on peut tester notre programme :





Examinons le code qui, comparé à la version graphique sans base de données a été modifié. Rappelons le code de la classe *impots* utilisée jusqu'ici :

```
// création d'une classe impots

public class impots{

  // les données nécessaires au calcul de l'impôt
  // proviennent d'une source extérieure

  private double[] limites, coeffR, coeffN;

  // constructeur
  public impots(double[] LIMITES, double[] COEFFR, double[] COEFFN) throws Exception{
    // on vérifie que les 3 tableaux ont la même taille
    boolean OK=LIMITES.length==COEFFR.length && LIMITES.length==COEFFN.length;
    if (! OK) throw new Exception ("Les 3 tableaux fournis n'ont pas la même taille("+
              LIMITES.length+","+COEFFR.length+","+COEFFN.length+")");
    // c'est bon
    this.limites=LIMITES;
    this.coeffR=COEFFR;
    this.coeffN=COEFFN;
  }//constructeur

  // calcul de l'impôt
  public long calculer(boolean marié, int nbEnfants, int salaire){
    // calcul du nombre de parts
    double nbParts;
    if (marié) nbParts=(double)nbEnfants/2+2;
    else nbParts=(double)nbEnfants/2+1;
    if (nbEnfants>=3) nbParts+=0.5;
    // calcul revenu imposable & Quotient familial
    double revenu=0.72*salaire;
    double QF=revenu/nbParts;
    // calcul de l'impôt
    limites[limites.length-1]=QF+1;
    int i=0;
    while(QF>limites[i]) i++;
    // retour résultat
    return (long)(revenu*coeffR[i]-nbParts*coeffN[i]);
  }//calculer
}//classe
```

Cette classe construit les trois tableaux *limites*, *coeffR*, *coeffN* à partir de trois tableaux passés en paramètres à son constructeur. On décide de lui adjoindre un nouveau constructeur permettant de construire les trois mêmes tableaux à partir d'une base de données :

```
  public impots(String dsnIMPOTS, String userIMPOTS, String mdpIMPOTS)
    throws SQLException,ClassNotFoundException{
```

```
    // dsnIMPOTS : nom DSN de la base de données
    // userIMPOTS, mdpIMPOTS : login/mot de passe d'accès à la base
```

Pour l'exemple, nous décidons de ne pas implémenter ce nouveau constructeur dans la classe *impots* mais dans une classe dérivée *impotsJDBC* :

```
// paquetages importés
import java.sql.*;
import java.util.*;

public class impotsJDBC extends impots{
  // rajout d'un constructeur permettant de construire
  // les tableaux limites, coeffr, coeffn à partir de la table
  // impots d'une base de données
  public impotsJDBC(String dsnIMPOTS, String userIMPOTS, String mdpIMPOTS)
   throws SQLException,ClassNotFoundException{

    // dsnIMPOTS : nom DSN de la base de données
    // userIMPOTS, mdpIMPOTS : login/mot de passe d'accès à la base

    // les tableaux de données
    ArrayList aLimites=new ArrayList();
    ArrayList aCoeffR=new ArrayList();
    ArrayList aCoeffN=new ArrayList();

    // connexion à la base
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection connect=DriverManager.getConnection("jdbc:odbc:"+dsnIMPOTS,userIMPOTS,mdpIMPOTS);
    // création d'un objet Statement
    Statement S=connect.createStatement();
    // requête select
    String select="select limites, coeffr, coeffn from impots";
    // exécution de la requête
    ResultSet RS=S.executeQuery(select);
    while(RS.next()){
      // exploitation de la ligne courante
      aLimites.add(RS.getString("limites"));
      aCoeffR.add(RS.getString("coeffr"));
      aCoeffN.add(RS.getString("coeffn"));
    }// ligne suivante
    // fermeture ressources
    RS.close();
    S.close();
    connect.close();
    // transfert des données dans des tableaux bornés
    int n=aLimites.size();
    limites=new double[n];
    coeffR=new double[n];
    coeffN=new double[n];
    for(int i=0;i<n;i++){
      limites[i]=Double.parseDouble((String)aLimites.get(i));
      coeffR[i]=Double.parseDouble((String)aCoeffR.get(i));
      coeffN[i]=Double.parseDouble((String)aCoeffN.get(i));
    }//for
  }//constructeur
}//classe
```

Le constructeur lit le contenu de la table *impots* de la base qu'on lui a passé en paramètres et remplit les trois tableaux *limites*, *coeff*R, *coeff*N. Un certain nombre d'erreurs peuvent se produire. Le constructeur ne les gère pas mais les "remonte" au programme appelant :

```
  public impotsJDBC(String dsnIMPOTS, String userIMPOTS, String mdpIMPOTS)
   throws SQLException,ClassNotFoundException{
```

Si on prête attention au code précédent, on verra que la classe *impotsJDBC* utilise directement les champs *limites*, *coeff*R, *coeff*N de sa classe de base *impots*. Ceux-ci étant déclarés privés :

```
  private double[] limites, coeffR, coeffN;
```

la classe *impotsJDBC* n'a pas d'accès à ces champs directement. Nous faisons donc une première modification à la classe de base en écrivant :

```
  protected double[] limites=null;
  protected double[] coeffR=null;
  protected double[] coeffN=null;
```

L'attribut *protected* permet aux classes dérivées de la classe *impots* d'avoir un accès direct aux champs déclarés avec cet attribut. Il nous faut faire une seconde modification. Le constructeur de la classe fille *impotsJDBC* est déclarée comme suit :

```
  public impotsJDBC(String dsnIMPOTS, String userIMPOTS, String mdpIMPOTS)
    throws SQLException,ClassNotFoundException{
```

On sait qu'avant de construire un objet d'une classe fille, on doit d'abord construire un objet de la classe parent. Pour ce faire, le constructeur de la classe fille doit appeler explicitement le constructeur de la classe parent avec une instruction *super(....)*. Ici ce n'est pas fait car on ne voit pas quel constructeur de la classe parent on pourrait appeler. Il n'y en a pour l'instant qu'un et il ne convient pas. Le compilateur va alors chercher dans la classe parent un constructeur sans paramètres qu'il pourrait appeler. Il n'en trouve pas et cela génère une erreur à la compilation. Nous ajoutons donc un constructeur sans arguments à notre classe *impots* :

```
  // constructeur vide
  protected impots(){}
```

Nous le déclarons "*protected*" afin qu'il ne puisse être utilisé que par des classes filles. Le squelette de la classe *impots* est devenu maintenant le suivant :

```
public class impots{

  // les données nécessaires au calcul de l'impôt
  // proviennent d'une source extérieure

  protected double[] limites=null;
  protected double[] coeffR=null;
  protected double[] coeffN=null;

  // constructeur vide
  protected impots(){}

  // constructeur
  public impots(double[] LIMITES, double[] COEFFR, double[] COEFFN) throws Exception{
...........
  }//constructeur

  // calcul de l'impôt
  public long calculer(boolean marié, int nbEnfants, int salaire){
............
  }//calculer
}//classe
```

L'action du menu *Initialiser* de notre application devient la suivante :

```
  void mnuInitialiser_actionPerformed(ActionEvent e) {
    // on récupère la chaîne de connexion
    Pattern séparateur=Pattern.compile("\\s*;\\s*");
    String[] champs=séparateur.split(txtConnexion.getText().trim());
    // il faut trois champs
    if(champs.length!=3){
      // erreur
      txtStatus.setText("Chaîne de connexion (DSN;uid;mdp) incorrecte");
      // retour à l'interface visuelle
      txtConnexion.requestFocus();
      return;
    }//if
    // on charge les données
    try{
      // création de l'objet impotsJDBC
      objImpots=new impotsJDBC(champs[0],champs[1],champs[2]);
      // confirmation
      txtStatus.setText("Données chargées");
      // le salaire peut être modifié
      txtSalaire.setEditable(true);
      // plus de chgt possible
      mnuInitialiser.setEnabled(false);
      txtConnexion.setEditable(false);
    }catch(Exception ex){
      // problème
      txtStatus.setText("Erreur : " + ex.getMessage());
      // fin
      return;
    }//catch
  }
```

Une fois l'objet *objImpots* créé, l'application est identique à l'application graphique déjà écrite. Le lecteur est invité à s'y reporter.

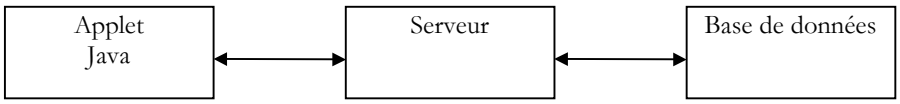
5.4 Exercices

5.4.1 Exercice 1

Fournir une interface graphique au programme *sql3* précédent.

5.4.2 Exercice 2

Une applet Java ne peut accéder à une base de données que par l'intermédiaire du serveur à partir duquel elle a été chargée. En effet, une applet n'ayant pas accès au disque de la machine sur laquelle elle est exécutéz, la base de données ne peut être sur la machine du client utilisant l'applet. On est donc dans la situation suivante :



La machine qui exécute l'applet, le serveur et la machine qui détient la base de données peuvent être trois machines différentes. On suppose ici que la base de données se trouve sur le serveur.

Problème 1

Écrire en Java l'application serveur suivante :

- l'application serveur travaille sur un port qui lui est passé en paramètre
- lorsqu'un client se connecte, l'application serveur envoie le message
  200 - Bienvenue - Envoyez votre requête
- le client envoie alors les paramètres nécessaires à la connexion à une base de données et la requête qu'il veut exécuter sous la forme :
  Pilote Java/URL base/UID/MDP/requête SQL

  Les paramètres sont séparés par la barre oblique. Les quatre premiers paramètres sont ceux du programme *sql3* décrit dans ce chapitre.
- L'application serveur crée alors une connexion avec la base de données précisée qui doit se trouver sur la même machine qu'elle et exécute la requête SQL dessus. Les résultats sont renvoyés au client sous la forme :
  100 - ligne1
  100 - ligne2
  ...
  s'il s'agit du résultat d'une requête Select ou

  101 - nbLignes

  pour rendre le nombre de lignes affectées par une requête de mise à jour. Si une erreur de connexion à la base ou d'exécution de la requête se produit, l'application renvoie

  500 - Message d'erreur
- une fois, la requête exécutée, l'application serveur ferme la connexion.

Problème 2

Créez une applet Java permettant d'interroger le serveur précédent. On pourra s'inspirer de l'interface graphique de l'exercice 1 précédent. Comme le port du serveur peut varier, celui-ci fera l'objet d'une saisie dans l'interface de l'applet. Il en sera de même pour tous les paramètres nécessaires à l'envoi de la ligne :

Pilote Java/URL base/UID/MDP/requête SQL

que le client doit envoyer au serveur.

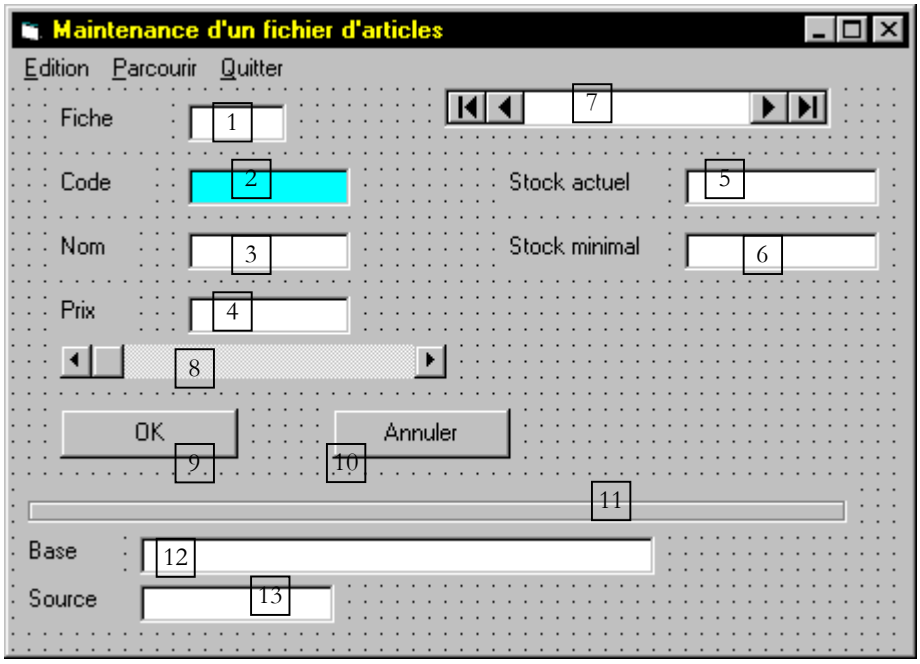
5.4.3 Exercice 3

Le texte suivant expose un problème destiné initialement à être traité en Visual Basic. Adaptez-le pour un traitement en Java dans une applet. Cette dernière s'appuiera sur le serveur de l'exercice 2. L'interface graphique pourra être modifiée pour tenir compte du nouveau contexte d'exécution.

On se propose de créer une application mettant en lumière les différentes opérations de mise à jour possibles d'une table d'une base de données ACCESS. La base de données ACCESS s'appelle **articles.mdb**. Elle a une unique table nommée **articles** qui répertorie les articles vendus par une entreprise. Sa structure est la suivante :

nom	type
code	code de l'article sur 4 caractères
nom	son nom (chaîne de caractères)
prix	son prix (réel)
stock_actu	son stock actuel (entier)
stock_mini	le stock minimum (entier) en-deça duquel il faut réapprovisioner l'article

On se propose de visualiser et de mettre à jour cette table à partir du formulaire suivant :



Les contrôles de ce formulaire sont les suivants :

n°	type	nom	Fonction
1	textbox	fiche	numéro de la fiche visualisée enabled est à false
2	textbox	code	code de l'article
3	textbox	nom	nom de l'article
4	textbox	prix	prix de l'article
5	textbox	actuel	stock actuel de l'article
6	textbox	minimum	stock minimum de l'article
7	data	data1	Contrôle data associé à la base de données databasename=chemin du fichier articles.mdb recordsource=articles connect=access
8	HScrollBar	position	permet de naviguer dans la table
9	button	OK	permet de valider une mise à jour - n'apparaît que lors de celle-ci
10	button	Annuler	permet d'annuler une mise à jour - n'apparaît que lors de celle-ci
11	frame	frame1	pour faire joli
12	textbox	basename	nom de la base ouverte

			enabled est à false
13	textbox	sourcename	nom de la table ouverte
			enabled est à false

Création de la feuille sous VB

contrôle	Particularités
data1	les champs databasename et recordsource sont renseignés. databasename doit désigner la base Access **articles.mdb** de votre répertoire et recordsource la table **articles.**
code	c'est un textbox qu'on veut lier au champ code de l'enregistrement courant de data1. Pour cela on renseigne deux champs : **datasource** : on met data1 pour indiquer que le textbox est lié à la table associée à **data1** **datafield** : on choisit le champ **code** de la table **articles** Après ces opérations, le textbox code contiendra toujours le champ code de l'enregistrement courant de data1. Inversement, modifier le contenu de ce textbox modifiera le champ code de la fiche courante. On fait de même pour les autres textbox
nom	datasource : data1 datafield : nom
prix	datasource : data1 datafield : prix
actuel	datasource : data1 datafield : stock_actu
minimum	datasource : data1 datafield : stock_mini

Les menus

La structure des menus est la suivante

Edition	**P**arcourir	**Q**uitter
Ajouter	Précédent	
Modifier	Suivant	
Supprimer	Premier	
	Dernier	

Le rôle des différentes options est le suivant :

menu	name	fonction
Ajouter	mnuajouter	pour ajouter un nouvel enregistrement à la table articles
Supprimer	mnusupprimer	pour supprimer de la table articles, l'enregistrement actuellement visualisé
Modifier	mnumodifier	pour modifier dans la table articles, l'enregistrement actuellement visualisé
Précédent	mnuprecedent	pour passer à l'enregistrement précédent
Suivant	mnusuivant	pour passer à l'enregistrement suivant
Premier	mnupremier	pour passer au premier enregistrement
Dernier	mnudernier	pour passer au dernier enregistrement
Quitter	mnuquitter	pour quitter l'application

Chargement de la feuille

Lors de l'événement *form_load*, la table associée à *data1* est ouverte (*data1.refresh*). Si l'ouverture échoue, un message d'erreur est affiché et le programme se termine (*end*). Sinon, le formulaire est affiché avec visualisé, le premier enregistrement de la table *articles*. Aucune saisie dans les champs n'est possible (propriété enabled à false). Cette saisie n'est possible qu'avec les options Ajouter et Modifier. Les boutons OK et Annuler sont cachés (visible=false).

Partie 1

On se propose de construire les procédures liées aux différentes options du menu ainsi qu'aux boutons OK et Annuler. Dans un premier temps, on ignorera les points suivants :

. l'autorisation/inhibition de certaines options du menu : par exemple, l'option **Suivant** doit être inhibée si on est positionné sur le dernier enregistrement de la table
. la gestion du scroller horizontal

menu Parcourir/Suivant

fait passer à l'enregistrement suivant (data1.RecordSet.MoveNext) si on n'est pas en fin de fichier (data1.RecordSet.EOF). Met à jour le textbox fiche (data1.recordset.absoluteposition/ data1.recordset.recordcount).

menu Parcourir/Précédent

fait passer à l'enregistrement précédent (data1.RecordSet.MovePrevious) si on n'est pas en début de fichier (data1.RecordSet.BOF). Met à jour le textbox fiche.

menu Parcourir/Premier

fait passer au premier enregistrement (data1.RecordSet.MoveFirst) si le fichier n'est pas vide (data1.recordset.recordcount=0). Met à jour le textbox fiche.

menu Parcourir/Dernier

fait passer au dernier enregistrement (data1.RecordSet.MoveLast) si le fichier n'est pas vide (data1.recordset.recordcount=0). Met à jour le textbox fiche.

menu Edition/Ajouter

- permet d'ajouter un enregistrement à la table
- met en mode Ajout d'enregistrement (data1.recordset.addnew)
- autorise les saisies sur les 5 champs code, nom, prix,... (enabled=true)
- inhibe les menus Edition, Parcourir, Quitter (enabled=false)
- affiche les boutons OK et Annuler (visible=true)

bouton OK

- valide une modification d'enregistrement (data1.recordset.Update)
- cache les boutons OK et Annuler (visible=false)
- autorise les options Edition, Parcourir, Quitter (enabled=true)
- met à jour le textbox fiche

bouton Annuler

- invalide une modification d'enregistrement (data1.recordset.CancelUpdate)
- cache les boutons OK et Annuler (visible=false)
- autorise les options Edition, Parcourir, Quitter (enabled=true)
- met à jour le textbox fiche

menu Edition/Modifier

- permet de modifier l'enregistrement visualisé sur le formulaire
- met en mode Edition d'enregistrement (data1.recordset.edit)
- autorise les saisies sur les 4 champs nom, prix,... (enabled=true) mais pas sur le champ code (enabled=false)
- inhibe les menus Edition, Parcourir, Quitter (enabled=false)
- affiche les boutons OK et Annuler (visible=true)

menu Edition/Supprimer

- permet de supprimer (data1.recordset.delete) l'enregistrement visualisé de la table
- passe à la fiche suivante (data1.recordset.movenext)

menu Quitter

- décharge la feuille (unload me)

événement form_unload (cancel as integer)

- activé par l'opération **unload me** ou la fermeture du formulaire par Alt-F4 ou un double-clic sur la case système, donc pas forcément par l'option **quitter**.
- pose la question « Voulez-vous vraiment quitter l'application » avec deux boutons Oui/Non (msgbox avec style=vbyes+vbno)
- si la réponse est Non (=vbno) on met cancel à -1 et on quitte la procédure form_unload. cancel à -1 indique que la fermeture de la fenêtre est refusée.
- si la réponse est oui (=vbyes), la base est fermée (data1.recordset.close, data1.database.close).

Partie 2 - Gestion des menus

Ici, on s'intéresse à l'autorisation/inhibition des menus. Après chaque opération changeant la fiche courante on appellera une procédure qu'on pourra appeler **Oueston**. Celle-ci vérifiera les conditions suivantes :

. si le fichier est vide, on
  inhibera les menus Parcourir, Edition/Supprimer,
  autorisera les autres
. si la fiche courante est la première fiche, on
  inhibera Parcourir/Précédent
  autorisera le reste
. si la fiche courante est la dernière, on
  inhibera Parcourir/Suivant
  autorisera le reste

Partie 3 - Gestion du variateur horizontal

Un variateur horizontal a trois champs importants :

min : sa valeur minimale
max : sa valeur maximale
value : sa valeur actuelle

Initialisation du variateur

Au chargement (form_load), le variateur sera initialisé ainsi :

min=0
max=data1.recordset.recordcount-1
value=1

On notera que juste après l'ouverture de la base (data1.refresh), le nombre d'enregistrements de la table représenté par data1.recordset.recordcount est faux. Il faut aller en fin de table (MoveLast), puis revenir en début de table (MoveFirst) pour qu'il soit correct.

Action directe sur le variateur

La position du curseur du variateur représente la position dans la table.

Lorsque l'utilisateur modifie le variateur (nommé position ici), l'événement position_change est déclenché. Dans cet événement, on changera l'enregistrement courant de la table pour que celui-ci reflète le déplacement opéré sur le variateur. Pour cela, on utilisera le champ **absoluteposition** de data1.recordset. Lorsqu'on affecte la valeur **i** à ce champ, la fiche n° i de la table devient la fiche courante. Les fiches sont numérotées à partir de 0 et ont donc un n° dans l'intervalle [0,data1.recordset.recordcount-1]. Dans la procédure position_change, il nous suffit d'écrire

data1.recordset.absoluteposition=position.value

pour que la fiche courante visualisée sur le formulaire reflète le déplacement opéré sur le variateur.

Ceci fait, on appellera ensuite, la procédure **Oueston** pour mettre à jour les menus.

Mise à jour du variateur

Puisque la position du curseur du variateur doit refléter la position dans la table, il faut mettre à jour la valeur du variateur à chaque fois qu'il y a un changement de fiche courante dans la table, produit par l'un ou l'autre des menus. Comme chacun de ceux-ci appelle la procédure Oueston, le mieux est de placer cette mise à jour également dans cette procédure. Il suffit d'écrire ici :

position.value=data1.recordset.absoluteposition

Partie 4 - Option Chercher

On ajoute l'option **Parcourir/Chercher** qui a pour but de permettre à l'utilisateur de visualiser un article dont il donne le code. Lorsque cette option est activée, se passent les séquences suivantes :

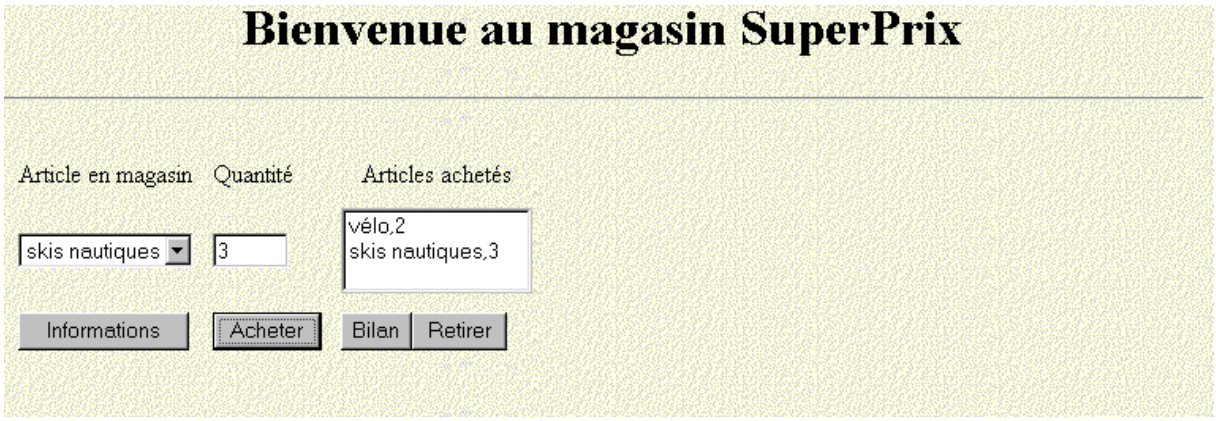
- on se met en Ajout de fiche (Addnew), ceci dans le seul but de ne pas modifier la fiche courante sur laquelle on était lorsque l'option a été activée,
- on autorise la saisie dans le champ code et on efface le contenu du champ fiche,
- on inhibe les menus et affiche les boutons **OK** et **Annuler**
- lorsque l'utilisateur appuie sur OK, il nous faut chercher la fiche correspondant au code tapé par l'utilisateur. Or la procédure OK_click sert déjà aux options **Parcourir/Ajouter** et **Parcourir/Modifier**. Pour distinguer entre ces cas, on est amenés à gérer une variable globale qu'on appellera ici **état** qui aura trois valeurs possibles : « ajouter », « modifier » et « chercher ». Les procédures liées aux boutons OK et Annuler utiliseront cette variable pour savoir dans quel contexte elles sont appelées.
- si **état** vaut « chercher », dans la procédure liée à OK, on
  - annule l'opération addnew (data1.recordset.cancelupdate) car on n'avait pas l'intention d'ajouter une fiche. Il faut noter, qu'alors la fiche courante redevient celle qui était présente à l'écran avant l'opération Parcourir/Chercher.
  - construit le critère de recherche lié au code et on lance la recherche (data1.recordset.findfirst critère),
  - si la recherche échoue (data1.recordset.nomatch=true), on le signale à l'utilisateur, puis on se remet en mode ajout (addnew) et on quitte la procédure OK. L'utilisateur devra retaper un nouveau code ou prendre l'option Annuler.
  - si la recherche aboutit, la fiche trouvée devient la nouvelle fiche courante. On remet les menus, cache les boutons OK/Annuler, inhibe la saisie dans le champ code et on quitte la procédure.
- si **état** vaut « chercher », dans la procédure liée à Annuler, on
  - annule l'opération addnew (data1.recordset.cancelupdate). On reviendra alors automatiquement sur la fiche courante d'avant l'opération Parcourir/Chercher.
  - remet les menus, cache les boutons OK/Annuler, inhibe la saisie dans le champ code et on quitte la procédure.

Un article doit être repéré de façon unique par son code. Faites en sorte que dans l'option Parcourir/Ajouter, l'ajout soit refusé si l'enregistrement à ajouter a un code article qui existe déjà dans la table.

5.4.4 Exercice 4

On présente ici une application Web s'appuyant sur le serveur de l'exercice 2. C'est un embryon d'application de commerce électronique.

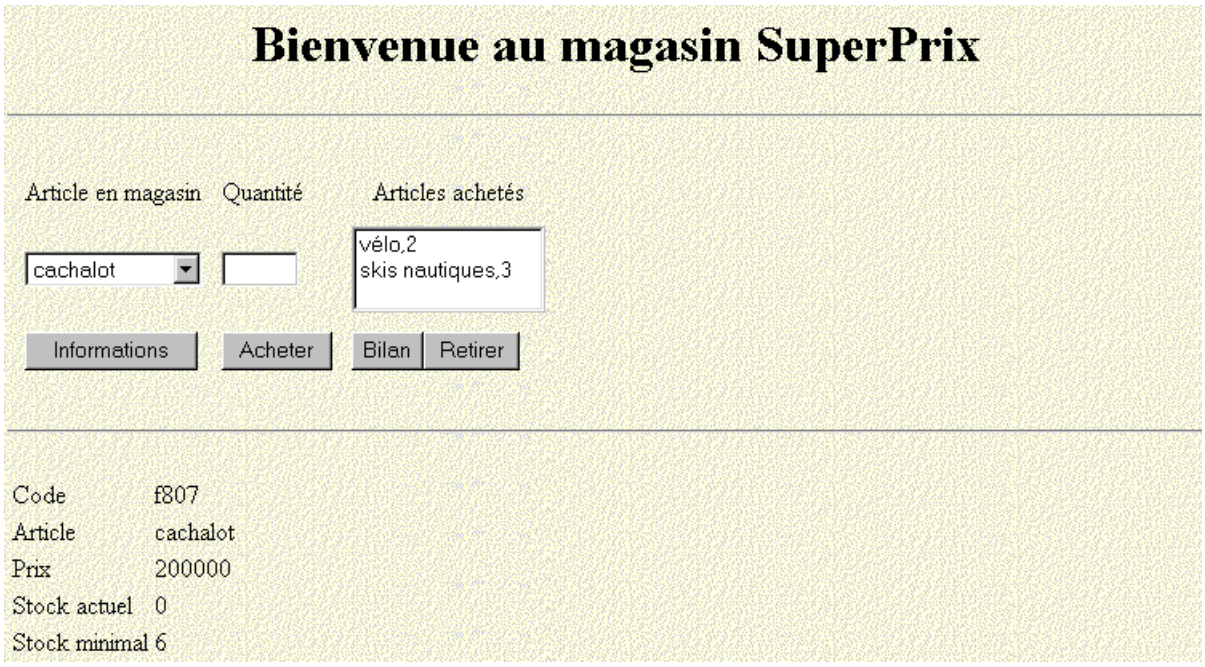
Le client commande des articles grâce à l'interface Web suivante :



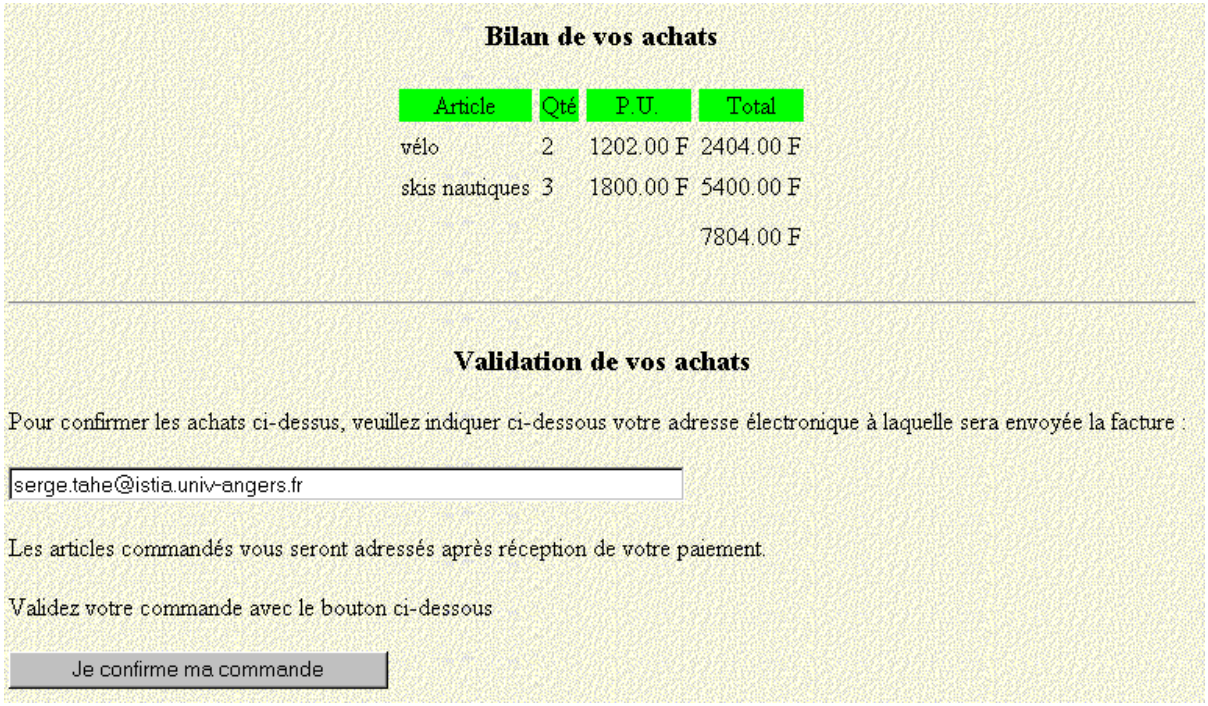
Il peut faire les opérations suivantes :

- il sélectionne un article dans la liste déroulante
- il précise la quantité désirée
- il valide son achat avec le bouton *Acheter*
- son achat est affiché dans la liste des articles achetés
- il peut retirer des articles de cette liste, en sélectionnant un article et en usant du bouton *Retirer*
- lorsqu'il actionne le bouton *Bilan*, il obtient le bilan suivant :

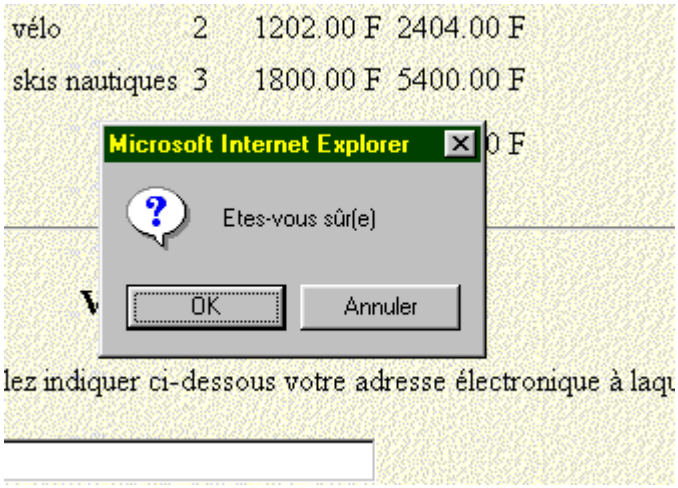
Le bilan permet à l'utilisateur de connaître le détail de sa facture. L'utilisateur a accès à des détails concernant un article sélectionné dans la liste déroulante avec le bouton *Informations* :



Lorsque l'utilisateur a demandé le bilan de sa facture, il peut la valider avec la page suivante. Pour cela, il doit donner son adresse électronique et confirmer sa commande avec le bouton adéquat.



Avant d'enregistrer sa commande, l'application demande confirmation :



Une fois la commande confirmée, l'application la traite et envoie une page de confirmation :

En réalité, l'application ne traite pas la commande. Elle se contente d'envoyer un courrier électronique à l'utilisateur lui demandant de régler le montant des achats :

```
Cher client,

Vous trouverez ci-dessous le détail de votre commande au magasin SuperPrix. Elle vous sera livrée après réception de votre chèque établi à l'ordre de SuperPrix et à envoyer à l'adresse suivante :

SuperPrix
ISTIA
62 av Notre-Dame du Lac
49000 Angers
France

Nous vous remercions vivement de votre commande

---------------------------------------
Votre commande
---------------------------------------
article, quantité, prix unitaire, total
=======================================
vélo, 2, 1202.00 F, 2404.00 F
skis nautiques, 3, 1800.00 F, 5400.00 F
Total à payer : 7804 F
```

Question : Créer l'équivalent de cette application Web avec une applet Java.

6. Les Threads d'exécution

6.1 Introduction

Lorsqu'on lance une application, elle s'exécute dans un flux d'exécution appelé un **thread**. La classe modélisant un *thread* est la classe *java.lang.Thread* dont voici quelques propriétés et méthodes :

`currentThread()`	donne le thread actuellement en cours d'exécution
`setName()`	fixe le nom du thread
`getName()`	nom du thread
`isAlive()`	indique si le thread est actif(true) ou non (false)
`start()`	lance l'exécution d'un thread
`run()`	méthode exécutée automatiquement après que la méthode start précédente ait été exécutée
`sleep(n)`	arrête l'exécution d'un thread pendant n millisecondes
`join()`	opération bloquante - attend la fin du thread pour passer à l'instruction suivante

Les constructeurs les plus couramment utilisés sont les suivants :

`Thread()`	crée une référence sur une tâche asynchrone. Celle-ci est encore inactive. La tâche créée doit posséder la méthode **run** : ce sera le plus souvent une classe dérivée de *Thread* qui sera utilisée.
`Thread(Runnable object)`	idem mais c'est l'objet *Runnable* passé en paramètre qui implémente la méthode *run*.

Regardons une première application mettant en évidence l'existence d'un thread principal d'exécution, celui dans lequel s'exécute la fonction *main* d'une classe :

```
// utilisation de threads

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

public class thread1{
  public static void main(String[] arg)throws Exception {
    // init thread courant
    Thread main=Thread.currentThread();
    // affichage
    System.out.println("Thread courant : " + main.getName());
    // on change le nom
    main.setName("myMainThread");
    // vérification
    System.out.println("Thread courant : " + main.getName());

    // boucle infinie
    while(true){
        // on récupère l'heure
       Calendar calendrier=Calendar.getInstance();
       String H=calendrier.get(Calendar.HOUR_OF_DAY)+":"
       +calendrier.get(Calendar.MINUTE)+":"
       +calendrier.get(Calendar.SECOND);
      // affichage
      System.out.println(main.getName() + " : " +H);
      // arrêt temporaire
      Thread.sleep(1000);
    }//while
  }//main
}//classe
```

Les résultats écran :

```
Thread courant : main
Thread courant : myMainThread
myMainThread : 15:34:9
myMainThread : 15:34:10
myMainThread : 15:34:11
myMainThread : 15:34:12
Terminer le programme de commandes (O/N) ? o
```

L'exemple précédent illustre les points suivants :

- la fonction m*ain* s'exécute bien dans un thread
- on a accès aux caractéristiques de ce thread par *Thread.currentThread()*
- le rôle de la méthode s*leep*. Ici le thread exécutant m*ain* se met en sommeil régulièrement pendant 1 seconde entre deux affichages.

6.2 Création de threads d'exécution

Il est possible d'avoir des applications où des morceaux de code s'exécutent de façon "simultanée" dans différents threads d'exécution. Lorsqu'on dit que des *thread*s s'exécutent de façon simultanée, on commet souvent un abus de langage. Si la machine n'a qu'un processeur comme c'est encore souvent le cas, les *thread*s se partagent ce processeur : ils en disposent, chacun leur tour, pendant un court instant (quelques millisecondes). C'est ce qui donne l'illusion du parallélisme d'exécution. La portion de temps accordée à un *thread* dépend de divers facteurs dont sa priorité qui a une valeur par défaut mais qui peut être fixée également par programmation. Lorsqu'un *thread* dispose du processeur, il l'utilise normalement pendant tout le temps qui lui a été accordé. Cependant, il peut le libérer avant terme :

- en se mettant en attente d'un événement (*wait, join*)
- en se mettant en sommeil pendant un temps déterminé (*sleep*)

1. Un **thread T** peut être créé de diverses façons

   - en dérivant la classe *Thread* et en redéfinissant la méthode *run* de celle-ci.
   - en implémentant l'interface *Runnable* dans une classe et en utilisant le constructeur *new Thread(Runnable)*. *Runnable* est une interface qui ne définit qu'une seule méthode : *public void run()*. L'argument du constructeur précédent est donc toute instance de classe implémentant cette méthode *run*.

   Dans l'exemple qui suit, les threads sont construits à l'aide d'une classe anonyme dérivant la classe Thread :

   ```
   // on crée le thread i
   tâches[i]=new Thread() {
     public void run() {
       affiche();
      }
    };//déf tâches[i]
   ```

   La méthode *run* se contente ici de renvoyer sur une méthode *affiche*.

2. L'exécution du thread T est lancé par **T.start()** : cette méthode appartient à la classe Thread et opère un certain nombre d'initialisations puis lance automatiquement la méthode *run* du Thread ou de l'interface *Runnable*. Le programme qui exécute l'instruction *T.start()* n'attend pas la fin de la tâche T : il passe aussitôt à l'instruction qui suit. On a alors deux tâches qui s'exécutent en parallèle. Elles doivent souvent pouvoir communiquer entre elles pour savoir où en est le travail commun à réaliser. C'est le problème de synchronisation des threads.

3. Une fois lancé, le thread s'exécute de façon autonome. Il s'arrêtera lorsque la fonction *run* qu'il exécute aura fini son travail.

4. On peut attendre la fin de l'exécution du Thread T par **T.join()**. On a là une instruction bloquante : le programme qui l'exécute est bloqué jusqu'à ce que la tâche T ait terminé son travail. C'est également un moyen de synchronisation.

Examinons le programme suivant :

```
// utilisation de threads

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

public class thread2{
  public static void main(String[] arg) {
    // init thread courant
    Thread main=Thread.currentThread();
    // on donne un nom au thread courant
    main.setName("myMainThread");
    // début de main
    System.out.println("début du thread " +main.getName());

    // création de threads d'exécution
    Thread[] tâches=new Thread[5];
    for(int i=0;i<tâches.length;i++){
      // on crée le thread i
```

```
      tâches[i]=new Thread() {
        public void run() {
          affiche();
        }
       };//déf tâches[i]
      // on fixe le nom du thread
      tâches[i].setName(""+i);
      // on lance l'exécution du thread i
      tâches[i].start();
    }//for

    // fin de main
    System.out.println("fin du thread " +main.getName());
  }//Main

  public static void affiche() {
    // on récupère l'heure
    Calendar calendrier=Calendar.getInstance();
    String H=calendrier.get(Calendar.HOUR_OF_DAY)+":"
      +calendrier.get(Calendar.MINUTE)+":"
      +calendrier.get(Calendar.SECOND);
    // affichage début d'exécution
    System.out.println("Début d'exécution de la méthode affiche dans le Thread " +
      Thread.currentThread().getName()+ " : " + H);
    // mise en sommeil pendant 1 s
    try{
      Thread.sleep(1000);
    }catch (Exception ex){}
    // on récupère l'heure
    calendrier=Calendar.getInstance();
    H=calendrier.get(Calendar.HOUR_OF_DAY)+":"
      +calendrier.get(Calendar.MINUTE)+":"
      +calendrier.get(Calendar.SECOND);
    // affichage fin d'exécution
    System.out.println("Fin d'exécution de la méthode affiche dans le Thread "
    +Thread.currentThread().getName()+ " : " + H);
  }// affiche
}//classe
```

Le thread principal, celui qui exécute la fonction m*ain*, crée 5 autres threads chargés d'exécuter la méthode statique *affiche*. Les résultats sont les suivants :

```
début du thread myMainThread
Début d'exécution de la méthode affiche dans le Thread 0 : 15:48:3
fin du thread myMainThread
Début d'exécution de la méthode affiche dans le Thread 1 : 15:48:3
Début d'exécution de la méthode affiche dans le Thread 2 : 15:48:3
Début d'exécution de la méthode affiche dans le Thread 3 : 15:48:3
Début d'exécution de la méthode affiche dans le Thread 4 : 15:48:3
Fin d'exécution de la méthode affiche dans le Thread 0 : 15:48:4
Fin d'exécution de la méthode affiche dans le Thread 1 : 15:48:4
Fin d'exécution de la méthode affiche dans le Thread 2 : 15:48:4
Fin d'exécution de la méthode affiche dans le Thread 3 : 15:48:4
Fin d'exécution de la méthode affiche dans le Thread 4 : 15:48:4
```

Ces résultats sont très instructifs :

- on voit tout d'abord que le lancement de l'exécution d'un thread n'est pas bloquante. La méthode m*ain* a lancé l'exécution de 5 threads en parallèle et a terminé son exécution avant eux. L'opération

  ```
  // on lance l'exécution du thread i
  tâches[i].start();
  ```

  lance l'exécution du thread *tâches[i]* mais ceci fait, l'exécution se poursuit immédiatement avec l'instruction qui suit sans attendre la fin d'exécution du thread.
- tous les threads créés doivent exécuter la méthode *affiche*. L'ordre d'exécution est imprévisible. Même si dans l'exemple, l'ordre d'exécution semble suivre l'ordre de lancement des threads, on ne peut en conclure de généralités. Le système d'exploitation a ici 6 threads et un processeur. Il va distribuer le processeur à ces 6 threads selon des règles qui lui sont propres.
- on voit dans les résultats une conséquence de la méthode s*leep*. Dans l'exemple, c'est le thread 0 qui exécute le premier la méthode *affiche*. Le message de début d'exécution est affiché puis il exécute la méthode s*leep* qui le suspend pendant 1 seconde. Il perd alors le processeur qui devient ainsi disponible pour un autre thread. L'exemple montre que c'est le thread 1 qui va l'obtenir. Le thread 1 va suivre le même parcours ainsi que les autres threads. Lorsque la seconde de sommeil du thread 0 va être terminée, son exécution peut reprendre. Le système lui donne le processeur et il peut terminer l'exécution de la méthode *affiche*.

Modifions notre programme pour le terminer la méthode *main* par les instructions :

```
    // fin de main
    System.out.println("fin du thread " +main.getName());
     // arrêt de l'application
     System.exit(0);
```

L'exécution du nouveau programme donne :

```
début du thread myMainThread
Début d'exécution de la méthode affiche dans le Thread 0 : 16:5:45
Début d'exécution de la méthode affiche dans le Thread 1 : 16:5:45
Début d'exécution de la méthode affiche dans le Thread 2 : 16:5:45
Début d'exécution de la méthode affiche dans le Thread 3 : 16:5:45
fin du thread myMainThread
Début d'exécution de la méthode affiche dans le Thread 4 : 16:5:45
```

Dès que la méthode *main* exécute l'instruction :

```
    System.exit(0);
```

elle arrête tous les threads de l'application et non simplement le thread *main*. La méthode *main* pourrait vouloir attendre la fin d'exécution des threads qu'elle a créés avant de se terminer elle-même. Cela peut se faire avec la méthode j*oin* de la classe *Thread* :

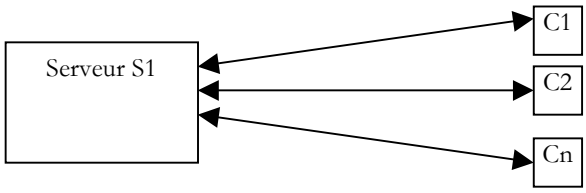
```
    // attente de tous les threads
    for(int i=0;i<tâches.length;i++){
      // on attend le thread i
      tâches[i].join();
    }//for

    // fin de main
    System.out.println("fin du thread " +main.getName());
     // arrêt de l'application
     System.exit(0);
```

On obtient alors les résultats suivants :

```
début du thread myMainThread
Début d'exécution de la méthode affiche dans le Thread 0 : 16:11:9
Début d'exécution de la méthode affiche dans le Thread 1 : 16:11:9
Début d'exécution de la méthode affiche dans le Thread 2 : 16:11:9
Début d'exécution de la méthode affiche dans le Thread 3 : 16:11:9
Début d'exécution de la méthode affiche dans le Thread 4 : 16:11:9
Fin d'exécution de la méthode affiche dans le Thread 0 : 16:11:10
Fin d'exécution de la méthode affiche dans le Thread 1 : 16:11:10
Fin d'exécution de la méthode affiche dans le Thread 2 : 16:11:10
Fin d'exécution de la méthode affiche dans le Thread 3 : 16:11:10
Fin d'exécution de la méthode affiche dans le Thread 4 : 16:11:10
fin du thread myMainThread
```

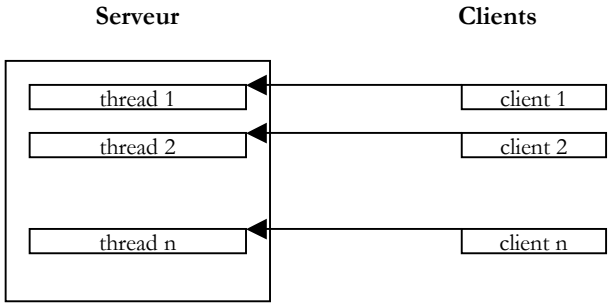
6.3 Intérêt des threads

Maintenant que nous avons mis en évidence l'existence d'un thread par défaut, celui qui exécute la méthode *Main*, et que nous savons comment en créer d'autres, arrêtons-nous sur l'intérêt pour nous des threads et sur la raison pour laquelle nous les présentons ici. Il y a un type d'applications qui se prêtent bien à l'utilisation des threads, ce sont les applications client-serveur de l'internet. Dans une telle application, un serveur situé sur une machine S1 répond aux demandes de clients situés sur des machines distantes C1, C2, ..., Cn.



Nous utilisons tous les jours des applications de l'internet correspondant à ce schéma : services Web, messagerie électronique, consultation de forums, transfert de fichiers... Dans le schéma ci-dessus, le serveur S1 doit servir les clients Ci de façon simultanée.

Si nous prenons l'exemple d'un serveur FTP (File Transfer Protocol) qui délivre des fichiers à ses clients, nous savons qu'un transfert de fichier peut prendre parfois plusieurs heures. Il est bien sûr hors de question qu'un client monopolise tout seul le serveur une telle durée. Ce qui est fait habituellement, c'est que le serveur crée autant de threads d'exécution qu'il y a de clients. Chaque thread est alors chargé de s'occuper d'un client particulier. Le processeur étant partagé cycliquement entre tous les threads actifs de la machine, le serveur passe alors un peu de temps avec chaque client assurant ainsi la simultanéité du service.



6.4 Une horloge graphique

Considérons l'application suivante qui affiche une fenêtre avec une horloge et un bouton pour arrêter ou redémarrer l'horloge :

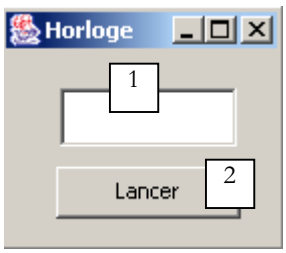
Pour que l'horloge vive, il faut qu'un processus s'occupe de changer l'heure toutes les secondes. En même temps, il faut surveiller les événements qui se produisent dans la fenêtre : lorsque l'utilisateur cliquera sur le bouton "Arrêter", il faudra stopper l'horloge. On a là deux tâches parallèles et asynchrones : l'utilisateur peut cliquer n'importe quand.

Considérons le moment où l'horloge n'a pas encore été lancée et où l'utilisateur clique sur le bouton "Lancer". On a là un événement classique et on pourrait penser qu'une méthode du thread dans lequel s'exécute la fenêtre peut alors gérer l'horloge. Seulement lorsqu'une méthode de l'application graphique s'exécute, le thread de celle-ci n'est plus à l'écoute des événements de l'interface graphique. Ceux-ci se produisent et sont mis dans une file d'attente pour être traités par l'application lorsque la méthode actuellement en cours d'exécution sera achevée. Dans notre exemple d'horloge, la méthode sera toujours en cours d'exécution puisque seul le clic sur le bouton "Arrêter" peut la stopper. Or cet événement ne sera traité que lorsque la méthode sera achevée. On tourne en rond.

La solution à ce problème serait que lorsque l'utilisateur clique sur le bouton "Lancer", une tâche soit lancée pour gérer l'horloge mais que l'application puisse continuer à écouter les événements qui se produisent dans la fenêtre. On aurait alors deux tâches distinctes qui s'exécuteraient en parallèle :

- gestion de l'horloge
- écoute des événements de la fenêtre

Revenons à notre horloge graphique :



n°	type	nom	rôle
1	JTextField (Editable=false)	txtHorloge	affiche l'heure
2	JButton	btnGoStop	arrête ou lance l'horloge

Le code utile de l'application construite avec JBuilder est la suivante :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

public class interfaceHorloge extends JFrame {
  JPanel contentPane;
  JTextField txtHorloge = new JTextField();
  JButton btnGoStop = new JButton();
```

```
  // attributs d'instance
  boolean finHorloge=true;
```

```
  //Construire le cadre
  public interfaceHorloge() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
```

```
  private void runHorloge(){
    // on boucle tant qu'on nous a pas dit d'arrêter
    while( ! finHorloge){
      // on récupère l'heure
      Calendar calendrier=Calendar.getInstance();
      String H=calendrier.get(Calendar.HOUR_OF_DAY)+":"
      +calendrier.get(Calendar.MINUTE)+":"
      +calendrier.get(Calendar.SECOND);
      // on l'affiche dans le champ T
      txtHorloge.setText(H);
      // attente d'une seconde
      try{
        Thread.sleep(1000);
      } catch (Exception e){
        // sortie avec erreur
        System.exit(1);
      }//try
    }// while
  }// runHorloge
```

```
  //Initialiser le composant
  private void jbInit() throws Exception  {
.....................
  }

  //Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée
  protected void processWindowEvent(WindowEvent e) {
.............
  }
```

```
  void btnGoStop_actionPerformed(ActionEvent e) {
    // on lance/arrête l'horloge
    // on récupère le libellé du bouton
    String libellé=btnGoStop.getText();
    // lancer ?
    if(libellé.equals("Lancer")){
      // on crée le thread dans lequel s'exécutera l'horloge
      Thread thHorloge=new Thread(){
        public void run(){
          runHorloge();
        }
      };//déf thread
      // on autorise le thread à démarrer
      finHorloge=false;
      // on change le libellé du bouton
      btnGoStop.setText("Arrêter");
      // on lance le thread
      thHorloge.start();
      // fin
      return;
    }//if
    // arrêter
    if(libellé.equals("Arrêter")){
```

```
      // on dit au thread de s'arrêter
      finHorloge=true;
      // on change le libellé du bouton
      btnGoStop.setText("Lancer");
      // fin
      return;
    }//if
  }
}
```

Lorsque l'utilisateur clique sur le bouton "Lancer", un thread est créé à l'aide d'une classe anonyme :

```
      Thread thHorloge=new Thread(){
        public void run(){
          runHorloge();
        }
```

La méthode *run* du thread renvoie à la méthode *runHorloge* de l'application. Ceci fait, le thread est lancé :

```
      // on lance le thread
      thHorloge.start();
```

La méthode *runHorloge* va alors s'exécuter :

```
  private void runHorloge(){
    // on boucle tant qu'on ne nous a pas dit d'arrêter
    while( ! finHorloge){
      // on récupère l'heure
      Calendar calendrier=Calendar.getInstance();
      String H=calendrier.get(Calendar.HOUR_OF_DAY)+":"
      +calendrier.get(Calendar.MINUTE)+":"
      +calendrier.get(Calendar.SECOND);
      // on l'affiche dans le champ T
      txtHorloge.setText(H);
      // attente d'une seconde
      try{
        Thread.sleep(1000);
      } catch (Exception e){
        // sortie avec erreur
        System.exit(1);
      }//try
    }// while
  }// runHorloge
```

Le principe de la méthode est le suivant :

1. affiche l'heure courante dans la boîte de texte *txtHorloge*
2. s'arrête 1 seconde
3. reprend l'étape 1 en ayant pris soin de tester auparavant le booléen *finHorloge* qui sera positionné à vrai lorsque l'utilisateur cliquera sur le bouton *Arrêter*.

6.5 Applet horloge

Nous transformons l'application graphique précédente en applet par la méthode habituelle et créons le document HTML *appletHorloge.htm* suivant :

```
<html>
  <head>
    <title>Applet Horloge</title>
  </head>
  <body>
    <h2>Une applet horloge</h2>
    <applet
      code="appletHorloge.class"
      width="150"
      height="130"
    ></applet>
    </center>
  </body>
</html>
```

Lorsque nous chargeons directement ce document dans IE en double-cliquant dessus, nous obtenons l'affichage suivant :

Tous les éléments nécessaires à l'applet sont dans cet exemple dans le même dossier :

```
E:\data\serge\Jbuilder\horloge\1>dir
13/06/2002  12:17               3 174 appletHorloge.class
13/06/2002  12:17                 658 appletHorloge$1.class
13/06/2002  12:17                 512 appletHorloge$2.class
13/06/2002  12:20                 245 appletHorloge.htm
```

Notre applet peut être améliorée. Nous avons dit qu'au chargement de l'applet, la méthode *init* était exécutée puis ensuite la méthode *start* si elle existe. De plus, lorsque l'utilisateur quitte la page, la méthode *stop* est exécutée si elle existe. Lorsqu'il revient sur la page de l'applet, la méthode *start* est de nouveau appelée. Lorsqu'une applet met en œuvre des threads d'animation visuelle, on utilise souvent les méthodes *start* et *stop* de l'applet pour lancer et arrêter les threads. Il est en effet inutile qu'un thread d'animation visuelle continue à travailler en arrière-plan alors que l'animation est cachée.

Nous ajoutons donc à notre applet les méthodes *start* et *stop* suivantes :

```
  public void stop(){
    // la page est cachée
    // suivi
    System.out.println("Page stop");
    // la page est cachée - on arrête le thread
    finHorloge=true;
  }

  public void start(){
    // la page réapparaît
    // suivi
    System.out.println("Page start");
    // on relance un nouveau thread horloge si nécessaire
    if(btnGoStop.getText().equals("Arrêter")){
      // on change le libellé
      btnGoStop.setText("Lancer");
      // et on fait comme si l'utilisateur avait cliqué dessus
      btnGoStop_actionPerformed(null);
    }//if
  }//start
```

Par ailleurs, nous avons ajouté un suivi dans la méthode *run* du thread pour savoir quand il démarre et s'arrête :

```
  private void runHorloge(){
    // suivi
    System.out.println("Thread horloge lancé");
    // on boucle tant qu'on nous a pas dit d'arrêter
    while( ! finHorloge){
      // on récupère l'heure
      Calendar calendrier=Calendar.getInstance();
      String H=calendrier.get(Calendar.HOUR_OF_DAY)+":"
      +calendrier.get(Calendar.MINUTE)+":"
      +calendrier.get(Calendar.SECOND);
      // on l'affiche dans le champ T
      txtHorloge.setText(H);
      // attente d'une seconde
      try{
        Thread.sleep(1000);
      } catch (Exception e){
        // sortie avec erreur
        return;
```

```
      }//try
    }// while
    // suivi
    System.out.println("Thread horloge terminé");
  }// runHorloge
```

Maintenant nous exécutons l'applet avec *AppletViewer* :

```
E:\data\serge\Jbuilder\horloge\1>appletviewer appletHorloge.htm
Page start                              // applet lancé - page affichée
Thread horloge lancé                    // le thread est lancé en conséquence
Page stop                               // applet mis en icône
Thread horloge terminé                  // le thread est arrêté en conséquence
Page start                              // applet réaffichée
Thread horloge lancé                    // le thread est relancé
Thread horloge terminé                  // appui sur bouton arrêter
Thread horloge lancé                    // appui sur bouton lancer
Page stop                               // applet en icône
Thread horloge terminé                  // thread arrêté en conséquence
Page start                              // réaffichage applet
Thread horloge lancé                    // thread relancé en conséquence
```

Avec *AppletViewer*, l'événement *start* se produit lorsque la fenêtre d'*AppletViewer* est visible et l'événement *stop* lorsqu'on la met en icône. Les résultats ci-dessus montrent que lorsque le document HTML est caché, le thread est bien arrêté s'il était actif.

6.6 Synchronisation de tâches

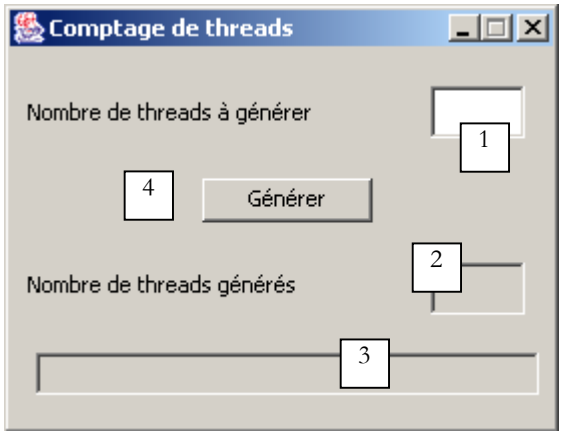
Dans notre exemple précédent, il y avait deux tâches :

- la tâche principale représentée par l'application elle-même
- la tâche chargée de l'horloge

La coordination entre les deux tâches était assurée par la tâche principale qui positionnait un booléen pour arrêter le thread de l'horloge. Nous abordons maintenant le problème de l'accès concurrent de tâches à des ressource communes, problème connu aussi sous le nom de "partage de ressources". Pour l'illustrer, nous allons d'abord étudier un exemple.

6.6.1 Un comptage non synchronisé

Considérons l'interface graphique suivante :

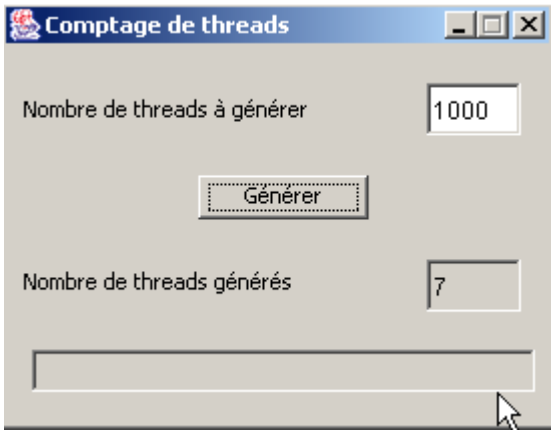


n°	type	nom	rôle
1	JTextField	txtAGénérer	indique le nombre de threads à générer
2	JTextfield (non éditable)	txtGénéres	indique le nombre de threads générés
3	JTextField (non éditable)	txtStatus	donne des informations sur les erreurs rencontrées et sur l'application elle-même
4	JButton	btnGénérer	lance la génération des threads

Le fonctionnement de l'application est le suivant :

- l'utilisateur indique le nombre de threads à générer dans le champ 1
- il lance la génération de ces threads avec le bouton 4
- les threads lisent la valeur du champ 2, l'incrémentent et affichent la nouvelle valeur. Au départ ce champ contient la valeur 0.

Les threads générés se partagent une ressource : la valeur du champ 2. Nous cherchons à montrer ici les problèmes que l'on rencontre dans une telle situation. Voici un exemple d'exécution :



On voit qu'on a demandé la génération de 1000 threads et qu'il en a été compté que 7. Le code utile de l'application est le suivant :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class interfaceSynchro extends JFrame {
  JPanel contentPane;
  JLabel jLabel1 = new JLabel();
  JTextField txtAGénérer = new JTextField();
  JButton btnGénérer = new JButton();
  JTextField txtStatus = new JTextField();
  JTextField txtGénérés = new JTextField();
  JLabel jLabel2 = new JLabel();
```

```
  // variables d'instance
    Thread[] tâches=null;   // les threads
    int[] compteurs=null;   // les compteurs
```

```
  //Construire le cadre
  public interfaceSynchro() {
..........
  }

  //Initialiser le composant
  private void jbInit() throws Exception  {
.......................
  }

  //Remplacé, ainsi nous pouvons sortir quand la fenêtre est fermée
  protected void processWindowEvent(WindowEvent e) {
...................
  }
```

```
  void btnGénérer_actionPerformed(ActionEvent e) {
    //génération des threads

    // on lit le nombre de threads à générer
    int nbThreads=0;
    try{
      // lecture du champ contenant le nbre de threads
      nbThreads=Integer.parseInt(txtAGénérer.getText().trim());
      // positif >
      if(nbThreads<=0) throw new Exception();
    }catch(Exception ex){
      //erreur
      txtStatus.setText("Nombre invalide");
      // on recommnece
      txtAGénérer.requestFocus();
```

```
      return;
    }//catch

    // au départ pas de threads générés
    txtGénérés.setText("0");  // cpteur de tâches à 0

    // on génère et on lance les threads
    tâches=new Thread[nbThreads];
    compteurs=new int[nbThreads];
    for(int i=0;i<tâches.length;i++){
      // on crée le thread i
      tâches[i]=new Thread() {
       public void run() {
          incrémente();
        }
      };//thread i
      // on définit son nom
      tâches[i].setName(""+i);
      // on lance son exécution
      tâches[i].start();
    }//for
  }//générer
```

```
  // incremente
  private void incrémente(){
    // on récupère le n° du thread
    int iThread=0;
    try{
      iThread=Integer.parseInt(Thread.currentThread().getName());
    }catch(Exception ex){}
    // on lit la valeur du compteur de tâches
    try{
      compteurs[iThread]=Integer.parseInt(txtGénérés.getText());
    } catch (Exception e){}
    // on l'incrémente
    compteurs[iThread]++;

    // on patiente 100 millisecondes - le thread va alors perdre le processeur
    try{
      Thread.sleep(100);
    } catch (Exception e){
      System.exit(0);
    }

    // on affiche le nouveau compteur
    txtGénérés.setText("");
    txtGénérés.setText(""+compteurs[iThread]);
    // suivi
    System.out.println("Thread " + iThread + " : " + compteurs[iThread]);
  }// incremente
```

```
}// classe
```

Détaillons le code :

- la fenêtre déclare deux variables d'instance :

```
  // variables d'instance
    Thread[] tâches=null;   // les threads
    int[] compteurs=null;   // les compteurs
```

Le tableau *tâches* sera le tableau des threads générés. Le tableau *compteurs* sera associé au tableau *tâches*. Chaque tâche aura un compteur propre pour récupérer la valeur du champ *txtGénérés* de l'interface graphique.

- lors d'un clic sur le bouton *Générer*, la méthode *btnGénérer_actionPerformed* est exécutée.
- celle-ci commence par récupérer le nombre de threads à générer. Au besoin, une erreur est signalée si ce nombre n'est pas exploitable. Elle génère ensuite les threads demandés en prenant soin de noter leurs références dans un tableau et en donnant à chacun d'eux un numéro. La méthode *run* des threads générés renvoie sur la méthode *incrémente* de la classe. Les threads sont tous lancés (*start*). Le tableau des compteurs associés aux threads est également créé.
- la méthode *incrémente* :
  - lit la valeur actuelle du champ *txtGénérés* et la stocke dans le compteur appartenant au thread en cours d'exécution
  - s'arrête 100 ms, ceci afin de perdre volontairement le processeur

- affiche la nouvelle valeur dans le champ *txtGénérés*

Expliquons maintenant pourquoi le comptage des threads est incorrect. Supposons qu'il y ait 2 threads à générer. Ils s'exécutent dans un ordre imprévisible. L'un d'entre-eux passe le premier et lit la valeur 0 du compteur. Il la passe alors à 1 mais il ne l'écrit pas dans la fenêtre : il s'interrompt volontairement pendant 100 ms. Il perd alors le processeur qui est alors donné à un autre thread. Celui-ci opère de la même façon que le précédent : il lit le compteur de la fenêtre et récupère le 0 qui s'y trouve toujours. Il passe le compteur à 1 et comme le précédent s'interrompt 100 ms. Le processeur est alors de nouveau accordé au premier thread : celui-ci va écrire la valeur 1 dans le compteur de la fenêtre et se terminer. Le processeur est maintenant accordé au second thread qui lui aussi va écrire 1. On a un résultat incorrect.

D'où vient le problème ? Le second thread a lu une mauvaise valeur du fait que le premier avait été interrompu avant d'avoir terminé son travail qui était de mettre à jour le compteur dans la fenêtre. Cela nous amène à la notion de ressource critique et de section critique d'un programme:

- une ressource critique est une ressource qui ne peut être détenue que par un thread à la fois. Ici la ressource critique est le compteur 2 de la fenêtre.
- une section critique d'un programme est une séquence d'instructions dans le flux d'exécution d'un thread au cours de laquelle il accède à une ressource critique. On doit assurer qu'au cours de cette section critique, il est le seul à avoir accès à la ressource.

6.6.2 Un comptage synchronisé par méthode

Dans l'exemple précédent, chaque thread exécutait la méthode *incrémente* de la fenêtre. La méthode *incrémente* était déclarée comme suit :

```
private void incremente()
```

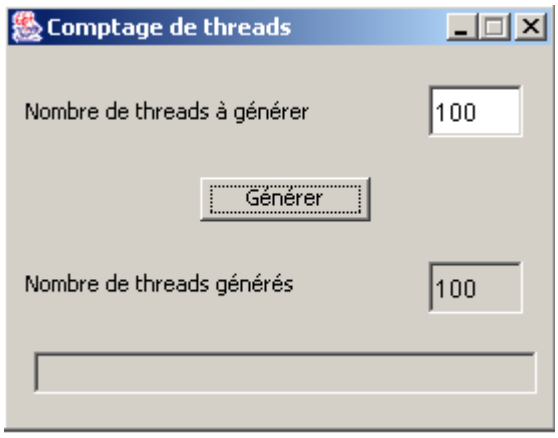
Maintenant nous la déclarons différemment :

```
// incremente
private synchronized void incrémente(){
```

Le mot clé **synchronized** signifie qu'un seul thread à la fois peut exécuter la méthode *incrémente*. Considérons les notations suivantes :

- l'objet fenêtre *F* qui crée les threads dans *btnGénérer_actionPerformed*
- deux threads T1 et T2 créés par F

Les deux threads sont créés par F puis lancés. Ils vont donc tous deux exécuter la méthode F.*run*. Supposons que T1 arrive le premier. Il exécute F.*run* puis F.*incremente* qui est une méthode synchronisée. Il lit la valeur 0 du compteur, l'incrémente puis s'arrête 100 ms. Le processeur est alors donné à T2 qui à son tour exécute F.*run* puis F.*incremente*. Et là il est bloqué car le thread T1 est en cours d'exécution de F.*incremente* et le mot clé ***synchronized*** assure qu'un seul thread à la fois peut exécuter F.*incremente*. T2 perd alors le processeur à son tour sans avoir pu lire la valeur du compteur. Au bout des 100 ms, T1 récupère le processeur, affiche la valeur 1 du compteur, quitte F.*incremente* puis F.*run* et se termine. T2 récupère alors le processeur et peut cette fois exécuter F.*incremente* car T1 n'est plus en cours d'exécution de cette méthode. T2 lit alors la valeur 1 du compteur, l'incrémente et s'arrête 100 ms. Au bout de 100 ms, il récupère le processeur, affiche la valeur 2 du compteur et se termine lui aussi. Cette fois, la valeur obtenue est correcte. Voici un exemple testé :



6.6.3 Comptage synchronisé par un objet

Dans l'exemple précédent, l'accès au compteur *txtGénérés* a été synchronisé par une méthode. Si la fenêtre qui crée les threads s'appelle F, on peut aussi dire que la méthode *F.incremente* représente une ressource qui ne devait être utilisée que par un seul thread à la fois. C'est donc une ressource critique. L'accès synchronisé à cette ressource a été garanti par le mot clé *synchronized* :

```
  private synchronized void incrémente()
```

On pourrait aussi dire que la ressource critique est l'objet F lui-même. C'est plus strict que dans le cas où la ressource critique est F.*incremente*. En effet, dans ce dernier cas, si un thread T1 execute *F.incremente*, un thread T2 ne pourra pas exécuter *F.incremente* mais pourra exécuter une autre méthode de l'objet F qu'elle soit synchronisée ou non. Dans le cas ou l'objet F est lui-même la ressource critique, lorsque un thread T1 exécute une section synchronisée de cet objet, toutre autre section synchronisée de l'objet devient inaccessible pour les autres threads. Ainsi si un thread T1 exécute la méthode synchronisée *F.incremente*, un thread T2 ne pourra pas exécuter non seulement *F.incremente* mais également toute autre section synchronisée de F, ceci même si aucun thread ne l'utilise. C'est donc une méthode plus contraignante.

Supposons donc que la fenêtre devienne la ressource critique. On écrira alors :

```
  // incremente
  private void incrémente(){
    // section critique
    synchronized(this){
      // on récupère le n° du thread
      int iThread=0;
      try{
        iThread=Integer.parseInt(Thread.currentThread().getName());
      }catch(Exception ex){}
      // on lit la valeur du compteur de tâches
      try{
        compteurs[iThread]=Integer.parseInt(txtGénérés.getText());
      } catch (Exception e){}
      // on l'incrémente
      compteurs[iThread]++;

      // on patiente 100 millisecondes - le thread va alors perdre le processeur
      try{
        Thread.sleep(100);
      } catch (Exception e){
        System.exit(0);
      }

      // on affiche le nouveau compteur
      txtGénérés.setText("");
      txtGénérés.setText(""+compteurs[iThread]);
    }//synchronized
  }// incremente
```

Tous les threads utilisent la fenêtre *this* pour se synchroniser. A l'exécution, on obtient les mêmes résultats corrects que précédemment. On peut en fait se synchroniser sur n'importe quel objet connu de tous les threads. Voici par exemple une autre méthode qui donne les mêmes résultats :

```
  // variables d'instance
    Thread[] tâches=null;   // les threads
    int[] compteurs=null;   // les compteurs
    Object synchro=new Object(); // un objet de synchronisation de threads

  // incremente
  private void incrémente(){
    // section critique
    synchronized(synchro){
..............
    }//synchronized
  }// incremente
```

La fenêtre crée un objet de type *Object* qui servira à la synchronisation des threads. Cette méthode est meilleure que celle qui se synchronise sur l'objet *this* parce que moins contraignante. Ici, si un thread T1 est dans la section synchronisée de *incrémente* et qu'un thread T2 veuille exécuter une autre section synchronisée du même objet *this* mais synchronisée par un autre objet que *synchro*, il le pourra.

6.6.4 Synchronisation par événements

Cette fois-ci, nous utilisons un booléen *peutPasser* pour signifier à un thread s'il peut entrer ou non dans une section critique. Une écriture sans synchronisation pourrait être la suivante :

```
while(! peutPasser);   // on attend que peutPasser passe à vrai
peutPasser=false;      // aucun autre thread ne doit passer
section critique;      // ici le thread est tout seul
peutPasser=true;       // un autre thread peut passer dans la section critique
```

La première instruction où un thread boucle en attendant que *peutPasser* passe à vrai est maladroite : le thread occupe le processeur inutilement. On parle d'attente active. On peut améliorer l'écriture comme suit :

```
while(! peutPasser){   // on attend que peutPasser passe à vrai
   Thread.sleep(100);// arrêt pendant 100 ms
}
peutPasser=false;      // aucun autre thread ne doit pas passer
section critique;      // ici le thread est tout seul
peutPasser=true;       // un autre thread peut passer dans la section critique
```

La boucle d'attente est ici meilleure : si le thread ne peut pas passer, il se met en sommeil pendant 100 ms avant de vérifier de nouveau s'il peut passer ou non. Le processeur va entre-temps être attribué à d'autres threads du système.

Ces deux méthodes sont en fait incorrectes : elle n'empêche pas deux threads de s'engouffrer en même temps dans la section critique. Supposons qu'un thread T1 détecte que *peutPasser* est à vrai. Il va alors passer à l'instruction suivante où il remet *peutPasser* à faux pour bloquer les autres Threads. Seulement, il peut très bien être interrompu à ce moment, soit parce que sa part de temps du processeur est épuisée, soit parce qu'une tâche plus prioritaire a demandé le processeur ou pour une autre raison. Le résultat est qu'il perd le processeur. Il le retrouvera un peu plus tard. Entre-temps d'autres tâches vont obtenir le processeur dont peut-être un thread T2 qui boucle en attendant que *peutPasser* passe à vrai. Lui aussi va découvrir que *peutPasser* est à vrai (le premier thread n'a pas eu le temps de le mettre à faux) et va passer lui-aussi dans la section critique. Ce qu'il ne fallait pas.

La séquence

```
while(! peutPasser){     // on attend que peutPasser passe à vrai
   try{
    Thread.sleep(100);  // arrêt pendant 100 ms
  } catch (Exception e) {}
}// while
peutPasser=false;       // aucun autre thread ne doit passer
```

est une séquence critique qu'il faut protéger par une synchronisation. S'inspirant de l'exemple précédent, on peut écrire :

```
  synchronized(synchro){
    while(! peutPasser){     // on attend que peutPasser passe à vrai
      try{
        Thread.sleep(100);  // arrêt pendant 100 ms
      } catch (Exception e) {}
    }//while
    peutPasser=false;       // aucun autre thread ne doit pas passer
  }// synchronized
  section critique;         // ici le thread est tout seul
  peutPasser=true;          // un autre thread peut passer dans la section critique
```

Cet exemple fonctionne correctement. On peut l'améliorer en évitant l'attente semi-active du thread lorsqu'il surveille régulièrement la valeur du booléen *peutPasser*. Au lieu de se réveiller régulièrement toutes les 100 ms pour vérifier l'état de *peutPasser*, il peut s'endormir et demander à ce qu'on le réveille lorsque *peutPasser* sera à vrai. On écrit cela de la façon suivante :

```
synchronized(synchro){
  if (! peutPasser) {
    try{
      synchro.wait();     // si on ne peut pas passer alors on attend
    } catch (Exception e){
      …
    }
  }
  peutPasser=false;     // aucun autre thread ne doit pas passer
}// synchronized
```

L'opération **synchro.wait()** ne peut être faite que par un thread "propriétaire" momentané de l'objet *synchro*. Ici, c'est la séquence :

```
synchronized(synchro){
```

```
    …
}// synchronized
```

qui assure que le thread est propriétaire de l'objet *synchro*. Par l'opération **synchro.wait()**, le thread cède la propriété du verrou de synchronisation. Pourquoi cela ? En général parce qu'il lui manque des ressources pour continuer à travailler. Alors plutôt que de bloquer les autres threads en attente de la ressource *synchro*, il la cède et se met en attente de la ressource qui lui manque. Dans notre exemple, il attend que le booléen *peutPasser* passe à vrai. Comment sera-t-il averti de cet événement ? De la façon suivante :

```
synchronized(synchro){
  if (! peutPasser) {
    try{
      synchro.wait();     // si on ne peut pas passer alors on attend
    } catch (Exception e){
      …
    }
  }
  peutPasser=false;     // aucun autre thread ne doit passer
}// synchronized
section critique...
synchronized(synchro){
      synchro.notify();
    }
```

Considérons le premier thread qui passe le verrou de synchronisation. Appelons le T1. Imaginons qu'il trouve le booléen *peutPasser* à vrai puisqu'il est le premier. Il le passe donc à faux. Il sort ensuite de la section critique verrouillée par l'objet *synchro*. Un autre thread pourra alors entrer dans la section critique pour tester la valeur de *peutPasser*. Il le trouvera faux et se mettra alors en attente d'un événement (*wait*). Ce faisant, il cède la propriété de l'objet *synchro*. Un autre thread peut alors entrer dans la section critique : lui aussi se mettra en attente car *peutPasser* est à faux. On peut donc avoir plusieurs threads en attente d'un événement sur l'objet *synchro*. Revenons au thread T1 qui lui est passé. Il exécute la section critique puis va indiquer qu'un autre thread peut maintenant passer. Il le fait avec la séquence :

```
synchronized(synchro){
      synchro.notify();
    }
```

Il doit d'abord reprendre possession de l'objet *synchro* avec l'instruction *synchronized*. Ca ne doit pas poser de problème puisqu'il est en compétition avec des threads qui, s'ils obtiennent momentanément l'objet *synchro* doivent l'abandonner par un *wait* parce qu'ils trouvent *peutPasser* à faux. Donc notre thread T1 va bien finir par obtenir la propriété de l'objet *synchro*. Ceci fait, il indique par l'opération *synchro.notify* que l'un des threads bloqués par un *synchro.wait* doit être réveillé. Ensuite il abandonne de nouveau la propriété de l'objet *synchro* qui va alors être donnée à l'un des threads en attente. Celui-ci poursuit son exécution avec l'instruction qui suit le *wait* qui l'avait mis en attente. A son tour, il va exécuter la section critique et exécuter un *synchro.notify* pour libérer un autre thread. Et ainsi de suite.

Voyons ce mode de fonctionnement sur l'exemple du comptage déjà étudié.

```
  void btnGénérer_actionPerformed(ActionEvent e) {
    //génération des threads

    // on lit le nombre de threads à générer
    int nbThreads=0;
    try{
      // lecture du champ contenant le nbre de threads
      nbThreads=Integer.parseInt(txtAGénérer.getText().trim());
      // positif >
      if(nbThreads<=0) throw new Exception();
    }catch(Exception ex){
      //erreur
      txtStatus.setText("Nombre invalide");
      // on recommnece
      txtAGénérer.requestFocus();
      return;
    }//catch

    // RAZ compteur de tâches
    txtGénérés.setText("0");  // cpteur de tâches à 0
    // 1er thread peut passer
    peutPasser=true;
    // on génère et on lance les threads
    tâches=new Thread[nbThreads];
    compteurs=new int[nbThreads];
    for(int i=0;i<tâches.length;i++){
      // on crée le thread i
      tâches[i]=new Thread() {
       public void run() {
```

```
          synchronise();
        }
      };//thread i
      // on définit son nom
      tâches[i].setName(""+i);
      // on lance son exécution
      tâches[i].start();
    }//for
  }//générer
```

Maintenant, les threads n'exécutent plus la méthode *incrémente* mais la méthode *synchronise* suivante :

```
  // étape de synchronisation des threads
  public void synchronise(){
    // on demande l'accès à la section critique
    synchronized(synchro){
      try{
        // peut-on passer ?
        if(! peutPasser){
          System.out.println(Thread.currentThread().getName()+ " en attente");
          synchro.wait();
        }
        // on est passé - on interdit aux autres threads de passer
        peutPasser=false;
      } catch(Exception e){
        txtStatus.setText(""+e);
         return;
      }//try
    }// synchronized
    // section critique
    System.out.println(Thread.currentThread().getName()+ " passé");
    incrémente();
    // on a fini - on libère un éventuel thread bloqué à l'entrée de la section critique
    peutPasser=true;
    System.out.println(Thread.currentThread().getName()+ " terminé");
    synchronized(synchro){
      synchro.notify();
    }// synchronized
  } // synchronise
```

La méthode *synchronise* a pour but de faire passer les threads un par un. Elle utilise pour cela une variable de synchronisation *synchro*. La méthode *incrémente* n'est maintenant plus protégée par le mot clé *synchronized* :

```
  // incremente
  private void incrémente(){
    // on récupère le n° du thread
    int iThread=0;
    try{
      iThread=Integer.parseInt(Thread.currentThread().getName());
    }catch(Exception ex){}
    // on lit la valeur du compteur de tâches
    try{
      compteurs[iThread]=Integer.parseInt(txtGénérés.getText());
    } catch (Exception e){}
    // on l'incrémente
    compteurs[iThread]++;
    // on patiente 100 millisecondes - le thread va alors perdre le processeur
    try{
      Thread.sleep(100);
    } catch (Exception e){
      System.exit(0);
    }
    // on affiche le nouveau compteur
    txtGénérés.setText("");
    txtGénérés.setText(""+compteurs[iThread]);
  }// incremente
```

Pour 5 threads, les résultats obtenus sont les suivants :

```
0 passé
1 en attente
2 en attente
3 en attente
4 en attente
0 terminé
1 passé
```

```
1 terminé
2 passé
2 terminé
3 passé
3 terminé
4 passé
4 terminé
```

Soient T0 à T4 les 5 threads générés par l'application. T0 prend le premier la propriété du verrou *synchro* et trouve *peutPasser* à vrai. Il met *peutPasser* à faux et passe : c'est le sens du premier message *passé*. Selon toute vraisemblance, il continue et exécute la section critique et notamment la méthode *incrémente*. Dans celle-ci, il va s'endormir pendant 100 ms (*sleep*). Il lâche donc le processeur. Celui-ci est accordé à un autre thread, le thread T1 qui obtient alors la propriété de l'objet *synchro*. Il découvre qu'il ne peut pas passer et se met en attente (*wait*). Il lâche alors la propriété de l'objet *synchro* ainsi que le processeur. Celui-ci est accordé au thread T2 qui subit le même sort. Pendant les 100 ms d'arrêt de T0, les threads T1 à T4 sont donc mis en attente. c'est le sens des 4 messages "*en attente*". Au bout de 100 ms, T0 récupère le processeur et termine son travail : c'est le sens du message "0 *terminé*". Il libère ensuite l'un des threads bloqués et se termine. Le processeur libéré est affecté alors à un thread disponible : celui qui vient d'être libéré. Ici c'est T1. Le thread T1 entre alors dans la section critique : c'est le sens du message "1 *passé*". Il fait ce qu'il a à faire et va à son tour s'arrêter 100 ms. Le processeur est alors disponible pour un autre thread mais tous sont en attente d'un événement : aucun d'eux ne peut prendre le processeur. Au bout de 100 ms, le thread T1 récupère le processeur et se termine : c'est le sens du message "1 *terminé*". Les Threads T1 à T4 vont avoir le même comportement que T1 : c'est le sens des trois séries de messages : "passé", "terminé".

7. Programmation TCP-IP

7.1 Généralités

7.1.1 Les protocoles de l'Internet

Nous donnons ici une introduction aux protocoles de communication de l'Internet, appelés aussi suite de protocoles **TCP/IP** (*Transfer Control Protocol / Internet Protocol*), du nom des deux principaux protocoles. Il est bon que le lecteur ait une compréhension globale du fonctionnement des réseaux et notamment des protocoles TCP/IP avant d'aborder la construction d'applications distribuées.

Le texte qui suit est une traduction partielle d'un texte que l'on trouve dans le document *"Lan Workplace for Dos - Administrator's Guide"* de NOVELL, document du début des années 90.

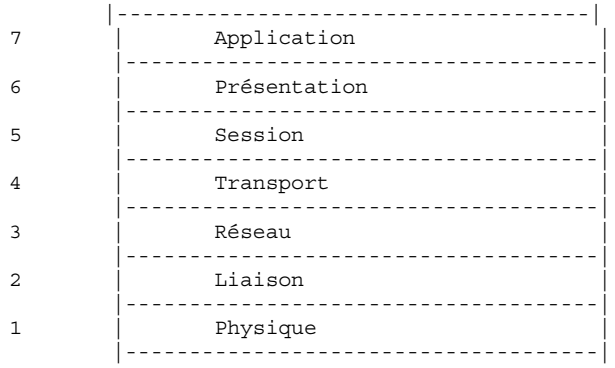
Le concept général de créer un réseau d'ordinateurs hétérogènes vient de recherches effectuées par le **DARPA** (**D**efense **A**dvanced **R**esearch **P**rojects **A**gency) aux Etats-Unis. Le DARPA a développé la suite de protocoles connue sous le nom de TCP/IP qui permet à des machines hétérogènes de communiquer entre elles. Ces protocoles ont été testés sur un réseau appelé **ARPAnet**, réseau qui devint ultérieurement le réseau **INTERNET**. Les protocoles TCP/IP définissent des formats et des règles de transmission et de réception indépendants de l'organisation des réseaux et des matériels utilisés.

Le réseau conçu par le DARPA et géré par les protocoles TCP/IP est un réseau à **commutation de paquets**. Un tel réseau transmet l'information sur le réseau, en petits morceaux appelés **paquets**. Ainsi, si un ordinateur transmet un gros fichier, ce dernier sera découpé en petits morceaux qui seront envoyés sur le réseau pour être recomposés à destination. TCP/IP définit le format de ces paquets, à savoir :

. origine du paquet
. destination
. longueur
. type

7.1.2 Le modèle OSI

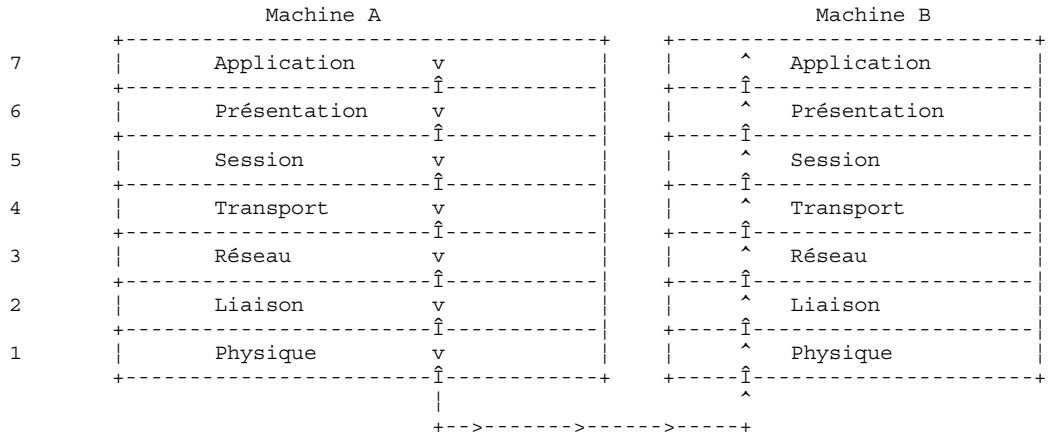
Les protocoles TCP/IP suivent à peu près le modèle de réseau ouvert appelé **OSI** (**O**pen **S**ystems **I**nterconnection Reference Model) défini par l'**ISO** (**I**nternational **S**tandards **O**rganisation). Ce modèle décrit un réseau idéal où la communication entre machines peut être représentée par un modèle à sept couches :



Chaque couche reçoit des services de la couche inférieure et offre les siens à la couche supérieure. Supposons que deux applications situées sur des machines A et B différentes veulent communiquer : elles le font au niveau de la couche *Application*. Elles n'ont pas

besoin de connaître tous les détails du fonctionnement du réseau : chaque application remet l'information qu'elle souhaite transmettre à la couche du dessous : la couche *Présentation*. L'application n'a donc à connaître que les règles d'interfaçage avec la couche *Présentation*.
Une fois l'information dans la couche *Présentation*, elle est passée selon d'autres règles à la couche *Session* et ainsi de suite, jusqu'à ce que l'information arrive sur le support physique et soit transmise physiquement à la machine destination. Là, elle subira le traitement inverse de celui qu'elle a subi sur la machine expéditeur.

A chaque couche, le processus expéditeur chargé d'envoyer l'information, l'envoie à un processus récepteur sur l'autre machine apartenant à la même couche que lui. Il le fait selon certaines règles que l'on appelle le **protocole** de la couche. On a donc le schéma de communication final suivant :



Le rôle des différentes couches est le suivant :

`Physique` Assure la transmission de bits sur un support physique. On trouve dans cette couche des équipements terminaux de traitement des données (E.T.T.D.) tels que terminal ou ordinateur, ainsi que des équipements de terminaison de circuits de données (E.T.C.D.) tels que modulateur/démodulateur, multiplexeur, concentrateur. Les points d'intérêt à ce niveau sont :

. le choix du codage de l'information (analogique ou numérique)
. le choix du mode de transmission (synchrone ou asynchrone).

`Liaison de données` Masque les particularités physiques de la couche Physique. Détecte et corrige les erreurs de transmission.

`Réseau` Gère le chemin que doivent suivre les informations envoyées sur le réseau. On appelle cela le *routage* : déterminer la route à suivre par une information pour qu'elle arrive à son destinataire.

`Transport` Permet la communication entre deux applications alors que les couches précédentes ne permettaient que la communication entre machines. Un service fourni par cette couche peut être le multiplexage : la couche transport pourra utiliser une même connexion réseau (de machine à machine) pour transmettre des informations appartenant à plusieurs applications.

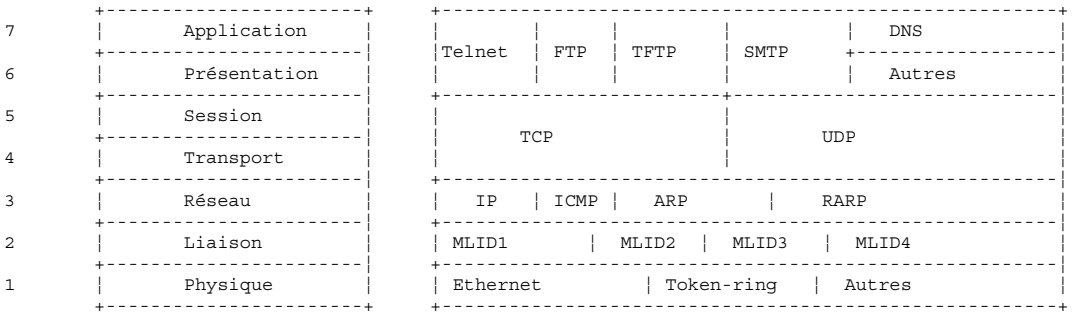
`Session` On va trouver dans cette couche des services permettant à une application d'ouvrir et de maintenir une session de travail sur une machine distante.

`Présentation` Elle vise à uniformiser la représentation des données sur les différentes machines. Ainsi des données provenant d'une machine A, vont être "habillées" par la couche *Présentation* de la machine A, selon un format standard avant d'être envoyées sur le réseau. Parvenues à la couche *Présentation* de la machine destinatrice B qui les reconnaîtra grâce à leur format standard, elles seront habillées d'une autre façon afin que l'application de la machine B les reconnaisse.

`Application` A ce niveau, on trouve les applications généralement proches de l'utilisateur telles que la messagerie électronique ou le transfert de fichiers.

7.1.3 Le modèle TCP/IP

Le modèle OSI est un modèle idéal encore jamais réalisé. La suite de protocoles TCP/IP s'en approche sous la forme suivante :



Couche Physique

En réseau local, on trouve généralement une technologie **Ethernet** ou **Token-Ring**. Nous ne présentons ici que la technologie Ethernet.

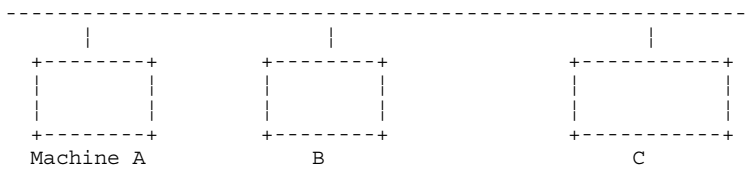
Ethernet

C'est le nom donné à une technologie de réseaux locaux à commutation de paquets inventée à PARC Xerox au début des années 1970 et normalisée par Xerox, Intel et Digital Equipment en 1978. Le réseau est physiquement constitué d'un câble coaxial d'environ 1,27 cm de diamètre et d'une longueur de 500 m au plus. Il peut être étendu au moyen de *répéteurs*, deux machines ne pouvant être séparées par plus de deux répéteurs. Le câble est passif : tous les éléments actifs sont sur les machines raccordées au câble. Chaque machine est reliée au câble par une carte d'accès au réseau comprenant :

- un transmetteur (*transceiver*) qui détecte la présence de signaux sur le câble et convertit les signaux analogiques en signaux numérique et inversement.
- un coupleur qui reçoit les signaux numériques du transmetteur et les transmet à l'ordinateur pour traitement ou inversement.

Les caractéristiques principales de la technologie Ethernet sont les suivantes :

- Capacité de 10 Mégabits/seconde.
- Topologie en bus : toutes les machines sont raccordées au même câble



- Réseau diffusant - Une machine qui émet transfère des informations sur le câble avec l'adresse de la machine destinatrice. Toutes les machines raccordées reçoivent alors ces informations et seule celle à qui elles sont destinées les conserve.

- La méthode d'accès est la suivante : le transmetteur désirant émettre écoute le câble - il détecte alors la présence ou non d'une onde porteuse, présence qui signifierait qu'une transmission est en cours. C'est la technique **CSMA** (*Carrier Sense Multiple Access*). En l'absence de porteuse, un transmetteur peut décider de transmettre à son tour. Ils peuvent être plusieurs à prendre cette décision. Les signaux émis se mélangent : on dit qu'il y a **collision**. Le transmetteur détecte cette situation : en même temps qu'il émet sur le câble, il écoute ce qui passe réellement sur celui-ci. S'il détecte que l'information transitant sur le câble n'est pas celle qu'il a émise, il en déduit qu'il y a collision et il s'arrêtera d'émettre. Les autres transmetteurs qui émettaient feront de même. Chacun reprendra son émission après un temps aléatoire dépendant de chaque transmetteur. Cette technique est appelée **CD** (*Collision Detect*). La méthode d'accès est ainsi appelée **CSMA/CD**.

- un adressage sur 48 bits. Chaque machine a une adresse, appelée ici adresse physique, qui est inscrite sur la carte qui la relie au câble. On appelle cet adresse, l'adresse *Ethernet* de la machine.

Couche Réseau

Nous trouvons au niveau de cette couche, les protocoles IP, ICMP, ARP et RARP.

IP (Internet Protocol)	Délivre des paquets entre deux noeuds du réseau
ICMP (Internet Control Message Protocol)	ICMP réalise la communication entre le programme du protocole IP d'une machine et celui d'une autre machine. C'est donc un protocole d'échange de messages à l'intérieur même du protocole IP.
ARP (Address Resolution Protocol)	fait la correspondance adresse Internet machine--> adresse physique machine
RARP (Reverse Address Resolution Protocol)	fait la correspondance adresse physique machine--> adresse Internet machine

Couches Transport/Session

Dans cette couche, on trouve les protocoles suivants :

TCP (Transmission Control Protocol)	Assure une remise fiable d'informations entre deux clients
UDP (User Datagram Protocol)	Assure une remise non fiable d'informations entre deux clients

Couches Application/Présentation/Session

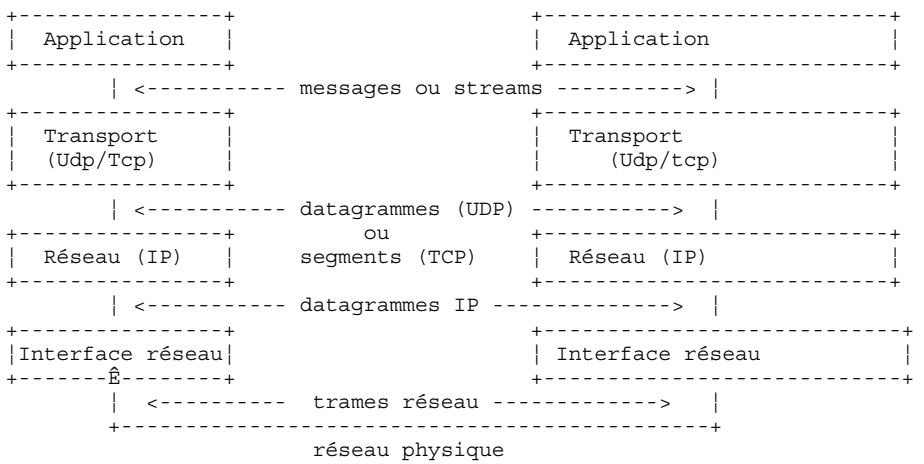
On trouve ici divers protocoles :

TELNET	Emulateur de terminal permettant à une machine A de se connecter à une machine B en tant que terminal
FTP (File Transfer Protocol)	permet des transferts de fichiers
TFTP (Trivial File Transfer Protocol)	permet des transferts de fichiers
SMTP (Simple Mail Transfer protocol)	permet l'échange de messages entre utilisateurs du réseau
DNS (Domain Name System)	transforme un nom de machine en adresse Internet de la machine
XDR (eXternal Data Representation)	créé par sun MicroSystems, il spécifie une représentation standard des données, indépendante des machines
RPC(Remote Procedures Call)	défini également par Sun, c'est un protocole de communication entre applications distantes, indépendant de la couche transport. Ce protocole est important : il décharge le programmeur de la connaissance des détails de la couche transport et rend les applications portables. Ce protocole s'appuie sur sur le protocole XDR
NFS (Network File System)	toujours défini par Sun, ce protocole permet à une machine, de "voir" le système de fichiers d'une autre machine. Il s'appuie sur le protocole RPC précédent

7.1.4 Fonctionnement des protocoles de l'Internet

Les applications développées dans l'environnement TCP/IP utilisent généralement plusieurs des protocoles de cet environnement. Un programme d'application communique avec la couche la plus élevée des protocoles. Celle-ci passe l'information à la couche du dessous et ainsi de suite jusqu'à arriver sur le support physique. Là, l'information est physiquement transférée à la machine

destinatrice où elle retraversera les mêmes couches, en sens inverse cette fois-ci, jusqu'à arriver à l'application destinatrice des informations envoyées. Le schéma suivant montre le parcours de l'information :



Prenons un exemple : l'application FTP, définie au niveau de la couche *Application* et qui permet des transferts de fichiers entre machines.

. L'application délivre une suite d'octets à transmettre à la couche *transport.*
. La couche *transport* découpe cette suite d'octets en *segments* TCP, et ajoute au début de chaque segment, le numéro de celui-ci. Les segments sont passés à la couche Réseau gouvernée par le protocole **IP**.
. La couche IP crée un paquet encapsulant le segment TCP reçu. En tête de ce paquet, elle place les adresses Internet des machines source et destination. Elle détermine également l'adresse physique de la machine destinatrice. Le tout est passé à la couche *Liaison de données & Liaison physique*, c'est à dire à la carte réseau qui couple la machine au réseau physique.
. Là, le paquet IP est encapsulé à son tour dans une **trame** physique et envoyé à son destinataire sur le câble.
. Sur la machine destinatrice, la couche *Liaison de données & Liaison physique* fait l'inverse : elle désencapsule le paquet IP de la trame physique et le passe à la couche IP.
. La couche *IP* vérifie que le paquet est correct : elle calcule une somme, fonction des bits reçus (*checksum*), somme qu'elle doit retrouver dans l'en-tête du paquet. Si ce n'est pas le cas, celui-ci est rejeté.
. Si le paquet est déclaré correct, la couche IP désencapsule le segment TCP qui s'y trouve et le passe au-dessus à la couche *transport*.
. La couche *transport*, couche *TCP* dans notre exemple, examine le numéro du segment afin de restituer le bon ordre des segments.
. Elle calcule également une somme de vérification pour le segment TCP. S'il est trouvé correct, la couche TCP envoie un accusé de réception à la machine source, sinon le segment TCP est refusé.
. Il ne reste plus à la couche TCP qu'à transmettre la partie données du segment à l'application destinatrice de celles-ci dans la couche du dessus.

7.1.5 Les problèmes d'adressage dans l'Internet

Un *noeud* d'un réseau peut être un ordinateur, une imprimante intelligente, un serveur de fichiers, n'importe quoi en fait pouvant communiquer à l'aide des protocoles TCP/IP. Chaque noeud a une **adresse physique** ayant un format dépendant du type du réseau. Sur un réseau Ethernet, l'adresse physique est codée sur 6 octets. Une adresse d'un réseau X25 est un nombre à 14 chiffres.

L'**adresse Internet** d'un noeud est une adresse **logique** : elle est indépendante du matériel et du réseau utilisé. C'est une adresse sur 4 octets identifiant à la fois un réseau local et un noeud de ce réseau. L'adresse Internet est habituellement représentée sous la forme de 4 nombres, valeurs des 4 octets, séparés par un point. Ainsi l'adresse de la machine Lagaffe de la faculté des Sciences d'Angers est notée 193.49.144.1 et celle de la machine Liny 193.49.144.9. On en déduira que l'adresse Internet du réseau local est 193.49.144.0. On pourra avoir jusqu'à 254 noeuds sur ce réseau.

Parce que les adresses Internet ou adresses IP sont indépendantes du réseau, une machine d'un réseau A peut communiquer avec une machine d'un réseau B sans se préoccuper du type de réseau sur lequel elle se trouve : il suffit qu'elle connaisse son adresse IP. Le protocole IP de chaque réseau se charge de faire la conversion adresse IP <--> adresse physique, dans les deux sens.

Les adresses **IP** doivent être toutes différentes. Des organismes officiels sont chargés de les distribuer. En fait, ces organismes délivrent une adresse pour des réseaux locaux, par exemple 193.49.144.0 pour le réseau de la faculté des sciences d'Angers. L'administrateur de ce réseau peut ensuite affecter les adresses IP 193.49.144.1 à 193.49.144.254 comme il l'entend. Cette adresse est généralement inscrite dans un fichier particulier de chaque machine reliée au réseau.

7.1.5.1 Les classes d'adresses IP

Une adresse IP est une suite de 4 octets notée souvent I1.I2.I3.I4, qui contient en fait deux adresses :

. l'adresse du réseau
. l'adresse d'un noeud de ce réseau

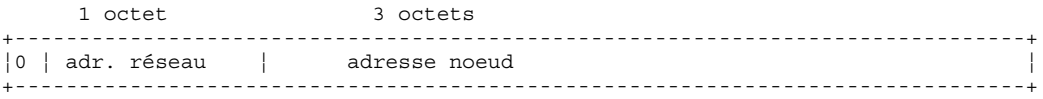
Selon la taille de ces deux champs, les adresses IP sont divisées en 3 classes : classes A, B et C.

Classe A

L'adresse IP : I1.I2.I3.I4 a la forme R1.N1.N2.N3 où

R1	est l'adresse du réseau
N1.N2.N3	est l'adresse d'une machine dans ce réseau

Plus exactement, la forme d'une adresse IP de classe A est la suivante :



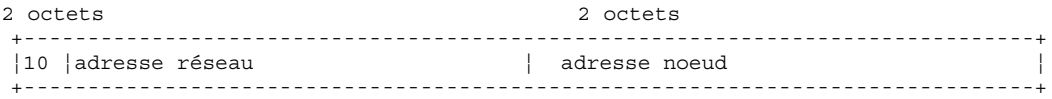
L'adresse réseau est sur 7 bits et l'adresse du noeud sur 24 bits. On peut donc avoir 127 réseaux de classe A, chacun comportant jusqu'à 2^{24} noeuds.

Classe B

Ici, l'adresse IP : I1.I2.I3.I4 a la forme R1.R2.N1.N2 où

R1.R2	est l'adresse du réseau
N1.N2	est l'adresse d'une machine dans ce réseau

Plus exactement, la forme d'une adresse IP de classe B est la suivante :



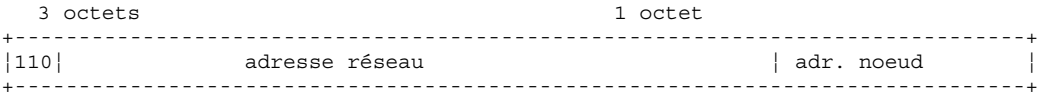
L'adresse du réseau est sur 2 octets (14 bits exactement) ainsi que celle du noeud. On peut donc avoir 2^{14} réseaux de classe B chacun comportant jusqu'à 2^{16} noeuds.

Classe C

Dans cette classe, l'adresse IP : I1.I2.I3.I4 a la forme R1.R2.R3.N1 où

R1.R2.R3	est l'adresse du réseau
N1	est l'adresse d'une machine dans ce réseau

Plus exactement, la forme d'une adresse IP de classe C est la suivante :



L'adresse réseau est sur 3 octets (moins 3 bits) et l'adresse du noeud sur 1 octet. On peut donc avoir 2^{21} réseaux de classe C comportant jusqu'à 256 noeuds.

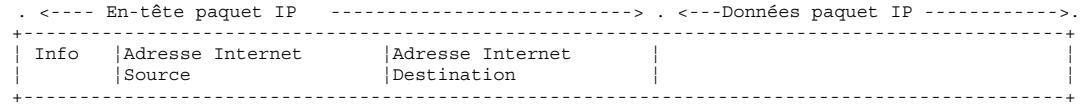
L'adresse de la machine *Lagaffe* de la faculté des sciences d'Angers étant 193.49.144.1, on voit que l'octet de poids fort vaut 193, c'est à dire en binaire 11000001. On en déduit que le réseau est de classe C.

Adresses réservées

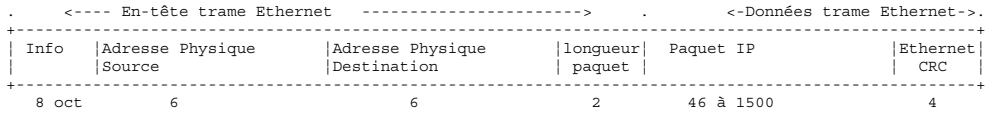
. Certaines adresses IP sont des adresses de réseaux plutôt que des adresses de noeuds dans le réseau. Ce sont celles, où l'adresse du noeud est mise à 0. Ainsi, l'adresse 193.49.144.0 est l'adresse IP du réseau de la Faculté des Sciences d'Angers. En conséquence, aucun noeud d'un réseau ne peut avoir l'adresse zéro.
. Lorsque dans une adresse IP, l'adresse du noeud ne comporte que des 1, on a alors une adresse de diffusion : cette adresse désigne **tous les noeuds du réseau**.
. Dans un réseau de classe C, permettant théoriquement 2^8=256 noeuds, si on enlève les deux adresses interdites, on n'a plus que 254 adresses autorisées.

7.1.5.2 Les protocoles de conversion Adresse Internet <--> Adresse physique

Nous avons vu que lors d'une émission d'informations d'une machine vers une autre, celles-ci à la traversée de la couche IP étaient encapsulées dans des paquets. Ceux-ci ont la forme suivante :



Le paquet IP contient donc les adresses Internet des machines source et destination. Lorsque ce paquet va être transmis à la couche chargée de l'envoyer sur le réseau physique, d'autres informations lui sont ajoutées pour former la trame physique qui sera finalement envoyée sur le réseau. Par exemple, le format d'une trame sur un réseau Ethernet est le suivant :



Dans la trame finale, il y a l'adresse physique des machines source et destination. Comment sont-elles obtenues ?

La machine expéditrice connaissant l'adresse IP de la machine avec qui elle veut communiquer obtient l'adresse physique de celle-ci en utilisant un protocole particulier appelé **ARP** (*Address Resolution Protocol*).

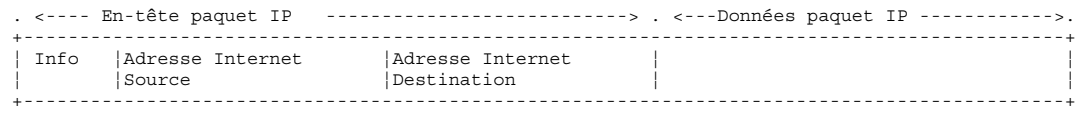
. Elle envoie un paquet d'un type spécial appelé paquet ARP contenant l'adresse IP de la machine dont on cherche l'adresse physique. Elle a pris soin également d'y placer sa propre adresse IP ainsi que son adresse physique.
. Ce paquet est envoyé à tous les noeuds du réseau.
. Ceux-ci reconnaissent la nature spéciale du paquet. Le noeud qui reconnaît son adresse IP dans le paquet, répond en envoyant à l'expéditeur du paquet son adresse physique. Comment le peut-il ? Il a trouvé dans le paquet les adresses IP et physique de l'expéditeur.
. L'expéditeur reçoit donc l'adresse physique qu'il cherchait. Il la stocke en mémoire afin de pouvoir l'utiliser ultérieurement si d'autres paquets sont à envoyer au même destinataire.

L'adresse IP d'une machine est normalement inscrite dans l'un de ses fichiers qu'elle peut donc consulter pour la connaître. Cette adresse peut être changée : il suffit d'éditer le fichier. L'adresse physique elle, est inscrite dans une mémoire de la carte réseau et ne peut être changée.

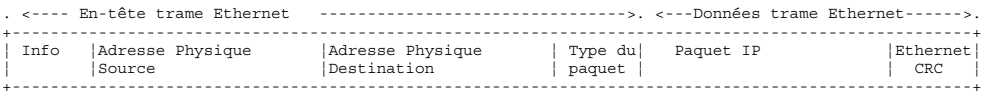
Lorsqu'un administrateur désire d'organiser son réseau différemment, il peut être amené à changer les adresses IP de tous les noeuds et donc à éditer les différents fichiers de configuration des différents noeuds. Cela peut être fastidieux et une occasion d'erreurs s'il y a beaucoup de machines. Une méthode consiste à ne pas affecter d'adresse IP aux machines : on inscrit alors un code spécial dans le fichier dans lequel la machine devrait trouver son adresse IP. Découvrant qu'elle n'a pas d'adresse IP, la machine la demande selon un protocole appelé **RARP** *(Reverse Address Resolution Protocol)*. Elle envoie alors sur un réseau un paquet spécial appelé paquet RARP, analogue au paquet ARP précédent, dans lequel elle met son adresse physique. Ce paquet est envoyé à tous les noeuds qui reconnaissent alors un paquet RARP. L'un d'entre-eux, appelé **serveur RARP**, possède un fichier donnant la correspondance adresse physique <--> adresse IP de tous les noeuds. Il répond alors à l'expéditeur du paquet RARP, en lui renvoyant son adresse IP. Un administrateur désirant reconfigurer son réseau, n'a donc qu'à éditer le fichier de correspondances du serveur RARP. Celui-ci doit normalement avoir une adresse IP fixe qu'il doit pouvoir connaître sans avoir à utiliser lui-même le protocole RARP.

7.1.6 La couche réseau dite couche IP de l'internet

Le protocole IP *(Internet Protocol)* définit la forme que les paquets doivent prendre et la façon dont ils doivent être gérés lors de leur émission ou de leur réception. Ce type de paquet particulier est appelé un **datagramme IP**. Nous l'avons déjà présenté :



L'important est qu'outre les données à transmettre, le datagramme IP contient les adresses Internet des machines source et destination. Ainsi la machine destinatrice sait qui lui envoie un message.
A la différence d'une trame de réseau qui a une longueur déterminée par les caractéristiques physiques du réseau sur lequel elle transite, la longueur du datagramme IP est elle fixée par le logiciel et sera donc la même sur différents réseaux physiques. Nous avons vu qu'en descendant de la couche réseau dans la couche physique le datagramme IP était encapsulé dans une trame physique. Nous avons donné l'exemple de la trame physique d'un réseau Ethernet :



Les trames physiques circulent de noeud en noeud vers leur destination qui peut ne pas être sur le même réseau physique que la machine expéditrice. Le paquet IP peut donc être encapsulé successivement dans des trames physiques différentes au niveau des noeuds qui font la jonction entre deux réseaux de type différent. Il se peut aussi que le paquet IP soit trop grand pour être encapsulé dans une trame physique. Le logiciel IP du noeud où se pose ce problème, décompose alors le paquet IP en *fragments* selon des règles précises, chacun d'eux étant ensuite envoyé sur le réseau physique. Ils ne seront réassemblés qu'à leur ultime destination.

7.1.6.1 Le routage

Le routage est la méthode d'acheminement des paquets IP à leur destination. Il y a deux méthodes : le routage direct et le routage indirect.

Routage direct

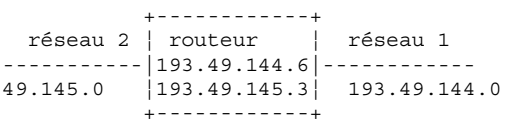
Le routage direct désigne l'acheminement d'un paquet IP directement de l'expéditeur au destinataire à l'intérieur du même réseau :

. La machine expéditrice d'un datagramme IP a l'adresse IP du destinataire.
. Elle obtient l'adresse physique de ce dernier par le protocole ARP ou dans ses tables, si cette adresse a déjà été obtenue.
. Elle envoie le paquet sur le réseau à cette adresse physique.

Routage indirect

Le routage indirect désigne l'acheminement d'un paquet IP à une destination se trouvant sur un autre réseau que celui auquel appartient l'expéditeur. Dans ce cas, les parties adresse réseau des adresses IP des machines source et destination sont différentes.

La machine source reconnaît ce point. Elle envoie alors le paquet à un noeud spécial appelé **routeur** (*router*), noeud qui connecte un réseau local aux autres réseaux et dont elle trouve l'adresse IP dans ses tables, adresse obtenue initialement soit dans un fichier soit dans une mémoire permanente ou encore via des informations circulant sur le réseau.
Un **routeur** est attaché à deux réseaux et possède une adresse IP à l'intérieur de ces deux réseaux.



Dans notre exemple ci-dessus :

. Le réseau n° 1 a l'adresse Internet 193.49.144.0 et le réseau n° 2 l'adresse 193.49.145.0.
. A l'intérieur du réseau n° 1, le routeur a l'adresse 193.49.144.6 et l'adresse 193.49.145.3 à l'intérieur du réseau n° 2.

Le routeur a pour rôle de mettre le paquet IP qu'il reçoit et qui est contenu dans une trame physique typique du réseau n° 1, dans une trame physique pouvant circuler sur le réseau n° 2. Si l'adresse IP du destinataire du paquet est dans le réseau n° 2, le routeur lui enverra le paquet directement sinon il l'enverra à un autre routeur, connectant le réseau n° 2 à un réseau n° 3 et ainsi de suite.

7.1.6.2 Messages d'erreur et de contrôle

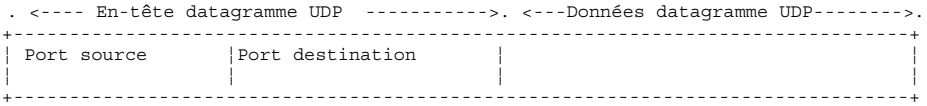
Toujours dans la couche réseau, au même niveau donc que le protocole IP, existe le protocole **ICMP** (*Internet Control Message Protocol*). Il sert à envoyer des messages sur le fonctionnement interne du réseau : noeuds en panne, embouteillage à un routeur, etc ... Les messages ICMP sont encapsulés dans des paquets IP et envoyés sur le réseau. Les couches IP des différents noeuds prennent les actions appropriées selon les messages ICMP qu'elles reçoivent. Ainsi, une application elle-même, ne voit jamais ces problèmes propres au réseau.
Un noeud utilisera les informations ICMP pour mettre à jour ses tables de routage.

7.1.7 La couche transport : les protocoles UDP et TCP

7.1.7.1 Le protocole UDP : User Datagram Protocol

Le protocole UDP permet un échange non fiable de données entre deux points, c'est à dire que le bon acheminement d'un paquet à sa destination n'est pas garanti. L'application, si elle le souhaite peut gérer cela elle-même, en attendant par exemple après l'envoi d'un message, un accusé de réception, avant d'envoyer le suivant.
Pour l'instant, au niveau réseau, nous avons parlé d'adresses IP de machines. Or sur une machine, peuvent coexister en même temps différents processus qui tous peuvent communiquer. Il faut donc indiquer, lors de l'envoi d'un message, non seulement l'adresse IP de la machine destinatrice, mais également le "nom" du processus destinataire. Ce nom est en fait un numéro, appelé **numéro de port**. Certains numéros sont réservés à des applications standard : port 69 pour l'application **tftp** *(trivial file transfer protocol)* par exemple.

Les paquets gérés par le protocole UDP sont appelés également des **datagrammes**. Ils ont la forme suivante :



Ces datagrammes seront encapsulés dans des paquets IP, puis dans des trames physiques.

7.1.7.2 Le protocole TCP : Transfer Control Protocol

Pour des communications sûres, le protocole UDP est insuffisant : le développeur d'applications doit élaborer lui-même un protocole lui permettant de détecter le bon acheminement des paquets.
Le protocole **TCP** *(Transfer Control Protocol)* évite ces problèmes. Ses caractéristiques sont les suivantes :

. Le processus qui souhaite émettre établit tout d'abord une **connexion** avec le processus destinataire des informations qu'il va émettre. Cette connexion se fait entre un port de la machine émettrice et un port de la machine réceptrice. Il y a entre les deux ports un chemin virtuel qui est ainsi créé et qui sera réservé aux deux seuls processus ayant réalisé la connexion.
. Tous les paquets émis par le processus source suivent ce chemin virtuel et arrivent dans l'ordre où ils ont été émis ce qui n'était pas garanti dans le protocole UDP puisque les paquets pouvaient suivre des chemins différents.
. L'information émise a un aspect continu. Le processus émetteur envoie des informations à son rhythme. Celles-ci ne sont pas nécessairement envoyées tout de suite : le protocole TCP attend d'en avoir assez pour les envoyer. Elles sont stockées dans une structure appelée *segment TCP*. Ce segment une fois rempli sera transmis à la couche IP où il sera encapsulé dans un paquet IP.
. Chaque segment envoyé par le protocole TCP est numéroté. Le protocole TCP destinataire vérifie qu'il reçoit bien les segments en séquence. Pour chaque segment correctement reçu, il envoie un accusé de réception à l'expéditeur.
. Lorsque ce dernier le reçoit, il l'indique au processus émetteur. Celui-ci peut donc savoir qu'un segment est arrivé à bon port, ce qui n'était pas possible avec le protocole UDP.
. Si au bout d'un certain temps, le protocole TCP ayant émis un segment ne reçoit pas d'accusé de réception, il retransmet le segment en question, garantissant ainsi la qualité du service d'acheminement de l'information.
. Le circuit virtuel établi entre les deux processus qui communiquent est *full-duplex* : cela signifie que l'information peut transiter dans les deux sens. Ainsi le processus destination peut envoyer des accusés de réception alors même que le processus source continue d'envoyer des informations. Cela permet par exemple au protocole TCP source d'envoyer plusieurs segments sans attendre d'accusé de réception. S'il réalise au bout d'un certain temps qu'il n'a pas reçu l'accusé de réception d'un certain segment n° *n*, il reprendra l'émission des segments à ce point.

7.1.8 La couche Applications

Au-dessus des protocoles UDP et TCP, existent divers protocoles standard :

TELNET

Ce protocole permet à un utilisateur d'une machine A du réseau de se connecter sur une machine B (appelée souvent machine hôte). TELNET émule sur la machine A un terminal dit universel. L'utilisateur se comporte donc comme s'il disposait d'un terminal connecté à la machine B. Telnet s'appuie sur le protocole TCP.

FTP : (File Transfer protocol)

Ce protocole permet l'échange de fichiers entre deux machines distantes ainsi que des manipulations de fichiers tels que des créations de répertoire par exemple. Il s'appuie sur le protocole TCP.

TFTP: (Trivial File Transfer Control)

Ce protocole est une variante de FTP. Il s'appuie sur le protocole UDP et est moins sophistiqué que FTP.

DNS : (Domain Name System)

Lorsqu'un utilisateur désire échanger des fichiers avec une machine distante, par FTP par exemple, il doit connaître l'adresse Internet de cette machine. Par exemple, pour faire du FTP sur la machine Lagaffe de l'université d'Angers, il faudrait lancer FTP comme suit : FTP 193.49.144.1

Cela oblige à avoir un annuaire faisant la correspondance machine <--> adresse IP. Probablement que dans cet annuaire les machines seraient désignées par des noms symboliques tels que :

machine DPX2/320 de l'université d'Angers
machine Sun de l'ISERPA d'Angers

On voit bien qu'il serait plus agréable de désigner une machine par un nom plutôt que par son adresse IP. Se pose alors le problème de l'unicité du nom : il y a des millions de machines interconnectées. On pourrait imaginer qu'un organisme centralisé attribue les noms. Ce serait sans doute assez lourd. Le contrôle des noms a été en fait distribué dans des **domaines**. Chaque domaine est géré par un organisme généralement très léger qui a toute liberté quant au choix des noms de machines. Ainsi les machines en France appartiennent au domaine **fr**, domaine géré par l'Inria de Paris. Pour continuer à simplifier les choses, on distribue encore le contrôle : des domaines sont créés à l'intérieur du domaine **fr**. Ainsi l'université d'Angers appartient au domaine **univ-Angers**. Le service gérant ce domaine a toute liberté pour nommer les machines du réseau de l'Université d'Angers. Pour l'instant ce domaine n'a pas été subdivisé. Mais dans une grande université comportant beaucoup de machines en réseau, il pourrait l'être.

La machine DPX2/320 de l'université d'Angers a été nommée *Lagaffe* alors qu'un PC 486DX50 a été nommé *liny*. Comment référencer ces machines de l'extérieur ? En précisant la hiérarchie des domaines auxquelles elles appartiennent. Ainsi le nom complet de la machine Lagaffe sera :

Lagaffe.univ-Angers.fr

A l'intérieur des domaines, on peut utiliser des noms relatifs. Ainsi à l'intérieur du domaine **fr** et en dehors du domaine **univ-Angers**, la machine Lagaffe pourra être référencée par

Lagaffe.univ-Angers

Enfin, à l'intérieur du domaine *univ-Angers*, elle pourra être référencée simplement par

Lagaffe

Une application peut donc référencer une machine par son nom. Au bout du compte, il faut quand même obtenir l'adresse Internet de cette machine. Comment cela est-il réalisé ? Suposons que d'une machine A, on veuille communiquer avec une machine B.

. si la machine B appartient au même domaine que la machine A, on trouvera probablement son adresse IP dans un fichier de la machine A.
. sinon, la machine A trouvera dans un autre fichier ou le même que précédemment, une liste de quelques **serveurs de noms** avec leurs adresses IP. Un *serveur de noms* est chargé de faire la correspondance entre un nom de machine et son adresse IP. La machine A va envoyer une requête spéciale au premier serveur de nom de sa liste, appelé requête DNS incluant donc le nom de la machine recherchée. Si le serveur interrogé a ce nom dans ses tablettes, il enverra à la machine A, l'adresse IP correspondante. Sinon, le serveur trouvera lui aussi dans ses fichiers, une liste de serveurs de noms qu'il peut interroger. Il le fera alors. Ainsi un certain nombre de serveurs de noms vont être interrogés, pas de façon anarchique mais d'une façon à minimiser les requêtes. Si la machine est finalement trouvée, la réponse redescendra jusqu'à la machine A.

XDR : (eXternal Data Representation)

Créé par sun MicroSystems, ce protocole spécifie une représentation standard des données, indépendante des machines.

RPC : (Remote Procedure Call)

Défini également par sun, c'est un protocole de communication entre applications distantes, indépendant de la couche transport. Ce protocole est important : il décharge le programmeur de la connaissance des détails de la couche transport et rend les applications portables. Ce protocole s'appuie sur sur le protocole XDR

NFS : Network File System

Toujours défini par Sun, ce protocole permet à une machine, de "voir" le système de fichiers d'une autre machine. Il s'appuie sur le protocole RPC précédent.

7.1.9 Conclusion

Nous avons présenté dans cette introduction quelques grandes lignes des protocoles Internet. Pour approfondir ce domaine, on pourra lire l'excellent livre de Douglas Comer :

Titre	TCP/IP : Architecture, Protocoles, Applications.
Auteur	Douglas COMER
Editeur	InterEditions

7.2 Gestion des adresses réseau en Java

7.2.1 Définition

Chaque machine de l'Internet est identifiée par une adresse ou un nom uniques. Ces deux entités sont gérées sous Java par la classe **InetAddress** dont voici quelque méthodes :

`byte [] getAddress()` donne les 4 octets de l'adresse IP de l'instance InetAddress courante

`String getHostAddress()`	donne l'adresse IP de l'instance InetAddress courante
`String getHostName()`	donne le nom Internet de l'instance InetAddress courante
`String toString()`	donne l'identité adresse IP/ nom internet de l'instance InetAddress courante
`InetAddress getByName(String Host)`	crée l'instance InetAddress de la machine désignée par Host. Génère une exception si Host est inconnu. Host peut être le nom internet d'une machine ou son adresse IP sous la forme I1.I2.I3.I4
`InetAddress getLocalHost()`	crée l'instance InetAddress de la machine sur laquelle s'exécute le programme contenant cette instruction.

7.2.2 Quelques exemples

7.2.2.1 Identifier la machine locale

```
import java.net.*;

public class localhost{
  public static void main (String arg[]){
    try{
      InetAddress adresse=InetAddress.getLocalHost();
  byte[] IP=adresse.getAddress();
  System.out.print("IP=");
  int i;
  for(i=0;i<IP.length-1;i++) System.out.print(IP[i]+".");
  System.out.println(IP[i]);
      System.out.println("adresse="+adresse.getHostAddress());
  System.out.println("nom="+adresse.getHostName());
  System.out.println("identité="+adresse);
    } catch (UnknownHostException e){
      System.out.println ("Erreur getLocalHost : "+e);
    }// fin try
  }// fin main
}// fin class
```

Les résultats de l'exécution sont les suivants :

```
IP=127.0.0.1
adresse=127.0.0.1
nom=tahe
identité=tahe/127.0.0.1
```

Chaque machine a une adresse IP interne qui est 127.0.0.1. Lorsqu'un programme utilise cette adresse réseau, il utilise la machine sur laquelle il fonctionne. L'intérêt de cette adresse est qu'elle ne nécessite pas de carte réseau. On peut donc tester des programmes réseau sans être connecté à un réseau. Une autre façon de désigner la machine locale est d'utiliser le nom **localhost**.

7.2.2.2 Identifier une machine quelconque

```
import java.net.*;

public class getbyname{
  public static void main (String arg[]){
    String nomMachine;
    // on récupère l'argument
    if(arg.length==0)
      nomMachine="localhost";
    else nomMachine=arg[0];
    // on tente d'obtenir l'adresse de la machine
    try{
      InetAddress adresse=InetAddress.getByName(nomMachine);
      System.out.println("IP : "+  adresse.getHostAddress());
      System.out.println("nom : "+ adresse.getHostName());
      System.out.println("identité : "+ adresse);
    } catch (UnknownHostException e){
      System.out.println ("Erreur getByName : "+e);
    }// fin try
  }// fin main
}// fin class
```

Avec l'appel **java getbyname**, on obtient les résultats suivants :

```
IP : 127.0.0.1
```

```
nom : localhost
identité : localhost/127.0.0.1
```

Avec l'appel **java getbyname shiva.istia.univ-angers.fr**, on obtient :

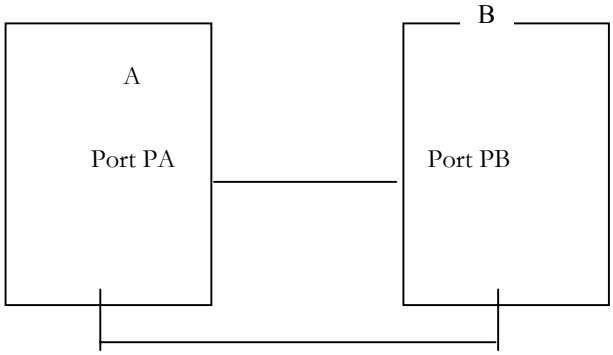
```
IP : 193.52.43.5
nom : shiva.istia.univ-angers.fr
identité : shiva.istia.univ-angers.fr/193.52.43.5
```

Avec l'appel **java getbyname www.ibm.com**, on obtient :

```
IP : 204.146.18.33
nom : www.ibm.com
identité : www.ibm.com/204.146.18.33
```

7.3 Communications TCP-IP

7.3.1 Généralités



Réseau physique

Lorsque une application *AppA* d'une machine A veut communiquer avec une application *AppB* d'une machine B de l'Internet, elle doit connaître plusieurs choses :

- l'adresse IP ou le nom de la machine B
- le numéro du port avec lequel travaille l'application *AppB*. En effet la machine B peut supporter de nombreuses applications qui travaillent sur l'Internet. Lorsqu'elle reçoit des informations provenant du réseau, elle doit savoir à quelle application sont destinées ces informations. Les applications de la machine B ont accès au réseau via des guichets appelés également des ports de communication. Cette information est contenue dans le paquet reçu par la machine B afin qu'il soit délivré à la bonne application.
- les protocoles de communication compris par la machine B. Dans notre étude, nous utiliserons uniquement les protocoles TCP-IP.
- le protocole de dialogue accepté par l'application *AppB*. En effet, les machines A et B vont se "parler". Ce qu'elles vont dire va être encapsulé dans les protocoles TCP-IP. Néammoins, lorsqu'au bout de la chaîne, l'application *AppB* va recevoir l'information envoyée par l'applicaton *AppA*, il faut qu'elle soit capable de l'interpréter. Ceci est analogue à la situation où deux personnes A et B communiquent par téléphone : leur dialogue est transporté par le téléphone. La parole va être codée sous forme de signaux par le téléphone A, transportée par des lignes téléphoniques, arrivée au téléphone B pour y être décodée. La personne B entend alors des paroles. C'est là qu'intervient la notion de protocole de dialogue : si A parle français et que B ne comprend pas cette langue, A et B ne pourront dialoguer utilement.

  Aussi les deux applications communicantes doivent -elles être d'accord sur le type de dialogue qu'elles vont adopter. Ainsi par exemple, le dialogue avec un service *ftp* n'est pas le même qu'avec un service *pop* : ces deux services n'acceptent pas les mêmes commandes. Elles ont un protocole de dialogue différent.

7.3.2 Les caractéristiques du protocole TCP

Nous n'étudierons ici que des communications réseau utilisant le protocole de transport TCP. Rappelons ici, les caractéristiques de ce protocole :

- Le processus qui souhaite émettre établit tout d'abord une **connexion** avec le processus destinataire des informations qu'il va émettre. Cette connexion se fait entre un port de la machine émettrice et un port de la machine réceptrice. Il y a entre les deux ports un chemin virtuel qui est ainsi créé et qui sera réservé aux deux seuls processus ayant réalisé la connexion.
- Tous les paquets émis par le processus source suivent ce chemin virtuel et arrivent dans l'ordre où ils ont été émis
- L'information émise a un aspect continu. Le processus émetteur envoie des informations à son rythme. Celles-ci ne sont pas nécessairement envoyées tout de suite : le protocole TCP attend d'en avoir assez pour les envoyer. Elles sont stockées dans une structure appelée *segment TCP*. Ce segment une fois rempli sera transmis à la couche IP où il sera encapsulé dans un paquet IP.
- Chaque segment envoyé par le protocole TCP est numéroté. Le protocole TCP destinataire vérifie qu'il reçoit bien les segments en séquence. Pour chaque segment correctement reçu, il envoie un accusé de réception à l'expéditeur.
- Lorsque ce dernier le reçoit, il l'indique au processus émetteur. Celui-ci peut donc savoir qu'un segment est arrivé à bon port.
- Si au bout d'un certain temps, le protocole TCP ayant émis un segment ne reçoit pas d'accusé de réception, il retransmet le segment en question, garantissant ainsi la qualité du service d'acheminement de l'information.
- Le circuit virtuel établi entre les deux processus qui communiquent est *full-duplex* : cela signifie que l'information peut transiter dans les deux sens. Ainsi le processus destination peut envoyer des accusés de réception alors même que le processus source continue d'envoyer des informations. Cela permet par exemple au protocole TCP source d'envoyer plusieurs segments sans attendre d'accusé de réception. S'il réalise au bout d'un certain temps qu'il n'a pas reçu l'accusé de réception d'un certain segment n° *n*, il reprendra l'émission des segments à ce point.

7.3.3 La relation client-serveur

Souvent, la communication sur Internet est dissymétrique : la machine A initie une connexion pour demander un service à la machine B : il précise qu'il veut ouvrir une connexion avec le service SB1 de la machine B. Celle-ci accepte ou refuse. Si elle accepte, la machine A peut envoyer ses demandes au service SB1. Celles-ci doivent se conformer au protocole de dialogue compris par le service SB1. Un dialogue demande-réponse s'instaure ainsi entre la machine A qu'on appelle machine **cliente** et la machine B qu'on appelle machine **serveur**. L'un des deux partenaires fermera la connexion.

7.3.4 Architecture d'un client

L'architecture d'un programme réseau demandant les services d'une application serveur sera la suivante :

```
ouvrir la connexion avec le service SB1 de la machine B
si réussite alors
    tant que ce n'est pas fini
      préparer une demande
      l'émettre vers la machine B
      attendre et récupérer la réponse
      la traiter
    fin tant que
finsi
```

7.3.5 Architecture d'un serveur

L'architecture d'un programme offrant des services sera la suivante :

```
ouvrir le service sur la machine locale
tant que le service est ouvert
    se mettre à l'écoute des demandes de connexion sur un port dit port d'écoute
    lorsqu'il y a une demande, la faire traiter par une autre tâche sur un autre port dit port de service
fin tant que
```

Le programme serveur traite différemment la demande de connexion initiale d'un client de ses demandes ultérieures visant à obtenir un service. Le programme n'assure pas le service lui-même. S'il le faisait, pendant la durée du service il ne serait plus à l'écoute des demandes de connexion et des clients ne seraient alors pas servis. Il procède donc autrement : dès qu'une demande de connexion est reçue sur le port d'écoute puis acceptée, le serveur crée une tâche chargée de rendre le service demandé par le client. Ce service est rendu sur un autre port de la machine serveur appelé **port de service**. On peut ainsi servir plusieurs clients en même temps.

Une tâche de service aura la structure suivante :

```
tant que le service n'a pas été rendu totalement
   attendre une demande sur le port de service
   lorsqu'il y en a une, élaborer la réponse
   transmettre la réponse via le port de service
fin tant que
libérer le port de service
```

7.3.6 La classe Socket

7.3.6.1 Définition

L'outil de base utilisé par les programmes communiquant sur Internet est la *socket*. Ce mot anglais signifie "prise de courant". Il est étendu ici pour signifier "prise de réseau". Pour qu'une application puisse envoyer et recevoir des informations sur le réseau Internet, il lui faut une prise de réseau, une *socket*. Cet outil a été initialement créé dans les versions d'Unix de l'université de Berkeley. Il a été porté depuis sur tous les systèmes Unix ainsi que dans le monde Windows. Il existe également sur les machines virtuelles Java sous deux formes : la classe *Socket* pour les applications clientes et la classe *ServerSocket* pour les applications serveur. Nous explicitons ici quelques-uns des constructeurs et méthodes de la classe *Socket* :

public Socket(String host, int port)	ouvre une connexion distante avec le port *port* de la machine *host*
`public int getLocalPort()`	rend le n° du port local utilisé par la socket
`public int getPort()`	rend le n° du port distant auquel la socket est connectée
`public InetAddress getLocalAdress()`	rend l'adresse InetAddress locale à laquelle la socket est liée
`public InetAddress getInetAdress()`	rend l'adresse InetAddress distante à laquelle la socket est liée
`public InputStream getInputStream()`	rend un flux d'entrée permettant de lire les données envoyées par le partenaire distant
`public OutputStream getOutputStream()`	rend un flux de sortie permettant d'envoyer des données au partenaire distant
`public void shutdownInput()`	ferme le flux d'entrée de la socket
`public void shutdownOutput()`	ferme le flux de sortie de la socket
`public void close()`	ferme la socket et ses flux d'E/S
`public String toString()`	rend une chaîne de caractères "représentant" la socket

7.3.6.2 Ouverture d'une connexion avec une machine Serveur

Nous avons vu que pour qu'une machine A ouvre une connexion avec un service d'une machine B, il lui fallait deux informations :

- l'adresse IP ou le nom de la machine B
- le numéro de port où officie le service désiré

Le constructeur

```
  public Socket(String  host, int  port);
```

crée une socket et la connecte à la machine *host* sur le port *port*. Ce constructeur génère une exception dans différents cas :

- mauvaise adresse
- mauvais port
- demande refusée
- …

Il nous faut gérer cette exception :

```
  Socket  sClient=null;
  try{
    sClient=new Socket(host,port);
  } catch(Exception e){
    // la connexion a échoué - on traite l'erreur
    ….
  }
```

Si la demande de connexion réussit, le client se voit localement attribuer un port pour communiquer avec la machine B. Une fois la connexion établie, on peut connaître ce port avec la méthode :

```
public int getLocalPort();
```

Si la connexion réussit, nous avons vu que, de son côté, le serveur fait assurer le service par une autre tâche travaillant sur un port dit de service. Ce numéro de port peut être connu avec la méthode :

```
public int getPort();
```

7.3.6.3 Envoyer des informations sur le réseau

On peut obtenir un flux d'écriture sur la socket et donc sur le réseau avec la méthode :

```
public OutputStream getOutputStream();
```

Tout ce qui sera envoyé dans ce flux sera reçu sur le port de service de la machine serveur. De nombreuses applications ont un dialogue sous forme de lignes de texte terminées par un passage à a ligne. Aussi la méthode *println* est-elle bien pratique dans ces cas là. On transforme alors le flux de sortie *OutputStream* en flux *PrintWriter* qui possède la méthode *println*. L'écriture peut générer une exception.

7.3.6.4 Lire des informations venant du réseau

On peut obtenir un flux de lecture des informations arrivant sur la socket avec la méthode :

```
public InputStream getInputStream();
```

Tout ce qui sera lu dans ce flux vient du port de service de la machine serveur. Pour les applications ayant un dialogue sous forme de lignes de texte terminées par un passage à la ligne on aimera utiliser la méthode *readLine*. Pour cela on transforme le flux d'entrée In*putStream* en flux *BufferedReader* qui possède la méthode *readLine()*. La lecture peut générer une exception.

7.3.6.5 Fermeture de la connexion

Elle se fait avec la méthode :

```
public void close();
```

La méthode peut générer une exception. Les ressources utilisées, notamment le port réseau, sont libérées.

7.3.6.6 L'architecture du client

Nous avons maintenant les éléments pour décrire l'architecture de base d'un client internet :

```
Socket  sClient=null;
try{
    // on se connecte au service officiant sur le port P de la machine M
  sClient=new Socket(M,P);

  // on crée les flux d'entrée-sortie de la socket client
  BufferedReader in=new BufferedReader(new InputStreamReader(sClient.getInputStream()));
  PrintWriter out=new PrintWriter(sClient.getOutputStream(),true);

  // boucle demande - réponse
  boolean  fini=false;
  String demande;
  String réponse;
  while (! fini){
    // on prépare la demande
    demande=…
    // on l'envoie
    out.println(demande);
    // on lit la réponse
    réponse=in.readLine();
    // on traite la réponse
    …
```

```
    }
    // c'est fini
    sClient.close();
  } catch(Exception e){
    // on gère l'exception
    ….
  }
```

Nous n'avons pas cherché à gérer les différents types d'exception générés par le constructeur *Socket* ou les méthodes *readline, getInputStream, getOutputStream, close* pour ne pas compliquer l'exemple. Tout a été réuni dans une seule exception.

7.3.7 La classe ServerSocket

7.3.7.1 Définition

Cette classe est destinée à la gestion des sockets **coté serveur**. Nous explicitons ici quelques-uns des constructeurs et méthodes de cette classe :

`public ServerSocket(int port)`	crée une socket d'écoutesur le port *port*
`public ServerSocket(int port, int count)`	idem mais fixe à *count* la taille de la file d'attente, c.a.d. le nombre maximal de connexions clientes mises en attente si le serveur est occupé lorsque la connexion cliente arrive.
`public int getLocalPort()`	rend le n° du port d'écoute utilisé par la socket
`public InetAddress getInetAdress()`	rend l'adresse InetAddress locale à laquelle la socket est liée
`public Socket accept()`	met le serveur en attente d'une connexion (opération bloquante). A l'arrivée d'une connexion cliente, rend une socket à partir de laquelle sera rendu le service au client.
`public void close()`	ferme la socket et ses flux d'E/S
`public String toString()`	rend une chaîne de caractères "représentant" la socket
`public void close()`	ferme la socket de service et libère les ressources qui lui sont associées

7.3.7.2 Ouverture du service

Elle se fait avec les deux constructeurs :

```
public ServerSocket(int  port);
public ServerSocket(int  port, int  count);
```

port est le port d'écoute du service : celui où les clients adressent leurs demandes de connexion. *count* est la taille maximale de la file d'attente du service (50 par défaut), celle-ci stockant les demandes de connexion des clients auxquelles le serveur n'a pas encore répondu. Lorsque la file d'attente est pleine, les demandes de connexion qui arrivent sont rejetées. Les deux constructeurs génèrent une exception.

7.3.7.3 Acceptation d'une demande de connexion

Lorsqu'un client fait une demande de connexion sur le port d'écoute du service, celui-ci l'accepte avec la méthode :

```
  public Socket accept();
```

Cette méthode rend une instance de *Socket* : c'est la socket de service, celle à travers laquelle le service sera rendu, le plus souvent par une autre tâche. La méthode peut générer une exception.

7.3.7.4 Lecture/Ecriture via la socket de service

La socket de service étant une instance de la classe *Socket*, on se reportera aux sections précédentes où ce sujet a été traité.

7.3.7.5 Identifier le client

Une fois la socket de service obtenue, le client peut être identifié avec la méthode

```
  public InetAddress getInetAddress()
```

de la classe *Socket*. On aura alors accès à l'adresse IP et au nom du client.

7.3.7.6 Fermer le service

Cela se fait avec la méthode

```
  public void close();
```

de la classe *ServerSocket*. Cela libère les ressources occupées, notamment le port d'écoute. La méthode peut générer une exception.

7.3.7.7 Architecture de base d'un serveur

De ce qui a été dit, on peut écrire la structure de base d'un serveur :

```
SocketServer sEcoute=null;
try{
  // ouverture du service
  int portEcoute=…
  int maxConnexions=…
  sEcoute=new ServerSocket(portEcoute,maxConnexions);

  // traitement des demandes de connexion
  boolean fini=false;
  Socket sService=null;
  while( ! fini){
    // attente et acceptation d'une demande
    sService=sEcoute.accept();

    // le service est rendu par une autre tâche à laquelle on passe la socket de service
    new Service(sService).start();

    // on se remet en attente des demandes de connexion
  }
  // c'est fini - on clôt le service
  sEcoute.close();
} catch (Exception e){
  // on traite l'exception
  …
}
```

La classe *Service* est un *thread* qui pourrait avoir l'allure suivante :

```
public class Service extends Thread{

  Socket sService;    // la socket de service

  // constructeur
  public Service(Socket S){
    sService=S;
  }

// run
public void run(){
  try{
    // on crée les flux d'entrée-sortie
  BufferedReader in=new BufferedReader(new InputStreamReader(sService.getInputStream()));
  PrinttWriter out=new PrintWriter(sService.getOutputStream(),true);

  // boucle demande - réponse
  boolean  fini=false;
  String demande;
  String réponse;
  while (! fini){
    // on lit la demande
    demande=in.readLine();

    // on la traite
    …

    // on prépare la réponse
    réponse=…

    // on l'envoie
    out.println(réponse);
```

```
  }
  // c'est fini
  sService.close();
  } catch(Exception e){
  // on gère l'exception
  ...
  }// try
} // run
```

7.4 Applications

7.4.1 Serveur d'écho

Nous nous proposons d'écrire un serveur d'écho qui sera lancé depuis une fenêtre DOS par la commande :

java serveurEcho port

Le serveur officie sur le port passé en paramètre. Il se contente de renvoyer au client la demande que celui-ci lui a envoyée accompagnée de son identité (IP+nom). Il accepte 2 connexions dans sa liste d'attente. On a là tous les constituants d'un serveur tcp. Le programme est le suivant :

```
// appel : serveurEcho port
// serveur d'écho
// renvoie au client la ligne que celui-ci lui a envoyée


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

public class serveurEcho{
  public final static String syntaxe="Syntaxe : serveurEcho port";
  public final static int nbConnexions=2;

  // programme principal
  public static void main (String arg[]){

   // y-a-t-il un argument
   if(arg.length != 1)
    erreur(syntaxe,1);

   // cet argument doit être entier >0
   int port=0;
   boolean erreurPort=false;
   Exception E=null;
   try{
    port=Integer.parseInt(arg[0]);
   }catch(Exception e){
      E=e;
      erreurPort=true;
   }
   erreurPort=erreurPort || port <=0;
   if(erreurPort)
    erreur(syntaxe+"\n"+"Port incorrect ("+E+")",2);

   // on crée la socket d'écoute
   ServerSocket ecoute=null;
   try{
    ecoute=new ServerSocket(port,nbConnexions);
   } catch (Exception e){
    erreur("Erreur lors de la création de la socket d'écoute ("+e+")",3);
   }

   // suivi
   System.out.println("Serveur d'écho lancé sur le port " + port);

   // boucle de service
   boolean serviceFini=false;
   Socket service=null;
   while (! serviceFini){
    // attente d'un client
    try{
      service=ecoute.accept();
    } catch (IOException e){
       erreur("Erreur lors de l'acceptation d'une connexion ("+e+")",4);
    }

    // on identifie la liaison
    try{
```

```
      System.out.println("Client ["+identifie(service.getInetAddress())+","+
      service.getPort()+"] connecté au serveur [" + identifie (InetAddress.getLocalHost())
      + "," + service.getLocalPort() + "]");
    } catch (Exception e) {
      erreur("identification liaison",1);
    }


    // le service est assuré par une autre tâche
    new traiteClientEcho(service).start();
   }// fin while
  }// fin main

// affichage des erreurs
  public static void erreur(String msg, int exitCode){
    System.err.println(msg);
    System.exit(exitCode);
  }

  // identifie
  private static String identifie(InetAddress Host){
    // identification de Host
    String ipHost=Host.getHostAddress();
    String nomHost=Host.getHostName();
    String idHost;
    if (nomHost == null) idHost=ipHost;
      else idHost=ipHost+","+nomHost;
    return idHost;
  }

}// fin class


// assure le service à un client du serveur d'écho

class traiteClientEcho extends Thread{

  private Socket service;     // socket de service
  private BufferedReader in;  // flux d'entrée
  private PrintWriter out;    // flux de sortie

  // constructeur
  public traiteClientEcho(Socket service){
    this.service=service;
  }

  // méthode run
  public void run(){

    // création des flux d'entrée et de sortie
    try{
      in=new BufferedReader(new InputStreamReader(service.getInputStream()));
    } catch (IOException e){
        erreur("Erreur lors de la création du flux déentrée de la socket de service ("+e+")",1);
    }// fin try
    try{
      out=new PrintWriter(service.getOutputStream(),true);
    } catch (IOException e){
        erreur("Erreur lors de la création du flux de sortie de la socket de service ("+e+")",1);
    }// fin try

    // l'identification de la liaison est envoyée au client
    try{
      out.println("Client ["+identifie(service.getInetAddress())+","+
      service.getPort()+"] connecté au serveur [" + identifie (InetAddress.getLocalHost())
      + "," + service.getLocalPort() + "]");
    } catch (Exception e) {
      erreur("identification liaison",1);
    }

    // boucle lecture demande/écriture réponse
    String demande,reponse;
    try{
      // le service s'arrête lorsque le client envoie une marque de fin de fichier
      while ((demande=in.readLine())!=null){
        // écho de la demande
        reponse="["+demande+"]";
        out.println(reponse);
        // le service s'arrête lorsque le client envoie "fin"
        if(demande.trim().toLowerCase().equals("fin")) break;
      }// fin while
    } catch (IOException e){
        erreur("Erreur lors des échanges client/serveur ("+e+")",3);
    }// fin try

    // on ferme la socket
    try{
```

```
      service.close();
    } catch (IOException e){
      erreur("Erreur lors de la fermeture de la socket de service ("+e+")",2);
    }// fin try
  }// fin run

  // affichage des erreurs
  public static void erreur(String msg, int exitCode){
    System.err.println(msg);
    System.exit(exitCode);
  }// fin erreur

  // identifie
  private String identifie(InetAddress Host){
    // identification de Host
    String ipHost=Host.getHostAddress();
    String nomHost=Host.getHostName();
    String idHost;
    if (nomHost == null) idHost=ipHost;
      else idHost=ipHost+","+nomHost;
    return idHost;
  }

}// fin class
```

Les deux classes nécessaires au service ont été réunies dans un même fichier source. Seule l'une d'entre-elles, celle qui a la fonction *main* a l'attribut *public*. La structure du serveur est conforme à l'architecture générale des serveurs tcp. On y ajouté une méthode (*identifie*) permettant d'identifier la liaison entre le serveur et un client. Voici quelques résultats :

Le serveur est lancé par la commande
java serveurEcho 187

Il affiche alors dans la fenêtre de contrôle, le message suivant :

```
Serveur d'écho lancé sur le port 187
```

Pour tester ce serveur, on utilise le programme *telnet* qui existe à la fois sous Unix et Windows. Telnet est un client tcp universel adapté à tous les serveurs qui acceptent des lignes de texte terminées par une marque de fin de ligne dans leur dialogue. C'est le cas de notre serveur d'écho. On lance un premier client *telnet* sous windows (2000 dans cet exemple) en tapant *telnet* dans une fenêtre DOS :

```
DOS>telnet
Microsoft (R) Windows 2000 (TM) version 5.00 (numéro 2195)
Client Telnet Microsoft
Client Telnet numéro 5.00.99203.1

Le caractère d'échappement est 'CTRL+$'

Microsoft Telnet> help

Les commandes peuvent être abrégées. Les commandes prises en charge sont :

close           ferme la connexion en cours
display         affiche les paramètres d'opération
open            ouvre une connexion à un site
quit            quitte telnet
set             définit les options (entrez 'set ?' pour afficher la liste)
status          affiche les informations d'état
unset           annule les options (entrez 'unset ?' pour afficher la liste)
? ou help       affiche des informations d'aide

Microsoft Telnet> set ?
NTLM            Active l'authentification NTLM.
LOCAL_ECHO      Active l'écho local.
TERM x          (où x est ANSI, VT100, VT52 ou VTNT))
CRLF            Envoi de CR et de LF

Microsoft Telnet> set local_echo

Microsoft Telnet> open localhost 187
```

Le programme *telnet* ne fait, par défaut, pas l'écho des commandes que l'on tape au clavier. Pour avoir cet écho on émet la commande :

```
Microsoft Telnet> set local_echo
```

Pour ouvrir une connexion avec le serveur, en lui précisant le port du service d'écho (*187*) et l'adresse de la machine sur lequel il se trouve (*localhost*) on émet la commande :

```
Microsoft Telnet> open localhost 187
```

Dans la fenêtre Dos du client, on reçoit alors le message :

```
Client [127.0.0.1,tahe,1059] connecté au serveur [127.0.0.1,tahe,187]
```

Dans la fenêtre du serveur, on a le message :

```
Serveur d'écho lancé sur le port 187
Client [127.0.0.1,tahe,1059] connecté au serveur [127.0.0.1,tahe,187]
```

Ici *tahe* et *localhost* désignent la même machine. Dans la fenêtre du client *telnet*, on peut taper des lignes de texte. Le serveur les renvoie en écho :

```
Client [127.0.0.1,tahe,1059] connectÚ au serveur [127.0.0.1,tahe,187]
je suis là
[je suis là]
au revoir
[au revoir]
```

On notera que le port du client (1059) est bien détecté mais que le port de service (187) est identique au port d'écoute (187), ce qui est inattendu. On pouvait en effet s'attendre à obtenir le port de la socket de service et non le port d'écoute. Il faudrait vérifier si on obtient les mêmes résultats sous Unix. Maintenant, lançons un second client *telnet*. La fenêtre du serveur devient :

```
Serveur d'écho lancé sur le port 187
Client [127.0.0.1,tahe,1059] connecté au serveur [127.0.0.1,tahe,187]
Client [127.0.0.1,tahe,1060] connecté au serveur [127.0.0.1,tahe,187]
```

Dans la fenêtre du second client, on peut aussi taper des lignes de texte :

```
Client [127.0.0.1,tahe,1060] connecté au serveur [127.0.0.1,tahe,187]
ligne1
[ligne1]
ligne2
[ligne2]
```

On voit ainsi que le serveur d'écho peut servir plusieurs clients à la fois. Les clients *telnet* peuvent être terminés en fermant la fenêtre Dos dans laquelle ils s'exécutent.

7.4.2 Un client java pour le serveur d'écho

Dans la partie précédente, nous avons utilisé un client *telnet* pour tester le service d'écho. Nous écrivons maintenant notre propre client :

```
// appel : clientEcho machine port
// client du serveur d'écho
// envoie des lignes au serveur qui les lui renvoie en écho

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

public class clientEcho{
  public final static String syntaxe="Syntaxe : clientEcho machine port";

  // programme principal
  public static void main (String arg[]){

   // y-a-t-il deux arguments
   if(arg.length != 2)
    erreur(syntaxe,1);

  // le premier argument doit être le nom d'une machine existante
  String machine=arg[0];
  InetAddress serveurAddress=null;
  try{
```

```
  serveurAddress=InetAddress.getByName(machine);
} catch (Exception e){
  erreur(syntaxe+"\nMachine "+machine+" inaccessible (" + e +")",2);
}

 // le port doit être entier >0
 int port=0;
 boolean erreurPort=false;
 Exception E=null;
 try{
  port=Integer.parseInt(arg[1]);
 }catch(Exception e){
    E=e;
    erreurPort=true;
 }
 erreurPort=erreurPort || port <=0;
 if(erreurPort)
  erreur(syntaxe+"\nPort incorrect ("+E+")",3);

 // on se connecte au serveur
 Socket sClient=null;
 try{
  sClient=new Socket(machine,port);
 } catch (Exception e){
  erreur("Erreur lors de la création de la socket de communication ("+e+")",4);
 }

// on identifie la liaison
try{
  System.out.println("Client : Client ["+identifie(InetAddress.getLocalHost())+","+
  sClient.getLocalPort()+"] connecté au serveur [" + identifie (sClient.getInetAddress())
  + "," + sClient.getPort() + "]");
} catch (Exception e) {
  erreur("identification liaison ("+e+")",5);
}

// création du flux de lecture des lignes tapées au clavier
BufferedReader IN=null;
try{
  IN=new BufferedReader(new InputStreamReader(System.in));
} catch (Exception e){
  erreur("Création du flux d'entrée clavier ("+e+")",6);
}
// création du flux d'entrée associée à la socket client
BufferedReader in=null;
try{
  in=new BufferedReader(new InputStreamReader(sClient.getInputStream()));
} catch (Exception e){
  erreur("Création du flux d'entrée de la socket client("+e+")",7);
}
// création du flux de sortie associée à la socket client
PrintWriter out=null;
try{
  out=new PrintWriter(sClient.getOutputStream(),true);
} catch (Exception e){
  erreur("Création du flux de sortie de la socket ("+e+")",8);
}

// boucle demandes - réponses
boolean serviceFini=false;
String demande=null;
String reponse=null;

// on lit le message envoyé par le serveur juste après la connexion
try{
  reponse=in.readLine();
} catch (IOException e){
    erreur("Lecture réponse ("+e+")",4);
}

// affichage réponse
System.out.println("Serveur : " +reponse);

while (! serviceFini){
  // lecture d'une ligne tapée au clavier
  System.out.print("Client : ");
  try{
    demande=IN.readLine();
  } catch (Exception e){
    erreur("Lecture ligne ("+e+")",9);
  }
  // envoi demande sur le réseau
  try{
    out.println(demande);
  } catch (Exception e){
    erreur("Envoi demande ("+e+")",10);
  }
```

```
    // attente/lecture réponse
    try{
      reponse=in.readLine();
    } catch (IOException e){
       erreur("Lecture réponse ("+e+")",4);
    }
    // affichage réponse
    System.out.println("Serveur : " +reponse);
    // est-ce fini ?
    if(demande.trim().toLowerCase().equals("fin")) serviceFini=true;
  }
  // c'est fini
  try{
    sClient.close();
  } catch(Exception e){
    erreur("Fermeture socket ("+e+")",11);
  }
}// main

// affichage des erreurs
  public static void erreur(String msg, int exitCode){
    System.err.println(msg);
    System.exit(exitCode);
  }

  // identifie
  private static String identifie(InetAddress Host){
    // identification de Host
    String ipHost=Host.getHostAddress();
    String nomHost=Host.getHostName();
    String idHost;
    if (nomHost == null) idHost=ipHost;
      else idHost=ipHost+","+nomHost;
    return idHost;
  }

}// fin class
```

La structure de ce client est conforme à l'architecture générale des clients *tcp*. Ici, on a géré les différentes exceptions possibles, une par une, ce qui alourdit le programme. Voici les résultats obtenus lorsqu'on teste ce client :

```
Client : Client [127.0.0.1,tahe,1045] connecté au serveur [127.0.0.1,localhost,187]
Serveur : Client [127.0.0.1,localhost,1045] connectÚ au serveur [127.0.0.1,tahe,187]
Client : 123
Serveur : [123]
Client : abcd
Serveur : [abcd]
Client : je suis là
Serveur : [je suis là]
Client : fin
Serveur : [fin]
```

Les lignes commençant par *Client* sont les lignes envoyées par le client et celles commençant par *Serveur* sont celles que le serveur a renvoyées en écho.

7.4.3 Un client TCP générique

Beaucoup de services créés à l'origine de l'Internet fonctionnent selon le modèle du serveur d'écho étudié précédemment : les échanges client-serveur se font pas échanges de lignes de texte. Nous allons écrire un client tcp générique qui sera lancé de la façon suivante : **java cltTCPgenerique serveur port**

Ce client TCP se connectera sur le port *port* du serveur *serveur*. Ceci fait, il créera deux threads :

1. un thread chargé de lire des commandes tapées au clavier et de les envoyer au serveur
2. un thread chargé de lire les réponses du serveur et de les afficher à l'écran

Pourquoi deux threads alors que dans l'application précédente ce besoin ne s'était pas fait ressentir ? Dans cette dernière, le protocole du dialogue était connu : le client envoyait une seule ligne et le serveur répondait par une seule ligne. Chaque service a son protocole particulier et on trouve également les situations suivantes :

- le client doit envoyer plusieurs lignes de texte avant d'avoir une réponse
- la réponse d'un serveur peut comporter plusieurs lignes de texte

Aussi la boucle envoi d'une unique ligne au serveur - réception d'une unique ligne envoyée par le serveur ne convient-elle pas toujours. On va donc créer deux boucles dissociées :

- une boucle de lecture des commandes tapées au clavier pour être envoyées au serveur. L'utilisateur signalera la fin des commandes avec le mot clé *fin*.
- une boucle de réception et d'affichage des réponses du serveur. Celle-ci sera une boucle infinie qui ne sera interrompue que par la fermeture du flux réseau par le serveur ou par l'utilisateur au clavier qui tapera la commande *fin*.

Pour avoir ces deux boucles dissociées, il nous faut deux threads indépendants. Montrons un exemple d'excécution où notre client tcp générique se connecte à un service SMTP (SendMail Transfer Protocol). Ce service est responsable de l'acheminement du courrier électronique à leurs destinataires. Il fonctionne sur le port 25 et a un protocole de dialogue de type échanges de lignes de texte.

```
Dos>java clientTCPgenerique istia.univ-angers.fr 25
Commandes :
<-- 220 istia.univ-angers.fr ESMTP Sendmail 8.11.6/8.9.3; Mon, 13 May 2002 08:37:26 +0200
help
<-- 502 5.3.0 Sendmail 8.11.6 -- HELP not implemented
mail from: machin@univ-angers.fr
<-- 250 2.1.0 machin@univ-angers.fr... Sender ok
rcpt to: serge.tahe@istia.univ-angers.fr
<-- 250 2.1.5 serge.tahe@istia.univ-angers.fr... Recipient ok
data
<-- 354 Enter mail, end with "." on a line by itself
Subject: test

ligne1
ligne2
ligne3
.
<-- 250 2.0.0 g4D6bks25951 Message accepted for delivery
quit
<-- 221 2.0.0 istia.univ-angers.fr closing connection
[fin du thread de lecture des réponses du serveur]
fin
[fin du thread d'envoi des commandes au serveur]
```

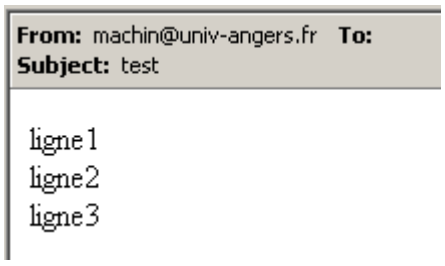
Commentons ces échanges client-serveur :

- le service SMTP envoie un message de bienvenue lorsqu'un client se connecte à lui :

```
<-- 220 istia.univ-angers.fr ESMTP Sendmail 8.11.6/8.9.3; Mon, 13 May 2002 08:37:26 +0200
```

- certains services ont une commande *help* donnant des indications sur les commandes utilisables avec le service. Ici ce n'est pas le cas. Les commandes SMTP utilisées dans l'exemple sont les suivantes :
  - **mail from:** *expéditeur*, pour indiquer l'adresse électronique de l'expéditeur du message
  - **rcpt to:** *destinataire*, pour indiquer l'adresse électronique du destinataire du message. S'il y a plusieurs destinataires, on ré-émet autant de fois que nécessaire la commande **rcpt to:** pour chacun des destinataires.
  - **data** qui signale au serveur SMTP qu'on va envoyer le message. Comme indiqué dans la réponse du serveur, celui-ci est une suite de lignes terminée par une ligne contenant le seul caractère point. Un message peut avoir des entêtes séparés du corps du message par une ligne vide. Dans notre exemple, nous avons mis un sujet avec le mot clé **Subject:**
- une fois le message envoyé, on peut indiquer au serveur qu'on a terminé avec la commande **quit**. Le serveur ferme alors la connexion réseau. Le thread de lecture peut détecter cet événement et s'arrêter.
- l'utilisateur tape alors **fin** au clavier pour arrêter également le thread de lecture des commandes tapées au clavier.

Si on vérifie le courrier reçu, nous avons la chose suivante (Outlook) :



On remarquera que le service SMTP ne peut détecter si un expéditeur est valide ou non. Aussi ne peut-on jamais faire confiance au champ *from* d'un message. Ici l'expéditeur *machin@univ-angers.fr* n'existait pas.

Ce client tcp générique peut nous permettre de découvrir le protocole de dialogue de services internet et à partir de là construire des classes spécialisées pour des clients de ces services. Découvrons le protocole de dialogue du service POP (Post Office Protocol) qui permet de retrouver ses méls stockés sur un serveur. Il travaille sur le port 110.

```
Dos> java clientTCPgenerique istia.univ-angers.fr 110
Commandes :
<-- +OK Qpopper (version 4.0.3) at istia.univ-angers.fr starting.
help
<-- -ERR Unknown command: "help".
user st
<-- +OK Password required for st.
pass monpassword
<-- +OK st has 157 visible messages (0 hidden) in 11755927 octets.
list
<-- +OK 157 visible messages (11755927 octets)
<-- 1 892847
<-- 2 171661
...
<-- 156 2843
<-- 157 2796
<-- .
retr 157
<-- +OK 2796 octets
<-- Received: from lagaffe.univ-angers.fr (lagaffe.univ-angers.fr [193.49.144.1])
<--     by istia.univ-angers.fr (8.11.6/8.9.3) with ESMTP id g4D6wZs26600;
<--     Mon, 13 May 2002 08:58:35 +0200
<-- Received: from jaume ([193.49.146.242])
<--     by lagaffe.univ-angers.fr (8.11.1/8.11.2/GeO20000215) with SMTP id g4D6wSd37691;
<--     Mon, 13 May 2002 08:58:28 +0200 (CEST)
...
<-- ------------------------------------------------------------------------
<-- NOC-RENATER2                  Tl.  : 0800 77 47 95
<-- Fax : (+33) 01 40 78 64 00 ,  Email : noc-r2@cssi.renater.fr
<-- ------------------------------------------------------------------------
<--
<-- .
quit
<-- +OK Pop server at istia.univ-angers.fr signing off.
[fin du thread de lecture des réponses du serveur]
fin
[fin du thread d'envoi des commandes au serveur]
```

Les principales commandes sont les suivantes :

- **user** *login*, où on donne son login sur la machine qui détient nos méls
- **pass** *password*, où on donne le mot de passe associé au login précédent
- **list**, pour avoir la liste des messages sous la forme numéro, taille en octets
- **retr** *i*, pour lire le message n° i
- **quit**, pour arrêter le dialogue.

Découvrons maintenant le protocole de dialogue entre un client et un serveur Web qui lui travaille habituellement sur le port 80 :

```
Dos> java clientTCPgenerique istia.univ-angers.fr 80
Commandes :
GET /index.html HTTP/1.0

<-- HTTP/1.1 200 OK
<-- Date: Mon, 13 May 2002 07:30:58 GMT
<-- Server: Apache/1.3.12 (Unix)  (Red Hat/Linux) PHP/3.0.15 mod_perl/1.21
<-- Last-Modified: Wed, 06 Feb 2002 09:00:58 GMT
<-- ETag: "23432-2bf3-3c60f0ca"
<-- Accept-Ranges: bytes
<-- Content-Length: 11251
<-- Connection: close
<-- Content-Type: text/html
<--
<-- <html>
<--
```

```
<-- <head>
<-- <meta http-equiv="Content-Type"
<-- content="text/html; charset=iso-8859-1">
<-- <meta name="GENERATOR" content="Microsoft FrontPage Express 2.0">
<-- <title>Bienvenue a l'ISTIA - Universite d'Angers</title>
<-- </head>
....
<-- face="Verdana"> - Dernire mise  jour le <b>10 janvier 2002</b></font></p>
<-- </body>
<-- </html>
<--
[fin du thread de lecture des réponses du serveur]
fin
[fin du thread d'envoi des commandes au serveur]
```

Un client Web envoie ses commandes au serveur selon le schéma suivant :

commande1
commande2
...
commanden
[ligne vide]

Ce n'est qu'après avoir reçu la ligne vide que le serveur Web répond. Dans l'exemple nous n'avons utilisé qu'une commande :

```
GET /index.html HTTP/1.0
```

qui demande au serveur l'URL **/index.html** et indique qu'il travaille avec le protocole HTTP version 1.0. La version la plus récente de ce protocole est 1.1. L'exemple montre que le serveur a répondu en renvoyant le contenu du fichier *index.html* puis qu'il a fermé la connexion puisqu'on voit le thread de lecture des réponses se terminer. Avant d'envoyer le contenu du fichier *index.html*, le serveur web a envoyé une série d'entêtes terminée par une ligne vide :

```
<-- HTTP/1.1 200 OK
<-- Date: Mon, 13 May 2002 07:30:58 GMT
<-- Server: Apache/1.3.12 (Unix)  (Red Hat/Linux) PHP/3.0.15 mod_perl/1.21
<-- Last-Modified: Wed, 06 Feb 2002 09:00:58 GMT
<-- ETag: "23432-2bf3-3c60f0ca"
<-- Accept-Ranges: bytes
<-- Content-Length: 11251
<-- Connection: close
<-- Content-Type: text/html
<--
<-- <html>
```

La ligne <*html*> est la première ligne du fichier */index.html*. Ce qui précède s'appelle des entêtes HTTP (HyperText Transfer Protocol). Nous n'allons pas détailler ici ces entêtes mais on se rappellera que notre client générique y donne accès, ce qui peut être utile pour les comprendre. La première ligne par exemple :

```
<-- HTTP/1.1 200 OK
```

indique que le serveur Web contacté comprend le protocole HTTP/1.1 et qu'il a bien trouvé le fichier demandé (200 OK), 200 étant un code de réponse HTTP. Les lignes

```
<-- Content-Length: 11251
<-- Connection: close
<-- Content-Type: text/html
```

disent au client qu'il va recevoir 11251 octets représentant du texte HTML (HyperText Markup Language) et qu'à la fin de l'envoi, la connexion sera fermée.

On a donc là un client tcp très pratique. Il fait sans doute moins que le programme *telnet* que nous avons utilisé précédemment mais il était intéressant de l'écrire nous-mêmes. Le programme du client tcp générique est le suivant :

```
// paquetages importés
import java.io.*;
import java.net.*;

public class clientTCPgenerique{
```

```
  // reçoit en paramètre les caractéristiques d'un service sous la forme
  // serveur port
  // se connecte au service
  // crée un thread pour lire des commandes tapées au clavier
  // celles-ci seront envoyées au serveur
  // crée un thread pour lire les réponses du serveur
  // celles-ci seront affichées à l'écran
  // le tout se termine avec la commande fin tapée au clavier

  // variable d'instance
  private static Socket client;

  public static void main(String[] args){

    // syntaxe
    final String syntaxe="pg serveur port";

    // nombre d'arguments
    if(args.length != 2)
      erreur(syntaxe,1);

    // on note le nom du serveur
    String serveur=args[0];

    // le port doit être entier >0
    int port=0;
    boolean erreurPort=false;
    Exception E=null;
    try{
      port=Integer.parseInt(args[1]);
    }catch(Exception e){
      E=e;
      erreurPort=true;
    }
    erreurPort=erreurPort || port <=0;
    if(erreurPort)
      erreur(syntaxe+"\n"+"Port incorrect ("+E+")",2);

    client=null;
    // il peut y avoir des problèmes
    try{
      // on se connecte au service
      client=new Socket(serveur,port);
    }catch(Exception ex){
      // erreur
      erreur("Impossible de se connecter au service ("+ serveur
        +","+port+"), erreur : "+ex.getMessage(),3);
      // fin
      return;
    }//catch

    // on crée les threads de lecture/écriture
     new ClientSend(client).start();
     new ClientReceive(client).start();

    // fin thread main
    return;
  }// main

  // affichage des erreurs
  public static void erreur(String msg, int exitCode){
    // affichage erreur
    System.err.println(msg);
    // arrêt avec erreur
    System.exit(exitCode);
  }//erreur
}//classe

class ClientSend extends Thread {
  // classe chargée de lire des commandes tapées au clavier
  // et de les envoyer à un serveur via un client tcp passé en paramètre

  private Socket client; // le client tcp

  // constructeur
  public ClientSend(Socket client){
    // on note le client tcp
    this.client=client;
  }//constructeur

  // méthode Run du thread
  public void run(){

    // données locales
    PrintWriter OUT=null;    // flux d'écriture réseau
     BufferedReader IN=null;   // flux clavier
    String commande=null;    // commande lue au clavier
```

```
    // gestion des erreurs
    try{
      // création du flux d'écriture réseau
      OUT=new PrintWriter(client.getOutputStream(),true);
       // création du flux d'entrée clavier
       IN=new BufferedReader(new InputStreamReader(System.in));
      // boucle saisie-envoi des commandes
      System.out.println("Commandes : ");
      while(true){
        // lecture commande tapée au clavier
        commande=IN.readLine().trim();
        // fini ?
        if (commande.toLowerCase().equals("fin")) break;
        // envoi commande au serveur
        OUT.println(commande);
        // commande suivante
      }//while
    }catch(Exception ex){
      // erreur
      System.err.println("Envoi : L'erreur suivante s'est produite : " + ex.getMessage());
    }//catch
    // fin - on ferme les flux
    try{
      OUT.close();client.close();
    }catch(Exception ex){}
    // on signale la fin du thread
    System.out.println("[Envoi : fin du thread d'envoi des commandes au serveur]");
  }//run
}//classe

class ClientReceive extends Thread{
  // classe chargée de lire les lignes de texte destinées à un
  // client tcp passé en paramètre

  private Socket client; // le client tcp

  // constructeur
  public ClientReceive(Socket client){
    // on note le client tcp
    this.client=client;
  }//constructeur

  // méthode Run du thread
  public void run(){

    // données locales
    BufferedReader IN=null;    // flux lecture réseau
    String réponse=null;   // réponse serveur

    // gestion des erreurs
    try{
      // création du flux lecture réseau
      IN=new BufferedReader(new InputStreamReader(client.getInputStream()));
      // boucle lecture lignes de texte du flux IN
      while(true){
        // lecture flux réseau
        réponse=IN.readLine();
        // flux fermé ?
        if(réponse==null) break;
        // affichage
        System.out.println("<-- "+réponse);
      }//while
    }catch(Exception ex){
      // erreur
      System.err.println("Réception : L'erreur suivante s'est produite : " + ex.getMessage());
    }//catch
    // fin - on ferme les flux
    try{
      IN.close();client.close();
    }catch(Exception ex){}
    // on signale la fin du thread
    System.out.println("[Réception : fin du thread de lecture des réponses du serveur]");
  }//run
}//classe
```

7.4.4 Un serveur Tcp générique

Maintenant nous nous intéressons à un serveur

- qui affiche à l'écran les commandes envoyées par ses clients
- leur envoie comme réponse les lignes de texte tapées au clavier par un utilisateur. C'est donc ce dernier qui fait office de serveur.

Le programme est lancé par : **java serveurTCPgenerique portEcoute**, où *portEcoute* est le port sur lequel les clients doivent se connecter. Le service au client sera assuré par deux threads :

- un thread se consacrant exclusivement à la lecture des lignes de texte envoyées par le client
- un thread se consacrant exclusivement à la lecture des réponses tapées au clavier par l'utilisateur. Celui-ci signalera par la commande **fin** qu'il clôt la connexion avec le client.

Le serveur crée deux threads par client. S'il y a n clients, il y aura 2n threads actifs en même temps. Le serveur lui ne s'arrête jamais sauf par un Ctrl-C tapé au clavier par l'utilisateur. Voyons quelques exemples.

Le serveur est lancé sur le port 100 et on utilise le client générique pour lui parler. La fenêtre du client est la suivante :

```
E:\data\serge\MSNET\c#\réseau\client tcp générique> java clientTCPgenerique localhost 100
Commandes :
commande 1 du client 1
<-- réponse 1 au client 1
commande 2 du client 1
<-- réponse 2 au client 1
fin
L'erreur suivante s'est produite : Impossible de lire les données de la connexion de transport.
[fin du thread de lecture des réponses du serveur]
[fin du thread d'envoi des commandes au serveur]
```

Les lignes commençant par <-- sont celles envoyées du serveur au client, les autres celles du client vers le serveur. La fenêtre du serveur est la suivante :

```
Dos> java serveurTCPgenerique 100
Serveur générique lancé sur le port 100
Thread de lecture des réponses du serveur au client 1 lancé
1 : Thread de lecture des demandes du client 1 lancé
<-- commande 1 du client 1
réponse 1 au client 1
1 : <-- commande 2 du client 1
réponse 2 au client 1
1 : [fin du Thread de lecture des demandes du client 1]
fin
[fin du Thread de lecture des réponses du serveur au client 1]
```

Les lignes commençant par <-- sont celles envoyées du client au serveur. Les lignes *N :* sont les lignes envoyées du serveur au client n° N. Le serveur ci-dessus est encore actif alors que le client 1 est terminé. On lance un second client pour le même serveur :

```
Dos> java clientTCPgenerique localhost 100
Commandes :
commande 3 du client 2
<-- réponse 3 au client 2
fin
L'erreur suivante s'est produite : Impossible de lire les données de la connexion de transport.
[fin du thread de lecture des réponses du serveur]
[fin du thread d'envoi des commandes au serveur]
```

La fenêtre du serveur est alors celle-ci :

```
Dos> java serveurTCPgenerique 100
Serveur générique lancé sur le port 100
Thread de lecture des réponses du serveur au client 1 lancé
1 : Thread de lecture des demandes du client 1 lancé
<-- commande 1 du client 1
réponse 1 au client 1
1 : <-- commande 2 du client 1
réponse 2 au client 1
1 : [fin du Thread de lecture des demandes du client 1]
fin
[fin du Thread de lecture des réponses du serveur au client 1]
Thread de lecture des réponses du serveur au client 2 lancé
2 : Thread de lecture des demandes du client 2 lancé
<-- commande 3 du client 2
réponse 3 au client 2
2 : [fin du Thread de lecture des demandes du client 2]
fin
[fin du Thread de lecture des réponses du serveur au client 2]
```

```
^C
```

Simulons maintenant un serveur web en lançant notre serveur générique sur le port 88 :

```
Dos> java serveurTCPgenerique 88
Serveur générique lancé sur le port 88
```

Prenons maintenant un navigateur et demandons l'URL *http://localhost:88/exemple.html*. Le navigateur va alors se connecter sur le port *88* de la machine *localhost* puis demander la page */exemple.html* :

Regardons maintenant la fenêtre de notre serveur :

```
Dos>java serveurTCPgenerique 88
Serveur générique lancé sur le port 88
Thread de lecture des réponses du serveur au client 2 lancé
2 : Thread de lecture des demandes du client 2 lancé
<-- GET /exemple.html HTTP/1.1
<-- Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/msword, */*
<-- Accept-Language: fr
<-- Accept-Encoding: gzip, deflate
<-- User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705; .NET CLR 1.0.2
914)
<-- Host: localhost:88
<-- Connection: Keep-Alive
<--
```

On découvre ainsi les entêtes HTTP envoyés par le navigateur. Cela nous permet de découvrir peu à peu le protocole HTTP. Lors d'un précédent exemple, nous avions créé un client Web qui n'envoyait que la seule commande GET. Cela avait été suffisant. On voit ici que le navigateur envoie d'autres informations au serveur. Elles ont pour but d'indiquer au serveur quel type de client il a en face de lui. On voit aussi que les entêtes HTTP se terminent par une ligne vide.

Elaborons une réponse à notre client. L'utilisateur au clavier est ici le véritable serveur et il peut élaborer une réponse à la main. Rappelons-nous la réponse faite par un serveur Web dans un précédent exemple :

```
<-- HTTP/1.1 200 OK
<-- Date: Mon, 13 May 2002 07:30:58 GMT
<-- Server: Apache/1.3.12 (Unix)  (Red Hat/Linux) PHP/3.0.15 mod_perl/1.21
<-- Last-Modified: Wed, 06 Feb 2002 09:00:58 GMT
<-- ETag: "23432-2bf3-3c60f0ca"
<-- Accept-Ranges: bytes
<-- Content-Length: 11251
<-- Connection: close
<-- Content-Type: text/html
<--
<-- <html>
```

Essayons de donner une réponse analogue :

```
...
<-- Host: localhost:88
<-- Connection: Keep-Alive
<--
2 : HTTP/1.1 200 OK
2 : Server: serveur tcp generique
2 : Connection: close
2 : Content-Type: text/html
2 :
2 : <html>
2 :   <head><title>Serveur generique</title></head>
2 :   <body>
```

```
2 :      <center>
2 :        <h2>Reponse du serveur generique</h2>
2 :      </center>
2 :    </body>
2 : </html>
2 : fin
L'erreur suivante s'est produite : Impossible de lire les données de la connexion de transport.
[fin du Thread de lecture des demandes du client 2]
[fin du Thread de lecture des réponses du serveur au client 2]
```

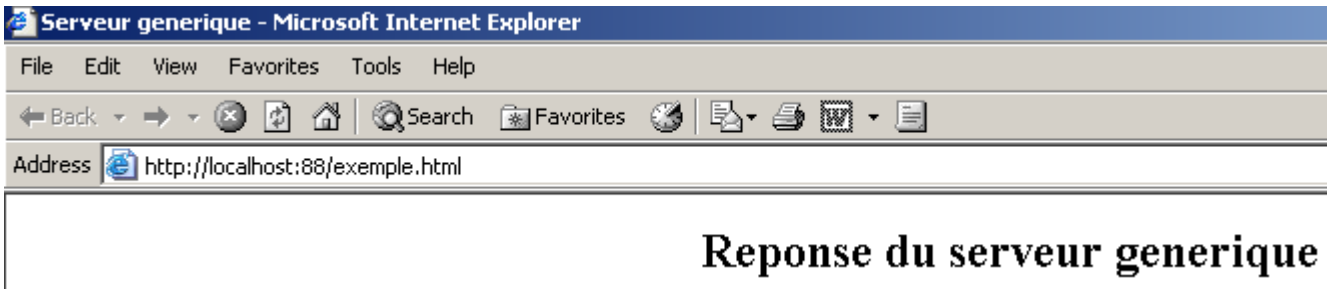
Les lignes commençant par *2 :* sont envoyées du serveur au client n° 2. La commande *fin* clôt la connexion du serveur au client. Nous nous sommes limités dans notre réponse aux entêtes HTTP suivants :

```
HTTP/1.1 200 OK
2 : Server: serveur tcp generique
2 : Connection: close
2 : Content-Type: text/html
2 :
```

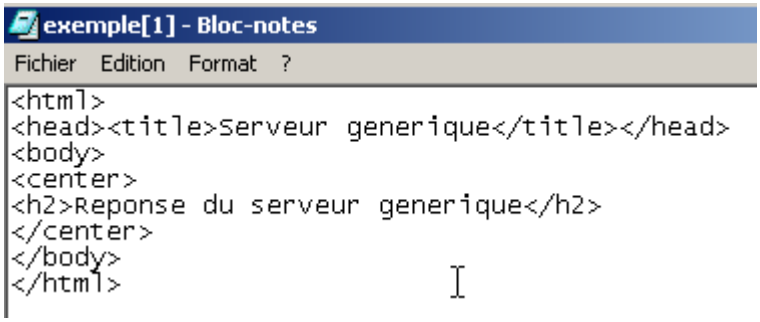
Nous ne donnons pas la taille du fichier que nous allons envoyer (*Content-Length*) mais nous contentons de dire que nous allons fermer la connexion (*Connection: close*) après envoi de celui-ci. Cela est suffisant pour le navigateur. En voyant la connexion fermée, il saura que la réponse du serveur est terminée et affichera la page HTML qui lui a été envoyée. Cette dernière est la suivante :

```
2 : <html>
2 :   <head><title>Serveur generique</title></head>
2 :   <body>
2 :     <center>
2 :       <h2>Reponse du serveur generique</h2>
2 :     </center>
2 :    </body>
2 : </html>
```

L'utilisateur ferme ensuite la connexion au client en tapant la commande *fin*. Le navigateur sait alors que la réponse du serveur est terminée et peut alors l'afficher :



Si ci-dessus, on fait *View/Source* pour voir ce qu'a reçu le navigateur, on obtient :



c'est à dire exactement ce qu'on a envoyé depuis le serveur générique.

Le code du serveur tcp générique est le suivant :

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

```
public class serveurTCPgenerique{

  // programme principal
  public static void main (String[] args){

  // reçoit le port d'écoute des demandes des clients
  // crée un thread pour lire les demandes du client
  // celles-ci seront affichées à l'écran
  // crée un thread pour lire des commandes tapées au clavier
  // celles-ci seront envoyées comme réponse au client
  // le tout se termine avec la commande fin tapée au clavier

  final String syntaxe="Syntaxe : pg port";
  // variable d'instance
     // y-a-t-il un argument
   if(args.length != 1)
    erreur(syntaxe,1);

    // le port doit être entier >0
    int port=0;
    boolean erreurPort=false;
    Exception E=null;
    try{
      port=Integer.parseInt(args[0]);
    }catch(Exception e){
      E=e;
      erreurPort=true;
    }
    erreurPort=erreurPort || port <=0;
    if(erreurPort)
      erreur(syntaxe+"\n"+"Port incorrect ("+E+")",2);

   // on crée le servive d'écoute
  ServerSocket ecoute=null;
  int nbClients=0;  // nbre de clients traités
    try{
      // on crée le service
      ecoute=new ServerSocket(port);
      // suivi
      System.out.println("Serveur générique lancé sur le port " + port);

      // boucle de service aux clients
      Socket client=null;
      while (true){ // boucle infinie - sera arrêtée par Ctrl-C
        // attente d'un client
        client=ecoute.accept();

        // le service est assuré des threads séparés
        nbClients++;

        // on crée les threads de lecture/écriture
         new ServeurSend(client,nbClients).start();
         new ServeurReceive(client,nbClients).start();

        // on retourne à l'écoute des demandes
      }// fin while
    }catch(Exception ex){
      // on signale l'erreur
      erreur("L'erreur suivante s'est produite : " + ex.getMessage(),3);
    }//catch
  }// fin main

  // affichage des erreurs
  public static void erreur(String msg, int exitCode){
    // affichage erreur
    System.err.println(msg);
    // arrêt avec erreur
    System.exit(exitCode);
  }//erreur
}//classe

class ServeurSend extends Thread{
  // classe chargée de lire des réponses tapées au clavier
  // et de les envoyer à un client via un client tcp passé au constructeur

  Socket client;// le client tcp
  int numClient;  // n° de client

  // constructeur
  public ServeurSend(Socket client, int numClient){
    // on note le client tcp
    this.client=client;
    // et son n°
    this.numClient=numClient;
  }//constructeur

  // méthode Run du thread
```

```
  public void run(){

    // données locales
    PrintWriter OUT=null;  // flux d'écriture réseau
    String réponse=null;   // réponse lue au clavier
     BufferedReader IN=null; // flux clavier

    // suivi
    System.out.println("Thread de lecture des réponses du serveur au client "+ numClient + " lancé");
    // gestion des erreurs
    try{
      // création du flux d'écriture réseau
      OUT=new PrintWriter(client.getOutputStream(),true);
       // création du flux clavier
       IN=new BufferedReader(new InputStreamReader(System.in));
      // boucle saisie-envoi des commandes
      while(true){
        // identification client
        System.out.print("--> " + numClient + " : ");
        // lecture réponse tapée au clavier
        réponse=IN.readLine().trim();
        // fini ?
        if (réponse.toLowerCase().equals("fin")) break;
        // envoi réponse au serveur
        OUT.println(réponse);
        // réponse suivante
      }//while
    }catch(Exception ex){
      // erreur
      System.err.println("L'erreur suivante s'est produite : " + ex.getMessage());
    }//catch
    // fin - on ferme les flux
    try{
      OUT.close();client.close();
    }catch(Exception ex){}
    // on signale la fin du thread
    System.out.println("[fin du Thread de lecture des réponses du serveur au client "+ numClient+ "]");
  }//run
}//classe

class ServeurReceive extends Thread{
  // classe chargée de lire les lignes de texte envoyées au serveur
  // via un client tcp passé au constructeur

  Socket client;// le client tcp
  int numClient;  // n° de client

  // constructeur
  public ServeurReceive(Socket client, int numClient){
    // on note le client tcp
    this.client=client;
    // et son n°
    this.numClient=numClient;
  }//constructeur

  // méthode Run du thread
  public void run(){

    // données locales
    BufferedReader IN=null;    // flux lecture réseau
    String réponse=null;   // réponse serveur

    // suivi
    System.out.println("Thread de lecture des demandes du client "+ numClient + " lancé");
    // gestion des erreurs
    try{
      // création du flux lecture réseau
      IN=new BufferedReader(new InputStreamReader(client.getInputStream()));
      // boucle lecture lignes de texte du flux IN
      while(true){
        // lecture flux réseau
        réponse=IN.readLine();
        // flux fermé ?
        if(réponse==null) break;
        // affichage
        System.out.println("<-- "+réponse);
      }//while
    }catch(Exception ex){
      // erreur
      System.err.println("L'erreur suivante s'est produite : " + ex.getMessage());
    }//catch
    // fin - on ferme les flux
    try{
      IN.close();client.close();
    }catch(Exception ex){}
    // on signale la fin du thread
    System.out.println("[fin du Thread de lecture des demandes du client "+ numClient+"]");
```

```
  }//run
}//classe
```

7.4.5 Un client Web

Nous avons vu dans l'exemple précédent, certains des entêtes HTTP qu'envoyait un navigateur :

```
<-- GET /exemple.html HTTP/1.1
<-- Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/msword, */*
<-- Accept-Language: fr
<-- Accept-Encoding: gzip, deflate
<-- User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705; .NET CLR 1.0.2
914)
<-- Host: localhost:88
<-- Connection: Keep-Alive
<--
```

Nous allons écrire un client Web auquel on passerait en paramètre une URL et qui afficherait à l'écran le contenu de cette URL. Nous supposerons que le serveur Web contacté pour l'URL supporte le protocole HTTP 1.1. Des entêtes précédents, nous n'utiliserons que les suivants :

```
<-- GET /exemple.html HTTP/1.1
<-- Host: localhost:88
<-- Connection: close
```

- le premier entête indique quelle page nous désirons
- le second quel serveur nous interrogeons
- le troisième que nous souhaitons que le serveur ferme la connexion après nous avoir répondu.

Si ci-dessus, nous remplaçons GET par HEAD, le serveur ne nous enverra que les entêtes HTTP et pas la page HTML.

Notre client web sera appelé de la façon suivante : **java clientweb URL cmd**, où **URL** est l'URL désirée et **cmd** l'un des deux mots clés GET ou HEAD pour indiquer si on souhaite seulement les entêtes (HEAD) ou également le contenu de la page (GET). Regardons un premier exemple. Nous lançons le serveur IIS puis le client web sur la même machine :

```
dos>java clientweb http://localhost HEAD
HTTP/1.1 302 Object moved
Server: Microsoft-IIS/5.0
Date: Mon, 13 May 2002 09:23:37 GMT
Connection: close
Location: /IISSamples/Default/welcome.htm
Content-Length: 189
Content-Type: text/html
Set-Cookie: ASPSESSIONIDGQQQGUUY=HMFNCCMDECBJJBPPBHAOAJNP; path=/
Cache-control: private
```

La réponse

```
HTTP/1.1 302 Object moved
```

signifie que la page demandée a changé de place (donc d'URL). La nouvelle URL est donnée par l'entête **Location:**

```
Location: /IISSamples/Default/welcome.htm
```

Si nous utilisons GET au lieu de HEAD dans l'appel au client Web :

```
dos>java clientweb http://localhost GET
HTTP/1.1 302 Object moved
Server: Microsoft-IIS/5.0
Date: Mon, 13 May 2002 09:33:36 GMT
Connection: close
Location: /IISSamples/Default/welcome.htm
Content-Length: 189
Content-Type: text/html
Set-Cookie: ASPSESSIONIDGQQQGUUY=IMFNCCMDAKPNNGMGMFIHENFE; path=/
```

```
Cache-control: private

<head><title>L'objet a changé d'emplacement</title></head>
<body><h1>L'objet a changé d'emplacement</h1>Cet objet peut être trouvé <a HREF="/IISSamples/Default/we
lcome.htm">ici</a>.</body>
```

Nous obtenons le même résultat qu'avec HEAD avec de plus le corps de la page HTML. Le programme est le suivant :

```
// paquetages importés
import java.io.*;
import java.net.*;

public class clientweb{

  // demande une URL
  // affiche le contenu de celle-ci à l'écran

  public static void main(String[] args){
    // syntaxe
    final String syntaxe="pg URI GET/HEAD";

    // nombre d'arguments
    if(args.length != 2)
      erreur(syntaxe,1);

    // on note l'URI demandée
    String URLString=args[0];
    String commande=args[1].toUpperCase();

    // vérification validité de l'URI
    URL url=null;
    try{
      url=new URL(URLString);
    }catch (Exception ex){
      // URI incorrecte
      erreur("L'erreur suivante s'est produite : " + ex.getMessage(),2);
    }//catch
    // vérification de la commande
    if(! commande.equals("GET") && ! commande.equals("HEAD")){
      // commande incorrecte
      erreur("Le second paramètre doit être GET ou HEAD",3);
    }

    // on extrait les infos utiles de l'URL
    String path=url.getPath();
    if(path.equals("")) path="/";
    String query=url.getQuery();
    if(query!=null) query="?"+query; else query="";
    String host=url.getHost();
    int port=url.getPort();
    if(port==-1) port=url.getDefaultPort();

    // on peut travailler
    Socket  client=null;          // le client
    BufferedReader IN=null;         // le flux de lecture du client
    PrintWriter OUT=null;         // le flux d'écriture du client
    String réponse=null;          // réponse du serveur
    try{
      // on se connecte au serveur
      client=new Socket(host,port);

      // on crée les flux d'entrée-sortie du client TCP
      IN=new BufferedReader(new InputStreamReader(client.getInputStream()));
      OUT=new PrintWriter(client.getOutputStream(),true);

      // on demande l'URL - envoi des entêtes HTTP
      OUT.println(commande + " " + path + query + " HTTP/1.1");
      OUT.println("Host: " + host + ":" + port);
      OUT.println("Connection: close");
      OUT.println();
      // on lit la réponse
      while((réponse=IN.readLine())!=null){
        // on traite la réponse
        System.out.println(réponse);
      }//while
      // c'est fini
      client.close();
    } catch(Exception e){
      // on gère l'exception
      erreur(e.getMessage(),4);
    }//catch
  }//main

  // affichage des erreurs
```

```
  public static void erreur(String msg, int exitCode){
    // affichage erreur
    System.err.println(msg);
    // arrêt avec erreur
    System.exit(exitCode);
  }//erreur
}//classe
```

La seule nouveauté dans ce programme est l'utilisation de la classe **URL**. Le programme reçoit une URL (*Uniform Resource Locator*) ou URI (*Uniform Resource Identifier*) de la forme *http://serveur:port/cheminPageHTML?param1=val1;param2=val2;...*. La classe **URL** nous permet de décomposer la chaîne de l'URL en ses différents éléments. Un objet URL est construit à partir de la chaîne URLstring reçue en paramètre :

```
    // vérification validité de l'URL
    URL url=null;
    try{
      url=new URL(URLString);
    }catch (Exception ex){
      // URI incorrecte
      erreur("L'erreur suivante s'est produite : " + ex.getMessage(),2);
    }//catch
```

Si la chaîne URL reçue en paramètre n'est pas une URL valide (absence du protocole, du serveur, ...), une exception est lancée. Cela nous permet de vérifier la validité du paramètre reçu. Une fois l'objet URL construit, on a accès aux différents éléments de celui-ci. Ainsi si l'objet *url* du code précédent a été construit à partir de la chaîne

http://serveur:port/cheminPageHTML?param1=val1;param2=val2;...

on aura :

url.getHost()=*serveur*
url.getPort()=*port* ou **-1** si le port n'est pas indiqué
url.getPath()=*cheminPageHTML* ou **la chaîne vide** s'il n'y a pas de chemin
url.getQuery()=*param1=val1;param2=val2;...* ou **null** s'il n'y a pas de requête
uri.getProtocol()=*http*

7.4.6 Client Web gérant les redirections

Le client Web précédent ne gère pas une éventuelle redirection de l'URL qu'il a demandée. Le client suivant la gère.

1. il lit la première ligne des entêtes HTTP envoyés par le serveur pour vérifier si on y trouve la chaîne *302 Object moved* qui signale une redirection
2. il lit les entêtes suivants. S'il y a redirection, il recherche la ligne *Location: url* qui donne la nouvelle URL de la page demandée et note cette URL.
3. il affiche le reste de la réponse du serveur. S'il y a redirection, les étapes 1 à 3 sont répétées avec la nouvelle URL. Le programme n'accepte pas plus d'une redirection. Cette limite fait l'objet d'une constante qui peut être modifiée.

Voici un exemple :

```
Dos>java clientweb2 http://localhost GET
HTTP/1.1 302 Object moved
Server: Microsoft-IIS/5.0
Date: Mon, 13 May 2002 11:38:55 GMT
Connection: close
Location: /IISSamples/Default/welcome.htm
Content-Length: 189
Content-Type: text/html
Set-Cookie: ASPSESSIONIDGQQQGUUY=PDGNCCMDNCAOFDMPHCJNPBAI; path=/
Cache-control: private

<head><title>L'objet a chang d'emplacement</title></head>
<body><h1>L'objet a chang d'emplacement</h1>Cet objet peut tre trouv <a HREF="/IISSamples/Default/we
lcome.htm">ici</a>.</body>

<--Redirection vers l'URL http://localhost:80/IISSamples/Default/welcome.htm-->

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Connection: close
```

```
Date: Mon, 13 May 2002 11:38:55 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Mon, 16 Feb 1998 21:16:22 GMT
ETag: "0174e21203bbd1:978"
Content-Length: 4781

<html>

<head>
<title>Bienvenue dans le Serveur Web personnel</title>
</head>
....
</body>
</html>
```

Le programme est le suivant :

```
// paquetages importés
import java.io.*;
import java.net.*;
import java.util.regex.*;

public class clientweb2{

  // demande une URL
  // affiche le contenu de celle-ci à l'écran

  public static void main(String[] args){
    // syntaxe
    final String syntaxe="pg URL GET/HEAD";

    // nombre d'arguments
    if(args.length != 2)
      erreur(syntaxe,1);

    // on note l'URI demandée
    String URLString=args[0];
    String commande=args[1].toUpperCase();

    // vérification validité de l'URI
    URL url=null;
    try{
      url=new URL(URLString);
    }catch (Exception ex){
      // URI incorrecte
      erreur("L'erreur suivante s'est produite : " + ex.getMessage(),2);
    }//catch
    // vérification de la commande
    if(! commande.equals("GET") && ! commande.equals("HEAD")){
      // commande incorrecte
      erreur("Le second paramètre doit être GET ou HEAD",3);
    }

    // on peut travailler
    Socket  client=null;          // le client
    BufferedReader IN=null;         // le flux de lecture du client
    PrintWriter OUT=null;         // le flux d'écriture du client
    String réponse=null;          // réponse du serveur
    final int nbRedirsMax=1;      // pas plus d'une redirection acceptée
    int nbRedirs=0;               // nombre de redirections en cours
    String premièreLigne;         // 1ère ligne de la réponse
    boolean redir=false;          // indique s'il y a redirection ou non
    String locationString="";       // la chaîne URL d'une éventuelle redirection

    // expression régulière pour trouver une URL de redirection
    Pattern location=Pattern.compile("^Location: (.+?)$");

    // gestion des erreurs
    try{
      // on peut avoir plusieurs URL à demander s'il y a des redirections
      while(nbRedirs<=nbRedirsMax){

        // on extrait les infos utiles de l'URL
        String protocol=url.getProtocol();
        String path=url.getPath();
        if(path.equals("")) path="/";
        String query=url.getQuery();
        if(query!=null) query="?"+query; else query="";
        String host=url.getHost();
        int port=url.getPort();
        if(port==-1) port=url.getDefaultPort();

        // on se connecte au serveur
```

```
      client=new Socket(host,port);

      // on crée les flux d'entrée-sortie du client TCP
      IN=new BufferedReader(new InputStreamReader(client.getInputStream()));
      OUT=new PrintWriter(client.getOutputStream(),true);

      // on demande l'URL - envoi des entêtes HTTP
      OUT.println(commande + " " + path + query + " HTTP/1.1");
      OUT.println("Host: " + host + ":" + port);
      OUT.println("Connection: close");
      OUT.println();

      // on lit la première ligne de la réponse
      premièreLigne=IN.readLine();
      // écho écran
      System.out.println(premièreLigne);

      // redirection ?
       if(premièreLigne.endsWith("302 Object moved")){
        // il y a une redirection
        redir=true;
        nbRedirs++;
      }//if

      // entêtes HTTP suivants jusqu'à trouver la ligne vide signalant la fin des entêtes
      boolean locationFound=false;
      while(!(réponse=IN.readLine()).equals("")){
        // on affiche la réponse
        System.out.println(réponse);
        // s'il y a redirection, on recherche l'entête Location
        if(redir && ! locationFound){
          // on compare la ligne à l'expression relationnelle location
          Matcher résultat=location.matcher(réponse);
          if(résultat.find()){
            // si on a trouvé on note l'URL de redirection
            locationString=résultat.group(1);
            // on note qu'on a trouvé
            locationFound=true;
          }//if
        }//if
        // entête suivant
      }//while

      // lignes suivantes de la réponse
      System.out.println(réponse);
      while((réponse=IN.readLine())!=null){
        // on affiche la réponse
        System.out.println(réponse);
      }//while
      // on ferme la connexion
      client.close();
      // a-t-on fini ?
      if ( ! locationFound || nbRedirs>nbRedirsMax)
        break;
      // il y a une redirection à opérer - on construit la nouvelle URL
      URLString=protocol +"://"+host+":"+port+locationString;
      url=new URL(URLString);
      // suivi
      System.out.println("\n<--Redirection vers l'URL "+URLString+"-->\n");
    }//while
   } catch(Exception e){
     // on gère l'exception
     erreur(e.getMessage(),4);
   }//catch
  }//main

  // affichage des erreurs
  public static void erreur(String msg, int exitCode){
    // affichage erreur
    System.err.println(msg);
    // arrêt avec erreur
    System.exit(exitCode);
  }//erreur
}//classe
```

7.4.7 Serveur de calcul d'impôts

Nous reprenons l'exercice IMPOTS déjà traité sous diverses formes. Rappelons la dernière mouture :

Une classe de base **impôts** a été créée. Ses attributs sont trois tableaux de nombres :

```
public class impots{
```

```
  // les données nécessaires au calcul de l'impôt
  // proviennent d'une source extérieure

  protected double[] limites=null;
  protected double[] coeffR=null;
  protected double[] coeffN=null;

  // constructeur vide
  protected impots(){}

  // constructeur
  public impots(double[] LIMITES, double[] COEFFR, double[] COEFFN) throws Exception{
```

La classe **impots** a deux constructeurs :

- un constructeur à qui on passe les trois tableaux de données nécessaires au calcul de l'impôt

```
  public impots(double[] LIMITES, double[] COEFFR, double[] COEFFN) throws Exception{
```

- un constructeur sans paramètres utilisable uniquement par des classes fille

```
  protected impots(){}
```

A partir de cette classe a été dérivée la classe **impotsJDBC** qui permet de remplir les trois tableaux *limites*, *coeff*R, *coeff*N à partir du contenu d'une base de données :

```
public class impotsJDBC extends impots{
  // rajout d'un constructeur permettant de construire
  // les tableaux limites, coeffr, coeffn à partir de la table
  // impots d'une base de données
  public impotsJDBC(String dsnIMPOTS, String userIMPOTS, String mdpIMPOTS)
    throws SQLException,ClassNotFoundException{

     // dsnIMPOTS : nom DSN de la base de données
     // userIMPOTS, mdpIMPOTS : login/mot de passe d'accès à la base
```

Une application graphique avait été écrite. L'application utilisait un objet de la classe **impotsJDBC**. L'application et cet objet étaient sur la même machine. Nous nous proposons de mettre le programme de test et l'objet *impotsJDBC* sur des machines différentes. Nous aurons une application client-serveur où l'objet *impotsJDBC* distant sera le serveur. La nouvelle classe s'appelle *ServeurImpots* et est dérivée de la classe *impotsJDBC :*

```
// paquetages importés
import java.net.*;
import java.io.*;
import java.sql.*;

public class ServeurImpots extends impotsJDBC {

  // attributs
  int portEcoute;      // le port d'écoute des demandes clients
  boolean actif;     // état du serveur

  // constructeur
  public ServeurImpots(int portEcoute,String DSNimpots, String USERimpots, String MDPimpots)
    throws IOException, SQLException, ClassNotFoundException {
    // construction parent
    super(DSNimpots, USERimpots, MDPimpots);
    // on note le port d'écoute
    this.portEcoute=portEcoute;
    // pour l'instant inactif
    actif=false;
    // crée et lance un thread de lecture des commandes tapées au clavier
    // le serveur sera géré à partir de ces commandes
    Thread admin=new Thread(){
      public void run(){
        try{
          admin();
        }catch (Exception ignored){}
      }
    };
    admin.start();
  }//ServeurImpots
```

Le seul paramètre nouveau dans le constructeur est le port d'écoute des demandes des clients. Les autres paramètres sont passés directement à la classe de base *impotsJDBC*. Le serveur d'impôts est contrôlé par des commandes tapées au clavier. Aussi crée-t-on un thread pour lire ces commandes. Il y en aura deux possibles : *start* pour lancer le service, *stop* pour l'arrêter définitivement. La méthode *admin* qui gère ces commandes est la suivante :

```
  public void admin() throws IOException{
    // lit les commandes d'administration du serveur tapées au clavier
    // dans une boucle sans fin
    String commande=null;
     BufferedReader IN=new BufferedReader(new InputStreamReader(System.in));
    while(true){
      // invite
      System.out.print("Serveur d'impôts>");
      // lecture commande
      commande=IN.readLine().trim().toLowerCase();
      // exécution commande
      if(commande.equals("start")){
        // actif ?
        if(actif){
          //erreur
          System.out.println("Le serveur est déjà actif");
          // on continue
          continue;
        }//if
        // on crée et lance le service d'écoute
        Thread ecoute=new Thread(){
          public void run(){
            ecoute();
          }
        };
        ecoute.start();
      }//if
      else if(commande.equals("stop")){
        // fin de tous les threads d'exécution
        System.exit(0);
      }//if
      else {
        // erreur
        System.out.println("Commande incorrecte. Utilisez (start,stop)");
      }//if
    }//while
  }//admin
```

Si la commande tapée au clavier est *start*, un thread d'écoute des demandes clients est lancé. Si la commande tapée est *stop*, tous les threads sont arrêtés. Le thread d'écoute exécute la méthode *ecoute* :

```
  public void ecoute(){
    // thread d'écoute des demandes des clients
    // on crée le service d'écoute
    ServerSocket ecoute=null;
    try{
      // on crée le service
      ecoute=new ServerSocket(portEcoute);
      // suivi
      System.out.println("Serveur d'impôts lancé sur le port " + portEcoute);

      // boucle de service
      Socket liaisonClient=null;
      while (true){ // boucle infinie
        // attente d'un client
        liaisonClient=ecoute.accept();

        // le service est assuré par une autre tâche
        new traiteClientImpots(liaisonClient,this).start();

        // on retourne à l'écoute des demandes
      }// fin while
    }catch(Exception ex){
      // on signale l'erreur
      erreur("L'erreur suivante s'est produite : " + ex.getMessage(),3);
    }//catch
  }//thread d'écoute
```

On retrouve un serveur tcp classique écoutant sur le port *portEcoute*. Les demandes des clients sont traitées par la méthode *run* du thread traiteCientImpots au constructeur duquel on passe deux paramètres :

1. l'objet *Socket liaisonClient* qui va permettre d'atteindre le client
2. l'objet *impotsJDBC this* qui va donner accès à la méthode *this.calculer* de calcul de l'impôt.

```
// ------------------------------------------------------
// assure le service à un client du serveur d'impôts

class traiteClientImpots extends Thread{

  private Socket liaisonClient;     // liaison avec le client
```

```
  private BufferedReader IN;      // flux d'entrée
  private PrintWriter OUT;        // flux de sortie
  private impotsJDBC objImpots;     // objet Impôt

    // constructeur
    public traiteClientImpots(Socket liaisonClient,impotsJDBC objImpots){
    this.liaisonClient=liaisonClient;
    this.objImpots=objImpots;
  }//constructeur
```

La méthode r*un* traite les demandes des clients. Ce sont des lignes de texte qui peuvent avoir deux formes :

1. *calcul marié(o/n) nbEnfants salaireAnnuel*
2. *fincalculs*

La forme 1 permet le calcul d'un impôt, la forme 2 clôt la liaison client-serveur.

```
  // méthode run
  public void run(){
    // rend le service au client
    try{
      // flux d'entrée
      IN=new BufferedReader(new InputStreamReader(liaisonClient.getInputStream()));
      // flux de sortie
      OUT=new PrintWriter(liaisonClient.getOutputStream(),true);
      // envoi d'un msg de bienvenue au client
      OUT.println("Bienvenue sur le serveur d'impôts");

      // boucle lecture demande/écriture réponse
      String demande=null;
      String[] champs=null;// les éléments de la demande
      String commande=null;// la commande du client : calcul ou fincalculs
      while ((demande=IN.readLine())!=null){
        // on décompose la demande en champs
        champs=demande.trim().toLowerCase().split("\\s+");
        // deux demandes acceptées : calcul et fincalculs
        commande=champs[0];
        if(! commande.equals("calcul") && ! commande.equals("fincalculs")){
          // erreur client
          OUT.println("Commande incorrecte. Utilisez (calcul,fincalculs).");
          // commande suivante
          continue;
        }//if
        if(commande.equals("calcul")) calculerImpôt(champs);
        if(commande.equals("fincalculs")){
          // msg d'au-revoir au client
          OUT.println("Au revoir...");
          // libération des ressources
          try{ OUT.close();IN.close();liaisonClient.close();}
          catch(Exception ex){}
          // fin
          return;
        }//if
        //demande suivante
      }//while
    }catch (Exception e){
      erreur("L'erreur suivante s'est produite ("+e+")",2);
    }// fin try
  }// fin Run
```

Le calcul de l'impôt est effectué par la méthode *calculerImpôt* qui reçoit en paramètre le tableau des champs de la demande faite par le client. La validité de la demande est vérifiée et éventuellement l'impôt calculé et renvoyé au client.

```
  // calcul d'impôts
    public void calculerImpôt(String[] champs){
    // traite la demande : calcul marié nbEnfants salaireAnnuel
    // décomposée en champs dans le tableau champs

    String marié=null;
    int nbEnfants=0;
    int salaireAnnuel=0;

    // validité des arguments
    try{
      // il faut au moins 4 champs
      if(champs.length!=4) throw new Exception();
      // marié
      marié=champs[1];
      if (! marié.equals("o") && ! marié.equals("n")) throw new Exception();
      // enfants
      nbEnfants=Integer.parseInt(champs[2]);
      // salaire
```

```
      salaireAnnuel=Integer.parseInt(champs[3]);
    }catch (Exception ignored){
      // erreur de format
      OUT.println(" syntaxe : calcul marié(O/N) nbEnfants salaireAnnuel");
      // fini
      return;
    }//if
    // on peut calculer l'impôt
    long impot=objImpots.calculer(marié.equals("o"),nbEnfants,salaireAnnuel);
    // on envoie la réponse au client
    OUT.println(""+impot);
  }//calculer
```

Un programme de test pourrait être le suivant :

```
// appel : serveurImpots port dsnImpots userImpots mdpImpots

import java.io.*;

public class testServeurImpots{
  public static final String syntaxe="Syntaxe : pg port dsnImpots userImpots mdpImpots";

  // programme principal
  public static void main (String[] args){

    // il faut 4 arguments
    if(args.length != 4)
      erreur(syntaxe,1);

    // le port doit être entier >0
    int port=0;
    boolean erreurPort=false;
    Exception E=null;
    try{
      port=Integer.parseInt(args[0]);
    }catch(Exception e){
      E=e;
      erreurPort=true;
    }
    erreurPort=erreurPort || port <=0;
    if(erreurPort)
      erreur(syntaxe+"\n"+"Port incorrect ("+E+")",2);

    // on crée le serveur d'impôts
    try{
      new ServeurImpots(port,args[1],args[2],args[3]);
    }catch(Exception ex){
      //erreur
      System.out.println("L'erreur suivante s'est produite : "+ex.getMessage());
    }//catch
  }//Main

  // affichage des erreurs
  public static void erreur(String msg, int exitCode){
    // affichage erreur
    System.err.println(msg);
    // arrêt avec erreur
    System.exit(exitCode);
  }//erreur
}// fin class
```

On passe au programme de test les données nécessaires à la construction d'un objet *ServeurImpots* et à partir de là il crée cet objet.

Tentons une première exécution :

```
dos>java testServeurImpots 124 mysql-dbimpots admimpots mdpimpots
Serveur d'impôts>start
Serveur d'impôts>Serveur d'écho lancé sur le port 124
stop
```

La commande

```
dos>java testServeurImpots 124 mysql-dbimpots admimpots mdpimpots
```

crée un objet *ServeurImpots* qui n'écoute pas encore les demandes des clients. C'est la commande **start** tapée au clavier qui lance cette écoute. La commande **stop** arrête le serveur. Utilisons maintenant un client. Nous utiliserons le client générique créé précédemment. Le serveur est lancé :

```
dos>java testServeurImpots 124 mysql-dbimpots admimpots mdpimpots
Serveur d'impôts>start
Serveur d'impôts>Serveur d'écho lancé sur le port 124
```

Le client générique est lancé dans une autre fenêtre Dos :

```
Dos>java clientTCPgenerique localhost 124
Commandes :
<-- Bienvenue sur le serveur d'impôts
```

On voit que le client a bien récupéré le message de bienvenue du serveur. On envoie d'autres commandes :

```
x
<-- Commande incorrecte. Utilisez (calcul,fincalculs).
calcul
<--  syntaxe : calcul marié(O/N) nbEnfants salaireAnnuel
calcul o 2 200000
<-- 22506
calcul n 2 200000
<-- 33388
fincalculs
<-- Au revoir...
[fin du thread de lecture des réponses du serveur]
fin
[fin du thread d'envoi des commandes au serveur]
```

On retourne dans la fenêtre du serveur pour l'arrêter :

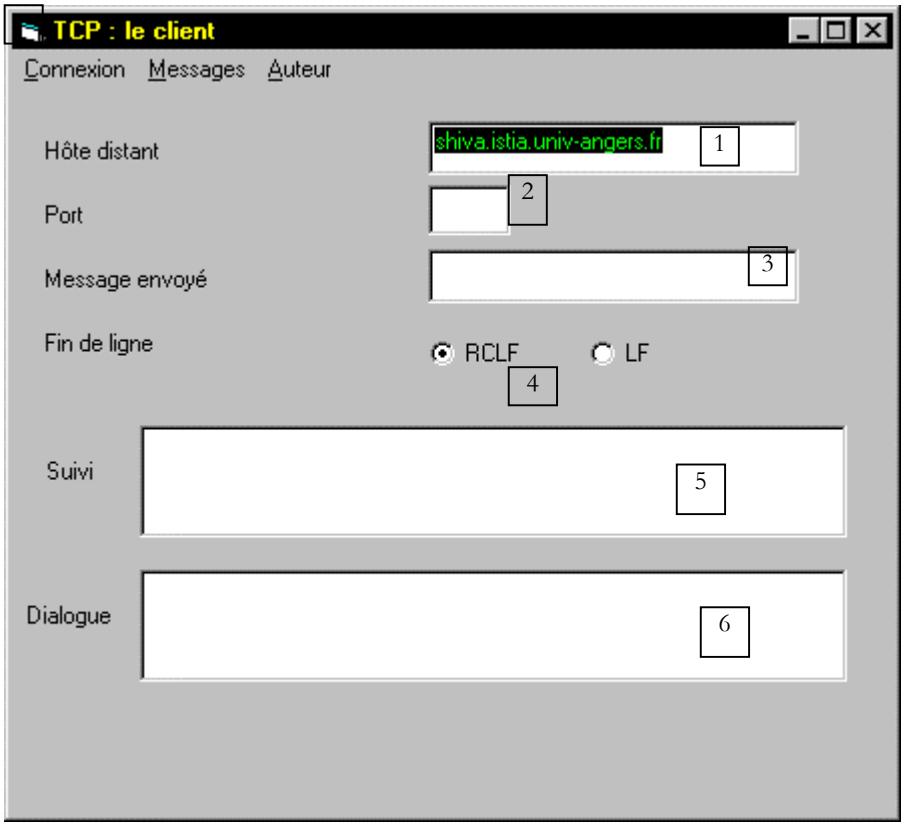
```
dos>java testServeurImpots 124 mysql-dbimpots admimpots mdpimpots
Serveur d'impôts>start
Serveur d'impôts>Serveur d'écho lancé sur le port 124
stop
```

7.5 Exercices

7.5.1 Exercice 1 - Client TCP générique graphique

7.5.1.1 Présentation de l'application

On se propose de créer un programme capable de dialoguer sur l'Internet avec les principaux services TCP. On l'appellera un client tcp générique. Lorsqu'on a compris cette application, on trouve que tous les clients tcp se ressemblent. La fenêtre du programme est la suivante :



La signification des différents contrôles est la suivante :

n°	nom	type	rôle
1	*TxtRemoteHost*	*JTextField*	nom de la machine offrant le service désiré
2	*TxtPort*	*JTextField*	port du service demandé
3	*TxtSend*	*JTextField*	texte du message qui sera envoyé au serveur par le client
4	*OptRCLF* *OptLF*	*JCheckBox*	boutons permettant d'indiquer comment se terminent les lignes dans le dialogue client/serveur RCLF : retour chariot (#13) + passage à la ligne (#10) LF : passage à la ligne (#10)
5	*LstSuivi*	*JList*	affiche des messages sur l'état de la communication entre le client & le serveur
6	*LstDialogue*	*JList*	affiche les messages échangés par le client (->) et le serveur (<-)
7	*CmdAnnuler*	*JButton*	caché - situé sous la liste de dialogue - Apparaît lorsque la connexion est en cours et permet de l'interrompre si le serveur ne répond pas

Les options de menu disponibles sont les suivantes :

option	sous-options	rôle
Connexion	Connecter	connecte le client au serveur
	Déconnecter	ferme la connexion
	Quitter	Termine le programme
Messages	Envoyer	Envoie le message du contrôle TxtSend au serveur
	RazSuivi	Efface la liste LstSuivi
	RazDialogue	Efface la liste LstDialogue
Auteur		affiche une boîte de copyright

7.5.1.2 FONCTIONNEMENT DE L'APPLICATION

Initialisation de l'application

Lorsque la feuille principale de l'application est chargée, les actions suivantes prennent place :

- la feuille est centrée sur l'écran
- seules les options de menu *Connexion/Quitter* & *Auteur* sont actives
- le bouton *Annuler* est caché
- les listes *LstSuivi & LstDialogue* sont vides

Menu Connexion/Connecter

Cette option n'est disponible que lorsque les champs *Hôte Distant* et *n° de port* sont non vides et qu'une connexion n'est pas actuellement active. Un clic sur cette option entraîne les opérations suivantes :

- la validité du port est vérifiée : ce doit être un entier >0
- un thread est lancé pour assurer la connexion au serveur
- le bouton *Annuler* apparaît afin de permettre à l'utilisateur d'interrompre la connexion en cours
- toutes les options de menu sont désactivées sauf *Quitter* & *Auteur*

La connexion peut se terminer de plusieurs façons :

1. L'utilisateur a appuyé sur le bouton *Annuler* : on arrête le thread de connexion et on remet le menu dans son état de départ. On indique dans le suivi qu'il y a eu fermeture de la connexion par l'utilisateur.
2. La connexion se termine avec une erreur : on fait la même chose que précédemment et de plus dans le suivi, on indique la cause de l'erreur.
3. La connexion se termine correctement : on enlève le bouton *Annuler*, indique dans le suivi que la connexion a été faite, autorise le menu *RazSuivi*, inhibe le menu *Connecter*, autorise le menu *Déconnecter*

Menu Connexion/Déconnecter

Cette option n'est disponible que lorsqu'une connexion avec le serveur existe. Lorsqu'elle est activée elle clôt la connexion avec le serveur et remet le menu dans son état de départ. On indique dans le suivi que la connexion a été fermée par le client.

Menu Connexion/Quitter

Cette option clôt une éventuelle connexion active avec le serveur et termine l'application.

Menu Messages/Envoyer

Cette option n'est accessible que si les conditions suivantes sont réunies :

- la connexion avec le serveur a été faite
- il y a un message à envoyer

Si ces conditions sont réunies, on envoie au serveur le texte présent dans le champ *TxtSend* (3) terminé par la séquence RCLF si l'option RCLF a été cochée, la séquence LF sinon. Une éventuelle erreur à l'émission est signalée dans la liste de suivi.

Menus RazSuivi et RazDialogue

Vident respectivement les listes *LstSuivi* et *LstDialogue*. Ces options sont inhibées lorsque les listes cooréspondantes sont vides.

Le bouton Annuler

Ce bouton situé en bas du formulaire n'apparaît que lorsque le client est en cours de connexion au serveur. Cette connexion peut ne pas se faire parce que le serveur ne répond pas ou mal. Le bouton ***Annuler*** donne alors à l'utilisateur, la possibilité d'interrompre la demande de connexion.

Les listes de suivi

La liste LstSuivi (**5**) fait le suivi de la connexion. Elle indique les moments clés de la connexion :

- son ouverture par le client
- sa fermeture par le serveur ou le client
- toutes les erreurs qui peuvent se produire tant que la liaison est active

La liste LstDialogue (***6***) fait le suivi du dialogue qui s'instaure entre le client et le serveur. Un thread surveille en tâche de fond ce qui arrive sur la socket de communication du client et l'affiche dans la liste 6.

L'option Auteur

Ce menu ouvre une fenêtre dite de Copyright :

Gestion des erreurs

Les erreurs de connexion sont signalées dans la liste de suivi ***6***, celles liées au dialogue client/serveur dans la liste de dialogue ***7***. Lors d'une erreur de liaison, le dialogue client/serveur est fermé et le formulaire remis dans son état initial prêt pour une nouvelle connexion.

7.5.1.3 TRAVAIL A FAIRE

Réaliser le travail décrit précédemment sous deux formes :

- application autonome
- applet

7.5.2 Exercice 2 - Un serveur de ressources

7.5.2.1 INTRODUCTION

Une institution possède plusieurs puissants serveurs de calcul accessibles sur Internet. Toute machine souhaitant utiliser ces services de calcul envoie un fichier de données sur le port 756 d'un des serveurs. Ce fichier comprend diverses informations : login, mot de passe, commandes indiquant le type de calcul désiré, les données sur lesquelles faire le calcul. Si le fichier de données est correct, le serveur de clacul choisi l'utilise et renvoie les résultats au client sous la forme d'un fichier texte.

Les avantages d'une telle organisation sont multiples :

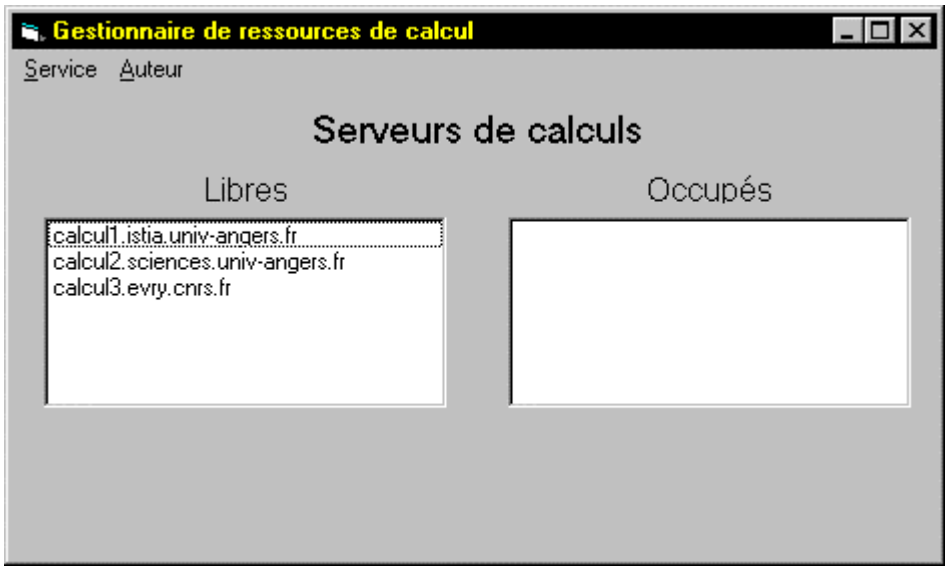
- tout type de client (PC, Mac, Unix,...) peut utiliser ce service
- le client peut être n'importe où sur l'Internet
- les moyens de calcul sont optimisés : seules quelques machines puissantes sont nécessaires. Ainsi, une petite organisation sans moyens de calcul peut utiliser ce service moyennant une contribution financière calculée d'après le temps de calcul utilisé.

Malgré la puissance des machines, un calcul peut parfois durer plusieurs heures : le serveur n'est alors pas disponible pour d'autres clients. Se pose alors pour un client, le problème de trouver un serveur de calcul disponible. Pour cela, on utilise alors un « gestionnaire de ressources de calcul », appelé serveur GRC par la suite. Ce service est placé sur une unique machine et travaille sur le port **864** en mode **tcp**. C'est à lui que s'adresse un client désirant un accès à un serveur de calcul. Le serveur GRC qui détient la liste complète des serveurs de calcul lui répond en lui envoyant le nom d'un serveur actuellement inactif. Le client n'a plus alors qu'à envoyer ses données au serveur qu'on lui aura désigné.

On se propose d'écrire le serveur GRC.

7.5.2.2 L'INTERFACE VISUELLE

L'interface visuelle sera la suivante :



L'interface présente deux listes de serveurs :

- à gauche, la liste des serveurs inactifs donc disponibles pour des calculs
- à droite, la liste des serveurs occupés par les calculs d'un client.

La structure du menu est la suivante :

Menu Principal	Menu secondaire	Rôle
Service	Lancer	Lance le service tcp sur le port 864
	Arrêter	Arrête le service
	Quitter	Termine l'application
Auteur		Informations de Copyright

La structure des contrôles présents sur le formulaire est la suivante :

Nom	Type	Rôle
listLibres	JList	Liste des serveurs libres
listOccupés	JList	Liste des serveurs occupés

7.5.2.3 FONCTIONNEMENT DE L'APPLICATION

Chargement de l'application

Au chargement de l'application, la liste **listLibres** est remplie avec la liste des noms des serveurs de calcul gérés par le GRC. Ceux-ci sont définis dans un fichier **Serveurs** passé en paramètre. Ce fichier contient une liste de noms de serveurs à raison d'un par ligne et est donc utilisé pour remplir la liste **listLibres**. Le menu **Lancer** est autorisé, le menu **Arrêter** est inhibé.

Option Service/Lancer

Cette option

- lance le service d'écoute du port 864 de la machine
- inhibe le menu **Lancer**
- autorise le menu **Arrêter**

Option Service/Arrêter

Cette option interrompt le service :

- la liste des serveurs occupés est vidée
- la liste des serveurs libres est remplie avec le contenu du fichier **Serveurs**
- le menu **Lancer** est autorisé
- le menu **Arrêter** est inhibé

Option Service/Quitter

L'application se termine.

Dialogue client/serveur

Le dialogue client/serveur se fait par échange de lignes de texte terminées par la séquence RCLF. Le serveur GRC reconnaît deux commandes : **getserveur** et **finservice**. Nous détaillons le rôle de ces deux commandes :

1 getserveur

Le client demande s'il y a un serveur de calcul disponible pour lui.

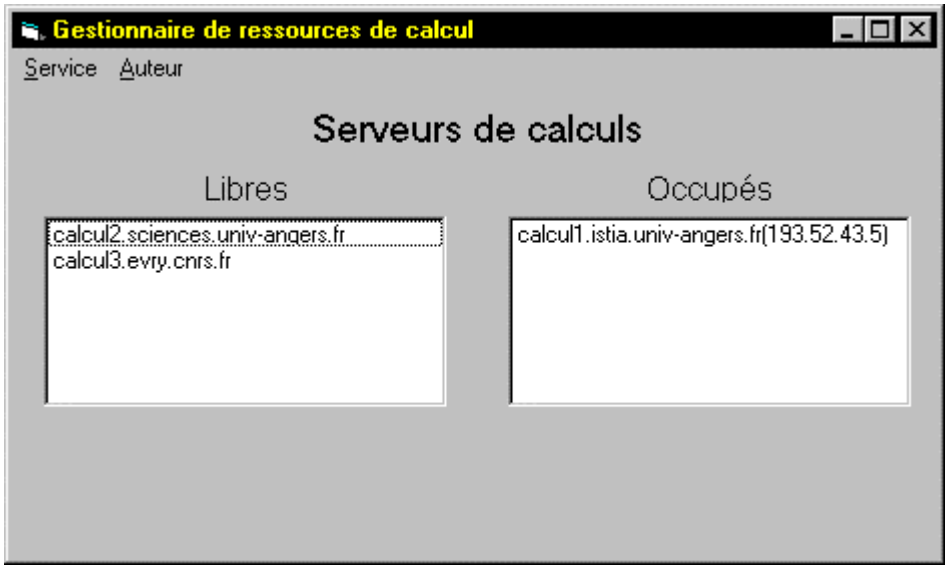
Le serveur GRC prend alors le premier serveur trouvé dans sa liste de serveurs libres et renvoie son nom au client sous la forme :

100-nom du serveur

Par ailleurs, il passe le serveur accordé au client dans la liste des serveurs occupés sous la forme :

serveur (IP du client)

comme le montre l'exemple suivant où le serveur *calcul1.istia.univ-angers.fr* est occupé à servir le client d'adresse IP *193.52.43.5* :



Un client ne peut envoyer une commande **getserveur** si un serveur de calcul lui a déjà été affecté. Ainsi avant de répondre au client, le serveur GRC vérifie que l'adresse IP du client n'est pas déjà présent parmi celles enregistrées dans la liste des serveurs occupés. Si c'est le cas, le serveur GRC répond :

501-Vous avez actuellement une demande en cours

Enfin, il y a le cas où aucun serveur de calcul n'est disponible : la liste des serveurs libres est vide. Dans ce cas, le serveur GRC répond :

502- Il n'y a aucun serveur de calcul disponible

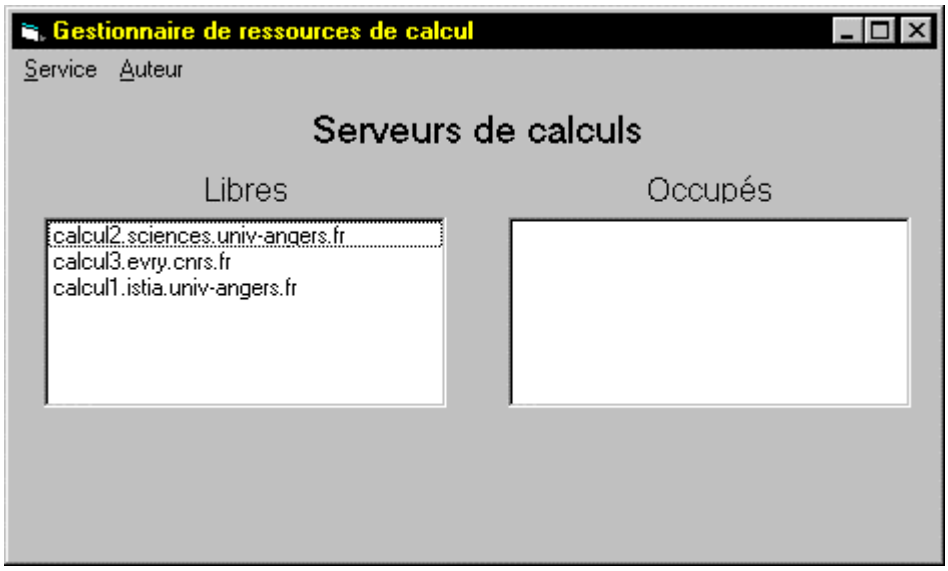
Dans tous les cas, après avoir répondu au client, le serveur GRC clôt la connexion avec celui-ci afin de pouvoir servir d'autres clients.

2 finservice

Le client signifie qu'il n'a plus besoin du serveur de calcul qu'il utilisait.

Le serveur GRC vérifie d'abord que le client est bien un client qu'il servait. Pour cela, il vérifie que l'adresse IP du client est présente parmi celles enregistrées dans la liste des serveurs occupés. Si ce n'est pas le cas, le serveur GRC répond :
503-Aucun serveur ne vous a été attribué

Si le client est reconnu, le serveur GRC lui répond :
101-Fin de service acceptée
et passe le serveur de calcul attribué à ce client dans la liste des serveurs libres. Pour reprendre l'exemple précédent, si le client envoie la commande **finservice**, l'affichage du serveur GRC devient :



Après l'envoi de la réponse, quelque soit celle-ci, le serveur GRC clôt la connexion.

7.5.2.4 TRAVAIL A FAIRE

Ecrire l'application comme un programme autonome qui pourra être testé par exemple avec un client *telnet* ou avec le client tcp générique de l'exercice précédent.

7.5.3 Exercice 3 - un client smtp

7.5.3.1 INTRODUCTION

Nous souhaitons ici, construire un client pour le service **SMTP** (SendMail Transfer Protocol) qui permet d'envoyer le courrier. Sous Unix ou Windows, le programme *telnet* est un client travaillant avec le protocole *tcp*. Il peut « converser » avec tout service *tcp* acceptant des commandes au format texte terminées par la séquence RCLF, c'est à dire les caractères de code ASCII 13 et 10. Voici un exemple de conversation avec le service *smtp* d'envoi du courrier :

```
$ telnet istia.univ-angers.fr 25     // appel du service smtp
```

// réponse du serveur *smtp*

```
Trying 193.52.43.2...
Connected to istia.univ-angers.fr.
Escape character is '^]'.
220-Istia.Istia.Univ-Angers.fr Sendmail 8.6.10/8.6.9 ready at Tue, 16 Jan 1996 07:53:12 +0100
220 ESMTP spoken here
```

// commentaires ----------
Le programme *telnet* peut appeler tout service par la syntaxe
** *telnet* machine_service port_service**

Les échanges client/serveur se font avec des lignes de texte terminées par la séquence RCLF.
Les réponses du service *smtp* sont de la forme :
numéro-Message ou
numéro Message
Le serveur smtp peut envoyer plusieurs lignes de réponse. La dernière ligne de la réponse est signalée par un numéro suivi d'un espace alors que pour les lignes précédentes de la réponse, le numéro est suivi d'un tiret -.
Un numéro supérieur ou égal à 500 signale un message d'erreur.

// fin de commentaires

```
help            // commande émise au clavier
```

// réponse du serveur *smtp*

```
214-Commands:
214- HELO EHLO MAIL RCPT DATA
214- RSET NOOP QUIT HELP VRFY
214- EXPN VERB
214-For more info use "HELP <topic>".
214-To report bugs in the implementation send email to
214- sendmail@CS.Berkeley.EDU.
214-For local information send email to Postmaster at your site.
214 End of HELP info
```

```
mail from: serge.tahe@istia.univ-angers.fr // nouvelle commande émise au clavier
```

// commentaires ---------
La commande mail a la syntaxe suivante :
mail from: adresse électronique de l'expéditeur du message
// fin de commentaires

// réponse du serveur *smtp*

```
250 serge.tahe@istia.univ-angers.fr... Sender ok
```

// commentaires
Le serveur *smtp* ne fait aucune vérification de validité de l'adresse de l'expéditeur : il la prend telle qu'on la lui a donnée
// fin de commentaires

```
rcpt to: user1@istia.univ-angers.fr   // nouvelle commande émise au clavier
```

// commentaires ---------
La commande rcpt a la syntaxe suivante :
rcpt to: adresse électronique du destinataire du message
Si l'adresse électronique est une adresse de la machine sur laquelle travaille le serveur *smtp*, il vérifie qu'elle existe bien sinon il ne fait aucune vérification. Si vérification il y a eu et qu'une erreur a été détectée elle sera signalée avec un numéro >= 500.

On peut émettre autant de commandes *rcpt to* que l'on veut : cela permet d'envoyer un message à plusieurs personnes.
// fin de commentaires

// réponse du serveur *smtp*

```
250 user1@istia.univ-angers.fr... Recipient ok
```

```
data            // nouvelle commande émise au clavier
```

// commentaires ---------
La commande data a la syntaxe suivante :
data
ligne1
ligne2
...
.
Elle est suivie des lignes de texte composant le message, celui-ci devant se terminer par une ligne comportant le seul caractère « point ».
Le message est alors envoyé au destinataire indiqué par la commande rcpt.
// fin de commentaires

// réponse du serveur *smtp*

```
354 Enter mail, end with "." on a line by itself
```

// texte du message tapé au clavier

```
subject: essai smtp

essai smtp a partir de telnet
.
```

// commentaires
Dans les lignes de texte de la commande data, on peut mettre une ligne subject: pour préciser le sujet du courrier. Cette ligne doit être suivie d'une ligne vide.

// réponse du serveur *smtp*

```
250 HAA11627 Message accepted for delivery
```

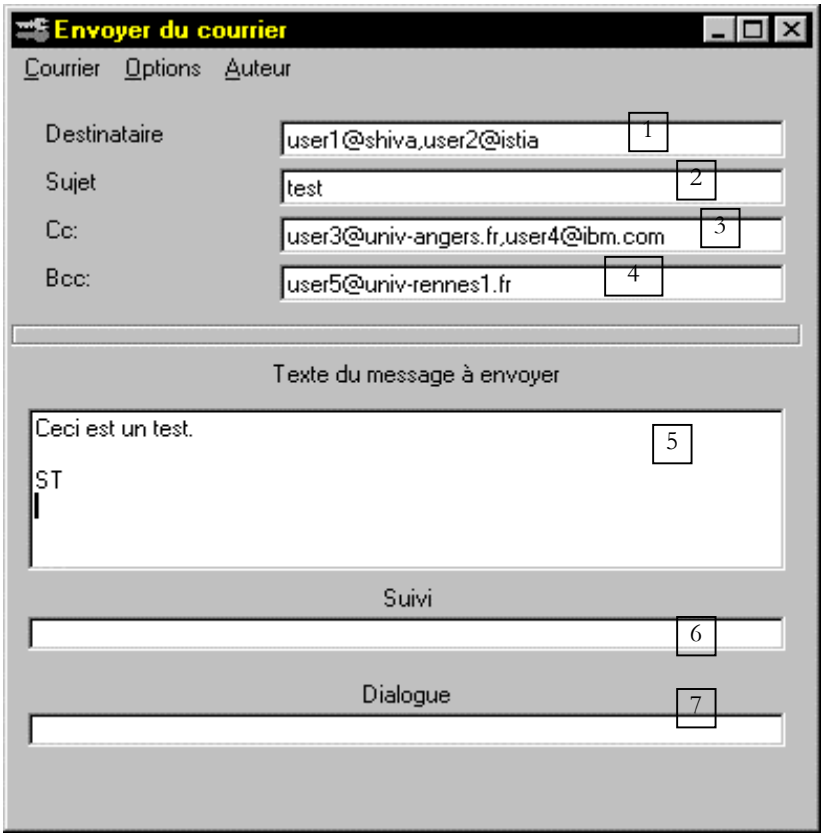
```
quit                    // nouvelle commande émise au clavier
```

// commentaires
La commande quit clôt la connexion au service *smtp*
// fin de commentaires

// réponse du serveur *smtp*
```
221 Istia.Istia.Univ-Angers.fr closing connection
```

7.5.3.2 L'INTERFACE VISUELLE

On se propose de construire un programme avec l'interface visuelle suivante :



Les contrôles ont le rôle suivant :

Numéro	Type	Rôle
1	JTextField	Suite d'adresses électroniques séparées par une virgule
2	JTextField	Texte du sujet du message
3	JTextField	Suite d'adresses électroniques séparées par une virgule
4	JTextField	Suite d'adresses électroniques séparées par une virgule
5	JTextArea	Texte du message
6	JList	liste de suivi
7	JList	liste de dialogue
8	JButton	bouton « Annuler » non représenté, apparaissant lorsque le client demande la connexion au serveur SMTP. Permet à l'utilisateur d'interrompre cette demande si le serveur ne répond pas.

7.5.3.3 LES MENUS

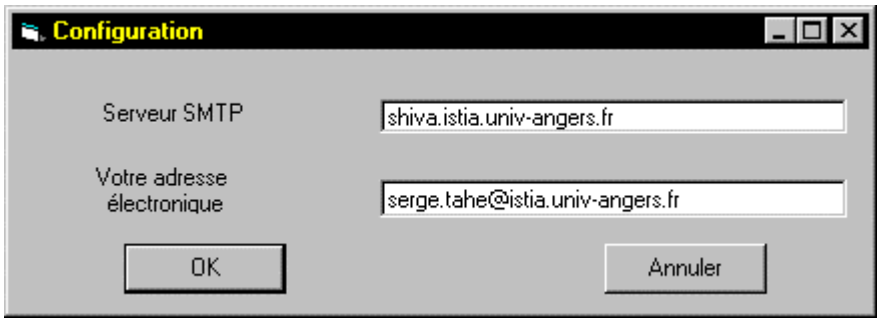
La structure des menus de l'application est la suivante :

Menu Principal	Menu secondaire	Rôle
Courrier		
	Envoyer	Envoie le message du contrôle 5
	Quitter	Quitte l'application
Options		
	Masquer Suivi	Rend invisible le contrôle 6
	Raz Suivi	Vide la liste de suivi 6
	Masquer Dialogue	Rend invisible la liste de dialogue 7
	Raz Dialogue	Vide la liste de dialogue 7
	Configurer	Permet à l'utilisateur de préciser - l'adresse du serveur smtp utilisé par le programme - son adresse électronique
	Sauvegarder...	Sauvegarde la configuration précédente dans un fichier .ini
Auteur		Informations de Copyright

7.5.3.4 FONCTIONNEMENT DE L'APPLICATION

Menu Options/Configurer

Ce menu provoque l'apparition de la fenêtre suivante :



Les deux champs doivent être remplis pour que le bouton ***OK*** soit actif. Les deux informations doivent être mémorisées dans des variables globales afin d'être disponibles pour d'autres modules.

Menu Courrier/Envoyer

Cette option n'est accessible que si les conditions suivantes sont réunies :

- la configuration a été faite
- il y a un message à envoyer
- il y a un sujet
- il y a au moins un destinataire dans les champs 1, 3 et 4

Si ces conditions sont réunies, la séquence des événements est la suivante :

- le formulaire est mis dans un état où toutes les actions pouvant interférer dans le dialogue client/serveur sont inhibées
- il y a connexion sur le port 25 du serveur précisé dans la configuration
- le client dialogue ensuite avec le serveur smtp selon le protocole décrit plus haut
- le ***mail from:*** utilise l'adresse électronique de l'expéditeur donnée dans la configuration
- le ***rcpt to:*** s'utilise pour chacune des adresses électroniques trouvées dans les champs 1, 3 et 4
- dans les lignes envoyées après la commande ***data***, on trouvera les textes suivants :
    - une ligne ***Subject:*** texte du sujet du contrôle 2
    - une ligne ***Cc:*** adresses du contrôle 3

- une ligne ***Bcc:*** adresses du contrôle 4
- le texte du message du contrôle 5
- le point terminal

Le bouton Annuler

Ce bouton situé en bas du formulaire n'apparaît que lorsque le client est en cours de connexion au serveur *smtp*. Cette connexion peut ne pas se faire parce que le serveur *smtp* ne répond pas ou mal. Le bouton ***Annuler*** donne alors à l'utilisateur, la possibilité d'interrompre la demande de connexion.

Les listes de suivi

La liste (***6***) fait le suivi de la connexion. Elle indique les moments clés de la connexion :
- son ouverture par le client
- sa fermeture par le serveur ou le client
- toutes les erreurs de connexion

La liste (*7*) fait le suivi du dialogue *smtp* qui s'instaure entre le client et le serveur.

Ces deux listes sont associées à des options du menu :

Masquer Suivi	Rend invisible la liste de suivi 6 ainsi que le libellé qui est au-dessus. Si la hauteur occupée par ces deux contrôles est *H*, tous les contrôles situés dessous sont remontés d'une hauteur ***H*** et la taille totale du formulaire est diminuée de ***H***. Par ailleurs, ***Masquer Suivi*** rend invisible l'option ***RazSuivi*** ci-dessous.
Raz Suivi	Vide la liste de suivi ***6***
Masquer Dialogue	Rend invisible la liste de dialogue ***7***, le libellé qui est dessus ainsi que l'option de menu ***RazDialogue*** ci-dessous. Comme pour ***Masquer Suivi***, la position des contrôles situés dessous (bouton ***Annuler*** peut-être) est recalculée et la taille de la fenêtre diminuée.
Raz Dialogue	Vide la liste de dialogue ***7***

L'option Auteur

Ce menu ouvre une fenêtre dite de Copyright :

Gestion des erreurs

Les erreurs de connexion sont signalées dans la liste de suivi ***6***, celles liées au dialogue client/serveur dans la liste de dialogue ***7***. Lors d'une erreur, l'utilisateur en est averti par une boîte d'erreur et la liste contenant la cause de l'erreur est affichée si elle était auparavant masquée. Par ailleurs, le dialogue client/serveur est fermé et le formulaire remis dans son état initial.

7.5.3.5 GESTION D'UN FICHIER DE CONFIGURATION

Il est souhaitable que l'utilisateur n'ait pas à reconfigurer le logiciel à chaque fois qu'il l'utilise. Pour cela si l'option ***Options/Sauvegarder la configuration*** en quittant est **cochée**, la fermeture du programme sauve les deux informations acquises par l'option ***Options/Configurer*** ainsi que l'état des deux listes de suivi dans un fichier ***sendmail.ini*** situé dans le même répertoire que le .exe du programme. Ce fichier a la forme suivante :

```
SmtpServer=shiva.istia.univ-angers.fr
ReplyAddress=serge.tahe@istia.univ-angers.fr
Suivi=0
Dialogue=1
```

Les lignes ***SmtpServer*** et ***ReplyAddress*** reprennent les deux informations acquises par l'option Options/Configurer. Les lignes ***Suivi*** et ***Dialogue*** donnent l'état des listes de Suivi et de Dialogue : 1 (présente), 0 (absente).

Au chargement du programme, le fichier ***sendmail.ini*** est lu s'il existe et le formulaire est configuré en conséquence. Si le fichier ***sendmail.ini*** n'existe pas on fait comme si on avait :

```
SmtpServer=
ReplyAddress=
Suivi=1
Dialogue=1
```

Si le fichier ***sendmail.ini*** existe mais est incomplet (lignes manquantes), la ligne manquante est remplacée par la ligne correspondante ci-dessus. Ainsi la ligne *Suivi=...* est manquante, on fait comme si on avait *Suivi=1*.

Toutes les lignes ne correspondant pas au modèle :
mot clé= valeur
sont ignorées ainsi que celles où le mot clé est invalide. Le mot clé peut-être en majuscules ou minuscules : cela n'importe pas.

Dans l'option ***Options/Configurer***, les valeurs ***SmtpServer*** et ***ReplyAddress*** actuellement en cours sont présentées. L'utilisateur peut alors les modifier s'il le désire.

7.5.3.6 TRAVAIL A FAIRE

Réaliser le travail décrit précédemment. Il est conseillé de traiter la gestion du fichier de configuration en dernier.

7.5.4 Exercice 4 - client POPPASS

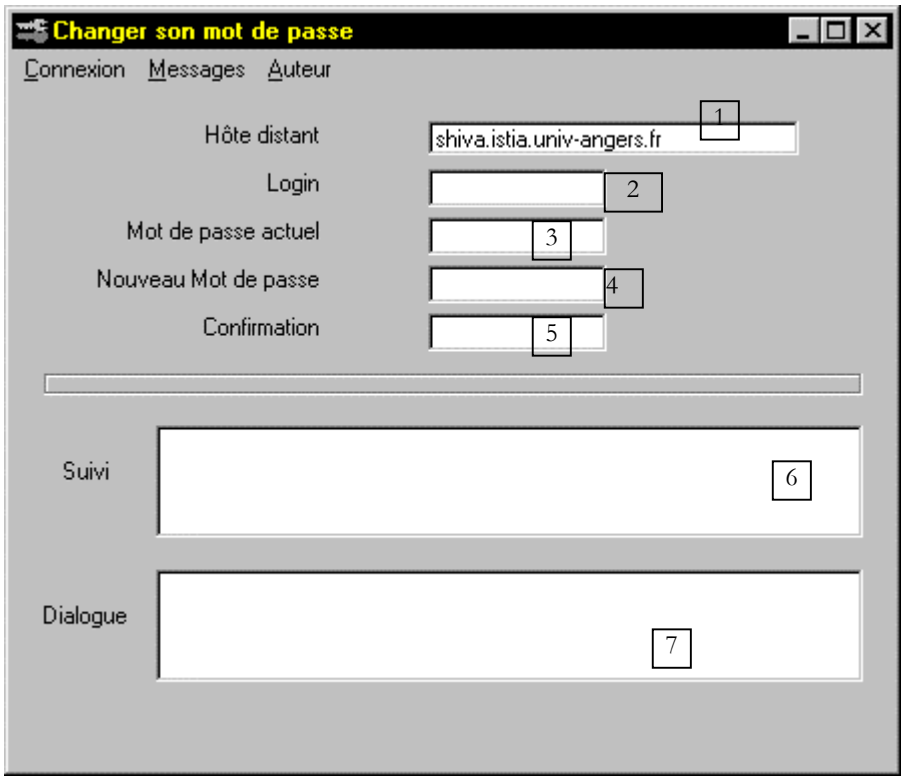
7.5.4.1 Introduction

On se propose de créer un client TCP capable de dialoguer avec le serveur POPPASSD qui travaille sur le port 106. Ce service permet de changer son mot de passe sur une machine UNIX. Le protocole du dialogue Client/Serveur est le suivant :

1 - Les dialogues se font par échange de messages terminés par la séquence RCLF
2 - Le client envoie des commandes au serveur
- Le serveur répond par des messages commençant par des nombres à 3 chiffres : XXX. Si XXX=200, la commande a été correctement exécutée, sinon il y a eu une erreur.

3 - La chronologie des échanges est la suivante :
- A - le client se connecte
  - le serveur répond par un message de bienvenue
- B - le client envoie USER login
  - le serveur répond en demandant le mot de passe si le login est accepté, par une erreur sinon
- C - le client envoie PASS mot_de_passe
  - le serveur répond en demandant le nouveau mot de passe si le mot de passe est accepté, par une erreur sinon
- D - le client envoie NEWPASS nouveau_mot_de_passe
  - le serveur répond en confirmant que le nouveau mot de passe a été accepté, par une erreur sinon
- E - le client envoie la commande QUIT
  - le serveur envoie un message de fin & ferme la connexion

7.5.4.2 Le formulaire du client



La signification des différents contrôles est la suivante :

n°	nom	type	rôle
1	txtRemoteHost	JTextField	nom du serveur
2	txtLogin	JTextField	login de l'utilisateur
3	txtMdp	JTextField	Mot de Passe de l'utilisateur
4	txtNewMdp	JTextField	Nouveau mot de passe de l'utilisateur
5	txtConfirmation	JTextField	Confirmation du nouveau mot de passe
6	lstSuivi	JList	Messages du suivi de la connexion
7	lstDialogue	JList	Messages du dialogue Client/Serveur
10	cmdAnnuler	JButton	non représenté - Bouton apparaissant lorsque la connexion au serveur est en cours. Permet de l'arrêter .

7.5.4.3 Les menus

Titre	Nom du contrôle	Rôle
Connexion	mnuconnexion	
Connecter	mnuconnecter	lance la connexion au serveur
Quitter	mnuQuitter	quitte l'application
Messages	mnuMessages	
RazSuivi	mnuRazSuivi	efface la liste lstSuivi
RazDialogue	mnuRazDialogue	efface la liste lstDialogue
Auteur	mnuAuteur	affiche la boîte de copyright

7.5.4.4 Fonctionnement de l'application

Initialisation de l'application

Lorsque la feuille principale de l'application est chargée, les actions suivantes prennent place :

- la feuille est centrée sur l'écran
- seules les options de menu *Connexion/Quitter* & *Auteur* sont actives

- le bouton *Annuler* est caché
- les listes *LstSuivi & LstDialogue* sont vides

Menu Connexion/Connecter

Cette option n'est disponible que lorsque les champs 1 à 5 ont été remplis. Un clic sur cette option entraîne les opérations suivantes :

- un thread est lancé pour assurer la connexion au serveur
- le bouton *Annuler* apparaît afin de permettre à l'utilisateur d'interrompre la connexion en cours
- toutes les options de menu sont désactivées sauf *Quitter* & *Auteur*

La séquence des événements est ensuite la suivante :

1. L'utilisateur a appuyé sur le bouton *Annuler* : on arrête le thread de connexion et on remet le menu dans son état de départ. On indique dans le suivi qu'il y a eu fermeture de la connexion par l'utilisateur.
2. La demande de connexion est acceptée par le serveur. On entame ensuite le dialogue avec le serveur pour changer le mot de passe. Les échanges de ce dialogue sont inscrits dans la liste *LstDialogue*. Une fois le dialogue terminé, la connexion avec le serveur est close et le menu du formulaire remis dans son état initial.
3. Tant que le dialogue est actif, le bouton Annuler reste présent afin de permettre à l'utilisateur de clôre la connexion s'il le veut.
4. Si au cours de la communication se produit une erreur quelconque, la connexion est close et la cause de l'erreur affichée dans la liste de suivi *LstSuivi*.

Menu Connexion/Quitter

Cette option clôt une éventuelle connexion active avec le serveur et termine l'application.

Menus RazSuivi et RazDialogue

Vident respectivement les listes *LstSuivi* et *LstDialogue*. Ces options sont inhibées lorsque les listes cooréspondantes sont vides.

Le bouton Annuler

Ce bouton situé en bas du formulaire n'apparaît que lorsque le client est en cours de connexion ou connecté au serveur. Le bouton ***Annuler*** donne à l'utilisateur, la possibilité d'interrompre la communication avec le serveur.

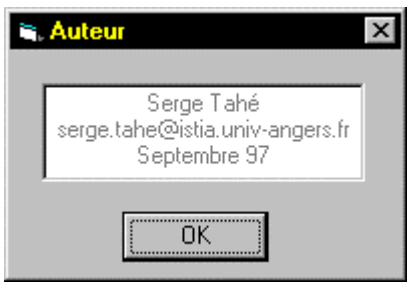
Les listes de suivi

La liste LstSuivi (***5***) fait le suivi de la connexion. Elle indique les moments clés de la connexion :
- son ouverture par le client
- sa fermeture par le serveur ou le client
- toutes les erreurs de qui peuvent se produire tant que la liaison est active

La liste LstDialogue (***6***) fait le suivi du dialogue qui s'instaure entre le client et le serveur.

L'option Auteur

Ce menu ouvre une fenêtre dite de Copyright :



Gestion des erreurs

Les erreurs de communication sont signalées dans la liste de suivi ***6***, celles liées au dialogue client/serveur dans la liste de dialogue ***7***. Lors d'une erreur de liaison, le dialogue client/serveur est fermé et le formulaire remis dans son état initial prêt pour une nouvelle connexion.

7.5.4.5 TRAVAIL A FAIRE

Réaliser le travail décrit précédemment sous forme d'aplication autonome puis d'applet.

8. JAVA RMI

8.1 Introduction

Nous avons vu comment créer des applications réseau à l'aide des outils de communication appelés *sockets*. Dans une application client/serveur bâtie sur ces outils, le lien entre le client et le serveur est le protocole de communication qu'ils ont adopté pour converser. Les deux applications peuvent être écrites avec des langages différents : Java par exemple pour le client, Perl pour le serveur ou toute autre combinaison. On a bien deux applications distinctes reliées par un protocole de communication connu des deux. Par ailleurs, l'accès au réseau via les sockets n'est pas transparent pour une application Java : elle doit utiliser la classe *Socket*, classe créée spécialement pour gérer ces outils de communication que sont les *sockets*.

JAVA RMI (Remote Method Invocation) permet de créer des applications réseau aux caractéristiques suivantes :

1. Les applications client/serveur sont des applications Java aux deux extrémités de la communication
2. Le client peut utiliser des objets situés sur le serveur comme s'ils étaient locaux
3. La couche réseau devient transparente : les applications n'ont pas à se soucier de la façon dont sont transportées les informations d'un point à un autre.

Le dernier point est un facteur de portabilité : si la couche réseau d'une application RMI venait à changer, l'application elle-même n'aurait pas à être ré-écrite. Ce sont les classes RMI du langage Java qui devront être adaptées à la nouvelle couche réseau.

Le principe d'une communication RMI est le suivant :

1. Une application Java classique est écrite sur une machine A. Elle va jouer le rôle de serveur. Pour ce faire certains de ses objets vont faire l'objet d'une « publication » sur la machine A sur laquelle l'application s'exécute et deviennent alors des services.
2. Une application Java classique est écrite sur une machine B. Elle va jouer le rôle de client. Elle aura accès aux objets/services publiés sur la machine A, c'est à dire que via une référence distante, elle va pouvoir les manipuler comme s'ils étaient locaux. Pour cela, elle aura besoin de connaître la structure de l'objet distant auquel elle veut accéder (méthodes & propriétés).

8.2 Apprenons par l'exemple

La théorie qui sous-entend l'interface RMI n'est pas simple. Pour y voir plus clair, nous allons suivre pas à pas l'écriture d'une application client/serveur utilisant le package RMI de Java. Nous prenons une application que l'on trouve dans de nombreux ouvrages sur RMI : le client invoque une unique méthode d'un objet distant qui lui renvoie alors une chaîne de caractères. Nous présentons ici, une légère variante : le serveur fait l'écho de ce que lui envoie le client. Nous avons déjà présenté dans cet ouvrage, une telle application s'appuyant elle, sur les sockets.

8.2.1 L'application serveur

8.2.1.1 Étape 1 : l'interface de l'objet/serveur

Un objet distant est une instance de classe qui doit implémenter l'interface *Remote* définie dans le package *java.rmi*. Les méthodes de l'objet qui seront accessibles à distance sont celles déclarées dans une interface dérivée de l'interface *Remote* :

```
import java.rmi.*;

// l'interface distante
public interface interEcho extends Remote{
  public String echo(String msg) throws java.rmi.RemoteException;
}
```

Ici, on déclare donc une interface *interEcho* déclarant une méthode *echo* comme accessible à distance. Cette méthode est susceptible de générer une exception de la classe *RemoteException*, classe qui regroupe tous les erreurs liées au réseau.

8.2.1.2 *Étape 2 : écriture de l'objet serveur*

Dans l'étape suivante, on définit la classe qui implémente l'interface distante précédente. Cette classe doit être dérivée de la classe *UnicastRemoteObject*, classe qui dispose des méthodes autorisant l'invocation de méthodes à distance.

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;

// classe implémentant l'écho distant
public class srvEcho extends UnicastRemoteObject implements interEcho{

  // constructeur
  public srvEcho() throws RemoteException{
    super();
  }// fin constructeur

  // méthode réalisant l'écho
  public String echo(String msg) throws RemoteException{
    return  "[" + msg + "]";
  }// fin écho
}// fin classe
```

Dans la classe précédente, nous trouvons :

1. la méthode qui fait l'écho
2. un constructeur qui ne fait rien si ce n'est appeler le constructeur de la classe mère. Il est là pour déclarer qu'il peut générer une exception de type *RemoteException*.

Nous allons créer une instance de cette classe avec une méthode *main*. Pour qu'un objet/service soit accessible de l'extérieur, il doit être créé et enregistré dans l'annuaire des objets accessibles de l'extérieur. Un client désirant accéder à un objet distant procède en effet de la façon suivante :

1. il s'adresse au service d'annuaire de la machine sur laquelle se trouve l'objet qu'il désire. Ce service d'annuaire opère sur un port que le client doit connaître (1099 par défaut). Le client demande à l'annuaire, une référence d'un objet/service dont il donne le nom. Si ce nom est celui d'un objet/service de l'annuaire, celui-ci renvoie au client une référence via laquelle le client va pouvoir dialoguer avec l'objet/service distant.
2. à partir de ce moment, le client peut utiliser cet objet distant comme s'il était local

Pour en revenir à notre serveur, nous devons créer un objet de type *srvEcho* et l'enregistrer dans l'annuaire des objets accessibles de l'extérieur. Cet enregistrement se fait avec la méthode de classe *rebind* de la classe *Naming* :

Naming.rebind(String nom, Remote obj)

avec

nom	le nom qui sera associé à l'objet distant
obj	l'objet distant

Notre classe *srvEcho* devient donc la suivante :

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;

// classe implémentant l'écho distant
public class srvEcho extends UnicastRemoteObject implements interEcho{

  // constructeur
  public srvEcho() throws RemoteException{
    super();
  }// fin constructeur

  // méthode réalisant l'écho
  public String echo(String msg) throws RemoteException{
    return  "[" + msg + "]";
  }// fin écho
```

```
  // création du service
  public static void main (String arg[]){
    try{
      srvEcho serveurEcho=new srvEcho();
      Naming.rebind("srvEcho",serveurEcho);
      System.out.println("Serveur d'écho prêt");
    } catch (Exception e){
      System.err.println(" Erreur "  + e + "  lors du lancement du serveur d'écho ");
    }
  }// main
}// fin classe
```

Lorsqu'on lit le programme précédent, on a l'impression qu'il va s'arrêter aussitôt après avoir créé et enregistré le service d'écho. Ce n'est pas le cas. Parce que la classe *srvEcho* est dérivée de la classe *UnicastRemoteObject*, l'objet créé s'exécute indéfiniment : il écoute les demandes des clients sur un port anonyme c'est à dire choisi par le système selon les circonstances. La création du service est asynchrone : dans l'exemple, la méthode *main* crée le service et continue son exécution : elle affichera bien « Serveur d'écho prêt ».

8.2.1.3 Étape 3 : compilation de l'application serveur

Au point où on en est, on peut compiler notre serveur. Nous compilons le fichier *interEcho.java* de l'interface *interEcho* ainsi que le fichier *srvEcho.java* de la classe *srvEcho*. Nous obtenons les fichiers *.class* correspondant : *interEcho.class* et *srvEcho.class*.

8.2.1.4 Étape 4 : écriture du client

On écrit un client à qui on passe en paramètre l'URL du serveur d'écho et qui

1. lit une ligne tapée au clavier
2. l'envoie au serveur d'écho
3. affiche la réponse que celui-ci envoie
4. reboucle en 1 et s'arrête lorsque la ligne tapée est « fin ».

Cela donne le client suivant :

```
import java.rmi.*;
import java.io.*;

public class cltEcho {

  public static void main(String arg[]){
    // syntaxe : cltEcho URLService

    // vérification des arguments
    if(arg.length!=1){
      System.err.println("Syntaxe : pg url_service_rmi");
      System.exit(1);
    }

    // dialogue client-serveur
    String urlService=arg[0];
    BufferedReader in=null;
    String msg=null;
    String reponse=null;
    interEcho serveur=null;

    try{
      // ouverture du flux clavier
      in=new BufferedReader(new InputStreamReader(System.in));
      // localisation du service
      serveur=(interEcho) Naming.lookup(urlService);
      // boucle de lecture des msg à envoyer au serveur d'écho
      System.out.print("Message : ");
      msg=in.readLine().toLowerCase().trim();
      while(! msg.equals("fin")){
        // envoi du msg au serveur et réception de la réponse
        reponse=serveur.echo(msg);
        // suivi
        System.out.println("Réponse serveur : " + reponse);
        // msg suivant
        System.out.print("Message : ");
        msg=in.readLine().toLowerCase().trim();
      }// while
      // c'est fini
      System.exit(0);
    // gestion des erreurs
    } catch (Exception e){
```

```
      System.err.println("Erreur : " + e);
      System.exit(2);
    }// try
  }// main
}// classe
```

Il n'y a rien de bien particulier dans ce client si ce n'est l'instruction qui demande une référence du serveur :

```
      serveur=(interEcho) Naming.lookup(urlService);
```

On se rappelle que notre service d'écho a été enregistré dans l'annuaire des services de la machine où il se trouve avec l'instruction :

```
      Naming.rebind("srvEcho",serveurEcho);
```

Le client utilise donc lui aussi une méthode de la classe *Naming* pour obtenir une référence du serveur qu'il veut utiliser. La méthode **lookup** utilisée admet comme paramètre l'url du service demandé. Celle-ci a la forme d'une url classique :

rmi://machine:port/nom_service

avec

rmi	facultatif - protocole rmi
machine	nom ou adresse IP de la machine sur laquelle opère le serveur d'écho - facultatif, par défaut *localhost*.
port	port d'écoute du service d'annuaire de cette machine - facultatif, par défaut *1099*
nom_service	nom sous lequel a été enregistré le service demandé (*srvEcho* pour notre exemple)

Ce qui est récupéré, c'est une instance de l'interface distante *interEcho*. Si on suppose que le client et le serveur ne sont pas sur la même machine, lorsqu'on compile le client *cltEcho.java*, on doit disposer dans le même répertoire, du fichier *interEcho.class*, résultat de la compilation de l'interface distante *interEcho*, sinon on aura une erreur de compilation sur les lignes qui référencent cette interface.

8.2.1.5 Étape 5 : génération des fichiers .class nécessaires à l'application client-serveur

Afin de bien comprendre ce qui est du côté serveur et ce qui est du côté client, on mettra le serveur dans un répertoire *echo\serveur* et le client dans un répertoire *echo\client*.

Le répertoire du serveur contient les fichiers source suivants :

```
E:\data\java\RMI\echo\serveur>dir *.java

INTERE~1 JAV           158  09/03/99  15:06 interEcho.java
SRVECH~1 JAV           759  09/03/99  15:07 srvEcho.java
```

Après compilation de ces deux fichiers source, on a les fichiers *.class* suivants :

```
E:\data\java\RMI\echo\serveur>dir *.class

SRVECH~1 CLA         1 129  09/03/99  15:58 srvEcho.class
INTERE~1 CLA           256  09/03/99  15:58 interEcho.class
```

Dans le répertoire du client, on trouve le fichier source suivant :

```
E:\data\java\RMI\echo\client>dir *.java

CLTECH~1 JAV         1 427  09/03/99  16:08 cltEcho.java
```

ainsi que le fichier *interEcho.class* qui a été généré lors de la compilation du serveur :

```
E:\data\java\RMI\echo\client>dir *.class

INTERE~1 CLA           256  09/03/99  15:59 interEcho.class
```

Après compilation du fichier source on a les fichiers *.class* suivants :

```
E:\data\java\RMI\echo\client>dir *.class
```

```
CLTECH~1 CLA         1 506  09/03/99  16:08 cltEcho.class
INTERE~1 CLA           256  09/03/99  15:59 interEcho.class
```

Si on tente d'exécuter le client *cltEcho*, on obtient l'erreur suivante :

```
E:\data\java\RMI\echo\client>j:\jdk12\bin\java cltEcho rmi://localhost/srvEcho
Erreur : java.rmi.UnmarshalException: error unmarshalling return; nested exception is:
        java.lang.ClassNotFoundException: srvEcho_Stub
```

Si on tente d'exécuter le serveur *srvEcho*, on obtient l'erreur suivante :

```
E:\data\java\RMI\echo\serveur>j:\jdk12\bin\java srvEcho
Erreur java.rmi.StubNotFoundException: Stub class not found: srvEcho_Stub; nested exception is:
        java.lang.ClassNotFoundException: srvEcho_Stub  lors du lancement du serveur d'écho
```

Dans les deux cas, la machine virtuelle Java indique qu'elle n'a pas trouvé la classe *srvEcho_stub*. Effectivement, nous n'avons encore jamais entendu parler de cette classe. Dans le client, la localisation du serveur s'est faite avec l'instruction suivante :

```
      serveur=(interEcho) Naming.lookup(urlService);
```

Ici, *urlservice* est la chaîne `rmi://localhost/srvEcho` avec

rmi	protocole rmi
localhost	machine où opère le serveur - ici la même machine sur laquelle le client. La syntaxe est normalement machine:port. En l'absence du port, c'est le port 1099 qui sera utilisé par défaut. A l'écoute de ce port, se trouve le service d'annuaire du serveur.
srvEcho	c'est le nom du service particulier demandé

A la compilation, aucune erreur n'avait été signalée. Il fallait simplement que le fichier *interEcho.class* de l'interface distante soit disponible.

A l'exécution, la machine virtuelle réclame la présence d'un fichier *srvEcho_stub.class* si le service demandé est le service *srvEcho*, de façon générale un fichier *X_stub.class* pour un service *X*. Ce fichier n'est nécessaire qu'à l'exécution pas à la compilation du client. Il en est de même pour le serveur. Qu'est-ce donc que ce fichier ?

Sur le serveur, se trouve la classe *srvEcho.class* qui est notre objet/service distant. Le client, s'il n'a pas besoin de cette classe, a néammoins besoin d'une sorte d'image d'elle afin de pouvoir communiquer avec. En fait, le client n'adresse pas directement ses requêtes à l'objet distant : il les adresse à son image locale *srvEcho_stub.class* située sur la même machine que lui. Cette image locale *srvEcho_stub.class* dialogue avec une image de même nature (*srvEcho_stub.class*) située cette fois sur le serveur. Cette image est créée à partir du fichier *.class* du serveur avec un outil de Java appelé *rmic*. Sous Windows, la commande :

```
E:\data\java\RMI\echo\serveur>j:\jdk12\bin\rmic srvEcho
```

va produire, à partir du fichier *srvEcho.class* deux autres fichiers *.class* :

```
E:\data\java\RMI\echo\serveur>dir *.class

SRVECH~2 CLA         3 264  09/03/99  16:57 srvEcho_Stub.class
SRVECH~3 CLA         1 736  09/03/99  16:57 srvEcho_Skel.class
```

Il y a bien là, le fichier *srvEcho_stub.class* dont le client et le serveur ont besoin à l'exécution. Il y a également un fichier *srvEcho_Skel.class* dont pour l'instant on ignore le rôle. On fait une copie du fichier *srvEcho_stub.class* dans le répertoire du client et du serveur et on supprime le fichier *srvEcho_Skel.class*. On a donc les fichiers suivants :

du côté serveur :

```
E:\data\java\RMI\echo\serveur>dir *.class

SRVECH~1 CLA         1 129  09/03/99  15:58 srvEcho.class
INTERE~1 CLA           256  09/03/99  15:58 interEcho.class
SRVECH~1 CLA         3 264  09/03/99  16:01 srvEcho_Stub.class
```

du côté client :

```
E:\data\java\RMI\echo\client>dir *.class

CLTECH~1 CLA         1 506  09/03/99  16:08 cltEcho.class
INTERE~1 CLA           256  09/03/99  15:59 interEcho.class
SRVECH~1 CLA         3 264  09/03/99  16:01 srvEcho_Stub.class
```

8.2.1.6 Étape 6 : Exécution de l'application client-serveur d'écho

Nous sommes prêts pour exécuter notre application client-serveur. Dans un premier temps, le client et le serveur fonctionneront sur la même machine. Il faut tout d'abord lancer notre application serveur. On se rappelle que celle-ci :

- crée le service
- l'enregistre dans l'annuaire des services de la machine sur laquelle opère le serveur d'écho

Ce dernier point nécessite la présence d'un service d'annuaire. Celui-ci est lancé par la commande :

```
start j:\jdk12\bin\rmiregistry
```

rmiregistry est le service d'annuaire. Il est ici lancé en tâche de fond dans une fenêtre Dos de Windows par la commande **start**. L'annuaire actif, on peut créer le service d'écho et l'enregistrer dans l'annuaire des services. Là encore, il est lancé en tâche de fond par une commande *start* :

```
E:\data\java\RMI\echo\serveur>start j:\jdk12\bin\java srvEcho
```

Le serveur d'écho s'exécute dans une nouvelle fenêtre DOS et affiche comme on le lui avait demandé :

```
Serveur d'écho prêt
```

Il ne nous reste plus qu'à lancer et tester notre client :

```
E:\data\java\RMI\echo\client>j:\jdk12\bin\java cltEcho rmi://localhost/srvEcho
Message : msg1
Réponse serveur : [msg1]
Message : msg2
Réponse serveur : [msg2]
Message : fin
```

8.2.1.7 Le client et le serveur sur deux machines différentes

Dans l'exemple précédent, le client et le serveur étaient sur une même machine. On les place maintenant sur des machines différentes :

- le serveur sur une machine windows
- le client sur une machine linux

Le serveur est lancé commé précédemment sur la machine Windows. Sur la machine linux, on a transféré les fichiers .class du client :

```
shiva[serge]:/home/admin/serge/java/rmi/client#
$ dir
total 9
drwxr-xr-x   2 serge    admin        1024 Mar 10 10:02 .
drwxr-xr-x   4 serge    admin        1024 Mar 10 10:01 ..
-rw-r--r--   1 serge    admin        1506 Mar 10 10:02 cltEcho.class
-rw-r--r--   1 serge    admin         256 Mar 10 10:02 interEcho.class
-rw-r--r--   1 serge    admin        3264 Mar 10 10:02 srvEcho_Stub.class
```

Le client est lancé :

```
$ java cltEcho rmi://tahe.istia.univ-angers.fr/srvEcho
Message : msg1
```

```
Erreur : java.rmi.ServerException: RemoteException occurred in server thread; nested exception is:
      java.rmi.UnmarshalException: error unmarshalling call header; nested exception is:
      java.rmi.UnmarshalException: skeleton class not found but required for client version
```

On a donc une erreur : la machine virtuelle Java réclame apparemment le fichier *srvEcho_skel.class* qui avait été produit par l'utilitaire *rmic* mais qui jusqu'à maintenant n'avait pas servi. On le recrée et on le transfère également sur la machine linux :

```
shiva[serge]:/home/admin/serge/java/rmi/client#
$ dir
total 11
drwxr-xr-x   2 serge    admin        1024 Mar 10 10:17 .
drwxr-xr-x   4 serge    admin        1024 Mar 10 10:01 ..
-rw-r--r--   1 serge    admin        1506 Mar 10 10:02 cltEcho.class
-rw-r--r--   1 serge    admin         256 Mar 10 10:02 interEcho.class
-rw-r--r--   1 serge    admin        1736 Mar 10 10:17 srvEcho_Skel.class
-rw-r--r--   1 serge    admin        3264 Mar 10 10:02 srvEcho_Stub.class
```

On a la même erreur que précédemment... Alors on réfléchit et on relit les docs sur RMI. On finit par se dire que c'est peut-être le serveur lui-même qui a besoin du fameux fichier *srvEcho_Skel.class*. On relance alors, sur la machine Windows, le serveur avec les deux fichiers *srvEcho_Stub.class* et *srvEcho_Skel.class* présents *:*

```
E:\data\java\RMI\echo\serveur>dir *.class

SRVECH~1 CLA         1 129  09/03/99  15:58 srvEcho.class
INTERE~1 CLA           256  09/03/99  15:58 interEcho.class
SRVECH~2 CLA         3 264  10/03/99   9:05 srvEcho_Stub.class
SRVECH~3 CLA         1 736  10/03/99   9:05 srvEcho_Skel.class

E:\data\java\RMI\echo\serveur>start j:\jdk12\bin\java srvEcho
```

puis sur la machine linux, on teste de nouveau le client et cette fois-ci, ça marche :

```
shiva[serge]:/home/admin/serge/java/rmi/client#
$ dir *.class
-rw-r--r--   1 serge    admin        1506 Mar 10 10:02 cltEcho.class
-rw-r--r--   1 serge    admin         256 Mar 10 10:02 interEcho.class
-rw-r--r--   1 serge    admin        3264 Mar 10 10:02 srvEcho_Stub.class

shiva[serge]:/home/admin/serge/java/rmi/client#
$ java cltEcho rmi://tahe.istia.univ-angers.fr/srvEcho
Message : msg1
Réponse serveur : [msg1]
Message : msg2
Réponse serveur : [msg2]
Message : fin
```

On en déduit donc, que côté serveur les deux fichiers *srvEcho_Stub.class* et *srvEcho_Skel.class* doivent être présents. Côté client, seul le fichier *srvEcho_Stub.class* a été nécessaire jusqu'à maintenant. Il s'était avéré indispensable lorsque le client et le serveur étaient sur la même machine Windows. Sous linux, on l'enlève pour voir...

```
shiva[serge]:/home/admin/serge/java/rmi/client#
$ dir *.class
-rw-r--r--   1 serge    admin        1506 Mar 10 10:02 cltEcho.class
-rw-r--r--   1 serge    admin         256 Mar 10 10:02 interEcho.class

shiva[serge]:/home/admin/serge/java/rmi/client#
$ java cltEcho rmi://tahe.istia.univ-angers.fr/srvEcho
*** Security Exception: No security manager, stub class loader disabled ***
java.rmi.RMISecurityException: security.No security manager, stub class loader disabled
        at sun.rmi.server.RMIClassLoader.getClassLoader(RMIClassLoader.java:84)
        at sun.rmi.server.MarshalInputStream.resolveClass(MarshalInputStream.java:88)
        at java.io.ObjectInputStream.inputClassDescriptor(ObjectInputStream.java)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java)
        at java.io.ObjectInputStream.inputObject(ObjectInputStream.java)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java)
        at sun.rmi.registry.RegistryImpl_Stub.lookup(RegistryImpl_Stub.java:105)
        at java.rmi.Naming.lookup(Naming.java:60)
        at cltEcho.main(cltEcho.java:28)
Erreur : java.rmi.UnexpectedException: Unexpected exception; nested exception is:
        java.rmi.RMISecurityException: security.No security manager, stub class loader disabled
```

On a une erreur intéressante qui a l'air de dire que la machine virtuelle Java a voulu charger la fameuse classe *stub* mais qu'il a échoué en l'absence d'un « security manager ». On se dit qu'on a vu quelque chose sur ce thème dans les docs. On s'y replonge... et on trouve que le serveur doit créer et installer un gestionnaire de sécurité (security manager) qui garantisse aux clients qui demandent le chargement de classes que celles-ci sont saines. En l'absence de ce *security manager*, ce chargement de classes est impossible. Ca semble coller : notre client linux a demandé la classe *srvEcho_stub.class* dont il a besoin au serveur et celui-ci a refusé en disant qu'aucun gestionnaire de sécurité n'avait été installé. On modifie donc le code de la fonction *main* du serveur de la façon suivante :

```
// création du service
public static void main (String arg[]){

  // installation d'un gestionnaire de sécurité
  System.setSecurityManager(new RMISecurityManager());

  // lancement et enregistrement du service
  try{
    srvEcho serveurEcho=new srvEcho();
    Naming.rebind("srvEcho",serveurEcho);
    System.out.println("Serveur d'écho prêt");
  } catch (Exception e){
    System.err.println(" Erreur "  + e + "  lors du lancement du serveur d'écho ");
  }
}// main
```

On compile et on produit les fichiers *srvEcho_stub.class* et *srvEcho_Skel.class* avec l'outil *rmic*. On lance le service d'annuaire (*rmiregistry*) puis le serveur et on a une erreur qu'auparavant on n'avait pas !

```
Erreur java.security.AccessControlException: access denied (java.net.SocketPermission 127.0.0.1:1099
connect,resolve)  lors du lancement du serveur d'écho
```

Le gestionnaire de sécurité semble avoir été trop efficace. On lit de nouveau les docs... On s'aperçoit que lorsqu'un gestionnaire de sécurité est actif, il faut préciser au lancement d'un programme ses droits. Cela se fait avec l'option suivante :

```
start j:\jdk12\bin\java -Djava.security.policy=mypolicy srvEcho
```

où

java.security.policy est un mot clé

mypolicy est un fichier texte définissant les droits du programme. Ici, c'est le suivant :

```
grant {
   // Allow everything for now
   permission java.security.AllPermission;
};
```

Le programme a ici tous les droits.

On recommence. On se place dans le répertoire du serveur et on fait successivement :

- lancement du service d'annuaire : `start j:\jdk12\bin\rmiregistry`
- lancement du serveur : `start j:\jdk12\bin\java -Djava.security.policy=mypolicy srvEcho`

Et cette fois, le serveur d'écho (le client pas encore) se lance correctement. Maintenant vous pouvez faire l'expérience suivante :

- arrêtez le serveur d'écho puis le service d'annuaire
- relancez le service d'annuaire en étant dans un autre répertoire que celui du serveur
- revenez sur le répertoire du serveur lancez le serveur d'écho - vous obtenez l'erreur suivante :

```
Erreur java.rmi.ServerException: RemoteException occurred in server thread; nes
ted exception is:
java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:
java.lang.ClassNotFoundException: srvEcho_Stub  lors du lancement du serveur d'écho
```

On en déduit que le répertoire à partir duquel le service d'annuaire est lancé a de l'importance. Ici, Java n'a pas trouvé la classe *srvEcho_stub.class* parce que le service d'annuaire n'a pas été lancé du répertoire du serveur. Lors du lancement du serveur, on peut préciser dans quel répertoire se trouvent les classes nécessaires au serveur :

```
start j:\jdk12\bin\java
-Djava.security.policy=mypolicy
-Djava.rmi.server.codebase=file:/e:/data/java/rmi/echo/serveur/
srvEcho
```

La commande est sur une unique ligne. Le mot clé `java.rmi.server.codebase` sert à indiquer l'URL du répertoire contenant les classes nécessaires au serveur. Ici, cette URL précise le protocole `file` qui est le protocole d'accès aux fichiers locaux et le répertoire contenant les fichiers *.class* du serveur. Si donc on procède comme suit :

- arrêt du service d'annuaire
- relancer le service d'annuaire à partir d'un autre répertoire que celui du serveur
- dans le répertoire du serveur, lancer celui-ci avec la commande (une seule ligne) :

```
 start j:\jdk12\bin\java
-Djava.security.policy=mypolicy
-Djava.rmi.server.codebase=file:/e:/data/java/rmi/echo/serveur/
srvEcho
```

Le serveur est alors lancé correctement. On peut donc passer au client. On le teste :

```
shiva[serge]:/home/admin/serge/java/rmi/client#
$ dir *.class
-rw-r--r--   1 serge    admin        1506 Mar 10 14:28 cltEcho.class
-rw-r--r--   1 serge    admin         256 Mar 10 10:02 interEcho.class

shiva[serge]:/home/admin/serge/java/rmi/client#
$ java cltEcho rmi://tahe.istia.univ-angers.fr/srvEcho
*** Security Exception: No security manager, stub class loader disabled ***
java.rmi.RMISecurityException: security.No security manager, stub class loader disabled
        at sun.rmi.server.RMIClassLoader.getClassLoader(RMIClassLoader.java:84)
        at sun.rmi.server.MarshalInputStream.resolveClass(MarshalInputStream.java:88)
        at java.io.ObjectInputStream.inputClassDescriptor(ObjectInputStream.java)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java)
        at java.io.ObjectInputStream.inputObject(ObjectInputStream.java)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java)
        at sun.rmi.registry.RegistryImpl_Stub.lookup(RegistryImpl_Stub.java:105)
        at java.rmi.Naming.lookup(Naming.java:60)
        at cltEcho.main(cltEcho.java:31)
Erreur : java.rmi.UnexpectedException: Unexpected exception; nested exception is:
        java.rmi.RMISecurityException: security.No security manager, stub class loader disabled
```

On obtient la même erreur signalant l'absence d'un gestionnaire de sécurité. On se dit qu'on s'est peut-être trompé et que c'est le client qui doit se créer son gestionnaire de sécurité. On laisse le serveur avec son gestionnaire de sécurité mais on en crée un pour le client également. La fonction main du client *cltEcho.java* devient alors :

```
public static void main(String arg[]){
  // syntaxe : cltEcho machine port
  // machine : machine où opère le serveur d'écho
  // port : port où opère l'annuaire des services sur la machine du service d'écho

  // vérification des arguments
  if(arg.length!=1){
    System.err.println("Syntaxe : pg url_service_rmi");
    System.exit(1);
  }

  // installation d'un gestionnaire de sécurité
  System.setSecurityManager(new RMISecurityManager());

  // dialogue client-serveur
  String urlService=arg[0];
  BufferedReader in=null;
  String msg=null;
  String reponse=null;
  interEcho serveur=null;

  try{
    ....
  } catch (Exception e){
    ....
  }// try
}// main
```

On procède ensuite comme suit :

- on recompile *cltEcho.java*
- on transfère les fichiers .class sur la machine linux

```
shiva[serge]:/home/admin/serge/java/rmi/client#
$ dir *.class
-rw-r--r--   1 serge    admin        1506 Mar 10 14:28 cltEcho.class
-rw-r--r--   1 serge    admin         256 Mar 10 10:02 interEcho.class
```

- on lance le client

```
$ java cltEcho rmi://tahe.istia.univ-angers.fr/srvEcho

java.io.FileNotFoundException: /e:/data/java/rmi/echo/serveur/srvEcho_Stub.class
        at java.io.FileInputStream.<init>(FileInputStream.java)
        at sun.net.www.protocol.file.FileURLConnection.connect(FileURLConnection.java:150)
        at sun.net.www.protocol.file.FileURLConnection.getInputStream(FileURLConnection.java:170)
        at sun.applet.AppletClassLoader.loadClass(AppletClassLoader.java:119)
        at sun.applet.AppletClassLoader.findClass(AppletClassLoader.java:496)
        at sun.applet.AppletClassLoader.loadClass(AppletClassLoader.java:199)
        at sun.rmi.server.RMIClassLoader.loadClass(RMIClassLoader.java:159)
        at sun.rmi.server.MarshalInputStream.resolveClass(MarshalInputStream.java:97)
        at java.io.ObjectInputStream.inputClassDescriptor(ObjectInputStream.java)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java)
        at java.io.ObjectInputStream.inputObject(ObjectInputStream.java)
        at java.io.ObjectInputStream.readObject(ObjectInputStream.java)
        at sun.rmi.registry.RegistryImpl_Stub.lookup(RegistryImpl_Stub.java:105)
        at java.rmi.Naming.lookup(Naming.java:60)
        at cltEcho.main(cltEcho.java:31)
File not found when looking for: srvEcho_Stub
Erreur : java.rmi.UnmarshalException: Return value class not found; nested exception is:
        java.lang.ClassNotFoundException: srvEcho_Stub
```

Malgré les apparences, on progresse : l'erreur n'est plus la même. On constate que le client a pu demander au serveur la classe *srvEcho_Stub.class*, mais que celui-ci ne l'a pas trouvé. Donc, le client doit avoir un gestionnaire de sécurité s'il veut pouvoir demander des classes au serveur.

Si on regarde l'erreur précédente, on voit que le fichier *srvEcho_Stub.class* a été cherché dans le répertoire `e:/data/java/rmi/echo/serveur/` et qu'il n'a pas été trouvé. C'est pourtant bien là qu'il est. Si on regarde plus précisément, la liste des méthodes impliquées dans l'erreur, on trouve celle-ci : `sun.net.www.protocol.file.FileURLConnection.getInputStream`. Le client semble avoir ouvert un flux avec un objet de type *FileURLConnection*. On se dit que tout ça a un rapport avec la façon dont on a lancé notre serveur :

```
start j:\jdk12\bin\java
-Djava.security.policy=mypolicy
-Djava.rmi.server.codebase=file:/e:/data/java/rmi/echo/serveur/
srvEcho
```

Le message d'erreur semble faire référence à la valeur du mot clé *java.rmi.server.codebase*. Lorsqu'on regarde les docs de nouveau, on voit que la valeur de ce mot clé est, dans les exemples donnés, toujours : http://.., c.a.d. que le protocole utilisé est http. Il n'apparaît pas clairement comment le client demande et obtient ses classes auprès du serveur. Peut-être les demande-t-il avec l'URL du mot clé *java.rmi.server.codebase*, URL précisée au lancement du serveur. On décide donc de lancer le serveur par la nouvelle commande suivante :

```
start j:\jdk12\bin\java -Djava.security.policy=mypolicy
-Djava.rmi.server.codebase=http://tahe.istia.univ-angers.fr/rmi/echo/ srvEcho
```

Le protocole est maintenant http. On est obligé de déplacer les fichiers *.class* à un endroit accessible au serveur http de la machine sur laquelle seront stockées les classes. Dans notre exemple, le serveur tourne sur une machine windows avec un serveur http PWS de Microsoft. La racine de ce serveur est *d:\Inetpub\wwwroot*. Aussi procède-t-on de la façon suivante :

- on crée le répertoire *d:\Inetpub\wwwroot\rmi\echo*
- on y place les fichiers *.class* du serveur ainsi que le fichier *mypolicy*
- on lance le serveur Web si ce n'est déjà fait
- on relance le service d'annuaire (rmiregistry)
- on relance le serveur avec la commande

```
start j:\jdk12\bin\java -Djava.security.policy=mypolicy
-Djava.rmi.server.codebase=http://tahe.istia.univ-angers.fr/rmi/echo/ srvEcho
```

- sur la machine linux, on lance le client :

```
shiva[serge]:/home/admin/serge/java/rmi/client#
$ dir *.class
-rw-r--r--   1 serge    admin        1622 Mar 10 14:37 cltEcho.class
-rw-r--r--   1 serge    admin         256 Mar 10 10:02 interEcho.class

shiva[serge]:/home/admin/serge/java/rmi/client#
$ java cltEcho rmi://tahe.istia.univ-angers.fr/srvEcho
Message : msg1
Réponse serveur : [msg1]
Message : msg2
Réponse serveur : [msg2]
Message : fin
```

Ouf ! Ca marche. Le client a bien réussi à récupérer le fameux fichier *srvEcho_Stub.class*.

Tout ça nous a donné des idées, et on se demande si le client qui se trouve sur la machine Windows du serveur, fonctionnerait lui aussi sans le fichier *srvEcho_Stub.class*. On passe dans le répertoire du client, on supprime le fichier *srvEcho_Stub.class* s'il s'y trouve et on lance le client de la même façon que sous Linux :

```
E:\data\java\RMI\echo\client>dir *.class

CLTECH~1 CLA         1 622  10/03/99  14:12 cltEcho.class
INTERE~1 CLA           256  09/03/99  15:59 interEcho.class

E:\data\java\RMI\echo\client>j:\jdk12\bin\java -Djava.security.policy=mypolicy cltEcho
rmi://tahe.istia.univ-angers.fr/srvEcho

Message : nouveau message
Réponse serveur : [nouveau message]
Message : fin
```

8.2.1.8 Résumé

Côté serveur Windows :

- le serveur a un gestionnaire de sécurité
- il a été lancé avec des options : `start j:\jdk12\bin\java -Djava.security.policy=mypolicy -Djava.rmi.server.codebase=http://tahe.istia.univ-angers.fr/rmi/echo/ srvEcho`

Côté client Linux ou Windows

- le client a un gestionnaire de sécurité
- sur linux, il a été lancé par `java cltEcho rmi://tahe.istia.univ-angers.fr/srvEcho`
- sur windows, il a été lancé par `j:\jdk12\bin\java -Djava.security.policy=mypolicy cltEcho rmi://tahe.istia.univ-angers.fr/srvEcho`

8.2.1.9 Serveur d'écho sur linux, clients sur Windows et Linux

On passe maintenant le serveur sur une machine Linux et on teste des clients linux et windows. La démarche à suivre est la suivante :

- on passe les fichiers *.class* du serveur sur la machine linux

```
shiva[serge]:/home/admin/serge/WWW/rmi/echo/serveur#
$ dir
total 11
drwxr-xr-x   2 serge    admin        1024 Mar 10 16:15 .
drwxr-xr-x   3 serge    admin        1024 Mar 10 16:09 ..
-rw-r--r--   1 serge    admin         256 Mar 10 16:09 interEcho.class
```

```
-rw-r--r--   1 serge    admin        1245 Mar 10 16:09 srvEcho.class
-rw-r--r--   1 serge    admin        1736 Mar 10 16:09 srvEcho_Skel.class
-rw-r--r--   1 serge    admin        3264 Mar 10 16:09 srvEcho_Stub.class
```

- parce que la classe *srvEcho_Stub.class* va être demandée par les clients, le répertoire choisi pour les classes du serveur est un répertoire accessible au serveur http de la machine Linux. Ici l'URL de ce répertoire est *http://shiva.istia.univ-angers.fr/~serge/rmi/echo/serveur*
- le service d'annuaire est lancé en tâche de fond : */usr/local/bin/jdk/rmiregistry &*
- le serveur est lancé en tâche de fond : */usr/local/bin/jdk/bin/java -Djava.rmi.server.codebase=http://shiva.istia.univ-angers.fr/~serge/rmi/echo/serveur/ srvEcho &*

On peut tester les clients. Le client Windows d'abord.

- on passe dans le répertoire du client sur la machine windows
- on lance le client par la commande :

```
E:\data\java\RMI\echo\client>j:\jdk12\bin\java -Djava.security.policy=mypolicy cltEcho
rmi://shiva.istia.univ-angers.fr/srvEcho
Message : msg1
Réponse serveur : [msg1]
Message : fin
```

On teste le client Linux :

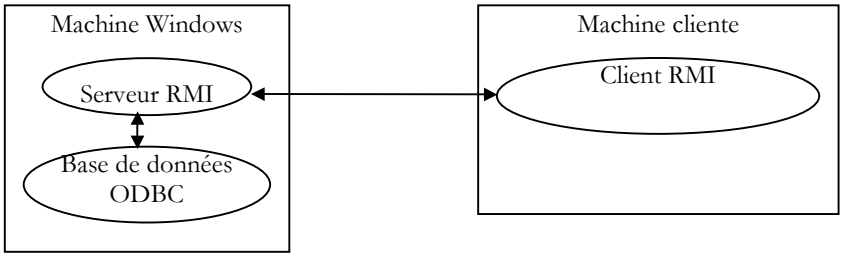
```
shiva[serge]:/home/admin/serge/java/rmi/echo/client#
$ java cltEcho srvEcho
Message : msg1
Réponse serveur : [msg1]
Message : msg2
Réponse serveur : [msg2]
Message : fin
```

A noter que pour le client linux qui fonctionne sur la même machine que le serveur d'écho, il n'a pas été besoin de préciser de machine dans l'URL du service demandé.

8.3 Deuxième exemple : Serveur SQL sur machine Windows

8.3.1 Le problème

Nous avons vu dans le chapitre JDBC comment gérer des bases de données relationnelles. Dans les exemples présentés, les applications et la base de données utilisée étaient sur la même machine Windows. On se propose ici, d'écrire un serveur RMI sur une machine Windows, qui permettrait aux clients distants d'exploiter les bases ODBC publiques du poste sur lequel se trouve le serveur.



Le client RMI pourrait faire 3 opérations :

- se connecter à la base de son choix
- émettre des requêtes SQL
- fermer la connexion

Le serveur exécute les requêtes SQL du client et lui envoie les résultats. C'est son travail essentiel et c'est pourquoi nous l'appellerons un serveur SQL.

Nous appliquons les différentes étapes vues précédemment avec le serveur d'écho.

8.3.2 Étape 1 : l'interface distante

L'interface distante est l'interface qui liste les méthodes du serveur RMI qui seront accessibles aux clients RMI. Nous utiliserons l'interface suivante :

```
import java.rmi.*;

// l'interface distante
public interface interSQL extends Remote{
  public String connect(String pilote, String url, String id, String mdp)
    throws java.rmi.RemoteException;
  public String[] executeSQL(String requete, String separateur)
    throws java.rmi.RemoteException;
  public String close()
    throws java.rmi.RemoteException;
}
```

Le rôle des différentes méthodes est le suivant :

connect le client se connecte à une base de données distante dont il donne le *pilote*, l'*url* JDBC ainsi que son identité *id* et son mot de passe *mdp* pour accéder à cette base. Le serveur lui rend une chaîne de caractères indiquant le résultat de la connexion :

200 - Connexion réussie
500 - Echec de la connexion

executeSQL le client demande l'exécution d'une requête SQL sur la base à laquelle il est connecté. Il indique le caractère qui doit séparer les champs dans les résultats qui lui sont renvoyés. Le serveur renvoie un tableau de chaînes :

100 n
pour une requête de mise à jour de la base, n étant le nombre de lignes mises à jour
500 msg d'erreur
si la requête a généré une erreur
501 Pas de résultats
si la requête n'a généré aucun résultat
101 ligne1
101 ligne2
101 ...
si la requête a généré des résultats. Les lignes ainsi renvoyées par le serveur sont les lignes résultats de la requête.

close le client ferme sa connexion avec la base distante. Le serveur renvoie une chaîne indiquant le résultat de cette fermeture :

200 Base fermée
500 Erreur lors de la fermeture de la base (msg d'erreur)

8.3.3 Étape 2 : Écriture du serveur

Le source Java du serveur SQL suit. Sa compréhension nécessite d'avoir assimilé la gestion JDBC des bases de données ainsi que la construction des serveurs RMI. Les commentaires du programme devraient faciliter sa compréhension.

```
// packages importés
import java.rmi.*;
import java.rmi.server.*;
import java.sql.*;
import java.util.*;

// classe srvSQL
public class srvSQL extends UnicastRemoteObject implements interSQL{

  // données globales de la classe
  private Connection DB;

  // ------------- constructeur
  public srvSQL() throws RemoteException{
    super();
  }

  // --------------- connect
  public String connect(String pilote, String url, String id,
    String mdp) throws RemoteException{

    // connexion à la base url grâce au driver pilote
    // identification avec identité id et mot de passe mdp

    String resultat=null;    // résultat de la méthode
    try{
      // chargement du pilote
      Class.forName(pilote);
      // demande de connexion
      DB=DriverManager.getConnection(url,id,mdp);
      // ok
      resultat="200 Connexion réussie";
    } catch (Exception e){
      // erreur
      resultat="500 Echec de la connexion (" + e + ")";
    }
    // fin
    return resultat;
  }

  // ------------- executeSQL
  public String[] executeSQL(String requete, String separateur)
    throws RemoteException{

    // exécute une requête SQL sur la base DB
    // et met les résultats dans un tableau de chaînes

    // données nécessaires à l'exécution de la requête
    Statement S=null;
    ResultSet RS=null;
    String[] lignes=null;
    Vector resultats=new Vector();
    String ligne=null;

    try{
      // création du conteneur de la requête
      S=DB.createStatement();
      // exécution requête
      if (! S.execute(requete)){
        // requête de mise à jour
        // on renvoie le nombre de lignes mises à jour
        lignes=new String[1];
        lignes[0]="100 "+S.getUpdateCount();
        return lignes;
      }
      // c'était une requête d'interrogation
      // on récupère les résultats
      RS=S.getResultSet();
      // nombre de champs du Resultset
      int nbChamps=RS.getMetaData().getColumnCount();
      // on les exploite
      while(RS.next()){
        // création de la ligne des résultats
        ligne="101 ";
        for (int i=1;i<nbChamps;i++)
          ligne+=RS.getString(i)+separateur;
        ligne+=RS.getString(nbChamps);
        // ajout au vecteur des résultats
        resultats.addElement(ligne);
      }// while
      // fin de l'exploitation des résultats
      // on libère les ressources
      RS.close();
      S.close();
      // on rend les résultats
      int nbLignes=resultats.size();
      if (nbLignes==0){
```

```
        lignes=new String[1];
        lignes[0]="501 Pas de résultats";
      } else {
        lignes=new String[resultats.size()];
        for(int i=0;i<lignes.length;i++)
          lignes[i]=(String) resultats.elementAt(i);
      }//if
      return lignes;
    } catch (Exception e){
      // erreur
      lignes=new String[1];
      lignes[0]="500 " + e;
      return lignes;
    }// try-catch
  }// executeSQL

  // --------------- close
  public String close() throws RemoteException {
    // ferme la connexion à la base de données
    String resultat=null;
    try{
      DB.close();
      resultat="200 Base fermée";
    } catch (Exception e){
      resultat="500 Erreur à la fermeture de la base ("+e+")";
    }
    // renvoi du résultat
    return resultat;
  }

  // ----------- main
  public static void main (String[] args){

    // gestionnaire de sécurité
    System.setSecurityManager(new RMISecurityManager());

    // lancement du service
    srvSQL serveurSQL=null;
    try{
      // création
      serveurSQL=new srvSQL();
      // enregistrement
      Naming.rebind("srvSQL",serveurSQL);
      // suivi
      System.out.println("Serveur SQL prêt");
    } catch (Exception e){
      // erreur
      System.err.println("Erreur lors du lancement du serveur SQL ("+ e +")");
    }// try-catch
  }// main

}// classe
```

8.3.4 Écriture du client RMI

Le client du serveur RMI est appelé avec les paramètres suivants :

urlserviceAnnuaire pilote urlBase id mdp separateur

urlserviceAnnuaire	url RMI du service d'annuaire qui a enregistré le serveur SQL
pilote	pilote que doit utiliser le serveur SQL pour gérer la base de données
urlBase	url JDBC de la base de données à gérer
id	identité du client ou null si pas d'identité
mdp	mot de passe du client ou null si pas de mot de passe
separateur	caractère que le serveur SQL doit utiliser pour séparer les champs des lignes résultats d'une requête

Voici un exemple de paramètres possibles :

srvSQL sun.jdbc.odbc.JdbcOdbcDriver jdbc:odbc:articles null null ,

avec ici :

urlserviceAnnuaire	**srvSQL** nom rmi du serveur SQL
pilote	**sun.jdbc.odbc.JdbcOdbcDriver**

	le pilote usuel des bases avec interface odbc
urlBase	**jdbc:odbc:articles** pour utiliser une base articles déclarée dans la liste des bases publiques ODBC de la machine Windows
id	**null** pas d'identité
mdp	**null** pas de mot de passe
separateur	**,** les champs des résultats seront séparés par une virgule

Une fois lancé avec les paramètres précédents, le client suit les étapes suivantes :

- il se connecte au serveur RMI **srvSQL**, donc un serveur RMI sur la même machine que le client
- il demande la connexion à la base de données articles
  connect("sun.jdbc.odbc.JdbcOdbcDriver", "jdbc:odbc:articles", "", "")
- il demande à l'utilisateur de taper une requête SQL au clavier
- il l'envoie au serveur SQL
  executeSQL(requete, ",");
- il affiche à l'écran les résultats renvoyés par le serveur
- il redemande à l'utilisateur de taper une requête SQL au clavier. Il s'arrêtera lorsque la requête est **fin**.

Le texte Java du client suit. Les commentaires devraient suffire à sa compréhension.

```
import java.rmi.*;
import java.io.*;

public class cltSQL {

  // données globales de la classe
  private static String syntaxe =
    "syntaxe : cltSQL urlServiceAnnuaire pilote urlBase id mdp separateur";
  private static BufferedReader in=null;
  private static interSQL serveurSQL=null;

  public static void main(String arg[]){
    // syntaxe : cltSQL urlServiceAnnuaire separateur pilote url id mdp
    // urlServiceAnnuaire : url de l'annuaire des services RMI à contacter
    // pilote : pilote à utiliser pour la base de données à exploiter
    // urlBase : url jdbc de la base à exploiter
    // id : identité de l'utilisateur
    // mdp : son mot de passe
    // separateur : chaine séparant les champs dans les résultats d'une requête

    // vérification du nb d'arguments
    if(arg.length!=6)
      erreur(syntaxe,1);

    // init paramètres de la connexion à la base de données
    String urlService=arg[0];
    String pilote=arg[1];
    String urlBase=arg[2];
    String id, mdp, separateur;
    if(arg[3].equals("null")) id=""; else id=arg[3];
    if(arg[4].equals("null")) mdp=""; else mdp=arg[4];
    if(arg[5].equals("null")) separateur=" "; else separateur=arg[5];

    // installation d'un gestionnaire de sécurité
    System.setSecurityManager(new RMISecurityManager());

    // dialogue client-serveur
    String requete=null;
    String reponse=null;
    String[] lignes=null;
    String codeErreur=null;

    try{
      // ouverture du flux clavier
      in=new BufferedReader(new InputStreamReader(System.in));
      // suivi
      System.out.println("--> Connexion au serveur RMI en cours...");
      // localisation du service
      serveurSQL=(interSQL) Naming.lookup(urlService);
      // suivi
      System.out.println("--> Connexion à la base de données en cours");
      // demande de connexion initiale à la base de données
      reponse=serveurSQL.connect(pilote,urlBase,id,mdp);
      // suivi
```

```
      System.out.println("<-- "+reponse);
      // analyse réponse
      codeErreur=reponse.substring(0,3);
      if(codeErreur.equals("500"))
        erreur("Abandon sur erreur de connexion à la base",3);
      // boucle de lecture des requêtes à envoyer au serveur SQL
      System.out.print("--> Requête : ");
      requete=in.readLine().toLowerCase().trim();
      while(! requete.equals("fin")){
        // envoi de la requête au serveur et réception de la réponse
        lignes=serveurSQL.executeSQL(requete,separateur);
        // suivi
        afficheLignes(lignes);
        // requête suivante
        System.out.print("--> Requête : ");
        requete=in.readLine().toLowerCase().trim();
      }// while
      // suivi
      System.out.println("--> Fermeture de la connexion à la base de données distante");
      // on clôt la connexion
      reponse=serveurSQL.close();
      // suivi
      System.out.println("<-- " + reponse);
      // fin
      System.exit(0);
    // gestion des erreurs
    } catch (Exception e){
      erreur("Abandon sur erreur : " + e,2);
    }// try
  }// main

  // ----------- AfficheLignes
  private static void afficheLignes(String[] lignes){
    for (int i=0;i<lignes.length;i++)
      System.out.println("<-- " + lignes[i]);
  }// afficheLignes

  // ------------ erreur
  private static void erreur(String msg, int exitCode){
    // affichage msg d'erreur
    System.err.println(msg);
    // libération éventuelle des ressources
    try{
      in.close();
      serveurSQL.close();
    } catch(Exception e){}
    // on quitte
    System.exit(exitCode);
  }// erreur

}// classe
```

8.3.5 Étape 3 : création des fichiers .class

- le serveur est compilé

```
E:\data\java\RMI\sql\serveur>j:\jdk12\bin\javac interSQL.java

E:\data\java\RMI\sql\serveur>j:\jdk12\bin\javac srvSQL.java

E:\data\java\RMI\sql\serveur>dir *.class

INTERS~1 CLA           451  12/03/99  17:54 interSQL.class
SRVSQL~1 CLA         3 238  12/03/99  17:54 srvSQL.class
```

- les fichiers Stub et Skel sont créés

```
E:\data\java\RMI\sql\serveur>j:\jdk12\bin\rmic srvSQL

E:\data\java\RMI\sql\serveur>dir *.class


INTERS~1 CLA           451  12/03/99  17:54 interSQL.class
SRVSQL~1 CLA         3 238  12/03/99  17:54 srvSQL.class
SRVSQL~2 CLA         4 491  12/03/99  17:56 srvSQL_Stub.class
SRVSQL~3 CLA         2 414  12/03/99  17:56 srvSQL_Skel.class
```

- on transfére les fichiers interSQL.class, srvSQL_Stub.class, srvSQL_Skel.class dans le répertoire du client

```
E:\data\java\RMI\sql\client>dir

CLTSQL~1 JAV         3 486  11/03/99  11:39 cltSQL.java
INTERS~1 CLA           451  11/03/99  10:55 interSQL.class
SRVSQL~1 CLA         4 491  11/03/99  13:19 srvSQL_Stub.class
SRVSQL~2 CLA         2 414  11/03/99  13:19 srvSQL_Skel.class
```

- on compile le client

```
E:\data\java\RMI\sql\client>j:\jdk12\bin\javac cltSQL.java

E:\data\java\RMI\sql\client>dir *.class

INTERS~1 CLA           451  11/03/99  10:55 interSQL.class
CLTSQL~1 CLA         2 839  12/03/99  18:00 cltSQL.class
SRVSQL~1 CLA         4 491  11/03/99  13:19 srvSQL_Stub.class
SRVSQL~2 CLA         2 414  11/03/99  13:19 srvSQL_Skel.class
```

8.3.6 Étape 4 : Tests avec serveur & client sur même machine windows

- le service d'annuaire est lancé dans un répertoire autre que celui du serveur et du client

```
F:\>start j:\jdk12\bin\rmiregistry
```

- on place le fichier **mypolicy** suivant dans les répertoires du client et du serveur

```
grant {
   // Allow everything for now
   permission java.security.AllPermission;
};
```

- on lance le serveur

```
E:\data\java\RMI\sql\serveur>start j:\jdk12\bin\java -Djava.security.policy=mypolicy
-Djava.rmi.server.codebase=file:/e:/data/java/rmi/sql/serveur/ srvSQL
```

- on lance le client

```
E:\data\java\RMI\sql\client>j:\jdk12\bin\java -Djava.security.policy=mypolicy cltSQL srvSQL
sun.jdbc.odbc.JdbcOdbcDriver jdbc:odbc:articles null null ,

--> Connexion au serveur RMI en cours...
--> Connexion à la base de données en cours
<-- 200 Connexion réussie
--> Requête : select nom, stock_actu from articles order by stock_actu desc
<-- 101 vélo,31
<-- 101 essai3,13
<-- 101 skis nautiques,13
<-- 101 canoé,13
<-- 101 panthère,11
<-- 101 léopard,11
<-- 101 cachalot,10
<-- 101 fusil,10
<-- 101 arc,10
--> Requête : update articles set stock_actu=stock_actu-1 where stock_actu<=11
<-- 100 5
--> Requête : select nom,stock_actu from articles order by stock_actu asc
<-- 101 cachalot,9
<-- 101 fusil,9
<-- 101 arc,9
<-- 101 panthère,10
<-- 101 léopard,10
<-- 101 essai3,13
<-- 101 skis nautiques,13
<-- 101 canoé,13
```

```
<-- 101 vélo,31
--> Requête : fin
--> Fermeture de la connexion à la base de données distante
<-- 200 Base fermée
```

8.3.7 Étape 5 : Tests avec serveur sur machine windows et client sur machine linux

- on arrête éventuellement le serveur et le service d'annuaire
- on transfère les fichiers .class du client sur une machine linux

```
shiva[serge]:/home/admin/serge/java/rmi/sql/client#
$ dir *.class
-rw-r--r--   1 serge    admin        2839 Mar 11 14:37 cltSQL.class
-rw-r--r--   1 serge    admin         451 Mar 11 14:37 interSQL.class
```

- les fichiers du serveur sont mis dans un répertoire accessible au serveur http de la machine windows

```
D:\Inetpub\wwwroot\rmi\sql>dir

INTERS~1 CLA           451  11/03/99  10:55 interSQL.class
SRVSQL~1 CLA         3 238  11/03/99  13:19 srvSQL.class
SRVSQL~2 CLA         4 491  11/03/99  13:19 srvSQL_Stub.class
SRVSQL~3 CLA         2 414  11/03/99  13:19 srvSQL_Skel.class
MYPOLICY                81  08/06/98  15:01 mypolicy
```

- on relance le service d'annuaire

```
F:\>start j:\jdk12\bin\rmiregistry
```

- on relance le serveur avec des paramètres différents que ceux utilisés dans le test précédent

```
D:\Inetpub\wwwroot\rmi\sql>start j:\jdk12\bin\java -Djava.security.policy=mypolicy
-Djava.rmi.server.codebase=http://tahe.istia.univ-angers.fr/rmi/sql/ srvSQL
```

- on lance le client sur la machine linux

```
/usr/local/bin/jdk/bin/java cltSQL rmi://tahe.istia.univ-angers.fr/srvSQL sun.jdbc.odbc.JdbcOdbcDriver
jdbc:odbc:articles null null ,

--> Requête : select nom,stock_actu,stock_mini from articles order by nom
<-- 101 arc,9,8
<-- 101 cachalot,9,6
<-- 101 canoé,13,7
<-- 101 essai3,13,9
<-- 101 fusil,9,8
<-- 101 léopard,10,7
<-- 101 panthère,10,7
<-- 101 skis nautiques,13,8
<-- 101 vélo,31,8
--> Requête : update articles set stock_actu=stock_mini where stock_mini<=7
<-- 100 4
--> Requête : select nom,stock_actu,stock_mini from articles order by nom
<-- 101 arc,9,8
<-- 101 cachalot,6,6
<-- 101 canoé,7,7
<-- 101 essai3,13,9
<-- 101 fusil,9,8
<-- 101 léopard,7,7
<-- 101 panthère,7,7
<-- 101 skis nautiques,13,8
<-- 101 vélo,31,8
--> Requête : fin
--> Fermeture de la connexion à la base de données distante
<-- 200 Base fermée
```

8.3.8 Conclusion

On a là une application intéressante en ce qu'elle permet l'accès à une base de données à partir d'un poste quelconque du réseau. On aurait très bien pu l'écrire de façon traditionnelle avec des sockets et ce qui est d'ailleurs demandé dans un exercice du chapitre sur les bases de données. Si on écrivait cette application de façon traditionnelle :

- on aurait un client et un serveur qui pourraient être écrits avec des langages différents
- le client et le serveur communiqueraient par échange de lignes de texte et auraient un dialogue qui pourrait être du genre :

client : connect machine port pilote urlBase id mdp
où les deux premiers paramètres spécifient où trouver le serveur, les quatre suivants indiquant les paramètres de connexion à la base de données à exploiter

Le serveur pourrait répondre par quelque chose du genre :
200 - Connexion réussie
500 - Echec de la connexion

client : executeSQL requete, separateur

pour demander au serveur d'exécuter une requête SQL sur la base connectée au client. **separateur** étant le caractère séparateur des champs dans les lignes de la réponse.

Le serveur pourrait répondre quelque chose du genre

100 n
pour une requête de mise à jour de la base, n étant le nombre de lignes mises à jour
500 msg d'erreur
si la requête a généré une erreur
501 Pas de résultats
si la requête n'a généré aucun résultat
101 ligne1
101 ligne2
101 ...
si la requête a généré des résultats. Les lignes ainsi renvoyées par le serveur sont les lignes résultats de la requête.

client : close
pour fermer la connexion avec la base distante. Le serveur pourrait renvoyer une chaîne indiquant le résultat de cette fermeture :

200 Base fermée
500 Erreur lors de la fermeture de la base (msg d'erreur)

On voit ici que si on est capable de construire une application traditionnelle avec un protocole du genre précédent, on peut en déduire une structure possible du serveur RMI. Là où dans le protocole on a une phrase du client vers le serveur du genre :

commande param1 param2 ... paramq

on pourra avoir au sein du serveur RMI une méthode

String commande(param1,param2,..., paramq)

et cette méthode accessible au client devra donc faire partie de l'interface publiée du serveur.

Pour en terminer là, notons que notre serveur ne gère actuellement qu'un client : il ne peut en gérer plusieurs, tel qu'il est écrit actuellement. En effet, si un client se connecte à une base B1, un objet *Connection* DB=DB1 est créé par le serveur. Si un second client demande une connexion à une base B2, le serveur le note en faisant *Connection* DB=DB2, cassant ainsi la connexion du premier client à la base B1.

8.4 Exercices

8.4.1 Exercice 1

Étendre le serveur SQL précédent afin qu'il gère plusieurs clients.

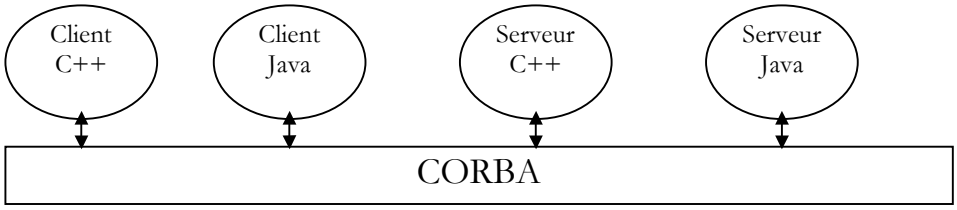
8.4.2 Exercice 2

Écrire l'applet Java de commerce électronique présentée dans les exercices du chapitre JDBC afin qu'elle travaille avec le serveur RMI de l'exercice précédent.

9. Construction d'applications distribuées CORBA

9.1 Introduction

Dans le chapitre précédent, nous avons vu comment créer des applications distribuées en Java avec le package RMI. Nous abordons ici le même problème avec cette fois-ci l'architecture CORBA. CORBA (*Common Object Request Broker Architecture*) est une spécification définie par l'OMG (*Object Management Group*) qui regroupe de nombreuses entreprises du monde informatique. CORBA définit un « bus logiciel » accessible à des applications écrites en différents langages :



Nous allons voir que la construction avec CORBA d'une application distribuée est proche de la méthode employée avec Java RMI : les concepts se ressemblent. CORBA présente l'avantage de l'interopérabilité avec des applications écrites dans d'autres langages.

9.2 Processus de développement d'une application CORBA

9.2.1 Introduction

Pour développer une application client-serveur CORBA, nous suivrons les étapes suivantes :

1. écriture de l'interface du serveur avec IDL *(Interface Definition Language)*
2. génération des classes « squelette » et « stub » du serveur
3. écriture du serveur
4. écriture du client
5. compilation de l'ensemble des classes
6. lancement d'un annuaire des services CORBA
7. lancement du serveur
8. lancement du client

Nous prenons comme premier exemple, celui du serveur d'écho déjà utilisé dans le contexte RMI. Le lecteur pourra ainsi voir les différences entre les deux méthodes.

L'application a été testée avec le jdk1.2.

9.2.2 Écriture de l'interface du serveur

Comme avec Java RMI, vis à vis du client, le serveur est défini par son interface. Si les classes implémentant le serveur ne sont pas nécessaires au client, celles de son interface le sont. Là où Java RMI utilisait une interface Java donnant naissance aux classes « squelette » et « stub » du serveur, l'architecture Java CORBA nécessite la description de l'interface avec un langage autre que Java. Cette interface donnera naissance à plusieurs classes utilisées pour certaines par le client, pour d'autres par le serveur.

La description de l'interface d'écho sera la suivante :

```
module echo{
  interface iSrvEcho{
    string echo(in string msg);
```

```
  };
};
```

La description de l'interface sera stockée dans un fichier **echo.idl**. Elle est écrite dans le langage IDL (*Interface Definition Language*) de l'OMG. Pour être exploitable, elle doit être analysée par un programme qui va créer des fichiers source dans le langage utilisé pour développer l'application CORBA. Ici, nous utiliserons le programme **idltojava.exe** qui à partir de l'interface précédente va créer les fichiers source *.java* nécessaires à l'application. Le programme *idltojava.exe* n'est pas livré avec le JDK. On peut se le procurer sur le site de Sun **http://java.sun.com**.

Analysons les quelques lignes de l'interface *idl* précédente :

module echo
: est équivalent à **package echo** de Java. La compilation de l'interface va donner naissance au package java *echo* c.a.d. un répertoire contenant des classes Java.

interface iSrvEcho
: est équivalent à **interface iSrvEcho** de Java. Va donner naissance à une interface Java.

string echo(in string msg)
: est équivalent à l'instruction Java **String echo(String msg)**. Les types du langage IDL ne correspondent pas exactement à ceux du langage Java. On trouvera les correspondances un peu plus loin dans ce chapitre. Dans le langage IDL, les paramètres d'une fonction peuvent être des paramètres d'entrée (**in**), de sortie (**out**), d'entrée-sortie (**inout**). Ici, la méthode *echo* reçoit un paramètre d'entrée *msg* qui est une chaîne de caractères et renvoie une chaîne de caractères comme résultat.

L'interface précédente est celle de notre serveur d'écho. On rappelle qu'une interface distante décrit les méthodes de l'objet serveur accessibles aux clients. Ici, seule la méthode *echo* sera disponible pour les clients.

9.2.3 Compilation de l'interface IDL du serveur

Une fois l'interface du serveur définie, on produit les fichiers Java correspondants.

```
E:\data\java\corba\ECHO>dir *.idl

ECHO     IDL             78  15/03/99  13:56 ECHO.IDL

E:\data\java\corba\ECHO>d:\javaidl\idltojava.exe -fno-cpp echo.idl
```

L'option *-fno-cpp* sert à indiquer qu'il n'y a pas à utiliser de pré-processeur (utilisé avec le C/C++ le plus souvent). La compilation du fichier *echo.idl* produit un sous-répertoire *echo* avec dedans les fichiers suivants :

```
E:\data\java\corba\ECHO>dir echo

_ISRVE~1 JAV         1 095  17/03/99  17:19 _iSrvEchoStub.java
ISRVEC~1 JAV           311  17/03/99  17:19 iSrvEcho.java
ISRVEC~2 JAV           825  17/03/99  17:19 iSrvEchoHolder.java
ISRVEC~3 JAV         1 827  17/03/99  17:19 iSrvEchoHelper.java
_ISRVE~2 JAV         1 803  17/03/99  17:19 _iSrvEchoImplBase.java
```

Le fichier *iSrvEcho.java* est le fichier Java décrivant l'interface du serveur :

```
/*
 * File: ./ECHO/ISRVECHO.JAVA
 * From: ECHO.IDL
 * Date: Mon Mar 15 13:56:08 1999
 *   By: D:\JAVAIDL\IDLTOJ~1.EXE Java IDL 1.2 Aug 18 1998 16:25:34
 */

package echo;
public interface iSrvEcho
    extends org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {
    String echo(String msg)
;
}
```

On voit que c'est quasiment la traduction mot à mot de l'interface IDL. Si on a la curiosité de regarder le contenu des autres fichiers *.java*, on trouvera des choses plus complexes. Voici ce que dit la documentation sur le rôle de ces différents fichiers :

iSrvEcho.java
: l'interface du serveur

_iSrvEchoImplbase.java
: implémente l'interface *iSrvEcho* précédente. C'est une classe abstraite, « squelette » du serveur fournissant au serveur les fonctionnalités CORBA nécessaires à l'application distribuée.

_iSrvEchoStub.java
: C'est l'image (« stub ») du serveur dont usera le client. Elle fournit au client les fonctionnalités CORBA pour atteindre le serveur.

iSrvEchoHelper.java
: Fournit les méthodes nécessaires à la gestion des références d'objets CORBA

iSrvEchoHolder.java
: Fournit les méthodes nécessaires à la gestion des paramètres d'entrée-sortie des méthodes de l'interface.

9.2.4 Compilation des classes générées à partir de l'interface IDL

Une bonne idée est de compiler les classes précédentes. On verra dans un autre exemple qu'on peut ici découvrir des erreurs dûes à un fonctionnement incorrect du générateur *idltojava*. Ici cela se passe bien, et après compilation, on a dans le répertoire du package *echo* les fichiers suivants :

```
E:\data\java\corba\ECHO\echo>dir

_ISRVE~1 JAV         1 095  17/03/99  17:19 _iSrvEchoStub.java
ISRVEC~1 JAV           311  17/03/99  17:19 iSrvEcho.java
ISRVEC~2 JAV           825  17/03/99  17:19 iSrvEchoHolder.java
ISRVEC~3 JAV         1 827  17/03/99  17:19 iSrvEchoHelper.java
_ISRVE~2 JAV         1 803  17/03/99  17:19 _iSrvEchoImplBase.java
_ISRVE~1 CLA         2 275  18/03/99  11:25 _iSrvEchoImplBase.class
_ISRVE~2 CLA         1 383  18/03/99  11:25 _iSrvEchoStub.class
ISRVEC~1 CLA           251  18/03/99  11:25 iSrvEcho.class
ISRVEC~2 CLA         2 078  18/03/99  11:25 iSrvEchoHelper.class
ISRVEC~3 CLA           858  18/03/99  11:25 iSrvEchoHolder.class
```

9.2.5 Écriture du serveur

9.2.5.1 Implémentation de l'interface iSrvEcho

Nous avons défini plus haut l'interface *iSrvEcho*. Nous écrivons maintenant la classe implémentant cette interface. Elle sera dérivée de la classe *_iSrvEchoImplbase.java* qui comme indiqué ci-dessus implémente déjà l'interface *iSrvEcho*.

```
// packages importés
import echo.*;

// classe implémentant l'écho distant
public class srvEcho extends _iSrvEchoImplBase{
  // méthode réalisant l'écho
  public String echo(String msg){
    return  "[" + msg + "]";
  }// fin écho
}// fin classe
```

Le code se comprend de lui-même. Cette classe est enregistrée dans le fichier *srvEcho.java* dans le répertoire parent de celui du package de l'interface *iSrvEcho*.

On peut compiler pour vérifier :

```
E:\data\java\corba\ECHO>j:\jdk12\bin\javac srvEcho.java

E:\data\java\corba\ECHO>dir

ECHO     IDL              78  15/03/99  13:56 ECHO.IDL
SRVECH~1 CLA             488  18/03/99  11:30 srvEcho.class
SRVECH~1 JAV             252  15/03/99  14:02 srvEcho.java
ECHO           <REP>         17/03/99  17:19 echo
```

9.2.5.2 Écriture de la classe de création du serveur

Comme pour une application client-serveur RMI, un serveur CORBA doit être enregistré dans un annuaire pour être accessible par des clients. C'est cette procédure d'enregistrement qui, au niveau développement, diffère selon qu'on a une application CORBA ou RMI. Voici celle du serveur CORBA d'écho enregistrée dans le fichier *serveurEcho.java* :

```
// packages importés
import echo.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

//---------- classe serveurEcho
public class serveurEcho{
  // ------- main : lance le serveur d'écho
  // syntaxe pg machineAnnuaire portAnnuaire nomService
  // machine : machine supportant l'annuaire CORBA
  // port : port de l'annuaire CORBA
  // nomService : nom du service à enregistrer

  public static void main(String arg[]){
  // les arguments sont-ils là ?
  if(arg.length!=3){
    System.err.println("Syntaxe : pg machineAnnuaire portAnnuaire nomService");
    System.exit(1);
  }
  // on récupère les arguments
  String machine=arg[0];
  String port=arg[1];
  String nomService=arg[2];

    try{

  // il nout faut un objet CORBA pour travailler
  String[] initORB={"-ORBInitialHost",machine,"-ORBInitialPort",port};
      ORB orb=ORB.init(initORB,null);
  // on met le service dans l'annuaire des services
  // il s'appellera srvEcho
      org.omg.CORBA.Object objRef=
        orb.resolve_initial_references("NameService");
      NamingContext ncRef=NamingContextHelper.narrow(objRef);
      NameComponent nc= new NameComponent(nomService,"");
      NameComponent path[]={nc};
  // on crée le serveur et on l'associe au service srvEcho
  srvEcho serveurEcho=new srvEcho();
      ncRef.rebind(path,serveurEcho);
  orb.connect(serveurEcho);
  // suivi
  System.out.println("Serveur d'écho prêt");
  // attente des demandes des clients
      java.lang.Object sync=new java.lang.Object();
      synchronized(sync){
        sync.wait();
      }
    } catch(Exception e){
  // il y a eu une erreur
      System.err.println("Erreur " + e);
      e.printStackTrace(System.err);
    }
  }// main
}// serveurEcho
```

Nous expliquons ci-après les grandes lignes du lancement du serveur sans entrer dans les détails qui sont complexes au premier abord. Il faut retenir de l'exemple précédent ses grandes lignes qui vont revenir dans tout serveur CORBA.

9.2.5.2.1 Les paramètres du serveur

Un serveur CORBA doit s'enregistrer auprès d'un service d'annuaire opérant sur une machine et un port donnés. Notre application recevra ces deux données en paramètres. Le service ainsi enregistré doit porter un nom, ce sera le troisième paramètre.

9.2.5.2.2 Créer l'objet d'accès au service d'annuaire CORBA

Pour atteindre le service d'annuaire et enregistrer notre serveur d'écho, nous avons besoin d'un objet appelé ORB (Object Request Broker) obtenu avec la méthode de classe suivante :

ORB ORB.init(String [] args, Properties prop)

args
: tableau de paires de chaînes de caractères, chaque paire étant de la forme (paramètre,valeur)

prop
: propriétés de l'application

L'exemple utilise la séquence suivante pour obtenir l'objet ORB :

```
String[] initORB={"-ORBInitialHost",machine,"-ORBInitialPort",port};
  ORB orb=ORB.init(initORB,null);
```

Les couples (paramètre, valeur) utilisés sont les suivants :

`("-ORBInitialHost",machine)`
: ce couple précise la machine ou opère l'annuaire des services CORBA, ici la machine passée en paramètre au serveur.

`("-ORBInitialPort",port )`
: ce couple précise le port ou opère l'annuaire des services CORBA, ici le port passé en paramètre au serveur.

Le second paramètre de la méthode *init* est laissé à *null*. Si on avait laissé le premier paramètre également à *null*, le couple (machine,port) utilisé aurait été par défaut **(localhost,900)**.

9.2.5.2.3 Enregistrer le serveur dans l'annuaire des services CORBA

L'enregistrement du serveur dans l'annuaire se fait avec les opérations suivantes :

```
// on met le service dans l'annuaire des services
// il s'appellera srvEcho
    org.omg.CORBA.Object objRef=
      orb.resolve_initial_references("NameService");
    NamingContext ncRef=NamingContextHelper.narrow(objRef);
    NameComponent nc= new NameComponent(nomService,"");
    NameComponent path[]={nc};
// on crée le serveur et on l'associe au service srvEcho
srvEcho serveurEcho=new srvEcho();
     ncRef.rebind(path,serveurEcho);
    orb.connect(serveurEcho);
```

La première partie du code consiste à préparer le nom du service. Ce nom est symbolisé dans le code par la variable **path**. Le nom d'un service comprend plusieurs composantes :

- une composante initiale *objRef*, objet générique qui doit être ramené à un type *NamingContext*, ici *ncRef*.
- le nom du service, ici *nomService* qui a été passé en paramètre au serveur

Ces composantes du nom (*NameComponent*) sont rassemblées dans un tableau, ici *path*. C'est ce tableau qui « nomme » de façon précise le service créé. Une fois le nom créé, il reste

- à l'associer à un exemplaire du serveur (la classe srvEcho construite précédemment)

```
srvEcho serveurEcho=new srvEcho();
```

- à l'enregistrer dans l'annuaire

```
ncRef.rebind(path,serveurEcho);
orb.connect(serveurEcho);
```

9.2.5.3 Compilation de la classe de lancement serveur

On compile la classe précédente :

```
E:\data\java\corba\ECHO>j:\jdk12\bin\javac serveurEcho.java

E:\data\java\corba\ECHO>dir

ECHO     IDL            78  15/03/99  13:56 ECHO.IDL
SERVEU~1 CLA         1 793  18/03/99  13:18 serveurEcho.class
SERVEU~1 JAV         1 806  16/03/99  15:38 serveurEcho.java
SRVECH~1 CLA           488  18/03/99  11:30 srvEcho.class
SRVECH~1 JAV           252  15/03/99  14:02 srvEcho.java
ECHO          <REP>         17/03/99  17:19 echo
```

9.2.6 Écriture du client

9.2.6.1 Le code

Nous écrivons un client afin de tester notre service d'écho. On passera au client les trois mêmes paramètres qu'au serveur :

machine	machine où se trouve l'annuaire des services CORBA
port	port sur lequel opère cet annuaire
nomService	nom du service d'écho

Le client se connecte au service d'écho puis demande à l'utilisateur de taper des messages au clavier. Ceux-ci sont envoyés au serveur d'écho qui les renvoie. Un suivi de ce dialogue est fait à l'écran.

Le client CORBA du service d'écho ressemble beaucoup au client RMI déjà écrit. Là encore, le client doit se connecter à un service d'annuaire pour obtenir une référence de l'objet-serveur auquel il veut se connecter. La différence entre les deux clients réside là et uniquement là. Voici le code du client CORBA d'écho :

```
// packages importés
import java.io.*;
import echo.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

// ---------- classe cltEcho
public class cltEcho {

  public static void main(String arg[]){
    // syntaxe : cltEcho machineAnnuaire portAnnuaire nomservice
    // machine : machine où opère l'annuaire des services CORBA
    // port : port où opère l'annuaire des services
    // nomService : nom du service d'écho

    // vérification des arguments
    if(arg.length!=3){
      System.err.println("Syntaxe : pg machineAnnuaire portAnnuaire nomservice");
      System.exit(1);
    }

    // on récupère les paramètres
    String machine=arg[0];
    String port=arg[1];
    String nomService=arg[2];

    // on fait la liaison avec le serveur d'écho
    iSrvEcho serveurEcho=getServeurEcho(machine,port,nomService);

    // dialogue client-serveur
    BufferedReader in=null;
    String msg=null;
    String reponse=null;
    iSrvEcho serveur=null;

    try{
      // ouverture du flux clavier
      in=new BufferedReader(new InputStreamReader(System.in));
      // boucle de lecture des msg à envoyer au serveur d'écho
      System.out.print("Message : ");
      msg=in.readLine().toLowerCase().trim();
      while(! msg.equals("fin")){
        // envoi du msg au serveur et réception de la réponse
        reponse=serveurEcho.echo(msg);
        // suivi
        System.out.println("Réponse serveur : " + reponse);
        // msg suivant
        System.out.print("Message : ");
        msg=in.readLine().toLowerCase().trim();
      }// while
      // c'est fini
      System.exit(0);
    // gestion des erreurs
    } catch (Exception e){
      System.err.println("Erreur : " + e);
      System.exit(2);
    }// try
  }// main

  // ---------------------- getServeurEcho
  private static iSrvEcho getServeurEcho(String machine, String port,
```

```
      String nomService){

    // demande une référence du serveur d'écho
    // suivi
    System.out.println("--> Connexion au serveur CORBA en cours...");
    // la référence du serveur d'écho
    iSrvEcho serveurEcho=null;
    try{
      // on demande un objet CORBA pour travailler
      String[] initORB={"-ORBInitialHost",machine,"-ORBInitialPort",port};
        ORB orb=ORB.init(initORB,null);
      // on utilise le service d'annuaire pour localiser le serveur d'écho
        org.omg.CORBA.Object objRef=
            orb.resolve_initial_references("NameService");
        NamingContext ncRef=NamingContextHelper.narrow(objRef);
      // le service recherché s'appelle srvEcho - on le demande
        NameComponent nc= new NameComponent(nomService,"");
        NameComponent path[]={nc};
        serveurEcho=iSrvEchoHelper.narrow(ncRef.resolve(path));
    } catch (Exception e){
      System.err.println("Erreur lors de la localisation du serveur d'écho ("
        + e + ")");
      System.exit(10);
    }// try-catch
    // on rend la référence au serveur
    return serveurEcho;
  }// getServeurEcho

}// classe
```

9.2.6.2 La connexion du client au serveur

Le client CORBA ci-dessus se connecte au serveur par l'instruction :

```
    // on fait la liaison avec le serveur d'écho
    iSrvEcho serveurEcho=getServeurEcho(machine,port,nomService);
```

A l'issue de cette opération, le client détient une référence du serveur d'écho. Ensuite un client CORBA ne diffère pas d'un client RMI. La méthode privée qui assure la connexion au serveur est la suivante :

```
  // ---------------------- getServeurEcho
  private static iSrvEcho getServeurEcho(String machine, String port, String nomService){
    // demande une référence du serveur d'écho
    // suivi
    System.out.println("--> Connexion au serveur CORBA en cours...");
    // la référence du serveur d'écho
    iSrvEcho serveurEcho=null;
    try{
      // on demande un objet CORBA pour travailler
      String[] initORB={"-ORBInitialHost",machine,"-ORBInitialPort",port};
       ORB orb=ORB.init(initORB,null);
      // on construit le nom du serveur d'écho
       org.omg.CORBA.Object objRef=
         orb.resolve_initial_references("NameService");
       NamingContext ncRef=NamingContextHelper.narrow(objRef);
       NameComponent nc= new NameComponent(nomService,"");
       NameComponent path[]={nc};
    // on demande le service
       serveurEcho=iSrvEchoHelper.narrow(ncRef.resolve(path));
    } catch (Exception e){
      System.err.println("Erreur lors de la localisation du serveur d'écho ("
        + e + ")");
      System.exit(10);
    }// try-catch
    // on rend la référence au serveur
    return serveurEcho;
  }// getServeurEcho
```

On retrouve les mêmes séquences de code que dans le serveur :

- on crée un objet ORB qui nous permettra de contacter l'annuaire des services CORBA

```
String[] initORB={"-ORBInitialHost",machine,"-ORBInitialPort",port};
      ORB orb=ORB.init(initORB,null);
```

- on définit les différentes composantes du nom du service d'écho

```
    org.omg.CORBA.Object objRef=orb.resolve_initial_references("NameService");
      NamingContext ncRef=NamingContextHelper.narrow(objRef);
```

```
      NameComponent nc= new NameComponent(nomService,"");
      NameComponent path[]={nc};
```

- on demande au service d'annuaire une référence du service d'écho (c'est là qu'on diffère du serveur)

```
serveurEcho=iSrvEchoHelper.narrow(ncRef.resolve(path));
```

9.2.6.3 Compilation

```
E:\data\java\corba\ECHO>j:\jdk12\bin\javac cltEcho.java

E:\data\java\corba\ECHO>dir

CLTECH~1 CLA         2 599  18/03/99  13:51 cltEcho.class
CLTECH~1 JAV         2 907  16/03/99  16:15 cltEcho.java
ECHO     IDL            78  15/03/99  13:56 ECHO.IDL
SERVEU~1 CLA         1 793  18/03/99  13:18 serveurEcho.class
SERVEU~1 JAV         1 806  16/03/99  15:38 serveurEcho.java
SRVECH~1 CLA           488  18/03/99  11:30 srvEcho.class
SRVECH~1 JAV           252  15/03/99  14:02 srvEcho.java
ECHO           <REP>        17/03/99  17:19 echo
```

9.2.7 Tests

9.2.7.1 Lancement du service d'annuaire

Sur une machine Windows, nous lançons le service d'annuaire de la façon suivante :

```
E:\data\java\corba\ECHO>start j:\jdk12\bin\tnameserv -ORBInitialPort 1000
```

ce qui a pour effet de lancer le service d'annuaire sur le port 1000 de la machine.

Le service d'annuaire **tnameserv** produit un affichage d'écran qui ressemble à ce qui suit :

```
Initial Naming Context:
IOR:000000000000002849444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436f
6e746578743a312e300000000001000000000000003000010000000000a69737469612d30303900
044700000018afabcafe000000027620dd9a0000000800000000000000000
TransientNameServer: setting port for initial object references to: 1000
```

C'est peu lisible, mais on retiendra la dernière ligne : le service est actif sur le port 1000.

9.2.7.2 Lancement du serveur d'écho

Le service d'écho est lancé avec trois paramètres :

```
E:\data\java\corba\ECHO>start j:\jdk12\bin\java serveurEcho localhost 1000 srvEcho
```

Le serveur affiche :

```
   Serveur d'écho prêt
```

9.2.7.3 Lancement du client sur le même poste que celui du serveur

```
E:\data\java\corba\ECHO>j:\jdk12\bin\java cltEcho localhost 1000 srvEcho
--> Connexion au serveur CORBA en cours...
Message : msg1
Réponse serveur : [msg1]
Message : msg2
Réponse serveur : [msg2]
Message : fin
```

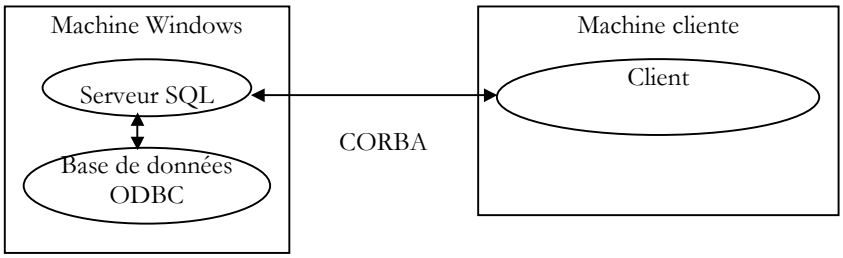
9.2.7.4 Lancement du client sur poste Windows autre que celui du serveur

```
E:\data\java\corba\ECHO>j:\jdk12\bin\java cltEcho tahe.istia.univ-angers.fr 1000 srvEcho
--> Connexion au serveur CORBA en cours...
Message : abcd
Réponse serveur : [abcd]
Message : efgh
Réponse serveur : [efgh]
Message : fin
```

9.3 Exemple 2 : un serveur SQL

9.3.1 Introduction

Nous reprenons ici l'écriture du serveur SQL déjà étudié dans le contexte de Java RMI, toujours pour mettre en évidence les points communs aux deux méthodes ainsi que leurs différences. Nous rappelons le rôle de ce serveur SQL : il se trouve sur une machine Windows, et permet aux clients distants d'exploiter les bases ODBC publiques de ce poste Windows.



Le client CORBA pourrait faire 3 opérations :

- se connecter à la base de son choix
- émettre des requêtes SQL
- fermer la connexion

Le serveur exécute les requêtes SQL du client et lui envoie les résultats. C'est son travail essentiel et c'est pourquoi nous l'appellons un serveur SQL. Nous appliquons les différentes étapes vues précédemment avec le serveur d'écho.

9.3.2 Écriture de l'interface IDL du serveur

Rappelons pour mémoire l'interface RMI que nous avions utilisée pour le serveur :

```
import java.rmi.*;

// l'interface distante
public interface interSQL extends Remote{
  public String connect(String pilote, String url, String id, String mdp)
    throws java.rmi.RemoteException;
  public String[] executeSQL(String requete, String separateur)
    throws java.rmi.RemoteException;
  public String close()
    throws java.rmi.RemoteException;
}
```

Le rôle des différentes méthodes était le suivant :

connect le client se connecte à une base de données distante dont il donne le pilote, l'url JDBC ainsi que son identité id et son mot de passe mdp pour accéder à cette base. Le serveur lui rend une chaîne de caractères indiquant le résultat de la connexion :

200 - Connexion réussie
500 - Echec de la connexion

executeSQL le client demande l'exécution d'une requête SQL sur la base à laquelle il est connecté. Il indique le caractère qui doit séparer les champs dans les résultats qui lui sont renvoyés. Le serveur renvoie un tableau de chaînes :

100 n
pour une requête de mise à jour de la base, n étant le nombre de lignes mises à jour
500 msg d'erreur
si la requête a généré une erreur
501 Pas de résultats
si la requête n'a généré aucun résultat
101 ligne1
101 ligne2
101 ...
si la requête a généré des résultats. Les lignes ainsi renvoyées par le serveur sont les lignes résultats de la requête.

close le client ferme sa connexion avec la base distante. Le serveur renvoie une chaîne indiquant le résultat de cette fermeture :

200 Base fermée
500 Erreur lors de la fermeture de la base (msg d'erreur)

L'interface IDL du serveur sera la suivante :

```
module srvSQL{

  typedef sequence<string> resultats;

  interface interSQL{
    string connect(in string pilote, in string urlBase, in string id, in string mdp);
    resultats executeSQL(in string requete, in string separateur);
    string close();
  };// interface
};// module
```

La seule nouveauté par rapport à ce qu'on a vu avec l'interface IDL du serveur d'écho est l'utilisation du mot clé **sequence**. Ce mot clé permet de définir un tableau à une dimension. La définition se fait en deux étapes :

- définition d'un type pour désigner le tableau, ici **resultats** :

```
typedef sequence<string> resultats;
```

> Le mot clé *typedef* est bien connu des programmeurs C/C++ : il permet de définir un nouveau type. Ici, le type *resultats* est défini comme équivalent au type **sequence<string>**, c.a.d. un tableau dynamique (non dimensionné) de chaînes de caractères.

- utilisation du nouveau type là où on en a besoin

```
resultats executeSQL(in string requete, in string separateur);
```

La méthode *executeSQL* rend donc un tableau de chaînes de caractères.

9.3.3 Compilation de l'interface IDL du serveur

L'interface IDL précédente est mise dans le fichier *srvSQL.idl*. On compile ce fichier :

```
E:\data\java\corba\sql>d:\javaidl\idltojava -fno-cpp srvSQL.idl

E:\data\java\corba\sql>dir

SRVSQL   IDL           275  19/03/99   9:59 srvSQL.idl
SRVSQL         <REP>        19/03/99   9:41 srvSQL
```

On constate que la compilation a donné naissance à un répertoire portant le nom du module de l'interface IDL(*srvSQL*). Regardons le contenu de ce répertoire :

```
E:\data\java\corba\sql>dir srvSQl

RESULT~1 JAV           833  19/03/99  10:00 resultatsHolder.java
RESULT~2 JAV         1 883  19/03/99  10:00 resultatsHelper.java
_INTER~1 JAV         2 474  19/03/99  10:00 _interSQLStub.java
INTERS~1 JAV           448  19/03/99  10:00 interSQL.java
INTERS~2 JAV           841  19/03/99  10:00 interSQLHolder.java
INTERS~3 JAV         1 855  19/03/99  10:00 interSQLHelper.java
_INTER~2 JAV         4 535  19/03/99  10:00 _interSQLImplBase.java
```

On rappelle que les fichiers *Helper* et *Holder* sont des classes liées aux paramètres d'entré-sortie et résultats des méthodes de l'interface distante. Le répertoire srvSQL contient tous les fichiers .java liés à l'interface *interSQL* définie dans le fichier .idl. In contient également des fichiers liés au type *resultats* créé dans l'interface IDL.

Le fichier *interSQL.java* est le fichier Java de l'interface de notre serveur. Il est important de vérifier que ce qui a été produit automatiquement correspond à notre attente. Le fichier *interSQL.java* généré est ici le suivant :

```
/*
 * File: ./SRVSQL/INTERSQL.JAVA
 * From: SRVSQL.IDL
 * Date: Fri Mar 19 09:59:48 1999
 *   By: D:\JAVAIDL\IDLTOJ~1.EXE Java IDL 1.2 Aug 18 1998 16:25:34
 */

package srvSQL;
public interface interSQL
    extends org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {
    String connect(String pilote, String urlBase, String id, String mdp)
;
    String[] executeSQL(String requete, String separateur)
;
    String close()
;
}
```

On constate qu'on a la même interface que celle utilisée pour le client-serveur RMI. On peut donc continuer. Compilons tous ces fichiers .java :

```
E:\data\java\corba\sql\srvSQL>j:\jdk12\bin\javac *.java

Note: _interSQLImplBase.java uses or overrides a deprecated API.  Recompile with
 "-deprecation" for details.
1 warning

E:\data\java\corba\sql\srvSQL>dir *.class

_INTER~1 CLA         3 094  19/03/99  10:01 _interSQLImplBase.class
_INTER~2 CLA         1 953  19/03/99  10:01 _interSQLStub.class
INTERS~1 CLA           430  19/03/99  10:01 interSQL.class
INTERS~2 CLA         2 096  19/03/99  10:01 interSQLHelper.class
INTERS~3 CLA           870  19/03/99  10:01 interSQLHolder.class
RESULT~1 CLA         2 047  19/03/99  10:01 resultatsHelper.class
RESULT~2 CLA           881  19/03/99  10:01 resultatsHolder.class
```

9.3.4 Écriture du serveur SQL

Nous écrivons maintenant le code du serveur SQL. Rappelons que cette classe doit dériver de la classe abstraite *_nomInterfaceImplBase* produite par la compilation du fichier IDL. Hormis cette particularité et si on excepte les séquences de code liées à l'enregistrement du service dans un annuaire, le code du serveur CORBA est identique à celui du serveur RMI :

```
// packages importés
import java.sql.*;
import java.util.*;
import srvSQL.*;

// classe SQLServant
public class SQLServant extends _interSQLImplBase{

  // données globales de la classe
```

```
private Connection DB;

// --------------- connect
public String connect(String pilote, String url, String id,
  String mdp){

  // connexion à la base url grâce au driver pilote
  // identification avec identité id et mot de passe mdp

  String resultat=null;    // résultat de la méthode
  try{
    // chargement du pilote
    Class.forName(pilote);
    // demande de connexion
    DB=DriverManager.getConnection(url,id,mdp);
    // ok
    resultat="200 Connexion réussie";
  } catch (Exception e){
    // erreur
    resultat="500 Echec de la connexion (" + e + ")";
  }
  // fin
  return resultat;
}

// ------------- executeSQL
public String[] executeSQL(String requete, String separateur){

  // exécute une requête SQL sur la base DB
  // et met les résultats dans un tableau de chaînes

  // données nécessaires à l'exécution de la requête
  Statement S=null;
  ResultSet RS=null;
  String[] lignes=null;
  Vector resultats=new Vector();
  String ligne=null;

  try{
    // création du conteneur de la requête
    S=DB.createStatement();
    // exécution requête
    if (! S.execute(requete)){
      // requête de mise à jour
      // on renvoie le nombre de lignes mises à jour
      lignes=new String[1];
      lignes[0]="100 "+S.getUpdateCount();
      return lignes;
    }
    // c'était une requête d'interrogation
    // on récupère les résultats
    RS=S.getResultSet();
    // nombre de champs du Resultset
    int nbChamps=RS.getMetaData().getColumnCount();
    // on les exploite
    while(RS.next()){
      // création de la ligne des résultats
      ligne="101 ";
      for (int i=1;i<nbChamps;i++)
        ligne+=RS.getString(i)+separateur;
      ligne+=RS.getString(nbChamps);
      // ajout au vecteur des résultats
      resultats.addElement(ligne);
    }// while
    // fin de l'exploitation des résultats
    // on libère les ressources
    RS.close();
    S.close();
    // on rend les résultats
    int nbLignes=resultats.size();
    if (nbLignes==0){
      lignes=new String[1];
      lignes[0]="501 Pas de résultats";
    } else {
      lignes=new String[resultats.size()];
      for(int i=0;i<lignes.length;i++)
        lignes[i]=(String) resultats.elementAt(i);
    }//if
    return lignes;
  } catch (Exception e){
    // erreur
    lignes=new String[1];
    lignes[0]="500 " + e;
    return lignes;
  }// try-catch
}// executeSQL
```

```
  // --------------- close
  public String close(){
    // ferme la connexion à la base de données
    String resultat=null;
    try{
      DB.close();
      resultat="200 Base fermée";
    } catch (Exception e){
      resultat="500 Erreur à la fermeture de la base ("+e+")";
    }
    // renvoi du résultat
    return resultat;
  }
}// classe SQLServant
```

Cette classe est placée dans le fichier *SQLServant.java* que nous compilons :

```
E:\data\java\corba\sql>dir

SRVSQL   IDL           275  19/03/99   9:59 srvSQL.idl
SRVSQL         <REP>        19/03/99   9:41 srvSQL
SQLSER~1 JAV         2 941  15/03/99   9:09 SQLServant.java

E:\data\java\corba\sql>j:\jdk12\bin\javac SQLServant.java

E:\data\java\corba\sql>dir *.class

SQLSER~1 CLA         2 568  19/03/99  10:19 SQLServant.class
```

9.3.5 Écriture du programme de lancement du serveur SQL

La classe précédente représente le serveur SQL une fois lancé. Auparavant, il doit être enregistré dans un annuaire des services CORBA. Comme pour le service d'écho, nous le ferons avec une classe spéciale à qui nous passerons, au moment de l'exécution, trois paramètres :

machine	machine où se trouve l'annuaire des services CORBA
port	port sur lequel opère cet annuaire
nomService	nom du service SQL

Le code de cette classe est quasi identique à celui de la classe qui faisait la même chose pour le service d'écho. Nous avons mis en gros caractères la ligne qui diffère entre les deux classes : elle ne crée pas le même objet-serveur. On voit donc qu'on a toujours la même mécanique de lancement du serveur. Si on isole cette mécanique dans une classe, comme il a été fait ici, elle devient quasi transparente au développeur.

```
// packages importés
import srvSQL.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

//----------- classe serveurSQL
public class serveurSQL{
  // ------- main : lance le serveur SQL
  public static void main(String arg[]){
    // serveurSQL machine port service

    //a-t-on le bon nombre d'arguments
    if(arg.length!=3){
      System.err.println("Syntaxe : pg machineAnnuaireCorba portAnnuaireCorba nomService");
      System.exit(1);
    }
    // on récupère les arguments
    String machine=arg[0];
    String port=arg[1];
    String nomService=arg[2];
    String[] initORB={"-ORBInitialHost",machine,"-ORBInitialPort",port};
     try{
      // il nout faut un objet CORBA pour travailler
       ORB orb=ORB.init(initORB,null);
      // on met le service dans l'annuaire des services
       org.omg.CORBA.Object objRef=
         orb.resolve_initial_references("NameService");
       NamingContext ncRef=NamingContextHelper.narrow(objRef);
       NameComponent nc= new NameComponent(nomService,"");
       NameComponent path[]={nc};
    // on crée le serveur et on l'associe au service srvSQL
```

```
    SQLServant serveurSQL=new SQLServant();
      ncRef.rebind(path,serveurSQL);
    orb.connect(serveurSQL);
    // suivi
    System.out.println("Serveur SQL prêt");
    // attente des demandes des clients
      java.lang.Object sync=new java.lang.Object();
      synchronized(sync){
        sync.wait();
      }
    } catch(Exception e){
      // il y a eu une erreur
      System.err.println("Erreur " + e);
      e.printStackTrace(System.err);
    }
  }// main
}// srvSQL
```

Nous compilons cette nouvelle classe :

```
E:\data\java\corba\sql>j:\jdk12\bin\javac serveurSQL.java

E:\data\java\corba\sql>dir *.class

SQLSER~1 CLA          2 568  19/03/99  10:19 SQLServant.class
SERVEU~1 CLA          1 800  19/03/99  10:33 serveurSQL.class
```

9.3.6 Écriture du client

Le client du serveur CORBA est appelé avec les paramètres suivants :

machine port nomServiceAnnuaire pilote urlBase id mdp separateur

machine	machine où se trouve l'annuaire des services CORBA
port	port sur lequel opère cet annuaire
nomService	nom du service SQL
pilote	pilote que doit utiliser le serveur SQL pour gérer la base de données désirée
urlBase	url JDBC de la base de données à gérer
id	identité du client ou null si pas d'identité
mdp	mot de passe du client ou null si pas de mot de passe
separateur	caractère que le serveur SQL doit utiliser pour séparer les champs des lignes résultats d'une requête

Voici un exemple de paramètres possibles :

localhost 1000 srvSQL sun.jdbc.odbc.JdbcOdbcDriver jdbc:odbc:articles null null ,

avec ici :

machine	machine sur laquelle se trouve l'annuaire des services CORBA
port	port sur lequel opère cet annuaire
srvSQL	**srvSQL** nom CORBA du serveur SQL
pilote	**sun.jdbc.odbc.JdbcOdbcDriver** le pilote usuel des bases avec interface odbc
urlBase	**jdbc:odbc:articles** pour utiliser une base articles déclarée dans la liste des bases publiques ODBC de la machine Windows
id	**null** pas d'identité
mdp	**null** pas de mot de passe
separateur	**,** les champs des résultats seront séparés par une virgule

Une fois lancé avec les paramètres précédents, le client suit les étapes suivantes :

- il se connecte à la machine **machine** sur le port **port** pour demander le service CORBA **srvSQL**

- il demande la connexion à la base de données articles
  connect("sun.jdbc.odbc.JdbcOdbcDriver", "jdbc:odbc:articles", "", "")
- il demande à l'utilisateur de taper une requête SQL au clavier
- il l'envoie au serveur SQL
  executeSQL(requete, ",");
- il affiche à l'écran les résultats renvoyés par le serveur
- il redemande à l'utilisateur de taper une requête SQL au clavier. Il s'arrêtera lorsque la requête est **fin**.

Le texte Java du client suit. Les commentaires devraient suffire à sa compréhension. On constatera que :

- le code est identique à celui du client RMI déjà étudié. Il en diffère par le processus de demande du service à l'annuaire, processus isolé dans la méthode *getServeurSQL*.
- la méthode *getServeurSQL* est identique à celle écrite pour le client d'écho

On voit donc que :

- un client CORBA ne diffère d'un client RMI que par sa façon de contacter le serveur
- cette façon est identique pour tous les clients CORBA

```
import srvSQL.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import java.io.*;

public class clientSQL {

  // données globales de la classe
  private static String syntaxe =
    "syntaxe : cltSQL machine port service pilote urlBase id mdp separateur";
  private static BufferedReader in=null;
  private static interSQL serveurSQL=null;

  public static void main(String arg[]){
    // syntaxe : cltSQL machine port separateur pilote url id mdp
    // machine port : machine & port de l'annuaire des services CORBA à contacter
    // service : nom du service
    // pilote : pilote à utiliser pour la base de données à exploiter
    // urlBase : url jdbc de la base à exploiter
    // id : identité de l'utilisateur
    // mdp : son mot de passe
    // separateur : chaîne séparant les champs dans les résultats d'une requête

    // vérification du nb d'arguments
    if(arg.length!=8)
      erreur(syntaxe,1);

    // init paramètres de la connexion à la base de données
    String machine=arg[0];
    String port=arg[1];
    String service=arg[2];
    String pilote=arg[3];
    String urlBase=arg[4];
    String id, mdp, separateur;
    if(arg[5].equals("null")) id=""; else id=arg[5];
    if(arg[6].equals("null")) mdp=""; else mdp=arg[6];
    if(arg[7].equals("null")) separateur=" "; else separateur=arg[7];

    // paramètres du service d'annuaire CORBA
    String[] initORB={"-ORBInitialHost",arg[0],"-ORBInitialPort",arg[1]};
    // client CORBA - on demande une référence du serveur SQL
    interSQL serveurSQL=getServeurSQL(machine,port,service);

    // dialogue client-serveur
    String requete=null;
    String reponse=null;
    String[] lignes=null;
    String codeErreur=null;

    try{
      // ouverture du flux clavier
      in=new BufferedReader(new InputStreamReader(System.in));
      // suivi
      System.out.println("--> Connexion à la base de données en cours");
      // demande de connexion initiale à la base de données
      reponse=serveurSQL.connect(pilote,urlBase,id,mdp);
      // suivi
```

```
      System.out.println("<-- "+reponse);
      // analyse réponse
      codeErreur=reponse.substring(0,3);
      if(codeErreur.equals("500"))
        erreur("Abandon sur erreur de connexion à la base",3);
      // boucle de lecture des requêtes à envoyer au serveur SQL
      System.out.print("--> Requête : ");
      requete=in.readLine().toLowerCase().trim();
      while(! requete.equals("fin")){
        // envoi de la requête au serveur et réception de la réponse
        lignes=serveurSQL.executeSQL(requete,separateur);
        // suivi
        afficheLignes(lignes);
        // requête suivante
        System.out.print("--> Requête : ");
        requete=in.readLine().toLowerCase().trim();
      }// while
      // suivi
      System.out.println("--> Fermeture de la connexion à la base de données distante");
      // on clôt la connexion
      reponse=serveurSQL.close();
      // suivi
      System.out.println("<-- " + reponse);
      // fin
      System.exit(0);
    // gestion des erreurs
    } catch (Exception e){
      erreur("Abandon sur erreur : " + e,2);
    }// try
  }// main

  // ----------- AfficheLignes
  private static void afficheLignes(String[] lignes){
    for (int i=0;i<lignes.length;i++)
      System.out.println("<-- " + lignes[i]);
  }// afficheLignes

  // ------------ erreur
  private static void erreur(String msg, int exitCode){
    // affichage msg d'erreur
    System.err.println(msg);
    // libération éventuelle des ressources
    try{
      in.close();
      serveurSQL.close();
    } catch(Exception e){}
    // on quitte
    System.exit(exitCode);
  }// erreur

  // ---------------------- getServeurSQL
  private static interSQL getServeurSQL(String machine, String port, String service){
    // demande une référence du serveur SQL
    // machine : machine de l'annuaire des services CORBA
    // port : port de l'annuaire des services CORBA
    // service : nom du service CORBA à demander

    // suivi
    System.out.println("--> Connexion au serveur CORBA en cours...");
    // la référence du serveur SQL
    interSQL serveurSQL=null;
    // paramètres du service d'annuaire CORBA
    String[] initORB={"-ORBInitialHost",machine,"-ORBInitialPort",port};
        try{
      // on demande un objet CORBA pour travailler - on passe pour cela le port
      // d'écoute de l'annuaire des services CORBA
       ORB orb=ORB.init(initORB,null);
      // on utilise le service d'annuaire pour localiser le serveur SQL
       org.omg.CORBA.Object objRef=
         orb.resolve_initial_references("NameService");
       NamingContext ncRef=NamingContextHelper.narrow(objRef);
      // le service recherché s'appelle srvSQL - on le demande
       NameComponent nc= new NameComponent(service,"");
       NameComponent path[]={nc};
       serveurSQL=interSQLHelper.narrow(ncRef.resolve(path));
    } catch (Exception e){
      System.err.println("Erreur lors de la localisation du serveur SQL ("
        + e + ")");
      System.exit(10);
    }// try-catch
    // on rend la référence au serveur
    return serveurSQL;
  }// getServeurSQL

}// classe
```

Compilons la classe du client :

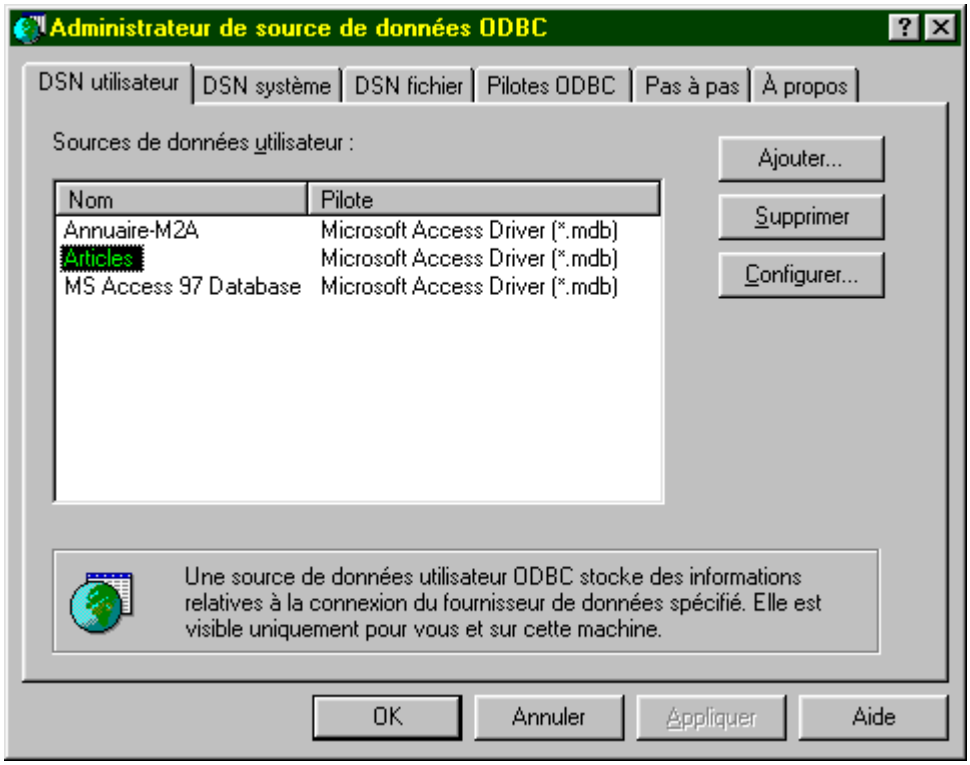
```
E:\data\java\corba\sql>j:\jdk12\bin\javac clientSQL.java

E:\data\java\corba\sql>dir *.class

SQLSER~1 CLA          2 568  19/03/99  10:19 SQLServant.class
SERVEU~1 CLA          1 800  19/03/99  10:33 serveurSQL.class
CLIENT~1 CLA          3 774  19/03/99  10:45 clientSQL.class
```

Nous sommes prêts pour les tests.

9.3.7 Tests

9.3.7.1 Pré-requis

On suppose qu'une base ACSESS appelée Articles est publiquement disponible sur la machine Windows du serveur SQL :



Cette base a la structure suivante :

nom	type
code	code de l'article sur 4 caractères
nom	son nom (chaîne de caractères)
prix	son prix (réel)
stock_actu	son stock actuel (entier)
stock_mini	le stock minimum (entier) en-deça duquel il faut réapprovisioner l'article

9.3.7.2 Lancement du service d'annuaire

```
E:\data\java\corba\sql>start j:\jdk12\bin\tnameserv -ORBInitialPort 1000

Le service d'annuaire est lancé sur le port 1000. Il affiche dans une fenêtre DOS quelque chose du genre :

Initial Naming Context:
IOR:000000000000002849444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436f
```

```
6e746578743a312e3000000000010000000000000030000100000000000a69737469612d30303900
052800000018afabcafe000000027693d3fd00000008000000000000000
TransientNameServer: setting port for initial object references to: 1000
```

9.3.7.3 Lancement du serveur SQL

On lance le serveur Sql :

```
E:\data\java\corba\sql>start j:\jdk12\bin\java serveurSQL localhost 1000 srvSQL
```

Il affiche dans une fenêtre DOS :

```
   Serveur SQL prêt
```

9.3.7.4 Lancement d'un client sur la même machine que le serveur

Voici les résultats obtenus avec un client sur la même machine que le serveur :

```
E:\data\java\corba\sql>j:\jdk12\bin\java clientSQL localhost 1000 srvSQL sun.jdbc.odbc.JdbcOdbcDriver
jdbc:odbc:articles null null ,
--> Connexion au serveur CORBA en cours...
--> Connexion à la base de données en cours
<-- 200 Connexion réussie
--> Requête : select nom, stock_actu, stock_mini from articles
<-- 101 vélo,31,8
<-- 101 arc,9,8
<-- 101 canoé,7,7
<-- 101 fusil,9,8
<-- 101 skis nautiques,13,8
<-- 101 essai3,13,9
<-- 101 cachalot,6,6
<-- 101 léopard,7,7
<-- 101 panthère,7,7
--> Requête : delete from articles where stock_mini<7
<-- 100 1
--> Requête : select nom, stock_actu, stock_mini from articles
<-- 101 vélo,31,8
<-- 101 arc,9,8
<-- 101 canoé,7,7
<-- 101 fusil,9,8
<-- 101 skis nautiques,13,8
<-- 101 essai3,13,9
<-- 101 léopard,7,7
<-- 101 panthère,7,7
--> Requête : fin
--> Fermeture de la connexion à la base de données distante
<-- 200 Base fermée
```

9.3.7.5 Lancement d'un client sur une autre machine que celle du serveur

Voici les résultats obtenus avec un client sur une autre machine que celle du serveur :

```
E:\data\java\corba\sql>j:\jdk12\bin\java clientSQL tahe.istia.univ-angers.fr 1000 srvSQL
sun.jdbc.odbc.JdbcOdbcDriver jdbc:odbc:articles null null ,
--> Connexion au serveur CORBA en cours...
--> Connexion à la base de données en cours
<-- 200 Connexion réussie
--> Requête : select * from articles
<-- 101 a300,v_lo,1202,31,8
<-- 101 d600,arc,5000,9,8
<-- 101 d800,canoé,1502,7,7
<-- 101 x123,fusil,3000,9,8
<-- 101 s345,skis nautiques,1800,13,8
<-- 101 f450,essai3,3,13,9
<-- 101 z400,léopard,500000,7,7
<-- 101 g457,panthère,800000,7,7
--> Requête : fin
--> Fermeture de la connexion à la base de données distante
```

```
<-- 200 Base fermée
```

9.4 Correspondances IDL - JAVA

Nous donnons ici, les correspondances entre types simples IDL et JAVA :

type IDL	type Java
boolean	**boolean**
char	**char**
wchar	**char**
octet	**byte**
string	**java.lang.String**
wstring	**java.lang.String**
short	**short**
unsigned short	**short**
long	**int**
unsigned long	**int**
long long	**long**
unsigned long long	**long**
float	**float**
double	**double**

Rappelons que pour définir un tableau d'éléments de type T dans l'interface IDL, on utilise l'instruction :

typedef sequence<T> nomType;

et qu'ensuite on utilise *nomType* pour référencer le type du tableau. Ainsi, dans l'interface du serveur SQL on a utilisé la déclaration :

typedef sequence<string> resultats;

pour que *resultats* désigne un tableau *String[]* sous Java.