

Le langage Java

Daniel Tounissoux *

29 septembre 2003

Table des matières

1 Quelques principes	4
2 Les principaux types primitifs	6
3 Opérateurs et expressions	8
4 Les instructions de contrôle	10
5 Classes et objets	11
6 Méthodes et constructeurs	16
7 Les tableaux	20
8 Héritage	23
9 Classes abstraites et interfaces	27
10 Interface graphique : les bases	28
11 Les principaux composants visuels	30
11.1 Les boutons : JButton	30
11.2 Les panneaux : JPanel	32
11.3 Les étiquettes : JLabel	32
11.4 Les fenêtres d'édition : JTextField	32
11.5 Les cases à cocher : JCheckBox	33
11.6 Les boutons radio : JRadioButton	33

*Université Claude-Bernard Lyon1 43 Bd du 11 Novembre 69622 Villeurbanne Cedex

12 Gestion des événements :ActionListener	35
13 Les gestionnaires de mise en forme	37
13.1 Le gestionnaire BorderLayout	37
13.2 Le gestionnaire FlowLayout	37
13.3 Le gestionnaire GridLayout	38
13.4 Le gestionnaire BoxLayout	38
13.5 Le gestionnaire CardLayout	38
13.6 Le gestionnaire GridBagLayout	39
13.7 En conclusion	39
14 Composants visuels complexes	40
14.1 Menus déroulants :JMenuBar	40
14.2 Les barres d'outils :JToolBar	41
14.3 Boîtes de listes :JList	43
14.4 Boîtes combinées :JComboBox	43
14.5 Panneau de défilement :JScrollPane	44
14.6 Panneau texte :JTextArea	45
14.7 Panneaux à onglets :JTabbedPane	46
14.8 Les grilles :JTable	47
15 Les fichiers textes	50
15.1 Opérations d'écriture	50
15.2 Opérations de lecture	51
15.3 Complément :la classe StreamTokenizer	51
16 Les exceptions	53
16.1 Protection d'un bloc :try et catch	53
16.2 Lancer une exception :throw	55
16.3 Le bloc finally	56
17 Les boîtes de dialogue	57
17.1 La classe JOptionPane	57
17.2 La classe JDialog	58
17.3 La classe JFileChooser	59
18 Les graphiques	61

<i>TABLE DES MATIÈRES</i>	3
19 Les événements de bas niveau liés à la souris	62
Annexe	64
1 Javadoc	64
2 Fichiers jar	64

1 Quelques principes

Le langage Java est un langage “objet”. Un programme est constitué de classes. Chaque classe fait l’objet d’un fichier comportant le même nom que la classe et l’extension `.java`.

On prendra soin de respecter la casse (majuscules/minuscules) même dans les noms de fichiers sous Windows.

On pourra distinguer :

- *les applications* qui peuvent s’exécuter directement comme n’importe quel programme, et qui peuvent comporter ou non une interface graphique.
- *les applets* qui doivent être exécutées dans une page HTML.

Nous nous intéressons ici à l’écriture d’applications.

Un application est caractérisée par la présence dans une classe d’une procédure *main* qui sert de point d’entrée au programme. Il est toujours possible d’insérer dans une classe une procédure *main* même si celle-ci ne sert qu’à effectuer des tests.

Exemple. Ecriture d’un programme sans interface graphique permettant d’afficher “bonjour”.

On crée le fichier `Bonjour.java` suivant :

```
public class Bonjour {
    public Bonjour() {
    }
    public static void main(String arg[]){
        System.out.println("bonjour");
    }
} // Bonjour
```

le programme est compilé par :

```
javac Bonjour.java
```

(ne pas oublier l’extension `java`)

Un fichier *Bonjour.class* est créé. Ce n’est pas un exécutable : il ne peut être exécuté que par la machine virtuelle java en faisant :

```
java Bonjour
```

(ne pas indiquer l’extension `class`). Le résultat du programme doit s’afficher dans la fenêtre console.

Ceci appelle un certain nombres de remarques.

- Le syntaxe du langage est très proche du C/C++.

- Habituellement le nom des classes commence par une majuscule (attention, Java fait la différence entre les majuscules et les minuscules)
- La procédure `Bonjour` est un constructeur. Elle ne précise aucun type à retourner.
- La procédure `main` est déclarée *static*, ce qui signifie qu'elle peut être exécutée sans avoir besoin d'instancier un objet de la classe.
- Ce programme n'instancie aucun objet de la classe. En général un objet de la classe sera instancié, comme on le verra dans des exemples plus loin.
- Le paramètre *arg* de la procédure `main` sert à passer des arguments à la ligne de commande.
- *System* représente une classe comportant le champ *out* qui est lui même un objet (représentant la console) et comprenant lui même les méthodes `print` et `println`.

Remarque. La compilation peut être exécutée avec *jikes*, compilateur java développé par IBM et distribué gratuitement (il existe une version Windows et une version Linux). *Jikes* est écrit en C, tandis que *javac* est écrit en java, aussi *jikes* est-il plus rapide que *javac*. La compilation peut s'effectuer en indiquant explicitement la librairie runtime *rt.jar* de java. Par exemple, on écrira :

```
jikes -classpath /usr/lib/jdk1.3/jre/lib/rt.jar Bonjour.java
```

(modifier le chemin en fonction de la configuration du système)

2 Les principaux types primitifs

Ce sont les seuls types du langage qui ne sont pas des classes (pour des raisons d'efficacité).

- type *entier signé* **byte** (1 octet) **short** (2 octets) **int** (4 octets) **long** (8 octets)

- types *flottant* **float** (4 octets) et **double** (8 octets). On peut utiliser la notation exponentielle pour les constantes. Implicitement une constante telle que 12.5 représente un type **double**, on peut écrire 12.5f pour imposer le type **float**.

- type *caractère* **char** (4 octets, codage unicode) 'a' 'A'... '\n' (saut de ligne) '\r' (retour chariot) '\t' (tabulation)

- type *booléen* **boolean** : peut prendre deux valeurs **true** et **false**

les déclarations se font comme en C:

```
int i,n ;
float x,y ;
```

les initialisations peuvent se faire en même temps :

```
int i=0;int n=i;
boolean ok=false ;
```

A la compilation Java signale comme une erreur les variables qui n'ont pas été initialisées.

Attention : la durée de vie d'une variable est limitée au bloc `{ }` dans lequel elle a été déclarée. Ainsi l'écriture :

```
{double x=1.5; x *=2 ;}
System.out.println(x);
```

provoque une erreur de compilation : la variable **x** dans **println** n'est pas déclarée.

Constantes

Il n'existe pas réellement de constantes en Java, mais seulement des variables que l'on ne peut pas modifier. Elles sont déclarées par le mot clef **final**. Leur valeur n'est pas nécessairement définie au moment de la déclaration. Par exemple, on pourra avoir :

```
final int k;
...
k=10;
```

toute tentative de modification de **k** conduit à une erreur de compilation.

La portée (durée de vie) d'une variable correspond au bloc dans lequel elle est déclarée. Ainsi la syntaxe suivante est-elle correcte :

```
for (int i=1;i<=5;i++){
    final int k=i;
    System.out.println(k);
}
```

On voit aussi sur cet exemple qu'une variable peut être déclarée seulement à l'instant où on en a besoin.

Classes associées aux types primitifs

L'utilisation des types primitifs peut poser quelques problèmes du fait que ces types ne sont pas des classes. Aussi on dispose de classes associées à chacun des types primitifs; parmi celles-ci les classes *Integer*, *Float*, *Double*, *Boolean*, *Character*.

La classe **Integer** possède de nombreuses méthodes opérant sur des entiers et des chaînes de caractères. Voici quelques exemples utilisant la classe *Integer*; les autres classes **Float**, **Double**, **Boolean** possèdent des possibilités analogues :

Pour définir un entier **Integer** à partir d'un entier **int**, on pourra écrire :

```
int i=3;
Integer ii=new Integer(i);
```

on crée ainsi une instance de la classe **Integer**.

Transformation d'un **Integer** en **int**.

```
int k=ii.intValue() ;
```

intValue() est une méthode statique de la classe **Integer** qui retourne un *int*.

On peut imprimer directement un **Integer** :

```
System.out.println(ii) ;
```

On peut même écrire :

```
int i=2 ; Integer ii=new Integer(4);
System.out.println(""+ii+i) ;
```

on obtiendra l'impression de 42. Grâce à la présence de la chaîne "", le compilateur a transformé *i* et *ii* en **String**.

On peut transformer un **Integer** en **String**:

```
String = ii.toString();
```

Ou bien :

```
String = ""+ii;
```

ce qui fonctionne aussi bien si *ii* est du type **int** !

(en fait la fonction *toString()* est une méthode de toutes les classes, et elle est systématiquement appelée par la fonction **print**).

Inversement, pour transformer une chaîne en **int** :

```
String s="22"; int ii=Integer.parseInt(s);
```

parseInt est une méthode statique de la classe **Integer**, permettant d'interpréter une chaîne comme un entier **int**

3 Opérateurs et expressions

Affectation

Le symbole de l'affectation est `=`

```
int a ;  
a=2 ;
```

`a=2` réalise l'affectation de la valeur 2 à la variable `a`. mais cette affectation est également une expression qui prend la valeur 2. Ainsi

```
int a,x ;  
x=(a=2) ;
```

est correct, et peut être simplifié en :

```
int a,x ; x=a=2 ;
```

Les instructions telles que :

```
x+=2; // ajoute la valeur 2 à la valeur de x
```

s'utilisent comme en C.

On dispose de `--` , `*=` , `/=`

Notons que `+=` peut être appliqué à des chaînes de caractères `String`

Opérateurs arithmétiques

- + addition
- soustraction
- * multiplication
- / division
- % modulo

Ces opérateurs peuvent opérer sur des réels ou des entiers (!)

`/` sur des entiers fournit le résultat de la division entière

`12.5 % 3.5` donne environ 2 (réel).

Les opérateurs `++` (incréméntation de une unité) et `--` (décréméntation de une unité) s'utilisent comme en C, mais sont aussi utilisables sur des réels.

Conversions implicites et explicites

En principe les opérandes d'une expression doivent tous être de même type. Des conversions vers le type le plus complexe sont effectuées automatiquement.

Une conversion explicite peut être effectuée de la manière suivante :

```
int a; float x; x=(float)a ;
```

Attention, dans l'autre sens :

```
int a; float x=3.6f; a=(int)x ;
```

`a` recevra la valeur tronquée 3.

Si on écrit `float x=3.6` on a une erreur car 3.6 est interprété comme du type double: on voit que le transtypage n'est pas implicite dans l'affectation on a un message " possible perte de précision ".

Les opérateurs relationnels

`<` (`<=`) inférieur
`>` (`>=`) supérieur
`==` égal
`!=` différent

Les opérateurs logiques

`!` négation
`&` et
`|` ou inclusif
`&&` et (avec court-circuit)
`||` ou inclusif (avec court-circuit)

4 Les instructions de contrôle

if

```
if (<condition>) {<instruction(s)>;} [else {<instruction(s)>;}]
```

switch

```
switch (<expression>){
case <cste1> : <instructions1>;
case <cste2> : <instructions2>;
...
case <csten> : <instructionsn>;
[default : <instructions>]
}
```

Le résultat de l'expression, et donc les constantes ne peuvent être que du `typeid`, `short`, `byte` ou `char`.

Si `expression` prend la valeur `cste2`, alors `instructions2` est exécutée, puis `instructions3` ...

Ce type d'instruction est le plus souvent employée sous la forme :

```
switch (<expression>){
case <cste1> : <instructions1>;break;
case <cste2> : <instructions2>;break;
...
case <csten> : <instructionsn>;break;
[default : <instructions>]
}
```

do...while

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

while...

```
while (<condition>) {instructions;}
```

for

```
for (<initialisation>;<test d'arrêt>;<incrémentation>) {instructions;}
```

break, continue

Dans une boucle `break` arrête le déroulement normal de la boucle, tandis que `continue` permet de passer au tour de boucle suivant.

5 Classes et objets

Syntaxe

Le plus souvent une classe sera définie par la syntaxe du type suivant :

```
public class MaClasse {
    // partie déclaration des champs
    private int a,b,s ;

    // constructeurs
    MaClasse(<paramètre>){[instructions;]}

    // méthodes
    void plus(){instructions;}

    int max(){instructions;}

    void affiche(instructions;)

    // méthode main
    public static void main(String args[]){ instructions ; }
}
```

`a,b` sont des champs de la classe. La déclaration de ces variables peut comprendre des initialisations : par exemple :

```
int a=3 ;
String s="bonjour" ;
```

`MaClasse(...)` est un constructeur de la classe.

`plus()` et `max()` sont des méthodes (fonctions) de la classe. `plus()` retourne une valeur `int`, `max()` ne retourne pas de valeur. Le constructeur est une méthode qui porte le même nom que la classe, et ne retourne pas de valeur (pas même `void`).

`main()` est une méthode particulière qui sert de point d'entrée et permet d'exécuter un programme qui utilisera les méthodes de la classe. Le fait que `main` soit déclarée `static` fait que cette méthode pourra être appelée même si cette classe n'a pas été instanciée.

Les éléments publics pourront être appelés de n'importe quelle classe.

utilisation d'une classe

Si la classe ne possède pas de fonction `main`, ses méthodes ne pourront être utilisées qu'à partir d'une autre classe. Les méthodes non statiques ne pourront être appelées qu'en instanciant un objet de cette classe, et cela à partir d'une autre classe. Cela pourra se faire de la manière suivante :

```
MaClasse x=new MaClasse();
```

ou en 2 temps :

```
MaClasse x;
...
x=new MaClasse();
```

La première instruction est une déclaration. La deuxième crée une instance `x` de la classe `MaClasse` en faisant appel au constructeur ; les instructions éventuelles contenues dans le constructeur sont exécutées à cette occasion.

Si la classe possède une fonction `main` (nécessairement statique) cette fonction pourra être appelée directement sans instanciation de la classe, mais ne pourront être appelées que des fonctions elles-mêmes statiques.

Le plus souvent la fonction `main` créera elle-même une instance de la classe :

```
public static void main(String args[]){
    MaClasse x=new MaClasse();
    x.plus();
}
```

Exemple

```
class MaClasse {

    private int a,b,s ;

    MaClasse(){

    void plus(){s=a+b;}

    int max(){if (a>b) return a; else return b;}

    void affiche(){System.out.println("max= "+max()+" somme= "+s);}

    public static void main(String args[]){
        MaClasse x=new MaClasse();
        x.a=Integer.parseInt(args[0]);
        x.b=Integer.parseInt(args[1]);
        x.plus();
        x.affiche();
    }
} // MaClasse
```

Cette classe pourra être utilisée comme un programme en faisant :

```
java MaClasse 3 4
```

le résultat est :

```
max= 4 somme= 7
```

On peut voir aussi sur cet exemple comment on peut utiliser les paramètres de la ligne de commande, et comment on peut transformer une chaîne (`String`) en un entier en utilisant la méthode statique `parseInt` de la classe `Integer`.

Affectation de classes

Si `a` et `b` sont deux objets du mêmes type (instances d'une même classe), on peut affecter `a` à `b`:

```
b=a;
```

Dans ce cas c'est l'adresse de `b` qui sera affectée à `a`, comme le montre l'exemple suivant :

```
class MaClasse {

    int val ;

    MaClasse(){

    public static void main(String args[]){
        MaClasse a= new MaClasse();
        MaClasse b= new MaClasse();
        a.val=1;b=a;System.out.println(a.val+" "+b.val);
        b.val=2;System.out.println(a.val+" "+b.val);
    }
} // MaClasse
```

Le résultat obtenu est :

```
1 1
2 2
```

Pour obtenir une vraie copie d'une classe on devra prévoir une méthode copie dans la classe, par exemple :

```
class MaClasse {

    int val ;

    MaClasse(){

    MaClasse copie(){
        MaClasse x=new MaClasse();
        x.val=val;
        return x;
    }

    public static void main(String args[]){
        MaClasse a= new MaClasse();
        MaClasse b= new MaClasse();

        a.val=1;b=a.copie();System.out.println(a.val+" "+b.val);
        b.val=2;System.out.println(a.val+" "+b.val);
    }
}
```

```

    }

} // MaClasse
dont l'exécution donnera :

1 1
1 2

```

Naturellement, il faudra être prudent si la classe contient des objets.

Comparaison de classes

On peut comparer deux objets à l'aide de l'opérateur de comparaison `==`. Mais cet opérateur compare l'adresse des deux objets et non pas leur contenu : ceci est fréquemment source d'erreurs.

La méthode `equals` permet de comparer le contenu de deux objets. Naturellement lors de la construction de nouvelles classes cette méthode devra être redéfinie (voir les paragraphes concernant les méthodes et l'héritage pour les définitions des termes *méthodes* et *redéfinition*).

La classe `String` permet d'illustrer les pièges liés à la comparaison des objets.

```

public class Essai {

    Essai(){
        String s=new String("bonjour");
        String t=new String("bonjour");
        System.out.println(t==s);
        System.out.println(t.equals(s));
    }

    public static void main(String arg[]){
        Essai e=new Essai();
    }
}

```

L'exécution donne :

```

false
true

```

Les deux chaînes n'ont pas la même adresse, mais elles ont le même contenu.

Par contre l'exemple suivant :

```

public class Essai {

    Essai(){
        String s="bonjour";
        String t="bonjour";
        System.out.println(t==s);
        System.out.println(t.equals(s));
    }
}

```

```
    }

    public static void main(String arg[]){
        Essai e=new Essai();
    }
}
```

donne :

```
true
true
```

parce que, pour des raisons d'optimisation, le compilateur ne crée qu'une instance pour les deux chaînes !

On évitera bien des problèmes en utilisant systématiquement `equals` pour comparer deux chaînes et plus généralement deux objets.

Classes internes

Depuis la version 1.1, Java permet de construire des classes internes à une autre classe suivant le modèle suivant :

```
public class MaClasse{
    //...méthodes et données de la classe externe

    public class ClasseInterne{
        //...méthodes et données de la classe interne
    }
}
```

La classe externe pourra instancier des objets du type de la classe interne. La classe interne a accès aux champs et méthodes de la classe interne, et réciproquement.

Le principal intérêt des classes internes est d'alléger le code, en plaçant dans une classes interne un certains nombre de champs et méthodes. Il est clair que si une classes interne risque de pouvoir être réutilisée, on a intérêt à en faire une classe ordinaire, bien que il soit possible d'instancier une classe interne publique à partir d'une autre classe non englobante. La syntaxe serait alors la suivante :

```
MaClasse.ClasseInterne x;
```

Une classe interne peut être définie à l'intérieur d'une méthode, on parle alors de classe interne locale. La classe interne locale ne peut alors être utilisée qu'à l'intérieur de cette méthode.

6 Méthodes et constructeurs

méthodes

Les classes contiennent des champs ordinaires (variables de type simple ou complexes) et des méthodes. Les champs et les méthodes constituent les membres de la classe.

Les **méthodes** sont des fonctions (procédures) qui font partie intégrantes de la classes. Elles peuvent être déclarées statiques (**static**): dans ce cas elles pourront être utilisées même si la classe n'est pas instanciée (toutefois elles ne devront avoir accès qu'à des données elles-mêmes statiques).

Contrairement à C++ La déclaration et la définition d'une méthode se font simultanément, comme dans l'exemple précédent :

```
MaClasse copie(){
    MaClasse x=new MaClasse();
    x.val=val;
    return x;
}
```

MaClasse est ici le type de retour: ce type peut être une classe, un type simple, ou bien **void**.

La méthode peut comprendre des paramètres également de type quelconques.

```
public class MaClasse {

    public MaClasse(){}

    void prod(int u,int v){
        int i,j;
        for (i=1;i<=u;i++){
            for (j=1;j<=v;j++){
                System.out.print(i*j+" ");
                System.out.println();
            }
        }

        public static void main(String args[]){
            MaClasse x=new MaClasse();
            x.prod(4,5);
        }

    } // MaClasse
```

Cet exemple montre comment définir une méthode **prod** et comment on l'appelle. Attention dans la déclaration le type doit être exprimé pour chaque variable; ainsi la déclaration **void prod(int a, b)** serait incorrect.

Il est possible de définir plusieurs méthodes avec des paramètres différents (types ou nombre) : on dit que les méthodes ont des signatures différentes.

Pour reprendre l'exemple précédent :

```
public class MaClasse {

    public MaClasse(){}

    void prod(int u,int v){
        int i,j;
        for (i=1;i<=u;i++){
            for (j=1;j<=v;j++){
                System.out.print(i*j+" ");
                System.out.println();
            }
        }

    void prod(int u){ prod(u,u); }

    public static void main(String args[]){
        MaClasse x=new MaClasse();
        x.prod(4);
    }

} // MaClasse
```

On a défini une seconde procédure **prod** que l'on appellera avec un seul paramètre. Selon le nombre de paramètres, java déterminera la procédure à appeler.

Attention : en Java tous les paramètres sont passés par valeur. Ceci n'est pas très gênant si on considère que :

- la plupart des méthodes opèrent sur les champs de la classe, et ainsi, le plus souvent, elles n'ont pas de paramètres,
- les paramètres sont souvent des classes, donc des adresses, et il est possible d'en modifier le contenu

constructeurs

Les constructeurs sont des méthodes particulières qui servent à initialiser les instances de classes à l'aide de l'instruction **new**. Leurs particularités sont :

- les constructeurs ont le même nom que la classe qui les contiennent,
- ils n'ont pas de type de retour.

Dans une classe il peut y avoir plusieurs constructeurs avec des signatures différentes.

On rencontrera souvent des situations comme dans l'exemple suivant où on a 3 constructeurs :

```
public class MaClasse {
```

```

double val ;

public MaClasse(int val){
    this.val=val;
    System.out.println("constructeur "+1);
}

public MaClasse(float val){
    this.val=val;
    System.out.println("constructeur "+2);
}

public MaClasse(double val){
    this.val=val;
    System.out.println("constructeur "+3);
}

void ecrire(){System.out.println(val);}

public static void main(String args[]){
    MaClasse x=new MaClasse(4.0);
    x=new MaClasse(4.0f);
    x=new MaClasse(4);
    x.ecrire();
}

} // MaClasse

```

L'exécution donne :

```

constructeur 3
constructeur 2
constructeur 1
4.0

```

Cet exemple montre comment on utilise un constructeur pour définir une nouvelle instance de classe.

On peut noter que la déclaration peut être séparée de l'instanciation, par exemple :

```

MaClasse x ;
...
x=new MaClasse(5.0);

```

Une classe peut ne pas posséder de constructeur. Le système définit alors un constructeur par défaut sans paramètre qui pourra être utilisé.

Ainsi dans l'exemple du paragraphe précédent sur les méthodes, on peut supprimer l'instruction :

```

public MaClasse(){ }

```

Notons enfin que la partie déclaration des champs peut comprendre des appels à un constructeur, par exemple :

```
public class AutreClasse {  
    MaClasse x=new MaClasse(3.5);  
  
    public AutreClasse(){ ....}  
    ...  
} // AutreClasse
```

7 Les tableaux

Généralités

Si on veut définir un tableau de réels on pourra écrire :

```
int n=5;
...
float t[]=new float[n];
```

La valeur `n` fournie n'est prise en compte qu'au moment de l'exécution, et non au moment de la compilation comme en Pascal ; elle peut donc être variable.

L'accès aux éléments du tableau se fera avec `t[0], ..., t[4]`.

La déclaration peut être séparée de la réservation mémoire :

```
float t[];
...
t=new float[5];
```

(la déclaration `float t[]` est équivalente à `float[] t`.)

On peut définir directement le contenu, en effectuant la réservation mémoire de façon implicite :

```
float t[]={1,2,3,4,5};
```

Le passage de paramètres tableaux dans les procédures se fait par des déclarations du type :

```
void proc(float t[],...)
```

Dans la procédure la taille pourra être obtenu par `t.length`. Attention `length` est un champ et non pas une méthode (sinon on écrirait `t.length()`).

Lors de l'appel de la procédure c'est l'adresse du tableau qui est passée, il est donc possible de modifier les valeurs du tableau.

Affectations

Les tableaux se comportent à peu près comme des classes. Lors d'une affectation, c'est l'adresse qui est recopiée, comme le montre l'exemple suivant :

```
public class MaClasse {

    int t1[]={1,1,1},t2[]={2,2,2} ;

    public static void main(String args[]){

        MaClasse a= new MaClasse();
        a.t1=a.t2 ;
        a.t1[0]=3 ;
        System.out.println(a.t2[0]);
    }
} // MaClasse
```

L'exécution provoque l'affichage de la valeur 3.

L'ancienne valeur du tableau `t1` est perdue, et la place sera récupérée par le ramasse-miettes.

Tableaux à plusieurs indices

Il n'existe pas proprement dit de tableaux à plusieurs indices en Java, mais il est possible de les simuler.

On peut écrire :

```
int t[][] ;
```

ce qui signifie que `t` est une référence à un tableau dont chacun des éléments est une référence à un tableau d'entiers.

Il faudra donc initialiser `t` par :

```
t=new int[2][] ;
int t1[]=new int[3] ;t[0]=t1 ; // ou bien t[0]=new int[3]
int t2[]=new int[4] ;t[1]=t2 ; // ou bien t[1]=new int[4]
```

ou plus simplement :

```
t={ new int[3] , new int[4]} ;
```

Le premier élément de `t` est donc un vecteur à 3 éléments, tandis que le deuxième est un vecteur à 4 éléments.

Ceci se simplifie si on veut une matrice ordinaire ; on pourra écrire :

```
int n=5; int p=4 ;
t=new int[n][p] ;
```

Voici un petit exemple qui résulte de ce qui précède :

```
public class MaClasse {
    int t[][]={{1,1,1},{2,2,2,2}} ;

    public static void main(String args[]){

        MaClasse a= new MaClasse();
        System.out.println(a.t[0][0]);
    }
} // MaClasse
```

Le nombre de lignes de `t` est donné par `t.length`, et le nombre d'éléments de la première ligne par `t[0].length`

Le type de base (ici `int`) peut être un type primitif quelconque (`int`, `double`, `boolean`, ..., ou une classe quelconque ;

La classe Vector

Cette classe fait partie du package `util`, et doit être importée explicitement en écrivant au début du fichier :

```
import java.util.Vector ;
```

ou bien :

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

La classe `Vector` permet de manipuler des vecteurs d'objets quelconques, de façon dynamique. Elle contient (entre autres les) deux constructeurs suivants :

```
Vector()  
Vector(int n)
```

qui construisent respectivement un vecteur vide, et un vecteur de capacité initiale de `n` éléments.

Les principales méthodes sont ;

```
void add(int index, Object element)  
void add(Object element)  
Object elementAt(int index) ou Object get(int index)  
boolean isEmpty()  
void remove(int index)  
int size()
```

Voici un petit exemple d'utilisation :

```
import java.util.* ;  
  
class MaClasse {  
  
    static Vector v=new Vector();  
  
    public static void main(String a[]){  
        v.add(new String("un "));  
        v.add(new String("deux "));  
        System.out.println((String)v.get(0)+(String)v.get(1)+v.size());  
    }  
  
} // MaClasse
```

Les classes `LinkedList` et `ArrayList` sont d'utilisation très semblable, et diffèrent surtout par leur implémentation qui les rend plus ou moins efficaces suivant les circonstances.

8 Héritage

Généralités

L'héritage permet de définir une nouvelle classe à partir d'une classe existante en lui apportant quelques modifications.

L'en-tête d'une classe dérivée a une syntaxe du type suivant :

```
public class AutreClasse extends MaClasse { ... }
```

On dit que `AutreClasse` hérite ou étend `MaClasse`, ou qu'elle en est une classe dérivée. On dit aussi que `MaClasse` est une classe ancêtre de `AutreClasse`.

Dans `AutreClasse` on pourra trouver :

- de nouveaux champs
- de nouvelles méthodes
- des méthodes portant le même nom et la même signature que dans `MaClasse` : on dit que ces méthodes sont redéfinies, et elles viennent remplacer les méthodes correspondantes de `MaClasse`.
- des méthodes portant le même nom et des signatures différentes : on dit que ces méthodes sont surdéfinies.

Java n'admet que l'héritage simple (une classe ne peut dériver directement de plusieurs classes), bien que l'utilisation d'interfaces (que nous verrons ultérieurement) puisse être considérée comme une forme d'héritage multiple.

Par rapport à C++ ou Delphi, où une méthode peut être déclarée virtuelle ou non, Java ne dispose qu'un type de méthode qui correspond au type virtuel de C++.

Exemple

La classe `Stat` calcule les statistiques élémentaires sur les n premiers entiers

```
public class Stat {
    double x[]; int n ;

    public Stat(int n){
        this.n=n ; x=new double[n];
        for (int i=0;i<n;i++) x[i]=i;
    }

    public double moyenne(){
        double m=0;
        for (int i=0;i<n;i++) m+=x[i];
        m=m/n; return m;
    }

    private double moyenneCarres(){
        double m=0;
        for (int i=0;i<n;i++) m+=x[i]*x[i];
        m=m/n ; return m;
    }
}
```

```

    }

    public double variance(){
        double m=moyenne();
        return moyenneCarres()-m*m ;
    }

    public double ecartType(){
        return Math.sqrt(variance());
    }

    public void afficheStat(){
        System.out.println("moyenne = "+moyenne());
        System.out.println("variance = "+variance());
        System.out.println("ecart-type = "+ecartType());
    }

    public static void main(String args[]){
        Stat s= new Stat(10);
        s.afficheStat();
    }
} // Stat

```

La classe StatPoids réalise le même objectif, mais chaque valeur est supposée affectée d'un poids:

```

public class StatPoids extends Stat {
    int p[]; // tableau des poids

    public StatPoids(int n) {
        super(n);
        p=new int[n];
        for (int i=0;i<n;i++) p[i]=n-i;
    }

    private double sommePoids(){
        int s=0; int i;
        for (i=0;i<n;i++) s+=p[i];
        return s;
    }

    public double moyenne(){
        double m=0;
        for (int i=0;i<n;i++) m+=x[i]*p[i];
        m=m/sommePoids(); return m;
    }

    private double moyenneCarres(){
        double m=0;
        for (int i=0;i<n;i++) m+=p[i]*x[i]*x[i];
    }
}

```

```

        m=m/sommePoids() ; return m;
    }

    public static void main(String args[]){
        Stat s= new StatPoids(10);
        s.afficheStat();
    }
} // StatPoids

```

Si vous pensez qu'il est inutile de calculer les statistiques sur la série des nombres entiers, pensez que si vous voulez calculer les statistiques sur d'autres séries il suffira de construire une classe dérivée qui redéfinira la méthode qui définit la série (ici le constructeur `Stat` ou `StatPoids`).

Remarques

La classe `StatPoids` redéfinit le constructeur. Mais on évite de tout réécrire en faisant appel au constructeur de la classe ancêtre : c'est ce que réalise l'instruction `super()`.

Plus généralement, il est possible de faire appel à une méthode ancêtre en écrivant `super.xxxx` (ou `xxxx` est le nom de la méthode). `super` désigne la classe ancêtre.

L'instruction `Stat s= new StatPoids(10);` peut sembler incorrecte. En fait la classe `StatPoids` est une classe `Stat` puisse qu'elle en dérive. En fait on aurait même pu définir `s` par `Object s= new StatPoids(10);` ceci aurait posé un problème à la compilation car `afficheStat` n'est pas un membre de `Object` ; on aurait dû réaliser un transtypage pour imposer au compilateur de considérer `s` comme un objet de type `Stat` ou `StatPoids`, et on aurait écrit :

```

    public static void main(String args[]){
        Object s= new StatPoids(10);
        ((StatPoids)s).afficheStat();
    }

```

L'intérêt de tout cela c'est qu'au moment où on déclare une variable objet, il n'est pas indispensable de connaître exactement son type. Le type définitif sera défini au moment de l'appel du constructeur.

Droits d'accès

Les membres d'une classe peuvent être précédés de l'un des attributs *public*, *private* ou *protected* ; ces attributs définissent les droits d'accès à partir d'autres classes :

- *public* : accessibilité à partir de n'importe quelle autre classe,
- *private* : accessibilité uniquement à partir de la classe,
- *protected* : accessibilité uniquement à partir des classes dérivées,
- absence d'attribut : accessibilité uniquement à partir des classes du même *package* (nous ne parlons pas de ce concept ici).

Classes anonymes

Il est possible de définir une classe interne sans lui donner de nom en la dérivant d'une autre classe. Supposons que nous disposions d'une classe `Classe0`. On souhaite utiliser cette classe en modifiant quelques méthodes. On pourrait évidemment définir une classe `Classe1` étendant `Classe0` ; mais on peut arriver au même résultat et éviter de créer `Classe1`, en utilisant une syntaxe de la forme :

```
c = new Classe0(){  
    // surdéfinition de certaines des méthodes de Classe0  
}
```

La classe créée, définissant l'objet `c`, n'a pas reçu de nom, on dit que c'est une classe anonyme.

9 Classes abstraites et interfaces

Classes abstraites

La déclaration d'une classe *abstraite* est précédée de l'attribut *abstract* ; une telle classe ne peut pas être instanciée et sert de base pour dériver d'autres classes, en s'assurant que ces classes posséderont les champs ou méthodes dont elle dispose. Une telle classe est nécessairement publique.

Une classe abstraite peut posséder des méthodes elles-mêmes abstraites : ce sont des méthodes dont on indique seulement l'en-tête. Si une classe possède une méthode abstraite, elle est nécessairement abstraite.

Exemple :

```
public abstract class ClasseBase {

    String Titre="ClassBase" ;

    abstract void afficheTitre() ;
} // ClasseBase
```

Notons qu'une classe abstraite peut dériver d'une classe non abstraite (cette possibilité est rarement utilisée).

Interfaces

Les interfaces ressemblent beaucoup aux classes abstraites, mais elles sont d'une utilisation plus riche ; elles permettent une sorte d'héritage multiple.

Les interfaces sont repérées par le mot clef *interface*. Elles ne comportent que des constantes et n'implémentent aucune méthode.

Exemple.

```
public interface Int {
    static final String TITRE="XXXX";
    void afficheTitre();
}
```

Toute classe utilisant cette interface devra implémenter la méthode `AfficheTitre` elle sera déclarée comme suit en utilisant le mot clef *implements* :

```
public class StatPoids extends Stat implements Int {
    ...
}
```

Cette classe devra implémenter `afficheTitre()`. On voit que cette classe étend `Stat` et `Int`. Une classe peut faire appel à plusieurs interfaces, on écrira simplement :

```
public class StatPoids extends Stat implements Int, Int2, Int3 {
```

Une application importante des interfaces sera vue au §12. Événements. L'interface `ActionListener` sera implémentée pour gérer les événements clics de souris, au moyen de la méthode `ActionPerformed(ActionEvent e)`.

10 Interface graphique : les bases

Java permet de construire facilement des interfaces graphiques qui simplifient et agrémentent l'utilisation des programmes.

Deux paquetages facilitent la création d'interfaces graphiques : **awt** et **swing**. **swing** est disponible depuis la version 1.2 le Java, il est basé sur **awt** mais offre plus de possibilités. La plupart des composants ont une version dans chacun des paquetages ; ceux basé sur **swing** ont un nom qui commence en général par la lettre J. L'utilisation des deux versions présente de petites différences. Nous présentons ici l'utilisation de **swing**.

Création d'une fenêtre de base : **JFrame**

JFrame est le composant de base qui permet de créer une fenêtre dans laquelle on pourra ajouter des composants visuels tels que boutons, graphiques, fenêtres d'édition ...

Le programme suivant permet d'afficher une fenêtre héritant de **JFrame** :

```
import javax.swing.* ;

public class Fenetre extends JFrame {

    public Fenetre() {
        super("première fenêtre");
        setSize(300,200);
        setLocation(100,100);
    }

    public static void main(String args[]){
        Fenetre f=new Fenetre();
        f.show(); // ou bien f.setVisible(true);
    }

} // Fenetre
```

On notera l'instruction `import javax.swing.* ;` qui permet d'importer le paquetage **swing** ainsi que l'appel au constructeur ancêtre, la définition de la taille et de la position de la fenêtre.

Si la fenêtre créée, possède toutes les fonctionnalités classiques, on pourra constater que le fait de fermer la fenêtre ne ferme pas l'application (sous Linux, on pourra la fermer en tapant Control-C).

Pour pouvoir fermer l'application en même temps que la fenêtre, on procédera comme suit, en ajoutant à la fenêtre un composant **WindowAdapter** dont on aura surchargé la méthode **windowClosing** de sorte qu'elle puisse fermer l'application. L'utilisation de ce composant nécessite l'utilisation de l'importation du paquetage **event** (`import java.awt.event.* ;`).

```
import javax.swing.* ;
import java.awt.event.*;

public class Fenetre extends JFrame {

    public Fenetre() {
        super("première fenêtre");
        setSize(300,200);
        setLocation(100,100);
        WindowListener l=new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        };
        addWindowListener(l);
    }

    public static void main(String args[]){
        Fenetre f=new Fenetre();
        f.setVisible(true);
    }
}
```

Naturellement une classe **Fenetre** pourra être instanciée à partir d'une classe, et l'instance créée pourra être affichée.

Pour connaître toutes les possibilités de **JFrame** on se référera à la documentation de Sun disponible sur internet.

11 Les principaux composants visuels

11.1 Les boutons : JButton

Les boutons sont parmi les composants les plus utilisés. Nous profitons de cette présentation pour montrer comment on peut ajouter des composants à une fenêtre.

Les boutons héritent de la classe `JComponent`.

```
import javax.swing.* ;
import java.awt.event.*;
import java.awt.*;

public class Fenetre extends JFrame {

    public Fenetre() {
        super("JButton");
        setSize(300,200);
        setLocation(100,100);
        WindowListener l=new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        };
        addWindowListener(l);
        Container cont=getContentPane();
        cont.setLayout(new FlowLayout());
        JButton b=new JButton("essai");
        cont.add(b);
    }

    public static void main(String args[]){
        Fenetre f=new Fenetre();
        f.setVisible(true);
    }

} // Fenetre
```

le résultat est le suivant :

Les instructions permettant de disposer le bouton sont les suivantes :

- `import java.awt.*` : pour pouvoir accéder aux composants `Container` et `FlowLayout`.
- `Container cont=getContentPane()` : permet d'accéder au conteneur de la fenêtre dans lequel tous les composants seront ajoutés;
- `cont.setLayout(new FlowLayout())` : définit la façon dont seront disposés les composants; `FlowLayout` s'appelle un *gestionnaire de mise en forme* (*LayoutManager*) ; il s'agit du gestionnaire le plus simple: les composants seront disposés dans l'ordre de la gauche vers la droite et de haut en bas, suivant la taille du conteneur. Il existe d'autres gestionnaires plus élaborés qui seront décrits plus tard.
- `JButton b=new JButton("essai")` : crée le bouton.
- `cont.add(b)` : ajoute le bouton au conteneur de la fenêtre.

Il est possible de se passer d'un gestionnaire de mise en forme (cette possibilité est rarement utilisée car elle présente des inconvénients.) Pour cela on remplacera les quatre lignes décrites précédemment par ce qui suit :

```
Container cont=getContentPane();
cont.setLayout(null);
JButton b=new JButton("essai");b.setBounds(10,10,50,20);
cont.add(b);
```

La méthode `setBounds` héritée de `Component` permet de définir la position et la taille du bouton. En général, avec un gestionnaire de mise en forme, ces paramètres sont gérés automatiquement en fonction de la taille du conteneur.

La méthode `setMargin` permet de contrôler les marges du bouton, autour de son contenu (texte ou image). Ainsi pour définir une marge de 1 pixel, on écrira :

```
b.setMargin(new Insets(1,1,1,1));
```

Il est possible de placer un icône sur un bouton, en utilisant un constructeur approprié comme suit (en supposant que l'on dispose d'une icône `fileclose.png` dans un fichier) :

```
ImageIcon icon = new ImageIcon("fileclose.png");
b2=new JButton(icon);
```

Remarque importante. Il est à noter que, dans les classes ainsi définies, les boutons ne savent pas répondre aux événements de la souris. Nous verrons un peu plus loin comment ajouter cette possibilité.

11.2 Les panneaux : JPanel

Les panneaux sont des conteneurs permettant de regrouper un ensemble de composants. Ils peuvent disposer de leur propre gestionnaire de mise en forme. Ils seront le plus souvent utilisés comme suit :

```
Container cont=getContentPane();
cont.setLayout(new FlowLayout());
JPanel p=new JPanel();p.setLayout(new FlowLayout());
JButton b1=new JButton("Fichier");
JButton b2=new JButton("Quitter");
p.add(b1); p.add(b2);
cont.add(p);
```

Les panneaux peuvent être munis de bords permettant de les mettre en relief. Pour cela on utilise une classe `BorderFactory` ; par exemple après la création du panneau dans l'exemple précédent, on pourra écrire :

```
Border bord=BorderFactory.createTitledBorder("essai bord");
// ou bien : Border bord=BorderFactory.createEtchedBorder();
p.setBorder(bord);
```

Il faudra importer le paquetage `border` avec la directive :

```
import javax.swing.border.* ;
```

11.3 Les étiquettes : JLabel

Le composant `JLabel` permet d'afficher un texte non éditable. L'utilisation est toujours identique :

```
JLabel l1=new JLabel("Fichier");
cont.add(l1);
```

Il est possible de placer l'étiquette dans un panneau, ou de placer un bord sur l'étiquette ...

11.4 Les fenêtres d'édition : JTextField

Les fenêtres d'édition sont des champs de saisie qui permettent de saisir sur une ligne, une chaîne de caractère ou une valeur numérique.

Une fenêtre d'édition pourra être ajoutée à un panneau comme suit

```
JTextField t1=new JTextField("Fichier",20);
p.add(t1);
cont.add(p);
```

Les paramètres du constructeur `TextField`, peuvent être `(String,int)` (chaîne par défaut et longueur de la fenêtre) ou bien `(String)` ou bien `(int)`.

Pour avoir accès au contenu de la fenêtre, on pourra écrire :

```
String s=t1.getText();
```

Si on souhaite traiter le contenu comme un entier, on écrira :

```
int n=Integer.parseInt(t1.getText());
```

et, pour obtenir un réel :

```
float n=Float.parseFloat(t1.getText());
```

11.5 Les cases à cocher : `JCheckBox`

Les cases à cocher servent à désigner une option qui peut être validée ou non ; il leur est associé une étiquette qui sert à préciser la nature de l'option.

La création s'effectue par une instruction du type suivant :

```
JCheckBox cb1 = new JCheckBox("Quadrillage ",false);  
p.add(cb1);
```

La valeur booléenne de la case s'obtient par :

```
boolean valeur=cb1.isSelected();
```

Les cases à cocher peuvent être groupées dans un conteneur `JCheckBoxGroup` de sorte qu'une seule case ne peut être cochée à un instant donné. Cette possibilité est plus souvent utilisée avec les boutons radio.

11.6 Les boutons radio : `JRadioButton`

Les boutons radio ont les mêmes fonctions que les cases à cocher. Ils sont le plus souvent groupés dans un `ButtonGroup` ; grâce à cela, deux boutons ne peuvent être cochés simultanément.

Il faut ajouter les boutons au groupe, ainsi qu'au panneau, comme dans le schéma suivant :

```
ButtonGroup bg=new ButtonGroup();  
JRadioButton br1 = new JRadioButton("Loi uniforme",true);  
bg.add(br1);  
JRadioButton br2 = new JRadioButton("Loi normale ",false);  
bg.add(br2);  
p.add(br1);  
p.add(br2);
```

Notons qu'il est tout à fait possible de faire un tableau de boutons radio pour simplifier l'écriture :

```
ButtonGroup bg=new ButtonGroup();
JRadioButton b[] = new JRadioButton[5];
for (int i=0;i<=4;i++){
    b[i] = new JRadioButton("Loi normale",false);
    bg.add(b[i]);p.add(b[i]);
}
```

Naturellement l'étiquette de chaque bouton devra être modifiée à l'aide de la méthode `setText(String)`, par exemple :

```
b[1].setText("Loi uniforme");
```

12 Gestion des événements : ActionListener

On distingue les événements de bas niveau tels que click de souris et les événements de haut niveau (dits sémantiques), tels que l'activation d'un bouton (qui peut être réalisée aussi par la barre d'espace). Le composant `ActionListener` est un écouteur qui permet de gérer simplement un certain nombre d'événements de haut niveau générés par les différents types de boutons (cases à cocher ...), de menus, de champs de saisie. Ceci est suffisant pour un grand nombre d'applications.

Supposons que nous disposions dans la classe `Fenetre` de deux boutons `b1` et `b2` ainsi que de deux méthodes `m1()` et `m2()`. On souhaite associer à un click sur `b1` l'appel à `m1()` et à un click sur `b2` l'appel à `m2()`.

Pour cela la fenêtre principale (ou un conteneur contenant `b1` et `b2`) doit implémenter l'interface `ActionListener` (que l'on peut traduire par écouteur d'événement). L'en tête de la classe `Fenetre` pourra être :

```
public class Fenetre extends JFrame implements ActionListener {
```

ceci impose de redéfinir la méthode `actionPerformed(ActionEvent ev)` de l'interface `ActionListener` ; on devra donc trouver dans la classe fenêtre une méthode sur le modèle suivant :

```
    public void actionPerformed(ActionEvent ev){
        .....
    }
```

Par ailleurs les boutons devront être munis d'un écouteur d'événements, après leur intanciation, comme suit :

```
b1=new JButton("bouton1");
b1.addActionListener(this);cont.add(b1);
b2=new JButton("bouton2");
b2.addActionListener(this);cont.add(b2);
```

Il ne reste plus qu'à compléter la méthode `actionPerformed` comme suit :

```
    public void actionPerformed(ActionEvent ev){
        Object source=ev.getSource();
        if (source==b1) m1() ;
        if (source==b2) m2() ;
    }
```

Attention à inclure la directive suivante :

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

pour que le compilateur trouve l'interface `ActionListener`.

Voici un exemple qui illustre le fonctionnement de 2 boutons permettant l'afficher un texte dans un champ de saisie ; nous avons ajouté un bouton qui permet de fermer l'application.

```
import javax.swing.* ;
import java.awt.event.*;
import java.awt.*;

public class Fenetre extends JFrame
    implements ActionListener {

    JButton b1,b2,b3;
    JTextField t ;

    public Fenetre() {
        super("Événements");
        setSize(300,200);
        setLocation(100,100);
        WindowListener l=new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        };
        addWindowListener(l);
        Container cont=getContentPane();
        cont.setLayout(new FlowLayout());
        b1=new JButton("Bouton1");
        b1.addActionListener(this);  cont.add(b1);
        b2=new JButton("Bouton2");
        b2.addActionListener(this);  cont.add(b2);
        t=new JTextField(10); cont.add(t);
        b3=new JButton("fin");
        b3.addActionListener(this);  cont.add(b3);
    }

    public void actionPerformed(ActionEvent ev){
        Object source=ev.getSource();
        if (source==b1) t.setText("bouton1") ;
        if (source==b2) t.setText("bouton2") ;
        if (source==b3) System.exit(0);;
    }

    public static void main(String args[]){
        Fenetre f=new Fenetre();
        f.setVisible(true);
    }
} // Fenetre
```

13 Les gestionnaires de mise en forme

Les gestionnaires de mise en forme permettent de disposer les composants sur une fenêtre. Ils peuvent être associés à un panneau ou au conteneur de la fenêtre principale.

13.1 Le gestionnaire BorderLayout

C'est souvent le gestionnaire par défaut. On l'associe au conteneur de la fenêtre en écrivant :

```
Container cont=getContentPane();
cont.setLayout(new BorderLayout());
```

Ce gestionnaire divise le panneau en 5 zones : **Center**, **North**, **South**, **East**, **West**. Il suffira de placer les composants en précisant la zone de la manière suivante :

```
JButton b=new JButton("Exe");
cont.add("North",b); // attention à la majuscule
```

Pour éviter qu'un composant tel qu'un bouton occupe toute la zone, on pourra d'abord le placer sur un panneau, et ajouter ce panneau à la zone.

Le plus souvent on placera des boutons sur la zone **North** et un panneau sur la zone **Center**, pour afficher un dessin ou un texte par exemple.

13.2 Le gestionnaire FlowLayout

Ce gestionnaire place les composants, dans l'ordre où ils sont ajoutés, les uns à la suite des autres, sur une même ligne s'il y a assez de place, sur des lignes différentes sinon. Sur chaque ligne, les composants peuvent être placés au choix à gauche, à droite ou au centre en passant au constructeur une des valeurs entières suivantes :

```
FlowLayout.LEFT
FlowLayout.RIGHT
FlowLayout.CENTER (valeur par défaut)
```

par exemple :

```
Container cont=getContentPane();
cont.setLayout(new FlowLayout(FlowLayout.LEFT));
cont.add(b);
```

La combinaison **FlowLayout** et **BorderLayout** permet de construire facilement des présentations élaborées.

13.3 Le gestionnaire GridLayout

Ce gestionnaire place les composants sur une grille régulière, chaque composant occupant une cellule; par exemple:

```
Container cont=getContentPane();
JButton b[];
b=new JButton[12];
JPanel p=new JPanel(new GridLayout(3,4));
for (int i=0;i<=11;i++){
    b[i]=new JButton("bouton"+i);
    p.add(b[i]);
}
cont.add(p);
```

Pour varier nous avons placé des boutons sur un panneau et non directement sur le conteneur principal.

Ce gestionnaire est un peu rigide si l'on souhaite utiliser des composants différents; en effet dans ce cas il est préférable que les composants n'aient pas des tailles identiques.

13.4 Le gestionnaire BoxLayout

Ce gestionnaire permet de disposer des composants suivant une seule ligne ou une seule colonne. Il s'utilise en général par l'intermédiaire d'un conteneur `Box` qui est doté par défaut d'un gestionnaire `BoxLayout`.

Pour créer un box horizontal, on utilise:

```
Box boxh=Box.createHorizontalBox();
```

et un box vertical:

```
Box boxv=Box.createVerticalBox();
```

Les composants seront ensuite ajoutés avec la méthode `add`:

```
JLabel l=new JLabel("Essai ");JTextField t=new JTextField(" ");
boxh.add(l); boxh.add(t);
```

Rien n'empêche de placer plusieurs box horizontaux dans un box vertical. Chaque composant aura sa dimension définie en fonction de sa nature et de la place dont il dispose; ainsi les `JTextField` auront une dimension variable et les boutons une dimension constante.

La combinaison des box et d'un `BorderLayout` permet de résoudre simplement la plupart des problèmes.

13.5 Le gestionnaire CardLayout

Ce gestionnaire permet de disposer des composants suivant une pile, un seul élément de la pile étant visible à un instant donné.

Nous citons seulement ce gestionnaire pour mémoire, le composant Swing `JTabbedPane` (boîte à onglet) étant d'utilisation plus pratique.

13.6 Le gestionnaire GridBagLayout

Nous citons également ce gestionnaire pour mémoire. Il permet des constructions complexes mais est d'utilisation complexe. Il organise les composants suivant une grille, mais un composant peut utiliser plusieurs cellules horizontalement et verticalement ; de plus des poids permettent d'ajuster la largeur et la hauteur des composants.

13.7 En conclusion

Les gestionnaires peuvent sembler d'utilisation complexe. Ils permettent en fait de construire des interfaces qui garderont un aspect correct dans différents environnements (système, résolution, polices de caractères ...).

Si vraiment on est allergique à ces gestionnaires, il ne faut pas oublier que, comme on l'a déjà évoqué, on peut mettre le gestionnaire à `null` ; on précisera alors la taille et la position des composants.

Il est possible de construire ses propres gestionnaires, ou d'en trouver tout construits sur internet (certains sites en proposent des dizaines).

14 Composants visuels complexes

14.1 Menus déroulants : JMenuBar

Pour créer un menu déroulant, il y a essentiellement 3 opérations :

- création de la barre menu :

```
JMenuBar menuBar=new JMenuBar();
```
- création des menus :

```
JMenu couleur=new JMenu("couleur"); menuBar.add(couleur);
JMenu taille=new JMenu("taille"); menuBar.add(taille);
```
- création des options :

```
rouge=new JMenuItem("rouge");
rouge.addActionListener(this);couleur.add(rouge);
bleu=new JMenuItem("bleu");
bleu.addActionListener(this);couleur.add(bleu);
```

Naturellement, il faudra ajouter la barre des menus à la fenêtre, et adjoindre aux différentes options comme indiqué ci dessus.

Voici un petit exemple qui utilise un menu pour changer la couleur d'un panneau et changer la taille de la fenêtre principale.

```
import javax.swing.* ;
import java.awt.event.*;
import java.awt.*;

public class Fenetre extends JFrame implements ActionListener{

    JMenuItem rouge,bleu,petit,grand ;
    JPanel p0;

    public Fenetre() {
        super("essai menu");
        setSize(300,200);
        setLocation(100,100);
        WindowListener l=new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        };
        addWindowListener(l);
        Container cont=getContentPane();
        cont.setLayout(new BorderLayout());

        JMenuBar menuBar=new JMenuBar();
        JMenu couleur=new JMenu("couleur");menuBar.add(couleur);
        JMenu taille=new JMenu("taille");menuBar.add(taille);
```

```

        rouge=new JMenuItem("rouge");
        rouge.addActionListener(this);couleur.add(rouge);
        bleu=new JMenuItem("bleu");
        bleu.addActionListener(this);couleur.add(bleu);

        grand=new JMenuItem("grand");
        grand.addActionListener(this);taille.add(grand);
        petit=new JMenuItem("petit");
        petit.addActionListener(this);taille.add(petit);

        p0=new JPanel();

        cont.add("North",menuBar);
        cont.add(p0);
    }

    public void actionPerformed(ActionEvent ev){
        Object source=ev.getSource();
        if (source==rouge) p0.setBackground(Color.red);
        if (source==bleu ) p0.setBackground(Color.blue);
        if (source==grand ) {setSize(500,400);validate();}
        if (source==petit ) {setSize(300,200);validate();}
    }

    public static void main(String args[]){
        Fenetre f=new Fenetre();
        f.setVisible(true);
    }
} // Fenetre

```

L'instruction `validate()` permet d'ajuster la taille du panneau à la nouvelle taille de la fenêtre.

14.2 Les barres d'outils: JToolBar

Les barres d'outils servent le plus souvent à grouper des boutons, bien qu'elles puissent contenir d'autres composants.

Voici un exemple analogue au précédent, mais géré par une barre d'outils.

```

import javax.swing.* ;
import java.awt.event.*;
import java.awt.*;

public class Fenetre extends JFrame implements ActionListener{

    JButton rouge,bleu,petit,grand ;

```

```

JPanel p0;

public Fenetre() {
    super("essai barre d'outils");
    setSize(300,200);
    setLocation(100,100);
    WindowListener l=new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    };
    addWindowListener(l);
    Container cont=getContentPane();
    cont.setLayout(new BorderLayout());

    JToolBar toolBar=new JToolBar();

    rouge=new JButton("rouge");
    rouge.addActionListener(this);toolBar.add(rouge);
    bleu=new JButton("bleu");
    bleu.addActionListener(this);toolBar.add(bleu);
    grand=new JButton("grand");
    grand.addActionListener(this);toolBar.add(grand);
    petit=new JButton("petit");
    petit.addActionListener(this);toolBar.add(petit);

    p0=new JPanel();

    cont.add("North",toolBar);
    cont.add(p0);
}

public void actionPerformed(ActionEvent ev){
    Object source=ev.getSource();
    if (source==rouge) p0.setBackground(Color.red);
    if (source==bleu ) p0.setBackground(Color.blue);
    if (source==grand ) {setSize(500,400);validate();}
    if (source==petit ) setSize(300,200);
}

public static void main(String args[]){
    Fenetre f=new Fenetre();
    f.setVisible(true);
}

} // Fenetre

```

Par défaut, les barres d'outils peuvent être déplacées en dehors de leur conteneur.

Si on veut interdire cette possibilité, on écrira :

```
toolBar.setFloatable(false);
```

14.3 Boîtes de listes : JList

La boîte de liste sert à choisir une ou plusieurs valeurs dans une liste prédéfinie (en principe, le contenu d'une boîte de liste n'est pas modifiable).

La construction se fait de la façon suivante :

```
String couleurs[]={"bleu","blanc","rouge"};
JList list=new JList(couleurs);
cont.add(list);
```

Le mode de sélection s'effectue au moyen de l'un des paramètres suivant dont la signification est évidente :

```
SINGLE_SELECTION
SINGLE_INTERVAL_SELECTION
MULTIPLE_INTERVAL_SELECTION
```

On écrira par exemple :

```
list.setSelectionMode(SINGLE_SELECTION);
```

La valeur sélectionnée pourra être obtenue par :

```
String sel=(String)list.getSelectedValue();
```

Le transtypage est nécessaire car la valeur retournée est du type `Object`.

Pour récupérer plusieurs valeurs, on pourra écrire :

```
Object sel[]=list.getSelectedValues();
for (int i=0;i<valeurs.length;i++)
    System.out.println((String)sel[i]);
```

On pourra placer une telle boîte dans un panneau de défilement `JScrollPane` ; dans ces conditions, par défaut 8 éléments seront visibles, et on pourra faire défiler les autres éléments.

14.4 Boîtes combinées : JComboBox

Une boîte combinée associe un champ `TextField` éditable ou non, toujours visible, à une boîte de liste qui peut être cachée ou visible. Une telle boîte est souvent utilisée pour sélectionner par exemple une police de caractères.

La construction est très semblable à une boîte de liste :

```
String couleurs[]={"bleu","blanc","rouge"};
list=new JComboBox(couleurs);
cont.add(list);
```

On peut accéder à l'élément sélectionné avec `getSelectedItem()` qui s'utilise comme dans les boîtes de liste (il faut opérer un transtypage si on veut une chaîne de caractères), mais on peut aussi y accéder avec `getSelectedIndex()` qui fournit l'indice de l'élément sélectionné. Dans ce cas, si le champ éditable a été rempli par l'utilisateur, la valeur retournée par `getSelectedIndex()` est -1. Contrairement à une boîte `JList` il est facile de modifier le contenu d'une `JComboBox`. Pour ajouter en fin de liste on utilise :

```
list.addItem("vert");
```

pour insérer en une position quelconque :

```
list.addItemAt("vert",2);
```

et pour supprimer un élément :

```
list.removeItem("vert");
```

14.5 Panneau de défilement : `JScrollPane`

Un panneau de défilement permet de visualiser partiellement un composant de grande taille, en lui adjoignant des barres de défilement. Voici un exemple typique utilisant un panneau qui ne peut pas être contenu sur la fenêtre :

```
import javax.swing.* ;
import java.awt.event.*;
import java.awt.*;

public class Fenetre extends JFrame {

    public Fenetre() {
        super("essai ScrollPane");
        setBounds(100,100,300,200);
        WindowListener l=new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        };
        addWindowListener(l);
        Container cont=getContentPane();
        JPanel p=new JPanel();
        p.setPreferredSize(new Dimension(400,400));
        JScrollPane defil=new JScrollPane(p);
        cont.add(defil);
    }

    public static void main(String args[]){
        Fenetre f=new Fenetre();
        f.setVisible(true);
    }
} // Fenetre
```

L'instruction

```
p.setPreferredSize(new Dimension(400,400));
```

crée un panneau dont la taille préférée est de (400,400), et donc supérieure à la taille de la fenêtre (les barres de défilement sont ajoutées en fonction de la taille préférée; le paramètre de la fonction `setPreferredSize` doit être de type `dimension`).

L'instruction

```
JScrollPane defil=new JScrollPane(p);
```

crée un panneau de défilement contenant `p`. Il suffit d'ajouter ensuite ce panneau au conteneur principal. Les barres de défilement seront gérées automatiquement en fonction de la taille de la fenêtre.

14.6 Panneau texte : JTextArea

Ces panneaux sont destinés à recevoir du texte standard, soit pour afficher des résultats, soit pour saisir des données. Ils sont éditables par défaut. Si on ne veut pas qu'ils soit éditables, on écrira :

```
text.setEditable(false);
```

Voici un exemple typique d'utilisation :

```
import javax.swing.* ;
import java.awt.event.*;
import java.awt.*;

public class Fenetre extends JFrame {

    JPanel p;
    JScrollPane defil ;
    JTextArea text ;

    public Fenetre() {
        super("JTextArea");
        setBounds(100,100,300,200);
        WindowListener l=new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        };
        addWindowListener(l);

        Container cont=getContentPane();
        p=new JPanel();
        defil =new JScrollPane(p);
        text=new JTextArea(50,50);
        for (int i=1;i<=100;i++) text.append(""+i+" hello !\n");
    }
}
```

```

        p.add(text);
        cont.add(defil);
    }

    public static void main(String args[]){
        Fenetre f=new Fenetre();
        f.setVisible(true);
    }
} // Fenetre

```

Nous avons inclus le `JTextArea` dans un panneau et lui même dans un panneau de défilement. L'instruction

```
append()
```

permet d'ajouter du texte à la fin.

On peut ajouter du texte à une position quelconque avec :

```
append(String,int);
```

ou bien remplacer du texte avec :

```
replaceText(String,int,int);
```

la partie du texte comprise entre les deux valeurs sera remplacée par la chaîne.

Il est possible de changer la police de caractères.

14.7 Panneaux à onglets : `JTabbedPane`

Ce sont des conteneurs qui permettent de passer facilement de l'affichage d'un panneau à un autre en cliquant sur un onglet.

On pourra créer un panneau à onglets de la manière suivante :

```
JTabbedPane tp= new JTabbedPane();
```

On ajoutera des panneaux `JPanel` en précisant une chaîne qui sera reproduite sur l'onglet, et le panneau à onglets est ensuite ajouté au panneau principal (par exemple) :

```

container cont=getContentPane();
JPanel p0=new JPanel; tp.addTab("Element 1",p0);
JPanel p1=new JPanel; tp.addTab("Element 2",p1);
...
cont.add(tp);

```

14.8 Les grilles : JTable

Présentation élémentaire

On peut construire une grille vide de la façon suivante :

```
Container cont=getContentPane();
JTable t=new JTable(10,6);
t.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
JScrollPane p=new JScrollPane(t);
```

Le constructeur `JTable()` prend comme paramètres le nombre de lignes et de colonnes. En général on place la table dans un panneau de défilement.

L'instruction `t.setAutoResizeMode(JTable.AUTO_RESIZE_OFF)` est pratiquement indispensable, sinon la taille des cellules est ajustée à la dimension de la fenêtre, et cette taille risque d'être mal adaptée.

les méthodes suivantes permettent d'accéder aux éléments de la table :

```
Object getValueAt(int ligne,int colonne);
setValueAt(Object v,int ligne,int colonne);
```

Compléments

La présentation précédente dissimule les possibilités importantes des grilles `JTable`.

Comme d'autres composants complexes tels que les `JList`, les grilles sont gérées suivant le schéma modèle-vue-contrôleur, c'est à dire que la gestion du tableau des valeurs (modèle) est dissociée de son aspect graphique (vue); le lien entre les deux est assuré par le contrôleur. C'est ce qui explique la relative complexité de ces objets.

Le modèle est défini par un objet de la classe abstraite `AbstractTableModel`. L'utilisation de cette classe impose la définition d'une classe dérivée qui doit implémenter les 3 méthodes suivantes :

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

La classe `DefaultTableModel` est dérivée de la classe `AbstractTableModel` et réalise l'implémentation de ces 3 méthodes, et possède un constructeur qui prend comme paramètres les nombres de lignes et de colonnes ;

La classe `JTable` définit l'aspect graphique de la table; un constructeur prend comme paramètre une instance du modèle défini à partir d'un modèle `AbstractTableModel` ou d'un `DefaultTableModel`.

Une façon équivalente à la précédente de construire une grille vide serait d'écrire :

```
Container cont=getContentPane();
DefaultTableModel dtm=new DefaultTableModel(10,6){
JTable t= new JTable(dtm);
t.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
JScrollPane p=new JScrollPane(t);
cont.add(p);
```

En définissant son propre modèle, à partir d'une matrice, il est possible de gérer de façon automatique les éléments de la matrice à partir de la grille.

L'exemple suivant montre un exemple plus complet d'une `JTable`, qui affiche automatiquement le contenu d'un tableau à deux dimensions, en utilisant une classe locale `MyModel` qui étend la classe `AbstractTableModel` :

```
import javax.swing.* ;
import java.awt.event.*;
import java.awt.*;
import javax.swing.table.*;

public class Fenetre extends JFrame {

    public Fenetre() {
        super("Essai JTable");
        setBounds(100,100,300,200);
        WindowListener l=new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        };
        addWindowListener(l);

        Container cont=getContentPane();

        float m[][]=new float[10][5];
        for (int i=0;i<=9;i++)
            for (int j=0;j<=4;j++)
                m[i][j]=(float)Math.random();

        AbstractTableModel tm=new MyModel(m);
        JTable t=new JTable(tm);
        t.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        JScrollPane p=new JScrollPane(t);

        cont.add(p);
    }

    public static void main(String args[]){
        Fenetre f=new Fenetre();
        f.setVisible(true);
    }

    class MyModel extends AbstractTableModel{
        float m[][] ;
        MyModel(float[][] m0){super();m=m0;}

        public int getRowCount(){return m.length; }
        public int getColumnCount(){return m[0].length; }
```

```

        public Object getValueAt(int row, int column){
            return new Float((float)m[row][column]);
        }
    }

} // Fenetre

```

Si on veut rendre la grille éditable, on peut modifier le modèle comme suit :

```

MyModel(float[] [] m0){super();m=m0;}

public int getRowCount(){return m.length; }
public int getColumnCount(){return m[0].length; }
public Object getValueAt(int row, int column)
    {return new Float((float)m[row][column]);}
public void setValueAt(Object v, int row, int column){
    if (v==null) m[row][column]=0 ;
    else {
        try{
            m[row][column]=Float.parseFloat((String)v);
        }
        catch(Exception e){ m[row][column]=0 ;}
    }
}

public boolean isCellEditable(int i,int j){return true;}
}

```

15 Les fichiers textes

Java dispose d'outils puissants mais complexes pour gérer les fichiers. Nous nous intéressons ici aux opérations élémentaires permettant d'utiliser les fichiers textes.

Pour accéder aux classes nécessaires pour les opérations d'entrées-sorties, on ajoutera au début du fichier la directive :

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

15.1 Opérations d'écriture

Deux classes sont nécessaires pour écrire dans un fichier :

- `FileWriter` qui permet d'ouvrir le fichier en écriture,
- `PrintWriter` qui dispose des méthodes `print` et `println` permettant l'écriture effective.

Voici une méthode permettant d'écrire dans un fichier texte :

```
public void ecrireFichier(){
    try{
        FileWriter fo=new FileWriter("fichier.dat");
        PrintWriter f=new PrintWriter(fo);
        for (int i=0;i<=10;i++) f.println(i+" Hello !");
        f.close();
    }
    catch(Exception e){
        System.out.println("Erreur écriture");
    }
}
```

On notera que le constructeur de `FileWriter` prend comme paramètre le nom du fichier, et celui de `PrintWriter` l'instance de `FileWriter`. La méthode `println` transforme en une chaîne les entiers et les réels.

La méthode `close` ferme le fichier.

L'écriture dans un fichier doit être incluse dans un bloc `try` permettant de gérer une éventuelle exception.

Il est possible d'associer à l'opération de sortie un buffer de la manière suivante :

```
public void ecrireFichier(){
    try{
        FileWriter fo=new FileWriter("fichier.dat");
        BufferedWriter bw=new BufferedWriter(fo);
        PrintWriter f=new PrintWriter(bw);
        for (int i=0;i<=10;i++) f.println(i+" Hello !");
        f.close();
    }
    catch(Exception e){
        System.out.println("Erreur écriture");
    }
}
```

15.2 Opérations de lecture

Les deux classes suivantes permettent de lire dans un fichier :

- `FileReader` qui permet d'ouvrir le fichier en lecture,
- `BufferedReader` qui dispose des méthodes permettant la lecture effective.

Une méthode permettant de lire ligne par ligne aura l'allure suivante :

```
public void lireFichier(){
    try{
        FileReader fi=new FileReader("fichier.dat");
        BufferedReader f=new BufferedReader(fi);
        String line=f.readLine();
        while (line!=null){
            text.append(line+"\n");
            line=f.readLine();
        }
    }
    catch(Exception e){
        System.out.println("Erreur lecture");
    }
}
```

(`text` est supposée être une instance de `JTextArea`)

On peut obtenir le même résultat en envoyant les caractères lus dans une chaîne qui doit être du type `StringBuffer` car une chaîne du type `String` ne peut pas être modifiée. On procédera donc comme suit :

```
public void lireFichier(){
    StringBuffer s=new StringBuffer("");
    try{
        FileReader fi=new FileReader("fichier.dat");
        BufferedReader f=new BufferedReader(fi);
        String line=f.readLine();
        while (line!=null){
            s.append(line+"\n");
            line=f.readLine();
        }
        text.setText(new String(s));
    }
    catch(Exception e){
        System.out.println("Erreur lecture");
    }
}
```

15.3 Complément : la classe `StreamTokenizer`

Cette classe est particulièrement pratique pour lire, entre autres, les valeurs numériques contenues dans des fichiers.

L'exemple suivant lit les éléments d'un fichier texte, séparés par un ou plusieurs espaces ; pour chaque élément, il affiche s'il s'agit, d'un mot, d'un nombre ou de la marque de fin de ligne :

```
public void lireFichier(){
    int x; // type du token
    try{
        FileReader fi=new FileReader("fichier.dat");
        StreamTokenizer f=new StreamTokenizer(fi);
        f.eolIsSignificant(true);
        while (f.nextToken()!=StreamTokenizer.TT_EOF){
            if (f.ttype==StreamTokenizer.TT_WORD )
                text.append("<mot> "+f.sval+"\n");
            if (f.ttype==StreamTokenizer.TT_NUMBER )
                text.append("<nombre> "+f.nval+"\n");
            if (f.ttype==StreamTokenizer.TT_EOL)
                text.append("<fin de ligne>\n");
        }
    }
    catch(Exception e){
        System.out.println("Erreur lecture");
    }
}
```

L'ouverture :

```
FileReader fi=new FileReader("fichier.dat");
StreamTokenizer f=new StreamTokenizer(fi);
```

se fait en utilisant `StreamTokenizer` au lieu de `BufferedReader`.

L'instruction

```
f.eolIsSignificant(true);
```

signifie que la fin de ligne ne sera pas considérée comme un espace ordinaire.

La fonction

```
f.nextToken()
```

retourne un entier qui caractérise le type de l'élément suivant ; cet entier est stocké dans la variable `f.ttype`.

Suivant la valeur de `f.ttype` on affiche un message et la valeur de l'élément qui est stockée dans `nval` s'il s'agit d'une valeur numérique, et dans `sval` s'il s'agit d'un mot (valeur non numérique, c'est à dire une chaîne).

16 Les exceptions

Une *exception* se produit lorsque une erreur telle qu'une division par zéro, tentative de lecture d'un fichier qui n'existe pas ... Le traitement des exceptions est destiné à éviter un arrêt du programme et à permettre à l'utilisateur de corriger son erreur.

En Java une exception est une classe qui est instanciée lorsqu'une erreur s'est produite. Il existe une classe `Exception` dont dérivent toutes les autres exceptions parmi lesquelles on peut citer `IOException`, `EOFException`, ...

16.1 Protection d'un bloc : try et catch

Si certaines instructions risquent de provoquer une erreur, on devra les protéger. Ceci consiste à inclure les instructions dangereuses dans un bloc `try`, et faire suivre ce bloc d'une instruction `catch` comme suit :

```
try{
    <instructions>
}
catch(Exception e){
    <message ou instructions à réaliser si une exception s'est produite>
}
```

Si tout se passe bien à l'intérieur du bloc `try` le contrôle du programme se poursuit après l'instruction `catch`, sinon le contrôle est passé au bloc `catch` : on dit que l'exception a été interceptée.

Il peut y avoir plusieurs instructions `catch` correspondant à un traitement séparé des différentes exceptions pouvant être émises ; par exemple :

```
try{
    <instructions>
}
catch(IOException e){
    <traitement 1>
}
catch(EOFException e){
    <traitement 2>
}
```

Exemple : Dans l'exemple suivant on veut afficher dans un `JTextArea` le contenu d'un fichier dont le nom doit être saisi par l'utilisateur dans un `JTextField`. Il est clair que l'utilisateur peut se tromper et indiquer un fichier qui n'existe pas, d'où la nécessité de protéger ces instructions. Ceci peut être réalisé de la manière suivante :

```
import javax.swing.* ;
import java.awt.event.*;
import java.awt.*;
```

```

import java.io.* ;

public class Fenetre extends JFrame implements ActionListener {

    JPanel p=new JPanel();
    JButton b=new JButton("exe");
    JTextField tf = new JTextField("",10);
    JTextArea text=new JTextArea(50,50);

    public Fenetre() {
        super("Exceptions");
        setBounds(100,100,300,200);
        WindowListener l=new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        };
        addWindowListener(l);

        Container cont=getContentPane();
        b.addActionListener(this);
        p.add("North",b);p.add("North",tf);cont.add("North",p);
        cont.add("Center",text);
    }

    public void lireFichier(){
        try{
            FileReader fi=new FileReader(tf.getText());
            BufferedReader f=new BufferedReader(fi);
            String line=f.readLine();
            while (line!=null){
                text.append(line+"\n");
                line=f.readLine();
            }
        }
        catch(Exception e){
            System.out.println("Erreur lecture");
        }
    }

    public void actionPerformed(ActionEvent ev){
        Object source=ev.getSource();
        if (source==b) lireFichier();
    }

    public static void main(String args[]){
        Fenetre f=new Fenetre();
        f.setVisible(true);
    }
} // Fenetre

```

En fait deux exceptions sont lancées, l'une `FileNotFoundException` par le constructeur `FileReader` et l'autre `IOException` par la fonction `ReadLine()` (la documentation en ligne Java fournit ce genre d'information). Ces deux exceptions ont été traitées ensemble, mais on aurait pu les traiter séparément dans des blocs `catch` séparés.

16.2 Lancer une exception : throw

Le mécanisme des exceptions permet d'obliger l'utilisateur de traiter ces exceptions, mais il lui laisse le choix de l'instant où il veut le faire.

Par exemple, le constructeur `FileReader` prévoit une exception mais il ne la traite pas : le traitement est réalisé dans la fonction `lireFichier`. Il pourrait se faire que l'on ne veuille pas traiter cette exception dans `lireFichier`, mais seulement quand on clique sur le bouton par exemple. Il suffit pour cela de relancer l'exception dans `lireFichier` et de la traiter dans `ActionPerformed`, ce qui pourra se faire avec la clause `throws` comme suit (nous réécrivons seulement la partie modifiée) :

```
public void lireFichier() throws Exception {
    FileReader fi=new FileReader(tf.getText());
    BufferedReader f=new BufferedReader(fi);
    String line=f.readLine();
    while (line!=null){
        text.append(line+"\n");
        line=f.readLine();
    }
}

public void actionPerformed(ActionEvent ev){
    Object source=ev.getSource();
    try{
        if (source==b) lireFichier();
    }
    catch(Exception e){
        System.out.println("Erreur lecture");
    }
}
```

Nous avons dit que l'utilisateur doit traiter toute exception émise. Que se passerait-il si on n'avait pas indiqué `throws Exception` dans `lireFichier` ni créé un bloc `try` dans cette fonction ? Nous aurions eu, à la compilation, des messages d'erreur la forme suivante :

```
Fenetre.java:32: unreported exception
java.io.FileNotFoundException; must be caught or declared
to be thrown
```

```
    FileReader fi=new FileReader(tf.getText());
    ^
```

```
Fenetre.java:34: unreported exception java.io.IOException;
must be caught or declared to be thrown
```

```
String line=f.readLine();
```

Notons que l'utilisateur peut créer ses propres classes d'exception (souvent locales) comme suit :

```
class MonException extends Exception{};
```

A n'importe quel instant une fonction peut déclencher l'exception, généralement à l'issue d'un test, de la façon suivante :

```
throw new MonException();
```

Naturellement l'en-tête de la fonction devra indiquer `throws MonException` ; et l'exception devra être traitée par un bloc `try` lors de l'appel de la fonction.

Notons que même si une exception a été traitée dans un bloc `try...catch`, elle peut être relancée explicitement, comme dans l'exemple suivant :

```
try{ ... }  
catch(Exception e){ <message> ; throw e; }
```

Conclusion Retenons que si une fonction lance implicitement ou explicitement une exception, celle-ci doit être traitée localement, par un bloc `try...catch`, ou son en-tête doit mentionner cette exception au moyen d'une clause `throws`.

16.3 Le bloc `finally`

Après le dernier bloc `catch` on peut ajouter un bloc `finally` :

```
try{ ... }  
catch(Exception e){ <message> ; }  
finally{...}
```

Le contenu du bloc `finally` sera exécuté dans tous les cas, que l'exception ait été déclenchée ou pas.

17 Les boîtes de dialogue

Les boîtes de dialogue servent à envoyer un message à l'utilisateur, demander une confirmation, saisir une information ... Naturellement on peut créer des boîtes de dialogue en les faisant dériver de `JFrame`, mais Java nous propose un ensemble de boîtes de dialogues prédéfinies et d'utilisation facile.

Les boîtes de dialogue sont généralement **modales** ce qui signifie qu'elles seules sont actives et donc que l'application ne peut être poursuivie qu'après leur fermeture.

Les possibilités présentées ici, bien que permettant de faire face à de nombreuses situations, ne représentent qu'un faible pourcentage des possibilités réelles.

17.1 La classe `JOptionPane`

Cette classe possède un certain nombre de méthodes *statiques* qui permettent d'afficher des boîtes de dialogue courantes.

La méthode `showMessageDialog`

Cette méthode peut être utilisée avec l'en-tête suivant :

```
void showMessageDialog(Component parent, Object message)
```

Dans ce cas elle affiche une boîte de dialogue ayant pour titre "Message") et comportant une icône, un message et un bouton "OK". Le composant `parent` désigne la fenêtre à l'intérieur de laquelle on veut afficher la boîte (il pourra être mis à `null`, dans ce cas la fenêtre par défaut sera choisie, le plus souvent l'écran entier); `message` peut théoriquement être un objet quelconque, mais sera le plus souvent du type `String`.

Le bouton "OK" ferme automatiquement la boîte de dialogue.

Le plus fréquemment l'appel se fera par :

```
showMessageDialog(this, "Hello ...")
```

La méthode peut également être utilisée de la façon suivante :

```
void showMessageDialog(Component parent, Object message,  
String title, int messageType)
```

ce qui permet de choisir le titre; `messageType` est une constante entière qui permet de définir l'icône et pourra être choisie parmi :

```
JOptionPane.ERROR_MESSAGE  
JOptionPane.WARNING_MESSAGE  
JOptionPane.QUESTION_MESSAGE  
JOptionPane.PLAIN_MESSAGE
```

(la dernière option correspond à l'absence d'icône)

La méthode `showConfirmDialog`

L'utilisation est très semblable à `showMessageDialog`, mais la boîte affichée attend une confirmation par l'intermédiaire de plusieurs boutons tels que "Yes" "No" ...

Voici l'en-tête sous sa forme la plus simple :

```
int showConfirmDialog(Component parent, Object message)
```

Dans ce cas les boutons affichés seront "Yes" "No" "Cancel". La fonction retourne une valeur entière, correspondant au bouton pressé, parmi les suivantes :

```
JOptionPane.OK_OPTION    (=0)
JOptionPane.YES_OPTION    (=0)
JOptionPane.NO_OPTION     (=1)
JOptionPane.CANCEL_OPTION (=2)
JOptionPane.CLOSED_OPTION (=-1)
```

L'autre version de la méthode est :

```
int showConfirmDialog(Component parent, Object message,
    String title, int optionType)
```

ce qui permet de choisir le titre; `OptionType` peut être choisi parmi :

```
JOptionPane.DEFAULT_OPTION
JOptionPane.YES_NO_OPTION
JOptionPane.YES_NO_CANCEL_OPTION
JOptionPane.OK_CANCEL_OPTION
```

La première option (`DEFAULT-OPTION`) correspond à l'affichage d'un unique bouton "OK".

La méthode `showInputDialog`

Cette fonction permet d'afficher une boîte comportant une fenêtre d'édition; son en-tête sous sa forme la plus simple est la suivante :

```
String showInputDialog(Component, Object message)
```

La boîte affichée comporte un bouton "OK" et un bouton "Cancel". La valeur retournée est le contenu de la fenêtre d'édition.

Signalons que cette fonction permet d'afficher une boîte de dialogue contenant une liste d'options.

17.2 La classe `JDialog`

Cette classe d'utilisation plus conforme à l'esprit de la programmation objet java permet de construire des boîtes de dialogue personnalisées. Cette classe s'utilise comme `JFrame`, mais les fenêtres construites peuvent être rendues modales; on devra leur ajouter des boutons des champs de saisie ..., et elles devront le plus souvent implémenter `ActionListener` pour répondre aux actions des boutons et autres composants.

Le principal constructeur est :

```
JDialog(Frame parent, String title, boolean modal)
```

Le dernier paramètre permet de rendre la boîte modale ou non.

Un fois la boîte construite, son affichage proprement dit se fera à l'aide de :

```
setVisible(true)
```

La fin du dialogue se fait avec :

```
setVisible(false)
```

Cette dernière instruction ne détruit pas la boîte ; la place mémoire pourra être récupérée par la méthode `dispose()` qui récupère aussi la place des composants ajoutés, sans avoir à être surchargée.

17.3 La classe `JFileChooser`

C'est sans doute une des boîtes de dialogue des plus utiles ; elle permet de sélectionner un nom de fichier pour chargement ou sauvegarde. Cette classe `swing` représente une grande amélioration par rapport à la classe `awt` `FileDialog`.

Les principales méthodes sont :

```
int showOpenDialog(Component parent)
int showSaveDialog(Component parent)
```

La valeur retournée peut-être :

```
JFileChooser.CANCEL_OPTION
JFileChooser.APPROVE_OPTION
JFileChooser.ERROR_OPTION
```

Voici une utilisation typique de cette classe.

```
JFileChooser chooser = new JFileChooser();
int returnVal = chooser.showOpenDialog(parent);
if(returnVal == JFileChooser.APPROVE_OPTION) {
    System.out.println("Vous avez sélectionné le fichier : " +
        chooser.getSelectedFile().getName());
}
```

Souvent on ne souhaite pas afficher tous les fichiers, mais seulement ceux possédant une certaine extension, par exemple ".txt" ou ".dat". On utilisera pour cela une classe dérivant de la classe `FileFilter` en redéfinissant les méthodes

```
boolean accept(File f);
String getDescription();
```

`accept(f)` doit retourner vrai si et seulement si le fichier `f` possède l'extension voulue (dans notre exemple ".txt" ou ".dat"), et `getDescription()` retourne une chaîne de caractères permettant la description des fichiers concernés (par exemple "txt, dat : fichiers de données").

Nous affecterons à l'objet `chooser` défini plus haut, le filtre grâce à la méthode `setFileFilter()` :

```
void setFileFilter(FileFilter ff);
```

Par exemple, nous définissons une classe `FileFilterDat` comme suit :

```
public class FileFilterDat extends FileFilter {

    public boolean accept(File f){
        boolean res=false ;
        StringTokenizer st=new StringTokenizer(f.getName(),".");
        String ext=st.nextToken();
        while (st.hasMoreTokens()) ext=st.nextToken();
        if (ext!=null){
            if (ext.equals("dat") || ext.equals("txt"))
                res=true;
        }
        return res;
    }

    public String getDescription(){
        return "dat, txt (fichiers de données)";
    }
}
```

L'utilisation de la boîte de sélection se fera alors comme suit :

```
JFileChooser chooser = new JFileChooser();
chooser.setFileFilter(new FileFilterDat());
int returnVal = chooser.showOpenDialog(parent);
if(returnVal == JFileChooser.APPROVE_OPTION) {
    System.out.println("Vous avez sélectionné le fichier : " +
        chooser.getSelectedFile().getName());
}
```

La boîte est fermée automatiquement lorsqu'on clique sur le bouton "Open" ou "Cancel".

Pour utiliser les classes évoquées dans ce paragraphe on utilisera :

```
import javax.swing.filechooser.FileFilter ;
// ou import javax.swing.filechooser.*;
import java.io.File ;
// ou import java.io.*;
import java.util.StringTokenizer ;
// ou import java.util.*;
```

18 Les graphiques

Le plus souvent, les graphiques sont réalisés dans un panneau `JPanel`.

Pour que le dessin soit permanent, il doit être réalisé à l'intérieur de la méthode `paintComponent` de `JPanel`, d'en tête :

```
void paintComponent(Graphics g)
```

qui est appelée périodiquement pour maintenir l'affichage du panneau.

La classe `Graphics` permet de définir un *contexte graphique*, mettant à notre disposition les outils pour dessiner, grâce aux méthodes `drawLine`, `drawRect`, `drawOval`, `drawString`

En pratique, on construit une classe dérivée de `JPanel` qui redéfinit la méthode `paintComponent` en incluant la réalisation du dessin, comme le montre le petit exemple suivant (on vérifiera que le dessin est permanent en masquant un instant le graphique par une fenêtre quelconque).

```
import javax.swing.* ;
import java.awt.*;

public class Fenetre extends JFrame {

    public Fenetre(){
        super("essai dessin");
        setBounds(10,10,200,200);
        JPanel p=new JPanelGr();
        getContentPane().add("Center",p);    }

    public static void main(String args[]){
        Fenetre f=new Fenetre();
        f.setVisible(true);    }

    class JPanelGr extends JPanel {
        public void paintComponent(Graphics g){
            super.paintComponent(g);
            g.drawRect(5,5,40,50);        }
    }
} // Fenetre
```

Certaines modifications du dessin qui ne sont pas réalisées au début risquent de ne pas être prises en compte immédiatement ; pour éviter cet inconvénient on peut faire appel à la méthode `repaint()` de `JPanel` dont le rôle est de forcer l'appel à `paintComponent`.

19 Les événements de bas niveau liés à la souris

Appui et relâchement d'un bouton : l'interface `MouseListener`

L'interface `MouseListener` possède 5 méthodes permettant de gérer les appuis et relâchements d'un bouton de la souris :

- `void mouseClicked(MouseEvent e)` : click de souris
- `void mousePressed(MouseEvent e)` : appui sur un bouton de la souris
- `void mouseReleased(MouseEvent e)` : relâchement du bouton
- `void mouseEntered(MouseEvent e)` : entrée du curseur sur le composant
- `void mouseExited(MouseEvent e)` : sortie du composant

En général un composant (panneau ou un bouton) implémente cette interface, ce qui signifie qu'il doit implémenter ces 5 méthodes (éventuellement certaines pourront être vides).

Plutôt que `MouseListener` on peut utiliser la classe abstraite `MouseAdapter` qui implémente ces 5 méthodes vides. L'avantage c'est que le panneau qui étend `MouseAdapter` ne peut implémenter que certaines de ces méthodes.

On écrira :

```
public class JPanel1 implements MouseListener { ....
```

ou bien

```
public class JPanel1 implements MouseAdapter { ....
```

Voici un exemple permettant de tester l'interface `MouseListener` :

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;

public class EssaiMouse extends JFrame implements MouseListener {
    public EssaiMouse () {
        super(" MouseListener");
        setSize(200,200);
        addMouseListener(this);
    }
    public void mouseClicked(MouseEvent e){
        System.out.println("click "+e.getX()+" "+e.getY());
    }
    public void mousePressed(MouseEvent e){
        System.out.println("Press "+e.getX()+" "+e.getY());
    }
    public void mouseReleased(MouseEvent e){
        System.out.println("Release "+e.getX()+" "+e.getY());
    }
    public void mouseEntered(MouseEvent e){
        System.out.println("Enter "+e.getX()+" "+e.getY());
    }
}
```

```

    public void mouseExited(MouseEvent e){
        System.out.println("Exit "+e.getX()+" "+e.getY());
    }
    public static void main(String a[]){
        EssaiMouse ess=new EssaiMouse();
        WindowListener l=new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        };
        ess.addWindowListener(l);
        ess.show();
    }
} // EssaiMouse

```

On notera :

- la déclaration `import java.awt.event.*`
- l'instruction `addMouseListener(this);`
- la gestion de la position de la souris grâce aux méthodes `getX()` et `getY()` de la classe `MouseEvent`.

Identification du bouton

La fonction `getModifiers()` de la classe `MouseEvent` retourne un entier permettant de récupérer l'information sur le bouton concerné. Les masques suivants facilitent cette opération :

- `InputEvent.BUTTON1_MASK`
- `InputEvent.BUTTON2_MASK`
- `InputEvent.BUTTON3_MASK`

Par exemple, la fonction `mouseClicked` pourra être modifiée comme suit pour afficher un message quand on clique sur le bouton 1 :

```

    public void mouseClicked(MouseEvent e){
        if ((e.getModifiers() & InputEvent.BUTTON1_MASK) !=0)
            System.out.println("Bouton 1");
    }

```

Déplacements de la souris : l'interface `MouseMotionListener`

Cette interface dont le fonctionnement est analogue à `MouseListener` contient deux méthodes :

- `void MouseMoved(MouseEvent e)` : déplacement de la souris sur le composant
- `void MouseDragged(MouseEvent e)` : déplacement de la souris sur le composant, bouton enfoncé

Naturellement, au lieu d'utiliser l'interface `MouseMotionListener` on peut utiliser la classe abstraite `MouseMotionAdapter` qui implémente les mêmes méthodes.

Annexe

1 Javadoc

Il est possible de générer automatiquement une documentation à partir des commentaires des fichiers sources.

Pour cela on place avant chaque méthode publique, champ ou classe un commentaire commençant par `/**` au lieu de `/*`, de la façon suivante :

```
/** * <commentaires sur la fonction>* @param <liste des paramètres >* @re-  
turn <valeur retournée>* @view <liste des fonctions en rapport avec cette  
fonction>*/
```

On lance alors javadoc en lui donnant comme paramètre le ou les fichiers sources, ce qui va avoir pour effet de générer des fichiers HTML semblables à ceux de la documentation de Sun.

(Pour plus de détails, consulter la doc "Tool documentation")

2 Fichiers jar

Les fichiers `.jar` sont des fichiers compactés analogues aux les fichiers `.zip`. Un utilitaire `jar` fourni avec `java` permet de les gérer.

Java a la possibilité d'exécuter les classes contenues dans des fichiers compactés `.jar` ou `.zip`.

Par exemple on crée, un fichier `bonjour.jar` comme suit :

```
# jar cvf bonjour.jar Bonjour.class
```

On pourra exécuter la classe `Bonjour.class` (supposée contenir une fonction `main` ne demandant pas de paramètre sur la ligne de commande), en tapant :

```
# java -classpath bonjour.jar Bonjour
```

Tout ceci fonctionne aussi bien avec un fichier `bonjour.zip`. Il est naturellement possible de placer toutes les classes dans un fichier `jar`.

Mais les fichiers `jar` offrent des possibilités supplémentaires par rapport aux fichiers `zip` : il est possible d'inclure dans le fichier `jar` des informations supplémentaires sur la classe principale. On procède ainsi :

1- Créer un fichier texte `xx.txt` contenant :

```
Manifest-Version: 1.0  
Main-Class: Bonjour
```

(`Bonjour` représente la classe principale. Attention : faire suivre le nom de la classe principale `Bonjour` uniquement d'un retour chariot, sans espace)

2- Créer le fichier `jar` en tapant

```
# jar cvfm bonjour.jar xx.txt Bonjour.class
```

3- Vérifier que le fichier `bonjour.jar` a bien été créé.

4- Exécuter le programme en tapant :

```
# java -jar bonjour.jar
```

Sous Windows, il suffit de double-cliquer sur l'icône du fichier pour l'exécuter.

Des indications supplémentaires pourront être trouvées en tapant `jar`, et dans la documentation de Sun.