SCIENCES SUP

Cours et exercices corrigés

IUT • BTS • Licence • Écoles d'ingénieurs

LA PROGRAMMATION OBJET EN JAVA

Michel Divay

DUNOD

LA PROGRAMMATION OBJET EN JAVA

Consultez nos catalogues sur le Web

www.dunod.com

LA PROGRAMMATION OBJET EN JAVA

Michel Divay
Professeur des Universités en informatique à Rennes 1

DUNOD

Illustration de couverture : *Contexture, digitalvision®*

ISBN 2 10 049697 2

Table des matières

Avant-propos

Java est un langage de programmation objet. Les principaux concepts de la programmation objet et du langage de programmation Java sont présentés dans ce livre à l'aide de nombreux exemples issus de domaines variés. Le livre n'essaie pas d'être exhaustif mais présente des notions et les illustre sur des exemples complets que le lecteur est encouragé à tester lui-même. Java définit de nombreuses fonctions (méthodes) comportant tout un éventail de paramètres. La documentation en ligne de Java est là pour détailler le rôle de chacun des paramètres et les variantes des fonctions. Le livre s'appuie également sur de nombreuses figures (environ 150) illustrant les concepts et les résultats attendus ou obtenus. Il est toujours difficile de présenter séquentiellement des notions qui en fait sont très imbriquées et interdépendantes comme le sont les concepts de programmation objet, d'héritage, de polymorphisme, d'interfaces graphiques, d'événements, de processus ou d'applets Java. De nombreux renvois tout au cours du livre évitent les répétitions fastidieuses.

Le **premier chapitre** présente la **syntaxe de base** du langage Java et fait le parallèle avec le langage C pour montrer les similitudes et les différences de syntaxe. On y passe rapidement en revue les notions de constantes, de variables, de tableaux, d'opérateurs, d'instructions de contrôles, de passages de paramètres et de récursivité.

Le **deuxième chapitre** aborde la **notion de classes et d'encapsulation des données**. On y définit les notions d'attributs et de méthodes, de variables d'instance et de variables de classe spécifiques de la programmation objet. La **notion de référence** d'objet et **d'allocation dynamique** des objets est illustrée par de nombreux schémas. La compréhension de cette notion est importante en Java car tous les objets sont repérés par des références. On peut grouper certaines classes dans des **paquetages** (*packages* ou bibliothèques) de façon à pouvoir les utiliser dans différentes applications. Les anomalies d'un programme peuvent donner naissance à des **exceptions** qui se propagent au fil des appels de fonctions jusqu'à ce qu'elles soient prises en compte au niveau décidé par le programmeur. Ce chapitre se termine par

une présentation des classes de gestion des chaînes de caractères et des classes de gestion d'ensembles d'éléments.

Le **chapitre 3** introduit la notion fondamentale en programmation objet **d'héritage** qui permet de définir une classe contenant les caractéristiques communes à plusieurs classes, celles-ci spécialisant la classe de base pour leurs besoins propres. En Java, une classe ne peut hériter que d'une seule classe. Néanmoins, un concept **d'interface** permet d'accéder à des objets de différentes classes suivant un autre critère. La notion d'héritage conduit à la notion de **polymorphisme** faisant qu'une même instruction peut en fait appeler lors de l'exécution des fonctions différentes de différents objets (mais ayant le même nom).

Le **chapitre 4** traite de la notion de liste permettant d'insérer ou d'extraire des éléments d'un ensemble. Ce chapitre illustre sur un exemple de structure de données très classique, les concepts d'héritage et de polymorphisme. Cette classe est très générale et peut être utilisée sans modification dans de nombreuses applications. La création d'une liste ordonnée nécessite un critère de comparaison de deux éléments. Ceci conduit à l'utilisation d'une interface Java de comparaison.

Les interfaces graphiques Java (**chapitre 5**) utilisent abondamment les concepts d'héritage. Une classe répertorie les caractéristiques communes aux composants graphiques (leur emplacement, leur taille, les couleurs d'arrière-plan et de premier plan, etc.). Puis chaque composant se spécialise éventuellement en plusieurs niveaux de sous-composants. Les composants peuvent être ajoutés à des composants conteneurs pour former des composants plus complexes. Les composants peuvent s'afficher dans des fenêtres et réagir à des événements les concernant provoqués à partir de la souris ou du clavier. Cette notion de groupement des composants en un composant plus complexe ayant ses propres propriétés est fondamentale en Java. Elle permet la création de composants réutilisables dans diverses applications ou dans des applets sur le Web. De nouveaux composants sont créés et réutilisés (sans modification) au fil des exemples.

Le **chapitre 6** présente l'interface graphique de la librairie Swing. Celle-ci reprend les concepts de l'interface graphique de **AWT** en y ajoutant de nombreuses possibilités qui permettent de développer des interfaces graphiques au look plus professionnel. La mise en œuvre de certains composants peut devenir toutefois assez complexe (gestion des listes, des arbres, des tables par exemple) si on veut utiliser toutes les possibilités de mise en œuvre.

Le **chapitre 7** est consacré à l'étude des flux d'information en entrée et en sortie. Il présente les deux grandes familles de flux en Java concernant les données binaires et les textes. Des exemples de flux sur des fichiers et sur des sites distants sont développés.

Un programme Java peut exécuter plusieurs animations en parallèle grâce au concept de processus (**thread**). Le programme se découpe en plusieurs programmes se partageant l'unité centrale à tour de rôle. Le **chapitre 8** présente cette notion et indique comment faire communiquer plusieurs threads entre eux pour les synchroniser ou pour échanger de l'information.

Java est très connu pour ses applets : programmes Java s'exécutant dans une page Web et se présentant souvent sous une forme graphique ou une animation. Le **chapitre 9** reprend certains des exemples développés dans les chapitres précédents et explique comment les convertir en applet.

Les classes développées sont présentées dans un paquetage ce qui permet de les réutiliser facilement. Des exercices corrigés, mais qui peuvent être mis au point de façon différente, sont proposés tout au cours de ce livre.

Les programmes Java de ce livre sont disponibles sur le site :

www.iut-lannion.fr/MD/MDLIVRES/LivreJava

Vous pouvez adresser vos remarques à l'adresse électronique suivante :

Michel.Divay@univ-rennes1.fr

D'avance merci.

Michel DIVAY

Chapitre **1**

Présentation du langage Java

1.1 INTRODUCTION GÉNÉRALE

Java est un langage objet permettant le développement d'applications complètes s'appuyant sur les structures de données classiques (tableaux, fichiers) et utilisant abondamment l'allocation dynamique de mémoire pour créer des objets en mémoire. La notion de structure, ensemble de données décrivant une entité (un objet en Java) est remplacée par la notion de classe au sens de la programmation objet. Le langage Java permet également la définition d'interfaces graphiques (GUI : Graphical User Interface) facilitant le développement d'applications interactives et permettant à l'utilisateur de "piloter" son programme dans un ordre non imposé par le logiciel.

Le langage est aussi très connu pour son interactivité sur le Web facilitant l'insertion dans des pages Web, au milieu d'images et de textes, de programmes interactifs appelés "applets"[1]. Pour des problèmes de sécurité, ces applets sont contrôlées et souvent limitées dans leur interaction avec le système d'exploitation de l'ordinateur sur lequel elles se déroulent : limitation des accès aux fichiers locaux ou aux appels système de la machine.

Un programme Java est portable au sens où il peut s'exécuter sur des ordinateurs fonctionnant avec différents systèmes d'exploitation. Les programmes écrits en Pascal ou en langage C sont aussi portables par compilation du code source sur la machine où le programme doit s'exécuter. Java est portable d'une plate-forme (matériel et système d'exploitation) à une autre sans recompilation. Le compilateur

1. Applet (nf) est l'acronyme de **App**lication **l**ight w**e**igh**t.**

produit un langage intermédiaire appelé "bytecode" qui est interprété sur les différentes machines. Il suffit donc de communiquer le bytecode et de disposer d'un interpréteur de bytecode pour obtenir l'exécution d'un programme Java. Les navigateurs du Web ont intégré un interpréteur de bytecode qui leur permet d'exécuter des programmes (applets) Java.

Cette portabilité est possible grâce à une normalisation indépendante de la plateforme : les entiers, par exemple, ont toujours la même longueur (en nombre d'octets) en Java quel que soit l'ordinateur, ce qui n'est pas le cas des langages C ou C++ (le type `int` peut occuper 2 ou 4 octets suivant les machines). Cette portabilité est souvent résumée de manière un peu commerciale par *write once, run anywhere* (écrivez une seule fois, exécutez n'importe où). Les programmes en Java ont l'extension `.java`, et le bytecode généré a l'extension `.class` (voir figures 1.1 et 1.2).

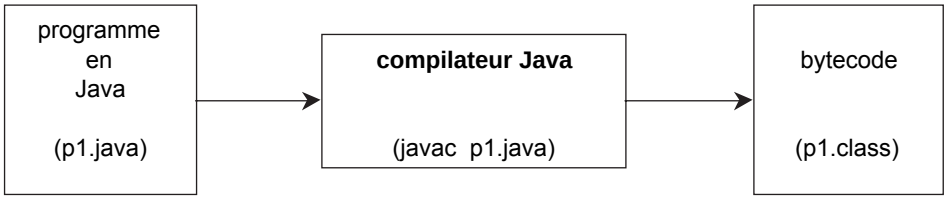


Figure 1.1 — La compilation du programme p1.java en son équivalent p1.class en bytecode.

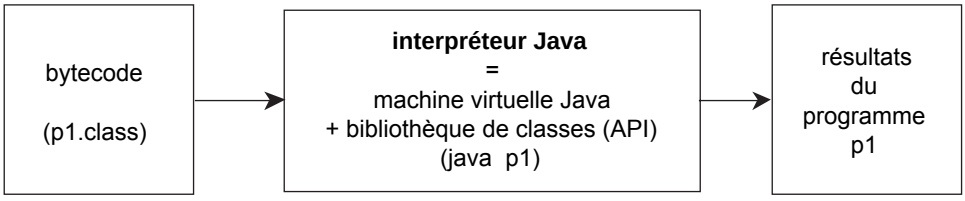


Figure 1.2 — L'interprétation du bytecode p1.class sur n'importe quel ordinateur disposant d'un interpréteur de bytecode (indépendamment du langage machine de cet ordinateur).

La souplesse de la portabilité a cependant une contrepartie. L'interprétation d'un programme est plus lente que l'exécution directe en langage machine qui résulterait d'une compilation classique. La richesse du langage Java tient aussi à sa bibliothèque de classes (API : Application Programming Interface) qui permet de traiter des applications très diverses (interfaces graphiques, chaînes de caractères, images, sons, mathématiques, etc.). Là également, il suffit de disposer du bytecode des bibliothèques pour pouvoir les utiliser sur n'importe quel ordinateur. Certaines classes peuvent cependant être spécifiques d'un système d'exploitation (gestion de l'heure, des droits d'accès aux fichiers, des processus, des interfaces graphiques s'appuyant ou non sur le système d'exploitation sous-jacent, etc.).

1.2 LA SYNTAXE DE JAVA

Java est un langage objet qui s'appuie sur la syntaxe du langage C et du C++. Java est un langage objet : on doit utiliser les classes et les objets ; C++ est un langage orienté objet, sorte de langage hybride permettant à la fois la programmation classique sans objets et la programmation avec objets. Ce chapitre présente la syntaxe des instructions de base de Java permettant de définir des variables de types dits "primitifs" et de contrôler le déroulement du programme. Les instructions sont présentées dans leur analogie avec C. Ce premier chapitre évite au maximum (mais pas entièrement) les concepts de classe et d'objet définis au chapitre 2.

La figure 1.3 présente un programme réalisant la somme des nb éléments d'un tableau tabEnt d'entiers.

```
// SomTab.java somme des nb éléments d'un tableau d'entiers

class SomTab {

 public static void main (String[] args) {
    int[] tabEnt = {1, 2, 3};// tableau de int nommé tabEnt
  //int tabEnt[] = {1, 2, 3};identique à l'instruction précédente
    int somme    = 0;
  //for (int i=0; i<3;             i++) somme += tabEnt[i];
    for (int i=0; i<tabEnt.length; i++) somme += tabEnt[i];
    System.out.println ("Somme : " + somme);
  }

}  // class SomTab
```

Figure 1.3 — La présentation de la syntaxe de base de Java.

Le résultat de l'exécution est le suivant :

```
Somme : 6
```

SomTab est une classe. Dans un premier temps, on peut considérer la classe SomTab comme un module comportant une fonction main. Les mots-clés `public` et `static` sont définis dans le chapitre 2. `String[] args` déclare un tableau de chaînes de caractères nommé `args` ; il contient les arguments de la ligne de commande lors du lancement du programme (voir la notion de tableau page 12 et la classe `String` page 80).

Les instructions suivantes déclarent et initialisent un tableau d'entiers. La deuxième ligne est en commentaires et constitue une variante de la ligne 1. La ligne 2 déclare et initialise avec la même syntaxe qu'en C ou C++, un tableau tabEnt de trois entiers.

```
int[] tabEnt = {1, 2, 3};
//int tabEnt[] = {1, 2, 3};
```

La syntaxe de la ligne 1 évoque mieux un tableau d'entiers (int[]). Les deux syntaxes sont acceptées en Java (voir tableau en page 12).

De même, les deux instructions suivantes (la première en commentaires) effectuent une boucle faisant la somme des entiers. En Java, la longueur du tableau (**tabEnt.length**) est connue et contrôlée. Il vaut donc mieux l'utiliser pour contrôler la boucle (ligne 2 du code suivant).

```
//for (int i=0; i < 3;                i++) somme += tabEnt[i];
  for (int i=0; i < tabEnt.length;   i++) somme += tabEnt[i];
```

L'écriture sur la sortie standard (l'écran par défaut) se fait par appel de la fonction (méthode) `println()` de l'objet `out` de la classe `System` (voir chapitre 7). "Somme :" est une chaîne de caractères à écrire suivie (c'est le sens du +, opération de concaténation de chaînes de caractères) de la valeur de l'entier somme (convertie automatiquement en chaîne de caractères) :

```
System.out.println ("Somme : " + somme);
```

Remarque : cette présentation sur un exemple fait bien apparaître l'analogie avec la syntaxe du C, et met en évidence quelques variantes qui sont développées dans les chapitres suivants.

1.2.1 Les conventions d'écriture des identificateurs

En Java, la convention adoptée (par les concepteurs de Java) mais non obligatoire consiste à écrire les noms des classes avec un identificateur commençant par une majuscule. Sur la figure 1.3, SomTab est une classe, de même que System (dans System.out.println). Par contre, les identificateurs de variables ou de fonctions commencent par une minuscule (main, tabEnt, somme, i, println). De plus, si l'identificateur consiste en la juxtaposition de mots, chaque mot interne commence par une majuscule : tabEnt (variable) pour table d'entiers ou SomTab (classe) pour somme du tableau. Les constantes symboliques sont écrites entièrement en majuscules (voir page 8 : GAUCHE, HAUT).

1.3 RÉSUMÉ DES INSTRUCTIONS JAVA DE BASE

L'exemple précédent (figure 1.3) a permis d'avoir une idée générale de la syntaxe de Java. Ce paragraphe décrit de façon concise et à l'aide d'exemples la syntaxe des instructions Java. Ce tour d'horizon présente dans l'ordre suivant : les types primitifs, les constantes, les variables, les expressions effectuant des opérations sur ces types primitifs et les tableaux (à une ou plusieurs dimensions) regroupant sous un même nom un ensemble de variables de même type.

1.3.1 Les commentaires

Comme en C, il existe deux types de commentaires. Les commentaires pour un ensemble de lignes délimités par /* au début et */ à la fin :

```
/* commentaires entre ces 2 marques,
   éventuellement sur plusieurs lignes */
```

et les commentaires pour fin de ligne (de // jusqu'à la fin de la ligne) :

```
i = 0; // commentaires jusqu'à la fin de la ligne
```

1.3.2 Les types de données primitifs (ou de base)

Les types primitifs (simples ou de base) correspondent à ceux du langage C auxquels s'ajoutent les types *byte* et *boolean*. Les chaînes de caractères sont gérées (différemment du C) sous forme de classes et présentées en 2.11 page 80. Un entier signé indique un nombre entier ayant un signe : + (par défaut) ou -. Le tableau suivant indique la liste des types d'entiers, de réels, de booléens ou de caractères disponibles et leurs limites dues à leur représentation en machine. En Java, un caractère est codé en format Unicode et occupe 2 octets. Les réels sont codés suivant la norme IEEE754 en un signe, une mantisse et un exposant. Le nombre d'octets occupés par chaque type de variable est normalisé et est le même sur tous les ordinateurs.

byte	un entier signé sur 8 bits	de –128 à +127
short	un entier signé sur 16 bits	de –32 768 à +32 767
int	un entier signé sur 32 bits	de –2 147 483 648 à 2 147 483 647
long	un entier signé sur 64 bits	de l'ordre de (±) 9 milliards de milliards
float	un réel sur 32 bits	de l'ordre de 1.4E-45 à 3.4E38
double	un réel sur 64 bits	de l'ordre de 4.9E-324 à 1.79E308
boolean	true ou false	
char	un caractère Unicode sur 16 bits	entier positif entre 0 et 65 535

Le type chaîne de caractères (type String) est défini comme une classe Java (voir page 12).

Exemples de déclaration de variables avec ou sans valeur initiale :

```
// TypesPrimitifs.java  les types primitifs Java

class TypesPrimitifs {

  public static void main (String[] args) {
    // entiers
    byte    b;                  // b : variable locale non initialisée
    short   s;
    int     i;
    long    l;
    byte    b1  = 50;          // b1 : variable locale initialisée à 50
    short   s1  = 5000;
    int     i1  = 50000;
    int     i2  = Integer.MIN_VALUE;              // le plus petit int
    int     i3  = Integer.MAX_VALUE;              // le plus grand int
    int     i4  = 0x1F;             // constante entière en hexadécimal
    int     i5  = 012;                    // constante entière en octal
    long    l1  = 10000000;
```

```
// réels
float   f;
float   f1  = 2.5f;                    // ajout de f pour type float
float   f2  = 2.5e3f;                   // 2.5 x 10 à la puissance 3
float   f3  = Float.MIN_VALUE;                // le plus petit float
double  d;
double  d1  = 2.5;     // pourrait s'écrire 2.5d (défault : double)
double  d2  = 2.5e3;                                    // ou 2.5e3d

// caractères
char    c;
char    c1  = 'é';
char    c2  = '\n';                  // passage à la ligne suivante
char    c3  = '\u00e9';                   // caractère Unicode du é
char    c4  = '\351';          // caractère é en octal : e9₁₆ = 351₈

// booléens
boolean bo;
boolean bo1 = true;
boolean bo2 = false;
```

Java vérifie qu'une variable locale (à la fonction main() dans cet exemple) est initialisée avant son utilisation. Si la variable n'est pas initialisée, le compilateur signale un message d'erreur comme sur les exemples ci-dessous.

```
// Les instructions suivantes donnent un message d'erreur :
// variable locale utilisée avant initialisation
//System.out.println ("b   : " + b);
//System.out.println ("s   : " + s);
//System.out.println ("i   : " + i);
//System.out.println ("l   : " + l);
//System.out.println ("c   : " + c);
//System.out.println ("bo : " + bo);
```

L'instruction suivante écrit en début de ligne (indiquée par \n), pour chaque variable, son nom suivi de sa valeur convertie en caractères. L'opérateur + est ici un opérateur de concaténation de chaînes de caractères.

```
// Écriture d'entiers : byte, short, int, long
System.out.println (
  "\nb1  : " + b1 +
  "\ns1  : " + s1 +
  "\ni1  : " + i1 +
  "\ni2  : " + i2 +
  "\ni3  : " + i3 +
  "\ni4  : " + i4 +
  "\ni5  : " + i5 +
  "\nl1  : " + l1
);

// Ecriture de réels (flottants) : float, double
System.out.println (
  "\nf1  : " + f1 +
```

```
            "\nf2  : " + f2 +
            "\nf3  : " + f3 +
            "\nd1  : " + d1 +
            "\nd2  : " + d2
        );

        // Ecriture de caractères
        System.out.println ("c1  : " + c1);
        System.out.println ("c3  : " + c3);
        System.out.println ("c4  : " + c4);

        // Ecriture de booléens
        System.out.println ("bo1 : " + bo1);
        System.out.println ("bo2 : " + bo2);
```

Les instructions suivantes provoquent une erreur de compilation : les valeurs sont en dehors des plages de valeurs autorisées pour un type donné.

```
        //byte b2 = 128;                              // byte de -128 à +127
        //byte b3 = -129;
        //short n1 = -32769;                  // short de -32 768 à +32 767
        //short n2 = 50000;
    }  // main

}  // class TypesPrimitifs
```

Exemple de résultats d'exécution de la classe TypesPrimitifs :

```
b1  : 50
s1  : 5 000
i1  : 50 000
i2  : -2 147 483 648                                  le plus petit int
i3  : 2 147 483 647                                   le plus grand int
i4  : 31                                                     1F en hexa
i5  : 10                                                   12 en octal
l1  : 10 000 000
f1  : 2.5
f2  : 2500.0
f3  : 1.4E-45                                        le plus petit float
d1  : 2.5
d2  : 2500.0
c1  : é
c3  : é                                  caractère Unicode 00e9 en hexa
c4  : é                                                  351 en octal
bo1 : true
bo2 : false
```

Remarque : pour chaque type, des constantes indiquant les maximums et les minimums sont définies : `Byte.MIN_VALUE`, `Byte.MAX_VALUE`, `Short.MIN_VALUE`, etc.

Exercice 1.1 – Programme d'écriture des valeurs limites des types primitifs

> Écrire le programme complet donnant les limites minimums et maximums de chacun des types entiers et réels.

1.3.3 Les constantes symboliques

Les constantes symboliques sont déclarées comme des variables précédées de *fina* . Leur contenu ne peut pas être modifié par la suite. L'identificateur de la constante est souvent écrit en majuscules (voir page 4). Exemples de déclaration de constantes symboliques :

```
final int GAUCHE = 1;
final int HAUT   = 2;
final int DROITE = 3;
final int BAS    = 4;

final double PI  = 3.1415926535;
```

1.3.4 Les opérateurs arithmétiques, relationnels, logiques

Les **opérateurs arithmétiques** réalisent des opérations sur des variables entières ou réelles. Les opérations possibles sont listées ci-après.

```
+         addition            n = a + b;
-         soustraction        n = a - b;
*         multiplication      n = a * b;
/         division            n = a / b;
+=        addition            n += b;                  à n, ajouter b
-=        soustraction        n -= b;                à à n, soustraire b
*=        multiplication      n *= b;               multiplier n par b
/=        divisionr           n /= b;                  diviser n par b
```

Le modulo fournit le reste de la division entière de deux nombres :

```
%         modulo              n = a % b; reste de la division de a par b
%=        modulo              n %= b; reste de la division de n par b
```

Les **opérateurs d'incrémentation** de variables (entières ou réelles) sont des raccourcis ajoutant ou retranchant la valeur 1 à une variable. La valeur est prise en compte avant l'incrémentation dans le cas où l'opérateur suit la variable (j++ : prendre la valeur de j, puis l'incrémenter) ; elle est prise en compte après l'incrémentation dans le cas contraire (++j : incrémenter j, puis prendre sa valeur).

```
++        incrémente la variable                          j++; ou ++j;
--        décrémente la variable                          j++; ou ++j;
```

Les **opérateurs (r elationnels)** de **comparaison** de variables ou de constantes (d'expressions arithmétiques en général) délivrent un résultat booléen.

```
>         supérieur           a > 0        (a est-il supérieur à 0 ?)
<         inférieur
<=        inférieur ou égal
>=        supérieur ou égal   n >= 1      (n supérieur ou égal à 1 ?)
==        égal
!=        non égal (différent) a != 0            (a différent de 0 ?)
```

Les **opérateurs logiques** définissent des expressions fournissant un résultat booléen. Ci-après, c, n et `nbCouleurs` sont des entiers ; "trouve" est un booléen.

```
&&        et logique          (c >= 0) && (c <= 15)
||        ou logique          (n < 0 || n >= nbCouleurs)
!         négation            (!trouve)
```

***Remarque* :** il existe aussi en Java un opérateur "&" qui demande toujours l'évaluation de ses deux opérandes alors que l'opérateur "&&" n'évalue pas le deuxième opérande si le premier est faux (précédemment, si c vaut -1, le premier test (c >= 0) fournit "faux" ; avec l'opérateur &&, le deuxième test (c <= 15) n'est pas évalué). Il existe de même un opérateur | qui évalue dans tous les cas ses deux opérandes. Pour l'opérateur ||, si le premier test est vrai, il est inutile d'évaluer le second (si n vaut -1, n < 0 est vrai ; il est inutile de tester si n >= nbCouleurs sur l'exemple précédent).

Les **opérateurs de décalage** de bits à gauche ou à droite réalisent des multiplications ou divisions rapides quand le quotient est une puissance de 2 :

```
<<        décalage à gauche    A << 2      (revient à multiplier par 4)
>>        décalage à droite    A >> 2         (revient à diviser par 4)
```

***Remarque* :** l'**opérateur = de l'affectation** peut être considéré comme délivrant un résultat qui est celui de l'affectation. Cela justifie l'écriture suivante qui peut être interprétée comme suit : c reçoit la valeur de lireChar() ; le résultat de l'affectation est comparé à '\n'.

```
if ( (c = lireChar()) == '\n')…
```

1.3.5 Les problèmes de dépassement de capacité (à l'exécution)

Chaque type primitif a une plage de valeurs autorisées et lorsqu'il s'agit de constantes, le compilateur vérifie la validité des affectations (voir page 5). Cependant, l'évaluation d'expressions arithmétiques peut produire à l'exécution des valeurs qui dépassent les plages de valeurs autorisées. Java ne signale aucun message d'erreur ; les résultats sont alors tout simplement faux. Exemple :

```
// Debordement.java         débordement lors d'opérations arithmétiques

class Debordement {

  public static void main (String[] args) {
    int i1 = 1000000000;  // 1 milliard
    int i2 = 2 * i1;
    int i3 = 3 * i1;
    System.out.println ("i1        : " + i1);
    System.out.println ("i2 = 2*i1 : " + i2);
    System.out.println ("i3 = 3*i1 : " + i3);      // débordement de i3

    int i4 = Integer.MAX_VALUE;
```

```
        int i5 = i4 + 1;
        System.out.println ("i4        : " + i4);
        System.out.println ("i5        : " + i5);      // débordement de i5
      }  // main

}  // class Debordement
```

Résultats d'exécution :

```
i1        : 1000000000
i2 = 2*i1 : 2000000000
i3 = 3*i1 : -1294967296         débordement ; i3 > Integer.MAX_VALUE
i4        : 2 147 483 647
i5        : -2147483648         débordement ; i5 > Integer.MAX_VALUE
```

Remarque : on voit d'après les résultats précédents, que le suivant du plus grand entier (Integer.MAX_VALUE), c'est le plus petit entier négatif !

1.3.6 Le transtypage (cast)

Le transtypage est nécessaire quand il risque d'y avoir perte d'information, comme lors de l'affectation d'un entier `long` (64 bits) à un entier `int` (32 bits), ou d'un réel `double` vers un réel `float`. On force alors le type, indiquant ainsi au compilateur qu'on est conscient du problème de risque de perte d'information. C'est aussi le cas lorsqu'on veut prendre la partie entière d'un réel. Les affectations suivantes sont **implicitement** autorisées entre entiers et/ou réels ; on peut affecter un `byte` à un `short`, ou un `float` à un `double` :

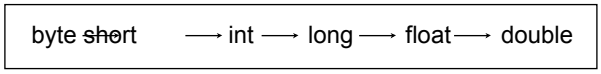


Figure 1.4 — Les conversions implicites autorisées en Java.

Quand les deux opérandes d'un opérateur ne sont pas du même type, celui qui est de type inférieur est converti dans le type de l'autre. Pour la variable l4 ci-après, on multiplie i3 (qui vaut 10) par la valeur réelle de 2.5f convertie en un entier long. Pour l5, i3 est converti en flottant ; le résultat en flottant est converti en entier long. Les résultats sont différents.

```
int  i3 = 10;
long l4 = i3 * (long) 2.5f;    // résultat : 20
long l5 = (long) (i3 * 2.5f);  // résultat : 25
```

Remarque : dans les expressions arithmétiques, les entiers de type `byte` ou `short` sont considérés comme des `int` (convertis d'office en `int`). Si b1 est un `byte`, b1 + 1 est considéré être un `int`. L'affectation b1 = b1 + 1 est illégale sans cast (de type `int -> byte`) alors que b1 = 5 est acceptée.

Exemple :

```
// TransTypage.java  les casts (conversions forcées)

class TransTypage {

  public static void main (String[] args) {
    byte   b1 = 15;
    int    i1 = 100;
    long   l1 = 2000;
    float  f1 = 2.5f;        // f pour float; par défaut de type double
    double d1 = 2.5;                                          // ou 2.5d

  //b1 = b1 + 1;        // cast nécessaire : b1 + 1 est converti en int
    b1 = (byte) (b1 + 1);                                  // OK avec cast
  //byte b2 = i1;                  // cast nécessaire : de int -> byte
    byte b2 = (byte) i1;                                   // OK avec cast
    System.out.println ("b2 : " + b2);

    long l2 = i1;                       // OK sans cast : de int -> long
    System.out.println ("l2 : " + l2);

  //int i2 = l1;                    // cast nécessaire : de long -> int
    int i2 = (int) l1;                                     // OK avec cast
    System.out.println ("i2 : " + i2);

    float f2 = l1;                   // OK sans cast : de long -> float
    System.out.println ("f2 : " + f2);

  //long l3 = f1;                 // cast nécessaire : de float -> long
    long l3 = (long) f1;                                   // OK avec cast
    System.out.println ("l3 : " + l3);

    double d2 = f1;                // OK sans cast : de float -> double
    System.out.println ("d2 : " + d2);

  //float f3 = d1;              // cast nécessaire : de double -> float
    float f3 = (float) d1;                                 // OK avec cast
    System.out.println ("f3 : " + f3);

    int  i3 = 10;
    long l4 = i3 * (long) 2.5f;                              // l4 vaut 20
    long l5 = (long) (i3 * 2.5f);                            // l5 vaut 25
    System.out.println ("\nl4 : " + l4 + "\nl5 : " + l5);

    // dans une expression arithmétique,
    // byte, short sont considérés comme des int
    byte b3 = 10;
  //short s4 = 2*b3;          // erreur : b3 est considéré comme un int
    short s4 = (short) (2*b3);     // cast nécessaire : de int -> short
    System.out.println ("s4 : " + s4);

    int  i4 = 10;
    long l6 = 1000;
  //int i5 = i4 + l6;          // erreur : le résultat est de type long
    int i5 = (int) (i4 + l6);       // cast nécessaire : de long -> int
```

```
        System.out.println ("i5 : " + i5);
      }  // main
    }  // class TransTypage
```

Résultats d'exécution de TransTypage :

```
b2 : 100
l2 : 100
i2 : 2000
f2 : 2000.0
l3 : 2
d2 : 2.5
f3 : 2.5
l4 : 20
l5 : 25
s4 : 20
i5 : 1010
```

1.3.7 Les chaînes de caractères : class String

En Java, les chaînes de caractères ne sont pas des types primitifs. Ce ne sont pas des tableaux de caractères terminés par un zéro de fin de chaîne comme en C. Les chaînes sont des objets de la classe *String* (voir page 80).

1.3.8 Les tableaux à une dimension

1.3.8.1 Les tableaux d'éléments de type primitif

Plusieurs éléments de même type peuvent être regroupés (sous un même nom) en tableau. On peut accéder à chaque élément à l'aide d'un d'indice précisant le rang de l'élément dans le tableau. Le nombre d'éléments maximum est contrôlé en Java. Il ne peut y avoir de débordement de tableaux. Un message d'erreur est délivré (sous forme d'exceptions : voir 74) si l'indice permettant d'accéder à un élément est en dehors de la plage autorisée. Le premier élément porte l'indice **0**. Les tableaux sont alloués dynamiquement en Java. Ils sont initialisés par défaut à 0 pour les nombres et à faux pour les tableaux de booléens.

Exemples de déclarations : tableau de types primitifs (type int) avec allocation et initialisation du tableau :

```
int tabEnt[] = {1, 2, 3};                  // (1) identique au langage C
int[] tabEnt = {1, 2, 3};          // (2) indique mieux un tableau d'int
```

(1) et (2) sont identiques et corrects en Java. La forme (2) indique mieux un tableau d'entiers nommé tabEnt, et initialisé avec les valeurs 1, 2 et 3.

Figure 1.5 — La déclaration int[] tabEnt.
tabEnt est une référence non initialisée vers un tableau d'entiers.

Exemples de déclarations avec allocation dynamique du tableau :

```
int[] tabEnt;     // référence (non initialisée) vers un tableau de int
```

déclare et définit la variable tabEnt comme une référence sur un tableau d'entiers, mais ne réserve pas de place mémoire pour ce tableau. L'allocation doit se faire avec `new` en indiquant le nombre d'éléments. Ci-après, on réserve 10 éléments de type `int`, numérotés de 0 à 9.

```
tabEnt = new int [10];    // allocation dynamique de 10 int pour tabEnt
```

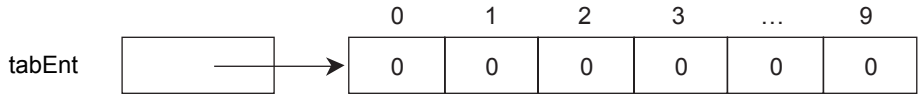


Figure 1.6 — Après exécution de tabEnt = new int [10].
tabEnt est une référence sur un tableau d'entiers initialisés à 0.

Ou encore en une seule instruction :

```
int[] tabEnt = new int [10];            // tableau d'int de 10 éléments
```

`tabEnt.length` fournit la longueur déclarée du tableau tabEnt (soit 10 sur l'exemple).

Exemple de programme Java utilisant les tableaux à une dimension :

```
// Tableau1D.java  tableau à une dimension

class Tableau1D {

  public static void main (String[] args) {
    // allocation d'un tableau tabEnt de 3 entiers
    // d'indice de 0 à 2
    int[] tabEnt = {1, 2, 3};
    // tabEnt.length = longueur de tabEnt
    for (int i=0; i < tabEnt.length; i++) {
      System.out.print (tabEnt[i] + " ");
    }
    System.out.println();                                  // à la ligne

    // allocation d'un tableau valEnt de 5 valeurs entières
    // d'indice de 0 à 4
    int[] valEnt = new int [5];        // initialisation à 0 par défaut
    valEnt [2] = 20;
    valEnt [4] = 50;
    //valEnt [5] = 80;    // message d'erreur à l'exécution (Exception)
    // valEnt.length = longueur de valEnt
    for (int i=0; i < valEnt.length; i++) {
      System.out.print (valEnt[i] + " ");
    }

    boolean[] reponse = new boolean [3];          // initialisé à false
    System.out.print ("\n\n");
```

```
    for (int i=0; i < reponse.length; i++) {
      System.out.print (reponse[i] + " ");
    }
  }  // main

}  // class Tableau1D
```

Résultats d'exécution :

```
1 2 3
0 0 20 0 50                          tableau initialisé à 0 par défaut
false false false                  tableau initialisé à false par défaut
```

1.3.8.2 Les tableaux d'objets

On déclare de la même façon un tableau d'objets (voir figure 2.17, page 61).

```
Balle[] tabBalle = new Balle [6];
```

tabBalle est un tableau d'éléments de type Balle pouvant référencer six objets de type Balle numérotés de 0 à 5. La déclaration du tableau ne crée pas les objets référencés. Les références sur les objets sont nulles. Les références des objets créés doivent être affectées au tableau par la suite.

Les chaînes de caractères étant gérées en Java sous forme de classes, on peut de même déclarer des tableaux d'objets de type String (chaînes de caractères).

```
String[] prenoms = new String [2];     // prenoms : tableau de 2 String
prenoms[0] = "Michel";       // prenoms[0] référence la chaîne "Michel"
prenoms[1] = "Josette";
```

`prenoms` est un tableau de `String` pouvant référencer deux `String` (chaînes de caractères). La déclaration et l'initialisation d'un tableau de chaînes de caractères peuvent se faire sous une forme équivalente plus compacte comme indiqué ci-après.

```
String[] prenoms = {"Michel", "Josette"};
```

1.3.9 Les tableaux à plusieurs dimensions

On peut généraliser la notion de tableau à une dimension pour créer des tableaux multidimensionnels. Un tableau à deux dimensions est un tableau de tableaux. Le

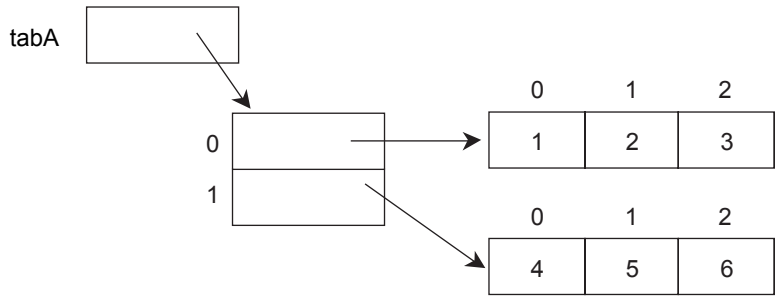


Figure 1.7 — Un tableau d'entiers à deux dimensions de deux lignes et trois colonnes.

nombre d'éléments dans chaque ligne du tableau à deux dimensions peut même varier en Java comme c'est le cas pour le tableau tabC du programme Java suivant (voir figure 1.8).

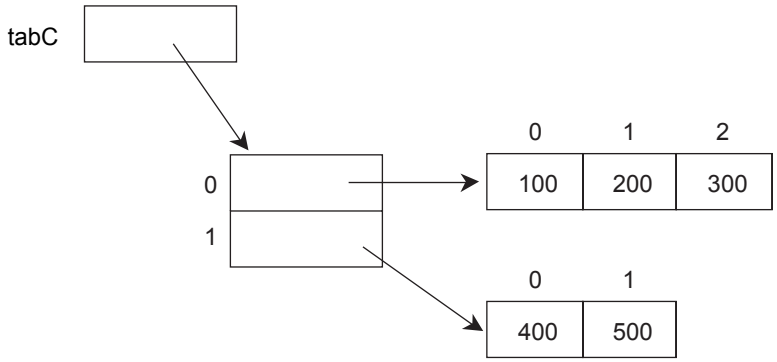


Figure 1.8 — Un tableau à deux dimensions de deux lignes et d'un nombre variable de colonnes.

Exemple de programme Java utilisant des tableaux à deux dimensions :

```
// Tableau2D.java  tableau à 2 dimensions

class Tableau2D {

  // liste les éléments du tableau à 2 dimensions
  static void listerTableau (int[][] tableau, String nom) {
    System.out.println ("\nlisterTableau  " + nom + ".length : "
                        + tableau.length + " lignes");
    for (int i=0; i < tableau.length; i++) {
      for (int j=0; j < tableau[i].length; j++) {
        System.out.print (tableau[i][j] + " "); // écrire tableau[i][j]
      }
      System.out.println();                              // à la ligne
    }
  }

  public static void main (String[] args) {
    int[][] tabA = {                          // 2 lignes et 2 colonnes
      {1, 2, 3},
      {4, 5, 6}
    };
    listerTableau (tabA, "tabA");

    int[][] tabB = new int [3][2];            // 3 lignes et 2 colonnes
    tabB[0][0] = 10;  tabB[0][1] = 20;
    tabB[1][0] = 30;  tabB[1][1] = 40;
    tabB[2][0] = 50;  tabB[2][1] = 60;
    listerTableau (tabB, "tabB");

    // nombre variable d'éléments par ligne
```

```
        // 1e ligne 3 éléments; 2e ligne 2 éléments
        int[][] tabC = {
          {100, 200, 300},
          {400, 500}
        };
        listerTableau (tabC, "tabC");
        System.out.println ("1e ligne : " + tabC[0].length + " colonnes");
        System.out.println ("2e ligne : " + tabC[1].length + " colonnes");
      }  // main

}  // class Tableau2D
```

Résultats d'exécution :

```
listerTableau  tabA.length : 2 lignes
1 2 3
4 5 6
listerTableau  tabB.length : 3 lignes
10 20
30 40
50 60
listerTableau  tabC.length : 2 lignes
100 200 300
400 500
1e ligne : 3 colonnes              3 éléments sur la première ligne
2e ligne : 2 colonnes              2 éléments sur la deuxième ligne
```

Remarque : de même que pour les tableaux à une dimension, on peut déclarer des tableaux à deux dimensions d'objets. L'instruction suivante déclare un tableau à deux dimensions d'objets de type `Balle` nommé `proposition`, et pouvant référencer 10 fois 4 éléments de type `Balle` ; les références vers les objets sont nulles.

```
Balle[][] proposition = new Balle [10][4];
```

1.3.10 Les instructions de contrôle (alternatives, boucles)

Les instructions de contrôle du déroulement lors de l'exécution des instructions ont la même syntaxe que pour le langage C. Leur syntaxe est présentée succinctement ci-dessous. L'espace n'étant pas significatif en Java, les instructions peuvent être présentées (indentées) de différentes façons.

1.3.10.1 Les alternatives

alternative simple :

```
if (...) {                      // le test est une expression booléenne
  ...              // utilisant éventuellement &&, || ou ! (voir page 9)
}
```

L'alternative peut être présentée sur une seule ligne si une seule instruction suit le test :

```
if (...) ...;
```

Exemple :

```
int lg;                                          // numMois de type int
if (numMois == 1) {                            // si numMois vaut 1 alors
  lg = 31;
  System.out.println ("janvier");
}

if (numMois == 1) lg = 31;                       // une seule instruction
```

alternative double :

```
if (...) {
  ...
} else {
  ...
}
```

Exemple :

```
if (somme > 1000) {                                 // somme de type int
  System.out.println ("Somme > 1000");
} else {
  System.out.println ("Somme <= 1000");
}
```

opérateur conditionnel : < condition > **?** valeur1 **:** valeur2 **;**

Si l'affectation d'une variable dépend d'une condition, l'instruction peut s'écrire d'une manière concise sous la forme donnée par les exemples suivants : affecter à la variable n, si trouve est vrai, la valeur i, sinon la valeur -1.

```
int n = trouve ? i : -1;
```

ou encore si une fonction retourne un résultat : retourner si trouve est vrai, la valeur i, sinon la valeur -1.

```
return trouve ? i : -1;
```

choix multiple :

```
switch (exp) {     // exp : expression de type byte, short, int ou char
case ...:
  ...
  break;
case ...:
  ...
  break;
default :                              // le cas default est facultatif
  ....
  // break;                          // optionnel si c'est le dernier cas
} // switch
```

Exemple 1 avec une expression entière :

```
// SwitchJava1.java                  test du switch java avec un entier

class SwitchJava1 {

  public static void main (String[] args) {
    int lgMois;

    int numMois = 3;        // changer la valeur pour le test du switch
  //int numMois = 13;                    // numéro de mois 13 incorrect

    switch (numMois) {                       // numMois est de type int
    case 1 :  case 3 :  case 5 :  case 7 :
    case 8 :  case 10 : case 12 :
      lgMois = 31;
      break;
    case 4 :  case 6 :  case 9 : case 11 :
      lgMois = 30;
      break;
    case 2 :
      lgMois = 28;
      break;
    default :
      lgMois = 0;
      System.out.println ("Numéro de mois " + numMois + " incorrect");
      // break;
    } // switch

    System.out.println ("lgMois : " + lgMois);
  }  // main

}  // class SwitchJava1
```

Exemple 2 avec une expression de type caractère :

```
// SwitchJava2.java               test du switch java avec un caractère

class SwitchJava2 {

  public static void main (String[] args) {
    boolean fini = false;
    char    car;

    car = 'O';              // changer la valeur pour le test du switch

    switch (car) {                              // car est de type char
    case 'O':  case 'o':
      fini = true;
      break;
    case 'N':  case 'n':
      fini = false;
      break;
    default :
      System.out.println ("Répondre par (O/N) ");
```

```
      } // switch

      System.out.println ("fini : " + fini);
    }  // main

}  // class SwitchJava2
```

1.3.10.2 Les boucles for, while et do ... while

boucle for :

```
for (exp1; exp2; exp3) {
  ...
}
```

- exp1 : évaluée avant de commencer la boucle.
- exp2 : condition évaluée en début d'itération ; la boucle continue si exp2 est vraie.
- exp3 : évaluée en fin de chaque itération.

Exemple :

pour i partant de 0 et
tant que i est inférieur à la longueur du tableau tab
ajouter tab[i] à somme,
et incrémenter i (en fin de chaque tour de boucle)

s'écrit comme suit sur une ligne (une seule instruction dans la boucle) :

```
for (int i=0; i < tab.length; i++) somme += tab[i];
```

ou encore surtout s'il y a plusieurs instructions dans la boucle :

```
for (int i=0; i < tab.length; i++) {
  somme += tab[i];
}
```

boucle while :

```
// tant que la condition est vraie, on exécute la boucle
while (<condition >) {
  ...
}
```

Exemple :

```
while (!fini) {        // fini est un booléen (tant que non fini faire)
  ...             // à un moment, le corps de la boucle met fini à vrai
}
```

boucle do ... while :

```
// exécuter la boucle tant que la condition de fin de boucle est vraie
do {
  ...                        // la boucle est exécutée au moins une fois
} while (<condition >);          // condition de poursuite de la boucle
```

1.3.11 Le passage des paramètres des fonctions (ou méthodes)

Les paramètres d'une fonction sont empilés dans la pile du processus lors de l'entrée dans la fonction et dépilés lors de la sortie de cette fonction. On peut passer un paramètre à une fonction de deux manières : en recopiant sa valeur ou son adresse dans la pile. Le **passage par valeur** ne permet pas de modifier la valeur de la variable du programme appelant puisqu'on travaille avec une copie.

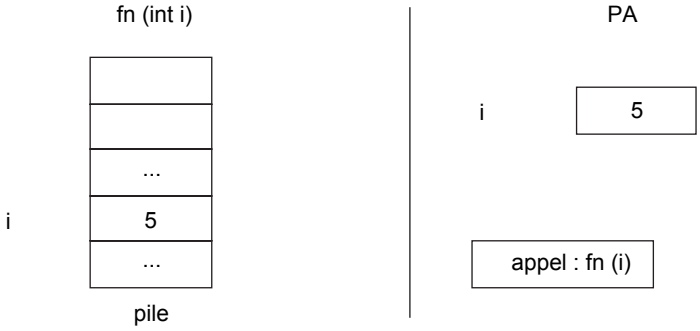


Figure 1.9 — Le passage par valeur de la variable i du PA (programme appelant) à une fonction.
La valeur i en paramètre de la fonction est recopiée dans la pile lors de l'appel dans PA de fn(i). La fonction utilise cette copie. Si la fonction modifie i (i = 2 par exemple), cette modification se fait dans la pile. La variable i du PA n'est pas modifiée au retour de la fonction.

Le **passage par adresse** permet de modifier la variable du programme appelant puisqu'on connaît son adresse. Dans le passage par adresse ci-après, les noms des tableaux dans le programme appelant et dans la fonction diffèrent légèrement (coef et coeff) pour éviter toute confusion. Ils pourraient être identiques ou carrément différents, cela ne change rien au principe du passage par adresse.

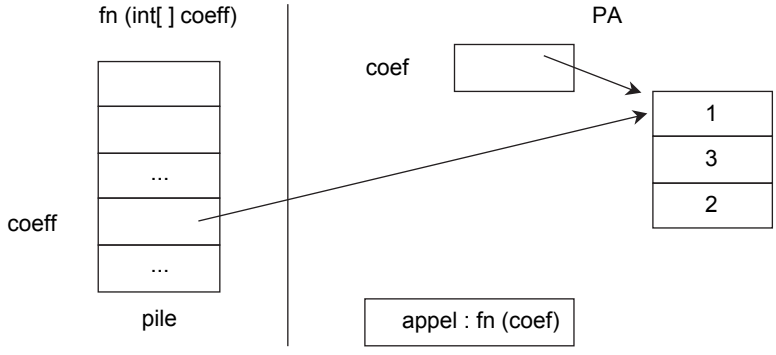


Figure 1.10 — Le passage par adresse du tableau :
on recopie dans la pile, l'adresse (la référence) du tableau coef.
La modification d'un élément du tableau coeff dans la fonction (coeff[1]=4) se répercute dans le programme appelant. Voir le programme Java ci-dessous.

1.3.11.1 Les tableaux (passage des paramètres par adresse)

Pour les **tableaux**, le passage des paramètres s'effectue par **adresse** (par référence). On peut accéder et modifier les valeurs du tableau passé en paramètre comme le montre le programme Java suivant.

```
// TableauParam.java  passage des paramètres
//                    de type tableau par reference (adresse)

class TableauParam {

  // liste les 2 tableaux note et coef
  static void listerTableau (float[] note, int[] coef) {
    System.out.println ("\nNote\tCoefficient");
    for (int i=0; i < note.length; i++) {
      System.out.println (note[i] + "\t\t" + coef[i]);
    }
  }

  // modifie le coefficient num du tableau coeff
  static void modifCoeff (int[] coeff, int num, int valeur) {
    coeff [num] = valeur;
  }

  public static void main (String[] args) {
    int[]   coef = {1, 3, 2};
    float[] note = {10.75f, 12f, 8.25f};

    listerTableau (note, coef);
    modifCoeff    (coef, 1, 4);              // le coefficient 1 vaut 4
    listerTableau (note, coef);
  }  // main

}  // class TableauParam
```

Résultats d'exécution :

```
Note          Coefficient
10.75              1
12.0               3
8.25               2
Note          Coefficient
10.75              1
12.0               4            le coefficient 1 a bien été modifié
8.25               2
```

1.3.11.2 Les types primitifs (passage des paramètres par valeur)

Par contre, les variables de type primitif (byte, short, int, long, float, double, char, boolean) sont passées par **valeur**. Une modification de cette variable dans une fonction est locale à la fonction et n'est pas répercutée dans le programme appelant. Exemple (**erroné**) de tentative de récupération d'un résultat par passage en paramètre d'un type primitif :

```
// SomMoyFaux.java  passage des paramètres par valeur
//                  pour les types primitifs

public class SomMoyFaux {

  // somme et moyenne des éléments du tableau tab
  static int somme (int[] tab, float moyenne) {       // RESULTATS FAUX
    int somme = 0;
    int n     = tab.length;

    for (int i=0; i < n; i++) somme += tab[i];
    moyenne = somme/n;
    System.out.println ("Dans la fonction somme  moyenne = "
                                                          + moyenne);
    return somme;
  }

  public static void main (String[] args) {
    int   somVect;
    float moy = 0;
    int[] vecteur = {10, 15, 20};

    somVect = somme (vecteur, moy);            // moy passée par valeur

    System.out.println ("Dans main  somVect = " + somVect
                                                  + ", moy = " + moy);
  } // main

}  // class SomMoyFaux
```

Résultats d'exécution :

```
Dans la fonction somme  moyenne = 15.0
Dans main  somVect = 45, moy = 0.0
                        faux ; la variable moy n'a pas été modifiée
```

1.3.11.3 Les objets (passage des paramètres par adresse)

Puisque les types primitifs sont passés par valeur et ne permettent pas de récupérer un résultat, on peut définir le type primitif dans une structure (une classe en Java, voir chapitre 2), et passer un objet de cette classe en paramètre qui lui est passé par adresse (ou référence).

```
// SomMoyOK.java  somme et moyenne
//                passage dans un objet de la moyenne

// la classe Reel s'apparente
// à un type de structure contenant une valeur réelle
class Reel {
  float valeur;
}

class SomMoyOK {

  // moyenne est un objet de la classe Reel
```

```
  // (structure contenant la variable valeur)
  static int somme (int[] tab, Reel moyenne) {
    int somTab = 0;
    int n      = tab.length;

    for (int i=0; i < n; i++) somTab += tab[i];
    // modifie le champ (l'attribut) valeur de l'objet moyenne
    moyenne.valeur = somTab/n;
    return somTab;
  }

  public static void main (String[] args) {
    int   somVect;
    Reel  moy = new Reel();                // allocation de l'objet moy
    int[] vecteur = {10, 15, 20};

    // l'objet moy est passé par adresse (référence)
    somVect = somme (vecteur, moy);
    System.out.println ("Somme " + somVect +
                                          "\nMoyenne " + moy.valeur);
  }

}  // class SomMoyOK
```

Résultats d'exécution :

```
Somme 45
Moyenne 15.0                    moy.valeur contient bien la moyenne
```

1.4 LA RÉCURSIVITÉ

Le langage Java comme C ou C++ gère la récursivité des fonctions. Une fonction peut s'appeler elle-même pour opérer sur de nouveaux paramètres. La fonction doit comporter un test d'arrêt (des appels récursifs) qu'il est conseillé de mettre en début de fonction.

1.4.1 Les boucles récursives

Une boucle (voir page 19) peut aussi s'écrire sous forme récursive. On recommence (appel récursif) le corps de la boucle tant qu'on n'a pas atteint la valeur limite de sortie de la boucle. La boucle est croissante ou décroissante suivant que l'écriture (l'action de la boucle) est faite avant ou après l'appel récursif.

```
// BouclesRecursives.java  boucles récursives

class BouclesRecursives {

  static void boucleDecroissante (int n) {
    if (n > 0) {
      System.out.println ("boucleDecroissante  valeur de n : " + n);
      boucleDecroissante (n-1);
    }
  }
```

```
  static void boucleCroissante (int n) {
    if (n > 0) {
      boucleCroissante (n-1);
      System.out.println ("boucleCroissante    valeur de n : " + n);
    }
  }

  public static void main (String[] args) {
    boucleDecroissante (5);
    System.out.println ("\n");
    boucleCroissante (5);
  }

} // class BouclesRecursives
```

Résultats d'exécution :

```
boucleDecroissante  valeur de n : 5
boucleDecroissante  valeur de n : 4
boucleDecroissante  valeur de n : 3
boucleDecroissante  valeur de n : 2
boucleDecroissante  valeur de n : 1
boucleCroissante    valeur de n : 1
boucleCroissante    valeur de n : 2
boucleCroissante    valeur de n : 3
boucleCroissante    valeur de n : 4
boucleCroissante    valeur de n : 5
```

1.4.2 La factorielle de N

La factorielle d'un nombre entier est le produit des nombres entiers inférieurs ou égaux au nombre dont on cherche factorielle. L'écriture se formule sous forme de récurrence :

```
n! = 1 si n = 0                          n factorielle vaut 1 si n vaut 0
n! = n * (n-1)!      n factorielle = n multiplié par factorielle de n-1
```

La fonction récursive **factRec**() découle directement de cette définition. **factIter**() est une version itérative équivalente.

```
// Factorielle.java  calcul de factorielle

//import mdpaquetage.es.*;   // paquetage es (voir remarque ci-dessous)

class Factorielle {

  // factorielle itératif
  // n >= 0 et n <= 20                          voir dépassement page 9
  static long factIter (int n) {                // factorielle itératif
    long f = 1;
    for (int i=1; i <= n; i++) f = f * i;
    return f;
  }
```

```
  // factorielle récursif
  // n >= 0 et n <= 20                          voir dépassement page 9
  static long factRec (int n) {                 // factorielle récursif
    if (n == 0) {
      return 1;
    } else {
      return n*factRec (n-1);
    }
  }

  public static void main (String[] args) {
    int n = 5;
    //System.out.print ("Valeur de n ? ");
    //int n = LectureClavier.lireEntier();     voir remarque ci-dessous
    System.out.println ("factIter  Factorielle (" + n + ") = "
                                                    + factIter (n));
    System.out.println ("factRec   Factorielle (" + n + ") = "
                                                    + factRec  (n));
  }

} // class Factorielle
```

Exemples de résultats :

```
factIter  Factorielle (5) = 120
factRec   Factorielle (5) = 120
```

Remarque : pour le test de factorielle, on peut se contenter de modifier directement la valeur de n. On peut aussi utiliser le paquetage `es` contenant la classe `LectureClavier` définissant une fonction (méthode) `lireEntier` qui permet d'entrer au clavier la valeur dont on calcule factorielle. Voir la notion de paquetage en page 65, et la définition du paquetage `es` en page 304.

1.4.3 La puissance de N

Le calcul de x^n (x à la puissance entière n) peut se faire en multipliant n fois x par lui-même. Une autre façon de procéder, surtout si n est grand, se formule comme suit :

x^n	$= 1$	si n = 0
x^n	$= x^{n/2} * x^{n/2}$	si n est pair
x^n	$= x^{(n-1)/2} * x^{(n-1)/2} * x$	si n est impair

Exemple :

x^4	$= x^2 * x^2$	n pair
x^5	$= x^2 * x^2 * x$	n impair

On applique alors la même méthode pour évaluer $x^{n/2}$. Le programme récursif en résultant est donné ci-après.

```
// PuissancedeN.java                    Puissance de N d'un nombre réel

class PuissancedeN {

  // puissance nième d'un nombre réel x (n entier >= 0)
  static double puissance (double x, int n) {
    double r;
    if (n == 0) {
      r = 1;
    } else {
      r = puissance (x, n/2);                          // appel récursif
      if (n%2 == 0) {        // n modulo 2 (reste de la division par 2)
        r = r*r;      // n pair
      } else {
        r = r*r*x;    // n impair
      }
    }
    return r;
  }

  public static void main (String[] args) {
    System.out.print (
      "\n3 puissance 4  = " + puissance (3, 4)  +
      "\n3 puissance 5  = " + puissance (3, 5)  +
      "\n2 puissance 0  = " + puissance (2, 0)  +
      "\n2 puissance 1  = " + puissance (2, 1)  +
      "\n2 puissance 2  = " + puissance (2, 2)  +
      "\n2 puissance 3  = " + puissance (2, 3)  +
      "\n2 puissance 10 = " + puissance (2, 10) +
      "\n2 puissance 32 = " + puissance (2, 32) +
      "\n2 puissance 64 = " + puissance (2, 64)
    );
  }  // main

}  // class PuissancedeN
```

Exemples de résultats :

```
3 puissance 4  = 81.0
3 puissance 5  = 243.0
2 puissance 0  = 1.0
2 puissance 1  = 2.0
2 puissance 2  = 4.0
2 puissance 3  = 8.0
2 puissance 10 = 1024.0
2 puissance 32 = 4.294967296E9
2 puissance 64 = 1.8446744073709552E19
```

1.5 CONCLUSION

Ce premier chapitre a présenté la syntaxe de base du langage Java. Pour les déclarations des types primitifs et des tableaux ou l'utilisation des instructions de contrôle, la syntaxe est très proche de celle du langage C. Il y a cependant des différences importantes : il n'y a pas de typedef, ni de déclarations de structures (struct en C), ni de variables globales, ni de directives de compilation (#include, #define, etc.), ni de pointeurs ; les entrées-sorties sont différentes (printf en C, System.out.println en Java), ainsi que la gestion des chaînes de caractères ; enfin on a un passage des paramètres par valeur ce qui pose des problèmes pour les types primitifs. La désallocation de mémoire est automatique, etc.

Ce chapitre introductif présente donc en quelque sorte, un moyen de **programmer en Java à la manière C**. Les notions de programmation objet ont été évitées. Les fonctions sont déclarées `static`, ce qui signifie qu'elles s'appliquent sans la définition explicite d'un objet.

Chapitre **2**

Les classes en Java

2.1 LES MODULES, LES CLASSES, L'ENCAPSULATION

La notion de module existe dans la plupart des langages non-objet. La définition générale d'un module est "une unité faisant partie d'un ensemble".

2.1.1 La notion de module et de type abstrait de données (TAD)

En programmation non-objet (Pascal, C), un module est un ensemble de fonctions traitant des données communes. Les éléments (constantes, variables, types, fonctions) déclarés dans la partie interface sont accessibles de l'extérieur du module, et sont utilisables dans un autre programme (un autre module ou un programme principal). Il suffit de référencer le module pour avoir accès aux éléments de sa partie interface. Celle-ci doit être la plus réduite possible, tout en donnant au futur utilisateur un large éventail de possibilités d'utilisation du module. Les déclarations de variables doivent être évitées au maximum. On peut toujours définir une variable locale au module à laquelle on accède ou que l'on modifie par des appels de fonctions de l'interface.

On parle alors *d'encapsulation* des données qui sont invisibles pour l'utilisateur du module et seulement accessibles à travers un jeu de fonctions. L'utilisateur du module n'a pas besoin de savoir comment sont mémorisées les données ; le module est pour lui un *type abstrait de données (TAD)*. Du reste, cette mémorisation locale peut évoluer, elle n'affectera pas les programmes des utilisateurs du module dès lors que les prototypes des fonctions d'interface restent inchangés.

En langage C, module.h constitue le fichier d'en-tête à inclure dans chaque fichier référençant des fonctions du module. Le corps du module est défini dans module.c.

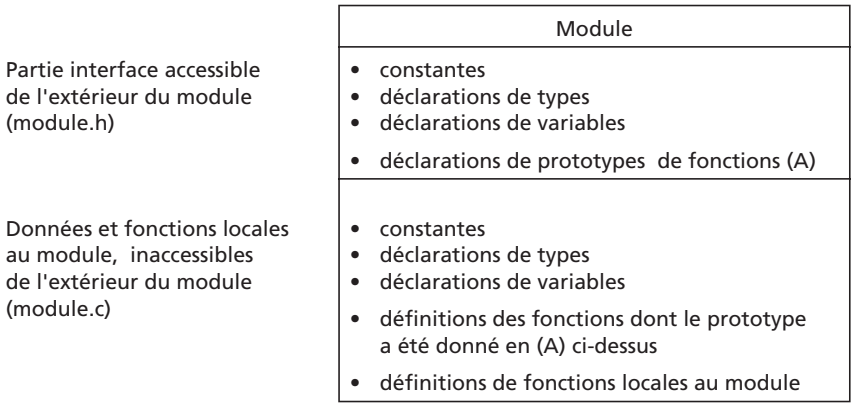


Figure 2.1 — La notion de module séparant ce qui est accessible de l'extérieur du module de ce qui est local au module, aussi bien pour les données que pour les fonctions.

Un exemple classique est celui de la pile. S'il n'y a qu'une pile à gérer, celle-ci peut être encapsulée dans les données locales (dans pile.c) ; la déclaration du fichier d'en-tête pourrait être la suivante en langage C.

```
// pile.h          version en langage C gérant une seule pile d'entiers

#ifndef PILE_H
#define PILE_H

void initPile   (int  max);                      // initialiser la pile
int  pileVide   ();                          // la pile est-elle vide ?
void empiler    (int  valeur);                    // empiler une valeur
int  depiler    (int* valeur);          // dépiler à l'adresse de valeur
void viderPile  ();                                  // vider la pile
void listerPile ();                    // lister les éléments de la pile

#endif
```

Si on veut gérer plusieurs piles, les données ne peuvent plus être dans la partie "données locales" du module. Celles-ci doivent être déclarées dans le programme appelant et passées en paramètres des fonctions du module gérant la pile. Le fichier d'en-tête pourrait s'écrire comme indiqué ci-dessous. La déclaration du type Pile doit être faite dans l'interface ; par contre les variables `max`, `sommet` et `element` ne devraient pas être accessibles de l'extérieur du module ce qui n'est pas le cas sur l'exemple suivant où le pointeur de pile donne accès aux composantes de la structure à partir du programme appelant.

```
// pile.h version en langage C gérant plusieurs piles d'entiers

#ifndef PILE_H
#define PILE_H

typedef struct {
  int  max;     // nombre max d'éléments dans la pile
  int  sommet;  // repère le dernier occupé de element
```

```
  int* element; // tableau d'entiers alloués dynamiquement
} Pile;

void initPile   (Pile* p, int  max);
int  pileVide   (Pile* p);
void empiler    (Pile* p, int  valeur);
int  depiler    (Pile* p, int* valeur);
void viderPile  (Pile* p);
void listerPile (Pile* p);

#endif
```

2.1.2 La notion de classe

Une classe est une extension de la notion de module. Les données et les fonctions traitant les données sont réunies ensemble dans une classe qui constitue un nouveau type. On peut déclarer des variables du type de cette classe. Cela revient à avoir la possibilité de dupliquer les modules en leur attribuant des noms différents, chaque module ayant ses propres variables locales visibles ou non de l'extérieur.

L'exemple de la pile peut être schématisé comme indiqué sur la figure 2.2 qui suit les conventions de la modélisation objet UML (Unified Modeling Language). On distingue les données de la classe appelées **attributs** en PO (Programmation Objet) et les fonctions appelées **méthodes**. Un signe moins(-) devant un élément indique un élément privé, visible seulement dans la classe ; un signe plus(+) indique un élément public, accessible par tous les utilisateurs de la classe ; l'absence de signe indique une visibilité (voir page 67) seulement pour les classes du paquetage (du répertoire). Les attributs `sommet` et `element`, et la méthode `erreur()` sont privés ; les autres méthodes sont publiques.

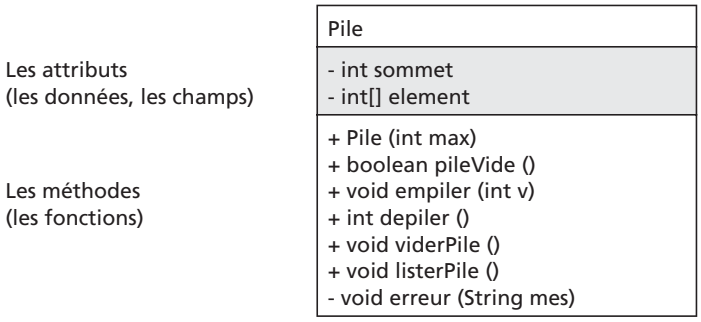


Figure 2.2 — La classe Pile (-indique un attribut ou une méthode privé).

La classe est un modèle qui s'apparente à la déclaration d'un nouveau type de données. On peut déclarer une variable (on parle d'**objet** ou d'**instance de la classe** en PO) du type de cette classe.

Exemple :

```
Pile p1 = new Pile (5);// p1 et p2 sont des instances de la classe Pile
Pile p2 = new Pile (100);      // p1 et p2 sont des objets de type Pile
```

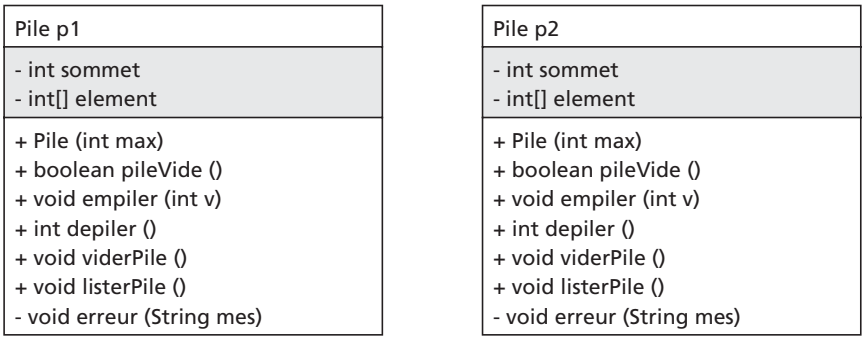


Figure 2.3 — Deux objets (deux instances) de la classe Pile.
Chaque objet a ses propres attributs (en grisé) qui sont privés (encapsulés dans l'objet). Les méthodes par contre ne sont pas dupliquées.

Chaque objet possède ses propres données (attributs). Les données en grisé sur la figure 2.3 sont **encapsulées** dans leur objet. La déclaration des objets p1 et p2 autorise seulement l'utilisation des méthodes de l'objet. Aucun accès direct (en dehors des méthodes de l'objet) n'est possible à `sommet` ou `element` de l'objet. On parle aussi d'**envoi de messages** à l'objet : chaque méthode s'adresse à l'objet pour qu'il modifie certaines de ses données ou fournisse de l'information sur son état : message p1.empiler(3) ; message p2.viderPile() ; etc.

D'autres structures de données seraient envisageables pour mémoriser la pile. Si l'interface est respectée, un changement de structures de données dans la classe Pile n'affectera pas les programmes utilisateurs qui n'ont pas accès aux données privées. On parle aussi d'un **contrat** passé entre la classe et ses utilisateurs, les clauses du contrat étant définies par les méthodes publiques.

2.1.3 La classe Pile

Le codage en Java de la classe Pile est indiqué ci-après. Le mot clé "private" indique que les attributs `sommet` et `element`, et la méthode `erreur()` sont privés.

```
// Pile.java  gestion d'une pile d'entiers

public class Pile {
  // sommet et element sont des attributs privés
  private int   sommet;         // repère le dernier occupé (le sommet)
  private int[] element;      // tableau d'entiers alloué dynamiquement

  // erreur est une méthode privée utilisable seulement
  // dans la classe Pile
  private void erreur (String mes) {
    System.out.println ("***erreur : " + mes);
  }

  public Pile (int max) {      // voir 2.2.1 Le constructeur d'un objet
    sommet  = -1;
    element = new int [max];               // allocation de max entiers
  }
```

```
  public boolean pileVide () {
    return sommet == -1;
  }

  public void empiler (int v) {
    if (sommet < element.length - 1) {
       sommet++;
       element [sommet] = v;
    }  else {
      erreur ("Pile saturee");
    }
  }

  public int depiler () {
    int v = 0;                  // v est une variable locale à depiler()
    if (!pileVide()) {
      v = element [sommet];
      sommet--;
    }  else {
      erreur ("Pile vide");
    }
    return v;
  }

  public void viderPile () {
    sommet = -1;
  }

  public void listerPile () {
    if (pileVide()) {
      System.out.println ("Pile vide");
    } else {
      System.out.println ("Taille de la pile : " + element.length);
      for (int i=0; i <= sommet; i++) {
        System.out.print (element[i] + " ");
      }
      System.out.println();                              // à la ligne
    }
  }

}  // class Pile
```

2.1.4 La mise en œuvre de la classe Pile

Dans le programme `PPPile` ci-dessous, on crée l'objet `pile1` de type `Pile` sur lequel on applique, en fonction des réponses au menu de l'utilisateur, des méthodes de la classe `Pile`. Les données sont encapsulées. L'accès à l'entier `sommet` de la `Pile pile1` serait autorisé si `sommet` était déclaré `public` dans la classe `Pile` ; dans ce cas `pile1.sommet` serait valide dans `PPPile.java` pour accéder à la variable `sommet` de la pile. En vertu du principe d'encapsulation, il est préférable que cet accès soit interdit.

```
// PPPile.java  Programme Principal de la Pile

import mdpaquetage.es.*;    // lireEntier                (voir page 304)

class PPPile {                          // Programme Principal de la Pile

  static int menu () {
    System.out.print (
      "\n\nGESTION D'UNE PILE\n\n" +
      "0 - Fin\n" +
      "1 - Initialisation de la pile\n"  +
      "2 - La pile est-elle vide ?\n"    +
      "3 - Insertion   dans la pile\n"   +
      "4 - Retrait     de   la pile\n"   +
      "5 - Vidage      de   la pile\n"   +
      "6 - Listage     de   la pile\n\n" +
      "Votre choix ? "
    );
    return LectureClavier.lireEntier();                 (voir page 304)
  }

  public static void main (String[] args) {
    // pile1, valeur et fini sont des variables locales à main()
    Pile    pile1 = new Pile (5) ;              // créer un objet Pile
    int     valeur;
    boolean fini = false;

    while (!fini) {

      switch (menu()) {

      case 0 :  // fin
        fini = true;
        break;

      case 1 :  // initialisation
        // une pile par défaut est créée lors du new Pile (5)
        System.out.print ("Taille de la pile ? ");
        int taille = LectureClavier.lireEntier();
        pile1 = new Pile (taille) ;
        System.out.println ("Initialisation d'une pile vide");
        break;

      case 2 :  // la pile est-elle vide ?
        if (pile1.pileVide() ) {
          System.out.println ("La pile est vide");
        } else {
          System.out.println ("La pile n'est pas vide");
        }
        break;

      case 3 :  // empiler une valeur
        System.out.print ("Valeur a empiler ? ");
```

```
            valeur = LectureClavier.lireEntier();
            pile1.empiler (valeur) ;
            break;

          case 4 :  // dépiler une valeur
            valeur = pile1.depiler();
            System.out.println ("Valeur depilee : " + valeur);
            break;

          case 5 :  // vider la pile
            pile1.viderPile();
            break;

          case 6 :  // lister la pile
            pile1.listerPile();
            break;
          } // switch
        } // while (!fini)

      } // main

    } // class PPPile
```

Exemples de résultats d'exécution de PPPile.java :

```
GESTION D'UNE PILE
0 - Fin
1 - Initialisation de la pile
2 - La pile est-elle vide ?
3 - Insertion   dans la pile
4 - Retrait     de   la pile
5 - Vidage      de   la pile
6 - Listage     de   la pile
Votre choix ? 1
Taille de la pile ? 4
Initialisation d'une pile vide
Menu
Votre choix ? 2
La pile est vide
Menu
Votre choix ? 4
***erreur : Pile vide
Valeur depilee : 0
Menu
Votre choix ? 3
Valeur a empiler ? 12
Menu
Votre choix ? 4
Valeur depilee : 12
```

Si la pile est saturée :

```
Menu
Votre choix ? 3
Valeur a empiler ? 14
***erreur : Pile saturee
```

Remarque : les cas d'erreurs pourraient être gérés différemment en utilisant la notion d'exceptions pour les détecter et les prendre en compte (voir les exceptions page 74).

2.2 LA NOTION DE CONSTRUCTEUR D'UN OBJET

Un objet est une instance, un exemplaire construit dynamiquement sur le modèle que décrit la classe.

2.2.1 Le constructeur d'un objet

Dans l'exemple précédent de la pile, la méthode suivante est un peu particulière.

```
public Pile (int max) {
  sommet  = -1;
  element = new int [max];              // allocation de max entiers
}
```

Elle porte seulement le nom de la classe en cours de définition. Cette méthode est un **constructeur** chargé de construire l'objet (allouer de la mémoire et initialiser les attributs de l'objet). Ce constructeur n'est jamais appelé directement ; il est pris en compte lors de la demande de création de l'objet avec new :

```
Pile pile1 = new Pile (5) ;
```

L'attribut `sommet` de l'objet `pile1` est initialisé à -1, et un tableau de 5 entiers repéré par la référence `element` est alloué. pile1 est une référence sur l'objet (son adresse). L'objet contient lui-même une référence `element` sur un tableau d'entiers.

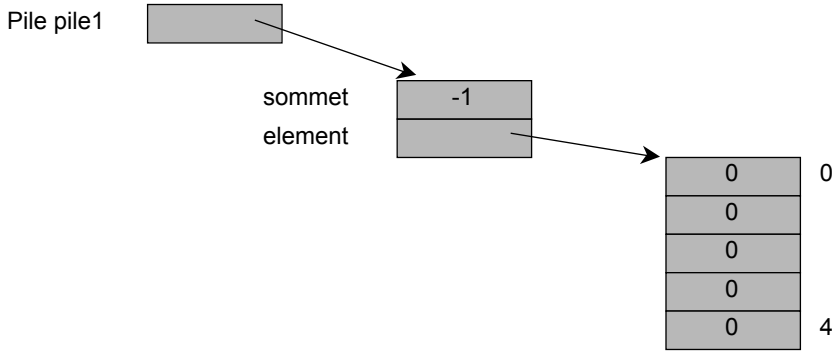


Figure 2.4 — La construction dynamique d'un objet de type Pile avec Pile pile1 = new Pile (5).

2.2.2 L'initialisation d'un attribut d'un objet

L'initialisation d'un attribut d'un objet est faite :

- d'abord par défaut lors de la création de l'attribut. Pour les types primitifs, l'attribut est initialisé à 0 pour les entiers, les réels et les caractères et à faux pour les booléens. Un attribut d'un objet peut être une référence sur un autre objet (ou un tableau) comme sur la figure 2.4 ; dans ce cas, la référence est initialisée à null (adresse inconnue) : l'objet devra être créé par la suite, éventuellement dans le constructeur à l'aide de new().
- en utilisant la valeur indiquée lors de sa déclaration comme attribut :

```
class Pile {
  private int sommet = -1;      // repère le dernier occupé (le sommet)
  ...
```

  initialise l'attribut `sommet` à -1 lors de sa déclaration.
- dans le constructeur de l'objet : le constructeur Pile (int max) initialise `sommet` à -1 ; il initialise la référence `element` en créant un tableau d'entiers.

> ***Remarque*** : il n'y a pas à désallouer la mémoire en Java. Cela est fait automatiquement par un processus ramasse-miettes (garbage collector) qui récupère la mémoire des objets qui ne sont plus référencés. En C, on désalloue avec la fonction free() et en C++, on peut définir un destructeur pour une classe.

2.3 LES ATTRIBUTS STATIC

2.3.1 La classe Ecran

On veut simuler un écran graphique comportant un certain nombre de lignes et de colonnes. Chaque point de l'écran contient un caractère correspondant à un pixel. On dessine sur cet écran à l'aide d'un jeu de fonctions définies comme suit :

- **Ecran** (int nbLig, int nbCol) : crée un écran de nbLig lignes sur nbCol colonnes.
- **effacerEcran** () : efface l'écran pour un nouveau dessin.
- **crayonEn** (int numLigCrayon, int numColCrayon) : positionne le crayon sur une ligne et une colonne.
- **changerCouleurCrayon** (int couleurCrayon) : change la couleur du crayon.
- **ecrirePixel** (int nl, int nc) : écrit un pixel avec la couleur du crayon (en fait ici, un caractère) pour un numéro de ligne et de colonne donnés en (nl, nc).
- **ecrireMessage** (int nl, int nc, String mes) : écrit le message mes en (nl, nc).
- **tracerCadre** () : trace un cadre autour de l'écran.
- **avancer** (int d, int n) : avance le crayon dans une direction d donnée (gauche, haut, droite, bas) d'un nombre de pas n déterminé.
- **afficherEcra** () : affiche le contenu de l'écran (les caractères de l'écran).
- **rectangle** (int xCSG, int yCSG, int xCID, int yCID) : trace un rectangle à partir des coordonnées CSG (coin supérieur gauche) et CID (coin inférieur droit).

- **spirale1** (int n, int max) : trace une spirale en partant de l'intérieur de la spirale.
- **spirale2** (int n, int d) : trace une spirale en partant de l'extérieur de la spirale.

L'écran graphique est schématisé sur la figure 2.5.

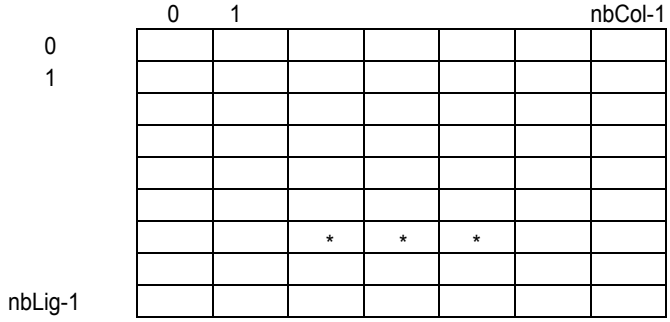


Figure 2.5 — L'écran graphique simulé par des caractères.

Les spécifications de la classe Ecran sont résumées sur le schéma UML de la figure 2.6 et représentent le jeu de méthodes disponibles pour l'utilisateur de la classe Ecran.

	Ecran
Les attributs (les données)	
Les méthodes (les fonctions)	**Ecran** (int nbLig, int nbCol) void **effacerEcran** () void **crayonEn** (int numLigCrayon, int numColCrayon) void **changerCouleurCrayon** (int couleurCrayon) void **ecrirePixel** (int nl, int nc) void **ecrireMessage** (int nl, int nc, String mes) void **tracerCadre** () void **avancer** (int d, int n) void **afficherEcran** () void **rectangle** (int xCSG, int yCSG, int xCID, int yCID) void **spirale1** (int n, int max) void **spirale2** (int n, int d)

Figure 2.6 — La classe Ecran (vue **utilisateur**).

Tous les attributs sont privés et toutes les méthodes sont publiques. L'utilisateur ne connaît pas les structures de données utilisées. Il communique seulement par l'intermédiaire d'un jeu de méthodes.

2.3.2 L'implémentation de la classe Ecran

Les données étant encapsulées dans la classe `Ecran`, le choix des structures de données est libre pour le concepteur de la classe Ecran. Cela n'a pas d'incidence pour les utilisateurs de la classe qui n'ont accès qu'aux méthodes (voir figure 2.6).

Les attributs privés suivants sont définis lors de l'implémentation de la classe Ecran :

- `nbLig`, `nbCol` : nombre de lignes et de colonnes de l'écran.
- `numLigCrayon`, `numColCrayon` : numéro de la ligne et de la colonne où se trouve le crayon.
- `couleurCrayon` : numéro (de 0 à 15) de la couleur du crayon. On affiche en fait sur l'écran un caractère dépendant de la couleur du crayon.
- `zEcran` (zone de l'écran) est une référence sur un tableau de caractères à deux dimensions alloué dynamiquement dans le constructeur de la classe `Ecran` ; `nbLig` et `nbCol` sont initialisés dans ce constructeur.

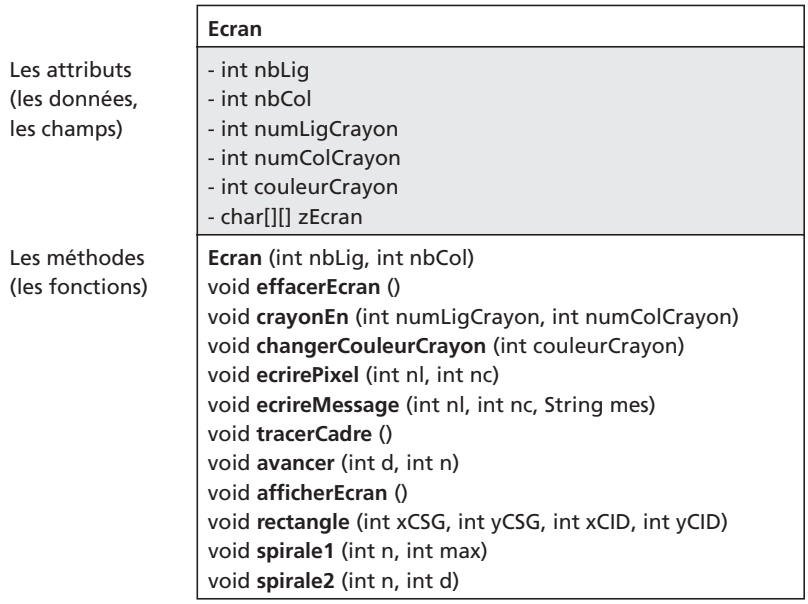


Figure 2.7 — La classe Ecran avec ses attributs privés (vue **concepteur**).

2.3.3 La classe Ecran en Java : le rôle de this

L'absence du mot-clé `public` devant la déclaration de la classe Ecran et devant chacune des méthodes de cette classe implique en fait une visibilité des méthodes s'étendant au paquetage, soit au répertoire contenant Ecran. Cela indique que le programme principal et la classe Ecran doivent être dans le même répertoire (voir page 67). Une constante symbolique se déclare avec final (voir page 8). Le mot clé "static" est présenté dans le paragraphe suivant.

```
// Ecran.java  simulation d'un écran graphique

class Ecran {
  // constantes symboliques pour les couleurs noir et blanc
  // les autres couleurs pourraient aussi être définies
  static final int  NOIR  =  0;
  static final int  BLANC =  15;
```

```
// constantes symboliques pour les directions
static final int  GAUCHE = 1;
static final int  HAUT   = 2;
static final int  DROITE = 3;
static final int  BAS    = 4;

// les attributs (données) de l'objet sont privés
private int      nbLig;         // nombre de lignes   de l'écran
private int      nbCol;         // nombre de colonnes de l'écran
private int      numLigCrayon;  // numéro de ligne    du crayon
private int      numColCrayon;  // numéro de colonne  du crayon
private int      couleurCrayon; // couleur            du crayon
private char[][] zEcran;        // référence sur le tableau à 2D
```

Le constructeur Ecran doit allouer et initialiser les attributs (données) privés de l'objet courant. Cette initalisation se fait souvent en donnant au paramètre le même nom que l'attribut qu'il doit initialiser. Ainsi, le constructeur Ecran a deux paramètres nbLig et nbCol (nombre de lignes et de colonnes) qui servent à initialiser les attributs privés nbLig et nbCol. Par convention, **this désigne l'objet courant** (c'est une référence sur l'objet courant) et il permet de lever l'ambiguïté de nom : **this.nbLig** désigne **l'attribut** alors que **nbLig** seul désigne le **paramètre**. Le principe s'applique également lorsqu'on doit changer la valeur d'un attribut comme dans les méthodes crayonEn() ou changerCouleurCrayon() ci-après.

```
// tableau de caractères zEcran alloué dynamiquement
// dans le constructeur Ecran()
Ecran (int nbLig, int nbCol) {          // nbLig désigne le paramètre
  this.nbLig = nbLig ;                 // this.nbLig désigne l'attribut
  this.nbCol = nbCol ;
  zEcran = new char [nbLig][nbCol];    // référence sur la zone écran
  changerCouleurCrayon (NOIR);              // par défaut crayon noir
  crayonEn (nbLig/2, nbCol/2);                // au milieu de l'écran
  effacerEcran();
}

// mettre l'écran à blanc
void effacerEcran () {                              // voir figure 2.5
  for (int i=0; i < nbLig; i++) {
    for (int j=0; j < nbCol; j++) zEcran [i][j] = ' ';
  }
}

// le crayon est mis en numLigCrayon, numColCrayon
void crayonEn (int numLigCrayon, int numColCrayon) {
  this.numLigCrayon = numLigCrayon ;
  this.numColCrayon = numColCrayon ;
}

// la couleur du crayon est couleurCrayon de 0 à 15
void changerCouleurCrayon (int couleurCrayon) {
  if ( (couleurCrayon >= 0) && (couleurCrayon <= 15) ) {
    this.couleurCrayon = couleurCrayon ;
  }
}
```

Les fonctions suivantes sont les fonctions élémentaires dessinant un pixel (en fait un caractère dans notre simulation), écrivant un message sous forme d'une chaîne de caractères, ou traçant un cadre entourant l'écran. La fonction avancer() avance suivant une direction donnée en laissant une trace de la couleur courante du crayon (en fait écrit des caractères correspondant à la couleur courante). Les dessins n'apparaissent qu'à la demande lors de l'appel de afficherEcran().

```
// écrire un caractère en (nl, nc) en fonction de couleurCrayon
void ecrirePixel (int nl, int nc) {
  String couleurs = "*123456789ABCDE.";      // 16 couleurs de 0 à 15
  if ( (nl >= 0) && (nl < nbLig) && (nc >= 0) && (nc < nbCol) ) {
    zEcran [nl][nc] = couleurs.charAt (couleurCrayon);
  }
}

// écrire le message mes en (nl, nc)
void ecrireMessage (int nl, int nc, String mes) {
  for (int i=0; i < mes.length(); i++) {
    if ( (nl >= 0) && (nl < nbLig) && (nc >= 0) && (nc < nbCol) ) {
      zEcran [nl][nc] = mes.charAt (i);        // voir String page 80
      nc++;
    }
  }
}

// tracer un cadre autour de l'écran
void tracerCadre () {
  for (int nc=0; nc < nbCol; nc++) {
    zEcran [0][nc] = '-';                        // le haut de l'écran
    zEcran [nbLig-1][nc] = '-';                   // le bas de l'écran
  }
  for (int nl=0; nl < nbLig; nl++) {
    zEcran [nl][0] = '|';                           // le côté gauche
    zEcran [nl][nbCol-1] = '|';                      // le côté droit
  }
}

// avancer dans la direction d de n caractères (pixels)
void avancer (int d, int n) {
  switch (d) {
  case DROITE :
    for (int i=numColCrayon; i < numColCrayon+n; i++) {
      ecrirePixel (numLigCrayon, i);
    }
    numColCrayon += n-1;
    break;
  case HAUT :
    for (int i=numLigCrayon; i > numLigCrayon-n; i--) {
      ecrirePixel (i, numColCrayon);
    }
```

```
      numLigCrayon += -n+1;
      break;
    case GAUCHE :
      for (int i=numColCrayon; i > numColCrayon-n; i--) {
        ecrirePixel (numLigCrayon, i);
      }
      numColCrayon += -n+1;
      break;
    case BAS :
      for (int i=numLigCrayon; i < numLigCrayon+n; i++) {
        ecrirePixel (i, numColCrayon);
      }
      numLigCrayon += n-1;
      break;
    }
  }

  void afficherEcran () {
    for (int i=0; i < nbLig; i++) {
      for (int j=0; j < nbCol; j++) {
        System.out.print (zEcran [i][j]);
      }
      System.out.print ("\n");
    }
    System.out.print ("\n");
  }
```

Les méthodes suivantes dessinent des figures géométriques sur un objet de la classe Ecran en utilisant les fonctions élémentaires vues ci-dessus. On peut dessiner un rectangle, ou une spirale en partant du plus petit côté intérieur ou en partant du grand côté extérieur.

```
  // tracer un rectangle défini par 2 points CSG et CID
  // CSG : coin supérieur gauche; CID : coin inférieur droit
  void rectangle (int xCSG, int yCSG, int xCID, int yCID) {
    int longueur = xCID-xCSG+1;
    int largeur  = yCID-yCSG+1;

    crayonEn (yCSG,   xCSG);
    avancer  (BAS,    largeur);
    avancer  (DROITE, longueur);
    avancer  (HAUT,   largeur);
    avancer  (GAUCHE, longueur);
  }

  // tracer une spirale en partant du centre de la spirale
  // et en se déplacant de n pas dans la direction courante.
  // augmenter n, et tourner à gauche de 90°.
  // On continue tant que n est < à max
  void spirale1 (int n, int max) {                          // récursif
    if (n < max) {
      avancer  (DROITE, n);
```

```
        avancer  (HAUT  , n+1);
        avancer  (GAUCHE, n+2);
        avancer  (BAS,    n+3);
        spirale1 (n+4,    max);
      }
    }

    // tracer une spirale en partant du plus grand côté
    // extérieur de la spirale, et en décrémentant n
    // tant que n est supérieur à 0.
    void spirale2 (int n, int d) {                          // récursif
      if (n >= 1) {
        avancer (d, n);        // avancer dans la direction d de n pixels
        spirale2 (n-1, 1 + d%4 );         // d modulo 4 soit 0, 1, 2 ou 3
      }
    }

  }  // classe Ecran
```

2.3.4 Les attributs static (les constantes static final)

Le programme suivant utilise la classe Ecran définie précédemment pour effectuer des dessins sur deux objets différents ecran1 et ecran2. Chaque objet dispose de ses propres données (attributs) privées.

Ecran ecran1
- int nbLig - int nbCol - int numLigCrayon - int numColCrayon - int couleurCrayon - char[][] zEcran
Ecran (int nbLig, int nbCol) void effacerEcran () void crayonEn (int numLigCrayon, int numColCrayon) void changerCouleurCrayon (int couleurCrayon) void ecrirePixel (int nl, int nc) void ecrireMessage (int nl, int nc, String mes) void tracerCadre () void avancer (int d, int n) void afficherEcran () void rectangle (int xCSG, int yCSG, int xCID, int yCID) void spirale1 (int n, int max) void spirale2 (int n, int d)

Ecran ecran2
- int nbLig - int nbCol - int numLigCrayon - int numColCrayon - int couleurCrayon - char[][] zEcran
Ecran (int nbLig, int nbCol) void effacerEcran () void crayonEn (int numLigCrayon, int numColCrayon) void changerCouleurCrayon (int couleurCrayon) void ecrirePixel (int nl, int nc) void ecrireMessage (int nl, int nc, String mes) void tracerCadre () void avancer (int d, int n) void afficherEcran () void rectangle (int xCSG, int yCSG, int xCID, int yCID) void spirale1 (int n, int max) void spirale2 (int n, int d)

Figure 2.8 — Deux objets de type Ecran.
Chaque objet dispose de ses propres attributs (en grisé sur le schéma).

Certains attributs peuvent être **communs à** tous les objets de **la classe** et exister indépendamment de tout objet de la classe. Ces attributs sont déclarés `static`. On peut trouver des variables `static` et des constantes `static`. Les constantes (NOIR, BLANC, GAUCHE, HAUT, DROITE, BAS) n'ont pas besoin d'être dupliquées dans chaque objet de la classe. Sur la figure 2.9, les constantes (car déclarées `final`) sont `static` donc existent indépendamment de tout objet. Ces constantes ne sont pas déclarées `public` ; elles sont visibles dans le paquetage (voir page 67). Elles pourraient être déclarées `private`, et visibles seulement dans les méthodes de l'objet.

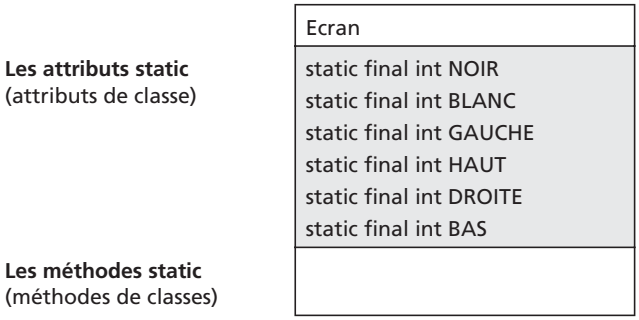


Figure 2.9 — Les attributs **static** sont mémorisés **une seule fois** et sont **communs** à tous les objets de la classe Ecran (cas des constantes par exemple).

La déclaration **static** indique que ces constantes sont spécifiques de la classe Ecran mais non spécifiques d'un objet particulier (comme ecran1 ou ecran2). On appelle ces variables des **attributs de classe**. Dans les méthodes de la classe, elles sont directement référencées en indiquant leur nom (voir l'utilisation de GAUCHE, DROITE, etc., dans la méthode avancer() de la classe Ecran). A l'extérieur de la classe, elles sont référencées en les préfixant du nom de la classe (voir ci-dessous dans la classe PPEcran : **Ecran.HAUT** par exemple signifie la variable (constante) static HAUT de la classe Ecran).

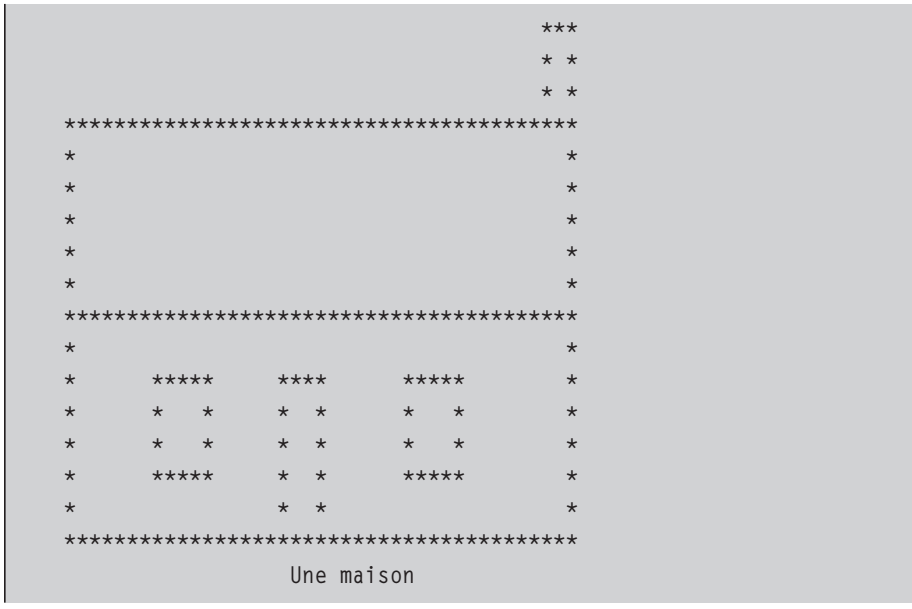
2.3.5 La mise en œuvre de la classe Ecran

La mise en œuvre de la classe consiste simplement en une suite d'appels des méthodes disponibles. L'utilisateur de la classe ne connaît pas les structures de données utilisées pour l'implémentation.

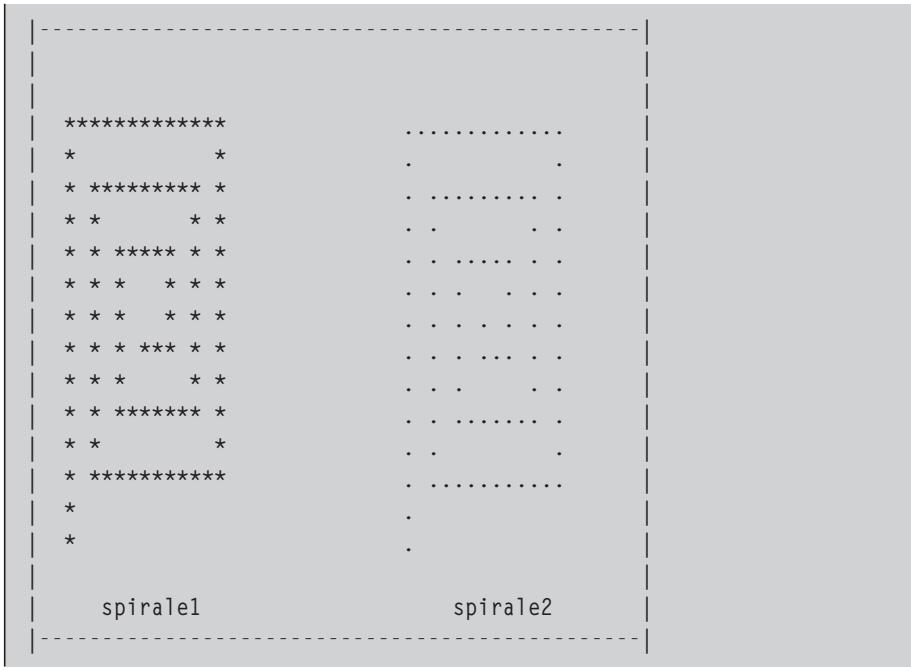
```
// PPEcran.java  Programme Principal Ecran

class PPEcran {                              // Programme Principal Ecran

  public static void main (String[] args) {

    // la maison
    Ecran ecran1 = new Ecran (20, 50);          // constructeur de Ecran
    ecran1.rectangle ( 3, 10, 43, 17);  // maison
```

```
        ecran1.rectangle ( 3,  4, 43, 10);  // toiture
        ecran1.rectangle (20, 12, 23, 17);  // porte
        ecran1.rectangle (41,  1, 43,  4);  // cheminée
        ecran1.rectangle (10, 12, 14, 15);  // fenêtre gauche
        ecran1.rectangle (30, 12, 34, 15);  // fenêtre droite
        ecran1.ecrireMessage (19, 20, "Une maison");
        ecran1.afficherEcran();

        // spirale1
        Ecran ecran2 = new Ecran (20, 50);         // constructeur de Ecran
        ecran2.tracerCadre();
        ecran2.crayonEn (10, 9);
        ecran2.spirale1 (3, 15);           // premier segment de longueur 3
        ecran2.ecrireMessage (18, 5, "spirale1");

        // spirale2
        ecran2.crayonEn (16, 30);
        ecran2.changerCouleurCrayon (Ecran.BLANC);      // constante static
        ecran2.spirale2 (14, Ecran.HAUT);   // premier segment vers le haut
        ecran2.ecrireMessage (18, 33, "spirale2");
        ecran2.afficherEcran();
    }
}  // class PPEcran
```

Exemple de résultats :



Avec un cadre autour de l'écran :



Remarque : les méthodes de cette classe Ecran s'apparentent aux fonctions disponibles dans le langage Logo conçu pour apprendre la programmation aux enfants, de façon ludique, en guidant une tortue qui laisse une trace ou non lors de son déplacement. Pour réaliser un dessin, un ensemble de commandes permet d'avancer la tortue d'un certain nombre de pas, de la faire tourner d'un certain nombre de degrés, etc.

2.4 LA SURCHARGE DES MÉTHODES, LES MÉTHODES STATIC

2.4.1 La classe Complex des nombres complexes

Les nombres complexes sont mis en œuvre grâce à une classe Complex. Un nombre complexe est caractérisé par deux nombres réels : sa partie réelle (pReel) et sa partie imaginaire (pImag). Les **attributs** retenus sont donc deux `double` pReel et pImag qui sont privés. Un autre choix de mémorisation (module et argument par exemple) n'affectera pas les utilisateurs de la classe puisque ses attributs sont privés. Les méthodes sont toutes publiques, et constituent le jeu de fonctions disponibles pour traiter des nombres complexes :

- double **partieRC** () : fournit la partie réelle du nombre complexe.
- double **partieIC** () : fournit la partie imaginaire du nombre complexe.

- double **module** () : fournit le module du nombre complexe.
- double **argument** () : fournit l'argument du nombre complexe.
- Complex **oppose** () : fournit l'opposé du nombre complexe.
- Complex **conjugue** () : fournit le conjugué du nombre complexe
- Complex **inverse** () : fournit l'inverse du nombre complexe.
- Complex **puissance** (int n) : fournit la puissance n° du nombre complexe.
- Complex **plus** (Complex z) : fournit l'addition du nombre complexe et de z.
- Complex **moins** (Complex z) : fournit la soustraction du nombre complexe et de z.
- Complex **multiplier** (Complex z) : fournit la multiplication du nombre complexe et de z.
- Complex **diviser** (Complex z) : fournit la division du nombre complexe et de z.
- String **toString** () : fournit la chaîne de caractères d'édition du nombre complexe.
- void **ecritC** () : écrit le nombre complexe sous la forme (pReel + pImag i).
- void **ecritCP** () : écrit le nombre complexe sous la forme polaire (module, argument).
- void **ecrit** () : appelle ecritC() et ecritCP.

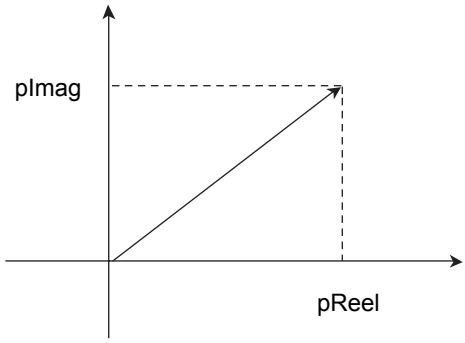


Figure 2.10 — Le nombre complexe (pReel = 0.707, pImag = 0.707). Le module vaut 1 et l'argument PI/4 soit 45 degrés.

Le module correspond à la longueur de l'hypoténuse du triangle de côté pReel et pImag. L'argument correspond à l'angle en radians (PI/4 = 0.785 radians par exemple ou 45 degrés).

2.4.2 La surcharge des méthodes

En programmation objet et en Java en particulier, un même nom de fonction peut être utilisé pour plusieurs fonctions. C'est notamment le cas lorsque les fonctions réalisent des actions conceptuellement voisines. On dit qu'il y a **surcharge** des fonctions. Il doit cependant y avoir possibilité de distinguer quelle est la fonction à mettre en œuvre. Le compilateur se base sur le nombre de paramètres, le type des paramètres ou l'ordre des types des paramètres pour lever l'ambiguïté. Les méthodes

surchargées doivent pouvoir se différencier sur cette base (appelée signature) lors de l'appel. On ne peut donc pas **surcharger** dans une **même classe** deux méthodes ayant rigoureusement même nombre, même type et même ordre des paramètres. La valeur de retour n'est pas prise en compte pour cette différenciation.

2.4.2.1 La surcharge des constructeurs

Cette surcharge des fonctions s'applique aussi pour les constructeurs (voir page 35). La construction d'un nombre complexe se fait à l'aide d'un des trois constructeurs suivants :

```
Complex (double pReel, double pImag)
Complex ()
Complex (double module, double argument, int typ)
```

Le premier constructeur construit un objet Complex à l'aide de deux `double` représentant la partie réelle et la partie imaginaire. Le deuxième constructeur est un constructeur par défaut qui construit un Complex avec une partie réelle et une partie imaginaire mises à 0. Le troisième constructeur construit un Complex en partant de ses valeurs polaires (son module et son argument). L'argument est donné en radians ou en degrés suivant la valeur du troisième paramètre (RADIANS ou DEGRES, constantes static de la classe Complex). Les instructions suivantes sont valides pour déclarer et initialiser un Complex (créer un objet Complex) ; PI est une constante static de la classe Complex :

```
Complex c1 = new Complex (2, 1.50);
Complex z  = new Complex ();                          (0, 0) par défaut

Complex p2 = new Complex (1, Complex.PI/2, Complex.RADIANS);
Complex p3 = new Complex (1, 90, Complex.DEGRES);
```

2.4.2.2 La surcharge des méthodes de la classe Complex

Dans une même classe, plusieurs méthodes peuvent avoir le même nom dès lors qu'on peut les distinguer grâce au nombre de paramètres et à leur type. La classe Complex a quatre méthodes surchargées qui correspondent aux quatre opérations arithmétiques. Le fait que certaines méthodes soient déclarées `static` est expliqué dans le prochain paragraphe. Suivant le nombre de paramètres (un ou deux), on déclenche la première ou la deuxième méthode **plus**(). Il en est de même pour les autres opérations.

```
public        Complex plus       (Complex z);
public static Complex plus       (Complex z1, Complex z2);
public        Complex moins      (Complex z);
public static Complex moins      (Complex z1, Complex z2);
public        Complex multiplier (Complex z);
public static Complex multiplier (Complex z1, Complex z2);
public        Complex diviser    (Complex z);
public static Complex diviser    (Complex z1, Complex z2);
```

2.4.3 Les attributs et méthodes static

2.4.3.1 Les attributs static

L'exemple de la figure 2.9 a montré la possibilité de définir des attributs `static` communs à tous les objets d'une classe. La figure 2.11 indique trois constantes `static` pour la classe Complex qui définissent la valeur PI et deux constantes indiquant si l'argument est en radians ou en degrés (voir `static final` en 2.3.4). La variable

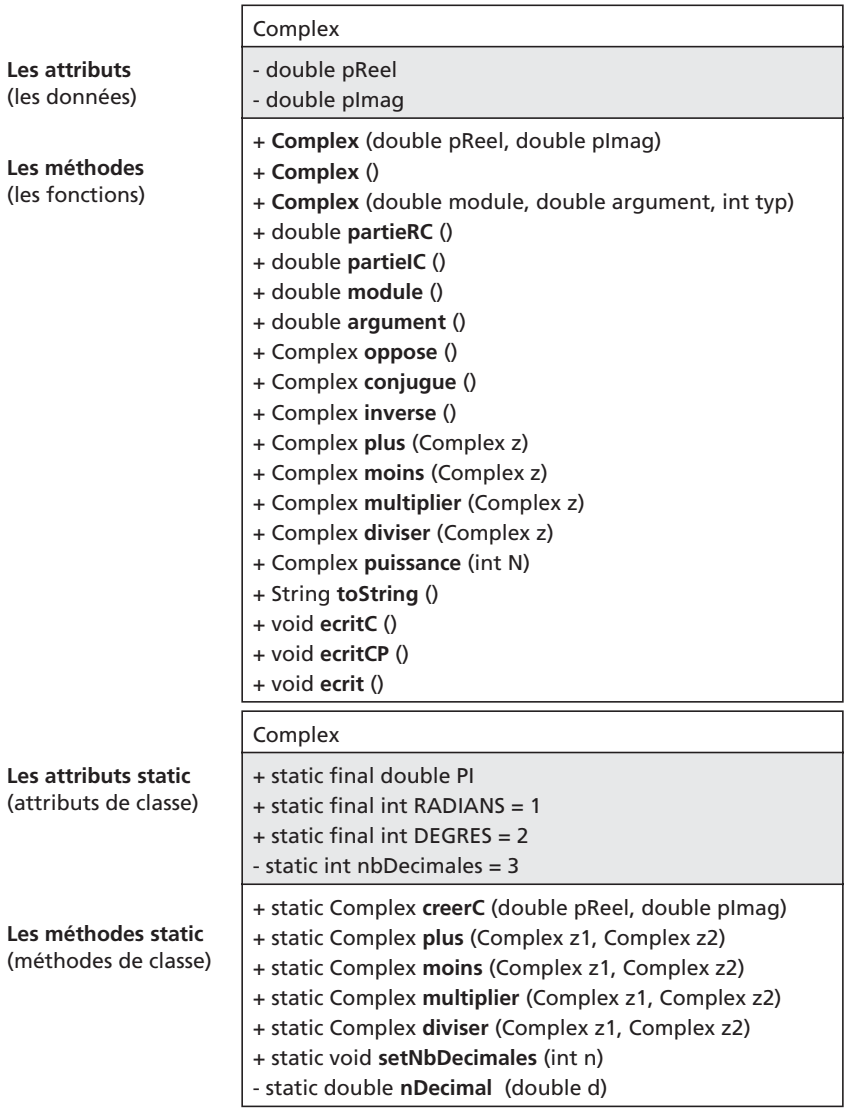


Figure 2.11 — La classe Complex.

Les attributs et méthodes d'instance sont séparés des attributs et méthodes de classes pour mieux mettre en évidence leurs différences. Les attributs et méthodes static sont normalement soulignés pour les distinguer des attributs et méthodes d'instance.

nbDecimales est utilisée pour indiquer le nombre de décimales à fournir pour l'écriture des valeurs des complexes. Cette variable est commune pour tous les nombres Complex. C'est une variable de la classe Complex, non une variable d'un objet de la classe Complex comme le sont pReel et pImag. D'où les attributs suivants pour la classe Complex (pReel et pImag sont des **attributs d'instance** ; nbDecimales est un **attribut de classe**) :

```
private double pReel; // partie Réelle
private double pImag; // partie Imaginaire

private static int nbDecimales = 3;  // nombre de décimales de 0 à 10
```

Les attributs `static` sont initialisés par défaut à 0 pour les nombres entiers ou réels, à false pour les booléens et à null pour les références (voir 2.2.2, page 36).

2.4.3.2 Les méthodes static

Les méthodes d'un objet s'applique implicitement à l'objet qui a déclenché la méthode. Si p1 est un Complex,

```
p1.partieRC();
```

fournit la partie réelle de p1. La méthode partieRC() est conçue pour délivrer la partie réelle de l'objet qui l'a appelée. De même, p1 et p2 étant des Complex, la méthode :

```
Complex p3 = p1.plus (p2);
```

fournit le résultat p3 de l'addition p1 plus p2.

On peut aussi demander que la méthode ne porte sur aucun objet implicite. L'addition de deux Complex peut se faire en mentionnant les deux Complex en paramètres et en ne préfixant pas l'appel de la méthode de l'objet concerné. La méthode est dite `static`. Les méthodes `static` sont des fonctions au sens classique du terme. En fonction de leurs paramètres, elles réalisent une action ou délivrent un résultat indépendamment de tout objet. Elles sont mises en œuvre (en dehors de leur classe) en les faisant précéder du nom de la classe les définissant.

Sur l'exemple de la figure 2.11, la méthode `creerC()`, fournit un nombre complexe à partir de deux réels `pReel` et `pImag`. Il n'y a pas d'objet implicite sur lequel porterait l'action. Les méthodes static ne doivent pas utiliser les attributs d'instance (les variables non static), ni les méthodes d'instance (non static) de l'objet. Elles utilisent leurs paramètres et les attributs static de la classe. De même, les opérations arithmétiques sur les complexes définies de façon `static`, réalisent une opération sur les deux paramètres `Complex z1` et `z2` et fournissent un résultat `Complex`. Aucun autre objet `Complex` implicite n'est mis en cause. Pour additionner deux `Complex`, on peut utiliser la méthode `plus()` ou la méthode `static plus()`. L'appel se fait de manière différente. Les méthodes `static`, comme les attributs `static`, sont précédées du nom de leur classe (exemple : **`Complex`**`.plus()`).

```
Complex c1 = new Complex (2, 1.50);
Complex c2 = new Complex (-2, 1.75);
Complex c3 = c1.plus (c2);                  // c1 est l'objet implicite

Complex c3 = Complex.plus (c1, c2);            // pas d'objet implicite
```

La figure 2.11 définit une méthode static de création d'un Complex et quatre méthodes static réalisant addition, soustraction, multiplication et division. On utilise aussi généralement une méthode static quand on accède ou modifie un attribut static. La méthode setNbDecimales() modifie le nombre de décimales voulues pour l'écriture des résultats. Ce nombre de décimales est fourni par la variable static nbDecimales (non attachée à un objet). La modification se fait grâce à la méthode static suivante :

```
static void setNbDecimales (int n);
```

L'appel de la fonction en dehors de la classe Complex se fait en préfixant la méthode du nom de la classe (la méthode setNbDecimales() de la classe Complex) :

```
Complex.setNbDecimales (3);
```

La méthode static nDecimal() tronque un réel double de façon à garder seulement nbDecimales. Cette méthode est privée donc locale à la classe Complex.

2.4.4 La classe Java Complex

```
// Complex.java  opérations de base
//               sur les nombres complexes

package mdpaquetage.complex;                    // voir paquetage page 65
```

Les méthodes suivantes construisent un objet de la classe Complex.

```
public class Complex {
  // les constantes de classes
  public static final double PI      = Math.PI;
  public static final int    RADIANS = 1;
  public static final int    DEGRES  = 2;
  // la variable de classe (attribut static)
  private static int nbDecimales = 3;  // nombre de décimales de 0 à 10

  // les attributs d'instance (les champs) de l'objet
  private double pReel;  // partie Réelle
  private double pImag;  // partie Imaginaire

  // constructeur d'un objet Complex
  // à partir de pReel (partie réelle) et pImag (partie imaginaire)
  public Complex (double pReel, double pImag) {
    this.pReel = pReel;                    // voir rôle de this page 38
    this.pImag = pImag;
  }
```

Un constructeur peut appeler un autre constructeur ayant un jeu de paramètres différents. Ainsi, **this** (0, 0) fait appel au constructeur de cette classe Complex ayant deux paramètres, soit le constructeur **Complex** (double pReel, double pImag) défini ci-dessus.

```
  // constructeur sans paramètre
  public Complex () {
    this (0, 0) ;           // appel d'un autre constructeur de Complex
  }
```

```
// constructeur à partir des composantes en polaire;
// argument en radians (RADIANS) ou en degrés (DEGRES) suivant typ
public Complex (double module, double argument, int typ) {
  if (typ == RADIANS) {
    pReel = module * Math.cos (argument);
    pImag = module * Math.sin (argument);
  } else {                                                  // en degrés
    pReel = module * Math.cos ((argument/180)*PI);
    pImag = module * Math.sin ((argument/180)*PI);
  }
}

// création d'un Complex
// à partir de pReel (partie réelle) et pImag (partie imaginaire)
public static Complex creerC (double pReel, double pImag) {
  return new Complex (pReel, pImag);
}
```

Les méthodes suivantes fournissent des informations sur un nombre Complex. On peut connaître la partie réelle, la partie imaginaire, le module ou l'argument d'un nombre Complex. Ces informations sont des réels (double).

```
// partie Réelle d'un Complex
public double partieRC () {
  return pReel;
}

// partie Imaginaire d'un Complex
public double partieIC () {
  return pImag;
}

// module d'un nombre Complex
public double module () {     // sqrt (square root) du paquetage lang
  return java.lang.Math.sqrt (pReel*pReel + pImag*pImag);
}

// argument d'un nombre Complex
public double argument () {    // atan2 : arc de tangente pImag/pReel
  return java.lang.Math.atan2 (pImag, pReel);
}
```

Les méthodes suivantes fournissent un nouveau nombre Complex calculé à partir de l'objet Complex courant. On peut créer l'opposé, le conjugué ou l'inverse d'un Complex.

```
// opposé d'un nombre Complex
public Complex oppose () {
  return new Complex (-pReel, -pImag);
}

// conjugué d'un nombre Complex
public Complex conjugue () {
  return new Complex (pReel, -pImag);
}
```

```
// inverse d'un nombre Complex
public Complex inverse () {
  return new Complex (1/module(), -argument(), RADIANS);
}
```

On peut réaliser des opérations arithmétiques sur les Complex : additionner deux Complex, les soustraire, les multiplier ou les diviser. On peut également multiplier un Complex plusieurs fois par lui-même, ce qui définit la puissance entière d'un nombre complexe.

```
// addition z1.plus(z)
public Complex plus (Complex z) {
  return new Complex (pReel + z.pReel, pImag + z.pImag);
}

// addition z = z1 + z2 de deux Complex (méthode static)
public static Complex plus (Complex z1, Complex z2) {
  return new Complex (z1.pReel + z2.pReel, z1.pImag + z2.pImag);
}

// soustraction z1.moins(z)
public Complex moins (Complex z) {
  return new Complex (pReel - z.pReel, pImag - z.pImag);
}

// soustraction  z = z1 - z2
public static Complex moins (Complex z1, Complex z2) {
  return new Complex (z1.pReel - z2.pReel, z1.pImag - z2.pImag);
}

// multiplication z1.multiplier(z)
public Complex multiplier (Complex z) {
  return new Complex (module() * z.module(),
                      argument() + z.argument(), RADIANS);
}

// multiplication z = z1 * z2
public static Complex multiplier (Complex z1, Complex z2) {
  return new Complex ( z1.module() * z2.module(),
                       z1.argument() + z2.argument(), RADIANS);
}

// division z1.diviser(z)
public Complex diviser (Complex z) {
  return this.multiplier (z.inverse());
}

// division z = z1 / z2
public static Complex diviser (Complex z1, Complex z2) {
  return multiplier (z1, z2.inverse());
}

// puissance nième (n entier >= 0) d'un Complex
```

```
public Complex puissance (int n) {
  Complex p = new Complex (1.0, 0.0);
  for (int i=1; i <= n; i++) p = multiplier (p, this);
                                                    // voir page 38
  return p;
}
```

Les méthodes suivantes écrivent les valeurs du nombre complexe sous forme cartésienne (parties réelle et imaginaire) et/ou sous forme polaire (module et argument). La méthode toString() (voir String page 80) fournit une chaîne de caractères sous les deux formes cartésienne et polaire. La chaîne de caractères fournie pour le nombre Complex de module 1 et d'argument PI/4 serait par exemple :

```
(0.707 + 0.707 i)    (1.0, 0.785)                    PI/4 = 0.785
```

La méthode ecritC() écrit la forme cartésienne ; la méthode ecritCP() écrit les coordonnées polaires et la méthode ecrit() écrit les deux formes. La méthode nDecimal() est expliquée dans le paragraphe suivant.

```
// fournit une chaîne de caractères comprenant les 2 formes
// cartésienne et polaire
public String toString () {                         // voir page 82
  return "(" + nDecimal (pReel)      + " + "
             + nDecimal (pImag)      + " i)"
    + "   (" + nDecimal (module())   + ", "
             + nDecimal (argument()) + ") ";
}

// écriture d'un nombre complexe : parties réelle et imaginaire
public void ecritC () {
  System.out.print (
    "(" + nDecimal (pReel) + " + " + nDecimal (pImag) + " i)"
  );
}

// écriture en polaire d'un nombre complexe : module et argument
public void ecritCP () {
  System.out.println (
    " (" + nDecimal (module())   + ", "
         + nDecimal (argument()) + ") "
  );
}

public void ecrit () {
  ecritC ();
  ecritCP();
}
```

Il n'existe pas sous Java de formatage simple des sorties. La méthode setNbDecimales (int n) fixe pour les variables de type double, le nombre de décimales à fournir. La méthode nDecimal (double d) tronque le nombre réel de façon à ne garder que n décimales.

```
  // modifie le nombre de décimales pour les résultats
  // n de 0 à 10; au-delà de 10, tous les chiffres sont affichés
  public static void setNbDecimales (int n) {
    if (n >= 0 && n <= 10) {
      nbDecimales = n;
    } else {
      nbDecimales = 15;  // toutes les décimales
    }
  }

  // modifie le double de facon à le tronquer si nbDecimales <= 10
  private static double nDecimal (double d) {
    if (nbDecimales <= 10) {
      long temp = (long) Math.pow (10, nbDecimales);
      return ((long) (d*temp)) / ((double)temp);
    } else {
      return d;
    }
  }

}  // class Complex
```

2.4.5 La mise en œuvre de la classe Complex

Disposant de la classe Complex, il est facile de créer et de réaliser des opérations arithmétiques sur ces nombres. Il s'agit simplement de faire appels aux constructeurs et aux méthodes de l'objet. Les appels des méthodes statiques sont donnés en commentaires pour bien mettre en évidence la façon de les appeler. Ci-après, les nombres complexes créés avec les méthodes *static* ou avec les méthodes d'instance sont identiques (cas de c3, c4, c5, c6 et de p3, p4, p5, p6).

```
// PPComplex.java  Programme Principal des Complex

import mdpaquetage.complex.*;                //(voir paquetage page 65)
                                    // définissant la classe Complex

public class PPComplex {              // Programme Principal des Complex

  public static void main (String[] args) {

    //Complex.setNbDecimales (2);                          // 3 par défaut

    // Coordonnées cartésiennes
    Complex c1 = new Complex (2,  1.50);
    Complex c2 = new Complex (-2, 1.75);
    System.out.print ("\nc1 =         " + c1);
    System.out.print ("\nc2 =         " + c2);

    /*
    //Méthodes statiques
    Complex c3 = Complex.plus       (c1, c2);
    Complex c4 = Complex.moins      (c1, c2);
    Complex c5 = Complex.multiplier (c1, c2);
```

```
  Complex c6 = Complex.diviser    (c1, c2);
  */

  Complex c3 = c1.plus       (c2);
  Complex c4 = c1.moins      (c2);
  Complex c5 = c1.multiplier (c2);
  Complex c6 = c1.diviser    (c2);

  Complex c7 = c1.puissance (3);

  System.out.print (
    "\nc3 = c1 + c2 " + c3 +
    "\nc4 = c1 - c2 " + c4 +
    "\nc5 = c1 * c2 " + c5 +
    "\nc6 = c1 / c2 " + c6 +
    "\nc7 = c1 ** 3 " + c7
  );


  // Coordonnées polaires
  System.out.print ("\n\n");
  Complex p1 = new Complex (1, Complex.PI/4, Complex.RADIANS);
//Complex p2 = new Complex (1, Complex.PI/2, Complex.RADIANS);
  Complex p2 = new Complex (1, 90, Complex.DEGRES);
  System.out.print ("\np1 =         " + p1);
  System.out.print ("\np2 =         " + p2);

  /*
  //Méthodes statiques
  Complex p3 = Complex.plus       (p1, p2);
  Complex p4 = Complex.moins      (p1, p2);
  Complex p5 = Complex.multiplier (p1, p2);
  Complex p6 = Complex.diviser    (p1, p2);
  */

  Complex p3 = p1.plus       (p2);
  Complex p4 = p1.moins      (p2);
  Complex p5 = p1.multiplier (p2);
  Complex p6 = p1.diviser    (p2);

  Complex p7 = p1.puissance (3);

  System.out.print (
    "\np3 = p1 + p2 " + p3 +
    "\np4 = p1 - p2 " + p4 +
    "\np5 = p1 * p2 " + p5 +
    "\np6 = p1 / p2 " + p6 +
    "\np7 = p1 ** 3 " + p7
  );

  System.out.print ("\n" +
    "\nPartie réelle de p1     : " + p1.partieRC() +
    "\nPartie imaginaire de p1 : " + p1.partieIC() +
    "\nModule de p1            : " + p1.module()   +
```

```
        "\nArgument de p1          : " + p1.argument() + "\n"
      );

      System.out.print ("\nCoordonnées polaires p1 : ");
      p1.ecritCP();

    }  // main

  }  // class PPComplex
```

Exemples de résultats d'exécution de PPComplex :

```
c1 =         (2.0    +  1.5   i)  (2.5,    0.643)
c2 =         (-2.0   +  1.75  i)  (2.657,  2.422)
c3 = c1 + c2 (0.0    +  3.25  i)  (3.25,   1.57)
c4 = c1 - c2 (4.0    + -0.25  i)  (4.007, -0.062)
c5 = c1 * c2 (-6.625 +  0.5   i)  (6.643,  3.066)
c6 = c1 / c2 (-0.194 + -0.92  i)  (0.94,  -1.779)
c7 = c1 ** 3 (-5.5   + 14.625 i)  (15.625, 1.93)

p1 =         (0.707  +  0.707 i)  (1.0,    0.785)         PI/4 = 45°
p2 =         (0.0    +  1.0   i)  (1.0,    1.57)          PI/2 = 90°
p3 = p1 + p2 (0.707  +  1.707 i)  (1.847,  1.178)
p4 = p1 - p2 (0.707  + -0.292 i)  (0.765, -0.392)
p5 = p1 * p2 (-0.707 +  0.707 i)  (1.0,    2.356)
p6 = p1 / p2 (0.707  + -0.707 i)  (1.0,   -0.785)
p7 = p1 ** 3 (-0.707 +  0.707 i)  (1.0,    2.356)

Partie réelle de p1     : 0.7071067812024209
Partie imaginaire de p1 : 0.7071067811706742
Module de p1            : 1.0
Argument de p1          : 0.785398163375

Coordonnées polaires p1 : (1.0, 0.785)
```

Remarque : la figure 5.45 page 231 donne une représentation graphique des nombres complexes.

2.5 L'IMPLÉMENTATION DES RÉFÉRENCES

Chaque objet en Java est **toujours** repéré par une **référence**. Il s'agit en fait de son adresse en mémoire. En C ou C++, on parlerait de pointeur sur l'objet, et il faudrait deux notations pour distinguer l'objet et l'adresse (pointeur) sur l'objet. En Java, le problème ne se pose pas ; on a une seule notation pour une référence sur l'objet.

2.5.1 La classe Personne (référençant elle-même deux objets Personne)

La classe Personne contient quatre attributs : deux objets nom et prénom de type String, et deux objets de type Personne. On définit deux constructeurs (voir constructeur

page 35). Le premier initialise les attributs nom, prenom, pere et mere. Le deuxième constructeur initialise nom et prenom à l'aide des paramètres, et pere et mere par défaut à null. La méthode toString() fournit une chaîne comportant le nom et le prénom de la Personne.

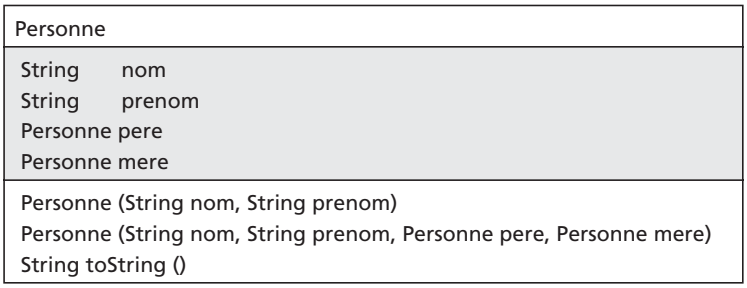


Figure 2.12 — La classe Personne (référençant deux objets Personne pere et mere).

D'où le programme Java :

```
// PPPersonne1.java  Programme Principal Personne numéro 1
//                   référençant deux objets Personne (pere et mere)

class Personne {
  String   nom;                     // les quatre attributs de Personne
  String   prenom;
  Personne pere;      // Personne contient deux références sur Personne
  Personne mere;         // un attribut référence est initialisé à null
                           // voir initialisation d'un attribut page 36

  Personne (String nom, String prenom, Personne pere, Personne mere) {
    this.nom    = nom;                  // voir le rôle de this page 38
    this.prenom = prenom;
    this.pere   = pere;
    this.mere   = mere;
  }

  // les références de pere et mere sont à null par défaut
  Personne (String nom, String prenom) {
    // appel du constructeur défini ci-dessus avec 4 paramètres
    this (nom, prenom, null, null);              // voir this() page 50
  }

  // retourne une chaîne de caractères formée avec le nom et le prénom
  public String toString () {                          // voir page 82
    return nom + " " + prenom;
  }

} // Personne
```

La classe ci-dessous définit deux instances `berthe` et `jacques` de `Personne`.
`berthe` est la mère de `jacques`.
`jacques` et `berthe` sont des variables locales à la fonction (méthode) main().

```
class PPPersonne1 {                    // Programme Principal Personne 1
  public static void main (String[] args) {
    Personne berthe  = new Personne ("Dupont", "Berthe");
    Personne jacques = new Personne ("Durand ", "Jacques",
                                                      null, berthe);
    System.out.print (
      "\njacques         : "              + jacques +
      "\nmère de jacques : "              + jacques.mere +
      "\nprénom de la mère de jacques : " + jacques.mere.prenom
    );
  }
} // class PPPersonne1
```

Exemple de résultats d'exécution :

```
jacques         : Durand  Jacques
mère de jacques : Dupont Berthe
prénom de la mère de jacques : Berthe
```

2.5.2 L'implémentation des références

En Java, les objets sont toujours repérés par référence c'est-à-dire par leur adresse en mémoire.

`Personne jacques;` est une référence non initialisée sur un futur objet Personne (voir figure 2.13).

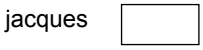


Figure 2.13 — L'implémentation de Personne jacques.

`Personne jacques = new Personne ("Durand", "Jacques");` entraîne la création d'un objet de type `Personne` référencé par `jacques`.

`jacques.nom` est une référence sur un objet de type `String` initialisé avec "Durand".

`jacques.prenom` est une référence sur l'objet de type `String` initialisé avec `"Jacques"`.

`jacques.pere` est une référence null (notée '/' sur la figure 2.14), de même que `jacques.mere`.

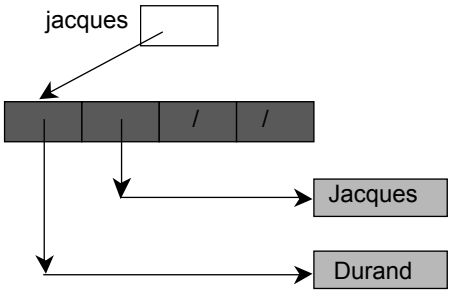


Figure 2.14 — L'implémentation de Personne jacques = new Personne ("Durand", "Jacques").

La figure 2.15 représente l'implémentation des deux instructions Java suivantes :

```
Personne berthe  = new Personne ("Dupont", "Berthe");
Personne jacques = new Personne ("Durand ", "Jacques", null, berthe);
```

Toutes les flèches sont des références sur des objets. `jacques.mere` est une référence sur l'objet `mere` de l'objet référencé par `jacques`. En raccourci, c'est l'objet `mere` de l'objet `jacques`.

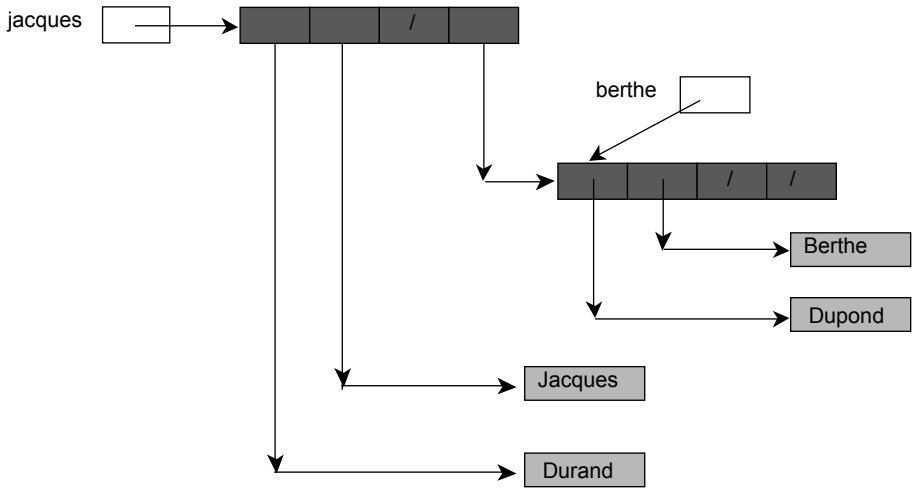


Figure 2.15 — L'implémentation des références en Java
(deux objets de type Personne, quatre objets de type String).

2.5.3 L'affectation des références, l'implémentation des tableaux d'objets

Ce deuxième exemple permet de construire des références sur des personnes comme schématisé sur la figure 2.16. Le père de jacques référence la Personne jules ; la mère de jacques référence la personne berthe. Toutes les flèches de la figure schématisent des références.

```
// PPPersonne2.java  Programme Principal Personne numéro 2

class Personne {
  idem Exemple 1                          voir classe Personne page 57
} // Personne

class PPPersonne2 {                    //Programme Principal Personne 2

  public static void main (String[] args) {
    Personne jules   = new Personne ("Durand", "Jules");
    Personne berthe  = new Personne ("Dupond", "Berthe");
    Personne jacques = new Personne ("Durand", "Jacques",
                                                      jules, berthe);

    System.out.print (
      "\nNom  de berthe   : " + berthe.nom   +
```

```
        "\nNom  de jacques  : " + jacques.nom  +
        "\nPère de jacques  : " + jacques.pere +
        "\nMère de jacques  : " + jacques.mere
    );
```

Voir la figure 2.16 suite à la création de ces trois objets. Voir aussi les résultats d'impression en fin de ce paragraphe.

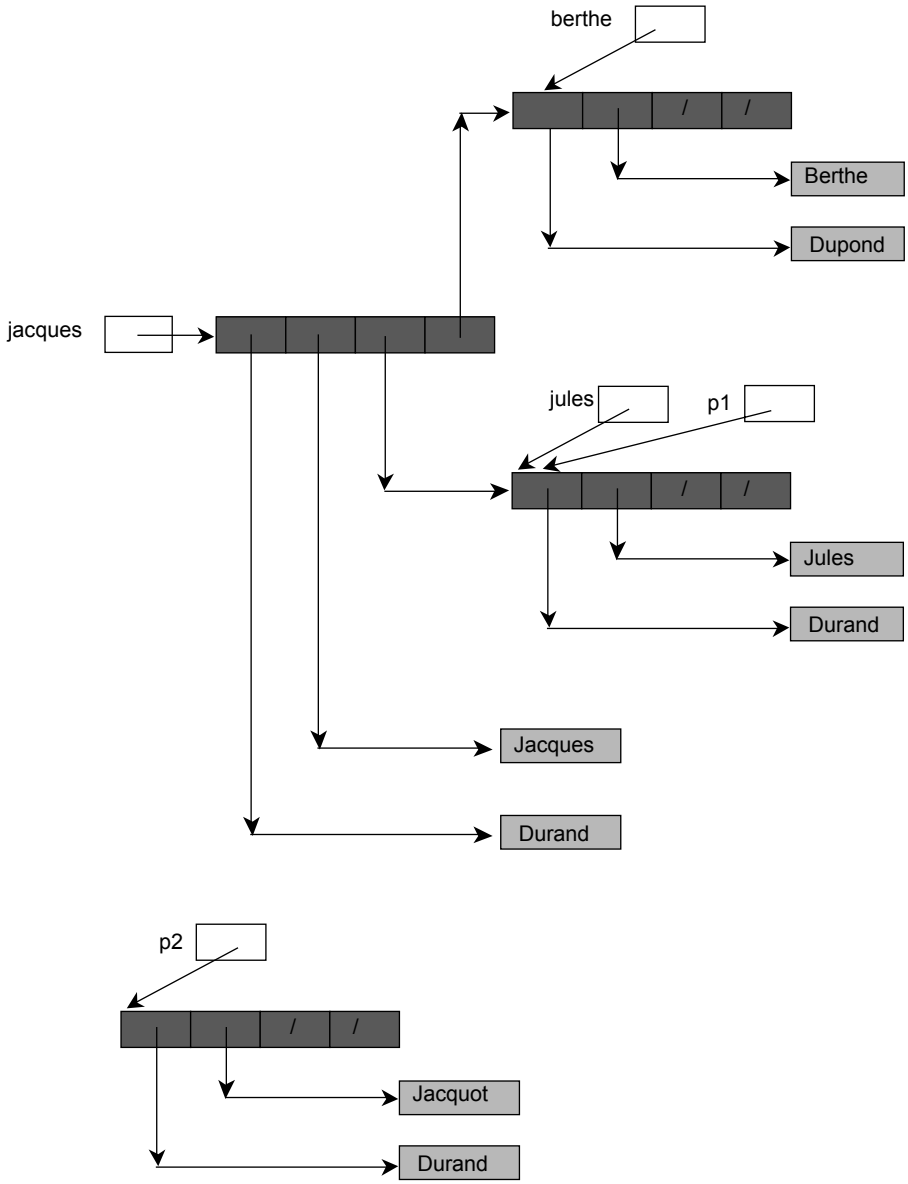


Figure 2.16 — La différence entre les instructions : Personne p1 = jules ; et Personne p2 = new Personne (jacques.nom, "Jacquot").

Les instructions suivantes illustrent l'affectation de références (voir figure 2.16) :

Personne jules = new Personne ("Durand", "Jules") ;

Personne p1 = jules ;

p1 et jules référencent la même personne (le même objet) ; la modification de jules.prenom est mise en évidence en écrivant p1.prenom.

```
// Deux références p1 et jules sur un même objet
Personne p1  = jules;      // p1 pointe sur le même objet que jules
jules.prenom = "Julot";
System.out.print (
  "\nPersonne p1      : " + p1 +
  "\nPersonne jules   : " + jules      // voir résultats ci-dessous
);
```

Mais, par la suite, on crée un nouvel objet Personne à partir du nom de jacques et du prénom "Jacquot". Les deux objets jacques et p2 sont deux objets différents ayant chacun leurs données comme le montre l'écriture des deux objets.

```
// Deux objets p2 et jacques différents
Personne p2 = new Personne (jacques.nom, "Jacquot");
System.out.print (
  "\nPersonne p2      : " + p2 +
  "\nPersonne jacques : " + jacques    // voir résultats ci-dessous
);
```

La création d'un tableau de Personne se fait en réservant un tableau de références nulles (voir figure 2.17), en créant ensuite chacun des objets et en mémorisant leur référence.

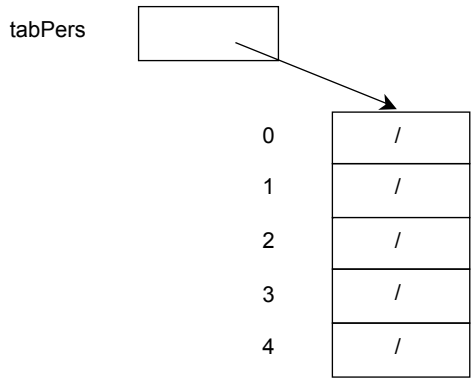


Figure 2.17 — La déclaration Personne[] tabPers = new Personne[5]; les références sur les Personne sont nulles (null représenté par / sur la figure).

```
// Tableau de personnes de 5 Personne
Personne[] tabPers = new Personne[5];

tabPers [0] = new Personne ("nom0", "prenom0");
tabPers [1] = new Personne ("nom1", "prenom1");
```

```
        tabPers [2] = new Personne ("nom2", "prenom2");
        tabPers [3] = new Personne ("nom3", "prenom3");
        tabPers [4] = new Personne ("nom4", "prenom4");

        Personne p3 = tabPers[4];
        System.out.println ("\nPersonne p3      : " + p3);
    }  // main

} //class PPPersonne2
```

La figure 2.18 schématise le tableau tabPers de cinq objets Personne créé dans le programme précédent.

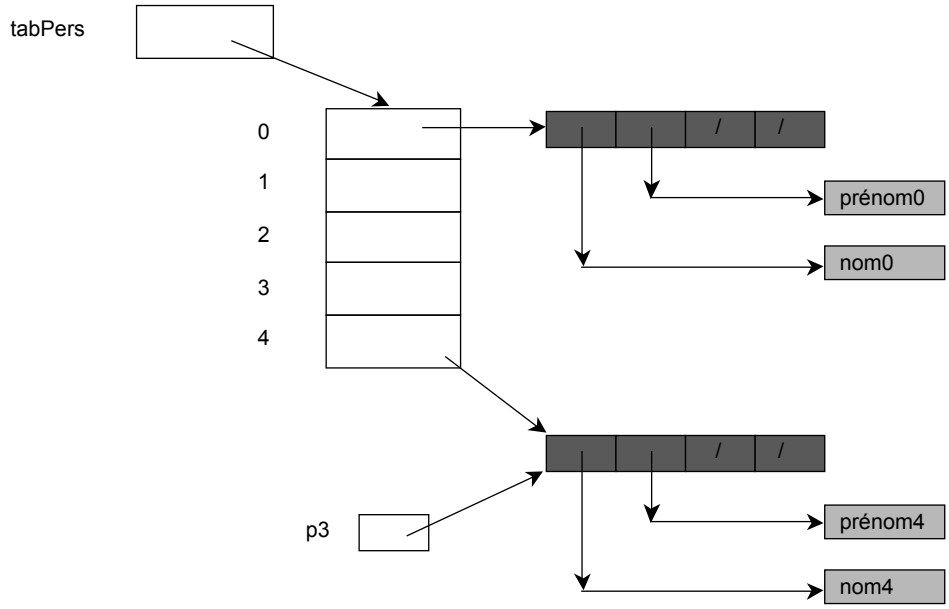


Figure 2.18 — Un tableau d'objets Personne.
Les objets 1, 2 et 3 ne sont pas représentés.

Exemple de résultats d'exécution :

```
Nom  de berthe   : Dupond
Nom  de jacques  : Durand
Père de jacques  : Durand Jules
Mère de jacques  : Dupond Berthe

Personne p1      : Durand Julot                  Personne p1 = jules
Personne jules   : Durand Julot

Personne p2      : Durand Jacquot    Personne p2 = new Personne(...)
Personne jacques : Durand Jacques

Personne p3      : nom4 prenom4              Personne p3 = tabPers[4]
```

2.5.4 La désallocation de la mémoire (ramasse-miettes, garbage collector)

Dans les exemples précédents, de nombreux objets ont été créés à l'aide de new(). En Java, la libération (la désallocation) de l'espace mémoire alloué et devenu inutile n'est pas à la charge de l'utilisateur. Le système sait détecter les objets devenus inutiles car ils ne sont plus référencés ; ils sont devenus inaccessibles. La place mémoire est automatiquement récupérée par un processus appelé ramasse-miettes ou en anglais *garbage collector*.

2.6 EXEMPLE : LA CLASSE DATE

La classe `Date` gère les dates. Les attributs `jour`, `mois` et `an` sont mémorisés comme des entiers. On définit les méthodes suivantes :

Date (int j, int m, int an) : constructeur créant un objet Date.
String **toString** () : chaîne de caractères correspondant à la date de l'objet.
boolean **bissex** () : fournit true si l'année est bissextile, false sinon.
int **nbJoursEcoules** () : nombre de jours écoulés depuis le début de l'année.
int **nbJoursRestants** () : nombre de jours restant dans l'année.

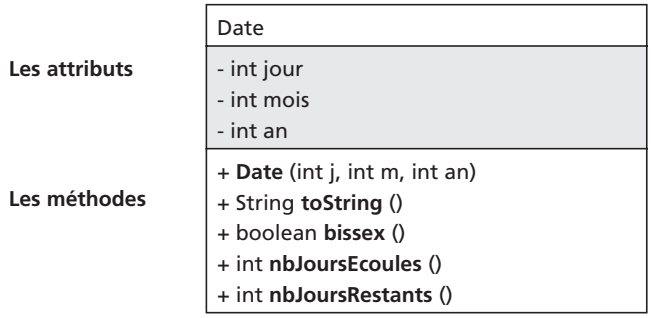


Figure 2.19 — La classe Date : ses attributs privés et ses méthodes *public* d'instance.

On définit, en plus, les méthodes `static` suivantes :
static boolean **bissex** (int annee) : fournit true si l'année est bissextile, false sinon.
static long **nbJoursEntre** (Date date1, Date date2) : fournit le nombre de jours écoulés entre les deux dates date1 et date2.

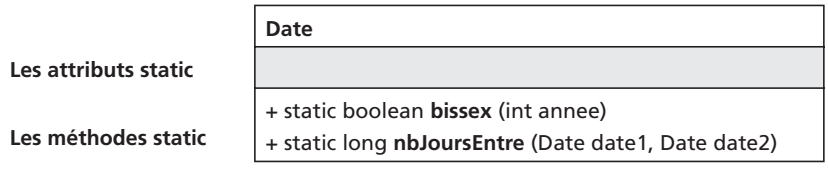


Figure 2.20 — Les méthodes static de la classe Date.

Les attributs (données) de la classe sont privés. Exemple d'utilisation de la classe **Date** :

```
class PPDate {                                    Programme Principal Date
  public static void main (String[] args) {
    System.out.print (                                  // méthodes static
      "\nBissextile 1900 : " + Date.bissex (1900) +
      "\nBissextile 1920 : " + Date.bissex (1920) +
      "\nBissextile 1989 : " + Date.bissex (1989) +
      "\nBissextile 2000 : " + Date.bissex (2000)
    );

    Date d0 = new Date (3, 2, 2003);
    System.out.print ("\nBissextile 2003 : " + d0.bissex());
    System.out.println ("\n\nAu " + d0 +
      "\n  nombre de jours écoulés  : " + d0.nbJoursEcoules() +
      "\n  nombre de jours restants : " + d0.nbJoursRestants()
    );

    Date d1 = new Date (1, 1, 1900);
    Date d2 = new Date (1, 1, 2000);
    System.out.println (
      "\nNombre de jours écoulés entre" +
      "\n  le    " + d1 +
      "\n  et le " + d2 +
      "\n  = " + Date.nbJoursEntre (d1, d2)           // méthode static
    );
  }  // main

}  // class PPDate
```

Exemple de résultats d'exécution :

```
Bissextile 1900 : false
Bissextile 1920 : true
Bissextile 1989 : false
Bissextile 2000 : true
Bissextile 2003 : false

Au  3/2/2003
  nombre de jours écoulés  : 34
  nombre de jours restants : 331

Nombre de jours écoulés entre
  le     1/1/1900
  et le  1/1/2000
  = 36525
```

Remarque : il y a surcharge (voir page 45) de la méthode bissex() qui s'utilise pour une année (un entier) ou une Date. **Date.bissex** (1900) appelle la méthode static bissex() de la classe Date. d0.**bissex**() appelle la méthode d'instance bissex() pour la Date d0. Voir les attributs et méthodes static en 2.4.3, page 48.

Exercice 2.1 – La classe Date

Écrire en Java la classe Date décrite ci-dessus.

2.7 LES PAQUETAGES (*PACKAGES*)

La notion de paquetage (*package* en anglais) s'apparente à la notion de **bibliothèque de pr ogrammes**. Plusieurs classes sont regroupées logiquement dans un même paquetage (un même répertoire). L'accès aux classes et donc aux méthodes du paquetage (pour les classes extérieures à ce paquetage) se fait alors en indiquant explicitement ou implicitement le ou les paquetages à consulter. Si plusieurs méthodes portent le même nom, il ne peut y avoir conflit que pour un même paquetage. Si des méthodes portant le même nom sont défines dans deux paquetages différents, elles sont identifiées sans ambiguïté grâce au nom de paquetage. L'ensemble des paquetages peut être **hiérarchisé** sur plusieurs niveaux en insérant les paquetages dans des sous-paquetages (sous-répertoires du répertoire du paquetage initial). Toute la hiérarchie des paquetages peut être réunie dans un seul fichier (d'extension .zip ou .jar). Ainsi, la bibliothèque standard des classes Java est hiérarchisée en paquetage comme suit (seule une partie de la hiérarchie est mentionnée) :

hiérarchie	**les paquetages**	**les classes**
java		
awt	java.awt	*interfaces graphiques (boutons, fenêtres, etc.)*
event	java.awt.event	*événements (clic de souris, clavier, etc.)*
image	java.awt.image	*traitement d'images*
io	java.io	*input-output (entrées-sorties)*
lang	java.lang	*classes de base (String, Math, System, etc.)*
util	java.util	*utilitaires (Random, Vector, Calendar, etc.)*
applet	java.applet	*traitement des applets*
net	java.net	*accès au réseau (URL, Socket, etc.)*

Chaque paquetage contient une ou plusieurs classes et chaque classe définit une ou plusieurs méthodes. Ainsi, le paquetage java.lang définit la classe String permettant de gérer les chaînes de caractères grâce à un ensemble de méthodes de traitement de chaînes de caractères. Le paquetage java.lang contient également une classe Math concernant les opérations mathématiques (valeur absolue, maximum de deux nombres, sinus, cosinus, etc.).

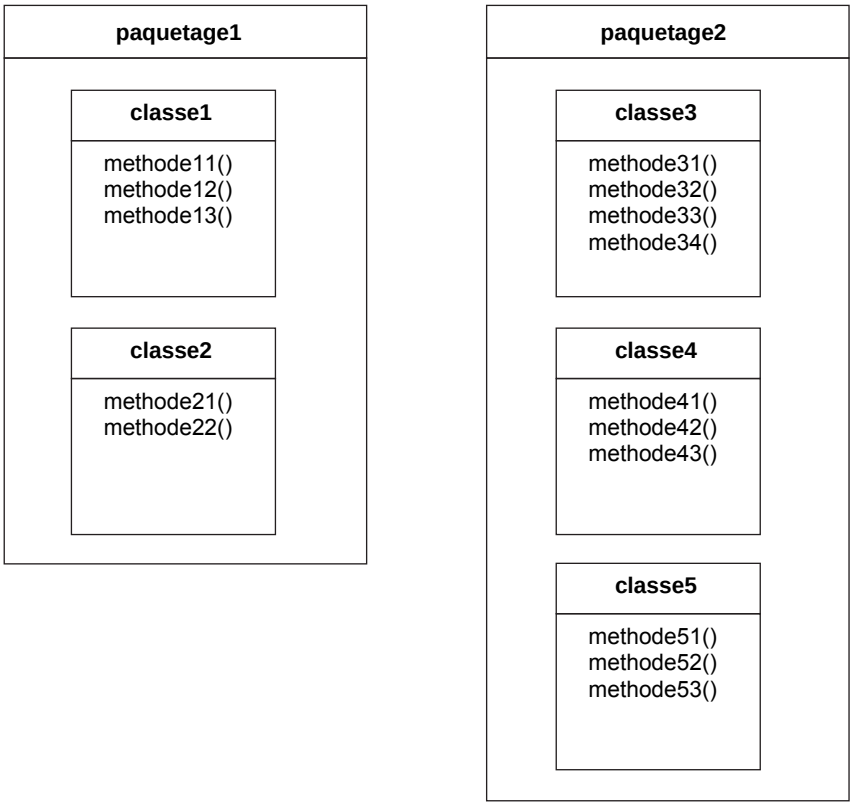


Figure 2.21 — Un paquetage contient une ou plusieurs classes. Chaque classe définit une ou plusieurs méthodes.

2.7.1 Le rangement d'une classe dans un paquetage

L'instruction **package** < nom_de_package > comme **première** instruction d'un programme Java indique que toutes les classes du programme (du fichier) sont insérées dans le paquetage mentionné. Ces classes seront accessibles pour tout programme Java référençant le paquetage. En l'absence d'instruction package, le compilateur considère que les classes du programme appartiennent à un paquetage (package) "par défaut". Page 50, "package mdpaquetage.complex" indique que la classe Complex (et donc toutes ses méthodes) est rangée dans le paquetage mdpaquetage.complex. Une option du compilateur permet de mentionner le répertoire racine du paquetage à créer. Voir page 371 en annnexe.

2.7.2 Le référencement d'une classe d'un paquetage

Pour référencer une classe d'un paquetage, il suffit de l'indiquer à l'aide d'une instruction `import` comme indiqué sur l'exemple ci-dessous où le programme importe (utilise) la classe classe1 de paquetage1.

```
import paquetage1.classe1;
```

On peut importer toutes les classes d'un paquetage, en écrivant :

```
import paquetage1.*;
```

Page 54, "import mdpaquetage.complex.*" indique que l'on utilise (importe) toutes les classes du paquetage complex. Importer une classe signifie qu'on peut créer des objets de cette classe ou utiliser des méthodes de la classe. On pourrait aussi ne pas importer la classe et préfixer chaque référence à la classe ou à une méthode de la classe par le nom du paquetage où elle se trouve. Le compilateur et l'interpréteur doivent connaître l'emplacement réel des répertoires racines des paquetages utilisés (rôle de la variable d'environnement `CLASSPATH`, ou d'une option du compilateur et de l'interpréteur, ou d'une option d'un menu déroulant d'un logiciel intégré). Le paquetage java.lang est importé d'office ; il n'y a pas à le mentionner. Voir page 371 en annnexe.

2.8 LES DROITS D'ACCÈS AUX ATTRIBUTS ET AUX MÉTHODES : PUBLIC, PRIVATE OU "DE PAQUETAGE"

En Java, il y a quatre types d'accès à un attribut ou à une méthode. Devant un attribut ou une méthode on peut déclarer : private, public ou ne rien déclarer, ce qui signifie un droit de paquetage. Un autre type d'accès (protected) est vu au chapitre suivant en relation avec le concept d'héritage. Voir page 94.

Un attribut d'instance **public** est accessible via un objet de toutes les méthodes de toutes les classes (celles de l'objet, celles du paquetage et celles des autres paquetages). La variable **d12** de Classe1 (voir figure 2.22) est accessible de toutes les méthodes. Exemple :

```
Classe1 c1 = new Classe1();
System.out.println (c1.d12);
```

Un attribut **private** peut être accédé uniquement par les méthodes de la classe. Seules les méthodes de Classe1 (methode11(), methode12() et methode13()) peuvent accéder à **d11**.

Si on ne déclare **rien**, l'attribut est accessible dans les méthodes de la classe et dans les méthodes des classes du paquetage. Ainsi, **d13** est accessible dans les méthodes de Classe1 (methode11(), methode12() et methode13()) et dans les méthodes de Classe2 (methode21() et methode22()) référençant un objet de Classe1 :

```
Classe1 c2 = new Classe1();
System.out.println (c2.d13);
```

Les deux instructions ci-dessus conduisent à une erreur dans une des méthodes de Classe3 ou de Classe4.

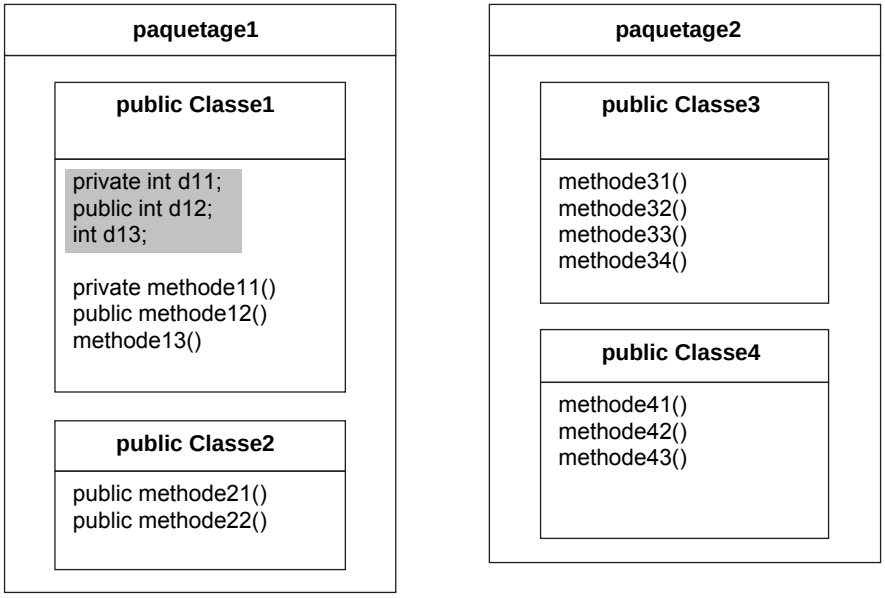


Figure 2.22 — Les droits d'accès private, public ou de paquetage.

Remarque : le principe est le même pour une méthode qui peut être privée (**private**) et utilisable seulement dans sa classe, **public** et utilisable dans toutes les méthodes de toutes les classes (à partir d'un objet de la classe pour les méthodes d'instance, ou en préfixant avec le nom de la classe pour les méthodes static de classe). **Par défaut**, la méthode est accessible dans les classes du paquetage ; methode13() est accessible dans le paquetage1. Si Classe1 n'est pas déclarée public, alors que ses méthodes sont déclarées public, ses méthodes ne seront néanmoins pas accessibles d'un autre paquetage.

2.8.1 Le paquetage paquetage1

La compilation de Classe1.java entraîne le rangement de Classe1.class dans le répertoire paquetage1 du répertoire mdpaquetage à partir du répertoire racine des paquetages (à définir à l'aide d'une option du compilateur).

```
// Classe1.java

package mdpaquetage.paquetage1 ;       // Classe1 élément de paquetage1

public class Classe1 {
  private int d11 = 1;                          // visible dans Classe1
  public  int d12 = 2;                               // visible partout
  int d13 = 3;                                 // visible dans paquetage1

  public String toString () {
    return "d11 " + d11 + ", d12 " + d12 + ", d13 " + d13;
  }
```

```
  private void methode11 () {                // accessible dans Classe1
    System.out.println ("methode11 " + toString());
  }

  public void methode12 () {                   // accessible de partout
                                               // si Classe1 est public
    System.out.println ("methode12 " + toString());
  }

  void methode13 () {                      // accessible dans paquetage1
    System.out.println ("methode13 " + toString());
  }

}  // class Classe1
```

De même pour la compilation de Classe2 qui est rangée dans le paquetage1.

```
// Classe2.java

package mdpaquetage.paquetage1 ;       // Classe2 élément de paquetage1

public class Classe2 {

  public void methode21 () {
    Classe1 c1 = new Classe1();

    //erreur d'accès :                                    d11 est privé
    //System.out.println ("c1.d11 "   + c1.d11);

    System.out.println ("c1.d12 " + c1.d12);             // d12 public
    System.out.println ("c1.d13 " + c1.d13);  // d13 dans le paquetage1

  //c1.methode11();                      // Erreur methode11 est privée
    c1.methode12();                          // methode12() est public
    c1.methode13();               // methode13() est dans le paquetage1
  }

}  // class Classe2
```

2.8.2 L'utilisation du paquetage paquetage1

La définition de Classe3 dans un autre paquetage (ou dans le paquetage par défaut si on n'indique pas d'instruction package) conduit aux accès suivants.

```
// Classe3.java

import mdpaquetage.paquetage1.*;                    // Classe1, Classe2

public class Classe3 {        // Classe3 dans un paquetage "par défaut"

  public void methode31 () {
    Classe1 c1 = new Classe1();

    // erreur d'accès : d11 est un membre privé
    //System.out.println ("c1.d11 " + c1.d11);

    System.out.println ("c1.d12 " + c1.d12);   // OK car d12 est public
    // erreur d'accès : d13 n'est pas défini dans le paquetage courant
    //System.out.println ("c1.d13 " + c1.d13);
```

```
    //c1.methode11();                    // erreur : methode11 est privée
      c1.methode12();          // OK : la méthode écrit les 3 valeurs de c1
    //c1.methode13();       // erreur methode13 est dans un autre paquetage
   }

    public static void main (String[] args) {
      new Classe3().methode31();
    }

  }  // class Classe3
```

Le résultat de l'exécution est donné ci-dessous. L'intérêt est surtout de savoir ce qu'on peut accéder. Les instructions provoquant une erreur de compilation sont mises en commentaires et commentées dans le programme Java précédent.

```
c1.d12 2
methode12 d11 1, d12 2, d13 3
```

2.9 L'AJOUT D'UNE CLASSE AU PAQUETAGE MDAWT

2.9.1 La classe Couleur (méthodes static)

L'utilisation d'une interface graphique nécessite souvent des changements de couleurs pour le texte ou pour le dessin. La classe Java Color définit des constantes *static* pour les couleurs de base (Color.red, Color.blue, etc.). La classe Couleur proposée ci-dessous définit des noms en français, attribue un numéro à une couleur ou fournit une couleur en fonction de son numéro. Cette classe est enregistrée dans le paquetage mdawt auquel d'autres classes viendront s'adjoindre par la suite. C'est une classe utilitaire pouvant être mise en œuvre dans diverses applications d'où l'intérêt de la mettre dans un paquetage. Les méthodes sont toutes `public static`. Le tableau ci-dessous définit les couleurs en anglais et en français.

0	Color.black	noir
1	Color.blue	bleu
2	Color.cyan	cyan
3	Color.darkGray	gris foncé
4	Color.gray	gris
5	Color.green	vert
6	Color.lightgray	gris clair
7	Color.magenta	magenta
8	Color.orange	orange
9	Color.pink	rose
10	Color.red	rouge
11	Color.white	blanc
12	Color.yellow	jaune

La classe **Couleur** définit l'attribut static nbCouleur et les méthodes suivantes :

- public static final int **nbCouleur** = 13 : attribut nombre de couleurs.
- public static Color **getColor** (int n) : la nième couleur de la classe Color.
- public static Color **getColor** (String nomCouleur) : la couleur de la classe Color pour un nom de couleur en français.
- public static int **numeroCouleur** (Color c) : le numéro de couleur de Color c.
- public static int **numeroCouleur** (String nomCouleur) : le numéro de couleur pour un nom de couleur en français.
- public static String **nomCouleur** (Color c) : le nom français de Color c.
- public static String **nomCouleur** (int n) : le nom français de la nième couleur.

2.9.2 Implémentation de la classe Couleur

Les tableaux des couleurs tabCA et tabCF sont privés, non modifiables et `static`. Ils existent indépendamment de tout objet de la classe Couleur.

```
// Couleur.java

package mdpaquetage.mdawt ; // classe Couleur rangée dans le paquetage mdawt

import java.awt.*;  // Color

public class Couleur {

  // tableau d'objets de la classe Color;
  // tableau des couleurs (anglais)
  private static final Color[] tabCA = {
    // 13 couleurs définies comme des constantes static (classe Color)
    Color.black,  Color.blue,   Color.cyan,      Color.darkGray,
    Color.gray,   Color.green,  Color.lightGray, Color.magenta,
    Color.orange, Color.pink,   Color.red,       Color.white,
    Color.yellow
  };

  // tableau des noms français équivalents des 13 couleurs
  // du tableau tabCA
  // tableau des couleurs (français)
  private static final String[] tabCF = {
    "noir",       "bleu",       "cyan",          "gris foncé",
    "gris",       "vert",       "gris clair",    "magenta",
    "orange",     "rose",       "rouge",         "blanc",
    "jaune"
  };

  public static final int nbCouleur = 13;         // nombre de couleurs

  // fournit l'objet de numéro numColor du tableau tabCA
  public static Color getColor (int numColor) {
    if (numColor < 0 || numColor >= nbCouleur) numColor = 0;
    return tabCA[numColor];
  }

  // fournit l'objet de type Color ayant le nom français nomCouleur
  // fournit Color.black si le nom n'existe pas dans tabCF
```

```
  public static Color getColor (String nomCouleur) {
    boolean trouve = false;
    int     i      = 0;
    while ( !trouve && (i < nbCouleur) ) {
      trouve = nomCouleur.equals (tabCF[i]);     // voir String page 80
      if (!trouve) i++;
    }
    if (!trouve) i = 0;  // Color.black
    return tabCA[i];
  }

  // fournit le numéro dans le tableau tabCA de l'objet Color c
  // fournit -1 si c n'existe pas
  public static int numeroCouleur (Color c) {
    boolean trouve = false;
    int     i      = 0;
    while ( !trouve && (i < nbCouleur) ) {
      trouve = c.equals (tabCA[i]);
      if (!trouve) i++;
    }
    if (!trouve) i = -1;
    return i;
  }

  // fournit le numéro dans le tableau tabCF de nomCouleur
  // fournit -1 si nomCouleur n'existe pas
  public static int numeroCouleur (String nomCouleur) {
    boolean trouve = false;
    int     i      = 0;
    while ( !trouve && (i < nbCouleur) ) {
      trouve = nomCouleur.equals (tabCF[i]);
      if (!trouve) i++;
    }
    if (!trouve) i = -1;
    return i;
  }

  // fournit le nom français de l'objet Color c
  public static String nomCouleur (Color c) {
    int n = numeroCouleur (c);
    if (n == -1) {
      return "inconnu";
    } else {
      return tabCF[n];
    }
  }

  // fournit le nom français de numCouleur
  public static String nomCouleur (int numCouleur) {
    if (numCouleur < 0 || numCouleur >= nbCouleur) {
      return "inconnu";
    } else {
      return tabCF[numCouleur];
    }
  }

}  // class Couleur
```

Remarque : Les méthodes getColor(), numeroCouleur() et nomCouleur() sont surchargées (voir page 46).

2.9.3 La mise en œuvre de la classe Couleur

Le programme suivant vérifie le bon fonctionnement des méthodes de la classe Couleur qui est utilisée dans les prochains chapitres. Les méthodes étant static, elles sont précédées du nom de la classe Couleur lors de leur appel en dehors de la classe Couleur (voir page 49).

```
// PPCouleur.java  Programme Principal Couleur

import java.awt.*;              // Color
import mdpaquetage.mdawt.*;     // Couleur

class PPCouleur {

  // méthode de test de la classe Couleur
  public static void main (String[] args) {
    System.out.println ("couleur " + Couleur.getColor ("noir") );
    System.out.println ("couleur " + Couleur.getColor (2) );
    System.out.println ("couleur " + Couleur.nomCouleur (Color.red) );

    for (int i=0; i < Couleur.nbCouleur; i++) {
      System.out.println ("couleur " + Couleur.nomCouleur(i) );
    }
    System.out.println ("couleur rouge, numéro : "
                        + Couleur.numeroCouleur ("rouge") );
  }

}  // class PPCouleur
```

Exemple de résultats d'exécution :

```
couleur java.awt.Color[r = 0,g = 0,b = 0]                    noir
couleur java.awt.Color[r = 0,g = 255,b = 255]                cyan
couleur rouge                                              Color.red
couleur noir                                                     0
couleur bleu                                                     1
couleur cyan
couleur gris foncé
couleur gris
couleur vert
couleur gris clair
couleur magenta
couleur orange
couleur rose
couleur rouge                                                   10
couleur blanc
couleur jaune                                                   12
couleur rouge, numéro : 10
```

2.10 LES EXCEPTIONS

Une exception est un événement exceptionnel risquant de compromettre le bon déroulement du programme. Un débordement de tableau ou de pile, la lecture d'une donnée erronée ou d'une fin de fichier prématurée constituent des exemples d'exceptions. Lors de la détection d'une anomalie, une fonction (en C par exemple) peut essayer de la traiter directement ; elle peut aussi fournir des informations de retour (par l'intermédiaire d'une variable) indiquant le type de l'anomalie et laissant au programme appelant le soin de décider de la suite à donner à cette anomalie.

En Java, la fonction peut lancer une exception en indiquant un type d'exception. L'exception se "propage" de retour de méthode en retour de méthode jusqu'à ce qu'une méthode la traite. Si aucune méthode ne la prend en compte, il y a un message d'erreur d'exécution. Suite à une exception, toutes les instructions sont ignorées, sauf les instructions chargées de capter cette exception.

2.10.1 Les exceptions de type RunTimeException

2.10.1.1 Exemple1 : les débordements de tableaux (dans main)

L'exécution du programme ci-dessus provoque un message d'erreur indiquant une exception du type ArrayIndexOutOfBoundsException, et l'arrêt du programme. Le tableau de 4 entiers tabEnt a des indices de 0 à 3. L'accès à la valeur d'indice 5 provoque une exception non captée.

```
// TestException1.java

public class TestException1 {

  public static void main (String[] args) {
    int tabEnt[] = { 1, 2, 3, 4 };

    // Exception de type java.lang.ArrayIndexOutOfBoundsException
    // non controlée par le programme; erreur d'exécution
    tabEnt [4] = 0;                                  // erreur et arrêt

    System.out.println ("Fin du main");    // le message n'apparaît pas
  } // main

}  // class TestException1
```

Dans le programme suivant, la même erreur d'accès à un élément est captée par le programme. Lors de la rencontre de l'erreur, les instructions en cours sont abandonnées et un bloc de traitement (catch) de l'exception est recherché et exécuté. Le traitement se poursuit après ce bloc sauf indication contraire du catch (return ou arrêt par exemple). Le message "Fin du main" est écrit après interception de l'exception.

```
public class TestException2 {

  public static void main (String[] args) {
    try {
      int tabEnt[] = { 1, 2, 3, 4 };
      // Exception de type ArrayIndexOutOfBoundsException
```

```
      // controlée par le programme
      tabEnt [4] = 0;                    // Exception et suite au catch
      System.out.println ("Fin du try");                // non exécutée
    } catch (Exception e) {
      System.out.println ("Exception : " + e);      // écrit le message
    }
    System.out.println ("Fin du main");                     // exécutée
  }

}  // class TestException2
```

Exemples de résultats d'exécution :

```
Exception : java.lang.ArrayIndexOutOfBoundsException
Fin du main
```

2.10.1.2 Exemple2 : les débordements de tableaux (dans une méthode)

Ci-dessous, l'instruction tabEnt[4] provoque une exception dans la méthode initTableau(). L'instruction qui suit n'est pas prise en compte ; l'exception n'est pas traitée dans initTableau(). Les instructions qui suivent l'appel de initTableau() dans main() (le programme appelant) sont ignorées également. Par contre, il existe un bloc catch qui traite l'exception. Le message "Fin de main" est lui pris en compte après le traitement de l'exception.

```
// PPTestException3.java  Exception lancée dans initTableau,
//                                captée dans main()

class TestException3 {
  int[] tabEnt = new int[4];

  void initTableau () {
    tabEnt[0] = 1;  tabEnt[1] = 2;
    tabEnt[2] = 3;  tabEnt[3] = 4;
    tabEnt[4] = 5;            // l'exception se produit ici; non captée
    System.out.println("Fin de initTableau");          // non exécutée
  }

  void ecrireTableau () {
    for (int i=0; i < tabEnt.length; i++) {
      System.out.print (" " + i);
    }
    System.out.println();
  }

} // class TestException3

public class PPTestException3 {
  public static void main (String[] args) {
    try {
      TestException3 te = new TestException3();
      te.initTableau();      // l'exception se produit dans initTableau
```

```
      te.ecrireTableau();                                  // non exécutée
    } catch (Exception e) {                          // exception captée ici
      System.out.println ("Exception : " + e);              // exécutée
    }
    System.out.println ("Fin de main");                    // exécutée
  }  // main

}  // class PPTestException3
```

Résultats d'exécution :

```
Exception : java.lang.ArrayIndexOutOfBoundsException
Fin de main
```

L'exception peut aussi être traitée dans initTableau() comme indiqué ci-dessous. Le programme se poursuit normalement une fois le bloc catch exécuté (retour de initTableau(), appel de ecrireTableau() et message de fin).

```
// PPTestException4.java  Exception lancée par initTableau()
//                                captée par initTableau()
class TestException4 {
  int[] tabEnt = new int[4];

  void initTableau () {
    try {
    tabEnt[0] = 1;  tabEnt[1] = 2;
    tabEnt[2] = 3;  tabEnt[3] = 4;
    tabEnt[4] = 5;                        // l'exception se produit ici
    } catch (Exception e) {                                 // captée ici
      System.out.println ("Exception : " + e);      // écrit le message
    }
    System.out.println("Fin de initTableau");               // exécutée
  }

  void ecrireTableau () {
    for (int i=0; i < tabEnt.length; i++) {
      System.out.print (" " + tabEnt[i]);
    }
    System.out.println();
  }

} // class TestException4

public class PPTestException4 {
  public static void main (String[] args) {
    TestException4 te = new TestException4();
    te.initTableau();                       // l'exception a été captée
    te.ecrireTableau();                                     // exécutée
    System.out.println ("Fin de main");                    // exécutée
  } // main

}  // class PPTestException4
```

Exemples de résultats d'exécution :

```
Exception : java.lang.ArrayIndexOutOfBoundsException
Fin de initTableau
 1 2 3 4                                                    ecrireTableau
Fin de main
```

2.10.1.3 Autres exemples d'exceptions de type RunTimeException

Les deux instructions suivantes provoquent une exception de type NullPointerException. s1 est une référence `null` sur un (futur) objet de type String. On ne peut pas accéder au troisième caractère de l'objet String référencé par `null`, d'où l'exception.

```
String s1 = null;  System.out.println (s1.charAt (3));
```

Le programme suivant donne des exemples où une exception est lancée : chaîne à convertir en entier ne contenant pas un entier (lecture erronée d'un entier par exemple), division par zéro, transtypage non cohérent de deux objets de classes différentes, essai d'ouverture d'un fichier n'existant pas (voir chapitre 7).

```
// TestException5.java  différentes exceptions

import java.awt.*;

public class TestException5 {
  public static void main (String[] args) {
    String s1 = null;
    //exception NullPointerException ci-dessous
    //System.out.println (s1.charAt(3));

    // convertit la chaîne "231" en un entier binaire
    int n1 = Integer.parseInt ("231");
    System.out.println ("n1 : " + n1);
    // exception car la chaîne "231x" n'est pas un entier :
    // exception NumberFormatException ci-dessous
    //int n2 = Integer.parseInt ("231x");
    //System.out.println ("n2 : " + n2);

    //int d = 0;
    //int quotient = 17 / d;   // exception de type ArithmeticException
    //System.out.println ("quotient : " + quotient);

    //Object p = new Point();        // voir héritage, chapitre suivant
    //Color coul = (Color) p;         // exception ClassCastException :
                                      // p n'est pas de la classe Color

    System.out.println ("Fin de main");
  } // main

}  // class TestException5
```

2.10.2 La définition et le lancement d'exceptions utilisateurs (throw et throws)

On peut reprendre l'exemple de la classe Pile (page 31) et remplacer par des exceptions les messages d'erreurs correspondant au dépilement d'une pile vide et à l'insertion dans une pile saturée. La nouvelle méthode erreur(), au lieu d'écrire un message, crée et lance une Exception contenant le message mes. Les méthodes erreur(), empiler() et depiler() peuvent lancer une exception et doivent le signaler dans leur en-tête à l'aide de "**throws Exception**".

Si une méthode traite l'exception à l'aide de catch, elle n'a pas à la transmettre, et n'indique donc pas de "throws Exception". Ci-dessous, l'exception est détectée dans la classe Pile. L'utilisateur de la classe peut décider comme il veut du traitement à faire suite à cette exception. Ce traitement n'est pas figé dans la classe Pile.

```
// Pile.java  gestion d'une pile d'entiers
// la classe Pile est rangée dans le paquetage pile
package mdpaquetage.pile ;

public class Pile {
  private int   sommet;         // repère le dernier occupé (le sommet)
  private int[] element;      // tableau d'entiers alloué dynamiquement

  private void erreur (String mes) throws Exception {
    throw new Exception (mes);
  }

  public Pile (int max) {
    sommet  = -1;
    element = new int [max];               // allocation de max entiers
  }

  public boolean pileVide () {
    return sommet == -1;
  }

  public void empiler (int v) throws Exception {
    if (sommet < element.length - 1) {
       sommet++;
       element [sommet] = v;
    }  else {
      erreur ("Pile saturee");                      // throws Exception
    }
  }

  public int depiler () throws Exception {
    int v = 0;
    if ( !pileVide() ) {
      v = element [sommet];
      sommet--;
    }  else {
      erreur ("Pile vide");                         // throws Exception
    }
    return v;
  }
```

```
  public void viderPile () {
    sommet = -1;
  }

  public void listerPile () {
    idem page 31
  }

}  // class Pile
```

Le programme principal de la pile ressemble également à celui de la page 32. Les différences sont indiquées ci-dessous. Si on tente d'empiler une valeur dans une pile saturée (cas 3), l'exception est traitée dans le try {} le plus externe en dehors de la boucle, et le programme s'arrête. Si on tente de dépiler dans une pile vide (cas 4), l'exception est traitée dans le catch qui suit immédiatement, et la boucle while continue.

L'utilisateur a toute liberté pour décider de ce qu'il veut faire suite à la réception d'une exception. De plus, le message fourni au constructeur de l'exception (new Exception (mes)) dans la méthode erreur() de la classe Pile est réécrit par la méthode toString() de l'exception appelée dans `System.out.println ("Erreur sur pile : " + e)`.

```
// PPPile.java  Programme Principal de la Pile

import mdpaquetage.pile.*;  // class Pile
import mdpaquetage.es.*;    // lireEntier (pour menu())   voir page 304

class PPPile {

  Le menu (voir page 32)

  public static void main (String[] args) {
    Pile    pile1  = new Pile (5);
    boolean fini   = false;
    int     valeur;

    try {
      while (!fini) {

        switch (menu()) {

        case 0, case 1, case 2 (voir page 32)


        case 3 : // empiler une valeur
          System.out.print ("Valeur a empiler ? ");
          valeur = LectureClavier.lireEntier();
          pile1.empiler (valeur);          // peut lancer une Exception
          break;

        case 4 :  // dépiler une valeur
          try {
            valeur = pile1.depiler();      // peut lancer une Exception
            System.out.println ("Valeur depilee : " + valeur);
          } catch (Exception e) {     // capte l'exception de depiler()
            System.out.println ("Erreur sur pile : " + e);
          }
          break;
```

```
            case 5 :  // vider la pile
              pile1.viderPile();
              break;

            case 6 :  // lister la pile
              pile1.listerPile();
              break;
            } // switch
          } // while (!fini)

        } catch (Exception e) {           // capte l'exception de empiler()
          System.out.println ("Erreur sur pile : " + e);
        } // try

      } // main

    } // class PPPile
```

Si la pile est vide, on obtient le message suivant ; le programme continue.

```
Votre choix ? 4
Erreur sur pile : java.lang.Exception : Pile vide
```

Si la pile est saturée, le programme s'arrête après ce message :

```
Votre choix ? 3
Valeur a empiler ? 15
Erreur sur pile : java.lang.Exception : Pile saturee
```

Remarque : les exceptions constituent en fait une hiérarchie d'exceptions auxquelles on peut appliquer les concepts de l'héritage. Voir page 106.

2.11 LA CLASSE STRING

2.11.1 Les différentes méthodes de la classe String

En Java, les chaînes de caractères sont gérées à l'aide de la classe String fournie en standard dans le paquetage java.lang. Cette gestion diffère du langage C où les chaînes de caractères sont mémorisées comme une suite de caractères terminée par 0. En vertu du principe d'encapsulation, l'accès direct aux données d'un objet de la classe String n'est pas possible et les structures de données utilisées sont inconnues. La classe String met à la disposition de l'utilisateur un large éventail de méthodes gérant les chaînes de caractères. Les principales méthodes sont résumées ci-dessous. Elles sont toutes publiques sinon on ne pourrait pas les utiliser.

Remarque : la chaîne de caractères d'un objet de type String **ne peut pas être modifié** . En cas de modification, les méthodes fournissent un **nouvel objet** à partir de la chaîne de l'objet courant et répondant aux caractéristiques de la méthode. Une même chaîne de caractères (String) peut être référencée plusieurs fois puisqu'on est sûr qu'elle ne peut pas être modifiée. La classe **StringBuffer** par contre permet la création et la modification d'une chaîne de caractères.

Les premières méthodes sont des constructeurs d'un objet de type String.

- **String** () : crée une chaîne vide
- **String** (String) : crée une chaîne à partir d'une autre chaîne
- **String** (StringBuffer) : crée une chaîne à partir d'une autre chaîne de type StringBuffer

> ***Remarque*** : String s = "Monsieur" ; est accepté par le compilateur et est équivalent à String s = new String ("Monsieur") ;

Les méthodes ci-après permettent d'accéder à un caractère de la chaîne, de comparer deux chaînes, de concaténer deux chaînes, de fournir la longueur d'une chaîne ou de fournir la chaîne équivalente en majuscules ou en minuscules.

- **char charAt** (int n) : fournit le n ième caractère de la chaîne de l'objet courant.
- **int compareTo** (String s) : compare la chaîne de l'objet et la chaîne s ; fournit 0 si =, <0 si inférieur, > 0 sinon.
- **String concat** (String s) : concatène la chaîne de l'objet et la chaîne s (crée un nouvel objet).
- **boolean equals** (Object s) : compare la chaîne de l'objet et la chaîne s.
- **boolean equalsIgnoreCase** (String s) : compare la chaîne de l'objet et la chaîne s (sans tenir compte de la casse).
- **int length** () : longueur de la chaîne.
- **String toLowerCase** () : fournit une nouvelle chaîne (nouvel objet) convertie en minuscules.
- **String toUpperCase** () : fournit une nouvelle chaîne (nouvel objet) convertie en majuscules.

Les méthodes indexOf() fournissent l'indice dans la chaîne de l'objet courant d'un caractère ou d'une sous-chaîne en partant du premier caractère (par défaut) ou d'un indice donné. Les méthodes lastIndex() fournissent l'indice du dernier caractère ou de la dernière sous-chaîne recherchés dans la chaîne de l'objet référencé.

- **int indexOf** (int c) : indice du caractère c dans la chaîne ; -1 s'il n'existe pas.
- **int indexOf** (int, int) : indice d'un caractère de la chaîne en partant d'un indice donné.
- **int indexOf** (String s) : indice de la sous-chaîne s.
- **int indexOf** (String, int) : indice de la sous-chaîne en partant d'un indice.
- **int lastIndexOf** (int c) : indice du dernier caractère c.
- **int lastIndexOf** (int, int) : indice du dernier caractère en partant de la fin et d'un indice.
- **int lastIndexOf** (String) : indice de la dernière sous-chaîne.
- **int lastIndexOf** (String, int) : indice de la dernière sous-chaîne en partant d'un indice.

Les méthodes suivantes permettent de remplacer un caractère par un autre (en créant un nouvel objet), d'indiquer si la chaîne commence ou finit par une sous-chaîne

donnée, de fournir une sous-chaîne en indiquant les indices de début et de fin, ou de créer une chaîne en enlevant les espaces de début et de fin de chaîne.

- **String replace** (char c1, char c2) : crée une nouvelle chaîne en remplaçant le caractère c1 par c2.
- **boolean startsWith** (String s) : teste si la chaîne de l'objet commence par s.
- **boolean startsWith** (String s, int n) : teste si la chaîne de l'objet commence par s en partant de l'indice n.
- **boolean endsWith** (String s) : teste si la chaîne de l'objet se termine par s.
- **String substring** (int d) : fournit la sous-chaîne de l'objet commençant à l'indice d.
- **String substring** (int d, int f) : fournit la sous-chaîne de l'objet entre les indices d (inclus) et f (exclu).
- **String trim** () : enlève les espaces en début et fin de chaîne.

Ces méthodes **static** fournissent (indépendamment de tout objet) une chaîne de caractères correspondant au paramètre (de type boolean, int, float, etc.)

- static String valueOf (**boolean b**) : fournit true ou false suivant le booléen b.
- static String valueOf (**char**) : fournit la chaîne correspondant au caractère.
- static String valueOf (**char**[]) : fournit la chaîne correspondant au tableau de caractères.
- static String valueOf (**char[], int, int**) : fournit la chaîne correspondant au tableau de caractères entre deux indices.
- static String valueOf (**double**) : fournit un double sous forme de chaîne.
- static String valueOf (**floa**) : fournit un float sous forme de chaîne.
- static String valueOf (**int**) : fournit un int sous forme de chaîne.
- static String valueOf (**long**) : fournit un long sous forme de chaîne.

D'autres méthodes sont disponibles pour par exemple passer d'une chaîne de caractères à un tableau de byte ou de char, pour comparer des sous-chaînes, etc. De même, d'autres constructeurs existent pour créer un objet String à partir d'un tableau de byte ou de char.

2.11.2 L'opérateur + de concaténation (de chaînes de caractères), la méthode toString()

L'opérateur + est un opérateur de concaténation de chaînes de caractères souvent utilisé pour la présentation des résultats à écrire. Il peut aussi s'utiliser pour former une chaîne de caractères. Les objets de type String ne sont pas modifiables ; il faut donc créer un nouvel objet de type String si on concatène deux objets de type String. Si une chaîne (un objet) devient inaccessible, son espace mémoire est automatiquement récupéré (voir page 63).

Si les deux opérandes de l'opérateur + sont de type String, le résultat est de type String. Si un des deux opérandes est de type String et l'autre de type primitif (entier, réel, caractère, booléen, voir page 5), le type primitif est converti en un String (en une chaîne de caractères). Si un des deux opérandes est de type String et l'autre est un objet, l'objet est remplacé par une chaîne de caractères fournie par la méthode

toString() de l'objet (ou celle de la classe Object, ancêtre de toutes les classes, si toString() n'est pas redéfinie dans l'objet ; voir l'héritage au chapitre 3). L'évaluation est faite de gauche à droite s'il y a plusieurs opérateurs +.

La méthode toString() a déjà été redéfinie dans les classes Complex (voir page 53) et Personne (voir page 57) pour fournir les caractéristiques de ces objets (notamment lors de l'écriture avec System.out.println()).

```
// TestString1.java  test de l'opérateur + de concaténation

class TestString1 {

  public static void main (String[] args) {
    String ch1 = "Louis";
    int    i1  = 13;
    String ch2 = ch1 + i1;                                    // Louis13
    System.out.println ("ch2 : " + ch2);
    String ch3 = i1 + ch1;                                    // 13Louis
    System.out.println ("ch3 : " + ch3);
    String ch4 = ch1 + i1 + 2;                               // Louis132
    System.out.println ("ch4 : " + ch4);
    String ch5 = ch1 + (i1 + 2);                              // Louis15
    System.out.println ("ch5 : " + ch5);
    //String ch6 = i1 + 2;     // impossible de convertir int en String
    //System.out.println ("ch6 : " + ch6);
    String ch7 = "" + i1 + 2;                                     // 132
    System.out.println ("ch7 : " + ch7);
    String ch8 = i1 + 2 + "";                                      // 15
    System.out.println ("ch8 : " + ch8);
    String ch9 = Integer.toString (i1 + 2);                        // 15
    System.out.println ("ch9 : " + ch9);
 }  // main

}  // class TestString1
```

Résultats d'exécution :

```
ch2 : Louis13
ch3 : 13Louis
ch4 : Louis132
ch5 : Louis15
ch7 : 132
ch8 : 15
ch9 : 15
```

2.11.3 Exemple d'utilisation des méthodes de la classe String

Le programme Java ci-dessous met en œuvre un certain nombre des méthodes présentées pour la classe String.

```
// TestString2.java

class TestString2 {
```

```
public static void main (String[] args) {
  String s1 = new String();
  String s2 = "";
  String s3 = null;
  String s4 = new String ("Madame");
  String s5 = "Mademoiselle";
  String s6 = "Monsieur";
  String s7 = s4 + ", " + s5 + ", " + s6;
  String s8 = "Madame";

  System.out.println (
    "\ns1 : " + s1 + ", s1.length() : " + s1.length() +
    "\ns2 : " + s2 + ", s2.length() : " + s2.length() +
    "\ns4 : " + s4 + ", s4.length() : " + s4.length()
  );
  System.out.println ("s3 : " + s3);
  //s3.length() provoque une Exception (s3 est null)
  //System.out.println (", s3.length() : " + s3.length());
  System.out.println ("s7 : " + s7);

  String  s9    = s4.concat (", " + s6);
  String  s10   = s4.toUpperCase();
  String  s11   = s4.toLowerCase();
  char    c1    = s4.charAt (2);
  int     comp1 = s4.compareTo (s5);
  int     comp2 = s4.compareTo (s8);
  int     comp3 = s6.compareTo (s4);
  boolean eq1   = s4.equals (s8);       // equals compare les chaînes
  boolean eq2   = (s4 == s8);            // == compare les références
  boolean eq3   = s6.equals (s4);
  System.out.println (
    "\ns4  : "    + s4  +
    "\ns5  : "    + s5  +
    "\ns6  : "    + s6  +
    "\ns7  : "    + s7  +
    "\ns8  : "    + s8  +
    "\ns9  : "    + s9  +
    "\ns10 : "    + s10 +
    "\ns11 : "    + s11 +
    "\n" +
    "\ncaractère 2 de s4       : " + c1    +
    "\ncomp1 de s4 et s5       : " + comp1 +
    "\ncomp2 de s4 et s8       : " + comp2 +
    "\ncomp3 de s6 et s4       : " + comp3 +
    "\neq1   de s4 et s8       : " + eq1   +
    "\neq2   de s6 et s4       : " + eq2   +
    "\neq3   de s4 et s8       : " + eq3
  );

  System.out.println (
    "\nindexOf     a  pour s4  : " + s4.indexOf ('a')    +
    "\nindexOf     a  pour s4  : " + s4.indexOf ('a', 2) +
```

```
      "\nindexOf     x  pour s4  : " + s4.indexOf ("x")    +
      "\nindexOf     si pour s6  : " + s6.indexOf ("si")   +
      "\nLastIndexOf a  pour s4  : " + s4.lastIndexOf ("a")
    );

    System.out.println (
      "\ns4 commence par Mad     : " + s4.startsWith ("Mad")  +
      "\ns6 finit    par ieur    : " + s6.endsWith   ("ieur") +
      "\ns6 fin à partir de 2    : " + s6.substring (2)       +
      "\ns6 sous-chaîne de 2 à 4 : " + s6.substring (2, 4)
    );

    String  s12 = "  127 ";
    boolean b   = true;
    int     i1  = 243;
    System.out.println (
      "\ns12 sans espaces        : " + s12.trim()         +
      "\nvalueOf b               : " + String.valueOf (b) +
      "\nvalueOf i1              : " + String.valueOf (i1)
    );

  }  // main

}  // class TestString2
```

Remarque : valueOf() est une méthode `static` appelée en la préfixant du nom de la classe String.

Exemples de résultats :

```
s1 : , s1.length() : 0
s2 : , s2.length() : 0
s4 : Madame, s4.length() : 6
s3 : null
s7 : Madame, Mademoiselle, Monsieur
s4  : Madame
s5  : Mademoiselle
s6  : Monsieur
s7  : Madame, Mademoiselle, Monsieur
s8  : Madame
s9  : Madame, Monsieur
s10 : MADAME
s11 : madame
caractère 2 de s4       : d
comp1 de s4 et s5       : -4                              //s4 < s5
comp2 de s4 et s8       : 0
comp3 de s6 et s4       : 14                            // s6 > s4
eq1   de s4 et s8       : true                       // les chaînes
eq2   de s4 == s8       : false                   // les références
eq3   de s6 et s4       : false
```

```
indexOf     a  pour s4  : 1
indexOf     a  pour s4  : 3              // en partant de l'indice 2
indexOf     x  pour s4  : -1
indexOf     si pour s6  : 3
LastIndexOf a  pour s4  : 3

s4 commence par Mad     : true
s6 finit    par ieur    : true
s6 fin à partir de 2    : nsieur
s6 sous-chaîne de 2 à 4 : ns
s12 sans espaces        : 127

valueOf b               : true
valueOf i1              : 243
```

2.12 LA CLASSE STRINGBUFFER

2.12.1 Les principales méthodes de la classe StringBuffer

Un objet de type **StringBuffer** représente une chaîne de caractères modifiable. On peut changer un caractère, ajouter des caractères au tampon interne de l'objet défini avec une capacité qui peut être changée si besoin est.

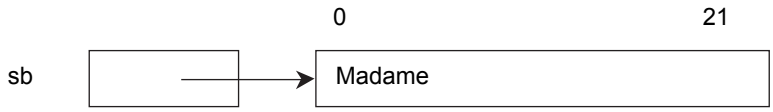


Figure 2.23 — Un objet sb de la classe StringBuffer.
Il contient six caractères et a une capacité de 22 caractères, numérotés de 0 à 21.

On peut construire un objet de type StringBuffer contenant 0 caractère avec une capacité par défaut de 16 ou définie à l'aide d'un paramètre entier. On peut aussi construire un objet de type StringBuffer à partir d'un objet de type String. Si la capacité devient insuffisante, il y a réallocation automatique avec agrandissement de la capacité. L'utilisateur peut aussi redéfinir la capacité d'un l'objet. Les principales méthodes sont présentées succinctement ci-après.

- **StringBuffer** () : construit un tampon ayant 0 caractère mais d'une capacité de 16 caractères.
- StringBuffer (**int n**) : construit un tampon de capacité n caractères.
- StringBuffer (**String**) : construit un objet de type StringBuffer à partir d'un objet de type String.

Les méthodes de type **StringBuffer append** () concatènent au tampon de l'objet courant la représentation sous forme de caractères de leur argument (boolean, int, etc.).

- StringBuffer append (**boolean**);
- StringBuffer append (**char**);

- StringBuffer append (**char[]**);
- StringBuffer append (**char[], int, int**);
- StringBuffer append (**double**);
- StringBuffer append (**floa**);
- StringBuffer append (**int**);
- StringBuffer append (**long**);
- StringBuffer append (**Object**) : ajoute toString() de l'objet.
- StringBuffer append (**String**).

Les méthodes de type **StringBuffer insert** () insèrent dans le tampon de l'objet, à partir de l'indice du paramètre 1, la représentation sous forme de caractères de leur argument.

- StringBuffer insert **(int, boolean**) ;
- StringBuffer insert (**int, char**) ;
- StringBuffer insert (**int, char[]**) ;
- StringBuffer insert (**int, double**) ;
- StringBuffer insert (**int, floa**) ;
- StringBuffer insert (**int, int**) ;
- StringBuffer insert (**int, long**) ;
- StringBuffer insert (**int, Object**) :ajoute toString() de l'objet.
- StringBuffer insert (**int, String**).

D'autres méthodes fournissent la capacité de l'objet de type StringBuffer, l'accès à un caractère à partir de son indice, la longueur de la chaîne, la chaîne inverse, ou l'objet de type String équivalent.

- **int capacity** () : fournit la capacité de cet objet (nombre maximum de caractères) ;
- **char charAt** (int n) : fournit le caractère d'indice n ;
- **int length** () : fournit la longueur (nombre de caractères) ;
- **StringBuffer reverse** () : inverse les caractères de l'objet (gauche-droite devient droite-gauche) ;
- **String toString** () : convertit en un objet de type String ;
- **ensureCapacity** (int n) : la capacité de l'objet est au moins n sinon un nouveau tampon est alloué ;
- **void setCharAt** (int index, char ch) : le caractère index reçoit ch (0 <= index < longueur).

2.12.2 Exemple de programme utilisant StringBuffer

```
// TestStringBuffer.java

class TestStringBuffer {

  static void ecrire (StringBuffer sb, String nom) {
    // écrit les caractéristiques de sb, plus son nom
    System.out.println (
      nom + " : "            + sb            + ", " +
```

```
      nom + ".length() : "   + sb.length()   + ", " +
      nom + ".capacity() : " + sb.capacity()
    );
  }

  public static void main (String[] args) {
    // constructeurs d'objets de la classe StringBuffer
    StringBuffer sb1 = new StringBuffer ();
    StringBuffer sb2 = new StringBuffer ("Bonjour");
    StringBuffer sb3 = new StringBuffer ("Madame");
    StringBuffer sb4 = new StringBuffer ("Mademoiselle");
    StringBuffer sb5 = new StringBuffer ("Monsieur");
    StringBuffer sb6 = new StringBuffer (sb3 + ", " + sb4 + ", "
                                                          + sb5);

    // écriture des caractéristiques des objets
    ecrire (sb3, "sb3");
    ecrire (sb4, "sb4");
    ecrire (sb5, "sb5");
    ecrire (sb6, "sb6");

    // on inverse sb3, on lui ajoute "," suivie de l'inverse de sb4
    // sb3 est modifié
    sb3.reverse();
    sb3.append (", " + sb4.reverse() );     // sb3 et sb4 sont modifiés
    ecrire (sb3, "sb3");

    // on ajoute à sb3 une virgule suivie de l'inverse de sb5
    sb3.append (", " + sb5.reverse() );
    ecrire (sb3, "sb3");

    // on convertit sb3 en un objet de la classe String
    String s3 = sb3.toString();
    System.out.println ("s3  : " + s3.toUpperCase() );

    // accès et modification d'un caractère de sb2
    char c1 = sb2.charAt (2);  // c1 vaut n
    sb2.setCharAt (2, 'u');    // sb2 vaut Boujour
    System.out.println ("c1 : " + c1 + ", sb2  : " + sb2 );

    // ajout et insertion de caractères dans sb1
    int     i = 572;
    boolean b = true;
    sb1.append (i);                 //572
    ecrire (sb1, "sb1");
    sb1.insert (3, b);              //572true
    ecrire (sb1, "sb1");
    sb1.insert (3, "  ");           //572  true
    ecrire (sb1, "sb1");
  }  // main

}  // class TestStringBuffer
```

Exemples de résultats :

```
sb3 : Madame, sb3.length() : 6, sb3.capacity() : 22
                        sb3 a une longueur de 6 et une capacité de 22
sb4 : Mademoiselle, sb4.length() : 12, sb4.capacity() : 28
sb5 : Monsieur, sb5.length() : 8, sb5.capacity() : 24
sb6 : Madame, Mademoiselle, Monsieur, sb6.length() : 30,
                                                    sb6.capacity() : 46
sb3 : emadaM, ellesiomedaM, sb3.length() : 20, sb3.capacity() : 22
 sb3 a maintenant une longueur de 20 et une capacité inchangée de 22
sb3 : emadaM, ellesiomedaM, rueisnoM, sb3.length() : 30,
                                                    sb3.capacity() : 46
          sb3 a une longueur de 30 et une capacité (augmentée) de 46
s3  : EMADAM, ELLESIOMEDAM, RUEISNOM
c1 : n, sb2  : Boujour
sb1 : 572,       sb1.length() : 3, sb1.capacity() : 16
sb1 : 572true,   sb1.length() : 7, sb1.capacity() : 16
sb1 : 572  true, sb1.length() : 9, sb1.capacity() : 16
```

2.13 LA CLASSE VECTOR

La classe Vector s'apparente à un tableau à une dimension dont on aurait pas à indiquer le nombre maximum d'éléments. L'espace nécessaire en cas d'agrandissement du Vector est automatiquement alloué. Sur l'exemple ci-dessous, un Vector vide a une capacité de dix objets. Lorsqu'on essaie de ranger un 11 i^{e} élément, la capacité passe automatiquement à 20.

On peut accéder, modifier ou détruire un élément du Vector à partir de son rang. Les éléments du Vector sont des objets (de type String pour v1, et de type Complex pour v2 ci-dessous). Voir Complex page 45.

```
// TestVector.java

import java.util.*;                // Vector
import mdpaquetage.complex.*;

class TestVector {

  public static void main (String[] args) {
    Vector v1 = new Vector ();
    v1.addElement ("chaine 0");
    v1.addElement ("chaine 1");
    v1.addElement ("chaine 2");
    v1.addElement ("chaine 3");
    v1.addElement ("chaine 4");
    System.out.println ("v1 size     : " + v1.size());
    System.out.println ("v1 capacity : " + v1.capacity());

    System.out.print ("\nv1 : ");
```

```
    for (int i=0; i < v1.size(); i++) {
      System.out.print (v1.elementAt(i));
      System.out.print ((i < v1.size()-1) ? ", " : "");
    }

    // en fournissant la référence du Vector
    System.out.println ("\nv1 bis : " + v1);


    Vector v2 = new Vector ();
    v2.addElement (new Complex (10, 1));   // élément 0
    v2.addElement (new Complex (20, 2));   // 1
    v2.addElement (new Complex (30, 3));   // 2
    v2.addElement (new Complex (40, 4));   // 3
    v2.addElement (new Complex (50, 5));   // 4
    //v2.addElement ("chaine de caractères");

    // modifier l'élément 1
    v2.setElementAt (new Complex (200, 20), 1);
    // supprimer l'élément 3
    v2.removeElementAt(3);

    System.out.println ("\n\nv2 : ");
    for (int i=0; i < v2.size(); i++) {
      Complex c = (Complex) v2.elementAt(i);
      System.out.print( "\n" + i + " : "); c.ecritC();
    }
    //System.out.println ("\nv2 bis : " + v2);

  }  // main

}  // class TestVector
```

Résultats de l'exécution du programme ci-dessus :

```
v1 size     : 5
v1 capacity : 10
v1 : chaine 0, chaine 1, chaine 2, chaine 3, chaine 4
v1 bis : [chaine 0, chaine 1, chaine 2, chaine 3, chaine 4]
v2 :
0 : (10.0 + 1.0 i)
1 : (200.0 + 20.0 i)
2 : (30.0 + 3.0 i)
3 : (50.0 + 5.0 i)
```

2.14 LA CLASSE HASHTABLE

Une hashtable est une table qui permet de retrouver un objet à partir de son identifiant. Ci-dessous, les clés sont : "etudiant1", "etudiant2", "etudiant3". A partir d'une de ces clés, on peut retrouver les caractéristiques de l'étudiant correspondant.

La mise en œuvre est simple. La méthode **put**() permet de mémoriser la clé et l'objet qu'elle caractérise. La méthode **get**() permet à partir de la clé de retrouver l'objet correspondant à cette clé. On peut supprimer un élément à partir de sa clé (remove()).

La classe Etudiant est définie dans le paquetage etudiant. Voir page 264. Elle permet de créer un objet Etudiant à partir de ses caractéristiques.

```
// TestHashtable.java

import java.util.*;              // Vector
import mdpaquetage.etudiant.*;

class TestHashtable {

  public static void main (String[] args) {
    Hashtable h1 = new Hashtable ();
    // insertion d'éléments dans la Hashtable
    h1.put ("etudiant1",
              new Etudiant ("etudiant1", "prenom1", "A", "A1", true));
    h1.put ("etudiant2",
             new Etudiant ("etudiant2", "prenom2", "B", "B1", false));
    h1.put ("etudiant3",
              new Etudiant ("etudiant3", "prenom3", "A", "A2", true));
    System.out.println ("h1 size     : " + h1.size());
    System.out.println ("h1          : " + h1);

    // recherche et destruction d'un élément de la table
    String   cherche  = "etudiant1";
    Etudiant etudiant = (Etudiant) h1.get (cherche);
    if (etudiant != null) {
      System.out.println ("caractéristiques : " + etudiant);
    }

    h1.remove (cherche);
    etudiant = (Etudiant) h1.get (cherche);
    if (etudiant == null) {
      System.out.println (cherche + " inconnu");
    }
    System.out.println ("h1          : " + h1);

  }  // main

}  // class TestHashtable
```

2.15 LES CLASSES INTERNES

Une classe en Java peut être insérée dans une autre classe. Ci-dessous, B est une classe interne à la classe A. Les classes internes sont surtout utilisées pour la **gestion des événements** (voir page 163). Un objet d'une classe interne est associé à l'objet de la classe englobante l'ayant créé. Un objet interne peut accéder aux attributs et méthodes de l'objet externe associé. Un objet externe peut accéder aux champs et méthodes de l'objet qu'il a créé.

public **class A** {
attributs de A // par exemple B b1 = new B() ; B b2 = new B() ;
méthodes de A // les méthodes d'instance (non static) peuvent créer
et utiliser des objets de la classe B
class B { // **classe interne** : aucun objet de type B n'est a priori créé
// lorsqu'on crée un objet de type A.
// un objet de classe B peut être créé dans une méthode de A
// ou en tant qu'attribut de A.
attributs d'instance de B
méthodes d'instance de B // elles peuvent accéder aux attributs
// et méthodes de A.
} // class B
} // class A

Exemple de programme Java :

```
// A.java

public class A {
  int    va  = 10;
  String mot = "bonjour";
  B nb = new B();   // objet B (classe interne) attribut de la classe A

  A (int va) {
    this.va = va;
  }

  void a1 () {
    System.out.println ("\n\na1 : création de objetb1");
    B objetb1 = new B(100);               // objet de la classe interne
    // accès à l'attribut (vb) de objetb1 de la classe interne B
    System.out.println ("a1 : objetb1.vb : " + objetb1.vb);
    objetb1.b1();                 // appel d'une méthode de la classe B

    System.out.println ("\n\na1 : création de objetb2");
    B objetb2 = new B();                  // objet de la classe interne
    // accès à l'attribut (vb) de objetb2 de la classe interne B
    System.out.println ("a1 : objetb2.vb : " + objetb2.vb);
    objetb2.b1();                 // appel d'une méthode de la classe B
  } // a1

  void a2 () {
    System.out.println ("*** methode a2 ");
  }

  // classe interne B
  public class B {
    private int vb = 5;  // attribut de la classe interne

    B (int vb) {
      this.vb = vb;
    }
```

```
    B () {
    }

    void b1 () {
      // un objet de la classe B peut accéder à un attribut de A
      // (ex : accès à l'attribut "va" de la classe A)
      System.out.println ("  b1 : mot " + mot + ", va "
                                                + va + ", vb " + vb);
      a2();  // appel d'une méthode de la classe A
    }
  }  // fin de la classe interne B

  public static void main (String[] args) {
    System.out.println ("\n\ncréation de objeta1");
    A objeta1 = new A(10);
    objeta1.a1();
    System.out.println();

    System.out.println ("\n\ncréation de objeta2");
    A objeta2 = new A(50);
    objeta2.a1();
  }  // main

}  // fin de class A
```

2.16 CONCLUSION

En programmation objet, une classe (un objet) existe indépendamment d'une application. On essaie de répertorier toutes ses caractéristiques intrinsèques sans tenir compte d'une application particulière. Une application donnée n'utilise souvent qu'un sous-ensemble des méthodes disponibles de la classe d'un objet.

C'est tout particulièrement le cas des classes String et StringBuffer qui offrent tout un éventail de méthodes dont seulement un sous-ensemble est utilisé dans une application. C'est également vrai pour la classe Pile où les différentes opérations ont été définies indépendamment de toutes applications. La classe Ecran offre également à l'utilisateur, une panoplie de méthodes lui permettant de réaliser un dessin.

Le principe d'encapsulation des données fait que les données (les attributs, les caractéristiques) d'un objet ne sont accessibles que par l'intermédiaire des méthodes qui constituent une interface non basée sur les structures de données. Le concepteur de la classe peut modifier et optimiser ses structures de données. Dès lors qu'il ne modifie pas l'interface, cela n'a aucune incidence sur les programmes utilisateurs.

La classe peut être vue comme une extension de la notion de module ou des concepts de type abstrait de données en programmation structurée. Le concepteur de la classe peut ajouter des méthodes ou surcharger des méthodes ; cela n'affectera pas les applications utilisant déjà cette classe. Il peut enrichir l'interface ; il ne doit pas modifier les prototypes des méthodes déjà en place.

Chapitre **3**

L'héritage

3.1 LE PRINCIPE DE L'HÉRITAGE

3.1.1 Les définitions

Le deuxième grand principe de la programmation objet après l'encapsulation (voir figure 2.3, page 31) est le concept d'héritage. Une classe B qui hérite d'une classe A hérite des attributs et des méthodes de la classe A sans avoir à les redéfinir. B est une **sous-classe** de A ; ou encore A est la **super-classe** de B ou la classe de base. On dit encore que B est une **classe dérivée** de la classe A, ou que la classe B étend la classe A. Une classe ne peut avoir qu'une seule super-classe ; il n'y a pas d'héritage multiple en Java. Par contre, elle peut avoir plusieurs sous-classes (voir figure 3.2).

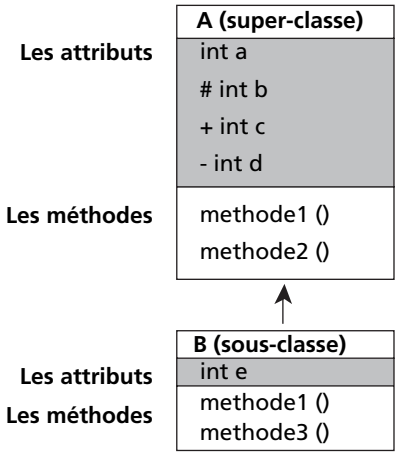


Figure 3.1 — Le principe de l'héritage : la classe B hérite de la classe A.
indique un attribut protected.

3.1.2 La redéfinition des méthodes et l'utilisation du mot-clé protected

Les droits d'accès aux données (private, public et de paquetage) ont été présentés à la page 67. Le concept d'héritage ajoute un nouveau droit d'accès dit protégé (**protected)** : celui pour une classe dérivée d'accéder aux attributs ou aux méthodes de sa super-classe qui ont été déclarés `protected`. Un attribut déclaré `protected` est accessible dans son **paquetage** et dans ses **classes dérivées**. Si un attribut de la super-classe est privé (private), la sous-classe ne peut y accéder. Sur la figure 3.1, un objet de la classe B peut accéder aux attributs a, b, c et e (a, b et c sont obtenus par héritage ; d privé est inaccessible de la classe B). La classe B contient les méthodes methode1() et methode3() de sa classe B, mais peut aussi accéder aux méthodes methode1() et methode2() de la classe A.

Soit b1 un objet de la classe B obtenu par B b1 = new B() :

accès aux attributs :

b1 peut accéder à l'attribut `c` (public : b1.c), à l'attribut `b` (protected dont accessible dans la sous-classe : b1.b), à l'attribut `a` si classe B et classe A sont dans le même paquetage (b1.a). b1 ne peut pas accéder à l'attribut `d` qui est privé (b1.d interdit). Voir page 67.

Accès aux méthodes :

- b1 peut activer une méthode spécifique de B, methode3() par exemple, comme suit : b1.methode3().
- b1 peut activer une méthode non redéfinie de sa super-classe A comme si elle faisait partie de sa classe B. Exemple : b1.methode2().
- methode1() est redéfinie dans la sous-classe B. On dit qu'il y a **redéfinition de méthodes**. Contrairement aux méthodes surchargées d'une classe qui doivent se différencier par les paramètres (voir page 46), une méthode redéfinie doit avoir un **prototype identique** à la fonction de sa super-classe. b1 exécutera methode1() de sa classe B sur l'instruction qui suit : b1.methode1(). L'objet peut aussi activer la méthode de sa super-classe A en préfixant l'appel de la méthode de **super** : **b1.super.methode1().** On suppose sur cet exemple que methode1() de la classe A et methode1() de la classe B ont le même nombre de paramètres, chacun des paramètres ayant le même type, et la valeur de retour est la même.

Un des avantages de la programmation objet est de pouvoir réutiliser et enrichir des classes sans avoir à tout redéfinir. La super-classe contient les attributs et méthodes **communs** aux classes dérivées. La classe dérivée ajoute des attributs et des méthodes qui spécialisent la super-classe. Si une méthode de la super-classe ne convient pas, on peut la redéfinir dans la sous-classe. Cette méthode redéfinie peut appeler la méthode de la super-classe si elle convient pour effectuer une partie du traitement. On peut utiliser sans les définir les méthodes de la super-classe si elles conviennent.

Remarque : si une classe est déclarée **abstraite,** on ne peut instancier de variables de cette classe. C'est un modèle pour de futures classes dérivées. On peut aussi interdire en Java la dérivation d'une classe en la déclarant `final`.

```
final class A {
  ...
}
```

3.2 LES CLASSES PERSONNE, SECRETAIRE, ENSEIGNANT, ETUDIANT

Dans un établissement d'enseignement et de manière schématique, on trouve trois sortes de personnes : des administratifs (au sens large, incluant des techniciens) représentés par la catégorie secrétaire, des enseignants et des étudiants. Chaque personne est caractérisée par son nom et prénom, son adresse (rue et ville) qui sont des attributs privés et communs à toutes les personnes. On peut représenter une personne suivant un schéma UML comme indiqué sur la figure 3.2. Les variables d'instances (attributs) sont le nom, le prénom, la rue et la ville. nbPersonnes est une variable de classe (donc static) qui comptabilise le nombre de Personne dans l'établissement. Cette variable de classe existe en un exemplaire indépendant de tout objet. Sur la figure 2.9, page 43 et sur la figure 2.11, page 48, les variables `static` apparaissent à part ; elles apparaissent en souligné avec les variables d'instance dans les classes de la figure 3.2 pour simplifier le schéma ; il est important de bien comprendre qu'elles ont un comportement différent des variables d'instance.

On définit les méthodes `public` suivantes de la classe **Personne** :

- le constructeur **Personne** (String nom, String prenom, String rue, String ville) : crée et initialise un objet de type Personne.
- String **toString** () : fournit une chaîne de caractères correspondant aux caractéristiques (attributs) d'une personne.
- **ecrirePersonne** () : pourrait écrire les caractéristiques d'une personne. Sur l'exemple ci-dessous, elle ne fait rien. Elle est déclarée **abstraite**.
- static **nbPersonnes** () : écrit le nombre total de personnes et le nombre de personnes par catégorie. C'est une méthode de classe.
- **modifier ersonne** (String rue, String ville) : modifie l'adresse d'une personne et appelle **ecrirePersonne** () pour vérifier que la modification a bien été faite.

3.2.1 La classe abstraite Personne

Si dans l'établissement, il n'y a exclusivement que trois catégories de personnes, une Personne n'existe pas. Seuls existent des secrétaires, des enseignants et des étudiants. Néanmoins, la classe Personne est utile dans la mesure où elle permet de regrouper les attributs et les méthodes communs aux trois catégories de personnel. La classe Personne est déclarée **abstraite**. On ne peut pas créer (instancier) d'objet de type Personne :

```
Personne p = new Personne ( "Dupond", "Jacques", "rue des mimosas",
                                                        "Paris");
```

fournit une **erreur** de compilation.

3.2.2 Les classes Secretaire, Enseignant et Etudiant

Une **Secretaire** est une Personne. Elle possède toutes les caractéristiques d'une Personne (nom, prenom, rue, ville) plus les caractéristiques spécifiques d'une secrétaire soit sur l'exemple, un numéro de bureau. Les méthodes suivantes sont définies pour un objet de la classe **Secretaire** :

- **Secretaire** (String nom, String prenom, String rue, String ville, String numeroBureau) : le constructeur d'un objet de la classe Secretaire doit fournir les caractéristiques pour construire une Personne, plus les spécificités de la classe Secretaire (numéro de bureau).

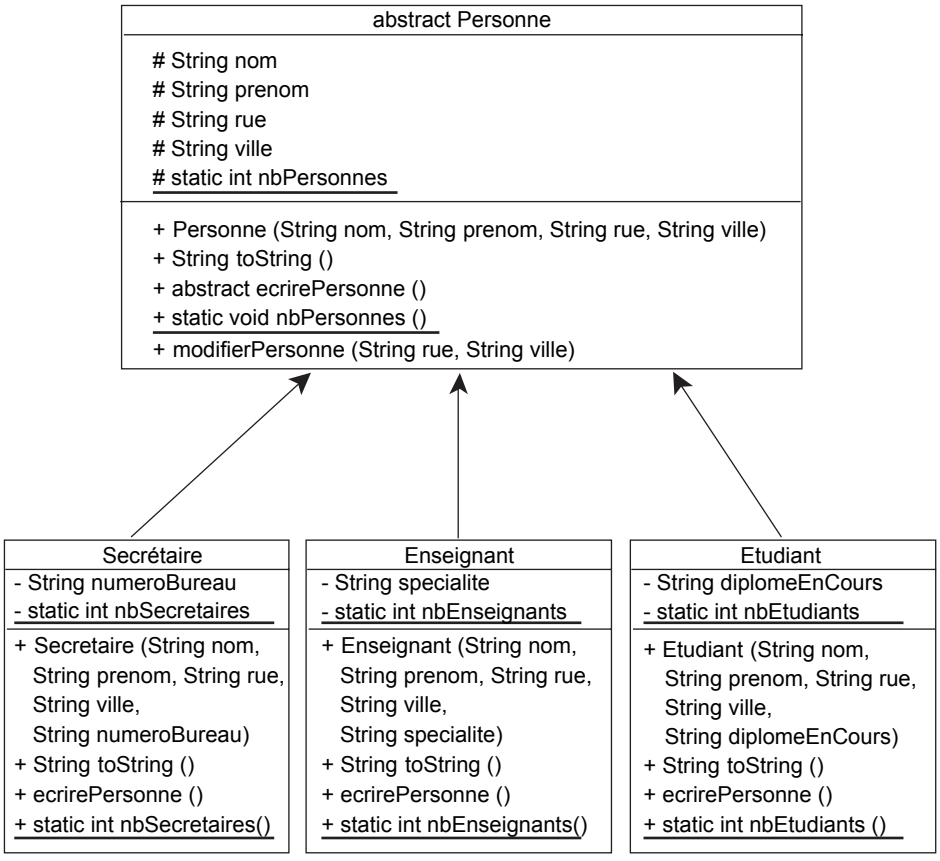


Figure 3.2 — Arbre d'héritage.

Les classes Secretaire, Enseignant et Etudiant sont des classes dérivées de la classe Personne. La classe Personne est abstraite et ne peut être instanciée.

- String **toString** () : fournit une chaîne contenant les caractéristiques d'une Secretaire.
- **ecrirePersonne** () : écrit "Secrétaire : " suivi des caractéristiques d'une Secretaire.

De même, un **Enseignant** est une Personne enseignant une spécialité (mathématiques, informatique, anglais, gestion, etc.). Un **Etudiant** est une Personne préparant un diplôme (diplomeEnCours). Les méthodes pour Enseignant et Etudiant sont similaires à celles de Secretaire. Une variable privée `static` dans chaque classe compte le nombre de personnes créées dans chaque catégorie. Une méthode `static` du même nom que la variable fournit la valeur de cette variable `static` (nbSecretaires, nbEnseignants, nbEtudiants).

3.2.3 Les mots-clés abstract, extends, super

Le programme Java correspondant à l'arbre d'héritage de la figure 3.2 est donné ci-après. Une classe contenant une méthode abstraite est forcément abstraite même si on ne l'indique pas explicitement. Ci-dessous, les attributs *protected* nom, prenom, rue, ville et la variable `static` nbPersonnes sont accessibles dans les classes dérivées Secretaire, Enseignant ou Etudiant même si les classes ne sont pas dans le même paquetage. Voir protected page 94.

```
// PPPersonne.java  Programme Principal Personne
//                  exemples d'héritage

// une personne
abstract class Personne {         // classe abstraite, non instanciable
  protected String nom;
  protected String prenom;
  protected String rue;
  protected String ville;
  protected static int nbPersonnes = 0;           // nombre de Personne

  // constructeur de Personne
  Personne (String nom, String prenom, String rue, String ville) {
    this.nom    = nom;                             // voir this page 38
    this.prenom = prenom;
    this.rue    = rue;
    this.ville  = ville;
    nbPersonnes++;
  }

  // fournir les caractéristiques d'une Personne sous forme
  // d'un objet de la classe String
  public String toString() {
    return nom + " " + prenom + " " + rue + " " + ville;
  }

  // méthode abstraite          (à redéfinir dans les classes dérivées)
  abstract void ecrirePersonne ();

  static void nbPersonnes () {
    System.out.println (
      "\nNombre d'employés     : " + nbPersonnes +
```

```
        "\nNombre de secrétaires : " + Secretaire.nbSecretaires() +
        "\nNombre d'enseignants  : " + Enseignant.nbEnseignants() +
        "\nNombre d'étudiants    : " + Etudiant.nbEtudiants()
      );
    }

    void modifierPersonne (String rue, String ville) {
      this.rue     = rue;
      this.ville   = ville;
      ecrirePersonne();               // de la classe dérivée de l'objet
    }

  }  // Personne
```

Une **Secretaire est une Personne** ayant une caractéristique supplémentaire : un numéro de bureau. La variable static nbSecretaires est incrémentée dans le constructeur de l'objet Secretaire. L'appel **super** (nom, prenom, rue, ville) fait appel au constructeur de la super-classe chargé d'initialiser les données communes à toute Personne : nom, prenom, rue et ville. De même, l'appel **super.toString()** fournit une chaîne de caractères en utilisant la méthode toString() de la super-classe de Secretaire (soit la classe Personne). Cet appel fournit une chaîne de caractères donnant les caractéristiques communes d'une personne. A cette chaîne, la méthode toString() de Secretaire ajoute le numéro de bureau qui lui est spécifique d'un objet Secretaire. Le principe est le même pour les méthodes toString() de Enseignant et Etudiant : toString() de la super-classe fournit les caractéristiques communes et toString() de la classe dérivée, les spécificités de la classe. On voit bien sur cet exemple que le "savoir-faire" de la super-classe est réutilisé sans devoir le réécrire. On en hérite et on l'enrichit.

```
class Secretaire extends Personne {               // héritage de classe
  private String numeroBureau;
  private static int nbSecretaires = 0;          // nombre de Secretaire

  Secretaire (String nom, String prenom, String rue, String ville,
              String numeroBureau) {
    // le constructeur de la super-classe
    super (nom, prenom, rue, ville);  // appel du constructeur Personne
    this.numeroBureau = numeroBureau;
    nbSecretaires++;
  }

  public String toString () {
    // super.toString() : toString() de la super-classe
    return super.toString() + "\n  N° bureau : " + numeroBureau;
  }

  void ecrirePersonne () {
    System.out.println ("Secrétaire : " + toString());
  }

  static int nbSecretaires () { return nbSecretaires; }

}  // Secretaire
```

Un **Enseignant est une Personne** enseignant une spécialité. La variable `static` nbEnseignants compte le nombre d'enseignants ; elle est incrémentée dans le constructeur Enseignant.

```
class Enseignant extends Personne {                // héritage de classe
  private String specialite;
  private static int nbEnseignants = 0;           // nombre d'Enseignant

  Enseignant (String nom, String prenom, String rue, String ville,
              String specialite) {
    super (nom, prenom, rue, ville);  // appel du constructeur Personne
    this.specialite = specialite;
    nbEnseignants++;
  }

  public String toString () {
    return super.toString() + "\n  spécialité : " + specialite;
  }

  void ecrirePersonne () {
    System.out.println ("Enseignant : " + toString());
  }

  static int nbEnseignants () { return nbEnseignants; }

} // Enseignant
```

Un **Etudiant est une P ersonne** préparant un diplôme. La variable `static` nbEtudiants compte le nombre d'étudiants ; elle est incrémentée dans le constructeur Etudiant.

```
class Etudiant extends Personne {                  // héritage de classe
  private String diplomeEnCours;
  private static int nbEtudiants = 0;               // nombre d'Etudiant

  Etudiant (String nom, String prenom, String rue, String ville,
            String diplomeEnCours) {
    super (nom, prenom, rue, ville);  // appel du constructeur Personne
    this.diplomeEnCours = diplomeEnCours;
    nbEtudiants++;
  }

  public String toString () {
    return super.toString() + "\n  Diplome en cours : "
                                                    + diplomeEnCours;
  }

  void ecrirePersonne () {
    System.out.println ("Etudiant : " + toString());
  }

  public String diplomeEnCours () {
    return diplomeEnCours;
  }

  static int nbEtudiants () { return nbEtudiants; }
```

```
} // Etudiant
```

La classe **PPPersonne** crée des objets Secretaire, Enseignant et Etudiant. La méthode static nbPersonnes() est appelée en la préfixant de la classe Personne.

```
class PPPersonne {                          // Programme Principal Personne
  public static void main(String[] args) {
    Secretaire chantal = new Secretaire ("Dupond", "Chantal",
      "rue des mimosas", "Rennes", "A123");
    chantal.ecrirePersonne();

    Enseignant michel = new Enseignant ("Martin", "Michel",
      "bd St-Antoine", "Rennes", "Maths");
    michel.ecrirePersonne();

    Etudiant e1 = new Etudiant ("Martin", "Guillaume",
      "bd St-Jacques", "Bordeaux", "licence info");
    e1.ecrirePersonne();

    Etudiant e2 = new Etudiant ("Dufour", "Stéphanie",
      "rue des saules", "Lyon", "DUT info");
    e2.ecrirePersonne();

    Personne.nbPersonnes();                          // méthode static

    System.out.println ("\n\nAprès modification :");
    chantal.modifierPersonne ("rue des sorbiers", "Nantes");
    michel.modifierPersonne ("rue des lilas", "Rennes");

  }  // main

}  // class PPPersonne
```

Exemples de résultats d'exécution de PPPersonne :

```
Secrétaire : Dupond Chantal rue des mimosas Rennes
  N° bureau : A123
Enseignant : Martin Michel bd St-Antoine Rennes
  spécialité : Maths
Etudiant : Martin Guillaume bd St-Jacques Bordeaux
  Diplome en cours : licence info
Etudiant : Dufour Stéphanie rue des saules Lyon
  Diplome en cours : DUT info

Nombre d'employés     : 4
Nombre de secrétaires : 1
Nombre d'enseignants  : 1
Nombre d'étudiants    : 2

Après modification :
Secrétaire : Dupond Chantal rue des sorbiers Nantes
  N° bureau : A123
Enseignant : Martin Michel rue des lilas Rennes
  spécialité : Maths
```

Remarque : on pourrait ne pas déclarer la méthode ecrirePersonne() abstraite en fournissant une méthode vide. Mais alors, les classes dérivées ne seraient plus obligées de la redéfinir.

3.2.4 Les méthodes abstraites, la liaison dynamique, le polymorphisme

3.2.4.1 Le polymorphisme d'une méthode redéfinie : ***ecrirePersonne()***

La méthode modifierPersonne modifie l'adresse (rue, ville) d'une Personne quelle que soit sa catégorie. Elle se trouve dans la classe commune Personne. Elle peut être appelée pour un objet Secretaire, Enseignant ou Etudiant.

```
void modifierPersonne (String rue, String ville) {
    this.rue   = rue;
    this.ville = ville;
    ecrirePersonne();       // ecrirePersonne() de la classe de l'objet
  }
```

Par contre, que fait la méthode ecrirePersonne() ? Si le compilateur faisait une **liaison statique**, il générerait des instructions prenant en compte la méthode ecrirePersonne() de la classe Personne et qui en fait ne fait rien sur cet exemple.

Quand le compilateur prévoit une **liaison dynamique** , la méthode ecrirePersonne() qui est exécutée dépend à l'exécution de la classe de l'objet. La méthode est d'abord recherchée dans la classe de l'objet, et en cas d'échec dans sa super-classe. Si chaque sous-classe redéfinit la méthode, celle de la super-classe n'est jamais exécutée et peut être déclarée abstraite. Elle sert de prototype lors de la compilation de modifierPersonne() de la classe Personne. Si la méthode est déclarée abstraite, les sous-classes doivent la redéfinir sinon, il y a un message d'erreur à la compilation.

Les instructions suivantes appellent la méthode modifierPersonne() de Personne pour deux objets différents dont la classe est dérivée de Personne.

```
chantal.modifierPersonne ("rue des sorbiers", "Nantes");
michel.modifierPersonne  ("rue des lilas", "Rennes");
```

On obtient les résultats indiqués ci-après. Pour l'objet chantal, la méthode modifierPersonne() appelle la méthode ecrirePersonne() de Secretaire. Pour l'objet michel, la méthode modifierPersonne() appelle la méthode ecrirePersonne() de Enseignant. On parle alors de **polymorphisme**. Le même appel de la méthode ecrirePersonne() de modifierPersonne() déclenche des méthodes redéfinies différentes (portant le même nom) de différentes classes dérivées. Le polymorphisme découle de la notion de redéfinition de méthodes entre une super-classe et une sous-classe.

```
Secrétaire : Dupond Chantal rue des sorbiers Nantes
  N° bureau : A123
Enseignant : Martin Michel rue des lilas Rennes
  spécialité : Maths
```

3.2.4.2 Le polymorphisme d'une variable de la super-classe

Le programme PPPersonne vu ci-dessus pourrait être écrit comme suit. La variable locale p est une référence sur une Personne. On ne peut pas créer d'instance de Personne (la classe est abstraite), mais on peut référencer à l'aide d'une variable de type Personne, un **descendant** de Personne (Secretaire, Enseignant ou Etudiant). L'instruction **p.ecrirePersonne() est polymorphe** ; la méthode appelée est celle de la classe de l'objet.

```
class PPPersonne2 {                    // Programme Principal Personne 2
  public static void main (String[] args) {
    Personne p;

    p = new Secretaire ("Dupond", "Chantal",
      "rue des mimosas", "Rennes", "A123");
    p.ecrirePersonne();                // ecrirePersonne() de Secretaire

    p = new Enseignant ("Martin", "Michel",
      "bd St-Antoine", "Rennes", "Maths");
    p.ecrirePersonne();                // ecrirePersonne() de Enseignant

    p = new Etudiant ("Martin", "Guillaume",
      "bd St-Jacques", "Bordeaux", "licence info");
    p.ecrirePersonne();                  // ecrirePersonne() de Etudiant

    p = new Etudiant ("Dufour", "Stéphanie",
      "rue des saules", "Lyon", "DUT info");
    p.ecrirePersonne();                  // ecrirePersonne() de Etudiant

    Personne.nbPersonnes();                              // méthode static
```

Peut-on à partir d'une référence de Personne, accéder aux attributs ou aux méthodes non redéfinies d'une classe dérivée ? Réponse : oui en forçant le type si l'objet est bien du type dans lequel on veut le convertir (sinon, il y a lancement d'une exception de type ClassCastException). Voir page 77.

```
    // À partir d'une référence de Personne,
    // accéder à un objet dérivé

    // p référence le dernier Etudiant créé ci-dessus.
    // Erreur de compilation : diplomeEnCours n'est pas un méthode
    // de la classe Personne
    //System.out.println ("Diplôme en cours : " + p.diplomeEnCours());

    // Erreur de compilation : impossible de convertir Personne
    // en Secretaire implicitement
    //Secretaire sec = p;

    // Erreur d'exécution : Exception ClassCastException :
    // p n'est pas une Secretaire (c'est un Etudiant)
    //Secretaire sec = (Secretaire) p;

    Etudiant et1 = (Etudiant) p;                  // OK p est un Etudiant
    System.out.println ("Diplôme en cours : " + et1.diplomeEnCours());
```

```
        // teste si p est un objet de la classe Etudiant
        if (p instanceof Etudiant) {
          System.out.println ("p est de la classe Etudiant");
        }
    }  // main
}  // PPPersonne2
```

3.2.4.3 Le polymorphisme de *toString()*

Si on remplace la méthode abstraite de la classe Personne par :

```
void ecrirePersonne () {
  System.out.println (toString());     // ou System.out.println (this);
};
```

et si on supprime ecrirePersonne() dans les sous-classes Secretaire, Enseignant et Etudiant, on obtient le résultat d'exécution ci-dessous. `System.out.println (this);` est une écriture simplifiée équivalente à `System.out.println (toString())`.

La méthode `ecrirePersonne()` appelée est celle unique de la classe `Personne` ; par contre la méthode `toString()` appelée dans ecrirePersonne() est celle de la classe de l'objet (`Secretaire`, `Enseignant` ou `Etudiant`) en raison de la liaison dynamique. Si plus tard, on définit une classe `Technicien`, la méthode `ecrirePersonne()` de `Personne` fera appel à la méthode `toString()` de `Technicien`, et ce sans aucune modification. Il suffira de définir une méthode toString() pour la classe Technicien. En l'absence de cette définition, c'est la méthode toString() de Personne qui sera utilisée. La méthode toString() de Personne est **polymorphe** puisque le même appel de méthode met en œuvre à l'exécution des méthodes dérivées redéfinies différentes suivant l'objet qui déclenche l'appel.

Résultats de l'exécution de PPPersonne2 :

ecrirePersonne() utilise :

```
Dupond Chantal rue des mimosas Rennes       toString() de Secretaire
  N° bureau : A123
Martin Michel bd St-Antoine Rennes          toString() de Enseignant
  spécialité : Maths
Martin Guillaume bd St-Jacques Bordeaux       toString() de Etudiant
  Diplome en cours : licence info
Dufour Stéphanie rue des saules Lyon
  Diplome en cours : DUT info

Nombre d'employés     : 4
Nombre de secrétaires : 1
Nombre d'enseignants  : 1
Nombre d'étudiants    : 2

Diplôme en cours : DUT info

P est de la classe Etudiant
```

Remarque : en Java, par défaut, la liaison est dynamique. Cependant, si la classe est déclarée `final`, c'est-à-dire qu'elle ne peut pas avoir de sous-classes, la liaison sera statique pour les méthodes de cette classe car la liaison statique est plus rapide. De même, une méthode peut être déclarée `final` auquel cas, elle ne pourra pas être redéfinie dans une sous-classe, et aura une liaison statique. En C++, les liaisons sont par défaut statiques. Il faut demander une liaison dynamique pour une méthode en la faisant précéder du mot-clé virtual.

Exercice 3.1 – Ajouter un nom pour une Pile

On dispose de la classe `Pile` définie en 2.10.2, page 78. Écrire une classe `PileNom` dérivée de `Pile` et comportant en plus, un nom de type `String`, un constructeur de `PileNom` et une méthode `listerPile()` qui écrit le nom de la `Pile` et liste les éléments de la pile. Écrire une classe PPPile de test.

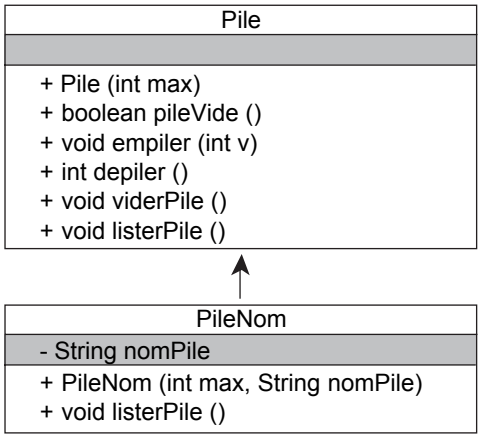


Figure 3.3 — La classe PileNom dérive de la classe Pile (héritage de classe).

3.3 LA SUPER-CLASSE OBJECT

En Java, toutes les classes dérivent implicitement d'une classe Object qui est la racine de l'arbre d'héritage. Cette classe définit une méthode toString() qui écrit le nom de la classe et un numéro d'objet. Les classes dérivées peuvent redéfinir cette méthode. Voir toString(), page 82. Puisque toute classe dérive de la classe Object, une référence de type Object peut référencer n'importe quel type d'objet : une référence de la super-classe peut être utilisée dans les classes dérivées. Voir polymorphisme d'une variable, page 103.

```
Object objet;
objet = new Secretaire ("Dupond", "Chantal",
                                  "rue des mimosas", "Rennes", "A123");
objet = new String ("Bonjour");
objet = new Complex (2, 1.50);
```

Le chapitre 4 utilise ce concept pour créer des listes d'objets (voir figure 4.3).

3.4 LA HIÉRARCHIE DES EXCEPTIONS

Les exceptions constituent un arbre hiérarchique utilisant la notion d'héritage.

```
Object                                  la classe racine de toutes les classes
  Throwable                  getMessage(), toString(), printStackTrace(), etc.
    Error                          erreurs graves, abandon du programme
    Exception                                  anomalies récupérables
        RunTimeException                          erreur à l'exécution
            ArithmeticException          division par zéro par exemple
            ClassCastException       mauvaise conversion de classes (cast)
            IndexOutOfBoundsException   en dehors des limites (tableau)
            NullPointerException                      référence nulle
            IllegalArgumentException
                NumberFormatException                 nombre mal formé
        IOException                          erreurs d'entrées-sorties
            FileNotFoundException                     fichier inconnu
```

Dans un catch, on peut capter l'exception à son niveau le plus général (Exception), ou à un niveau plus bas (FileNotFoundException par exemple) de façon à avoir un message plus approprié. On peut avoir plusieurs catch pour un même try. Voir exceptions page 74.

3.5 LES INTERFACES

Java n'autorise que l'héritage simple. Une classe ne peut avoir qu'une seule superclasse. Une certaine forme d'héritage multiple est obtenue grâce à la notion d'interface. Sur la figure 3.4, on définit de manière formelle, une classe AA qui hérite de la classe A et une classe BB qui hérite de la classe B. L'interface Modifiable permet de voir les classes AA et BB comme ayant une propriété commune, celle d'être modifiable. Une interface définit uniquement des constantes et des prototypes de méthodes.

```
interface Modifiable {
  static final int MIN = -5;
  static final int MAX = +5 ;                          // constante

  void zoomer (int n) ;                                // prototype
}  // Modifiable
```

Chacune des classes désirant être reconnues comme ayant la propriété Modifiable doit le faire savoir en implémentant l'interface Modifiable,

```
class AA extends A implements Modifiable { ... }
class BB extends B implements Modifiable { ... }
```

et en définissant les méthodes données comme prototypes dans l'interface (soit la seule méthode void zoomer (int) sur l'exemple).

On peut définir des variables de type Modifiable (on ne peut pas instancier d'objet de type Modifiable) référencant des objets des classes implémentant l'interface. Une variable Modifiable peut référencer un objet AA ou un objet BB. On peut même faire un tableau de Modifiable contenant des objets implémentant l'interface Modifiable (AA ou BB). Une variable de type Modifiable ne donne accès qu'aux méthodes de l'interface. Pour accéder à une des méthodes d'un objet AA à partir d'une variable de type Modifiable, il faut forcer le type (cast) en AA. Si le transtypage échoue (l'objet n'est pas un AA), il y a lancement d'une exception de type ClassCastException.

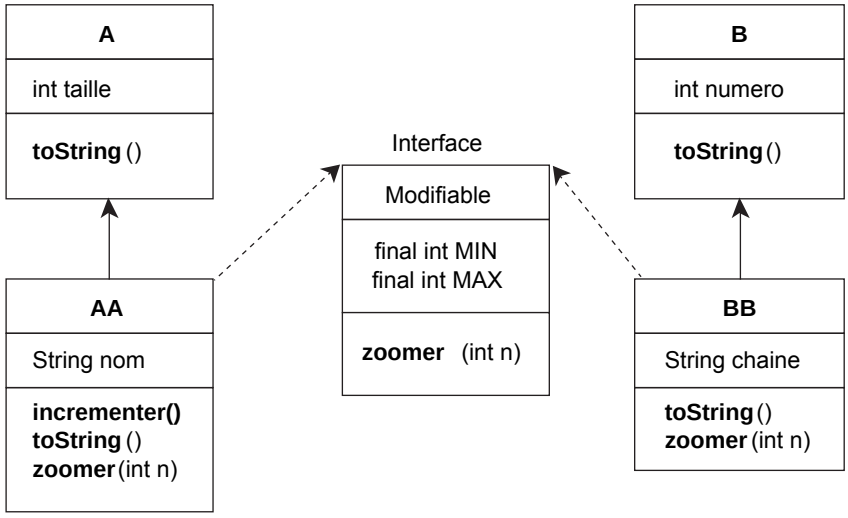


Figure 3.4 — Héritage et interface.

Le programme Java suivant permet de définir des variables de type Modifiable référençant des AA ou des BB, et de déclencher la méthode zoomer() pour des Modifiable.

```
// PPInterface.java  Programme Principal Interface

class A {
  int taille;

  A (int taille) {
    this.taille = taille;
  }

  // String des caractéristiques de A
  public String toString() {
    return "A : " + taille;
  }

}  // A

// AA hérite de A
// et implémente l'interface Modifiable
class AA extends A implements Modifiable {
```

```
  String nom;

  AA (String nom, int taille) {
    super (taille);
    this.nom = nom;
  }

  void incrementer () {
    taille++;
  }

  // String des caractéristiques de AA
  public String toString() {
    return "AA : " + super.toString() + ", AA : " + nom;
  }

  // méthode de l'interface Modifiable redéfinie pour AA
  public void zoomer(int n) {
    if (n < MIN) n = MIN;
    if (n > MAX) n = MAX;
    if (n ==0)  n = 1;
    // n > 0, on multiplie par n
    // n < 0, on divise par |n|
    taille = n > 0 ? taille*n : (int)(taille / (double) -n);
  }

}  // AA


class B {
  int valeur;

  B (int valeur) {
    this.valeur = valeur;
  }

  // String des caractéristiques de B
  public String toString() {
    return "B : " + valeur;
  }

}  // B

// BB hérite de B
// et implémente l'interface Modifiable
class BB extends B implements Modifiable {
  String chaine;

  BB (String chaine, int valeur) {
    super (valeur);
    this.chaine = chaine;
  }

  // String des caractéristiques de BB
  public String toString() {
    return "BB : " + super.toString() + ", BB : " + chaine;
  }
```

```
  // méthode de l'interface Modifiable redéfinie pour BB
  public void zoomer(int n) {
    if (n < MIN) n = MIN;
    if (n > MAX) n = MAX;
    if (n ==0)  n = 1;
    // n > 0, on multiplie par 2*n
    // n < 0, on divise par 2*|n|
    valeur = n > 0 ? valeur*2*n : (int)(valeur / (double) (-2*n));
  }

}  // BB

interface Modifiable {
  static final int MIN = -5;
  static final int MAX = +5;

  void zoomer(int n);
}  // Modifiable

public class PPInterface {                Programme Principal Interface
  public static void main (String[] args) {

    // un tableau de Modifiable contenant des AA ou des BB
    Modifiable[] tabMod = {
      new AA ("b1", 10),
      new AA ("b2", 20),
      new BB ("b3", 30),
      new BB ("b3", 50)
    };

    // on déclenche la méthode zoomer() pour les éléments du tableau
    // les AA sont divisés par 2
    // les BB sont divisés par 4
    for (int i=0; i < tabMod.length; i++) {
      tabMod[i].zoomer (-2);  // -2 : réduction
    }

    for (int i=0; i < tabMod.length; i++) {
      //System.out.println (tabMod[i].toString());
      System.out.println (tabMod[i]);
    }

    // erreur : un Modifiable ne peut pas accéder à incrementer()
    for (int i=0; i < tabMod.length; i++) {
      // erreur : incrementer n'est pas une méthode
      // de la classe Modifiable
      //tabMod[i].incrementer();
      System.out.println (tabMod[i].getClass().getName());
    }

    Modifiable m1 = tabMod[0];
    ((AA)m1).incrementer();         // il faut effectuer un transtypage
    System.out.println ("m1 : " + m1);
```

```
        Modifiable m2 = tabMod[2];                    // m2 est de type BB
        //((AA)m2).incrementer();           // Exception ClassCastException
        System.out.println ("m2 : " + m2);

    }  // main

}  // PPInterface
```

Les résultats sont les suivants :

```
                                        zoomer(-2) a modifié les valeurs
AA : A : 5, AA : b1                                                    /2
AA : A : 10, AA : b2                                                   /2
BB : B : 7, BB : b3                                                    /4
BB : B : 12, BB : b3                                                   /4
AA                  getClass().getName() fournit le nom de la classe
AA
BB
BB
m1 : AA : A : 6, AA : b1                // m1 incrémentée (de 5 à 6)
m2 : BB : B : 7, BB : b3
```

3.6 CONCLUSION

La notion d'héritage est une notion importante en programmation objet. Cette notion permet de spécialiser une classe sans avoir à tout réécrire, en utilisant sans modification, les méthodes de la super-classe si elles conviennent ou en réécrivant les méthodes qui ne conviennent plus sachant qu'une partie du traitement peut souvent être faite par les méthodes de la super-classe.

Les interfaces graphiques en Java (chapitre 5) sont organisées sous forme d'un arbre d'héritage. Tous les composants (boutons, fenêtres, zones de saisie, etc.) ont des caractéristiques communes (dimension, couleur du fond, couleur du texte, etc.). Chaque composant doit définir ses spécificités en terme d'attributs et de méthodes.

Le polymorphisme permet de définir dans la super-classe des méthodes tenant compte des spécificités des classes dérivées. En cas de redéfinition de méthodes, c'est la méthode de la classe dérivée de l'objet qui est prise en compte, et non celle de la super-classe. Les interfaces permettent de regrouper des classes ayant une ou plusieurs propriétés communes et de déclencher des traitements pour les objets des classes implémentant cette interface. Une classe ne peut hériter que d'une seule super-classe mais peut implémenter plusieurs interfaces.

Chapitre **4**

Le paquetage liste

4.1 LA NOTION DE LISTE

La liste chaînée constitue un bon exemple de l'intérêt de la programmation objet. Les classes Liste (liste simple) et ListeOrd (liste ordonnée) définies ci-dessous sont réutilisées (sans modification) dans de nombreux exemples divers au cours de ce livre. Ces classes sont rangées dans un paquetage et leur utilisation se résume à déclarer une instruction "import mdpaquetage.listes.*;" (voir paquetage, page 66). Ce chapitre est aussi un bon exemple de synthèse des notions vues préalablement : classe abstraite, héritage, polymorphisme, paquetage.

4.1.1 La classe abstraite ListeBase

Un certain nombre de méthodes sont communes à une liste ordonnée ou non ordonnée. Les méthodes *public* et communes aux deux types de listes sont répertoriées dans une classe abstraite appelée ListeBase. Voir classe abstraite en 3.2.1.

Les méthodes *public* suivantes sont définies pour un utilisateur de la classe **ListeBase** ou d'une classe dérivée :

- **ListeBase** () : constructeur de ListeBase.
- boolean **listeVide** () : la liste est-elle vide ?
- int **nbElement** () : fournit le nombre d'objets dans la liste.
- Object **extraireEnTeteDeListe** () : fournit une référence sur l'objet extrait en tête de la liste.
- Object **extraireEnFinDeListe** () : fournit une référence sur l'objet extrait en fin de la liste.

- Object **extraireNieme** (int numero) : extrait l'objet de rang numero et fournit sa référence ou null si l'élément n'existe pas. Le premier objet a le numéro 1.
- void **ouvrirListe** () : se positionne sur le premier objet de la liste.
- boolean **finList** () : a-t-on atteint la fin de la liste ?
- Object **objetCourant** () : fournit une référence sur l'objet courant. L'objet suivant (du courant) devient objet courant pour le prochain appel de objetCourant().
- void **listerListe** (String separateur) : liste les objets de la liste en insérant entre chaque objet la chaîne de caractères separateur.
- Object **chercherUnObjet** (Object objetCherche) : fournit une référence sur un objet ayant les caractéristiques (identifiant) de objetCherche.
- boolean **extraireUnObjet** (Object objet) : extraire de la liste, l'objet référencé par objet.

4.1.2 Les classes dérivées de ListeBase : Liste et ListeOrd

4.1.2.1 Classe Liste

Une Liste (liste simple non ordonnée) peut utiliser toutes les méthodes de ListeBase. L'insertion est cependant spécifique et diffère de l'insertion dans une liste ordonnée.

Les méthodes *public* suivantes sont définies pour la classe Liste :

- void **insererEnTeteDeListe** (Object objet) : insère l'objet en tête de la liste.
- void **insererEnFinDeListe** (Object objet) : insère l'objet en fin de la liste.

4.1.2.2 Classe ListeOrd

L'insertion dans une liste ordonnée requiert les méthodes spécifiques suivantes en plus des méthodes de la classe ListeBase :

- **ListeOrd** () : constructeur d'une liste ordonnée croissante (par défaut).
- **ListeOrd** (int type) : constructeur d'une liste croissante ou décroissante suivant type (qui vaut CROISSANT ou DECROISSANT).
- void **insererEnOrdre** (Object objet).

4.1.3 Polymorphisme des méthodes toString() et compareTo()

4.1.3.1 Polymorphisme de toString()

La méthode **listerListe**() de la classe ListeBase liste les objets de la liste en écrivant leurs caractéristiques. La méthode listerListe() appelle, lors de l'exécution, la méthode toString() de l'objet de la liste, si cette méthode est redéfinie dans la classe de cet objet, sinon on recherche dans l'arbre d'héritage une méthode toString(). Comme toutes les classes héritent par défaut de la classe Object, et que cette classe contient une méthode toString(), on trouvera en dernier recours cette méthode qui fournira des informations sur l'implantation de l'objet, et non sur ces caractéristiques.

Pour que listerListe() fournisse les informations désirées, il faut donc définir la méthode toString() dans la classe de l'objet. Exemple : toString() doit être définie dans la classe Personne si on veut pouvoir lister une liste de Personne avec listerListe(). Voir polymorphisme en 3.2.4.

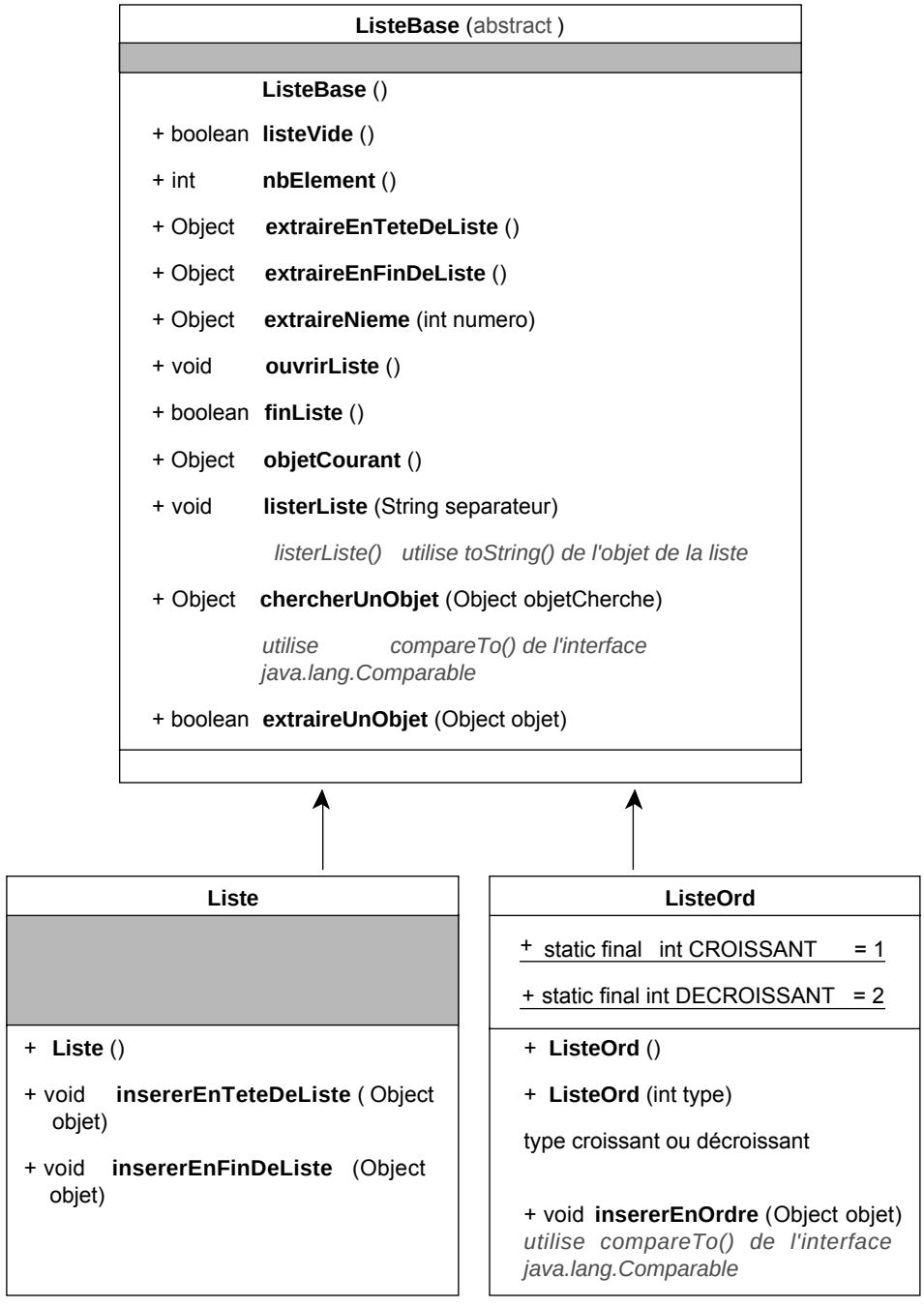


Figure 4.1 — Arbre d'héritage des listes (vue utilisateur).

4.1.3.2 Polymorphisme de compareTo()

La méthode Object **chercherUnObjet** (Object objetCherche) recherche une référence sur l'objet de la liste ayant certaines des caractéristiques de objetCherche. Pour cela, il faut être en mesure de comparer deux objets et d'identifier l'objet cherché à partir de certaines des caractéristiques de l'objet en paramètre. Là également, le polymorphisme permet de résoudre ce problème d'une manière très générale. La méthode chercherUnObjet() appellera, à l'exécution, la méthode compareTo() de la classe de l'objet de la liste. Par exemple, si la liste est une liste de Personne, la méthode chercherUnObjet() appellera la méthode compareTo() de la classe Personne.

L'interface Comparable définit le prototype de la fonction :

```
int compareTo (Object objet);
```

qui fournit :

- une valeur < 0 si l'objet courant est < à l'objet *objet* en paramètre ;
- 0 s'il y a égalité ;
- et une valeur > 0 sinon.

Sur l'exemple de la classe Personne, la classe doit implémenter l'interface Comparable en définissant la méthode compareTo() pour la classe Personne (comparaison de deux Personne).

De même, la méthode **insererEnOrdre** (Object objet) doit consulter la méthode compareTo() pour insérer un objet à sa place dans une liste ordonnée.

4.1.4 L'implémentation des classes ListeBase, Liste et ListeOrd

Les méthodes *public* de la classe étant définies, l'implémentation peut se faire de différentes façons. L'utilisateur de la classe n'a pas à connaître cette implémentation. Le développeur de la classe doit bien sûr connaître tous les détails de cette implémentation.

4.1.4.1 La classe Element

Une liste est constituée d'éléments chaînés entre eux. Chaque élément contient une référence sur un objet spécifique de l'application et une référence sur l'élément suivant de la liste. Un objet de type `Element` contient une référence sur l'objet nommé `reference` (voir Object page 105) et une référence sur l'élément suivant nommé `suivant`.

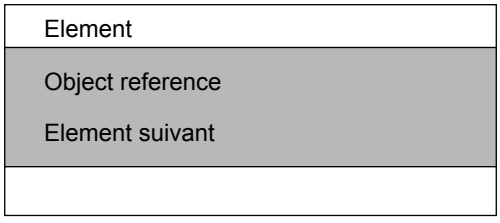


Figure 4.2 — La classe Element (du paquetage listes).

```
// Element.java  définit un élément de liste contenant
//     une référence sur l'objet de la liste et
//     une référence sur l'élément suivant de la liste

package mdpaquetage.listes;

class Element {       // visibilité = le paquetage listes
  Object  reference;  // une référence sur un Object
  Element suivant;    // une référence sur un Element
}
```

4.1.4.2 La classe ListeBase

Dans l'implémentation choisie, une liste d'éléments est repérée par une tête de liste constituée de trois références sur le premier et le dernier éléments et sur le prochain élément à traiter lors du parcours, et d'un entier indiquant le nombre d'éléments dans la liste.

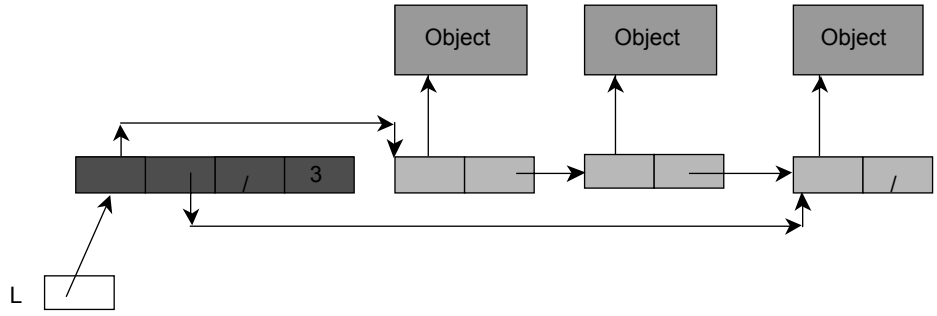


Figure 4.3 — Une liste d'objets.

4.1.4.3 La classe ListeBase en Java

Le programme Java de la classe ListeBase est le suivant :

```
// ListeBase.java  Cette classe de gestion de listes est générique;
//                 elle gère des listes d'objets avec tête de liste.

package mdpaquetage.listes;

public abstract class ListeBase {
  protected Element premier;        // visibilité = paquetage listes
  protected Element dernier;        // et classes dérivées
  protected Element courant;
  protected int     nbElt;          // nombre d'objets dans la liste

// constructeur d'une Liste vide
ListeBase () {
premier = null;
dernier = null;
courant = null;
nbElt   = 0;
  }
```

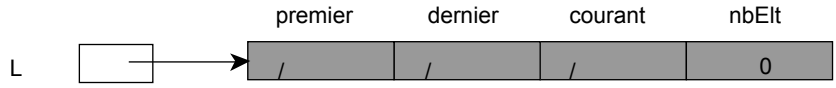


Figure 4.4 — Une liste vide.

```
// la liste est-elle vide ?
public boolean listeVide () {
return nbElt == 0;
}

// fournir le nombre d'objets dans la liste
public int nbElement () {
return nbElt;
}
```

Les méthodes suivantes permettent d'extraire un objet de la liste en tête ou en fin de liste :

```
// extraire l'objet en tête de la liste
public Object extraireEnTeteDeListe () {
    Element extrait = premier;                    // null si liste vide
    if (!listeVide()) {
      premier = premier.suivant;
      nbElt--;
      if (premier == null) dernier = null;        // liste devenue vide
    }
    return extrait != null ? extrait.reference : null;
}

// extraire l'objet en fin de la liste
public Object extraireEnFinDeListe () {
    Object extrait;
    if (listeVide()) {
      extrait = null;
      } else if (premier == dernier) {              // un seul élément
      extrait = premier.reference;
      premier = null;
      dernier = null;
      nbElt   = 0;
         } else {                              // au moins deux éléments
      Element element = premier;                      // element courant
      while (element.suivant != dernier) element = element.suivant;
      extrait = extraireApres (element);
      }
      return extrait;
}
```

La méthode suivante permet d'extraire un objet à partir de son rang dans la liste :

```
// fournir la référence de l'objet numero; null s'il n'existe pas.
// Le premier objet a le numéro 1;
public Object extraireNieme (int numero) {
  Object extrait;                       // référence sur l'objet extrait
  if (numero <= 0 || numero > nbElt) {
```

```
        extrait = null;
        } else if (numero == 1) {
        extrait = extraireEnTeteDeListe();
            } else {
        Element element = premier;
        for (int j=1; j < numero-1; j++) element = element.suivant;
        extrait = extraireApres (element);
        }
        return extrait;
    }
```

Les méthodes suivantes permettent de parcourir la liste :

```
    // se positionner sur le premier objet de la liste
    public void ouvrirListe () {
        courant = premier;
    }

    // A-t-on atteint la fin de la liste ?
    public boolean finListe () {
        return courant == null;
    }

    // fournir l'objet courant de la liste,
    // et se positionner sur l'objet suivant qui devient le courant
    public Object objetCourant () {
        Element element = courant;
        if (courant != null) {
          courant = courant.suivant;
        }
        return element != null ? element.reference : null;
    }
```

On parcourt la liste en écrivant les caractéristiques de chacun des objets de la liste à l'aide de la méthode **toString() de la classe de l'objet référ encé** (voir polymorphisme page 104). La méthode listerListe() peut fonctionner pour des **objets inconnus à ce jour** mais qui définiront une méthode toString(). Les éléments sont séparés par la chaîne `separateur` qui peut être par exemple, un espace, une virgule suivie d'un espace, ou un retour à la ligne ("\n").

```
    // lister la liste
    public void listerListe (String separateur) {
      if (listeVide()) {
          System.out.println ("Liste vide");
      } else {
        ouvrirListe();
        while (!finListe()) {
            Object objet = objetCourant();
            System.out.print (objet);                // ou objet.toString()
            if (!finListe()) System.out.print (separateur);
        }
      }
    } // listerListe

  } // class Liste
```

La méthode **chercherUnObjet()** permet d'obtenir une référence sur l'objet de la liste ayant les caractéristiques de l'objet en paramètre. Les caractéristiques à définir pour objetCherche sont celles utilisées par la méthode compareTo() de l'objet et qui constituent l'identifiant de l'objet (le nom et prénom d'une personne par exemple).

```
// fournir une réérence sur l'objet "objetCherche" de la liste;
  // null si l'objet n'existe pas
  public Object chercherUnObjet (Object objetCherche) {
    boolean trouve = false;
    Object  objet  = null;      // référence courante
    ouvrirListe ();
    while (!finListe () && !trouve) {
      objet  = objetCourant ();
      // interface java.lang.Comparable
      trouve = ((Comparable)objet).compareTo
                                        ((Comparable)objetCherche) == 0;
  }
  return trouve ? objet : null;
}
```

La méthode extraireUnObjet() extrait de la liste l'objet dont la référence est donnée en paramètre.

```
// extraire de la liste, l'objet référencé par objet
public boolean extraireUnObjet (Object objet) {
  Element precedent = null;
  Element ptc       = null;

  // repère l'élement précédent
  boolean trouve = false;
  ouvrirListe ();
  while (!finListe () && !trouve) {
    precedent = ptc;
    ptc       = elementCourant ();
    trouve = (ptc.reference == objet) ? true : false;
  }
  if (!trouve) return false;

  Object extrait = extraireApres (precedent);
  return true;
}
```

Les méthodes suivantes ont un attribut **protected**. Elles ne sont utilisables que dans le paquetage listes ou dans les classes dérivées de ListeBase.

```
// insérer objet en tête de la liste
protected void insererEnTeteDeListe (Object objet) {
  Element nouveau   = new Element();
  nouveau.reference = objet;
  nbElt++;
  nouveau.suivant   = premier;
```

```
    premier           = nouveau;
    if (dernier == null) dernier = nouveau;
  }

  // insérer objet après precedent dans la liste;
  // si precedent est null, insérer en tête de liste
  protected void insererApres (Element precedent, Object objet) {
    if (precedent == null) {
      insererEnTeteDeListe (objet);
    } else {
      Element nouveau   = new Element();
      nouveau.reference = objet;
      nbElt++;
      nouveau.suivant   = precedent.suivant;
      precedent.suivant = nouveau;
      if (precedent == dernier) dernier = nouveau;
    }
  }

  // insérer objet en fin de la liste
  protected void insererEnFinDeListe (Object objet) {
    insererApres (dernier, objet);
  }

  // extraire l'élément de la liste se trouvant après precedent;
  // si precedent vaut null, on extrait le premier de la liste;
  // retourne null si l'élément à extraire n'existe pas
  protected Object extraireApres (Element precedent) {
    Element extrait;
    if (precedent == null) {
      return extraireEnTeteDeListe();
    } else {
      extrait = precedent.suivant;
      if (extrait != null) {
        precedent.suivant = extrait.suivant;
        nbElt--;
        if (extrait == dernier) dernier = precedent;
      }
      return extrait != null ? extrait.reference : null;
    }
  }

  // fournir une référence sur l'élément courant (pas l'objet)
  // de la liste,
  // et se positionner sur le suivant qui devient le courant
  protected Element elementCourant () {
    Element element = courant;
    if (courant != null) {
      courant = courant.suivant;
    }
    return element;
  }
```

4.1.4.4 L 'implémentation de la classe Liste

Cette classe hérite de la classe ListeBase. Elle définit les méthodes spécifiques aux listes simples (non-ordonnées) d'insertion en tête de liste et en fin de liste.

```
// Liste.java  classe de gestion de listes simples génériques

package mdpaquetage.listes;

public class Liste extends ListeBase {

  // insérer objet en tête de la liste
  public void insererEnTeteDeListe (Object objet) {
    super.insererEnTeteDeListe (objet);
  }

  // insérer objet en fin de la liste
  public void insererEnFinDeListe (Object objet) {
    super.insererEnFinDeListe (objet);
  }

} // class Liste
```

4.1.4.5 L'implémentation de la classe ListeOrd

Les listes ordonnées sont des listes qui ont une propriété supplémentaire : les objets de la liste sont dans un ordre croissant ou décroissant. Le principe de l'héritage convient parfaitement pour réutiliser les propriétés de ListeBase et y ajouter la nouvelle propriété de liste ordonnée. Le critère d'ordre varie suivant la classe des objets de la liste.

Mis à part ce critère d'ordre, l'insertion dans une liste ordonnée croissante se fait toujours de la même façon. Il faut trouver une place dans la liste telle que l'élément à insérer soit entre un plus petit et un plus grand si l'ordre est croissant. Le critère d'ordre est défini par une méthode compareTo() définie dans la classe des objets de la liste, qui compare l'objet courant et l'objet en paramètre, et fournit un entier avec les conventions suivantes : 0 si les deux objets sont égaux, < 0 si l'objet courant est inférieur à l'objet en paramètre, > 0 sinon.

On peut définir une méthode très générale d'insertion dans une liste ordonnée à condition d'être sûr de disposer de la méthode **compareTo**() pour comparer les objets de la liste. L'interface Comparable est justement définie pour cela. Un objet d'une classe implémentant l'interface Comparable peut être considéré de type Comparable. Toute classe implémentant l'interface Comparable doit redéfinir la méthode compareTo(). Voir interface page 106.

```
// l'interface Comparable du paquetage standard java.lang

public interface Comparable {
  int compareTo (Object objet);      // méthode d'interface donc public
}
```

L'instruction suivante :

```
boolean ordre = ((Comparable)objet1).compareTo ((Comparable)objet2) < 0;
```

- pour des objets Personne (dont la classe implémente Comparable), force le type des objets à Comparable et déclenche la méthode compareTo() de Personne ;
- pour des objets NbEntier (dont la classe implémente Comparable), force le type des objets à Comparable et déclenche la méthode compareTo() de NbEntier.

Le booléen ordre précédent est vrai si objet1 < à objet2.

La classe ListeOrd dérive de ListeBase et ajoute une propriété de liste ordonnée. La liste peut être ordonnée en ordre croissant ou en ordre décroissant suivant le type en paramètre du constructeur ListeOrd() ; par défaut, la liste est ordonnée croissante. La méthode enOrdre() indique si les deux objets en paramètres sont en ordre (suivant l'ordre de la liste). La méthode insererEnOrdre() insère un objet dans une liste ordonnée.

```
// ListeOrd.java  liste ordonnée

// classes Liste et ListeOrd dans le même paquetage
package mdpaquetage.listes;

public class ListeOrd extends ListeBase {
  public static final int CROISSANT   = 1;
  public static final int DECROISSANT = 2;
  private boolean ordreCroissant;

  public ListeOrd () {
    ordreCroissant = true;
  }

  public ListeOrd (int type) {
    ordreCroissant = true;
    if (type == DECROISSANT) ordreCroissant = false;
  }

  // enOrdre si objet1 < objet2 (ordre croissant)
  // enOrdre si objet1 > objet2 (ordre décroissant)
  private boolean enOrdre (Object objet1, Object objet2) {
    boolean ordre = false;
    try {
      ordre = ((Comparable)objet1).compareTo ((Comparable)objet2) < 0;
    } catch (Exception e) {  // ClassCastException
      System.out.println
                        ("erreur sur la redéfinition de compareTo\n");
      System.exit (1);
    }
    if (!ordreCroissant) ordre = !ordre;
    return ordre;
  }  // enOrdre
```

L'insertion dans une liste ordonnée n'ayant pas d'élément est en fait une insertion en tête de liste. Si l'objet à insérer (en paramètre) est plus petit que le premier élément, c'est aussi une insertion en tête de liste. Sinon, il faut parcourir la liste à la recherche d'un emplacement tel que l'élément à insérer soit en ordre avec son élément à droite

repéré par element.reference. On garde une référence sur l'élément précédent de la liste pour pouvoir insérer après cet élément.

```
  public void insererEnOrdre (Object objet) {
    Element precedent = null;
    Element element;

    if (listeVide ()) {   // liste vide
      insererEnTeteDeListe (objet);
    } else {
      element = premier;  // droit d'accès du paquetage listes
      if ( enOrdre (objet, element.reference) ) {
        insererEnTeteDeListe (objet);
      } else {    // insertion en milieu ou fin de liste
        boolean trouve = false;
        while ((element != null) && !trouve) {
          precedent = element;
          element   = element.suivant;
          if (element != null)
                          trouve = enOrdre (objet, element.reference);
        }
        // insertion en milieu ou fin de liste après precedent
        // Liste et ListeOrd dans le meme paquetage
        insererApres (precedent, objet);
      }
    }
  }  // insererEnOrdre

}  // class ListeOrd
```

4.2 L'UTILISATION DE LA CLASSE LISTE POUR UNE LISTE DE PERSONNES

La gestion d'une liste de personnes est facile à mettre en œuvre en utilisant la classe Liste définie précédemment. Une Personne est caractérisée seulement par son nom et son prénom. On définit deux méthodes : un constructeur et la méthode toString() fournissant une chaîne de caractères composée du nom et du prénom.

La classe Personne s'écrit comme suit en Java :

```
// PPListePersonnes.java  gestion d'une liste de personnes
//                     en utilisant la classe Liste du paquetage listes

import mdpaquetage.listes.*;      // import du paquetage sur les listes

class Personne {  // une personne
  private String nom;
  private String prenom;

  Personne (String nom, String prenom) {
    this.nom    = nom;
```

```
    this.prenom = prenom;
  }

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

}  // class Personne
```

On ajoute un nom de liste pour chacune des listes de personnes en étendant la classe Liste :

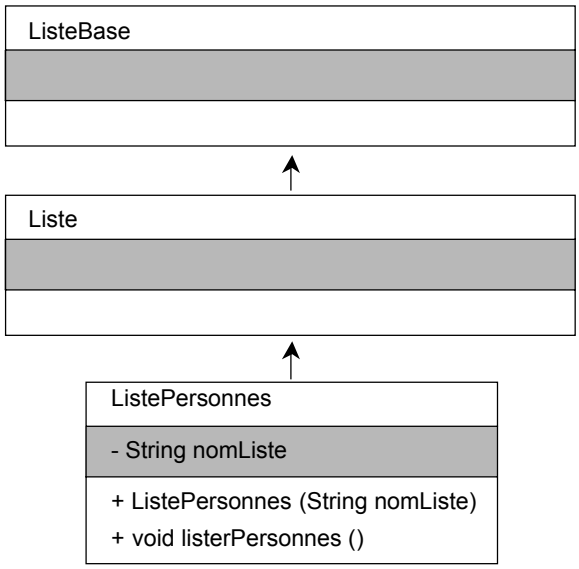


Figure 4.5 — La classe ListePersonnes hérite de la classe Liste.

```
// une liste de personnes comportant en plus un nom de liste
class ListePersonnes extends Liste {
  private String nomListe;

  ListePersonnes (String nomListe) {
    super();           // appel du constructeur de Liste (voir page 98)
                                  // inséré d'office; pourrait être omis
    this.nomListe = nomListe;
  }

  // lister les noms des personnes de la liste
  void listerPersonnes () {
    System.out.println (
      "\n" + nomListe + " (" + nbElement() + " éléments) : "
    );
    listerListe ("\n");    // de la super-classe Liste
                           // qui utilise toString() de Personne
  }

}  // ListePersonnes
```

La classe permettant de tester les deux classes Personne et ListePersonnes :

```
class PPListePersonnes {          // Programme Principal Liste Personnes

  public static void main (String[] args) {
    ListePersonnes list1 = new ListePersonnes ("Etudiants");

    if (list1.listeVide()) {
      System.out.println ("Liste de personnes vide");
    }

    Personne p1 = new Personne ("Dupond", "Jules");
    Personne p2 = new Personne ("Durand", "Jacques");
    list1.insererEnFinDeListe (p1);
    list1.insererEnFinDeListe (p2);

    list1.listerPersonnes();

    Personne extrait = (Personne) list1.extraireEnTeteDeListe();
    //Personne extrait = (Personne) list1.extraireNieme (2);
    System.out.println ("\n\nExtrait : " + extrait);
    extrait = (Personne) list1.extraireEnTeteDeListe();
    System.out.println ("Extrait : " + extrait);
    list1.listerPersonnes();
  }  // main

}  // class PPListePersonnes
```

Remarque : list1.extraireEnTeteDeliste() fournit une référence sur un Object. On doit forcer le type de l'objet en un type Personne. Si l'objet n'est pas de type Personne, il y aura lancement d'une exception.

Un exemple de résultats d'exécution :

```
Liste de personnes vide
Etudiants (2 éléments) :
Dupond Jules
Durand Jacques
Extrait : Dupond Jules
Extrait : Durand Jacques
Etudiants (0 éléments) :
Liste vide
```

4.3 L'UTILISATION DE LA CLASSE LISTEORD (PERSONNE, NBENTIER)

Disposant du paquetage listes contenant les classes ListeBase et ListeOrd, il est très facile de gérer des listes ordonnées de n'importe quel type d'objet. Le programme principal suivant effectue la création d'une liste ordonnée croissante de Personne et d'une liste ordonnée décroissante de nombres entiers.

```
// PPListOrd.java  Programme Principal Liste Ordonnée
//                 de Personne ou de NbEntier

import mdpaquetage.listes.*;
```

La classe **Personne** :

Si la classe Personne est déjà définie et si on ne veut (peut) pas la modifier, on peut dériver une nouvelle classe PersonneC qui implémente la méthode compareTo() au lieu de définir la classe Personne comme ci-après.

```
class Personne implements Comparable {
  private String nom;
  private String prenom;

  Personne (String nom, String prenom) {
    this.nom    = nom;
    this.prenom = prenom;
  }

  public String toString () {
    return "Personne : " + nom + " " + prenom;
  }
```

Le critère de tri pour les personnes est la concaténation de leur nom et prenom.

```
  // compare nom et prenom des deux objets
  public int compareTo (Object objet) {
    Personne p1 = this;                              // objet courant
    Personne p2 = ((Personne) objet);              // objet en paramètre
    String nom1 = p1.nom + " " + p1.prenom;
    String nom2 = p2.nom + " " + p2.prenom;
    return nom1.compareTo (nom2);               // comparaison de String
  }

}  // Personne
```

La classe NbEntier :

```
class NbEntier implements Comparable {
  int valeur;

  NbEntier (int valeur) {
    this.valeur = valeur;
  }

  public String toString () {
    return "" + valeur;
  }

  // compare la valeur des deux objets
  public int compareTo (Object objet) {
    NbEntier n1 = this;                              // objet courant
    NbEntier n2 = (NbEntier) objet;                // objet en paramètre
    if (n1.valeur < n2.valeur)  return -1;
    if (n1.valeur == n2.valeur) return 0;
    return 1;
  }

}  // class NbEntier
```

La classe utilisatrice est très simple. Il suffit de déclarer un objet de type ListeOrd et de lui ajouter des éléments. La méthode listerListe() de la classe ListeBase est utilisable sans modification pour lister les Personne et les nombres entiers, l'écriture des caractéristiques des objets étant assurée par la méthode toString() de la classe de l'objet (polymorphisme).

```
class PPListOrd {

  public static void main (String[] args) {

    // création d'une liste ordonnée croissante de Personne
    ListeOrd l1 = new ListeOrd();
    l1.insererEnOrdre (new Personne ("Dupond",   "Albert"));
    l1.insererEnOrdre (new Personne ("Duval",    "Jacques"));
    l1.insererEnOrdre (new Personne ("Dupré",    "Sébastien"));
    l1.insererEnOrdre (new Personne ("Dufour",   "Monique"));
    l1.insererEnOrdre (new Personne ("Dumoulin", "Nathalie"));
    l1.insererEnOrdre (new Personne ("Dupond",   "Lydie"));

    System.out.println ("\nlisterListe :");
    System.out.println ("nombre d'éléments : " + l1.nbElement());
    l1.listerListe ("\n");                              // de ListeBase
    System.out.println ("\n");

    // création d'une liste ordonnée décroissante de NbEntier
    ListeOrd l2 = new ListeOrd (ListeOrd.DECROISSANT);
    l2.insererEnOrdre (new NbEntier (25));
    l2.insererEnOrdre (new NbEntier (15));
    l2.insererEnOrdre (new NbEntier (55));
    l2.insererEnOrdre (new NbEntier (45));
    l2.insererEnOrdre (new NbEntier (25));
    l2.insererEnOrdre (new NbEntier (5));
    l2.insererEnOrdre (new NbEntier (17));

    System.out.println ("\nlisterListe :");
    l2.listerListe ("\n");                              // de ListeBase
    System.out.println ("\n");

  }  // main

}  // class PPListOrd
```

Exemples de résultats :

```
listerListe :                  liste ordonnée croissante de Personne
nombre d'éléments : 6
Personne : Dufour Monique
Personne : Dumoulin Nathalie
Personne : Dupond Albert
Personne : Dupond Lydie
Personne : Dupré Sébastien
Personne : Duval Jacques
```

```
listerListe :         liste ordonnée décroissante de nombres entiers
55
45
25
25
17
15
5
```

4.4 L'UTILISATION DE LA CLASSE LISTEORD POUR UNE LISTE DE MONÔMES (POLYNÔMES)

Un polynôme d'une variable est une liste ordonnée de monômes. Chaque monôme est caractérisé par un coefficient et un exposant. Exemple de polynôme :

$$P(x) = 2\,x^2 + 3.5\,x + 5 = 2\,x^2 + 3.5\,x^1 + 5\,x^0$$

est constitué des monômes de coefficient et exposant suivants : (2,2), (3.5,1), (5,0). La classe Monome est schématisée sur la figure 4.6. Les attributs sont accessibles dans le paquetage et on définit deux méthodes : un constructeur et la redéfinition de la méthode toString() qui fournit une chaîne de caractères représentant les caractéristiques du monôme.

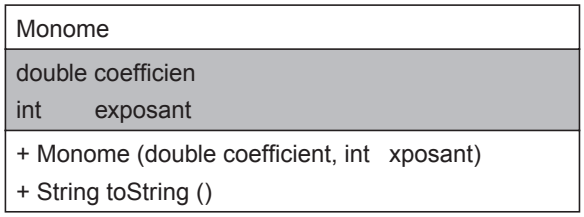


Figure 4.6 — La classe Monome.

D'où le programme Java qui définit la classe Monome :

```
// PPPolynome.java  Programme Principal sur les Polynômes

import mdpaquetage.listes.*;  // import du paquetage listes

class Monome implements Comparable {
  // visibilité des attributs = paquetage courant
  double coefficient;   // coefficient du monôme
  int    exposant;      // exposant

  // le constructeur de Monome
  Monome (double coefficient, int exposant) {
    this.coefficient = coefficient;
    this.exposant    = exposant;
  }
```

```
  // la méthode toString() fournit les caractéristiques du monôme
  public String toString() {
    if (exposant == 0) {
      return coefficient + "";     // conversion en String voir page 82
    } else if (exposant == 1) {
      return coefficient + " x";
    } else {
      return coefficient + " x**" + exposant;
    }
  }

  // la méthode de l'interface Comparable
  public int compareTo (Object objet) {
    Monome m1 = (Monome) this;
    Monome m2 = (Monome) objet;
    if (m1.exposant <  m2.exposant) return -1;
    if (m1.exposant == m2.exposant) return 0;
    return 1;
  }

}  // classe Monome
```

Chaque polynôme est une liste de monômes à laquelle on souhaite ajouter un nom. La classe Polynome est schématisée sur la figure 4.7 ; elle hérite de la classe ListeOrd et définit un nouvel attribut nomPoly. Les listes sont ordonnées suivant la valeur décroissante des exposants.

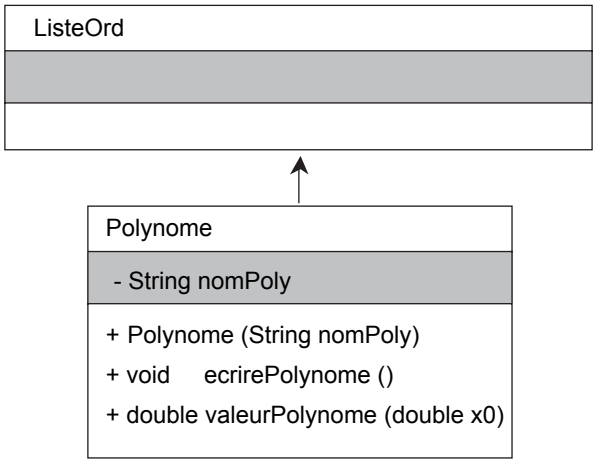


Figure 4.7 — La classe Polynome hérite de la classe ListeOrd et ajoute ses spécificités.

D'où le programme Java pour la classe Polynome :

```
// un polynôme = une liste de monômes
class Polynome extends ListeOrd {
  private String nomPoly;                          // le nom du polynôme
```

```
  // constructeur de Polynome
  Polynome (String nomPoly) {
    super (DECROISSANT);                  // liste ordonnée décroissante
    this.nomPoly = nomPoly;
  }

  // écrire les monômes de la liste
  void ecrirePolynome () {
    System.out.print (nomPoly + " : ");
    listerListe (" + ");                        // de la classe ListeBase
    System.out.println();
  }

  // calculer la valeur du polynôme pour x0
  double valeurPolynome (double x0) {
    double valeur = 0;
    if (listeVide()) {
      System.out.println ("polynôme vide");
    } else {
      ouvrirListe();
      while (!finListe()) {
        Monome monome = (Monome) objetCourant();
        valeur += monome.coefficient*Math.pow (x0, monome.exposant);
      }
    }
    return valeur;
  }

}  // classe Polynome
```

La classe **PPPolyno** suivante permet de tester les classes Monome et Polynome.

```
// Programme Principal des Polynômes
class PPPolynome {

  public static void main (String[] args) {
    Polynome p1 = new Polynome ("PA");
    if (p1.listeVide()) {
      System.out.println ("p1 vide");
    }

    p1.insererEnOrdre (new Monome (3.5, 1));
    p1.insererEnOrdre (new Monome (2,   2));
    p1.insererEnOrdre (new Monome (5,   0));

    p1.ecrirePolynome();

    System.out.println ("p1(2) = " + p1.valeurPolynome(2));

    Monome extrait = (Monome) p1.extraireEnTeteDeListe();
    System.out.print ("\nExtrait : " + extrait + "\n");

    p1.ecrirePolynome();
  }  // main

}  // class PPPolynome
```

Remarque : p1.extraireEnTeteDeliste() fournit une référence sur un Object. On doit forcer le type de l'objet en un type Monome. Si l'objet n'est pas de type Monome, il y aura lancement d'une exception.

Exemple de résultats d'exécution :

```
p1 vide
PA : 2.0 x**2 + 3.5 x + 5.0
p1(2) = 20.0

Extrait : 2.0 x**2
PA : 3.5 x + 5.0
```

Remarque : si les classes Monome et Polynome ne sont pas dans le même paquetage, ou si `coefficient` et `exposant` sont déclarés `private`, l'instruction suivante de la méthode valeurPolynome() :

```
valeur += monome.coefficient*Math.pow(x0,monome.exposant);
```

déclenchera un message d'erreur car ces variables sont alors inaccessibles à partir de la référence `monome` d'un objet de type `Monome`. Il faudrait dans ce cas écrire une méthode *public* fournissant le `coefficient`, et une autre l'`exposant`. Voir droits d'accès page 67.

Le programme ci-dessus teste les classes Monome, Polynome et PPPolynome dans le même paquetage (par défaut). Pour plus de généralités, il faudrait mettre Monome et Polynome dans un paquetage `polynome` en ajoutant public aux classes et aux méthodes. La classe PPPolynome devrait importer ce paquetage `polynome`.

4.5 L'UTILISATION DES CLASSES LISTE ET LISTEORD POUR UNE LISTE DE CARTES

On veut simuler un jeu de 52 cartes sur ordinateur. Pour cela, on enregistre chaque carte dans une liste en mémorisant sa valeur de 1 à 13 et sa couleur de 1 à 4. On constitue la classe Carte comme indiqué sur la figure 4.8.

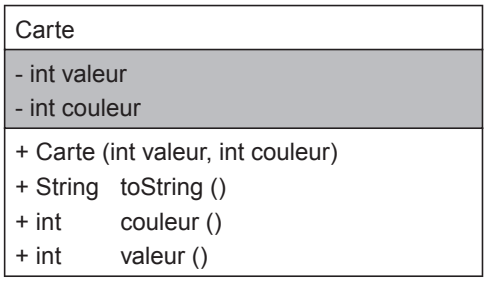


Figure 4.8 — La classe Carte.

Un paquet de cartes peut être schématisé à l'aide des méthodes suivantes de la classe **PaquetCartes** (voir figure 4.9)

- void **insererEnFinDePaquet** (Carte c) : insère la carte c en fin du paquet de cartes.
- void **listerCartes** () : liste les cartes en les faisant précéder de leur numéro (de 1 à n).
- void **creerTas** () : crée un tas de 52 cartes rangées par couleur et valeur.
- PaquetCartes **battreLesCartes** () : crée un paquet de cartes battues.
- void **distribuerLesCartes** (ListeOrd[] joueur) : crée un tableau de quatre listes ordonnées représentant les cartes des quatre joueurs après distribution des cartes. Les cartes sont ordonnées d'abord sur la couleur puis ensuite sur la valeur.

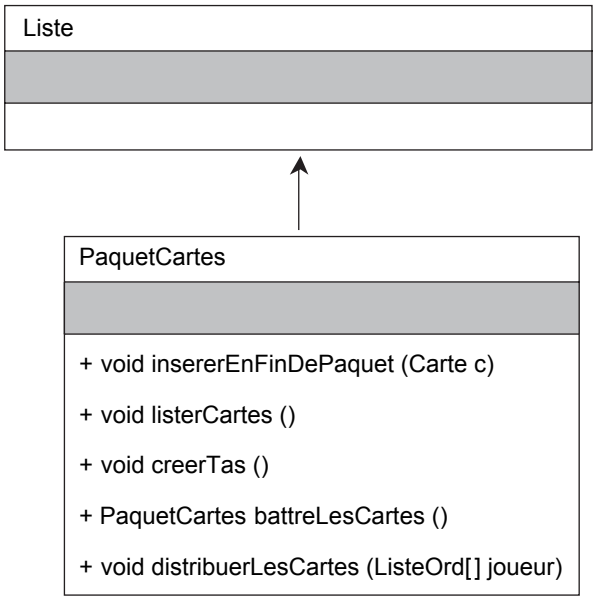


Figure 4.9 — La classe PaquetCartes hérite de la classe Liste.

Exercice 4.1 – Les cartes

> Coder et tester en Java, la classe Carte et la classe PaquetCartes schématisée sur la figure 4.9.

4.6 LA NOTION DE LISTES POLYMORPHES

Un des grands principes de la programmation objet concerne le polymorphisme. Une même variable peut référencer des objets de classes différentes dès lors que les objets appartiennent à la classe ou aux sous-classes (voir page 103). Sur la figure 4.10, on définit les classes `Personne` et `Chat` (comme sous-classes de `Object` par défaut). La variable `vivant` de type `Object` peut référencer un objet de type `Personne` ou un

objet de type `Chat`. Si la variable `vivant` référence une Personne, les méthodes sont d'abord cherchées dans la classe Personne ; c'est le cas pour l'instruction suivante System.out.println (vivant) ; qui fait appel dans ce cas à toString() de Personne. Par contre, si `vivant` référence un Chat, la même instruction déclenchera toString() de Chat. Voir 3.2.4.3, page 104. La variable `vivant` est polymorphe et peut référencer n'importe quel Object puisque tous les objets dérivent de Object.

```
Object   vivant;           // une référence d'un objet dérivé de Object
Personne p = new Personne ("Dupré", "Roland");
System.out.println (vivant);    // utilise toString() de Personne
vivant = new Chat ("Pastille"); // vivant référence un Chat
System.out.println (vivant);    // utilise toString() de Chat
```

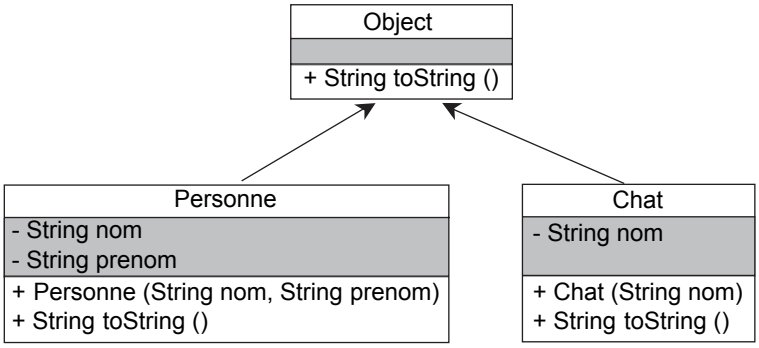


Figure 4.10 — Les classes Personne et Chat héritent par défaut de la classe Object.

On code en Java les classes Personne et Chat. Une Personne a un nom et un prénom. Un chat a seulement un nom.

```
// Polymorphe.java  liste polymorphe de Personne et de Chat

import mdpaquetage.listes.*;  // classe Liste

class Personne {
  private String nom;
  private String prenom;

  Personne (String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }

  public String toString () {
    return "Personne : " + nom + " " + prenom;
  }
}  // Personne

class Chat {
  private String nom;
```

```
  Chat (String nom) {
    this.nom = nom;
  }

  public String toString () {
    return "Chat : " + nom;
  }

}  // Chat
```

La classe **Polymorphe** suivante crée une liste contenant des objets de différentes classes dès lors que ces classes sont des classes dérivées de la classe `Object`. La liste `l1` contient des objets `Personne` et des objets `Chat`.

```
public class Polymorphe {

  public static void main (String[] args) {
    // polymorphisme de la variable vivant
    Object   vivant;    // une référence d'un Object (pas une instance)
    Personne p = new Personne ("Dupré", "Roland");
    vivant = p;                        // vivant référence une Personne
    System.out.println (vivant);      // utilise toString() de Personne
    vivant = new Chat ("Pastille");         // vivant référence un Chat
    System.out.println (vivant);          // utilise toString() de Chat
    System.out.println ();

    // liste polymorphe contenant des Personne et des Chat
    Liste l1    = new Liste();
    Personne p1 = new Personne ("Dupond", "Albert");
    l1.insererEnFinDeListe (p1);

    Chat c1 = new Chat ("Minette");
    l1.insererEnFinDeListe (c1);

    l1.insererEnFinDeListe (new Personne ("Dufour", "Marie"));
    l1.insererEnFinDeListe (new Chat ("Minou"));

    l1.listerListe ("\n");                        // de la classe Liste
  }  // main

}  // class Polymorphe
```

Résultats d'exécution :

```
Personne : Dupré Roland
Chat : Pastille
Personne : Dupond Albert          la liste des Personne et des Chat
Chat : Minette
Personne : Dufour Marie
Chat : Minou
```

Remarque : la méthode `void listerListe (String separateur)` de la classe `Liste` liste les caractéristiques des divers objets de la liste et ce, grâce à la liaison dynamique (voir 3.2.4, page 102). L'instruction System.out.print

(objet) ; de `listerListe()` (voir 4.1.4.3, page 117) de la classe `Liste` déclenche la méthode `toString()` de la classe de l'objet référencé par `objet`. Si `objet` référence un objet `Personne`, c'est la méthode `toString()` de `Personne` qui est activée ; si `objet` référence un objet `Chat`, c'est la méthode `toString()` de `Chat` qui est exécutée.

4.7 CONCLUSION

L'héritage est un notion qui implique des concepts différents de la programmation classique : surcharge de méthodes, redéfinition de méthodes, liaison dynamique, interfaces. On peut enrichir ou spécialiser une classe en ajoutant des attributs ou des méthodes à une classe existant déjà.

La liste est un bon exemple où l'utilisation de l'héritage permet d'avoir une classe ListeBase répertoriant les attributs et les méthodes communes à tous types de listes. Elle permet de définir très simplement les classes dérivées Liste et ListeOrd réutilisables dans différentes applications (liste de personnes, polynômes, cartes à jouer). Une liste ordonnée est un cas particulier de liste qu'il faut spécialiser par dérivation. La classe ListeBase est très générale mais a pourtant un "trou" qu'il faut combler dans l'application : il faut redéfinir la méthode toString() utilisée par listerListe() pour indiquer les caractéristiques de l'objet à écrire (une méthode toString() est définie par défaut dans la classe Object). De même l'utilisation d'une liste ordonnée a aussi un "trou" à combler : il faut impérativement implémenter la méthode de comparaison des objets à insérer dans la liste.

Pour un utilisateur non familier avec la programmation objet, il est toujours un peu surprenant de devoir définir dans les classes dérivées des méthodes qu'on n'appelle jamais. Le mystère réside dans le polymorphisme de l'appel des méthodes.

Chapitre **5**

Les interfaces graphiques

5.1 L'INTERFACE GRAPHIQUE JAVA AWT

Une interface graphique est un ensemble de **composants** graphiques permettant à un utilisateur de communiquer avec un logiciel. Les éléments graphiques de l'interface Java sont définis dans le paquetage (package) **AWT** (abstract window toolkit). AWT est une librairie de classes graphiques portables s'exécutant sans recompilation sur tout ordinateur disposant d'un interpréteur de code Java (machine virtuelle Java). Cependant, la machine virtuelle s'appuie en partie sur les ressources graphiques du système d'exploitation.

L'apparence des fenêtres et boutons diffère d'un système d'exploitation à l'autre avec AWT. Il existe une autre librairie d'interfaces graphiques appelée **Swing** entièrement autonome, qui ne dépend pas du système d'exploitation, mais qui peut prendre l'aspect de tel ou tel système d'exploitation à la demande (voir chapitre 6).

5.1.1 Les classes Point, Dimension, Rectangle, Color, Cursor, Font, Graphics

Les classes Point, Dimension, Rectangle Color, Cursor, Font et Graphics sont souvent utilisées dans les exemples suivants. Ces méthodes utilisées dans la suite de ce livre sont décrites succinctement ci-après.

La classe **Point** définit un point repéré par ses attributs entiers x et y sur un plan. **x** et **y** sont **public** ; un Point peut être construit à partir de deux paramètres indiquant x et y du Point, ou à partir d'un autre Point. En l'absence de paramètres, c'est le Point origine (0,0) qui est choisi par défaut. Des méthodes permettent de changer les valeurs du Point, de le translater ou de fournir (toString()) une chaîne donnant ses caractéristiques.

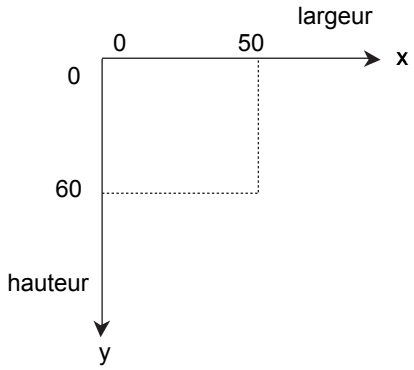


Figure 5.1 — Le système des coordonnées graphiques : l'axe des y est vers le bas.

La classe **Dimension** définit deux attributs entiers représentant les dimensions d'un objet en largeur et en hauteur. Les attributs **width** et **height** sont **public**. Il y a trois constructeurs : par défaut, les deux valeurs sont nulles ; un objet Dimension peut aussi être construit à partir d'un autre objet Dimension, ou à partir de deux entiers. Les principales méthodes sont :

public Dimension **getSize** () :fournit un objet Dimension contenant largeur et hauteur.

public void **setSize** (Dimension d) :modifie les attributs de l'objet à partir de d.

public void **setSize** (int width, int height) : modifie les attributs à partir des deux paramètres.

public String **toString** () :fournit une chaîne indiquant les valeurs des deux attributs.

La classe **Rectangle** définit un rectangle sur le plan caractérisé par le point p repérant son coin supérieur gauche (CSG), et par sa dimension d (largeur et hauteur). Rectangle a 4 attributs public : **x** et **y,** coordonnées du point, et **width** et **height,** valeurs de d.

public **Rectangle** (Point p, Dimension d)

public **Rectangle** (int x, int y, int width, int height)

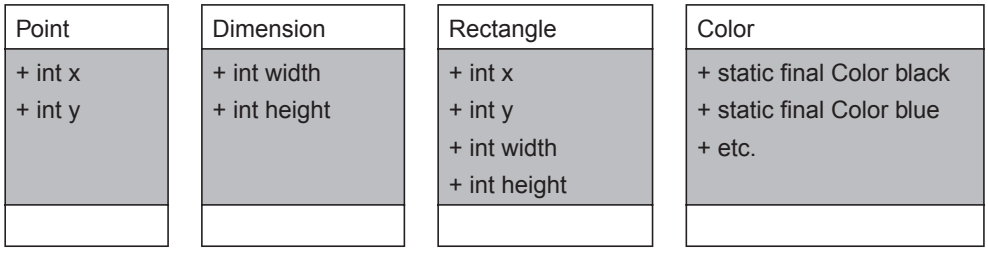


Figure 5.2 — Les attributs public des classes Point, Dimension, Rectangle et Color.

La classe **Color** permet d'accéder à des constantes `static` indiquant 13 couleurs prédéfinies :

```
Color.black   Color.blue   Color.cyan       Color.darkGray
Color.gray    Color.green  Color.lightGray  Color.magenta
Color.orange  Color.pink   Color.red        Color.white
Color.yellow
```

Une couleur est créée à partir de ses trois composantes (RVB : rouge, vert, bleu), chaque composante ayant une valeur sur 8 bits (de 0 à 255). On peut créer une nouvelle couleur en indiquant ses 3 composantes. Exemple : new Color (100, 100, 100) ; crée un objet référençant une nouvelle couleur.

La classe **Font** définit les caractéristiques d'une police de caractères. Une police de caractères (une fonte) est définie par le nom de la police, le style des caractères (normal, gras, italique) et la taille des caractères. PLAIN, BOLD et ITALIC sont des constantes static pour le style (qui peuvent s'ajouter : Font.BOLD + Font.ITALIC). Des accesseurs permettent d'obtenir ou de modifier chacune des caractéristiques de la fonte, ou de savoir si la fonte possède un style particulier. Exemple : Font f = new Font ("Courier", Font.BOLD, 12) ; crée un objet f de type Font, de police Courier, en gras et de taille 12.

La classe **FontMetrics** pour une fonte donnée (f par exemple) fournit des méthodes donnant la taille en pixels d'un caractère ou d'un message.

Exemple :

```
Font f = new Font ("Courier", Font.BOLD, 12);
FontMetrics fm = getFontMetrics (f);
int larg = fm.stringWidth (titre));  // largeur du titre de type String
```

La classe **Graphics** est une classe abstraite encapsulant (contenant) des informations permettant de dessiner des formes géométriques ou du texte pour un composant. Chaque composant a un contexte graphique. Les principales informations sont :

- le composant sur lequel il faut dessiner,
- le rectangle à redessiner au cas où le composant a été partiellement recouvert (par un autre composant) ;
- la couleur du dessin ;
- la fonte utilisée.

Un objet de la classe Graphics (un contexte graphique) contient toutes les informations pour écrire ou effectuer un dessin sur un composant. La classe Graphics définit de nombreuses méthodes pour tracer du texte ou des formes géométriques. Quelques méthodes sont données succinctement à titre indicatif parmi les plus utilisées :

- **drawImage** (Image, int, int, Color, ImageObserver) : trace une image.
- **drawLine** (int, int, int, int) : trace une ligne droite entre deux points.
- **drawRect** (int, int, int, int) : trace un rectangle.
- **drawOval** (int, int, int, int) : trace un ovale (ellipse) dans le rectangle défini.
- **drawArc** (int, int, int, int, int, int) : trace un arc (une partie d'ellipse).

- **drawString** (String, int, int) : trace une chaîne de caractères.
- **fillRec** (int, int, int, int) : remplit un rectangle.
- **getColor** () : fournit la couleur courante.
- **getFont** () : fournit la fonte courante.
- **setColor** (Color) : change la couleur courante.
- **setFont** (Font) : change la fonte courante.

Exemple d'utilisation (g est un objet de type Graphics)

```
g.setColor (Color.red);                    // couleur courante : rouge
g.drawString ("Bonjour", 5, 50);         // "Bonjour" en x = 5 et y = 0
g.drawRect (10, 10, 50, 50); // Rectangle de CSG (10,10); w = 50; h = 50
```

5.1.2 La classe Component (composant graphique)

Les composants graphiques (*classe Component* en anglais) de AWT constituent une hiérarchie de classes utilisant le principe de l'héritage. La classe Component de AWT dérive de la classe Object (voir page 105) et contient une bonne centaine de méthodes.

Ce chapitre présente le principe de l'interface graphique et quelques méthodes utilisées par la suite au fil des exemples. Pour le reste, il faut consulter la documentation sur la bibliothèque des classes.

La classe abstraite **Component** contient des méthodes permettant d'accéder ou de modifier les **caractéristiques communes à tous les composants** : dimension du composant, coordonnées du coin supérieur gauche du composant, couleur du texte ou du fond, type de curseur quand on passe sur le composant, police des caractères, est-il visible ou non ? La classe est **abstraite** : on ne peut pas créer directement d'objet Component ; seulement des objets dérivés (voir la classe abstraite Personne, page 96 et la figure 3.2, page 97).

Les composants sont disposés sur la fenêtre initiale. Certains composants peuvent recevoir d'autres composants : on les appelle des **conteneurs** (*container* en anglais). L'ensemble des composants d'une application constitue un arbre des composants, la racine de l'arbre étant la fenêtre initiale de l'application.

La disposition des composants dans la fenêtre peut être imposée par le programme qui indique, pour chaque composant, le coin supérieur gauche du composant (CSG), sa largeur et sa hauteur. Cependant, de nombreuses fenêtres sont modifiables en taille (par l'utilisateur) lors de l'exécution. Se pose alors le problème de la disposition des composants lorsque la fenêtre change de taille. Faut-il garder aux sous-composants la même taille et le même emplacement ? Dans ce cas, la modification de la taille de la fenêtre a peu d'intérêt.

En cas de réorganisation de la fenêtre, celle-ci peut être à la charge du programme qui tient compte de la taille réelle de la fenêtre, ou se faire suivant un plan type de réorganisation appelé "Gestionnaire de mise en page" (*LayoutManager* en anglais).

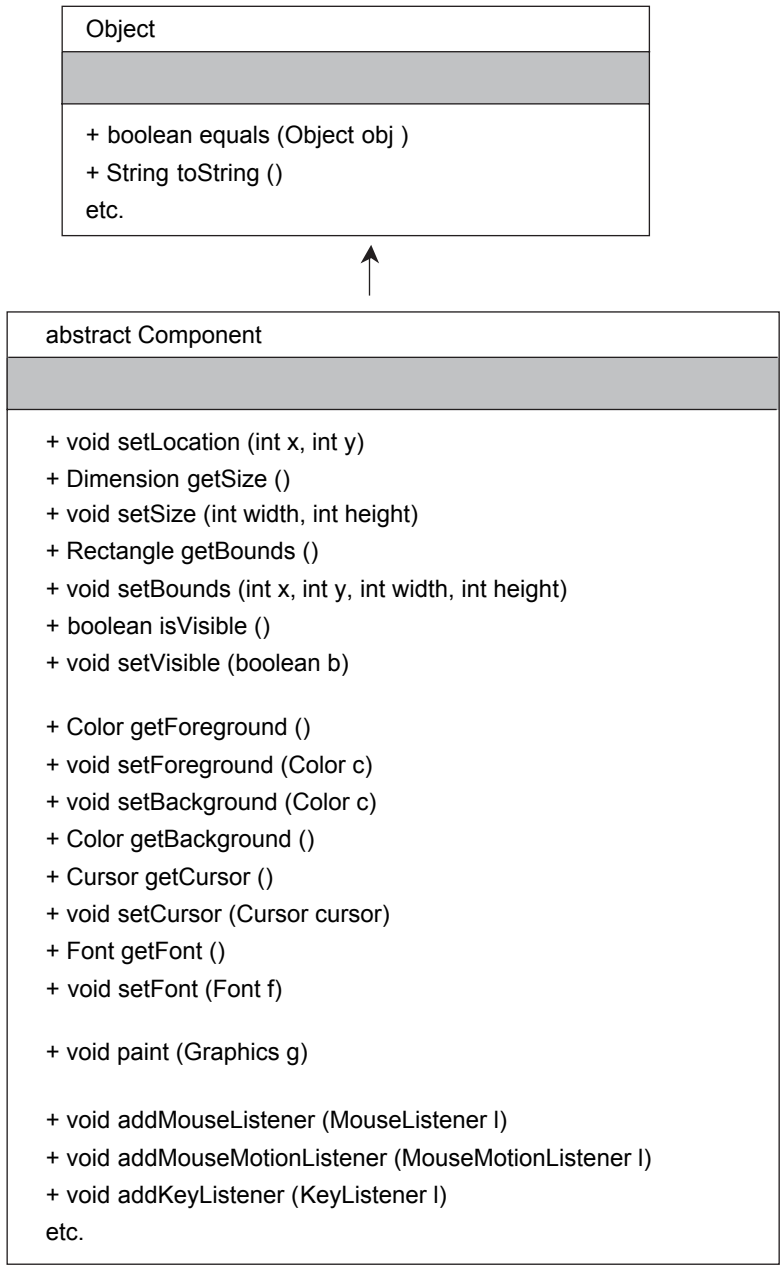


Figure 5.3 — La classe Component.
Il s'agit d'une classe abstraite qui hérite de Object et définit les caractéristiques communes aux composants graphiques.

Les principales méthodes de la classe Component sont décrites succinctement ci-dessous :

le positionnement du composant dans l'espace de son père :

- public Rectangle **getBounds**() : fournit un objet Rectangle indiquant la position du composant (CSG : coin supérieur gauche) par rapport au conteneur et ses dimensions (largeur et hauteur) lors de l'appel de cette méthode (voir Dimension, figure 5.2). Celles-ci ne sont pas figées, mais peuvent être modifiées, par l'utilisateur, par le programmeur, ou automatiquement lors de la mise en page.
- public Dimension **getSize** () : fournit la taille du composant (largeur et hauteur).
- public Point **getLocation** () : fournit le point CSG du composant.
- public void **setBounds** (int x, int y, int width, int height) : modifie le point CSG par rapport au conteneur et la taille du composant.
- public void **setSize** (int width, int height) : modifie la taille du composant.
- public void **setLocation** (int x, int y) : modifie le point CSG du composant.
- public void **setVisible** (boolean b) : le composant est visible si b est vrai, invisible sinon. Par défaut, les composants sont visibles, sauf les fenêtres qui sont a priori invisibles (Frame par exemple).
- les couleurs du texte et du fond, la police des caractères, le curseur
- public Color **getBackground** () : fournit la couleur du fond du composant. Si aucune couleur n'est définie pour le composant, c'est celle du conteneur qui est utilisée.
- public Color **getForeground** () : fournit la couleur du premier plan (texte ou dessin) du composant. Si aucune couleur n'est définie pour le composant, c'est celle du conteneur qui est utilisée.
- public void **setBackground** (Color c) : modifie la couleur du fond du composant.
- public void **setForeground** (Color c) : modifie la couleur du premier plan du composant.
- public Font **getFont** () : fournit la fonte courante (voir Font page 137) qui est par défaut celle du conteneur.
- void **setFont** (Font *f*) : la fonte f devient fonte courante.
- le dessin du composant
- public Graphics **getGraphics** () : fournit un objet de type Graphics pour ce composant.
- public void **paint** (Graphics *g*) : dessine le composant en utilisant le contexte graphique g.
- public void **update** (Graphics *g*) : redessine le fond et appelle la méthode paint (g).
- public void **repaint** () : appelle update (g), g étant le contexte graphique du composant.

le traitement des événements concernant ce composant

Des événements provoqués par la souris ou le clavier peuvent concerner un composant graphique et le faire réagir. Si on veut que le composant soit sensible à certaines catégories d'événements, il faut le demander explicitement en mettant un

objet à l'écoute de l'événement (un Listener) et en indiquant ce qui doit être fait lorsque l'événement survient.

- void **addMouseListener** (MouseListener *l*) : l'objet l indique ce que doit faire le composant lors d'un clic de souris, lorsque le curseur entre dans son espace (le rectangle qu'il occupe) ou en sort, etc. Il gère les événements de la souris.
- void **addMouseMotionListener** (MouseMotionListener *l*) : idem ci-dessus mais l'objet de type MouseMotionListener indique ce que doit faire le composant lors du mouvement de la souris (déplacement de la souris bouton appuyé ou non).
- void **addKeyListener** (KeyListener *l*) : idem mais pour les événements en provenance du clavier et concernant le composant déclenchant la méthode.

5.1.3 La hiérarchie des composants graphiques de base

La classe **Component** est dérivée en sous-classes répertoriant les spécificités de chaque composant. La hiérarchie des composants est donnée sur la figure 5.4. On peut facilement définir de nouveaux composants en dérivant cette classe Component ou une de ses sous-classes.

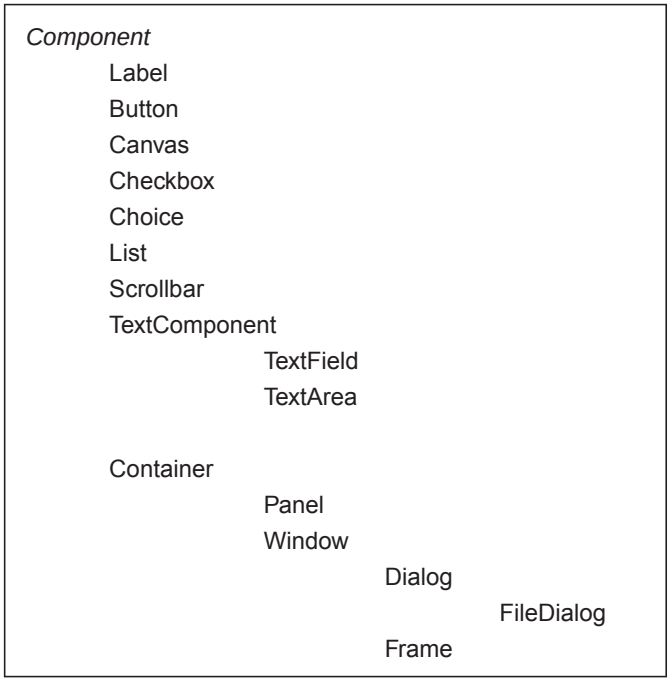


Figure 5.4 — La hiérarchie des composants (Component) avec la librairie AWT de Java.

La classe **Component** est une classe **abstraite** définissant les caractéristiques communes aux objets pouvant être représentés graphiquement à l'écran. Ces objets ont des propriétés communes (taille, emplacement sur l'écran, etc.), et des méthodes

communes (fournir la taille de l'objet, afficher ou masquer l'objet, invalider l'objet, gérer les événements concernant l'objet, etc.). Le principe de l'héritage (voir chapitre 3) convient parfaitement pour ces composants graphiques qui se spécialisent par héritage. La figure 5.5 donne des exemples de composants graphiques de AWT.

> ***Remarque*** : ces composants ne peuvent pas être affichés directement. Ils doivent être ajoutés (méthode add()) à un objet conteneur (une fenêtre par exemple) (voir 5.1.5, page 149).

La classe **Label** définit une ligne de texte que le programme peut changer en cours d'exécution. Pour l'utilisateur, ce Label est une chaîne constante qu'il ne peut pas modifier sur son écran. Ce composant dispose de toutes les méthodes de Component (héritage). On peut lui définir un emplacement, une taille, une couleur de fond ou de texte, une fonte particulière. A cela, s'ajoute quelques spécificités des Label concernant la justification (gauche, centré, droite) dans l'espace attribué.

Attributs et méthodes :

- Label.CENTER, Label.LEFT, Label.RIGHT : constantes static pour justifier le texte.
- **Label** (), **Label** (String eti), **Label** (String eti, int justification) : constructeurs de Label
- String **getText** () : fournit le texte du Label.
- void **setText** (String text) : modifie le texte du Label.

La classe **Button** définit un bouton affichant un texte et déclenchant une action lorsqu'il est activé à l'aide de la souris. Exemple : un bouton Quitter.

Méthodes :

- **Button** (String label) : constructeur d'un bouton ayant l'étiquette label.
- String **getLabel** () : fournit l'étiquette du bouton.
- void **setLabel** (String label) : modifie l'étiquette du bouton.
- void **addActionListener** (ActionListener l) : enregistre un objet de type ActionListener qui indique ce qu'il faut faire quand on appuie ou relâche le bouton. Cet enregistrement est retiré à l'aide de la méthode **removeActionListener**(). Le bouton devient alors inopérant.
- void **setActionCommand** (String command) : définit la chaîne de caractères qui est délivrée lorsque le bouton est activé.
- public String **getActionCommand** () : fournit la chaîne définie par setActionCommand() ou l'étiquette du bouton par défaut.

La classe **Canvas** définit une zone rectangulaire de dessin.

Méthodes :

Canvas () : constructeur de la zone de dessin.

void **paint** (Graphics *g*) : repeint le composant avec la couleur du fond.

Les applications doivent dériver la classe Canvas et redéfinir la méthode paint() en fonction de l'application.

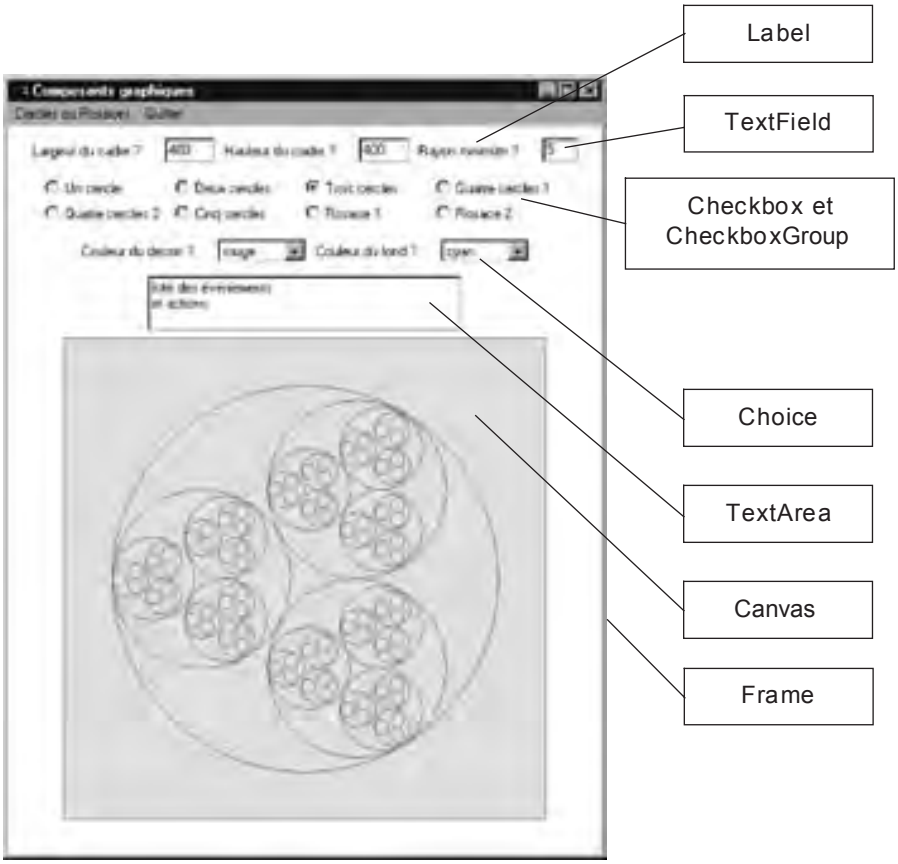


Figure 5.5 — Des exemples de composants graphiques de la librairie AWT.

La classe **Checkbox** définit une boîte à cocher. Celle-ci a une étiquette et un état booléen coché ou non.

Méthodes :

- public **Checkbox** (String label, boolean etat) : crée une boîte à cocher d'étiquette label avec l'état coché ou non suivant le booléen etat.
- public **Checkbox** (String label, boolean etat, **CheckboxGroup** group) : constructeur qui insère la boîte dans un groupe de boîtes.
- String **getLabel** () : fournit l'étiquette de la boîte à cocher.
- boolean **getState** () : fournit l'état de la boîte à cocher.
- **setLabel** (String label) : modifie l'étiquette.
- void **setState** (boolean etat) : modifie l'état.
- void **addItemListener** (ItemListener l) : l'objet l de type ItemListener indique ce qu'il faut faire si la boîte à cocher est sélectionnée. **removeItemListener** (ItemListener l) annule l'opération précédente ; l'écouteur l devient inopérant.

Plusieurs boîtes à cocher peuvent être groupées de façon qu'une seule boîte soit cochée à la fois. **CheckboxGroup**() crée un groupe de boîtes à cocher ; les boîtes de type **Checkbox** sont ajoutées au groupe à l'aide d'un constructeur ayant un paramètre supplémentaire mentionnant le groupe :

public Checkbox **getSelectedCheckbox** () : fournit la boîte active du groupe.
void **setSelectedCheckbox** (Checkbox b) : b devient la boîte active du groupe.

La classe **Choice** définit un menu contenant une liste d'options dont une seule peut être activée (choisie). On peut ajouter ou enlever par programme des éléments à cette liste en fin de liste ou à une position donnée de la liste. On peut connaître le nombre d'éléments dans la liste, le rang ou l'étiquette de l'élément sélectionné par l'utilisateur, etc. Voir figure 5.5.

Méthodes :

- **Choice** () : constructeur du menu de choix. La liste des choix est vide.
- void **add** (String item) : ajoute la chaîne item à la liste de choix.
- int **getItemCount** () : fournit le nombre d'éléments dans la liste de choix.
- int **getSelectedIndex** () : fournit l'index de l'élément sélectionné.
- String **getSelectedItem** () : fournit l'étiquette de l'option sélectionnée.
- void **select** (int pos) : l'item de numéro pos devient l'item courant sélectionné et affiché.
- void **addItemListener** (ItemListener l) : l'objet l de type ItemListener indique ce qu'il faut faire si une option de la boîte de choix est sélectionnée. **removeItemListener**() annule l'opération précédente.

La classe **List** définit une liste déroulante d'éléments. Le nombre d'éléments affichés peut être modifié. On peut également sélectionner un ou plusieurs éléments de la List. (Voir figure 7.11, page 302).

Méthodes :

- **List** () : constructeur d'une liste vide. Un seul élément sélectionnable.
- **List** (int n) : constructeur d'une liste vide. n éléments affichés à la fois. Un seul élément sélectionnable.
- **List** (int n, boolean multi) : constructeur d'une liste vide. n éléments affichés à la fois. Un seul élément sélectionnable si multi est faux, plusieurs si multi est vrai.
- void **add** (String s) : ajoute la chaîne s dans la liste.
- String **getSelectedItem** () : fournit l'étiquette de l'élément sélectionné.
- String[] **getSelectedItems** () : fournit un tableau des étiquettes des l'éléments sélectionnés en cas de sélection multiple.
- String **getSelectedIndex** () : fournit l'indice de l'élément sélectionné.
- String[] **getSelectedIndexes** () : fournit un tableau des indices des l'éléments sélectionnés en cas de sélection multiple.

La classe **Scrollbar** définit une barre de défilement vertical ou horizontal (voir figures 5.39 et 5.40, page 224). Cette barre représente les valeurs entières comprises entre les deux entiers minimum et maximum. Le curseur représente la valeur courante

de la barre. La valeur courante peut être modifiée (avec un incrément paramétrable) suite à des clics de la souris sur cette barre.

Méthodes :

- Scrollbar.HORIZONTAL, Scrollbar.VERTICAL : constantes static indiquant le type de la barre de défilement (horizontale ou verticale).
- **Scrollbar** (int orientation, int valeur, int visible, int minimum, int maximum) : constructeur d'une barre de défilement avec l'orientation donnée (verticale ou horizontale), de valeur courante `valeur` et représentant les valeurs entre `minimum` et `maximum`. visible indique la largeur du curseur du Scrollbar.
- int **getValue** () : fournit la valeur courante.
- void **setValue** (int valeur) : change la valeur courante.
- void **setBlockIncrement** (int *v*) : définit l'incrément de bloc quand on clique sur la barre à droite ou à gauche du curseur.
- void **addAdjustmentListener** (AdjustmentListener l) : enregistre un écouteur chargé de gérer les événements de la barre de défilement (clics de la souris).

La classe **TextComponent** est la super-classe des classes de texte éditable (TextArea et TextField). Elle regroupe les caractéristiques communes à ces deux classes.

Méthodes :

- String **getText** () : fournit le texte du composant.
- void **setText** (String t) : modifie le texte t du composant.
- String **getSelectedText** () : fournit le texte sélectionné (par l'utilisateur) dans le texte du composant. L'utilisateur a pu sélectionner avec la souris, une partie seulement du texte.
- void **setEditable** (boolean b) : le texte du composant est éditable ou non suivant la valeur du booléen.

La classe **TextField** est une zone de saisie d'une *seule ligne* de texte. Les caractères entrés peuvent être remplacés par un caractère choisi dit écho (pour les mots de passe par exemple).

Méthodes :

- **TextField** (String t, int n) : le composant contient la chaîne de caractères t affichable sur *une ligne* de n colonnes.
- void **setEchoChar** (char *c*) : c est le caractère écrit en écho.

La classe **TextArea** est une zone de saisie ou d'affichage de texte sur *plusieurs lignes*. Cette zone peut avoir ou non des barres de défilement lorsque la fenêtre est trop petite pour afficher tout le texte horizontalement ou verticalement.

- **TextArea** (String t, int nl, int nc, int scrollbars) : zone de texte de nl lignes et nc colonnes contenant initialement la chaîne t. Le paramètre scrollbars correspond à une des quatre constantes static indiquant 0, 1 barre verticale, 1 barre horizontale ou 2 barres de défilement.

- void **append** (String s) : ajoute s en fin du texte de la zone de texte.
- void **insert** (String s, int pos) : insère s à la position pos dans le texte de la zone de texte.
- void **replaceRange** (String s, int debut, int fin) : remplace le texte entre debut et fin par la chaîne s.

5.1.4 Les gestionnaires de répartition des composants (gestionnaires de mise en page)

Le point repérant le coin supérieur gauche (CSG) et la dimension du rectangle occupé peuvent être indiqués ou calculés par le programme pour chacun des composants. Les composants peuvent aussi être **disposés automatiquement** en suivant les caractéristiques définies dans un objet chargé de la mise en page (*Layout* en anglais). Ceci est particulièrement utile lorsque la fenêtre initiale change de taille. Plusieurs types de mises en page (Layout) sont prédéfinis :

- **absence de Layout** : les composants sont rangés en fonction de leur position et dimensions définies explicitement dans le programme.
- **BorderLayout** : l'espace du conteneur est divisé en cinq zones repérées par : le centre, le nord, le sud, l'est et l'ouest. La taille de ces différentes zones est ajustée en fonction des composants (de type Component) qui y sont insérés.
- **FlowLayout** : les composants sont rangés les uns à la suite des autres sur la même ligne en fonction de leurs dimensions. Si l'élément ne peut pas tenir sur la fin de la ligne, il est mis sur la ligne suivante. Si le conteneur (Container) change de dimensions, les composants peuvent être amenés à changer de place (changement de taille d'une fenêtre).
- **GridLayout** : les composants sont rangés dans des cases de même taille disposées en lignes et en colonnes.
- **CardLayout** : les composants sont présentés un par un comme un paquet de cartes vu de dessus. On ne voit que le composant du dessus. On peut visualiser un des composants en fonction de son numéro ou de son nom. On peut aussi visualiser le premier, le dernier ou le suivant.
- **GridBagLayout** : les composants sont rangés dans des cases de même taille disposées en lignes et en colonnes mais en tenant compte de contraintes : tel composant doit être le dernier de sa ligne par exemple ; tel autre composant occupe deux cases, etc.

Les différents types de Layout sont illustrés sur les exemples suivants.

absence de Layout : l'emplacement et la taille des composants sont **imposés par le programme** à l'aide de setLocation(), setSize() ou setBounds() (voir figure 5.3, page 139). En cas de réduction de la fenêtre, les composants ne sont pas déplacés et peuvent être en partie ou totalement masqués sauf bien sûr si le programme tient compte des dimensions de la fenêtre pour placer les composants (voir figure 5.6). L'absence de Layout est obtenue par : setLayout (null). Les six composants sont deux Motif "champignon", un Label, un TextArea "Exemple ...", un Motif "Clé de sol" et un Motif "Palette". La classe Motif est décrite à la page 153.

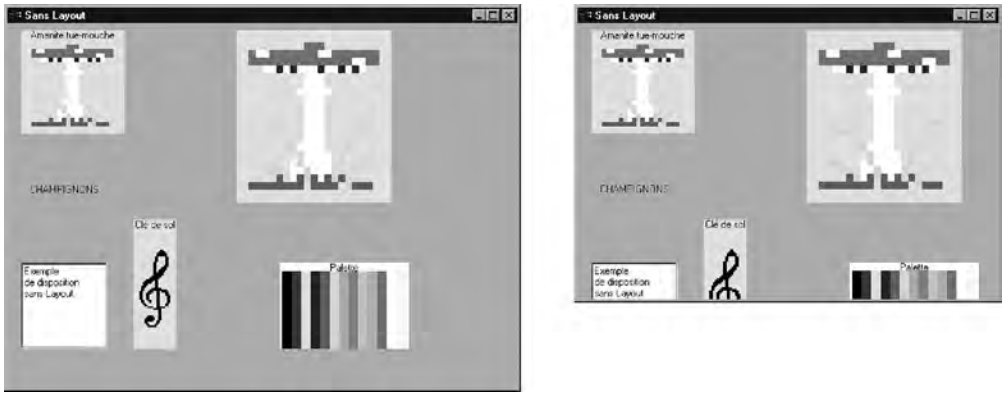


Figure 5.6 — Dessin de six Component sans Layout.
L'emplacement est imposé par le programme. Sur le dessin de droite, la fenêtre a été réduite ; les composants n'ont changé ni de taille, ni de place.

BorderLayout : les composants sont rangés suivant les directions nord, sud, ouest, est et centre. Le conteneur peut avoir au plus cinq composants. Sur l'exemple de la figure 5.7, le Label CHAMPIGNONS est centré au nord, le Motif palette est à l'ouest, le Motif champignon est au centre, le Motif "Clé de sol" est à l'est, et le Label "Layout de type BorderLayout" est au sud, cadré à gauche. Les composants sont automatiquement répartis dans les 5 zones en fonction de leur taille et de la taille de la fenêtre.

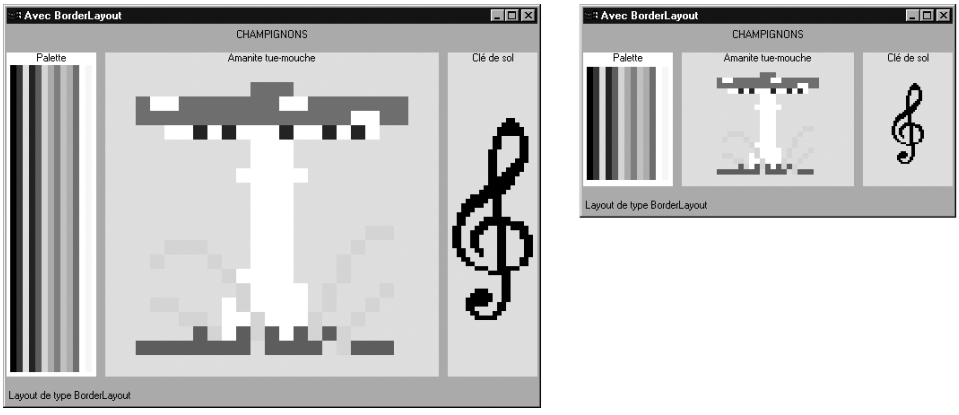


Figure 5.7 — Dessin de cinq Component avec un BorderLayout sur deux fenêtres de taille différente.
Le gestionnaire de mise en page recalcule automatiquement la position et la taille des composants.

FlowLayout : les composants sont disposés les uns à la suite des autres suivant les dimensions de la fenêtre. Si les dimensions changent (la fenêtre est agrandie ou réduite), la disposition peut être modifiée et des composants déplacés d'une ligne à l'autre. Sur la figure 5.8, la réduction de la taille de la fenêtre entraîne un déplacement du composant de type TextArea contenant le message "Exemple de ...".

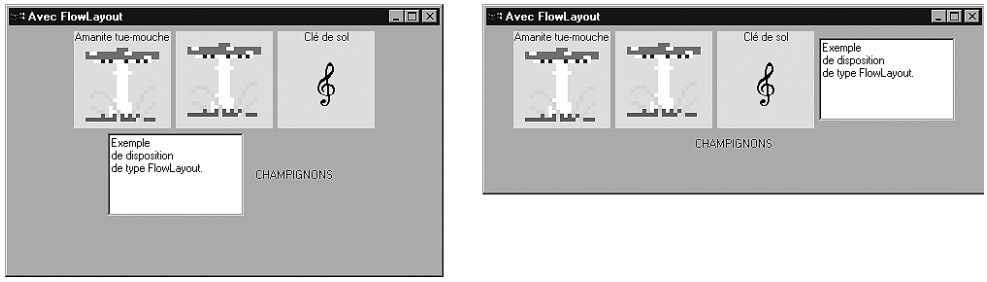


Figure 5.8 — Dessin de cinq Component avec un FlowLayout dans deux fenêtres de taille différente.

GridLayout : la disposition se fait suivant un tableau divisant l'espace en lignes et en colonnes. On peut indiquer le nombre maximum de lignes et/ou colonnes. Sur la figure 5.9, la mise en page est créée comme suit : new GridLayout (0, 3) ; 0 indique un nombre indéterminé de lignes ; 3 indique trois composants par ligne. Les composants sont redessinés dans l'espace qui leur est attribué et qui change si la taille de la fenêtre change.

Le dessin peut se faire en gardant les dimensions initiales de l'image, en gardant les proportions de l'image (modifier hauteur et largeur dans le même rapport) ou en occupant tout l'espace attribué ce qui modifie l'aspect de l'image. Ceci est pris en compte par la méthode de dessin du composant et non par le gestionnaire de mise en page.

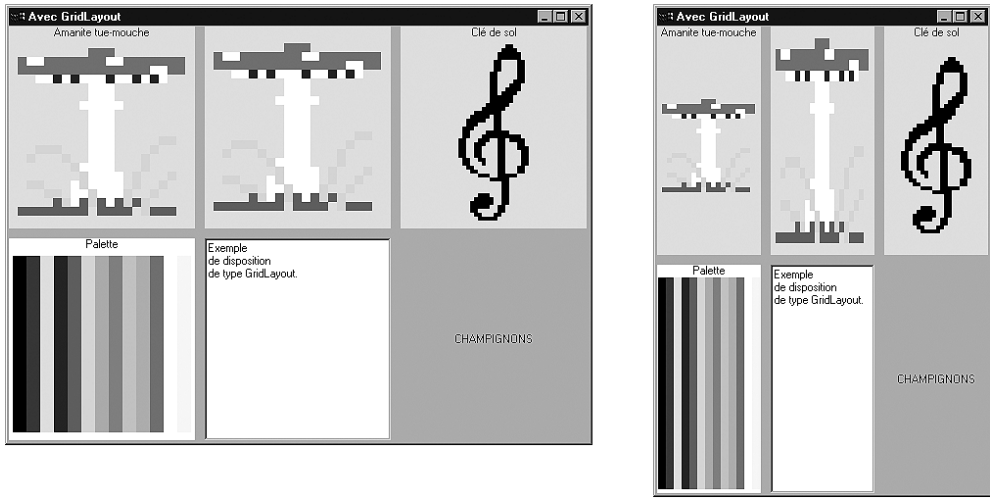


Figure 5.9 — Dessin de six Component avec un GridLayout (trois composants par ligne) dans deux fenêtres de taille différente.

CardLayout : les composants sont représentés comme un paquet de cartes vu de dessus (voir figure 5.10). On ne voit qu'une seule carte mais des méthodes permettent de faire défiler les cartes, ou même d'amener une carte au-dessus du paquet en donnant son numéro ou son nom.

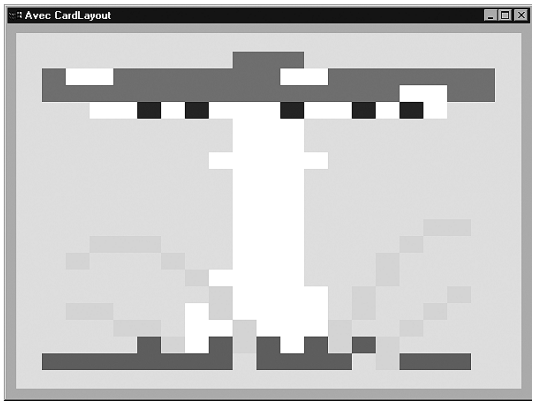


Figure 5.10 — Un seul Component visible avec un CardLayout ; les autres sont en dessous.

GridBagLayout : les composants sont disposés en lignes et en colonnes en suivant des contraintes de disposition.

Remarque : un composant peut redéfinir la méthode **getPreferredSize**() de Component pour indiquer sa taille préférée en l'absence d'autre indication. Les gestionnaires de mise en pages en tiennent plus ou moins compte suivant leur type pour décider de la largeur et de la hauteur du composant en l'absence d'une autre indication :

- absence de Layout ou FlowLayout : largeur et hauteur préférées utilisées.
- GridLayout : largeur et hauteur préférées non utilisées.
- BorderLayout : largeur et hauteur préférées non utilisées pour le centre. La largeur préférée est utilisée à l'ouest et à l'est (la hauteur n'est pas utilisée ; c'est celle attribuée au composant). La hauteur préférée est utilisée au nord et au sud (la largeur n'est pas utilisée).

Exemple de redéfinition dans un composant (voir Motif en 5.2.2) :

```
public Dimension getPreferredSize() {
  return new Dimension (100, 100);      // largeur et hauteur préférées
}
```

5.1.5 Les composants (Component) de type conteneur (Container)

5.1.5.1 Définition et exemple de conteneur

Un objet de type **Container** (conteneur en français) est un composant graphique qui peut contenir d'autres composants graphiques tels que ceux définis précédemment : Label, Button, Checkbox, Choice, Scrollbar, etc. Les composants ajoutés à un conteneur sont enregistrés dans une liste. On peut retirer un composant de la liste. À un conteneur, on peut ajouter un ou plusieurs autres conteneurs. On a donc en fait un arbre des composants.

La classe Container est une classe **abstraite** (voir page 96) définissant les méthodes communes aux conteneurs. Sous AWT, les classes dérivées de Container sont : Panel et Window, Window se dérivant à nouveau en Frame et Dialog (voir figure 5.4, page 141).

La figure 5.11 montre l'arbre des composants correspondant aux composants de la figure 5.5, page 143. La racine est une fenêtre de type Frame qui contient un conteneur Panel. Celui-ci contient 3 conteneurs de type Panel (non mis en évidence sur la figure car ils ont la couleur de fond du père), un composant de type TextArea et un autre de type Canvas. Le Panel 1 contient 3 composants Label et 3 composants TextField ; le Panel 2 contient les 8 composants de type boîtes à cocher ; et le Panel 3, les 2 composants de type Choice. La figure 5.22, page 172 met en évidence les différents panels.

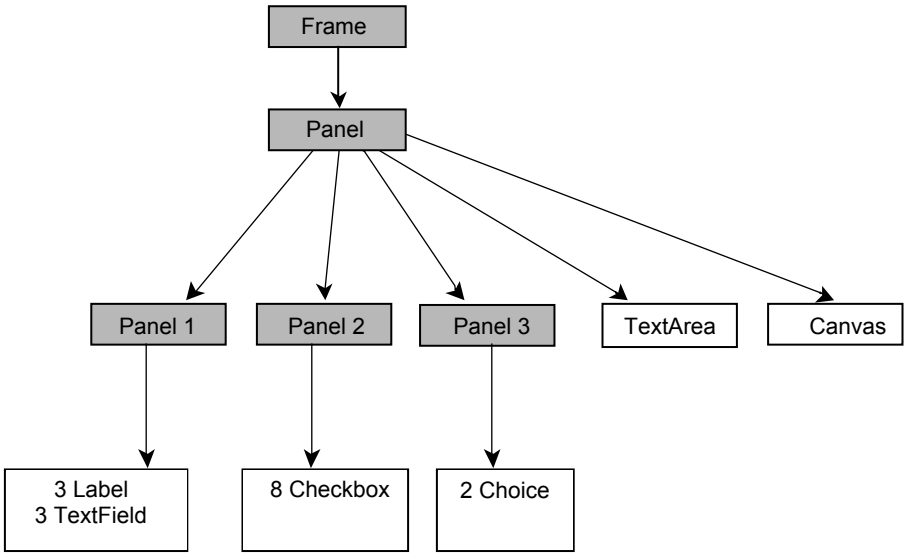


Figure 5.11 — L'arbre des composants de la figure 5.5.
En grisé, les conteneurs (Container).

5.1.5.2 Les méthodes de la classe Container de AWT

La classe abstraite Container (voir la classe abstraite Personne en 3.2.1 et la figure 3.2, page 97) permet de regrouper dans un composant dit conteneur les composants à répartir dans l'espace du conteneur soit par programme, soit en utilisant un gestionnaire de mise en page. Les principales méthodes de la classe Container sont indiquées ci-dessous ; elles définissent les caractéristiques communes à tous les conteneurs.

- Component **add** (Component *comp*) : ajoute le composant comp au conteneur.
- void **add** (Component *comp*, Object *constraints*) : ajout de comp avec des contraintes.
- void **remove** (Component *comp*) : enlève le composant comp du conteneur.
- void **remove** (int n) : enlève le nième composant du conteneur.

- Component **getComponent** (int n) : fournit le nième composant du conteneur.
- int **getComponentCount** () : fournit le nombre de composants dans le conteneur.
- Component **getComponentAt** (Point *p*) : fournit le composant se trouvant au point p.

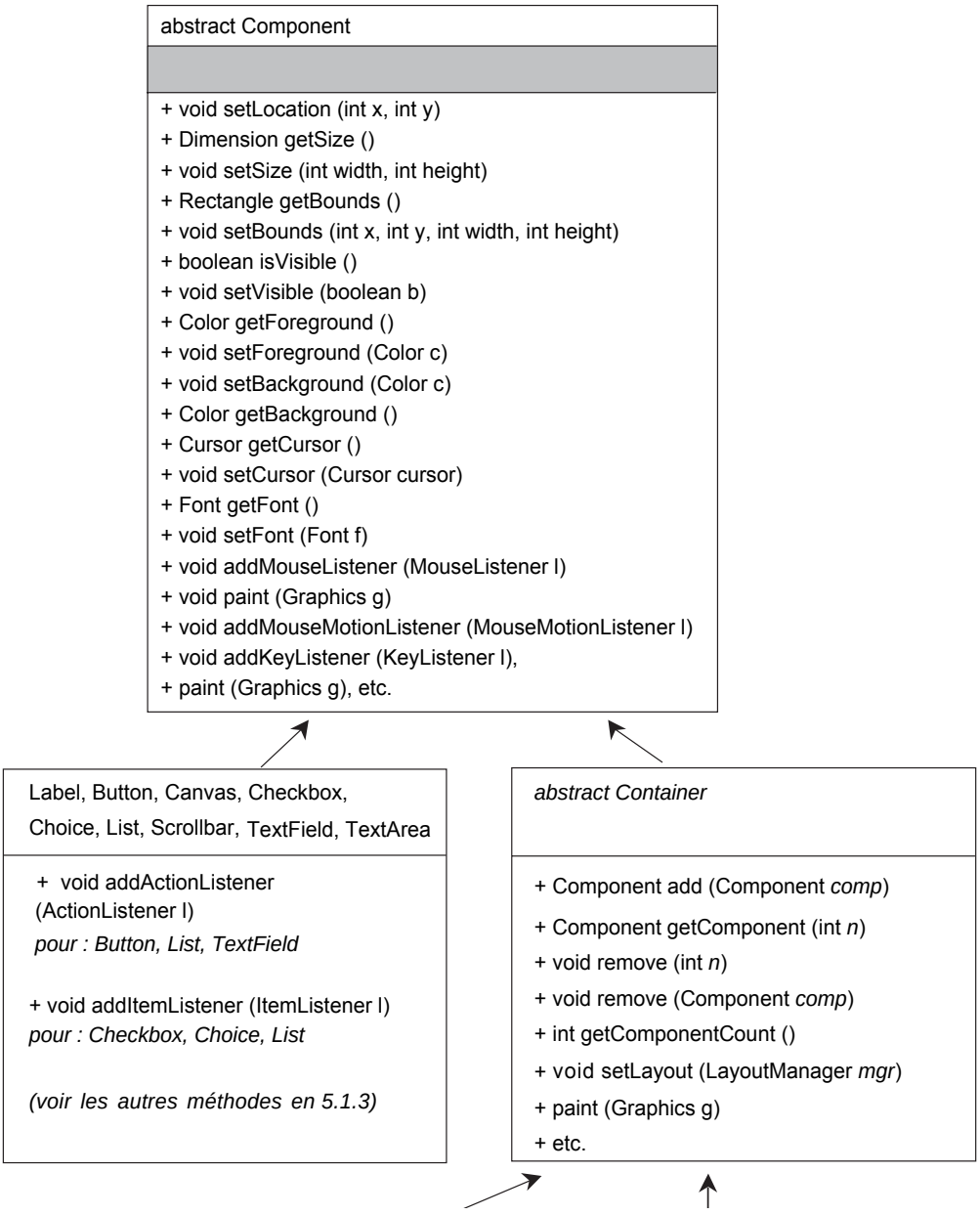


Figure 5.12 — L'arbre hiérarchique AWT des composants (Component) et des conteneurs (Container). (Suite de la figure au verso.)

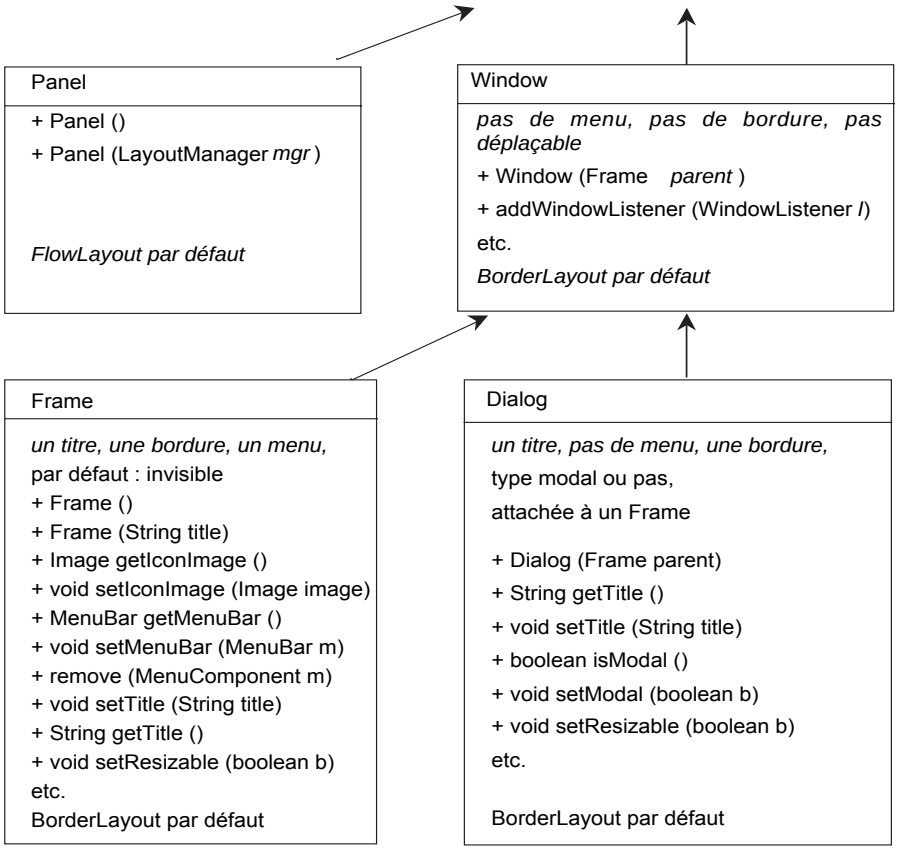


Figure 5.12 *(suite)* — L'arbre hiérarchique AWT des composants (Component) et des conteneurs (Container).

- void **paint** (Graphics *g*) : redessine le composant et ses sous-composants.
- void **setLayout** (LayoutManager *mgr*) : enregistre mgr comme objet indiquant la mise en page pour ce conteneur.
- void **validate** () : valide ce conteneur et ses composants en calculant l'emplacement et la taille de chacun des composants.

> ***Remarque*** : sur la figure 5.12, la classe Container dérive de Component. Un objet de la classe Container est un composant qui occupe une zone de l'espace de son père. Sur la figure 5.11, Panel 1, Panel 2 et Panel 3 sont des conteneurs occupant une partie de l'espace du père (de type Panel). La plupart des caractéristiques et des méthodes de Component restent valables pour un Container. Il a une taille, un emplacement défini par son coin supérieur gauche (CSG), une couleur de premier et d'arrière-plan, etc. La classe est abstraite et ne peut être instanciée ; elle doit donc être dérivée. Quand on ajoute un composant, sa couleur de premier plan et d'arrière-plan et la fonte utilisées sont héritées par défaut du conteneur.

La classe **Panel** définit un conteneur fournissant de l'espace dans lequel on peut ajouter d'autres composants y compris d'autres Panel. La mise en page est par défaut de type FlowLayout**.** La classe définit deux constructeurs, le deuxième précisant le type de gestionnaire de mise en page choisi. Les autres caractéristiques sont définies à l'aide des méthodes de la super-classe Component (espace occupé, couleur de fond, etc.). Voir figure 5.12.

La classe **Window** définit une fenêtre sans bordure et sans barre de menu. Par défaut, la mise en page est de type BorderLayout et la fenêtre est invisible. La fenêtre est sensible aux clics de la souris qui la font passer par défaut au premier plan. Un objet Window a obligatoirement pour père un objet de type Frame, Window ou Dialog qui doit être fourni au constructeur. On peut ajouter ou enlever des composants à la fenêtre puisque Window hérite de Container. On peut définir la taille et la couleur du fond puisque Window hérite de Component.

Méthodes :

- public **Window** (Frame *Window*) ; fenêtre Window attachée à une autre fenêtre de type Frame par exemple.
- void **dispose** () : détruit la fenêtre.
- public void **pack** () : dessine les composants de la fenêtre.
- public void **toFront** () : met la fenêtre au premier plan.
- public void **toBack** () : met la fenêtre à l'arrière plan.
- void **addWindowListener** (WindowListener *l*) : ajoute un objet l de type WindowListener qui indique ce qu'il faut faire lorsque la fenêtre est activée. **removeWindowListener**() supprime cette écoute.

La classe **Frame** définit une fenêtre principale d'application ayant un titre et une bordure. Par défaut, la mise en page est de type BorderLayout et la fenêtre est invisible. On peut ajouter un menu à une fenêtre de type Frame. On peut modifier sa taille, la mettre en icône, etc. Une même application peut créer plusieurs fenêtres de type Frame. La figure 5.5 définit une fenêtre de type Frame, de titre "Composants graphiques" ayant une barre de menu et des composants répartis dans différents conteneurs (voir figure 5.11).

La classe **Dialog** définit une fenêtre de dialogue (saisie ou affichage de messages). La fenêtre peut être de type modale ; on ne peut rien faire d'autre tant que la fenêtre n'est pas fermée. Voir figures 5.38, page 219.

> ***Remarque*** : un composant de type Window (et donc Frame et Dialog) ne peut être ajouté à un conteneur. C'est un composant de niveau 1 (top-level window).

5.2 UN NOUVEAU COMPOSANT : LA CLASSE MOTIF

5.2.1 La classe Motif

Pour bien comprendre le fonctionnement d'un composant, le mieux est d'en créer un nouveau. On veut créer un composant défini par la classe **Motif**, dérivant de Component. Un Motif est un dessin créé à partir d'une description donnée sous forme de texte

précisant les couleurs pour chaque pixel. Un champignon peut être schématisé comme l'indique la figure 5.13. Chaque lettre correspond à une couleur représentant un carré ayant cette couleur sur l'écran. Le Motif peut être de taille variable et doit pouvoir se dessiner dans le rectangle qui lui est attribué (soit par le programmeur, soit par le gestionnaire de mise en page du conteneur).

```
"                        "
"          KKK           "
" KLLKKKKKKKLLKKKKKKK    "
" KKKKKKKKKKKKKKKLLKK    "
"   LLDLDLLLDLLDLDL      "
"          LLL           "
"          LLL           "
"         LLLLL          "
"          LLL           "
"          LLL           "
"          LLL           "
"          LLL      FF   "
"   FFF    LLL     F     "
"  F   F   LLL    F      "
"        FLLLL    F      "
"         FLLLL F    F   "
"  FF    LFLLLL F   F    "
"    FF LLFLLLF   F      "
"     EFLEFELEFEF        "
" EEEEEEEE EEEE FEEE     "
"                        "
```

Figure 5.13 — Un composant Motif : chaque lettre représente une couleur.

Un objet de la classe **Motif** est caractérisé par les **attributs** suivants :

- le tableau de chaînes de caractères décrivant le motif (voir figure 5.13).
- le type du motif : taille fixe (l'image ne change pas de taille), proportionnelle (la proportion entre largeur et hauteur est conservée) ou variable (l'image occupe tout l'espace attribué cependant l'image peut être déformée).
- un titre (éventuellement null) ajouté en haut du Motif et centré dans l'espace attribué ;
- la palette des couleurs décrivant les couleurs attribuées à chacune des lettres représentant une couleur (Exemple : 'A' : rouge, 'B' : vert, etc.). Une palette par défaut est définie.
- sa largeur et sa hauteur en pixels si l'image est de taille fixe ;
- un booléen indiquant si on doit inverser l'image lors du dessin : les caractères à droite sont mis à gauche, et vice versa (inversion par rapport à un axe vertical en milieu d'image).
- un booléen indiquant si on doit dessiner le fond de l'image, ou si il doit rester transparent, ce qui permet de superposer des images.

Méthodes :

- **Motif** (String[] figure, int type) : constructeur (tableau décrivant la figure et le type du Motif ; on utilise la palette par défaut).
- **Motif** (String[] figure, int type, String titre) : constructeur avec un titre pour le Motif.
- **Motif** (String[] figure, int type, Color[] palette) : on définit une palette de couleurs.
- **Motif** (String[] figure, int type, String titre, Color[] palette) : titre et palette.
- void **setTaille** (int largeur, int hauteur) : définit la taille de l'image (si l'image est de taille fixe).

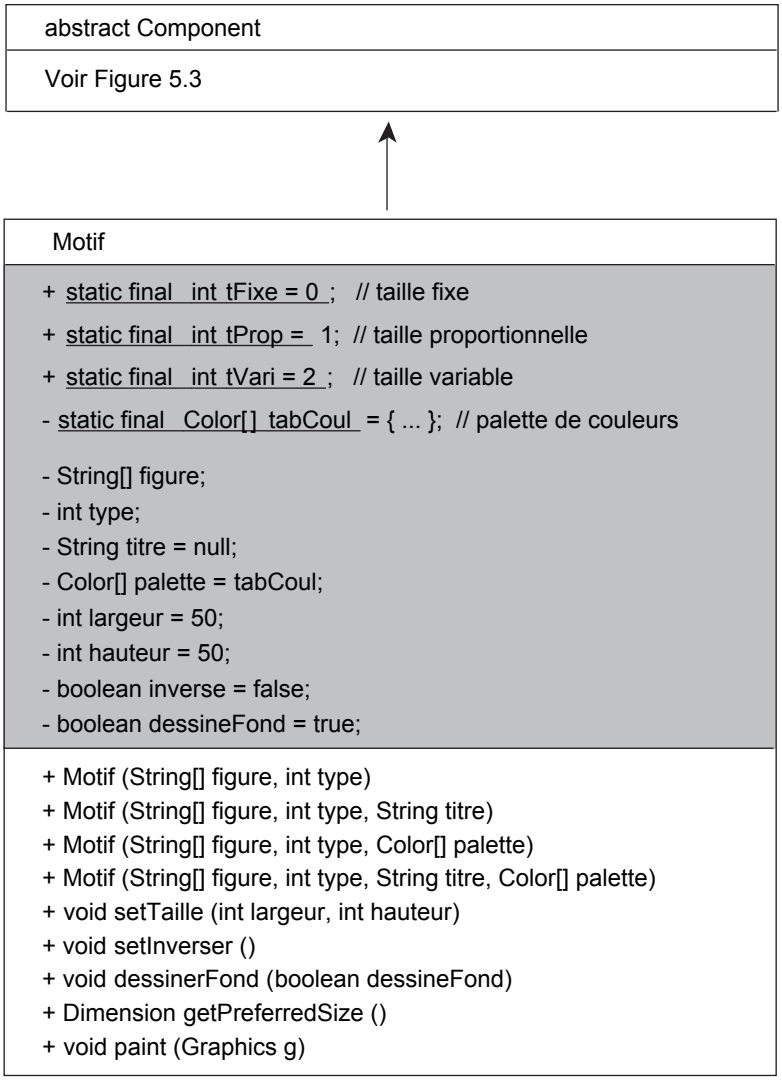


Figure 5.14 — Le composant Motif : ses attributs et ses méthodes.

- void **setInverser** () : dessiner ou non l'image inversée suivant un axe vertical en son milieu.
- void **dessinerFond** (boolean dessineFond) : dessiner ou non le fond de l'image suivant la valeur du booléen.
- Dimension **getPreferredSize** () : utilisée par certains gestionnaires de mise en page (voir page 149).
- void **paint** (Graphics g) : dessine le Motif dans son espace.

Comme **Motif hérite de Component** (ou de Canvas), il dispose de toutes les méthodes de Component, et en particulier les méthodes déterminant la taille ou la couleur du composant. Voir page 151.

5.2.2 Le programme Java de la classe Motif

La méthode getSize() appelée dans paint() fournit la taille du rectangle attribué au composant au moment de le redessiner. Cette taille peut changer d'un appel à l'autre notamment si la taille de la fenêtre varie. La classe Motif est rangée dans le paquetage mdawt de façon à pouvoir être utilisée de n'importe quelle classe Java.

```
// Motif.java  composant d'affichage d'un motif
//             dessiner en mode graphique à partir d'une
//             description en mode texte

package mdpaquetage.mdawt;  // paquetage mdawt

import java.awt.*;          // Component

public class Motif extends Canvas {             // ou extends Component
  public static final int tFixe = 0;  // taille fixe
  public static final int tProp = 1;  // taille proportionnelle
  public static final int tVari = 2;  // taille variable

  // tableau des 13 couleurs définies dans la classe Color
  // et codées de A à M pour décrire un Motif
  private static final Color[] tabCoul = {     // classe Color page 137
    // 13 couleurs définies comme des constantes static (classe Color)
    Color.black,  Color.blue,   Color.cyan,      Color.darkGray,
    Color.gray,   Color.green,  Color.lightGray, Color.magenta,
    Color.orange, Color.pink,   Color.red,       Color.white,
    Color.yellow
  };

  private String[] figure;        // les caractères décrivant la figure
  private int      type;                            // le type du dessin
  private String   titre   = null;                          // le titre
  private Color[]  palette = tabCoul;        // la palette des couleurs
  private int      largeur = 50;   // largeur par défaut si taille fixe
  private int      hauteur = 50;   // hauteur par défaut si taille fixe
  private boolean  inverse = false;               // inverser le Motif
  private boolean  dessineFond = true;             // dessiner un fond
```

Plusieurs constructeurs existent (voir surcharge des constructeurs page 47) permettant de construire un Motif à partir de paramètres différents. Tous les constructeurs ont un paramètre de type tableau de String représentant le Motif à dessiner (voir figure 5.13). Le paramètre *type* indiquant le type du dessin du Motif (fixe, proportionnel ou variable) lorsque l'espace alloué au Motif change. On peut aussi définir un titre et/ou une palette de couleurs indiquant la couleur pour chacune des lettres codant l'image.

```
// la figure, le type, le titre et la palette des couleurs
 public Motif (String[] figure, int type, String titre,
                                                    Color[] palette) {
   this.figure  = figure;
   this.type    = type;
   this.titre   = titre;
   this.palette = (palette == null) ? tabCoul : palette;
 }

  // la figure (de taille proportionelle)
  public Motif (String[] figure) {
    this (figure, tProp, null, null);
  }

 // la figure et le type
  public Motif (String[] figure, int type) {
    this (figure, type, null, null);            // voir this() page 50
  }

// la figure (de taille proportionnelle) et le titre
  public Motif (String[] figure, String titre) {
    this (figure, tProp, titre, null);
  }

  // la figure, le type et le titre
  public Motif (String[] figure, int type, String titre) {
    this (figure, type, titre, null);
  }

  // la figure, le type et la palette des couleurs
  public Motif (String[] figure, int type, Color[] palette) {
    this (figure, type, null, palette);
  }

  // constructeur de copie d'un Motif
  public Motif (Motif motif) {
    this (motif.figure, motif.type, motif.titre, motif.palette);
    setTaille (motif.largeur, motif.hauteur);
    this.inverse     = motif.inverse;
    this.dessineFond = motif.dessineFond;
    this.setBackground (motif.getBackground());
    this.setForeground (motif.getForeground());
  }
```

Si l'image est de taille fixe, ses dimensions par défaut sont de 50 pixels de large sur 50 de haut. La méthode setTaille() modifie les dimensions du Motif. La taille minimum du Motif correspond à un pixel par caractère du tableau figure. A chaque appel, la méthode setInverser() indique qu'il faut inverser l'image. Deux appels de setInverser() conduisent à l'image initiale.

```
// si l'image est de taille fixe,
// elle ne dépend pas de l'espace attribué.
public void setTaille (int largeur, int hauteur) {
  this.largeur = largeur;
  this.hauteur = hauteur;
  repaint();
}

// inverse l'image : la droite à gauche; la gauche à droite
public void setInverser () {
  this.inverse = !inverse;                    // change à chaque appel
  repaint();
}
```

Si on dessine le fond, le rectangle qu'occupe le motif est colorié avec la couleur du fond. Deux motifs ne peuvent pas se chevaucher, ce qui est possible si le fond est considéré comme transparent.

```
// dessine le fond du dessin ou pas (transparence)
public void dessinerFond (boolean dessineFond) {
  this.dessineFond = dessineFond;
  repaint();
}

// utilisée par FlowLayout, BorderLayout
public Dimension getPreferredSize() {                // voir page 149
  return new Dimension (100, 100);
}
```

La méthode paint() est chargée de redessiner le Motif à chaque fois qu'on le lui demande, en tenant compte des attributs de l'objet Motif et de l'espace attribué au composant par le programme ou par le gestionnaire de mise en page. Cet espace peut varier en cours d'exécution si on modifie la taille de la fenêtre principale.

Si le type du dessin est tFixe, on garde les valeurs de largeur et hauteur définies par défaut ou par setTaille(). Si le dessin est proportionnel ou variable, il doit s'adapter à l'espace attribué. On modifie largeur et hauteur du Motif en conséquence. S'il faut dessiner le fond du dessin, on remplit tout l'espace attribué par la couleur du fond du dessin. S'il y a un titre, il faut le centrer en haut de l'image en tenant compte de la fonte utilisée (voir page 137). Si la largeur est insuffisante, le titre n'est pas écrit. Il reste alors à calculer la largeur et la hauteur représentées par une lettre de la description du Motif. Pour chaque ligne et pour chaque colonne du Motif, on dessine un rectangle de la couleur indiquée par la lettre du Motif.

```
public void paint (Graphics g) {
  Dimension Di = getSize();        // dimension actuelle du composant
  if (dessineFond) {
    g.setColor (getBackground());
    g.fillRect (0, 0, largeur, hauteur);
    g.setColor (getForeground());
  }

  if (type == Motif.tProp || type == Motif.tVari) {
    hauteur = Di.height;
    largeur = Di.width;
  }  // sinon, on ne modifie pas largeur et hauteur

  // titre (le centrer, réserver de l'espace en haut)
  int margeT = 0;                                    // marge du titre
  if (titre != null) {
    FontMetrics fm = getFontMetrics (g.getFont());        // page 137
    int x = (largeur-fm.stringWidth(titre))/2; // x de début du titre
    int y = 10;
    if (x > 0) {                       // si assez large pour le titre
      g.drawString (titre, x, y);
      margeT = 10;
    }
  }

  int hauteurR = hauteur;                           // hauteur Restante
  if (hauteurR > margeT) hauteurR -= margeT;
  int nbLigne = figure.length;       // nombre de lignes de la figure
  if (nbLigne == 0) nbLigne = 1;
  int nbCar = figure[0].length();  // nombre de colonnes de la figure
  if (nbCar == 0) nbCar = 1;
  int dh = (int) (largeur / nbCar);                // delta horizontal
  int dv = (int) (hauteurR / nbLigne);               // delta vertical
  if (dh <= 0) dh = 1;
  if (dv <= 0) dv = 1;                              // au moins 1 pixel
  if (type == Motif.tFixe || type == Motif.tProp) {
    dh = Math.min (dh, dv);
    dv = dh;
  }
  int margeH = margeT + (hauteurR - nbLigne*dv) / 2;    // marge haut
  int margeG = (largeur - nbCar*dh) / 2;              // marge gauche

  for (int i=0; i < figure.length; i++) {
    String ligne;
    if (inverse) {                        // voir StringBuffer page 86
      StringBuffer sb = new StringBuffer (figure[i]).reverse();
      ligne = new String (sb);
    } else {
      ligne = figure[i];
    }
```

```
        for (int j=0; j < ligne.length(); j++) {
          char car = ligne.charAt (j);
          if (car != ' ') {
            int couleur = 0;
            if ((car >= 'A') && (car <= 'Z')) {    // au plus 26 couleurs
              couleur = (int) (car - 'A');          // rang de la couleur
            }
            if (couleur >= palette.length) couleur = 0;
            g.setColor (palette[couleur]);
            g.fillRect (margeG + j*dh, margeH + i*dv, dh, dv);
          }
        }
      }
    }  // paint

}  // class Motif
```

5.2.3 La mise en œuvre du composant Motif

La classe **PPMotif** hérite de **Frame** et dispose des méthodes de Frame, Window, Container et Component (voir figure 5.12, page 151). Pour un objet de type Frame, on peut définir un titre, une taille, une couleur de fond et une mise en page : celle choisie ci-dessous est de type GridLayout (voir 5.1.4) avec 3 colonnes. Les composants sont espacés de 10 pixels dans les 2 sens. La fenêtre se referme automatiquement quand on clique sur l'icône de fermeture.

La description caractère par caractère des motifs hirondelle, champignon, oiseau, paletteV et clesol est donnée en Annexe 10.1 en page 358, ainsi que les tableaux de couleurs palette1Oiseau et palette2Oiseau. Il existe une palette de couleurs par défaut (tabCoul) dans le composant Motif. Ces divers motifs et palettes sont définis dans la classe MotifLib du paquetage mdawt.

```
// PPMotif.java          Programme Principal de test du composant Motif

import java.awt.*;            // Frame
import mdpaquetage.mdawt.*;  // Motif et MotifLib

// Programme Principal Motif hérite de Frame
class PPMotif extends Frame {

  PPMotif () {

    // les caractéristiques de la fenêtre principale type Frame
    setTitle ("Avec le nouveau composant Motif");
    setBackground (Color.lightGray);  // défaut : blanc
    setBounds (10, 10, 400, 300);     // défaut : petite taille

    // mise en page par colonne de 3; 10 pixels entre composants
    setLayout (new GridLayout (0, 3, 10, 10));              // page 146

    // créer les composants
    Motif m1 = new Motif (MotifLib.hirondelle, "Hirondelle");
    Motif m2 = new Motif (MotifLib.champignon, "Amanite tue-mouche");
```

```
        Motif m3 = new Motif (MotifLib.oiseau, Motif.tProp,
                                              MotifLib.palette1Oiseau);
        Motif m4 = new Motif (MotifLib.paletteV, Motif.tVari, "Palette");
        Motif m5 = new Motif (MotifLib.clesol);
        //Button m5 = new Button ("Motif");
        Motif m6 = new Motif (MotifLib.oiseau, Motif.tFixe,
                                              MotifLib.palette2Oiseau);
        m6.setInverser();
        m6.setTaille (41, 60);                          // m6 de taille fixe

        // ajouter les composants au conteneur (de type Frame)
        add (m1);
        add (m2);
        add (m3);
        add (m4);
        add (m5);
        add (m6);

        addWindowListener (new FermerFenetre());          // voir page 221

        setVisible (true);                              // défaut : invisible
      } // constructeur PPMotif

      public static void main (String[] arg) {
        new PPMotif();
        //new PPMotif();  // si on veut un deuxième objet (fenêtre)
      } // main

    }  // PPMotif
```

Remarque : setTitle() est une méthode de la classe Frame ; setVisible(), setBackground(), setBounds() sont des méthodes de la classe Component ; setLayout() et add() sont des méthodes de la classe Container ; addWindowListener() est une méthode de la classe Window ; Motif(), setInverser() et setTaille() sont des méthodes de la classe Motif qui dérive de Component (voir figure 5.12, page 151).

La connaissance de l'arbre d'héritage est **fondamentale** pour une bonne compréhension du mécanisme de la programmation objet. Les méthodes disponibles pour un objet donné en dépendent. L'instruction suivante : `Button m5 = new Button ("Motif");` au lieu du Motif m5 dessinerait un bouton avec le libellé Motif au lieu du motif de la clé de sol. Le composant Motif se comporte de façon analogue au composant standard de Java.

Exemple de résultats : sur la figure 5.15, les motifs sont présentés sur deux lignes de trois colonnes de taille égale. Le Motif "palette de couleurs" (en bas à gauche) est de type variable et occupe tout l'espace qui lui est attribué quelle que soit sa taille. Les autres motifs sont de type proportionnel sauf le dernier oiseau qui est de type

fixe et dont sa taille ne varie pas si on modifie la taille de la fenêtre contenant les composants (voir figure 5.16).

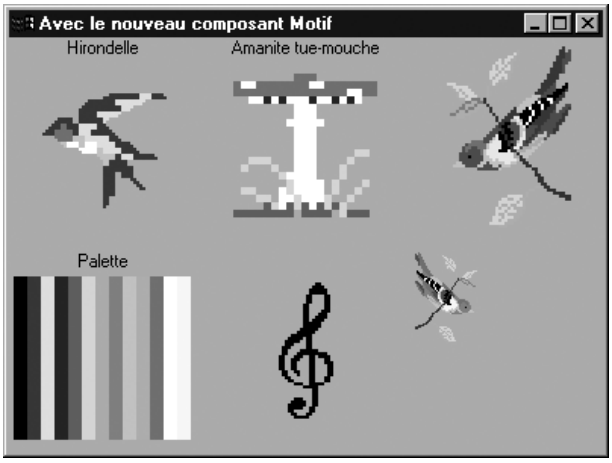


Figure 5.15 — Six composants de type Motif dans un conteneur de type Frame.

Sur la figure 5.16, la taille de la fenêtre a été modifiée. Le gestionnaire de mise en page (type GridLayout) a réparti l'espace de la fenêtre en deux lignes de trois colonnes et chaque motif (objet) a dû se redessiner en fonction de ses caractéristiques. L'oiseau de la deuxième ligne n'a pas changé de taille. La largeur de l'espace attribué ne permettant pas d'écrire le titre du champignon, le titre a disparu.

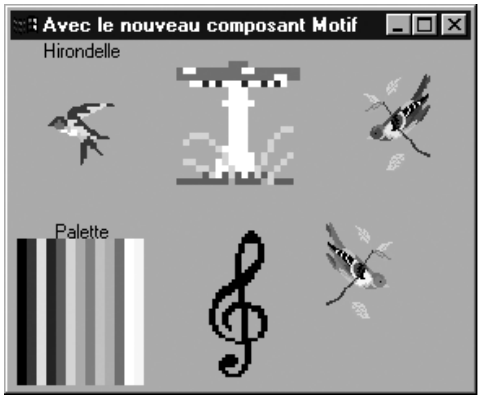


Figure 5.16 — Après réduction de la fenêtre de la figure 5.15.

Exercice 5.1 – La classe ParCinq

Écrire une classe **ParCinq** dérivée du conteneur **Panel** qui affiche 5 composants dans l'espace (le rectangle) qui lui est attribué comme indiqué sur la figure 5.17 (un dans chaque coin et un au centre) ; la figure contient trois objets ParCinq.

```
class ParCinq extends Panel {
  attributs

  public ParCinq (Component c1, Component c2,
                  Component c3, Component c4, Component c5) {
  ...
  }

  public void paint (Graphics g) {
    ...
  }

}  // class ParCinq
```

Figure 5.17 — Présentation par cinq : trois Panel de cinq Motif dans un Frame.

> Écrire la classe **DispositionP5** qui crée 3 objets de classe ParCinq dans une fenêtre de type Frame.

5.3 LA GESTION DES ÉVÉNEMENTS DES BOUTONS

5.3.1 Les écouteurs d'événements des boutons (ActionListener)

Dans l'exemple précédent, il n'y a pas d'interaction avec l'utilisateur. Les dessins s'affichent, et c'est tout. La seule interaction concerne la fenêtre qui se ferme en cliquant sur l'icône de fermeture.

La figure 5.18 présente une interface graphique comportant une série de boutons qui permettent de sélectionner un phonème (un son) du français, et de l'écrire dans la zone de texte située en dessous. Un bouton permet d'effacer la zone de texte. Pour qu'un bouton ait une action, il faut le demander en indiquant qu'on se met à l'écoute du bouton (un *Listener* en anglais) et en précisant l'objet (et donc les méthodes) qu'il faut exécuter quand ce bouton est sollicité. Pour un bouton (classe Button, voir

page 142), l'action doit être prise en compte lorsque qu'on appuie sur le bouton. Exemple :

```
Button bEffacer = new Button ("Effacer tout");
bEffacer.addActionListener (new ActionEffacer());
```

On crée le bouton bEffacer et on se met à l'écoute d'une action sur ce bouton ; si une action est réalisée sur le bouton, on exécute **actionPerformed**() de la classe **ActionEffacer.**

Dans l'exemple suivant, b est un tableau de Button. On crée un **objet ecoutePhoneme** de type ActionListener. Ce même objet ecoutePhoneme est mis à l'écoute de tous les boutons du tableau b. Si on clique sur un des boutons du tableau b, la méthode **actionPerformed()** de l'objet **ecoutePhoneme** de la classe **CaracterePhoneme** est exécutée.

```
ActionListener ecoutePhoneme = new CaracterePhoneme();
for (int i=0; i < nbPhon; i++) {
  b[i].addActionListener (ecoutePhoneme);          voir Button page 142
}
```

La méthode **actionPerformed (**ActionEvent evt**)** a un paramètre de type ActionEvent qui indique les caractéristiques de l'événement evt ayant eu lieu concernant ce bouton. evt.getActionCommand() fournit le nom (l'étiquette par défaut) du bouton concerné par cet événement (utilisée dans la classe **CaracterePhoneme**).

Exemple pour le bouton Effacer : si on clique sur le bouton Effacer, la méthode **actionPerformed**() de la classe **ActionEffacer** est exécutée. Ci-dessous, cette méthode met la chaîne vide dans la zone de texte (de type TextArea, voir 5.1.3). Un objet de la classe interne ActionEffacer peut accéder aux attributs (zoneTexte ci-dessous) et méthodes de l'objet associé qui l'a créé (voir classes internes en 2.15).

```
// classe interne
class ActionEffacer implements ActionListener {
  public void actionPerformed (ActionEvent evt) {
    zoneTexte.setText ("");    // voir TextComponent, TextArea page 145
  }
}
```

Exemple pour les boutons du tableau de boutons : si on clique sur un des boutons du tableau des phonèmes, la méthode **actionPerformed**() de la classe **CaracterePhoneme** est exécutée. evt.getActionCommand() fournit l'étiquette du bouton. Cette chaîne est ajoutée en fin du texte de la zone de texte.

```
// classe interne
class CaracterePhoneme implements ActionListener {
  public void actionPerformed (ActionEvent evt) {
    String etiquette = evt.getActionCommand();          voir Button 142
    zoneTexte.append (etiquette);                  voir TextArea en 145
  }
}
```

Dans cet exemple, le programme doit agir en fonction des actions de l'utilisateur. Il doit réagir aux événements déclenchés par l'utilisateur quand il appuie sur un bouton. L'ordre des actions n'est pas imposé par le programme. L'utilisateur a l'impression de "piloter" son programme librement.

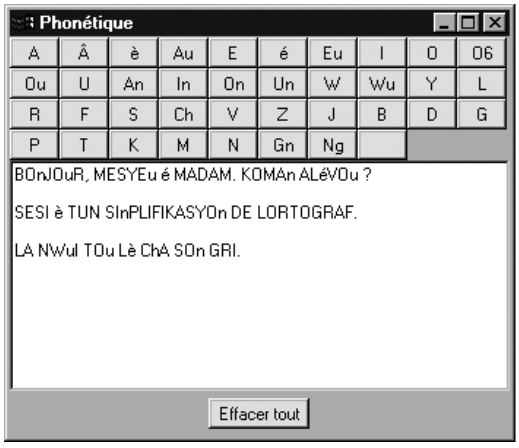


Figure 5.18 — Écriture en phonétique à l'aide d'un tableau de boutons.
À chaque son du français (phonème) correspond un bouton. L'écriture pourrait se faire en utilisant le code phonétique international (IPA). Un son serait alors écrit avec un seul caractère.

La structure d'arbre des composants (Component) est donnée sur la figure 5.19. Le conteneur principal est de type Panel. Il contient au nord un premier Panel dans lequel sont disposés les 38 boutons (Button) correspondant aux phonèmes. Le conteneur principal contient également une zone de texte au centre et un deuxième Panel au sud contenant le bouton "Effacer tout".

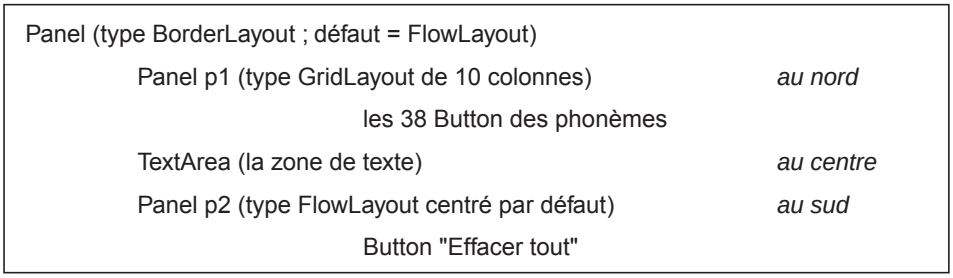


Figure 5.19 — L'arbre des composants de la figure 5.18.

5.3.2 Le composant Phonetique

La description de l'interface graphique se trouve dans le constructeur de la classe Phonetique. Ce qui concerne les actions se trouvent dans la méthode traitement() et dans les classes internes qui suivent et qui indiquent l'action des différents écouteurs.

La classe Phonetique hérite de la classe Panel (voir figure 5.12). C'est un nouveau composant.

```
//Phonetique.java  écriture en phonétique
//                 utilisation d'un tableau de boutons

package mdpaquetage.mdawt;       // classe Phonetique rangée dans mdawt

import java.awt.*;        // Graphics
import java.awt.event.*;  // ActionListener

public class Phonetique extends Panel {
  // les différents phonèmes (ou sons) du français et leur code
  static final String[] codPhon = {
    "A",  "Â",  "è",  "Au",  "E",  "é",  "Eu", "I",  "O",  "O6",
    "Ou", "U",  "An", "In",  "On", "Un", "W",  "Wu", "Y",  "L",
    "R",  "F",  "S",  "Ch",  "V",  "Z",  "J",  "B",  "D",  "G",
    "P",  "T",  "K",  "M",   "N",  "Gn", "Ng", " "
  };
  static final int nbPhon = codPhon.length;
  Button[] b;
  TextArea zoneTexte;
  Button   bEffacer;
```

Le constructeur Phonetique crée et ajoute les composants au conteneur courant de type Panel. Il appelle la méthode traitement() qui enregistre les écouteurs chargés d'effectuer les actions correspondant aux événements provenant des boutons.

```
public Phonetique () {
    setBackground (Color.lightGray);
    setLayout     (new BorderLayout());              // Panel principal

    // p1 : Panel des boutons des phonèmes au nord     voir figure 5.19
    // type GridLayout pour p1, nb colonnes = 10
    Panel p1 = new Panel (new GridLayout (0, 10));
    b = new Button [nbPhon];  // tableau de boutons (codes phonétiques)
    for (int i=0; i < nbPhon; i++) {
      b[i] = new Button (codPhon[i]);
      b[i].setBackground (Color.cyan);
      p1.add (b[i]);
    }
    add (p1, "North");

    // la zone de texte au centre
    zoneTexte = new TextArea ("");
    add (zoneTexte, "Center");

    // le bouton "Effacer tout" dans panel p2 au sud
    bEffacer = new Button ("Effacer tout");
    bEffacer.setBackground (Color.cyan);
    Panel p2 = new Panel();
    p2.add (bEffacer);
    add (p2,"South");
```

```
    System.out.print (
      "\ngetComponentCount()    : " + getComponentCount()    +
      "\nparamString()          : " + paramString()          +
      "\np1.getComponentCount() : " + p1.getComponentCount() +
      "\np2.getComponentCount() : " + p2.getComponentCount()
    );
    setVisible (true);

    traitement();
  }  // constructeur Phonetique
```

La méthode de traitement et les différentes classes internes (voir § 2.15) chargées du traitement.

```
   // GESTION DES EVENEMENTS

  void traitement() {
    // prendre en compte les boutons des phonèmes
    ActionListener ecoutePhoneme = new CaracterePhoneme();
    for (int i=0; i < nbPhon; i++) {
      b[i].addActionListener (ecoutePhoneme);
    }
    // prendre en compte l'action du bouton Effacer
    bEffacer.addActionListener (new ActionEffacer());
  }

  // classes internes à la classe Phonetique              voir page 91

  // un bouton phonème appuyé
  class CaracterePhoneme implements ActionListener {
    public void actionPerformed (ActionEvent evt) {
      String etiquette  = evt.getActionCommand();        voir page 142
      zoneTexte.append (etiquette);                      voir page 145
    }
  }

  // pour le bouton Effacer
  class ActionEffacer implements ActionListener {
    public void actionPerformed (ActionEvent evt) {
      zoneTexte.setText ("");
    }
  }

}  // class Phonetique
```

Les instructions suivantes en fin du constructeur de Phonetique permettent de vérifier le nombre de composants ajoutés à chaque conteneur (le Panel racine et les deux Panel).

```
System.out.print (
  "\ngetComponentCount()    : " + getComponentCount()    +  // page 150
  "\nparamString()          : " + paramString()          +  // de Panel
  "\np1.getComponentCount() : " + p1.getComponentCount() +
  "\np2.getComponentCount() : " + p2.getComponentCount()
);
```

Le résultat est le suivant. Il y a bien trois composants au niveau de Panel. Le Panel p1 contient bien les 38 boutons, et le Panel p2 le bouton Effacer (voir figure 5.19).

```
getComponentCount()    : 3
paramString() : panel0,0,0,0x0,invalid,layout = java.awt.BorderLayout
p1.getComponentCount() : 38                    les 38 boutons phonèmes
p2.getComponentCount() : 1                    le bouton "Effacer tout"
```

5.3.3 La mise en œuvre du composant Phonetique dans une application

La classe Phonetique constitue un nouveau composant qu'on peut ajouter par exemple à un conteneur Frame définissant une fenêtre principale d'application.

```
//PPPhonetique.java  Programme Principal Phonétique;
//                   utilisation du composant Phonetique

import java.awt.*;              // Frame
import mdpaquetage.mdawt.*;     // Phonetique, FermerFenetre

class PPPhonetique extends Frame {

  PPPhonetique () {
    setTitle  ("Phonétique");                      // du conteneur Frame
    setBounds (20, 20, 350, 300);
    add (new Phonetique(), "Center");         // BorderLayout par défaut
    addWindowListener (new FermerFenetre());                // voir 221
    setVisible (true);
  }

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

}  // PPPhonetique
```

Voir le résultat d'exécution sur la figure 5.18.

5.4 LES MENUS DÉROULANTS

La figure 5.20 montre un exemple de menu déroulant qu'on peut ajouter seulement à une fenêtre de type Frame sous AWT. On y distingue un objet de type MenuBar, deux objets de type Menu avec les étiquettes "Cercles ou Rosaces" et "Quitter", et pour le premier Menu, huit éléments de menu de type MenuItem. Un ou plusieurs des MenuItem pourraient être de type Menu, ce qui constituerait un arbre de Menu et MenuItem.

Tous les objets concernant le menu dérivent de la classe abstraite (voir page 96) **MenuComponent** qui regroupe les caractéristiques communes des objets concernant les menus (voir figure 5.21). On y trouve essentiellement le type de la fonte qui peut être défini individuellement pour chaque objet, et un nom qui peut être utilisé lors du traitement.

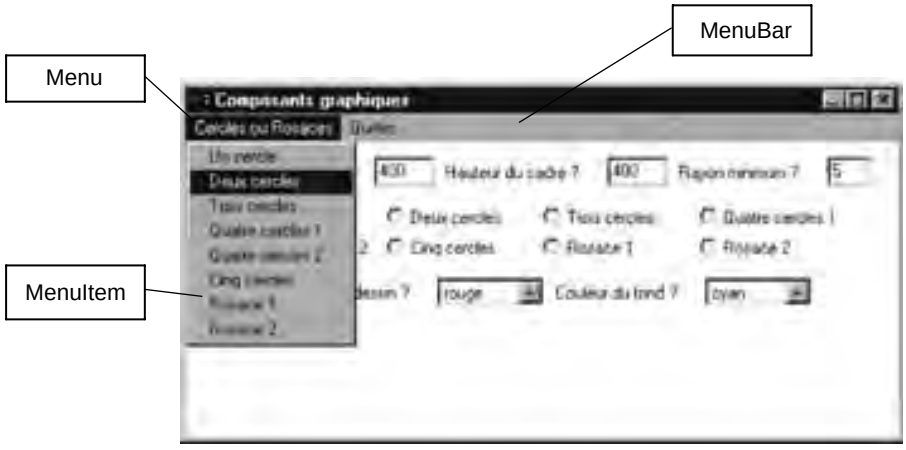


Figure 5.20 — Un exemple de menu déroulant.
On a cliqué sur le premier Menu qui est ouvert et présente huit éléments de Menu.

Principales méthodes de MenuComponent :

- Font **getFont** () : fournit la fonte d'un objet de type MenuComponent.
- void **setFont** (Font f) : définit la fonte pour cet objet.
- String **getName** () : nom de l'objet.
- void **setName** (String nom) : définit le nom qui est fourni par getName().

La classe **MenuBar** caractérise la barre de menus. On peut lui ajouter des menus. La figure 5.20 en compte deux. On peut obtenir le nombre de Menu dans la barre (getMenuCount()), accéder à un Menu à partir de son rang, supprimer un Menu de la barre à partir de son rang ou à partir de la référence sur le Menu à détruire.

Principales méthodes de MenuBar :

- **MenuBar** () : constructeur d'une barre de menu.
- Menu **add** (Menu m) : ajoute le Menu m à la barre de menu.
- int **getMenuCount** () : indique le nombre de Menu pour cette barre de menus.
- Menu **getMenu** (int n) : fournit le n ième Menu.
- void **remove** (int n) : supprime le n ième Menu.
- void **remove** (MenuComponent m) : supprime le Menu de référence m.

La classe **MenuItem** caractérise un élément de menu : celui-ci a une étiquette. Il peut être ou non accessible. S'il est inaccessible, il apparaît dans le menu avec un grisé plus clair, mais il ne peut être choisi. La méthode setEnabled() valide ou invalide un élément de menu. La méthode isEnabled() teste si l'élément est valide ou non. On peut enregistrer un écouteur pour un élément d'un menu avec addActionListener() ou le retirer avec removeActionListener().

Principales méthodes de MenuItem :

- **MenuItem** () : crée un élément de menu sans étiquette.
- **MenuItem** (String s) : crée un élément de menu avec l'étiquette s.

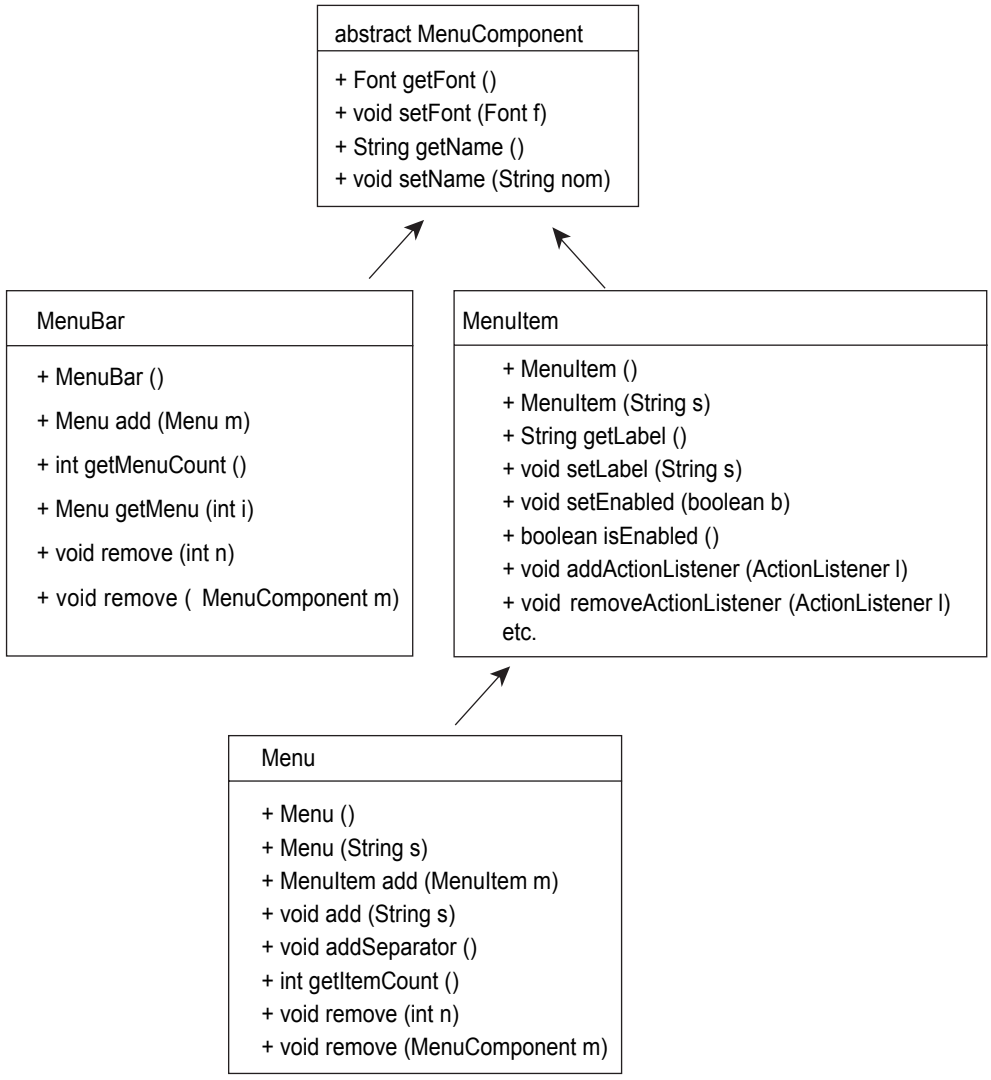


Figure 5.21 — L'arbre d'héritage pour les menus.

- String **getLabel** () : fournit l'étiquette de l'élément de menu.
- void **setLabel** (String s) : définit l'étiquette de l'élément de menu.
- void **setEnabled** (boolean b) : définit si l'élément est valide ou non suivant la valeur de b.
- boolean **isEnabled** () : indique si l'élément est valide ou non.
- void **addActionListener** (ActionListener l) : spécifie l'objet écouteur pour cet élément.
- void **removeActionListener** (ActionListener l) : supprime l'écoute de cet élément de menu.

La classe **Menu** définit un menu de la barre de menus et permet d'ajouter ou d'enlever des éléments au menu, d'obtenir la référence d'un des éléments, de compter le nombre d'éléments, etc.

Principales méthodes de Menu :

- **Menu** () : crée un nouveau menu.
- **Menu** (String s) : crée un nouveau menu d'étiquette s.
- MenuItem **add** (MenuItem m) : ajoute un élément au menu.
- void **add** (String s) : ajoute s au menu.
- void **addSeparator** () : ajoute une barre séparant les éléments du menu.
- int **getItemCount** () : fournit le nombre d'éléments dans ce menu.
- void **remove** (int n) : enlève le nième élément du menu.

L'arbre d'héritage est présenté sur la figure 5.21.

Le menu de la figure 5.20 est créé comme suit. On crée d'abord la barre de menus référencée par la variable `barre`. La méthode setMenuBar() de la classe Frame installe la barre de menus référencée par `barre`. On pourrait avoir plusieurs barres de menus ; une seule est installée à la fois. On crée le premier menu avec son titre "Cercles ou Rosaces" qui est ajouté à la barre de menu (méthode add() de MenuBar). On crée ensuite un tableau menu1 d'éléments de menu (de MenuItem) à partir du tableau choixLibelle[] de chaînes de caractères. Ces éléments sont ajoutés à menu1. On fait de même pour le deuxième menu de titre "Quitter". On lui ajoute un seul élément "Fin de Programme".

```
static final String[] choixLibelle = {
  "Un cercle",        "Deux cercles",     "Trois cercles",
  "Quatre cercles 1", "Quatre cercles 2", "Cinq cercles",
  "Rosace 1",         "Rosace 2"
};

// les menus
MenuBar barre = new MenuBar();                    // classe MenuBar
setMenuBar (barre);                 // classe Frame; voir page 151

// Première colonne de menu
Menu menuCercle = new Menu ("Cercles ou Rosaces");
barre.add (menuCercle);                           // classe MenuBar
MenuItem[] menu1 = new MenuItem [choixLibelle.length];
for (int i=0; i < choixLibelle.length; i++) {
  menu1[i] = new MenuItem (choixLibelle[i]);
  menuCercle.add (menu1[i]);                          // classe Menu
}

// Deuxième colonne du menu
Menu menuQuitter = new Menu ("Quitter");
barre.add (menuQuitter);                          // classe MenuBar
MenuItem menu2 = new MenuItem ("Fin du programme");
menuQuitter.add (menu2);                            // classe Menu
```

5.5 LA GESTION DES ÉVÉNEMENTS DES COMPOSANTS ET DES MENUS

L'exemple de l'écriture en phonétique précédent (voir § 5.3, page 163) a montré comment gérer les événements des boutons. L'exemple proposé dans ce paragraphe gère des événements variés concernant des boutons, des boîtes à cocher, des boîtes de choix, des zones de texte et une zone de dessin. L'exemple utilise une panoplie assez complète de composants AWT.

Cet exemple permet de dessiner dans une zone de dessin des formes géométriques basées sur des cercles et qu'on peut faire varier à l'aide de différents paramètres entrés grâce à l'interface graphique. On peut choisir la largeur et la hauteur de la zone de dessin ainsi qu'un rayon minimum pour le tracé récursif des cercles. Un groupe de boîtes à cocher permet de choisir le dessin à afficher dans la zone de dessin. Les couleurs du dessin et de l'arrière-plan du dessin peuvent être choisies à l'aide de deux boîtes de choix proposant un éventail de couleurs.

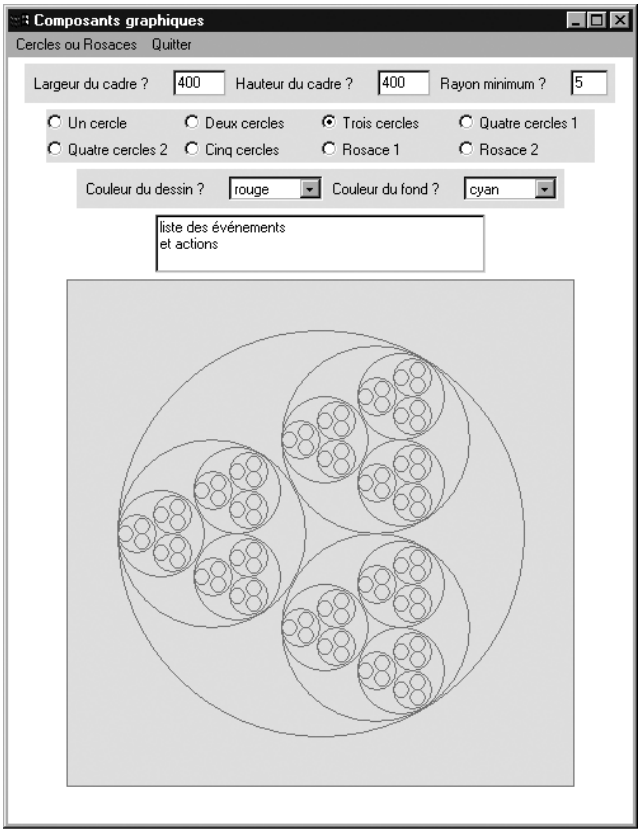


Figure 5.22 — Composants graphiques : gestion des événements et des menus.
Les trois conteneurs de la classe Panel sont mis en évidence (pour l'explication) grâce à une couleur de fond différente de celle de la fenêtre principale.

Dans un but pédagogique et pour vérification, une trace des événements est affichée dans la zone de texte. Un menu avec deux choix au niveau de la barre de menus fournit une autre façon de sélectionner un des dessins. Si on clique sur le premier menu, il s'ouvre et propose les mêmes choix que les cases à cocher. Le deuxième menu permet seulement de quitter l'application (similaire à la case de fermeture de la fenêtre).

5.5.1 L'interface graphique ComposantsDemo

5.5.1.1 Les différents composants de cette interface ComposantsDemo

ComposantsDemo est un nouveau composant formé à partir d'autres composants standard AWT. L'arbre des composants de l'interface est donné aux figures 5.11 et 5.23. Sa création ne présente aucune difficulté puisqu'il s'agit de créer des composants en utilisant un des constructeurs du composant, et en insérant ce composant dans son conteneur (Panel principal ou Panel p1, p2 ou p3 sur cet exemple). Il faut cependant mentionner certaines spécificités de certains composants (voir 5.1.3, page 141) :

- les zones de saisie (TextField) sont initialisées avec une valeur qui est d'abord convertie en une chaîne de caractères par Integer.toString (int).
- pour les cases à cocher (Checkbox), il faut d'abord créer un groupe de cases, et créer ensuite chaque case à cocher en indiquant le groupe dont elle fait partie (un des constructeurs de Checkbox a un paramètre de ce type).
- pour les boîtes d'options (Choice), il faut ajouter (add) les différentes options à la boîte, et éventuellement indiquer la valeur sélectionnée (select) au départ.

```
Frame
    Panel principal ComposantsDemo (FlowLayout)
        Panel p1 (FlowLayout)
            3 TextField (tF1, tF2, tF3) précédés de leur Label (l1, l2, l3)
        Panel p2 (GridLayout de 4 colonnes)
            tableau tabChoix de Checkbox
            dans un même groupe cbG (CheckboxGroup)
        Panel p3 (FlowLayout)
            2 Choice (boîtes de choix) : choixCT, choixCF et leur Label (l4, l5)
        TextArea zoneText
        FigGeo dessin : zone de dessin de la classe FigGeo
    MenuBar barre
        Menu menuCercle
            tableau menu1 de 8 MenuItem
        Menu menuQuitter
            MenuItem menu2
```

Figure 5.23 — Arbre des composants de la figure 5.22.

5.5.1.2 Le programme Java du composant ComposantsDemo

On définit comme attributs toutes les références des objets de cette interface graphique. Les objets non référencés en dehors du constructeur : certaines étiquettes (Label) comme l1, l2, etc., sont déclarés localement dans le constructeur. Ce dernier crée tous les objets de l'interface graphique suivant la structure de la figure 5.23.

```
// ComposantsDemo.java

package mdpaquetage.mdawt;

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

public class ComposantsDemo extends Panel {
  TextField  tF1, tF2, tF3; // text Field 1, 2, 3
  Checkbox[] tabChoix;      // tableau des Checkbox
  Choice     choixCT;       // choix Couleur du Texte
  Choice     choixCF;       // choix Couleur du Fond
  TextArea   zoneText;      // zone de texte
  int        lDessin = 400; // largeur du dessin        FigGeo page 181
  int        hDessin = 400; // hauteur du dessin
  int        figInit = 2;   // figure initiale du dessin
  Color      c       = Color.red;  // couleur premier plan du dessin
  Color      cF      = Color.cyan; // couleur du fond      du dessin
  FigGeo     dessin  = new FigGeo (figInit, c, cF);          voir 5.5.2
  CheckboxGroup cbG;        // le groupe des Checkbox
  // les libellés des Checkbox
  static final String[] choixLibelle = {
    "Un cercle",        "Deux cercles",     "Trois cercles",
    "Quatre cercles 1", "Quatre cercles 2", "Cinq cercles",
    "Rosace 1",         "Rosace 2"
  };

  // les accesseurs
  public static String[] choixLibelle ( ) { return choixLibelle; };
  public TextArea zoneText ()   { return zoneText; };
  public FigGeo   dessin   ()   { return dessin; };
  public Checkbox[] tabChoix () { return tabChoix; };

  public ComposantsDemo () {
    // les caractéristiques du Panel principal
    setBackground (Color.white);                // défaut : FlowLayout
    setBounds (20, 20, 500, 650);

    Panel panel1 = new Panel();                              // page 153
    add (panel1);

    // Les zones de saisies (précédées de leur libellé)
    Label l1  = new Label ("Largeur du cadre ? ");           // page 142
    tF1       = new TextField (Integer.toString (lDessin));
    tF1.setBackground (Color.white);
    panel1.add (l1);
    panel1.add (tF1);
```

```
Label l2  = new Label ("Hauteur du cadre ? ");
tF2       = new TextField (Integer.toString (hDessin));
tF2.setBackground (Color.white);
panel1.add (l2);
panel1.add (tF2);

Label l3  = new Label ("Rayon minimum ? ");
tF3       = new TextField (Integer.toString (dessin.getRMin()));
tF3.setBackground (Color.white);
panel1.add (l3);
panel1.add (tF3);

// Les différentes boîtes à cocher
Panel panel2 = new Panel (new GridLayout (0,4));        // page 146
cbG = new CheckboxGroup();                      // création du groupe
// on crée un tableau de Checkbox
tabChoix = new Checkbox [choixLibelle.length];
for (int i=0; i < tabChoix.length; i++) {
  // les Checkbox sont ajoutées au groupe cbG
  tabChoix[i] = new Checkbox (choixLibelle[i], cbG,
                                    i ==figInit ? true : false);
  panel2.add (tabChoix[i]);
}
add (panel2);

// Les 2 boites de choix (de couleur)
Panel panel3 = new Panel();
Label l4 = new Label ("Couleur du dessin ? ");
panel3.add (l4);
choixCT = new Choice ();                          // Couleur du Texte
for (int i=0; i < Couleur.nbCouleur; i++) {
  choixCT.add (Couleur.nomCouleur(i));              // voir page 70
}
choixCT.select (Couleur.numeroCouleur(c));
panel3.add (choixCT);

Label l5 = new Label ("Couleur du fond ? ");
panel3.add (l5);
choixCF = new Choice ();  // Couleur du fond
for (int i=0; i < Couleur.nbCouleur; i++) {
  choixCF.add (Couleur.nomCouleur(i));
}
choixCF.select (Couleur.numeroCouleur(cF));
panel3.add (choixCF);
add (panel3);

// La zone de texte (indiquant les actions en cours
zoneText = new TextArea ("liste des événements\net actions",
                                                          2, 40);
add (zoneText);

// La zone de dessin                               // FigGeo page 181
dessin.setSize (lDessin, hDessin);
```

```
    add (dessin);

    // pour mettre en évidence l'espace grisé des 3 Panel
    //panel1.setBackground (Color.cyan);
    //panel2.setBackground (Color.cyan);
    //panel3.setBackground (Color.cyan);

    setVisible (true);            // on affiche quand tout est en place

    traitement();

  }  // constructeur ComposantsDemo
```

5.5.1.3 Le traitement des événements (ActionListener, ItemListener)

Comme pour les boutons dans l'exemple précédent (voir § 5.3, page 163), il faut indiquer les composants qui peuvent déclencher une **action** et les **mettre sur écouteur** (sur Listener). Button, TextField et MenuItem déclenchent une action ; ils utilisent **addActionListener**() pour enregistrer l'objet qui doit traiter l'action. Ci-dessous, les objets tF1, tF2, tF3, les 8 MenuItem de menu1 et le MenuItem de menu2 utilise un **ActionListener** (voir figure 5.12, page 151).

Pour d'autres composants, il y a un **choix parmi plusieurs éléments**. C'est le cas des cases à cocher d'un groupe ou des boîtes de choix. Ces composants sont écoutés par un **ItemListener**. Le tableau tabChoix des cases à cocher et les boîtes de choix choixCT et choixCF utilisent **addItemListener**() pour enregistrer l'objet chargé de les surveiller et de traiter l'événement.

```
// TRAITEMENT DES COMPOSANTS

  void traitement () {  // les Listener
    // les TextField
    tF1.addActionListener (new Largeur());
    tF2.addActionListener (new Hauteur());
    tF3.addActionListener (new Rayon());
    // les cases à cocher : un même écouteur pour les 8 cases
    ItemListener ecouteCase = new TypeFigures();
    for (int i=0; i < tabChoix.length; i++) {
      tabChoix[i].addItemListener (ecouteCase);
    }
    // les 2 boîtes de choix (une seule classe pour les 2)
    // mais deux objets créés
    // on peut n'en créer qu'un comme pour les cases à cocher
    choixCT.addItemListener (new ChoixCouleurs());
    choixCF.addItemListener (new ChoixCouleurs());
  }  // traitement
```

Les traitements à faire suivant les événements intervenus sont définis dans les méthodes des classes de chaque Listener. Chaque type de Listener doit redéfinir une ou plusieurs méthodes dont le prototype est donné dans une interface (voir page 106) référencée par "implements ActionListener" ou "implements ItemListener". Pour un ActionListener, il faut redéfinir la méthode : void **actionPerformed** (ActionEvent evt) dont le prototype est défini dans l'interface **ActionListener**. Pour un ItemListener,

il faut redéfinir la méthode : void **itemStateChanged** (ItemEvent evt) dont le prototype se trouve dans l'interface **ItemListener**.

La méthode **actionPerformed**() de la classe interne Largeur a un paramètre evt contenant les caractéristiques de l'événement ayant conduit à cet appel. La méthode getText() fournit la chaîne de caractères du TextField tF1 (seul abonné sur ce listener). Cette chaîne est réécrite pour vérification dans la zone de texte, et convertie en binaire grâce à la méthode parseInt(). On modifie en conséquence les dimensions du composant FigGeo (voir page 181). Le principe est le même pour gérer les événements concernant les TextField modifiant la hauteur ou le rayon. Voir TextField page 145.

```
// classes internes accédant aux attributs de ComposantsDemo
// comme choixCT, zoneText, etc. (voir classes internes, page 89)

// Largeur du dessin
// actionPerformed() déclenchée si on modifie le TextField largeur
class Largeur implements ActionListener {
  public void actionPerformed (ActionEvent evt) {
    String s = tF1.getText ();
    zoneText.setText ("action Largeur\n" + s);
    lDessin  = Integer.parseInt (s);
    dessin.setSize (lDessin, hDessin);  // setSize de Component
  }
}

// Hauteur du dessin
// actionPerformed() déclenchée si on modifie le TextField hauteur
class Hauteur implements ActionListener {
  public void actionPerformed (ActionEvent evt) {
    String s = tF2.getText ();
    zoneText.setText ("action Hauteur\n" + s);
    hDessin  = Integer.parseInt (s);
    dessin.setSize (lDessin, hDessin);  // setSize de Component
  }
}

// Rayon minimum du dessin
// actionPerformed() déclenchée si on modifie le TextField rayon
class Rayon implements ActionListener {
  public void actionPerformed (ActionEvent evt) {
    String s = tF3.getText ();
    zoneText.setText ("action Rayon\n" + s);
    dessin.setRMin (Integer.parseInt (s));
  }
}
```

La méthode **itemStateChanged**() de la classe interne **typeFigures** a un paramètre evt contenant les caractéristiques de l'événement ayant conduit à cet appel. La méthode getSource() fournit la référence sur la boîte à cocher activée (**Checkbox**). Comme les références ont été sauvegardées dans le tableau tabChoix[], on peut retrouver le numéro de la boîte à cocher activée. Ce numéro est communiqué à l'objet dessin

par la méthode setNumFig() qui demande alors à l'objet `dessin` de se redessiner à l'aide de la méthode repaint(). Voir Checkbox page 143.

```
// Bouton radio (typeFigures)
// itemStateChanged() déclenchée si on modifie un des éléments
// du groupe de boîtes à cocher
class TypeFigures implements ItemListener {
  public void itemStateChanged (ItemEvent evt) {
    Checkbox ch     = (Checkbox) evt.getSource();
    // retrouver le numéro de la Checkbox
    int      n      = 0;
    boolean  trouve = false;
    while (!trouve && n < tabChoix.length){
      trouve = ch == tabChoix[n];
      if (!trouve) n++;
    }
    dessin.setNumFig (n);
    String s = (String) ch.getLabel();
    zoneText.setText ("boîtes à cocher : " + s + "\n"
                      + "de numéro : " + n);
  }
}
```

Modification de la couleur du dessin ou de la couleur du fond du dessin grâce aux deux boîtes de choix (**Choice**). getSource() permet d'identifier la boîte activée et getSelectedIndex() fournit le numéro du choix sélectionné. Voir Choice, page 144.

```
// Choice (colors)
// itemStateChanged() déclenchée si on modifie un des éléments
// d'une des boites de choix : les 2 boîtes ont des écouteurs
// de la même classe
class ChoixCouleurs implements ItemListener {
  public void itemStateChanged (ItemEvent evt) {
    Choice ch = (Choice) evt.getSource();
    int nC = ch.getSelectedIndex();      // numéro de couleur choisie
    if (ch == choixCT) {
      dessin.setCouleur (Couleur.getColor(nC));
    } else {
      dessin.setCouleurF (Couleur.getColor(nC));
    }

    zoneText.setText ("boîtes de choix de couleurs "
      + ((ch == choixCT) ? "du texte : " : "du fond : ")
      + (String) ch.getSelectedItem() + "\n"      // libellé du choix
      + "numéro de couleur : " + nC );
  }
}  // ChoixCouleurs

}  // class ComposantsDemo
```

5.5.1.4 Exemple d'utilisation du composant ComposantsDemo

La mise en œuvre du composant dans une application se fait en ajoutant le composant à une fenêtre de type Frame.

```
// PPComposantsDemo.java

import java.awt.*;  // Frame
import mdpaquetage.mdawt.*;  // classes ComposantsDemo, FermerFenetre

// pour une application sans menu
class PPComposantsDemo extends Frame {

  PPComposantsDemo () {
    setTitle ("ComposantsDemo");
    setBounds (10, 10, 500, 650);
    ComposantsDemo p = new ComposantsDemo();
    add (p, "Center");                          // défaut : BorderLayout
    addWindowListener (new FermerFenetre());                // voir 221
    setVisible (true);
  } // PPComposantsDemo

  public static void main (String[] args) {
    new PPComposantsDemo();                                  // premier objet
    PPComposantsDemo c2 = new PPComposantsDemo();           // deuxième objet
    c2.setLocation (600, 10);
  }

}  // PPComposantsDemo
```

Les deux schémas de la figure 5.24 correspondent à l'utilisation du composant ComposansDemo.

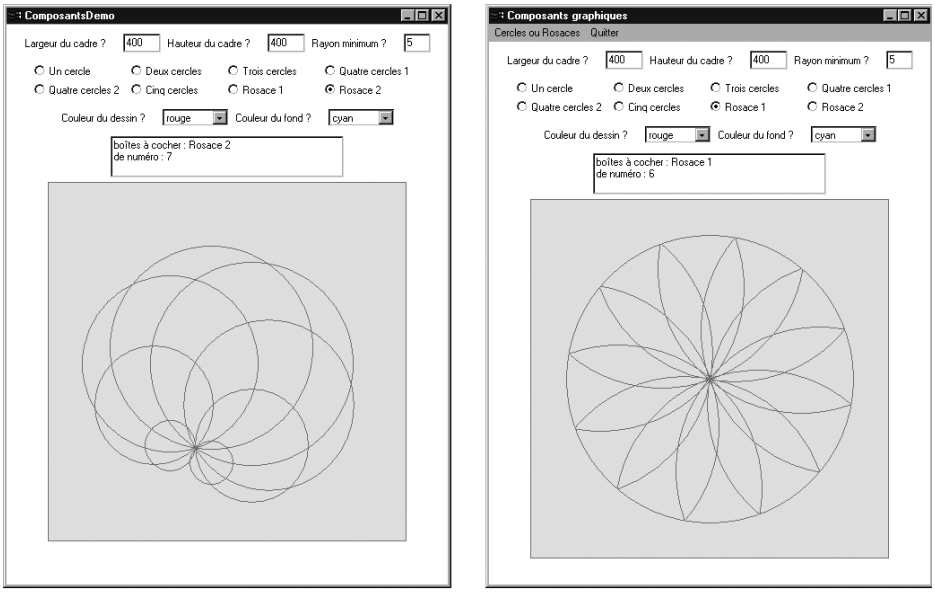


Figure 5.24 — Exemples de dessins de cercles et de rosaces.
Le deuxième schéma comporte un menu déroulant.

5.5.1.5 Exemple d'utilisation d'un menu déroulant

L'ajout d'un menu déroulant à la fenêtre principale de type Frame se fait comme expliqué en 5.4 en ajoutant une barre de menus au conteneur de la fenêtre principale de type Frame et en définissant des écouteurs pour les éléments de menus (MenuItem).

```
// PPComposantsDemoMenu.java avec menu déroulants

import java.awt.*;           // Frame
import java.awt.event.*;     // ActionListener
import mdpaquetage.mdawt.*;  // classses ComposantsDemo, FermerFenetre

class PPComposantsDemoMenu extends Frame {
  // tableau des libellés des Checkbox de ComposantsDemo
  static String[] choixLibelle = ComposantsDemo.choixLibelle();
  ComposantsDemo comp = new ComposantsDemo ();
  TextArea   zoneText = comp.zoneText();
  FigGeo     dessin   = comp.dessin();
  Checkbox[] tabChoix = comp.tabChoix();
  MenuItem[] menu1;
  MenuItem   menu2;

  PPComposantsDemoMenu () {
    setTitle  ("ComposantsDemo avec menu");                 // du Frame
    setBounds (10, 10, 550, 650);

    // Les menus
    MenuBar barre = new MenuBar();
    setMenuBar (barre);
    // Première colonne de menu
    Menu menuCercle = new Menu ("Cercles ou Rosaces");
    barre.add (menuCercle);
    menu1 = new MenuItem [choixLibelle.length];
    for (int i=0; i < choixLibelle.length; i++) {
      menu1[i] = new MenuItem (choixLibelle[i]);
      menuCercle.add (menu1[i]);
    }
    // Deuxième colonne du menu
    Menu menuQuitter = new Menu ("Quitter");
    barre.add (menuQuitter);
    menu2 = new MenuItem ("Fin du programme");
    menuQuitter.add (menu2);

    // un même écouteur pour les 8 items du premier menu
    ActionListener ecouteMenu = new ActionMenu1();
    for (int i=0; i < choixLibelle.length; i++) {
      menu1[i].addActionListener (ecouteMenu);
    }
    // un écouteur pour l'item du second menu
    menu2.addActionListener (new ActionQuitter());
    add (comp, "Center");                       // défaut : BorderLayout

    // la fenêtre de type Frame
    addWindowListener (new FermerFenetre());          // voir page 221

    setVisible (true);
  }
```

Le traitement du premier **menu** de la barre des menus est confié à la classe **actionMenu1**. Il faut retrouver le numéro de l'élément de menu activé et le communiquer à l'objet dessin pour qu'il se redessine. Il faut également changer en conséquence (du choix du menu), l'état de la boîte à cocher.

```
// pour la première colonne du menu
// actionPerformed() déclenchée si on sélectionne
// un élément du premier menu : tous les éléments
// ont le même écouteur
class ActionMenu1 implements ActionListener {
  public void actionPerformed (ActionEvent evt) {
    MenuItem ch = (MenuItem) evt.getSource();
    int      n  = 0;
    boolean  trouve = false;
    while (!trouve && n < menu1.length){
      trouve = ch == menu1[n];
      if (!trouve) n++;
    }
    dessin.setNumFig (n);
    // boîte à cocher à changer en fonction du menu choisi
    tabChoix[n].setState (true);

    String S = (String) ch.getLabel();
    zoneText.setText ("Menu 1 : " + S + ",\n"
                      + "numéro du choix : " + n );
  }
}
```

Pour le deuxième menu, le seul choix est de quitter l'application.

```
// pour "Fin de programme" du menu
class ActionQuitter implements ActionListener {
  public void actionPerformed (ActionEvent evt) {
    System.exit (0);
  }
}
```

La méthode main() crée deux objets de type PPComposantsDemoMenu.

```
public static void main (String[] args) {
    new PPComposantsDemoMenu();
    PPComposantsDemoMenu c2 = new PPComposantsDemoMenu();
    c2.setLocation (600, 10);
  }

}  // PPComposantsDemoMenu
```

Le deuxième schéma de la figure 5.24 comporte un menu déroulant correspondant au programme ci-dessus.

5.5.2 La classe *FigGeo* d'affichage du dessin

Les composants Button, TextField, etc., ont leur propre méthode paint() pour se redessiner lorsque la fenêtre change de taille ou lorsqu'elle revient au premier plan

après avoir été masquée par exemple. Le composant Motif créé en 5.2.1, page 153 a sa propre méthode paint(). La zone de dessin des figures géométriques de la classe FigGeo est un objet qui connaissant le numéro de figure à dessiner et les couleurs du texte et du fond du dessin peut refaire le dessin quand il le faut. Les cercles dessinés récursivement le sont dès lors que le rayon est supérieur à un rayon minimum rMin.

Les dimensions de la zone de dessin sont connues par l'intermédiaire de la méthode getSize() qui fournit les dimensions attribuées au composant au moment de le repeindre, ce qui est le rôle de la méthode paint(). Des méthodes (accesseurs) permettent de connaître ou de changer ces paramètres de l'extérieur de la classe FigGeo, les attributs étant privés. La classe FigGeo est rangée dans le paquetage mdawt.

```
// FigGeo.java

package mdpaquetage.mdawt;                          // paquetage mdawt
import java.awt.*;

public class FigGeo extends Canvas {
  private int    rMin = 5;      // rayon Minimum des cercles récursifs
  private int    numFig;        // numéro de Figures de 0 à 7
  private int    angle  = 250;  // angle pour point commun rosace2
  private double module = 0.8;  // module pour point commun rosace2
                                // unité = rayon du cercle

  // numéro de figure, couleur premier et arrière-plan
  public FigGeo (int numFig, Color couleur, Color couleurF) {
    this.numFig = numFig;
    setForeground (couleur);  // de Component
    setBackground (couleurF);
  }

  // numéro de figure
  public FigGeo (int numFig) {
    this (numFig, Color.red, Color.cyan);
  }

  // pour FlowLayout et BorderLayout
  public Dimension getPreferredSize () {
    return new Dimension (400, 400);
  }

  // les accesseurs : accéder ou modifier un attribut
  public int   getRMin ()      { return rMin;     }
  public int   getNumFig ()    { return numFig;   }
  public void  setRMin (int rMin) {
    this.rMin = rMin;
    repaint();  // il faut redessiner
  }
  public void setCouleur (Color couleur) {
    setForeground (couleur);
    repaint();
  }
```

```
public void setCouleurF (Color couleurF) {
  setBackground (couleurF);
  repaint();
}
public void  setNumFig   (int numFig)     {
  this.numFig = numFig;
  repaint();
}

// trace un cercle de centre x, y et de rayon gR
void cercle (Graphics g, int x, int y, int gR) {
  g.drawOval (x-gR, y-gR, 2*gR, 2*gR);              // voir page 137

}

// trace récursivement un cercle de rayon gR centré en (x,y)
void unCercle (Graphics g, int x, int y, int gR) {
  if (gR > rMin) {
    cercle (g, x, y, gR);
    int pR = gR/2;             // gR : grand Rayon; pR : petit Rayon
    unCercle (g, x, y, pR);                          // appel récursif
  }
}

// trace récursivement deux cercles
// le cercle initial est centré en (x,y) et a pour rayon gR
void  deuxCercles (Graphics g, int x, int y, int gR) {
  if (gR > rMin) {
    cercle (g, x, y, gR);
    int pR = gR/2;
    deuxCercles (g, x+pR, y, pR);                 // appels récursifs
    deuxCercles (g, x-pR, y, pR);
  }
}

// trace récursivement trois cercles;
// le cercle initial est centré en (x,y) et a pour rayon gR
void troisCercles (Graphics g, int x, int y, int gR) {
  if (gR > rMin) {
    cercle (g, x, y, gR);
    int pR = (int) (0.4641*gR);
    //int pR = (int) ((2*Math.sqrt(3)-3)*gR);  // = 0.4641*gR
    int h = gR - pR;
    troisCercles (g, x-h,          y,    pR);     // appels récursifs
    troisCercles (g, (int)(x+h/2), y+pR, pR);
    troisCercles (g, (int)(x+h/2), y-pR, pR);
  }
}

// quatre cercles non récursifs
void quatreCercles1 (Graphics g, int x, int y, int gR) {
    //cercle (g, x, y, gR);  // cercle englobant = 5 cercles
    int pR = gR/2;
```

```
    cercle (g, x-pR, y,    pR);
    cercle (g, x+pR, y,    pR);
    cercle (g, x,    y+pR, pR);
    cercle (g, x,    y-pR, pR);
}

// quatre cercles non récursifs
// en fait, 4 fois trois quarts de cercle (270 degrés)
void quatreCercles2 (Graphics g, int x, int y, int gR) {
    g.drawArc (x-gR, y-gR, gR, gR,   0, 270);        // voir page 137

    g.drawArc (x,    y-gR, gR, gR, -90, 270);
    g.drawArc (x,    y,    gR, gR, 180, 270);
    g.drawArc (x-gR, y,    gR, gR,  90, 270);
}

// cinq cercles non récursifs
// un cercle central + le dessin précédent
void cinqCercles (Graphics g, int x, int y, int gR) {
    cercle (g, x, y, gR/2);
    quatreCercles2 (g, x, y, gR);
}

// rosace double
void rosace1 (Graphics g, int x, int y, int gR) {
  cercle (g, x, y, gR);                          // le cercle englobant

  // la première rosace commence avec un angle de 10°
  for (int n=1; n <= 2; n++) {        // deux rosaces décalées de 30°
    int d = n ==1 ? 10 : 40;
    for (int i=0; i <= 5; i++) {
      g.drawArc (
        (int)(x-gR+gR*Math.cos(2*Math.PI*(d+i*60.)/360.)),
        (int)(y-gR+gR*Math.sin(2*Math.PI*(d+i*60.)/360.)),
        2*gR, 2*gR, 120-i*60-d, 120);
    }
  }
}

// modifie l'argument et le module du point commun rosaces2
void setRosace2 (int angle, double module) {
  this.angle  = angle;
  this.module = module;
}

// les centres des 8 cercles sont régulièrement
// espacés sur un cercle de rayon gR et de centre (x,y);
// les 8 cercles passent par un point commun P
// défini par son module et son argument par rapport
// au cercle (x, y, gR)
void rosace2 (Graphics g, int x, int y, int gR) {
  // (xP,yP) : le point commun P défini par angle et module
  int xP = (int) (x + module*gR*Math.cos (2*Math.PI*angle/360.));
```

```
    int yP = (int) (y - module*gR*Math.sin (2*Math.PI*angle/360.));
    // les 8 cercles centrés sur le cercle de rayon gR centré en (x,y)
    for (int i=0; i <= 7; i++) {
      // (rX, rY) : le centre du i° cercle
      int rX = (int)(x+gR*Math.cos (2*Math.PI*i*45./360.));
      int rY = (int)(y+gR*Math.sin (2*Math.PI*i*45./360.));
      // le rayon du cercle de centre (rX,rY) passant par le point P
      int rC = (int) Math.sqrt ((rX-xP)*(rX-xP) + (rY-yP)*(rY-yP));
      cercle (g, rX, rY, rC);
    }
    // pour dessiner en bleu le cercle des centres des cercles
    //g.setColor (Color.blue);
    //cercle (g, x, y, gR);
  }  // rosace2
```

Les huit cercles dessinés par rosace2() sont centrés sur un cercle de centre (x,y) et de rayon gR. Ils passent par un point commun P défini par son module et son argument par rapport au cercle (x, y, gR). Normalement, ce cercle est invisible. Le point commun P peut être changé de place pour obtenir des dessins différents. Ce point commun peut se trouver à l'intérieur ou à l'extérieur du cercle.

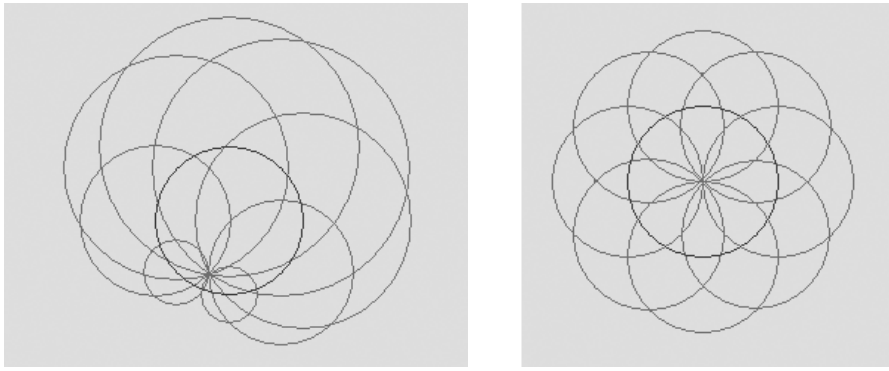


Figure 5.25 — Le cercle bleu sur lequel sont centrés les autres cercles qui passent également par un point commun.

Sur la figure de droite, le point commun est le centre du cercle bleu. Normalement, le cercle bleu est invisible.

La méthode paint() est appelée lorsqu'il faut repeindre le dessin. Elle repeint d'abord le fond et entoure le dessin d'un cadre. Elle dessine alors une figure suivant le numéro numFig de la figure à dessiner.

```
public void paint (Graphics g) {
  Dimension gR = getSize();                  // de la classe Component
  g.setColor (couleurF);                     // voir Graphics page 137

  g.fillRect (0, 0, gR.width-1, gR.height-1);     // dessiner le fond
  g.setColor (couleur);
```

```
        g.drawRect(0, 0, gR.width-1, gR.height-1);       // tracer le cadre
        int xCentre = gR.width/2;
        int yCentre = gR.height/2;    // le centre de l'espace du composant
        int rayon   = (int) (gR.height/2.5);

        switch (numFig) {
        case 0 :
          unCercle (g, xCentre, yCentre, rayon);
          break;
        case 1 :
          deuxCercles (g, xCentre, yCentre, rayon);
          break;
        case 2 :
          troisCercles (g, xCentre, yCentre, rayon);
          break;
        case 3 :
          quatreCercles1 (g, xCentre, yCentre, rayon);
          break;
        case 4 :
          quatreCercles2 (g, xCentre, yCentre, rayon);
          break;
        case 5 :
          cinqCercles (g, xCentre, yCentre, rayon);
          break;
        case 6 :
          rosace1 (g, xCentre, yCentre, rayon);
          break;
        case 7 :
          rayon = (int) (rayon/2.5);  // sinon trop grand
          // on décale le centre du cercle vers le point commun
          xCentre += (int) rayon*module*Math.cos(2*Math.PI*angle/360.);
          yCentre -= (int) rayon*module*Math.sin(2*Math.PI*angle/360.);
          rosace2 (g, xCentre, yCentre, rayon);
          break;
        }
      }  // main

    }  // FigGeo
```

Exercice 5.2 – Réutilisation du composant FigGeo

> En utilisant la classe FigGeo définie ci-dessus, écrire une classe PPFigGeo qui dessine les huit figures que peut afficher cette classe FigGeo. Les huit figures sont présentées par colonne de trois figures comme indiqué sur la figure 5.26. La neuvième figure est une variante de la figure huit avec un point commun pour les cercles défini avec setRosace2 (45, 0.9) (angle = 45°, module = 0.9).

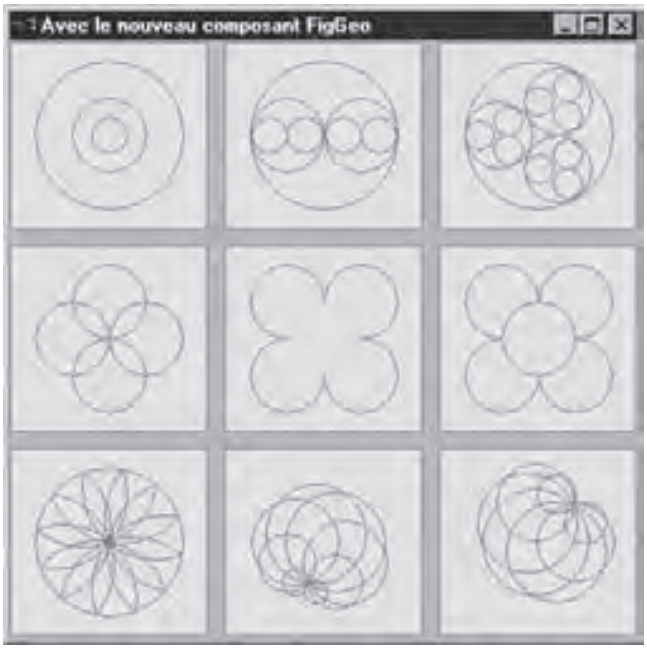


Figure 5.26 — Les huit figures de la classe FigGeo, plus une variante du huitième dessin.

5.6 LE JEU DU PENDU

5.6.1 L'interface graphique

Le jeu du pendu consiste à découvrir un mot en proposant un caractère à chaque fois. Si le caractère proposé figure dans le mot à découvrir, il est affiché à sa place. Si le caractère apparaît plusieurs fois dans le mot, toutes les occurrences du caractère dans le mot sont affichées. Si le caractère proposé ne figure pas dans le mot, il y a échec, et une potence se construit étape par étape au fil des échecs. On a le droit à huit échecs au plus, sinon on est pendu. L'interface graphique est proposée sur la figure 5.27. On y trouve deux Label, deux TextField, une zone de dessin (Potence) et un Button. Le mot à trouver doit être **non éditable** par l'utilisateur.

5.6.2 La classe Potence

La potence peut être considérée comme un objet à part avec ses caractéristiques bien particulières. On y trouve l'état dans lequel se trouve la potence (étape de la construction) : il y a neuf états (de 0 à 8). On doit aussi savoir si on doit afficher le message "trouvé" ou le nombre de coups restants. La potence est caractérisée par deux attributs privés : `etat` et `trouve`. La classe Potence hérite de Component et à ce titre a une taille (fournie par le programmeur ou le gestionnaire de mise en page) et une couleur de texte et de fond. C'est un **nouveau composant** qui peut être réutilisé dans n'importe quelle application.

Figure 5.27 — Le jeu du pendu.

La classe Java Potence contient des méthodes d'initialisation des attributs `etat` et `trouve`, des accesseurs permettant de connaître ou de modifier un de ses deux attributs, et une méthode permettant à la potence de se dessiner en fonction de ses deux caractéristiques. Afin de faciliter son réemploi, la classe Potence est rangée dans le paquetage mdawt. Les méthodes doivent être `public` (y compris le constructeur) si on veut utiliser cette classe à partir un autre paquetage.

```
// Potence.java                          composant de dessin de la potence

package mdpaquetage.mdawt;           // classe Potence du paquetage mdawt
import java.awt.*;

public class Potence extends Component {
  public static final int maxEtat = 9;  // de 0 à 8 inclus
  private int      etat   = 0;          // état de la construction
  private boolean  trouve = false;      // afficher le message trouvé
  Dimension        di;                  // pour les méthodes l() et h()

  public Potence () {
    etat   = 0;
    trouve = false;
  }

  // réinitaliser l'état de la Potence
  public void init () {
    etat   = 0;
    trouve = false;
    repaint();
  }

  // positionner l'état de la Potence
  public void setEtat (int etat) {
    if (etat >= 0 && etat < maxEtat) this.etat = etat;
    repaint();
  }
```

```
// donner l'état de la Potence
public int getEtat () {
  return etat;
}

// positionner trouve de la Potence à vrai
public void setTrouve () {
  this.trouve = true;
  repaint();
}

// incrémenter l'état de la Potence
public void incEtat () {
  if (etat+1 < maxEtat) {
    etat++;
    repaint();
  }
}

// utile pour FlowLayout et BorderLayout
public Dimension getPreferredSize() {
  return new Dimension (140, 160);                    // voir page 149
}
```

La méthode **paint**() dessine la potence à chaque appel en tenant compte de son état, et du fait que *trouvé* est vrai ou non. La méthode getSize() de Component fournit les dimensions de l'espace attribué à la potence. Le fond est redessiné et un cadre est ajouté autour de cet espace attribué. La potence s'adapte à son espace, en largeur et en hauteur sans respect des dimensions initiales.

```
// mise à l'échelle en largeur de v
int l (int v) {
  double k = Math.min (di.width/140., di.height/160.);
  return (int) (v*k);
  // return v*di.width/140;  // non proportionnelle
}

// mise à l'échelle en hauteur de v
int h (int v) {
  double k = Math.min (di.width/140., di.height/160.);
  return (int) (v*k);
  // return v*di.height/160;  // non proportionnelle
}

// le dessin de la Potence
public void paint (Graphics g) {
  // le dessin s'adapte à l'espace attribué
  di = getSize ();  // de Component
  g.clearRect (0, 0, di.width-1, di.height-1);              // effacer
  g.drawRect  (0, 0, di.width-1, di.height-1);     // tracer le cadre

  // s'adapte à l'espace du composant
  int taille = 12*(di.width/120);
  if (taille < 8) taille = 8;
  g.setFont (new Font ("TimesRoman", Font.PLAIN, taille));
```

```
      if (etat >= 1) g.drawLine (l(30), h(120), l(90), h(120));
      if (etat >= 2) g.drawLine (l(30), h(120), l(30), h( 40));
      if (etat >= 3) g.drawLine (l(60), h(120), l(30), h( 90));
      if (etat >= 4) g.drawLine (l(30), h( 40), l(80), h( 40));
      if (etat >= 5) g.drawLine (l(30), h( 60), l(50), h( 40));
      if (etat >= 6) g.drawLine (l(70), h( 40), l(70), h( 60));
      if (etat >= 7) g.drawOval (l(65), h( 60), l(10), h( 10));  // tête
      if (etat >= 8) {
        g.drawLine (l(70), h(70), l(70), h(85));              // le corps
        g.drawLine (l(70), h(70), l(65), h(75));
        g.drawLine (l(70), h(70), l(75), h(75));
        g.drawLine (l(70), h(85), l(65), h(95));
        g.drawLine (l(70), h(85), l(75), h(95));
      }

      if (trouve) {
        g.drawString ("Bravo, vous avez trouvé", l(10), h(150));
      } else if (etat == 8) {
        g.drawString ("Vous êtes pendu !!! ",    l(10), h(150));
      } else if (etat == 7) {
        g.drawString ("Reste 1 coup à jouer",    l(10), h(150));
      } else {  // etat >= 0 et < 7
        g.drawString ("Reste " + (8-etat) + " coups à jouer",
                                                      l(10), h(150));
      }
    }  // paint

}  // Potence
```

La figure 5.28 donne les différents états de la potence. Il suffit de créer un tableau d'objets de la classe Potence dans ses différents états et de les ajouter à la fenêtre principale de type Frame. La méthode paint() de chaque objet de type Potence est appelée pour dessiner la potence dans l'espace attribué par le gestionnaire de mise en page de type GridLayout (3 éléments par ligne).

```
// PPDessinPotence.java                  Programme Principal de Dessin
//                                              des états de la Potence

import java.awt.*; // Frame
import mdpaquetage.mdawt.*;    // Potence, FermerFenetre

// Programme Principal de Dessin des états de la Potence
class PPDessinPotence extends Frame {                // hérite de Frame

  PPDessinPotence () {
    // les caractéristiques de la fenêtre principale
    setTitle ("Les " + Potence.maxEtat + " états de la potence");
    setBackground (Color.lightGray);  // défaut : blanc
    setBounds (10,10, 400, 550);      // défaut : petite taille

    // mise en page par colonnes de 3; 10 pixels entre composants
    setLayout (new GridLayout (0, 3, 10, 10));

    // un tableau d'objets Potence
    Potence[] dessin = new Potence [Potence.maxEtat];       // de 0 à 8
    for (int i=0; i < dessin.length; i++) {
```

```
        dessin[i] = new Potence();
        dessin[i].setBackground (Color.cyan);
        dessin[i].setForeground (Color.red);
        dessin[i].setEtat (i);
        add (dessin[i]);
      }

      addWindowListener (new FermerFenetre());          // voir page 221
      setVisible (true);                              // défaut : invisible
    } // constructeur PPDessinPotence

    public static void main (String[] arg) {
      new PPDessinPotence();
    } // main

  } // class PPDessinPotence
```

La figure 5.28 donne les neuf états de la Potence.

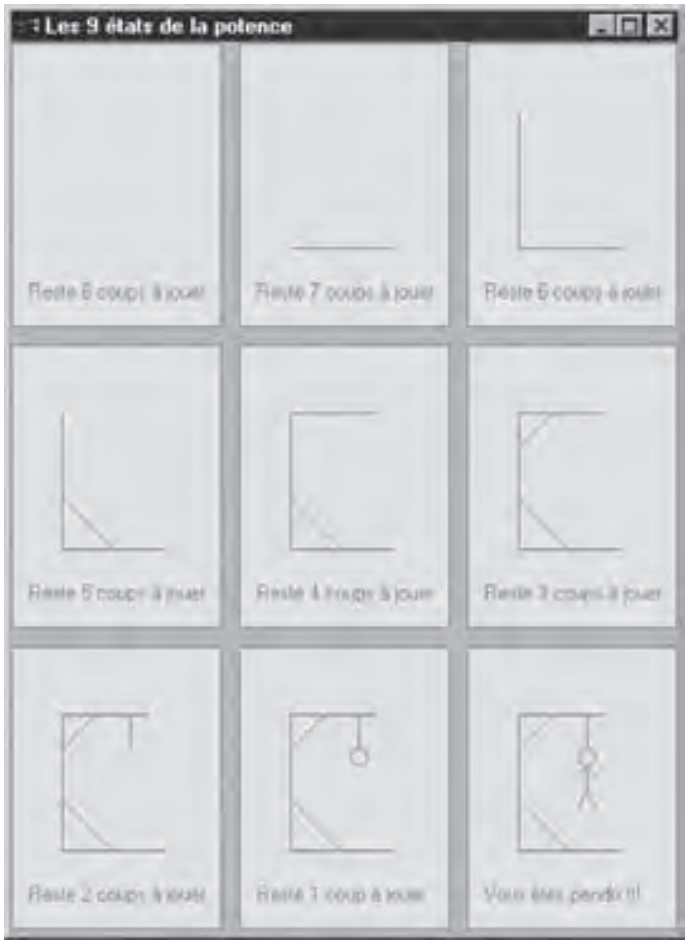


Figure 5.28 — Les neuf états de la potence.

Exercice 5.3 – Programme Java du pendu : utilisation du composant Potence

> Écrire le **composant Pendu** permettant de jouer au pendu en créant l'interface utilisateur de la figure 5.29 et en utilisant le composant Potence.

```
public class Pendu extends Panel {  // composant Pendu
   // cbPendu : color back Pendu; color back Potence et front Potence
   public Pendu (Color cbPendu, Color cbPotence, Color cfPotence);
   public Pendu (); // couleurs par défaut
   // changement du tableau des mots à découvrir;
   // un tableau est fourni par défaut
   public void setTableMots (String[] tabMots);
}
```

Exemple de mise en œuvre du **composant Pendu** :

```
// PPendu.java  jeu du pendu

import java.awt.*;           // Frame
import mdpaquetage.mdawt.*;  // class Pendu, FermerFenetre

class PPendu extends Frame {

  PPendu () {
    setTitle     ("Jeu du pendu");
    setBounds    (10, 10, 340, 450);
    // setResizable (false);
    addWindowListener (new FermerFenetre());          // voir page 221
    add (new Pendu(), "Center");
    setVisible (true);
  }

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

}  // PPendu
```

Exemples de jeux avec mot trouvé ou non :

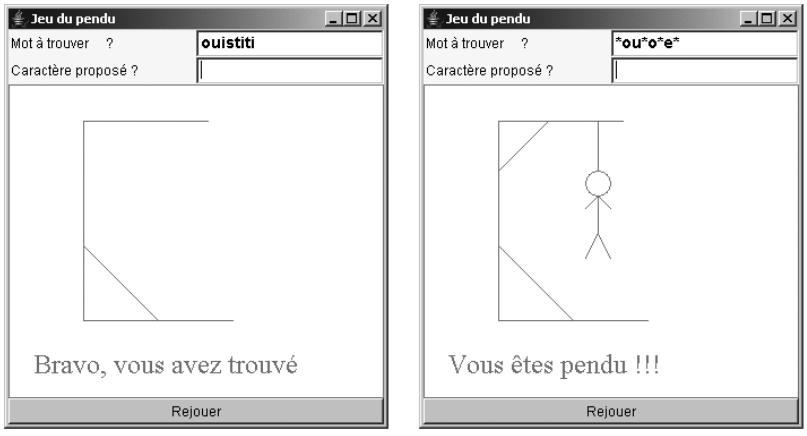


Figure 5.29 — Exemples d'interface utilisateur pour le jeu du pendu.

5.7 LE COMPOSANT BALLE

5.7.1 La classe Balle

Une balle est un objet graphique caractérisé par une couleur, un diamètre et un booléen indiquant si la balle doit être centrée automatiquement dans l'espace qui lui est attribué par le programmeur ou le gestionnaire de mise en page (LayoutManager) ou être centrée en un point (x,y).

Outre des constructeurs initialisant les caractéristiques de Balle, et des accesseurs fournissant ou modifiant certaines de ses caractéristiques, on peut définir plusieurs méthodes spécifiques de la classe **Balle**.

Méthodes de la classe Balle :

- public String **toString** () : fournit la chaîne des caractéristiques de Balle.
- public Dimension **getPreferredSize**() : la taille préférée (FlowLayout, BorderLayout).
- public void **paint** (Graphics g) : repeint la balle. Si la balle doit être centrée, on calcule les coordonnées de son centre par rapport à l'espace qui lui a été attribué. Si la balle ne doit pas être centrée, le point (x,y) reste le centre de la balle.

Le programme Java est donné ci-après. Balle dérive de **Component** (voir figure 5.12, page 157). La classe Balle est ajoutée au paquetage mdawt de façon qu'elle puisse être utilisée dans différentes applications.

```
// Balle.java                          composant d'affichage d'une balle

package mdpaquetage.mdawt;  // Balle est rangée dans le paquetage mdawt

import java.awt.*;                                          // Graphics

public class Balle extends Canvas {
  // Les attributs de l'objet Balle
  protected Color clrBalle;   // couleur de la Balle
  protected int   diametre;   // diamètre de la balle
  protected int   x;          // position en x du centre de la balle
  protected int   y;          // position en y du centre de la balle
  // auCentre : true  : la Balle est centrée dans son espace
  // auCentre : false : la Balle est centrée en (x,y)
  protected boolean auCentre;
```

Les différents constructeurs suivants initialisent les attributs de Balle. Le premier et deuxième constructeurs ont des paramètres initialisant tous les attributs d'une Balle centrée en (x,y) ou au Point centre.

Le troisième constructeur initialise une Balle centrée automatiquement dans son espace.

Le dernier constructeur positionne aléatoirement le centre d'une balle à l'intérieur d'un rectangle de Dimension d.

```
// création d'un Balle centrée au point (x,y)
  public Balle (Color clrBalle, int diametre, int x, int y) {
    this.clrBalle = clrBalle;
```

```
    this.diametre = diametre;
    this.x        = x;
    this.y        = y;
    this.auCentre = false;
  }

  // création d'un Balle centrée au Point centre
  public Balle (Color clrBalle, int diametre, Point centre) {
    this (clrBalle, diametre, centre.x, centre.y);
  }

  // création d'un Balle centrée dans son espace
  public Balle (Color clrBalle, int diametre) {
    this (clrBalle, diametre, 0, 0);
    this.auCentre = true;
  }

  // création d'un Balle de centre aléatoire
  // dans le rectangle de Dimension d
  public Balle (Color clrBalle, int diametre, Dimension d) {
    this (clrBalle, diametre,
         (int) (Math.random() * (d.width-diametre)  + diametre/2),
         (int) (Math.random() * (d.height-diametre) + diametre/2));
  }
```

Les accesseurs fournissent ou modifient un ou plusieurs attributs de Balle. La méthode toString() fournit une chaîne de caractères contenant les caractéristiques de l'objet Balle en cours de traitement.

```
  // les accesseurs en consultation et en modification
    public Color getClrBalle () { return clrBalle; }
    public int   getDiametre () { return diametre; }
    public int   getX ()        { return x; }
    public int   getY ()        { return y; }

    // modifier la couleur et le diamètre de la Balle
    public void setCouleur (Color clrBalle) {
      this.clrBalle = clrBalle;
      repaint();
    }

    // modifier la couleur et le diamètre de la Balle
    public void setDiametre (int diametre) {
      this.diametre = diametre;
      repaint();
    }

    // modifier le centre (x,y) de la Balle
    public void setXY (int x, int y) {
      this.x = x;
      this.y = y;
      repaint();
    }
```

```
// écrire les caractéristiques de la Balle
public String toString () {
  return ( "Balle " + clrBalle + " de diamètre " + diametre
           + " en (" + x + "," + y + "), auCentre = " + auCentre);
}

// utile pour FlowLayout et BorderLayout
public Dimension getPreferredSize() {                  // voir page 149
  return new Dimension (50, 50);
}
```

La méthode **paint**() redessine la balle quant c'est nécessaire. Si la balle est ajoutée à un conteneur utilisant un gestionnaire de mise en page, on lui attribue automatiquement un espace dans le conteneur. Si la balle est centrée, elle se redessine au milieu de cet espace sans tenir compte des coordonnées de son centre. La méthode paint() est appelée automatiquement quand la Balle est ajoutée au conteneur (méthode add()). Si la balle n'est pas centrée dan son espace, elle se redessine centrée en (x,y) de son espace (son contexte graphique), ou du contexte graphique du conteneur si on appelle explicitement paint() de Balle avec le contexte graphique d'un conteneur. La balle a un reflet tracé avec deux arcs de couleur blanche.

```
// dessiner la Balle dans le contexte graphique g
// Balle centrée : au milieu de l'espace attribué à la Balle
// Balle non centrée : (x,y) coordonnées du centre de la balle
public void paint (Graphics g) {
  // calculer les coordonnées de (x,y)
  int x, y;                                        // variables locales
  // di : espace alloué pour la balle ajoutée au Container (add).
  // si on appelle directement paint() d'un autre contexte
  // graphique, hauteur et largeur valent 0.
  // cas de PP2Balle.java
  Dimension di = getSize();          // de l'espace alloué à cet objet
  int hauteur = di.height;
  int largeur = di.width;
  g.clearRect (0, 0, largeur, hauteur);            // couleur du fond
  if (auCentre) {
    x = largeur/2;              // x et y au milieu de l'espace alloué
    y = hauteur/2;
  } else {
    x = this.x;                      // (x,y) au point (this.x,this.y)
    y = this.y;
  }

  // la balle
  int r = diametre/2;                                          // rayon
  g.setColor (clrBalle);
  g.fillOval (x-r, y-r, diametre, diametre);

  // le reflet de la balle
  g.setColor (Color.white);
  int r3 = r/3;                                          // 1/3 du rayon
```

```
    // un arc démarrant à 90° sur 100° dans le sens trigonométrique
    g.drawArc (x-2*r3, y-2*r3, 2*r3, 2*r3, 90, 100);
    g.drawArc (x-r/2,  y-r/2,    r3,   r3, 90, 100);
  }  // paint

}  // classe Balle
```

5.7.2 La mise en œuvre du composant Balle avec un gestionnaire de mise en page

Le gestionnaire de mise en page attribue un espace à chaque composant. Le gestionnaire utilisé ci-après répartit les composants par colonne de trois éléments. Les trois premières balles sont centrées dans leur espace : les coordonnées du centre sont automatiquement recalculées. Les trois dernières sont positionnées aux coordonnées (x, y) définies lors de leur création. Les coordonnées sont relatives à l'espace du contexte graphique (voir figure 5.30). La couleur du fond diffère d'un élément à l'autre pour mettre en évidence l'espace attribué à chaque composant. Les balles sont ajoutées (**add**()) au conteneur, le gestionnaire de mise en page s'occupe d'appeler la méthode paint() quand c'est nécessaire.

```
// PPBalle.java  Programme Principal de test de la classe Balle

import java.awt.*;           // Color
import mdpaquetage.mdawt.*;  // Balle, FermerFenetre

class PPBalle extends Frame {

  Balle[] tabBalles = {  // tableau de Balle
    // balles centrées dans leur espace
    new Balle (Color.black, 10),          // sa couleur et son diamètre
    new Balle (Color.red,   20),
    new Balle (Color.blue,  60),
    // balles centrées en (x,y) de leur espace
    new Balle (Color.black, 10, new Point (20, 20)),
    new Balle (Color.red,   20, new Point (50, 80)),
    new Balle (Color.blue,  30, new Point (60, 100))
  };

  PPBalle() {
    setTitle  ("test de Balle");
    setBounds (20, 20, 300, 300);
    setLayout (new GridLayout (0, 3));        // 3 composants par ligne

    for (int i=0; i < tabBalles.length; i++) {
      // on alterne la couleur du fond
      tabBalles[i].setBackground (i % 2 == 0 ?
                                        Color.yellow : Color.cyan);
      add (tabBalles[i]);
    }
    addWindowListener (new FermerFenetre());          // voir page 221
    setVisible (true);
```

```
  } // constructeur PPBalle

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

}  // class PPBalle
```

Les trois balles du haut sont automatiquement centrées si on change la taille de la fenêtre. Les trois balles du bas restent à leur position par rapport à leur espace soit (20, 20), (50, 80), (60, 100).

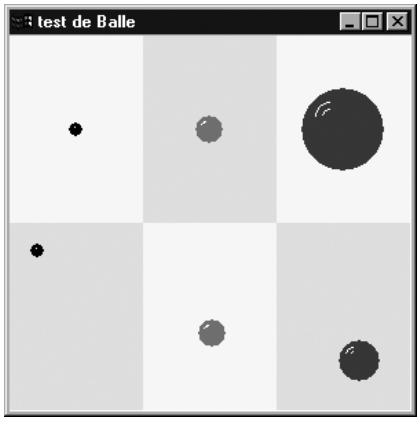


Figure 5.30 — Mise en page de composants de la classe Balle.
Ligne du haut, les balles sont centrées dans leur espace. Ligne du bas, le centre de chaque balle est imposé. Les balles ont un reflet blanc formé de deux arcs.

5.7.3 La mise en œuvre du composant Balle dans le contexte graphique du conteneur

Si la balle se déplace dans l'espace d'un conteneur, on peut la dessiner dans ce conteneur sans l'ajouter à ce conteneur (pas de add() de la Balle) mais en appelant directement la méthode paint() de Balle avec le contexte graphique du conteneur.

La méthode paint (Graphics g) ci-dessous fait référence au contexte graphique du conteneur PP2Balle de type Frame. La balle n'est pas ajoutée au conteneur ; la méthode paint() est appelée explicitement pour chacune des balles. L'instruction : `tabBalles[i].paint(g) ;` se déroule dans le contexte graphique g de la fenêtre principale de type Frame de PP2Balle.

```
// PP2Balle.java    Programme Principal n° 2 de test de la classe Balle

import java.awt.*;           // Color
import mdpaquetage.mdawt.*;  // Balle, FermerFenetre

class PP2Balle extends Frame {

  Balle[] tabBalles = {       // pas de centrage automatique des balles
    new Balle (Color.black, 10, new Point ( 20, 200)),
    new Balle (Color.red,   20, new Point (150, 150)),
```

```
    new Balle (Color.blue,  30, new Point (250, 270)),
    new Balle (Color.black, 10, new Point (200, 50)),
    new Balle (Color.red,   20, new Point ( 50, 80)),
    new Balle (Color.blue,  30, new Point (180, 100))
  };

  PP2Balle() {
    // caractéristiques de la fenêtre principale
    setTitle ("2° test de Balle");
    setBounds (20, 20, 300, 300);
    addWindowListener (new FermerFenetre());          // voir page 221
    setVisible (true);
  }

  // g est le contexte graphique de PP2Balle soit de type Frame
  // les balles ne sont pas ajoutées au conteneur
  public void paint (Graphics g) {
    for (int i=0; i < tabBalles.length; i++) {
      tabBalles[i].paint (g);
    }
  }

  public static void main (String[] args) {
    PP2Balle p = new PP2Balle();
    // sortie standard pour contrôle
    for (int i=0; i < p.tabBalles.length; i++) {
      System.out.println (p.tabBalles[i]);
    }
  }  // main

}  // class PP2Balle
```

Sur la figure 5.31, les balles sont placées à leur position par rapport à la fenêtre principale (par rapport au contexte graphique de Frame). Les deux balles (bleues) les plus grosses sont en (180, 100) et en (250, 270).

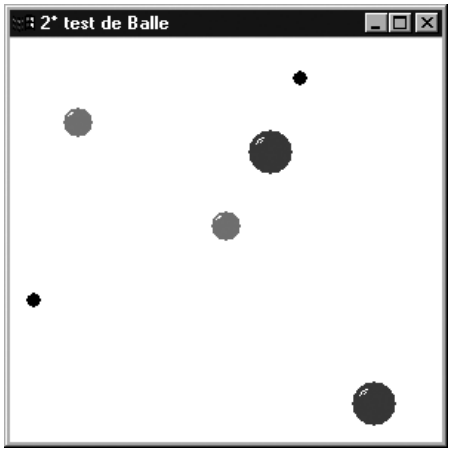


Figure 5.31 — Positionnement de composants dans un contexte graphique.

Les résultats imprimés sur la sortie standard sont donnés ci-dessous :

```
Balle java.awt.Color[r = 0,g = 0,b = 0]   de diamètre 10 en (20,200),  auCentre = false
Balle java.awt.Color[r = 255,g = 0,b = 0] de diamètre 20 en (150,150), auCentre = false
Balle java.awt.Color[r = 0,g = 0,b = 255] de diamètre 30 en (250,270), auCentre = false
Balle java.awt.Color[r = 0,g = 0,b = 0]   de diamètre 10 en (200,50),  auCentre = false
Balle java.awt.Color[r = 255,g = 0,b = 0] de diamètre 20 en (50,80),   auCentre = false
Balle java.awt.Color[r = 0,g = 0,b = 255] de diamètre 30 en (180,100), auCentre = false
```

Exercice 5.4 Balle mobile dans un rectangle

On veut déplacer une balle dans un rectangle donné. La balle doit rebondir sur les côtés du rectangle et ne jamais sortir de ce cadre. Une balle mobile a en plus d'une balle normale, une vitesse en x et une vitesse en y. On utilise le concept **d'héritage** pour définir une BalleMobile à partir d'une Balle.

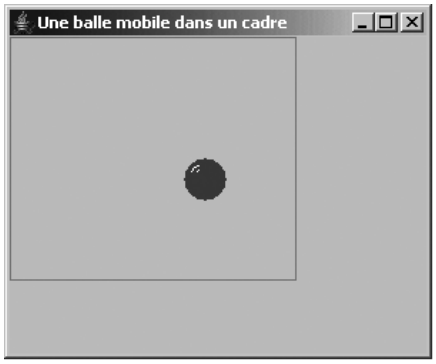


Figure 5.32 — Une balle mobile dans un rectangle (classe BalleMobile)

Écrire la classe BalleMobile suivante :

```
public class BalleMobile extends Balle {
  private int   vitx;       // vitesse suivant l'axe des x
  private int   vity;       // vitesse suivant l'axe des y

  // la Balle est mobile, centrée en (x,y)
  // de vitesse vitx (axe des x) et vity (axe des y)
  public BalleMobile (Color clrBalle, int diametre, int x, int y,
                                                int vitx, int vity);

  // création d'un BalleMobile de centre et vitesses aléatoires
  // dans le rectangle de Dimension d
  public BalleMobile (Color clrBalle, int diametre, Dimension d);
```

les accesseurs pour les vitesses en x et en y (à définir)

```
  // déplacer la balle dans un cadre largeur x hauteur,
  // sans sortir du cadre;
  // recalculer x et y en tenant compte de la vitesse
```

```
    // et du diamètre de la balle
    public void deplacer (int largeur, int hauteur);

}  // BalleMobile
```

Écrire un programme PPBalleMobile permettant de faire évoluer un balle mobile dans le contexte graphique d'un Panel.

5.8 LES INTERFACES MOUSELISTENER ET MOUSEMOTIONLISTENER

En Java, pour qu'un composant réalise une action suite à un événement le concernant, il faut le demander explicitement à l'aide d'un écouteur (Listener). Les boutons ont un **ActionListener** (voir page 163) indiquant ce qu'il faut faire quand ils sont "actionnés". Certains composants ont un **ItemListener** (voir page 176) écoutant chaque élément (item) du composant. Une fenêtre a un **WindowListener** prenant en compte les différents événements déclenchés sur cette fenêtre (voir page 221). La prise en compte des événements de la souris se fait de manière similaire grâce à deux écouteurs (les interfaces **MouseListener** et **MouseMotionListener**).

MouseListener prend en compte l'appui ou le relâchement du bouton de la souris dans l'espace du composant, l'entrée ou la sortie du curseur de la souris de cet espace, ou un clic sur le composant. A noter qu'un clic génère aussi les événements appui et relâchement. Cinq méthodes de l'interface MouseListener traitent ces 5 événements :

- void **mouseClicked** (MouseEvent evt) : clic sur l'espace du composant.
- void **mouseEntered** (MouseEvent evt) : le curseur entre dans l'espace du composant.
- void **mouseExited** (MouseEvent evt) : le curseur sort de l'espace du composant.
- void **mousePressed** (MouseEvent evt) : bouton de la souris appuyé sur le composant.
- void **mouseReleased** (MouseEvent evt) : bouton de la souris relâché sur le composant.

MouseMotionListener prend en compte les déplacements de la souris quand le curseur se trouve dans l'espace du composant ; il gère deux événements : le déplacement du curseur sans appui sur le bouton, et le déplacement avec bouton de la souris constamment appuyé. Deux méthodes de l'interface MouseMotionListener traitent ces 2 événements :

- void **mouseMoved** (MouseEvent evt) : survol du composant, bouton non appuyé.
- void **mouseDragged** (MouseEvent evt) : survol du composant, bouton appuyé.

La mise en œuvre de ces interfaces (voir Interface page 106) nécessite la définition de toutes les méthodes, même celles qui ne sont pas utilisées dans l'application. On peut bien sûr redéfinir comme vides les méthodes non utilisées. C'est ce que font les adaptateurs (**Adapter**) qui implémentent comme vides les méthodes de l'interface. Il suffit **d'hériter d'un Adapter** pour avoir d'office toutes les méthodes vides. Il reste alors à définir uniquement celles qui sont utiles pour l'application.

Tous les composants héritant de Component peuvent se mettre à l'écoute des deux sortes d'événements de la souris. Il faut le demander à l'aide des méthodes suivantes de Component (voir figure 5.3, page 139) qui enregistrent un écouteur pour un composant donné.

- void **addMouseListener** (MouseListener *l*);
- void **addMouseMotionListener** (MouseMotionListener *l*).

La figure 5.33 présente des composants standard AWT (Label, Button) et des composants créés dans les chapitres précédents (Motif, FigGeo, Potence et Balle). Tous dérivent de Component et peuvent réagir aux événements de la souris si on le leur demande. Ils sont créés par les instructions Java suivantes :

```
Label   etiq    = new Label  ("étiquette");
Button  bouton  = new Button ("bouton");
Motif   oiseau  = new Motif  (MotifLib.oiseau, Motif.tProp,
                                                MotifLib.palette1Oiseau);
Potence potence = new Potence();
Balle   balle   = new Balle  (Color.blue, 20);
FigGeo  dessin  = new FigGeo (6);
```

Tous les composants de la figure 5.33 peuvent être déplacés sur l'écran grâce à la demande faite avec **addMouseMotionListener** () et à la programmation de la méthode void **mouseDragged** (MouseEvent evt) de la classe SourisMvt. Le même objet (ecouteSouris) est utilisé pour les six composants car l'effet de la souris est le même dans tous les cas. Pour déplacer un composant, il suffit de maintenir la souris appuyée dans le coin supérieur gauche du composant et de déplacer l'objet.

```
MouseMotionListener ecouteSouris = new SourisMvt();
etiq.addMouseMotionListener    (ecouteSouris);  // Label
bouton.addMouseMotionListener  (ecouteSouris);  // Button
oiseau.addMouseMotionListener  (ecouteSouris);  // Motif
potence.addMouseMotionListener (ecouteSouris);  // Potence
balle.addMouseMotionListener   (ecouteSouris);  // Balle
dessin.addMouseMotionListener  (ecouteSouris);  // FigGeo
```

Les composants de type Motif, FigGeo, Potence et Balle ont des comportements très différents suite à un clic de la souris dans leur espace. On souhaite par exemple qu'un clic sur le Motif oiseau fasse pivoter l'oiseau, un clic sur un objet FigGeo affiche la figure suivante, un clic sur la Potence la fasse passer à l'étape de construction suivante, et un clic sur la Balle la fasse se gonfler jusqu'à un maximum, puis se dégonfler. L'étiquette est insensible au clic, et le bouton est par nature même (dans les méthodes de Button) sensible au clic ; sur la figure, son comportement n'est pas modifié mais il n'effectue aucune action ; il change d'aspect. On enregistre un écouteur MouseListener pour oiseau, potence, balle et dessin et on redéfinit la méthode void **mouseClicked** (MouseEvent evt) ; pour les classes SourisOiseau, SourisPotence, SourisBalle et SourisDessin.

```
oiseau.addMouseListener  (new SourisOiseau());  // Motif
potence.addMouseListener (new SourisPotence()); // Potence
balle.addMouseListener   (new SourisBalle());   // Balle
dessin.addMouseListener  (new SourisDessin());  // FigGeo
```

On souhaite de plus que la couleur du fond de la fenêtre englobante de type Frame change suivant que le curseur se trouve à l'intérieur ou à l'extérieur de la fenêtre. Pour cela, il faut enregistrer un écouteur de type MouseListener, et définir les méthodes void **mouseEntered** (MouseEvent evt) et void **mouseExited** (MouseEvent evt) de la classe SourisFrame.

```
addMouseListener (new SourisFrame());     // pour l'objet courant Frame
```

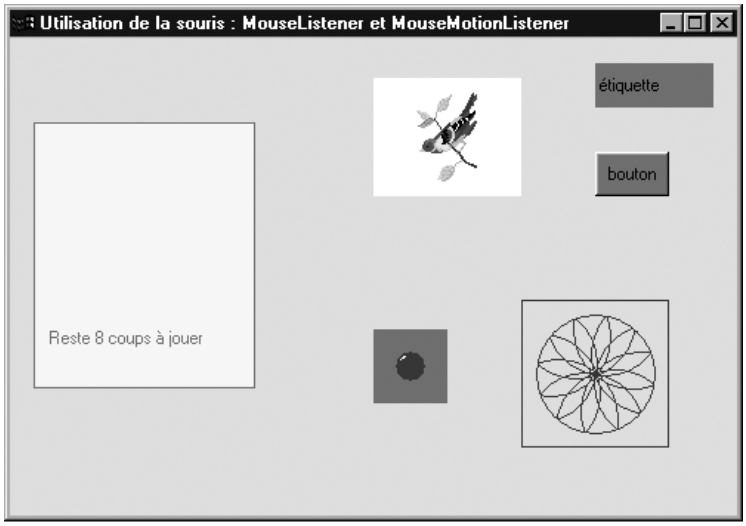


Figure 5.33 — Utilisation de la souris : MouseListener et MouseMotionListener.
Tous les composants peuvent être déplacés avec la souris. Ils peuvent aussi réagir à un clic de la souris : l'oiseau se retourne ; la balle se gonfle et se dégonfle dans son espace ; la potence se construit à chaque clic ; le dessin (figure géométrique) change. L'étiquette est insensible aux clics de la souris. Le bouton est dessiné différemment s'il est appuyé ou sélectionné. La couleur du fond de la fenêtre change suivant que la souris est à l'intérieur de la fenêtre ou non.

Le programme Java permettant aux composants de réagir aux événements provoqués par la souris est donné ci-dessous. La mise en page est faite **sans gestionnaire** de mise en page. Les positions initiales des composants sont déterminées par le programme et modifiées à l'aide de la souris.

```
// PPSouris.java

import java.awt.*;           // Frame
import java.awt.event.*;     // MouseListener

import mdpaquetage.mdawt.*;  // Balle, FigGeo, Motif, Potence

public class Souris extends Frame {
  Label   etiq    = new Label  ("étiquette");
  Button  bouton  = new Button ("bouton");
  Motif   oiseau  = new Motif  (MotifLib.oiseau, Motif.tProp,
                                          MotifLib.palette1Oiseau);
```

```
Potence potence = new Potence();
Balle   balle   = new Balle  (Color.blue, 20);
FigGeo  dessin  = new FigGeo (6, Color.blue, Color.cyan);

Souris () {
  // caractéristiques de la fenêtre principale
  setTitle  ("Utilisation de la souris : MouseListener " +
                                          "et MouseMotionListener");
  setLayout (null) ;           // pas de gestionnaire de mise en page
  setBounds (100, 100, 500, 350);
  setBackground (Color.cyan);

  etiq.setBounds (400, 40, 80, 30);
  etiq.setBackground (Color.red);
  add (etiq);

  bouton.setBounds (400, 100, 50, 30);
  bouton.setBackground (Color.red);
  add (bouton);

  dessin.setBounds (350, 200, 100, 100);
  dessin.setBackground (Color.red);
  add (dessin);

  oiseau.setBounds (250, 50, 100, 80);
  oiseau.setBackground(Color.white);
  add (oiseau);

  potence.setBounds     (20, 80, 150, 180);
  potence.setBackground (Color.yellow);
  potence.setForeground (Color.red);
  add (potence);

  balle.setBounds (250, 220, 100, 100);
  balle.setBackground (Color.red);
  add (balle);

  oiseau.addMouseListener  (new SourisOiseau());
  potence.addMouseListener (new SourisPotence());
  balle.addMouseListener   (new SourisBalle());
  dessin.addMouseListener  (new SourisDessin());
  addMouseListener         (new SourisFrame());

  MouseMotionListener ecouteSouris = new SourisMvt();
  etiq.addMouseMotionListener    (ecouteSouris) ;
  bouton.addMouseMotionListener  (ecouteSouris) ;
  oiseau.addMouseMotionListener  (ecouteSouris) ;
  potence.addMouseMotionListener (ecouteSouris) ;
  balle.addMouseMotionListener   (ecouteSouris) ;
  dessin.addMouseMotionListener  (ecouteSouris) ;

  addWindowListener (new FermerFenetre());          // voir page 221
  setVisible (true);
}  // constructeur Souris
```

```
// classes INTERNES à la classe Souris                    voir page 91

// écoute les événements "souris" sur la fenêtre Frame;
// change la couleur du fond quand on sort ou entre du Frame.
class SourisFrame extends MouseAdapter {     // Adapter voir page 200

  public void mouseEntered (MouseEvent evt) {
    setBackground (Color.cyan);
  }

  public void mouseExited (MouseEvent evt) {
    setBackground (Color.lightGray);
  }

}  // SourisFrame

// l'oiseau change de position à chaque clic
class SourisOiseau extends MouseAdapter {          // Adapter voir 200
  public void mouseClicked (MouseEvent evt) {
    oiseau.setInverser();                            // voir page 158
  }
}  // SourisOiseau

// la potence change d'état à chaque clic
class SourisPotence extends MouseAdapter {   // Adapter voir page 200
  public void mouseClicked (MouseEvent evt) {
    if (potence.getEtat() == Potence.maxEtat) {
      potence.setEtat(0);                                // voir 5.6.2
    } else {
      potence.incEtat();
    }
  }
}  // SourisPotence

// la balle se gonfle jusqu'à occuper tout son espace
// puis se dégonfle ensuite au fil des clics
class SourisBalle extends MouseAdapter {     // Adapter voir page 200
  int delta = 5;
  public void mouseClicked (MouseEvent evt) {
    int dia = balle.getDiametre();
    int largeur = balle.getSize().width;
    if (dia >= largeur) delta = -5;
    if (dia <= 10) delta = 5;
    balle.setDiametre (dia + delta);
  }
}  // SourisBalle

// chaque clic entraîne un changement de figure géométrique
class SourisDessin extends MouseAdapter {    // Adapter voir page 200
  public void mouseClicked (MouseEvent evt) {
    int n = dessin.getNumFig();  // de 0 à 7
    dessin.setNumFig (n >= 7 ? 0 : n+1);
  }
}  // SourisDessin

// le composant est déplacé avec la souris
```

```
  class SourisMvt extends MouseMotionAdapter { // Adapter voir page 200
    public void mouseDragged (MouseEvent evt) {
      Component composant = (Component) evt.getSource();
      // position du csg du composant
      Point csg = composant.getLocation();
      // position de la souris dans le composant
      // x et y peuvent être négatifs (avec mouseDragged)
      int x = evt.getX();
      int y = evt.getY();
      if ( (Math.abs (x) < 10) && (Math.abs (y) < 10) ) {
        composant.setLocation (csg.x + x, csg.y + y);
      }
    }
  }  // SourisMvt

}  // class Souris

class PPSouris {
  public static void main (String[] args) {
     new Souris();
  }
}
```

L'espace attribué à l'oiseau et à la balle est mis en évidence en le coloriant pour mieux en comprendre le fonctionnement. Il pourrait se confondre avec la couleur du fond du panneau en ne définissant pas setBackground() pour oiseau et balle dans le constructeur Souris(). La figure 5.34 indique l'état des composants après déplacements et clics sur les composants. La potence est en partie construite ; la balle a grossi ; l'oiseau a changé de côté ; la zone de dessin affiche une autre figure. Les composants ont changé de place.

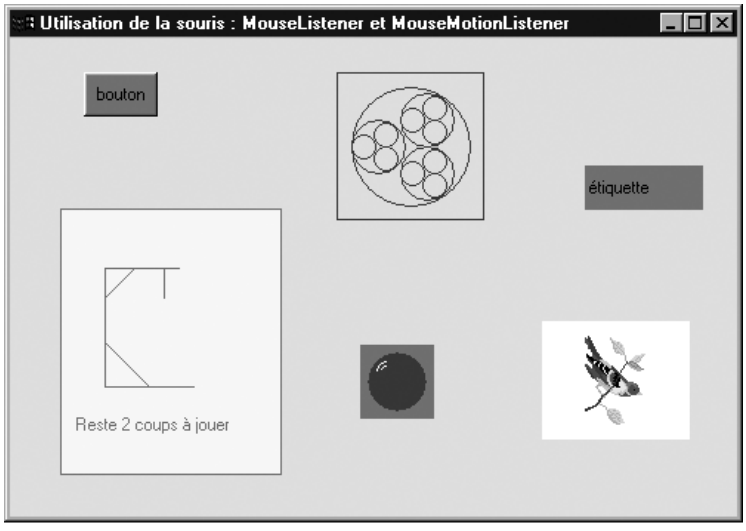


Figure 5.34 — État de l'écran après différents clics et déplacements de composants. À comparer avec la figure 5.33.

5.9 LE JEU DU MASTERMIND (EXEMPLE DE SYNTHÈSE)

5.9.1 La présentation du jeu

Le jeu du MasterMind consiste à découvrir les couleurs d'un nombre n de balles. On dispose de nc couleurs disponibles pour chacune des balles. Le joueur doit faire des propositions de couleurs jusqu'à trouver l'ordre et la couleur des balles.

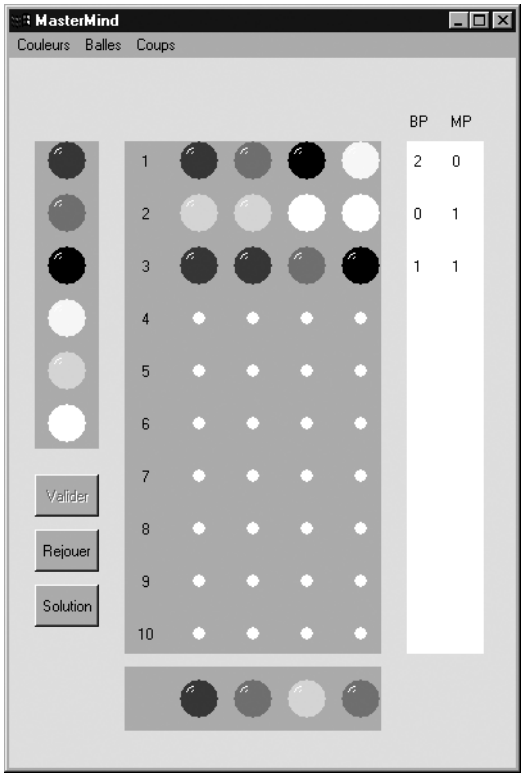


Figure 5.35 — MasterMind avec six couleurs disponibles et quatre couleurs à découvrir en dix coups maximum.

A chaque proposition de n couleurs (une pour chaque balle), l'ordinateur répond en indiquant le nombre BP de couleurs Bien Placées et le nombre de MP de couleurs Mal Placées. Sur la figure 5.35, il y a six couleurs disponibles ; le joueur doit découvrir la couleur des quatre balles et l'ordre des quatre couleurs. La solution étant (bleu, rouge, vert, rouge), la première proposition du joueur (bleu, rouge, noir, jaune) conduit à 2 BP (le bleu et le rouge), et zéro MP. La deuxième proposition (vert, vert, blanc, blanc) conduit à 0 BP et 1 MP (le vert). La troisième proposition (bleu, bleu, rouge, noir) fournit 1 BP (le bleu) et 1 MP (le rouge).

Il y a $6^4 = 1\,296$ combinaisons possibles à découvrir en 10 coups maximum. La complexité du jeu dépend du nombre de couleurs disponibles et du nombre de balles

dont on cherche la couleur. Sur la figure 5.37, il n'y a que deux couleurs disponibles (rouge et bleu) et 3 couleurs à découvrir. Il n'y a que $2^3 = 8$ combinaisons possibles à découvrir en 7 coups.

solution	bleu	rouge	vert	rouge
première proposition	bleu	rouge	noir	jaune
résultats	BP	BP		

5.9.2 Le composant MasterMind

La mise en page est effectuée sans gestionnaire de mise en page pour le panel principal (racine). Celui-ci contient cinq composants Panel : le Panel des couleurs disponibles,

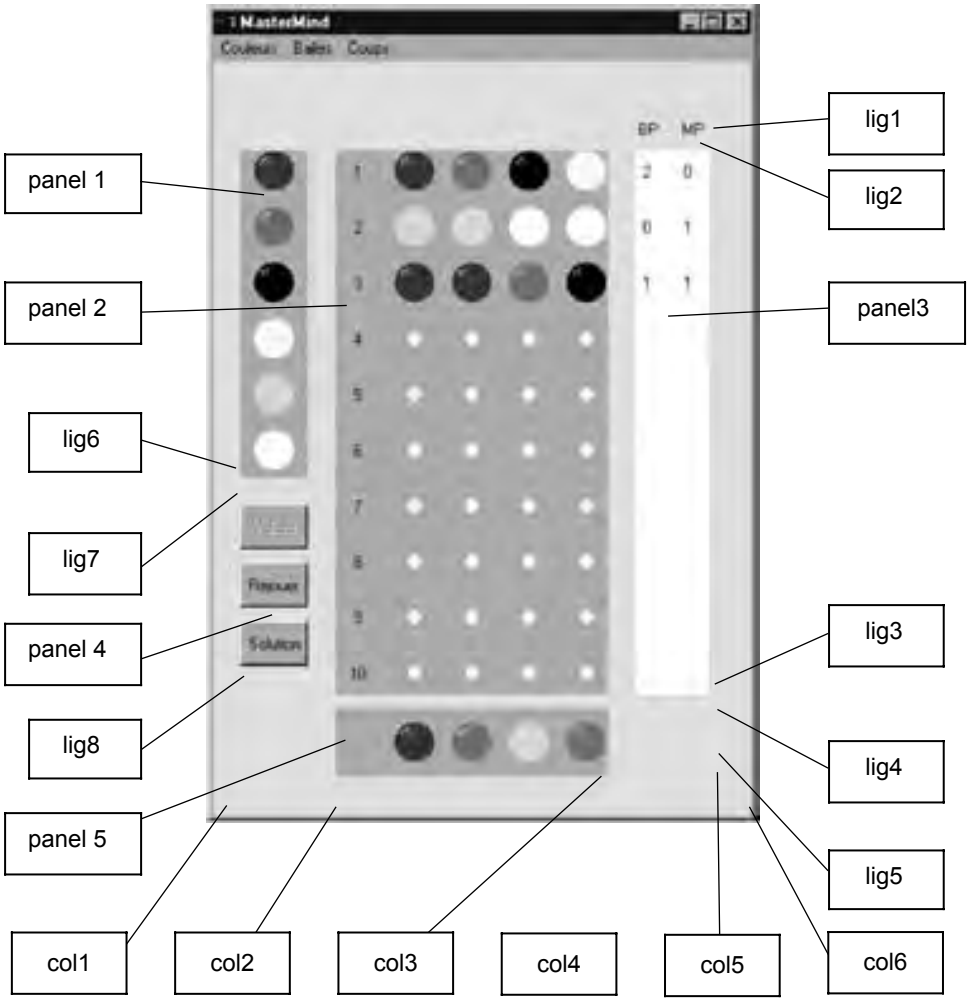


Figure 5.36 — Disposition des panels et des composants.

le Panel des propositions du joueur, le Panel des résultats, le Panel des boutons et le Panel de la solution. Le Panel solution est invisible au départ. Le programme doit indiquer l'emplacement et la taille des panels dans le panel racine. Les panels autres que le panel racine définissent un gestionnaire de type GridLayout() qui range les composants (de type Balle, ou Label pour les résultats) par colonnes.

La figure 5.36 indique la disposition retenue repérée par des numéros de colonnes (col1, col2, etc.) ou des numéros de lignes (lig1, lig2, etc.). Ces valeurs ne sont pas des constantes car le nombre de couleurs disponibles (NB_COUL), le nombre de couleurs à découvrir (NB_BALLES) et le nombre de coups autorisés (MAX_COUPS) peuvent varier d'un jeu à l'autre.

5.9.2.1 Le constructeur de l'interface du composant MasterMind

Le constructeur MasterMind construit l'interface graphique du composant MasterMind en ajoutant les composants (Balle ou Label) aux différents conteneurs de type Panel (voir classe Balle page 193). Les balles sont centrées dans l'espace qui leur est attribué par le gestionnaire du Panel de type GridLayout. Il suffit de les ajouter au Panel pour qu'elles soient dessinées automatiquement.

Les couleurs disponibles sont enregistrées dans un tableau de Balle et ajoutées à panel 1. Les propositions du joueur sont enregistrées dans un tableau de Balle à deux dimensions. Initialement, tous les emplacements des balles sont marqués par une petite balle blanche. Ces petites balles blanches sont ajoutées au panel 2. Le panel 3 est initialisé avec des étiquettes (Label) vides pour chaque résultat possible. Les boutons sont ajoutés au panel 4. La solution à trouver est mémorisée dans un tableau de balles ajoutées au conteneur panel 5 qui est invisible au départ.

Le constructeur doit encore initialiser les structures de données utilisées et enregistrer les écouteurs pour les balles pouvant être actives et pour les boutons.

Le programme Java construisant cette interface dans le constructeur de MasterMind est donné ci-dessous.

```
// MasterMind.java  composant MasterMind

package mdawt;

import java.awt.*;           // Frame
import java.awt.event.*;     // ActionListener
import mdpaquetage.mdawt.*;  // Balle
import mdpaquetage.utile.*;  // aleat                    voir page 366

public class MasterMind extends Panel {

  // Présentation
  private static final int DIA_BALLE = 30;       // diamètre des balles
  private static final int LG_BALLE  = DIA_BALLE + 10;

  private int NB_BALLES; // de 2 à 6
  private int NB_COUL;    // de 2 à 6
  private int MAX_COUPS;  // de 5 à 15
  private int col1;
  private int col2;
```

```
private int col3;
private int col4;
private int col5;
private int col6;

private int lig1;
private int lig2;
private int lig3;
private int lig4;
private int lig5;
private int lig6;
private int lig7;
private int lig8;
private int lig9;

private Button   bValider   = new Button ("Valider");
private Button   bRejouer   = new Button ("Rejouer");
private Button   bSolution  = new Button ("Solution");

private Panel panel1 = new Panel();  // les balles disponibles
private Panel panel2 = new Panel();  // les propositions
private Panel panel3 = new Panel();  // les résultats (bP, mP)
private Panel panel4 = new Panel();  // les boutons
private Panel panel5 = new Panel();  // la solution

// Les données de l'application
// le nombre de balles, de couleurs et de coups est variable
// et défini dans le constructeur

// Les 6 couleurs disponibles
private Color[] tc = {
  Color.blue,   Color.red,   Color.black,
  Color.yellow, Color.green, Color.white
};
private Balle[] ballesDispo;   // NB_COUL balles disponibles
private Balle[][] proposition; // MAX_COUPS propositions au plus
private Balle[] tBalSol;       // les NB_BALLES balles solutions
private Balle balleC;          // la balle courante
private Color[] couleursSol;   // les couleurs de la solution
private Label[] bPetMP;        // Résultats des prop. : BP et MP
private int nbCoup;            // nombre de coups joués

MouseListener ecouteurProp = new Proposition();

// constructeur de l'objet MasterMind
public MasterMind (int NB_COUL, int NB_BALLES, int MAX_COUPS) {
  this.NB_BALLES = NB_BALLES;
  this.NB_COUL   = NB_COUL;
  this.MAX_COUPS = MAX_COUPS;

  // dépendent des 3 variables NB_COUL, NB_BALLES, MAX_COUPS
  col1 = 20;
  col2 = col1+LG_BALLE+30;
  col3 = col2+(NB_BALLES+1)*LG_BALLE;
```

```
col4 = col3+20;
col5 = col4+60;
col6 = col5+20;

lig1 = 35;
lig2 = lig1+30;
lig3 = lig2+MAX_COUPS*(LG_BALLE);
lig4 = lig3+10;
lig5 = lig4+10 + LG_BALLE+20;
lig6 = lig2+NB_COUL*(LG_BALLE);
lig7 = lig6+20;
lig8 = lig7+3*(30+10);
lig9 = lig8+10;

// alloue les tableaux de références      // voir référence page 59
ballesDispo = new Balle [NB_COUL];
proposition = new Balle [MAX_COUPS][NB_BALLES];
bPetMP      = new Label [MAX_COUPS];
tBalSol     = new Balle [NB_BALLES];
couleursSol = new Color [NB_BALLES];
balleC      = ballesDispo [0];                 // la balle courante

// le Panel principal
setLayout     (null);             // defaut : FlowLayout pour Panel
setSize       (col6, Math.max (lig5, lig9));
setBackground (Color.cyan);

// Label bP : bien placé
Label bP = new Label  ("BP      MP");
bP.setBounds (col4, lig1, col5-col4, 30);
add (bP);

// la zone des couleurs disponibles
panel1.setBounds     (col1, lig2, 50, lig6-lig2);
panel1.setBackground (Color.lightGray);
panel1.setLayout     (new GridLayout (0, 1, 10, 10));
add (panel1);

// la zone des propositions
panel2.setBounds     (col2, lig2, col3-col2, lig3-lig2);
panel2.setBackground (Color.lightGray);
panel2.setLayout     (new GridLayout (0, NB_BALLES+1, 10, 10));
add (panel2);

// la zone des résultats BP et MP
panel3.setBounds     (col4, lig2, col5-col4, lig3-lig2);
panel3.setBackground (Color.white);
panel3.setLayout     ((new GridLayout (0, 1, 10, 10)));
add (panel3);

// la zone des boutons
panel4.setBounds     (col1, lig7, 50, lig8-lig7);
panel4.setLayout     (new GridLayout (0, 1,10,10));
add (panel4);
```

```
    // la zone de la solution (non visible au départ)
    panel5.setBounds     (col2, lig4, col3-col2, LG_BALLE+10);
    panel5.setBackground (Color.lightGray);
    panel5.setLayout     (new GridLayout (0, NB_BALLES+1, 10, 10));
    panel5.setVisible    (false);         // défaut est vrai pour Panel
    add (panel5);

    // panel1 : les couleurs disponibles
    for (int i=0; i < ballesDispo.length; i++) {
      ballesDispo [i] = new Balle (tc[i], DIA_BALLE);
      panel1.add (ballesDispo [i]);
    }

    // panel2 : petites balles marquant les emplacements
    for (int i=0; i < proposition.length; i++) {
      //p2g.drawString (Integer.toString(i+1), 10, HL+i*HL);
      panel2.add (new Label (Integer.toString(i+1), Label.CENTER));
      for (int j=0; j < proposition[i].length; j++) {
        proposition [i][j] = new Balle (Color.white, 10);
        panel2.add (proposition [i][j]);
      }
      // panel3 : les résultats (un Label vide)
      bPetMP [i] = new Label ();
      panel3.add (bPetMP [i]);
    }

    // panel4 : les boutons
    bValider.setSize (50, 30);
    panel4.add (bValider);
    bRejouer.setSize (50, 30);
    panel4.add (bRejouer);
    bSolution.setSize (50, 30);
    panel4.add (bSolution);

    // panel5 : tableau des balles pour la solution
    // la solution à afficher (panel5 invisible au début)
    panel5.add (new Label());                // Label vide pour aligner
    for (int i=0; i < tBalSol.length; i++) {
      tBalSol [i] = new Balle (Color.white, DIA_BALLE);
      panel5.add (tBalSol [i]);
    }

    initDonnees();      // des données : 1˚ fois et pour rejouer
    traitement ();      // les différents écouteurs
    //afficherMenu();   // afficher le menu déroulant (en exercice)
    setVisible (true);  // on affiche le Panel principal
  } // constructeur MasterMind

  // valeurs par défaut
  public MasterMind() {
    this (6, 4, 10);
  }
```

```
// initialisation des données : la 1˚ fois et pour rejouer
void initDonnees() {
  nbCoup = 0;

  // génération des couleurs à chercher
  for (int i=0; i < couleursSol.length; i++) {
    couleursSol [i] = tc[Utile.aleat(NB_COUL)]; // de 0 à NB_COUL-1
  }

  // toutes les propositions sont de petites balles blanches;
  // on supprime les écouteurs s'il y en a
  // qui sont restés du jeu précédent.
  for (int i=0; i < proposition.length; i++) {
    for (int j=0; j < proposition[i].length; j++) {
      proposition [i][j].setCoulDia (Color.white, 10);
      proposition [i][j].removeMouseListener (ecouteurProp);
    }
    bPetMP [i] .setText ("");                  // les Label sont vides
  }

  // au début, on écoute les balles de la ligne 0 des propositions
  for (int j=0; j < proposition[0].length; j++) {
    proposition [0][j].addMouseListener (ecouteurProp);
  }

  panel5.setVisible (false);              // la solution est invisible
  // chaque balle de la solution mémorise sa couleur
  for (int i=0; i < tBalSol.length; i++) {
      tBalSol [i].setCoulDia (couleursSol[i], DIA_BALLE);
  }
  bValider.setEnabled (true);

}  // initDonnees
```

5.9.3 Les actions des composants (les écouteurs)

Les instructions précédentes ont permis d'effectuer la mise en page des composants de l'interface graphique. Comme vu dans les paragraphes précédents, les composants effectuent des actions à la demande. Pour cela, il faut les mettre en attente (à l'écoute) d'événements les concernant en définissant des Listener. Les composants qui doivent entraîner une action quand ils sont sollicités sont les suivants :

- les boutons Valider, Rejouer et Solution ;
- les balles (panel1) indiquant les couleurs disponibles ;
- les balles de la ligne de la nouvelle proposition (panel2). Un clic sur une balle d'une autre ligne est sans effet.

Pour les boutons et les couleurs disponibles, les écouteurs sont constamment en place et sont ajoutés dans le constructeur MasterMind() via la méthode traitement(). Les écouteurs de la ligne de proposition varient. Quand on valide une proposition, il faut enlever les écouteurs de la ligne de la proposition, passer à la ligne suivante et y enregistrer des écouteurs pour chaque balle de cette nouvelle ligne.

5.9.3.1 Les écouteurs des événements des boutons du MasterMind

Pour les boutons **Valider**, **Rejouer** et **Solution**, on crée et enregistre un objet respectivement des classes **ActionValider**, **ActionRejouer** et **ActionSolution**. Comme vu page 163, l'action des boutons est décrite par la méthode **actionPerformed**() de l'écouteur.

- actionPerformed() de **ActionValider** compare les couleurs proposées avec les couleurs de la solution et fournit le nombre de couleurs bien placées et le nombre de couleurs mal placées. Ceci se fait dans la méthode valider() qui utilise un objet de type BPEtMP ; on récupère deux résultats ; il faut donc les passer dans un objet (une structure en langage C) ; la fonction valider() retourne faux si la ligne des propositions est incomplète. Si la proposition est acceptable, on enlève les écouteurs de la ligne courante repérée par nbCoup ; on incrémente nbCoup et on enregistre des écouteurs pour chaque balle de cette nouvelle ligne, sauf si on a fini (on a trouvé, ou on a atteint le nombre de coups maximum).
- actionPerformed() de **ActionRejouer** consiste à réinitialiser les structures de données et à réafficher la fenêtre de jeu.
- actionPerformed() de **ActionSolution** consiste à afficher la solution dans son panel, à enlever les écouteurs de la ligne courante : on ne peut plus changer la proposition. On ne peut pas valider une seconde fois (le bouton Valider est inhibé).

```
// TRAITEMENT DES ACTIONS DES COMPOSANTS GRAPHIQUES

  void traitement () {
    bValider.addActionListener  (new ActionValider());
    bRejouer.addActionListener  (new ActionRejouer());
    bSolution.addActionListener (new ActionSolution());

    // les balles de couleurs disponibles
    MouseListener sourisEcouteur = new Disponible();
    for (int i=0; i < ballesDispo.length; i++) {
      ballesDispo[i].addMouseListener (sourisEcouteur);
    }
  }

  // pour récupérer les deux résultats de valider()
  class BPEtMP {  // Bien Placé et Mal Placé
    int bP = 0;
    int mP = 0;
  }

  // Traitements des événements des boutons

  // action pour le bouton Valider
  class ActionValider implements ActionListener {
    public void actionPerformed (ActionEvent evt) {
      Color[] propCoul = new Color[NB_BALLES];
      BPEtMP  resu     = new BPEtMP();
      for (int i=0; i < NB_BALLES; i++) {
        propCoul[i] = proposition [nbCoup][i].getClrBalle();
      }
```

```
      if (valider (couleursSol, propCoul, resu) ) {
        bPetMP [nbCoup].setText ( " " + resu.bP + "        "
                                                          + resu.mP);
        bPetMP [nbCoup].setBackground (Color.white);
        for (int j=0; j < proposition[nbCoup].length; j++) {
          proposition [nbCoup][j].removeMouseListener (ecouteurProp);
        }
        nbCoup++;
        if (resu.bP != NB_BALLES) {                // on n'a pas trouvé
          if (nbCoup < MAX_COUPS) {       // il reste des coups à jouer
            for (int j=0; j < proposition[nbCoup].length; j++) {
              proposition [nbCoup][j].addMouseListener (ecouteurProp);
            }
          } else {                    // on a épuisé les MAX_COUPS coups
            panel5.setVisible (true);
          }
        }
      }
    }  // class ActionValider

    // action pour le bouton Rejouer
    class ActionRejouer implements ActionListener {
      public void actionPerformed (ActionEvent evt) {
        panel2.setVisible (false);
        initDonnees();
        panel2.setVisible (true);
      }
    }

    // action pour le bouton Solution
    class ActionSolution implements ActionListener {
      public void actionPerformed (ActionEvent evt) {
        panel5.setVisible (true);
        // on ne peut plus jouer si on a demandé la solution
        for (int j=0; j < proposition[nbCoup].length; j++) {
          proposition [nbCoup][j].removeMouseListener (ecouteurProp);
        }
        bValider.setEnabled (false);             // bouton Valider inhibé
      }
    }
```

5.9.3.2 *Les écouteurs des événements de la souris pour le MasterMind*

Chacune des balles du panel "balles disponibles" est écoutée par un même objet (sourisEcouteur) de la classe Disponible (voir la méthode traitement() ci-dessus). Si on clique sur une des balles de cette zone, la méthode mousePressed() de la classe `Disponible` est exécutée ; elle range dans balleC la référence de la balle choisie.

```
// Traitements des événements de la souris

  // pour le choix d'une couleur disponible (de panel1);
  // modifie balleC (balle courante)
  class Disponible extends MouseAdapter {           // Adapter page 200
    public void mousePressed (MouseEvent evt) {
      balleC = (Balle) evt.getSource();
    }
  }
```

dansBalle() fournit le numéro de la balle du tableau tBalles sur laquelle on a cliqué.

```
// fournit le numéro dans tBalles de la Balle bClique
static int dansBalle (Balle[] tBalles, Balle bClique) {
    int i = 0;
    boolean trouve = false;
    while (!trouve && i < tBalles.length) {
      trouve = tBalles[i] == bClique;
      if (!trouve) i++;
    }
    return trouve ? i : -1;
}
```

Si on clique sur une des balles de la ligne de proposition écoutée par ecouteProp (voir ecouteProp page 209) la méthode mousePressed() de la classe Proposition est exécutée. On recherche le rang dans la ligne de la balle cliquée, et on recopie les caractéristiques de la balle courante dans cette balle.

```
// pour une proposition de couleur (panel2)
// on recopie les caractéristiques de la balle courante
// dans le tableau proposition, et on réaffiche la balle.
class Proposition extends MouseAdapter {          // Adapter page 200
  public void mousePressed (MouseEvent evt) {
    Balle bClique = (Balle) evt.getSource();
    int numB = dansBalle (proposition[nbCoup], bClique);
    if (numB != -1) {
      proposition[nbCoup][numB].setCoulDia (balleC.getClrBalle(),
                          balleC.getDiametre());
    }
  }
}
```

La méthode ci-dessous est purement algorithmique et appelée dans la classe ActionValider. Il s'agit de comparer deux tableaux de couleurs et d'indiquer le nombre de couleurs bien placées (occupant le même rang dans les tableaux) et le nombre de couleurs existant dans les deux tableaux mais placées à un rang différent. Deux tableaux de booléens marquent les éléments déjà pris en compte soit dans la solution, soit dans la proposition de façon à ne pas les compter deux fois.

```
// couleursSol : les couleurs de la solution à trouver
// propCoul    : la proposition des NB_BALLES couleurs
// le résultat BP et MP est placé dans un objet BPEtMP
```

```
  // S'il n'y a pas NB_BALLES propositions, on refuse de valider
  public boolean valider (Color[] couleursSol, Color[] propCoul,
                          BPEtMP resu) {
    int lg = couleursSol.length;
    // marqué dans la solution ou dans la proposition
    boolean[]  marqueSol  = new boolean [lg];
    boolean[]  marqueProp = new boolean [lg];

    int nB = 0;                     // nombre de points sans proposition
    for (int i=0; i < lg; i++) {
      if (proposition[nbCoup][i].getDiametre() == 10) nB++;
    }
    if (nB ! = 0) return false;           // il manque des propositions

    for (int i=0; i < lg; i++) {                  // à false par défaut
      marqueSol[i]  = false;
      marqueProp[i] = false;
    }

    int bP = 0;                                  // nombre de bien placées
    for (int i=0; i < lg; i++) {
      if (propCoul[i] == couleursSol[i]) {
        bP++;
        marqueSol[i]  = true;
        marqueProp[i] = true;
      }
    }
    resu.bP = bP;

    int mP = 0;                                   // nombre de mal placées
    for (int i=0; i < lg; i++) {
      if (marqueProp[i] ==false) {  // pas traité
        int j = 0;
        boolean trouve = false;
        while ( !trouve && (j < lg) ) {
          if (!marqueSol[j]) {
            trouve = propCoul[i] == couleursSol[j];
            if (trouve) {
              mP++;
              marqueProp [i] = true;
              marqueSol  [j] = true;
            }
          }
          if (!trouve) j++;
        }
      }
    }
    resu.mP = mP;

    return true;            // résultat dans la variable (l'objet) resu
  }  // valider

}  // class MasterMind
```

5.9.4 La mise en œuvre du composant MasterMind dans une application

La mise en œuvre du composant MasterMind dans une application consiste simplement à ajouter ce composant dans un conteneur de type Frame.

```
// PPMasterMind.java  sans menu déroulant

import java.awt.*;           // Frame
import mdpaquetage.mdawt.*;  // MasterMind

class PPMasterMind extends Frame {
  int NB_COUL   = 2;
  int NB_BALLES = 3;
  int MAX_COUPS = 7;

  PPMasterMind () {
     setTitle  ("MasterMind");
     setBounds (10, 10, 450, 600);
     addWindowListener (new FermerFenetre());          // voir page 221
     MasterMind m = new MasterMind (NB_COUL, NB_BALLES, MAX_COUPS);
     add (m, "Center");                          // defaut BorderLayout
     setVisible (true);
  }  // constructeur PPMasterMind

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

}  // PPMasterMind
```

La figure 5.37 correspond à l'exécution du programme précédent avec deux couleurs disponibles, trois balles et sept coups maximum.

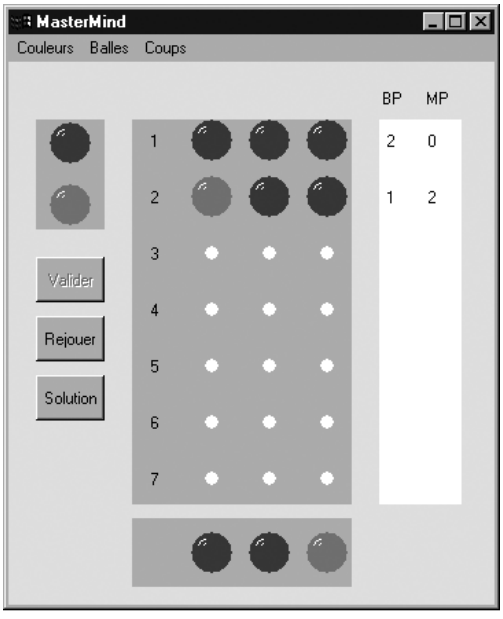


Figure 5.37 — MasterMind avec deux couleurs disponibles (bleu et rouge) et trois couleurs à découvrir (bleu, bleu, rouge) sur l'exemple) en sept coups maximum.

Exercice 5.5 Menu déroulant pour MasterMind

La figure 5.37 indique la présence d'un menu déroulant permettant de sélectionner le nombre de couleurs disponibles (de 2 à 6), le nombre de balles à découvrir (de 2 à 6) et le nombre de coups maximum (de 5 à 15). Écrire :

la méthode **afficherMen** () qui crée les trois menus de la barre de menus avec des éléments (MenuItem) présentant les différentes valeurs possibles. Enregistrer les écouteurs des classes **ActionMenu1**, **ActionMenu2** et **ActionMenu3** pour les trois menus.

les classes **ActionMenu1**, **ActionMenu2** et **ActionMenu3** qui récupérent la valeur sélectionnée, et créent un nouvel objet de type **MasterMind** ayant cette caractéristique.

Modifier la classe **PPMasterMind** pour qu'elle affiche le menu précédent.

5.10 LES FENÊTRES DE TYPE WINDOW, DIALOG ET FRAME

5.10.1 Les différentes fenêtres

La figure 5.12, page 151 indique trois types de fenêtres : Window, Dialog et Frame. Dans les exemples précédents, les fenêtres conteneurs utilisées étaient de type Frame.

Une fenêtre de type Window est une fenêtre sans titre, sans cadre et sans cases de fermeture ou de réduction. La figure 5.38 donne au deuxième rang à droite un exemple de fenêtre de type **Window**. C'est un conteneur, on peut lui ajouter des composants. On ne peut pas la déplacer ni modifier sa taille.

Une fenêtre de type **Dialog** a un bouton de fermeture mais pas de bouton de réduction ou d'agrandissement. Elle est faite pour un dialogue simple et pour être ensuite refermée. Elle peut être **modale**, c'est-à-dire que l'utilisateur doit fermer la boîte de dialogue avant de pouvoir agir sur un autre composant de l'application (même la sélection ou le déplacement des autres fenêtres est interdit et provoque un bip de refus). La figure 5.38 indique en bas deux exemples de fenêtre de dialogue : l'une est modale et l'autre non. La fenêtre modale est la dernière ouverte et bloque toute intervention sur les autres fenêtres de l'application jusqu'à sa fermeture. La fenêtre modale bloque sur l'instruction setVisible (true) la concernant.

Le programme Java suivant correspond à la figure 5.38. Il ouvre une fenêtre principale de type Frame et quatre autres fenêtres de type : Frame, Window, Dialog et Dialog. Toutes ces fenêtres sont des conteneurs ; on peut leur ajouter des composants standard ou maison. Une fenêtre de dialogue est affichée à la demande. Elle est rarement affichée dans un constructeur d'objet mais plutôt dans une méthode en cours d'exécution, en créant une nouvelle fenêtre de dialogue (Message) ou en la rendant visible.

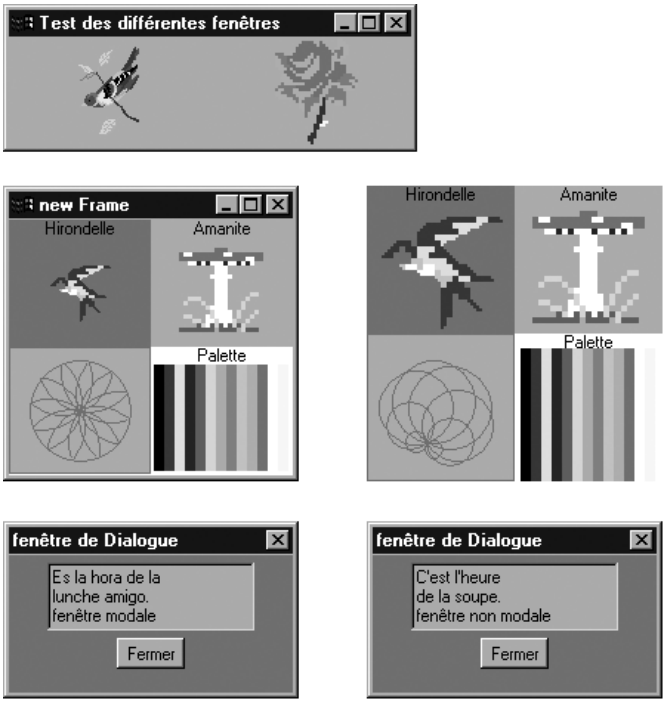


Figure 5.38 — Les différents types de fenêtres : Frame, Window et Dialog.
Les fenêtres sont des conteneurs : on peut leur ajouter des composants.

```
// Fenetres.java  test des 3 types de fenêtres :
//               Frame, Window et Dialog

import java.awt.*;          // Dialog
import java.awt.event.*;    // ActionListener
import mdpaquetage.mdawt.*; // Motif, FigGeo, FermerFenetre

class Message extends Dialog {

  // constructeur d'une fenêtre de dialogue
  // attachée à f, avec un titre et un message à afficher
  // fenêtre modale = bloquante jusqu'à sa fermeture
  Message (Frame f, String titre, String message, boolean modal) {
    super (f, titre, modal);                    // super-classe Dialog
    // defaut pour Dialog : BorderLayout
    setLayout (new FlowLayout (FlowLayout.CENTER));
    setBounds (450, 400, 200, 120);
    setBackground (Color.red);
    setResizable (false);          // on ne peut pas modifier la taille

    TextArea tA1 = new TextArea (message, 3, 20);
    tA1.setEditable (false);   // on ne modifie pas le texte du message
    tA1.setBackground (Color.lightGray);
    add (tA1);
```

```
    Button bFermer = new Button ("Fermer");
    add (bFermer);
    bQuitter.addActionListener (new ActionFermer());   // bouton Fermer
    addWindowListener (new FermerDialog());      // fenêtre de dialogue

    setVisible (true);         // bloquant ici si la fenêtre est modale
  }  // constructeur Message

  // pour le bouton Fermer
  class ActionFermer implements ActionListener {
    public void actionPerformed (ActionEvent evt) {
      dispose();                        // ferme la fenêtre de dialogue
    }
  }

  // pour fermer la fenêtre de dialogue avec la case de fermeture
 class FermerDialog extends WindowAdapter {
    public void windowClosing (WindowEvent evt) {
      dispose();                        // ferme la fenêtre de dialogue
    }
  }

}  // class Message
```

La classe Message ci-dessus affiche une fenêtre de dialogue en créant un nouvel objet de type Message comportant un message et un bouton "Fermer". Si la fenêtre est modale, seule sa fermeture (avec le bouton Fermer ou avec la case de fermeture) donne accès aux autres fenêtres.

Sur le motif m4 (champignon de la fenêtre "new Frame") ci-dessous, le curseur à la forme d'une main. Sur les deux figures géométriques (de Frame ou Window, deuxième ligne de la figure 5.38), il a la forme d'un sablier. Voir setCursor() ci-dessous.

```
class Fenetres extends Frame {

  Fenetres () {
    // la fenêtre principale de type Frame
    setTitle      ("Test des différentes fenêtres");
    setBackground (Color.lightGray);  // défaut : blanc
    setBounds     (10, 10, 280, 100); // défaut : petite taille
    setLayout     (new GridLayout (0, 2));
    Motif m1 = new Motif (MotifLib.oiseau,
                          Motif.tProp, MotifLib.palette1Oiseau);
    Motif m2 = new Motif (MotifLib.rose,
                             Motif.tProp, MotifLib.paletteRose);
    add (m1);
    add (m2);

    // création d'une nouvelle fenêtre de type Frame
    Frame w1 = new Frame ("new Frame");
    w1.setLayout (new GridLayout (0,2));
    w1.setBounds (700, 50, 200, 200);
    Motif m3 = new Motif (MotifLib.hirondelle, Motif.tProp,
                                                  "Hirondelle");
    m3.setBackground (Color.red);
```

```
        Motif m4 = new Motif (MotifLib.champignon,
                                                Motif.tProp, "Amanite");
        m4.setBackground (Color.lightGray);
        m4.setCursor (new Cursor (Cursor.HAND_CURSOR));
        FigGeo d1 = new FigGeo (6, Color.red, Color.lightGray);
        d1.setCursor (new Cursor (Cursor.WAIT_CURSOR));
        Motif m5 = new Motif (MotifLib.paletteV, Motif.tVari, "Palette");
        m5.setBackground (Color.white);
        w1.add (m3);
        w1.add (m4);
        w1.add (d1);
        w1.add (m5);
        //w1.toFront();                  // si les fenêtres se chevauchent
        w1.setVisible (true);                  // par défaut : non visible

        // création d'une fenêtre Window rattachée à un Frame
        Window w2 = new Window (this);         // this = fenêtre principale
        w2.setLayout (new GridLayout (0,2));
        w2.setBounds (700,400, 200, 200);
        w2.add (new Motif (m3));
        w2.add (new Motif (m4));
        FigGeo d2 = new FigGeo (7, Color.red, Color.lightGray);
        d2.setCursor (new Cursor (Cursor.WAIT_CURSOR));
        w2.add (d2);
        w2.add (new Motif (m5));
        w2.setVisible(true);                   // par défaut : non visible

        addWindowListener    (new FermerFenetre());  // principale
        w1.addWindowListener (new FermerFenetre());  // w1 (Frame)

        setVisible (true);       // défaut : invisible (fenêtre principale)
    }  // constructeur Fenetres

    public static void main (String[] args) {
      Fenetres f = new Fenetres();
      Message mes1 = new Message (f, "fenêtre de Dialogue",
        "C'est l'heure\nde la soupe.\nfenêtre non modale", false);
      mes1.setLocation (100, 150);
      Message mes2 = new Message (f, "fenêtre de Dialogue",
        "Es la hora de la\nlunche amigo.\nfenêtre modale", true);
    }
} // class Fenetres
```

5.10.2 La classe FermerFenetre (WindowListener) pour AWT

La fermeture d'une fenêtre de type Frame se fait toujours de la même façon en installant un objet de type **WindowListener** chargé d'indiquer ce qu'il faut faire quand on clique sur l'icône de fermeture. On peut définir la classe **FermerFenetre** et l'installer dans le paquetage mdawt. Il suffit alors d'installer dans l'application l'écouteur à l'aide de l'instruction :

```
addWindowListener (new FermerFenetre());
```

La classe **FermerFenetre** redéfinit la méthode **windowClosing**() qui indique qu'il faut fermer l'application si on clique sur l'icône de la fenêtre de nom "frame0" (la fenêtre principale) et fermer la fenêtre sinon (voir figure 5.38).

```
// FermerFenetre.java

package mdpaquetage.mdawt;

import java.awt.*;
import java.awt.event.*;  // WindowAdapter

// pour fermer la fenêtre ou l'application
public class FermerFenetre extends WindowAdapter {
  public void windowClosing (WindowEvent evt) {
    //System.out.println (evt.getWindow().getName());
    if (evt.getWindow().getName().equals("frame0")) {
        System.exit(0);                          // fermer l'application
    } else {
      evt.getWindow().dispose();                    // fermer la fenêtre
    }
  }
}
```

5.11 LA BARRE DE DÉFILEMENT (ADJUSTMENTLISTENER)

Une barre de défilement (voir Scrollbar page 144) visualise une variable dans l'ensemble de ses valeurs possibles. Un curseur se déplace dans une fenêtre et indique la valeur actuelle (voir figure 5.39). Cette valeur oscille entre un minimum et un maximum. La valeur actuelle est représentée comme un pourcentage de la différence entre le maximum et le minimum. Un clic sur la barre en A décrémente la valeur (d'une unité par défaut mais qui peut être définie à l'aide de la méthode **setUnitIncrement**()) et l'incrémente en B. Un clic en C ou D décrémente ou incrémente d'une valeur définie par la méthode **setBlockIncrement**(). Chaque clic sur la barre de défilement provoque un événement qui est pris en compte par un écouteur de type **AdjustmentListener**. La méthode **getValue**() pour un objet de type **Scrollbar** fournit la valeur que représente le curseur.

Constructeurs

- public static final int HORIZONTAL;
- public static final int VERTICAL ;
- public **Scrollbar** () : barre verticale.
- public **Scrollbar** (int orientation) : barre verticale ou horizontale suivant orientation.
- public **Scrollbar** (int orientation, int value, int visible, int minimum, int maximum) : barre avec une orientation (verticale ou horizontale), une valeur actuelle, la taille (du curseur) correspondant par exemple à la partie visible de ce que représente la barre (une partie d'un fichier par exemple, non utilisé sur l'exemple ci-dessous), et un intervalle de valeurs entre minimum et maximum.

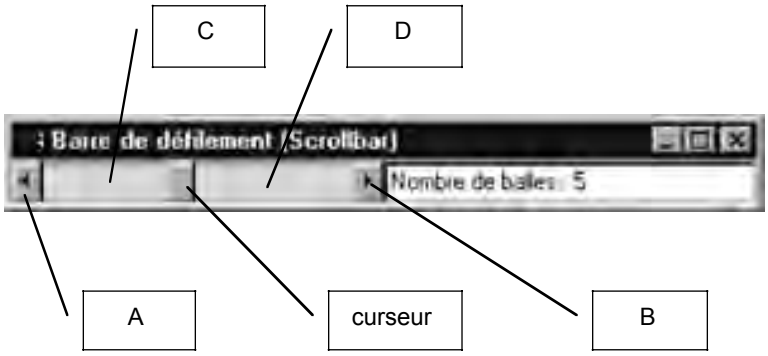


Figure 5.39 — Barre de défilement (Scrollbar).
Une barre de défilement horizontale (à gauche) et une zone de texte à droite (TextField) indiquant la valeur du curseur. On peut cliquer dans les zones A, B, C ou D. On peut aussi déplacer le curseur avec la souris pour faire varier la valeur représentée par la barre de défilement.

Méthodes :

Il existe des méthodes (accesseurs) pour fournir ou modifier les paramètres de la barre de défilement (minimum, maximum, etc.). Des clics dans différentes parties de la barre de défilement modifient la valeur actuelle. Les méthodes utilisées dans l'exemple sont :

- void **setBlockIncrement** (int *v*) : v = incrément en C ou D.
- void **setUnitIncrement** (int *v*) : v = incrément en A ou B.
- int **getValue** () : fournit la valeur représentée par le curseur.

La figure 5.40 donne un exemple d'utilisation d'une barre de défilement. La fenêtre contient trois composants : une barre de défilement représentant le nombre de balles à dessiner, une zone de texte indiquant le nombre de balles schématisé par le curseur et une zone de dessin comportant les balles placées aléatoirement dans la zone de dessin (voir la classe Balle page 193).

```
// Defilement.java  test de la barre de défilement

import java.awt.*;           // Canvas
import java.awt.event.*;     // AdjustmentListener
import mdpaquetage.mdawt.*;  // Balle, FermerFenetre
```

La classe **DessinNBalles** contient une méthode balles() pour générer un nombre de balles dépendant de l'entier valBarre et une méthode de dessin paint() de ces balles. Le nombre de balles à dessiner peut être changé avec la méthode setNbBalles().

```
// Dessin de N balles
class DessinNBalles extends Canvas {
  private int valBarre = 5;  // nombre de balles au départ
  private Balle[] tabBalles; // tableau de balles

  // créer valBarre Balle dans le tableau tabBalles
  // de diamètre 20 et de couleur rouge,
```

```
  // aléatoirement placées dans le rectangle d.
  public void balles (Dimension d) {
    int diametre = 20;                          // diamètre des balles
    tabBalles = new Balle [valBarre];
    for (int i=0; i < valBarre; i++) {
      tabBalles[i] = new Balle (Color.red, diametre, d);
    }
  }

  // modifier le nombre de balles
  public void setNbBalles (int valBarre) {
    this.valBarre = valBarre;
    repaint();
  }

  // dessiner les balles du tableau tabBalles
  // dans le contexte graphique de ce Component de type Canvas
  // Les balles ne sont pas ajoutées au Canvas.
  public void paint (Graphics g) {
    Dimension di = getSize();
    balles (di);          // place les balles dans le tableau tabBalles
    for (int i=0; i < valBarre; i++) tabBalles[i].paint(g);
  }

}  // DessinNBalles
```

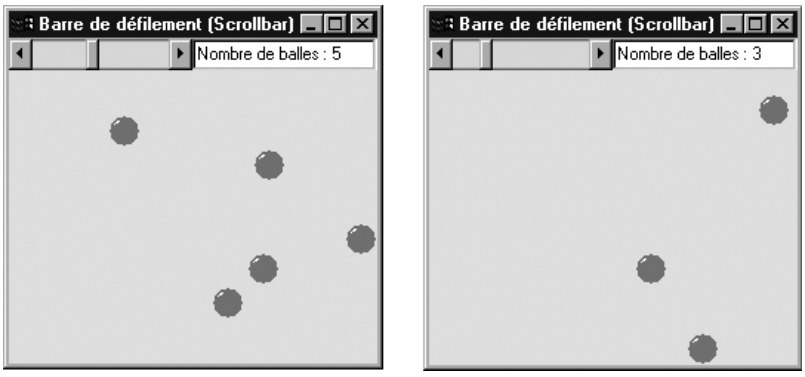


Figure 5.40 — Barre de défilement (Scrollbar) commandant le nombre de balles.

Le constructeur de la classe Defilement crée dans un Frame, un Panel contenant la barre de défilement (Scrollbar) et la zone de texte (TextField), et un objet de la classe DessinNbBalles défini ci-dessus. Un **écouteur** de la classe BarreDefilement est ajouté à la barre de défilement. La méthode **adjustmentValueChanged**() est appelée à chaque clic sur la barre ou déplacement du curseur.

```
public class Defilement extends Frame {
  Scrollbar     bDefil;
  TextField     tF1;
  DessinNBalles dessin;
```

```
  public Defilement() {
    setTitle   ("Barre de défilement (Scrollbar)");
    setBounds  (20, 20, 500, 500);
    // Frame de type BorderLayout par défaut

    // panel1 pour Scrollbar et TextField
    Panel panel1 = new Panel (new GridLayout(0, 2));
    int valBarre = 5;                  // valeur de départ du Scrollbar
    bDefil = new Scrollbar (Scrollbar.HORIZONTAL, valBarre, 0, 1, 11);
    bDefil.setBlockIncrement (3);
    //bDefil.setUnitIncrement (2);                        // par défaut 1
    panel1.add (bDefil);
    tF1 = new TextField ("Nombre de balles : " + valBarre);
    tF1.setEditable (false);
    panel1.add (tF1);
    add (panel1, "North");

    dessin = new DessinNBalles();
    dessin.setNbBalles (valBarre);       // valBarre = nombre de balles
    dessin.setBackground (Color.cyan);
    add (dessin, "Center");

    bDefil.addAdjustmentListener (new BarreDefilement());
    addWindowListener            (new FermerFenetre());// voir page 221
    setVisible (true);
  }

  class BarreDefilement implements AdjustmentListener {
    // méthode invoquée à chaque clic sur le Scrollbar
    public void adjustmentValueChanged (AdjustmentEvent evt) {
      int valBarre = bDefil.getValue();
      dessin.setNbBalles (valBarre);
      tF1.setText ("Nombre de balles : " + valBarre);;
    }
  }

  public static void main (String[] args) {
    new Defilement();
    //new Defilement();      // pour avoir un deuxième objet Defilement
  }

}  // class Defilement
```

Exercice 5.6 – Ballon créé par héritage de la classe Balle et réagissant aux clics et aux mouvements de la souris

Écrire une classe **Ballon** décrivant un nouveau composant **interactif** héritant de la classe Balle et permettant de déplacer le Ballon dans son espace en utilisant la souris (bouton appuyé). De plus, un clic dans la partie *gauche* de l'espace du Ballon doit le faire grossir, un clic dans la partie *droite* doit le faire se dégonfler. Lorsque le curseur de la souris entre dans l'espace du Ballon, celui-ci doit changer de couleur (devenir plus sombre par exemple). Il doit retrouver sa couleur initiale lorsque le curseur quitte

cet espace. Ballon est un nouveau composant qui peut être ajouté à un conteneur tout comme les composants prédéfinis (Button ou Scrollbar par exemple).

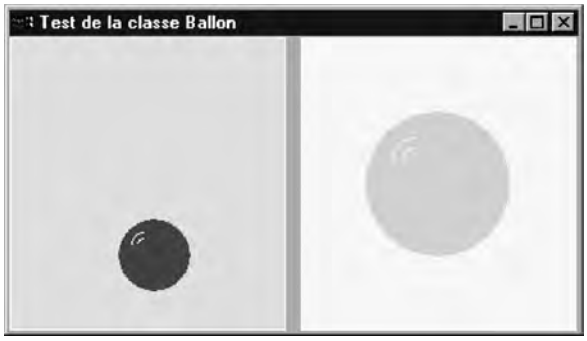


Figure 5.41 — Deux ballons gonflables et dégonflables.
Le premier est déplaçable ; le deuxième est centré dans son espace. La couleur du ballon change (de claire à foncée) lorsque le curseur entre ou sort de l'espace du ballon.

Le programme de mise en œuvre de la classe Ballon est donné ci-dessous.

```
// PPBallon.java  Programme Principal de test de la classe Ballon

import java.awt.*;           // Frame
import mdpaquetage.mdawt.*;  // Ballon, FermerFenetre

class PPBallon extends Frame {

  PPBallon () {
    setTitle  ("Test de la classe Ballon");
    setBounds (20, 20, 500, 300);
    setLayout (new GridLayout (0, 2, 10, 10));
    setBackground (Color.lightGray);

    // Ballon rouge de diamètre 50 centré en (100,150)
    // et déplaçable avec la souris
    Ballon ba1 = new Ballon (Color.red, 50, new Point (100, 150));
    ba1.setBackground (Color.cyan);
    add (ba1);

    // Ballon vert de diamètre 100 centré dans son espace
    Ballon ba2 = new Ballon (Color.green, 100);
    ba2.setBackground (Color.yellow);
    add (ba2);

    addWindowListener (new FermerFenetre());          // voir page 221
    setVisible (true);
  }  // constructeur PPBallon

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

}  //  PPBallon
```

Exercice 5.7 – Tracé de droites rebondissant sur les côtés d'un rectangle (Economiseur)

On veut dessiner des droites parallèles en mouvement rebondissant sur les côtés d'un rectangle (les côtés du cadre d'une fenêtre) comme indiqué sur la figure 5.42. Le nombre de droites parallèles et la durée en millisecondes entre deux déplacements peuvent être modifiés à l'aide de deux champs de saisie (de type TextField).

En utilisant la classe prédéfinie Point (voir figure 5.2, page 136), définir une classe **Droite** qui contient deux Point repérant les deux extrémités d'un segment de droite. Écrire les constructeurs et méthodes suivants :

– **Droite** (Point p1, Point p2) : constructeur d'une Droite à partir de deux Point.

– **Droite** (Droite d) : constructeur de copie.

– public String **toString** () : fournit les caractéristiques d'une Droite.

– void **dessinerDroite** (Graphics g) : dessine la Droite.

Écrire la classe **DessinDroites** dérivant de Component qui dessine les droites d'un tableau de Droite attribut de cette classe. Les méthodes disponibles sont les suivantes :

– void **setTableau** (Droite[] tabDroites) : tabDroites devient le tableau de Droite de l'objet.

– public void **paint** (Graphics g) : dessine les Droite du tableau de l'objet.

Des dessins mobiles sur l'écran sont parfois utilisés pour éviter que la même image ne se forme toujours au même endroit de l'écran. Ces dessins sont appelés économiseurs d'écran d'où le nom **Economiseur** choisi pour la classe de mise en œuvre du déplacement des droites. La classe dérive de Panel ; c'est un nouveau composant qui peut être mis en œuvre par le programme suivant :

```
// PPEconomiseur.java  Programme Principal Economiseur d'écran

import java.awt.*;           // Frame
import mdpaquetage.mdawt.*;  // Economiseur, FermerFenetre

class PPEconomiseur {
  public static void main (String[] args) {
    Frame f = new Frame();
    f.setTitle  ("Mouvement de droites");
    f.setBounds (10, 10, 800, 500);
    f.addWindowListener (new FermerFenetre());         // voir page 221
    Economiseur e1 = new Economiseur (300);
    f.add (e1, "Center");                    // BorderLayout par défaut
    f.setVisible (true);
    e1.deplacer();                                     // boucle infinie
  }
}  // PPEconomiseur
```

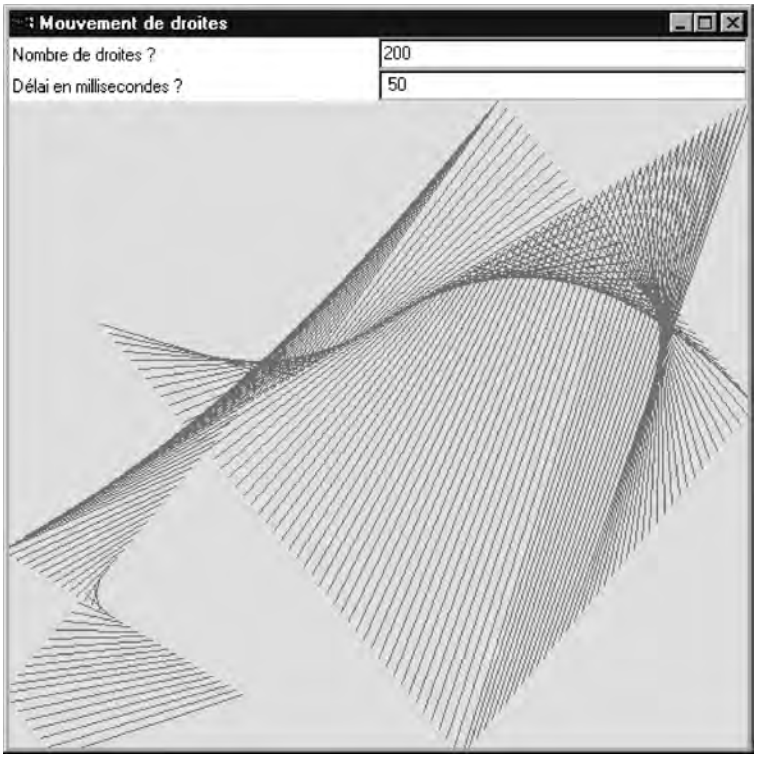


Figure 5.42 — Tracé de droites parallèles en mouvement.

Écrire les constructeurs, les méthodes et la classe interne suivants de la classe **Economiseur** dérivant de Panel :

– **public Economiseur** (int nbDroites, Color couleurDroites, Color couleurFond, int delai, int vx1, int vy1, int vx2, int vy2) : construit un objet à partir des paramètres nbDroites (nombre de Droite), des couleurs des Droite et du fond de la zone de dessin, du délai entre deux mouvements des Droite sur l'écran et des vitesses en x et en y de chacune des deux extrémités des Droite. Le constructeur construit initialement un tableau de nbDroites identiques, ce qui fait qu'au départ, une seule droite s'affiche. Les coordonnées des deux extrémités de la Droite de départ sont aléatoires dans la zone de dessin.

– **public Economiseur** (int nbDroites) : constructeur par défaut.

– **public void deplacer** () : boucle infinie de déplacement des Droite.

– class **GererParametres** (classe interne de gestion des deux valeurs des TextField).

Remarque : la méthode deplacer() gère un tableau de Droite. À chaque étape, les Droite sont décalées dans le tableau ; on perd ainsi la première Droite qui

n'est plus affichée. On ajoute en dernière position, les caractéristiques de la nouvelle Droite à afficher en tenant compte des vitesses des deux extrémités de la Droite qui ne sont pas les mêmes.

Exercice 5.8 – Écriture d'un programme de tracés de courbes (utilise une classe abstraite)

Écrire une classe **DessinCourbe** traçant une courbe dans une zone de dessin (dérivant de Component). Les constructeurs et les méthodes sont les suivants :

DessinCourbe (double tVal[], double xa, double xh) ; construit un objet à partir d'un tableau tVal de valeurs. L'abscisse du premier point est xa et les points sont espacés d'un pas xh.

– public double **min** () : fournit la valeur minimale de tVal.

– public double **max** () : fournit la valeur maximale de tVal.

– public void **paint** (Graphics g) : trace la courbe et éventuellement les axes de l'origine.

Écrire la classe **PPCourbe1** qui trace les courbes de la figure 5.43 en initialisant un tableau de double et en utilisant la classe DessinCourbe.

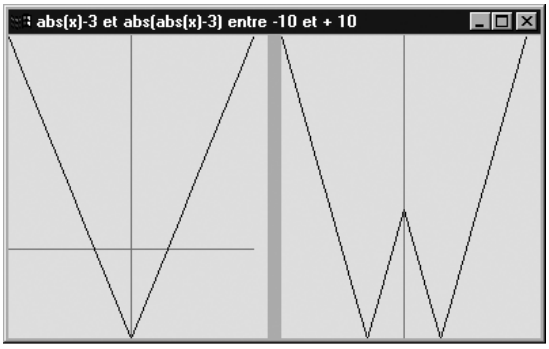


Figure 5.43 — |x|-3 et ||x|-3| entre -10 et 10.

Écrire une classe **abstraite Courbe** dérivant de Panel et ayant les attributs suivants :

double tVal[] : tableau des valeurs à tracer

double xa : x de départ

double xh : écart entre deux points sur l'axe des x

et les méthodes suivantes :

– **Courbe** (String nord, double xa, double xb, double xh) : constructeur de Courbe créant un composant au nord du Panel contenant le message nord, un composant au sud indiquant les valeurs minimum et maximum et un composant au centre de type **DessinCourbe** dessinant les valeurs du tableau tVal.

– public abstract double **fn** (double x) : la fonction à tracer à définir dans la classe dérivée.

– public void **tabulation** () : la méthode de tabulation calculant les valeurs rangées dans tVal de la courbe pour la fonction abstraite fn() supposée définie sur l'intervalle des valeurs.

Écrire les classes **Courbe1**, **Courbe2**, **Courbe3**, **Courbe4** dérivant de la classe **Courbe** et définissant la fonction abstraite **fn**() de Courbe pour chacune de ces classes.

Écrire la classe **PPCourbe2** qui trace les courbes de la figure 5.44.

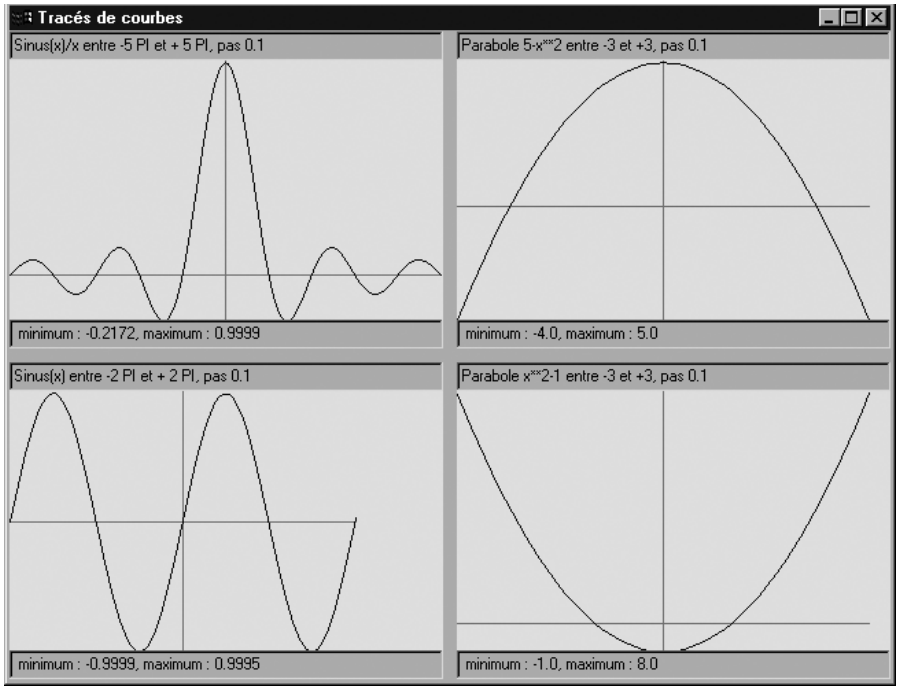


Figure 5.44 — Tracé de courbes (sinusoïdes et paraboles).

Exercice 5.9 – Dessin de nombres complexes dans un contexte graphique

Une classe Complex opérant sur les nombres complexes a été présentée en page 45. L'utilisation du graphisme permet d'illustrer les concepts géométriques liés aux nombres complexes. On crée une classe **ComplexG** (complex graphique) dérivant de la classe Complex et mémorisant en plus des parties réelle et imaginaire, la couleur du trait représentant un complexe, un booléen indiquant si on doit afficher la valeur du complexe, et un message éventuellement nul qui précède cette valeur. La classe ComplexG est définie ci-dessous. Elle hérite de la classe Complex. Elle fournit deux constructeurs et une méthode dessinant dans un contexte graphique.

La classe ComplexG n'est pas de type Component et ne dispose pas de son propre contexte graphique. La méthode void **dessiner** (Graphics g, Point origine, int unite) ; trace une droite entre le point origine et le point représentant la valeur complexe en tenant compte de l'unité qui est la même sur les deux axes et de la couleur de l'objet ComplexG.

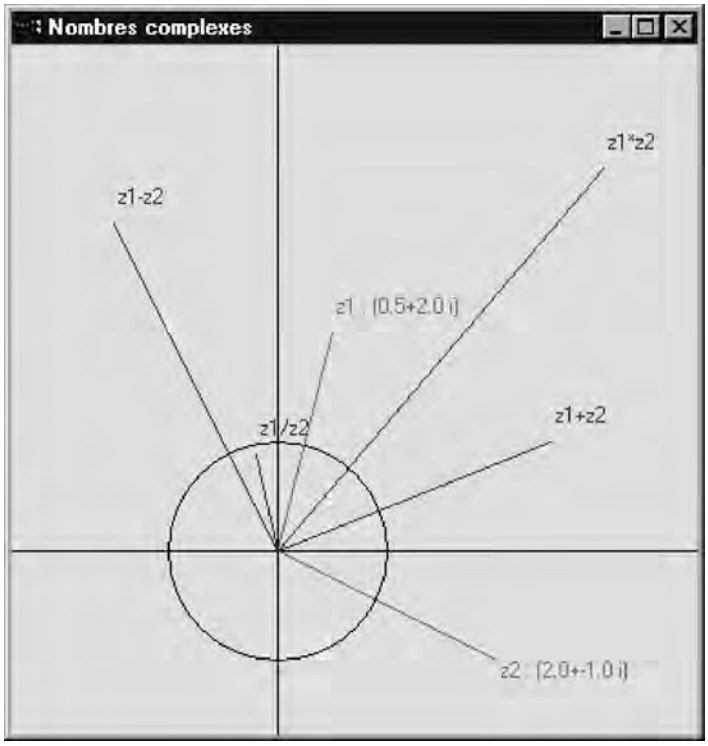


Figure 5.45 — Représentation graphique des nombres complexes et du cercle unité.

L'origine et l'unité sont fixées. Le dessin ne se modifie pas si la taille de la fenêtre change. Le dessin peut être rendu ajustable en modifiant une option.

La classe ComplexG :

```
// ComplexG.java

package mdpaquetage.complex;

import java.awt.*;     // Graphics

// Un nombre Complex plus la couleur du trait pour le dessin
class ComplexG extends Complex {
  Color   c;        // Couleur du trait schématisant le nombre complexe
  boolean ecrireValeur;                    // écrire ou non la valeur
  String  message;             // faire précéder l'écriture de message
```

```
  ComplexG (Complex z, Color c, String message, boolean ecrireValeur){
    super (z.partieRC(), z.partieIC());         // super-classe Complex
    this.c = c;
    this.ecrireValeur = ecrireValeur;
    this.message = message;
  }

  ComplexG (Complex z, Color c) {
    this (z, c, null, true);
  }

  // dessiner un nombre complexe dans le contexte graphique g
  // connaissant les coordonnées du point origine
  // et la valeur en pixels de l'unité
  void dessiner (Graphics g, Point origine, int unite) {
    g.setColor (c);
    int finX = origine.x + (int)(partieRC()*unite);
    int finY = origine.y - (int)(partieIC()*unite);
    double pR = ((int)(partieRC()*100))/100.0;        // deux décimales
    double pI = ((int)(partieIC()*100))/100.0;
    g.drawLine (origine.x, origine.y, finX, finY);
    String etiquette = message ==null ? "" : message;
    if (ecrireValeur) {
      etiquette += "(" + pR + "+" + pI + " i)";
    }
    g.drawString (etiquette, finX + 2, pI > 0 ?finY-10 : finY+10);
  }

}  // ComplexG
```

Écrire une classe **DessinZ** qui dessine plusieurs ComplexG dans une même fenêtre de type Panel. Les nombres complexes à représenter sont mémorisés dans une liste (voir page 111). L'unité et l'origine peuvent être fixées une fois pour toutes ou être ajustables suivant l'espace attribué pour ce dessin. Les méthodes suggérées sont les suivantes :

– **DessinZ** (int unite, Point origine) : constructeur pour un dessin avec une unité et une origine fixées quelles que soient les dimensions de l'espace.

– **DessinZ** () : constructeur pour un dessin à ajuster.

– public void **ajouter** (ComplexG z) : ajouter z à la liste des complexes à dessiner.

– void **ajusterUnite** (Dimension d) : calculer unité et origine.

– void **dessinerAxes** (Graphics g, Dimension d) : dessiner les axes de dimension d.

– public void **paint** (Graphics g) :dessiner les complexes de la liste.

Écrire la classe **PPComplex** dessinant les complexes des figures 5.45 et 5.46.

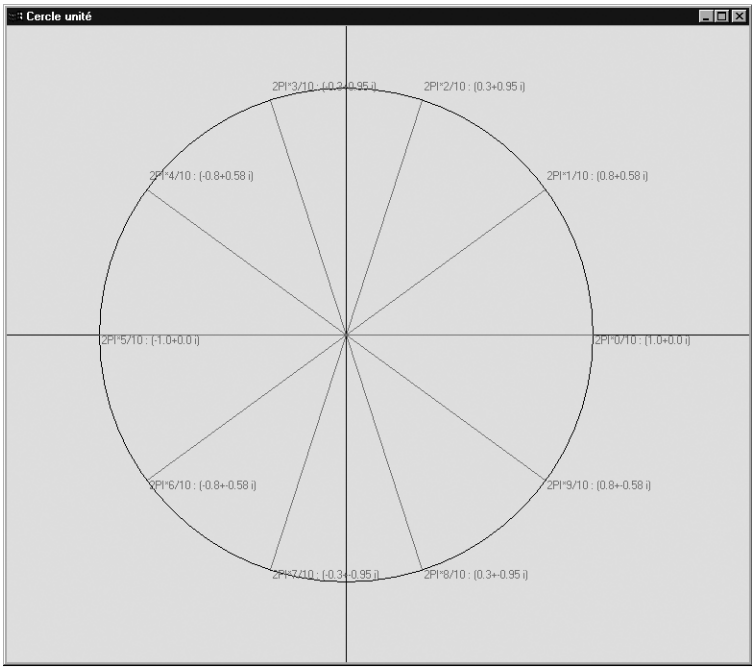


Figure 5.46 — Nombres complexes sur le cercle unité ($e^{i*j*PI/10}$ pour j de 0 à 9).
Le dessin s'ajuste de manière proportionnelle en fonction des dimensions de la fenêtre.

5.12 LA GESTION DU CLAVIER

5.12.1 Les écouteurs de type KeyListener

La prise en compte de caractères frappés au clavier s'effectue de la même façon que pour tous les événements. Un composant qui désire recevoir des informations en provenance du clavier doit le demander en enregistrant un écouteur de type **KeyListener** à l'aide de la méthode **addKeyListener**() (voir page 139). L'interface KeyListener définit les prototypes des méthodes **keyPressed**(), **keyReleased**() et **keyTyped**(). La classe **KeyAdapter** redéfinit des méthodes vides (voir Adapter page 200), ce qui permet de ne redéfinir par héritage que les méthodes réellement utiles. Le composant qui reçoit les caractères frappés est celui qui a le focus.

5.12.2 La création d'une classe MenuAide de menu d'aide (touche F1)

On définit une classe MenuAide qui affiche pour un composant un fichier d'aide quand la touche F1 du clavier est pressée et que le composant a le focus. Un objet de la classe MenuAide est caractérisé par les attributs suivants :

- le composant l'ayant appelé ;
- le nom du fichier de type texte à afficher ;
- un titre pour la fenêtre d'aide.

Les méthodes sont les suivantes :

- **MenuAide** (Component comp, String nomFichier, String titre) : constructeur.
- void **creerFenetre** () : crée et affiche la fenêtre d'aide (de type Frame).

Le constructeur de **MenuAide** enregistre un objet écouteur de la classe **GererAideClavier** implémentant l'interface **KeyListener**. La classe interne **GererAideClavier** définit la méthode **keyPressed**() qui indique ce qu'il faut faire lorsque la touche F1 est activée alors que ce composant a le focus. Dans ce cas, la méthode creerFenetre() affiche la fenêtre d'aide à partir des informations du fichier.

```
// MenuAide.java  classe permettant d'ouvrir un menu d'aide
//                en appuyant sur la touche F1 du clavier

package mdpaquetage.mdawt;

import java.awt.*;        // Frame
import java.awt.event.*;  // KeyListener
import java.io.*;         // BufferedReader

// classe MenuAide permettant d'ouvrir un menu d'aide
// spécifique du composant ayant le focus
// quand on appuie sur la touche F1 du clavier
public class MenuAide extends Frame {
  Component comp;        // le composant ayant le focus recevant F1
  String    nomFichier;  // nom du fichier contenant le menu d'aide
  String    titre;       // titre de la fenêtre d'aide

  public MenuAide (Component comp, String nomFichier, String titre) {
    this.comp       = comp;
    this.nomFichier = nomFichier;
    this.titre      = titre;
    // comp est réceptif aux événements du clavier le concernant
    comp.addKeyListener (new GererAideClavier());
  }

  // créer une fenêtre d'aide de type Frame contenant un TextArea.
  // lire le fichier d'aide et l'afficher dans le TextArea
  private void creerFenetre() {
    setTitle  (titre);
    setBounds (650, 150, 300, 200);
    TextArea aide = new TextArea ();
    aide.setEditable (false);
    add (aide, "Center");  // Frame : défaut BorderLayout

    // lecture du fichier d'aide (voir chapitre 6)
    BufferedReader fe = null;
    String textAide = "";
    try {  // ouverture et lecture du fichier fe
      fe = new BufferedReader (new FileReader(nomFichier));
```

```
        String ch;
        // lecture des lignes
        while ( (ch = fe.readLine()) != null) {
          textAide += ch + '\n';
          aide.setText (textAide);
        }
      } catch (Exception e) {
        aide.setText ("Erreur lors de la lecture du fichier d'aide\n"
                                                                    + e);
      }

      addWindowListener (new FermerFenetreAide());
      setVisible(true);
    } // creerFenetre

    // pour ouvrir la fenêtre d'aide
    // si elle n'est pas déjà ouverte
    class GererAideClavier extends KeyAdapter {
      public void keyPressed (KeyEvent evt) {
        int code = evt.getKeyCode();
        if (code == KeyEvent.VK_F1) {
          creerFenetre();
        }
      }
    }

    // pour fermer la fenêtre d'Aide
    class FermerFenetreAide extends WindowAdapter {
      public void windowClosing (WindowEvent evt) {
        dispose();
      }
    }

  }  // class MenuAide
```

5.12.3 Exemple de mise en œuvre du menu d'aide

5.12.3.1 Mise en œuvre dans une application

Pour une application (jeu du pendu par exemple, voir 5.6), l'ajout d'un menu d'aide se fait de façon très simple avec la classe définie ci-dessus. Il suffit de créer le fichier "pendu.txt" à afficher lorsqu'on appuie sur F1 expliquant les règles du jeu, et de créer un objet MenuAide dans le constructeur :

```
new MenuAide (this, "pendu.txt", "Comment jouer au Pendu ? ");
```

Chaque composant d'une interface graphique peut avoir un menu d'aide approprié grâce à un objet de type MenuAide. Ci-dessous, le composant tF (type TextField) défini un menu d'aide affiché lorsque ce composant a le focus et qu'on appuie sur F1.

```
new MenuAide (tF,  "penduCar.txt", "Entrez un caractère");
```

5.12.3.2 Utilisation d'un FocusListener

La figure 5.47 présente trois composants dans une fenêtre de type Frame : un TextArea, un composant Balle (voir page 196) et un composant Motif (voir page 153). Ces trois composants enregistrent un objet écouteur ml (implémentant **MouseListener**) de la classe **PrendFocus** qui définit la méthode **mouseEntered**() indiquant ce qu'il faut faire lorsque le curseur de la souris entre ou sort de l'espace du composant. Cette méthode appelle **requestFocus**() qui permet au composant d'avoir le focus lorsque le curseur de la souris se trouve dans son espace.

Les trois composants enregistrent également un objet écouteur fl de type **ChangeColor** implémentant l'interface **FocusListener** qui indique ce qu'il faut faire lorsque le focus change pour ce composant. L'interface FocusListener définit deux méthodes :

- public void **focusGained** (FocusEvent evt) : le composant prend le focus.
- public void **focusLost** (FocusEvent evt) : le composant perd le focus.

Sur l'exemple suivant, le fond de la fenêtre est en jaune quand le composant a le focus, sinon, il a sa couleur normale.

Les trois composants créent un objet **MenuAide** les concernant. Si on appuie sur F1 lorsque le curseur se trouve dans l'espace du composant et donc qu'il a le focus (voir ci-dessus), une fenêtre de menu d'aide s'ouvre avec un texte d'aide approprié. Sur la figure 5.47, on a appuyé sur F1 alors que le curseur était dans l'espace de la balle. Un menu d'aide différent s'afficherait si le curseur était dans la zone de saisie ou sur le Motif oiseau. Pour un composant donné, si la fenêtre d'aide est déjà ouverte, l'appui sur F1 est sans effet.

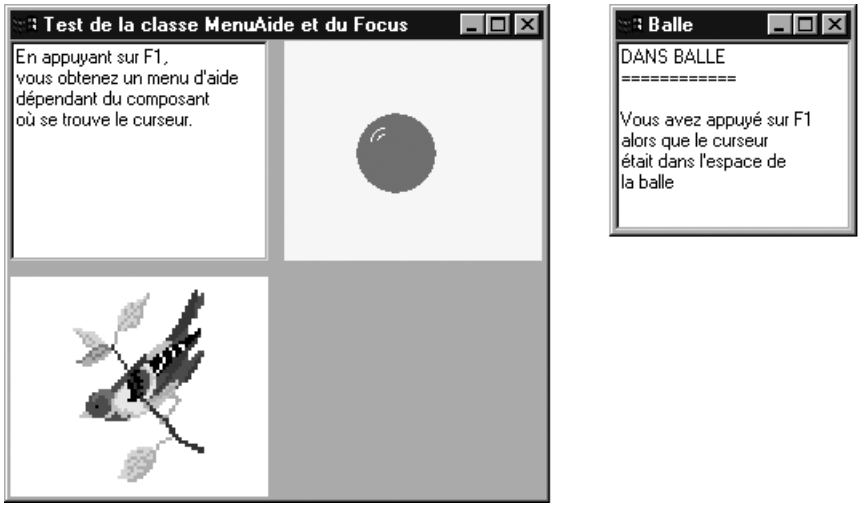


Figure 5.47 — Test de la classe MenuAide et du Focus.
Un menu d'aide activé en appuyant sur la touche F1 et dépendant du composant ayant le focus qui est mis en évidence par un arrière-plan jaune.

```
// PPMenuAide.java  Programme Principal de test du menu d'aide

import java.awt.*;           // Frame
import java.awt.event.*;     // MouseAdapter
import mdpaquetage.mdawt.*;  // MenuAide, FermerFenetre

class PPMenuAide extends Frame {

  PPMenuAide() {
    setTitle   ("Test de la classe MenuAide et du Focus");
    setBounds  (20, 20, 500, 500);
    setLayout  (new GridLayout (0, 2, 10, 10));
    setBackground (Color.lightGray);

    String s =  "En appuyant sur F1, "
              + "\nvous obtenez un menu d'aide"
              + "\ndépendant du composant"
              + "\noù se trouve le curseur.";
    TextArea tf1 = new TextArea (s);
    // TextArea b1 = new TextArea (s);
    // TextArea m1 = new TextArea (s);

    Balle b1 = new Balle (Color.red, 50, new Point (70, 70));
    Motif m1 = new Motif (MotifLib.oiseau, Motif.tProp,
                          MotifLib.palette1Oiseau );

    b1.setBackground (Color.blue);
    m1.setBackground (Color.white);
    add (tf1);
    add (b1);
    add (m1);

    MouseListener ml = new PrendFocus();
    b1.addMouseListener  (ml);
    m1.addMouseListener  (ml);
    tf1.addMouseListener (ml);

    FocusListener fl = new ChangeColor();
    b1.addFocusListener  (fl);
    m1.addFocusListener  (fl);
    tf1.addFocusListener (fl);

    addWindowListener    (new FermerFenetre());        // voir page 221

    new MenuAide (b1,  "balle.txt",     "Balle");
    new MenuAide (m1,  "motif.txt",     "Motif");
    new MenuAide (tf1, "textField.txt", "TextField");
    setVisible (true);
  }  // constructeur PPMenuAide
```

```
    // pour gérer l'entrée ou la sortie de la souris de l'espace
    class PrendFocus extends MouseAdapter {
      public void mouseEntered (MouseEvent evt) {
        Component comp = (Component) evt.getSource();
        comp.requestFocus();
      }

      public void mouseExited (MouseEvent evt) {
        requestFocus();  // le Frame prend le focus
      }
    }

    // pour gérer la prise ou la perte de focus
    class ChangeColor implements FocusListener {
      Color anc;

      public void focusGained (FocusEvent evt) {
        Component comp = (Component) evt.getSource();
        anc = comp.getBackground();
        comp.setBackground (Color.yellow);
      }

      public void focusLost (FocusEvent evt) {
        Component comp = (Component) evt.getSource();
        comp.setBackground (anc);
      }
    }

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

  }  //  PPMenuAide
```

5.13 CONCLUSION

L'interface graphique Java montre bien la richesse et la souplesse de mise en œuvre des composants graphiques. La programmation objet est très utile et permet de mettre en commun les caractéristiques des composants (la classe Component) et d'obtenir par héritage des composants plus spécifiques. La bibliothèque AWT définit un certain nombre de composants standard (étiquettes, zone de saisie de texte, boîtes à cocher, listes déroulantes, menus déroulants, etc.).

D'autres composants ont été définis de manière similaire (Motif, Balle, FigGeo, MasterMind, DessinCourbe). Les composants peuvent s'afficher dans un espace attribué par le programmeur ou plus souvent par un gestionnaire de mise en page qui obéit à certaines contraintes pour disposer les composants dans la fenêtre ou le panneau recevant ces composants et jouant le rôle de conteneur de composants.

Les composants peuvent aussi réagir à certains événements comme un clic de la souris ou une entrée au clavier. Pour cela, ils doivent le demander en définissant un objet écouteur dépendant du type des événements attendus. Les prototypes des

méthodes permettant de prendre en compte les événements sont définis dans des interfaces (au sens Java). La classe de l'objet écouteur doit définir les méthodes traitant les événements de l'écouteur.

La compréhension de l'arbre d'héritage des divers éléments graphiques (voir figure 5.12) est fondamentale. De là dépendent les diverses méthodes qu'on peut appliquer à un composant. La consultation de la documentation en ligne est aussi une nécessité. Le nombre de classes et de méthodes est très important en Java et il est impossible (de plus ce serait fastidieux) de tout expliquer ou illustrer en détail par des exemples.

Ce livre est là pour expliquer les concepts de base (héritage, composants graphiques, écouteurs d'événements, etc.) et développer des exemples réalistes illustrant ces concepts.

Chapitre **6**

La librairie Swing

6.1 GÉNÉRALITÉS

La librairie AWT utilise certaines des fonctions du système d'exploitation (pour l'affichage des fenêtres par exemple). Un même composant AWT aura une apparence différente sur différentes plate-formes. La portabilité est assurée, mais l'apparence peut être différente à partir du même fichier de bytecode (un même .class).

La librairie Swing est totalement gérée par la machine virtuelle ce qui permet de ne plus se limiter au sous-ensemble de composants graphiques communs à toutes les plates-formes. Swing définit de nouveaux composants. Cependant, la bibliothèque Swing nécessite plus de temps CPU, car la machine virtuelle doit dessiner les moindres détails des composants. Cette indépendance vis-à-vis du système d'exploitation permet même de décider, sur n'importe quelle machine, de l'apparence (le look and feel) que l'on veut donner à ses composants (style Linux, style Windows, etc.). Certains composants peuvent être complexes à mettre en œuvre si on veut les exploiter dans tous leurs détails (les listes, les arbres ou les tables par exemple). Swing est fourni en standard avec le JDK (Java Development Kit).

> ***Remarque*** : tous les look and feel ne sont pas disponibles en standard dans le JDK.

6.2 LA CLASSE JCOMPONENT

La bibliothèque Swing contient de nombreux composants, chacun d'entre eux ayant de nombreuses méthodes. Le but de ce chapitre est de montrer à travers explications et exemples les potentialités de cette bibliothèque. Il est bien sûr impossible d'être

exhaustif. La documentation en ligne est nécessaire, et est là pour fournir le détail des nombreuses méthodes des composants.

La librairie Swing se trouve dans le paquetage : javax.swing. La figure 6.1 présente les composants Swing les plus classiques et qui ont pour certains un équivalent dans la librairie AWT. Le composant Label de AWT par exemple se nomme JLabel en Swing. Il y a cependant des différences plus profondes entre ces 2 composants aux noms similaires. Sur la figure 5.12, page 151, le composant AWT Label hérite de Component. Sur la figure 6.1, le composant Swing JLabel hérite de JComponent qui hérite de Container qui hérite de Component.

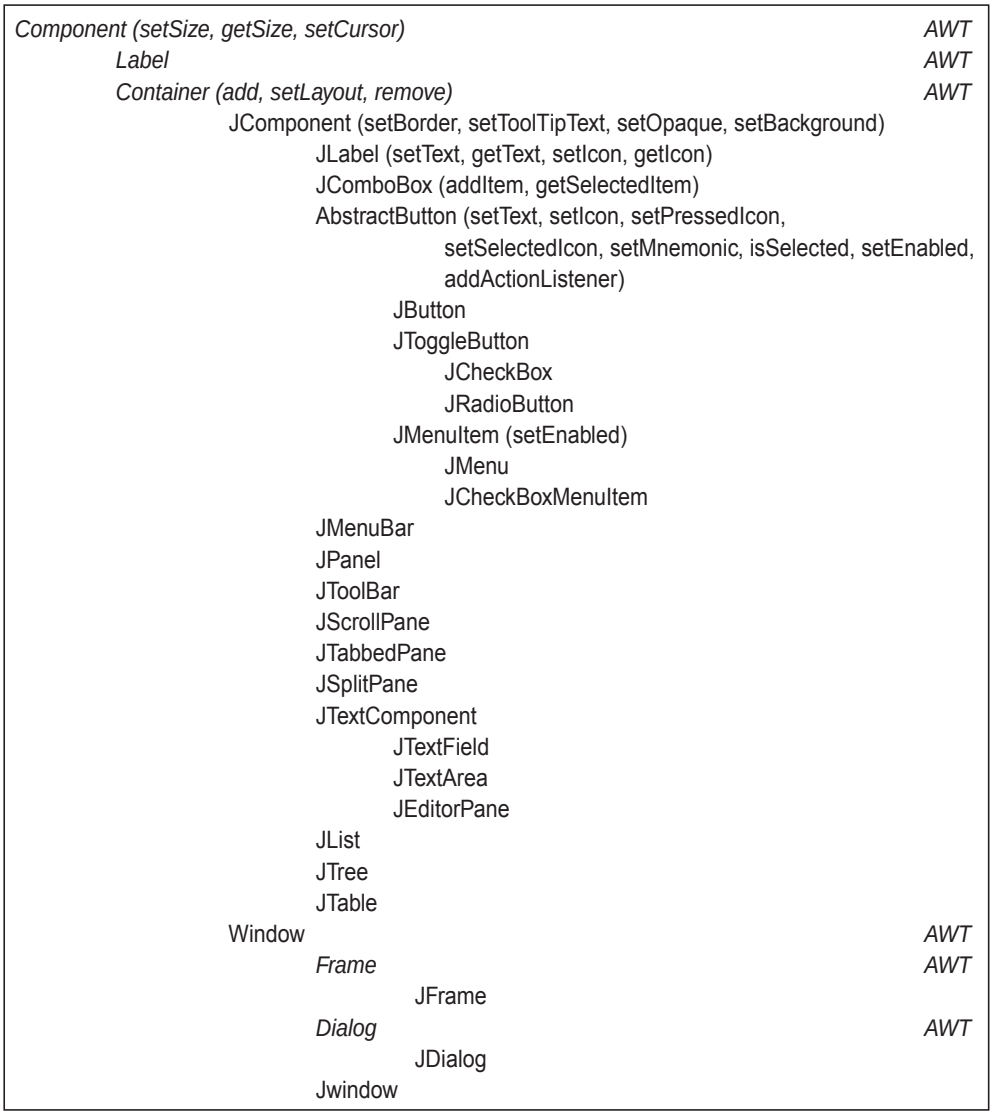


Figure 6.1 — Les composants Swing. En italique, les composants AWT.

Un JComponent est un Container. La classe JComponent contient les méthodes héritées de ses ascendants Container et Component de AWT. Autour d'un JComponent, on peut mettre différents types de bordures. On peut aussi afficher des bulles d'aide sur les JComponent.

Exemples :

mettre une bordure sur le JComponent c :

```
c.setBorder (BorderFactory.createTitleBorder ("Titre"));
```

afficher une bulle d'aide pour le JComponent c :

```
c.setToolTipText ("Bulle d'aide");
```

6.3 LA CLASSE JLABEL

Un JLabel est un composant qui permet d'afficher un texte non modifiable par l'utilisateur (modifiable seulement par programme) et/ou un icône. Un JLabel ne peut pas être sélectionné et par conséquent, il ne peut pas avoir le focus. Voir Label de AWT en page 142.

Un JLabel a un fond transparent qui prend la couleur de son conteneur. En indiquant que le fond est opaque, on peut définir une couleur pour le JLabel. Un icône avec ou sans texte peut être défini pour un JLabel, en précisant éventuellement leur emplacement respectif à l'aide de méthodes spécifiques de JLabel. Le texte peut être écrit en HTML et délimité par les deux balises < html > et </html > ; il peut donc comporter des caractères gras, italiques, de différentes couleurs, etc.

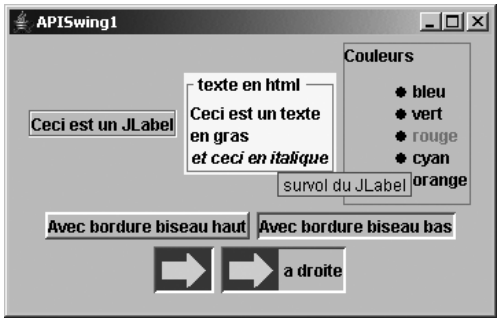


Figure 6.2 — Exemples de JLabel avec texte (en html), icônes, bordures, et bulles d'aide.

Le programme suivant définit et affiche les JLabel de la figure 6.2.

```
// APISwing1.java JLabel

import java.awt.*;     // Container
import javax.swing.*;  // JLabel, JPanel, JFrame

class APISwing1 extends JFrame {

  APISwing1 () {
    setTitle ("APISwing1");
    Container f = (Container) getContentPane;
```

```
setBounds (10, 10, 350, 250);

JPanel p1 = new JPanel ();
f.add(p1);

// JLabel : texte + bordure
JLabel la = new JLabel ("Ceci est un JLabel");
la.setBorder (BorderFactory.createEtchedBorder());
p1.add (la);

// JLabel : texte en html + bordure + fond jaune + bulle d'aide
JLabel lb = new JLabel
          ("<html > Ceci est un texte < br > < b > en gras </b > "
               + "<br > < i > et ceci en italique </i > </html > ");
lb.setBorder (BorderFactory.createTitledBorder
                                                  (" texte en html "));
lb.setOpaque (true);
lb.setBackground (Color.YELLOW);
lb.setToolTipText ("survol du JLabel");
p1.add (lb);

// JLabel : liste en html + bordure rouge
JLabel lc = new JLabel (
  "<html > Couleurs"  +
  "<ul > "            +
  "<li > bleu </li > "   +
  "<li > vert </li > "   +
  "<li > < font COLOR = ff0000 > rouge </font > </li > " +
  "<li > cyan </li > "   +
  "<li > orange </li > " +
  "</ul > </html > "
);
lc.setBorder (BorderFactory.createLineBorder(Color.red));
p1.add (lc);

// JLabel : texte + bordure
JLabel ld = new JLabel ("Avec bordure biseau haut");
ld.setBorder (BorderFactory.createRaisedBevelBorder());
p1.add (ld);

// JLabel : texte + bordure
JLabel le = new JLabel ("Avec bordure biseau bas");
le.setBorder (BorderFactory.createLoweredBevelBorder());
p1.add (le);

// JLabel : icône + bordure + curseur
JLabel lf = new JLabel (new ImageIcon ("flechedroite.gif"));
lf.setBorder (BorderFactory.createLoweredBevelBorder());
lf.setCursor(new Cursor(Cursor.HAND_CURSOR));
p1.add (lf);

// JLabel : texte + icône + bordure + curseur
JLabel lg = new JLabel ("a droite",
                new ImageIcon ("flechedroite.gif"), JLabel.CENTER);
lg.setBorder (BorderFactory.createLoweredBevelBorder());
```

```
    lg.setCursor(new Cursor(Cursor.HAND_CURSOR));
    p1.add (lg);

    setVisible (true);
  }  // constructeur APISwing1

  public static void main (String[] args) {
    new APISwing1();
  }

}  // class APISwing1
```

6.4 LA CLASSE JCOMBOBOX

Un JComboBox est un composant qui permet de choisir une option dans une liste déroulante. Voir Choice en AWT , page 144. On peut indiquer le nombre d'éléments à afficher quand la liste est ouverte. Un JComboBox peut être créé à partir d'un tableau ou d'un Vector d'objets. On peut aussi lui ajouter des éléments avec la méthode addItem(). Le JComboBox peut être rendu éditable. La méthode getSelectedItem() fournit le rang de l'élément sélectionné.

```
String[]  choix = {"vrai", "faux"};
JComboBox c     = new JComboBox (choix);
```

Remarque : les données à afficher dans le composant peuvent être mises en forme (en cas de présentation particulière) par les méthodes d'un objet effectuant la présentation voulue (implémenter l'interface ListCellRenderer). Cette mise en forme est traitée pour les JList en page 261.

6.5 LA CLASSE ABSTRACTBUTTON (POUR LES BOUTONS)

La classe abstraite AbstractButton définit une classe qui regroupe les caractéristiques de tous types de bouton. Les méthodes de cette classe permettent de définir un texte éventuellement illustré d'un icône, de définir un raccourci clavier ou un écouteur qui définit l'action à prendre en compte pour ce bouton. Des icônes différents suivant l'état du bouton (normal, appuyé, survolé ou désactivé) peuvent être définis. On peut gérer la place de l'icône par rapport au texte.

définir un texte et/ou un icône pour le bouton

```
setText, setIcon
```

définir un raccourci clavier

```
setMnemonic (char c)
```

définir un icône

```
setIcon                                  pour l'état désélectionné
setPressedIcon                                 pour l'état appuyé
setRolloverIcon                          pour le survol du curseur
setDisabledIcon                 en cas de désactivation du bouton
```

définir un écouteur de type ActionListener

`addActionListener`

6.5.1 JButton (avec texte et/ou image, hérite de AbstractButton)

Des méthodes de JButton (héritage de AbstractButton) permettent de définir plusieurs icônes suivant l'état du JButton (normal, pressé, survolé, désactivé). Le JButton peut avoir une bordure, et une marque lorsqu'il a le focus. La figure 6.3 présente dans sa partie JButton, un bouton normal (A), un bouton ayant un icône et un texte (B), un bouton ayant seulement un icône (une flèche droite) et en dernier, un bouton désactivé (D).

Figure 6.3 — Les boutons Swing.

```
ActionListener al = new ActionButton();

  JPanel lesJButton () {
    // JButton (hérite de AbstractButton)
    JPanel p = new JPanel (new FlowLayout(FlowLayout.LEFT));
    p.setBorder (BorderFactory.createTitledBorder("JButton"));

    JButton ba = new JButton ("Bouton A");
    ba.setCursor (new Cursor(Cursor.HAND_CURSOR));
    getRootPane().setDefaultButton(ba);          // Default JButton
    p.add (ba);

    JButton bb = new JButton ("Bouton B");
    bb.setMnemonic ('o');                                // Alt + o
    bb.setIcon         (new ImageIcon("bullet1.gif"));  // bleu
    bb.setPressedIcon  (new ImageIcon("bullet2.gif"));  // rouge
    bb.setRolloverIcon (new ImageIcon("bullet3.gif"));  // vert
```

```
  bb.setCursor (new Cursor(Cursor.HAND_CURSOR));
  p.add (bb);

  JButton bc = new JButton (new ImageIcon("flechedroite.gif"));
  bc.setToolTipText ("Bouton avec icône seul");
  p.add (bc);

  JButton bd = new JButton ("Bouton D");
  bd.setEnabled (false);
  p.add (bd);

  return p;
}  // lesJButton
```

6.5.2 JToggleButton (hérite de AbstractButton)

Un JToggleButton est un bouton qui a deux états comme un bouton de boîte à outils et pouvant indiquer une activation ou non d'une option (deux états). Par défaut et en présence de texte pour le JToggleButton, les deux états sont indiqués par un fond permutant blanc ou grisé. Pour les JToggleButton n'ayant qu'un icône (pas de texte), on peut définir un icône pour l'état normal, et un pour l'état sélectionné (héritage de AbstractButton). La figure 6.3 montre dans la partie JToggleButton 1, trois JToggle-Button avec les textes a, b et c, ce dernier ayant en plus un icône "flèche droite".

```
// les JToggleButton avec texte et/ou icône
JPanel lesJToggleButton1 () {
  JPanel p = new JPanel (new FlowLayout(FlowLayout.LEFT));
  p.setBorder (BorderFactory.createTitledBorder("JToggleButton 1"));

  JToggleButton jta = new JToggleButton (" a ", true);
  jta.setBorder (BorderFactory.createRaisedBevelBorder);
  p.add (jta);

  JToggleButton jtb = new JToggleButton (" b ");
  jtb.setBorder(BorderFactory.createRaisedBevelBorder);
  p.add (jtb);

  JToggleButton jtc =
        new JToggleButton (" c ", new ImageIcon("flechedroite.gif"));
  jtc.setSelectedIcon (new ImageIcon ("flechedroite2.gif"));
  jtc.setBorder (BorderFactory.createRaisedBevelBorder);
  p.add (jtc);

  // les écouteurs
  jta.addActionListener (al);
  jtb.addActionListener (al);
  jtc.addActionListener (al);

  return p;
}  // lesJToggleButton1
```

Une manière plus courante d'utiliser les JToggleButton est de définir deux icônes différents pour l'état activé et désactivé. Le fond peut par exemple être plus clair (ou en grisé) lorsque le bouton est non actif.

```
// les JToggleButton avec 2 icônes (une pour chaque état)
JPanel lesJToggleButton2 () {
  JPanel p = new JPanel (new FlowLayout(FlowLayout.LEFT));
  p.setBorder (BorderFactory.createTitledBorder("JToggleButton 2"));

  jt1 = new JToggleButton (new ImageIcon ("home.gif"));
  jt1.setSelectedIcon     (new ImageIcon ("home2.gif"));
  jt1.setBorder (BorderFactory.createRaisedBevelBorder);
  p.add (jt1);

  jt2 = new JToggleButton (new ImageIcon ("flechedroite.gif"));
  jt2.setSelectedIcon     (new ImageIcon ("flechedroite2.gif"));
  jt2.setBorder (BorderFactory.createRaisedBevelBorder);
  p.add (jt2);

  jt3 = new JToggleButton (new ImageIcon ("flechegauche.gif"));
  jt3.setSelectedIcon     (new ImageIcon ("flechegauche2.gif"));
  jt3.setBorder (BorderFactory.createRaisedBevelBorder);
  p.add (jt3);

  // les écouteurs
  jt1.addActionListener (al);
  jt2.addActionListener (al);
  jt3.addActionListener (al);

  return p;
}  // lesJToggleButton2
```

6.5.3 JCheckBox (case à cocher, hérite de JToggleButton)

Chaque case à cocher est indépendante des autres. Sur la figure 6.3, le premier JCheckBox est standard. Le deuxième se compose d'un icône (flèche) et d'une étiquette. Un icône supplémentaire pour l'état sélectionné peut être indiqué (voir AbstractButton).

```
// les JCheckBox : les cases à cocher
JPanel lesJCheckBox() {
  JPanel p = new JPanel (new GridLayout(1, 3));
  p.setBorder (BorderFactory.createTitledBorder ("JCheckBox"));

  JCheckBox cb1 = new JCheckBox ("choix a");
  p.add(cb1);

  JCheckBox cb2 =
       new JCheckBox ("choix b", new ImageIcon("flechedroite2.gif"));
  cb2.setSelectedIcon (new ImageIcon("flechedroite.gif"));
  cb2.setMnemonic ('b'); // Alt + b
  cb2.setCursor (new Cursor(Cursor.HAND_CURSOR));
  p.add (cb2);

  // les écouteurs
  cb1.addActionListener (al);
  cb2.addActionListener (al);

      return p;
}  // lesJCheckBox
```

6.5.4 RadioButton (hérite de JToggleButton)

Les JRadioButton sont présentés par groupe. Un seul élément du groupe à la fois peut être sélectionné. Il faut créer un objet de la classe ButtonGroup, et lui ajouter des JRadioButton. Le JRadioButton peut être précédé d'un icône différenciant ses deux états : sélectionné ou pas.

```
// les boutons radio
JPanel lesJRadioButton () {
  JPanel p = new JPanel (new GridLayout (1,3));
  p.setBorder (BorderFactory.createTitledBorder("JRadioButton"));

  ButtonGroup  bg  = new ButtonGroup();  // pas un Component
  JRadioButton rb1 = new JRadioButton ("bouton 1");
  bg.add (rb1);
  p.add  (rb1);
  JRadioButton rb2 = new JRadioButton ("bouton 2");
  bg.add (rb2);
  p.add  (rb2);
  JRadioButton rb3 = new JRadioButton ("bouton 3");
  rb3.setIcon         (new ImageIcon  ("home.gif"));
  rb3.setSelectedIcon (new ImageIcon  ("home2.gif"));
  bg.add (rb3);
  p.add  (rb3);

  // les écouteurs
  rb1.addActionListener (al);
  rb2.addActionListener (al);
  rb3.addActionListener (al);

  return p;
}  // lesJRadioButton
```

Exercice 6.1 – Les boutons

> À partir des méthodes données ci-dessus, compléter le programme qui permet d'obtenir les différents boutons de la.

6.5.5 JMenuItem (hérite de AbstractButton)

Un JMenuItem hérite de AbstractButton (voir page 241). Il hérite donc de toutes les caractéristiques d'un AbstractButton avec donc la possibilité d'afficher du texte et des icônes. Il est utilisé comme élément de base dans les menus. Il peut avoir un écouteur de type ActionListener.

6.6 LES MENUS DÉROULANTS

Les menus sous Swing sont assez proches des menus de AWT avec quelques spécificités supplémentaires. Un JMenuItem hérite de AbstractButton : on peut donc lui ajouter un icône. Voir les menus déroulants sous AWT page 168.

6.6.1 JMenuBar (hérite de JComponent)

Un JMenuBar définit la barre de menus à ajouter à une fenêtre.

```
new JMenuBar()                          créer une barre de menus
setJMenuBar (JMenuBar m)       ajouter une barre de menus à une fenêtre
add (JMenu m)                      ajouter un menu à une barre de menus
```

6.6.2 JMenu (hérite de JMenuItem)

JComponent
AbstractButton
JMenuItem
JMenu
JCheckBoxMenuItem

Un objet de type JMenu peut se subtituer à un objet de type JMenuItem. Un objet de type JMenu contient des JMenuItem. Certains d'entre eux peuvent être des JMenu. Un JMenu constitue une option de la barre de menus qui s'ouvre lors d'un clic de souris. add (JMenuItem) est une méthode de JMenu.

```
addSeparator()         ajoute une barre de séparation entre 2 JMenuItem
add (JMenuItem m)   ajouter un JMenuItem (ou un autre JMenu) à un JMenu
```

6.6.3 JMenuItem (hérite de AbstractButton)

En tant qu'AbstractButton, JMenuItem peut contenir un texte ou/et un icône, et avoir un raccourci mnémonique. On peut lui ajouter un écouteur de type ActionListener.

```
JMenuItem (String s)                 créer un élément de base d'un menu
```

6.6.4 JCheckBoxMenuItem

C'est un JMenuItem avec une case à cocher. Son état est obtenu à l'aide de la méthode isSelected(), ou imposé à l'aide de setSelected().

Figure 6.4 — JMenuBar, JMenu, JMenuItem, JCheckBoxMenuItem.

Le programme suivant effectue la construction des menus de la figure 6.4.

```
// APIActionJMenuBar.java
// mise en oeuvre d'une barre de menus

import java.awt.*;           // Color, MenuShortcut
import javax.swing.*;        // JFrame
import java.awt.event.*;     // ActionListener

// test des actions d'une barre de menus
class APIActionJMenuBar extends JFrame {
  JMenuBar   barre1;
  JMenuBar   barre2;  // on change de barre en cours de route
  JTextArea  ta;
  ActionListener actionMenu = new ActionMenu ();

  APIActionJMenuBar () {
    setTitle  ("APIActionJMenuBar");
    setBounds (10, 10, 400, 400);

    // créer et afficher la barre de menus 1
    barre1 = creerMenuBar1();
    setJMenuBar (barre1);
    // créer la barre de menu 2
    barre2 = creerMenuBar2();

    ta = new JTextArea ("");
    ta.setBackground (Color.cyan);
    getContentPane.add (ta, "Center");

    setVisible (true);
  } // APIActionJMenuBar
```

La méthode creerMenuBar1() crée la barre de menus (JMenuBar) de la figure 6.4. Cette barre contient deux JMenu ("menuA 1" et "menuA 2"). Le JMenu "menuA 1" contient trois JMenuItem. Le JMenuItem "sous-menuA 11" contient un icône et il a une bulle d'aide (c'est un AbstractButton). Le JMenuItem "sous-menuA 12" est invalidé (en grisé). Le JMenuItem "menuA 13" est un JMenu et conduit à deux nouveaux JMenuItem ("sous-menuA 131" et " sous-menuA 132"). Le premier de ces sous-menus contient une case à cocher (JCheckBoxMenuItem).

```
  JMenuBar creerMenuBar1 () {
    JMenuBar barre = new JMenuBar();

    // menu A, colonne 1
    // item 1
    JMenu menuA1 = new JMenu ("menuA 1");
    barre.add (menuA1);
    JMenuItem mA11 = new JMenuItem ("sous-menuA 11");
    mA11.setIcon (new ImageIcon("md12.gif"));
    mA11.setToolTipText ("menu item a11");
    mA11.setBackground (Color.RED); // quand non sélectionné
    menuA1.add (mA11);
    // item 2
    JMenuItem mA12 = new JMenuItem ("sous-menuA 12");
    mA12.setEnabled(false);
```

```
  menuA1.add (mA12);
  menuA1.addSeparator();
  // item 3
  JMenu mA13 = new JMenu ("menuA 13");
  menuA1.add (mA13);
  JMenuItem mA131 = new JCheckBoxMenuItem ("sous-menuA 131");
  JMenuItem mA132 = new JMenuItem         ("sous-menuA 132");
  mA13.add (mA131);
  mA13.add (mA132);

  // menu A, colonne 2
  JMenu menuA2 = new JMenu ("menuA 2");
  barre.add (menuA2);
  JMenuItem mA21 = new JMenuItem ("sous-menuA 21");
  JMenuItem mA22 = new JMenuItem ("sous-menuA 22");
  menuA2.add (mA21);
  menuA2.add (mA22);

  // un écouteur actionMenu pour les actions des JmenuItem
  mA11.addActionListener  (actionMenu);
  mA12.addActionListener  (actionMenu);
  mA21.addActionListener  (actionMenu);
  mA22.addActionListener  (actionMenu);
  mA131.addActionListener (actionMenu);
  mA132.addActionListener (actionMenu);
  // chaînes à fournir lors de l'activation du menu
  mA11.setActionCommand   ("11");
  mA12.setActionCommand   ("12");
  mA21.setActionCommand   ("21");
  mA22.setActionCommand   ("22");
  mA131.setActionCommand  ("131");
  mA132.setActionCommand  ("132");

  return barre;
}  // creerMenuBar1
```

La méthode creerMenuBar2() crée une deuxième barre de menus. Une seule barre de menus peut être affichée à la fois. Cependant, on peut changer de barre de menus en cours d'exécution du programme.

```
JMenuBar creerMenuBar2 () {
  JMenuBar barre = new JMenuBar();

  // menu B, colonne 1
  JMenu     menuB1 = new JMenu     ("menuB 1");
  JMenuItem mB11   = new JMenuItem ("sous-menuB 11");
  JMenuItem mB12   = new JMenuItem ("sous-menuB 12");
  menuB1.add (mB11);
  menuB1.add (mB12);
  barre.add  (menuB1);

  // menu B, colonne 2
  JMenu     menuB2 = new JMenu     ("menuB 2");
  JMenuItem mB21   = new JMenuItem ("sous-menuB 21");
  JMenuItem mB22   = new JMenuItem ("sous-menuB 22");
```

```
    menuB2.add (mB21);
    menuB2.add (mB22);
    barre.add  (menuB2);

   // pour les JMenuItem seulement
    mB11.addActionListener  (actionMenu);
    mB12.addActionListener  (actionMenu);
    mB21.addActionListener  (actionMenu);
    mB22.addActionListener  (actionMenu);
    return barre;
  }
```

Les actions des JMenuItem consistent (ci-dessous à titre de test) à écrire les caractéristiques de l'action dans le JTextArea, et à changer à chaque sélection de barre de menus.

```
  // classe interne pour les actions
  // action pour les menus de barre1 et barre2
  class ActionMenu implements ActionListener {
    boolean barre1Used = true;

    public void actionPerformed (ActionEvent evt) {
      JMenuItem ch = (JMenuItem) evt.getSource();
      ta.append ("\n ActionMenu : " + ch.getText() + " " +
                 ch.getActionCommand());
      if (!barre1Used) {           // change toute la barre de menus
        setJMenuBar (barre1);
      } else {
        setJMenuBar (barre2);
      }
      barre1Used = ! barre1Used;
   }
  }

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

}  // APIActionJMenuBar
```

6.7 LES CONTAINERS TYPE PANNEAU

Swing définit plusieurs types de conteneurs auxquels on peut ajouter des composants. Le panneau de base est le JPanel. D'autres panneaux plus complexes existent : le nom se termine par Pane comme dans (JScroll, JTabbed, JSplit, JDesktop) Pane.

6.7.1 JPanel (dérive de JComponent)

Un JPanel est un conteneur qui peut recevoir d'autres JComponent. Un JPanel hérite de JComponent : on peut donc lui ajouter une bordure, changer sa couleur de fond. Par défaut, le JPanel est opaque. Il peut être transparent et laisser apercevoir ce qu'il

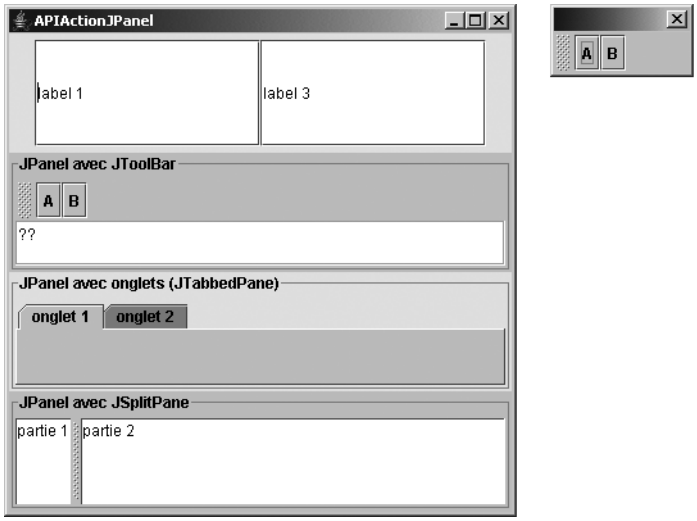


Figure 6.5 — Différents Panels. Un JToolBar peut être déplacé en une fenêtre indépendante.

y a en dessous de lui. Par défaut, la mise en page est de type FlowLayout. On peut connaître le nombre de JComponent ajoutés à un JPanel. On peut enlever des composants à partir de leur référence ou à partir de leur numéro dans le JPanel. Voir figures 6.1 et 6.5.

```
JPanel lesJComponent () {
  JPanel p = new JPanel(new GridLayout (0,2));
  p.setBorder(BorderFactory.createEmptyBorder(5,20,5,20)); // marges
  System.out.println (p.getComponentCount());
  p.add (new JTextField ("label 1"));
  JLabel la = new JLabel ("label 2");
  p.add (la);
  p.add (new JTextField ("label 3"));
  System.out.println ("\nnb JComponent " + p.getComponentCount());
  System.out.println ("\ncomponent 1   " + p.getComponent(1));
  //p.remove(1);  // le deuxième composant
  p.remove(la);
  //p.removeAll;
  System.out.println ("\nnb JComponent " + p.getComponentCount());
  System.out.println ("\ncomponent 1   " + p.getComponent(1));

  p.setBackground (Color.cyan);
  return p;
}
```

6.7.2 JToolBar (hérite de JComponent)

Un JToolBar est un composant qui permet d'afficher des icônes représentant des actions souvent réalisées. Un JToolBar peut être déplacé à l'aide de la souris et devenir une fenêtre indépendante. Voir figures 6.1 et 6.5.

```
JPanel lesJToolBar () {
  JPanel p = new JPanel(new BorderLayout());  // BorderLayout
  p.setBorder (BorderFactory.createTitledBorder
                                          ("JPanel avec JToolBar"));
  JTextArea ta = new JTextArea ("??");
  ta.setBorder(BorderFactory.createEtchedBorder());
  p.add (ta, BorderLayout.CENTER);

  JToolBar jtb = new JToolBar();
  p.add (jtb, BorderLayout.NORTH);
  JButton bouton = new JButton ("A");
  jtb.add (bouton);
  bouton = new JButton ("B");
  jtb.add (bouton);
  return p;
}
```

6.7.3 JScrollPane (hérite de JComponent)

On peut ajouter un JComponent dans un JScrollPane si le composant risque d'être trop grand pour l'espace qui lui est imparti. Si le composant n'est pas entièrement visible, des ascenseurs apparaissent qui permettent de faire défiler la vue du composant. Plusieurs composants peuvent être ajoutés à un JPanel qui lui est ajouté au JScrollPane. Un JScrollPane ne gère qu'un seul composant.

6.7.4 JTabbedPane (hérite de JComponent)

Le panneau JTabbedPane permet la définition de plusieurs panneaux sélectionnables à l'aide d'onglets. On peut aussi indiquer la position des onglets dans le panneau à onglets (par défaut, en haut). addTab (String titre, Component composant) ajoute un onglet avec le titre et le composant indiqués. Voir figure 6.5.

```
JPanel lesJTabbedPane () {
  JPanel p = new JPanel(new BorderLayout()); // BorderLayout
  p.setBorder (BorderFactory.createTitledBorder
                            ("JPanel avec onglets (JTabbedPane)"));
  JTabbedPane tb = new JTabbedPane();
  p.add (tb, BorderLayout.CENTER);
  tb.addTab ("onglet 1", null);       // null : onglet sans composant
  tb.addTab ("onglet 2", null);
  p.setBackground (Color.cyan);
  return p;
}
```

6.7.5 SplitPane (hérite de JComponent)

Un JSplitPane correspond à un double panneau avec une cloison mobile entre les deux panneaux. La séparation peut se faire suivant l'axe horizontal ou vertical. Cette séparation peut être déplacée à l'aide de la souris.

```
JPanel lesJSplitPane () {
  JPanel p = new JPanel (new BorderLayout());  // BorderLayout
  p.setBorder (BorderFactory.createTitledBorder
                                        ("JPanel avec JSplitPane"));
  JSplitPane sp = new JSplitPane (JSplitPane.HORIZONTAL_SPLIT,
              new JTextArea("partie 1"), new JTextArea ("partie 2"));
  p.add (sp, BorderLayout.CENTER);
  return p;
}
```

6.7.6 Le gestionnaire de répartition BoxLayout

Le gestionnaire de mise en page BoxLayout de Swing permet de répartir les composants horizontalement sur une ligne, ou verticalement sur une colonne. On peut ajouter entre les composants des composants de gestion d'espace de taille fixe ou de taille variable (glue).

La figure 6.6 est gérée au niveau du **JFrame** par un **BoxLayout vertical** répartissant deux composants Panel (p1 et p2) séparés par un espace fixe de 10 pixels défini par Box.createVerticalStrut(10). Les étiquettes et les boutons gardent leur taille si la taille de la fenêtre est modifiée. Un JTextField par contre s'adapte en largeur et en hauteur.

Le **Panel p1** géré par un **BoxLayout vertical** contient verticalement deux JLabel, un Motif, un espace de 10 pixels et un JTextArea. Le **Panel p2** géré par un **BoxLayout horizontal** contient horizontalement un espace de taille variable (glue), un JLabel, un espace de 10 pixels, un JLabel et un espace de 10 pixels. L'agrandissement de la fenêtre sur la figure 6.6 a gardé le même aspect pour le panel p2 : la partie espace variable avant JLabel3 et JLabel4 a été augmentée horizontalement.

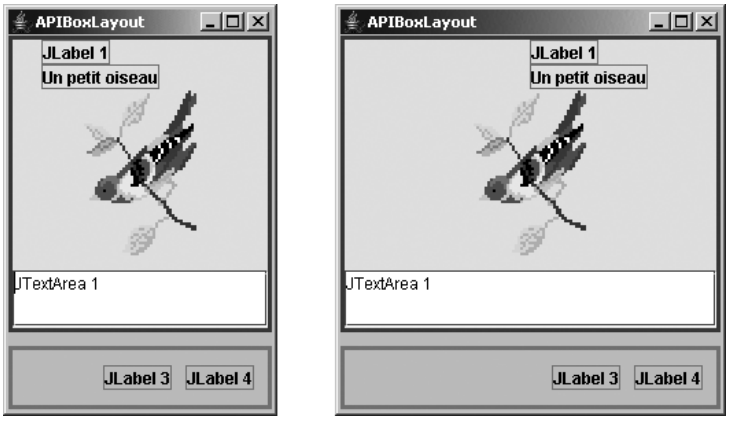


Figure 6.6 — Mise en page avec un BoxLayout.

Le programme permettant de tester le gestionnaire de mise en page de type BoxLayout est donné ci-dessous.

```
// APIBoxLayout.java

import java.awt.*;           // Color
import javax.swing.*;        // JFrame
import mdpaquetage.mdawt.*;  // Motif et MotifLib

class APIBoxLayout extends JFrame {

  APIBoxLayout () {
    setTitle  ("APIBoxLayout");
    setBounds (10, 10, 200, 300);
    setBackground (Color.lightGray);
    Container f = (Container) getContentPane();

    BoxLayout bo = new BoxLayout (f, BoxLayout.Y_AXIS) ;
    f.setLayout (bo);

    // Panel p1 : une colonne de composants
    JPanel p1 = new JPanel();
    f.add (p1);
    p1.setLayout (new BoxLayout (p1, BoxLayout.Y_AXIS));
    p1.setBorder (BorderFactory.createLineBorder(Color.blue, 3));
    p1.setBackground (Color.cyan);

    // les composants de p1 (verticalement)
    // JLabel + JLabel + Motif + 10 pixels
                                          + JTextArea dans JScrollPane
    JLabel la1 = new JLabel ("JLabel 1");
    JLabel la2 = new JLabel ("Un petit oiseau");
    Motif  m3  = new Motif  (MotifLib.oiseau, Motif.tProp,
                             MotifLib.palette1Oiseau);
    JTextArea ta = new JTextArea ("JTextArea 1");
    la1.setBorder (BorderFactory.createLineBorder(Color.red));
    la2.setBorder (BorderFactory.createLineBorder(Color.red));
    p1.add (la1);
    p1.add (la2);
    p1.add (m3);
    p1.add (Box.createVerticalStrut(10));  // espace de 10 pixels
    p1.add (new JScrollPane(ta));

    f.add (Box.createVerticalStrut(10));   // espace de 10 pixels

    // Panel p2 : une ligne de composants
    JPanel p2 = new JPanel ();
    f.add (p2);
    p2.setLayout (new BoxLayout (p2, BoxLayout.X_AXIS));
    p2.setBorder (BorderFactory.createLineBorder(Color.red,3));

    // les composants de p2 (horizontalement)
    // glue + JLabel 3 + 10 pixels + JLabel 4 + 10 pixels
    JLabel la3 = new JLabel ("JLabel 3");
    JLabel la4 = new JLabel ("JLabel 4");
    la3.setBorder (BorderFactory.createLineBorder(Color.red));
    la4.setBorder (BorderFactory.createLineBorder(Color.red));
    p2.add (Box.createHorizontalGlue());
    p2.add (la3);
    p2.add (Box.createHorizontalStrut(10));
```

```
        p2.add (la4);
        p2.add (Box.createHorizontalStrut(10));

        setVisible (true);
      } // APIBoxLayout

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

    }  // APIActionJPanel
```

6.8 LES CONTAINERS DE TYPE FENÊTRE

Les Container de type fenêtre de Swing (JFrame, JDialog, JWindow) sont très proches de ceux de AWT. Ils dérivent directement de leur homologue en AWT (Frame, Dialog, Window). Voir fenêtres AWT page 218 et figure 6.1.

6.8.1 JFrame (hérite de Frame)

Les JComponent sont ajoutés au "content pane" du JFrame, et non directement au frame comme pour AWT, un JFrame se composant en fait de trois panneaux. L'ajout de composants et le Layout se réfèrent toujours au ContentPane d'un JFrame. On peut utiliser les instructions suivantes par exemple :

```
JFrame frame = new JFrame();  // BorderLayout
Container f = frame.getContentPane();
f.add (new JLabel("label 1"));
```

Le comportement lors du clic sur la case de fermeture est aussi différent de AWT qui sans écouteur ne fait rien. Sous Swing, par défaut, la fenêtre se ferme, sans fermer l'application (`HIDE_ON_CLOSE`). On peut aussi demander les actions suivantes :

```
setDefaultCloseOperation (WindowConstants.DO_NOTHING_ON_CLOSE);
                                                        // idem AWT
setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
                                              // fin de l'application
```

Figure 6.7 — Fenêtres de type JFrame et JDialog.

```
// APIJFrame.java  test des fenêtres de Swing

import java.awt.*;           // Container
import javax.swing.*;        // JFrame

class APIJFrame extends JFrame {
```

```
    APIJFrame () {
      setTitle  ("APIJFrame");
      setBounds (10, 10, 200, 100);
      setBackground (Color.lightGray);
      Container f = getContentPane();
      f.setLayout (new BorderLayout());
      f.add (new JLabel (" fenêtre de type JFrame"));
      setVisible (true);

      JDialog d1 = new JDialog (this, "JDialog", false);
      d1.setBounds (10, 150, 200, 100);
      Container cd = d1.getContentPane();
      cd.setBackground (Color.cyan);
      cd.add (new JLabel (" fenêtre de type JDialog"));
      d1.setVisible (true);

      JWindow w1 = new JWindow (this);
      w1.setBounds (10, 300, 200, 100);
      Container cw = w1.getContentPane();
      cw.setBackground (Color.cyan);
      cw.add (new JLabel (" fenêtre de type JWindow"));
      w1.setVisible (true);

    } // constructeur APIJFrame

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

  }  // APIJFrame
```

6.8.2 JDialog (hérite de Dialog)

Une fenêtre JDialog est dépendante d'une fenêtre JFrame (ou d'une autre fenêtre JDialog). Lorsque cette dernière est détruite, la fenêtre JDialog l'est également. Voir page 218.

6.8.3 JWindow (hérite de Window)

Une fenêtre de type JWindow est une fenêtre sans bandeau, non déplaçable, non redimensionnable et sans case de fermeture. Voir page 218.

6.9 LES COMPOSANTS TEXT (JTEXTCOMPONENT)

JTextComponent est la classe de base pour les composants texte Swing. Voir figure 6.1.

6.9.1 JTextField, JTextArea, JEditorPane

Un JTextField permet l'édition d'une ligne de texte. Un JTextArea permet l'édition de plusieurs lignes de texte sans mise en page. Un JEditorPane permet de visualiser du texte sans mise en page (comme un JTextArea), un texte mis en page au format HTML ou au format RTF.

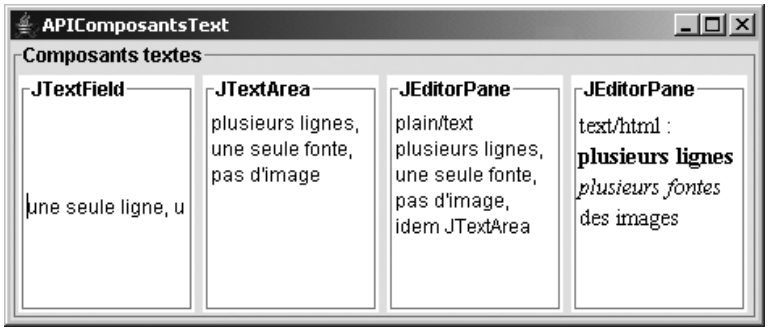


Figure 6.8 — Les composants de type texte en Swing.

```
// APIComposantsText.java

import java.awt.*;           // Color
import javax.swing.*;        // JFrame
import javax.swing.event.*;

class APIComposantsText extends JFrame {

  APIComposantsText () {
    setTitle  ("APIComposantsText");
    setBounds (10, 10, 450, 200);
    setBackground (Color.lightGray);
    Container frame = getContentPane();
    frame.setLayout (new BorderLayout());

    JPanel p1 = new JPanel (new GridLayout(1,4,5,5));
    frame.add (p1, "Center");
    p1.setBackground (Color.cyan);
    p1.setBorder(BorderFactory.createTitledBorder
                                              ("Composants textes"));

    JTextField tf = new JTextField
                    ("une seule ligne, une seule fonte, pas d'image");
    tf.setBorder (BorderFactory.createTitledBorder("JTextField"));
    p1.add (tf);

    JTextArea ta = new JTextArea
               ("plusieurs lignes, \nune seule fonte, \npas d'image");
    ta.setBorder (BorderFactory.createTitledBorder("JTextArea"));
    ta.setLineWrap (true);
    p1.add (ta);

    JEditorPane te1 = new JEditorPane ("plain/text",
      "plain/text \nplusieurs lignes, \nune seule fonte,
                              \npas d'image, " + "\nidem JTextArea");
    p1.add (te1);
    te1.setBorder (BorderFactory.createTitledBorder("JEditorPane"));

    JEditorPane te2 = new JEditorPane ("text/html",
      "text/html : <br > < b > plusieurs lignes </b > < br > " +
      "<i > plusieurs fontes </i > < br > " + "des images < br > " );
```

```
    p1.add (te2);
    te2.setBorder (BorderFactory.createTitledBorder("JEditorPane"));
    setVisible (true);

    nouveauFrame();              // créer le composant de la figure 6.9

} // APIComposantsText
```

Figure 6.9 — Composants texte de classe JEditorPane.

addHyperlinkListener() définit un écouteur sur le composant de classe JEditorPane(). Un clic sur un hyperlien provoque un événement pris en compte par la méthode hyperlinkUpdate(). Sur la figure 6.9, on passe d'une image à l'autre en cliquant sur les hyperliens.

```
void nouveauFrame () {
  JFrame frame = new JFrame ();
  frame.setTitle  ("JEditorPane");
  frame.setBounds (10, 300, 100, 250);
  frame.setBackground (Color.lightGray);
  Container f = frame.getContentPane();
  JEditorPane te;
  try {
    te = new JEditorPane ("file : nepal.html");
    f.add (te);
    te.setBorder (BorderFactory.createTitledBorder
                                          ("avec fichier html"));
    te.setEditable (false);     // sinon écouteur Hyperlien inopérant
    te.addHyperlinkListener (new HyperLien());
  } catch (Exception e) {
    System.out.println (e);
  }
  frame.setVisible (true);
}
```

```
    // classe interne
    class HyperLien implements HyperlinkListener {
      public void hyperlinkUpdate (HyperlinkEvent evt) {
        System.out.println("hyperlinkUpdate " + evt.getURL);
        JEditorPane t = (JEditorPane) (evt.getSource());
        if (evt.getEventType == HyperlinkEvent.EventType.ACTIVATED) {
          try {
            t.setPage (evt.getURL());
          } catch (Exception e) {
            e.printStackTrace();
          }
        }
      }
    }

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

}  // APIComposantsText
```

6.10 LA CLASSE JLIST (HÉRITE DE JCOMPONENT)

Une JList est un composant qui permet la sélection de un ou plusieurs éléments d'une liste. La sélection se fait avec la souris et avec les touches Maj et Ctrl qui permettent d'étendre la sélection, ou de faire des sélections non contigües. Les éléments de la liste sont des objets. Le texte affiché est alors celui fourni par la méthode toString() de cet objet. Dans le cas le plus simple, l'objet peut être une chaîne de caractères (String). En cas de sélection multiple, il vaut mieux définir un bouton valider qui est actionné en fin de sélection. Comme tout JComponent, la JList peut avoir une bordure ou une bulle d'aide. Voir figure 6.1.

6.10.1 Exemple simple avec des String

Le JPanel de gauche de la figure 6.10 contient deux JList à sélection unique. La première affiche une bulle d'aide "choix unique". La deuxième JList est dans un JScrollPane. Les deux JList sont créées à partir d'un tableau de String, ce qui constitue un cas de base.

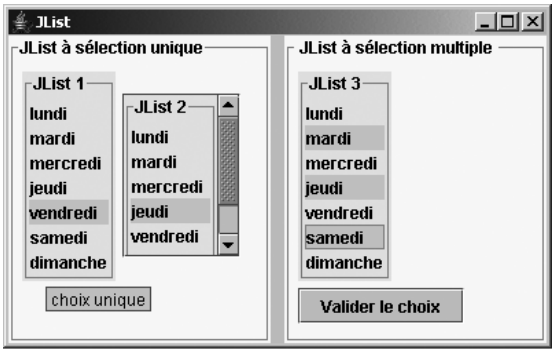


Figure 6.10 — Les JList.

```
String[] jour = { "lundi", "mardi", "mercredi", "jeudi",
                  "vendredi", "samedi", "dimanche" };

JPanel JlistChoixUnique() {
  // JList à sélection unique (par défaut, sélection multiple)
  // créée à partir d'un tableau de String
  JPanel p = new JPanel (new FlowLayout(FlowLayout.LEFT));
  p.setBackground (Color.yellow);
  p.setBorder (BorderFactory.createTitledBorder
                                    ("JList à sélection unique"));

  JList ja = new JList (jour);
  ja.setSelectedIndex(1);  // de 0 à n-1
  ja.setSelectionMode (ListSelectionModel.SINGLE_SELECTION);
  ja.setBackground  (Color.cyan);
  ja.setToolTipText ("choix unique");
  ja.setBorder (BorderFactory.createTitledBorder ("JList 1"));
  ja.addListSelectionListener (new ChoixSimpleListe());
  p.add (ja);

  // JList à sélection unique dans un JScrollPane
  JList jb = new JList (jour);
  jb.setSelectionMode (ListSelectionModel.SINGLE_SELECTION);
  jb.setBackground (Color.cyan);
  jb.setVisibleRowCount (5);
  JScrollPane js = new JScrollPane (jb);
  jb.setBorder (BorderFactory.createTitledBorder ("JList 2"));
  p.add (js);

  return p;
}
```

L'écouteur **ChoixSimpleListe** de la JList en sélection simple est donné ci-dessous. La méthode getSelectedIndex() fournit le rang de l'élément de la liste choisi. La méthode getSelectedValue() fournit la référence de l'objet sélectionné dans la liste.

```
// classe interne
// choix à sélection simple
class ChoixSimpleListe implements ListSelectionListener {
  public void valueChanged (ListSelectionEvent evt) {
   JList jl = (JList) evt.getSource();
   System.out.println ("\n\n *** " + jl.getSelectedIndex());
                                                        // indice
   System.out.println (" *** " + jl.getSelectedValue());
                                                    // le texte
  }
}
```

Le JPanel de droite de la figure 6.10 contient une JList à sélection multiple, et un bouton de validation en fin de choix multiple.

```
JPanel JlistChoixMultiple() {
  // JList à sélection multiple par défaut + JButton Valider
  // en utilisant les touches MAJ (plages de valeurs)
```

```
    // ou CTRL (valeurs isolées)
    JPanel p = new JPanel (new FlowLayout(FlowLayout.LEFT));
    p.setBackground (Color.yellow);
    p.setBorder (BorderFactory.createTitledBorder
                                    (" JList à sélection multiple "));

    JList jlm = new JList (jour);
    jlm.setSelectedIndex (1);  // de 0 à n-1
    jlm.setBackground (Color.cyan);
    jlm.setBorder (BorderFactory.createTitledBorder ("JList 3"));
    p.add (jlm);

    JButton ba = new JButton ("Valider le choix");
    ba.addActionListener (new ValiderChoix(jlm));
    p.add (ba);

    return p;
  }
```

L'écouteur du bouton **ValiderChoix** passe en paramètre la JList qu'il traite. Le constructeur de ValiderChoix mémorise cette JList. La méthode **getSelectedIndices()** fournit un tableau d'entiers des rangs des objets sélectionnés. La méthode **getSelectedValues()** fournit un tableau des références des objets sélectionnés.

```
// écouteur du JButton Valider pour le choix à sélection multiple
class ValiderChoix implements ActionListener {
  JList jlm;

  ValiderChoix (JList jlm) {            // constructeur de l'écouteur
    this.jlm = jlm;
  }

  public void actionPerformed (ActionEvent evt) {
    // le tableau des indices des éléments sélectionnés
    int[] tabselect = jlm.getSelectedIndices();
    for (int i=0; i < tabselect.length; i++) {
      System.out.println ("i : " + i + ", " + tabselect[i]);
    }
    // le tableau des références des objets sélectionnés
    Object[] tabselectv = jlm.getSelectedValues();
    for (int i=0; i < tabselect.length; i++) {
      System.out.println ("i : " + i + " : " + tabselectv[i]);
    }
  }
}
```

Exercice 6.2 – JList des jours de la semaine

Faire le programme permettant d'obtenir le composant de la figure 6.10.

6.10.2 Les classes Etudiant et LesEtudiants

Au lieu de créer la liste à partir d'un tableau des String à afficher dans la JList (tableau des noms des jours de la semaine dans l'exemple précédent), on peut passer en paramètre du constructeur de la JList, un tableau de références sur des objets, ou un Vector d'objets. La méthode toString() est automatiquement appelée et la JList est constituée à partir des informations délivrées par toString() de l'objet.

On utilise des objets Etudiant au lieu d'objets de type String comme dans l'exemple précédent. Les classes Etudiant et LesEtudiants sont définies comme suit :

```
// un étudiant
public class Etudiant {
  public Etudiant (String nom,     String prenom,   String groupe,
                   String sousgrp, boolean boursier);
  public String  toString();

  // accesseurs
  public String  getnom();
  public String  getprenom();
  public String  getgroupe();
  public String  getsousgrp();   // sous-groupe
  public boolean getboursier();

  public void setnom      (String  nom);
  public void setprenom   (String  prenom);
  public void setgroupe   (String  groupe);
  public void setsousgrp  (String  sousgrp);
  public void setboursier (boolean boursier);

}  // class Etudiant

// un ensemble d'étudiants
public class LesEtudiants {
  public LesEtudiants();                  // constructeur des étudiants
  public int      nbEtudiants    ();              // nombre d'étudiants
  public boolean  existe         (int n);   // l'étudiant n existe t-il
  public Etudiant getEtudiant    (int n);
  public void     addEtudiant    (Etudiant etudiant);
  public void     removeEtudiant (int n);
}
```

6.10.3 Le ListModel de la JList Etudiant

On peut construire la JList des étudiants à partir d'un tableau ou d'un Vector d'Etudiant. Une autre façon encore plus générale est de confier à des méthodes **prédéfinie** d'un objet de type **ListModel**, la fourniture des informations et des objets à insérer dans la JList. C'est le rôle de la classe ListeModelEtudiants ci-dessous. La méthode getSize() fournit le nombre d'étudiants à insérer dans la liste. La méthode getElementAt (int n) fournit l'objet n de la JList. On fait complètement abstraction de la provenance des données (un tableau d'étudiants, une base de données étudiants, etc.).

```
// ListModelEtudiant.java

package mdpaquetage.etudiant;

import javax.swing.*;  // AbstractListModel

// ListModel qui fournit les données de la JList
public class ListModelEtudiant extends AbstractListModel {
   LesEtudiants lesEtudiants;
   boolean      editable = false;

   public ListModelEtudiant (LesEtudiants lesEtudiants) {
     this.lesEtudiants = lesEtudiants;
   }

    public int getSize() {
      return lesEtudiants.nbEtudiants();
    }

    public Object getElementAt (int n) {
      return lesEtudiants.getEtudiant(n);
    }

 }  // ListModelEtudiant
```

6.10.4 Le Renderer (des cellules) de la JList Etudiant

De même, la présentation de chaque élément de la JList peut être confiée à des méthodes **prédéfinie** d'un objet de mise en forme des éléments appelé "**Renderer**" La méthode getListCellRendererComponent() est appelée pour afficher chaque élément (cellule) de la JList composée d'objets "Etudiant".

```
// RendererJListEtudiant.java  // JList

package mdpaquetage.etudiant;

import java.awt.*;          // Container, BorderLayout
import javax.swing.*;       // JPanel, JFrame, JLabel

// RENDERER de JList

// classe interne de présentation des éléments de la JList;
// la méthode est appelée pour chaque élément de JList.
// chaque élément de la JList est rendu à l'aide d'un JLabel.
// ce pourrait être un autre JComponent (standard ou maison).
class RendererJList1 extends JLabel implements ListCellRenderer {
  // la méthode retourne ici un JLabel pour chaque élément à afficher
  static final ImageIcon bullet = new ImageIcon ("bullet.gif");

  public Component getListCellRendererComponent(
      JList list,
      Object value,          // objet à afficher (Etudiant)
      int index,             // indice de l'élément
      boolean isSelected,    // état sélectionné ?
      boolean cellHasFocus)  // la JList a-t-elle le focus ?
    {
```

```
    JLabel   eti = this;  // héritage
    Etudiant etu = (Etudiant) value;
    String   s   = etu.getnom() + " " + etu.getprenom();
    if (isSelected) {
      eti.setText (s);
      eti.setIcon (bullet);
      setBackground (list.getSelectionBackground());
      setForeground (Color.red);
    } else {
      eti.setText ("    " + s);
      eti.setIcon (null);  // enlever l'icône s'il y en avait un
      setBackground (list.getBackground());
      setForeground (list.getForeground());
    }
    eti.setEnabled (list.isEnabled());
    eti.setFont (list.getFont());
    eti.setOpaque (true);  // on veut voir le fond du JLabel
    return eti;  // on pourrait retourner un autre composant
  }  // getListCellRendererComponent

}  // RendererJList1

public class RendererJListEtudiant {
  // mise en page de la JList
  public RendererJListEtudiant (JList jlist) {
    // le renderer s'applique aux éléments de la JList
    RendererJList1 rd1 = new RendererJList1();
    jlist.setCellRenderer (rd1);
  }

}  // RendererJListEtudiant
```

6.10.5 Utilisation du ListModel et du Renderer

La figure 6.11 représente l'interface graphique de choix d'un ou de plusieurs étudiants dans une JList avec une présentation standard à gauche et personnalisée à droite.

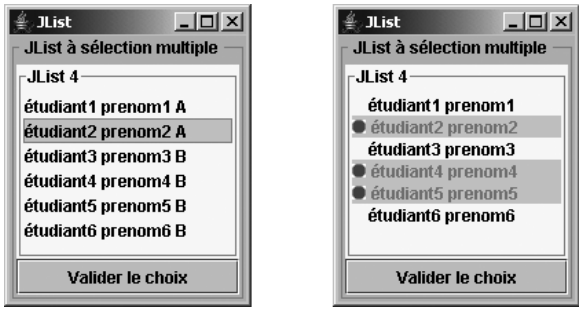


Figure 6.11 — JList d'objets Etudiant avec le Renderer standard à gauche, et un Renderer spécifique à droite (avec icône et couleurs personnalisés).

Lors de l'appui sur le bouton Valider, le programme suivant écrit sur la sortie standard, les numéros et libellés des éléments sélectionnés. Le programme utilise un ListModel pour obtenir les données à insérer dans la JList. Il utilise un Renderer pour présenter la liste : les éléments sélectionnés sont précédés d'un icône (une image) et les couleurs de fond et de premier plan sont changées.

```
i : 0 : 1                          indices et libelles des sélectionnés
i : 1 : 3                                    voir figure 6.11 de droite
i : 2 : 4
i : 0 : étudiant2 prenom2 A
i : 1 : étudiant4 prenom4 B
i : 2 : étudiant5 prenom5 B
```

```
// APISwing4.java  Programme Principal de Swing

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

class APISwing4 extends JFrame {

  // constructeur
  APISwing4 () {
    setTitle ("JList");
    Container f = (Container) getContentPane();
    f.setLayout (new GridLayout (0,1));
    setBounds (10, 10, 180, 220);
    f.add (panelJList());
    setVisible (true);
  }  // constructeur

  JPanel panelJList () {
    // JList à sélection multiple par défaut
    // utiliser les touches MAJ (plages de valeurs)
    // ou CTRL (valeurs isolées)
    JPanel p = new JPanel(new BorderLayout());
    p.setBackground (Color.lightGray);
    p.setBorder(BorderFactory.createTitledBorder
                                    (" JList à sélection multiple "));

    // JList créée en utilisant un ListModel (un objet qui fournit
    // les données dont le libellé doit être mis dans la JList)
    // objet listModelEtudiant fournissant les données de la liste
    LesEtudiants      lesEtudiants = new LesEtudiants();
    ListModelEtudiant listModelEtudiant;
    listModelEtudiant = new ListModelEtudiant (lesEtudiants);
    JList jlm = new JList (listModelEtudiant);

    // mise en page de la JList jlm avec un Renderer
```

```
    // ligne suivante en commentaire = > renderer standard
    RendererJListEtudiant rd = new RendererJListEtudiant (jlm);

    jlm.setSelectedIndex (1);  // de 0 à n-1
    jlm.setBackground (Color.yellow);
    jlm.setBorder (BorderFactory.createTitledBorder ("JList 4"));
    p.add (jlm);

    JButton ba = new JButton ("Valider le choix");
    ba.addActionListener (new ValiderChoix(jlm));
    p.add (ba, "South");
    return p;
  }  // panelJList

  // classes internes

  // validation des choix à l'aide du JButton Valider
  class ValiderChoix implements ActionListener {
    JList jlm;

    ValiderChoix (JList jlm) {
      this.jlm = jlm;
    }

    public void actionPerformed (ActionEvent evt) {
      // le tableau des indices des éléments sélectionnés
      int[] tabselect = jlm.getSelectedIndices();
      System.out.println ("\n\n");
      for (int i=0; i < tabselect.length; i++) {
        System.out.println ("i : " + i + " : " + tabselect[i]);
      }

      // le tableau des valeurs des éléments sélectionnés
      Object[] tabselectv = jlm.getSelectedValues();
      for (int i=0; i < tabselectv.length; i++) {
        System.out.println ("i : " + i + " : " + tabselectv[i]);
      }
    }
  }

  public static void main (String[] args) {
    new APISwing4();
  }

}  // class APISwing4
```

6.11 LES ARBRES ET LA CLASSE JTREE

6.11.1 Représentation d'un arbre en mémoire (TreeNode)

La classe DefaultMutableTreeNode permet de représenter un nœud d'un arbre. Chaque nœud (sauf la racine) a un nœud père et de 0 à n nœuds fils. La racine de l'arbre est un nœud qui n'a pas de nœud père. Un nœud peut contenir une référence sur un

objet lié à l'application. La structure de l'arbre et/ou le contenu des objets référencés peuvent changer, d'où le terme de Mutable. Cette classe contient des méthodes pour fournir :

- le nombre de fils d'un nœud ;
- le n ième fils d'un nœud ;
- le nœud du père ;
- la référence de l'objet du nœud ;
- les nœuds en partant d'un nœud donné (différents types de parcours) ;
- etc.

```
// un synonyme plus mnémonique (qui ne fait rien de plus)
class Noeud extends DefaultMutableTreeNode {
  Noeud (Object objet) {
    super (objet);
  }
}
```

La méthode **creerArbre**() permet de créer l'arbre des groupes d'étudiants comme indiqué ci-dessous et sur la figure 6.13. Cette initialisation pourrait se faire par lecture des informations dans un fichier.

Etudiants
- A
  - A1
    - Etudiant 1
  - A2
    - Etudiant 2
- B
  - B1
    - Etudiant 3
    - Etudiant 4
    - Etudiant 5
  - B2
    - Etudiant 6

```
// créer l'arbre des groupes et des étudiants
Noeud creerArbre () {
  Noeud racine   = new Noeud ("Etudiants");
  Noeud groupeA  = new Noeud ("A");
  Noeud groupeA1 = new Noeud ("A1");
  Noeud groupeA2 = new Noeud ("A2");
  Noeud groupeB  = new Noeud ("B");
  Noeud groupeB1 = new Noeud ("B1");
  Noeud groupeB2 = new Noeud ("B2");
  racine.add  (groupeA);
  racine.add  (groupeB);
```

```
    groupeA.add (groupeA1);
    groupeA.add (groupeA2);
    groupeB.add (groupeB1);
    groupeB.add (groupeB2);

    // chaque Noeud feuille contient une référence vers un Etudiant
    for (int i=0; i < lesEtudiants.nbEtudiants(); i++) {
      Etudiant etudiant = lesEtudiants.getEtudiant(i);
      Noeud  nd = new Noeud (etudiant) ;
      String sg = etudiant.getsousgrp();
      if (sg.equals("A1")) {
        groupeA1.add (nd);
      } else if (sg.equals("A2")) {
        groupeA2.add (nd);
      } else if (sg.equals("B1")) {
       groupeB1.add (nd);
      } else if (sg.equals("B2")) {
       groupeB2.add (nd);
      }
    }
    return racine;
  }  // creerArbre
```

La méthode infosNoeud() fournit sur la sortie standard des informations sur le Nœud. Elle utilise les méthodes de la classe DefaultMutableTreeNode.

```
// des informations sur un Noeud
void infosNoeud (Noeud nd) {
  System.out.println ("infosNoeud nd " + nd);
  System.out.println ("infosNoeud getParent " + nd.getParent());
  TreeNode[] tn = nd.getPath();
  for (int i=0; i < tn.length; i++) {
    System.out.println (i + " : " + tn[i]);
  }
  int n = nd.getChildCount();
  for (int i=0; i < n; i++) {
    System.out.println( "Child " + i + " : " + nd.getChildAt(i));
  }
}  // infosNoeud
```

Exemple de résultat en partant du nœud groupeB1

```
infosNoeud nd B1
infosNoeud getParent B                                    getParent
0 : Etudiants                                               getPath
1 : B                                                       getPath
2 : B1                                                      getPath
Child 0 : étudiant3 prenom3 B                             getChildAt
Child 1 : étudiant4 prenom4 B                             getChildAt
Child 2 : étudiant5 prenom5 B                             getChildAt
```

La méthode enumererLesNoeuds() permet de faire un parcours complet des nœuds de l'arbre.

```
// parcours en profondeur en partant d'un noeud donné;
// résultats dans le JEditorPane je (en HTML)
void enumererLesNoeuds (Noeud nd) {
  String s = new String("<h2 > Parcours récursif de l'arbre </h2 > ");
//Enumeration e = nd.breadthFirstEnumeration();  // en largeur
  Enumeration e = nd.depthFirstEnumeration();    // en profondeur
  while (e.hasMoreElements()) {
     s += e.nextElement() + "<br > ";
  }
  je.setText(s);
}
```

Remarque : dans cette partie, il s'agit de représenter les nœuds de l'arbre en mémoire sans représentation graphique. De nombreuses autres méthodes existent qui permettent de traiter les nœuds de l'arbre, ou de sélectionner certains nœuds de l'arbre.

6.11.2 La classe JTree

Un JTree permet de représenter graphiquement des informations structurées. Les méthodes de JTree donne des informations sur l'état du dessin (pas de l'arbre en mémoire). Un nœud du dessin peut être réduit ou développé. Un nœud du dessin peut être sélectionné. Les nœuds peuvent être sélectionnés par leur chemin ou leur rang (numéro dans l'arbre). On peut récupérer les chemins des nœuds sélectionnés. La sélection multiple de nœuds est possible comme pour les JList en utilisant les touches Ctrl et Alt.

On peut ajouter des écouteurs à un JTree pour prendre en compte différents événements : la sélection simple ou multiple dans le JTree affiché ou l'expansion/réduction d'un nœud intermédiaire.

Comme pour les JList, on peut définir un Renderer pour changer la présentation des informations des nœuds de l'arbre. On peut ajouter des icônes par exemple.

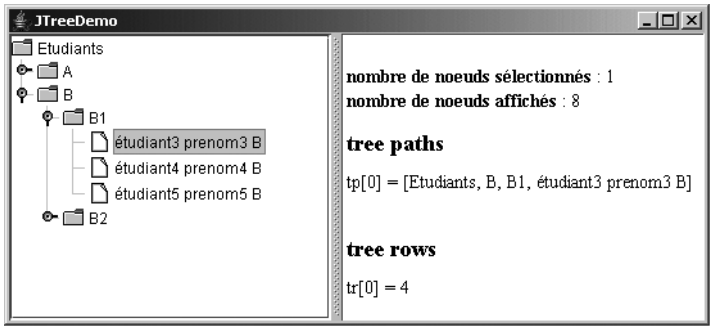


Figure 6.12 — Un JTree avec le Renderer standard.

6.11.3 L'écouteur de sélection simple ou multiple

EcouteurArbre est l'écouteur sur l'arbre des étudiants. Chaque clic sur un nœud de l'arbre permet d'obtenir des informations dans le JEditorPane (en HTML) comme indiqué sur la figure 6.13.

```
// l'écouteur pour controler un clic sur un noeud
class EcouteurArbre implements TreeSelectionListener {
  public void valueChanged (TreeSelectionEvent evt) {
    JTree arbre = (JTree) evt.getSource();
    if (arbre.getSelectionCount() == 0) return;  // pb réduction

    String s = new String(); // en html
    s += "<br > < b > nombre de noeuds sélectionnés </b > : "
                                        + arbre.getSelectionCount();
    s += "<br > < b > nombre de noeuds affichés </b > : "
                                              + arbre.getRowCount();
    s += "<h2 > tree paths </h2 > ";

    // cas de sélection multiple : écrit les paths et les rows
    TreePath[] tp = arbre.getSelectionPaths();
    for (int i=0; i < tp.length; i++) {
      s += "tp[" + i +"] =  " + tp[i] + "<br > ";
    }
    s += "<h2 > tree rows </h2 > ";
    int[] tr = arbre.getSelectionRows();
    for (int i=0; i < tr.length; i++) {
      s += "tr[" + i +"] = " + tr[i] + "<br > ";
    }
    je.setText(s);

  }  // valueChanged
}  // EcouteurArbre
```

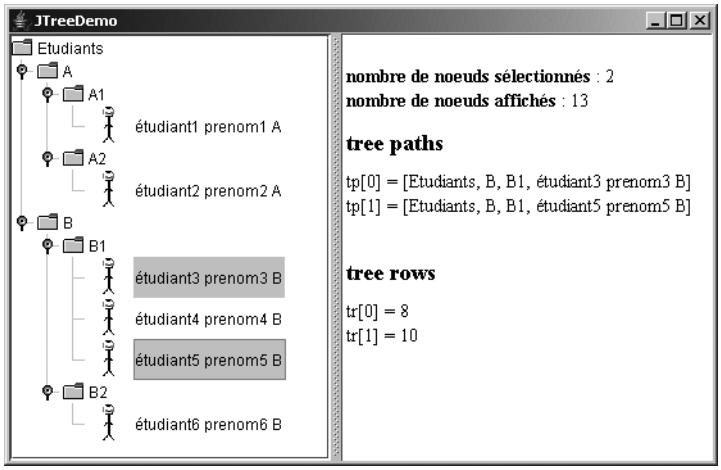


Figure 6.13 — Un JTree avec un Renderer personalisé.

On peut aussi agir à l'aide d'un écouteur à chaque fois que l'arbre est développé ou réduit en ajoutant l'écouteur suivant.

```
 // l'écouteur pour contrôler expansion et réduction de l'arbre
class EcouteurExpansion implements TreeExpansionListener {

  public void treeCollapsed (TreeExpansionEvent evt) {
    JTree arbre = (JTree) evt.getSource();
   je.setText ("<i > réduction de " + evt.getPath() + "</i > < br > ");
  }  // treeCollapsed

  public void treeExpanded (TreeExpansionEvent evt) {
   je.setText ("<i > expansion de " + evt.getPath() + "</i > < br > ");
  }  // treeCollapsed

}
```

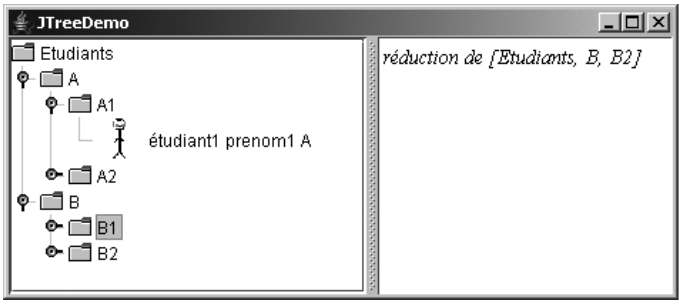


Figure 6.14 — Réduction ou expansion d'un arbre.

6.11.4 Le Renderer de l'arbre (JTree) des étudiants

Comme pour les JList, on peut contrôler les informations affichées pour chacun des nœuds et par exemple ajouter un icône comme sur la figure 6.13. L'écouteur de la classe RendererArbre peut se définir de plusieurs façons. On peut partir de l'écouteur par défaut (qui existe dans tous les cas) et le modifier (possibilité 1) ou définir entièrement la méthode qui met en forme les libellés des nœuds (possibilité 2). Ci-dessous, un icône est ajouté pour les feuilles de l'arbre (nœud des étudiants).

```
// RendererJTreeEtudiant.java  // JTree

package mdpaquetage.etudiant;

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

// RENDERER de JTree

/* possibilité 1
// on récupère le TreeCellRenderer par défaut, et on le modifie.
```

```
public class RendererJTreeEtudiant extends DefaultTreeCellRenderer {
  public RendererJTreeEtudiant (JTree arbre) {
    setLeafIcon (new  ImageIcon("md12.gif"));
    arbre.setCellRenderer (this);
  }
}
*/

// possibilité 2 : on définit le TreeCellRenderer
// mise en page d'une cellule (nœud) de l'arbre
class RendererJTree1 implements TreeCellRenderer {
  public Component getTreeCellRendererComponent (
    JTree tree,
    Object value,
    boolean selected,
    boolean expanded,
    boolean leaf,
    int row,
    boolean hasFocus) {
      if (leaf) {
        return new JLabel("" + value,
                                new ImageIcon("md12.gif"),JLabel.LEFT);
      } else {
        return new JLabel("" + value);
      }
  }
}

public class RendererJTreeEtudiant {
  public RendererJTreeEtudiant (JTree arbre) {
    arbre.setCellRenderer (new RendererJTree1());
  }
}
```

6.11.5 Utilisation du Renderer de l'arbre des étudiants

Le constructeur JTreeDemoEtu() permet de créer le composant de la figure 6.13 utilisant un Renderer pour le JTree (et un JEditorPane pour afficher les résultats).

```
class JTreeDemoEtu extends JPanel {
  JEditorPane je = new JEditorPane("text/html","");
  LesEtudiants lesEtudiants = new LesEtudiants();

  // le constructeur de l'arbre des étudiants
  public JTreeDemoEtu() {
    super (new GridLayout(1,0));
    Noeud racine = creerArbre();
    enumererLesNoeuds (racine);
    JTree arbre = new JTree (racine);
    arbre.setEditable (true);  // false par défaut

    // l'arbre dans un JScrollPane
```

```
    // et le JEditorPane dans un JSplitPane
    JScrollPane scr = new JScrollPane (arbre);
    JSplitPane  sp  = new JSplitPane  (JSplitPane.HORIZONTAL_SPLIT);
    sp.setDividerLocation (250);
    sp.setLeftComponent   (scr);
    sp.setRightComponent  (je);
    add (sp);

    arbre.addTreeSelectionListener (new EcouteurArbre());
    arbre.addTreeExpansionListener (new EcouteurExpansion());
    // ligne suivante en commentaire = Renderer standard
    RendererJTreeEtudiant rd = new RendererJTreeEtudiant(arbre);
  }  // constructeur JTreeDemoEtu
```

6.12 LA CLASSE JTABLE

Une JTable est un composant utilisé pour afficher et éditer des tables à deux dimensions. La première ligne peut contenir les titres des colonnes.

6.12.1 Exemple de JTable élémentaire

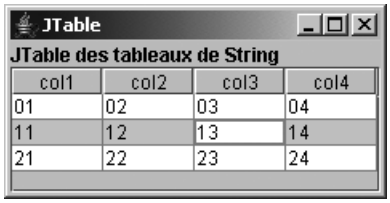


Figure 6.15 — JTable avec Renderer standard.

La classe suivante permet de créer la JTable de la figure 6.15 à partir d'un tableau de String des en-têtes, et d'un tableau à deux dimensions de String représentant les valeurs de la JTable.

```
// JTableDemo0.java

import java.awt.*;       // Container, BorderLayout
import javax.swing.*;    // JFrame, JLabel

public class JTableDemo0 extends JFrame {

  public JTableDemo0() {
    Container f = (JPanel) getContentPane();
    setSize  (new Dimension(200, 120));
    setTitle ("JTable");

    // JTable élémentaire créée à partir d'un tableau
                                                (par défaut éditable)
    f.add (new JLabel("JTable des tableaux de String"),
                                                 BorderLayout.NORTH);
```

```
    JScrollPane jscr = new JScrollPane (creerJtable0());
    f.add (jscr);
    setVisible (true);
  }  // constructeur JTableDemo0

  // créer une JTable élémentaire à partir de tableaux de String
  JTable creerJtable0() {
    String[] titres  = {"col1", "col2", "col3", "col4" };
    String[][] tabch = {              // table de chaines de caractères
        { "01", "02", "03", "04"},
        { "11", "12", "13", "14"},
        { "21", "22", "23", "24"},
    };
    return new JTable (tabch, titres);
  }

  public static void main (String[] args) {
    new JTableDemo0();
  }

}  // JTableDemo0
```

6.12.2 Le TableModel de la JTable

Comme pour la JList, les données de la JTable peuvent être fournies par les méthodes d'un objet de type **TableModel**. C'est le rôle de la classe TableModelEtudiant ci-dessous. La méthode getSize() fournit le nombre de lignes de la table. La méthode getColumnName (int n) fournit l'en-tête de la colonne n. La méthode getColumn-Count() fournit le nombre de colonnes, etc. La méthode getValueAt (l, c) fournit la valeur à la ligne l et à la colonne c. Cette classe fournit toutes les informations concernant les données de la JTable.

```
// TableModelEtudiant.java

package mdpaquetage.etudiant;

import javax.swing.table.*;  // AbstractTableModel

// TableModel qui fournit les données pour la JTable
// ou modifie les données à partir de la JTable (cas de l'édition)
public class TableModelEtudiant extends AbstractTableModel {
  LesEtudiants lesEtudiants;
  boolean      editable = false;

  // constructeur
  public TableModelEtudiant (LesEtudiants lesEtudiants) {
    this.lesEtudiants = lesEtudiants;
  }

  // fournir l'en-tête de la colonne n
  public String getColumnName (int n) {
    switch (n) {
    case 0 : return "nom";
    case 1 : return "prenom";
```

```
    case 2 : return "groupe";
    case 3 : return "sousgrp";
    case 4 : return "boursier";
    }
    return "";
  }

  public int getColumnCount() { return 5; }
  public int getRowCount() {
    return lesEtudiants.nbEtudiants();
  }

  public Class getColumnClass (int c) {
    return getValueAt (0,c).getClass();  // classe du 1° élément
                                                          de la colonne
  }

  // fournir l'objet pour une ligne et une colonne données
  public Object getValueAt (int ligne, int col) {
    Etudiant etudiant = lesEtudiants.getEtudiant (ligne);
    switch (col) {
    case 0 : return etudiant.getnom();
    case 1 : return etudiant.getprenom();
    case 2 : return etudiant.getgroupe();
    case 3 : return etudiant.getsousgrp();
    case 4 : return new Boolean (etudiant.getboursier());
    }
    return "?";  // erreur
  }  // getValueAt

  public boolean isCellEditable (int ligne, int col) {
    return editable;
  }

  public void setCellEditable (boolean editable) {
    this.editable = editable;
  }

  // appelé en fin de modification en édition de la case ligne x col
  // l'éditeur a fourni l'objet saisi pour cette case;
  // modifier les données en conséquence.
  public void setValueAt (Object objet, int ligne, int col) {
    Etudiant etudiantEnCours = lesEtudiants.getEtudiant (ligne);
    String s = objet.toString();
    switch (col) {
    case 0 : etudiantEnCours.setnom     (s); break;
    case 1 : etudiantEnCours.setprenom  (s); break;
    case 2 : etudiantEnCours.setgroupe  (s); break;
    case 3 : etudiantEnCours.setsousgrp (s); break;
    case 4 : etudiantEnCours.setboursier(((Boolean)objet)
                                          .booleanValue()); break;
    }
  }  // setValueAt

}  // TableModelEtudiant
```

6.12.3 Le Renderer de la JTable des étudiants

La présentation de la JTable est confiée à un ou plusieurs renderers. On peut avoir un renderer différent par colonne de la JTable par exemple. Ci-après, on utilise le même renderer pour toutes les colonnes. Le renderer a cependant une action différente suivant le numéro de la colonne.

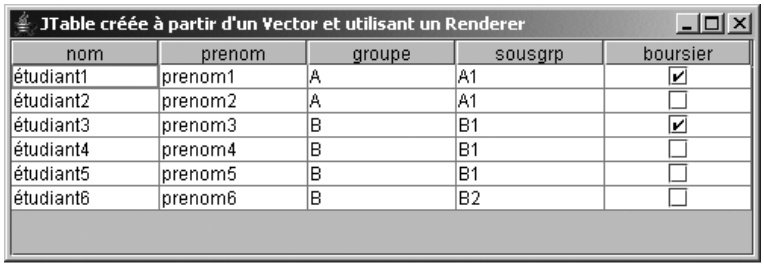


Figure 6.16 — JTable avec un Renderer standard.

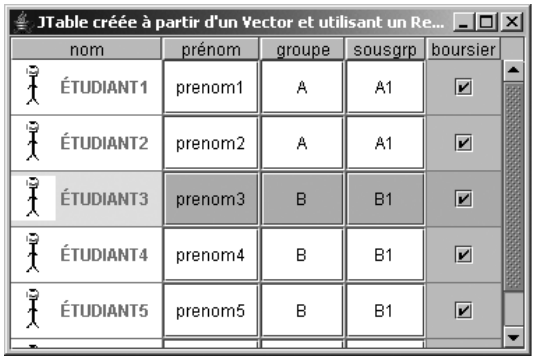


Figure 6.17 — JTable avec un Renderer personnalisé.

```
// RendererJTableEtudiant.java  // JTable

package mdpaquetage.etudiant;

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

// RENDERER de JTable

// mise en page du Component d'une cellule suivant la cellule
// et ses caractéristiques (JLabel, JTextField, JCheckBox)
class RendererJTable1 implements TableCellRenderer {
  JLabel     c  = new JLabel ();           // instancier une seule fois
  JTextField cf = new JTextField ("");

  // fournir le composant à afficher la cellule en row x column;
```

```
  // pas forcément le même composant pour toutes les cellules
  public Component getTableCellRendererComponent (
      JTable  table,
      Object  obj,
      boolean isSelected,
      boolean hasFocus,
      int     row,
      int     column) {

    JComponent co = null;
    switch (column) {
    case 0 :                       // colonne 0 de la JTable : un JLabel
      //JLabel c = new JLabel ();
      c.setOpaque (true);
      c.setForeground (Color.RED);
      c.setBackground (Color.white);
      c.setIcon ( new ImageIcon("md12.gif"));
      c.setHorizontalAlignment (SwingConstants.LEFT);
      c.setText (((String)obj).toUpperCase());
      co = c;
      break;
    case 1 : case 2 : case 3 :      // colonnes 1, 2, 3 : un JTextField
      //JTextField cf = new JTextField ("");
      cf.setHorizontalAlignment (SwingConstants.CENTER);
      cf.setText ((String)obj);
      co = cf;
      break;
    case 4 :                                  // colonne 4 : un JCheckBox
      // instancier à chaque fois : mettre plutôt en attribut
      JCheckBox jc = new JCheckBox(null, null,
                                   ((Boolean)obj).booleanValue());
      jc.setHorizontalAlignment (SwingConstants.CENTER);
      co = jc;
      break;
    }  // switch

    if (co != null) {                                  // les couleurs
      if (isSelected) co.setBackground (Color.lightGray);
      if (hasFocus)   co.setBackground (Color.cyan);
    }
    return co;  // le composant à afficher
  }
}  // RendererJTable1
```

Le renderer s'applique aux colonnes 0 1 2 3 4. On pourrait avoir des Renderer différents suivant les colonnes.

```
public class RendererJTableEtudiant {
  // mise en page de la JTable
  public RendererJTableEtudiant (JTable jtable) {
    // le renderer s'applique aux colonnes 0 1 2 3 4
    RendererJTable1 rd1 = new RendererJTable1();
    jtable.getColumnModel().getColumn(0).setCellRenderer(rd1);
    jtable.getColumnModel().getColumn(1).setCellRenderer(rd1);
```

```
        jtable.getColumnModel().getColumn(2).setCellRenderer(rd1);
        jtable.getColumnModel().getColumn(3).setCellRenderer(rd1);
        jtable.getColumnModel().getColumn(4).setCellRenderer(rd1);
        // taille des colonnes 0 et 4, et des lignes
        jtable.getColumnModel().getColumn(0).setPreferredWidth (90);
        jtable.getColumnModel().getColumn(4).setPreferredWidth (30);
        jtable.setRowHeight (40);
       }

} // RendererJTableEtudiant
```

6.12.4 Le CellEditor de la JTable

Pendant la phase d'édition d'une cellule, un composant standard ou un composant personnalisé permet d'effectuer la saisie ou la modification de la donnée de la cellule. Sur la figure 6.18, un composant JComboBox permet de choisir vrai ou faux pour le champ boursier d'un étudiant. Lorsque l'édition est terminée, le Renderer présente la cellule sous la forme d'un JCheckBox comme pour les autres lignes. L'édition aurait d'ailleurs pu se faire en utilisant un JCheckBox (voir le code en commentaire dans le programme ci-dessous). On définit deux éditeurs de cellules : Editeur1 pour une cellule des colonnes 0 à 3 éditées dans un JTextField ; Editeur2 pour une cellule de la colonne 4 (un booléen) éditée dans un JComboBox.

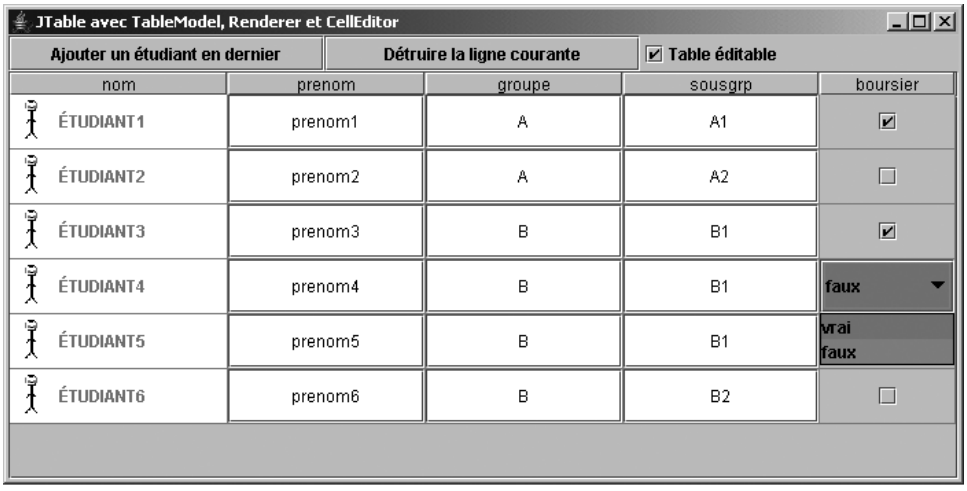


Figure 6.18 — Édition d'une cellule (CellEditor) du tableau avec un JComboBox. L'édition terminée, la case est présentée avec une case à cocher (indiquée par le Renderer) comme pour les autres étudiants.

```
// EditorJTableEtudiant.java  // JTable

package mdpaquetage.etudiant;

import java.awt.*;          // Container, BorderLayout
```

```
import javax.swing.*;       // JPanel, JFrame, JLabel
import javax.swing.table.*;

// Editeur pour les colonnes 0, 1, 2 et 3 (type String)
class Editeur1 extends AbstractCellEditor implements TableCellEditor {
  // le composant à afficher dans la case à éditer
  JTextField tf = new JTextField ();

  // on édite dans un TextField
  //pour une cellule des colonnes 0 à 3
  public Component getTableCellEditorComponent (
    JTable  table,
    Object  value,
    boolean isSelected,
    int     row,
    int     col
   ) {

    tf.setText (value.toString());
    tf.setBackground (Color.RED);
    tf.setHorizontalAlignment (SwingConstants.CENTER);
    return tf;
  }

  // l'objet à fournir en fin d'édition
  public Object getCellEditorValue () {
    return tf.getText();
  }
}  // class Editeur1

// Editeur pour la colonne 4
// En l'absence de cet écouteur, l'éditeur par défaut
// des booléens est pris en compte.
// L'édition peut se faire dans un JCheckBox (ou dans un JComboBox)
class Editeur2 extends AbstractCellEditor implements TableCellEditor {
  //JCheckBox c = new JCheckBox ();  // si édition avec JCheckBox
  String[] choix = {"vrai", "faux"};
  JComboBox c = new JComboBox (choix);

  // On édite dans un JComboBox pour une cellule de la colonne 4
  public Component getTableCellEditorComponent (
    JTable  table,
    Object  value,
    boolean isSelected,
    int     row,
    int     col
   ) {
    boolean bvalue = ((Boolean)value).booleanValue();
    c.setBackground (Color.RED);

    // si édition avec un CheckBox
    //c.setHorizontalAlignment (SwingConstants.CENTER);
    //c.setSelected (bvalue);
```

```
    // JComboBox
    if (bvalue) c.setSelectedIndex(0); else c.setSelectedIndex(1);

    // on pourrait aussi retourner un composant maison
    return c;
  }

  // l'objet Boolean à fournir en fin d'édition
  public Object getCellEditorValue () {
    //boolean valeur = c.isSelected();          // JCheckBox
    boolean valeur = c.getSelectedIndex()==0;   // JComboBox
    return new Boolean (valeur);
  }
}  // class Editeur2
```

L'éditeur de cellules numéro 1 s'applique aux colonnes 0 1 2 3, l'éditeur numéro 2 à la colonne 4.

```
// mise en page des diverses colonnes lors de l'édition
 public class EditorJTableEtudiant {
  // mise en Ppge de la JTable
  public void EditorJTableEtudiant (JTable jtable) {
    // Editeur
    Editeur1  editeur1 = new Editeur1 ();  // String
    Editeur2  editeur2 = new Editeur2 ();  // boolean

    jtable.getColumnModel().getColumn(0).setCellEditor(editeur1);
    jtable.getColumnModel().getColumn(1).setCellEditor(editeur1);
    jtable.getColumnModel().getColumn(2).setCellEditor(editeur1);
    jtable.getColumnModel().getColumn(3).setCellEditor(editeur1);
    // par défaut, un éditeur de JCheckBox est utilisé
    jtable.getColumnModel().getColumn(4).setCellEditor(editeur2);
    jtable.setCellSelectionEnabled (false);
   }

 }  // EditorJTableEtudiant
```

6.12.5 Utilisation de la JTable des étudiants

Le programme ci-dessous crée un JFrame. Les données à afficher dans la JTable proviennent de l'objet de la classe TableModelEtudiant. La JTable est mise en page à l'aide de l'objet RendererJTableEtudiant. L'édition des cellules est prise en compte par l'objet de la classe EditorJTableEtudiant.

```
// JTableDemo2.java TableModel, Renderer et Editeur de cellules

import java.awt.*;          // Container, BorderLayout
import java.awt.event.*;
import javax.swing.*;       // JPanel, JFrame, JLabel
import javax.swing.table.*;
import mdpaquetage.etudiant.*;
```

```
public class JTableDemo2 extends JFrame {
  LesEtudiants        lesEtudiants = new LesEtudiants();
  TableModelEtudiant  tableModelEtudiant;
  JTable              jtable;
  JCheckBox b3;

  // JTable avec un  TableModel, Renderer et un CellEditor
  public JTableDemo2() {
    setBounds (500, 10, 700, 350);
    setTitle ("JTable avec un TableModel");
    Container f = getContentPane();

    // Objet tableModelEtudiant fournissant les données de la table
    tableModelEtudiant = new TableModelEtudiant (lesEtudiants);
    jtable             = new JTable (tableModelEtudiant);

    // mise en page de la table
    // ligne suivante en commentaire = Renderer standard
    RendererJTableEtudiant rd = new RendererJTableEtudiant(jtable);

    // Edition des cellules
    EditorJTableEtudiant ed = new EditorJTableEtudiant (jtable);

    // la JTable au centre dans un JScrollPane
    JScrollPane scrollpane = new JScrollPane (jtable);
    f.add (scrollpane);

    // 2 boutons au nord dans un JPanel
    JPanel pb = new JPanel (new GridLayout(1,0));
    JButton b1 = new JButton ("Ajouter un étudiant en dernier");
    ActionButton a = new ActionButton();
    b1.addActionListener (a);
    b1.setActionCommand ("B1");
    pb.add (b1);
    JButton b2 = new JButton ("Détruire la ligne courante");
    b2.addActionListener (a);
    b2.setActionCommand ("B2");
    pb.add (b2);
    b3 = new JCheckBox ("Table éditable", false);
    b3.addActionListener (a);
    b3.setActionCommand ("B3");
    pb.add (b3);
    f.add (pb, "North");

    setVisible (true);
  }  // constructeur JTableDemo2

  // class interne
  class ActionButton implements ActionListener {

    public void actionPerformed (ActionEvent evt) {
      int row = jtable.getEditingRow();
      String s = evt.getActionCommand();
```

```
        if (s.equals("B1")) {
          lesEtudiants.addEtudiant(new Etudiant("?", "", "",
                                                    "", false));
        } else if (s.equals("B2")) {
          lesEtudiants.removeEtudiant(row);
        } else {
          tableModelEtudiant.setCellEditable(b3.isSelected());
        }
        tableModelEtudiant.fireTableDataChanged();
      }
    }

    public static void main (String[] args) {
      new JTableDemo2();
    }

}  // JTableDemo2
```

6.13 CONCLUSION

La librairie Swing définit de nombreux composants graphiques permettant de développer des interfaces graphiques conviviales et variées. On retrouve bien sûr les composants de base comme en AWT, avec cependant de nouvelles possibilités : bulles d'aide, bordures variées, textes et icônes, etc. Les conteneurs sont également plus nombreux et permettent des présentations mieux adaptées : onglets, barres d'outils, barres de division, etc. Le principe des écouteurs est similaire dans sa mise en œuvre à celui de AWT. De nombreux écouteurs ont été ajoutés pour s'adapter aux besoins des composants Swing.

Les composants JList, JTree et JTable se veulent très généraux. Dans leur version de base, ils peuvent être mis en œuvre très facilement. Leurs versions plus évoluées permettent de définir des objets se chargeant de gérer la provenance des données, la mise en page des données du composant et de l'édition des données du composant. Ils peuvent alors être assez complexes à mettre en œuvre. Par l'éventail de ses possibilités, la librairie Swing permet de définir des interfaces graphiques plus professionnelles que AWT. Les concepts acquis, il faut consulter la documentation pour mettre en œuvre la multitude des détails disponibles.

Chapitre **7**

Les entrées-sorties

7.1 INTRODUCTION

En Java, les entrées et les sorties d'informations se présentent comme des flux. L'information est transférée sous forme d'octets à partir ou vers une source qui peut être variée : un fichier, un périphérique, une connexion sur un autre ordinateur (internet par exemple), ou même en provenance d'une zone de mémoire centrale. Au niveau le plus élémentaire, l'information n'est qu'une suite d'octets.

Dans certaines applications, l'utilisateur souhaite accéder aux informations sous forme de texte qu'il peut afficher dans différentes fenêtres ou modifier à l'aide d'un éditeur de texte ; les nombres figurent alors sous leur forme ascii (ou Unicode).

Dans d'autres applications utilisant des données, les informations sont transférées sous leur forme binaire (cas des nombres entiers ou réels). Ces données binaires ne

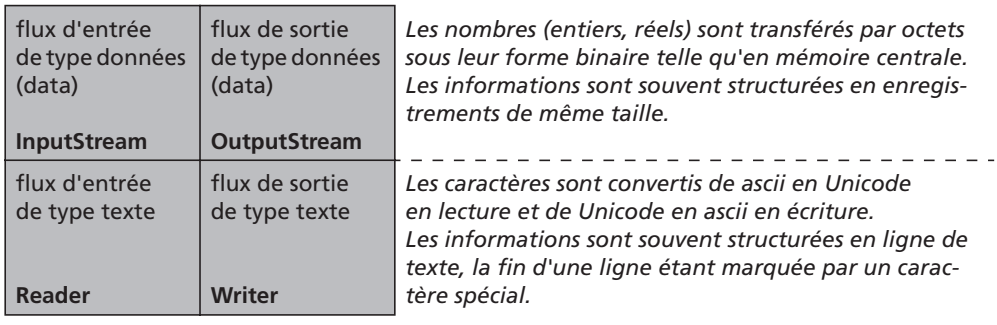


Figure 7.1 — Les différents types de flux représentés en Java par les quatre classes abstraites InputStream, OutputStream, Reader et Writer.

peuvent pas être lues par un éditeur de texte, seulement par un programme. On distingue les **flux de type données (data)** des **flux de type text** . Pour chacun des deux cas, le flux peut être un flux d'entrée ou un flux de sortie.

Les classes abstraites (voir 3.2.1, page 96) définissent les caractéristiques communes aux flux suivant leur type. Des classes dérivées spécialisent ces flux suivant la source ou la destination du flux : fichier (File), tube (Pipe), zone mémoire (tableau en mémoire).

7.2 LES EXCEPTIONS SUR LES FICHIERS

Des anomalies peuvent se produire lors de l'utilisation de fichiers. Ces anomalies peuvent être prises en compte par les programmes en captant les exceptions (voir pages 74 et 106). Si on essaie d'ouvrir un fichier qui n'existe pas, il y a déclenchement d'une exception de type **FileNotFoundException** ; si on atteint la fin d'un fichier, il y a lancement d'une exception de type **EOFException** (ce qui n'est pas vraiment une anomalie).

7.3 LES FLUX DE TYPE DONNÉES (DATA) POUR UN FICHIER

7.3.1 Les classes (File, Data)InputStream, (File, Data)OutputStream

La classe **FileInputStream** dérive de la classe InputStream et concerne les lectures de données dans un **fichie** . La classe construit un objet de type FileInputStream à partir du nom d'un fichier (voir constructeur figure 7.2). Des méthodes permettent pour cet objet de **lire un octet ou un tableau d'octets** . La lecture d'un caractère Unicode demande deux appels de read() ou la lecture dans un tableau de deux caractères. La lecture d'un entier ou d'un réel se fait octet par octet. L'écriture à l'aide d'un objet de type **FileOutputStream** est également rudimentaire : **écriture d'un octet ou d'un tableau d'octets** . Cet objet est **l'objet de base** sur lequel d'autres objets vont se contruire.

flux d'entrée de type données (data) pour les fichiers **FileInputStream**	flux de sortie de type données (data) pour les fichiers **FileOutputStream**
FileInputStream (String nom)	**FileOutputStream** (String nom)
int **read** () (-1 si fin de fichier) int **read** (byte[] b) etc.	**write** (int) écrit un octet **write** (byte[] b) etc.

Figure 7.2 — Les flux (stream) d'entrées ou de sorties (input, output) de type données pour les **fichiers** (File).

Pour faciliter les lectures de caractères Unicode ou de nombres sous leur forme binaire, Java dispose de la classe **DataInputStream** dont un objet peut être construit à partir d'un objet de type InputStream, donc d'un objet de type FileInputStream dérivé de InputStream. Cela signifie que DataInputStream peut être utilisé avec d'autres flux dérivés de InputStream (pas seulement les flux de fichiers).

Un objet de type DataInputStream permet la lecture de données de types primitifs (lecture d'un entier, lecture d'un réel, etc.). De même, un objet de type **DataOutputStream** permet l'écriture de types primitifs. En l'absence de ces deux classes, il faudrait lire ou écrire les types primitifs octet par octet (ce que permet seulement un FileInputStream ou un FileOutputStream).

flux d'entrée de type données (data) avec lectures de types primitifs **DataInputStream**	flux de sortie de type données (data) avec écritures de types primitifs **DataOutputStream**
DataInputStream (InputStream)	**DataOutputStream** (OutputStream)
int **read** () int **read** (byte[] b)	**write** (int)
readChar ()	**writeChar** (int) **writeChars** (String)
readInt () **readLong** () **readFloat** () **readDouble** () ... etc.	**writeInt** (int) **writeLong** (long) **writeFloat** (float) **writeDouble** (double) ... etc.

Figure 7.3 — Les flux de type données (data) pour les types primitifs des fichiers : on peut lire ou écrire un entier, un réel, une chaîne de caractères.

7.3.2 Exemple de classes utilisant des fichiers de type données (data)

7.3.2.1 La classe Etudiant

On gère des notes d'étudiants en utilisant un fichier de données. Pour cela, on enregistre pour chaque étudiant, son nom (10 caractères maximum) et prénom (10 caractères maximum) et ses trois notes d'examens (un réel entre 0 et 20). Les méthodes suivantes de la classe **Etudiant** permettent de créer un étudiant, d'enregistrer ses 3 notes, de modifier une de ses notes et de fournir avec toString(), ses caractéristiques.

```
// DataEtudiant.java
//   écrire et lire des données d'un fichier
//   de données binaires (type data)
```

```
//   et les afficher dans une fenêtre

import java.io.*;      // DataOutputStream
import java.swing.*;   // JFrame

class Etudiant {
  String  nom;
  String  prenom;
  float[] note = new float [3];

  Etudiant (String nom, String prenom) {
    this.nom    = new String (nom);
    this.prenom = new String (prenom);
    for (int i=0; i < note.length; i++) note[i] = 0;
  }

  // création avant lireEtu
  Etudiant () {
    this ("", "");
  }

  // enregistrer les trois notes de l'étudiant
  void modifNote (float n1, float n2, float n3) {
    note [0] = n1;
    note [1] = n2;
    note [2] = n3;
  }

  // enregistrer (modifier) la i' note de l'étudiant de 1 à 3
  void modifNote (int i, float n) {
    if (i >= 1 && i <= note.length) note [i-1] = n;
  }

  // les caractéristiques de l'étudiant
  public String toString () {
    return nom + " " + prenom + " "
               + note[0] + " " + note[1] + " "+ note[2];
  }
```

La méthode **ecrireEtu**() écrit les caractéristiques d'un Etudiant dans un fichier de type **DataOutputStream**. Le flux étant de type data (ou encore appelé binaire), les informations sont écrites dans le fichier sans transformation, telles qu'elles sont en mémoire. La méthode **writeChars**() écrit la chaîne de dix caractères dans le fichier fs ; chaque caractère est en fait écrit sous la forme de 2 octets (caractère Unicode). La méthode **writeFloat**() écrit un réel sur 32 bits dans le fichier, soit 4 octets. Un flux de type FileOutputStream ne permettrait que des écritures octet par octet. Voir le dump en hexadécimal de la figure 7.5.

```
  // écrire les caractéristiques d'un étudiant dans le fichier fs
  void ecrireEtu (DataOutputStream fs) {
    try {
      fs.writeChars (nom);    // écrit en Unicode
      fs.writeChars (prenom); // nom et prenom ajustés à 10 caractères
```

```
        for (int i=0; i < note.length; i++) {
          fs.writeFloat(note[i]);
        }
      } catch (IOException e) {
        System.out.println (e);
      }
    }
```

La méthode **lireEtu**() lit les caractéristiques d'un Etudiant dans le fichier de type **DataInputStream**. La méthode **readChar**() lit un caractère Unicode (2 octets) à partir du fichier fe. La méthode **readFloat**() lit un réel sur 32 bits (un Float). La méthode *read*() mise en commentaires ne permet de lire qu'un octet à la fois. Il faudrait alors reconstituer le réel en juxtaposant les 4 octets. C'est la seule façon de lire pour des flux de type FileInputStream.

```
    // lire les caractéristiques d'un étudiant dans le ficher fe
    void lireEtu (DataInputStream fe) {
      try {
        for (int i=0; i < 10; i++) {
          char c = fe.readChar();               // lit 2 octets (Unicode)
          //char c1 = (char) fe.read();        // lit le 1° octet Unicode
          //char c = (char) fe.read();         // lit le 2° octet Unicode
          nom    += c;
        }
        for (int i=0; i < 10; i++) prenom += fe.readChar();
        for (int i=0; i < note.length; i++) {
          note[i] = fe.readFloat();
          //int i1 = fe.read();                // lit le 1° octet du réel
          //int i2 = fe.read();                // lit le 2° octet du réel
          //int i3 = fe.read();                // lit le 3° octet du réel
          //int i4 = fe.read();                // lit le 4° octet du réel
        }
      } catch (IOException e) {
        System.out.println ("lireEtu " + e);
      }
    }

    // insérer ici la méthode lireDEtu présentée en 7.3.3, page 292

  } // class Etudiant
```

7.3.2.2 *La classe DataEtudiant*

La classe **DataEtudiant** affiche dans deux fenêtres différentes (voir figure 7.4), les informations écrites dans le fichier `etudiant.dat`, puis celles lues à partir du fichier ainsi créé.

```
  // définit une fenêtre contenant une zone de texte
  class DataEtudiant extends JFrame {
    JTextArea ta = new JTextArea (50, 20);

    DataEtudiant (String titre) {
      setTitle   (titre);  // type Frame
      setSize    (300, 100);
```

```
    ta.setEditable (false);
    getContentPane().add (ta);
    setVisible (true);
  }  // constructeur DataEtudiant
```

La méthode **ecritdata** ouvre un flux de type **DataOutputStream** à partir d'un flux de type **FileOutputStream** sur le fichier `etudiant.dat`. Les trois objets Etudiant créés sont écrits dans ce fichier fs par la méthode ecrireEtu() et affichés dans une fenêtre de type Frame.

```
// écriture de données dans le fichier etudiant.dat
void ecritdata () {
  String chT = new String ("Données à écrire\n");
  DataOutputStream fs = null;
  try {
    fs = new DataOutputStream (new FileOutputStream ("etudiant.dat"));

    // les noms et prénoms sont sur 10 caractères;
    // on pourrait ajuster à 10 par programme
    Etudiant e1 = new Etudiant ("Dupont    ", "Jules     ");
    e1.modifNote (12.5f, 8.5f, 14.5f);
    Etudiant e2 = new Etudiant ("Duval     ", "Jacques   ");
    e2.modifNote (3.5f, 7.5f, 11.5f);
    Etudiant e3 = new Etudiant ("Dupré     ", "Sébastien ");
    e3.modifNote (9.5f, 17.25f, 10.5f);

    e1.ecrireEtu (fs);
    e2.ecrireEtu (fs);
    e3.ecrireEtu (fs);
    chT += e1 + "\n" + e2 + "\n" + e3;
    fs.close();
  } catch (IOException e) {
    System.out.println (e);
  }

  ta.setText (chT);
}  // ecritdata
```

La méthode **litdata** ouvre un flux de type **DataInputStream** à partir d'un flux de type **FileInputStream** pour le fichier `etudiant.dat`. Trois objets Etudiant sont créés et initialisés par lecture dans le fichier fe, puis affichés dans une fenêtre de type Frame.

```
// lecture de données dans le fichier etudiant.dat
void litdata () {
  String chT = new String ("Données lues\n");
  DataInputStream fe = null;
  try {
    fe = new DataInputStream (new FileInputStream ("etudiant.dat"));
  } catch (FileNotFoundException e) {
    System.out.println (e);
    System.exit (1);
  }

  Etudiant e1 = new Etudiant();
  e1.lireEtu (fe);
```

```
        chT += e1 + "\n";

        Etudiant e2 = new Etudiant();
        e2.lireEtu (fe);
        chT += e2 + "\n";

        Etudiant e3 = new Etudiant();
        e3.lireEtu (fe);
        chT += e3 + "\n";

        ta.setText (chT);
      }  // litData

      public static void main (String[] args) {
        // on écrit les caractéristiques des étudiants
        DataEtudiant de1 = new DataEtudiant ("Ecriture des données");
        de1.ecritdata();
        de1.setLocation (20, 20);

        // on relit les caractéristiques des étudiants
        DataEtudiant de2 = new DataEtudiant ("Lecture des données");
        de2.litdata();
        de2.setLocation (400, 20);
      }

    }  // class DataEtudiant
```

Exemples de résultats d'écriture et de lecture dans un fichier de type data.

Figure 7.4 — Écriture et relecture des données du fichier "etudiant.dat".

Le dump en hexadécimal de la figure 7.5 indique une partie du contenu du fichier etudiant.dat après écriture des caractéristiques des trois étudiants. On y retrouve pour le premier étudiant, les 10 caractères Unicode (20 octets) de la chaîne "Dupond ", suivis des 10 caractères Unicode du prénom "Jules ", suivis des trois notes réelles sur 4 octets : (41 48 0 0), (41 8 0 0) et (41 68 0 0).

```
.D.u.p.o.n.t. .        0 44   0 75   0 70   0 6f   0 6e   0 74   0 20   0 20
. . .J.u.l.e.s.        0 20   0 20   0 4a   0 75   0 6c   0 65   0 73   0 20
. . . . AH..A...       0 20   0 20   0 20   0 20  41 48   0  0  41  8   0  0
Ah...D.u.v.a.l.       41 68   0  0   0 44   0 75   0 76   0 61   0 6c   0 20
. . . . .J.a.c.q       0 20   0 20   0 20   0 20   0 4a   0 61   0 63   0 71
.u.e.s. . . @`..       0 75   0 65   0 73   0 20   0 20   0 20  40 60   0  0
```

Figure 7.5 — Dump en hexadécimal du début du fichier etudiant.dat de type "données binaires".

7.3.3 L'accès direct sur un flux d'un fichier de type data (binaire)

On peut accéder directement à un ou plusieurs octets d'un **fichie** de type data en utilisant la classe **RandomAccessFile**. Celle-ci permet d'accéder à un fichier en lecture, en écriture ou en lecture-écriture suivant le deuxième paramètre du constructeur de l'objet RandomAccessFile, le premier indiquant le nom du fichier. Un fichier ouvert en lecture définit les **mêmes méthodes** que **DataInputStream** ; s'il est ouvert en écriture, il dispose des méthodes de **DataOutputStream**, et s'il est ouvert en lecture-écriture des méthodes des deux.

La méthode **seek**() permet de se positionner sur un des octets du fichier ; la lecture ou l'écriture se fait alors à partir de cet octet. L'exemple ci-dessous reprend le fichier `etudiant.dat` créé précédemment. On accède en lecture à un enregistrement Etudiant en fonction de son numéro entier de 0 à 2.

La méthode **lireDEtu()** suivante de lecture directe d'un enregistrement est ajoutée à la classe Etudiant (voir page 289).

```
// lire directement les caractéristiques de l'étudiant num
void lireDEtu (RandomAccessFile fe, int num) {
  try {
    fe.seek (num*(20*2+3*4));      // 20 caractères Unicode + 3 réels
    for (int i=0; i < 10; i++) nom    += fe.readChar();
    for (int i=0; i < 10; i++) prenom += fe.readChar();
    for (int i=0; i < note.length; i++) note[i] = fe.readFloat();
  } catch (IOException e) {
    System.out.println ("Erreur lors de l'accès aléatoire " + e);
  }
}
```

Le programme **LectureDirecte** accède directement à un enregistrement du fichier étudiant.dat créé précédemment à partir du numéro de l'étudiant (qui ici pour simplifier est aussi le numéro d'enregistrement sinon, il faudrait créer un index ou une fonction (hash-code) établissant la relation entre numéro d'étudiant et numéro d'enregistrement).

```
// LectureDirecte.java  lecture directe (dans un ordre aléatoire)
//                      des notes d'un étudiant

import java.io.*;   // IOException

// class Etudiant dans le même paquetage

class LectureDirecte {

  public static void main (String[] args) throws IOException {
    // lecture aléatoire dans un fichier binaire
    RandomAccessFile fr = new RandomAccessFile ("etudiant.dat", "r");
    Etudiant etu = new Etudiant();
    etu.lireDEtu (fr, 2);           // on lit directement Etudiant n° 2
    System.out.println (etu);
  } // main

}  // class LectureDirecte
```

Exemple de résultat pour l'étudiant numéro 2 (le premier étudiant a le numéro 0) :

```
Dupré      Sébastien  9.5 17.25 10.5
```

7.4 LES FLUX DE TYPE TEXTE POUR UN FICHIER

7.4.1 Les classes FileReader, BufferedReader

Les fichiers de type texte ne contiennent que des caractères directement affichables sur écran ou imprimables. Si un fichier de type texte contient un nombre, celui-ci doit être écrit sous sa forme ascii, et non sous sa forme binaire comme vu en 7.3. Java utilise une représentation Unicode des caractères en mémoire centrale. Cependant, de nombreux logiciels (les éditeurs de texte par exemple) utilise le code ascii représentant un caractère sur un octet. Le langage Java dispose de deux classes permettant de faire cette conversion :

- en lecture : conversion de ascii en Unicode
- en écriture : conversion de Unicode en ascii

flux d'entrée de type texte pour les fichiers ascii **FileReader**	flux de sortie de type texte pour les fichiers ascii **FileWriter**
FileReader (String nom) int **read** () -1 si EOF etc.	**FileWriter** (String nom) void **write** (int) void **write** (String) etc.

Figure 7.6 — Les flux de type text en lecture ou en écriture (Reader ou Writer) pour les fichiers (File).

La classe **FileReader** ne permet que la lecture de caractères. La classe **BufferedReader** permet de faire des lectures de lignes entières à l'aide de la méthode **readLine**().

flux tamponné d'entrée de type texte **BufferedReader**	flux tamponné de sortie de type texte **BufferedWriter**
BufferedReader (Reader fe) int **read** () String **readLine** () null si EOF etc.	**BufferedWriter** (Writer fs) void **newLine** () void **write** (int c) void **write** (String, int, int) etc.

Figure 7.7 — Les flux tamponnés (Buffered) de type text en lecture ou en écriture (Reader ou Writer).

7.4.2 Exemples de classes utilisant un fichier de type texte

On veut lire des mots dans un fichier afin d'initialiser un tableau de mots. Le jeu du pendu (voir page 187) utilise un tableau des mots à découvrir. Ce tableau peut être initalisé directement dans le programme, ou à l'aide d'un fichier ce qui permet de changer les mots sans toucher au programme, de définir des niveaux de jeu ou des vocabulaires spécifiques, voire de jouer dans une autre langue (l'anglais par exemple). Les mots lus dans le fichier sont affichés dans une fenêtre de type Frame.

```
// LireMots.java  lire des lignes d'un fichier de type text
//                et les afficher dans une fenêtre

import java.io.*;      // FileReader
import javax.swing.*;  // JFrame

public class LireMots extends JFrame {
  static final int MAXMOTS = 200;
  String[]  tabMots = new String [MAXMOTS];
  int       nbMots  = 0;
  JTextArea ta      = new JTextArea();
  String    chT     = "";

  LireMots (String titre) {
    setTitle (titre);
    setSize  (300, 300);
    ta.setEditable (false);
    getContentPane().add (ta, "Center");      // BorderLayout
    setVisible (true);
  }
```

La lecture à l'aide d'un objet de la classe **BufferedReader** appliquée à un objet de la classe **FileReader** permet de lire des caractères (read()) ou des lignes entières à l'aide de **readLine**(). Les mots sont stockés dans un tableau tabMots de String et affichés dans une fenêtre de type Frame.

```
// lecture de mots (readline) dans le fichier de type texte mots.dat
// et rangement dans le tableau tabMots
void litMots() {
  BufferedReader fe = null;
  try {
    // ouverture du fichier mots.dat
    fe = new BufferedReader (new FileReader ("mots.dat"));
    // Initialisation du tableau tabMots
    String ch;
    // readLine return null si EOF
    while ( (ch = fe.readLine()) != null) {
      if (nbMots < MAXMOTS) tabMots [nbMots++] = ch;
    }
    for (int i=0; i < nbMots; i++) chT += i + ":" + tabMots[i] +"\n";
    ta.setText (chT);
  } catch (IOException e) {
    ta.setText ("Erreur fichier : " + e);
  }
}  // litMots
```

```
    public static void main (String[] args) {
      new LireMots ("Lecture de lignes").litMots();
    }
  }
```

Exemple de résultats avec l'appel de la méthode litMots().

Figure 7.8 — Lecture et affichage du fichier de texte "mots.dat".

Remarque : la lecture à l'aide d'un objet de la classe FileReader permet de lire seulement des caractères ascii à partir d'un fichier. Les fins de ligne sont également lus comme un caractère. On pourrait se baser sur celles-ci pour délimiter les mots, ce qui est automatiquement fait par la classe BufferedReader. La méthode suivante est donnée pour montrer la différence de possibilités entre BufferedReader et FileReader. Ci-dessous, les limites de mots ne sont pas automatiquement reconnues.

```
// lecture caractère par caractère (read) dans le fichier
// de type texte "mots.dat";
// pas de readline() pour FileReader
void litCar() {
  FileReader fe = null;
  try {
    fe = new FileReader ("mots.dat");
    int c;
    while ( (c = fe.read()) != -1) chT += (char) c;
    ta.setText (chT);
  } catch (IOException e) {
    ta.setText ("Erreur fichier : " + e);
  }
}  // litCar
```

7.4.3 La lecture d'un fichier de données en mode texte

Le fait que le fichier soit un fichier de texte n'interdit pas qu'il contienne des données. Seulement celles-ci sont sous forme ascii et doivent être converties de ascii en binaire s'ils s'agit de nombres sur lesquels on veut faire des calculs. De plus, on doit pouvoir connaître les limites de chacun des champs soit parce que les informations

sont délimités par un caractère séparateur, soit parce que les champs ont une longueur fixe qui doit être scrupuleusement respectée.

L'exemple développé ci-après concerne la gestion des factures d'abonnés au téléphone. On connaît pour chaque abonné, son nom, son prénom, son numéro de téléphone, son âge et le montant de sa facture.

```
// LireTelephone.java  lire des lignes d'un fichier de type texte
//                     et les afficher dans une fenêtre

import java.awt.*;          // Font
import javax.swing.*;       // JFrame
import java.io.*;           // BufferedReader
import java.util.*;         // StringTokenizer

class Telephone {
  String nom     = "";
  String prenom  = "";
  String tel     = "";
  int    age     = 0;
  float  facture = 0;

  public String toString () {
   return  nom + " " + prenom + " " + tel + " " +
           age + " " + facture;
  }

}  // Telephone
```

La classe LireTelephone présente deux façons de lire un fichier de type texte contenant des données de champs de longueur fixe ou délimités par un caractère séparateur. Elle affiche les résultats de la lecture dans deux fenêtres différentes.

```
public class LireTelephone extends JFrame {
  JTextArea ta  = new JTextArea (50,20);
  String    chT = "";

  LireTelephone (String titre) {
    setTitle       (titre);
    setSize        (350, 80);
    ta.setEditable (false);
    ta.setFont     (new Font ("Courier", Font.BOLD, 12));
    getContentPane().add (ta, "Center");
}
```

La méthode **litTelSep**() ouvre le fichier et lit les lignes. Chaque ligne est découpée en ses différentes composantes séparées par des caractères / à l'aide d'un objet de la classe **StringTokenizer** construit avec deux paramètres : le premier indique la chaîne à découper, le deuxième indique le caractère séparateur. Chaque appel à la méthode **nextToken**() pour cet objet fournit la chaîne suivante délimitée par le séparateur. S'il s'agit d'une chaîne contenant des nombres, elle est ensuite convertie en binaire (en entier pour age, en réel pour facture).

Exemple de fichier à traiter : telsep.dat

```
Dupont/Jules/0299254488/32/1209.21
Duval/Jacques/0133447843/60/2134.10
```

```
// lecture (readline) dans le fichier de type texte "telsep.dat";
// les champs sont séparés par des séparateurs /
void litTelSep () {
  BufferedReader fe = null;
  try {
    fe = new BufferedReader (new FileReader ("telsep.dat"));
    String ch;
    // readLine() return null si EOF
    while ( (ch = fe.readLine()) != null) {
      Telephone te = new Telephone();
      StringTokenizer client = new StringTokenizer (ch, "/");
      te.nom     = client.nextToken();
      te.prenom  = client.nextToken();
      te.tel     = client.nextToken();
      te.age     = Integer.parseInt(client.nextToken());
      te.facture = Float.valueOf (client.nextToken()).floatValue();
      System.out.println (te);
      chT       += te + "\n";
    }
    ta.setText (chT);
    setVisible (true);
  } catch (IOException e) {
    ta.setText ("Erreur fichier : " + e);
  }
}  // litTelSep
```

La méthode **litTel**() indique une autre façon de lire des données à partir d'un fichier de type texte en répartissant les données sur des champs de taille fixe.

Exemple de fichier à traiter : tel.dat

```
Dupont    Jules     0299254488  32   1209.21
Duval     Jacques   0133447843  60   2134.10
```

```
// lecture (readline) dans le fichier de type texte "tel.dat";
// les champs sont de longueur fixe
void litTel () {
  BufferedReader fe = null;
  try {
    fe = new BufferedReader (new FileReader ("tel.dat"));
    String ch;
    // readLine return null si EOF
    while ( (ch = fe.readLine()) != null) {
      Telephone te = new Telephone();
      te.nom     = ch.substring (0,10);
      te.prenom  = ch.substring (10,20);
      te.tel     = ch.substring (20,30);
      String s   = ch.substring (30,34).trim(); // enlève les blancs
```

```
            te.age     = Integer.parseInt (s);
            te.facture = Float.valueOf(ch.substring (34,44)).floatValue();
            System.out.println (te);
            chT       += te + "\n";
            ta.setText (chT);
            setVisible (true);
          }
        } catch (IOException e) {
          ta.setText ("Erreur fichier : " + e);
        }
      }  // litTel

      public static void main (String[] args) {
        LireTelephone t1 = new LireTelephone ("Lecture avec séparateur");
        t1.setLocation (20, 20);
        t1.litTelSep();

        LireTelephone t2 = new LireTelephone ("Lecture avec taille fixe");
        t2.setLocation (400, 20);
        t2.litTel();
      }

    }  // LireTelephone
```

Résultats d'exécution : dans les deux cas, on retrouve les valeurs définies dans le fichier sous forme de texte, et converties en binaire pour `age` et `facture`.

Figure 7.9 — Lecture de données dans un fichier de type texte, les champs étant délimités par un séparateur ou de taille fixe.

7.4.4 L'accès direct à la définition d'un mot du dictionnaire sur un fichier de type texte

Les méthodes d'accès direct à un enregistrement vues pour un fichier binaire (voir 7.3.3) restent valables pour un fichier de type texte. On dispose d'un fichier de type texte contenant les définitions des mots d'un dictionnaire.

Lors de la création de ce fichier des définitions, on crée également un fichier d'index contenant le mot, l'indice du premier octet de la définition dans le fichier des définitions, et la longueur de la définition. Cet index est chargé en mémoire dans un tableau en début de traitement. Pour simplifier, sur l'exemple ci-dessous, l'index est créé directement dans un tableau (voir page suivante).

Les mots de l'index sont présentés dans une liste déroulante (voir JList page 261) permettant de choisir le mot dont on veut la définition. Le rang de l'élément choisi dans la liste déroulante permet de connaître l'indice du premier octet de la définition du mot dans le fichier de type text.

anecdote	0	53
cellule	53	13
fleuve	66	37
incognito	103	35
ras	138	17
trémolo	155	23

Figure 7.10 L'index du fichier des définitions : la définition de fleuve se trouve à l'octet 66 du fichier des définitions sur une longueur de 37 octets.

```
// Definition.java   lecture directe de la définition d'un mot

import java.awt.*;           // Color
import javax.swing.*;        // JFrame
import java.io.*;            // RandomAccessFile
import java.util.*;          // Vector
import javax.swing.event.*;  // ListSelectionListener
```

Une entrée du tableau d'index se compose du mot, de l'indice de son premier caractère dans le fichier des définitions et de la longueur de cette définition.

```
// une entrée de la table d'index
class Entree {
  String  mot;  // fleuve
  long    depl; // 66 (déplacement)
  int     lg;   // 37

  Entree (String mot, int depl, int lg) {
    this.mot  = mot;
    this.depl = depl;
    this.lg   = lg;
  }

  public String toString () {
    return mot + " " + depl + " " + lg;
  }

  // lire directement la définition correspondant à cette entrée
  String lireDefinition (RandomAccessFile fe) {
    String rep = new String();
    try {
      fe.seek (depl);
      for (int i=0; i < lg; i++) rep += (char) fe.read();
    } catch (IOException e) {
       System.out.println ("Erreur accès aléatoire " + e);
    }
    return rep;
  }

}  // Entree
```

La classe Index crée le tableau d'index donnant accès à la définition de chacun des mots du dictionnaire. La méthode **rechercherEntree**() fournit le rang dans le tableau du mot en paramètre. Sur un index volumineux, cette recherche devrait être une recherche dichotomique et non une recherche séquentielle comme ci-dessous.

```
class Index {
  Entree[] index = {
    new Entree ("anecdote",    0, 53),
    new Entree ("cellule",    53, 13),
    new Entree ("fleuve",     66, 37),
    new Entree ("incognito", 103, 35),
    new Entree ("ras",       138, 17),
    new Entree ("trémolo",   155, 23)
  };
  int nbIndex = index.length;

  // retourne l'entrée de l'index correspondant au mot
  Entree rechercherEntree (String mot) {
    boolean trouve = false;
    int     i      = 0;
    while (!trouve && (i < nbIndex) ) {
      trouve = mot.equals (index[i].mot);
      if (!trouve) i++;
    }
    return trouve ? index[i] : null;
  }

}  // Index
```

La classe **Definitio** affiche une liste déroulante des mots du dictionnaire et fournit la définition du mot sélectionné.

```
class Definition extends JFrame {
  RandomAccessFile fr;
  Index ind;
  JTextArea zoneText;

  Definition () {
    setTitle  ("Consultation d'un dictionnaire");
    Container f = getContentPane();
    setBounds (200, 150, 300, 250);
    setBackground (Color.lightGray);

    // les données de la liste
    ind = new Index ();
    Vector v = new Vector ();
    for (int i=0; i < ind.nbIndex; i++) {
      v.addElement (ind.index[i].mot);
    }
    // la liste
    JList lst = new JList (v);
    lst.setSelectionMode (ListSelectionModel.SINGLE_SELECTION);
    lst.setBackground (Color.cyan);
```

```
        lst.setVisibleRowCount (4);
        lst.setBorder
                (BorderFactory.createTitledBorder ("Dictionnaire"));
        f.add (new JScrollPane(lst), "North");

        // la zone de texte
        zoneText = new JTextArea ();
        zoneText.setEditable (false);
        zoneText.setBackground (Color.yellow);
        zoneText.setBorder
                (BorderFactory.createTitledBorder ("Définition du mot"));
        f.add (zoneText, "Center");

        // ouverture du fichier des définitions "dico.txt"
        try {
          fr = new RandomAccessFile ("dico.txt", "r");
        } catch (IOException e) {
          System.out.println (e);
        }

        // un écouteur pour les mots de la liste
        lst.addListSelectionListener (new choixMot());

        setVisible (true);
      }  // Definition
```

L'écouteur est sollicité lorsque un choix a été effectué dans la liste déroulante. La méthode **getSelectedValue** () fournit le mot sélectionné dont on retrouve le rang dans le tableau d'index à l'aide de la méthode **rechercherEntree**() définie dans la classe Index. Il est alors possible de lire directement la définition du mot à partir des informations fournies par le tableau d'index.

```
class choixMot implements ListSelectionListener {
    public void valueChanged (ListSelectionEvent evt) {
      JList  list   = (JList) evt.getSource();
      String mot    = "" + list.getSelectedValue ();  // toString()
      Entree entree = ind.rechercherEntree (mot);
      if (entree != null) {
        String def = entree.lireDefinition (fr);
        zoneText.setText(def);
      }
    }
  }

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

}  // class Definition
```

Exemple de résultats : l'utilisateur a sélectionné **fleu e**.

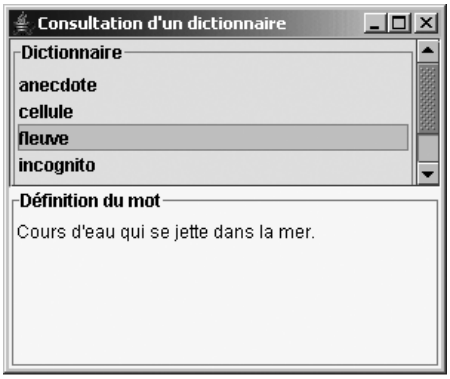


Figure 7.11 — Accès direct à la définition d'un mot dans un fichier de type text.

7.5 EXEMPLES DE SOURCES DE FLUX BINAIRE

La notion de flux indique que les informations peuvent provenir de différentes sources.

7.5.1 Les flux en lecture (type InputStream)

```
// DiversFlux.java    lecture d'octets avec InputStream
//                    sur plusieurs flux d'entrée

import java.io.*;   // IOException

class DiversFlux {
```

La méthode lireOctets() lit des octets sur le flux de type InputStream `is` sans connaître la source réelle du flux défini lors de l'appel de cette méthode. Elle réécrit les caractères sur la sortie standard pour vérification.

```
  static void lireOctets (InputStream is) {
    byte[] tamponOctet = new byte [1024];
    int nb = 0;
    try {                                  // read de la classe dérivée
      nb = is.read (tamponOctet);                    // lit des octets
    } catch (IOException e) {
      System.out.println (e);
    }
    System.out.println ("nombre de bytes lus : " + nb);
    for (int i=0; i < nb; i++) {
      if (i % 10 == 0) System.out.println();  // 10 par ligne
      System.out.print (i + "(" + tamponOctet[i] + ") ");
    }
    System.out.println();
  }  // lireOctets
```

La méthode lireEntiers() lit des entiers sur le flux de type DataInputStream `is` sans connaître la source réelle du flux défini lors de l'appel de cette méthode. Elle réécrit les entiers sur la sortie standard.

```
static void lireEntiers (DataInputStream is) {
  int[] tamponInt = new int [20];
  int nb = 0;
  try {
    while (nb < 20) {  // jusqu'à 20 ou EOF
      int lu = is.readInt ();  // lit des entiers
      tamponInt[nb++] = lu;    // si EOF on sort sur readInt()
    }
  } catch (EOFException e) {
    // sortie EOF normale; on exécute finally
  } catch (IOException e) {
    System.out.println (e);
  } finally {
    System.out.println ("nombre d'entiers lus : " + nb);
    for (int i=0; i < nb; i++) {
      System.out.println (i + " " + tamponInt[i]);
    }
  }
}  // lireEntiers
```

Le programme ci-dessous lit des octets provenant du clavier (System.in), d'un fichier `mots.dat` et d'une zone mémoire (un tableau `entree` de byte) en utilisant un flux de type InputStream. Il affiche les informations lues sur ces trois flux par la méthode lireOctets().

```
public static void main (String args[]) throws IOException {
  // pour FileNotFoundException
  byte[] entree = { 0, 0, 2, 3, 0, 1, 0, 2};

  InputStream           is1 = System.in;
  FileInputStream       is2 = new FileInputStream ("noms.dat");
  ByteArrayInputStream  is3 = new ByteArrayInputStream (entree);

  System.out.println ("\nsource = entree standard\n");
  lireOctets (is1);
  System.out.println ("\nsource =  noms.dat\n");
  lireOctets (is2);
  System.out.println ("\nsource = tableau d'octets\n");
  lireOctets (is3);
```

La lecture de la zone mémoire `entree` (tableau de byte) se fait ci-dessous avec un objet de type DataInputStream qui permet la lecture d'entiers avec lireEntiers().

```
  // lecture d'entiers à partir d'un tableau de byte
  ByteArrayInputStream  is4 = new ByteArrayInputStream (entree);
  DataInputStream       dis = new DataInputStream (is4);
  lireEntiers (dis);

} // main

}  // class DiversFlux
```

```
source = entree standard

le petit chat                                          tapé au clavier
nombre de bytes lus : 15                                 avec CR et LF

0(108) 1(101) 2(32) 3(112) 4(101) 5(116) 6(105) 7(116) 8(32) 9(99)
10(104) 11(97) 12(116) 13(13) 14(10)
```

```
source =  noms.dat

nombre de bytes lus : 44

0(68) 1(117) 2(112) 3(111) 4(110) 5(100) 6(32) 7(65) 8(108) 9(98)
10(101) 11(114) 12(116) 13(13) 14(10) 15(68) 16(117) 17(118) 18(97) 19(108)
20(32) 21(67) 22(111) 23(114) 24(101) 25(110) 26(116) 27(105) 28(110) 29(13)
30(10) 31(68) 32(117) 33(102) 34(111) 35(117) 36(114) 37(32) 38(77) 39(105)
40(99) 41(104) 42(101) 43(108)
```

```
source = tableau d'octets          le tableau entree lu comme un flux
nombre de bytes lus : 8

0(0) 1(0) 2(2) 3(3) 4(0) 5(1) 6(0) 7(2)
```

Lors de l'appel de lireEntiers(), les octets (0,0,2,3) du tableau `entree` ont fourni l'entier 515 (2*256+3), et les octets (0,1,0,2) ont fourni l'entier 65538 (1*65536 + 2).

```
source = tableau d'octets lu comme un tableau d'entiers
nombre d'entiers lus : 2
0 515
1 65538
```

7.5.2 Les flux en écriture (type OutputStream)

On peut faire de même en écriture, et envoyer les informations vers la sortie standard (System.out), un fichier ou une zone mémoire.

> ***Remarque*** : entre deux processus (ou threads) on peut définir une communication grâce à des flux passant par un tube. Un processus écrit dans le tube ; un autre lit dans le tube. Les flux sont alors de type **PipedInputStream** et **PipedOutputStream** (voir figure 8.5, page 333).

7.6 EXEMPLES DE SOURCES DE FLUX DE TYPE TEXTE

7.6.1 Le flux de type texte pour l'entrée standard (System.in)

La lecture de nombres ou de caractères sur le flux d'entrée standard (le clavier par défaut) se fait comme pour les autres flux vus précédemment. Pour les flux de type

texte, la classe **InputStreamReader** est chargée de faire la conversion du code ascii au code Unicode. La classe BufferedReader appliquée à ce flux permet (voir figure 7.7) de disposer de la méthode readLine() qui lit une ligne entière. System.in est un objet de type InputStream désignant l'entrée standard. Les deux méthodes suivantes font partie du paquetage es (utilisé pour Factorielle en 1.4.2 et menu de la pile en 2.1.4).

```
// LectureClavier.java

package mdpaquetage.es;

import java.io.*;   // IOException

public class LectureClavier {
  static BufferedReader fe = new BufferedReader
    (new InputStreamReader (System.in));

  // lire une chaine au clavier
  public static String lireChaine (){
    // lecture tamponnée de caractères au clavier;
    // lecture tamponnée permet readline()
    String reponse = "";
    try {
      reponse = fe.readLine();
    } catch (Exception e) {
      System.out.println ("Erreur lireChaine");
      System.exit (1);
    }
    return reponse;
  }

  // lire un entier au clavier
  public static int lireEntier (){
    String  reponse  = "";
    int     n        = 0;
    boolean luEntier = false;
    while (!luEntier) {
      try {
        reponse  = fe.readLine();
        n        = Integer.parseInt (reponse);
        luEntier = true;
      } catch (Exception e) {
        System.out.println ("Valeur non entière," +
                            "\nretapez une valeur entière");
      }
    }
    return n;
  }

}  // class LectureClavier
```

7.6.2 La lecture à partir d'une URL

7.6.2.1 Le flux de texte en provenance d'une URL

Une URL (Uniform Ressource Locator) caractérise un fichier à charger soit localement avec le protocole *fil* , soit sur un site distant avec un protocole *http* par exemple. A partir de cette adresse URL, on peut ouvrir un flux d'entrée en mode text grâce à la classe InputStreamReader chargée de faire la conversion ascii en Unicode.

La classe **URL** permet de référencer une ressource (un fichier) sur un site local ou distant.

- **URL** (String) :crée une URL à partir d'une chaîne de caractères
- **getContent** () : fournit l'URL.
- **getFile** () : fournit le nom de fichier de l'URL.
- **getHost** () : fournit le nom de l'ordinateur hôte.
- **getPort** () : fournit le port de l'ordinateur hôte.
- **getProtocol** () : fournit le protocole.
- **openStream** () : ouvre une connection sur une URL et fournit un flux de type InputStream.
- **toString** () : fournit une chaîne représentant l'URL.

Le programme ci-après lit les informations en provenance d'une URL dans un tableau de caractères de nom `tampon` de 4096 caractères. On pourrait bien sûr faire une boucle pour lire toutes les informations. L'interface graphique est celle de la figure 7.12.

```
// LectureURL.java    connexion à une URL

import java.awt.*;            // Container
import javax.swing.*;         // JFrame
import java.awt.event.*;      // addActionListener
import java.io.*;             // IOException
import java.net.*;            // URL

class LectureURL extends JFrame {
  JTextArea ta = new JTextArea ();

  LectureURL () {
    setTitle ("Connexion Internet");
    Container f = getContentPane();
    setBounds (10, 10, 500, 400);
    JPanel p1 = new JPanel ();  // p1 : FlowLayout
    p1.setLayout (new BoxLayout (p1, BoxLayout.X_AXIS));
    JLabel l1 = new JLabel ("adresse Internet ");
    JTextField tf = new JTextField ("http://www.", 50);
    tf.setBackground (Color.yellow);
    p1.add (l1);
    p1.add (tf);
    f.add (p1, "North");

    ta.setText ("");
```

```
    ta.setEditable (false);
    ta.setBackground (Color.cyan);
    f.add (ta, "Center");

    tf.addActionListener (new Connecter());
    setVisible (true);
  }  // constructeur lectureURL
```

La classe interne **Connecter** définit le rôle de l'écouteur activé lorsqu'on valide par return, une adresse URL saisie dans la zone de texte de type TextField. On ouvre un flux sur cette adresse et on lit les 4096 premiers caractères.

```
  class Connecter implements ActionListener {
    public void actionPerformed (ActionEvent evt) {
      JTextField t = (JTextField) evt.getSource();
      String s = (String) t.getText();
      String zt = "";
      try {
        URL url = new URL (s) ;
        InputStream fluxE = url.openStream();

        // les caractéristiques de l'URL
        System.out.println ("s           : " + s);
        System.out.println ("url         : " + url);
        System.out.println ("getRef      : " + url.getRef());
        System.out.println ("getFile     : " + url.getFile());
        System.out.println ("getHost     : " + url.getHost());
        System.out.println ("getPort     : " + url.getPort());
        System.out.println ("getProtocol : " + url.getProtocol());

        InputStreamReader fe = new InputStreamReader (fluxE) ;
        char[] tampon = new char [4096];
        int nb = fe.read (tampon);  // lit des char (au plus 4096)
        System.out.println ("nb " + nb);
        for (int i=0; i < nb; i++) zt += tampon[i];
        ta.setText (zt);
      } catch (IOException e) {
        System.err.println ("Erreur URL : " + e);
      }
    }
  }  // class Connecter

  public static void main (String args[]) throws IOException {
    new LectureURL();
  } // main

}  // class LectureURL
```

Remarque : InputStreamReader permet de passer d'un flux d'octets (ascii) à un flux de caractères Unicode pour un flux d'entrée. Si on veut lire des lignes plutôt que des caractères, il faut faire une lecture tamponnée comme indiqué ci-après :

```
BufferedReader fe = new BufferedReader
                                        (new InputStreamReader (fluxE));
for (int i=0; i < 10; i++) zt += fe.readLine() + '\n';
ta.setText (zt);
```

7.6.2.2 Exemple d'exécution avec une URL de type fichier local

Sur l'exemple ci-dessous, on récupère dans la fenêtre, le source du programme java.

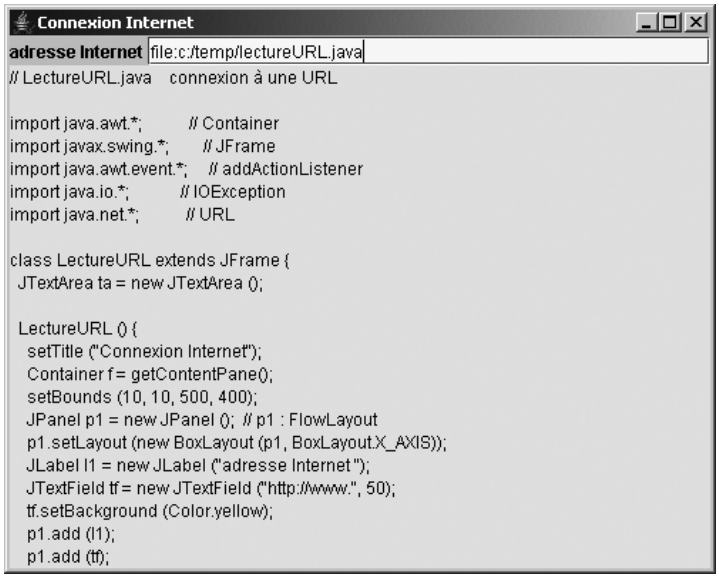


Figure 7.12 — Lecture d'une URL de type file.

Résultats sur la sortie standard :

```
s           : file : c :/temp/lectureURL.java
url         : file : c :/temp/lectureURL.java
getRef      : null
getFile     : c :/temp/lectureURL.java
getHost     :
getPort     : -1
getProtocol : file
nb 2523                                    nombre de caractères lus
```

7.6.2.3 Exemple d'exécution avec une URL de type fichier distant

En se connectant à l'URL *www.culture.fr*, on récupère un flux de caractères correspondant au code HTML envoyé par le serveur du site distant. On a ici le début d'un navigateur (comme Netscape ou Internet Explorer). Il faudrait cependant interpréter le code HTML et disposer les éléments dans la fenêtre pour pouvoir ensuite les activer.

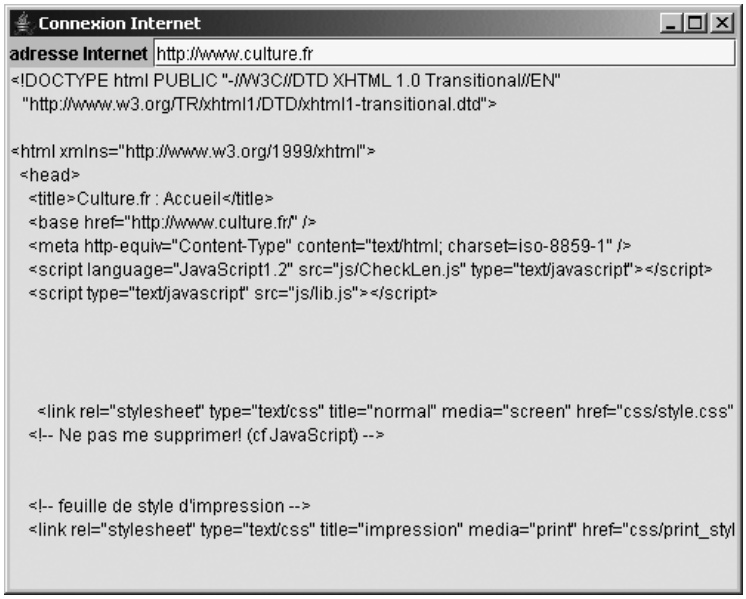


Figure 7.13 — Connexion au site Internet www.culture.fr : réception d'un flux de caractères.

Résultats sur la sortie standard :

```
s           : http://www.culture.fr
url         : http://www.culture.fr
getRef      : null
getFile     :
getHost     : www.culture.fr
getPort     : -1
getProtocol : http
nb 2637                                    nombre de caractères lus
```

Exercice 7.1 – Programme Java du pendu avec URL

Écrire le **composant PenduURL** héritant du composant **Pendu** (qui ne doit pas être modifié ; voir exercice 5.3, page 192) et qui permet d'initialiser l'ensemble des mots du jeu à découvrir à partir d'une URL (locale ou distante).

```
public class PenduURL extends Pendu {
  public PenduURL ();                          // mots par défaut de Pendu
  public PenduURL (URL urlFichier);                  // mots de urlFichier
  // mots de urlFichier et couleurs
  public PenduURL (URL urlFichier, Color cbPendu, Color cbPotence,
                   Color cfPotence);

}  // PenduURL
```

7.6.3 La classe PrintWriter : écriture formatée de texte

L'écriture formatée sur la sortie standard se fait avec System.out.println(). On exécute la méthode println() de l'objet `out` de la classe `System`. Ces possiblités de formatage peuvent s'appliquer à un fichier en sortie de type texte utilisant un objet de la classe PrintWriter.

```
// EcritureWriter.java

import java.io.*;   // IOException

class EcritureWriter {

  public static void main (String[] args) throws IOException {
    FileWriter fw1 = new FileWriter ("resultats.res");
    PrintWriter pw = new PrintWriter (fw1);

    float f1 = 2.5f;
    float f2 = 12.5f;

    pw.println ("f1 = " + f1 + ", f2 = " + f2);
    pw.println ("f1 + f2 = " + (f1+f2));
    pw.close();
  } // main

}  // class EcritureWriter
```

Contenu du fichier *resultats.res* après exécution du programme :

```
f1 = 2.5, f2 = 12.5
f1 + f2 = 15.0
```

7.7 L'ÉCRITURE ET LA LECTURE D'OBJETS : LA SÉRIALISATION

Comme on peut écrire des entiers ou des réels, on peut également écrire des objets pour pouvoir les relire ensuite. Deux classes (**ObjectOutputStream** et **ObjectInputStream**) sont chargées de ces écritures et lectures d'objets qui sont transférés comme des séries d'octets d'où le terme de sérialisation. Ces deux classes enregistrent les valeurs des attributs des objets plus des informations supplémentaires sur la classe. Pour qu'un objet soit sérialisable, il doit implémenter **l'interface Serializable** qui ne définit aucune méthode mais autorise la sérialisation de l'objet.

On définit la classe Etudiant suivante caractérisant un étudiant par les attributs nom, prénom et un tableau de trois notes réelles de 0 à 20.

```
// lireObjet.java

import java.io.*;

class Etudiant implements Serializable {
  String  nom;
  String  prenom;
  float[] note = new float [3];
```

```
  Etudiant (String nom, String prenom,
                              float note1, float note2, float note3) {
    this.nom    = nom;
    this.prenom = prenom;
    note [0] = note1;
    note [1] = note2;
    note [2] = note3;
  }

  public String toString () {
    return nom + " " + prenom + " "
                           + note[0] + " " + note[1] + " " + note[2];
  }
}  // class Etudiant
```

Sérialisation des objets Etudiant : écrire les objets et les relire.

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

La séquence d'instructions ci-dessous ouvre en écriture le fichier **binaire** "etudObj.dat", y écrit deux objets de type Etudiant et ferme le fichier.

```
    try {
      ObjectOutputStream fs = new ObjectOutputStream (
                                  new FileOutputStream ("etuObj.dat"));
      Etudiant jules   = new Etudiant ("dupont", "jules",
                                                   7.5f,   12f, 11.25f);
      Etudiant jacques = new Etudiant ("duval",  "jacques",
                                                  10.5f, 13.5f, 15.25f);
      fs.writeObject (jules);
      fs.writeObject (jacques);
      fs.close();
    } catch (IOException e) {
      System.out.println (e);
    }
```

Les instructions suivantes ouvrent en lecture le fichier binaire "etudObj.dat" et y lisent deux objets de type Etudiant

```
    try {
      ObjectInputStream fe = new ObjectInputStream (
                                   new FileInputStream ("etuObj.dat"));
      Etudiant etu1 = (Etudiant) fe.readObject();
      Etudiant etu2 = (Etudiant) fe.readObject();
      fe.close();
      System.out.println (etu1);
      System.out.println (etu2);
    } catch (IOException e) {
      System.out.println (e);
    } catch (ClassNotFoundException e) {
      System.out.println (e);
    }

  }  // main
}  // class EcrireLireObjet
```

Exemples de résultats :

Les deux objets de type Etudiant ont été écrits dans le fichier etuObj.dat, relus à partir de ce fichier et affichés sur écran pour vérification. Le fichier etuObj.dat enregistre les données et des informations supplémentaires sur les objets et leur classe. La structure est très spécifique de Java.

```
dupont jules 7.5 12.0 11.25
duval jacques 10.5 13.5 15.25
```

7.8 LA CLASSE FILE

La classe **File** fournit des informations sur les fichiers (et les répertoires qui sont une catégorie particulière de fichiers) et permet de gérer les caractéristiques de ces fichiers. On peut tester l'existence d'un fichier, le renommer, le changer de répertoire, connaître le nombre d'octets dans le fichier, etc. Les informations fournies sont parfois dépendantes du sytème d'exploitation utilisé (gestion des dates, des droits d'accès au fichier, etc.).

Principales méthodes de la classe File :

- **File** (String) : constructeur d'un objet File (pas d'un fichier).
- boolean **canRead** () : droit de lecture du fichier.
- boolean **canWrite** () : droit d'écriture du fichier.
- boolean **delete** () : destruction réussie ou non du fichier.
- boolean **exists** () : le fichier existe-t-il ?
- String **getAbsolutePath** () : fournit le chemin complet du fichier.
- String **getName** () : fournit le nom du fichier (sans chemin).
- String **getPath** () : fournit le chemin du fichier.
- boolean **isDirectory** () : le fichier est-il un répertoire.
- boolean **isFile** () : le fichier est-il un fichier normal.
- long **lastModifie** () : date de dernière modification du fichier.
- long **length** () : longueur du fichier en octets.
- **String[] list** () : fournit un tableau de String des noms des fichiers du répertoire.
- boolean **mkdir** () : crée un répertoire à partir des informations de l'objet File.
- boolean **renameTo** (File dest) : renomme le fichier.
- String **toString** () : fournit les caractéristiques du fichier.

Le programme suivant met en œuvre certaines des méthodes de File :

```
// TestClassFile.java  test de la classe File

import java.io.*;       // IOException

class TestClassFile {

  public static void main (String[] args) throws IOException {
    // crée un objet de type File pour "fichier.dat"
```

```
        // si le fichier n'existe pas, il n'est pas créé par new File() :
        // les attributs de l'objet File sont alors initialisés par défaut
        File f = new File ("fichier.dat");

        // crée le fichier d'objet f; la plupart des constructeurs de
        // fichiers acceptent un nom de fichier ou un objet de type File
        FileWriter fe = new FileWriter (f);

        System.out.println (
          "\ngetName         : " + f.getName()         +
          "\ngetAbsolutePath : " + f.getAbsolutePath() +
          "\ncanWrite        : " + f.canWrite()        +
          "\ncanRead         : " + f.canRead()
        );

        File rep = new File ("java");
        if (rep.mkdir()) {
          System.out.println ("répertoire " + rep.getName() + " créé");
        } else {
          System.out.println ("répertoire "+ rep.getName() + " non créé");
        }

      } // main

    }  // class TestClassFile
```

Exemple de résultats :

```
getName         : fichier.dat
getAbsolutePath : C :\temp\fichier.dat
canWrite        : true
canRead         : true
répertoire java créé
```

7.9 LA GESTION D'UN CARNET D'ADRESSES (FICHIER DE TEXTE)

On veut réaliser la gestion d'un carnet d'adresses en utilisant l'interface graphique définie ci-dessous : un BoxLayout contenant 3 JPanel (Caractéristiques, Fonctions et Trace). Le JPanel Trace contient un JEditorPane, ce qui permet d'avoir des caractères en gras (format HTML).

Exercice 7.2 – Gestion d'un carnet d'adresses

> Écrire le programme Java permettant de réaliser la gestion d'un carnet d'adresses. Les adresses sont mémorisées dans un fichier de type texte. Chaque champ est séparé par un séparateur / comme indiqué ci-dessous. Utiliser un StringTokenizer.

```
Martin/Joël/33, rue des saules/35000/Rennes/02 99 48 47 34
```

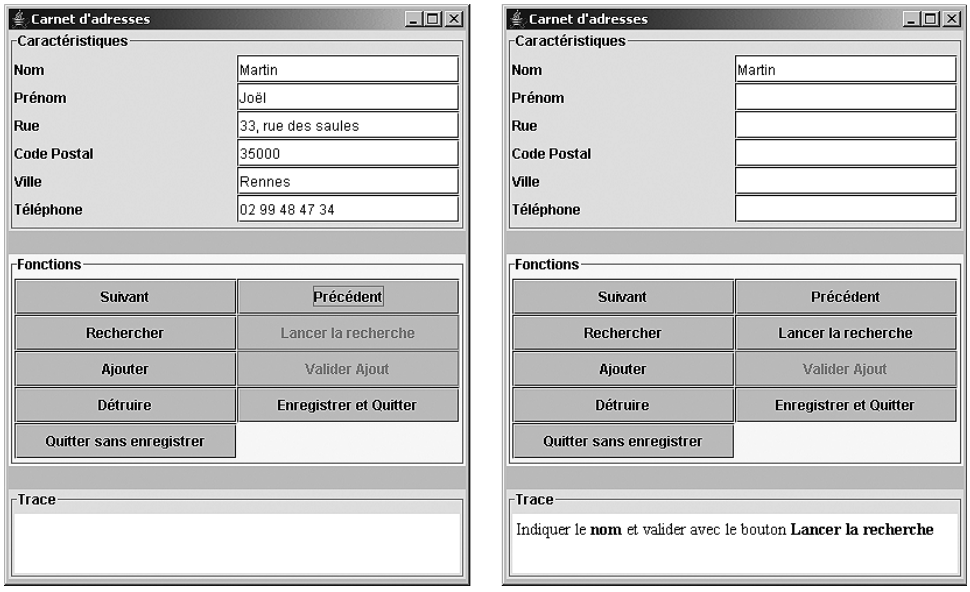


Figure 7.14 — Gestion d'un carnet d'adresses.

On garde une copie du fichier des adresses avant de regénérer le fichier mis à jour après modification.

7.10 LA GESTION DES RÉPERTOIRES

La sélection d'un fichier à traiter pour une application peut se faire en utilisant un objet de type **FileDialog** qui ouvre une fenêtre listant les fichiers du répertoire. On peut changer de répertoire et remonter ou descendre dans la hiérarchie des répertoires. Un bouton "Ouvrir" permet de sélectionner un des fichiers de la liste (et pas forcément de l'ouvrir).

```
// Repertoire.java  choix d'un fichier dans un répertoire

import java.awt.*;     // FileDialog
import javax.swing.*;  // JFrame

public class Repertoire extends JFrame {
  FileDialog fd;
  String     nomRep, nomFichier;

  Repertoire() {
    fd = new FileDialog (this, "Sélection d'un fichier");
    fd.setVisible (true); // fenêtre modale : bloque ici jusqu'au choix
    nomRep     = fd.getDirectory();
    nomFichier = fd.getFile();                // nom du fichier choisi
  }
```

```
    public static void main (String[] args) {
      Repertoire rep = new Repertoire();

      System.out.println ("Répertoire : " + rep.nomRep
                          + "\nFichier    : " + rep.nomFichier);

    }  // class Repertoire

  }  // Repertoire
```

La fgure 7.15 indique la fenêtre de dialogue ouverte lors du `fd.setVisible (true)` et qui permet la sélection d'un fichier d'un répertoire.

Figure 7.15 — Fenêtre de dialogue (FileDialog) pour sélectionner un fichier.

Exemple de résultats, après avoir choisi le fichier Repertoire.java et cliqué sur le bouton Ouvrir :

```
Répertoire : C:\TEMP\Repertoire\
Fichier    : Repertoire.java
```

7.11 CONCLUSION

La gestion des flux sous Java est a priori déroutante avec ses noms de classes à rallonges. On y retrouve pourtant les deux grands types de flux : les flux binaires (ou de données) et les flux de type texte. Les objets chargés de la mise en œuvre de ces flux ont plus ou moins de compétence.

Les objets de base ne savent que transmettre des octets (en lecture ou en écriture) et cela pourrait être suffisant pour gérer les flux d'entrée-sortie. D'autres objets sont plus spécialisés et savent lire les valeurs binaires des types primitifs (entiers, réels) ou sont capables de lire une ligne pour les fichiers de type text. On peut même lire ou écrire des objets en Java grâce au concept de sérialisation.

Un cas particulier de flux important concerne les fichiers. On peut faire de l'accès direct sur les deux types de fichier pour lire directement une information sans passer par les précédentes.

On peut aussi lire un flux en provenance d'un site distant.

Chapitre **8**

Les processus (les threads)

8.1 LE PROBLÈME

Un composant appelé Economiseur qui affiche des droites parallèles en mouvement sur l'écran est décrit en page 228. La méthode qui gère ces mouvements est appelée deplacer() ; elle effectue une boucle infinie qui consiste à déplacer et afficher les droites, et à mettre le programme en sommeil pendant un temps appelé délai en millisecondes, ce qui permet à d'autres processus de passer en unité centrale.

Si on lance deux composants Economiseur, le deuxième ne marche pas ; il ne se passe rien pour ce deuxième composant. Le premier composant effectuant une boucle infinie d'affichage suite à l'appel e1.deplacer() ; le deuxième appel e2.deplacer() ne sera jamais atteint.

```
e1.deplacer();                                          // boucle infinie
e2.deplacer();                          // boucle infinie : jamais exécutée
```

Il faudrait trouver un moyen de répartir les passages en unité centrale entre les deux composants du même programme.

```
// PPEconomiseurPb.java  économiseur d'écran
//                       pb pour exécuter 2 économiseurs

import java.awt.*;           // Container
import javax.swing.*;        // JFrame
import mdpaquetage.mdawt.*;  // Economiseur

public class PPEconomiseurPb {
  public static void main (String[] args) {
    JFrame    fr = new JFrame ();
    Container f  = fr.getContentPane();
```

```
        fr.setTitle  ("Mouvement de droites : 2 composants = PB ?? ");
        fr.setBounds (10, 10, 550, 300);
        f.setLayout  (new GridLayout (0, 2, 10, 10));
        Economiseur e1 = new Economiseur (100);
        Economiseur e2 = new Economiseur (10);
        f.add (e1);
        f.add (e2);
        fr.setVisible (true);
        e1.deplacer();  // boucle infinie
        e2.deplacer();  // boucle infinie : jamais exécutée
      }
    }  // PPEconomiseurPb
```

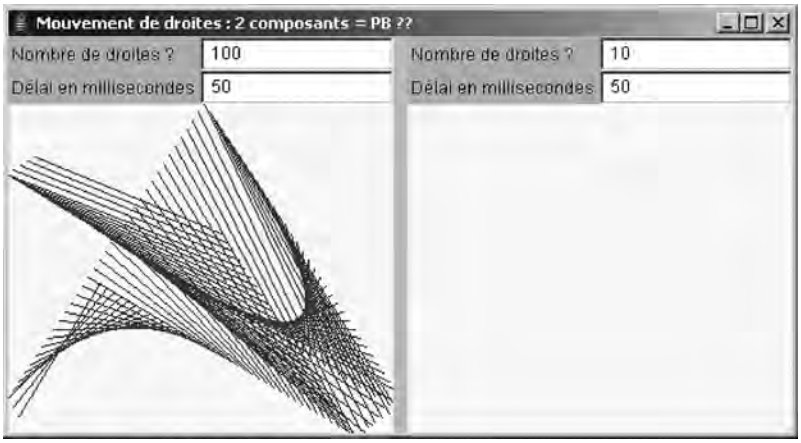


Figure 8.1 — Un seul des deux composants fonctionne.
Le premier est dans une boucle infinie ; le deuxième composant ne peut pas s'exécuter.

8.2 LES THREADS

Les systèmes d'exploitation actuels des ordinateurs exécutent en même temps plusieurs programmes (plusieurs processus). On peut exécuter un programme dans une fenêtre tout en utilisant un éditeur de texte dans une deuxième fenêtre, et en lisant son courrier dans une troisième. La commande Unix "ps" ou le gestionnaire des tâches sous Window listent les processus. En fait, les processus passent en unité centrale à tour de rôle, un à la fois, et ce très rapidement sauf s'ils sont bloqués en attente d'une ressource.

Une application peut être conçue de manière que plusieurs parties de programmes (plusieurs tâches appelées **thread** sous Java) se déroulent en parallèle (en fait se partagent l'unité centrale à tour de rôle). Ces tâches peuvent également se synchroniser, partager ou communiquer de l'information.

Pour résoudre le problème des économiseurs, il faut que chacun des économiseurs soit une tâche qui puisse passer avec les autres tâches de l'ordinateur en unité centrale.

Les animations en Java (les composants animés) font souvent appel aux threads : chaque animation est une tâche programmée de manière indépendante des autres et qui acquiert à son tour et quand c'est nécessaire, l'unité centrale.

Le partage de l'unité centrale entre les tâches d'une application est géré par l'interpréteur Java (la Machine Virtuelle Java) ; il n'est pas à la charge du programmeur. Les threads peuvent néanmoins avoir à se synchroniser ou à partager des données communes.

8.2.1 La classe Thread

La classe **Thread** fournit des méthodes pour gérer les objets Thread : les créer, changer leur priorité définie entre un minimum et un maximum avec une valeur moyenne par défaut. On peut attribuer un nom à un thread, le démarrer, l'interrompre, le détruire, le mettre en attente pendant un nombre de millisecondes, etc. Les principales méthodes sont données succinctement ci-après :

- static **int MAX_PRIORITY** : priorité maximale.
- static **int MIN_PRIORITY** : priorité minimale.
- static **int NORM_PRIORITY** : priorité par défaut.
- **Thread** (Runnable t) : crée un nouveau thread devant exécuter la méthode run() de t.
- void **setName** (String) : définit le nom du thread.
- String **getName** () : fournit le nom du thread.
- int **getPriority** () : fournit la priorité du thread.
- void **setPriority** (int) : change la priorité.
- static **Thread currentThread** () : fournit une référence sur le thread courant.
- static **void sleep** (long n) : le processus est suspendu pour n millisecondes.
- static **int activeCount** : nombre de threads du groupe.
- static **int enumerate** (Thread[] tabThread) : copie dans le tableau les références des threads actifs.
- void **start** () : la JVM appelle la méthode run() du thread.
- **run** () : appel de la méthode run() de l'objet.
- **suspend** () : suspend le thread.
- **resume** () : le thread redevient actif.
- void **destroy** () : détruit le thread.
- void **join** () : attend la fin du thread.
- String **toString** () : fournit certaines des caractéristiques du thread.

8.2.2 La première façon de créer un Thread (par héritage de la classe Thread)

Une classe C1 peut hériter de la classe Thread et redéfinir la méthode run() de Thread. Un objet objC1 de cette classe C1 peut être créé et un thread lancé avec la

méthode start() de la super-classe de cet objet : objC1.start(). La méthode start() de Thread appelle la méthode run() de la classe dérivée C1.

```
class C1 extends Thread {

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

} // class C1

class PP {
  public static void main (String[] args) {
    C1 objC1 = new C1();  // objC1 est de classe C1 et hérite de Thread
    objC1.start();         // start() de la super-classe Thread appelle
                                          // objC1.run() définie en (A)
  }
}
```

ou encore en appelant la méthode start() de la super-classe dans le constructeur :

```
class C1 extends Thread {

  C1 () {
    ...                    // l'objet en construction est de classe C1
    start();               // start() de la super-classe Thread appelle
                  // la méthode run() de C1 définie ci-dessous en (B)
  }

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

} // class C1

class PP {
  public static void main (String[] args) {
    C1 objC1 = new C1();                  // crée et démarre le thread
  }
}
```

8.2.3 La deuxième façon de créer un Thread (en implémentant l'interface Runnable)

Une classe C2 peut implémenter l'interface Runnable qui définit un prototype pour la méthode run() que C2 doit redéfinir. Un objet objC2 de cette classe C2 peut être créé et passé en paramètre du constructeur de Thread pour construire un objet `tache` de type Thread. Si la classe C2 hérite d'une autre classe, cette deuxième façon de faire est la seule possible puisqu'une classe ne peut hériter que d'une seule classe.

```
class C2 implements Runnable {

  public void run () {  // fonction de l'interface Runnable à redéfinir
  ...
  }

} // class C2
```

```
class PP {
  public static void main (String[] args) {
    C2 objC2 = new C2();
    Thread tache = new Thread (objC2);                // crée le Thread
    tache.start();                // le démarre en appelant tache.run()
  }
}
```

ou encore en lançant le thread dans le constructeur :

```
class C2 implements Runnable {

  C2 () {
    ...
    new Thread (this).start();             // crée et démarre le Thread
  }

  public void run () {
  ...
  }

} // class C2

class PP {
  public static void main (String[] args) {
    C2 objC2 = new C2();           // crée l'objet et démarre le Thread
  }
}
```

8.3 LA SOLUTION AU PROBLÈME DES ÉCONOMISEURS

8.3.1 La solution

Si on ne veut pas modifier ou si on ne dispose pas des sources Java de Economiseur, on peut créer une nouvelle classe Eco héritant de la classe Economiseur et implémentant l'interface Runnable comme vu au 8.2.3. Dans ce cas, les deux économiseurs effectuent tous les deux leur boucle infinie mais à tour de rôle. Les deux composants animés Economiseur fonctionnent correctement comme l'indique la figure 8.2.

```
// PPEconomiseur.java  économiseur d'écran
//                     OK pour exécuter 2 économiseurs

import java.awt.*;           // Container
import javax.swing.*;        // JFrame
import mdpaquetage.mdawt.*;  // Economiseur

class Eco extends Economiseur implements Runnable {

  public Eco (Economiseur e) {
    super (e);
    new Thread (this).start();      // pour démarrer le thread dans le
               // constructeur avec la méthode run() de l'objet courant
  }
```

```
  public void run () {
    deplacer();                                  // de la classe Economiseur
  }
} // Eco

class PPEconomiseur {
  public static void main (String[] args) {
    JFrame    fr = new JFrame ();
    Container f  = fr.getContentPane();

    fr.setTitle ("Mouvement de droites, 2 composants = The solution");
    fr.setBounds (10, 10, 550, 300);
    f.setLayout (new GridLayout (0, 2, 10, 10));

    Eco e1 = new Eco (new Economiseur (100));
    Eco e2 = new Eco (new Economiseur (50));  // ou new Eco (e1);
    f.add (e1);
    f.add (e2);
    fr.setVisible (true);

    // si on ne démarre pas le thread dans le constructeur,
    // on peut le démarrer ici; les deux s'excluent.
    // new Thread (e1).start();
    // new Thread (e2).start();
    // testThread (e1, e2);  // pour test de la classe Thread
  }  // main

}  // PPEconomiseur
```

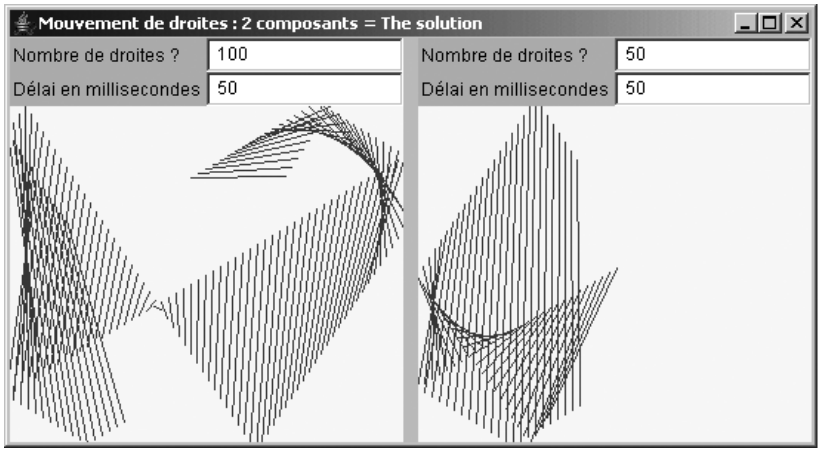


Figure 8.2 — Deux processus (thread) Economiseur se déroulant en parallèle.

Remarque : si on dispose des sources de Economiseur, on peut définir la classe Economiseur comme implémentant l'interface Runnable avec l'instruction suivante :

```
public class Economiseur extends Panel implements Runnable { }
```

On peut alors créer les objets Economiseur :

Economiseur e1 = new Economiseur (100) ;

Economiseur e2 = new Economiseur (50) ;

et lancer les threads :

new Thread (e1).start();

new Thread (e2).start();

ou de même lancer le thread dans le constructeur de Economiseur comme pour Eco ci-dessus. Il faut toutefois renommer la méthode deplacer() de Economiseur qui doit s'appeler run().

8.3.2 Le test de différentes méthodes de la classe Thread

On pourrait tester différentes méthodes de la **classe Thread** en insérant les instructions suivantes en fin de la méthode main() de PPEconomiseur.

```
// ne pas démarrer le Thread dans le constructeur Eco
static void testThread (Eco e1, Eco e2) {
  // différents tests des méthodes de Thread
  // non en rapport avec Economiseur
  new Thread (e1).start();
  Thread t2 = new Thread (e2); // l'objet t2 est créé
  t2.start(); // t2 est démarré

  // différents tests des méthodes de Thread
  // non en rapport avec Economiseur
  try {
    Thread.sleep (5000); // le thread main s'endort pendant 5 secondes
    System.out.println ("t2 suspendu");
    t2.suspend(); // le thread t2 est suspendu
    Thread.sleep (5000); // le thread main s'endort pendant 5 secondes
    t2.resume(); // le thread t2 est réactivé
  } catch (Exception e) {};

  System.out.println (Thread.MIN_PRIORITY + " "
      + Thread.MAX_PRIORITY + " " + Thread.NORM_PRIORITY);
  System.out.println ("currentThread : " + Thread.currentThread());
  int n = Thread.activeCount();  // nombre de thread actifs
  System.out.println ("activeCount   : "   + n);
  Thread[] tabThread = new Thread [n];
  Thread.enumerate (tabThread);  // tableau des threads
  for (int i=0; i < n; i++) {
    System.out.println (i + " : " + tabThread [i]);
  }
}  // testThread
```

Les résultats sur la sortie standard sont les suivants ; ils sont commentés en italiques. Un thread a un nom (fourni par setName() ou par defaut), une priorité entre 1 et 10 (5 par défaut) et appartient à un groupe (le groupe main ci-dessus). Il y a quatre threads : main, un thread d'affichage des fenêtres, et deux threads Eco.

```
                                    les composants t1 et t2 s'exécutent
t2 est suspendu   l'affichage du composant t2 s'arrête pour 5 secondes
t2 reprend                          l'affichage du composant t2 reprend

1 10 5                            priorités minimale, maximale, normale

     le thread courant = thread main de priorité 5 et du groupe main
currentThread : Thread[main,5,main]

                                          nombre de threads actifs = 4
activeCount   : 4

               enumerate un tableau de références des threads actifs
0 : Thread[main,5,main]                                           main
1 : Thread[TaskManager notify thread,10,main]      affichage fenêtre
2 : Thread[Thread-0,5,main]                             thread Eco 1
3 : Thread[Thread-1,5,main]                             thread Eco 2
```

8.4 LES MOTIFS ANIMÉS

Ce paragraphe présente un deuxième composant animé réalisé suivant les concepts de la programmation objet en héritant de la classe Motif qui affiche un motif à partir d'une description sous forme de lignes de texte. L'animation consiste à inverser l'image suivant un axe vertical en son milieu. La méthode setInverser() de Motif déclenche à chaque appel l'inversion du dessin à réaliser.

La classe **MotifAnime** hérite de Motif et implémente l'interface Runnable. On doit redéfinir la méthode run(). Le thread peut comme précédemment être lancé dans le constructeur de MotifAnime ou en utilisant un objet de cette classe (voir PPMotifAnime ci-après). La classe MotifAnime est rangée dans le paquetage mdawt de façon à faciliter sa réutilisation.

```
// MotifAnime.java

package mdpaquetage.mdawt;

public class MotifAnime extends Motif implements Runnable {

  public MotifAnime (Motif m) {
    super (m);
    new Thread (this).start();          // démarre dans le constructeur
                                       //ou après la création de l'objet
  }

  // boucle infinie
  public void run () {
    for (;;) {
      setInverser();      // motif inversé suivant un temps n aléatoire
      int n = 1000 + (int) ((Math.random()*10000)) % 2000;
      attente (n);
    }
```

```
  }

  // le thread s'endort pendant n millisecondes
  void attente (int n) {
    try {
      Thread.sleep (n);
    }
    catch (InterruptedException e) {
      System.out.println (e);
    }
  }

}  // MotifAnime
```

Exemple de résultats : les oiseaux bougent de manière aléatoire.

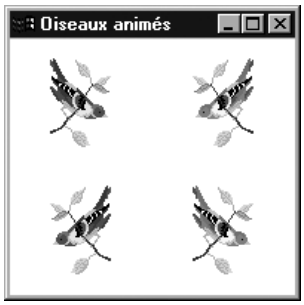


Figure 8.3 — Chaque oiseau est un processus (un thread) qui change de sens de manière aléatoire.

La mise en œuvre du composant MotifAnime est très simple. Il suffit de l'ajouter à un conteneur.

```
// PPMotifAnime.java

import java.awt.*;           // Container, Color, GridLayout
import javax.swing.*;        // JFrame
import mdpaquetage.mdawt.*;  // Motif

class PPMotifAnime extends JFrame {
  Motif m1   = new Motif (MotifLib.oiseau,
                          Motif.tProp, MotifLib.palette1Oiseau);
  Motif m2   = new Motif (MotifLib.oiseau,
                          Motif.tProp, MotifLib.palette2Oiseau);
  Motif m3   = new Motif (MotifLib.rose,
                            Motif.tProp, MotifLib.paletteRose);
  MotifAnime ma1 = new MotifAnime (m1);
  MotifAnime ma2 = new MotifAnime (m2);
  MotifAnime ma3 = new MotifAnime (m2);
  MotifAnime ma4 = new MotifAnime (m1);  // ou m3 pour animer une rose

  PPMotifAnime () {
    Container f = getContentPane();
    setTitle  ("Oiseaux animés");
```

```
    setBounds (20, 20, 200, 200);
    f.setLayout (new GridLayout (0, 2));
    setBackground (Color.white);

    f.add (ma1);
    f.add (ma2);
    f.add (ma3);
    f.add (ma4);
    setVisible (true);
    //new Thread (ma1).start();          // ici ou dans le constructeur
    //new Thread (ma2).start();                    // l'un ou l'autre
    //new Thread (ma3).start();
    //new Thread (ma4).start();
  }  // constructeur PPMotifAnime

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

}  // PPMotifAnime
```

8.5 LA GÉNÉRATION DE NOMBRES

8.5.1 Exemple 1 : deux processus générateurs de nombres non synchronisés

Le programme suivant crée deux threads chargés de générer chacun un nombre fixé de nombres aléatoires. Chaque nombre aléatoire nm généré entre 0 et nMilli est affiché et est utilisé pour mettre le thread en sommeil pendant nm millisecondes. Les threads sont indépendants et génèrent des nombres à leur rythme. La méthode join() de la classe Thread (voir 8.2.1) demande au thread courant d'attendre la fin d'un autre thread.

Le processus courant étant `main`, les instructions suivantes : p1.**join**(); et p2.**join**(); demande au processus main d'attendre la fin de p1 et la fin de p2 pour poursuivre. Sur l'exemple, le message "C'est fini" apparaît bien en dernier après tous les nombres générés par p1 et p2.

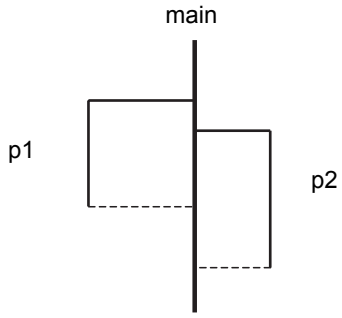


Figure 8.4 — Le thread main.

Il crée les deux threads p1 et p2. Il attend (méthode join) la fin de p1 et la fin de p2 pour se terminer.

Le programme Java ci-dessous crée les deux threads p1 et p2.

```
// PPGenerateurNB.java  2 threads générateurs de nombres
//                      sans synchronisation

import mdpaquetage.utile.*;  // aleat, pause              voir page 366

class GenerateurNb extends Thread {
  int numero;  // numéro du processus
  int nMilli;  // temps d'attente en millisecondes
  int nbNb;    // nombre de nombres à générer

  // le générateur de nombres
  GenerateurNb (int numero, int nMilli, int nbNb) {
    this.numero = numero;  // numéro du générateur
    this.nMilli = nMilli;  // aléatoire entre 0 et nMilli
    this.nbNb   = nbNb;    // nombre de nombres à générer
    start();  // le thread démarre ici dans le constructeur
  }

  // générer nbNb fois un nombre suivi d'une pause
  public void run() {
    for (int i=0; i < nbNb; i++) {
      int nm = Utile.aleat (nMilli);  //(int) (Math.random()*nMilli);
      System.out.println ("numero = " + numero
                          + ", i = " + i + ", nm = " + nm);
      Utile.pause (nm);
    }
  }
}  // class GenerateurNb
```

On crée les deux threads générateurs de nombres. Le thread main attend la fin de ces deux threads pour se terminer.

```
class PPGenerateurNb {
  public static void main(String[] args) {
    GenerateurNb p1 = new GenerateurNb (1, 10, 8);
    GenerateurNb p2 = new GenerateurNb (2, 20, 5);

    // p1.start();  // ici ou dans le constructeur
    // p2.start();

    try {
      p1.join();
      p2.join();
    } catch (Exception e) {
      System.out.println (e);
    }

    System.out.println ("C'est fini");
  }  // main

}  // class PPGenerateurNb
```

Exemple de résultats :

p1 a bien généré 8 nombres (i de 0 à 7) et p2, cinq nombres (i de 0 à 4). La génération des nombres est entremêlée et correspond au passage en unité centrale de chacun des threads. Le thread main écrit le message de fin quand tous les nombres sont générés (rôle de join).

```
numero = 1, i = 0, nm = 2
numero = 2, i = 0, nm = 0
numero = 1, i = 1, nm = 9
numero = 2, i = 1, nm = 0
numero = 2, i = 2, nm = 14
numero = 1, i = 2, nm = 4
numero = 1, i = 3, nm = 2
numero = 1, i = 4, nm = 7
numero = 2, i = 3, nm = 9
numero = 1, i = 5, nm = 3
numero = 1, i = 6, nm = 1
numero = 1, i = 7, nm = 5
numero = 2, i = 4, nm = 15
C'est fini
```

8.5.2 Wait, notify, notifyAll

Tout objet possède un jeton (un verrou) qui peut être disponible ou pris. Un thread peut prendre le jeton d'un objet s'il est disponible ou doit attendre sinon. Les threads peuvent se synchroniser sur le verrou de n'importe quel objet.

La **classe Object**, super-classe de toutes les classes, contient entre autres, les méthodes suivantes :

- wait () : un thread est mis en attente sur cet objet.
- notify () : réveille un thread en attente sur cet objet.
- notifyAll () : réveille tous les threads en attente sur cet objet.

8.5.3 La synchronisation avec sémaphores

Un sémaphore peut être considéré comme une boîte contenant des jetons et sur laquelle on peut réaliser deux opérations :

- prendre un jeton s'il en reste, ce qui permet de continuer, ou attendre l'arrivée d'un jeton s'il n'y en a plus.
- remettre le jeton dans la boîte ce qui permet éventuellement de débloquer un processus en attente.

Ces deux opérations sont traditionnellement appelées P pour "prendre" et V pour "remettre".

(voir Systèmes d'exploitation en bibliographie).

On peut créer une classe Semaphore initalisant une boîte de jetons dans le constructeur et définissant les opérations P et V sur un objet de la classe Semaphore. Les méthodes sont synchronisées (**synchronized**) : un seul thread à la fois peut exécuter P ou V sur un objet donné. Un seul thread à la fois modifie donc la variable nJeton.

```
// Semaphore.java

package mdpaquetage.processus;           // dans le paquetage processus

public class Semaphore {
  private int nJeton = 0;              // nombre de jetons du sémaphore
  String nom;                                    // nom du sémaphore

  public Semaphore (String nom, int nJeton) {
    this.nom    = new String (nom);
    this.nJeton = nJeton;
  }

  public Semaphore (int nJeton) {
    this ("", nJeton);
  }

  // c'est une méthode : P et V pourraient être en minuscules
  synchronized public void P () {                // Puis-je continuer ?
    try {
      // le thread est mis en attente sur l'objet si nJeton = 0.
      // quand il est réveillé, il reteste la valeur de nJeton.
      while (nJeton ==0) wait();
    } catch (InterruptedException e) {
      System.out.println (e);
    }
    nJeton--;                                          // il prend un jeton
  }

  synchronized public void V () {                    // Vas-y, continue !
    nJeton++;                                        // on remet un jeton
    notify();                                        // on le fait savoir
  }

} // Semaphore
```

8.5.4 Exemple 2 : deux processus synchronisés (tampon une place)

Dans certaines applications, les threads doivent se synchroniser. L'un effectue un travail dont l'autre à besoin. C'est typiquement le cas dans la situation dite du producteur consommateur où un thread produit des données et où un autre les exploite (les consomme).

On peut simuler cette application en créant un thread qui met une valeur dans une zone de mémoire commune, et un autre thread qui utilise cette valeur. Les deux processus doivent se synchroniser : le producteur ne doit pas changer la valeur si elle n'a pas été utilisée par le consommateur. Le consommateur ne doit utiliser la valeur qu'une seule fois et faire savoir qu'on peut en mettre une autre dans la zone commune.

Cette synchronisation peut se faire avec deux sémaphores : un sémaphore qui indique si la zone est libre, et un autre qui indique si elle est occupée.

```
libre   : sémaphore initialisé à 1; la zone est libre au départ
occupé  : sémaphore initialisé à 0

processus producteur;              processus consommateur;
  booleen fin = faux;                booleen fin = faux;

  tant que non fin faire             tant que non fin faire
    ...                                ...
    Puis-je produire ?                 Puis-je consommer ?
    P (libre);                         P (occupé);

    déposer les données                prélever les données
    dans zone_commune                  dans zone_commune;
                                       les traiter;

    Vas-y consomme                     Vas-y produis
    V (occupé);                        V (libre);
    ...                                ...
    fin = ??                           fin = ??
  finfaire;                          finfaire;
```

La programmation en Java est très proche du schéma producteur consommateur donné ci-dessus. Un objet de la classe ZoneC définit la zone commune partagée qui se limite ici à une valeur entière. Les objets de la classe Semaphore sont **accessibles des deux threads** car les références des objets sont passées dans les constructeurs de Producteur et Consommateur.

```
// ProdCons1.java  producteur consommateur
//                 avec zone commune partagée de 1 place

import mdpaquetage.processus.*;  // Semaphore, P et V
import mdpaquetage.utile.*;      // pause                 voir page 366

// zone commune partagée par p1 et c1
class ZoneC {
  int valeur = 0;
}

class Producteur extends Thread {
  Semaphore libre;   // référence du sémaphore libre
  Semaphore occupe;  // référence du sémaphore occupe
  ZoneC     entier;  // référence de la zone mémoire commune
  String    nom;     // nom du producteur

  Producteur (Semaphore libre, Semaphore occupe,
                                          ZoneC entier, String nom) {
    this.libre  = libre;
    this.occupe = occupe;
    this.entier = entier;
    this.nom    = nom;
  }
```

```
  public void run() {
    for (int i=0; i < 10; i++) {
      Utile.attente (100);        // le producteur produit à son rythme
      libre.P();
      entier.valeur = i*10;
      System.out.println (nom + " i = " + i
                          + "  valeur = " + entier.valeur);
      occupe.V();
    }
  }

}  // Producteur

class Consommateur extends Thread {
  Semaphore libre;   // référence du sémaphore libre
  Semaphore occupe;  // référence du sémaphore occupe
  ZoneC     entier;  // référence de la zone mémoire commune
  String    nom;     // nom du Consommateur

  Consommateur (Semaphore libre, Semaphore occupe,
                                          ZoneC entier, String nom) {
    this.libre  = libre;
    this.occupe = occupe;
    this.entier = entier;
    this.nom    = nom;
  }

  public void run() {
    for (int i=0; i < 10; i++) {
      Utile.attente (100);     // le consommateur consomme à son rythme
      occupe.P();
      System.out.println (nom + " i = " + i
                          + "  valeur = " + entier.valeur);
      libre.V();
    }
  }

}  //Consommateur
```

On crée les deux sémaphores, la zone de mémoire partagée et les deux threads Producteur et Consommateur qui sont démarrés avec start(). Le thread main attend la fin des deux threads qu'il a créés (join).

```
class ProdCons1 {

  public static void main (String[] args) {
    Semaphore    libre  = new Semaphore (1);
    Semaphore    occupe = new Semaphore (0);
    ZoneC        entier = new ZoneC();
    Producteur   p1  = new Producteur   (libre, occupe, entier, "p1");
    Consommateur c1  = new Consommateur (libre, occupe, entier, "c1");
```

```
        p1.start();                                  // démarrage des threads
        c1.start();
        try {
          p1.join();
          c1.join();
        } catch (InterruptedException e) {
          System.out.println (e);
        }
        System.out.println ("The end\n");
      }

    }  // class ProdCons1
```

Les résultats ci-dessous indique qu'il y a eu synchronisation. Le producteur p1 a produit une valeur et a attendu que le consommateur c1 l'ait prise avant d'en produire une autre. Voir les résultats du 8.5.1 où il n'y a pas de synchronisation.

```
p1 i = 0  valeur = 0
c1 i = 0  valeur = 0
p1 i = 1  valeur = 10
c1 i = 1  valeur = 10
p1 i = 2  valeur = 20
c1 i = 2  valeur = 20
p1 i = 3  valeur = 30
c1 i = 3  valeur = 30
p1 i = 4  valeur = 40
c1 i = 4  valeur = 40
p1 i = 5  valeur = 50
c1 i = 5  valeur = 50
p1 i = 6  valeur = 60
c1 i = 6  valeur = 60
p1 i = 7  valeur = 70
c1 i = 7  valeur = 70
p1 i = 8  valeur = 80
c1 i = 8  valeur = 80
p1 i = 9  valeur = 90
c1 i = 9  valeur = 90
The end
```

8.5.5 Exemple 3 : deux processus synchronisés (tampon de N places)

Dans l'exemple précédent, le tampon entre le producteur et le consommateur ne fait qu'une seule place. On peut utiliser un tampon de N places. Le producteur peut produire jusqu'à N éléments avant de devoir attendre que le consommateur les prenne. Il y a toujours une synchronisation entre les deux processus, mais dans une certaine limite, chacun peut aller à son rythme.

Exercice 8.1 – Producteur Consommateur avec N places

> Modifier le programme Producteur Consommateur de façon que la synchronisation se fasse avec un tampon de N places.

Exemple de résultats pour un tampon de 3 places. Au démarrage, le producteur a pu produire deux valeurs avant que le consommateur ne commence à utiliser les valeurs.

```
p1 i = 0  valeur = 0 0
p1 i = 1  valeur = 1 10
c1 i = 0        valeur = 0 0
p1 i = 2  valeur = 2 20
p1 i = 3  valeur = 0 30
c1 i = 1        valeur = 1 10
c1 i = 2        valeur = 2 20
p1 i = 4  valeur = 1 40
p1 i = 5  valeur = 2 50
c1 i = 3        valeur = 0 30
p1 i = 6  valeur = 0 60
c1 i = 4        valeur = 1 40
p1 i = 7  valeur = 1 70
c1 i = 5        valeur = 2 50
p1 i = 8  valeur = 2 80
c1 i = 6        valeur = 0 60
p1 i = 9  valeur = 0 90
c1 i = 7        valeur = 1 70
c1 i = 8        valeur = 2 80
c1 i = 9        valeur = 0 90
The end
```

8.6 LA SYNCHRONISATION DES (THREADS) BAIGNEURS

La synchronisation des processus informatiques peut souvent être illustrée sur des exemples de la vie courante. L'activité humaine regorge de situations où les êtres humains doivent se synchroniser pour accomplir une activité. Un baigneur se rendant à la piscine doit aussi tenir compte des autres et se synchroniser par rapport à eux. Il doit attendre s'il n'y a plus de panier pour y mettre ses vêtements ; il doit attendre si toutes les cabines sont occupées. Son activité peut se résumer comme suit :

processus **baigneur**
    arriver à la piscine
    P (panier) *Puis-je prendre un panier ?*
    prendre un panier
    P (cabine) *Puis-je avoir une cabine ?*
    choisir une cabine (déshabillage)

V (cabine) *Libérer la cabine*
se baigner
P (cabine) *Puis-je avoir une cabine ?*
choisir une cabine (rhabillage)
V (cabine) *Libérer la cabine*
V (panier) *Rendre le panier*
quitter la piscine

On peut utiliser deux sémaphores initialisés avec le nombre de paniers et le nombre de cabines. Pour séparer la simulation de l'affichage, chaque thread baigneur écrit son numéro et son état dans un tube commun à tous les baigneurs. Un autre thread est lui chargé de la présentation des résultats. Il suffit de changer ce thread pour avoir une présentation différente des résultats. Ci-après, les résultats sont visualisés graphiquement.

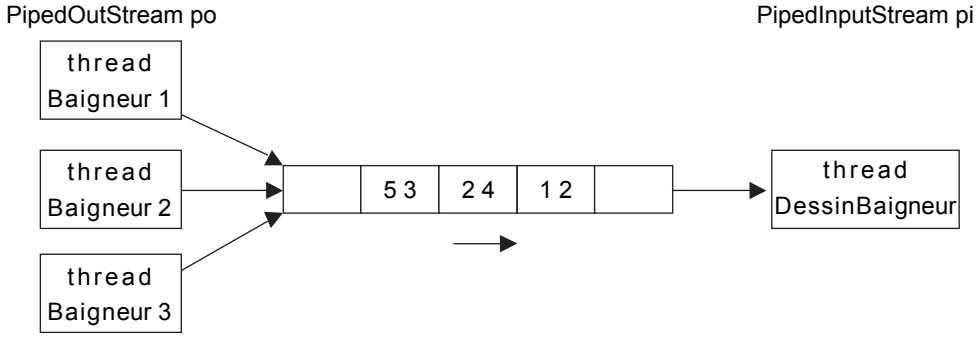


Figure 8.5 — Transmission des résultats des threads par l'intermédiaire d'un tube en mémoire centrale.

Les threads Baigneur écrivent dans le tube leur numéro et leur état (le baigneur 1 est dans l'état 2, le baigneur 2 dans l'état 4, etc.). Le processus DessinBaigneurs lit dans le tube les résultats de la simulation et affiche graphiquement l'état de chaque processus.

Le programme Java suivant définit les processus Baigneur. Le thread DessinBaigneurs est laissé en exercice. Les attributs panier, cabine et po sont static et existent en **un** exemplaire **accessible de tous les objets** de la classe Baigneur.

```
// PPBaigneurs.java Programme Principal Baigneurs

import java.awt.*;              // Container, Color
import javax.swing.*;           // JFrame
import java.io.*;               // PipedOutputStream
import mdpaquetage.utile.*;     // attente
import mdpaquetage.processus.*; // Semaphore, P() et V()

// chaque Baigneur est un thread qui doit se
// synchroniser avec les autres
class Baigneur extends Thread {
```

```
static Semaphore panier;                          // attribut de classe
static Semaphore cabine;                          // attribut de classe
static PipedOutputStream po = null;               // attribut de classe
int    numero;                                    // numéro du Baigneur

// à faire une fois pour initialiser les sémaphores et le tube
static void init (int nbPanier, int nbCabine) {
  panier = new Semaphore (nbPanier);
  cabine = new Semaphore (nbCabine);
  try {
    po = new PipedOutputStream (DessinBaigneurs.pi);
  } catch (IOException e) {
    System.out.println (e);
  }
}

// accède aux variables de classe panier, cabine (de type Semaphore)
// et po (pipe output) : le tube dans lequel chaque Baigneur écrit
// ses résultats (son numéro et son état)
Baigneur (int numero) {
  this.numero = numero;
  start();               // on démarre le thread dans le constructeur
}

// envoi de numero et etat dans le tube po
void message (int etat) {
  try {
    po.write (numero);
    po.write (etat);
  } catch (IOException e) {
    System.out.println (e);
  }
}

// attente aléatoire
void attente (int n) {
  Utile.attente (n);
}

// l'activité d'un Baigneur
public void run () {
  attente (100);
  message (0);  // arrive
  attente (60);

  panier.P();
  message (1);  // prend un panier
  attente (80);

  cabine.P();
  message (2);  // se déshabille
  attente (60);
```

```
        message (3);  // se baigne
        cabine.V();

        attente (150);

        cabine.P();
        message (4);  // se rhabille
        attente (80);

        message (5);  // quitte
        cabine.V();

        attente (40);
        panier.V();

        nb--;
        if (nb == 0) {
          try {
            po.close();
          } catch (Exception e) {
            System.out.println (e);
          }
        }
      }  // run

    }  // class Baigneur
```

Exercice 8.2 – Le dessin de la simulation des baigneurs

Écrire la classe **DessinBaigneurs** qui lit dans le tube les résultats fournis par les threads Baigneur et affiche graphiquement leur état comme indiqué sur la figure 8.6 qui évolue au fil des états des baigneurs.

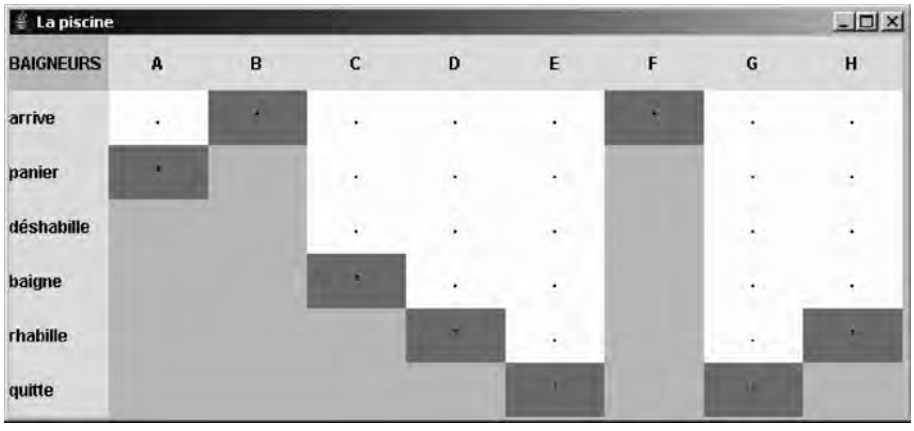


Figure 8.6 — Affichage de l'état des huit threads Baigneur (de A à H) synchronisés par des sémaphores.
B et F arrivent, A a pris un panier, C se baigne, etc.

> Écrire la classe PPBaigneurs qui crée et démarre les threads baigneurs et dessin et attend la fin des processus qu'elle a créés pour se terminer.

8.7 CONCLUSION

Java est un des rares langages de programmation à prévoir explicitement dans sa syntaxe, la gestion de plusieurs tâches (multithread) pour une même application. Celle-ci facilite la mise en œuvre de composants animés qui s'exécutent comme une tâche à part, et qu'on peut ajouter à un conteneur au même titre que les autres composants.

Les threads permettent également la synchronisation de processus qui accomplissent leur tâche tout en dépendant des autres pour le partage de ressources communes. Les threads peuvent se communiquer de l'information en utilisant une zone de mémoire commune (producteur consommateur) ou en passant par un tube : des processus écrivent dans le tube, un autre processus lit les informations contenues dans ce tube.

Chapitre **9**

Les applets

9.1 LES DÉFINITIONS

Une applet est un programme Java exécuté dans une page du Web ou par un applet viewer. L'applet utilise souvent des composants graphiques ou/et des animations. La classe Applet dérive de Panel (qui dérive de Container, voir page 151). Le bytecode de l'applet est téléchargé et exécuté localement dans la page Web par le navigateur qui implémente un interpréteur de bytecode.

Cependant, pour des raisons de sécurité, les applets ont des limites. Une applet ne peut pas accéder à un fichier local (elle ne peut donc ni accéder, ni détruire les fichiers d'un disque local) ; elle ne peut accéder seulement qu'aux fichiers du serveur d'où elle provient.

9.2 LA CLASSE APPLET

Les principales méthodes de la classe Applet sont résumées ci-après (voir la classe URL page 306) :

- AudioClip **getAudioClip** (URL) : fournit un objet AudioClip.
- AudioClip **getAudioClip** (URL, **String**) : fournit un objet AudioClip.
- URL **getCodeBase**() : fournit la base de l'URL du répertoire de l'applet (fichier .class).
- URL **getDocumentBase**() : fournit la base de l'URL du document de l'applet (fichier HTML).
- Image **getImage** (URL) : fournit un objet Image.

- Image **getImage** (URL, **String**) : fournit un objet Image.
- **getParameter** (String) : fournit la valeur du paramètre de nom donné.
- **init** () : chargement d'une applet.
- **play** (URL) : joue le fichier son à l'adresse indiquée.
- **play** (URL, **String**) : joue le fichier son à l'adresse indiquée.
- **resize** (Dimension) : modifie la taille de l'applet.
- **showStatus** (String) : affiche le message dans la fenêtre de status.
- **start** () : l'applet démarre son exécution.
- **stop** () : l'applet arrête son exécution.

9.3 LA CLASSE JAPPLET

La classe JApplet fait partie de la librairie Swing. Elle hérite de Applet. Les composants doivent être ajoutés au Container obtenu avec getContentPane() comme pour les JFrame. Par défault, une JApplet a une mise en page de type BorderLayout.

9.4 LA TRANSFORMATION D'UNE APPLICATION EN UNE APPLET

Si l'application a été conçue sous forme de composants réutilisables, le passage d'une application à une applet se fait très facilement. Si le composant est de type Panel (ou JPanel), il suffit de l'insérer dans une fenêtre de type Frame (ou JFrame) pour avoir une application et de l'insérer dans une classe dérivant de Applet (ou JApplet) pour en faire une applet.

Une applet n'utilise pas la méthode main() pour démarrer mais la méthode init(). La fenêtre principale n'est pas de type Frame mais de type Applet ; elle ne contient pas de méthodes setSize(), setBounds() ou setTitle(). La taille de l'applet est déterminée par les paramètres du fichier HTML lançant l'applet. Le gestionnaire de mise en page de Applet est par défaut celui de Panel dont dérive la classe Applet soit de type FlowLayout. Il est de type BorderLayout pour un JApplet.

9.5 L'APPLET COMPOSANTSDEMO

9.5.1 Avec Applet

L'application développée en figure 5.22, page 172 et comportant différents composants graphiques peut facilement être transformée en une applet. Il suffit d'ajouter l'objet ComposantsDemo au conteneur que constitue l'applet.

```
// AppletComposantsDemo.java

import java.awt.*;           // BorderLayout
import mdpaquetage.mdawt.*;  // classe ComposantsDemo
import java.applet.*;        // Applet
```

```
class AppletComposantsDemo extends Applet {

  public void init () {
    setLayout (new BorderLayout());  // défaut : FlowLayout
    ComposantsDemo p = new ComposantsDemo ();
    add (p, "Center");
  } // init

}  // AppletComposantsDemo
```

9.5.2 Avec JApplet

```
// JAppletComposantsDemo.java

import java.awt.*;           // Container
import javax.swing.*;        // JApplet
import mdpaquetage.mdawt.*;  // classe ComposantsDemo

public class AppletComposantsDemo extends JApplet {

  public void init () {
    Container f = this.getContentPane();
    ComposantsDemo p = new ComposantsDemo ();
    p.setBackground (Color.cyan);
    f.add (p, "Center");  // default : BorderLayout
  } // init

}  // AppletComposantsDemo
```

9.5.3 Mise en œuvre à l'aide d'un fichier HTML

La mise en œuvre de l'applet se fait à partir du fichier HTML *ComposantsDemo.htm* suivant qui définit la taille de l'applet (largeur = 550, hauteur = 650).

```
<HTML>
<HEAD>
<TITLE> Composants démo </TITLE>
</HEAD>
<BODY>
<applet                                          balise de début d'applet
  code = AppletComposantsDemo.class width = 550 height = 650>
</applet>                                         balise de fin d'applet
</BODY>
</HTML>
```

9.6 L'APPLET ECONOMISEUR

La classe Eco permet de considérer le composant Economiseur comme un thread s'exécutant en concurrence avec les autres comme vu dans le chapitre précédent en 8.3.

```
// AppletEconomiseur.java   Applet économiseur d'écran

import java.awt.*;           // Color
import javax.swing.*;        // JApplet
import mdpaquetage.mdawt.*;  // Economiseur
```

```
class Eco extends Economiseur implements Runnable {

  public Eco (Economiseur e) {
    super (e);
  }

  public void run () {
    deplacer();           // de Economiseur
  }

} // Eco
```

La classe **AppletEconomiseur** dérive de JApplet et possède toutes les méthodes de Applet vues ci-dessus. **Applet dérive de P anel**, c'est un **conteneur** auquel on peut ajouter des composants. Le démarrage de l'applet se fait grâce à la méthode **init**() de la classe AppletEconomiseur.

Dans le fichier HTML Economiseur.htm, l'applet se note entre les balises < applet > et </applet >. Le nom de l'applet est repéré par le mot-clé code, la largeur et la hauteur attribuées à l'applet dans la page web sont indiquées par width et heigth. Ces informations sont automatiquement prises en compte par le navigateur. Le mot-clé nombre (de valeur 4) est un paramètre de l'applet qui est pris en compte dans les instructions de l'applet grâce à la méthode **getparameter** ("nombre") qui sur l'exemple fourni 4, et permet de dessiner quatre économiseurs concurrents dans l'espace attribué à l'applet.

Fichier HTML *Economiseur.htm* indiquant la présence d'une applet dans une page web :

```
<HTML>
<HEAD>
<TITLE> Economiseur d'écran </TITLE>
</HEAD>
<BODY>
<applet                                          balise de début d'applet
  code = AppletEconomiseur.class width = 800 height = 500 nombre = 4 >
</applet>                                          balise de fin d'applet
</BODY>
</HTML>
```

La suite du programme Java :

```
class AppletEconomiseur extends JApplet {
  Economiseur[] tabEcono = {     // tableau de 4 composants Economiseur
    new Economiseur (300, Color.blue, Color.yellow, 50, 5,-3,4,6),
    new Economiseur (100, Color.red,  Color.cyan,   30, 6,-3,2,-6),
    new Economiseur (100, Color.blue, Color.yellow, 30, 6,-3,2,-6),
    new Economiseur (100, Color.red,  Color.cyan,   30, 6,-3,2,-6)
  };

  public void init () {
    // les paramètres de l'applet
    int       nbEco  = 2;  // valeur par défaut
    String    sNbEco = getParameter ("nombre");
    if (sNbEco != null) {  // null si nombre n'est pas défini
                           // dans le fichier Economiseur.htm
```

```
      nbEco = Integer.parseInt (sNbEco);  // String -> int
      if (nbEco > 4) nbEco = 4;
      if (nbEco < 0) nbEco = 1;
    }

    // le Layout (défaut : BorderLayout)
    int n = (nbEco ==1) ? 1 h 2;  // 2 composants par ligne
    Container f = this.getContentPane();
    f.setLayout (new GridLayout(0, n, 10, 10 ));

    // les composants
    Eco[] tabEco = new Eco [nbEco];
    for (int i=0; i < nbEco; i++) {
      tabEco[i] = new Eco (tabEcono[i]);
      f.add (tabEco[i]);
    }

    for (int i=0; i < nbEco; i++) {
      new Thread (tabEco[i]).start();  // on démarre les threads
    }
  }  // init

}  // AppletEconomiseur
```

Exemple d'applet :

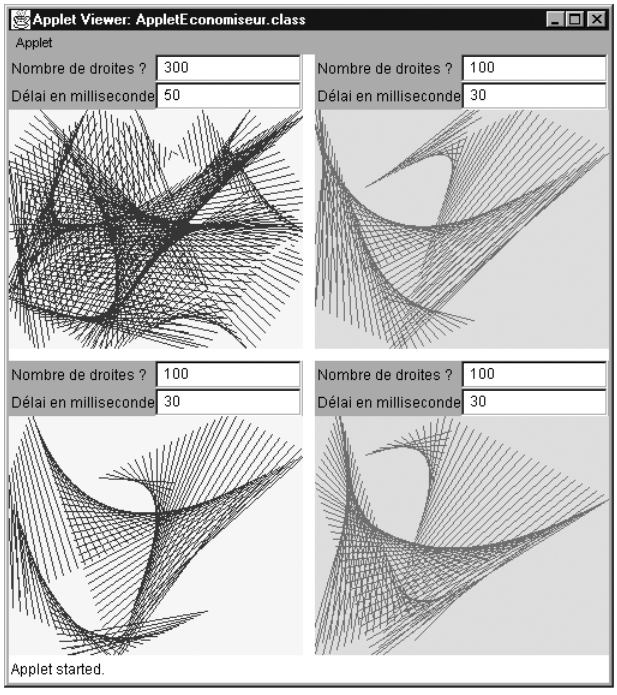


Figure 9.1 — Applet avec quatre composants Economiseur.

L'applet peut être lancée dans un navigateur ou en utilisant le logiciel appletviewer comme ci-dessus.

9.7 L'APPLET MOTIFANIMÉ

De même, le composant **MotifAnimé** défini dans le chapitre précédent dans le cadre d'une application peut facilement être réutilisé dans une applet.

Le fichier HTML *oiseaux.htm* suivant lance l'applet PPOiseauAnime.

```
<HTML>
<HEAD>
<TITLE> Oiseaux animés </TITLE>
</HEAD>
<BODY>
<applet
  code = PPOiseauAnime.class width = 150 height = 150 >
</applet>
</BODY>
</HTML>
```

La classe **PPOiseauAnime** crée une applet contenant quatre MotifAnime représentant des oiseaux qui sont animés comme vu au 8.4 page 323 pour les applications.

```
// PPOiseauAnime.java

import java.awt.*;           // Color, Container
import javax.swing.*;        // JApplet
import java.awt.*;           // GridLayout
import mdpaquetage.mdawt.*;  // Motif, MotifAnime

public class PPOiseauAnime extends Applet {
  Motif oiseau1   = new Motif (MotifLib.oiseau, Motif.tProp,
                                          MotifLib.palette1Oiseau);
  Motif oiseau2   = new Motif (MotifLib.oiseau, Motif.tProp,
                                          MotifLib.palette2Oiseau);
  Motif oiseau3   = new Motif (oiseau1);
  Motif oiseau4   = new Motif (oiseau2);
  MotifAnime ois1 = new MotifAnime (oiseau1);
  MotifAnime ois2 = new MotifAnime (oiseau2);
  MotifAnime ois3 = new MotifAnime (oiseau3);
  MotifAnime ois4 = new MotifAnime (oiseau4);

  // démarrage de l'applet
  public void init () {
    Container f = getContentPane();
    f.setLayout (new GridLayout (0,2));  // 2 composants par ligne
    setBackground (Color.white);
    f.add (ois1);                        // ajouté au conteneur JApplet
    f.add (ois2);
    f.add (ois3);
    f.add (ois4);
  }  // init

}  // PPOiseauAnime
```

Exemple d'affichage de l'applet :

Chaque oiseau est un processus (un thread) qui bouge de manière aléatoire.

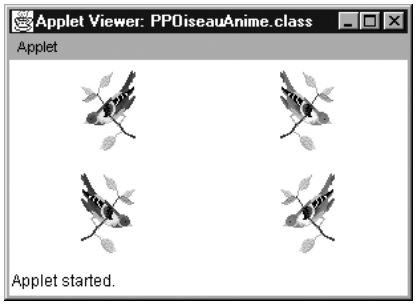


Figure 9.2 — Applet créée à partir de composants Motif.
Chaque oiseau est un thread qui bouge de manière aléatoire (en se retournant).

9.8 L'APPLET JEU DU PENDU

Le jeu du Pendu présenté page 187 peut aussi être très facilement converti en une applet. Les mots à trouver sont par défaut définis dans un tableau initialisé dans la classe Pendu. La classe PenduURL (voir page 309) permet d'initialiser l'ensemble des mots à trouver à partir d'une URL.

On se donne ainsi la possibilité d'initialiser ce tableau de mots à partir d'un fichier local s'il s'agit d'une application, ou d'un fichier provenant du serveur s'il s'agit d'une applet. Ce fichier des mots ne peut provenir que du même site que celui de l'applet (voir limites des applets en 9.1). La classe PenduURL a un constructeur acceptant un paramètre de type URL.

Le fichier HTML *pendu.htm* :

```
<HTML>
<HEAD>
<TITLE> Jeu du Pendu </TITLE>
</HEAD>
<BODY>
<applet code = AppletPendu.class width = 400 height = 450
  fichier = "motssimples.dat">
</applet>
</BODY>
</HTML>
```

L'applet "Jeu du Pendu" :

```
// AppletPendu.java  jeu du pendu avec des mots d'un tableau
//   ou d'un fichier en paramètre de l'applet
//   ou du fichier mots.dat par défaut

import java.awt.*;           // GridLayout
import javax.swing.*;        // JApplet
```

```
import java.net.*;           // URL
import mdpaquetage.mdawt.*;  // Pendu, PenduURL

public class AppletPendu extends JApplet {

  public void init () {
    Container f       = getContentPane();
    String    fichier = getParameter ("fichier");
    if (fichier == null) {
      f.add (new Pendu ());  // BorderLayout
    } else {
      URL url = getCodeBase();
      String nomFichier = url.getFile() + "/" + fichier;
      URL urlFichier;
      try {
        urlFichier = new URL(url.getProtocol(),
                                          url.getHost(), nomFichier);
      } catch (Exception e) {
        urlFichier = null;
      }
      f.add (new PenduURL(urlFichier));  // // BorderLayout
    }
  }
```

Remarque : il est alors très facile de définir un fichier HTML proposant plusieurs niveaux de jeu ou des jeux avec des vocabulaires très spécifiques : les animaux, les adjectifs, etc., voire même avec des mots étrangers (anglais par exemple).

```
<HTML>
<HEAD>
<TITLE> Le jeu du Pendu </TITLE>
</HEAD>
<BODY>
<H1> Le jeu du pendu </H1>
<UL>
<LI> < A HREF ="pendu.htm"  > Mots français simples     </A> </LI>
<LI> < A HREF ="niveau2.htm"> Mots français quelconques </A> </LI>
<LI> < A HREF ="niveau3.htm"> Mots anglais              </A> </LI>
</UL>
</BODY>
</HTML>
```

9.9 L'APPLET MASTERMIND

La réutilisation du MasterMind dans une applet est très facile telle qu'il a été conçu. Il suffit d'appeler dans la méthode init() de l'applet le constructeur du composant MasterMind avec un nombre de couleurs disponibles, un nombre de balles à découvrir et un nombre de coups maximum pouvant être définis dans le fichier HTML référençant l'applet. Le fichier HTML pourrait être modifié pour proposer des niveaux de jeux basés sur ces trois paramètres.

Le fichier *MasterMind.htm* :

```
<HTML>
<HEAD>
<TITLE> Jeu du MasterMind </TITLE>
</HEAD>
<BODY>
<H1> Le MASTER MIND Niveau 1 </H1>
<! nbCoul de 2 à 6; nbBalles de 2 à 6; nbCoups de 5 à 15>
<applet code = AppletMasterMind.class width = 350 height = 450
       nbCoul = 2 nbBalles = 3 nbCoups = 7>
</applet>
</BODY>
</HTML>
```

Le programme Java de l'applet **AppletMasterMind** :

```
// AppletMasterMind.java  Applet MasterMind

import java.awt.*;          // BorderLayout
import javax.swing.*;       // JApplet
import mdpaquetage.mdawt.*; // Balle, MasterMind

public class AppletMasterMind extends JApplet {

  public void init () {
    int NB_BALLES = 5;
    int NB_COUL   = 4;
    int MAX_COUPS = 10;

    String s = getParameter ("nbCoul");
    if (s!=null) NB_COUL   = Integer.parseInt (s);
    s = getParameter ("nbBalles");
    if (s!=null) NB_BALLES = Integer.parseInt (s);
    s = getParameter ("nbCoups");
    if (s!=null) MAX_COUPS = Integer.parseInt (s);

    Container f = this.getContentPane();  // BorderLayout
    f.add (new MasterMind (NB_COUL, NB_BALLES, MAX_COUPS), "Center");
  }

}  // AppletMasterMind
```

Les résultats seraient équivalents aux figures des pages 206 et 217 mais sans titre et sans menu déroulant.

9.10 LE DESSIN RÉCURSIF D'UN ARBRE (APPLET ET APPLICATION)

On veut dessiner un arbre comme schématisé sur la figure 9.3. Pour cela, on dessine un segment de droite. Ce segment se divise récursivement en un nombre aléatoire de segments de droites de longueur 2/3 du précédent et répartis suivant un angle d'ouverture variable. Lorsque le segment de droite devient inférieur à cinq pixels, il est tracé en vert et une fois sur 20, on dessine un fruit rouge.

Figure 9.3 — Dessin d'un arbre de la nature (un pommier et un pin parasol !).

Le composant arbre peut se mettre en œuvre dans une applet.

```
// AppletArbre.java  applet de dessin d'un arbre

import java.awt.*;          // BorderLayout
import javax.swing.*;       // JApplet

// pour une Applet
public class AppletArbre extends JApplet {

  public void init () {
    Container f = this.getContentPane();  // BorderLayout
    f.add (new PanelArbre());  // avec les trois zones de saisie
    //f.add (new Arbre());     // sans les trois zones de saisie
  }

}  //  AppletArbre
```

Ou tout aussi facilement dans une application :

```
// pour une application
class PPArbre extends JFrame {

  PPArbre () {
    Container f = this.getContentPane();
    setTitle  ("Arbres");
    setBounds (20, 20, 500, 500);
    f.add (new PanelArbre());
    //f.add (new Arbre());  // sans zone de saisie
    setVisible (true);
  }

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

}  // PPArbre
```

Exercice 9.1 – Le dessin récursif d'un arbre

> Écrire la classe **Arbre** permettant de dessiner un arbre dans une zone de dessin.
>
> Écrire la classe **PanelArbre** décrivant un composant constitué de trois zones de saisies et d'une zone de dessin de l'arbre comme indiqué sur la figure 9.3.

9.11 LE FEU D'ARTIFICE (APPLET ET APPLICATION)

On veut simuler un feu d'artifice sur écran. Ceci consiste à tracer un segment de droite et à le faire éclater en un nombre aléatoire de segments successeurs qui doivent être tracés à l'étape suivante, chacun de ces successeurs éclatant à nouveau et donnant une nouvelle génération de segments. La couleur des segments est aléatoire. Le composant FeuArt doit pouvoir être utilisé dans une application ou dans une applet comme ci-dessous. Le fichier HTML FeuArt.htm est le suivant :

```
<HTML>
<HEAD>
<TITLE> Feu d'artifice </TITLE>
</HEAD>
<BODY>
<applet code = AppletFeuArt width = 600 height = 300> </applet>
</BODY>
</HTML>
```

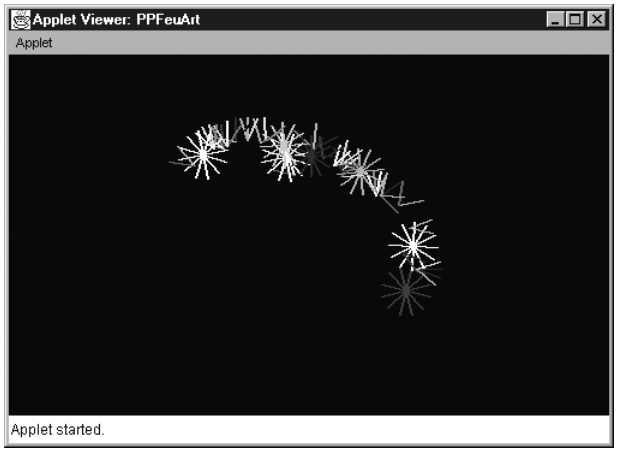


Figure 9.4 — L'applet feu d'artifice.

Exercice 9.2 – Feu d'artifice

> Écrire une classe **Segment** qui définit et dessine un Segment de droite connaissant sa longueur, son point origine, l'angle en degrés du segment de droite et sa couleur.

Écrire la classe suivante **class F euArt extends Can vas implements Runnable** {} qui dessine un feu d'artifice dans un composant Canvas et implémente l'interface Runnable. On utilise deux listes (voir chapitre 4) : la liste courante des segments à tracer, et la liste suivante des segments à tracer pour le prochain tour. A l'étape suivante, la liste suivante devient alors la liste courante, et on recrée une liste suivante, etc.

Écrire un programme de mise en œuvre de ce composant FeuArt dans une application autonome et dans une applet.

9.12 LE SON ET LES IMAGES

On peut facilement afficher des images et déclencher la lecture de fichiers de sons à partir d'une application Java ou d'une applet grâce aux classes **Image** et **AudioClip**.

9.12.1 Les images

Une image est caractérisée par la classe Image ; on peut déclarer des variables comme suit : `Image photo1`.

- Pour une applet, la méthode **getImage**() de la classe Applet (voir § 9.2) crée un objet image à partir d'une adresse URL (l'adresse de base d'où provient l'applet, plus le nom du fichier contenant l'image).
- Pour une application, il faut faire appel à une boîte à outils (classe Toolkit) qui définit une méthode `Image` **`getImage`**`(String)`.

Dans les deux cas la méthode **drawImage**() de la classe Graphics dessine l'image dans son espace ; les dimensions peuvent être changées en précisant une nouvelle largeur et une nouvelle hauteur de l'image.

9.12.2 Les sons (bruit ou parole)

La sonorisation se fait par lecture d'un fichier de sons. Le format du fichier son reconnu par le navigateur dépend de la version de la machine virtuelle Java. La version 1 n'accepte que les fichiers au format "au" ; la version 2 accepte aussi les fichiers au format "wav".

- Pour une applet, la méthode **getAudioClip**() de la classe Applet (voir § 9.2) crée un objet AudioClip à partir d'une adresse URL (l'adresse de base d'où provient l'applet, plus le nom du fichier contenant le son).
- Pour une application, il faut utiliser la séquence d'instructions suivantes :

```
AudioClip son = null;
URL url = ClassLoader.getSystemResource ("gong.au");
try {
  son = (AudioClip) url.getContent();
} catch (Exception e) {
  System.out.println (e);
}
```

Les méthodes play() et stop() de AudioClip permettent de déclencher le son ou de l'arrêter.

9.12.3 Un exemple d'utilisation d'images et de sons

On veut créer une application ou une applet présentant des images (voir figure 9.5) pour lesquelles on peut déclencher une sonorisation en cliquant sur l'image. Si le curseur de la souris sort de l'espace de l'image, le son doit s'arrêter.

Figure 9.5 — Affichage d'images dans une application ou dans une applet.
Un clic sur une image déclenche la lecture d'un fichier de son (bruit ou commentaires sur l'image) qui s'arrête si le curseur de la souris sort de l'espace de l'image.

On crée le composant ImageSon qui a deux attributs : un objet Image et un objet AudioClip (son). On ajoute à cet objet un écouteur réagissant à deux événements de la souris : un clic de la souris qui déclenche la lecture du fichier son : son.play(), et un événement de sortie de l'espace de l'image qui arrête le son : son.stop(). La méthode paint() ajuste l'image tout en gardant ses proportions dans l'espace qui lui a été attribué. La classe **ImageSon** est décrite ci-dessous.

```
// AppletImages.java  Applet ou Application
//                    de présentation d'images et de sons

import java.awt.*;           // Panel
import javax.swing.*;        // JPanel
import java.awt.event.*;     // MouseAdapter
import java.applet.*;        // AudioClip
import java.net.*;           // URL

class ImageSon extends JPanel {           // c'est un nouveau composant
  Image     photo;                      // constitué d'un attribut Image
  AudioClip son;                    // et d'un attribut AudioClip (son)

  ImageSon (Image photo, AudioClip son) {
    this.photo = photo;
    this.son   = son;
    addMouseListener (new JouerSon());         // écouteur du composant
  }
```

```
  // pas de fichier son
  ImageSon (Image photo) {
    this (photo, null);  // pas de son
  }

  // on ajuste l'image à l'espace attribué (par le Layout)
  public void paint (Graphics g) {
    int l    = photo.getWidth  (this);
    int h    = photo.getHeight (this);
    int largeur = getSize().width;
    int hauteur = getSize().height;

    // ajuster la taille de l'image à la taille disponible
    // en respectant les proportions
    double rl = l/(double)largeur;
    double rh = h/(double)hauteur;
    double r  = Math.max (rl, rh);
    g.drawImage (photo, 1, 1, (int)(l/r), (int) (h/r), this);
  }  // paint

  // jouer le fichier son lorsqu'on clique sur l'image;
  // l'arrêter quand le curseur sort de l'espace de l'image
  class JouerSon extends MouseAdapter {                // voir page 199
    public void mouseClicked (MouseEvent evt) {
      if (son != null) son.play();
    }
    public void mouseExited (MouseEvent evt) {
      if (son != null) son.stop();
    }
  }

}  // class ImageSon
```

La classe suivante **PPImages** présente une application utilisant le composant **ImageSon**.

```
class PPImages extends JFrame {  // Programme Principal Images

  PPImages (String nomrepimages, String nomrepaudio) {
    // les sons
    AudioClip son1 = null;
    AudioClip son3 = null;
    String fichierson1 = nomrepaudio + "oie.au";
    String fichierson2 = nomrepaudio + "mouton.au";
    URL url1 = ClassLoader.getSystemResource (fichierson1);
    URL url2 = ClassLoader.getSystemResource (fichierson2);
    try {
      son1 = (AudioClip) url1.getContent();
    } catch (Exception e) {
      System.out.println ("pas de fichier son " + fichierson1);
    }
    try {
      son3 = (AudioClip) url2.getContent();
```

```
    } catch (Exception e) {
      System.out.println ("pas de fichier son " + fichierson2);
    }

    // les images (pas d'exception comme pour le son)
    Toolkit tk = getToolkit();
    Image photo1 = tk.getImage (nomrepimages + "oie.gif");
    Image photo2 = tk.getImage (nomrepimages + "hamac.jpg");
    Image photo3 = tk.getImage (nomrepimages + "mouton.jpg");

    Container f = getContentPane();
    setTitle  ("Images et sons");
    f.setLayout (new GridLayout(0, 3, 10, 10));
    setBackground (Color.white);
    setBounds (10, 10, 600, 200);
    f.add (new ImageSon (photo1, son1));
    f.add (new ImageSon (photo2));
    f.add (new ImageSon (photo3, son3));

    setVisible (true);
  }

  public static void main (String[] args) {
    String nomrepimages = (args.length == 0) ? "." : args[0];
    String nomrepaudio  = (args.length == 1) ? "." : args[1];
    new PPImages (nomrepimages, nomrepaudio);
  }

}  // PPImages
```

La classe suivante **AppletImages** présente une applet utilisant le composant **ImageSon**. Cette applet a deux paramètres au niveau du fichier HTML repérés par les mots-clés `nom` pour l'image et `audio` pour le son.

```
public class AppletImages extends JApplet {
  public void init () {
    // audio et nom valent null si non défini
    String nom = getParameter ("nom");
    if (nom == null) {
      add (new Label ("pas de photo mentionnée"), "North");
    } else {
      String audio = getParameter ("audio");
      Image  photo = getImage (getCodeBase(), nom);
      AudioClip son = null;
      if (audio != null) son = getAudioClip(getCodeBase(), audio);

      Container f = getContentPane();  // BorderLayout
      setBackground (Color.white);
      f.add (new ImageSon(photo, son), "Center");
    }
  }
}  // AppletImages
```

Le fichier HTML images.htm suivant présente trois images dont les deux premières peuvent être sonorisées par un clic de souris. Bien sûr, les fichiers sons pourraient contenir un bruit approprié ou un commentaire sous forme de parole (fichier en .wav par exemple).

```
<HTML>
<HEAD>
  <TITLE> images </TITLE>
</HEAD>
<BODY>
<HR>
<H1> Exposition </H1>
<HR>
<applet  code = "AppletImages" width = 200 height = 150
        nom = "images/oie.gif" audio = "audio/oie.au"> </applet>
<applet  code = AppletImages width = 200 height = 150
nom = "images/hamac.jpg" audio = "audio/hamac.au"> </applet>
<applet  code = AppletImages width = 200 height = 150
nom = "images/mouton.jpg" audio = "audio/mouton.au"> </applet>

<HR>
</BODY>
</HTML>
```

9.13 LE MOUVEMENT PERPÉTUEL DES BALLES (APPLET ET APPLICATION)

On veut créer un jeu de balles rebondissant sur les bords d'un rectangle dont le coin supérieur gauche est fixé, mais dont la largeur et la hauteur varient entre une taille maximale (celle de la fenêtre initiale) et une taille minimale (cinq fois la taille d'une balle par exemple). Le nombre de balles et la taille des balles est variable. Les couleurs des balles sont aléatoires. On utilise le composant Balle défini en page 193. La mise en œuvre de l'applet peut se faire à l'aide du fichier HTML Balles.htm suivant :

```
<HTML>
<HEAD> <TITLE> Balles en mouvement </TITLE> </HEAD>
<BODY>
<applet code = AppletBalle width = 400 height = 400
        nombre = 12 diametre = 20 vitesse = 12>
</applet>
</BODY>
</HTML>
```

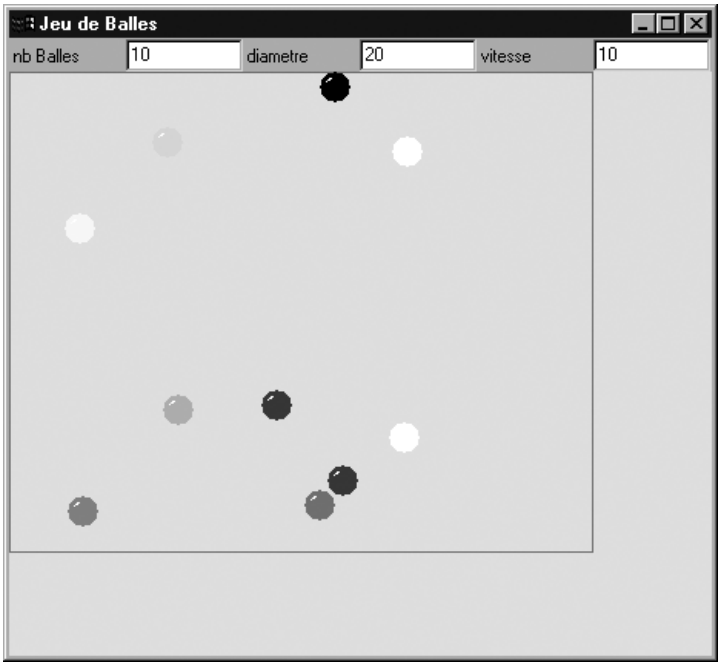


Figure 9.6 — Mouvement perpétuel de balles.

Exercice 9.3 – Mouvement perpétuel de balles

Écrire une classe **Cadre** qui permet d'initialiser les dimensions du cadre, de modifier sa taille et de l'afficher dans un contexte graphique.

Écrire une classe **ZoneBalles** extends Canvas {} qui définit la zone de dessin des balles ; le constructeur de la classe est le suivant : class ZoneBalles (Cadre cadre, BalleMobile[] tabBalles) {}. Il a en paramètres, la référence du cadre mobile, et un tableau des balles mobiles à dessiner.

Écrire une classe **PanelBalle extends JP anel implements Runnable** {} qui définit 3 zones de saisie comme sur la figure 9.6, et une zone de mouvement des balles. Cette classe implémente la fonction run() de l'interface Runnable.

Écrire un programme de mise en œuvre du composant PanelBalle dans une applet et dans une application. Plusieurs composants PanelBalle doivent pouvoir fonctionner en même temps.

Ajouter une sonorisation lorsqu'il n'y a que deux balles et que ces deux balles s'entrechoquent.

9.14 LE TREMPOLINE (APPLET ET APPLICATION)

On veut créer un composant animé représentant un sauteur au TREMPOLINE. Celui-ci se déplace verticalement dans l'espace qui lui est attribué en gardant une coordonnée en x constante. Les différentes positions du sauteur sont mémorisées sous forme d'images dans un tableau d'images. Le sauteur monte en 16 étapes (16 affichages d'images), tourne sur lui-même une ou plusieurs fois suivant la hauteur, et redescend en 16 étapes. La hauteur atteinte à chaque montée descente varie suivant un coefficient vitesse mémorisé dans un tableau de 12 éléments. Au début, la hauteur atteinte est faible (phase d'élan), puis elle devient importante et ensuite décroît.

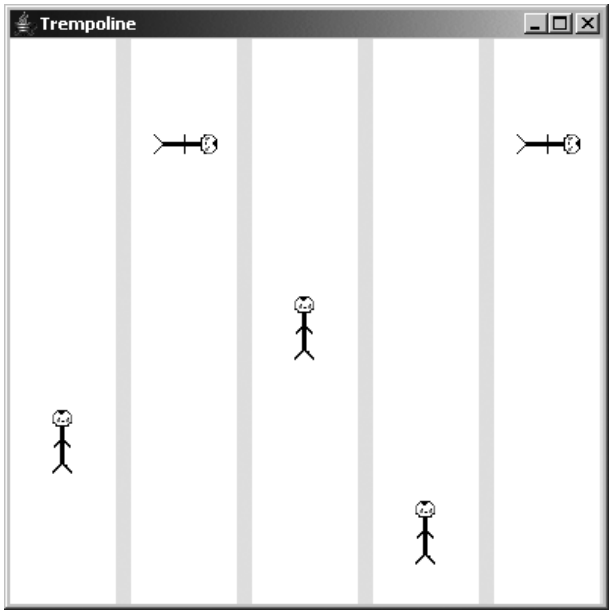


Figure 9.7 — 5 processus (thread) sauteurs au trempoline.

Les différents étapes de la montée descente sont représentées par des fichiers contenant les images à dessiner. Il y a cinq fichiers d'images très stylisées représentées sur la figure 9.8.

md1.gif	md2.gif	md3.gif	md4.gif	md5.gif
				

Figure 9.8 — Les différentes positions d'un sauteur au trempoline.

Exercice 9.4 – Trempoline

> Écrire la classe **class SauteurTrempoline extends JP anel implements Runnable ()** qui définit un composant simulant un sauteur au trempoline.
>
> Écrire une application et une applet mettant en œuvre dans une même fenêtre plusieurs sauteurs au trempoline régulièrement espacés sur l'axe des x.

9.15 LA BATAILLE NAVALE (APPLET ET APPLICATION)

Le jeu de bataille navale consiste à effectuer des tirs sur un espace contenant des bateaux de une case à l'aide de la souris. Deux bateaux peuvent se toucher.

- Si la case visée ne contient pas de bateaux et si les cases avoisinantes n'en contiennent pas, le programme dessine toutes les cases en jaune ; la case visée contient le message "Raté" et les cases avoisinantes Rien.
- Si la case visée ne contient rien mais une case avoisinante contient un bateau, le message "EnVue" s'affiche dans la case visée qui est coloriée en jaune.

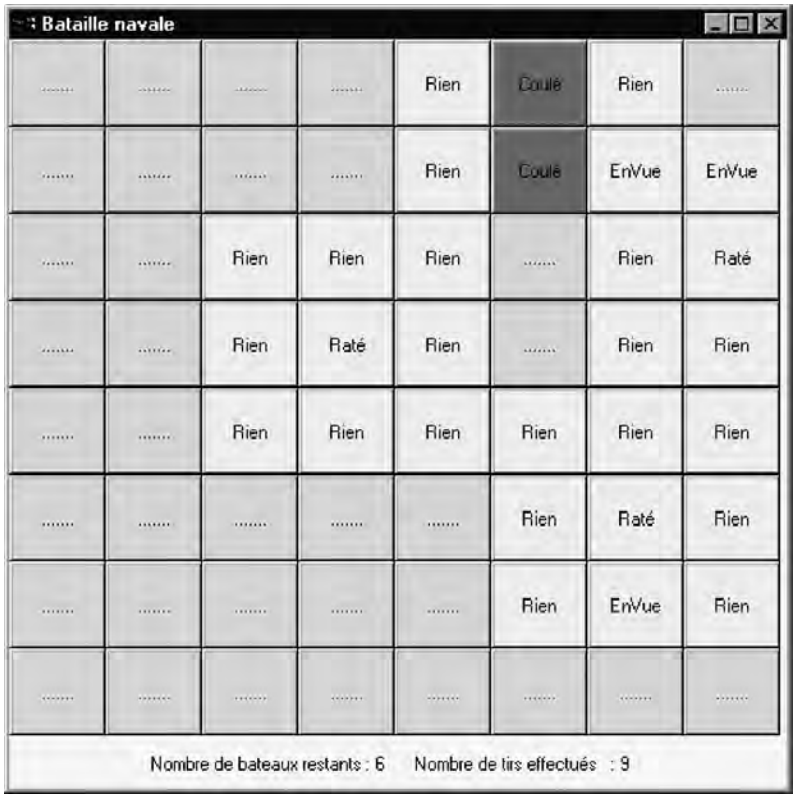


Figure 9.9 — Jeu de la bataille navale.

- Si la case contient un bateau, le message Coulé s'affiche et la case est coloriée en rouge ;
  - ➤ si les cases avoisinantes ne contiennent pas de bateau, celles-ci sont coloriées en jaune.
  - ➤ si un autre bateau est en vue, le message "Coulé" est remplacé par le message "EnVue", mais la case reste en rouge indiquant un bateau coulé.

  Remarque : la couleur jaune indique une absence de bateaux. La couleur rouge indique un bateau coulé. La couleur bleue indique une zone à explorer.

Exercice 9.5 – Bataille navale

> Écrire un composant **BatNav** qui crée un espace de 8x8 cases, place aléatoirement 8 bateaux de une case, et prend en compte les clics de la souris sur cet espace.
>
> Écrire une application et une applet mettant en œuvre la classe BatNav.
>
> Ajouter une sonorisation permettant de fournir les messages parlés "Coulé" et "EnVue".

9.16 CONCLUSIONS SUR LES APPLETS

Si l'application a bien été conçue comme un composant graphique contenu dans un Panel (ou JPanel), son utilisation sous la forme d'une application ou d'une applet est très facile. Il suffit d'ajouter le composant à une fenêtre de type Frame (ou JFrame) pour une application et à un conteneur dérivant de Applet (ou JApplet) pour une applet pouvant être prise en compte par un navigateur dans une page web. Certains composants peuvent être animés.

Si on souhaite pouvoir animer plusieurs composants dans une applet, il faut que chaque composant soit un thread qui sollicite l'unité centrale concurremment avec les autres threads et les autres processus de l'ordinateur.

Tout comme les applications, les applets peuvent avoir une interface graphique réagissant aux événements de la souris ou du clavier ; elles peuvent afficher des images et utiliser des fichiers sons.

9.17 CONCLUSION GÉNÉRALE

Java est un langage complet permettant de développer des applications exécutées localement et pouvant accéder aux ressources de l'ordinateur et à des sites distants. Java permet également l'insertion dans des pages web de programmes téléchargés qui sont exécutés localement mais qui, pour des raisons de sécurité, sont limités dans leur accès aux ressources de l'ordinateur local.

Les bibliothèques de classes permettent de développer des applications variées ayant des interfaces graphiques conviviales. Les notions fondamentales d'héritage et de polymorphisme permettent de developper des classes très générales qui sont des d'outils pouvant être réutilisés dans d'autres applications et sur d'autres ordinateurs indépendamment du système d'exploitation.

Chapitre **10**

Annexes

10.1 LA CLASSE MOTIFLIB : EXEMPLES DE MOTIF

Exemples de descriptions de dessins pouvant être créés à partir de la classe Motif :

```
// MotifLib.java  librairie de Motif disponibles

// champignon
// paletteV
// paletteH
// hirondelle
// canard
// oiseau
// clé de sol
// rose

package mdpaquetage.mdawt;

import java.awt.*;  // Color

public class MotifLib {

  public static final String[] champignon = { // 22 x 21
```

```
    "                    ",
    "         KKK        ",
    " KLLKKKKKKKLLKKKKKKK ",
    " KKKKKKKKKKKKKKKLLKK ",
    "   LLDLDLLLDLLDLDL   ",
    "         LLL        ",
    "         LLL        ",
    "        LLLLL       ",
    "         LLL        ",
    "         LLL        ",
    "         LLL        ",
    "         LLL     FF ",
    "   FFF   LLL    F   ",
    "  F   F  LLL   F     ",
    "       FLLLL   F     ",
    "        FLLLL F   F  ",
    "  FF   LFLLLL F  F   ",
    "    FF LLFLLLF  F    ",
    "     EFLEFELEFEF     ",
    " EEEEEEEE EEEE FEEE  ",
    "                    "
  };

  // dessin de la palette verticale des couleurs
  public static final String[] paletteV = {
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM",
    "ABCDEFGHIJKLM"
  };

  // dessin de la palette horizontale des couleurs
    public static final String[] paletteH = {
    "AAAAAAAAAAAAAAAAAAAA",
    "BBBBBBBBBBBBBBBBBBB",
    "CCCCCCCCCCCCCCCCCCC",
    "DDDDDDDDDDDDDDDDDDD",
    "EEEEEEEEEEEEEEEEEEE",
    "FFFFFFFFFFFFFFFFFFF",
    "GGGGGGGGGGGGGGGGGGG",
```

```
  "HHHHHHHHHHHHHHHHHHHH",
  "IIIIIIIIIIIIIIIIIII",
  "JJJJJJJJJJJJJJJJJJJJ",
  "KKKKKKKKKKKKKKKKKKK",
  "LLLLLLLLLLLLLLLLLLL",
  "MMMMMMMMMMMMMMMMMMM"
};

  public static final String[] hirondelle = { // 44 x 41
  "                                            ",
  "                                            ",
  "                                            ",
  "                                            ",
  "                        BBBBBBBBBBCCCCBBBB  ",
  "                      BBBBBBBBBBBBCCCCBBBBB ",
  "                  BBBBBBBBBCCCCCCCCCC       ",
  "                 BBBBBBBBBCCCCCCCCC         ",
  "                BBBBBBBBBBBBBCCCC           ",
  "               BBBBBBBBBBBBBBCCC            ",
  "               BBBBBBCCCCCCCC               ",
  "               BBBBBBCCCCCCC                ",
  "       BBBBB  BBBBBBBBCC                    ",
  "      BBBBBB  BBBBBBBBC                     ",
  "     BBBKKBBBBCCCCCCBBB                     ",
  "    BBBBKKBBBBCCCCCCBBBB                    ",
  "   BBBKKKKKKBBCCCCBBBBBBB                   ",
  "  BBBBKKKKKKBBCCCCBBBBBBBB                  ",
  "      KKKKKKIIIICCCCCCCCBBB                 ",
  "       KKKKKIIIICCCCCCCCBBBB                ",
  "        IILLLLIILLLLCCCCCBBBBB              ",
  "         ILLLLIILLLLCCCCCBBBBBB             ",
  "              LLLLLLIICCCCCCBBBBB           ",
  "               LLLLLIICCCCCCBBBBBB          ",
  "                    IIIIIIBBCCCCBBBBBBB     ",
  "                     IIIIIBBCCCCBBBBBBBB    ",
  "                      IIBBBBBB              ",
  "                      IIBBBBBB              ",
  "                      BBBBBBBBBB            ",
  "                      BBBBBBBBBB            ",
  "                      BBBB  BBBB            ",
  "                      BBBB  BBBB            ",
  "                      BBBB     BB           ",
  "                      BBBB      BB          ",
  "                      BBBB       BB         ",
  "                      BBBB        BB        ",
  "                      BBBB                  ",
  "                      BBBB                  ",
  "                      BB                    ",
  "                      BB                    ",
  "                      BB                    ",
  "                      B                     "
};
```

```
public static final String[] canard = { // 50 x 22
  /*"          1         2         3         4         "*/
  /*"01234567890123456789012345678901234567890123456789"*/
    "         AAAA                                      ",
    "        AAAAAA                                     ",
    "       AAAAAAAA                                    ",
    "       AAHAAAAA                                    ",
    "      FAAAAAAAA                                    ",
    "    FFFFAAAAAAA                                    ",
    "  FFFFFAAAAAAA                                     ",
    "         AAAA                                      ",
    "         AAAA                                      ",
    "          AAA                                      ",
    "          AAA                                      ",
    "         AAAAA                                     ",
    "        AAAAAA    DDDDD                           ",
    "        BBBAAAA CDDDDDDDDD                        ",
    "       BBBBBBBCCCDDDDDDDDDD                       ",
    "      BBBBBBBCCCDDDDDDDDDDDDD                     ",
    "     BBBBBBBCCCDDDEEEEDDDDEEEEEEE                 ",
    "     BBBBBBBCCCDEEEEEEEEDDEEEEEEE                 ",
    "     BBBBBBCCCCCEEEEEEEEEEEEEEEE                  ",
    "      BBBBCCCCCCCEEEEEEEEEEEEE                    ",
    "GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG",
};

public static final Color[] paletteCanard = {
    Color.darkGray,
    Color.WHITE,
    Color.ORANGE,
    Color.GREEN,
    Color.RED,
    Color.YELLOW,
    Color.BLUE,
    Color.BLACK
};

// un petit oiseau de toutes les couleurs
public static final String[] oiseau = { // 41 x 60
/*"         1         2         3         4 " */
/*"12345678901234567890123456789012345678901" */
  "                                       C  ", /* 1 */
  "                        A             CC D",
  "                       BA             CCDD",
  "                      BBA            CCCDD",
  "                   BABBAA            CCDDD",
  "                  BBABAA            CCCDD ",
  "                 BABBABA            CCCD  ",
  "                 BABABBA           CCCCD  ",
  "                BBBABAAA           CCCCD  ",
  "                BBABBBA           CCCC    ", /* 10 */
```

```
  "              BABBABA            CCCC    ",
  "              ABBAAA            CCCC     ",
  "BBB           ABA              CCCCE   F ",
  " ABBBBB      A                 CCCEE  FF ",
  "  ABABBB     A                CCCEEEFFFC ",
  "   ABABBB    A               CCCEEEFFFC  ",
  "   ABBABBB   A              GGGEEFFFFCC  ",
  "    AABBMMM J             GGGGEFFHHFCC   ",
  "      MMAAMMJ            GGGGFHFFFHCC    ",
  "     MAAAMAA J          GGGGFFFHFFCC   DD", /* 20 */
  "    MMAMMAA   J       CGGGFHFFFHFCCC  DDC",
  "   MAAMAAA    J      CCGGFFFHFFCCCCE DDC ",
  " MMAAAAA       J    CCCCFFFFHFFCCCEEDDCC ",
  "   AAA          J  CCCIIFFFHFFCCCCEEDDC  ",
  "                J CCCGGIGFFHFCCCCEEDDCC  ",
  "                 JCFFDDGIFHFCCCCEEEDCC   ",
  "                DJFFDDDDGIFDDDCEEEEDC    ",
  "               EDDJFFDDDGGDDDCCEEEECC    ",
  "              EDDFJFFFDDGIDDDCEEGEEC     ",
  "             EEDIFFJFDDDGIDDCEGGGE       ", /* 30 */
  "           EEEGIFFJFFDDIIDCEGKKE         ",
  "        LLLLEEGIIIFFJDDGIDCEGKKG         ",
  "      ELLLLLLGIIIIFFJFDIIEEKKKGK         ",
  "     ELLEELLLGIIIIFFFJIIEKKKGGKKK        ",
  "     ELEEEELGIIIIFFFIJJKKKGGG   KK       ",
  "    EEEEEEELGIIIIIIIIIKJGGGG    K        ",
  "    EEEFEEEEGGIIIIIIIIKKJGG    K         ",
  "    EEEEEEEEGGGIGIIIIIIGJJ    KK         ",
  "   GGEEEEEEGGEGGGGGIIGG  JJ   K          ",
  "  GGGGEEEEGGEEEGGEGGG     JJKK           ", /* 40 */
  "          EEEEEEEG         KJJ           ",
  "                           KKJ           ",
  "                             JJ          ",
  "                              JJ         ",
  "                              JJ         ",
  "                               JJ        ",
  "                             AA JJ       ",
  "                           AA    JJJ     ",
  "                          A        JJJJ  ",
  "                     BBBBB A          JJJJ", /* 50 */
  "                    BBAABBAA            JJ",
  "                   BBBBBBABA              ",
  "                   BAABAABBA              ",
  "                  BABBABAAAA              ",
  "                  BBBABBBAA               ",
  "                  BAABAAAAA               ",
  "                 BBBABBBAA                ",
  "                 BBABBAAA                 ",
  "                BBBBBAAA                  ",
  "                   AAA                    "  /* 60 */

/*"         1         2         3         4 " */
```

```
/*"12345678901234567890123456789012345678901" */
};

// une palette pour l'oiseau ci-dessus (le mâle !)
public static final Color[] palette1Oiseau = {
  new Color (0, 220, 0),   // A : vert des feuilles
  new Color (230, 230, 0), // B : jaune des feuilles
  Color.blue,              // C : dessus de queue
  new Color (150, 100, 0), // D : dessous de la queue
  Color.magenta.darker(),  // E : autour de l'oeil
  Color.black,             // F : noir de l'aile
  Color.orange,            // G : dessous du ventre
  Color.white,             // H : points blanc de l'aile
  new Color (240, 240, 0), // I : jaune à la'avant du corps
  new Color (150, 0, 0),   // J : tronc arbre
  Color.lightGray,         // K : pattes
  Color.red,               // L : dessus de la tête
  Color.green.darker()     // M : vert foncé des feuilles
};

// une palette pour l'oiseau ci-dessus (la femelle !)
public static final Color[] palette2Oiseau = {
  new Color (0, 220, 0),   // A : vert des feuilles
  new Color (230, 230, 0), // B : jaune des feuilles
  Color.red,               // C : dessus de queue
  new Color (150, 100, 0), // D : dessous de la queue
  Color.magenta,           // E : autour de l'oeil
  Color.black,             // F : noir de l'aile
  Color.orange,            // G : dessous du ventre
  Color.white,             // H : points blanc de l'aile
  Color.yellow,            // I : jaune à la'avant du corps
  new Color (150, 0, 0),   // J : tronc arbre
  Color.lightGray,         // K : pattes
  Color.red,               // L : dessus de la tête
  Color.green.darker()     // M : vert foncé des feuilles
};

// une clé de sol pour les musiciens
public static final String[] clesol = { // 20 x 49
```

```
  "                    ",
  "              AA    ",
  "             AAAA   ",
  "            AAAAAA  ",
  "            AAAAAA  ",
  "           AA    AA ",
  "           AA    AA ",
  "           AA    AA ",
  "          AA     AA ",
  "          AA     AA ",
  "          AA    AAA ",
  "          AA   AAA  ",
  "          AA  AAAA  ",
  "          AA AAAA   ",
  "          AAAAAA    ",
  "          AAAAAA    ",
  "         AAAAAA     ",
  "        AAAAAA      ",
  "      AAAAAAA       ",
  "     AAAAA AA       ",
  "    AAAAA  AA       ",
  "    AAA    AA       ",
  "   AAA     AA       ",
  "   AA    AAAAAA     ",
  "   AA   AAAAAAAAAA  ",
  " AA   AAA  AA AAAA  ",
  " AA  AAA   AA   AA  ",
  " AA  AA    AA   AAA ",
  " AA  AA    AA    AA ",
  " AA  AA    AA    AA ",
  "  AA  AA   AA    AA ",
  "  AA   A   AA    AA ",
  "   AA      AA   AA  ",
  "    AA     AA   AA  ",
  "      AAA  AAAAA    ",
  "         AAAAA      ",
  "            AA      ",
  "            AA      ",
  "            AA      ",
  "      AAA   AA      ",
  "     AAAAA  AA      ",
  "    AAAAAA  AA      ",
  "    AAAAA  AA       ",
  "    AA     AA       ",
  "     AA   AA        ",
  "      AAAAA         ",
  "                    "
};

// une rose pour les sentimentaux
public static final String[] rose = { // 34 x 36
```

```
  /*"         1         2         3    " */
  /*"12345678901234567890123456789012345" */
    "                           BB      ",
    "              AAAA  A      BBB     ",
    "             AAAAAA AA     BBBB    ",
    "            AAAAAAAAAAA  BBCBB    ",
    "          DDAAAAAAAAAAAABCCCBB    ",
    "     CCC  DAAAAAADAAAAAACDCCBB    ",
    "     BCCCAAAAAAAADDADAAAADDCBB     ",
    "     BBCCADAAAADDADDAAAADDBB       ",
    "     BBCCAADDDDDDADDAAAADAA        ",
    "     BBBCCAADDDDADDDAAAADAAAA      ",
    "      BBBADAAAAADDDDAAAADAAAA      ",
    "        AAADAAADDDDAAAADAAAAA      ",
    "       DDAAADDDDDDAAAADAAAAACCC    ",
    "       DDDAAAADDDAAAAAAAAAACCCCC   ",
    "       DDDDAAAAAAAAAAAAADAACCCCBBC ",
    "       CDDDDDDDAAAAADAADBBBCCCCBB  ",
    "      CCCCCCAAAAAAADAADDCCBBBBBCB  ",
    "     CBBCCCDDAAAADDAADDCCCCC BBB   ",
    "     BBCCCCDDDDDDDAAADDCCCCCC      ",
    "     BBBBCCCDDDDDCAADDCCCCCCC      ",
    "      BBBBBCCDDCCCCCBECBBCCCC      ",
    "            CBCCCCCCBEBBCBBBC      ",
    "             BBCCCBBBE   BBBC      ",
    "             BBCCBBBEE             ",
    "              BCCBB EE             ",
    "               CB   EE             ",
    "               C   EE              ",
    "                   EE              ",
    "                   EEFF            ",
    "                  EEFF             ",
    "                  EE               ",
    "                  E                ",
    "                 EE                ",
    "                EEE                ",
    "                EE                 ",
    "                 E                 "
  };

  // palette des couleurs de la rose
  public static final Color[] paletteRose = {
    Color.red,               // A : rouge
    new Color (0, 220, 0),   // B : vert clair
    Color.green.darker(),    // C : vert foncé
    Color.pink,              // D : rose
    new Color (150, 0, 0),   // E : marron
    Color.yellow             // F : épine
  };

} // class MotifLib
```

10.2 LE PAQUETAGE UTILE

Quelques méthodes utilitaires pour la génération de nombres aléatoires et pour la mise en sommeil des threads.

```
// Utile.java

package mdpaquetage.utile;

public class Utile {

  // génère un nombre entre 0 et N-1 inclus
  public static int aleat (int N) {
    return (int) ((Math.random()*997) % N);
  }

  // le thread s'endort pendant n millisecondes
  public static void pause (int n) {
    try {
      Thread.sleep (n);
    }
    catch (InterruptedException e) {
      System.out.println (e);
    }
  }

  // attente d'un nombre aléatoire de ms entre 0 et n
  public static void attente (int n) {
    try {
      Thread.sleep (aleat(n));
    } catch (InterruptedException e) {
      System.out.println (e);
    }
  }

} // Utile
```

10.3 SUGGESTIONS DE PROJETS DE SYNTHÈSE EN JAVA

Quelques exemples de réalisations possibles en Java.

10.3.1 Partition de musique

Faire un programme Java permettant de gérer une partition de musique. Chaque note est un composant qui peut être déplacé ou modifié avec la souris. Les divers instruments sont simulés en utilisant les classes du paquetage javax.sound.midi (Synthesizer, MidiChannel). Utiliser la classe Motif pour afficher les notes. On peut également utiliser la classe Liste pour enregistrer les notes de la partition dans une liste.

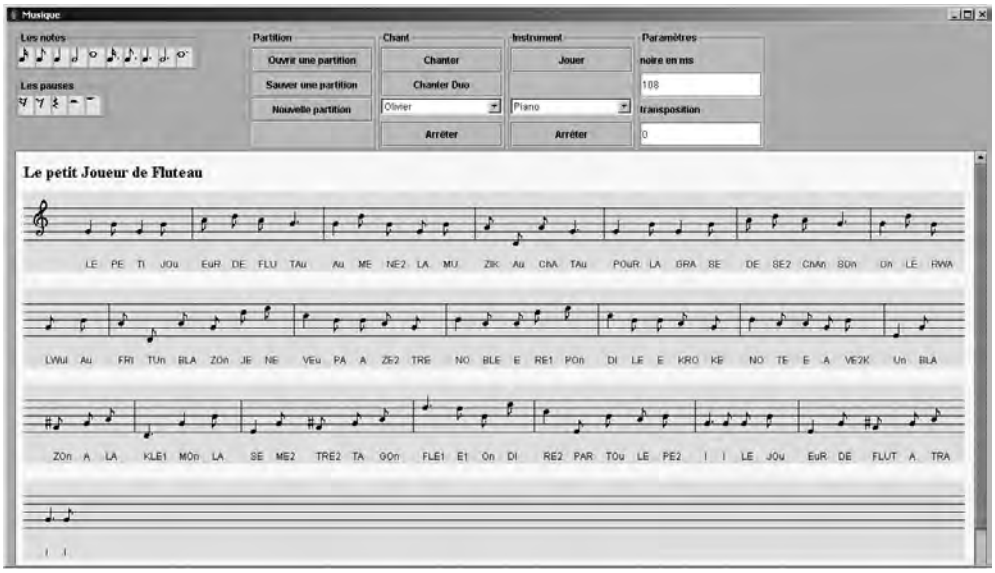


Figure 10.1 — Une partition de musique réalisée avec l'interface graphique Java.

10.3.2 Emploi du temps

On veut gérer un emploi du temps et l'afficher suivant différents critères : par enseignants, par matière, par salle, par groupe. Les données de l'emploi du temps sont enregistrées dans un fichier structuré comme indiqué ci-dessous. Utiliser JTree, JList, JTable et la table d'onglets pour réaliser les interfaces graphiques données ci-après.

La structure du fichier de l'emploi du temps :

nom	type	matière	salle	groupe	jour	heuredeb	duree
V.....	CM	ANALYSE	Amphi1	*	2	9	1

Sur la figure 10.2, un JTabbedPane (trois onglets), un JSplitPane contenant à gauche un JTree (Matieres/POO est sélectionné) et à droite, le dessin des pavés correspondant à l'enseignement de POO.

Sur la figure 10.3, un JTabbedPane (3 onglets), un JSplitPane contenant à gauche quatre JList (enseignants, salles, groupes, matières, salle) et à droite, la JTable des enseignements correspondant à la sélection (salle 13 sur l'exemple).

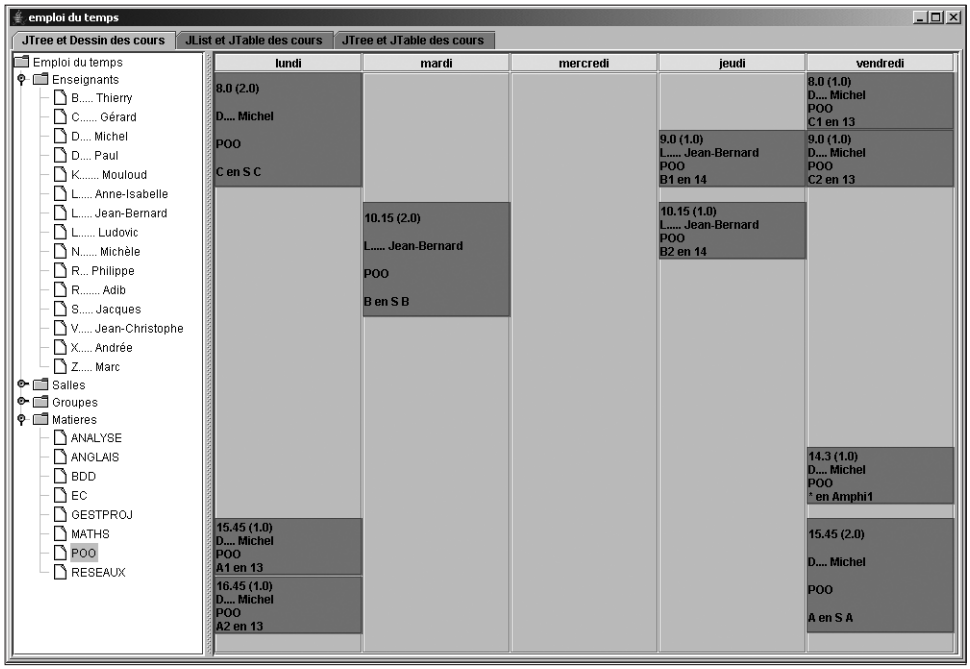


Figure 10.2 — JTree et Canvas.

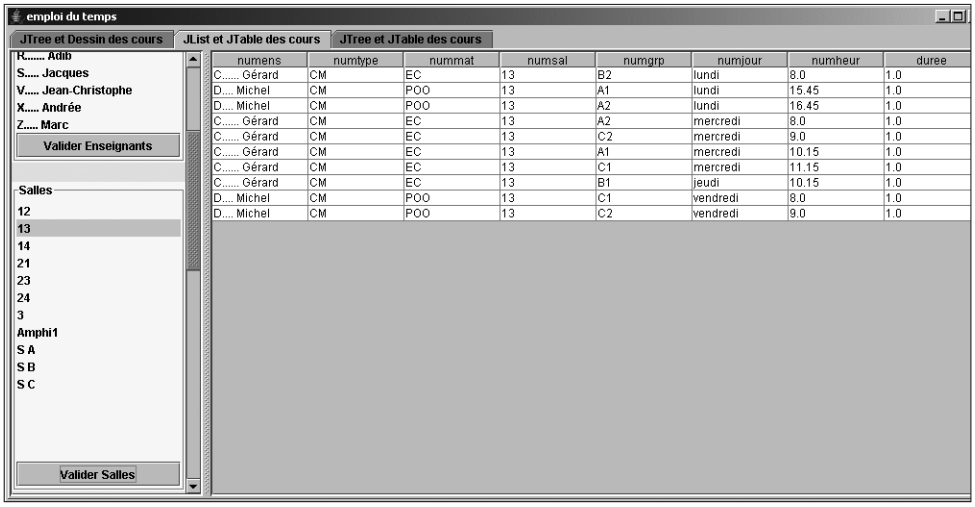


Figure 10.3 — JList et JTable.

Sur la figure 10.4, JTree à gauche, JTable à droite. Affichage des enseignements du groupe A1.

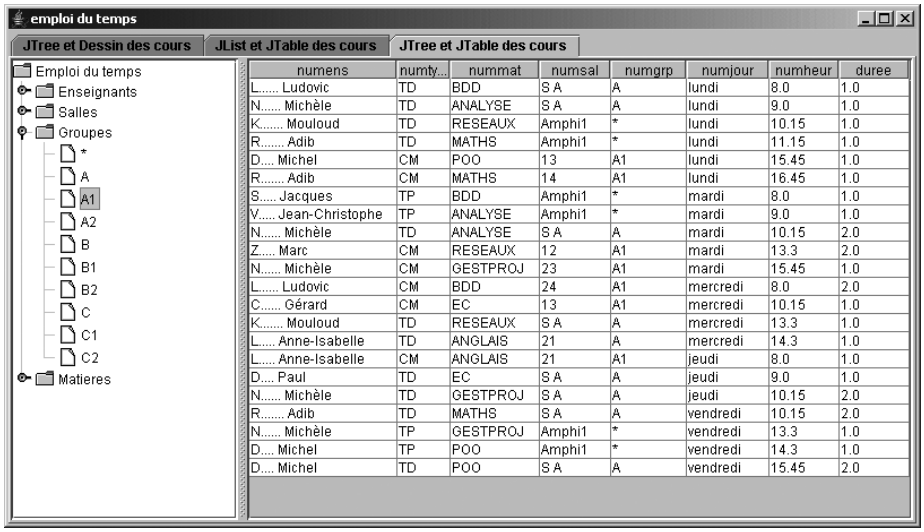


Figure 10.4 — JTree et JTable.

10.4 LA STRUCTURE DU PAQUETAGE MDPAQUETAGE

Ce paragraphe présente la structure du paquetage mdpaquetage utilisé au cours de ce livre. Les classes internes correspondant aux actions des écouteurs ne sont pas mentionnées. Le paquetage se trouve dans mdpaquetage.jar (voir fichiers jar, ci-après).

```
mdpaquetage
        mdawt
                Balle.class
                Ballon.class
                ComposantsDemo.class
                FigGeo.class
                Couleur.class
                Courbe.class
                DessinCourbe.class
                Droite.class
                DessinDroites.class
                Economiseur.class
                FermerFenetre.class
                MasterMind.class
                MenuAide.class
                Motif.class
```

```
          MotifAnime.class
          MotifLib.class
          Pendu.class
          Potence.class
          PenduURL.class
          Phonetique.class

paquetage1
          Classe1.class
          Classe2.class
complex
          Complex.class
          ComplexG.class
          DessinZ.class
listes
          Liste.class
          ListeBase.class
          Element.class
          ListeOrd.class
etudiant
          Editeur1.class
          Editeur2.class
          EditorJTable.class
          Etudiant.class
          ListModel.class
          Mess.class
          RendererJList1.class
          RendererJList.class
          RendererJTable1.class
          RendererJTable.class
          RendererJTree1.class
          RendererJTree.class
          TableModel.class
es
          LectureClavier.class
pile
          Pile.class
repertoire
          Repertoire.class
processus
          Semaphore.class
utile
          Utile.class
```

10.5 COMPILATION ET EXÉCUTION DE PROGRAMMES JAVA

10.5.1 Compilation d'un programme Java

Le fichier de commandes **compiljava**.bat suivant permet sous Windows de compiler un ou plusieurs programmes Java à partir de n'importe quel répertoire. Les fichiers résultant de cette compilation (les .class) sont envoyés dans le répertoire de nom de variable *repclass* (initialisé ici avec c :\temp). La racine des paquetages recherchés pendant la phase de compilation est indiquée par la variable *mdpaquetage* (initialisé ici avec c :\temp).

L'instruction Java :

```
package mdpaquetage.listes;
```

a pour effet de ranger les classes dans : c :\temp\mdpaquetage\listes. L'option -d de javac indique le répertoire de sortie y compris pour les paquetages à créer.

```
@echo off
rem compiljava

set repjava=c:\j2sdk1.4.2_04\bin                les exécutables du sdk
set mdpaquetage=c:\temp               le répertoire racine du paquetage
set repclass=c:\temp                             répertoire de sortie

rem avec jdk de sun
%repjava%\javac -classpath %mdpaquetage% -d %repclass%  %1 %2 %3 %4 %5
                                                              %6 %7 %8 %9

@echo on
```

10.5.2 Exécution d'un programme Java

Le fichier de commandes **exejava**.bat suivant permet sous Windows d'exécuter un programme Java de n'importe quel répertoire en indiquant le répertoire de la classe à exécuter et le répertoire de la racine du paquetage mdpaquetage. Ces répertoires pourraient être différents.

```
@echo off
rem exejava

set repjava =c:\j2sdk1.4.2_04\bin
set mdpaquetage =c:\temp              le répertoire racine du paquetage
set repclass =c:\temple                répertoire des .class à exécuter

rem avec jdk de sun
%repjava%\java -classpath %mdpaquetage%;%repclass%  %1 %2 %3 %4 %5
                                                              %6 %7 %8 %9

@echo on
```

Un fichier de commande **refaire**.bat permet de compiler et exécuter facilement un programme PPJava.java créant ou utilisant le paquetage mdpaquetage.

```
rem refaire.bat

call compiljava  PPJava.java
```

```
call exejava     PPJava
pause
```

10.5.3 Les fichiers archives (.jar)

On peut créer des fichiers archives à l'aide du logiciel jar.exe (du sdk java). Le paquetage mdpaquetage peut être remplacé par un seul fichier mdpaquetage.jar.

```
set repjava=c:\j2sdk1.4.2_04\bin
set repclass=c:\temp

rem c : create, v : verbose, f : nom du fichier en .jar
rem -C liste des fichiers ou répertoires à archiver
rem    examen récursif pour les répertoires (mdpaquetage ici)
%repjava%\jar cf %repclass%\mdpaquetage.jar -C %repclass% mdpaquetage
```

On peut le lister avec :

```
t : lister; f : le nom du fichier .jar suit les options.
%repjava%\jar tf %repclass%\mdpaquetage.jar
```

Remarque : lors de la compilation et de l'exécution, classpath peut indiquer un fichier de paquetage en .jar. On peut donc remplacer dans ce cas, dans compiljava.bat et exejava.bat :

```
set mdpaquetage=c:\temp
```

par

```
set mdpaquetage=c:\temp\mdpaquetage.jar
```

Ceci est surtout vrai quand le paquetage est opérationnel et mis à la disposition des utilisateurs.

Chapitre **11**

Corrigés des exercices

CHAPITRE 1

Exercice 1.1 : Programme d'écriture des valeurs limites des types primitifs

```
// LimitPrimitif.java  les limites des types primitifs   // voir page 8

class LimitPrimitif {

  public static void main (String[] args) {
    System.out.println (
      "\nByte.MIN_VALUE      : " + Byte.MIN_VALUE +
      "\nByte.MAX_VALUE      : " + Byte.MAX_VALUE +
      "\nShort.MIN_VALUE     : " + Short.MIN_VALUE +
      "\nShort.MAX_VALUE     : " + Short.MAX_VALUE +
      "\nInteger.MIN_VALUE   : " + Integer.MIN_VALUE +
      "\nInteger.MAX_VALUE   : " + Integer.MAX_VALUE +
      "\nLong.MIN_VALUE      : " + Long.MIN_VALUE +
      "\nLong.MAX_VALUE      : " + Long.MAX_VALUE +
      "\nFloat.MIN_VALUE     : " + Float.MIN_VALUE +
      "\nFloat.MAX_VALUE     : " + Float.MAX_VALUE +
      "\nDouble.MIN_VALUE    : " + Double.MIN_VALUE +
      "\nDouble.MAX_VALUE    : " + Double.MAX_VALUE
    );
  }  // main

}  // class LimitPrimitif
```

Résultats :

```
Byte.MIN_VALUE      : -128
Byte.MAX_VALUE      : 127
Short.MIN_VALUE     : -32 768
Short.MAX_VALUE     : 32 767
Integer.MIN_VALUE   : -2147483648
Integer.MAX_VALUE   : 2 147 483 647
Long.MIN_VALUE      : -9223372036854775808
Long.MAX_VALUE      : 9223372036854775807
Float.MIN_VALUE     : 1.4E-45
Float.MAX_VALUE     : 3.4028235E38
Double.MIN_VALUE    : 4.9E-324
Double.MAX_VALUE    : 1.7976931348623157E308
```

CHAPITRE 2

Exercice 2.1 : La classe Date

```
// Date.java                                                    // voir page 65

class Date {
  private int jour;
  private int mois;
  private int an;

  Date (int j, int m, int an) {
    jour    = j;
    mois    = m;
    this.an = an;
  }

  public String toString () {
    return " " + jour + "/" + mois + "/" + an;
  }

  // fournit vrai si année bissextile, faux sinon
  static boolean bissex (int annee) {
    if ( (annee % 400) == 0 ) return true;
    if ( (annee % 100) == 0 ) return false;
    if ( (annee % 4)   == 0 ) return true;
    return false;
  }

  // idem ci-dessus pour une Date
  boolean bissex () {
    return bissex (an);
  }
```

```
  // fournit le nombre de jours écoulés dans l'année
  int nbJoursEcoules () {
    int lgMoisP [] = {0,31,59,90,120,151,181,212,243,273,304,334};
    return lgMoisP [mois-1] + jour + (Date.bissex (an) ? 1 : 0);
  }

  // fournit le nombre de jours restants dans l'année
  int nbJoursRestants () {
    return 365 + (Date.bissex (an) ? 1 : 0) - nbJoursEcoules();
  }

  // nombre de jours écoulés entre 2 dates date1 < date2
  static long nbJourEntre (Date date1, Date date2) {
    int nje1 = date1.nbJoursEcoules ();
    int njr1 = date1.nbJoursRestants ();
    int nje2 = date2.nbJoursEcoules ();

    // Les deux dates de la même année
    if (date1.an == date2.an) return nje2-nje1;

    // année de date2 supérieure à année de date1
    long nbj = njr1;  // fin de la première année
    for (int i=date1.an+1; i < date2.an; i++ ) {
      nbj += 365 + (Date.bissex (i) ? 1 : 0);    // les années entières
    }
    nbj += nje2;  // fin de la dernière année
    return nbj;
  }

} // classe Date
```

CHAPITRE 3

Exercice 3.1 : Ajouter un nom pour une pile

```
// PPPile.java  Programme Principal de la Pile         // voir page 105

import mdpaquetage.pile.*;   // Classe Pile

class PileNom extends Pile {
  String nomPile;

  PileNom (int max, String nomPile) {
    super (max);                  // le constructeur de la super-classe
    this.nomPile = nomPile;
  }

  public void listerPile() {
    System.out.println ("\nNom de la pile : " + nomPile);
    super.listerPile();   // la méthode listerPile() de la super-classe
  }

}  // PileNom
```

```
public class PPPile {

  public static void main (String args []) {
    PileNom p1 = new PileNom (5, "Pile 1");
    try {
      p1.empiler (10);
      p1.empiler (20);
      p1.empiler (30);
    } catch (Exception e) {           // exception lancée par empiler()
      System.out.println (e);
    }
    p1.listerPile();

    PileNom p2 = new PileNom (10, "Pile 2");
    try {
      p2.empiler (100);
      p2.empiler (200);
      p2.empiler (300);
      p2.empiler (400);
    } catch (Exception e) {
      System.out.println (e);
    }
    p2.listerPile();

  } // main

} // class PPPile
```

CHAPITRE 4

Exercice 4.1 : Les cartes

```
// PPCartes.java  Programme Principal des cartes       // voir page 131

import mdpaquetage.listes.*;  // Liste

class Carte implements Comparable {
  private int couleur;
  private int valeur;

  Carte (int couleur, int valeur) {
    this.couleur = couleur;
    this.valeur  = valeur;
  }

  public String toString () {
    return "couleur : " + couleur + " valeur : " + valeur;
  }

  int couleur () {
    return couleur;
  }
```

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

  public int compareTo (Object objet) {
    Carte c1 = (Carte) this;
    Carte c2 = (Carte) objet;
    if (c1.couleur < c2.couleur) return -1;
    if (c1.couleur == c2.couleur) {
      if (c1.valeur <  c2.valeur) return -1;
      if (c1.valeur == c2.valeur) return 0;
    }
    return 1;
  }

}  // class Carte

// un paquet de cartes = une liste de cartes non ordonnées
class PaquetCartes extends Liste {

  //PaquetCartes() {                          // constructeur par défaut
  //  super();
  //}

  void insererEnFinDePaquet (Carte carte) {
    insererEnFinDeListe (carte);
  }

  // lister les cartes du paquet;
  // listerListe conviendrait mais n'écrit pas le n° de Carte
  void listerCartes () {
    int i = 0;
    ouvrirListe();
    while (!finListe()) {
      Carte carte = (Carte) objetCourant();
      i++;
      System.out.println ("i : " + i + " " + carte);
    }
  }

  void creerTas() {
    for (int i=1; i <=4; i++) {
      for (int j=1; j <=13; j++) {
        insererEnFinDePaquet (new Carte (i, j));
      }
    }
  }

  PaquetCartes battreLesCartes() {
    PaquetCartes pb = new PaquetCartes();               // paquet battu

    for (int i=5 2; i >=1; i--) {
      int aleat = (int)(Math.random() * i) + 1;
```

```
      Carte extrait = (Carte)extraireNieme (aleat);   // carte extraite
      if (extrait != null) pb.insererEnFinDeListe (extrait);
    }
    return pb;
  }

  void distribuerLesCartes (ListeOrd[] joueur) {
    for (int nj=0; nj < joueur.length; nj++) {
      joueur[nj] = new ListeOrd();
    }

    for (int i=1; i <=13; i++) {
      for (int nj=0; nj < 4; nj++) {
        Carte extrait = (Carte) extraireEnTeteDeListe();
        joueur[nj].insererEnOrdre (extrait);
      }
    }
  }

}  // class PaquetCartes

// class Programme Principal des Cartes
class PPCartes {

  public static void main (String args[]) {
    PaquetCartes   tas1   = new PaquetCartes();
    ListeOrd[] joueur = new ListeOrd[4];

    tas1.creerTas();
    System.out.println ("\nListe du tas1 après création\n");
    tas1.listerCartes();
    //tas1.ListerListe("\n");

    System.out.println ("\n\nListe du tas battu\n");
    PaquetCartes  tas2 = tas1.battreLesCartes();
    tas2.listerCartes();

    tas2.distribuerLesCartes (joueur);
    for (int nj=0; nj < joueur.length; nj++) {
      System.out.println ("\n\nListe du joueur " + nj + "\n");
      joueur[nj].listerListe("\n");
    }
  }  // main

}  // class PPCartes
```

CHAPITRE 5

Exercice 5.1 : La classe parcinq

```
// ParCinq.java  mise en page de 5 composants sur un Panel // voir page 162

import java.awt.*;           // Panel
import mdpaquetage.mdawt.*;  // Motif
```

```
// Panel contenant 5 Component
class ParCinq extends Panel {
  Component[] t = new Component [5];

  public ParCinq (Component c1, Component c2,
                  Component c3, Component c4, Component c5) {
    setLayout (null);            // pour le Panel
    setBackground (Color.cyan);  // du Panel
    t[0] = c1;
    t[1] = c2;
    t[2] = c3;
    t[3] = c4;
    t[4] = c5;
    for (int i=0; i < 5; i++) add (t[i]);
  }

  public void paint (Graphics g) {
    super.paint(g);
    Rectangle r = getBounds();
    int l = (r.width-20)/3;
    int h = (r.height-60)/3;
    t[0].setBounds (10,     30,     l, h);
    t[1].setBounds (10+2*l, 30,     l, h);
    t[2].setBounds (10+l,   40+h,   l, h);
    t[3].setBounds (10,     50+2*h, l, h);
    t[4].setBounds (10+2*l, 50+2*h, l, h);
  }

} // ParCinq

// trois fois cinq composants dans un Frame
class DispositionP5 {

  public static void main (String[] args) {
    Motif  m1 = new Motif (MotifLib.champignon);
    Motif  m2 = new Motif (MotifLib.rose,Motif.tFixe,
                                          MotifLib.paletteRose);
    m2.setTaille (80, 80);
    Motif  m3 = new Motif (MotifLib.oiseau, Motif.tProp,
                                          MotifLib.palette1Oiseau);
    Motif  m4 = new Motif (MotifLib.oiseau, Motif.tProp,
                                          MotifLib.palette2Oiseau);
    m4.setInverser();
    Motif  m5 = new Motif (MotifLib.hirondelle, Motif.tFixe);

    Frame f = new Frame ("3 fois par 5");
    f.setBounds (10,10, 800, 300);
    f.setVisible (true);  // nécessaire
    f.setBackground (Color.lightGray);  // du Frame
    f.setLayout (new GridLayout (0, 3, 10, 10));

    ParCinq p1 = new ParCinq (m1, m2, m3, m4, m5);
    f.add (p1);
```

```
    ParCinq p2 = new ParCinq (new Motif(m1), new Motif(m2),
                              new Motif(m3),new Motif(m4), new Motif(m5));
    f.add (p2);

    ParCinq p3 = new ParCinq (new Motif(m1), new Motif(m2),
                              new Motif(m3),new Motif(m4), new Motif(m5));
    f.add (p3);

    f.addWindowListener (new FermerFenetre());          // voir page 221

  }  // main

}  // DispositionP5
```

Remarque : on aurait pu utiliser l'héritage pour la classe DispositionP5 en écrivant :

```
class DispositionP5 extends Frame { ...}
```

et en supprimant f devant les appels de méthodes (f.setBounds(), etc.) comme pour la classe PPMotif.

Exercice 5.2 : Réutilisation du composant FigGeo

```
// PPFigGeo.java  Programme Principal de test du composant FigGeo;
//        dessin des 8 figures possibles + une variante// voir page 186

import java.awt.*;           // Frame
import mdpaquetage.mdawt.*;  // PPFigGeo, FermerFenetre

// Programme Principal de test de FigGeo
class PPFigGeo extends Frame {

  PPFigGeo () {  // hérite de Frame
    setTitle   ("Avec le nouveau composant FigGeo");
    setBackground (Color.lightGray);  // défaut : blanc
    setBounds  (10,10, 400, 400);     // défaut : petite taille
    // mise en page par colonne de 3; 10 pixels entre composants
    setLayout (new GridLayout (0, 3, 10, 10));

    for (int i=0; i < 8; i++) {
      add (new FigGeo (i));
    }
    // un neuvième dessin pour compléter
    FigGeo rosace2bis = new FigGeo (7);
    rosace2bis.setRosace2 (45, 0.9);
    add (rosace2bis);
    addWindowListener (new FermerFenetre());          // voir page 221
    setVisible (true);  // défaut : invisible
  } // constructeur PPFigGeo

  public static void main (String[] arg) {
    new PPFigGeo();
  } // main

}  // PPFigGeo
```

Exercice 5.3 : Programme java du pendu - utilisation du composant potence

```
// Pendu.java  jeu du pendu avec les mots d'un tableau  // voir page 192
// Pendu est un nouveau composant de type Panel

package mdpaquetage.mdawt;

import java.awt.*;            // Panel, TextField
import java.awt.event.*;      // ActionListener
import mdpaquetage.utile.*;   // aleat

public class Pendu extends Panel {  // composant Pendu

  // initialisation directe du tableau tabMots
  private String[] tabMots = {
    "beaucoup",       "baudruche",  "colibri",    "coing",
    "couenne",        "dodeliner",  "dompter",    "doigt",
    "fat",            "foyer",      "gencive",    "gel",
    "grandiloquent",  "hameçon",    "immonde",    "insulaire",
    "obséquieux",     "leader",     "maestro",    "minaret",
    "morille",        "navet",      "ouistiti",   "parjure",
    "pharynx",        "poinçon",    "presbyte",   "quintuple",
    "ravitaillement", "reptile",    "république", "somptueux",
    "surexcité",      "trémolo",    "vouvoyer",
    "anticonstitutionnellement"
  };
  int nbMots = tabMots.length;

  String       mot;          // mot courant à découvrir
  StringBuffer motEnCours;   // les caractères trouvés du mot en cours
  TextField    tF1      = new TextField ("");  // mot en cours
  TextField    tF2      = new TextField ("");  // le caractère proposé
  Potence      potence  = new Potence();
  Button       bRejouer = new Button ("Rejouer");

  // cbPendu : color back Pendu, color back Potence et front Potence
  public Pendu (Color cbPendu, Color cbPotence, Color cfPotence) {
    setLayout (new BorderLayout()); // defaut : FlowLayout
    setBackground (cbPendu);

    Panel pnord = new Panel();
    pnord.setLayout(new GridLayout(2,2));
    add (pnord, "North");

    Label l1 = new Label ("Mot à trouver     ? ");
    pnord.add (l1);
    tF1.setBackground (Color.white);
    tF1.setEditable  (false);
    tF1.setFocusable (false);
    pnord.add (tF1);

    Label l2 = new Label ("Caractère proposé ? ");
    pnord.add (l2);
```

```
    tF2.setBackground (Color.white);
    pnord.add (tF2);

    potence.setBackground (cbPotence);
    potence.setForeground (cfPotence);
    add (potence,  "Center");
    add (bRejouer, "South");

    Font f = new Font ("TimesRoman", Font.BOLD, 14);
    tF1.setFont (f);
    tF2.setFont (f);

    setVisible (true);
    initJouer ();  // initialisation et choix du premier mot
    traitement();
  }  // constructeur Pendu

  public Pendu () {
    this (Color.yellow, Color.white, Color.red);
  }

  // changement du tableau des mots à découvrir
  public void setTableMots (String[] tabMots) {
    if (tabMots != null) {
      this.tabMots = tabMots;
      initJouer();
    }
  }

  // À faire à chaque jeu : (ré)initialiser les données
  private void initJouer () {
    potence.init();
    int nAleat = Utile.aleat (tabMots.length);            voir page 366
    mot        = tabMots [nAleat];
    motEnCours = new StringBuffer ("");
    for (int i=0; i < mot.length(); i++) motEnCours.append ("*");
    tF1.setText (new String (motEnCours));
    tF2.setEditable (true); // mis à faux si pendu
    tF2.setText ("");
    tF2.requestFocus();     // curseur dans tF2
  }  // initJouer


  // TRAITEMENT DES EVENEMENTS

  // on déclare les composants à prendre en compte (écouter)
  void traitement () {
    tF2.addActionListener      (new CaractereProposee());
    bRejouer.addActionListener (new ActionRejouer());
  }

  // Classes internes à la classe Pendu                  voir page 91
```

```
  // un caractère proposé
  class CaractereProposee implements ActionListener {
    public void actionPerformed (ActionEvent evt) {
      String s = tF2.getText();
      if (s.length() == 0) return;  // sinon exception
      char car = s.charAt (0);
      // car existe-t-il dans le mot à trouver ?
      boolean trouve = false;
      for (int i=0; i < mot.length(); i++) {
        if ( mot.charAt(i) == car ) {
          motEnCours.setCharAt (i, car);
          trouve = true;
        }
      }
      if (!trouve) {  // pas trouvé car
        potence.incEtat();
        if (potence.getEtat() == Potence.maxEtat-1)
                                              tF2.setEditable (false);
      } else if (mot.equals (new String (motEnCours))) {
        potence.setTrouve();
        tF2.setEditable (false);
      }
      tF1.setText (new String (motEnCours));
      tF2.setText ("");
      tF2.requestFocus();
    }
  }  // caractereProposee

  // pour le bouton Rejouer
  class ActionRejouer implements ActionListener {
    public void actionPerformed (ActionEvent evt) {
      initJouer ();
    }
  }

}  // class Pendu
```

Exercice 5.4 : Balle mobile dans un rectangle

//voir page 199

La classe BalleMobile

```
// BalleMobile.java  composant d'affichage d'une balle

package mdpaquetage.mdawt;

import java.awt.*;  // Graphics

public class BalleMobile extends Balle {
  private int vitx;       // vitesse suivant l'axe des x
  private int vity;       // vitesse suivant l'axe des y

  // la Balle est mobile, centrée en (x,y)
```

```
// de vitesse vitx (axe des x) et vity (axe des y)
public BalleMobile (Color clrBalle, int diametre, int x, int y,
              int vitx, int vity) {
  super (clrBalle, diametre, x, y);
  this.vitx = vitx;
  this.vity = vity;
}

// création d'un BalleMobile de centre et vitesses aléatoires
// dans le rectangle de Dimension d
public BalleMobile (Color clrBalle, int diametre, Dimension d) {
  super (clrBalle, diametre, d);
  vitx = (int) (2 + Math.random() * 6);  // de 2 à 8
  vity = (int) (2 + Math.random() * 6);
}

// les accesseurs en consultation et en modification
public int getVitesseX () { return vitx; }
public int getVitesseY () { return vity; }

// modifier la vitesse en x de la balle
public void setVitesseX (int vitesse) {
  vitx = vitesse;
  repaint();
}

// modifier la vitesse en y de la balle
public void setVitesseY (int vitesse) {
  vity = vitesse;
  repaint();
}

// modifier les vitesses en x et en y d'un rapport k
public void setVitesse (int k) {
  vitx = vitx * k;
  vity = vity * k;
  repaint();
}

// écrire les caractéristiques de la balle
public String toString () {
  return ( super.toString() + ", vitx = " + vitx + ",
                                                  vity = " + vity);
}

// déplacer la balle dans un cadre largeur x hauteur,
// sans sortir du cadre;
// recalculer x et y en tenant compte de la vitesse
// et du diamètre de la balle
public void deplacer (int largeur, int hauteur) {
  int d2 = diametre/2;
  x += vitx;
  y += vity;
  int ldrte = largeur-d2; // limite droite
```

```
    int lbas  = hauteur-d2; // limite bas
    if (x < d2)    { x = d2;    vitx =  Math.abs(vitx); };
    if (y < d2)    { y = d2;    vity =  Math.abs(vity); };
    if (x > ldrte) { x = ldrte; vitx = -Math.abs(vitx); };
    if (y > lbas)  { y = lbas;  vity = -Math.abs(vity); };
  }

}  // BalleMobile
```

Le programme principal de test de le classe BalleMobile

```
// PPBalleMobile.java
//    déplacer une balle dans un cadre

import java.awt.*;           // Graphics
import mdpaquetage.mdawt.*;  // BalleMobile
import mdpaquetage.utile.*;  // pause

// Programme Principal de test de BalleMobile
class PPBalleMobile extends Frame {

  PPBalleMobile () {
    setTitle  ("Une balle mobile dans un cadre");
    setBounds (100, 100, 300, 300);
    Panel p1 = new Panel ();
    add (p1);
    setVisible (true);

    // cadre où évolue la balle largeur x hauteur
    int largeur = 200;
    int hauteur = 150;
    // la balle
    BalleMobile bm = new BalleMobile (Color.blue, 30,
                                      largeur/2, hauteur/2, 8, 5);

    // le mouvement de la balle dans le contexte graphique de p1
    Graphics g = p1.getGraphics();
    for (;;) {
      g.clearRect (0, 0, largeur, hauteur);
      g.setColor  (Color.red);
      g.drawRect  (0, 0, largeur, hauteur);
      bm.deplacer (largeur, hauteur);
      bm.paint    (g);
      Utile.pause (100);
    }
  }  // constructeur PPBalleMobile

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

}  // PPBalleMobile
```

Exercice 5.5 : Menu déroulant pour MasterMind

```
// PPMasterMind.java  avec menu déroulant              // voir page 218

import java.awt.*;           // Frame
import java.awt.event.*;     // ActionListener pour menu
import mdpaquetage.mdawt.*;  // MasterMind

class PPMasterMind extends Frame {
  int NB_COUL   = 2;
  int NB_BALLES = 3;
  int MAX_COUPS = 7;
  MasterMind m;

  PPMasterMind () {
     setTitle  ("MasterMind Mind");
     setBounds (10, 10, 450, 600);
     addWindowListener  (new FermerFenetre());
     m = new MasterMind (NB_COUL, NB_BALLES, MAX_COUPS);
     add (m, "Center");
     afficherMenu();
     setVisible (true);
  }  // constructeur PPMasterMind

  void afficherMenu() {
   // Les menus
    MenuBar barre = new MenuBar();
    setMenuBar (barre);
    Menu menu1 = new Menu ("Couleurs");
    barre.add (menu1);
    MenuItem[] menuItem1 = new MenuItem [5]; // de 2 à 6
    for (int i=0; i < menuItem1.length; i++) {
      menuItem1[i] = new MenuItem (Integer.toString (i+2));
      menu1.add (menuItem1[i]);
    }
    // pour les items du premier menu
    for (int i=0; i < menuItem1.length; i++) {
      menuItem1[i].addActionListener (new ActionMenu1());
    }

    Menu menu2 = new Menu ("Balles");
    barre.add (menu2);
    MenuItem[] menuItem2 = new MenuItem [5]; // de 2 à 6
    for (int i=0; i < menuItem2.length; i++) {
      menuItem2[i] = new MenuItem (Integer.toString (i+2));
      menu2.add (menuItem2[i]);
    }
    // pour les items du premier menu
    for (int i=0; i < menuItem2.length; i++) {
      menuItem2[i].addActionListener (new ActionMenu2());
    }
```

```
        Menu menu3 = new Menu ("Coups");
        barre.add (menu3);
        MenuItem[] menuItem3 = new MenuItem [10]; // de 5 à 14
        for (int i=0; i < menuItem3.length; i++) {
          menuItem3[i] = new MenuItem (Integer.toString (i+5));
          menu3.add (menuItem3[i]);
        }
        // pour les items du premier menu
        for (int i=0; i < menuItem3.length; i++) {
          menuItem3[i].addActionListener (new ActionMenu3());
        }
      }  // afficherMenu

      // pour la première colonne du menu
      class ActionMenu1 implements ActionListener {
        public void actionPerformed (ActionEvent event) {
          String s = (String) event.getActionCommand();
          NB_COUL  = Integer.parseInt (s);
          remove (m);
          m = new MasterMind (NB_COUL, NB_BALLES, MAX_COUPS);
          add (m);
          validate();
        }
      }  // class ActionMenu1

      // pour la deuxième colonne du menu
      class ActionMenu2 implements ActionListener {
        public void actionPerformed (ActionEvent event) {
          String s = (String) event.getActionCommand();
          NB_BALLES  = Integer.parseInt (s);
          remove (m);
          m = new MasterMind (NB_COUL, NB_BALLES, MAX_COUPS);
          add (m);
          validate();
        }
      }  // class ActionMenu2

      // pour la troisième colonne du menu
      class ActionMenu3 implements ActionListener {
        public void actionPerformed (ActionEvent event) {
          String s = (String) event.getActionCommand();
          MAX_COUPS  = Integer.parseInt (s);
          remove (m);
          m = new MasterMind (NB_COUL, NB_BALLES, MAX_COUPS);
          add (m);
          validate();
        }
      }  // class ActionMenu3

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

    }  // PPMasterMind
```

Exercice 5.6 : Ballon créé par héritage de la classe balle et réagissant aux clics et aux mouvements de la souris

```
// Ballon.java  = une Balle (dé)gonflable et déplaçable  // voir page 225
//              avec la souris

package mdpaquetage.mdawt;

import java.awt.*;        // Color
import java.awt.event.*;  // MouseListener

public class Ballon extends Balle {

// centré au Point p
  public Ballon (Color couleur, int diametre, Point p) {
    super (couleur, diametre, p);
    addMouseListener       (new Gonfler());
    addMouseMotionListener (new Deplacer());
  }

  // centré dans son espace; non déplaçable
  public Ballon (Color couleur, int diametre) {
    super (couleur, diametre);
    addMouseListener       (new Gonfler());
  }

  // classes internes
  class Gonfler extends MouseAdapter {

    // à gauche du centre, on gonfle; à droite, on dégonfle
    public void mouseClicked (MouseEvent evt) {
      // Ballon bClique = (Ballon) evt.getSource();
      int   diametre = getDiametre();  // héritage de Balle
      int xCentre = getX();
      if (evt.getX() < xCentre) {
        setDiametre (diametre+5);
      } else {
        setDiametre (diametre > 15 ? diametre-5 : 10);
      }
      requestFocus();
    }

    // couleur sombre
    public void mouseEntered (MouseEvent evt) {
      Color clrBalle = getClrBalle();
      setCouleur (clrBalle.darker());
    }

    // couleur claire
    public void mouseExited (MouseEvent evt) {
      Color clrBalle = getClrBalle();
      setCouleur (clrBalle.brighter());
    }

  }  // class Gonfler
```

```
  // déplacer le Ballon dans son espace
  class Deplacer extends MouseMotionAdapter {
    public void mouseDragged ( MouseEvent evt) {
      setXY (evt.getX(), evt.getY());
    }
  }  // class Deplacer

  public void paint (Graphics g) {  // de Ballon
    super.paint(g);            // de Balle
    Dimension di = getSize (); // entourer la zone de dessin
    g.drawRect (0, 0, di.width-1, di.height-1);
  }

}  // class Ballon
```

Exercice 5.7 : Tracé de droites rebondissant sur les cotés d'un rectangle (Economiseur)

```
// Economiseur.java  économiseur d'écran              // voir page 227

package mdpaquetage.mdawt;  // FermerFenetre

import java.awt.*;           // Point, Graphics
import java.awt.event.*;     // ActionListener
import mdpaquetage.utile.*;  // aleat, pause             voir page 366

// définir une Droite à partir de 2 points
class Droite {
  Point p1;
  Point p2;

  // le constructeur à partir de deux Point
  Droite (Point p1, Point p2) {
    this.p1 = new Point (p1);
    this.p2 = new Point (p2);
  }

  // le constructeur de copie
  Droite (Droite d) {
    this (d.p1, d.p2);
  }

  public String toString () {
    return "Droite : " + p1 + p2;
  }

  // dessiner la Droite de cette classe
  void dessinerDroite (Graphics g) {
    g.drawLine (p1.x, p1.y, p2.x, p2.y);
  }

} // Droite

// dessiner des droites à partir d'un tableau de droites
class DessinDroites extends Component {
```

```
  Droite[] tabDroites = null;
  // double buffer pour éviter le scintillement
  protected Image    dBuf = null;  // Double Buffer
  protected Graphics gdBuf;        // Double Buffer

  // on obtient la référence sur le tableau des droites
  void setTableau (Droite[] tabDroites) {
    this.tabDroites = tabDroites;
    repaint();
  }

  // Double buffer
  // normalement update() efface le composant et appelle paint(g).
  // Ok si on ne redéfinit pas update
  // mais léger scintillement avec AWT
  public void update (Graphics g) {
    int w = getSize().width;
    int h = getSize().height;
    if (dBuf == null) {
      dBuf  = createImage (w, h);
      gdBuf = dBuf.getGraphics();
    }
    // on dessine dans le contexte graphique gdBuf
    paint (gdBuf);
    // on affiche l'image
    g.drawImage (dBuf, 0, 0, this);
  }

  // dessiner les droites du tableau tabDroites
  public void paint (Graphics g) {
    if (tabDroites != null) {
      for (int i=0; i < tabDroites.length; i++) {
        tabDroites[i].dessinerDroite (g);
      }
    }
  }

}  // DessinDroites

public class Economiseur extends Panel {
  int       largeur;
  int       hauteur;
  int       nbDroites;       // Nombre de droites en parallèle
  int       delai;           // délai en ms entre 2 affichages
  Color     couleurDroites;  // couleur des droites
  Color     couleurFond;     // couleur du fond des droites
  TextField tF1;             // pour le nombre de droites
  TextField tF2;             // pour le délai entre affichage
  DessinDroites dessin;      // zone de dessin des droites
  Droite[]  tabDroites;      // tableau des droites
  int       vx1, vy1;        // vitesse en x et y du début des //
  int       vx2, vy2;        // vitesse en x et y de la fin des //
```

```
// caractérisé par : le nombre et la couleur des droites,
// le délai en ms entre 2 affichages
// la vitesse en x et y de début et de fin des droites
public Economiseur (int nbDroites, Color couleurDroites,
  Color couleurFond, int delai, int vx1, int vy1, int vx2, int vy2) {
  this.nbDroites      = nbDroites;
  this.couleurDroites = couleurDroites;
  this.couleurFond    = couleurFond;
  this.delai  = delai;
  this.vx1    = vx1;
  this.vx2    = vx2;
  this.vy1    = vy1;
  this.vy2    = vy2;

  // le Panel
  setLayout     (new BorderLayout());
  setBackground (couleurFond);        // du Panel

  // les composants ajoutés à Panel (BorderLayout)
  // panel1 ajouté au nord avec les 2 paramètres
  Label l1 = new Label ("Nombre de droites ? ");
  tF1 = new TextField (" " + nbDroites);
  Label l2 = new Label ("Délai en millisecondes ? ");
  tF2 = new TextField (" " + delai);
  Panel panel1 = new Panel (new GridLayout (0,2));
  panel1.add (l1);
  panel1.add (tF1);
  panel1.add (l2);
  panel1.add (tF2);
  panel1.setBackground (Color.lightGray);
  add (panel1, "North");

  // la zone de dessin ajoutée au milieu
  // couleur du fond = couleur du fond de Frame
  dessin = new DessinDroites();
  dessin.setForeground (couleurDroites);
  add (dessin, "Center");

  // traitement : les actions prises en compte
  tF1.addActionListener (new GererParametres());
  tF2.addActionListener (new GererParametres());

  setVisible (true);
}  // constructeur Economiseur

// délai = 30; vit1 = (5, -3); vit2 = (4, 6)
public Economiseur (int nbDroites) {
  this (nbDroites, Color.blue, Color.yellow, 50, 5, -3, 4, 6);
}

public Economiseur (Economiseur e) {
   this (e.nbDroites, e.couleurDroites, e.couleurFond,
             e.delai, e.vx1, e.vy1, e.vx2, e.vy2);
}
```

```
// créer le tableau des n droites identiques
void initDonnees () {
  largeur   = dessin.getSize().width;
  hauteur   = dessin.getSize().height;
  if (largeur == 0) largeur = 50;
  if (hauteur == 0) hauteur = 50;
  int posx1 = Utile.aleat (largeur);
  int posy1 = Utile.aleat (hauteur);
  int posx2 = Utile.aleat (largeur);
  int posy2 = Utile.aleat (hauteur);
  Droite d  = new Droite (new Point (posx1,posy1),
                          new Point (posx2,posy2) );
  tabDroites = new Droite [nbDroites];  // allouer le tableau
  for (int i=0; i < tabDroites.length; i++) {
      tabDroites[i] = new Droite (d);
  }
  dessin.setTableau (tabDroites);
}  // initDonnees

// déplacer le Point p en restant dans le rectangle di
// si numero = 1, utiliser les vitesses (vx1, vy1)
// sinon utiliser (vx2, vy2)
// on modifie les attributs : vx1, vy1, vx2, vy2
void deplacer (Point p, Dimension di, int numero) {
  if (numero == 1 ) {
    if ( (p.x < 0) || (p.x > di.width) )  vx1 = -vx1;
    if ( (p.y < 0) || (p.y > di.height) ) vy1 = -vy1;
    p.x += vx1;
    p.y += vy1;
  } else {
    if ( (p.x < 0) || (p.x > di.width) )  vx2 = -vx2;
    if ( (p.y < 0) || (p.y > di.height) ) vy2 = -vy2;
    p.x += vx2;
    p.y += vy2;
  }
}

// boucle infinie de déplacement des droites
public void deplacer () {
  initDonnees();
  for (;;) {
    // on décale les droites dans le tableau tabDroites
    // la première valeur est perdue
    for (int i=0; i < tabDroites.length-1; i++) {
      tabDroites[i] = new Droite (tabDroites[i+1]);
    }
    // on ajoute la dernière droite du tableau
    Dimension di = dessin.getSize();
    deplacer (tabDroites[tabDroites.length-1].p1, di, 1);
```

```
        deplacer (tabDroites[tabDroites.length-1].p2, di, 2);
        dessin.repaint();
        Utile.pause (delai);
      }
    }

    // classes internes des écouteurs des TextField
    class GererParametres implements ActionListener {
      public void actionPerformed (ActionEvent evt) {
        TextField tf = (TextField) evt.getSource();
        String    s  = tf.getText();
        if (tf.equals(tF1)) {                          // des références
          nbDroites = Integer.parseInt (s);
          initDonnees();
        } else {
          delai = Integer.parseInt (s);
        }
      }  // actionPerformed
    } // GererParametres

}  // Economiseur
```

Exercice 5.8 : Écriture d'un programme de tracés de courbes (utilise une classe abstraite)

Le composant DessinCourbe (première partie)

```
// DessinCourbe.java                                    // voir page 229

package mdpaquetage.mdawt;

import java.awt.*;  // Canvas

public class DessinCourbe extends Component {
  double  tVal[];     // référence sur le tableau des valeurs
  private double xa;  // abscisse du premier point
  private double xh;  // distance entre les points

  // dessiner une courbe à partir des valeurs du tableau tVal
  // en partant de xa par pas de xh
  public DessinCourbe (double tVal[], double xa, double xh){
    this.tVal = tVal;
    this.xa   = xa;
    this.xh   = xh;
    setBackground (Color.cyan);
  }

  // si les valeurs sont communiquées par la suite
  // lecture dans un fichier par exemple
  public DessinCourbe (){
    this (null, 0, 0);
  }
```

```
// fournir la valeur minimum du tableau tVal
public double min () {
  double mini = tVal[0];
  for (int i=1; i < tVal.length; i++) {
    if (tVal[i] < mini) mini = tVal[i];
  }
  return mini;
}

// fournir la valeur maximum du tableau tVal
public double max () {
  double maxi = tVal[0];
  for (int i=1; i < tVal.length; i++) {
    if (tVal[i] > maxi) maxi = tVal[i];
  }
  return maxi;
}

// modifier les paramètres de la courbe
public void ModifDessinCourbe (double tVal[]) {
  this.tVal = tVal;
}

// tracer la courbe dans l'espace attribué (par le gestionnaire)
public void paint (Graphics g) {
  Dimension d       = getSize();  // dimension de la zone de dessin
  int       nb      = tVal.length;
  int       lgLigne = d.height;   // hauteur du dessin
  int       h       = d.width/(nb-1); // nombre de pixels
                                                    entre 2 valeurs
  double    pas;   // distance en pixels entre 2 points
                                                    sur l'axe des x
  double    x;     // valeur de x courante
  double    maxi;  // valeur maximale
  double    mini;  // valeur minimale

  super.paint(g);
  g.clearRect (0, 0, d.width, d.height);

  if (h ==0) h = 1;
  mini = min();
  maxi = max();
  if (maxi == mini) maxi = mini+1;
  pas  = (maxi - mini) / (lgLigne - 1);
  x    = xa;

  // les axes en rouge
  Color anc = g.getColor();
  g.setColor (Color.red);
  if (mini*maxi < 0) {
    int l = d.height - (int) ((0 - mini) / pas);
    g.drawLine (0, l, (nb-1)*h, l);
  }
```

```
    if (xa * (xa+(nb-1)*xh) < 0) {
      int v = (int) Math.round ((0-xa) / xh);
      v = v*h;
      g.drawLine (v, 0, v, lgLigne);
    }
    g.setColor (anc);

    // tracé des points
    int ka = d.height - (int) ((tVal[0] - mini) / pas);
    for (int j=1; j < nb; j++) {
      // tracé du segment entre les points j-1 et j
      int k = d.height - (int) ((tVal[j] - mini) / pas);
      g.drawLine ((j-1)*h, ka, j*h, k);
      ka = k;
    }
  }  // paint

}  // class DessinCourbe
```

La mise en œuvre du composant DessinCourbe

```
// PPCourbe1.java  Programme Principal de tracé de Courbe

import java.awt.*;           // Frame
import mdpaquetage.mdawt.*;  // DessinCourbe

class PPCourbe1 extends Frame {

  PPCourbe1 () {
    setTitle  ("abs(x)-3 et abs(abs(x)-3) entre -10 et + 10");
    setLayout (new GridLayout (0, 2, 10, 10));
    setBounds (10, 10, 400, 250);
    setBackground (Color.lightGray);

    // initialisation des tableaux de double tabV1 et tabV2
    double[] tabV1 = new double [21];
    double[] tabV2 = new double [21];
    for (int i=-10; i <=10; i++) {
      tabV1 [i+10] = Math.abs (i) -3;
      tabV2 [i+10] = Math.abs (Math.abs(i)-3);
    }

    DessinCourbe crb1, crb2;
    crb1 = new DessinCourbe (tabV1, -10, 1);
    crb2 = new DessinCourbe (tabV2, -10, 1);

    add (crb1);
    add (crb2);
    addWindowListener (new FermerFenetre());          // voir page 221
    setVisible (true);
  }  // PPCourbe1

  public static void main (String args []) {
    new PPCourbe1();
  }  // main

}  // class PPCourbe1
```

La classe abstraite Courbe (deuxième partie)

```
// Courbe.java

import java.awt.*;           // Panel
import mdpaquetage.mdawt.*;  // DessinCourbe

public abstract class Courbe extends Panel {
  double tVal[]; // tableau des valeurs
  double xa;     // x de départ
  double xh;     // écart entre deux points sur l'axe des x
  static final double PUIS10 = 10000d; // pour garder 4 décimales

  // xb : x de fin
  public Courbe (String nord, double xa, double xb, double xh) {
    setLayout (new BorderLayout());
    TextField tf1 = new TextField (nord);  // ou new Label (nord)
    tf1.setEditable (false);
    add (tf1, "North");

    // calculer les valeurs du tableau dont on veut la courbe
    this.xa = xa;
    this.xh = xh;
    int nb  = (int) (1 + Math.round ((xb-xa)/xh));
    tVal    = new double [nb];  // allocation du tableau tVal
    tabulation();  // calcule les nb valeurs du tableau tVal
    DessinCourbe crb = new DessinCourbe (tVal, xa, xh);

    // ajouter la courbe au centre
    add (crb, "Center");
    // et au sud, un libellé avec les valeurs min et max
    TextField tf2 = new TextField (  // ou Label
         " minimum : " + ((int)(C.min() * PUIS10)) / PUIS10 +
        ", maximum : " + ((int)(C.max() * PUIS10)) / PUIS10);
    tf2.setEditable (false);
    add (tf2, "South");

    setVisible (true);
  }  // constructeur de Courbe

  public abstract double fn (double x);

  // tabulation utilise la méthode abstraite fn
  public void tabulation () {
    double x = xa;
    for (int i=0; i < tVal.length; i++) {
      tVal[i] = fn (x);
      x += xh;
    }
  }

}  // class Courbe
```

Les courbes de la figure 5.44 :

```
// PPCourbe2.java Programme Principal de tracé de courbe

import java.awt.*;           // Frame
import mdpaquetage.mdawt.*;  // FermerFenetre

class Courbe1 extends Courbe {

  public double fn (double x) {
    return Math.sin (x) / x;
  }

  Courbe1 (String nord, double xa, double xb, double xh) {
    super (nord, xa, xb, xh);
  }
}

class Courbe2 extends Courbe {

  public double fn (double x) {
    return 5-x*x;
  }

  Courbe2 (String nord, double xa, double xb, double xh) {
    super (nord, xa, xb, xh);
  }
}

class Courbe3 extends Courbe {

  public double fn (double x) {
    return Math.sin (x);
  }

  Courbe3 (String nord, double xa, double xb, double xh) {
    super (nord, xa, xb, xh);
  }
}

class Courbe4 extends Courbe {

  public double fn (double x) {
    return x * x - 1;
  }

  Courbe4 (String nord, double xa, double xb, double xh) {
    super (nord, xa, xb, xh);
  }
}


class PPCourbe2 extends Frame {

  PPCourbe2 () {
    setTitle   ("Tracés de courbes");
    setLayout  (new GridLayout (0, 2, 10, 10));
    setBounds  (10, 10, 650, 500);
```

```
    setBackground (Color.lightGray);

    Courbe1 f1 = new Courbe1 (
      "Sinus(x)/x entre -5 PI et + 5 PI, pas 0.1",
      -5*Math.PI, 5*Math.PI, 0.1
    );

    Courbe2 f2 = new Courbe2 (
      "Parabole 5-x**2 entre -3 et +3, pas 0.1",
      -3, 3, 0.1
    );

    Courbe3 f3 = new Courbe3 (
      "Sinus(x) entre -2 PI et + 2 PI, pas 0.1",
      -2*Math.PI, 2*Math.PI, 0.1
    );

    Courbe4 f4 = new Courbe4 (
      "Parabole x**2-1 entre -3 et +3, pas 0.1",
      -3, 3, 0.1
    );

    add (f1);
    add (f2);
    add (f3);
    add (f4);
    addWindowListener (new FermerFenetre());          // voir page 221

    setVisible (true);
  }  // constructeur PPCourbe2

  public static void main (String args []) {
    new PPCourbe2();
  }

}  // class PPCourbe
```

Exercice 5.9 : Dessin de nombres complexes dans un contexte graphique

```
// DessinZ.java                                          // voir page 230

package mdpaquetage.complex;

import java.awt.*;            // Graphics
import mdpaquetage.listes.*;  // listes

// dessin d'une liste de nombres complexes
public class DessinZ extends Panel {
  int     marge   = 50;    // marge autour du dessin
  int     unite   = 10;    // unite = 10 pixels sur les axes
  Point   origine;         // coordonnées de l'origine
  boolean ajuste  = true;  // au milieu de la fenêtre, echelle ajustée
  Liste   li      = new Liste ();  // la liste des ComplexG
```

```
// dessin ajusté dans l'espace par défaut (origine et unité)
public DessinZ () {
}

// dessin ajusté dans l'espace;
// on fixe la marge autour du dessin
public DessinZ (int marge) {
  this.marge = marge;
}

// dessin non ajusté; axe en origine
public DessinZ (int unite, Point origine) {
  this.unite    = unite;
  this.origine  = origine;
  this.ajuste   = false; // l'origine est fixé
}

// ajouter un ComplexG à la liste li
public void ajouter (ComplexG z) {
  li.insererEnFinDeListe (z);
}

// parcourir la liste li pour trouver les valeurs
// maximales et minimales en x et en y
void ajusterUnite (Dimension d) {
  double maxXP = 0; // max des X positifs
  double maxYP = 0; // max des Y positifs
  double maxXN = 0; // max des X négatifs (minimum)
  double maxYN = 0; // max des Y négatifs (minimum)
  li.ouvrirListe();
  while (!li.finListe()) {
    ComplexG ptc = (ComplexG) li.objetCourant ();
    double pR = ptc.partieRC();
    double pI = ptc.partieIC();
    if (pR > maxXP) maxXP = pR;
    if (pR < maxXN) maxXN = pR;
    if (pI > maxYP) maxYP = pI;
    if (pI < maxYN) maxYN = pI;
  }

  // intervalle à représenter
  double valX = maxXP - maxXN;
  double valY = maxYP - maxYN;

  // calcul de la valeur en pixels de l'unité
  // suivant les dimensions de di
  // et calcul du point origine (x =0, y =0) en pixels
  unite = Math.min ((int)((d.width-2*marge)  / valX),
                    (int)((d.height-2*marge) / valY));
  if (unite ==0) unite = 1;
  origine = new Point ( marge + (int) ( Math.abs(maxXN)*unite),
                        marge + (int) ( Math.abs(maxYP)*unite) );
}  // ajusterUnite
```

```
  // dessiner les axes à partir du Point origine
  // et suivant les dimensions de d
  void dessinerAxes (Graphics g, Dimension d) {
    g.drawLine (0, origine.y, d.width-1, origine.y); // axe horizontal
    g.drawLine (origine.x,  0, origine.x, d.height-1); // axe vertical
    g.drawOval (origine.x-unite, origine.y-unite, 2*unite, 2*unite);
                                                        // cercle unité
  }

  // dessiner les ComplexG de la liste li
  public void paint (Graphics g) {
    Dimension d = getSize ();                    // au moment de paint()
    if (ajuste) ajusterUnite(d);
    dessinerAxes (g,d);

    li.ouvrirListe();
    while (!li.finListe()) {
      ComplexG ptc = (ComplexG) li.objetCourant ();
      ptc.dessiner (g, origine, unite);
    }
  }

}  //  class DessinZ
```

Le programme de mise en œuvre des classes ComplexG et DessinZ. On aurait pû ajouter les deux composants DessinZ dans une même fenêtre de type Frame.

```
// PPComplex.java

import java.awt.*;              // Frame
import mdpaquetage.complex.*;  // Complex, ComplexG, DessinZ
import mdpaquetage.mdawt.*;    // FermerFenetre

class PPComplex1 extends Frame {

  PPComplex1 () {
    setTitle      ("Nombres complexes");
    setBounds     (10, 10, 500, 500);
    setVisible    (true);
    setBackground (Color.cyan);
    addWindowListener (new FermerFenetre());           // voir page 221

    Complex z1 = new Complex (0.5, 2);
    Complex z2 = new Complex (2, -1);
    Complex z3 = z1.plus(z2);
    Complex z4 = z1.moins(z2);
    Complex z5 = z1.multiplier(z2);
    Complex z6 = z1.diviser(z2);

    // unité = 60 pixels; origine = (150, 300);
    DessinZ dz = new DessinZ (60, new Point (150, 300));
    dz.ajouter (new ComplexG (z1, Color.red,  "z1 : ", true));
    dz.ajouter (new ComplexG (z2, Color.red,  "z2 : ", true));
    dz.ajouter (new ComplexG (z3, Color.blue, "z1+z2", false));
    dz.ajouter (new ComplexG (z4, Color.blue, "z1-z2", false));
```

```
    dz.ajouter (new ComplexG (z5, Color.blue, "z1*z2", false));
    dz.ajouter (new ComplexG (z6, Color.blue, "z1/z2", false));
    add (dz);
    setVisible (true);
  }  // PPComplex1

  public static void main(String[] args) {
    new PPComplex1();
  }

}  // class PPComplex1

class PPComplex2 extends Frame {

  PPComplex2 () {
    // la fenêtre de type Frame
    setTitle      ("Cercle unité");
    setBounds     (10, 10, 700, 700);
    setVisible    (true);
    setBackground (Color.cyan);
    addWindowListener (new FermerFenetre());          // voir page 221

    // Zone de dessin des Complex
    // avec une marge de 100 pixels tout autour
    DessinZ dz = new DessinZ (100);
    // les Complex à dessiner
    Complex C = new Complex (1, 2*Complex.PI/10, Complex.RADIANS);
    for (int i=0; i < 10; i++) {
      dz.ajouter (new ComplexG (C.puissance(i), Color.red,
                                ("2PI*" +i + "/10 : "), true));
    }

    add (dz);
    setVisible (true);
  }  // PPComplex2

  public static void main(String[] args) {
    new PPComplex2();
  }

}  // class PPComplex2
```

CHAPITRE 6

Exercice 6.1 : Les boutons

// voir page 248

```
// APISwing2.java  les boutons

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

```
import javax.swing.*;

class APISwing2 extends JFrame {
  JToggleButton jt1;
  JToggleButton jt2;
  JToggleButton jt3;
  ActionListener al = new ActionJToggleButton();

  // constructeur APISwing2
  APISwing2 () {
    setTitle ("APISwing2");
    Container f = (Container) getContentPane();
    f.setLayout (new GridLayout (0,1));
    setBounds (10,10, 400, 400);

    f.add (lesJButton());
    f.add (lesJToggleButton1());
    f.add (lesJToggleButton2());
    f.add (lesJCheckBox());
    f.add (lesJRadioButton());

    setVisible (true);
  }  // constructeur APISwing2

  JPanel lesJButton        () {}  // voir cours
  JPanel lesJToggleButton1 () {}
  JPanel lesJToggleButton2 () {}
  JPanel lesJCheckBox      () {}
  JPanel lesJRadioButton   () {}

  class ActionJToggleButton implements ActionListener {
    public void actionPerformed (ActionEvent evt) {
      AbstractButton jt = (AbstractButton) evt.getSource();
      if (jt.isSelected()) {
        System.out.println ("selected  " + jt.getText());
      } else {
        System.out.println ("deselected" + jt.getText());
      }
      if (jt == jt1) {
        System.out.println ("home");
      } else if (jt == jt2) {
        System.out.println ("fleche droite");
      } else if (jt == jt3) {
        System.out.println ("fleche gauche");
      }
    }
  }

  public static void main (String[] args) {
    new APISwing2();
  }

}  // class APISwing2
```

Exercice 6.2 : JList des jours de la semaine

// voir page 263

```
// APISwing3.java  JList(sélection unique ou multiple)

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

class APISwing3 extends JFrame {
  String[] jour = { "lundi", "mardi", "mercredi", "jeudi",
                    "vendredi", "samedi", "dimanche" };

  // constructeur
  APISwing3 () {
    setTitle ("JList");
    Container f = (Container) getContentPane();
    f.setLayout (new GridLayout (1,2,10,10));
    setBounds (10, 10, 400, 250);

    f.add (JlistChoixUnique());
    f.add (JlistChoixMultiple());

    setVisible (true);
  }  // APISwing3

  // voir cours
  JPanel JlistChoixUnique() {}
  class  ChoixSimpleListe  implements ListSelectionListener {}
  JPanel JlistChoixMultiple() {}
  class  ValiderChoix  implements ActionListener {}

  public static void main (String[] args) {
    new APISwing3();
  }

}  // class APISwing3
```

CHAPITRE 7

Exercice 7.1 : Programme java du pendu avec URL

// voir page 309

```
// PenduURL.java  jeu du pendu avec les mots d'un tableau
//             ou d'un fichier local ou distant (URL)
//  définit un composant PenduURL qui hérite de Pendu

package mdpaquetage.mdawt;

import java.awt.*;        // Panel
import java.io.*;         // BufferedReader
```

```
import java.net.*;        // URL

public class PenduURL extends Pendu {

  // on utilise Pendu (avec les mots de Pendu)
  public PenduURL () {
    super (Color.yellow, Color.white, Color.red);
  }

  // initialisation des mots de Pendu avec urlFichier
  public PenduURL (URL urlFichier) {
    this (urlFichier, Color.yellow, Color.white, Color.red);
  }

  // initialisation des mots de Pendu avec urlFichier
  // cbPendu : couleur background du jeu
  // cbPotence, cfPotence : couleurs (back,fore)ground de la Potence
  public PenduURL (URL urlFichier, Color cbPendu, Color cbPotence,
                Color cfPotence) {
     super (cbPendu, cbPotence, cfPotence);
     String[] tabMots = inittabMots (urlFichier);
     setTableMots (tabMots);  // ensemble des mots du jeu
  }  // constructeur Pendu

  // fournit un tableau des mots du jeu
  // à partir d'un fichier local ou d'un site distant
  private String[] inittabMots (URL urlFichier) {
    String[] tabMots;
    try {
      // compte le nombre de mots "nbMots" dans le fichier
      String ch;
      int nbMots = 0;
      InputStream is = urlFichier.openStream();
      BufferedReader fe =
                        new BufferedReader (new InputStreamReader(is));
      while ( (ch =fe.readLine()) != null) {
        nbMots++;
      }

      // Réinitialisation du tableau tabMots
      // Relecture du fichier fe
      tabMots = new String [nbMots];
      int i = 0;
      is = urlFichier.openStream();
      fe = new BufferedReader (new InputStreamReader(is));
      while ( (ch =fe.readLine()) != null) {
        //System.out.println ("inittabMots " + ch);
        tabMots[i++] = ch;
      }
    } catch (FileNotFoundException e) {
      System.out.println (e);  // on continue avec le tableau
      tabMots = null;
```

```
    } catch (Exception e) { // IOException, SecurityExceptionEx
      System.out.println ("initdebut " + e);
      tabMots = null;
    }
    return tabMots;
  }  // inittabMots

}  // class PenduURL
```

Le programme PPenduURl de test de la classe PenduURL.

```
// PPenduURL.java  jeu du pendu
//   avec un ensemble de mots par défaut
//   ou à partir des mots d'une URL

import javax.swing.*;          // JFrame
import java.net.*;             // URL
import java.io.*;              // IOException
import mdpaquetage.mdawt.*;    // class Pendu

public class PPenduURL extends JFrame {

  PPenduURL (String fichier) {
    setBounds    (10, 10, 340, 450);
    if (fichier == null) {
      setTitle     ("Jeu du pendu (mots par défaut)" );
      getContentPane().add (new PenduURL ());
    } else {
      setTitle     ("Jeu du pendu ( mots de " + fichier + ")" );
      URL urlFichier;
      try {
        urlFichier = new URL ("file :" + fichier);
        getContentPane().add (new PenduURL (urlFichier));
      } catch (IOException e) {
        getContentPane().add (new PenduURL ());
      }
    }
    setVisible (true);
  }

  public static void main (String args[]) {
    if (args.length == 1) {
      new PPenduURL (args[0]);
    } else {
      new PPenduURL (null);
      //System.out.println ("paramètre 1 : < fichier des mots > ");
    }
  }

}  // PPenduURL
```

Exercice 7.2 : Gestion d'un carnet d'adresses

```
// Carnet.java   gestion d'un carnet d'adresses            // page 313
//               gestion d'un fichier d'adresses en mode texte
//               utilise un tableau de String

import javax.swing.*;       // JFrame
import java.awt.*;          //
import java.awt.event.*;    // ActionListener
import java.io.*;           // PrintWriter
import java.util.*;         // StringTokenizer

class Adresse {
  static final  int nbChamp = 6;
  String[] tabCh = new String [nbChamp]; // une Adresse = 6 champs

  public String toString () {
    return tabCh[0] + " " + tabCh[1] + " " + tabCh[2]+ " " +
           tabCh[3] + " " + tabCh[4] + " " + tabCh[5];
  }
}  // Adresse

public class Carnet extends JFrame {
  final static int nbChamp = Adresse.nbChamp;
  final static int maxAdr  = 100;

  JLabel     l1  = new JLabel ("Nom");
  JLabel     l2  = new JLabel ("Prénom");
  JLabel     l3  = new JLabel ("Rue");
  JLabel     l4  = new JLabel ("Code Postal");
  JLabel     l5  = new JLabel ("Ville");
  JLabel     l6  = new JLabel ("Téléphone");
  JTextField tf1 = new JTextField (15);
  JTextField tf2 = new JTextField (15);
  JTextField tf3 = new JTextField (30);
  JTextField tf4 = new JTextField (15);
  JTextField tf5 = new JTextField (15);
  JTextField tf6 = new JTextField (15);
  JTextField[] tabTF  = {tf1, tf2, tf3, tf4, tf5, tf6 };

  JButton    b1  = new JButton ("Suivant");
  JButton    b2  = new JButton ("Précédent");
  JButton    b3  = new JButton ("Rechercher");
  JButton    b4  = new JButton ("Lancer la recherche");
  JButton    b5  = new JButton ("Ajouter");
  JButton    b6  = new JButton ("Valider Ajout");
  JButton    b7  = new JButton ("Détruire");
  JButton    b8  = new JButton ("Enregistrer et Quitter");
  JButton    b9  = new JButton ("Quitter sans enregistrer");
  JButton[] boutons = { b1, b2, b3, b4, b5, b6, b7, b8, b9 };
```

```
//JTextArea  ta  = new JTextArea("", 3, 20);
JEditorPane  ta  = new JEditorPane ("text/html", "");

String    nomFichier = "adresses.dat";
Adresse[] tabAdr     = new Adresse[maxAdr];
int       nbAdr      = 0;  // nombre d'adresses
int       numAdr     = 0;  // numéro de l'adresse courante de tabAdr
BufferedReader fe    = null;
PrintWriter    fs    = null;
ActionListener ecouteur = new ActionBouton();

public Carnet() {
  Container f = this.getContentPane();
  setTitle      ("Carnet d'adresses");
  setBackground (Color.cyan);
  setBounds     (20, 200, 400, 500);
  BoxLayout bo = new BoxLayout (f, BoxLayout.Y_AXIS);
  f.setLayout (bo);

  JPanel p1 = new JPanel(new GridLayout(0,2));
  p1.setBorder(BorderFactory.createTitledBorder("Caractéristiques"));
  p1.setBackground (Color.cyan);
  f.add (p1);
  p1.add (l1);  p1.add (tf1);
  p1.add (l2);  p1.add (tf2);
  p1.add (l3);  p1.add (tf3);
  p1.add (l4);  p1.add (tf4);
  p1.add (l5);  p1.add (tf5);
  p1.add (l6);  p1.add (tf6);

  f.add (Box.createVerticalStrut(20));  // espace de 10 pixels

  // les boutons dans p2
  JPanel p2 = new JPanel(new GridLayout(0,2));
  p2.setBorder(BorderFactory.createTitledBorder("Fonctions"));
  p2.setBackground (Color.yellow);
  f.add (p2);

  ActionBouton ecouteur = new ActionBouton();
  for (int i=0; i < boutons.length; i++) {
    p2.add(boutons[i]);
    boutons[i].addActionListener(ecouteur);
  }
  b4.setEnabled (false);
  b6.setEnabled (false);

  f.add (Box.createVerticalStrut(20));  // espace de 10 pixels

  // le JTextArea dans p3
  JPanel p3 = new JPanel(new GridLayout(0,1));
  f.add(p3);
  p3.setBorder(BorderFactory.createTitledBorder("Trace"));
  p3.setBackground (Color.cyan);
  p3.add(ta);
```

```
  lireFichierAdresses();
  for (int i=0; i < tabTF.length; i++) {
    tabTF[i].setText(tabAdr[0].tabCh[i]);
  }
  setVisible (true);
} // class Carnet

// lire les adresses du fichier nomFichier
// et les mémoriser dans le tableau tabAdr
void lireFichierAdresses () {
  try {
    // ouverture en entrée du fichier adresses.dat
    fe = new BufferedReader (new FileReader (nomFichier));
    // Lire le fichier et mémoriser les informations dans tabAdr
    String ch;
    while ((ch =fe.readLine())!=null) {
      Adresse ad = new Adresse();
      StringTokenizer adr = new StringTokenizer (ch, "/");
      for (int i=0; i < nbChamp; i++) {
        ad.tabCh[i] = adr.nextToken();
      }
      if (nbAdr < maxAdr) tabAdr [nbAdr++] = ad;
    }
    fe.close();
  } catch (IOException e) {
    System.out.println ("Erreur " + e);
    System.exit (1);
  }
}

// regénérer le fichier des adresses à partir du tableau tabAdr
void regenerer () {
  // renommer le fichier actuel pour sauvegarde
  // détruire l'ancienne sauvegarde
  File fe = new File (nomFichier);
  File sv = new File ("old" + nomFichier);
  sv.delete();
  fe.renameTo (sv);
  // regénérer le fichier nomFichier
  try {
    fs = new PrintWriter (new FileWriter (nomFichier));
  } catch (IOException err) {
    System.out.println ("Erreur " + err);
    System.exit(1);
  }
  for (int i=0; i < nbAdr; i++) {
    for (int j=0; j < 5; j++) {
      fs.print (tabAdr[i].tabCh[j] + "/");
    }
   fs.println (tabAdr[i].tabCh[5]);
  }
  fs.close();
```

```
}

int rechercherbouton (JButton b) {
  boolean trouve = false;
  int     i = 0;
  while (!trouve && i < boutons.length) {
    if ( b == boutons[i]) {
      trouve = true;
    } else {
      i++;
    }
  }
  return trouve ? i : -1;
}

int rechercher (String nom) {
  boolean trouve = false;
  int     i = 0;
  while (!trouve && i < nbAdr) {
    if ( nom.equals(tabAdr[i].tabCh[0]) ) {
      trouve = true;
    } else {
      i++;
    }
  }
  return trouve ? i : -1;
}

void setTF (int numAdr) {
  if (nbAdr == 0) {  // aucun élément
    resetTF ();
    ta.setText ("Le carnet d'adresses est vide");
    b1.setEnabled (false);
    b2.setEnabled (false);
    b3.setEnabled (false);
    b6.setEnabled (false);
  } else {
    for (int i=0; i < nbChamp; i++)
                          tabTF[i].setText(tabAdr[numAdr].tabCh[i]);
  }
}

void resetTF () {
  for (int i=0; i < tabTF.length; i++) {
    tabTF[i].setText("");
  }
}


// Actions des boutons Enregistrer, Suivant, Effacer, Quitter
class ActionBouton implements ActionListener {

  public void actionPerformed (ActionEvent evt) {
    JButton bouton = (JButton) evt.getSource();
    String s = "";
```

```
b4.setEnabled (false);
b6.setEnabled (false);

switch (rechercherbouton(bouton) +1) {
case 1 :
  numAdr++; if (numAdr >= nbAdr) numAdr = 0;
  ta.setText ("");
  setTF (numAdr);
  break;
case 2 :
  numAdr--; if (numAdr < 0) numAdr = nbAdr-1;
  ta.setText ("");
  setTF (numAdr);
  break;
case 3 :
  resetTF();
  ta.setText ("Indiquer le < b > nom </b > et valider " +
      "\navec le bouton < b > Lancer la recherche </b > ");
  tabTF[0].requestFocus();
  b4.setEnabled (true);
  break;
case 4 :
  int numR = rechercher (tabTF[0].getText());
  if (numR != -1) {
    numAdr = numR;
    ta.setText (tabTF[0].getText() + " trouvé");
    setTF (numAdr);
  } else {
    ta.setText (tabTF[0].getText() + " <b > inconnu </b > ");
  }
  break;
case 5 :
  resetTF();
  ta.setText ("<b > Compléter </b > les différents
                                    champs de l'adresse à ajouter"
    + "\net valider avec le bouton < b > Valider Ajout </b > ");
  b6.setEnabled (true);
  tabTF[0].requestFocus();
  break;
case 6 :
  Adresse ad = new Adresse ();
  for (int i=0; i < 6; i++) {
    ad.tabCh[i] = new String (tabTF[i].getText());
  }
  if (tabTF[0].getText().equals ("")) {
    ta.setText ("Indiquer le nom à ajouter");
  } else {
    numAdr = nbAdr;
    tabAdr[nbAdr++] = ad;
    ta.setText (tabTF[0].getText() + " ajouté");
  }
  break;
```

```
        case 7 :
          if (nbAdr > 0) {
            ta.setText (tabTF[0].getText() + " " + tabTF[1].getText()
                                                    + "  détruit");
            if (numAdr != nbAdr-1) {
              tabAdr [numAdr] = tabAdr [nbAdr-1];
            }
            nbAdr--;
            numAdr--; if (numAdr < 0) numAdr = 0;
            setTF (numAdr);
          } else {
            ta.setText ("Le carnet d'adresses est vide");
          }
          break;
        case 8 :
           regenerer();
           System.exit(0);
           break;
         case 9 :
          System.exit(0);
          break;
       }

      }  // actionPerformed

    }  // class ActionBouton


    public static void main (String[] args) {
      new Carnet();
    }
  }
```

CHAPITRE 8

Exercice 8.1: Producteur Consommateur avec n places

```
// ProdConsN.java  Producteur Consommateur            // voir page 332
//                 avec zone commune partagée de n places

import mdpaquetage.processus.*;
import mdpaquetage.utile.*;                              // voir § 10.2

// Producteur Consommateur avec un tampon de n places
class ZoneC {                      // zone commune partagée par p1 et c1
  final int nPlaces;
  int[] valeur;

  ZoneC (int nPlaces) {
    this.nPlaces = nPlaces;
    valeur = new int [nPlaces];
  }

} // class ZoneC
```

```
// class Producteur
class Producteur extends Thread {
  Semaphore libre;   // référence du sémaphore libre
  Semaphore occupe;  // référence du sémaphore occupe
  ZoneC     entier;  // référence de la zone mémoire commune
  String    nom;     // nom du producteur

  Producteur (Semaphore libre, Semaphore occupe,
              ZoneC entier, String nom) {
    this.libre  = libre;
    this.occupe = occupe;
    this.entier = entier;
    this.nom    = nom;
  }

  public void run() {
    for (int i=0; i < 10; i++) {
      Utile.attente (50);
      libre.P();
      int n = i % entier.nPlaces;
      entier.valeur [n] = i*10;
      System.out.println (nom + " i = " + i
              + "  valeur = " + n + " " + entier.valeur[n]);
      occupe.V();
    }
  }

}  // class Producteur


// class Consommateur
class Consommateur extends Thread {
  Semaphore libre;   // référence du sémaphore libre
  Semaphore occupe;  // référence du sémaphore occupe
  ZoneC     entier;  // référence de la zone mémoire commune
  String    nom;     // nom du Consommateur

  Consommateur (Semaphore libre, Semaphore occupe,
                ZoneC entier, String nom) {
    this.libre  = libre;
    this.occupe = occupe;
    this.entier = entier;
    this.nom    = nom;
  }

  public void run() {
    for (int i=0; i < 10; i++) {
      Utile.attente (100);
      occupe.P();
      int n = i % entier.nPlaces;
      System.out.println (nom + " i = " + i
           + "         valeur = " + n + " " + entier.valeur [n]);
      libre.V();
    }
  }

}  // class Consommateur
```

```
class ProdConsN {
  public static void main(String[] args) {
    int np = 3;  // zone commune de np places
    Semaphore    libre  = new Semaphore (np);
    Semaphore    occupe = new Semaphore (0);
    ZoneC        entier = new ZoneC (np);
    Producteur   p1  = new Producteur   (libre, occupe, entier, "p1");
    Consommateur c1  = new Consommateur (libre, occupe, entier, "c1");

    p1.start();                                  // démarrage des threads
    c1.start();
    try {
      p1.join();
      c1.join();
    } catch (InterruptedException e) {
      System.out.println (e);
    }

    System.out.println ("The end\n");
  }  // main
}  // class ProdConsN
```

Exercice 8.2 : Les dessins de la simulation des baigneurs

```
// la classe DessinBaigneurs est un thread qui lit dans le tube
// et dessine les états des baigneurs                          page 335

class DessinBaigneurs extends JFrame implements Runnable {
  int        nbProc;  // nombre de processus Baigneur
  int[]      etatB;   // tableau de l'état courant de chaque Baigneur
  JLabel[][] tE;      // tableau de libellés
  static PipedInputStream pi = new PipedInputStream();

  // les noms des états d'un Baigneur
  String[] tEtat = {
    "arrive",     // 0
    "panier",     // 1
    "déshabille", // 2
    "baigne",     // 3
    "rhabille",   // 4
    "quitte"      // 5
  };
  int nbEtat = tEtat.length;  // nombre d'états

  String[] nomb = { // Noms des baigneurs (26 au plus)
    "A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M",
    "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"
  };

  DessinBaigneurs (int nbProc) {
    Container f = getContentPane();
    this.nbProc = nbProc;
    etatB = new int [nbProc];
    tE    = new JLabel [nbEtat][nbProc];
    for (int i=0; i < etatB.length; i++) etatB [i] = 0;
```

```
    setTitle ("La piscine");
    f.setLayout (new GridLayout(0,nbProc+1));

    // Ligne de titre du haut
    JLabel l1 = new JLabel ("BAIGNEURS");
    f.add (l1);
    for (int j=0; j < nbProc; j++) {
      JLabel b = new JLabel (nomb[j], JLabel.CENTER);
      b.setOpaque(true);
      b.setBackground (Color.cyan);
      f.add (b);
    }

    for (int j=0; j < nbEtat; j++) {
      JLabel l2 = new JLabel (tEtat[j]);
      l2.setOpaque (true);
      l2.setBackground (Color.cyan);  // colonne 1 des noms d'états
      f.add (l2);
      for (int i=0; i < nbProc; i++) {
        JLabel l3 = new JLabel (" ", JLabel.CENTER);
        l3.setOpaque (true);
        tE[j][i] = l3;
        f.add (l3);
      }
    }

    setBounds (20, 20, 650, 300);
    setVisible (true);
  }  // constructeur de DessinBaigneurs

  // lit dans le tube pi et affiche suivant les valeurs lues
  public void run() {
    int numero;  // numéro du baigneur
    try {
      while ( (numero = pi.read()) != -1) {
        int etat = pi.read();

        // on modifie l'ancien état du baigneur
        tE[etatB[numero]][numero].setText (".");
        tE[etatB[numero]][numero].setBackground (Color.white);

        // on modifie le nouvel état du baigneur
        etatB [numero] = etat;
        tE[etat][numero].setText ("*");
        tE[etat][numero].setBackground (Color.red);
        Utile.pause (500);
      }  // while
      pi.close ();
    } catch (IOException e) {
      System.out.println (e);
    }
  }  // run

}  // class DessinBaigneurs
```

```
public class PPBaigneurs {

  public static void main(String[] args) {
    final int nBaigneur = 8;  // nombre de baigneurs
    final int nPanier   = 5;  // nombre de paniers
    final int nCabine   = 3;  // nombre de cabines

    // initialise les sémaphores
    // et ouvre en écriture le tube des résultats
    Baigneur.init (nPanier, nCabine);

    // crée les threads baigneurs
    Baigneur[] tbaigneur = new Baigneur [nBaigneur];
    for (int i=0; i < nBaigneur; i++) tbaigneur[i] = new Baigneur (i);

    // crée le thread DessinBaigneurs
    DessinBaigneurs dessin  = new DessinBaigneurs (nBaigneur);
    Thread tdessin = new Thread (dessin);
    tdessin.start();

    // attendre la fin de tous les threads Baigneur
    // et du thread de DessinBaigneurs
    try {
      for (int i=0; i < nBaigneur; i++) tbaigneur[i].join();
      tdessin.join();
    } catch (InterruptedException e) {
      System.out.println (e);
    }

    // le thread initial main se termine
    System.out.println ("C'est fini");
  }

}  // class PPBaigneurs
```

CHAPITRE 9

Exercice 9.1 : Les dessins récursifs d'un arbre

```
// dessiner un arbre                                        voir page 347
class Arbre extends Canvas {
  int     lg;                       // longueur du tronc (1° segment)
  Color   tronc     = Color.gray;   // couleur des branches
  Color   feuille   = Color.green;  // couleur des feuilles
  Color   fleur     = Color.red;    // couleur des fleurs
  int     ouverture = 180;          // ouverture des branches
  double  pc        = 0.1;         // % de aléatoire pour les longueurs
  int nbSegment     = 3;  // nombre aléatoire entre 0, ..., nbSegment-1
  boolean auto      = true;        // taille automatique dans l'espace
```

```
Arbre (int lg, Color tronc, Color feuille, Color fleur,
       int ouverture, int nbSegment) {
  this.lg        = lg;           // -1 si taille automatique
  this.tronc     = tronc;
  this.feuille   = feuille;
  this.fleur     = fleur;
  this.ouverture = ouverture;
  this.nbSegment = nbSegment;
  auto           = false;        // taille imposée
}

// lg en fonction de l'espace du composant
Arbre (Color tronc, Color feuille, Color fleur,
       int ouverture, int nbSegment) {
  this (-1, tronc, feuille, fleur, ouverture, nbSegment);
  auto           = true;  // taille imposée
}

Arbre (Color tronc, Color feuille, Color fleur) {
  this (-1, tronc, feuille, fleur, 180, 3);
}

Arbre () {
}

// tracer un segment dans un contexte graphique
//   lg      : longueur du segment
//   po      : point origine du segment
//   angle   : angle en degrés
//   couleur : couleur du segment
//   pa      : point arrivée
public void avance (Graphics g, int lg, Point po, int angle,
                    Color couleur, Point pa) {
  Color sauv = g.getColor();
  pa.x = (po.x + (int) (lg*Math.cos (angle*2*Math.PI/360.)));
  pa.y = (po.y - (int) (lg*Math.sin (angle*2*Math.PI/360.)));
  g.setColor (couleur);
  int nb = lg/10;
  for (int i=-nb/2; i <=nb/2; i++) {  // épaisseur du trait
    g.drawLine (po.x+i, po.y, pa.x+i, pa.y);
  }
  g.setColor(sauv);
}

// dessiner un segment de longueur lg en partant du Point po
// et suivant un angle et une couleur donnés;
// puis tracer récursivement de 3 à 5 segments
// en repartant de la fin du segment en cours.
public void arbreDessin (Graphics g, int lg, Point po,
                         int angle, Color couleur) {
  Point pa = new Point ();              // point d'arrivée du segment
  avance (g, lg, po, angle, couleur, pa);              // pa modifié
```

```
        int nb = 3 + Utile.aleat (nbSegment);                  // 3, 4, 5, …
        lg = 2*lg / 3;                    // longueur des prochains segments
        if (lg > 3) {
          // bouts des prochaines branches en vert,
          // et un fruit de temps en temps au point pa
          if (lg <= 5) {  // 3 ou 4 ou 5
            couleur = feuille;              // Le bout des branches en vert
            int n2 = Utile.aleat (30);              // entre 0 et 29 inclus
            if (n2 == 1) {                     // Une fois sur 30, une pomme
              g.setColor (fleur);
              g.fillOval (pa.x, pa.y, 8, 8);
            }
          }

          // les nb segments sont réparties suivant un angle ouverture
          int a = angle;
          int d = ouverture / nb;     // écart en degrés entre les segments
          int douv = 1 + (int) (ouverture*pc);
          int dlg  = 1 + (int) (lg*pc);
          for (int i=1; i <=nb; i++) {
            a = angle - (ouverture/2) - (d/2) + i*d; a = a % 360;
            int deltaOuv = -(douv/2) + Utile.aleat (douv);
            int deltaLg  = Utile.aleat (dlg);
            if (a >=-90 && a <=270)
              // si on veut éviter les branches vers le bas
              arbreDessin (g, lg + deltaLg, pa, a + deltaOuv, couleur);
          }

        }  // if

      }  //  arbreDessin

      public Dimension getPreferredSize(){
        return new Dimension (100, 100);
      }

      public void paint (Graphics g) {
        Dimension d = getSize();
        // il faut redessiner entièrement le dessin
        int lga = lg;
        if (auto) {
          lga  = 1 + Math.min (d.height, d.width)/8;  // pas 0
        }

        Point po = new Point (d.width/2, 7*d.height/8);
        arbreDessin (g, lga*3/2, po, 90, tronc);
      }

    } // class Arbre
```

La classe PanelArbre utilise le composant Arbre ci-dessus :

```
    // AppletArbre.java  applet de dessin d'un arbre
    // le dessin est différent à chaque appel de paint
```

```
// les caractéristiques du dessin ne sont pas sauvegardées

import java.awt.*;          // Color, GridLayout
import javax.swing.*;       // Graphics, Point, Color
import java.awt.event.*;    //

// dessiner un panel arbre avec 3 textField de modification
// des paramètres
public class PanelArbre extends JPanel {
  Arbre a = new Arbre (Color.gray, Color.green, Color.red, 150, 3);
  JTextField tF1, tF2, tF3;

  public PanelArbre() {
    setLayout (new BorderLayout());

    JPanel p1 = new JPanel ();
    p1.setLayout (new GridLayout(0, 2));
    add (p1, "North");

    JLabel l1 = new JLabel ("ouverture (60-360) ? ");
    p1.add (l1);
    tF1 = new JTextField (Integer.toString (a.ouverture));
    p1.add (tF1);
    JLabel l2 = new JLabel ("noeuds (de 1 à 5) ? ");
    p1.add (l2);
    tF2 = new JTextField (Integer.toString(a.nbSegment));
    p1.add (tF2);
    JLabel l3 = new JLabel ("aléatoire (de 0 à 15) ? ");
    p1.add (l3);
    tF3 = new JTextField (Integer.toString((int) (a.pc*100)));
    p1.add (tF3);
    p1.setBackground (Color.cyan);

    add (a, "Center");

    tF1.addActionListener (new Ouverture());
    tF2.addActionListener (new NbSegment());
    tF3.addActionListener (new PC());
  }  // PanelArbre


  class Ouverture implements ActionListener {
    public void actionPerformed (ActionEvent event) {
      String s = tF1.getText();
      int v = Integer.parseInt (s);
      if (v < 60)  v = 60;
      if (v > 360) v = 360;
      tF1.setText ("" + v);
      a.ouverture = v;
      a.repaint();
    }
  }
```

```
  class NbSegment implements ActionListener {
    public void actionPerformed (ActionEvent event) {
      String s = tF2.getText();
      int v = Integer.parseInt (s);
      if (v <= 0) v = 1;  // éviter modulo 0
      if (v  > 5) v = 5;
      tF2.setText ("" + v);
      a.nbSegment = v;
      a.repaint();
    }
  }

  class PC implements ActionListener {
    public void actionPerformed (ActionEvent event) {
      String s = tF3.getText();
      int v = Integer.parseInt (s);
      if (v < 0)  v = 0;
      if (v > 15) v = 15;
      tF3.setText ("" + v);
      a.pc = v/100d;
      a.repaint();
    }
  }

}  // class PanelArbre
```

Exercice 9.2 : Feu d'artifice

// voir page 347

```
// Segment.java  dessiner un segment de droite

import java.awt.*;      // Graphics, Point, Color

public class Segment {
  // un segment de droite
  int     lg;         // longueur du segment
  Point   po;         // coordonnées du point d'origine du segment
  Point   pa;         // coordonnées du point d'arrivée du segment
  int     angle;      // angle du segment
  Color   couleur;    // Couleur
  boolean terminal;  // segment terminal ou non

  // définir un segment de longueur lg, d'origine po
  // d'angle et couleur donnés.
  public Segment (int lg, Point po, int angle, Color couleur,
                  boolean terminal) {
    this.lg      = lg;
    this.po      = new Point (po.x, po.y);
    this.angle   = angle;
    this.couleur = couleur;
    this.pa      = new Point (0,0); // à calculer
    pa.x = (po.x + (int) (lg*Math.cos (angle*2*Math.PI/360.)));
```

```
    pa.y = (po.y - (int) (lg*Math.sin (angle*2*Math.PI/360.)));
    this.terminal = terminal;
  }

  // afficher le segment
  void paint (Graphics g) {
    Color sauv = g.getColor();
    g.setColor(couleur);
    // pour effectuer un trait plus ou moins fin suivant lg
    for (int i=0; i <=lg/15; i++) {
      g.drawLine (po.x+i, po.y, pa.x+i, pa.y);
    }
    g.setColor (sauv);
  }

  public String toString() {
    return "lg : " + lg + ", angle : " + angle;
  }

}  // class Segment
```

Le programme principal de test du feu d'artifice :

```
// FeuArt.java  dessiner un feu d'artifice (Application ou Applet)

import java.awt.*;            // Canvas, Point, Color
import mdpaquetage.listes.*;  // Liste
import mdpaquetage.mdawt.*;   // tableau des couleurs
import mdpaquetage.utile.*;   // aleat

class FeuArt extends Canvas implements Runnable {
  public static final int nbc =13; // nombre de couleurs
  Liste listCour = new Liste();

  FeuArt () {
    setBackground (Color.black);  // il fait nuit noire
    new Thread (this).start();    // on lance run() de FeuArt
  }

  private static int aleat (int nbS) {
    return Utile.aleat (nbS);
  }

  // ajouter un élément (Segment) en fin de la liste li
  void  ajouterSegment (Liste li, int lg, Point po, int angle,
                        Color couleur, boolean terminal) {
    Segment nouveau = new Segment (lg, po, angle, couleur, terminal);
    li.insererEnFinDeListe (nouveau);
  }

  // dessiner un feu d'artifice
  // lg    : longueur du premier Segment
  // po    : point d'origine
```

```
// angle : angle du premier segment
void dessinerFeuArt (Dimension d, int lg, Point po, int angle) {
  int dfin = (int) d.height/20;
  listCour = new Liste();
  ajouterSegment (listCour, lg, po, angle,
                              Couleur.getColor(aleat(nbc)), false);

  while (!listCour.listeVide ()) {
    Liste listSuiv = new Liste();
    listCour.ouvrirListe ();

    while (!listCour.finListe ()) {
      Segment segment = (Segment) listCour.objetCourant();
      if (!segment.terminal) {
        int hasard = aleat (5);
        int nbS;  // le Segment segment éclate en nbS Segments
        int dev;  // angle entre les Segments

        switch (hasard) {
        // le Segment éclate en nbS segments terminaux
        case 1 :
          nbS = 12;
          dev = (int) (360 / nbS);
          for (int i=1; i <=nbS; i++) {
            ajouterSegment (listSuiv, 2*segment.lg/3, segment.pa,
                  segment.angle + i*dev, segment.couleur, true);
          }
          break;
        // le Segment éclate en nbS segments non terminaux
        default :
          int lgP = 2*segment.lg / 3;  // longueur des prochains
                                                                segments
          if (lgP >= 10) {
            int ouverture = 60 + aleat (120);
            lgP  = lgP + aleat (lgP/8);
            nbS = 2 + aleat(3);
            dev = ouverture / nbS;
            for (int i=1; i <=nbS; i++) {
              ajouterSegment (listSuiv, lgP, segment.pa,
                     segment.angle -ouverture/2 - dev/2 + i*dev,
                     Couleur.getColor(aleat(nbc)), false);
            }
          }
          break;
        }
      }  // if
    }  // while

    listCour = listSuiv;
    repaint();
    Utile.pause (300);
  }  // while

} // dessinerFeuArt
```

```
  public void run () {
    for (;;) {  // for ever
      Dimension d = getSize();         // taille actuelle de la fenêtre
      int w4  = (int) d.width/4;
      int w2  = (int) d.width/2;
      int h4  = (int) d.height/4;
      int h10 = (int) d.height/10;
      // déclencher une fusée en un point du bas de l'écran
      // avec des coordonnées et un angle (de 45 à 135) aléatoires
      Point pp = new Point (w4 + aleat(w2), d.height);
      dessinerFeuArt (d, h4 + aleat(h10), pp, 45 + aleat(90));
      Utile.pause (600);  // on attend 600 ms entre chaque tir
    }
  }  // run

  public void paint (Graphics g) {
    // dessine les Segment de la liste courante
    listCour.ouvrirListe ();
    while (!listCour.finListe ()) {
      Segment segment = (Segment) listCour.objetCourant();
      segment.paint (g);
    }
  }  // paint

} // class FeuArt
```

Exercice 9.3 : Mouvement perpetuel de balles

voir page 353

La classe Cadre :

```
// Cadre.java  Applet ou Application
//                   jeu de balles dans un cadre mouvant

import java.awt.*;           // Graphics

// le Cadre où évoluent les balles : ce n'est pas un composant
public class Cadre {
  // Les attributs de l'objet cadre mobile
  int maxLarg;     // largeur max du cadre
  int maxHaut;     // hauteur max
  int largeur;     // largeur courante du cadre
  int hauteur;     // hauteur courante
  int vitLarg = 3; // vitesse en x du cadre
  int vitHaut = 2; // vitesse en y du cadre

  // crée un cadre de largeur x hauteur;
  // de taille maximum maxLarg x maxHaut
  public Cadre (Dimension di) {
    this.largeur = di.width;
    this.maxLarg = di.width;
    this.hauteur = di.height;
    this.maxHaut = di.height;
  }
```

```
  // retourne largeur courante ou hauteur courante du cadre
  int largeur () { return largeur; };
  int hauteur () { return hauteur; };

  // affiche le cadre en rouge
  public void afficher (Graphics g) {
    g.setColor (Color.red);
    g.drawRect (0, 0, largeur, hauteur);
  }

  // modifie la taille du cadre où évoluent les balles;
  // largeur et hauteur du cadre doivent être > à 4 diametre
  void modifTaille (int diametre) {
    int min = 4*diametre;  // taille minimum du cadre
    largeur += vitLarg;    // nouvelle largeur
    hauteur += vitHaut;    // nouvelle hauteur
    // si on sort du cadre, on change le signe de la vitesse
    if (largeur < min) { largeur = min; vitLarg *= -1;}
    if (hauteur < min) { hauteur = min; vitHaut *= -1;}
    if (largeur > maxLarg - diametre) {
      largeur = maxLarg - diametre; vitLarg *= -1;
    }
    if (hauteur > maxHaut - diametre) {
      hauteur = maxHaut - diametre; vitHaut *= -1;
    }
  }  // modifTaille

}  // class Cadre
```

La classe ZoneBalles de dessin des balles :

```
// ZoneBalles.java  Applet ou Application
//                  jeu de balles dans un cadre mouvant

import java.awt.*;           // Graphics
import mdpaquetage.mdawt.*;  // Balle, Couleur

// la zone où on affiche le cadre
// et les balles du tableau tabBalles
public class ZoneBalles extends Canvas {
  Cadre   cadre      = null;
  Balle[]  tabBalles = null;

  public ZoneBalles (Cadre cadre, Balle[] tabBalles) {
    this.cadre     = cadre;
    this.tabBalles = tabBalles;
    repaint();
  }

  // si la taille du Canvas n'est pas connue
  // lors de la construction de ZoneBalles
  void setZoneBalles (Cadre cadre, Balle[] tabBalles) {
    this.cadre     = cadre;
    this.tabBalles = tabBalles;
    repaint();
  }
```

```
  // dessiner le cadre et les balles
  public void paint (Graphics g) {
    // Afficher l'écran et les balles
    if (cadre != null) cadre.afficher (g);
    if (tabBalles != null) {
      for (int i=0; i < tabBalles.length; i++) tabBalles [i].paint (g);
    }
  }  // paint

} // ZoneBalles
```

Le Panel des balles (champs de saisie et zone de dessin) :

```
// PanelBalle.java  Applet ou Application
//                   jeu de balles dans un cadre mouvant

import java.awt.*;          // Graphics
import javax.swing.*;          // Graphics
import java.awt.event.*;    // ActionListener
import java.applet.*;       // Applet
import mdpaquetage.mdawt.*; // Balle, Couleur
import mdpaquetage.utile.*; // aleat


//le Panel des champs de saisie et de la zone de dessin
public class PanelBalle extends JPanel implements Runnable {
  Cadre      cadre;          // le cadre mobile entourant les balles
  int        nbBalles = 4;   // par défaut
  int        diametre = 15;  // par défaut
  int        vitesse  = 10;  // par défaut
  int        nb = 0;         // pour ModifTaille appelé 1 fois sur 8
  Image      dBuf = null;    // double buffer
  Graphics   gDBuf;          // double buffer
  AudioClip  son = null;
  JTextField tF1, tF2, tF3;
  ZoneBalles  p2;
  BalleMobile[]  tabBalles;  // Tableau de balles

  public PanelBalle() {
    this (4, 15, 10);
  }

  public PanelBalle (int nbBalles, int diametre, int vitesse) {
    this.nbBalles = nbBalles;
    this.diametre = diametre;
    this.vitesse = vitesse;

    setBackground (Color.cyan);
    setLayout (new BorderLayout());
    JPanel p1 = new JPanel ();
    p1.setLayout (new GridLayout(0, 6));
    JLabel l1 = new JLabel ("nb Balles ");
    p1.add (l1);
    tF1 = new JTextField (Integer.toString (nbBalles));
```

```
    p1.add (tF1);
    JLabel l2 = new JLabel ("diametre ");
    p1.add (l2);
    tF2 = new JTextField (Integer.toString (diametre));
    p1.add (tF2);
    JLabel l3 = new JLabel ("vitesse ");
    p1.add (l3);
    tF3 = new JTextField (Integer.toString (vitesse));
    p1.add (tF3);
    tF1.addActionListener (new nbBalles());
    tF2.addActionListener (new diametre());
    tF3.addActionListener (new vitesse());
    p1.setBackground (Color.lightGray);
    add (p1, "North");
    p2 = new ZoneBalles (cadre, tabBalles);
    add (p2, "Center");

  }  // PPBalle

  private int carre (int a, int b) {
    return a*a + b*b;
  }

  // change le signe des vitesses s'il y a choc en sens contraire
  // suivant x ou suivant y
  public void chocBalles () {
    for (int i=0; i < nbBalles; i++) {
      for (int j=i+1; j < nbBalles; j++) {
        // future position des 2 balles i et j
        int xi    = tabBalles[i].getX();
        int xj    = tabBalles[j].getX();
        int yi    = tabBalles[i].getY();
        int yj    = tabBalles[j].getY();
        int vitxi = tabBalles[i].vitX();
        int vitxj = tabBalles[j].vitX();
        int vityi = tabBalles[i].vitY();
        int vityj = tabBalles[j].vitY();

        int t1x = xi + vitxi;
        int t2x = xj + vitxj;
        int t1y = yi + vityi;
        int t2y = yj + vityj;

        boolean choc = carre (t1x - t2x, t1y - t2y)
           <= tabBalles[i].getDiametre()*tabBalles[j].getDiametre();

        if (choc) {
          if (nbBalles == 2 && son != null) son.play();
          tabBalles[i].setVitesseX (-vitxi);  // même sens en x
          if (vitxi * vitxj < 0) {
            // les balles i et j en sens contraire sur l'axe des x
```

```
          // on change le signe des 2 vitesses en y
          tabBalles[j].setVitesseX (-vitxj);
        }
        tabBalles[i].setVitesseY (-vityi);  // même sens en y
        if (vityi * vityj < 0) {
          // les balles i et j en sens contraire sur l'axe des y
          // on change le signe des 2 vitesses en y
          tabBalles[j].setVitesseY (-vityj);
        }
      }

    }  // for j
  }  // for i

}  // chocBalles

public void deplacerBalles (int diametre) {
  chocBalles();  // change le signe des vitesses si besoin
  for (int i=0; i < nbBalles; i++) {
    tabBalles[i].deplacer (cadre.largeur, cadre.hauteur);
  }
  // tous les 8 coups, on modifie la taille de l'écran
  nb++; if (nb % 8 == 0) cadre.modifTaille (diametre);
  p2.repaint();
}

void initDonnees () {
  Dimension di = p2.getSize();
  cadre = new Cadre (di);
  tabBalles = new BalleMobile [nbBalles];  // tableau de balles
  for (int i=0; i < nbBalles; i++) {
    int numCoul;  // numéro de la couleur
    int numFond = Couleur.numeroCouleur(getBackground());
    do {
      numCoul = Utile.aleat (13);
    } while (numCoul == numFond); // sauf la couleur du fond

    // la vitesse est aléatoire de 2 à 7
    tabBalles [i] = new BalleMobile (Couleur.getColor(numCoul),
                                     diametre, di);
  }  // for
  p2.setZoneBalles (cadre, tabBalles);
}

public void run () {
  initDonnees();
  while (true) {
    deplacerBalles (diametre);
    Utile.pause (200/vitesse);
  }
}

// Double buffer
```

```
  public void update (Graphics g) {
    int w = getSize().width;
    int h = getSize().height;
    if (dBuf == null) {
      dBuf  = createImage (w, h);
      gDBuf = dBuf.getGraphics();
    }
    // on dessine dans le contexte graphique gDBuf
    gDBuf.setColor (p2.getBackground());
    gDBuf.fillRect (0, 0, w, h);
    paint (gDBuf);
    Graphics gp2 = p2.getGraphics();
    gp2.drawImage (dBuf, 0, 0, this);  // on affiche le dessin
  }  // update

  class nbBalles implements ActionListener {
    public void actionPerformed (ActionEvent event) {
      String S = tF1.getText ();
      int v = Integer.parseInt (S);
      if (v < 0) v = 1;
      if (v > 50) v = 50;
      nbBalles = v;
      tF1.setText ("" + v);
      initDonnees();
    }
  }

  class diametre implements ActionListener {
    public void actionPerformed (ActionEvent event) {
      String S = tF2.getText ();
      int v = Integer.parseInt (S);
      if (v < 5)  v = 5;
      if (v > 50) v = 50;
      diametre = v;
      tF2.setText ("" + v);
      initDonnees();
    }
  }

  class vitesse implements ActionListener {
    public void actionPerformed (ActionEvent event) {
      String S = tF3.getText ();
      int v = Integer.parseInt (S);
      if (v < 2)  v = 2;
      if (v > 20) v = 20;
      vitesse = v;
      tF3.setText ("" + v);
      initDonnees();
    }
  }

}  // PanelBalle
```

L'applet utilisant deux composants PanelBalle :

```
// AppletBalle.java  Applet ou Application
//                   jeu de balles dans un cadre mouvant

import java.awt.*;           // Graphics
import javax.swing.*;        // JApplet
import mdpaquetage.mdawt.*;  // Balle, Couleur

// pour Applet
public class AppletBalle extends JApplet {

  public void init () {

    // nombre de balles
    int nbBalles = getParam ("nombre", 1, 20, 4);
    // diamètre des balles
    int diametre = getParam ("diametre", 5, 50, 12);
    // vitesse des balles
    int vitesse  = getParam ("vitesse", 1, 20, 10);

    Container f = this.getContentPane();
    f.setLayout (new GridLayout (0, 1));
    PanelBalle p1 = new PanelBalle (nbBalles, diametre, vitesse);
    PanelBalle p2 = new PanelBalle (nbBalles, diametre, vitesse);
    f.add (p1);
    f.add (p2);
    validate();  // calcule l'espace de chaque composant
                 // à faire avant l'appel de initDonnees();
                 // dans le Thread

    // AudioClip pour le choc de deux balles (pour une Applet)
    p1.son = getAudioClip (getCodeBase(), "audio/beep.au");
    p2.son = getAudioClip (getCodeBase(), "audio/beep.au");
    // Thread pour l'animation des balles
    new Thread (p1).start();
    new Thread (p2).start();
  } // init

  int getParam (String nom, int min, int max, int vdefaut) {
    int valeur = vdefaut;
    String sValeur = getParameter (nom);
    if (sValeur != null) {
     valeur = Integer.parseInt (sValeur);
     if (valeur > max) valeur = max;
     if (valeur < min)  valeur = min;
    }
    return valeur;
  }  // getParam

}  // AppletBalle
```

L'application utilisant deux composants PanelBalle :

```
// Application
class PPBalle extends JFrame {

  PPBalle () {
    Container f = this.getContentPane();
    setTitle ("Jeu de Balles");
    setBounds (10, 10, 500, 500);
    addWindowListener (new FermerFenetre());
    f.setLayout ( new GridLayout (0, 1, 10, 10));
    PanelBalle p1 = new PanelBalle ();
    PanelBalle p2 = new PanelBalle ();
    f.add (p1);
    f.add (p2);
    setVisible (true);

    Thread tache1 = new Thread (p1);
    tache1.start();
    Thread tache2 = new Thread (p2);
    tache2.start();
  }

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

}  // PPBalle
```

Exercice 9.4 : Trempoline

```
                                                          // voir page 355
// AppletTrempo.java     animation : applet ou application

import java.awt.*;           // Image, Dimension
import javax.swing.*;        // JPanel
import mdpaquetage.utile.*;  // Utile.aleat

// le composant trempoline = un sauteur au trempoline
class SauteurTrempoline extends JPanel implements Runnable {
  Image  trempoimg[];
  Image  courante;
  int    xpos;
  int    ypos;
  int    dh;
  static int marge = 40; // haut et bas
  static int vitesse[]  = {1, 2, 3, 4, 5, 6, 6, 5, 4, 3, 2, 1};
                                                        // 12 valeurs

  // on passe en paramètre le tableau des images du sauteur
  SauteurTrempoline (Image trempoimg[]) {
    this.trempoimg = trempoimg;
    setBackground (Color.white);
    courante = trempoimg[0];
  }
```

```
// animation du sauteur au trempoline
public void run () {

  while (true) {  // boucle infinie d'animation
    // la hauteur atteinte varie avec le temps (de 1 à 12)
    // monte 12 fois en (16 étapes de montée,
    // tourne, 16 étapes de descente
    pause (Utile.aleat (1000));
    for (int k=0; k < vitesse.length; k++) {
      // dans une application les dimensions peuvent être changées
      // en cours d'application
      Dimension d = getSize();
      ypos = d.height - marge;
      xpos = d.width/2 - 24;  // Taille image = 48
      dh   = (d.height-marge) / 100;

      // maximum = 16* (6/100) = 96/100
      for (int i=0; i < 16; i++) {
        ypos -= vitesse[k]*dh;
        courante = i < 8 ? trempoimg[0] : trempoimg[1];  // bras
        repaint();
        pause (10+i*5);
      }

      // tourne sur place
      int nTour = 1;
      if (vitesse[k] ==6) nTour = 3;
      if (vitesse[k] ==5) nTour = 2;
      if (vitesse[k] <=2) nTour = 0;
      for (int n=1; n <=nTour; n++) {
        for (int i=2; i <=4; i++) {
          courante = trempoimg[i];
          repaint();
          pause (200/nTour);
        }
      }

      // descend en 16 étapes
      for (int i=0; i < 16; i++) {
        ypos += vitesse[k]*dh;
        courante = trempoimg[0];  // bras en bas
        repaint();
        pause (10+90-i*5);
      }

    }  // for
  }  // while
}  // run

void pause (int time) {
  Utile.pause (time);
}
```

```
  // on dessine l'image courante
  public void paint(Graphics g) {
    g.drawImage (courante, xpos, ypos, this);
  }

}  // class Sauteurtrempoline
```

Application

```
// Application trempoline
class PPtrempo extends JFrame {

  String[] tremposrc = { "md1.gif", "md2.gif", "md3.gif",
                         "md4.gif", "md5.gif" };
  Image[]  trempoimg;

  PPtrempo (String nomrepimages) {
    // lecture des images du sauteur
    Toolkit tk = getToolkit();
    Image[] trempoimg = new Image[5];
    for (int i=0; i < trempoimg.length; i++) {
      trempoimg[i] = tk.getImage (nomrepimages + "/" + tremposrc[i]);
    }

    setTitle  ("trempoline");
    setBounds (10, 10, 400, 400);
    int n = getSize().width / 70;
    Container f = getContentPane ();
    f.setBackground(Color.cyan);
    f.setLayout (new GridLayout(0, n, 10, 10));

    // n sauteurs de trempoline
    // (dépend de la largeur de fenêtre de type Frame
    SauteurTrempoline[] tr = new SauteurTrempoline [n];
    for (int i=0; i < tr.length; i++) {
      tr[i] = new SauteurTrempoline (trempoimg);
      f.add (tr[i]);
    }
    setVisible(true);
    // les threads doivent être lancés après setVisible
    // pour que chacun connaisse la place qu'il occupe
    for (int i=0; i < tr.length; i++) {
      new Thread (tr[i]).start();
    }
  }

  public static void main (String[] args) {
    String nomrepimages = (args.length == 0) ? "." : args[0];
    new PPtrempo (nomrepimages);
  }

}  // PPTrempo
```

Applet

```
public class Applettrempo extends JApplet {
  String[] tremposrc = { "md1.gif", "md2.gif", "md3.gif",
                         "md4.gif", "md5.gif" };
  Image[]  trempoimg;

  int getParam (String nom, int min, int max, int vdefaut) {
    int valeur = vdefaut;
    String sValeur = getParameter (nom);
    if (sValeur != null) {
      valeur = Integer.parseInt (sValeur);
      if (valeur > max) valeur = max;
      if (valeur < min)  valeur = min;
    }
    return valeur;
  }

  public void init () {
    Image trempoimg[] = new Image[5];
    for (int i=0; i < trempoimg.length; i++) {
      trempoimg[i] = getImage (getCodeBase(),
                                              "images/" + tremposrc[i]);
    }

    int n = getSize().width / 70;
    n = getParam ("nombre", 1, n, n);
    Container f = getContentPane();
    f.setLayout (new GridLayout(0, n, 10, 10));
    setBackground (Color.white);

    SauteurTrempoline[] tr = new SauteurTrempoline [n];
    for (int i=0; i < tr.length; i++) {
      tr[i] = new SauteurTrempoline (trempoimg);
      f.add (tr[i]);
    }
    validate();
    // les threads doivent être lancés après validate
    // pour que chacun connaisse la place qu'il occupe
    for (int i=0; i < tr.length; i++) {
      new Thread (tr[i]).start();
    }
  }

}  // AppletTrempo
```

Exercice 9.5 : Bataille navale

// voir page 356

```
// AppletBatNav.java  Applet ou Application
import java.awt.*;           // Frame
import java.awt.event.*;     // ActionListener
import java.applet.*;        // interface AudioClip
```

```
import mdpaquetage.mdawt.*;  // FermerFenetre
import java.net.*;           // URL

class BatNav extends Panel {
  static final int NBBAT   = 8;  //nb Bateaux
  static final int INCONNU = 0;  // état de la case à découvrir
  static final int BATEAU  = 1;  // un bateau
  static final int ENVUE   = 2;  // en vue à partir de (i,j)
  static final int RATE    = 3;  // raté
  static final int RIEN    = 4;  // case vide

  int        nbCoules = 0;  // Nombre de bateaux coulés
  int        nbTirs   = 0;  // Nombre de tirs effectués
  Label      l1;  // bateaux restants
  Label      l2;  // nombre de tirs
  int[][]    bateau   = new int    [NBBAT][NBBAT];
  Button[][] bouton   = new Button [NBBAT][NBBAT];
  String[]   nomEtat  = { ".......", "Bateau", "EnVue",
                                                    "Raté", "Rien"};
  int[][]    position = new int [NBBAT][2]; // position des bateaux
                                                          coulés
  AudioClip  enVue = null;  // message oral "en vue"
  AudioClip  coule = null;  // message oral "coulé"

  // construit la fenêtre et initialise les données
  BatNav () {
    setLayout (new BorderLayout ());

    Panel p1 = new Panel();
    p1.setLayout (new GridLayout (0, NBBAT));
    ActionListener ecouteTir = new Tir();
    for (int i=0; i < NBBAT; i++) {
      for (int j=0; j < NBBAT; j++) {
        int c = bateau [i][j];
        if (c ==BATEAU) c = INCONNU;  // Ne pas afficher les bateaux
        bouton[i][j] = new Button (nomEtat[c]);
        bouton[i][j].setBackground (Color.cyan);
        bouton[i][j].addActionListener (ecouteTir);
        p1.add (bouton[i][j]);
      }
    }
    add(p1, "Center");

    Panel p2 = new Panel();
    p2.setLayout ( new FlowLayout ());
    l1 = new Label ("Nombre de bateaux restants : " + NBBAT);
    p2.add (l1);
    l2 = new Label ("Nombre de tirs effectués   : " + nbTirs);
    p2.add (l2);
    add(p2, "South");

    initDonnees ();
    setVisible (true);  // après initialisation des données
  }  // constructeur BatNav
```

```
// pour les sons
void setAudio (AudioClip enVue, AudioClip coule) {
  this.enVue = enVue;
  this.coule = coule;
}

// initialise les données (position des bateaux, etc.)
void initDonnees () {
  // initalise les données
  // au début toutes les positions sont inconnues
  for (int i=0; i < NBBAT; i++) {
    for (int j=0; j < NBBAT; j++) {
      bateau [i][j] = INCONNU;
    }
  }

  // placer les NBBAT bateaux aléatoirement
  // pas deux bateaux sur la même case
  int nb = NBBAT;
  while (nb > 0) {
    int i = (int) ((Math.random()*997) % NBBAT);
    int j = (int) ((Math.random()*997) % NBBAT);
    if (bateau [i][j] == INCONNU) {
      bateau [i][j] = BATEAU;
      nb--;
    }
  } // while
  //printbateaux();  // pour vérification
}  // initDonnees

// écrire la position des bateaux sur la sortie standard
// pour vérification; à mettre en commentaires
void printbateaux () {
  String[] etat = { ".", "B", "E", "T", "V"};
  for (int i=0; i < NBBAT; i++) {
    for (int j=0; j < NBBAT; j++) {
      int c = bateau [i][j];
      //if (c ==BATEAU) c = INCONNU; // ne pas afficher les bateaux
      System.out.print (etat[c] + " ");
    }
    System.out.println ();
  }
  System.out.println ();
}

void tir (int i, int j) {
  if (bateau [i][j] == BATEAU) {
    // on a tiré sur un bateau en (i,j)
    bateau [i][j]     = RIEN;
    // on enregistre la position du bateau coulé
    position [nbCoules][0] = i;
```

```
      position [nbCoules][1] = j;
      nbCoules++;
      if (coule!=null) coule.play();
      if (EnVue (i, j)) {
        bateau[i][j] = ENVUE;
      } else {
        RienAutour (i,j);
      }
    } else {
      // pas de bateau en (i,j). Des bateaux en vue ?
      if (EnVue (i, j)) {
         bateau[i][j] = ENVUE;
         if (enVue != null) enVue.play();
      } else {
         bateau [i][j] = RATE;
         RienAutour (i,j);
      }
    }
  }

  boolean EnVue (int i, int j) {
    boolean R = false;
   if ( (i > 0)       && (j > 0)       ) R = R || (bateau[i-1][j-1] == BATEAU);
   if ( (i > 0)                        ) R = R || (bateau[i-1][j]   == BATEAU);
   if ( (i > 0)       && (j < NBBAT-1) ) R = R || (bateau[i-1][j+1] == BATEAU);

   if (                  (j > 0)       ) R = R || (bateau[i]  [j-1] == BATEAU);
   if (                  (j < NBBAT-1) ) R = R || (bateau[i]  [j+1] == BATEAU);

   if ( (i < NBBAT-1) && (j > 0)       ) R = R || (bateau[i+1][j-1] == BATEAU);
   if ( (i < NBBAT-1)                  ) R = R || (bateau[i+1][j]   == BATEAU);
   if ( (i < NBBAT-1) && (j < NBBAT-1) ) R = R || (bateau[i+1][j+1] == BATEAU);
    return R;
  }

  void RienAutour (int i, int j) {
    // Les messages ENVUE ou RATE sont gardés
   if ( (i > 0)       && (j > 0)       && (bateau[i-1][j-1] == INCONNU)) bateau[i-1][j-1] = RIEN;
   if ( (i > 0)                        && (bateau[i-1][j  ] == INCONNU)) bateau[i-1][j]   = RIEN;
   if ( (i > 0)       && (j < NBBAT-1) && (bateau[i-1][j+1] == INCONNU)) bateau[i-1][j+1] = RIEN;
   if (                  (j > 0)       && (bateau[i]  [j-1] == INCONNU)) bateau[i]  [j-1] = RIEN;
   if (                  (j < NBBAT-1) && (bateau[i]  [j+1] == INCONNU)) bateau[i]  [j+1] = RIEN;

   if ( (i < NBBAT-1) && (j > 0)       && (bateau[i+1][j-1] == INCONNU)) bateau[i+1][j-1] = RIEN;
   if ( (i < NBBAT-1)                  && (bateau[i+1][j]   == INCONNU)) bateau[i+1][j]   = RIEN;
   if ( (i < NBBAT-1) && (j < NBBAT-1) && (bateau[i+1][j+1] == INCONNU)) bateau[i+1][j+1] = RIEN;
  }

 void RAZEnVue (int i, int j) {
   if ( (i > 0)       && (j > 0)       && (bateau[i-1][j-1] == ENVUE) && !EnVue (i-1,j-1)) bateau[i-1][j-1] = RIEN;
   if ( (i > 0)                        && (bateau[i-1][j]   == ENVUE) && !EnVue (i-1,j))   bateau[i-1][j]   = RIEN;
   if ( (i > 0)       && (j < NBBAT-1) && (bateau[i-1][j+1] == ENVUE) && !EnVue (i-1,j+1)) bateau[i-1][j+1] = RIEN;
```

```
    if (             (j > 0)        && (bateau[i]  [j-1] == ENVUE) && !EnVue (i,  j-1)) bateau[i]  [j-1] = RIEN;
    if (             (j < NBBAT-1)  && (bateau[i]  [j+1] == ENVUE) && !EnVue (i,  j+1)) bateau[i]  [j+1] = RIEN;

    if ( (i < NBBAT-1) && (j > 0)       && (bateau[i+1][j-1] == ENVUE) && !EnVue (i+1,j-1)) bateau[i+1][j-1] = RIEN;
    if ( (i < NBBAT-1)              && (bateau[i+1][j]   == ENVUE) && !EnVue (i+1,j  )) bateau[i+1][j]   = RIEN;
    if ( (i < NBBAT-1) && (j < NBBAT-1)  && (bateau[i+1][j+1] == ENVUE) && !EnVue (i+1,j+1)) bateau[i+1][j+1] = RIEN;
    }

    class Tir implements ActionListener {
      // on a appuyé sur un bouton (effectué un tir)
      public void actionPerformed (ActionEvent evt) {
        Object SrcEvt = evt.getSource();
        nbTirs++;

        // on cherche la référence (i,j) du bouton appuyé
        for (int i=0; i < NBBAT; i++) {
          for (int j=0; j < NBBAT; j++) {
            if (SrcEvt.equals (bouton[i][j])) {
              tir (i,j);
              RAZEnVue (i, j);
            }
          }
        }

        // modifier les nouveaux Label des boutons
        // suite au tir
        for (int i=0; i < NBBAT; i++) {
          for (int j=0; j < NBBAT; j++) {
            int c = bateau [i][j];
            if (c ==BATEAU) c = INCONNU;  // Ne pas afficher les bateaux
            bouton[i][j].setLabel (nomEtat[c]);
            if (c ==RATE || c ==ENVUE || c ==RIEN) {
              bouton[i][j].setBackground (Color.yellow);
            }
          }
        }

        // afficher en rouge les boutons des bateaux coulés
        // si le bouton contient "EnVue", laisser le message
        for (int n=0; n < nbCoules; n++) {
          int i = position[n][0];
          int j = position[n][1];
          if (bateau[i][j] == RIEN) bouton[i][j].setLabel ("Coulé");
          bouton[i][j].setBackground (Color.red);
        }
        l1.setText("Nombre de bateaux restants : " + (NBBAT-nbCoules));
        l2.setText("Nombre de tirs effectués   : " + nbTirs);
      }
    }  // class Tir

  }  // class BatNav
```

Pour une applet :

```
// AppletBatNav.java  Applet ou Application

import java.applet.*;       // AudioClip
import javax.swing.*;       // JFrame, JApplet
import java.net.*;          // URL

public class AppletBatNav extends JApplet {

  public void init () {
    AudioClip enVue = null, coule = null;
    try {
      enVue = getAudioClip (getCodeBase(), "audio/envue.au");
      coule = getAudioClip (getCodeBase(), "audio/coule.au");
    } catch (Exception e) {
      System.out.println ("Erreur : " + e);
    }
    BatNav bn = new BatNav ();
    bn.setAudio (enVue, coule);
    getContentPane().add (bn, "Center");
  }

}  // AppletBatNav
```

Pour une application :

```
class PPBatNav extends JFrame {
  AudioClip enVue = null, coule = null;

  PPBatNav (String nomrepaudio) {
    try {  // les fichiers son en .wav ou en .au
      URL fson1 = ClassLoader.getSystemResource (nomrepaudio
                                                    + "envue.au");
      enVue = (AudioClip) fson1.getContent();
      URL fson2 = ClassLoader.getSystemResource (nomrepaudio
                                                    + "coule.au");
      coule = (AudioClip) fson2.getContent();
    } catch (Exception e) {
      System.out.println ("Erreur : " + e);
    }
    setTitle ("Bataille navale");
    setBounds (10, 10, 500, 500);
    BatNav bn = new BatNav ();
    bn.setAudio (enVue, coule);
    getContentPane().add (bn, "Center");
    setVisible (true);
  }

  public static void main(String[] args) {
    String nomrepaudio  = (args.length == 0) ? "." : args[0];
    new PPBatNav (nomrepaudio);
  }
} // PPBatNav
```

Bibliographie

JAVA

Arnold K., Gosling J., *The Java Programming Language*, Addison-Wesley.

Berthié V., Briaud J-B., *Swing, la synthèse , développement des interfaces gr aphiques en Java*, Dunod.

Charon I., *Le langage Java*, Hermes.

Clavel G., Mirouze N., Munerot S., Pichon E., Soukal M.,Tiffanneau S., Java, *La synthèse, Des concepts objet aux architectures Web*, Dunod.

Delannoy C., *Programmer en Java*, Eyrolles.

Farinone J-M., *Java et le multimédia*, Dunod.

Harold E., *Programmation réseau avec Java*, O'Reilly.

Lai M., *Penser objet avec UML et Java*, Dunod.

Scott Oaks & Henry Wong, *Java Threads*, O'Reilly.

LES STRUCTURES DE DONNÉES

Divay M., *Algorithmes et structures de données génériques,* 2e éd., Cours et exercices corrigés en langage C, Dunod, 2004, ISBN 2 10 007450 4, 342 pages.

LES SYSTÈMES D'EXPLOITATION

Divay M., *Unix, Linux et les systèmes d'e xploitation*, 2e éd., Cours et exercices corrigés, Dunod, 2004, ISBN 2 10 007451 2, 430 pages.

Index