Université d'Aix-Marseille
Faculté des Sciences de Luminy

Le langage Java

Master CCI, ISMA, I2A, BBSG, etc.

Henri Garreta
Département d'Informatique – LIF

Le langage Java

Henri Garreta
Aix-Marseille Université, Faculté des Sciences de Luminy

Table des matières

Ce poly est en chantier, c'est son état normal.
Ceci est la version du 22 février 2012 (première publication : janvier 2000).

`henri.garreta@univ-amu.fr`

La dernière version de ce document peut être téléchargée à l'adresse
`http://henri.garreta.perso.luminy.univmed.fr/Polys/PolyJava.pdf`

1 Introduction

Ce polycopié est le support de plusieurs cours de Java donnés à la Faculté des Sciences de Luminy. Très succinct, il ne remplace pas la consultation d'ouvrages plus approfondis et illustrés, parmi lesquels :

pour débuter :
Patrick Niemeyer & Jonathan Knudsen
Introduction à Java, 2^{eme} édition
O'Reilly, 2002

deuxième lecture :
Ian Darwin
Java en action
O'Reilly, 2002

Ce texte s'adresse à des lecteurs connaissant le langage C. De nombreux éléments importants de Java, comme les expressions, les instructions, l'appel des fonctions, etc., ne sont pas expliqués ici, tout simplement parce qu'ils sont réalisés en Java comme en C. En outre, l'exposé n'est pas progressif : chaque notion est introduite comme si les autres concepts du langage avaient déjà été traités. Cela rend la lecture initiale de ce document plus éprouvante, mais la consultation ultérieure plus efficace.

Ces notes traitent de la version courante de la plate-forme Java (le langage et ses bibliothèques officielles) et de son environnement de développement, appelés désormais respectivement *Java^{TM} Platform Standard Edition 6* et *Java^{TM} SE Development Kit 6* (le numéro de version 1.6.x apparaît parfois à la place de 6). Cependant, pour la clarté des idées, nous avons introduit la plupart des éléments comme il convenait de le faire à l'époque de Java 1.4 (par exemple, l'héritage et le polymorphisme sont d'abord expliqués sans mentionner ni la généricité ni l'emballage automatique). Les principales nouveautés introduites lors du passage de Java 1.4 à Java 5 sont présentées à la section 12.

Un document est rigoureusement indispensable pour programmer en Java : la documentation en ligne de l'API (*Interface du Programmeur d'Applications*). C'est un fichier hypertexte qui contient tout ce qu'il faut savoir à propos de tous les paquets, interfaces, classes, variables et méthodes de la plate-forme Java. On peut le consulter en ligne et le télécharger à l'adresse `http://java.sun.com/javase/6/docs/api/`

Enfin, un excellent document pour apprendre Java est le *tutoriel* officiel, qu'on peut consulter en ligne ou télécharger à l'adresse `http://java.sun.com/tutorial/`

1.1 Pratique effective de Java

Dans cette section nous supposons que vous travaillez sur un système Windows, Linux, Solaris ou Mac OS X. Certains des produits mentionnés ici ont également des versions pour des systèmes plus confidentiels.

• Pour pratiquer le langage Java, c'est-à-dire pour saisir, compiler et exécuter vos programmes, il vous faut un *environnement de développement Java*, comportant un compilateur, une machine Java et des bibliothèques – au moins la bibliothèque standard. Vous obtenez tout cela en installant le *JDK* (*Java Development Kit*), que vous pouvez télécharger gratuitement depuis le site de Sun, à l'adresse `http://java.sun.com/javase/downloads`.

Vous pouvez utiliser n'importe quel éditeur de textes pour saisir vos programmes. Supposons que vous ayez tapé le texte de la célèbre classe `Bonjour` :

```
public class Bonjour {
    public static void main(String[] args) {
        System.out.println("Bonjour à tous!");
    }
}
```

Le fichier dans lequel vous avez enregistré ce texte *doit* s'appeler `Bonjour.java`. Vous en obtenez la compilation en tapant la commande

```
javac Bonjour.java
```

Si le programme est correct, la compilation se déroule sans produire aucun message. Vous lancez alors l'exécution de votre programme (pour obtenir l'affichage du texte *Bonjour à tous !*) en tapant la commande

```
java Bonjour
```

Notez bien qu'à la commande `javac` on donne un nom de fichier (*Bonjour.java*), tandis qu'à la commande `java` on doit donner un nom de classe (*Bonjour*).

• Quel que soit le système d'exploitation que vous utilisez, vous pouvez vous constituer un environnement de travail plus agréable en installant le *JDK* puis un petit éditeur de textes orienté Java. Vous en trouverez plusieurs sur le web, par exemple *Jext*, à l'adresse `http://www.jext.org/`.

• Vous obtenez un confort incomparablement supérieur si, après *JDK*, vous installez *Eclipse*, un puissant *IDE* (environnement de développement intégré) avec de nombreuses fonctionnalités pour aider à la programmation en Java, telles que la complétion automatique des expressions tapées, la compilation au fur et à mesure de la frappe, la suggestion de la manière de corriger les erreurs, etc. Vous pouvez *librement* télécharger *Eclipse* à l'adresse `http://www.eclipse.org/`

• Enfin, si vous voulez programmer des interfaces graphiques rien qu'en jouant de la souris [1] vous pouvez *gratuitement* installer *NetBeans* de Sun (`http://www.netbeans.org/`) ou *JBuilder Foundation* de Borland (`http://www.borland.fr/jbuilder/`).

1. Si vous débutez en Java nous vous déconseillons de commencer par là.

2 Considérations lexicales

2.1 Jeu de caractères

Java utilise le codage des caractères UTF-16, une implémentation de la norme *Unicode*, dans laquelle chaque caractère est représenté par un entier de 16 bits, ce qui offre 65 536 possibilités au lieu des 128 (ASCII) ou 256 (Latin-1) imposés par d'autres langages de programmation. Il y a donc de la place pour la plupart des alphabets nationaux ; en particulier, tous les caractères français y sont représentés sans créer de conflit avec des caractères d'autres langues.

Cela vaut pour les données manipulées par les programmes (caractères et chaînes) et aussi pour les programmes eux-mêmes. Il est donc possible en Java de donner aux variables et aux méthodes des noms accentués. Mais on ne vous conseille pas d'utiliser cette possibilité, car vous n'êtes jamais à l'abri de devoir un jour consulter ou modifier votre programme à l'aide d'un éditeur de textes qui ne supporte pas les accents.

2.2 Commentaires

Il y a trois sortes de commentaires. D'abord, un texte compris entre `//` et la fin de la ligne :

```
int nbLignes;   // nombre de variables du système
```

Ensuite, un texte compris entre `/*` et `*/`, qui peut s'étendre sur plusieurs lignes :

```
/* Recherche de l'indice ik du "pivot maximum"
   c.-à-d. tel que k <= ik < nl et
   a[ik][k] = max { |a[k][k]|, ... |a[nl - 1][k]| } */
```

Enfin, cas particulier du précédent, un texte compris entre `/**` et `*/` est appelé *commentaire de documentation* et est destiné à l'outil `javadoc` qui l'exploite pour fabriquer la documentation d'une classe à partir de son texte source [2]. Cela ressemble à ceci (un tel commentaire est associé à l'élément, ici une méthode, qu'il précède) :

```
/**
 * Résolution d'un système linéaire
 *
 * @param a Matrice du système
 * @param x Vecteur solution du système
 * @return <tt>true</tt> si la matrice est inversible, <tt>false</tt> sinon.
 * @author Henri G.
 */
public boolean resolution(double a[][], double x[]) {
    ...
```

2.3 Identificateurs

Les règles de formation des identificateurs sont les mêmes que dans le langage C, compte tenu des remarques du § 2.1. La note *Java Code Conventions* [3] fait les recommandations suivantes à propos de la manière de nommer les entités des programmes :

1. Les différents identificateurs, séparés par des points, qui forment un nom de paquet sont écrits principalement en minuscules, surtout le premier : `java.awt.event`, `monprojet.mesoutils`.
2. Le nom d'une classe ou d'une interface est fait d'un mot ou de la concaténation de plusieurs mots. Chacun commence par une majuscule, y compris le premier : `Point`, `AbstractListModel`, `ListeDeNombres`, etc.
3. Les noms des méthodes commencent par une minuscule. Lorsqu'ils sont formés par concaténation de mots, chacun sauf le premier commence par une majuscule : `toString()`, `vitesseDuVent(...)`, etc.
4. La même règle vaut pour les noms des variables, aussi bien les variables de classe et d'instance que les arguments et les variables locales des méthodes : `i`, `nombreDeMembres`, etc.
5. Enfin, les constantes de classe (c'est-à-dire les variables `static final`) sont écrites de préférence en majuscules : `Integer.MAX_VALUE`, `TextField.LEFT_ALIGNMENT`, etc.

2. Par exemple, la documentation en ligne de l'API, ce gigantesque document hypertexte qui est le compagnon inséparable de tout programmeur Java, est automatiquement produit au moyen de `javadoc` à partir des fichiers sources de la bibliothèque standard.

3. On peut télécharger cette note depuis le site `http://java.sun.com/docs/codeconv/`

2.4 Constantes littérales

Les règles pour l'écriture des constantes sont les mêmes qu'en C, avec les *additions* suivantes :

1. *Caractères*. Les caractères non imprimables peuvent être représentés aussi par la séquence `\u`*xxxx*, où *xxxx* sont les quatre chiffres hexadécimaux du code numérique du caractère. Par exemple, la chaîne suivante comporte les quatre caractères `A`, espace (code 32, ou 20 en hexadécimal), `B` et `ñ` (code 241, ou F1 en hexadécimal) :

   ```
   "A\u0020B\u00F1"    // la chaîne de quatre caractères "A Bñ"
   ```

2. *Entiers*. Une constante littérale entière est supposée être du type `int`, c'est-à-dire codée sur 32 bits, sauf si elle est trop grande pour être représentée dans ce type ; elle appartient alors au type `long`. La lettre `L` ou `l` indique que la constante doit être considérée comme appartenant au type `long` et codée sur 64 bits, même si sa valeur est petite. Exemple :

   ```
   1234                // une valeur du type int
   1234L               // une valeur du type long
   ```

   Dans l'initialisation d'une variable de type `long` par une valeur de type `int` le compilateur insère automatiquement la conversion nécessaire. On peut alors se demander dans quelle circonstance on est gêné par le fait qu'une valeur entière soit considérée `int` au lieu de `long`. Le problème réside le plus souvent dans l'incapacité de représenter les résultats des opérations. Par exemple, l'affectation

   ```
   long u = 4000000 * 6000000;      // ERREUR (débordement de la muliplication)
   ```

   est certainement incorrecte, car `4000000` et `6000000` seront considérés de type `int`, et donc la multiplication et son résultat aussi ; or ce résultat ne peut pas être représenté dans les 32 bits d'un entier (voyez le tableau 1, page 11). Bien entendu, que ce résultat soit *ensuite* affecté à une variable de type `long` ne le rend pas juste. De plus, s'agissant d'un débordement en arithmétique entière, aucune erreur ne sera signalée ni à la compilation ni à l'exécution (mais le résultat sera faux).

   Pour corriger ce problème il suffit placer l'opération et son résultat dans le type `long` ; à cause de la « règle du plus fort » [4], il suffit pour cela qu'un des opérandes le soit :

   ```
   long u = 4000000L * 6000000;      // OK
   ```

3. *Flottants*. Une constante flottante (c'est-à-dire un nombre comportant un point et/ou un exposant, comme `1.5`, `2e-12` ou `-0.54321E3`) représente une valeur du type double. Les suffixes `f` ou `F` peuvent être utilisés pour préciser le type de ce qui est écrit.

   Ainsi, une simple affectation telle que

   ```
   float x = 1.5;      // ERREUR (discordance de types)
   ```

   est signalée comme une erreur. Correction :

   ```
   float x = 1.5f;
   ```

   On peut aussi laisser la constante être du type `float` et en demander explicitement la conversion :

   ```
   float x = (float) 1.5;
   ```

4. *Booléens*. Le type boolean comporte deux valeurs, représentées par les constantes littérales suivantes, qui sont aussi des mots réservés :

   ```
   false               // la valeur booléenne faux
   true                // la valeur booléenne vrai
   ```

4. *Grosso modo* cette règle dit que si les deux opérandes d'une opération ne sont pas de même type alors le plus « fort » (c.-à-d. le plus étendu, ou le plus précis, ou un flottant face à un entier, etc.) provoque la conversion vers son type de l'autre opérande et le placement dans ce type de l'opération et de son résultat. Cette règle s'applique dans la grande majorité des opérations binaires.

3 Types

3.1 Les types de données de Java

Les types des données manipulées par les programmes Java se répartissent en deux catégories :
– les *types primitifs*,
– les *objets*, c'est-à-dire les *tableaux* et les *classes*.
La table 1 récapitule ce qu'il faut savoir sur les huit types primitifs.

TABLE 1 – Les types *primitifs* de Java

Type	*description*	*taille*	*ensemble de valeurs*	*val. init.*
Nombres entiers				
`byte`	Octet	8 bits	de -128 à 127	`0`
`short`	Entier court	16 bits	de $-32\,768$ à $32\,767$	
`int`	Entier	32 bits	de $-2\,147\,483\,648$ à $2\,147\,483\,647$	
`long`	Entier long	64 bits	de -2^{63} à $2^{63}-1$	
Nombres flottants				
`float`	Flottant simple précision	32 bits	de $-3,4028235\times10^{+38}$ à $-1,4\times10^{-45}$, 0 et de $1,4\times10^{-45}$ à $3,4028235\times10^{+38}$	`0.0`
`double`	Flottant double précision	64 bits	de $-1,7976931348623157\times10^{+308}$ à $-4,9\times10^{-324}$, 0 et de $4,9\times10^{-324}$ à $1,7976931348623157\times10^{+308}$	
Autres types				
`char`	Caractère	16 bits	Tous les caractères Unicode	`'\u0000'`
`boolean`	Valeur booléenne	1 bit	`false`, `true`	`false`

On notera que :
– tous les types entiers sont « signés » (c'est-à-dire représentent des ensembles contenant des nombres positifs et négatifs),
– portabilité oblige, on n'a pas reproduit en Java la bourde consistant à laisser la taille des `int` à l'appréciation de chaque compilateur.

3.2 Conversions entre types primitifs

La question de la compatibilité et de la convertibilité entre expressions de types primitifs est réglée en Java selon un petit nombre de principes simples :

1. Toute valeur d'un type numérique est convertible vers tout autre type numérique, selon les modalités expliquées ci-après.
2. Les conversions entre types numériques sans risque de perte d'information (c'est-à-dire : entier ou caractère vers entier de taille supérieure, flottant vers flottant de taille supérieure, entier ou caractère vers flottant) sont faites, à la compilation et à l'exécution, sans requérir aucune indication particulière.
   Par exemple, si les variables qui y figurent ont les types que leurs noms suggèrent, les affectations suivantes ne soulèvent le moindre problème, ni à la compilation ni à l'exécution :
   ```
   unInt = unChar;
   unDouble = unFloat;
   unFloat = unInt;
   ```
3. Les conversions avec risque de perte d'information (c'est-à-dire : entier vers entier de taille inférieure, flottant vers flottant de taille inférieure, flottant vers entier) sont refusées par le compilateur :
   ```
   unByte = unInt;                    // ERREUR
   ```
   sauf si le programmeur a explicitement utilisé l'*opérateur de transtypage* (ou *cast operator*) :
   ```
   unByte = (byte) unInt;             // Ok
   ```
   ATTENTION. Une expression comme la précédente est placée sous la responsabilité du programmeur, seul capable de garantir que la valeur de `unInt` pourra, au moment de l'affectation ci-dessus, être représentée dans une variable de type `byte`. Cette condition ne peut pas être vérifiée à la compilation

et elle ne l'est pas non plus à l'exécution. Cela peut conduire à des troncations inattendues, non signalées, et donc à des résultats absurdes. Par exemple, le programme suivant affiche la valeur 1 :

```
int unInt = 257;
byte unByte = (byte) unInt;     // DANGER
System.out.println(unByte);
```

4. Puisque les caractères sont représentés par des nombres, le type `char` est considéré comme numérique, pour ce qui nous occupe ici. Cependant, la conversion de n'importe quel type numérique vers le type `char` est considérée comme susceptible d'entraîner une perte de précision ; cela oblige le programmeur à utiliser l'opérateur de transtypage.
5. La conversion d'une expression booléenne vers un type numérique, ou réciproquement, est illégale et rejetée par le compilateur. Les expressions « à la mode de C » suivantes sont donc erronées en Java :

```
if (unInt) ...        // ERREUR (nécessite une conversion int → boolean)
unInt = (x < y);      // ERREUR (nécessite une conversion boolean → int)
```

Bien entendu, ces conversions peuvent être obtenues facilement :
- conversion booléen → entier : `(unBooleen ? 1 : 0)`
- conversion nombre → booléen : `(unNombre != 0)`

3.3 Références

3.3.1 Sémantique des valeurs et sémantique des références

Les expressions de types primitifs ont la *sémantique des valeurs*. Cela signifie que si e est une expression d'un type primitif T, alors à tout endroit [5] où elle figure *e représente une valeur* du type T. Par exemple, si `unInt` est une variable entière contenant la valeur 12, alors les trois expressions 12, `unInt` et `unInt`$/4+9$ ont la même signification : la valeur entière 12.

Notez que des trois expressions 12, `unInt` et `unInt`$/4+9$, une seule est associée à un emplacement dans la mémoire (la seconde). Si une expression a la sémantique des valeurs elle n'est pas forcément un renvoi à un objet existant dans la mémoire.

Tous les autres types de Java, c'est-à-dire les tableaux et les classes, ont la *sémantique des références*. Cela veut dire qu'une expression d'un type tableau ou classe est, en toute circonstance, une référence sur un objet existant dans la mémoire [6].

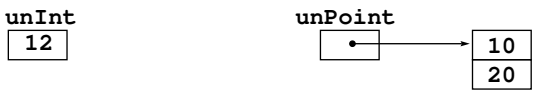


FIGURE 1 – Sémantique des valeurs et sémantique des références

On peut voir une référence comme un pointeur [7] géré de manière interne par Java. Par exemple, si `Point` est une classe formée de deux entiers `x` et `y` (les coordonnées d'un point) :

```
class Point {
    int x, y;
    ...
}
```

et si `unPoint` est une variable de type `Point` correctement initialisée :

```
Point unPoint = new Point(10, 20);
```

alors il convient de se représenter la variable `unPoint` et sa valeur comme le montre la figure 1.

Bien noter qu'en Java aucun signe ne rappelle, lors de l'emploi d'une expression de type référence, qu'il s'agit d'une sorte de pointeur (c'est-à-dire, on n'utilise pas les opérateurs `*` ou `->` de C). Puisque les objets sont *toujours* accédés par référence, une expression comme

5. Comme dans tous les langages qui ont la notion d'*affectation*, il y a une exception : placée au membre gauche d'une affectation, une expression ne représente pas une valeur mais un emplacement de la mémoire.
6. Plus précisément, les références sont des renvois à la *heap*, c'est-à-dire la partie de la mémoire dans laquelle se font les allocations dynamiques (par `new`, le moyen de créer un tableau ou un objet).
7. Attention, abus de langage ! Imaginer un pointeur est sans doute une bonne manière d'appréhender une référence, mais il ne faut pas oublier qu'en Java on s'interdit de parler de pointeurs, car le programmeur n'est pas censé connaître les détails du procédé par lequel Java désigne de manière interne les objets qu'il crée dans la mémoire.


```
unPoint.x
```

est sans ambiguïté ; elle signifie : le champ `x` de l'objet référencé par `unPoint`.

3.3.2 Référence "sur rien"

La valeur particulière `null` indique qu'une variable d'un type référence n'est pas une référence à un objet valide. C'est la valeur qu'il convient de donner à une telle variable lorsque l'objet référencé n'existe pas encore, ou bien lorsque la variable a cessé d'être utile en tant que référence sur un objet :

```
Point unPoint = null;       // déclaration et initialisation de unPoint
   ...                      // ici, unPoint ne représente pas un objet valide
unPoint = new Point(10, 20);
   ...                      // ici on peut travailler avec l'objet unPoint
unPoint = null;
   ...                      // à nouveau, unPoint n'est pas un objet valide
```

Le compilateur se charge d'initialiser à `null` les variables d'instance et de classe d'un type objet qui n'ont pas d'initialisation explicite. Les variables locales ne sont pas initialisées [8].

3.3.3 Cas des paramètres des méthodes

Les explications précédentes sont valables pour toutes sortes d'expressions, quels que soient les contextes où elles apparaissent (sauf les membres gauches des affectations, qui échappent à cette discussion). Cela vaut en particulier pour les paramètres effectifs des appels des méthodes ; par conséquent :

- un paramètre d'un type est *passé par valeur* : lors de l'appel de la méthode, la *valeur* du paramètre effectif est copiée dans le paramètre formel correspondant, qui est une variable locale de la méthode ; en particulier, si un tel paramètre effectif est une variable, l'appel de la méthode ne pourra pas en changer la valeur, quoi qu'elle fasse au paramètre formel correspondant ;
- un paramètre d'un type tableau ou classe est *passé par référence* [9] : lors de l'appel de la méthode, le paramètre formel est initialisé avec une *référence* sur le tableau ou l'objet que représente le paramètre effectif ; il en découle que le tableau ou l'objet en question pourra être réellement modifié par les actions que la méthode fera sur le paramètre formel.

3.4 Tableaux

3.4.1 Déclaration et initialisation

Tous les tableaux utilisés en Java sont des *tableaux dynamiques*. Cela veut dire que le nombre d'éléments :
- n'est pas un élément constitutif du type du tableau et n'a pas besoin d'apparaître dans sa déclaration,
- n'est pris en compte qu'au moment de l'exécution du programme, même s'il est connu pendant la compilation.

Il y a deux syntaxes rigoureusement équivalentes pour déclarer un tableau `tab` dont les éléments sont, par exemple, des `int` :

```
int tab[];              // tab désigne un tableau de int
```

et

```
int[] tab;              // tab désigne un tableau de int
```

Dans l'un et l'autre cas, la variable `tab` ainsi déclarée est une référence et, pour le moment, elle ne référence rien. Si les expressions précédentes sont la déclaration d'une variable d'instance ou de classe, `tab` aura été initialisée avec la valeur `null`. S'il s'agit de la déclaration d'une variable locale, `tab` est indéterminée et il aurait probablement été plus sage de la compléter par une initialisation explicite, comme :

```
int[] tab = null;       // tab désigne un tableau de int
```

Le tableau lui-même commence effectivement à exister lors de l'appel de l'opérateur `new` :

```
tab = new int[nombreDeComposantes];
```

8. Les variables locales ne sont pas initialisées mais toute utilisation d'une variable locale susceptible de ne pas avoir été initialisée est détectée par le compilateur et signalée comme une erreur.

9. En toute rigueur on peut dire que le passage d'un objet ou d'un tableau comme paramètre d'une méthode se traduit par le *passage par valeur de la référence* à l'objet ou au tableau en question.

où *nombreDeComposantes* est une expression, d'un type entier, qui définit la taille du tableau.

NOTE. On peut joindre dans la même expression la déclaration de la variable et la création du tableau correspondant :

`int[] tab = new int[`*nombreDeComposantes*`];`

cela est plus commode, car il y a moins de choses à écrire, et plus fiable, car on élimine la période pendant laquelle la variable existe sans référencer un tableau. En général c'est faisable, car Java permet d'écrire les déclarations là où on veut.

3.4.2 Accès aux éléments

Une fois le tableau créé, on accède à ses composantes en utilisant la même notation qu'en C :

`tab[`i`]`

où i est une expression de type entier [10]. Pour que l'expression ci-dessus soit légitime il faut et il suffit que $0 \leq i <$ *nombreDeComposantes*. Autrement dit, les éléments du tableau précédent sont

`t[0]`, `t[1]`, ... `t[`*nombreDeComposantes*-1`]`

Le schéma de la figure 2 est la bonne manière de se représenter l'accès `tab[`i`]` :

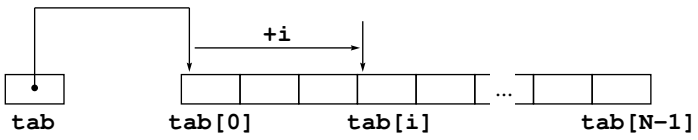


FIGURE 2 – Un tableau de `N` éléments

TAILLE EFFECTIVE D'UN TABLEAU. L'expression *unTableau*`.length` permet de connaître le nombre d'éléments d'un tableau. Cela est bien utile pour écrire des traitements de tableaux dont la taille est inaccessible lors de la compilation. Par exemple, le programme suivant affiche les arguments écrits sur la ligne de commande (à la suite des deux mots `java ArgumentsLigneCommande`) quel que soit leur nombre :

```
class ArgumentsLigneCommande {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

CONTRÔLE DE L'INDICE. Lors de l'accès à un élément d'un tableau, Java contrôle *toujours* la validité de l'indice. Autrement dit, lors de l'évaluation d'une expression comme `tab[i]` les deux erreurs possibles les plus fréquentes sont détectées et déclenchent une exception :

- `NullPointerException`, pour indiquer que `tab = null`, c'est-à-dire que la variable `tab` n'est pas une référence valide (en clair, le plus souvent : la variable `tab` a bien été déclarée, mais le tableau correspondant n'a pas encore été créé),
- `ArrayIndexOutOfBoundsException`, pour indiquer que la condition $0 \leq$ `i` $<$ `tab.length` n'est pas satisfaite.

3.4.3 Tableaux à plusieurs indices

Les tableaux à plusieurs indices sont dynamiques eux aussi. A titre d'exemple, examinons le cas des matrices (tableaux à deux indices). Les déclarations suivantes sont tout à fait équivalentes :

```
int mat[][];        // mat désigne un tableau de int à deux indices
int[] mat[];        //  "       "       "       "       "       "
int[][] mat;        //  "       "       "       "       "       "
```

Ces trois expressions introduisent `mat` comme une variable de type *tableau à deux indices* de `int`. On peut les lire « `mat` est un tableau-à-un-indice de tableaux-à-un-indice de `int` ». Aucun tableau n'existe pour le moment ; selon le contexte, la valeur de mat est soit `null`, soit indéterminée.

La matrice commence à exister lorsqu'on appelle l'opérateur `new`. Par exemple, de la manière suivante :

10. Ne pas oublier qu'une expression de type `char` est toujours spontanément convertible dans le type `int`.


```
mat = new int[NL][NC];
```

où `NL` et `NC` sont des expressions entières définissant le nombre de lignes et de colonnes de la matrice souhaitée. L'instruction précédente a le même effet que le code suivant :

```
mat = new int[NL][];                // un tableau de NL [références de] tableaux
for (int i = 0; i < NL; i++)
    mat[i] = new int[NC];           // un tableau de NC int
```

L'accès aux éléments d'un tableau à deux indices se fait comme en C :

```
mat[i][j]
```

L'évaluation de cette expression déclenchera la vérification successive des conditions :

1° `mat` $\neq$ `null`
2° $0 \leq$ `i` $<$ `mat.length`
3° `mat[i]` $\neq$ `null`
4° $0 \leq$ `j` $<$ `mat[i].length`

Ainsi, un tableau à deux indices dont les éléments sont – par exemple – des `int` est réalisé en Java comme un tableau à un indice dont les éléments sont des [références sur des] tableaux à un indice dont les éléments sont des `int`. La figure 3 représente cette configuration.

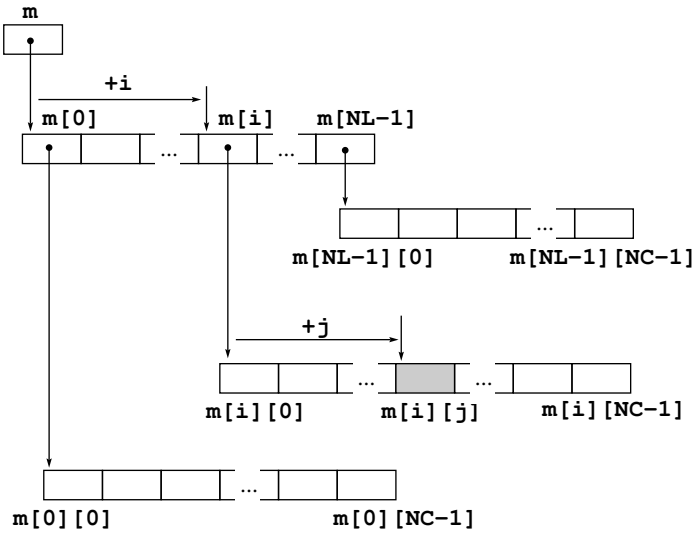


FIGURE 3 – Accès à `m[i][j]`, `m` étant une matrice à `NL` lignes et `NC` colonnes

NOTE. Java permet de manipuler, sous la même syntaxe, des tableaux rectangulaires et des tableaux qui ne le sont pas ; il suffit pour cela de faire soi-même l'initialisation des lignes. Par exemple, le code suivant crée une matrice triangulaire inférieure de dimension `N` dans laquelle chaque ligne a la longueur requise par son rang :

```
float[][] mt = new float[N][];
for (int i = 0; i < N; i++)
    mt[i] = new float[i + 1];
```

3.4.4 Tableaux initialisés et tableaux anonymes.

Deux syntaxes permettent d'indiquer un ensemble de valeurs devant remplir un tableau. Dans l'une comme dans l'autre, le nombre de valeurs fournies détermine la taille du tableau. Dans la première forme, analogue à celle qui existe en C, on initialise le tableau au moment de sa déclaration :

```
int[] tab = { 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 };
```

La deuxième forme permet de créer un tableau garni de valeurs, mais pas forcément à l'endroit de la déclaration du tableau :

```
int[] tab;
...
tab = new int[] { 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 };
```

En fait, cette deuxième syntaxe permet de créer des tableaux garnis *anonymes*, ce qui est bien utile lorsqu'un tableau ne sert qu'une seule fois. Par exemple, si la variable `tab` avait été déclarée et construite dans l'unique but de figurer dans l'appel d'une méthode (cela arrive) comme dans :

```
int tab[] = { 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 };
traiter(tab);
```

alors il aurait été plus simple de ne pas déclarer de variable `tab` :

```
traiter(new int[] { 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 });
```

Cas des matrices. Les mêmes mécanismes permettent de créer des tableaux multidimensionnels garnis, y compris lorsque leurs lignes ne sont pas de même longueur :

```
int[][] mat = { { 11, 12, 13, 14, 15 },
                { 20, 21, 22, 23, 24, 25, 26, 27, 28, 29 },
                { },
                null,
                { 51, 52, 53 } };
```

Notez la différence entre la troisième et la quatrième ligne du tableau précédent : la troisième existe mais est vide (c'est un tableau de 0 éléments), tandis que la quatrième n'existe pas.

Note. Contrairement à ce qui se passe dans d'autres langages, en Java l'initialisation des tableaux se fait pendant l'exécution du programme, non pendant la compilation. Première conséquence : initialiser un tableau par une liste de constantes est plus commode, mais pas notablement plus efficace, que la collection d'affectations qui aurait le même effet. Par exemple, la déclaration

```
int[] tab = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

a le même effet – mais est plus agréable à écrire et à lire – que la séquence

```
int[] tab = new int[10];
tab[0] = 1;
tab[1] = 2;
...
tab[9] = 10;
```

Deuxième conséquence : les expressions avec lesquelles on initialise des tableaux n'ont pas à être nécessairement des constantes. Des expressions quelconques, évaluées pendant l'exécution du programme, peuvent y figurer :

```
int[] tab = { x + 1, 2 * x + 1, 4 * x + 1, (int) Math.atan(y) };
```

3.5 Conversions entre types tableaux ou classes

Attention, section indigeste ! *Pour bien comprendre cette section il faut connaître les notions de classe et d'héritage.*

Le *type statique* d'une expression E est déterminé par les *déclarations des variables* apparaissant dans E, ainsi que par les constantes, opérateurs et méthodes qui forment E. Par exemple, avec la déclaration

```
Object e = "Bonjour";
```

le type statique de `e` est `Object`.

Le *type dynamique* d'une expression E est celui qui résulte des *valeurs effectives des variables* intervenant dans E, ainsi que des constantes, opérateurs et méthodes constituant E. Par exemple, avec la même déclaration ci-dessus, le type dynamique de `e` est `String`.

Dans ces conditions, soit E une expression, T_s son type statique, T_d son type dynamique à un instant donné et T_f un deuxième type. Nous nous intéressons à la conversion de la valeur de E vers le type T_f, lorsque T_s et T_f ne sont pas identiques et que ce ne sont pas deux types primitifs. Voici ce qu'il faut savoir à ce sujet :

1. Aucune conversion n'est possible entre un type primitif et un type tableau ou classe, ou réciproquement. Dans les points suivants on suppose donc que T_s et T_f sont tous les deux des types tableaux ou classes.

2. La conversion de la valeur de E vers le type T_f, lorsqu'elle est légitime, est une conversion *sans travail* : elle se réduit à considérer l'objet désigné par E comme ayant le type T_f, sans que cet objet subisse la moindre modification (ne pas oublier que E ne représente qu'une *référence* sur un objet).
3. Sauf le cas particulier examiné au point 7, la conversion de E vers le type T_f n'est acceptée *à la compilation* que si T_s et T_f sont des classes apparentées, c'est-à-dire des classes dont l'une est sous-classe, directe ou indirecte, de l'autre (autrement dit, T_s et T_f sont sur une même branche de l'arbre d'héritage).
4. Si T_f est une super-classe, directe ou indirecte, de T_s la conversion de E vers le type T_f est correcte et ne requiert aucune indication particulière. On peut appeler cela une *généralisation* du type de E.

   Par exemple, si la classe `Article` est une super-classe de la classe `ArticleSportif`, elle-même super-classe de `Velo`, qui est super-classe de `VTT`, alors l'affectation suivante est correcte :

   ```
   Article unArticle = new Velo( ...caractéristiques d'un vélo... );
   ```

   L'objet `Velo` créé ci-dessus n'est en rien modifié par son affectation à la variable `unArticle` mais, aussi longtemps qu'il est accédé à travers cette variable, il est considéré comme un `Article`, non comme une `Velo` (`Article` est un concept plus général que `Velo`).

5. Si T_f est une sous-classe (directe ou indirecte) de T_s, la conversion de E vers le type T_f est une sorte de *particularisation* du type de E et :
   - n'est acceptée à la compilation que si on utilise explicitement l'opérateur de transtypage. Exemple :

     ```
     Article unArticle;
     Velo unVelo;
     ...
     unVelo = unArticle;                 // ERREUR dès la compilation
     unVelo = (Velo) unArticle;          // Ok, pour la compilation
     ```

   - n'est correcte à l'exécution que si T_d est une sous-classe, directe ou indirecte, de T_f. Exemple :

     ```
     unArticle = new VTT( ... caractéristiques d'un VTT ... );
     ...
     unVelo = (Velo) unArticle;          // Ok, un VTT peut être vu comme une Velo
     ```

   - déclenche lorsqu'elle est incorrecte, à l'exécution, l'exception `ClassCastException`. Exemple :

     ```
     unArticle = new BallonDeBasket( ... caractéristiques d'un ballon ... );
     ...
     unVelo = (Velo) unArticle;          // déclenche ClassCastException (un ballon
                                         // ne peut pas être vu comme un vélo)
     ```

6. Aucune conversion n'est acceptée entre types tableaux différents (mais n'oubliez pas que le nombre d'éléments d'un tableau ne fait pas partie de la définition de son type : pour Java, un tableau de 10 entiers et un tableau de 20 entiers ont exactement le même type).
7. Enfin, les seules conversions correctes entre une classe et un tableau concernent nécessairement la classe `Object`, super-classe de toutes les classes et de tous les tableaux. Exemple :

   ```
   int[] uneTable = new int[nombre];
   ...
   Object unObjet = uneTable;    // Ok (Object super-classe de tous les objets)
   ...
   uneTable = (int[]) unObjet;   // Ok (d'après le type dynamique de unObjet)
   ```

En résumé. L'essentiel à retenir des règles précédentes :
- à la compilation, la conversion d'une expression de type objet ou tableau est acceptée sauf si on peut prédire *avec certitude* qu'elle échouera à l'exécution ; elle requiert parfois l'emploi explicite de l'opérateur de transtypage ;
- à l'exécution, la conversion du type d'un objet est toujours contrôlée par Java ; elle n'est acceptée que si elle correspond à une *généralisation* du type dynamique de l'objet.

3.6 Copie et comparaison des objets

Attention, section indigeste ! On mentionne ici des concepts, concernant notamment les superclasses et les interfaces, qui ne seront expliquées qu'à la section 7.

3.6.1 Copie

Une conséquence du fait que les objets et les tableaux sont toujours manipulés à travers des références est la suivante : l'affectation d'une expression de type objet ou tableau ne fait pas une vraie duplication de la valeur de l'expression, mais uniquement une duplication de la référence (on appelle parfois cela une *copie superficielle*). Par exemple, considérons :

```
Point p = Point(10, 20);
Point q = p;
```

A la suite de l'affectation précédente on peut penser qu'on a dans `q` un duplicata de `p`. Il n'en est rien, il n'y a pas deux objets Point, mais un seul objet Point référencé par deux variables distinctes, comme le montre la figure 4. Dans certaines situations cette manière de copier est suffisante, mais elle présente un inconvénient évident : chaque modification de la valeur de `p` se répercute automatiquement sur celle de `q`.

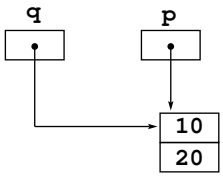


FIGURE 4 – Copie de la référence sans duplication de l'objet (*copie superficielle*)

Pour obtenir la duplication effective d'un objet, il faut appeler sa méthode `clone`. Tous les objets sont censés posséder une telle méthode [11] car elle est déclarée dans la classe `Object`, la super-classe de toutes les classes. Il appartient à chaque classe de redéfinir la méthode `clone`, pour en donner une version adaptée au rôle et aux détails internes de ses instances.

La méthode `clone` rend un résultat de type `Object` [12], il faut donc l'utiliser comme suit (la définition même de `clone` garantit que la conversion du résultat de `p.clone()` vers le type `Point` est légitime) :

```
Point q = (Point) p.clone();
```

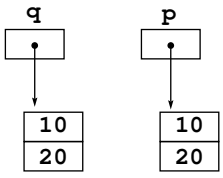


FIGURE 5 – Duplication effective d'un objet (*copie profonde*)

Si `clone` a été correctement définie (dans la classe `Point`) l'expression précédente affecte à `q` une *copie profonde* de l'objet qui est la valeur de `p`, comme montré sur la figure 5.

Il appartient au programmeur d'une classe de définir ce qu'il entend par *copie profonde*, notamment lorsque les objets ont pour membres d'autres objets. Sauf indication contraire on suppose que le clone d'un objet est un deuxième objet n'ayant avec le premier aucun bout de mémoire en commun (comme les points montrés ci-dessus), mais ce n'est pas une obligation. Par exemple, on verra à la section suivante que le clone standard d'un tableau d'objets n'est pas disjoint du tableau original.

11. La méthode `clone` existe dans tous les objets mais, si le programmeur n'a pas pris certaines précautions (consistent soit à définir explicitement la méthode `clone`, soit à déclarer que la classe implémente l'interface `Cloneable`), une exception `CloneNotSupportedException` sera lancée à l'exécution.

12. C'est regrettable mais nécessaire, car la définition initiale de `clone`, dans la classe `Object`, ne peut être déclarée rendant autre chose qu'un `Object` ; ensuite les rédéfinitions de `clone` dans les sous-classes doivent rendre strictement le même résultat, c'est une contrainte du mécanisme de la redéfinition des méthodes.

Cas des tableaux

La méthode `clone` fonctionne en particulier dans le cas des tableaux : si `t` est un tableau, le résultat de `t.clone()` est un tableau de même type et même taille que `t` dont les éléments ont été initialisés *par affectation* de ceux de `t`.

Par exemple, les instructions suivantes créent *deux* tableaux distincts, l'un référencé par les variables `tab1` et `tab2` et l'autre référencé par `tab3` :

```
int[] tab1 = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 };
int[] tab2 = tab1;                    // ceci n'est pas une duplication
int[] tab3 = (int[]) tab1.clone();    // ceci est une duplication effective
```

ATTENTION. Le résultat rendu par `t.clone()` est un tableau nouvellement créé, de même type et même taille que `t`, dont les éléments sont initialisés par affectation, *non par clonage*, de ceux de `t`. Autrement dit, la méthode `clone` des tableaux n'effectue pas une duplication profonde, mais seulement une *duplication à un niveau* : si les éléments du tableau sont des objets [13] le tableau initial et son clone ne sont pas disjoints car ils partagent les mêmes éléments. Il faut se demander si c'est cela qu'on voulait obtenir. Cette question est reprise dans la section suivante.

NOTE JAVA 5. La méthode `clone` étant introduite au niveau de la classe `Object`, elle est déclarée comme rendant un résultat de type `Object` ce qui, dans le cas des tableaux, obligeait – jusqu'à Java 1.4 – à l'utiliser munie d'un changement de type :

```
Point[] t = new Point[n];
...
Point[] q = (Point[]) t.clone();
```

A partir de la version 5 du langage ce n'est plus une obligation. Le compilateur réserve un traitement spécial à l'appel de `clone` *sur un tableau* de manière à rendre correctes des expressions comme :

```
Point[] t = new Point[n];
...
Point[] q = t.clone();
```

3.6.2 Définir la méthode `clone`

La méthode `clone` doit être redéfinie dans chaque classe où cela est utile ; elle doit être `public`. Exemple :

```
class Point {
    int x, y;
    public Point(int a, int b) {
        x = a; y = b;
    }
    public Object clone() {
        return new Point(x, y);
    }
}
```

On peut écrire la méthode `clone` à partir de rien, comme ci-dessus, mais on peut aussi alléger ce travail et utilisant la méthode `Object.clone()` héritée de la classe `Object`. Pour cela, la classe qu'on écrit doit comporter l'énoncé « `implements Cloneable;` » [14] sans quoi l'exception `CloneNotSupportedException` sera lancée à l'exécution. Exemple :

```
class Rectangle implements Cloneable {
    Point coinNO, coinSE;
    public Rectangle(int x1, int y1, int x2, int y2) {
        coinNO = new Point(x1, y1);
        coinSE = new Point(x2, y2);
    }
    public Object clone() throws CloneNotSupportedException {
        return super.clone();   // notre version de clone se réduit à
    }                           // la version héritée de la classe Object
}
```

13. Cas particulier remarquable : les matrices, qui sont des tableaux de tableaux. Puisque les tableaux sont des objets, il en résulte que la méthode `clone` ne duplique pas les coefficients des matrices, même lorsque ceux-ci sont d'un type primitif.

14. L'interface `Cloneable` est entièrement vide, l'énoncé `implements Cloneable` n'est qu'une marque par laquelle le programmeur s'accorde le « permis d'utiliser la méthode héritée `Object.clone()` ».

L'effet de la méthode héritée `Object.clone()` est la création d'un objet de même type que celui qu'on clone, puis l'initialisation de chacune des variables d'instance de l'objet créé par *affectation* de la variable d'instance correspondante de l'objet original. C'est donc une copie « à un niveau », moins superficielle que la copie des références, mais ce n'est pas la vraie copie en profondeur : si des membres sont à leur tour des objets cette sorte de copie ne suffit pas pour faire que l'objet cloné soit entièrement séparé de l'objet original.

Par exemple, la figure 6 montre à quoi ressemblerait le couple formé par un objet de la classe `Rectangle`, définie ci-dessus, et son clone, créés par les instructions suivantes :

```
Rectangle p = new Rectangle(10, 20, 30, 40);
Rectangle q = (Rectangle) p.clone();
```

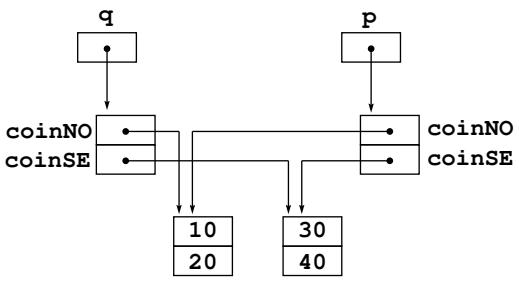


FIGURE 6 – La copie « à un niveau » que fait `Object.clone()`

Pour avoir une duplication complète la méthode `clone` aurait dû être écrite comme ceci :

```
class Rectangle {
    Point coinNO, coinSE;
    ...
    public Object clone() {
        return new Rectangle(coinNO.x, coinNO.y, coinSE.x, coinSE.y);
    }
}
```

ou bien, en supposant que la classe `Rectangle` a un constructeur sans argument :

```
class Rectangle {
    Point coinNO, coinSE;
    ...
    public Object clone() {
        Rectangle r = new Rectangle();
        r.coinNO = (Point) coinNO.clone();
        r.coinSE = (Point) coinSE.clone();
        return r;
    }
}
```

NOTE. Il est regrettable que l'utilisation de la méthode `clone` héritée de la classe `Object` soit alourdie par la nécessité d'attraper ou déclarer `CloneNotSupportedException`, puisque cette exception est *susceptible* d'être lancée, même lorsque, comme ici, il est certain qu'elle ne sera pas lancée (puisque l'énoncé `implements Cloneable` a bien été écrit dans la déclaration de la classe).

Dans l'exemple ci-dessus, cette exception a été *déclarée*. On aurait pu aussi bien l'*attraper* et la faire disparaître (puisque que de toute façon elle ne sera pass lancée) en écrivant `clone` de cette manière :

```
class Rectangle implements Cloneable {
    ...
    public Object clone() {
        try {                          // notre version de clone se réduit à
            return super.clone();      // la version héritée de la classe Object
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }
    ...
}
```


3.6.3 Comparaison

Des remarques analogues aux précédentes peuvent être faites au sujet de l'opérateur de comparaison. Si `a` et `b` sont d'un type classe ou tableau, la condition `a == b` ne traduit pas l'égalité des valeurs de `a` et `b`, mais l'égalité des références. Autrement dit, cette condition est vraie non pas lorsque `a` et `b` sont des objets égaux, mais lorsque `a` et `b` sont *le même* objet.

Dans beaucoup de situations une telle notion d'égalité est trop restrictive. C'est pourquoi tous les objets sont censés posséder la méthode `equals` qui implémente une notion d'égalité plus utile.

```
Point p = new Point(10, 20);
Point q = new Point(10, 20);
...
System.out.println(p == q);           // ceci affiche false
System.out.println(p.equals(q));      // ceci affiche true (si equals a été bien définie)
...
```

La méthode `equals` est définie une première fois au niveau de la classe `Object`, où elle n'est rien de plus que l'égalité des références :

```
class Object {
    ...
    public boolean equals(Object o) {
        return this == o;
    }
}
```

mais chaque classe peut la redéfinir pour en donner une version adaptée au rôle et aux détails internes de ses instances. Pour notre classe `Point` voici une première version, pas très juste :

```
class Point {
    ...
    public boolean equals(Point o) {
        return x == o.x && y == o.y;        // VERSION ERRONÉE !!!
    }
}
```

Bien que répondant en partie aux besoins, cette version de `Point.equals` est erronée car, à cause du type de l'argument, elle *ne constitue pas une redéfinition* de la méthode `Object.equals(Object o)` [15]. Voici une version plus correcte :

```
class Point {
    ...
    public boolean equals(Object o) {
        return o instanceof Point && x == ((Point) o).x && y == ((Point) o).y;
    }
}
```

Maintenant, la méthode ci-dessus est une vraie redéfinition de `Object.equals`, mais elle n'est pas encore parfaite. Il faut savoir, en effet, que x `instanceof` K ne signifie pas forcément que x est instance de K (« x est un K »), mais uniquement que x est instance d'une sous-classe de K (« x est une sorte de K »). Ainsi, si on suppose que `Pixel` est une sous-classe de `Point` (un `Pixel` est un `Point` avec, en plus, l'indication d'une couleur), à la suite de

```
Point cePoint = new Point(11, 22);
Pixel cePixel = new Pixel(11, 22, "rouge");
```

la condition « `cePoint.equals(cePixel)` » sera vraie. Est-ce ce que l'on souhaite ? Probablement non. Voici une nouvelle version de `equals` qui corrige ce problème :

```
class Point {
    ...
    public boolean equals(Object o) {
```

15. Il n'est pas élémentaire de comprendre pourquoi cette version simple de `equals` n'est pas correcte. Il faut imaginer la condition `p.equals(q)` lorsque les valeurs de `p` et `q` sont des objets `Point` alors que la variable `q` est déclarée de type `Object` (une telle situation se produit, par exemple, chaque fois qu'on considère une `Collection` dont les éléments sont des objets `Point`).

Si `q` est déclarée de type `Object`, `p.equals(q)` n'utilisera pas notre méthode `Point.equals(Point p)`, mais `Object.equals(Object o)` qui *n'est pas redéfinie* dans la classe `Point`.

```
            return o != null && o.getClass() == Point.class
                && ((Point) o).x == x && ((Point) o).y == y;
        }
    }
```

Toute surcharge de la méthode `Object.equals(Object o)` doit être une *relation d'équivalence* (c'est-à-dire une relation binaire *réflexive*, *symétrique* et *transitive*) telle que si `x` $\neq$ `null` alors `x.equals(null)` est faux. De plus, bien que ce ne soit pas une obligation, il semble naturel de faire en sorte que si `y = x.clone()` alors `x == y` soit certainement faux et `x.equals(y)` soit certainement vrai.

3.7 Chaînes de caractères

3.7.1 Les classes `String` et `StringBuffer`

Deux classes permettent de représenter les chaînes de caractères :
- les instances de la classe `String` sont *invariables* ; une fois créées, les caractères dont elles sont formées ne peuvent plus être modifiés,
- les instances de la classe `StringBuffer`, au contraire, sont destinées à subir des opérations qui modifient les caractères dont elles sont faites (remplacements, ajouts, suppressions, etc.).

Les détails de l'utilisation de ces deux classes, extrêmement utiles l'une et l'autre, sont décrits dans la documentation de l'API Java. Limitons-nous ici à signaler certains privilèges tout à fait originaux et intéressants de la classe `String` :

1. Il existe une notation spécifique pour les constantes littérales : toute suite de caractères encadrée par des guillemets produit une instance de la classe `String` :

   ```
   String nom = "Gaston Lagaffe";
   ```

2. L'opération de concaténation (mise bout à bout) de deux chaînes se note par l'opérateur [16] + :

   ```
   String formule = "Monsieur " + nom;    // vaut "Monsieur Gaston Lagaffe"
   ```

3. Toutes les classes possèdent la méthode `String toString()`, qui renvoie une expression d'un objet sous forme de chaîne de caractères. La classe `Object` fournit une version minimale de cette méthode, chaque classe peut en donner une définition plus adaptée. Exemple :

   ```
   class Point {
       int x, y;
       ...
       public String toString() {
           return "(" + x + "," + y + ")";
       }
   }
   ```

4. L'opérateur +, lorsqu'un de ses opérandes est une chaîne, provoque la conversion de l'autre opérande vers le type chaîne. Cette conversion est effectuée
   - dans le cas d'un type primitif, par l'appel d'une des méthodes statiques `valueOf` de la classe `String`,
   - dans le cas d'un objet, par l'appel de sa méthode `toString`

   Par exemple, si `p` est un objet `Point` (voir ci-dessus), l'expression

   ```
   "résultat: " + p
   ```

   équivaut à l'expression

   ```
   "résultat: " + p.toString()
   ```

   dont la valeur est, par exemple, la chaîne `"résultat: (10,20)"`.

5. La conversion vers le type chaîne est pratiquée également par certaines méthodes d'intérêt général. Par exemple, les méthodes `print` et `println` de la classe `PrintStream` (la classe de l'objet `System.out`) ont des définitions spécifiques pour les cas où leur argument est d'un type primitif ou `String` et récupèrent tous les autres cas à l'aide de la méthode `toString`. Par exemple, on peut imaginer `print(Object o)` ainsi écrite :

   ```
   public void print(Object o) {
       print(o.toString());        // print(Object) est ramenée à print(String)
   }
   ```

16. Les programmeurs C++ noteront que cet emploi du + est le seul cas de surcharge d'un opérateur en Java.

3.7.2 Copie et comparaison des chaînes

1. CLONE. La classe `String` ne comporte pas de méthode `clone` : puisque les instances de cette classe sont immuables, il n'est jamais nécessaire de les cloner.

2. EQUALS. La classe `String` offre une version de la méthode `equals` dont il est facile d'imaginer ce qu'elle fait : parcourir les deux chaînes en comparant les caractères correspondants. Deux chaînes construites séparément l'une de l'autre doivent être comparées à l'aide de cette méthode. Par exemple, si `reponse` est de type `String`, l'expression

```
reponse == "oui"
```

vaut souvent `false` indépendamment de la valeur de `reponse` et n'a donc aucun intérêt. Il faut écrire à la place

`reponse.equals("oui")` ou bien `"oui".equals(reponse)`.

3. INTERN. La méthode `equals` est utile mais relativement onéreuse, puisqu'elle doit examiner les deux chaînes en comparant les caractère homologues. Les programmes qui comportent des comparaisons fréquentes entre des chaînes (en nombre raisonnable) peuvent tirer avantage de la méthode `intern()` qui renvoie une chaîne ayant les mêmes caractères que la chaîne donnée, mais appartenant à un *ensemble de chaînes sans répétition*, au sein duquel deux chaînes égales sont forcément la même chaîne. Lorsqu'une chaîne appartient à cet ensemble on dira qu'elle a été *internalisée*.

Ainsi, par exemple, étant données deux variables `s1` et `s2` de type `String`, quelles que soient les valeurs des chaînes `t1` et `t2`, à la suite de

```
s1 = t1.intern();
s2 = t2.intern();
```

les conditions « `s1.equals(s2)` » et « `s1 == s2` » sont équivalents (mais l'évaluation de la deuxième est évidemment plus rapide).

NOTE. Les chaînes de caractères constantes écrites dans le programme source sont garanties internalisées, contrairement aux chaînes construites pendant l'exécution (chaque appel de `new` doit produire la création d'un *nouvel* objet). Ainsi, le code suivant affiche certainement « `true false true` » :

```
...
String a = "bonjour";
...
String b = "bonjour";
...
String c = new String("bonjour");
...
String d = c.intern();
...
System.out.println((a == b) + " " + (a == c) + " " + (a == d));
```

4 Expressions et instructions

Cette section ridiculement courte est consacrée aux différences notables existant entre les expressions [17] et instructions de Java et celles de C. Bonne nouvelle : pour l'essentiel, ce sont les mêmes. Les expressions sont formées avec les mêmes opérateurs, les instructions emploient les mêmes mots-clés, les unes et les autres sont soumises aux mêmes règles de syntaxe, renvoient les mêmes valeurs et ont les mêmes effets.

Exceptions : deux mécanismes sont réalisées de manière originale en Java et méritent d'être expliquées, même dans un polycopié succinct comme celui-ci :
- l'instruction de *rupture étiquetée*, expliquée ci-dessous,
- la *boucle for améliorée* expliquée à la section 12.3.2.

4.1 Rupture étiquetée

L'instruction `goto` de certains langages n'existe pas ici. Cependant, afin de bénéficier d'un service important que cette instruction rend parfois, la maîtrise de l'*abandon des boucles imbriquées*, on a ajouté ceci à Java :
- les instructions, et notamment les boucles (`for`, `while`, `do...while`, etc.), peuvent être étiquetées,
- l'instruction `break` peut être suivie d'une étiquette afin de provoquer l'abandon de plusieurs boucles imbriquées au lieu d'une seule.

Une étiquette est un identificateur qu'il n'est pas nécessaire de déclarer ; il suffit de l'écrire, suivi de « : », devant une instruction, pour qu'il soit connu à l'intérieur de celle-ci et inconnu à l'extérieur.

Par exemple, le code – purement démonstratif – suivant affiche les dates comprises entre le 1^{er} janvier 2000 et le 30 décembre 2005, en supposant que tous les mois ont 30 jours. A chaque affichage, un entier pseudo-aléatoire de l'intervalle $[0, 100[$ est tiré : si c'est un multiple de 3, le mois en cours est abandonné (on passe au 1^{er} du mois suivant) ; si c'est un multiple de 13, l'année en cours est abandonnée (on passe au 1^{er} janvier de l'année suivante) ; si le nombre est 0, le travail est abandonné :

```
...
grandeBoucle: for (int a = 2000; a <= 2005; a++)
    moyenneBoucle: for (int m = 1; m <= 12; m++)
        petiteBoucle: for (int j = 1; j <= 30; j++) {
            System.out.println(j + "/" + m + "/" + a);
            int x = (int) (Math.random() * 100);
            if (x == 0)
                break grandeBoucle;
            else if (x % 13 == 0)
                break moyenneBoucle;
            else if (x % 3 == 0)
                break petiteBoucle;
        }
...
```

Ce qui vient d'être dit à propos de l'instruction `break` étiquetée s'applique *mutatis mutandis* à l'instruction `continue` qui peut, elle aussi, être étiquetée. Par exemple, voici une rédaction équivalente du bout de programme précédent :

```
...
grandeBoucle: for (int a = 2000; a <= 2005; a++)
    moyenneBoucle: for (int m = 1; m <= 12; m++)
        for (int j = 1; j <= 30; j++) {
            System.out.println(j + "/" + m + "/" + a);
            int x = (int) (Math.random() * 100);
            if (x == 0)
                break grandeBoucle;
            else if (x % 13 == 0)
                continue grandeBoucle;
            else if (x % 3 == 0)
                continue moyenneBoucle;
        }
...
```

17. Nous voulons parler ici des expressions ne mettant en jeu que des types primitifs

5 Classes, paquets, fichiers et répertoires

L'unité de compilation en Java est la classe. Un fichier source se compose nécessairement de quelques classes complètes, souvent une seule, et le compilateur produit un fichier compilé (fichier d'extension `.class`) pour chaque classe rencontrée dans le fichier source, fût-elle une classe interne ou anonyme.

La syntaxe de la définition des classes est expliquée à partir de la section 6.2. Ici nous nous intéressons au regroupement des classes en paquets, et au rangement des classes et des paquets dans les fichiers et les répertoires.

5.1 Paquets et classes

Voici la quintessence de la notion de paquet (*package*) : *deux classes peuvent avoir le même nom si elles appartiennent à des paquets distincts.*

5.1.1 Les noms des paquets et l'instruction `package`

Qu'on le veuille on non, chaque classe appartient à un paquet. Si on ne s'est occupé de rien, il s'agit du paquet sans nom (*unnamed package*), qui contient toutes les classes pour lesquelles on n'a pas donné un nom de paquet explicite.

On peut changer cela à l'aide de l'instruction *package*, qui doit être la première instruction utile (i.e. autre qu'un commentaire) du fichier source :

`package` *nomDuPaquet*`;`

Toutes les classes définies dans un fichier commençant par la ligne précédente appartiennent *ipso facto* au paquet nommé *nomDuPaquet*. D'autres fichiers peuvent comporter cette même instruction ; les classes qu'ils définissent appartiennent alors à ce même paquet.

Un nom de paquet est un identificateur ou une suite d'identificateurs, chacun séparé du suivant par un point. Trois exemples : `monprojet`, `java.awt.event` et `fr.univ_mrs.dil.henri.outils`.

Les noms commençant par `java.` sont réservés aux paquets contenant les classes de la bibliothèque standard. Seul *Sun Microsystems* a le droit de les utiliser.

Si vous écrivez des programmes d'un intérêt vraiment général, vous devez pouvoir garantir que les noms de vos paquets sont uniques dans le monde entier. Le système proposé par *Sun* pour nommer ces paquets à vocation planétaire consiste à utiliser les éléments de votre nom de domaine Internet, placés à l'envers.

Par exemple, si votre domaine est `dil.univ-mrs.fr` alors vos paquets pourront être nommés comme ceci [18] :

```
package fr.univ_mrs.dil.unProjet.outils;
```

5.1.2 Les noms des classes et l'instruction `import`

Chaque classe possède un *nom court*, qui est un identificateur, et un *nom long* formé en préfixant le nom court par le nom du paquet. Par exemple, `java.awt.List` est le nom long d'une classe du paquet `java.awt` dont le nom court est `List`.

Une classe peut *toujours* être désignée [19] par son nom long. Elle peut en outre être désignée par son nom court :

- depuis [toute méthode de] toute classe du même paquet ;
- depuis [toute méthode de] toute classe écrite dans un fichier qui comporte l'instruction :
  `import` *nomDuPaquet*`.`*nomDeLaClasse*`;`
  ou bien l'instruction (dans cette forme on « importe » d'un coup toutes les classes du paquet indiqué) :
  `import` *nomDuPaquet*`.*;`

NOTE 1. Il ne faut pas donner à la formule précédente plus d'effet qu'elle n'en a : *unPaquet*`.*` signifie les classes du paquet *unPaquet*, cela n'inclut pas les classes des éventuels « sous-paquets » (les paquets dont les noms commencent par *unPaquet*). Ainsi, les deux directives suivantes, d'un usage fréquent, ne sont pas redondantes, la première n'implique pas la deuxième :

```
import java.awt.*;          // les classes du paquet java.awt
import java.awt.event.*;    // les classes du paquet java.awt.event
```

18. Les éléments d'un nom de paquet doivent être des identificateurs. C'est la raison pour laquelle les éventuels tirets - doivent être remplacés, comme ici, par des blancs soulignés _.

19. Qu'une classe puisse ou doive être désignée par tel ou tel nom ne préjuge en rien du fait qu'on ait on non le droit d'accéder à ses membres ; cela dépend du contexte et de règles expliquées à la section 6.4.

NOTE 2. Dans certains cas l'utilisation des noms longs des classes reste nécessaire pour éviter des ambiguïtés. Par exemple, `List` est une classe du paquet `java.awt` et une interface du paquet `java.util`. Dans un fichier où les deux sont employées (ce n'est pas rare), on ne peut pas utiliser le nom court dans les deux cas.

NOTE 3. On entend parfois : « il faut écrire telle instruction `import` pour *avoir le droit* d'utiliser telle classe ». Cela n'est *jamais* vrai : l'instruction `import` ne sert pas à rendre visibles des classes (toutes les classes qui se trouvent dans des fichiers et répertoires que la machine Java peut atteindre sont visibles [20]) ni à vous donner le droit d'accès à des classes (cela se pilote à travers les qualifications `public`, `private`, etc., voir la section 6.4). Le seul droit que l'instruction `import` vous donne est celui d'utiliser un nom court pour nommer une classe que vous auriez pu, sans cela, désigner par son nom long.

NOTE 4. Le paquet `java.lang` est spécial : ses classes peuvent être désignées par leur nom court sans qu'il soit nécessaire d'écrire `import java.lang.*` en tête du fichier. Ce paquet est fait de classes fondamentales, qui font partie de la définition du langage Java lui-même : `Object`, `String`, `Math`, `System`, `Integer`, etc. S'il fallait écrire `import java.lang.*` pour les utiliser, alors il faudrait l'écrire dans *tous* les fichiers.

5.2 Classes, fichiers et répertoires

5.2.1 Classes publiques et non publiques

La déclaration d'une classe peut être précédée ou non du qualifieur `public`. Si ce n'est pas le cas, la classe est dite « non publique » et elle n'est accessible que depuis les [méthodes des] autres classes *du même paquet* [21].

Si la déclaration d'une classe est précédée du qualifieur `public` alors la classe est dite publique ; elle est accessible depuis [les méthodes de] toutes les classes de tous les paquets.

Si un fichier source contient une classe publique, le nom du fichier doit être le nom de cette classe, complété par l'extension « `.java` ». Par exemple, un fichier contenant le texte

```
public class Point {
    ...
    définition de la classe Point
    ...
}
```

doit se nommer `Point.java`. Il en découle qu'un fichier source ne peut pas contenir plus d'une classe publique.

En résumé, mis à part les commentaires, un fichier source est donc composé des éléments suivants, dans l'ordre indiqué :
- éventuellement, une instruction `package` ;
- éventuellement, une ou plusieurs instructions `import` ;
- une ou plusieurs classes, dont au plus une est publique ; elle impose alors son nom au fichier.

5.2.2 Point d'entrée d'un programme.

Une classe est dite exécutable si, et seulement si elle comporte une méthode de signature

```
public static void main(String[] args)
```

(chaque mot de l'expression précédente est imposé strictement, sauf `args`). Le lancement de l'exécution d'une application Java équivaut à l'appel de la méthode `main` d'une classe exécutable ; si on travaille à une console, cela se fait par la commande

`java` *NomDeLaClasse* $argum_1$ ... $argum_k$

L'argument `args` de `main` est un tableau dont les éléments sont les chaînes $argum_1 \ldots argum_k$ qui figurent dans la commande qui a lancé le programme. Par exemple, voici un programme qui affiche n fois un certain mot m, n et m étant donnés lors du lancement. Fichier `Radotage.java` :

20. L'instruction `import` n'est donc pas l'homologue en Java de la directive `#include` du langage C
21. Les classes non publiques représentent donc des services offerts aux autres classes du même paquet, non des fonctionnalités universelles dont n'importe quel utilisateur peut avoir besoin (cela est le rôle des classes publiques).

```
public class Radotage {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        for (int i = 0; i < n; i++)
            System.out.print(args[1] + " ");
    }
}
```

Compilation et exécution de ce programme (dans le cas d'exécution présenté ici, `args[0]` est la chaîne `"5"` et `args[1]` la chaîne `"bonsoir"`) :

```
> javac Radotage.java
> java Radotage 5 bonsoir
bonsoir bonsoir bonsoir bonsoir bonsoir
>
```

5.2.3 Où placer les fichiers des classes ?

Un procédé, qui dépend plus ou moins du système d'exploitation sous-jacent, associe à chaque paquet un répertoire dans lequel doivent se trouver les fichiers compilés (fichiers d'extension `.class`) des classes du paquet ; il consiste à utiliser la structure hiérarchique du système de fichiers pour reproduire la structure hiérarchique des noms des paquets.

Ainsi, par exemple, les fichiers des classes des paquets `java.awt.event` et `monprojet.outils` doivent être placés, respectivement, dans des répertoires nommés, sur Unix, `java/awt/event/` et `monprojet/mesoutils/`. Sur Windows, ces répertoires seraient `java\awt\event\` et `monprojet\mesoutils\`.

Ces chemins sont *relatifs*[22] aux répertoires suivants :
- par défaut, le répertoire courant (nommé généralement « . »),
- chacun des répertoires de la liste définie par la valeur de la variable d'environnement `CLASSPATH`, lorsque cette variable est définie[23] (le cas par défaut n'est alors pas considéré),
- chacun des répertoires de la liste définie par la valeur de l'argument `-cp` ou `-classpath`, si cet argument a été spécifié lors du lancement de la machine java (l'éventuelle valeur de la variable `CLASSPATH` et le cas par défaut sont alors ignorés).

NOTE. Parfois les classes intervenant dans une compilation ou une exécution sont prises dans un fichier "`.jar`". Il faut savoir que, fondamentalement, un tel fichier n'est que la forme « zippée » d'une arborescence de répertoires et les indications précédentes s'appliquent pratiquement telles quelles, le fichier "`.jar`" jouant le rôle de répertoire de base.

Pour obtenir que le compilateur place les fichiers compilés dans un certain répertoires, en le créant s'il n'existe pas, il faut lancer la compilation avec l'option « `-d` *répertoireDeBase* ». Par exemple, si on souhaite que les noms des paquets correspondent à des chemins relatifs au répertoire courant (désigné par « . ») il faut activer le compilateur par la commande :

```
javac -d . MaClasse.java
```

Ainsi, si le fichier `MaClasse.java` contient une classe nommée `MaClasse` et commence par l'instruction `package monPaquet` la commande précédente produira un fichier appelé `MaClasse.class` placé dans un sous-répertoire `monPaquet` du répertoire courant .

Si `MaClasse` est une classe exécutable, on la lancera alors depuis le répertoire courant par l'une ou l'autre des commandes :

`java monPaquet/MaClasse` (sur UNIX) ou `java monPaquet\MaClasse` (sur Windows)

ou

`java monPaquet.MaClasse` (indépendamment du système)

OÙ PLACER LES FICHIERS SOURCES ? Il est conseillé de suivre la même règle pour le placement des fichiers sources (fichiers `.java`). Si on veut que la compilation d'une classe $\mathcal{C}$ produise la compilation des

22. Si r est un répertoire défini par un chemin absolu (i.e. partant de la racine du système de fichiers) A, on dit qu'un chemin B est *relatif à* r pour indiquer que B représente le chemin absolu obtenu en concaténant A et B.

Par exemple, si le chemin `monProjet/mesOutils/` est relatif au répertoire `/home/henri/` alors il définit le chemin absolu `/home/henri/monProjet/mesOutils/`.

23. Dans les versions récentes du *SDK*, sauf besoin spécifique il vaut mieux ne pas jouer avec la variable `CLASSPATH`. S'ils ont été bien installés, le compilateur et la machine Java accèdent normalement à toutes les bibliothèques de classes dont ils ont besoin dans le cadre d'une utilisation normale de Java.

classes dont $\mathcal{C}$ dépend [24], ces dernières doivent se trouver dans les répertoires qui correspondent à leurs paquets. Par exemple, si le fichier `MaClasse.java` commence par l'instruction « `package monPaquet;` » alors il vaut mieux placer ce fichier dans le répertoire `monPaquet`. Sa compilation sera alors provoquée

– soit par la compilation d'une classe qui mentionne la classe `monPaquet.MaClasse`,
– soit par la commande « `javac -d . monPaquet/MaClasse.java` »

5.2.4 Comment distribuer une application Java ?

Dans la section précédente on a expliqué où placer les classes pour que l'exécution d'un programme se passe bien, du moins lorsqu'elle a lieu sur la machine sur laquelle on a fait le développement. Mais que se passe-t-il lorsqu'un programme développé sur une machine doit être exécuté sur une autre ?

Dans le cas général, la compilation d'une application produit de nombreux fichiers classes, éventuellement rangés dans différents répertoires (autant de répertoires que de paquetages en jeu). Il en résulte que, alors que pour d'autres langages *une* application correspond à *un* fichier executable, en Java une application se présente comme une pluralité de fichiers et de répertoires ; une telle organisation rend complexe et risquée la distribution des applications. C'est pourquoi les exécutables Java sont distribués sous la forme d'archives *zip* ou, plutôt, *jar* [25], qu'il faut savoir fabriquer. Expliquons cela sur un exemple :

Imaginons qu'une application se compose des classes `Principale`, `Moteur`, `Fenetres`, etc., le point d'entrée – c'est-à-dire la méthode par laquelle l'exécution doit commencer [26] – étant dans la classe `Principale`. La compilation de ces classes a produit les fichiers `Principale.class`, `Moteur.class`, `Fenetres.class`, etc.

Avec un éditeur de textes quelconque, composez un fichier nommé `MANIFEST.MF` comportant l'unique ligne

```
Main-Class: Principale
```

et tapez la commande :

`jar -cvfm Appli.jar MANIFEST.MF Principale.class Moteur.class Donnees.class` *etc.*

Notez que, si les classes constituant l'application sont les seuls fichiers `.class` du répertoire de travail, vous pouvez taper la commande précédente sous la forme

```
jar -cvfm Appli.jar MANIFEST.MF *.class
```

Cela produit un fichier nommé `Appli.jar` que, si vous êtes curieux, vous pouvez examiner avec n'importe quel outil comprenant le format *zip* : vous y trouverez tous vos fichiers `.class` et un répertoire nommé `META-INF` contenant le fichier `MANIFEST.MF` un peu augmenté :

```
Manifest-Version: 1.0
Created-By: 1.5.0_05 (Sun Microsystems Inc.)
Main-Class: Principale
```

C'est ce fichier `Appli.jar` que vos distribuerez. Quiconque souhaitera exécuter votre application pourra le faire en tapant la commande [27]

```
java -jar Appli.jar
```

ou bien

```
java -classpath Appli.jar Principale
```

Dans ce dernier cas, l'application s'exécuterait correctement même si l'archive ne contenait pas de fichier manifeste, voire si au lieu de `Appli.jar` on avait utilisé une archive `Appli.zip` tout à fait ordinaire.

24. En principe, la compilation d'une classe $\mathcal{C}$ produit la compilation des classes que $\mathcal{C}$ mentionne, pour lesquelles cela est utile, c'est-à-dire celles qui ont été modifiées depuis leur dernière compilation. C'est un mécanisme proche de la commande *make* de Unix mais l'expérience montre qu'il y a parfois des ratés (des classes qui le devraient ne sont pas recompilées).

25. En réalité une archive *jar* est la même chose qu'une archive *zip*, sauf qu'une archive *jar* est censée contenir – sans que la chose soit rédhibitoire – un fichier *manifest*.

26. Le point d'entrée est la première instruction d'une méthode qui *doit avoir* le prototype `public static void main(String[] args)`, mais que rien n'empêche que des méthodes ayant ce prototype existent aussi dans plusieurs autres classes de l'application.

27. Sur un système Windows correctement configuré, on pourra même exécuter l'application en double-cliquant sur l'icône du fichier *jar*. Dans ce cas, il y a intérêt à ce qu'il s'agisse d'une application qui crée sa propre interface-utilisateur (par exemple, graphique) car autrement les entrées-sorties seront perdues, faute de console.

6 Les objets

6.1 ♡ Introduction : les langages orientés objets

Java est un langage orienté objets. Pour fixer les idées, voici une présentation très sommaire d'une partie du jargon et des concepts de base de cette famille de langages [28].

Objet. Les entités élémentaires qui composent les programmes sont les *objets*. Un objet est constitué par l'association d'une certaine quantité de mémoire, organisée en champs qu'on appelle des *variables d'instance*, et d'un ensemble d'opérations (fonctions, procédures...) qu'on appelle des *méthodes*.

Contrôle de l'accessibilité. Les variables et les méthodes peuvent être qualifiées *publiques*, c'est-à-dire accessibles depuis n'importe quel point du programme, ou *privées*, c'est-à-dire accessibles uniquement depuis les méthodes de l'objet en question.

Encapsulation. Un objet est vu par le reste du programme comme une entité opaque : on ne peut modifier son état, c'est-à-dire les valeurs de ses variables, autrement qu'en faisant faire la modification par une de ses méthodes publiques [29]. L'ensemble des méthodes publiques d'un objet s'appelle son *interface* [30].

L'importante propriété suivante est ainsi garantie : un objet ne dépend pas des implémentations (c'est-à-dire les détails internes) d'autres objets, mais uniquement de leurs interfaces.

Message. Un message est une requête qu'on soumet à un objet pour qu'il effectue une de ses opérations. En pratique, cela se traduit par l'appel d'une méthode de l'objet. Le message spécifie *quelle* opération doit être effectuée, mais non *comment* elle doit l'être : il incombe à l'objet récepteur du message de connaître les modalités effectives que prend pour lui l'exécution de l'opération en question.

Appellation « orienté objets ». Du point de vue du concepteur, toute procédure ou fonction appartient à un objet ; elle spécifie comment un objet donné réagit à un message donné. Dans la programmation orientée objets les entités les plus extérieures sont les objets, les actions sont définies à l'intérieur des objets, comme attributs de ces derniers.

Cela s'oppose à la programmation traditionnelle, ou « orientée actions », où les entités les plus extérieures sont les actions (procédures, fonctions, routines, etc.) les objets se définissant de manière subordonnée aux actions (ils en sont les données, les résultats, etc.).

Classe. Une classe est la description des caractéristiques communes à tous les objets qui représentent la même sorte de chose. Vue de l'extérieur, elle définit donc leur interface commune, seul aspect public des objets. Vue de l'intérieur, c'est-à-dire comme son concepteur la voit, la classe définit la structure de la mémoire privée de ces objets (les variables d'instance) et les réponses aux messages de leurs interfaces (les corps des méthodes).

Instance [31]. Les objets individuels décrits par une classe sont ses instances. *Chaque objet est instance d'une classe.* Il n'y a pas d'inconvénient à identifier la notion de classe avec celle de type, les instances s'identifient alors avec les valeurs du type : la classe est le type de ses instances.

L'instanciation est l'opération par laquelle les objets sont créés. C'est une notion dynamique (même dans les langages à types statiques) : tout objet est créé (a) entièrement garni et (b) à un moment bien déterminé de l'exécution d'un programme.

6.2 Classes, variables et méthodes

6.2.1 Déclaration des classes et des objets, instanciation des classes

Fondamentalement, une classe est constituée par un ensemble de *membres*, qui représentent
- des données, on les appelle alors *variables*, *champs*, ou encore *données membres*,
- des traitements, on les appelle alors *méthodes* ou *fonctions membres*.

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

28. Nous traitons ici de l'aspect *encapsulation* des langages orientés objets. On introduira plus loin les concepts liés à l'*héritage*.
29. En Java, la possibilité d'avoir des variables d'instance publiques tempère cette affirmation.
30. Attention, en Java le mot *interface* a un sens technique bien précis, voir la section 7.7.3
31. Le mot *instance* est très souvent employé en programmation orientée objets, avec le sens expliqué ici (qui se résume tout entier dans l'incantation « *chaque objet est une instance de sa classe* »). Si ce mot vous pose un problème, ne cherchez pas de l'aide parmi les sens qu'il a dans la langue française, aucun ne convient. Il a été emprunté par les pères fondateurs de la *POO* à la langue anglaise, où entre autres choses, *instance* signifie *cas*, comme dans *cas particulier*.

visibilité `class` *identificateur* `{`
déclaration d'un membre
...
déclaration d'un membre
`}`

ou *visibilité* est soit rien, soit le mot `public` (la classe est alors dite publique, voyez la section 5.2.1).

Les membres des classes se déclarent comme les variables et les fonctions en C, sauf qu'on peut les préfixer par certains qualifieurs, `private`, `public`, etc., expliqués plus loin. Exemple :

```
class Point {
    int x, y;       // coordonnées cartésiennes
    void montre() {
        System.out.println("(" + x + "," + y + ")");
    }
    double distance(Point p) {
        int dx = x - p.x;
        int dy = y - p.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

Une classe est un type de données : à la suite d'une déclaration comme la précédente, on pourra déclarer des variables de type `Point`, c'est-à-dire des variables dont les valeurs sont des *instances de la classe* `Point` et ont la structure définie par cette dernière. Exemple :

```
Point p;            // déclaration d'une variable Point
```

Notez bien que la déclaration précédente introduit une variable `p` qui pourra référencer des instances de la classe `Point`, mais qui, pour le moment, ne référence rien. Selon son contexte, une déclaration comme la précédente donnera à `p` :
- soit la valeur `null`, dans le cas de la déclaration d'une variable membre,
- soit une valeur conventionnelle « *variable non initialisée* », dans le cas de la déclaration d'une variable locale d'une méthode.

On obtient que `p` soit une référence valide sur un objet soit en lui affectant une instance existante, soit en lui affectant une instance nouvellement créée, ce qui s'écrit :

```
p = new Point();    // création d'une nouvelle instance de la classe Point
```

La justification des parenthèses ci-dessus, et d'autres informations importantes sur l'instanciation et l'initialisation des objets, sont données à la section 6.5.1.

NOTE. Étant donnée une classe *C*, on dit indifféremment *instance de la classe C* ou *objet C*.

6.2.2 Instances et membres d'instance

La déclaration donnée en exemple à la section précédente introduit une classe `Point`, comportant pour le moment deux *variables d'instance*, `x` et `y`, et deux *méthodes d'instance*, `montre` et `distance`. En disant que ces éléments sont des variables et des méthodes *d'instance* de la classe `Point` on veut dire que « chaque instance en a sa propre version », c'est-à-dire :
- pour les variables, qu'il en existe un jeu différent dans chaque instance, disjoint de ceux des autres instances : chaque objet `Point` a sa propre valeur de `x` et sa propre valeur de `y`,
- pour les méthodes, qu'elles sont liées à un objet particulier : un appel de la méthode `montre` n'a de sens que s'il est adressé à un objet `Point` (celui qu'il s'agit de montrer).

Ainsi, sauf dans quelques situations particulières que nous expliciterons plus loin, pour accéder à un membre d'instance il faut indiquer l'instance concernée. Cela se fait par la même notation que l'accès aux champs des structures en C :

`p.x` : un accès à la variable d'instance `x` de l'objet `p`
`p.montre()` : un appel de la méthode `montre` sur l'objet `p`

L'expression `p.montre()` est la version Java de la notion, fondamentale en programmation orientée objets, « envoyer le *message* `montre` à l'objet `p` ». A priori, le même message pourrait être envoyé à des objets différents. Il incombe à chaque destinataire, ici `p`, de connaître les détails précis que prend pour lui la réaction au message `montre`.

Accès à ses propres membres. Ainsi, une méthode d'instance est nécessairement appelée à travers un objet, le destinataire du message que la méthode représente. Dans le corps de la méthode, cet objet est implicitement référencé par toutes les expressions qui mentionnent des membres d'instance sans les associer à un objet explicite.

Par exemple, dans la méthode `distance` de la classe `Point` :

```
...
double distance(Point p) {
    int dx = x - p.x;
    int dy = y - p.y;
    return Math.sqrt(dx * dx + dy * dy);
}
...
```

les variables `x` et `y` qui apparaissent seules désignent les coordonnées de l'objet à travers lequel on aura appelé la méthode `distance`. D'autre part, les expressions `p.x` et `p.y` désignent les coordonnées de l'objet qui aura été mis comme argument de cet appel.

Ainsi, à l'occasion d'un appel de la forme (`a` et `b` sont des variables de type `Point`) :

```
a.distance(b);
```

le corps de la méthode `distance` équivaut à la suite d'instructions :

```
{ int dx = a.x - b.x;
  int dy = a.y - b.y;
  return Math.sqrt(dx * dx + dy * dy); }
```

Accès à soi-même. A l'intérieur d'une méthode d'instance, l'identificateur `this` désigne l'objet à travers lequel la méthode a été appelée. Par exemple, la méthode `distance` précédente peut aussi s'écrire, *mais ici cela n'a guère d'avantages*[32], comme ceci :

```
...
double distance(Point p) {
    int dx = this.x - p.x;
    int dy = this.y - p.y;
    return Math.sqrt(dx * dx + dy * dy);
}
...
```

Pour voir un exemple où l'emploi de `this` est nécessaire, imaginez qu'on nous oblige à écrire la méthode d'instance `distance` en la ramenant à un appel d'une méthode de classe :

```
static int distanceSymetrique(Point a, Point b);
```

c'est-à-dire une méthode qui, comme une fonction en C, n'est pas attachée à une instance particulière et calcule la distance entre les deux points donnés comme arguments. Cela donnerait :

```
...
int distance(Point p) {
    return distanceSymetrique(this, p);
}
...
```

6.2.3 Membres de classe (membres statiques)

Les membres *de classe* sont ceux dont la déclaration est qualifiée par le mot réservé `static`[33] :

```
class Point {
    int x, y;                                          // variables d'instance
    static int nombre;                                 // variable de classe
    void montre() {                                    // méthode d'instance
        System.out.println("(" + x + "," + y + ")");
    }
    static int distanceSymetrique(Point p, Point q) {  // méthode de classe
```

32. Notez cependant que certains stylistes de la programmation en Java recommandent d'employer systématiquement la notation `this.`*variabledInstance* ou `this.`*methodedInstance*`(...)` pour les *membres d'instance*, et proscrivent cette notation pour les *membres statiques* (cf. section 6.2.3) ; cela permet à un lecteur du programme de distinguer plus facilement ces deux sortes de membres lors de leur utilisation.

33. Par la suite on dira indifféremment *membre de classe* ou *membre statique* et *membre d'instance* ou *membre non statique*.

```
        return Math.abs(p.x - q.x) + Math.abs(p.y - q.y);
    }
}
```

Contrairement aux membres d'instance, les membres de classe ne sont pas attachés aux instances. Ce qui signifie :

- pour une variable, qu'*il n'y en a qu'une*. Alors que de chaque variable d'instance il y a un exemplaire pour chaque objet effectivement créé par l'exécution du programme, de chaque *variable de classe* il est créé – la première fois que la classe intervient dans l'exécution du programme – un unique exemplaire auquel toutes les instances accèdent,
- pour une méthode, qu'on peut l'appeler sans devoir passer par un objet. L'appel d'une *méthode de classe* ne se fait pas en faisant référence explicitement ou implicitement à une instance particulière (autre que celles qui figurent éventuellement à titre de paramètres) ; un tel appel ne peut dont pas être vu comme un message adressé à un objet.

  Conséquence importante : à l'intérieur d'une méthode de classe, l'objet `this` (objet implicite à travers lequel une méthode d'instance est nécessairement appelée) n'est pas défini.

Ainsi l'expression suivante, par exemple écrite dans une méthode étrangère à la classe `Point` :

```
Point.nombre
```

est un accès bien écrit à la variable de classe `nombre` et, `p` et `q` étant des objets `Point`, l'expression

```
Point.distanceSymetrique(p, q)
```

est un appel correct de la méthode de classe `distanceSymetrique`.

Remarque 1. On notera que les deux expressions précédentes peuvent s'écrire aussi

```
p.nombre
p.distanceSymetrique(p, q)
```

(le préfixe « `p.` » ne sert ici qu'à indiquer qu'il s'agit des membres `nombre` et `distance` de la classe `Point` ; ces expressions ne font aucun usage implicite de l'objet `p`) mais ces notations sont trompeuses – et déconseillées – car elles donnent aux accès à des membres de classe l'apparence d'accès à des membres d'instance.

Remarque 2. On aura compris que les méthodes de classe sont ce qui se rapproche le plus, en Java, des fonctions « autonomes », comme celles du langage C. Par exemple, la méthode `Math.sqrt`, mentionnée plus haut, n'est rien d'autre que la fonction racine carrée qui existe dans toutes les bibliothèques mathématiques. L'obligation d'enfermer de telles fonctions dans des classes n'est pas une contrainte mais une richesse, puisque cela permet d'avoir, dans des classes différentes, des fonctions qui ont le même nom.

De même, la notion de variable de classe est voisine de celle de variable globale du langage C. De telles variables existent dès que la classe est chargée, c'est-à-dire la première fois qu'elle intervient dans le programme, jusqu'à la terminaison de celui-ci.

Remarque 3. Pour bien mettre en évidence les différences entre les membres de classe et les membres d'instance, examinons une erreur qu'on peut faire quand on débute :

```
public class Essai {
    void test() {
        ...
    }
    public static void main(String[] args) {
        test();   // *** ERREUR ***
    }
}
```

Sur l'appel de `test` on obtient une erreur « *Cannot make a static reference to the non-static method test* ». Cela signifie que la méthode `test` ne peut être appelée qu'en se référant, explicitement ou implicitement, à un objet, alors que `main` étant `static`, elle n'est liée à aucun objet (en fait, dans ce programme on ne crée aucune instance de la classe `Essai`).

Deux solutions. La première consiste à créer une instance, éventuellement un peu artificielle, permettant d'appeler `test` en passant par un objet :


```
public class Essai {
    void test() {
        ...
    }
    public static void main(String[] args) {
        Essai unEssai = new Essai();
        unEssai.test();   // OK
    }
}
```

La deuxième, plus simple, consiste à rendre `test` statique :

```
public class Essai {
    static void test() {
        ...
    }
    public static void main(String[] args) {
        test();   // OK
    }
}
```

6.3 Surcharge des noms

En Java les noms peuvent être *surchargés* : un même nom peut désigner plusieurs entités distinctes, à la condition que lors de chaque utilisation du nom le compilateur puisse déterminer sans erreur quelle est l'entité désignée.

En pratique, cela signifie qu'on peut donner le même nom
- à des entités dont les rôles syntaxiques sont différents, comme une classe et un membre, ou bien une variable et une méthode,
- à des membres de classes distinctes,
- à des méthodes de la même classe ayant des signatures[34] différentes.

Par exemple, dans la classe `Math` on trouve, parmi d'autres, les méthodes `int abs(int x)` (valeur absolue d'un entier), `float abs(float x)` (valeur absolue d'un flottant), `double abs(double x)`, etc. Lorsque le compilateur rencontre une expression comme

```
u = Math.abs(v);
```

la méthode effectivement appelée est celle qui correspond au type effectif de `v`. L'intérêt de ce mécanisme est facile à voir : il dispense le programmeur d'avoir à connaître toute une série de noms différents (un pour chaque type d'argument) pour nommer les diverses implémentations d'un même concept, la valeur absolue d'un nombre.

On notera que le type du résultat d'une méthode ne fait pas partie de sa signature : deux méthodes d'une même classe, ayant les mêmes types d'arguments mais différant par le type du résultat qu'elles rendent ne peuvent pas avoir le même nom.

Remarque. Il ne faut pas confondre la *surcharge* avec la *redéfinition* des méthodes, que nous expliquerons à la section 7.3 :

1. La surcharge des méthodes consiste dans le fait que deux méthodes, souvent membres de la même classe, ont le même nom mais des signatures différentes. Sur un appel de méthode faisant apparaître ce nom, la détermination de la méthode qu'il faut effectivement appeler est faite d'après les arguments de l'appel. C'est un problème statique, c'est-à-dire résolu *pendant la compilation* du programme.
2. La redéfinition des méthodes consiste dans le fait que deux méthodes ayant le même nom et *la même signature* sont définies dans deux classes dont l'une est sous-classe, directe ou indirecte, de l'autre. Nous verrons que, sur l'appel d'une méthode ayant ce nom, la détermination de la méthode qu'il faut effectivement appeler est faite d'après le type *effectif* de l'objet destinataire du message. C'est un mécanisme dynamique, qui intervient *au moment de l'exécution* du programme.

34. La signature d'une méthode est la liste des types des ses arguments.

6.4 Contrôle de l'accessibilité

6.4.1 Membres privés, protégés, publics

La déclaration d'un membre d'une classe (variable ou méthode, d'instance ou de classe) peut être qualifiée par un des mots réservés `private`, `protected` ou `public`, ou bien ne comporter aucun de ces qualifieurs.

Cela fonctionne de la manière suivante :

- *membres privés* : un membre d'une classe $\mathcal{C}$ dont la déclaration commence par le qualifieur `private` n'est accessible que depuis [les méthodes de] la classe $\mathcal{C}$; c'est le contrôle d'accès le plus restrictif,
- un membre d'une classe $\mathcal{C}$ dont la déclaration ne commence pas par un des mots `private`, `protected` ou `public` est dit avoir l'accessibilité *par défaut* ; il n'est accessible que depuis [les méthodes de] la classe $\mathcal{C}$ et les autres classes du paquet auquel $\mathcal{C}$ appartient,
- *membres protégés* : un membre d'une classe $\mathcal{C}$ dont la déclaration commence par le qualifieur `protected` n'est accessible que depuis [les méthodes de] la classe $\mathcal{C}$, les autres classes du paquet auquel $\mathcal{C}$ appartient et les sous-classes, directes ou indirectes, de $\mathcal{C}$,
- *membres publics* : un membre de la classe $\mathcal{C}$ dont la déclaration commence par le qualifieur `public` est accessible partout où $\mathcal{C}$ est accessible ; c'est le contrôle d'accès le plus permissif.

Le concept de membre protégé est lié à la notion d'héritage. Pour cette raison, ces membres sont expliqués plus en détail à la section 7.5 (page 46).

6.4.2 ♡ L'encapsulation

Quand on débute dans la programmation orientée objets on peut trouver que le contrôle de l'accessibilité est une chinoiserie peu utile. Il faut comprendre, au contraire, que cette question est un élément fondamental de la méthodologie.

En effet, nous cherchons à écrire des programmes les plus modulaires possibles, or l'encapsulation est l'expression ultime de la modularité : faire en sorte que chaque objet du système que nous développons ne puisse pas dépendre de détails internes des autres objets. Ou, dit autrement, garantir que chaque objet s'appuie sur les spécifications des autres, non sur leurs implémentations.

Il en découle une règle de programmation simple : tout membre qui peut être rendu privé doit l'être [35].

Pour illustrer cette question, considérons notre classe `Point`, et supposons que nous souhaitions permettre à ses utilisateurs d'accéder à l'abscisse et à l'ordonnée des points. Première solution, rendre les membres `x` et `y` publics :

```
class Point {
    public int x, y;
    ...
}
```

Exemple d'utilisation : si `p` est un objet `Point`, pour transformer un point en son symétrique par rapport à l'axe des abscisses il suffira d'écrire :

```
p.y = - p.y;
```

Voici une autre définition possible de la classe `Point` :

```
class Point {
    private int x, y;
    public int getX() { return x; }
    public int getY() { return y; }
    public void setPos(int a, int b) {
        « validation de a et b »
        x = a; y = b;
    }
    ...
}
```

avec cette définition, pour transformer un point `p` en son symétrique par rapport à l'axe des abscisses il faut écrire :

```
p.setPos(p.getX(), - p.getY());
```

35. Certains langages vont plus loin, et font que toutes les variables sont d'office privées. Java permet de déclarer des variables d'instance et de classe publiques ; notre propos ici est de déconseiller de telles pratiques.

Malgré les apparences, la deuxième manière est la meilleure, car elle respecte le principe d'encapsulation, ce que ne fait pas la première. D'abord, cette deuxième définition permet de garantir la cohérence interne des objets créés par les utilisateurs de la classe `Point`, y compris les plus étourdis, puisque le seul moyen de modifier les valeurs des variables d'instance `x` et `y` est d'appeler la méthode `setPos`, dans laquelle le concepteur de la classe a écrit le code qui valide ces valeurs (partie « *validation de a et b* »).

Ensuite, la deuxième définition garantit qu'aucun développement utilisant les objets `Point` ne fera intervenir des détails de l'implémentation de cette classe, puisque les variables d'instance `x` et `y` sont privées, c'est-à-dire inaccessibles. Par exemple, si un jour il devenait nécessaire de représenter les points par leurs coordonnées polaires au lieu des coordonnées cartésiennes, cela pourrait être fait sans rendre invalide aucun programme utilisant la classe `Point` puisque, vu de l'extérieur, le comportement de la classe `Point` serait maintenu :

```
class Point {
    private double rho, theta;
    public int getX() { return (int) (rho * Math.cos(theta)); }
    public int getY() { return (int) (rho * Math.sin(theta)); }
    public void setPos(int a, int b) {
        rho = Math.sqrt(a * a + b * b);
        theta = Math.atan2(b, a);
    }
    ...
}
```

6.5 Initialisation des objets

En Java la création d'un objet est toujours dynamique : elle consiste en un appel de l'opérateur `new`, et peut se produire à n'importe quel endroit d'un programme. Il faut savoir que les variables d'instance des objets ne restent jamais indéterminées, elles ont des valeurs initiales précises.

Un élément de la méthodologie objet, très important pour la qualité des programmes, est que le concepteur d'une classe maîtrise complètement les actions à faire et les valeurs initiales à donner lors de la création des instances ; il peut ainsi garantir que les objets sont, dès leur création, *toujours cohérents*.

Notons pour commencer que, s'il n'y a pas d'autres indications, Java donne une valeur initiale à chaque membre d'une classe. Pour les membres de types primitifs, la valeur initiale correspondant à chaque type est celle qu'indique le tableau 1 (page 11). Pour les membres d'autres types, c'est-à-dire les tableaux et les objets, la valeur initiale est `null`. Ainsi, sans que le programmeur n'ait à prendre de précaution particulière, les instances d'une classe déclarée comme ceci

```
class Point {
    int x, y;
    ...
}
```

sont des points ayant $(0, 0)$ pour coordonnées.

Lorsque l'initialisation souhaitée est plus complexe que l'initialisation par défaut, mais assez simple pour s'écrire comme un ensemble d'affectations indépendantes de l'instance particulière créée, on peut employer une notation analogue à celle de C. Par exemple, voici une classe `Point` dont les instances sont des points aléatoirement disposés dans le rectangle $[0, 600[\times[0, 400[$:

```
class Point {
    int x = (int) (Math.random() * 600);
    int y = (int) (Math.random() * 400);
    ...
}
```

Lorsque l'initialisation requise est plus complexe qu'une suite d'affectations, ou lorsqu'elle dépend d'éléments liés au contexte dans lequel l'objet est créée, la notation précédente est insuffisante. Il faut alors définir un ou plusieurs *constructeurs*, comme expliqué à la section suivante.

6.5.1 Constructeurs

Un constructeur d'une classe est une méthode qui a le même nom que la classe et pas de type du résultat [36]. Exemple :

```
class Point {
    private int x, y;

    public Point(int a, int b) {
        validation de a et b
        x = a;
        y = b;
    }
    ...
}
```

Un constructeur est toujours appelé de la même manière : lors de la création d'un objet, c'est-à-dire comme opérande de l'opérateur `new` :

```
Point p = new Point(200, 150);
```

Dans un constructeur on ne doit pas trouver d'instruction de la forme « `return` *expression* `;` » : un constructeur ne rend rien (c'est l'opérateur `new` qui rend quelque chose, à savoir une référence sur l'objet nouveau), son rôle est d'initialiser les variables d'instance de l'objet en cours de création, et plus généralement d'effectuer toutes les opérations requises par la création de l'objet.

Comme les autres méthodes, les constructeurs peuvent être qualifiés `public`, `protected` ou `private`, ou avoir l'accessibilité par défaut. Une situation utile et fréquente est celle montrée ci-dessus : un ou plusieurs constructeurs publics servant surtout à initialiser un ensemble de variables d'instance privées.

La surcharge des méthodes vaut pour les constructeurs : une classe peut avoir plusieurs constructeurs, qui devront alors différer par leurs signatures :

```
class Point {
    private int x, y;

    public Point(int a, int b) {
        validation de a et b
        x = a;
        y = b;
    }
    public Point(int a) {
        validation de a
        x = a;
        y = 0;
    }
    ...
}
```

A l'intérieur d'un constructeur l'identificateur `this` *utilisé comme un nom de variable* désigne (comme dans les autres méthodes d'instance) l'objet en cours de construction. Cela permet parfois de lever certaines ambiguïtés ; par exemple, le constructeur à deux arguments donné plus haut peut s'écrire aussi [37] :

```
class Point {
    private int x, y;

    public Point(int x, int y) {
        validation de x et y
        this.x = x;
        this.y = y;
    }
    ...
}
```

36. Nous l'avons déjà dit, le type du résultat d'une méthode ne fait pas partie de sa signature. Néanmoins, le fait d'avoir ou non un type du résultat est un élément de la signature. Il en découle que, par exemple, dans une certaine classe `Point` peuvent coexister un constructeur `Point()` et une méthode `void Point()` distincts.

37. Parfois les noms des arguments des méthodes ne sont pas indifférents, ne serait-ce qu'en vue de la documentation (ne pas oublier que l'outil `javadoc` fabrique la documentation à partir du code source).

A l'intérieur d'un constructeur, l'identificateur `this`, *utilisé comme un nom de méthode*, désigne un autre constructeur de la même classe (celui qui correspond aux arguments de l'appel). Par exemple, voici une autre manière d'écrire le second constructeur de la classe `Point` montrée plus haut :

```
class Point {
    private int x, y;

    public Point(int x, int y) {
        validation de x et y
        this.x = x;
        this.y = y;
        ...
    }
    public Point(int x) {
        this(x, 0);
        ...
    }
    ...
}
```

NOTE 1. Lorsqu'un tel appel de `this(...)` apparaît, il doit être la *première instruction* du constructeur.

NOTE 2. L'expression qui crée un objet, « `new UneClasse()` », se présente toujours comme l'appel d'un constructeur, même lorsque le programmeur n'a écrit aucun constructeur pour la classe en question. Tout se passe comme si Java avait dans ce cas ajouté à la classe un constructeur implicite, réduit à ceci :

```
UneClasse() {
    super();
}
```

(la signification de l'expression `super()` est explique à la section 7.4)

6.5.2 Membres constants (`final`)

La déclaration d'un membre d'une classe peut commencer par le qualifieur `final`. Nous verrons à la section 7.6.3 ce que cela veut dire pour une méthode ; pour une variable, cela signifie qu'une fois initialisée, elle ne pourra plus changer de valeur ; en particulier, toute apparition de cette variable à gauche d'une affectation sera refusée par le compilateur. Une telle variable est donc plutôt une constante.

Par exemple, voici comment déclarer une classe dont les instances sont des points immuables :

```
class Point {
    final int x, y;

    Point(int a, int b) {
        validation de a et b
        x = a;
        y = b;
    }
    ...
}
```

N.B. Les affectations de `x` et `y` qui apparaissent dans le constructeur sont acceptées par le compilateur, car il les reconnaît comme des initialisations ; toute autre affectation de ces variables sera refusée.

Le programmeur a donc deux manières d'assurer que des variables d'instance ne seront jamais modifiées une fois les instances crées : qualifier ces variables `final` ou bien les qualifier `private` et ne pas définir de méthode qui permettrait de les modifier. La première manière est préférable, car

- la qualification `private` n'empêche pas qu'une variable soit modifiée depuis une méthode de la même classe,
- la qualification `final` informe le compilateur du caractère constant de la variable en question, celui-ci peut donc effectuer les optimisations que ce caractère constant permet.

CONSTANTES DE CLASSE. Les variables de classe peuvent elles aussi être qualifiées `final` ; elles sont alors très proches des « pseudo-constantes » qu'on définit en C par la directive `#define`. La coutume est de nommer ces variables par des identificateurs tout en majuscules :

```
public class Point {
    public static final int XMAX = 600;
    public static final int YMAX = 400;

    private int x, y;

    public Point(int x, int y) throws Exception {
        if (x < 0 || x >= XMAX || y < 0 || y > YMAX)
            throw new Exception("Coordonnées invalides pour un Point");
        this.x = x;
        this.y = y;
    }
    ...
}
```

6.5.3 Blocs d'initialisation statiques

Les constructeurs des classes sont appelés lors de la création des objets, ce qui est tout à fait adapté à l'initialisation des variables d'instance. Mais comment obtenir l'initialisation d'une variable de classe, lorsque cette initialisation est plus complexe qu'une simple affectation ?

Un *bloc d'initialisation statique* est une suite d'instructions, encadrée par une paire d'accolades, précédée du mot `static`. Ces blocs (et les autres affectations servant à initialiser des variables de classe) sont exécutés, dans l'ordre où ils sont écrits, lors du chargement de la classe, c'est-à-dire avant toute création d'une instance et avant tout accès à une variable ou à une méthode de classe. Exemple :

```
public class Point {
    public static int xMax;
    public static int yMax;

    static {
        Dimension taillEcran = Toolkit.getDefaultToolkit().getScreenSize();
        xMax = taillEcran.width;
        yMax = taillEcran.height;
    }
    ...
}
```

6.5.4 Destruction des objets

Dans beaucoup de langages, comme C et C++, l'opération de création des objets (`malloc`, `new`, etc.) est couplée à une opération symétrique de libération de la mémoire (`free`, `delete`, etc.) que le programmeur doit explicitement appeler lorsqu'un objet cesse d'être utile. Dans ces langages, l'oubli de la restitution de la mémoire allouée est une cause d'erreurs sournoises dans les programmes. Or, lorsque les objets sont complexes, la libération complète et correcte de la mémoire qu'ils occupent, effectuée par des méthodes appelées *destructeurs*, est un problème souvent difficile.

Voici une bonne nouvelle : en Java il n'y a pas d'opération de libération de la mémoire ; en particulier, on n'a pas besoin de munir les classes de destructeurs qui seraient le pendant des constructeurs[38]. Quand un objet cesse d'être utile on l'oublie, tout simplement. Exemple :

```
Point p = new Point(a, b);
...
p = new Point(u, v);      ici, le point P(a,b) a été « oublié »
...
p = null;                 maintenant, le point P(u,v) a été oublié également
...
```

Si un programme peut tourner longtemps sans jamais s'occuper de la destruction des objets devenus caducs c'est que la machine Java intègre un mécanisme, appelé *garbage collector*, qui récupère la mémoire

38. Attention, ne prenez pas pour un destructeur la méthode `finalize`, une notion difficile à cerner et à utiliser que nous n'expliquerons pas dans ce cours.

inutilisée. A cet effet Java maintient dans chaque objet le décompte des références que l'objet supporte, et récupère l'espace occupé par tout objet dont le nombre de références devient nul [39].

Ce mécanisme, qui implique le parcours de structures éventuellement très complexes, se met en marche automatiquement lorsque la mémoire libre vient à manquer, et aussi en tache de fond lorsque la machine est oisive ou effectue des opérations lentes, comme des accès réseau ou des lectures au clavier.

6.6 Classes internes et anonymes

Cette section pointue peut être ignorée en première lecture.

6.6.1 Classes internes

Si la seule utilité d'une classe $\mathcal{I}$ est de satisfaire les besoins d'une autre classe $\mathcal{E}$ alors on peut définir $\mathcal{I}$ à l'intérieur de $\mathcal{E}$. On dit que $\mathcal{I}$ est une *classe interne*, et que $\mathcal{E}$ est la classe *englobante* de $\mathcal{I}$. Cela a deux conséquences majeures :

- une conséquence statique, facile à comprendre : à l'intérieur de la classe englobante le nom de la classe interne peut être employé tel quel, mais à l'extérieur de la classe englobante le nom de la classe interne n'est utilisable que s'il est préfixé par le nom de la classe englobante ou par celui d'une instance de cette classe ; il en résulte une diminution, toujours bienvenue, du risque de collisions de noms,
- une conséquence dynamique, bien plus subtile : si elle n'est pas déclarée `static`, la classe interne est liée à une instance particulière de la classe englobante, à laquelle elle fait implicitement référence.

Exemple. La classe `Automobile` est destinée à représenter les divers modèles au catalogue d'un constructeur. Un même modèle est fabriqué avec différents moteurs. Il a été décidé que la classe `Moteur` n'a d'intérêt qu'à l'intérieur de la classe `Automobile` [40] :

```
class Automobile {
    String modele;
    Moteur moteur;

    Automobile(String m, int c) {
        modele = m;
        moteur = new Moteur(c);
    }

    class Moteur {
        int cylindree;

        Moteur(int c) {
            cylindree = c;
        }

        public String toString() {
            return "Moteur de " + cylindree + " cc du " + modele;
        }
    }

    public static void main(String[] args) {
        Automobile auto = new Automobile("coupé", 2400);
        System.out.println(auto.moteur);
    }
}
```

Regardez bien la méthode `toString` ci-dessus : les deux variables `cylindree` et `modele` qui y apparaissent ont des statuts différents, puisque `cylindree` appartient à une instance de `Moteur`, tandis que `modele` est liée à une instance d'`Automobile`.

39. On peut expliquer le fonctionnement du *garbage collector* de la manière suivante : un objet est *vivant* s'il est (a) la valeur d'une variable de classe d'une classe chargée, (b) la valeur d'une variable d'instance d'un objet *vivant* ou (c) la valeur d'une variable locale d'une méthode active.

Le travail du *garbage collector* se compose, au moins logiquement, de trois phases : (1) parcourir tous les objets vivants et « marquer » les cellules mémoire qu'ils occupent, (2) noter comme étant réutilisables toutes les cellules non marquées pendant la phase précédente et (3) de temps en temps, réarranger la mémoire pour regrouper les cellules réutilisables, de manière à lutter contre l'émiettement de l'espace disponible.

40. Il y là un choix de conception très discutable (même s'il paraît normal que le moteur soit à l'intérieur de l'automobile !) car cela interdit que deux modèles d'automobile distincts aient le même moteur.

Puisque la classe `Moteur` toute entière est un membre non statique de la classe `Automobile`, la première ne peut être accédée qu'à travers une instance particulière de la seconde. Ainsi, par exemple, le constructeur de `Moteur` est une méthode d'instance de `Automobile`.

Il en résulte que dans les méthodes d'instance de `Moteur` deux variables `this` sont définies : celle qui identifie un objet `Moteur` particulier et celle qui identifie l'objet `Automobile` dans lequel cet objet `Moteur` a été créé. On peut dire que ce deuxième `this` se comporte comme une variable de classe de la classe `Moteur`, à la condition de comprendre qu'il y en aura un différent pour chaque instance de la classe `Automobile`.

A l'intérieur d'une instance de `Moteur`, ces deux variables peuvent être atteintes par les appellations respectives `this` et `Automobile.this`.

6.6.2 Classes anonymes

Les classes anonymes sont des classes internes créées à la volée au moment de leur instanciation, comme les tableaux anonymes (cf. section 3.4.4). Cela est particulièrement utile lorsqu'il faut définir une sous-classe d'une classe donnée, ou une classe implémentant une interface donnée, alors qu'on n'a besoin que d'une seule instance de la nouvelle classe.

Par exemple, l'instruction suivante crée dans la foulée une classe anonyme, sous-classe de la classe `Point`, dans laquelle la méthode `toString` est redéfinie, et une instance (qui restera unique) de la classe anonyme ainsi définie :

```
Point o = new Point(0, 0) {
    public String toString() {      // l'objet o est un Point ordinaire, sauf
        return "[Origine]";         // pour ce qui concerne la méthode toString
    }
};
```

Si, après la compilation de la classe `Point` ci-dessus, on examine la liste des fichiers produits par le compilateur on trouvera les deux fichiers `Point.class` et `Point$1.class`. Le second traduit la création d'une classe anonyme à l'occasion de l'instanciation précédente.

Autre exemple, tiré de la gestion des événements dans `AWT` : pour attacher un auditeur d'événements « action » à un composant produisant de tels événements, comme un bouton ou un menu, il faut exécuter l'instruction :

composant`.addActionListener(`*auditeur*`);`

où *auditeur* est un objet d'une classe implémentant l'interface `ActionListener`. Or cette interface se compose d'une unique méthode, `actionPerformed`. On s'en sort donc à moindres frais avec une classe anonyme :

```
composant.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        traitement de l'événement
    }
});
```


7 Héritage

7.1 ♡ Introduction : raffiner, abstraire

Pour approcher la notion d'héritage, examinons deux sortes de besoins qu'il satisfait, légèrement différents. Le premier, que nous appellerons *raffinement*, correspond à la situation suivante : on dispose d'une classe A achevée, opérationnelle, définissant des objets satisfaisants, et il nous faut définir une classe B qui en est une *extension* : les objets B ont tout ce qu'ont les objets A plus quelques variables et méthodes ajoutées et/ou quelques méthodes « améliorées » (on dira *redéfinies*).

Imaginez, par exemple, qu'on dispose d'une classe `RectangleGeometrique` pour représenter des rectangles en tant qu'entités mathématiques, avec des variables d'instance (`x`, `y`, `largeur`, `hauteur`, etc.) et des méthodes (`translation`, `rotation`, etc.) et qu'il nous faille une classe `RectangleGraphique` dont les instances sont des entités rectangulaires capables de s'afficher sur un écran. Elles comportent donc tout ce qui fait un `RectangleGeometrique` et, en plus, quelques variables (`couleurFond`, `couleurBord`, `épaisseurBord`, etc.) et méthodes supplémentaires (`affichage`, `transformationCouleur`, etc.).

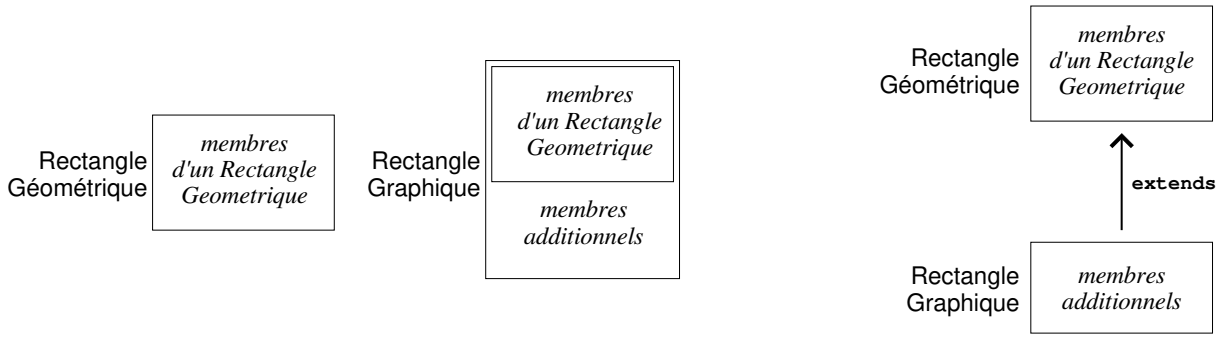


FIGURE 7 – Raffiner

Si nous ne disposions que des outils expliqués dans les chapitres précédents, nous devrions définir la classe `RectangleGraphique` en y dupliquant (cf. figure 7, à gauche) les informations qui se trouvent dans la définition de `RectangleGeometrique`.

Le mécanisme de l'héritage permet d'éviter cette duplication, en nous laissant déclarer dès le commencement que la classe `RectangleGraphique` *étend* la classe `RectangleGeometrique` et en possède donc tous les attributs (cf. figure 7, à droite). De cette manière, il ne nous reste à définir que les membres spécifiques que la deuxième classe (que nous appellerons la *sous-classe*) ajoute à la première (la *super-classe*).

Une deuxième sorte de besoins auxquels l'héritage s'attaque sont ceux qui se manifestent lorsqu'on a à développer plusieurs classes possédant un ensemble de membres communs. Sans l'héritage, ces derniers devraient être déclarés en autant d'exemplaires qu'il y a de classes qui en ont besoin. L'héritage va nous permettre de les déclarer une seule fois, dans une classe distincte, inventée pour l'occasion, dont la seule justification sera de contenir ces membres communs à plusieurs autres.

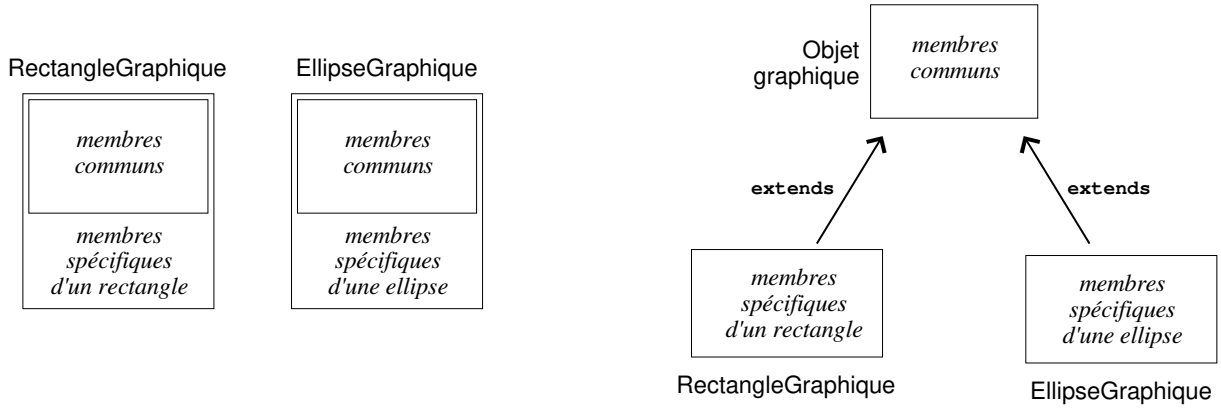


FIGURE 8 – Abstraire

Imaginons par exemple une application manipulant plusieurs sortes d'objets susceptibles de s'afficher sur un écran : des rectangles, des ellipses, des polygones, etc. Il faudra définir plusieurs classes distinctes, comportant des éléments spécifiques – on ne dessine pas un rectangle comme on dessine une ellipse – mais aussi des éléments communs, aussi bien des variables (la couleur du fond, la couleur et l'épaisseur du bord, etc.) que des méthodes (positionnement, translation, etc.)

Voyez la figure 8 (droite) : l'héritage permet de définir des classes `RectangleGraphique`, `EllipseGraphique`, etc., dans lesquelles seuls les éléments spécifiques sont spécifiés, car elles sont déclarées comme des extensions d'une classe `ObjetGraphique` où sont définis les éléments communs à toutes ces autres classes.

Il y a une différence très précise entre cette situation et la précédente : la classe `ObjetGraphique` *ne doit pas avoir des instances*. Un objet qui serait un `ObjetGraphique` sans être quelque chose de plus concret (un `RectangleGraphique`, une `EllipseGraphique`, etc.) n'aurait pas de sens. Pour exprimer cette propriété on dit que `ObjetGraphique` est une *classe abstraite*. Elle est un moyen commode et puissant pour désigner collectivement les diverses sortes d'objets destinés à l'affichage, mais pour la création d'une tel objet il faut toujours utiliser une classe concrète.

REMARQUE POINTUE. Voyant les deux exemples précédents on peut se demander s'il est possible de définir la classe `RectangleGraphique` comme sous-classe en même temps de `RectangleGeometrique` et d'`ObjetGraphique`. La réponse est *non*, en Java l'héritage est *simple* : pour des raisons aussi bien d'efficacité que de simplicité conceptuelle, une classe ne peut être un raffinement que d'une seule autre.

Cependant nous introduirons, le moment venu, la notion d'*interface* (une interface est une classe « très » abstraite) et on verra qu'une classe peut être sous-classe de plusieurs interfaces, la relation d'héritage se nommant alors *implémentation*. Ainsi, si notre classe `ObjetGraphique` peut être écrite sous-forme d'interface, on pourra finalement exprimer en Java que la classe `RectangleGraphique` est une extension de la classe `RectangleGeometrique` et une implémentation de l'interface `ObjetGraphique` ; cela nous apportera une grande partie des avantages qu'on aurait pu espérer de l'héritage multiple, s'il avait été possible.

7.2 Sous-classes et super-classes

7.2.1 Définition de sous-classe

L'héritage est le mécanisme qui consiste à définir une classe nouvelle, dite *classe dérivée* ou *sous-classe*, par extension d'une classe existante, appelée sa *classe de base* ou *super-classe*. La sous-classe possède d'office tous les membres de la super-classe, auxquels elle ajoute ses membres spécifiques.

L'héritage est indiqué par l'expression « `extends` *SuperClasse* » écrite au début de la définition d'une classe. Par exemple, définissons la classe `Pixel` comme sous-classe de la classe `Point` qui nous a déjà servi d'exemple. Un pixel est un point, étendu par l'ajout d'une variable d'instance qui représente une couleur (`Color` est une classe définie dans le paquet `java.awt`) :

```
class Point {
    int x, y;
    ...
}

class Pixel extends Point {
    Color couleur;
    ...
}
```

Avec la définition précédente, un `Pixel` se compose de trois variables d'instance : `x` et `y`, héritées de la classe `Point`, et `couleur`, une variable d'instance spécifique. Ces trois variables sont membres de plein droit de la classe `Pixel` et, si `pix` est un objet de cette classe, on peut écrire – sous réserve qu'on ait le droit d'accès – `pix.x` et `pix.y` exactement comme on peut écrire `pix.couleur`.

NOTE. En toute rigueur, si la définition d'une classe $\mathcal{D}$ commence par

```
class 𝒟 extends ℬ
    ...
```

on devrait dire que $\mathcal{D}$ est une sous-classe *directe* de B ou que $\mathcal{B}$ est une super-classe *directe* de $\mathcal{D}$.

Par ailleurs, on dit que $\mathcal{D}$ est une *sous-classe* de $\mathcal{B}$ si $\mathcal{D}$ est sous-classe directe de $\mathcal{B}$ ou d'une sous-classe de $\mathcal{B}$. Dans ce cas, on dit aussi que $\mathcal{B}$ est *super-classe* de $\mathcal{D}$. Une super-classe [resp. une sous-classe] qui n'est pas directe est dite indirecte.

Cela étant, nous dirons sous-classe [resp. super-classe] pour sous-classe directe [resp. super-classe directe], sauf dans les (rares) situations où cela créerait une confusion.

7.2.2 ♡ L'arbre de toutes les classes

En Java l'héritage est simple : chaque classe possède au plus une super-classe directe. En fait, il y a une classe particulière, nommée `Object`, qui n'a pas de super-classe, et toutes les autres classes ont exactement une super-classe directe. Par conséquent, l'ensemble de toutes les classes, muni de la relation « est sous-classe directe de », constitue une unique arborescence dont la racine est la classe `Object`.

Ainsi, les variables et méthodes qui composent un objet ne sont pas toutes définies dans la classe de celui-ci : il y en a une partie dans cette classe, une partie dans sa super-classe, une autre partie dans la super-classe de la super-classe, etc. Cela introduit la question de la *recherche des méthodes* : un message ayant été envoyé à un objet, la réponse pertinente doit être recherchée dans la classe de celui-ci ou, en cas d'échec, dans la super-classe de celle-là, puis dans la super-classe de la super-classe, et ainsi de suite, en remontant le long de la branche de l'arbre d'héritage, jusqu'à trouver une classe dans laquelle on a prévu de répondre au message en question.

Le procédé de recherche des méthodes est un problème important, dans tous les langages qui pratiquent l'héritage [41]. Il peut être traité à la compilation, on parle alors de *liaison statique*, ou bien à l'exécution, on dit qu'on pratique alors la *liaison dynamique*. La liaison statique est plus efficace, puisque les incertitudes sont levées avant que l'exécution ne commence, mais la liaison dynamique est beaucoup plus souple et puissante. Comme la suite le montrera, Java met en œuvre un savant mélange de liaison statique et dynamique.

7.3 Redéfinition des méthodes

7.3.1 Surcharge et masquage

Que se passe-t-il lorsque des membres spécifiques de la sous-classe ont le même nom que des membres hérités ? Deux cas :

1. Si les entités qui ont le même nom ont des rôles ou des signatures différentes, alors c'est un cas de *surcharge* et il est traité comme s'il s'agissait de deux membres de la même classe. Par exemple, rien ne s'oppose à ce que cohabitent sans conflit dans la sous-classe une variable d'instance propre `f` et une méthode héritée `f`, ou deux méthodes de signatures différentes dont une est héritée et l'autre non.
2. S'il s'agit au contraire de deux entités ayant le même rôle et, dans le cas des méthodes, la même signature alors il y a *masquage* : dans la sous-classe, le membre spécifique masque le membre hérité.

Il y a plusieurs manières d'accéder à un membre masqué. Imaginons par exemple qu'on a introduit une mesure de qualité dans les points, et aussi une notion de qualité dans les pixels, ces deux notions n'ayant pas de rapport entre elles [42] :

```
class Point {
    int x, y;
    int qualite;
    ...
}

class Pixel extends Point {
    Color couleur;
    int qualite;
    ...
}
```

Dans ces conditions, *à l'intérieur d'une méthode de la classe* `Pixel`, les expressions `qualite` et `this.qualite` désignent le membre `qualite` spécifique de la classe `Pixel`. Mais les expressions

```
super.qualite
```

et

```
((Point) this).qualite
```

désignent toutes deux le membre `qualite` hérité de la classe `Point`.

41. Notez que ce problème est partiellement simplifié et optimisé en Java par le fait qu'on n'y pratique que l'héritage simple. Dans les langages à héritage multiple, où une classe peut avoir plusieurs super-classes directes, le graphe d'héritage n'est plus un arbre. Dans ces langages, la recherche des méthodes devient une opération litigieuse et complexe.

42. Si la qualité d'un pixel (en tant que pixel) n'a pas de rapport avec la qualité d'un point, c'est une maladresse que de donner le même nom à ces variables d'instance ; il vaudrait certainement mieux que le membre `Pixel.qualite` ait un autre nom.

7.3.2 Redéfinition des méthodes

Si la définition dans une sous-classe d'une variable ayant le même nom qu'une variable de la super-classe apparaît souvent comme une maladresse de conception, la définition d'une méthode ayant le même nom et la même signature qu'une méthode de la super-classe est au contraire une technique justifiée et très utile.

En effet, la sous-classe étant une *extension*, c'est-à-dire une « amélioration », de la super-classe, les objets de la sous-classe doivent répondre à tous les messages auxquels répondent les objets de la super-classe, mais il est normal qu'il y répondent de manière « améliorée » : beaucoup de méthodes de la sous-classe font les mêmes choses que des méthodes correspondantes de la super-classe, mais elles le font « mieux », c'est-à-dire en prenant en compte ce que les objets de la sous-classe ont de plus que ceux de la super-classe.

Exemple, courant mais fondamental : la méthode `toString`[43] :

```
class Point {
    int x, y;
    ...
    public String toString() {
        return "(" + x + "," + y + ")";
    }
}
class Pixel extends Point {
    Color couleur;
    ...
    public String toString() {
        return "[(" + x + "," + y + ");" + couleur + "]";
    }
}
```

Il est facile de comprendre pourquoi la réponse à un appel comme `unPixel.toString()` peut être vue comme une « amélioration » de la réponse à `unPoint.toString()` : si `unPoint` représente un point et `unPixel` un pixel, les expressions

```
System.out.println(unPoint);
System.out.println(unPixel);
```

affichent, par exemple

```
(10,20)
[(30,40);java.awt.Color.red]
```

NOTE. Écrite comme elle l'est, la méthode `Pixel.toString` précédente a deux défauts :
- le même code (le traitement de x et y) est écrit dans `Point.toString` et dans `Pixel.toString`, ce qui constitue une redondance, toujours regrettable,
- plus grave, la classe `Pixel` s'appuie ainsi sur des détails *internes* de la classe `Point`.

Une manière bien préférable d'écrire les méthodes redéfinies de la sous-classe consiste à laisser les méthodes de la super-classe se charger du traitement de la partie héritée :

```
class Pixel extends Point {
    Color couleur;
    ...
    public String toString() {
        return "[" + super.toString() + ";" + couleur + "]";
    }
}
```

Contraintes de la redéfinition des méthodes

1. On ne peut pas profiter de la redéfinition d'une méthode pour en restreindre l'accessibilité : la redéfinition d'une méthode doit être au moins « aussi publique » que la définition originale. En particulier, la redéfinition d'une méthode publique[44] doit être qualifiée `public`.

2. La signature de la redéfinition d'une méthode doit être *identique* à celle de la méthode héritée. La *signature* d'une méthode se compose de trois éléments :

43. La méthode `toString` appartient à tous les objets, puisqu'elle est définie dans la classe `Object`.

44. On notera que cela concerne en particulier la redéfinition – obligatoire, dans ce cas – des méthodes des interfaces (cf. section 7.7.3) qui, par définition, sont toutes implicitement publiques.

– le nom de la méthode,
– la suite des types de ses arguments,
– le fait que la méthode soit ordinaire (d'instance) ou `static`.

Par exemple, si la définition d'une méthode commence par la ligne

```
public double moyenne(double[] tableau, int nombre, String titre) {
```

alors sa signature est le triplet :

`< moyenne, (double[], int, java.lang.String),` *non statique* `>`

3. Il faut faire très attention à la contrainte évoquée au point précédent, car les différences de signature *ne sont pas en tant que telles signalées par le compilateur.*

En effet, imaginons que la méthode ci-dessus soit définie dans une classe A, et que dans une sous-classe B de A on écrive une méthode (que l'on pense être une redéfinition de la précédente) commençant par

```
public double moyenne(double[] tableau, short nombre, String titre) {
```

Les signatures de ces deux méthodes n'étant pas identiques, la deuxième n'est pas une redéfinition de la première. Hélas pour le programmeur, cela ne constitue pas une erreur que le compilateur pourrait diagnostiquer, mais un simple cas de surcharge : la classe B possède deux méthodes nommées `moyenne` : la première prend un `int` pour second argument tandis que la deuxième prend un `short` [45].

4. On notera bien que le type du résultat renvoyé par une méthode ne fait pas partie de la signature de celle-ci. Dans ces conditions :
– une définition de méthode est une redéfinition si et seulement si la signature est identique à celle d'une méthode héritée, indépendamment du type du résultat,
– si une définition est une redéfinition, alors le type du résultat de la redéfinition *doit* être identique au type du résultat de la méthode héritée.

Note Java 5. La pratique montre que la contrainte exprimée ci-dessus est trop forte. Dans la mesure où on peut voir la redéfinition d'une méthode comme un raffinement de celle-ci, on souhaiterait parfois que la méthode raffinée puisse rendre un résultat qui est lui-même un raffinement du résultat renvoyé par la méthode originale.

C'est pourquoi en Java 5 la règle précédente est devenue :
– une définition de méthode est une redéfinition si et seulement si la signature est identique à celle d'une méthode héritée, indépendamment du type du résultat,
– si une définition est une redéfinition, alors le type du résultat de la redéfinition doit être soit identique, *soit une sous-classe* du type du résultat de la méthode héritée.

7.4 Héritage et constructeurs

L'initialisation d'un objet d'une sous-classe implique son initialisation en tant qu'objet de la super-classe. Autrement dit, tout constructeur de la sous-classe commence nécessairement par l'appel d'un constructeur de la super-classe.

Si le programmeur n'a rien prévu, ce sera le constructeur sans arguments de la super-classe, *qui doit donc exister*. Mais on peut expliciter l'appel d'un constructeur de la super-classe par l'instruction :

`super(`*arguments*`);`

Si elle est présente, cette instruction doit être la première du constructeur de la sous-classe. Par exemple, si la classe `Point` n'a qu'un constructeur, à deux arguments :

```
class Point {
    int x, y;
    Point(int x, int y) {
        validation de x et y
        this.x = x;
        this.y = y;
    }
    ...
}
```

alors le programmeur est obligé d'écrire un constructeur pour la classe `Pixel`, comme ceci :

45. La différence entre un `int` et un `short` étant minime (la valeur 10 est-elle de type `int` ou de type `short` ?), on montre là une situation qui, si elle était voulue, serait certainement une maladresse source de confusion et d'erreurs ultérieures.

```
class Pixel extends Point {
    Color couleur;
    Pixel(int x, int y, Color couleur) {
        super(x, y);
        this.couleur = couleur;
    }
    ...
}
```

7.5 Membres protégés

Comme il a été dit (cf. § 6.4) un membre d'une classe C dont la déclaration commence par le qualifieur `protected` est accessible depuis les méthodes de la classe C, celles des autres classes du paquet auquel C appartient, *ainsi que celles des sous-classes, directes ou indirectes, de C.*

En réalité la règle qui gouverne l'accès aux membres protégés est plus complexe que cela. Considérons un accès tel que :

unObjet`.`*unMembre*
ou
unObjet`.`*unMembre*`(`*arguments*`)`

dans lequel *unMembre* (une variable dans le premier cas, une méthode dans le second) est un membre protégé d'une certaine classe C. Pour être légitime, cette expression

- doit être écrite dans une méthode de C ou d'une classe du même paquet que C – en cela les membres protégés sont traités comme les membres ayant l'accessibilité par défaut –, ou bien
- doit être écrite dans une méthode d'une classe S qui est sous-classe de C, mais alors *unObjet* doit être instance de S ou d'une sous-classe de S (en pratique, *unObjet* sera souvent `this`).

Il est difficile de produire des applications simples et probantes illustrant l'emploi de tels membres [46]. Pour comprendre l'intérêt d'une *méthode* protégée il faut imaginer un traitement qui doit être défini dans une classe A alors qu'il ne peut être appelé que depuis les instances d'une sous-classe de A.

C'est le cas, par exemple, du constructeur d'une classe qui ne doit pas avoir d'instances [47], mais uniquement des sous-classes qui, elles, auront des instances. Voici une version très naïve de quelques classes pour gérer le stock d'une entreprise de location de choses. Il existe une classe `Article`, la plus générale, dont toutes les sortes de choses en location sont sous-classes :

```
package generalites;

public class Article {
    private long reference;
    private double prixParJour;

    protected Article(long reference, double prixParJour) {
        this.reference = reference;
        this.prixParJour = prixParJour;
    }
}
```

Il existe également un certain nombre de sous-classes de la précédente, représentant des catégories précises d'objets à louer, comme les véhicules :

46. Certains théoriciens de la programmation orientée objets estiment même que le concept de membre protégé est injustifié et contreproductif.
47. Plus loin, de telles classes seront appelées *classes abstraites*, voir § 7.7.2.

```
package applications;

public class Vehicule extends generalites.Article {
    private String immatriculation;
    private String carburant;

    public Vehicule(long reference, double prixParJour,
            String immatriculation, String carburant) {
        super(reference, prixParJour);
        this.immatriculation = immatriculation;
        this.carburant = carburant;
    }
}
```

Tous les objets loués sont manipulés dans le programme à travers des variables déclarées de type `Article` :

```
Article unArticle;
```

Il ne faut pas que la valeur d'une telle variable puisse être une instance directe de la classe `Article`, car un objet qui serait un `Article` sans être quelque chose de plus précis (un véhicule, un outil, un meuble, etc.) n'a pas de sens. Le constructeur de la classe `Article` doit donc être rendu moins accessible que `public` afin de garantir que cette classe ne sera jamais directement instanciée :

```
unArticle = new Article(111222333, 120.0);   // *** ERREUR ***
```

Or, en même temps, ce constructeur de `Article` doit être rendu plus accessible que `private` afin qu'il puisse être appelé (à travers l'expression « `super(reference, prixParJour);` ») depuis le constructeur d'une sous-classe telle que `Vehicule` :

```
unArticle = new Vehicule(111222333, 120.0, "1234 ABC 13", "G-O");   // OK
```

NOTE 1. Si les classes de notre exemple avaient été dans le même paquetage, la qualification `protected` n'aurait rien ajouté à la qualification par défaut. Comme on le voit, l'intérêt de la qualification protected est assez limité, surtout en Java.

NOTE 2. L'exemple ci-dessus tire son intérêt de la présence d'une classe qui ne doit pas avoir d'instances. De telles classes sont dites *abstraites* ; nous rencontrerons plus loin (cf. § 7.7.2) de bien meilleures manières de les prendre en considération.

7.6 ♡ Le polymorphisme

Le polymorphisme est un concept particulièrement riche présent dans beaucoup de langages orientés objets. L'aspect que nous en étudions ici tourne autour des notions de généralisation et particularisation.

La *généralisation* est la possibilité de tenir un objet pour plus général qu'il n'est, le temps d'un traitement. Par exemple, pour appliquer une transformation géométrique à un `Pixel` il suffit de savoir qu'il est une sorte de `Point`.

Par *particularisation* nous entendons la possibilité de retrouver le type particulier d'un objet alors que ce dernier est accédé à travers une variable d'un type plus général. Par exemple, retrouver le fait que la valeur d'une variable de type `Point` est en réalité un `Pixel`.

7.6.1 Généralisation et particularisation

GÉNÉRALISATION (conversion sous-classe → super-classe). Si $\mathcal{B}$ est une super-classe d'une classe $\mathcal{D}$, la conversion d'un objet $\mathcal{D}$ vers le type $\mathcal{B}$ est une opération naturelle et justifiée. En effet, à cause de l'héritage, toutes les variables et méthodes de la classe $\mathcal{B}$ sont dans la classe $\mathcal{D}$; autrement dit, tout ce qu'un $\mathcal{B}$ sait faire, un $\mathcal{D}$ sait le faire aussi bien (et peut-être mieux, à cause de la redéfinition possible des méthodes héritées). Par conséquent, en toute circonstance, *là où un $\mathcal{B}$ est attendu on peut mettre un $\mathcal{D}$.*

De telles conversions sont implicites et toujours acceptée par le compilateur. Nous les rencontrerons dans de très nombreuses situations. Par exemple, avec nos classes `Point` et `Pixel` :

```
class Point {
    int x, y;
    ...
    int distance(Point p) {
        return Math.abs(x - p.x) + Math.abs(y - p.y);
    }
```

```
    ...
}

class Pixel extends Point {
    Color couleur;
    ...
}
```

si les variables `unPoint` et `unPixel` ont les types que leurs noms suggèrent, l'expression

```
r = unPoint.distance(unPixel);
```

illustre la généralisation : lors de l'appel de la méthode `distance`, la valeur de la variable `unPixel` est affectée à l'argument `p` (de type `Point`) ; ainsi, durant le calcul de sa distance au point donné, le pixel est vu comme un `Point`, ce qui est suffisant car, dans le calcul d'une distance, seuls interviennent les membres que le pixel hérite de la classe `Point`.

Notez que c'est le même genre de considérations qui donnent un sens à l'expression :

```
r = unPixel.distance(unPoint);
```

Particularisation (conversion super-classe → sous-classe). Une conséquence du fait que les objets sont désignés par référence (cf. section 3.3) est qu'ils ne sont pas altéré par leur généralisation : lorsqu'un objet d'une classe $\mathcal{D}$ est accédé à travers une variable d'un type plus général $\mathcal{B}$ on ne peut pas atteindre les membres de la classe $\mathcal{D}$ qui ne sont pas dans $\mathcal{B}$, mais ces membres ne cessent pas pour autant d'exister dans l'objet en question.

Cela donne un sens à l'opération réciproque de la généralisation : donner à un objet $\mathcal{O}$ un type $\mathcal{T}$ plus particulier que celui à travers lequel il est connu à un instant donné. Bien entendu, cela n'est légitime que si $\mathcal{T}$ est soit le type effectif de $\mathcal{O}$, soit une généralisation de ce type.

Une telle conversion doit toujours être explicitée par le programmeur, elle n'est jamais implicite :

```
Point unPoint;
Pixel unPixel = new Pixel(a, b, c);
...
unPoint = unPixel;              // OK. Généralisation
...

unPixel = unPoint;              // ERREUR
...
unPixel = (Pixel) unPoint;      // OK. Particularisation
```

Les contraintes que doit vérifier une expression de la forme `(`$\mathcal{T}$`)` $\mathcal{E}$, où $\mathcal{T}$ est une classe et $\mathcal{E}$ une expression d'un type classe, ne sont pas les mêmes à la compilation et à l'exécution du programme :

- à la compilation, il faut que $\mathcal{T}$ soit sous-classe ou super-classe, directe ou indirecte, du type effectif de $\mathcal{E}$ (c'est-à-dire que les conversions ne sont acceptées à la compilation qu'entre deux classes qui se trouvent sur la même branche de l'arbre d'héritage),
- à l'exécution, il faut que la conversion représente une généralisation du type effectif de $\mathcal{E}$, sinon, une exception `ClassCastException` sera lancée.

Ces contraintes sont faciles à comprendre : il suffit de se rappeler que la généralisation et la particularisation sont des transformations sans travail (des « jeux de pointeurs »), qui ne peuvent en aucun cas ajouter à un objet des membres qu'il n'a pas.

Exemple. Une situation dans laquelle la généralisation et la particularisation apparaissent très naturellement est l'emploi d'outils généraux de la bibliothèque Java, comme les collections. Par exemple, une manière de représenter une ligne polygonale consiste à se donner une liste (ici, une liste chaînée, `LinkedList`) de points (ici, tirés au hasard) :

```
...
Point unPoint;
List lignePolygonale = new LinkedList();

for (int i = 0; i < nbr; i++) {
    unPoint = new Point((int)(XMAX * Math.random()), (int)(YMAX * Math.random()));
    lignePolygonale.add(unPoint);
}
...
```

Considérons l'expression « `lignePolygonale.add(unPoint)` ». Les `LinkedList` étant définies à un niveau de généralité maximal, l'objet `unPoint` ne peut être traité, dans les méthodes de la liste, que comme un simple `Object`, la classe la plus générale.

Du coup, lorsque ces points sont extraits de la liste, il faut explicitement leur redonner le type qui est le leur. Par exemple, cherchons à afficher les sommets d'une telle ligne polygonale :

```
...
Iterator iter = lignePolygonale.iterator();
while (iter.hasNext()) {
    unPoint = (Point) iter.next();
    application à unPoint d'un traitement nécessitant un Point
}
...
```

La conversion « `(Point) iter.next()` » est nécessaire, car le résultat rendu par `next` est un `Object`. Elle est certainement correcte, car dans la collection que `iter` parcourt nous n'avons mis que des objets `Point`.

Lorsqu'une liste n'est pas homogène comme ci-dessus, il faut s'aider de l'opérateur `instanceof` pour retrouver le type effectif des objets. Par exemple, supposons que `figure` soit une collection contenant des points et des objets d'autres types, voici comment en traiter uniquement les points :

```
...
Iterator iter = figure.iterator();
while (iter.hasNext()) {

    Object unObjet = iter.next();
    if (unObjet instanceof Point) {
        Point unPoint = (Point) unObjet;
        traitement de unPoint
    }
}
...
```

N.B. Le programme précédent traitera les points, mais aussi les pixels et les objets de toutes les sous-classes, directes et indirectes, de la classe `Point`. Telle est la sémantique de l'opérateur `instanceof` (c'est le comportement généralement souhaité).

7.6.2 Les méthodes redéfinies sont "virtuelles"

Soit `v` une variable d'un type classe $\mathcal{B}$ et `m` une méthode de $\mathcal{B}$; un appel tel que `v.m(`*arguments*`)` est légitime. Imaginons qu'au moment d'un tel appel, la valeur effective de `v` est instance de $\mathcal{D}$, une sous-classe de $\mathcal{B}$, dans laquelle `m` a été redéfinie. La question est : quelle version de `m` sera appelée ? Celle de $\mathcal{B}$, la classe de la variable `v`, ou bien celle de $\mathcal{D}$, la classe de la valeur effective de `v` ?

En Java, la réponse est : la seconde. Quel que soit le type sous lequel un objet est vu à la compilation, lorsqu'à l'exécution la machine appelle sur cet objet une méthode qui a été redéfinie, la définition effectivement employée est la plus particulière, c'est à dire *la plus proche du type effectif* de l'objet.

Exemple, trivial mais révélateur, avec la méthode `toStringtoString` :

```
Object unObjet = new Point(1, 2);
...
String str = unObjet.toString();
```

La chaîne affectée à `str` par l'expression ci-dessus est `"(1,2)"`, c'est-à-dire l'expression textuelle d'un objet `Point` (et non `"Point@12cf71"`, qui serait le résultat de la version de `toString` définie dans `Object`). Noter que c'est bien parce que `toString` est définie dans la classe `Object` que l'expression `unObjet.toString()` est acceptée à la compilation, mais à l'exécution c'est la version définie dans la classe `Point`, la classe de la valeur effective de `unObjet`, qui est employée.

Dans certains langages, comme C++, on appelle cela des méthodes virtuelles ; en Java, toutes les méthodes sont donc virtuelles. Cela a d'importantes conséquences méthodologiques, et rend le polymorphisme extrêmement intéressant et puissant.

Autre exemple (plus instructif). Considérons une application manipulant des classes d'objets devant être dessinés dans des fenêtres graphiques : des segments, des rectangles, des cercles, etc. Dans toutes ces classes il y a une méthode `seDessiner` prenant en charge la présentation graphique des objets en question.

La question est : comment manipuler ensemble ces objets différents, tout en ayant le droit d'utiliser ce qu'ils ont en commun ? Par exemple, comment avoir une collection de tels objets graphiques différents, et pouvoir appeler la méthode `seDessiner` de chacun ?

On l'aura compris, la solution consiste à définir une super-classe de toutes ces classes, dans laquelle la méthode `seDessiner` est introduite :

```
class ObjetGraphique {
    ...
    void seDessiner(Graphics g) {
    }
    ...
}
class Segment extends ObjetGraphique {
    int x0, y0, x1, y1;  // extrémités
    void seDessiner(Graphics g) {
        g.drawLine(x0, y0, x1, y1);
    }
    ...
}
class Triangle extends ObjetGraphique {
    int x0, y0, x1, y1, x2, y2;  // sommets
    void seDessiner(Graphics g) {
        g.drawLine(x0, y0, x1, y1);
        g.drawLine(x1, y1, x2, y2);
        g.drawLine(x2, y2, x0, y0);
    }
    ...
}
class Cercle extends ObjetGraphique {
    int x, y;  // centre
    int r;     // rayon
    void seDessiner(Graphics g) {
        g.drawOval(x - r, y - r, x + r, y + r);
    }
    ...
}
```

Exemple d'utilisation de ces classes : une `figure` est un tableau d'objets graphiques :

```
ObjetGraphique[] figure = new ObjetGraphique[nombre];
...
figure[i] = new Segment(arguments);
...
figure[j] = new Triangle(arguments);
...
figure[k] = new Cercle(arguments);
...
```

Les divers objets graphiques subissent une généralisation lors de leur introduction dans le tableau `figure`. Puisque `seDessiner` est définie dans la classe `ObjetGraphique`, on peut appeler cette méthode à travers les éléments de `figure`. L'intérêt de tout cela réside dans le fait que c'est la méthode *propre à chaque objet graphique particulier* qui sera appelée, non celle de la classe `ObjetGraphique` :

```
for (int i = 0; i < figure.length; i++)
    figure[i].seDessiner(g);    // ceci dépend de la valeur effective de figure[i]
```

Les méthodes redéfinies sont *nécessairement* virtuelles

La « virtualité » des méthodes peut donc se résumer ainsi : la méthode effectivement appelée par une expression comme *unObjet*`.`*uneMethode*`()` dépend de la valeur effective de *unObjet* au moment de cet appel, non de la déclaration de *unObjet* ou de propriétés statiques découlant du texte du programme.

Bien que la plupart du temps cette propriété soit extrêmement utile, on voudrait quelquefois pouvoir choisir la méthode a appeler parmi les diverses redéfinitions faites dans les super-classes. Il faut comprendre que cela ne se peut pas : le mécanisme des méthodes virtuelles en Java n'est pas débrayable.

Seule exception : si *uneMethode* est redéfinie dans une classe, l'expression `super.`*uneMethode*`()` permet d'appeler, depuis une méthode de la même classe, la version de `uneMethode` qui est en vigueur dans la super-classe. Mais il n'y a aucun autre moyen pour « contourner » une ou plusieurs redéfinitions d'une méthode.

On notera que les variables ne sont pas traitées comme les méthodes (mais la redéfinition des variables n'a pas d'intérêt, ce n'est qu'un masquage généralement maladroit). L'exemple suivant illustre la situation :

```
class A {
    String v = "variable A";
    String m() { return "méthode A"; }
}

class B extends A {
    String v = "variable B";
    String m() { return "méthode B"; }
}

class C extends B {
    String v = "variable C";
    String m() { return "méthode C"; }

    void test() {
        System.out.println( v );
        System.out.println( super.v );
        System.out.println( ((A) this).v );

        System.out.println( m() );
        System.out.println( super.m() );
        System.out.println( ((A) this).m() );
    }

    public static void main(String[] args) {
        new C().test();
    }
}
```

Affichage obtenu :

```
variable C
variable B
variable A          ← pour une variable, cela « marche »
méthode C
méthode B
méthode C           ← pour une méthode non
```

7.6.3 Méthodes et classes finales

Les méthodes virtuelles sont extrêmement utiles, mais elles ont un coût : de l'information cachée doit être ajoutée aux instances de la classe `ObjetGraphique` pour permettre à la machine Java de rechercher la méthode qui doit être effectivement appelée à l'occasion d'une expression comme

```
unObjetGraphique.seDessiner(g);
```

cette « recherche » prenant d'ailleurs du temps à l'exécution.

C'est la raison pour laquelle le programmeur peut optimiser son programme en déclarant qu'une méthode ne sera pas redéfinie dans les éventuelles sous-classes, ce qui permet au compilateur de lier statiquement les appels de cette méthode, au lieu de les lier dynamiquement comme le veut la règle générale.

Cela se fait par l'emploi du qualifieur `final`. Par exemple, s'il y a lieu de penser que le dessin d'un cercle sera le même dans la classe `Cercle` et dans les sous-classes de celle-ci, on peut déclarer cette méthode comme ceci :

```
class Cercle extends ObjetGraphique {
    int x, y;  // centre
    int r;     // rayon
```

```
    final void seDessiner(Graphics g) {            // cette méthode ne sera
        g.drawOval(x - r, y - r, x + r, y + r);    // pas redéfinie dans les
    }                                              // sous-classes de Cercle
    ...
}
```

Classes finales. Si une classe ne doit pas avoir de sous-classes, on peut effectuer d'un seul coup le travail d'optimisation précédent sur toutes ses méthodes en déclarant `final` la classe tout entière. Par exemple, si la classe `Triangle` n'est pas destinée à avoir des sous-classes, on peut l'écrire :

```
final class Triangle extends ObjetGraphique {
    int x0, y0, x1, y1, x2, y2;  // sommets
    void seDessiner(Graphics g) {
        g.drawLine(x0, y0, x1, y1);
        g.drawLine(x1, y1, x2, y2);
        g.drawLine(x2, y2, x0, y0);
    }
    ...
}
```

7.7 ♡ Abstraction

7.7.1 Méthodes abstraites

Revenons sur l'exemple de la section précédente :

```
class ObjetGraphique {
    ...
    void seDessiner(Graphics g) {
    }
    ...
}
```

Nous observons ceci : la méthode `seDessiner` doit être définie dans la classe `ObjetGraphique`, car le rôle de cette classe est de rassembler les éléments communs aux divers objets graphiques considérés, or la capacité de se représenter graphiquement est bien un de ces éléments, sans doute le principal. En même temps il nous est impossible, au niveau de la classe `ObjetGraphique`, d'écrire quoi que ce soit dans la méthode `seDessiner`, car nous n'en savons pas assez sur ce qu'il s'agit de dessiner.

On exprime cette situation en disant que `seDessiner` est une méthode *abstraite*. Sa déclaration dans la classe `ObjetGraphique` est une « promesse » : on annonce que tout objet graphique devra avoir une méthode `seDessiner`, qu'il n'est pas possible de fournir pour le moment.

En Java on peut déclarer qu'une méthode est abstraite à l'aide du qualifieur `abstract`. Le corps de la méthode doit alors être remplacé par un point virgule (et, comme on le verra à la section suivante, la classe contenant une telle méthode doit elle aussi être déclarée abstraite) :

```
abstract class ObjetGraphique {
    ...
    abstract void seDessiner(Graphics g);
    ...
}
```

Les méthodes abstraites d'une classe restent abstraites dans ses sous-classes, sauf si elles y ont une redéfinition qui leur donne un corps.

La magie des fonctions virtuelles, l'exemple canonique. Ce n'est pas parce qu'une méthode est abstraite qu'on ne peut pas l'appeler, y compris dans des classes où elle n'est pas encore définie. Par exemple, une manière (naïve) d'effacer un objet graphique consiste à le dessiner avec une plume qui a pour couleur la couleur du fond :

```
abstract class ObjetGraphique {
    ...
    abstract void seDessiner(Graphics g);
    ...
```



```
    void sEffacer(Graphics g) {
        g.setColor(getBackground());    // on prend la couleur du fond
        seDessiner(g);
    }
    ...
}
```

Une instance de la classe `ObjetGraphique` (mais nous verrons qu'il ne peut pas en exister) ne serait pas plus capable de s'effacer qu'elle ne le serait de se dessiner, mais toute sous-classe qui définit une version effective de `seDessiner` dispose aussitôt d'une version opérationnelle de `sEffacer`.

7.7.2 Classes abstraites

Une classe abstraite est une classe qui ne doit pas avoir d'instances ; elle n'est destinée qu'à avoir des sous-classes qui, elles, auront des instances. Cela peut provenir
- d'une impossibilité technique ; par exemple, une classe qui possède des méthodes abstraites (en propre ou héritées) ne peut pas avoir des instances, car ce seraient des objets dont une partie du comportement requis n'aurait pas été écrit,
- d'une impossibilité conceptuelle, c'est-à-dire un choix du concepteur. Par exemple, les classes *XxxAdapter* du paquet `java.awt.event` sont des classes entièrement faites de méthodes vides, dont d'éventuelles instances n'auraient aucun intérêt.

On indique qu'une classe est abstraite par le qualifieur `abstract` :

```
abstract class ObjetGraphique {
    ...
    abstract void seDessiner(Graphics g);
    ...
}
```

Il est obligatoire d'employer le qualifieur `abstract` devant toute classe qui possède des méthodes abstraites, propres *ou héritées*.

7.7.3 Interfaces

Parlant naïvement, on peut dire que les méthodes non abstraites d'une classe sont des « services » que celle-ci rend aux concepteurs de ses éventuelles sous-classes, tandis que les méthodes abstraites sont des « corvées » que la classe transmet aux concepteurs des sous-classes, qui devront en donner des définitions.

Une *interface* est une classe entièrement faite de membres publics qui sont
- des méthodes abstraites,
- variables statiques finales (c'est-à-dire des constantes de classe).

Une interface se déclare avec le mot-clé `interface` au lieu de `class`. Il n'y a alors pas besoin d'écrire les qualifieurs `public` et `abstract` devant les méthodes, ni `public`, `static` et `final` devant les variables.

Par exemple, si elle est entièrement faite de méthodes abstraites, notre classe `ObjetGraphique` peut être déclaré comme une interface :

```
interface ObjetGraphique {
    void seDessiner(Graphics g);
    ...
}
```

Autre exemple, extrait de la bibliothèque Java (plus précisément du paquet `java.util`) :

```
interface Collection {
    boolean isEmpty();
    int size();
    boolean contains(Object o);
    boolean add(Object o);
    Iterator iterator();
    ...
}
```

Une interface est une *spécification* : elle fixe la liste des méthodes qu'on est certain de trouver dans toute classe qui déclare être conforme à cette spécification.

Cela s'appelle une *implémentation* de l'interface, et s'indique avec le qualifieur `implements` :

```
class Tas implements Collection {
    public boolean isEmpty() {
        implémentation de la méthode isEmpty
    }
    public int size() {
        implémentation de la méthode size
    }
    ...
}
```

Notez que les méthodes des interfaces sont toujours publiques (implicitement) ; par conséquent, leurs définitions dans des sous-classes doivent être explicitement qualifiées `public`, sinon une erreur sera signalée.

Deux interfaces peuvent hériter l'une de l'autre. Par exemple, toujours dans le paquet `java.util` on trouve l'interface `SortedSet` (ensemble trié) qui est une sous-interface de `Set`, elle-même une sous-interface de `Collection`, etc.

D'autre part, la relation `implements` entre une classe et une interface est aussi une sorte d'héritage. Cet héritage est *multiple* : une classe peut implémenter plusieurs interfaces distincts. Par exemple, on pourrait définir une classe `Figure` qui serait une sorte de `Collection` (d'objets graphiques) et en même temps une sorte d'`ObjetGraphique` (puisque capable de se dessiner) :

```
class Figure implements Collection, ObjetGraphique {
    ...
    public void seDessiner(Graphics g) {
        appeler seDessiner sur chaque membre de la figure
    }

    public boolean add(Object o) {
        ajouter o (un objet graphique) à la figure
    }

    public boolean isEmpty() {
        la figure est-elle vide ?
    }
    ...
}
```

L'intérêt de cette sorte d'héritage multiple apparaît clairement : une instance de notre classe `Figure` pourra être mise à tout endroit où une `Collection` est attendue, et aussi à tout endroit où un `ObjetGraphique` est attendu.

Interface pour transmettre une fonction

Java ne comportant pas la notion de pointeur, encore moins de pointeur sur une fonction, comment fait-on dans ce langage pour qu'une méthode soit argument d'une autre ?

La solution consiste à déclarer la méthode comme membre d'une interface définie à cet effet, et à mettre l'interface comme type de l'argument en question. Voici par exemple comment définir, dans une classe `CalculNumerique`, une méthode `zero` qui recherche par dichotomie une solution approchée à ε-près de l'équation $f(x) = 0,\ x \in [\ a,\ b\ ]$:

```
class MonCalculNumerique {
    ...
    static double zero(double a, double b, double epsilon, Fonction f) {
        double c;
        if (f.val(a) > f.val(b)) {
            c = a; a = b; b = c;
        }
```



```
            while (Math.abs(b - a) > epsilon) {
                c = (a + b) / 2;
                if (f.val(c) < 0)
                    a = c;
                else
                    b = c;
            }
            return (a + b) / 2;
        }
        ...
    }
```

`Fonction` est une interface définie pour cette occasion. Elle se réduit à une méthode `val`, qui représente la fonction en question :

```
interface Fonction {
    double val(double x);
}
```

Notez à quel point `Fonction` et `zero` illustrent la notion d'interface et son utilité : « être un objet `Fonction` » ce n'est rien d'autre que « posséder une méthode nommée `val` qui prend un `double` et rend un `double` ». Il n'y a pas besoin d'en savoir plus pour écrire la méthode `zero` !

A titre d'exemple proposons-nous d'utiliser cette méthode pour calculer la valeur de $\sqrt{2}$ approchée à 10^{-8} près. Il s'agit de résoudre l'équation $x^2 - 2 = 0$ sur, par exemple, l'intervalle $[\ 0,\ 2\ ]$. La fonction à considérer est donc $f(x) = x^2 - 2$.

Première méthode, lourde, définir une classe exprès :

```
class X2moins2 implements Fonction {
    public double val(double x) {
        return x * x - 2;
    }
}

class Essai {
    public static void main(String[] args) {
        Fonction f = new X2moins2();
        double z = CalculNumerique.zero(0, 2, 1e-8, f);
        System.out.println(z);
    }
}
```

Remarquant que la classe `X2moins2` n'a qu'une instance, on peut alléger notre programme en utilisant à sa place une classe anonyme :

```
class Essai {
    public static void main(String[] args) {
        Fonction f = new Fonction() {
            public double val(double x) {
                return x * x - 2;
            }
        };
        double z = CalculNumerique.zero(0, 2, 1e-8, f);
        System.out.println(z);
    }
}
```

On peut même faire l'économie de `f` (mais le programme obtenu est-il plus lisible ?) :

```
public class Courant {
    public static void main(String[] args) {
        double z = CalculNumerique.zero(0, 2, 1e-8, new Fonction() {
            public double val(double x) {
                return x * x - 2;
            }
        });
        System.out.println(z);
    }
}
```

8 Exceptions

On dit qu'une exception se produit lorsque les conditions d'exécution sont telles que la poursuite du programme devient impossible ou incorrecte. Exemple : l'indice d'un tableau se trouve en dehors des bornes, la mémoire manque lors de la création d'un objet, etc.

Les exceptions sont généralement détectées par les fonctions de bas niveau, voire par la machine Java elle-même, mais la plupart du temps ce n'est que dans les fonctions de haut niveau qu'on peut programmer la réaction adéquate à de telles situations non prévues.

Le mécanisme des *exceptions* de Java est un moyen de communication permettant aux éléments des couches « basses » des programmes de notifier aux couches « hautes » la survenue d'une condition anormale.

Par exemple, nous avons déjà vu une classe `Point` dont les instances étaient soumises à validation. Une bonne manière de traiter l'invalidité des arguments de la construction d'un point consiste à « lancer » une exception :

```
class Point {
    static final int XMAX = 1024, YMAX = 768;
    int x, y;

    Point(int a, int b) throws Exception {
        if (a < 0 || a >= XMAX || b < 0 || b >= YMAX)
            throw new Exception("Coordonnées illégales");
        x = a;
        y = b;
    }
    ...
}
```

Les exceptions ont diverses sortes de causes :
- des causes synchrones[48], c'est-à-dire provoquées par l'exécution d'une instruction, comme la division par zéro, l'impossibilité d'allouer de la mémoire, l'impossibilité de charger une classe, etc.,
- cas particulier du précédent, l'exécution de l'instruction `throw`,
- des causes asynchrones ; en Java il n'y en a que deux : l'exécution, depuis un autre thread, de la méthode `Thread.stop` et la survenue d'une erreur interne de la machine Java.

Une fois lancée, une exception « remonte », c'est-à-dire :
- l'exécution du bloc de code dans lequel l'exception a été lancée est abandonné,
- si ce bloc n'est ni un bloc `try` ni le corps d'une méthode, alors il est lui-même vu comme une instruction ayant lancé l'exception,
- si ce bloc est le corps d'une méthode, alors l'appel de cette dernière, dans la méthode appelante, est vu comme une instruction ayant lancé l'exception,
- si ce bloc est un bloc `try` comportant une clause `catch` compatible avec l'exception, alors cette dernière est « attrapée » : le code associé à cette clause `catch` est exécuté, et l'exception ne va pas plus loin.

8.1 Interception des exceptions

Syntaxe :

```
try {
    bloc_0
}
catch(type_exception_1 arg_1) {
    bloc_1
}
catch(type_exception_2 arg_2) {
    bloc_2
}
...
finally {
    bloc_k
}
```

48. Ces causes synchrones montrent qu'il n'est pas correct de se limiter à dire qu'une exception est une situation *imprévisible*. Une division par zéro ou l'utilisation d'une référence nulle ne sont qu'apparemment imprévisibles, si on avait mieux analysé le programme et ses données on aurait pu les prévoir avec certitude. Mais, comme on le verra, il est très commode de ranger parmi les exceptions ces événements, malgré tout difficiles à prévoir et généralement aux conséquences graves.

Les blocs `catch` sont en nombre quelconque, le bloc `finally` est optionnel.

Cela fonctionne de la manière suivante : les instructions qui composent $bloc_0$ sont exécutées. Plusieurs cas sont possibles :

1. Si l'exécution de $bloc_0$ ne provoque le lancement d'aucune exception, les bouts de code composant $bloc_1$ ... $bloc_{k-1}$ sont ignorés.
2. Si une exception $\mathcal{E}$ est lancée par une des instructions de $bloc_0$, ce bloc est immédiatement abandonné et on recherche le premier i tel que $\mathcal{E}$ soit instance, directe ou indirecte, de $type\_exception_i$. Alors, $\mathcal{E}$ est affecté à l'argument arg_i et le bloc correspondant, $bloc_i$, est exécuté (d'une certaine manière, cette partie ressemble donc à un appel de fonction).
3. Si le type de $\mathcal{E}$ ne correspond à aucun des $type\_exception_i$ alors l'exception continue sa trajectoire : l'ensemble « `try...catch...` » est considéré comme une instruction ayant lancé $\mathcal{E}$.

L'éventuel code composant le bloc `finally` est exécuté quelle que soit la manière dont le bloc `try` a été quitté, y compris le cas n° 3. Autrement dit, ce bloc `finally` est le moyen d'assurer qu'un code sera exécuté à la suite de $bloc_0$ même si ce bloc se termine en catastrophe.

8.2 Lancement des exceptions

Une exception se lance par l'instruction

`throw` *exception*`;`

où *exception* est une expression dont la valeur doit être un objet d'une des classes de la hiérarchie expliquée à la section suivante.

8.2.1 Hiérarchie des exceptions

Toutes les classes d'exceptions appartiennent à un sous-arbre de l'arbre des classes dont la racine est la classe `Throwable` :

`Object`	
`Throwable`	Classe de base de toutes les exceptions.
`Error`	Erreurs graves (en particulier, exceptions asynchrones) qu'il n'est pas raisonnable de chercher à récupérer.
`Exception`	Exceptions méritant d'être détectées et traitées.
`RuntimeException`	Exceptions pouvant survenir durant le fonctionnement normal de la machine Java : - indice de tableau hors des bornes - accès à un membre d'une référence `null` - erreur arithmétique (division par zéro, etc.)
`IOException`	Exceptions pouvant survenir pendant les opérations d'entrée/sortie
Vos propres classes exceptions viennent ici	
Autres classes	

8.2.2 Déclaration des exceptions lancées par une méthode

Une méthode contenant une instruction pouvant lancer une exception d'un certain type $\mathcal{E}$ doit nécessairement

- soit attraper l'exception, au moyen d'un bloc « `try...catch...` » adéquat,
- soit déclarer qu'elle est susceptible de lancer, ou plutôt de laisser échapper, des exceptions du type $\mathcal{E}$.

Cette déclaration se fait par un énoncé de la forme

`throws` *listeDExceptions*

placé à la fin de l'en-tête de la méthode. Par exemple, le constructeur de la classe `Point` étant écrit ainsi

```
Point(int a, int b) throws Exception {
    if (a < 0 || a >= XMAX || b < 0 || b >= YMAX)
        throw new Exception("Coordonnées illégales");
    x = a;
```



```
        y = b;
    }
```

(notez que l'énoncé `throws Exception` est obligatoire) pour l'essayer, nous devrons écrire une méthode `main` soit de la forme :

```
public static void main(String[] args) {
    ...
    try {
        Point p = new Point(u, v);
        ...
    }
    catch (Exception e) {
        System.out.println("Problème: " + e.getMessage());
    }
    ...
}
```

soit de la forme :

```
public static void main(String[] args) throws Exception {
    ...
    Point p = new Point(u, v);
    ...
}
```

NOTE. Dans cet exemple nous nous contentons d'un travail assez sommaire, en ne définissant pas de classe spécifique pour les exceptions qui intéressent notre programme. En règle générale on définit de telles classes, car cela permet une récupération des erreurs plus sélective :

```
class ExceptionCoordonnesIllegales extends Exception {
    ExceptionCoordonnesIllegales() {
        super("Coordonnées illégales");
    }
}

class Point {
    Point(int a, int b) throws Exception {
        if (a < 0 || a >= XMAX || b < 0 || b >= YMAX)
            throw new ExceptionCoordonnesIllegales();
        x = a;
        y = b;
    }
    ...
    public static void main(String[] args) {
        ...
        try {
            Point p = new Point(u, v);
        }
        catch (ExceptionCoordonnesIllegales e) {
            System.out.println("Problème avec les coordonnées du point");
        }
        ...
    }
}
```

EXCEPTIONS CONTRÔLÉES ET NON CONTRÔLÉES. Les exceptions contrôlées sont celles qu'on est obligé de déclarer dans l'en-tête de toute méthode susceptible de les lancer ; les autres exceptions sont dites non contrôlées.

Toutes les exceptions sont contrôlées, sauf :

- les instances de `Error`, car elles expriment des événements irrattrapable,
- les instances de `RuntimeException`, car s'il fallait déclarer ces exceptions, cela concernerait *toutes* les méthodes.

8.3 Assert

Le service rendu par l'instruction *assert*[49] est double :
- *assert* est un mécanisme simple et concis permettant de vérifier, à l'exécution, qu'à certains endroits d'un programme des conditions qui doivent être remplies le sont effectivement,
- *assert* comporte un dispositif global et immédiat d'activation ou désactivation qui permet, au moment de l'exécution, de choisir entre plusieurs modes, allant d'un « mode atelier » muni de nombreuses vérifications (qui consomment du temps) à un « mode utilisateur final », plus rapide puisque les occurrences de *assert* y sont neutralisées.

L'instruction *assert* se présente sous deux formes :

`assert` *expression*$_1$ `;`

et

`assert` *expression*$_1$ `:` *expression*$_2$ `;`

expression$_1$ est une expression booléenne, *expression*$_2$ une expression d'un type quelconque. Dans les deux cas, l'exécution de l'instruction *assert* commence par évaluer *expression*$_1$. Si cette expression est vraie, l'exécution continue sans que rien ne se passe.

En revanche, si *expression*$_1$ se révèle fausse alors une erreur[50] de type `java.lang.AssertionError` est lancée. Si c'est la deuxième forme de *assert* qui a été utilisée, la valeur d'*expression*$_2$ figure dans le message associé à l'erreur.

Exemple :

```
void uneMethode(int x) {
    ...
    assert MIN < x && x < MAX;
    ...
}
```

Lors d'une exécution dans laquelle `uneMethode` est appelée avec un argument inférieur à `MIN` ou supérieur à `MAX` on obtiendra l'arrêt du programme – sauf si l'erreur est attrapée – avec un message du style :

```
java.lang.AssertionError
    at Tests.uneMethode(Tests.java:187)
    at Tests.main(Tests.java:12)
Exception in thread "main"
```

Si l'instruction *assert* avait été écrite sous la forme :

```
    ...
    assert MIN < x && x < MAX : "MIN < x && x < MAX, x = " + x;
    ...
```

alors le message aurait été

```
java.lang.AssertionError: MIN < x && x < MAX, x = -25
    at Tests.uneMethode(Tests.java:187)
    at Tests.main(Tests.java:12)
Exception in thread "main"
```

NOTE. L'instruction *assert* est un puissant outil pour découvrir des erreurs de conception et on ne saurait trop insister sur l'intérêt de son utilisation. Mais elle n'est pas la bonne manière de sauver des situations que le programmeur ne peut pas empêcher, comme des erreurs dans les données : saborder le programme n'est en général pas la bonne réaction à une faute de frappe commise par l'utilisateur !

De même, *assert* n'est pas la bonne manière d'envoyer des messages à l'utilisateur final : en principe non-informaticien, celui-ci n'est pas censé comprendre un diagnostic abscons mentionnant des classes dont il ignore l'existence et des numéros de ligne dont il n'a que faire. L'information que *assert* produit ne s'adresse qu'à l'auteur du programme.

En résumé, *assert* est très pratique mais il faut le remplacer par une autre construction (comme une instruction *if* qui produit un dialogue avec l'utilisateur sans tuer le programme) lorsque

49. Notez que, contrairement à ce qui se pase en C, le mécanisme *assert* n'est pas réalisé en Java par une bibliothèque séparée, mais par une vraie instruction, c'est-à-dire une extension de la syntaxe du langage.

50. Rappelons que les erreurs (classe `java.lang.Error`, sous-classe de `java.lang.Throwable`) sont une sorte d'exceptions *non contrôlées*. Elles sont destinées principalement – mais non exclusivement – à représenter des problèmes sérieux que les applications ne doivent pas chercher à rattraper.

– il s'agit de détecter des erreurs autres que de conception ou de programmation,
– il s'agit de produire des messages adressés à l'utilisateur final,
– il s'agit de vérifications qui doivent être faites même lorsque le mécanisme *assert* n'est pas actif (voir ci-dessous).

Activation et désactivation de *assert*

Par défaut, *assert* est désactivé : quand une application Java est exécutée, cela se passe comme si ces instructions avaient été enlevées du code.

Pour rendre le mécanisme actif, il faut lancer la machine Java avec l'option `-enableassertions` ou `-ea`, qui peut prendre un argument, de la manière suivante :

`-ea`	activation de *assert* partout dans l'application,
`-ea:`*unPackage*`...`	activation de *assert* dans le paquetage indiqué et tous ses sous-paquetages,
`-ea:...`	activation de *assert* dans le paquetage sans nom,
`-ea:`*uneClasse*	activation de *assert* dans la classe indiquée,
`-esa`	activation de *assert* dans toutes les classes du système.

Exemple. Exécution d'une application dont la méthode principale est dans la classe `MaClasse` en activant assert dans toutes les classes du paquetage courant :

```
java -ea ... MaClasse
```

9 Quelques classes utiles

9.1 Nombres et outils mathématiques

9.1.1 Classes-enveloppes des types primitifs

Java comporte une unique arborescence de classes, de racine `Object`, dans laquelle se trouvent les classes de toutes les données manipulées par les programmes... sauf les valeurs de types primitifs qui, pour d'évidentes raisons d'efficacité, gardent leur statut de données élémentaires, utilisables par les opérations basiques câblées dans les ordinateurs.

Afin que ces données simples puissent être traitées comme des objets lorsque cela est nécessaire, Java fournit huit classes-enveloppes, une pour chaque type primitif : `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` et `Character`. Le rôle principal de chaque instance d'une de ces classes est d'encapsuler une valeur du type primitif correspondant. Ces classes ont à peu près les mêmes méthodes ; parmi les principales :

- un constructeur prenant pour argument la valeur du type primitif qu'il s'agit d'envelopper,
- la réciproque : une méthode *xxx*`Value` (*xxx* vaut `byte`, `short`, etc.) pour obtenir la valeur de type primitif enveloppée,
- une méthode nommée `toString` pour obtenir cette valeur sous forme de chaîne de caractères,
- éventuellement, une méthode statique, appelée `parse`*Xxx*, permettant d'obtenir une valeur d'*un type primitif* à partir de sa représentation textuelle (cela s'appelle *parse* car former une valeur à partir d'un texte c'est faire une analyse lexicale),
- éventuellement, une méthode statique `valueOf` pour construire un tel objet à partir de sa représentation sous forme de texte. Par exemple, `IntegerValueOf(`*uneChaine*`)` donne le même résultat que `new Integer(Integer.parseInt(`*uneChaine*`))` ;

Pour chacune des huit classes, ces méthodes sont [51] :

Classe `Byte` :

`Byte(byte b)`	conversion `byte` → `Byte`
`byte byteValue()`	conversion `Byte` → `byte`
`String toString()`	conversion `Byte` → `String`
`static Byte valueOf(String s)`	conversion `String` → `Byte`
`static byte parseByte(String s)`	conversion `String` → `byte`

Classe `Short` :

`Short(short s)`	conversion `short` → `Short`
`short shortValue()`	conversion `Short` → `short`
`String toString()`	conversion `Short` → `String`
`static Short valueOf(String s)`	conversion `String` → `Short`
`static short parseShort(String s)`	conversion `String` → `short`

Classe `Integer` :

`Integer(int i)`	conversion `int` → `Integer`
`int intValue()`	conversion `Integer` → `int`
`String toString()`	conversion `Integer` → `String`
`static Integer valueOf(String s)`	conversion `String` → `Integer`
`static int parseInt(String s)`	conversion `String` → `int`

Classe `Long` :

`Long(long l)`	conversion `long` → `Long`
`long longValue()`	conversion `Long` → `long`
`String toString()`	conversion `Long` → `String`
`static Long valueOf(String s)`	conversion `String` → `Long`
`static long parseLong(String s)`	conversion `String` → `long`

Classe `Float` :

`Float(float f)`	conversion `float` → `Float`
`float floatValue()`	conversion `Float` → `float`
`String toString()`	conversion `Float` → `String`
`static Float valueOf(String s)`	conversion `String` → `Float`
`static float parseFloat(String s)`	conversion `String` → `float`

51. Les deux premières méthodes listées ici pour chacune de ces huit classes (le constructeur et la méthode *xxx*`Value()`) assurent deux opérations qu'on appelle familièrement l'*emballage* et le *déballage* d'une valeur primitive.

On notera qu'à partir de Java 5 ces opérations deviennent implicites (cf. section 12.2) : le compilateur se charge de les insérer là où elles sont nécessaires.

Classe `Double` :

`Double(double d)`	conversion `double` → `Double`
`double doubleValue()`	conversion `Double` → `double`
`String toString()`	conversion `Double` → `String`
`static Double valueOf(String s)`	conversion `String` → `Double`
`static double parseDouble(String s)`	conversion `String` → `double`

Classe `Boolean` :

`Boolean(boolean b)`	conversion `boolean` → `Boolean`
`boolean booleanValue()`	conversion `Boolean` → `boolean`
`String toString()`	conversion `Boolean` → `String`
`static Boolean valueOf(String s)`	conversion `String` → `Boolean`

Classe `Character` :

`Character(char c)`	conversion `char` → `Character`
`char charValue()`	conversion `Character` → `char`
`String toString()`	conversion `Character` → `String`

A propos de conversions, signalons également l'existence dans la classe `String` des méthodes suivantes :

`static String valueOf(int i)`	conversion `int` → `String`
`static String valueOf(long l)`	conversion `long` → `String`
`static String valueOf(float f)`	conversion `float` → `String`
`static String valueOf(double d)`	conversion `double` → `String`
`static String valueOf(boolean b)`	conversion `boolean` → `String`
`static String valueOf(char c)`	conversion `char` → `String`
`static String valueOf(Object o)`	conversion `Object` → `String`

9.1.2 Fonctions mathématiques

La classe `Math` représente la bibliothèque mathématique. Elle n'est pas destinée à avoir des instances : toutes ses méthodes sont statiques. On y trouve :

`static double E`, `static double PI`

Les constantes e (2.718281828459045) et π (3.141592653589793)

type `abs(`*type* `v)`

Valeur absolue ; *type* est un des types `int`, `long`, `float` ou `double`

type `max(`*type* `a,` *type* `b)`

Le plus grand de `a` et `b` ; *type* est un des types `int`, `long`, `float` ou `double`

type `min(`*type* `a,` *type* `b)`

Le plus petit de `a` et `b` ; *type* est un des types `int`, `long`, `float` ou `double`

`double floor(double v)`

Le plus grand entier inférieur ou égal à `v`.

`double ceil(double v)`

Le plus petit entier supérieur ou égal à `v`.

`double rint(double v)`

L'entier le plus proche de `v` ; s'il y en a deux, celui qui est pair.

`int round(int v)`

L'entier le plus proche de `v`.
La valeur de `round(v)` est la même que celle de `(int) floor(v + 0,5)`.

`double random()`

Un nombre pseudo-aléatoire dans $[0, 1[$. (S'agissant de nombres pseudo-aléatoires, la classe `java.util.Random` permet de faire des choses plus savantes.)

`double exp(double x)`, `double log(double x)`

Exponentielle et logarithme nepérien.

`double pow(double a, double b)`

Elevation de `a` à la puissance `b`.

La documentation ne donne pas d'indication sur l'algorithme employé ni sur son efficacité, mais il est raisonnable de penser que si `b` est entier (c'est-à-dire si `b = Math.ceil(b)`, alors a^b est calculé en faisant des multiplications, aussi peu nombreuses que possible. Dans le cas général, a^b est calculé par la formule $e^{b \log a}$.

`double sin(double x)`, `double cos(double x)`, `double tan(double x)`,
`double asin(double x)`, `double acos(double x)`, `double atan(double x)`

Fonctions trigonométriques habituelles.

`double atan2(double b, double a)`

Rend une valeur de l'intervalle $]-\pi, \pi]$ qui est l'argument du complexe $a + ib$ (pour $a \neq 0$ c'est donc la même chose que $\tan \frac{b}{a}$).

9.1.3 Nombres en précision infinie

Le paquet `java.math` (à ne pas confondre avec la classe `Math` du paquet `java.lang`) offre des classes permettant de manipuler des nombres avec autant de chiffres que nécessaire.

NOMBRES ENTIERS. Les instances de la classe `BigInteger` représentent des nombres entiers. Parmi les principaux membres de cette classe :

`BigInteger ZERO` : la constante 0
`BigInteger ONE` : la constante 1
`BigInteger(String s)` : conversion `String` → `BigInteger`
`BigInteger(byte[] t)` : conversion `byte[]` → `BigInteger`
`BigInteger add(BigInteger b)` : addition
`BigInteger subtract(BigInteger b)` : soustraction
`BigInteger multiply(BigInteger b)` : multiplication
`BigInteger divide(BigInteger b)` : quotient
`BigInteger remainder(BigInteger b)` : reste
`BigInteger[] divideAndRemainder(BigInteger b)` : quotient et reste
`BigInteger gcd(BigInteger b)` : P.G.C.D.
`BigInteger compareTo(BigInteger b)` : comparaison
`BigInteger max(BigInteger b)` : max
`BigInteger min(BigInteger b)` : min
`int intValue()` : conversion `BigInteger` → `int`
`long longValue()` : conversion `BigInteger` → `long`
`static BigInteger valueOf(long v)` : conversion `long` → `BigInteger`

EXEMPLE. Le programme suivant calcule la factorielle d'un entier donné en argument :

```
class Factorielle {
    public static void main(String[] args) {
        if (args.length <= 0)
            System.out.println("Emploi: java Factorielle <nombre>");
        else {
            int n = Integer.parseInt(args[0]);
            BigInteger k = BigInteger.ONE;
            BigInteger f = BigInteger.ONE;
            for (int i = 0; i < n; i++) {
                f = f.multiply(k);
                k = k.add(BigInteger.ONE);
            }
            System.out.println(f);
        }
    }
}
```

exemple d'utilisation :

```
>java Factorielle 500
1220136825991110068701238785423046926253574342803192842192413588385845370
31538819976054964475022032818630136164771482035841633787220781772004807 8
   etc.
000000000000000000000000000000000000000000000000000000000000000000000000
>
```

NOMBRES DÉCIMAUX. Les objets `BigDecimal` sont des nombres décimaux en grande précision. Plus précisément, un `BigDecimal` *nombre* est représenté par un couple formé d'un `BigInteger` *mantisse* et d'un int (32 bits) *echelle* tels que

$$nombre = \frac{mantisse}{10^{echelle}}$$

Parmi les principaux membres de cette classe :

`BigDecimal abs()` : valeur absolue

`BigDecimal add(BigDecimal val)` : addition

`BigDecimal subtract(BigDecimal val)` : soustraction

`BigDecimal multiply(BigDecimal val)` : multiplication

`BigDecimal divide(BigDecimal val, int roundingMode)` : division

`int compareTo(BigDecimal val)` : comparaison

`BigDecimal max(BigDecimal val)` : max

`BigDecimal min(BigDecimal val)` : min

`float floatValue()` : conversion `BigDecimal` → `float`

`double doubleValue()` : conversion `BigDecimal` → `double`

`int intValue()` : conversion `BigDecimal` → `int`

`long longValue()` : conversion `BigDecimal` → `long`

`BigInteger unscaledValue()` : obtention de la partie *mantisse*

`int scale()` : obtention de la partie *échelle*

`BigDecimal setScale(int scale)` : définition de la partie *échelle*

`BigDecimal movePointLeft(int n)`, `BigDecimal movePointRight(int n)` : déplacement de la virgule

9.2 Collections et algorithmes

NOTE JAVA 5. *Les classes et interfaces de la bibliothèque des collections et algorithmes ont été considérablement modifiées par l'introduction de la généricité en Java 5. Notez bien que les collections sont expliquées ici comme en Java 1.4 ; pour les nouveautés apparues en Java 5 consultez la section 12.5.*

Les programmes que nous réalisons couramment manipulent trois sortes de données : les valeurs de types primitifs, les objets et les *collections d'objets*. Dans les sections précédentes nous avons vu ce que Java propose pour les types primitifs et les objets.

Pour représenter les pluralités d'objets nous ne connaissons pour l'instant que les tableaux. Ceux-ci ont un avantage considérable, la possibilité d'accéder directement (i.e. en « temps constant ») à un élément à partir de son indice. Mais ils ont aussi un important inconvénient, le manque de souplesse dans la gestion de leur espace, puisqu'il faut connaître le nombre maximum de leurs éléments au moment de leur création (d'où un risque du débordement).

Cette section montre qu'à côté des tableaux, la bibliothèque Java propose une grande variété de collections, qui sont des structures de données beaucoup plus souples et riches.

9.2.1 Collections et tables associatives

Le paquet `java.util` contient les interfaces et les classes pour la manipulation des *collections* d'objets. Nous donnons ici la liste exhaustive de ces interfaces et classes, avec une explication du principe de chacune [52].

52. Il n'est pas aisé d'expliquer les collections et, encore moins, les algorithmes qui leur sont associés, car on ne peut pas se limiter aux spécification de ces classes. Par exemple, pour faire comprendre les différences qu'il y a entre une `ArrayList` et un `Vector` il faut se pencher sur les implémentations de ces classes, ce qui va contre le principe d'encapsulation.

Tout au long de cette section on gardera à l'esprit que les indications sur l'implémentation restent des *recommandations* et que, même si cela semble bizarre, une réalisation particulière de la bibliothèque pourrait ne pas les suivre, à condition de respecter les spécifications.

Pour les détails précis, on se référera avec profit aux pages concernant le paquetage `java.util` dans la documentation de l'API (`http://java.sun.com/javase/6/docs/api/`) et dans le tutoriel Java (`http://java.sun.com/tutorial/`).
Dans les descriptions suivantes la relation d'héritage entre les classes est indiquée par la marge laissée à gauche :

INTERFACES

`Collection`. Cette interface représente la notion de *collection d'objets* la plus générale. A ce niveau d'abstraction, les opérations garanties sont : obtenir le nombre d'éléments de la collection, savoir si un objet donné s'y trouve, ajouter (sans contrainte sur la position) et enlever un objet, etc.

On notera que les éléments des collections sont déclarés avec le type `Object`. Cela a deux conséquences principales[53] :

1. Tout objet particulier peut devenir membre d'une collection, et il ne « perd » rien en le devenant, mais la vue qu'on en a en tant que membre de la collection est la plus générale qui soit. Par conséquent, quand un objet est extrait d'une collection il faut lui « redonner » explicitement le type particulier qui est le sien. Au besoin, relisez la section 7.6.1.
2. Les valeurs de types primitifs (`boolean`, `int`, `float`, etc.) ne peuvent pas être directement ajoutées aux collections, il faut auparavant les « emballer » dans des instances des classes enveloppes correspondantes (`Boolean`, `Integer`, `Float`, etc.).

Point important, toutes les collections peuvent être *parcourues*. Cela se fait à l'aide d'un itérateur (interfaces `Iterator` ou `Enumeration`, voyez la section 9.2.2), un objet qui assure que tous les éléments de la collection sont atteints, chacun à son tour, et qui encapsule les détails pratiques du parcours, dépendant de l'implémentation de la collection.

`List`. Il s'agit de la notion de séquence[54], c'est-à-dire une collection où chaque élément a un rang. On trouve ici des opérations liées à l'accès direct aux éléments (insertion d'un élément à un rang donné, obtention de l'élément qui se trouve à un rang donné, etc.) même si, à ce niveau de généralité, on ne peut pas donner d'indication sur le coût de tels accès.

`Set`. Comme c'est souvent le cas, la notion d'ensemble est celle de *collection sans répétition* : un `Set` ne peut pas contenir deux éléments e_1 et e_2 vérifiant e_1`.equals(`e_2`)`.
Pour l'essentiel, l'interface `Set` n'ajoute pas des méthodes à l'interface `Collection`, mais uniquement la garantie de l'unicité des éléments lors de la construction d'un ensemble ou de l'ajout d'un élément.
Aucune indication n'est donnée *a priori* sur l'ordre dans lequel les éléments d'un `Set` sont visités lors d'un parcours de l'ensemble.

`SortedSet`. Un ensemble trié garantit que, lors d'un parcours, les éléments sont successivement atteints en ordre croissant
- soit selon un ordre propre aux éléments (qui doivent alors implémenter l'interface `Comparable`),
- soit selon une relation d'ordre (un objet `Comparator`) donnée explicitement lors de la création de l'ensemble.

`Map`. Cette interface formalise la notion de *liste associative* ou *dictionnaire*, c'est-à-dire une collection de couples (*clé*, *valeur*) appelés *associations*.

Les opérations fondamentales des dictionnaires sont l'ajout et la suppression d'une association et la recherche d'une association à partir de la valeur d'une clé.

53. Ces deux règles sont importantes et il faut s'assurer de bien les comprendre. Nous devons cependant signaler que dans Java 5 elles sont considérablement adoucies par deux éléments nouveaux :
- les types paramétrés (cf. section 12.5.1) permettent de travailler avec des collections dont les éléments sont plus précis que de simples `Object`,
- l'emballage et déballage automatique (cf. section 12.2) simplifient notablement l'introduction et l'extraction des éléments des collections.

54. Pour exprimer que les éléments d'une séquence ont un rang, la documentation officielle parle de *collections ordonnées*. Attention, l'expression est trompeuse, cela n'a rien à voir avec une quelconque relation d'ordre sur les éléments (les collections respectant une relation d'ordre sont appelées *collections triées*).

Sont fournies également des méthodes pour obtenir diverses *vues* du dictionnaire : l'ensemble des clés, la collection des valeurs, l'ensemble des associations.

L'unicité des clés est garantie : un objet `Map` ne peut pas contenir deux associations (c_1, v_1) et (c_2, v_2) telles que c_1`.equals(`c_2`)`.

`SortedMap`. Un dictionnaire dans lequel les associations sont placées dans l'ordre croissant des clés, ce qui se manifeste lors des parcours des trois collections associées au dictionnaire : l'ensemble des clés, la collection des valeurs et l'ensemble des associations.

Classes

`AbstractCollection`. Cette classe abstraite offre un début d'implémentation de l'interface `Collection`, destinée à alléger l'effort à fournir pour disposer d'une implémentation complète.

Pour avoir une collection immuable il suffit de définir une sous-classe de `AbstractCollection`, dans laquelle seules les méthodes `iterator()` et `size()` sont à écrire. Pour avoir une collection modifiable il faut en outre surcharger la méthode `add(Object o)`.

`AbstractList`. Classe abstraite qui est un début d'implémentation de l'interface `List`.

Attention, on suppose ici qu'il s'agit de réaliser une collection basée sur une structure de données, comme un tableau, dans laquelle l'accès direct est optimisé. Si la liste n'est pas basée sur une structure de données à accès direct il vaut mieux étendre la classe `AbstractSequentialList` au lieu de celle-ci.

Pour disposer d'une liste immuable, les méthodes `get(int index)` et `size()` sont à écrire. Si on souhaite une liste modifiable, il faut en outre surcharger la méthode `set(int index, Object element)`.

`AbstractSequentialList`. Cette classe abstraite est un début d'implémentation de l'interface `List`, qu'il faut utiliser – de préférence à `AbstractList` – lorsque la collection à définir est basée sur une structure de données à accès (uniquement) séquentiel.

Pour disposer d'une liste il faut étendre cette classe en définissant les méthodes `listIterator()` et `size()`. L'itérateur renvoyé par la méthode `listIterator` est utilisé par les méthodes qui implémentent l'accès direct [55] `get(int index)`, `set(int index, Object element)`, etc.

`LinkedList`. Une implémentation complète de l'interface `List`.

Une `LinkedList` est réputée être implémentée par une liste chaînée bidirectionnelle. En tout cas, toutes les opérations qui ont un sens pour de telles listes existent et ont, en principe, le coût qu'elles auraient dans ce cas.

Il en découle que les `LinkedList` sont particulièrement bien adaptées à la réalisation de piles (*LIFO*), de files d'attente (*FIFO*) et de queues à deux extrémités.

`ArrayList`. La même chose qu'un `Vector`, sauf que ce n'est pas synchronisé (à propos de synchronisation, voyez la section 10.2).

`Vector`. Un `Vector` est une liste implémentée par *un tableau qui prend soin de sa propre taille.*

Lorsqu'on a besoin d'un tableau qui grandit au fur et à mesure du déroulement d'un programme, les `Vector` sont une alternative intéressante aux tableaux ordinaires : l'accès direct aux éléments y est réalisé avec autant d'efficacité et, en plus, le programmeur n'a pas besoin d'estimer *a priori* la taille maximale de la structure, qui se charge de grandir selon les besoins, du moins si les insertions se font à la fin du tableau.

Deux propriétés, dont on peut éventuellement fixer la valeur lors de la construction du `Vector`, en commandent la croissance :
- `capacity` : la taille courante du tableau sous-jacent au `Vector`,
- `capacityIncrement` : le nombre d'unités dont la capacité est augmentée chaque fois que cela est nécessaire.

55. Par conséquent, l'accès au i^{eme} élément a un coût proportionnel à i. C'est la raison pour laquelle on doit prendre `AbstractSequentialList` pour super-classe uniquement lorsque la structure de données sous-jacente ne dispose pas d'accès direct.

De plus, un objet `Vector` est synchronisé [56].

`Stack`. Un objet `Stack` est une pile implémentée par un `Vector`. Cela se manifeste par des opérations spécifiques :
- `push(Object o)` : empiler un objet au sommet de la pile,
- `Object pop()` : dépiler l'objet au sommet de la pile,
- `Object peek()` : obtenir la valeur de l'objet au sommet de la pile sans le dépiler,
- `boolean empty()` : la pile est-elle vide ?
- `int search(Object o)` : rechercher un objet en partant du sommet de la pile.

`AbstractSet`. Cette classe abstraite est un début d'implémentation de l'interface `Set`. Le procédé est le même que pour `AbstractCollection` (voir ci-dessus) sauf que le programmeur doit prendre soin, en définissant les méthodes manquantes, d'obéir aux contraintes supplémentaires des ensembles, surtout pour ce qui est de l'unicité des éléments.

`HashSet`. Une implémentation (complète) de l'interface `Set` réalisée à l'aide d'une table d'*adressage dispersé*, ou *hashcode* (c'est-à-dire une instance de la classe `HashMap`).

C'est très efficace, car les opérations de base (`add`, `remove`, `contains` et `size`) se font en temps constant. Le seul inconvénient [57] est que lors des parcours de la structure les éléments apparaissent dans un ordre imprévisible.

`LinkedHashSet`. Une implémentation de l'interface Set qui utilise une table de *hascode* et, de manière redondante, une liste doublement chaînée.

Cela fonctionne comme un `HashSet`, et avec pratiquement la même efficacité, mais de plus, lors des parcours de la structure ses éléments sont retrouvés dans l'ordre dans lequel ils ont été insérés (c'est la première insertion qui détermine l'ordre).

Un `LinkedHashSet` est une bonne solution lorsqu'on ne peut pas supporter l'ordre généralement chaotique dans lequel sont parcourus les éléments d'un `HashSet` et que, en même temps, on ne peut pas payer le coût additionnel d'un `TreeSet`, voir ci-après.

`TreeSet`. Une implémentation de l'interface `Set` réalisée à l'aide d'un arbre binaire de recherche (c'est-à-dire une instance de la classe `TreeMap`).

Lors des parcours, les éléments de l'ensemble apparaissent en ordre croissant
- soit relativement à un ordre qui leur est propre, et dans ce cas ils doivent implémenter l'interface `Comparable`,
- soit relativement à une relation d'ordre, un objet `Comparator`, fournie lors de la création de la structure.

Les opérations de base (`add`, `remove` et `contains`) sont garanties prendre un temps $O(\log n)$ (n est le nombre d'éléments).

`AbstractMap`. Cette classe abstraite offre un début d'implémentation de l'interface `Map`, destinée à alléger l'effort à fournir pour disposer d'une implémentation complète.

Pour avoir un dictionnaire immuable il suffit de définir une sous classe de `AbstractMap` et d'y redéfinir la méthode `entrySet`. Pour un dictionnaire modifiable il faut en outre redéfinir la méthode `put`.

`HashMap`. Implémentation de l'interface `Map` à l'aide d'une table d'*adressage dispersé* ou *hashcode*. C'est très efficace, car les opérations de base (`get` et `put`) se font en temps constant, du moins si on suppose que la fonction de dispersion répartir uniformément les valeurs des clés.

La table de hashcode est réalisée par un tableau qui, à la manière d'un `Vector`, prend soin de grandir lorsque la table atteint un certain facteur de remplissage.

Le seul inconvénient est que lors des parcours de la structure les éléments apparaissent avec les clés dans un ordre imprévisible.

56. Cela veut dire que des dispositions sont prises pour interdire que deux threads accèdent de manière concurrente à un même `Vector`, ce qui donnerait des résultats imprévisibles si l'un des deux modifiait le vecteur. A ce propos voyez la section 10.2.

57. Ce n'est qu'un inconvénient pratique. En théorie, la notion de « ordre dans lequel les éléments appartiennent à un ensemble » n'a même pas de sens.

`LinkedHashMap`. Implémentation de l'interface `Map` à l'aide d'une table de *hashcode et* d'une liste doublement chaînée redondante.

Grâce à cette liste, lors des parcours de la structure les associations (*clé*,*valeur*) apparaissent de manière ordonnée, en principe dans l'ordre dans lequel elles ont été ajoutées à la table (dit *ordre d'insertion*).

Un argument du constructeur permet de spécifier que l'on souhaite, au lieu de l'ordre d'insertion, un ordre défini par les accès faits aux éléments une fois qu'ils ont été ajoutés : cela consiste à ordonner les éléments en allant du *moins récemment accédé* vers le *plus récemment accédé* (cela permet d'utiliser des objets `LinkedHashMap` pour réaliser des « structures *LRU* »).

`IdentityHashMap`. Cette classe implémente l'interface `Map` comme `HashMap`, sauf qu'elle utilise la relation d'équivalence définie par l'opérateur `==` (égalité « des références ») au lieu de celle définie par le prédicat `equals` (égalité « des valeurs »).

C'est donc une classe très technique, à utiliser avec une extrême prudence.

`WeakHashMap`. Cette classe implémente l'interface `Map` comme `HashMap`, sauf que les associations (*clé*,*valeur*) sont considérées comme étant *faiblement liées* à la table.

Cela veut dire que si la table est le seul objet qui référence une association donnée, alors cette dernière pourra être détruite par le *garbage collector* (cf. section 6.5.4) la prochaine fois que la mémoire viendra à manquer.

L'intérêt pratique est facile à voir : lorsqu'un objet devient sans utilité le programmeur n'a pas à se préoccuper de l'enlever des tables `WeakHashMap` dans lesquelles il a pu être enregistré et qui, sans cela, le feraient passer pour utile et le maintiendraient en vie.

Cela étant, c'est encore une classe très technique, à utiliser avec prudence.

`TreeMap`. Cette classe implémente l'interface *SortedMap* en utilisant un *arbre binaire rouge et noir*[58]. C'est très efficace, puisque les opérations fondamentales (`containsKey`, `get`, `put` et `remove`) sont garanties en $O(\log n)$ (n est le nombre d'associations dans le dictionnaire) et, de plus, lors des parcours de la structure les associations apparaissent avec les clés en ordre croissant :

- soit relativement à un *ordre naturel*, et dans ce cas les clés doivent implémenter l'interface `Comparable`,
- soit relativement à une relation d'ordre (un objet `Comparator`) fournie lors de la création du dictionnaire.

ATTENTION. Une relation d'ordre (méthode `compareTo` dans le cas de l'interface `Comparable`, méthode `compare` dans le cas de l'interface `Comparator`) doit être *compatible avec l'égalité* (cf. section 9.2.4) pour qu'un objet `TreeMap` qui l'utilise soit correct.

9.2.2 Itérateurs

Un objet qui implémente l'interface `Iterator` représente le parcours d'une collection. Il permet donc, au moyen d'opérations indépendantes de la collection particulière dont il s'agit, de se positionner sur le premier élément de la collection et d'obtenir successivement chaque élément de cette dernière.

Ce que « premier élément » veut dire et l'ordre dans lequel les éléments sont obtenus dépendent de la nature de la collection parcourue, comme il a été indiqué dans la section précédente.

Un objet `Iterator` encapsule une *position courante* qui représente l'état du parcours. Les méthodes constitutives de l'interface `Iterator` sont :

`boolean hasNext()` : Prédicat vrai si et seulement si le parcours que l'itérateur représente n'est pas fini (c'est-à-dire si la position courante est valide).

`Object next()` : Renvoie l'objet sur lequel l'itérateur est positionné et fait avancer la position courante.

`void remove()` : Supprime de la collection le dernier élément précédemment renvoyé par l'itérateur. Il faut avoir préalablement appelé `next`, et un appel de `remove` au plus est permis pour un appel de `next`.

58. A propos des arbres rouge et noir, voyez Thomas CORMEN, Charles LEISERSON et Ronald RIVEST, *Introduction à l'algorithmique*, Dunod, 1994.

L'opération `remove` est « facultative ». Plus précisément, elle est liée[59] au caractère modifiable de la collection qui est en train d'être parcourue :
– si la collection est modifiable, cette opération doit être opérationnelle,
– si la collection est immuable, cette opération doit se limiter à lancer une exception `UnsupportedOperationException`.

L'implémentation d'un itérateur dépend de la collection à parcourir ; par conséquent, on construit un itérateur nouveau, positionné au début d'une collection, en le demandant à cette dernière. Cela fonctionne selon le schéma suivant :

```
Collection uneCollection;
...
Iterator unIterateur = uneCollection.iterator();
while ( unIterateur.hasNext() ) {
    Object unObjet = unIterateur.next();
    exploitation de unObjet
}
```

EXEMPLE. Le programme purement démonstratif suivant affiche la liste *triée* des nombres *distincts*[60] obtenus en tirant pseudo-aléatoirement N nombres entiers dans l'intervalle [0, 100 [:

```
...
TreeSet nombres = new TreeSet();
for (int i = 0; i < N; i++) {
    int r = (int) (Math.random() * 100);
    nombres.add(new Integer(r));
}

Iterator it = nombres.iterator();
while (it.hasNext()) {
    Object x = it.next();
    System.out.print(x + " ");
}
...
```

ENUMERATION. Signalons l'existence de l'interface `Enumeration`, sensiblement équivalente à `Iterator`. Sans aller jusqu'à dire que `Enumeration` est désapprouvée (*deprecated*), la documentation officielle indique qu'il faut lui préférer `Iterator`... ce qui n'est pas toujours possible car un certain nombre de services sont réservés aux énumérations et n'ont pas de contrepartie pour les itérateurs (par exemple, la méthode `list` de la classe `Collections`, cf. section 9.2.4).

Les homologues des méthodes `hasNext` et `next`, dans `Enumeration`, sont `hasMoreElements` et `nextElement`. Le schéma, dans le cas des énumérations, est donc celui-ci :

```
Collection uneCollection;
...
Enumeration uneEnumeration = uneCollection.elements();
while ( uneEnumeration.hasMoreElements() ) {
    Object unObjet = uneEnumeration.nextElement();
    exploitation de unObjet
}
```

9.2.3 Quelques méthodes des collections

Voici quelques méthodes des collections parmi les plus importantes (il y en a beaucoup plus que cela) :

COLLECTION. Toutes les collections répondent aux messages suivants :

`boolean isEmpty()` est vrai si et seulement si la collection est vide,

`int size()` renvoie le nombre d'éléments de la collection,

`boolean contains(Object unObjet)` est vrai si et seulement si la collection contient un élément *elt* vérifiant *elt*`.equals(`*unObjet*`)`,

59. Cette question est cruciale pour les collections dont les opérations fondamentales sont définies à partir de l'itérateur correspondant, comme dans le cas de `AbstractSequentialList` (cf. section 9.2.1).
60. On a des nombres distincts parce qu'on les range dans un ensemble, et cette liste est triée parce qu'il s'agit un `TreeSet`.

`void add(Object unObjet)` ajoute l'objet indiqué à la collection ; la place à laquelle l'objet est ajouté dépend du type particulier de la collection,

`void remove(Object unObjet)` si la collection a des éléments *elt* vérifiant *elt*`.equals(`*unObjet*`)` alors l'un d'entre eux est enlevé de la collection ; dans le cas où il y en a plusieurs, savoir lequel est éliminé dépend du type de la collection.

LIST. Les listes ont ceci de plus que les simples collections :

`Object get(int i)` renvoie l'élément qui se trouve à la i^{eme} place,

`void set(int i, Object unObjet)` remplace le i^{eme} élément par l'objet indiqué,

`void add(Object unObjet)` ajoute l'objet indiqué à la fin de la liste,

`void add(int i, Object unObjet)` insère l'objet indiqué à la i^{eme} place (les éléments dont l'indice était $\geq i$ ont, après l'insertion, un indice augmenté de 1).

SET. Les ensembles n'ont pas des méthodes additionnelles, mais un certain comportement est garanti :

`boolean add(Object unObjet)`
si l'ensemble ne contient pas un élément *elt* vérifiant *elt*`.equals(`*unObjet*`)` alors cette méthode ajoute cet objet et renvoie `true` ; sinon, l'ensemble reste inchangé et cette méthode renvoie `false`.

MAP. Les « dictionnaires » sont des collections de paires (*clé*, *valeur*) :

`Object put(Object cle, Object valeur)` ajoute au dictionnaire l'association (*clé*, *valeur*) indiquée ; cette méthode renvoie la précédente *valeur* associée à la *clé* indiqué, ou `null` si une telle clé ne se trouvait pas dans la structure de données,

`Object get(Object cle)` renvoie la *valeur* associée à la *clé* indiquée, ou `null` si une telle clé n'existe pas.

Attention, obtenir `null` comme résultat n'implique pas nécessairement que la clé cherchée n'est pas dans la structure : si le couple (*clé*, `null`) existe on obtient le même résultat (pour savoir si une clé existe il vaut mieux utiliser `containsKey`),

`boolean containsKey(Object cle)` vrai si et seulement si un couple comportant la *clé* indiquée se trouve dans le dictionnaire,

`boolean containsValue(Object valeur)` vrai si et seulement si un ou plusieurs couples comportant la *valeur* indiquée existent dans le dictionnaire.

9.2.4 Algorithmes pour les collections

La classe `Collections` (notez le pluriel [61]) est entièrement faite de méthodes statiques qui opèrent sur des collections, pour les transformer, y effectuer des recherches, en construire de nouvelles, etc. Nous présentons ici quelques méthodes de la classe `Collections` parmi les plus importantes.

A PROPOS DE RELATION D'ORDRE. Certaines des opérations décrites ci-après créent ou exploitent des collections *triées*. Il faut savoir qu'il y a deux manières de spécifier la relation d'ordre par rapport à laquelle une structure est dite triée :

– L'interface `Comparable`. Dire d'une classe qu'elle implémente l'interface `Comparable` c'est dire que ses instances forment un ensemble muni d'une relation d'ordre, donnée par l'unique méthode de cette interface

```
int compareTo(Object o)
```

définie par : `a.compareTo(b)` est négatif, nul ou positif selon que la valeur de `a` est inférieure, égale ou supérieure à celle de `b`.

– L'interface `Comparator`. Les implémentations de cette interface sont des relations [62] qu'on passe comme arguments aux méthodes qui utilisent ou créent des collections ordonnées. Cette interface comporte essentiellement la méthode :

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

61. Il est rare qu'une classe soit désignée par un substantif pluriel ; une classe « ordinaire » porte plutôt le nom singulier qui décrit ses instances. On utilise des pluriels pour désigner des classes entièrement faites d'utilitaires se rapportant à une classe qui aurait le nom en question, au singulier. Composées de variables et méthodes statiques, ces classes ne sont pas destinées à avoir des instances, elles sont des *bibliothèques de fonctions*.

62. Ne pas confondre ces deux interfaces : un `Comparable` représente un objet sur lequel est définie une relation d'ordre, un `Comparator` représente la relation d'ordre elle-même.

définie, comme la précédente, par : `compare(a, b)` est négatif, nul ou positif selon que la valeur de `a` est inférieure, égale ou supérieure à celle de `b`.

Dans un cas comme dans l'autre, ces comparaisons supportent quelques contraintes (ce qui est dit ici sur `compare` s'applique également à `compareTo`) :

- on doit avoir *signe*`(compare(x,y)) == -`*signe*`(compare(y,x))` pour tout couple de valeurs `x` et `y` ;
- la relation est transitive : si `compare(x,y) > 0` et `compare(y,z) > 0` alors `compare(x,z) > 0` ;
- si `compare(x,y) == 0` alors *signe*`(compare(x,z)) ==` *signe*`(compare(y,z))` pour tout `z` ;
- en outre, bien que ce ne soit pas strictement requis, on fait généralement en sorte que

  `(compare(x,y) == 0) == x.equals(y)`

  Lorsque cette dernière contrainte n'est pas satisfaite, la documentation doit comporter l'avertissement *"Note : this comparator imposes orderings that are inconsistent with equals."*

Tri d'une collection.

`static void sort(List uneListe)` trie `uneListe` selon l'*ordre naturel* de ses éléments, ce qui requiert que ces éléments implémentent l'interface `Comparable` et soient comparables deux à deux.

`static void sort(List uneList, Comparator compar)` trie `uneListe` selon l'ordre déterminé par `compar`.

Dans l'un et l'autre cas le tri est garanti *stable* : deux éléments équivalents (pour la relation d'ordre) se retrouvent placés l'un par rapport à l'autre dans la liste triée comme ils étaient dans la liste avant le tri.

`static Comparator reverseOrder()` renvoie un comparateur qui représente l'ordre inverse de l'ordre naturel d'une collection. Par exemple, si l'expression `Collections.sort(`*uneListe*`)` est correcte, alors

`Collections.sort(`*uneListe*`, Collections.reverseOrder());`

est correcte aussi et produit le tri de la liste donnée par « ordre décroissant ».

`static Object max(Collection uneCollection)`

`static Object max(Collection uneCollection, Comparator compar)`

`static Object min(Collection uneCollection)`

`static Object min(Collection uneCollection, Comparator compar)` Ces quatre méthodes recherchent le maximum ou le minimum d'une collection donnée, soit par rapport à l'ordre naturel de la collection (dont les éléments doivent alors implémenter l'interface `Comparable`) soit par rapport à la relation d'ordre représentée par l'argument `compar`.

Recherches dans une collection triée

`static int binarySearch(List uneListe, Object uneValeur)` Recherche la valeur indiquée dans la liste donnée en utilisant l'algorithme de la *recherche binaire* (ou *dichotomique*). La liste doit être triée selon son ordre naturel, ce qui implique que ses éléments implémentent l'interface `Comparable` et sont comparables deux à deux.

`static int binarySearch(List uneListe, Object uneValeur, Comparator compar)` Recherche la valeur indiquée dans la liste donnée en utilisant l'algorithme de la *recherche binaire* (ou *dichotomique*). La liste doit être triée selon l'ordre défini par `compar`.

Dans l'un et l'autre cas, le coût d'une recherche est de l'ordre de $\log n$ si la liste est basée sur un vrai accès direct aux éléments (i.e. le coût de la recherche est $O(\log n)$ si le coût d'un accès est $O(1)$).

`static int indexOfSubList(List grandeListe, List petiteListe)` Recherche le début de la *première* occurrence de `petiteListe` en tant que sous-liste de `grandeListe`. Plus précisément, renvoie la plus petite valeur `i` ≥ 0 telle que

`grandeListe.subList(i, i + petiteListe.size()).equals(petiteListe)`

ou -1 si une telle valeur n'existe pas.

`static int lastIndexOfSubList(List grandeListe, List petiteListe)` Recherche le début de la *dernière* occurrence de `petiteListe` en tant que sous-liste de `grandeListe`. Plus précisément, renvoie la plus grande valeur `i` ≥ 0 telle que

`grandeListe.subList(i, i + petiteListe.size()).equals(petiteListe)`

ou -1 si une telle valeur n'existe pas.

Pour les deux méthodes ci-dessus la documentation indique que la technique employée est celle de la « force brute ». Peut-être ne faut-il pas en attendre une grande efficacité...

Utilitaires de base

`static List nCopies(int n, Object uneValeur)` Renvoie une liste immuable formée de `n` copies de la valeur `uneValeur`. Il s'agit de copie superficielle : l'objet `uneValeur` *n'est pas cloné* pour en avoir `n` exemplaires.

`static void copy(List destin, List source)` Copie les éléments de la liste `source` dans les éléments correspondants de la liste `destin`, qui doit être au moins aussi longue que `source`. Attention, cette opération ne crée pas de structure : les deux listes doivent exister.

`static void fill(List uneListe, Object uneValeur)` Remplace toutes les valeurs de `uneListe` par l'unique valeur `uneValeur`.

`static boolean replaceAll(List uneListe, Object ancienneValeur, Object nouvelleValeur)` Remplace dans `uneListe` tous les éléments égaux à `ancienneValeur` par `nouvelleValeur`. Renvoie `true` si au moins un remplacement a eu lieu, `false` sinon.

`static ArrayList list(Enumeration uneEnumeration)` Construit un objet `ArrayList` contenant les éléments successivement renvoyés par l'énumération indiquée, placés dans l'ordre dans lequel cette dernière les a donnés.

`static void swap(List uneListe, int i, int j)` Échange les valeurs de `uneListe` qui se trouvent aux emplacements `i` et `j`.

`static void reverse(List uneListe)` Renverse l'ordre des éléments de `uneListe`.

`static void rotate(List uneListe, int distance)` Fait « tourner » les éléments de `uneListe` : l'élément qui se trouvait à l'emplacement `i` se trouve, après l'appel de cette méthode, à l'emplacement `(i + distance)` *modulo* `uneListe.size()`

`static void shuffle(List uneListe)` Réarrange pseudo-aléatoirement les éléments de `uneListe`.

`static Set singleton(Object unObjet)` Renvoie un ensemble immuable constitué de l'unique élément représenté par `unObjet`.

`static List singletonList(Object unObjet)` Renvoie une liste immuable constituée de l'unique élément `unObjet`.

`static Map singletonMap(Object clé, Object valeur)` Renvoie une liste associative immuable constituée de l'unique association (`cle`, `valeur`).

La classe `Collections` contient encore deux autres séries de méthodes, pour lesquelles nous renvoyons à la documentation officielle :

- des méthodes `synchronizedCollection(Collection c)`, `synchronizedList(List l)`, `synchronizedMap(Map m)`, etc., pour obtenir une version synchronisée (i.e. pouvant faire face aux accès simultanés faits par plusieurs thread concurrents) d'une collection donnée,
- des méthodes `unmodifiableCollection(Collection c)` `unmodifiableList(List l)`, `unmodifiableMap(Map m)`, etc., pour obtenir une copie immuable d'une collection donnée.

9.2.5 Algorithmes pour les tableaux

La classe `Arrays` (encore un pluriel) est entièrement faite de méthodes statiques qui opèrent sur des tableaux. Voici certaines de ces méthodes parmi les plus importantes :

`static List asList(Object[] unTableau)` Construit et renvoie une liste dont les éléments sont ceux du tableau indiqué.

`static int binarySearch(`***type***`[] unTableau, `***type*** `uneValeur)` Recherche *binaire* (ou *dichotomique*).

type est un des mots `byte`, `short`, `int`, `long`, `float`, `double` ou `char`.

Ces méthodes recherchent `uneValeur` dans le tableau `unTableau`, qui *doit être trié* par rapport à l'ordre naturel du *type* en question. Elles renvoient un entier i qui représente

- si $i \geq 0$, l'emplacement dans le tableau de la valeur recherchée,
- si $i < 0$, la valeur $i = -(j+1)$ où j est l'emplacement dans le tableau dans lequel il faudrait mettre `uneValeur` si on voulait l'insérer tout en gardant trié le tableau.

`static int binarySearch(Object[] unTableau, Object uneValeur)`

`static int binarySearch(Object[] unTableau, Object uneValeur, Comparator compar)`
Même définition que précédemment. Le tableau doit être trié, dans le premier cas par rapport à l'ordre naturel de ses éléments (qui doivent alors implémenter l'interface `Comparable`), dans le second cas par rapport à la relation d'ordre exprimée par l'argument `compar`.

`static boolean equals(`***type***`[] unTableau,` ***type***`[] unAutreTableau)` Égalité de tableaux.

type est un des mots `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char` ou `Object` (autant dire n'importe quel type).

Ce prédicat est vrai si et seulement si les deux tableaux passé comme arguments sont égaux (au sens de `equals`, dans le cas des tableaux d'objets).

`static void fill(`***type***`[] unTableau,` ***type*** `uneValeur)`

`static void fill(`***type***`[] unTableau, int debut, int fin,` ***type*** `uneValeur)` Remplissage d'un tableau avec une même valeur.

type est un des mots `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char` ou `Object`.

La valeur indiquée est affectée à tous les éléments du tableau (premier cas) ou aux éléments de rangs `debut`, `debut`+1, ... `fin` (second cas).

`static void sort(type[] unTableau)`

`static void sort(type[] unTableau, int debut, int fin)` Tri d'un tableau, soit tout entier (premier cas) soit depuis l'élément d'indice `debut` jusqu'à l'élément d'indice `fin` (second cas).

type est un des mots `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char` ou `Object` (autant dire n'importe quel type).

Le tri se fait par rapport à l'*ordre naturel* des éléments du tableau ce qui, dans le cas où *type* est `Object`, oblige ces éléments à être d'un type qui implémente l'interface `Comparable`.

La méthode de tri employée est une variante élaborée du l'algorithme du tri rapide (*quicksort*) qui, nous dit-on, offre un coût de $n \times \log n$ même dans un certain nombre de cas pourris où le *quicksort* de base serait quadratique.

`static void sort(Object[] a, int fromIndex, int toIndex, Comparator compar)` Même chose que ci-dessus, mais relativement à la relation d'ordre définie par l'argument `compar`.

9.3 Réflexion et manipulation des types

La classe `java.lang.Class` et les classes du paquet `java.lang.reflect` (aux noms évocateurs, comme `Field`, `Method`, `Constructor`, `Array`, etc.) offrent la possibilité de pratiquer une certaine *introspection* : un objet peut inspecter une classe, éventuellement la sienne propre, accéder à la liste de ses membres, appeler une méthode dont le nom n'est connu qu'à l'exécution, etc.

Une instance de la classe `java.lang.Class` représente un type (c'est-à-dire un type primitif ou une classe). Supposons que nous ayons une variable déclarée ainsi :

```
Class uneClasse;
```

Nous avons plusieurs manières de lui donner une valeur :

- une *classe-enveloppe d'un type primitif* (`Integer`, `Double`, etc.) possède une constante de classe `TYPE` qui représente le type primitif en question ; de plus, l'expression *type*`.class` a le même effet. Ainsi, les deux expressions suivantes affectent le type `int` à la variable `uneClasse` :

```
uneClasse = Integer.TYPE;
uneClasse = int.class;
```

- si *uneClasse* est l'identificateur d'une classe, alors l'expression *uneClasse*`.class` a pour valeur la classe en question[63]. L'exemple suivant affecte le type `java.math.BigInteger` à la variable `uneClasse` :

```
uneClasse = java.math.BigInteger.class;
```

63. Notez la différence entre `TYPE` et `class` pour les classes-enveloppes des types primitifs : `Integer.TYPE` est le type `int`, tandis que `Integer.class` est le type `java.lang.Integer`.

– on peut aussi demander le chargement de la classe, à partir de son nom complètement spécifié. C'est un procédé plus onéreux que les précédents, où la plupart du travail de chargement (dont la détection d'éventuelles erreurs dans les noms des classes) était fait durant la compilation alors que, dans le cas présent, il sera fait pendant l'exécution. L'exemple suivant affecte encore le type `java.math.BigInteger` à la variable `uneClasse` :

```
uneClasse = Class.forName("java.math.BigInteger");
```

– enfin, le moyen le plus naturel est de demander à un objet quelle est sa classe. L'exemple suivant affecte encore le type `java.math.BigInteger` à la variable `uneClasse` :

```
Object unObjet = new BigInteger("92233720368547758079223372036854775807");
...
uneClasse = unObjet.getClass();
```

Pour survoler cette question assez pointue, voici un exemple qui illustre quelques unes des possibilités de la classe `Class` et du paquet `reflect`.

La méthode `demoReflexion` ci-dessous, purement démonstrative, prend deux objets quelconques `a` et `b` et appelle successivement *toutes* les méthodes qui peuvent être appelées sur `a` avec `b` pour argument, c'est-à-dire les méthodes d'instance de la classe de `a` qui ont un unique argument de type la classe de `b` :

```
import java.lang.reflect.Method;
import java.math.BigInteger;

public class DemoReflexion {

    static void demoReflexion(Object a, Object b) {
        try {
            Class classe = a.getClass();
            Method[] methodes = classe.getMethods();

            for (int i = 0; i < methodes.length; i++) {
                Method methode = methodes[i];
                Class[] params = methode.getParameterTypes();
                if (params.length == 1 && params[0] == b.getClass()) {
                    Object r = methode.invoke(a, new Object[] { b });
                    System.out.println(
                        a + "." + methode.getName() + "(" + b + ") = " + r);
                }
            }
        } catch (Exception exc) {
            System.out.println("Problème : " + exc);
        }
    }

    public static void main(String[] args) {
        demoReflexion(new BigInteger("1001"), new BigInteger("1003"));
    }
}
```

Le programme précédent affiche la sortie

```
1001.compareTo(1003) = -1
1001.min(1003) = 1001
1001.add(1003) = 2004
...
1001.gcd(1003) = 1
1001.mod(1003) = 1001
1001.modInverse(1003) = 501
```


9.4 Entrées-sorties

NOTE IMPORTANTE. Java 5 a introduit des outils simples et pratiques pour effectuer la lecture et l'écriture de données formatées. Ils sont expliquée dans la section 12.3.4 (page 159).

9.4.1 Les classes de flux

Java fournit un nombre impressionnant de classes prenant en charge les opérations d'entrée sortie. Voici une présentation très succincte des principales (la marge laissée à gauche reflète la relation d'héritage). Plusieurs exemples d'utilisation de ces classes sont donnés dans la section 9.4.2 :

FLUX D'OCTETS EN ENTRÉE

`InputStream`	Cette classe abstraite est la super-classe de toutes les classes qui représentent des flux d'octets en entrée. Comporte une unique méthode, `int read()`, dont chaque appel renvoie un octet extrait du flux.
`FileInputStream`	Un flux qui obtient les données lues, des octets, à partir d'un fichier. Construction à partir d'un `String` (le nom du fichier), d'un `File` ou d'un `FileDescriptor`.
`ByteArrayInputStream`	Un flux qui obtient les données lues, des octets, à partir d'un tableau d'octets indiqué lors de la construction.
`PipedInputStream`	Un flux qui obtient les données lues, des octets, à partir d'un `PipedOutputStream` auquel il est connecté. Ce deuxième flux doit appartenir à un thread distinct, sous peine de blocage. Construction à partir d'un `PipedOutputStream` ou à partir de de rien.
`FilterInputStream`	Un flux qui obtient les données lues, des octets, à partir d'un autre `InputStream`, auquel il ajoute des fonctionnalités.
`BufferedInputStream`	Ajoute aux fonctionnalités d'un autre `InputStream` la mise en tampon des octets lus. Construction à partir d'un `InputStream`.
`PushbackInputStream`	Ajoute aux fonctionnalités d'un autre `InputStream` la possibilité de « délire » un octet (un octet « délu » est retrouvé lors d'une lecture ultérieure). Construction à partir d'un `InputStream`.
`DataInputStream`	Un objet de cette classe obtient de l'`InputStream` sous-jacent des données appartenant aux types primitifs de Java. Voir `DataOutputStream`, plus loin. Construction à partir d'un `InputStream`.
`ObjectInputStream`	Un flux capable de dé-sérialiser les objets préalablement sérialisés dans un `ObjectOutputStream`. Voir les sections « sérialisation » et `ObjectOutputStream`. Construction à partir d'un `InputStream`.
`SequenceInputStream`	Un flux qui obtient les données lues, des octets, à partir de la concaténation logique d'une suite d'objets `InputStream`. Construction à partir de deux `InputStream` ou d'une énumeration dont les éléments sont des `InputStream`.

En résumé :
- les sous-classes directes de `InputStream` se distinguent par le type de la source des données lues,
- les sous classes de `FilterInputStream` sont diverses sortes de couches ajoutées par-dessus un autre flux.

FLUX D'OCTETS EN SORTIE

`OutputStream`	Classe abstraite, super-classe de toutes les classes qui représentent

	des flux (d'octets) en sortie. Un tel flux reçoit des octets et les envoie vers un certain puits (« puits » s'oppose à « source ») Comporte une méthode abstraite : `void write(int b)`, qui écrit un octet.
`FileOutputStream`	Un flux de sortie associé à un fichier du système sous-jacent. Construction à partir d'un `String`, d'un `File` ou d'un `FileDescriptor`.
`ByteArrayOutputStream`	Un flux de sortie dans lequel les données sont rangées dans un tableau d'octets. Construction à partir de rien.
`PipedOutputStream`	Voir `PipedInputStream`. Construction à partir d'un `PipedInputStream` ou à partir de rien.
`FilterOutputStream`	Un flux qui envoie les données à un autre flux (donné lors de la construction) après leur avoir appliqué un certain traitement. Construction à partir d'un `OutputStream`.
`BufferedOutputStream`	Un flux de sortie qui regroupe les octets logiquement écrits avant de provoquer un appel du système sous-jacent en vue d'une écriture physique. Construction à partir d'un `OutputStream`.
`DataOutputStream`	Un flux de sortie destiné à l'écriture de données des types primitifs de Java. Les données ne sont pas formatées (traduites en textes lisibles par un humain) mais elles sont écrites de manière portable. Construction à partir d'un `OutputStream`.
`PrintStream`	Un flux destiné à l'écriture de toutes sortes de données, sous une forme lisible par un humain. Ces flux ne gênèrent jamais d'exception ; à la place, ils positionnent une variable privée qui peut être testée à l'aide de la méthode `checkError`. De plus, ces flux gèrent le vidage (flushing) du tampon dans certaines circonstances : appel d'une méthode `println`, écriture d'un caractère `'\n'`, etc. Construction à partir d'un `OutputStream`.
`ObjectOutputStream`	Un flux de sortie pour écrire des données de types primitifs et des objets Java dans un `OutputStream`. Cela s'appelle la *sérialisation* des données. Les objets peuvent ensuite être lus et reconstitués en utilisant un `ObjectInputStream`. Construction à partir d'un `OutputStream`.
	Seuls les objets qui implémentent l'interface `Serializable` peuvent être écrits dans un flux `ObjectOutputStream`. La classe (nom et signature) de chaque objet est codée, ainsi que les valeurs des variables d'instance (sauf les variables déclarées `transient`).
	La sérialisation d'un objet implique la sérialisation des objets que celui-ci référence, il s'agit donc de mettre en séquence les nœuds d'un graphe, qui peut être cyclique. C'est pourquoi cette opération, assez complexe, est très utile.

Flux de caractères en entrée

Les flux de caractères, en entrée et en sortie. Ces flux sont comme les flux d'octets, mais l'information élémentaire y est le caractère (Java utilisant le codage Unicode, un caractère n'est pas la même chose qu'un octet).

`Reader` — Classe abstraite, super-classe de toutes les classes qui lisent des flux de caractères.
Deux méthodes abstraites : `int read(char[], int, int)` et `void close()`.

`BufferedReader` — Les objets de cette classe améliorent l'efficacité des opérations d'entrée en « bufférisant » les lectures sur un flux sous-jacent (un objet `Reader` donné en argument du constructeur).
Construction à partir d'un `Reader`.

`LineNumberReader` — Un flux d'entrée bufférisé qui garde trace des numéros des lignes lues.
(Une ligne est une suite de caractères terminée par `'\r'`, `'\n'` ou `'\r\n'`).
Construction à partir d'un `Reader`.

`CharArrayReader` — Un flux qui obtient les caractères lus à partir d'un tableau de caractères interne, qu'on indique lors de la construction du flux.
Construction à partir d'un `char[]`.

`InputStreamReader` — Un flux de caractères qui obtient les données dans un flux d'octets (un objet `InputStream` donné en argument du constructeur).
Un tel objet assure donc le travail de conversion des octets en caractères.
Construction à partir d'un `InputStream`.

`FileReader` — (Classe de confort) Un flux d'entrée de caractères qui obtient ses données dans un fichier.
On suppose que le codage des caractères par défaut et la taille par défaut du tampon sont adéquats. (Si tel n'est pas le cas il vaut mieux construire un `InputStreamReader` sur un `FileInputStream`)
Construction à partir d'un `String`, d'un `File` ou d'un `FileDescriptor`.

`FilterReader` — Classe abstraite pour la lecture de flux filtrés (?)
Possède, à titre de membre, un objet `Reader` dont il filtre les caractères lus.

`PushbackReader` — Flux d'entrée de caractères ayant la possibilité de « délire » un ou plusieurs caractères.
Construction à partir d'un `Reader`.

`PipedReader` — L'équivalent, pour des flux de caractères, d'un `PipedInputStream`.
Construction à partir d'un `PipedWriter` ou à partir de rien.

`StringReader` — Un flux de caractères dont la source est un objet `String`.
Construction à partir d'un `String`.

Flux de caractères en sortie

`Writer` — Classe abstraite, super-classe de toutes les classes qui écrivent des flux de caractères.
Méthodes abstraites : `void write(char[], int, int)`, `void flush()` et `void close()`.

`BufferedWriter` — Un flux de sortie dans lequel les caractères logiquement écrits sont groupés pour diminuer le nombre effectif d'écritures physiques.
Construction à partir d'un `Writer`.

`CharArrayWriter` — Un flux qui met les caractères produits dans un tableau de

	caractères. Construction à partir de rien.
`OutputStreamWriter`	Un flux qui met les caractères écrits dans un `OutputStream` préalablement construit. Un tel objet prend donc en charge la conversion des caractères en des octets. Construction à partir d'un `OutputStream`.
`FileWriter`	(Classe de confort) Un flux qui envoie les caractères écrits dans un fichier. Construction à partir d'un `String`, d'un `File` ou d'un `FileDescriptor`.
`FilterWriter`	Classe abstraite pour la définition de flux filtrés d'écriture de caractères.
`PipedWriter`	L'équivalent, pour un flux de caractères, d'un `PipedOutputStream`. Voir `PipedReader`. Construction à partir d'un `PipedReader` ou à partir de rien.
`StringWriter`	Un flux qui met les caractères écrits dans un objet `StringBuffer`, lequel peut être utilisé pour construire un `String`. Construction à partir de rien.
`PrintWriter`	Un flux de caractères destiné à l'envoi de représentations formatées de données dans un flux préalablement construit. Un `PrintWriter` se construit au-dessus d'un `OutputStream` ou d'un Writer préalablement créé.

Autres classes

`File`	Un objet File est la représentation abstraite d'un fichier, d'un répertoire ou, plus généralement, d'un chemin du système de fichiers sous-jacent.
`FileDescriptor`	Un `FileDescriptor` est un objet opaque qui représente une entité spécifique de la machine sous-jacente : un fichier ouvert, un socket ouvert ou toute autre source ou puits d'octets. La principale utilité d'un tel objet est la création d'un `FileInputStream` ou d'un `FileOutputStream`.
`RandomAccessFile`	Un objet `RandomAccessFile` représente un fichier supportant l'accès relatif. Cette classe implémente les interfaces `DataInput` et `DataOutput`, c'est-à-dire qu'elle supporte les opérations de lecture et d'écriture, ainsi que des opérations pour positionner et obtenir la valeur d'une certaine *position courante* dans le fichier
`StreamTokenizer`	Un objet de cette classe est un analyseur qui reconnaît des unités lexicales formées d'octets ou de caractères provenant d'une source de données. Les unités sont de quatre types : `TT_NUMBER` (la valeur est un double), `TT_WORD`, `TT_EOF` et (éventuellement) `TT_EOL`. Dans le même ordre d'idées, voir aussi `StringTokenizer`.

9.4.2 Exemples

1. Ecriture et lecture dans un fichier « binaire ». Les deux méthodes de la classe `FichierBinaire` suivante permettent d'enregistrer un tableau de nombres flottants (des `double`) dans un fichier, ou bien de reconstituer un tel tableau à partir de ses valeurs préalablement enregistrées. Le fichier en question est formé d'un nombre entier n suivi de n nombres flottants en double précision :

```
class FichierBinaire {
    public void sauver(double nombres[], String nomFichier) throws IOException {
        FileOutputStream fichier = new FileOutputStream(nomFichier);
        DataOutputStream donnees = new DataOutputStream(fichier);
        int combien = nombres.length;
        donnees.writeInt(combien);
        for (int i = 0; i < combien; i++)
            donnees.writeDouble(nombres[i]);
        donnees.close();
    }
    public double[] restaurer(String nomFichier) throws IOException {
        FileInputStream fichier = new FileInputStream(nomFichier);
        DataInputStream donnees = new DataInputStream(fichier);
        int combien = donnees.readInt();
        double[] result = new double[combien];
        for (int i = 0; i < combien; i++)
            result[i] = donnees.readDouble();
        donnees.close();
        return result;
    }
}
```

2. Écriture et lecture dans un fichier de texte. Même sorte de travail qu'à l'exemple précédente, mais le fichier est maintenant de texte. C'est moins efficace, puisque les données sont exprimées sous une forme textuelle (donc du travail supplémentaire lors de l'écriture et lors de la lecture), mais cela permet l'exploitation ou la modification du fichier par des outils généraux (imprimantes, éditeurs de texte, etc.). Pour faciliter l'éventuelle exploitation « humaine » du fichier, devant chaque donnée n_i on écrit son rang i :

```
class FichierDeTexte {
    public void sauver(double nombres[], String nomFichier) throws IOException {
        FileOutputStream fichier = new FileOutputStream(nomFichier);
        PrintWriter donnees = new PrintWriter(fichier);

        int combien = nombres.length;
        donnees.println(combien);
        for (int i = 0; i < combien; i++)
            donnees.println(i + " " + nombres[i]);
        donnees.close();
    }
    public double[] restaurer(String nomFichier) throws IOException {
        FileInputStream fichier = new FileInputStream(nomFichier);
        InputStreamReader adaptateur = new InputStreamReader(fichier);
        LineNumberReader donnees = new LineNumberReader(adaptateur);

        String ligne = donnees.readLine();
        int combien = Integer.parseInt(ligne);
        double[] result = new double[combien];
        for (int i = 0; i < combien; i++) {
            ligne = donnees.readLine();
            StringTokenizer unites = new StringTokenizer(ligne);
            unites.nextToken();               // on ignore le rang i
            result[i] = Double.parseDouble(unites.nextToken());
            }
        if (donnees.readLine() != null)
            throw IOException("Il y a plus de données que prévu");
        donnees.close();
        return result;
    }
```

3. Lecture sur l'entrée standard. La classe `Est` (comme Entree STandard) offre un petit nombre de méthodes statiques destinées à effectuer simplement des opérations de lecture à l'entrée standard (souvent le clavier) :

– `int lireInt()` : lecture d'un nombre entier,

– `double lireDouble()` : lecture d'un nombre flottant,
– `char lireChar()` : lecture d'un caractère,
– `String lireString()` : lecture de toute une ligne,
– `void viderTampon()` : remise à zéro du tampon d'entrée.

NOTE JAVA 5. La classe `Est` montrée ici reste un exemple intéressant d'utilisation de la bibliothèque des entrées-sorties, mais il faut savoir qu'à partir de la version 5 de Java le service rendu par cette classe est pris en charge – de manière un peu plus professionnelle – par la classe `java.util.Scanner` de la bibliothèque standard (cf. § 12.3.4, page 159).

Exemple d'utilisation de notre classe `Est` :

```
...
int numero;
String denomination;
double prix;
char encore;
...
do {
    System.out.print("numero? ");
    numero = Est.lireInt();
    System.out.print("dénomination? ");
    Est.viderTampon();
    denomination = Est.lireString();
    System.out.print("prix? ");
    prix = Est.lireDouble();
    System.out.print("Encore (o/n)? ");
    Est.viderTampon();
    encore = Est.lireChar();
} while (encore == 'o');
...
```

Voici la classe `Est` :

```
import java.io.*;
import java.util.StringTokenizer;

public class Est {

    public static int lireInt() {
        return Integer.parseInt(uniteSuivante());
    }

    public static double lireDouble() {
        return Double.parseDouble(uniteSuivante());
    }

    public static char lireChar() {
        if (tampon == null)
            tampon = lireLigne();
        if (tampon.length() > 0) {
            char res = tampon.charAt(0);
            tampon = tampon.substring(1);
            return res;
        } else {                            // Si tous les caractères ordinaires
            tampon = null;                  // de la ligne courante ont été lus,
            return '\n';                    // le caractère renvoyé est '\n'
        }
    }

    public static String lireString() {
        if (tampon == null)
            return lireLigne();
```

```
            else {
                String res = tampon;
                tampon = null;
                return res;
            }
        }

        public static void viderTampon() {
            tampon = null;
        }

        private static String uniteSuivante() {
            if (tampon == null)
                tampon = lireLigne();

            StringTokenizer unites = new StringTokenizer(tampon, limites);
            while ( ! unites.hasMoreTokens()) {
                tampon = lireLigne();
                unites = new StringTokenizer(tampon, limites);
            }

            String unite = unites.nextToken();
            tampon = tampon.substring(tampon.indexOf(unite) + unite.length());
            return unite;
        }

        private static String lireLigne() {
            try {                               // Les IOException sont transformées
                return entree.readLine();       // en RuntimeException (non contrôlées)
            } catch (IOException ioe) {
                throw new RuntimeException("Erreur console [" + ioe.getMessage() +"]");
            }
        }

        private static BufferedReader entree =
                            new BufferedReader(new InputStreamReader(System.in));
        private static String tampon = null;
        private static final String limites = " \t\n,;:/?!%#$()[]{}";
    }
```

9.4.3 Analyse lexicale

L'*analyse lexicale* est la transformation d'une suite de caractères en une suite d'*unités lexicales* ou *tokens* – des « mots » – obéissant à une *grammaire lexicale* souvent définie par un ensemble d'expressions régulières.

En Java, l'analyse lexicale la plus basique est prise en charge par les objets `StringTokenizer` et `StreamTokenizer`.

A partir de la version 1.4, des analyseurs plus sophistiqués peuvent être réalisés à l'aide des classes de manipulation d'expressions régulières, `java.util.regex.Pattern` et `java.util.regex.Matcher`. A ce sujet, voyez la section 9.5.

D'autre part, Java 5 introduit la classe `java.util.Scanner`, plus puissante que `StreamTokenizer` mais plus simple d'emploi que le couple `Pattern` et `Matcher`.

Pour illustrer l'emploi des objets `StreamTokenizer`, ou « analyseurs lexicaux de flux », voici un exemple classique (classique dans certains milieux !) : le programme suivant reconnaît et évalue des expressions arithmétiques lues à l'entrée standard, formées avec des nombres, les quatre opérations et les parenthèses, et terminées par le caractère '='.

Il est intéressant d'observer l'empilement de flux d'entrée réalisé dans le constructeur de la classe `Calculateur` : à la base, il y a `System.in`, qui est un flux d'octets (un `InputStream`) ; « autour » de ce flux on a mis `adaptateur`, qui est un flux de caractères (un `InputStreamReader`, donc un `Reader`) ; « autour » de ce dernier, on a construit `unites`, un flux d'unités lexicales (un `StreamTokenizer`).


```
class Calculateur {
    private StreamTokenizer unites;

    public Calculateur() throws IOException {
        InputStreamReader adaptateur = new InputStreamReader(System.in);
        unites = new StreamTokenizer(adaptateur);
    }



    public double unCalcul() throws Exception {
        unites.nextToken();
        double result = expression();
        if (unites.ttype != '=')
            throw new ErreurSyntaxe("'=' attendu à la fin de l'expression");
        return result;
    }



    private double expression() throws IOException, ErreurSyntaxe {
        double result = terme();
        int oper = unites.ttype;
        while (oper == '+' || oper == '-') {
            unites.nextToken();
            double tmp = terme();
            if (oper == '+')
                result += tmp;
            else
                result -= tmp;
            oper = unites.ttype;
        }
        return result;
    }



    private double terme() throws IOException, ErreurSyntaxe {
        double result = facteur();
        int oper = unites.ttype;
        while (oper == '*' || oper == '/') {
            unites.nextToken();
            double tmp = facteur();
            if (oper == '*')
                result *= tmp;
            else
                result /= tmp;
            oper = unites.ttype;
        }
        return result;
    }



    private double facteur() throws IOException, ErreurSyntaxe {
        double result = 0;



        if (unites.ttype == StreamTokenizer.TT_NUMBER) {
            result = unites.nval;
            unites.nextToken();
        }
```

```
        else if (unites.ttype == '(') {
            unites.nextToken();
            result = expression();
            if (unites.ttype != ')')
                throw new ErreurSyntaxe("')' attendue " + unites.ttype);
            unites.nextToken();
        }
        else
            throw new ErreurSyntaxe("nombre ou '(' attendu " + unites.ttype);
        return result;
    }
}

class ErreurSyntaxe extends Exception {
    ErreurSyntaxe(String message) {
        super(message);
    }
}
```

9.4.4 Mise en forme de données

Le paquet `java.text` est fait de classes et d'interfaces pour la mise en forme de textes, dates, nombres, etc. Les plus utiles, pour nous :

`Format`	Classe abstraite concernant la mise en forme des données sensibles aux particularités locales (nombres, dates, etc.).
`NumberFormat`	Classe abstraite, super-classe des classes concernant la mise en forme et la reconnaissance des nombres (aussi bien les types primitifs que leurs classes-enveloppes).
`DecimalFormat`	Classe concrète, prenant en charge l'écriture en base 10 des nombres.
`DecimalFormatSymbols`	Un objet de cette classe représente l'ensemble des symboles qui interviennent dans le formatage d'un nombre (séparateur décimal, séparateur de paquets de chiffres, etc.). Le constructeur par défaut de cette classe initialise un objet avec les valeurs locales de tous ces symboles.

Examinons quelques exemples :

Exemple 1. Écriture d'un nombre en utilisant le *format local* (i.e. le format en vigueur à l'endroit où le programme est exécuté) :

```
import java.text.NumberFormat;

public class AProposDeFormat {
    static public void main(String[] args) {
        NumberFormat formateur = NumberFormat.getInstance();

        double x = 1234567.23456789;
        System.out.println("format \"brut\"           : " + x);

        String s = formateur.format(x);
        System.out.println("format local par défaut : " + s);
        ...
```

Affichage obtenu (en France) :

```
format "brut"           : 1234567.23456789
format local par défaut : 1 234 567,235
```

Comme on le voit ci-dessus, il se peut que des nombres formatés en accord avec les particularités locales ne puissent pas être donnés à « relire » aux méthodes basiques de Java, comme `Integer.parseInt` ou

`Double.parseDouble`. Pour reconstruire les nombres que ces textes expriment on doit employer les méthodes correspondantes de la classe `NumberFormat`. Voici une possible continuation de l'exemple précédent :

```
            ...
            try {
                Number n = formateur.parse(s);
                x = n.doubleValue();
            } catch (ParseException exc) {
                System.out.println("Conversion impossible : " + exc.getMessage());
            }
            System.out.println("valeur \"relue\"          : " + x);
        }
    }
```

Affichage obtenu maintenant :

```
format "brut"           : 1234567.23456789
format local par défaut : 1 234 567,235
valeur "relue"          : 1234567.235
```

Exemple 2. Utilisation de ces classes pour obtenir une mise en forme précise qui ne correspond pas forcément aux particularités locales. Dans l'exemple suivant on écrit les nombres avec exactement deux décimales, la virgule décimale étant représentée par un point et les chiffres avant la virgule groupés par paquets de trois signalés par des apostrophes :

```
public class Formatage {
    static public void main(String[] args) {
        DecimalFormatSymbols symboles = new DecimalFormatSymbols();
        symboles.setDecimalSeparator('.');
        symboles.setGroupingSeparator('\'');
        DecimalFormat formateur = new DecimalFormat("###,###,###.00", symboles);

        double d = 1.5;
        System.out.println(formateur.format(d));

        d = 0.00123;
        System.out.println(formateur.format(d));

        int i = 12345678;
        System.out.println(formateur.format(i));
    }
}
```

l'affichage est ici :

```
1.50
.00
12'345'678.00
```

Variante, avec :

```
DecimalFormat formateur = new DecimalFormat("000,000,000.00", symboles);
```

l'affichage aurait été :

```
000'000'001.50
000'000'000.00
012'345'678.00
```

Exemple 3. Des formats spécifiques comme les précédents peuvent aussi être composés à l'aide de méthodes « `set`... » de la classe `DecimalFormat` :

```
public class Formatage {
    public static void main(String[] args) {
        DecimalFormat formateur = (DecimalFormat) NumberFormat.getInstance();

            // Modifier ce format afin qu'il ressemble à 999.99[99]
        DecimalFormatSymbols symboles = new DecimalFormatSymbols();
        symboles.setDecimalSeparator('.');
        formateur.setDecimalFormatSymbols(symboles);
        formateur.setMinimumIntegerDigits(3);
        formateur.setMinimumFractionDigits(2);
        formateur.setMaximumFractionDigits(4);

        double d = 1.5;
        System.out.println(d + " se formate en " + formateur.format(d));
        d = 1.23456789;
        System.out.println(d + " se formate en " + formateur.format(d));
    }
}
```

Exécution :

```
1.5 se formate en 001.50
1.23456789 se formate en 001.2346
```

EXEMPLE 4. La classe `Fomat` et ses voisines n'offrent pas un moyen simple d'effectuer un cadrage à droite des nombres ou, plus exactement, un cadrage calculé à partir de la position de la virgule. De telles mises en forme sont nécessaires pour constituer des colonnes de nombres correctement alignés.

Par exemple, proposons-nous d'afficher une table de conversion en degrés Celsius d'une série de températures exprimées en degrés Fahrenheit :

```
public class ConversionTemperature {
    public static void main(String[] args) {
        NumberFormat formateur = new DecimalFormat("##.###");

        for (int f = -40; f <= 120; f += 20) {
            double c = (f - 32) / 1.8;
            System.out.println(f + " " + formateur.format(c));
        }
    }
}
```

L'affichage obtenu n'est pas très beau :

```
-40 -40
-20 -28,889
0 -17,778
20 -6,667
40 4,444
60 15,556
80 26,667
100 37,778
120 48,889
```

Voici une deuxième version de ce programme, dans laquelle un objet `FieldPosition` est utilisé pour connaître, dans l'expression formatée d'un nombre, l'indice du caractère sur lequel se termine sa partie entière (c'est-à-dire, selon le cas, soit la fin du nombre, soit la position de la virgule). Cet indice permet de calculer le nombre de blancs qu'il faut ajouter à la gauche du nombre pour le cadrer proprement :


```
public class ConversionTemperature {
    public static void main(String[] args) {
        NumberFormat formateur = new DecimalFormat("##.###");
        FieldPosition pos = new FieldPosition(NumberFormat.INTEGER_FIELD);

        for (int c = -40; c <= 120; c += 20) {
            double f = (c - 32) / 1.8;

            String sc = formateur.format(c, new StringBuffer(), pos).toString();
            sc = ajoutEspaces(4 - pos.getEndIndex(), sc);

            String sf = formateur.format(f, new StringBuffer(), pos).toString();
            sf = ajoutEspaces(4 - pos.getEndIndex(), sf);

            System.out.println(sc + " " + sf);
        }
    }

    private static String ajoutEspaces(int n, String s) {
        return "                          ".substring(0, n) + s;
    }
}
```

L'affichage obtenu maintenant est nettement mieux aligné :

```
 -40  -40
 -20  -28,889
   0  -17,778
  20   -6,667
  40    4,444
  60   15,556
  80   26,667
 100   37,778
 120   48,889
```

9.4.5 Représentation des dates

La manipulation et l'affichage des dates sont traités en Java avec beaucoup de soin (portabilité oblige !), mais la question peut sembler un peu embrouillée car elle est répartie sur trois classes distinctes de manière un peu arbitraire :

`java.util.Date` - Un objet de cette classe encapsule un instant dans le temps (représenté par le nombre de millisecondes écoulées entre le 1^{er} janvier 1970, à `0:00:00` heures GMT, et cet instant).

D'autre part, ces objets ont des méthodes pour le calcul de l'année, du mois, du jour, etc., mais elles sont désapprouvées, ce travail doit désormais être donné à faire aux objets `Calendar`.

`java.util.GregorianCalendar` - Cette classe est une sous-classe de la classe abstraite `Calendar`. Ses instances représentent des dates, décomposées en plusieurs nombres qui expriment l'année, le mois, le jour dans le mois, dans l'année et dans la semaine, l'heure, les minutes, etc.

`java.text.DateFormat` - Les instances de cette classe, fortement dépendante des particularités locales, s'occupent de l'expression textuelle des dates.

Exemple 1. Voici comment mesurer le temps que prend l'exécution d'un certain programme (attention, il s'agit de « temps écoulé », qui comprend donc le temps pris par d'éventuelles autres tâches qui ont été entrelacées avec celle dont on cherche à mesurer la durée) :

```
...
void unCalcul() {
    Date d0 = new Date();

    un calcul prenant du temps...

    Date d1 = new Date();
    System.out.println("Temps écoulé: "
        + (d1.getTime() - d0.getTime()) / 1000.0 + " secondes");
```

```
    }
    ...
```

EXEMPLE 2. Un programme pour savoir en quel jour de la semaine tombe une date donnée :

```
class Jour {
    public static void main(String[] args) {
        if (args.length < 3)
            System.out.println("Emploi: java Jour <jour> <mois> <annee>");
        else {
            int j = Integer.parseInt(args[0]);
            int m = Integer.parseInt(args[1]);
            int a = Integer.parseInt(args[2]);
            GregorianCalendar c = new GregorianCalendar(a, m - 1, j);
            int s = c.get(GregorianCalendar.DAY_OF_WEEK);
            String[] w = { "dimanche", "lundi", "mardi", "mercredi",
                                                    "jeudi", "vendredi", "samedi" };
            System.out.println(w[s - 1]);
        }
    }
}
```

EXEMPLE 3. Affichage personnalisé de la date. Chaque fois qu'on demande à un objet `Maintenant` de se convertir en chaîne de caractères, il donne une expression de la date et heure courantes :

```
class Maintenant {
    private static String[] mois = { "janvier", "février", "mars", ... "décembre" };
    private static String[] jSem = { "dimanche", "lundi", "mardi", ... "samedi" };
    public String toString() {
        GregorianCalendar c = new GregorianCalendar();
        return jSem[c.get(c.DAY_OF_WEEK) - 1] + " " + c.get(c.DAY_OF_MONTH)
            + " " + mois[c.get(c.MONTH) - 1] + " " + c.get(c.YEAR)
            + " à " + c.get(c.HOUR) + " heures " + c.get(c.MINUTE);
    }
    public static void main(String[] args) {
        System.out.println("Aujourd'hui: " + new Maintenant());
    }
}
```

EXEMPLE 4. La même chose, à l'aide d'un objet `DateFormat` :

```
class Maintenant {
    DateFormat df = DateFormat.getDateTimeInstance(DateFormat.FULL,
        DateFormat.MEDIUM, Locale.FRANCE);
    public String toString() {
        return df.format(new Date());
    }
    public static void main(String[] args) {
        System.out.println("Aujourd'hui: " + new Maintenant());
    }
}
```

9.4.6 La sérialisation

La sérialisation est un mécanisme très puissant, très utile et très simple d'emploi pour sauvegarder des objets entiers dans des fichiers et de les restaurer ultérieurement. Les points clés en sont :

- pour que les instances d'une classe puissent être sérialisées il doit être dit que cette classe implémente l'interface `Serializable`, une interface vide par laquelle le programmeur exprime qu'il n'y a pas d'opposition à ce que les objets en question soient conservés dans un fichier,
- la sérialisation d'un objet implique la sauvegarde des valeurs de toutes ses variables d'instance, sauf celles déclarées `transient` (transitoires),
- la sérialisation d'un objet implique en principe celle des objets qu'il référence (ses objets membres) qui doivent donc être également sérialisables,
- le couple sérialisation (sauvegarde dans un fichier) – désérialisation (restauration depuis un fichier) comporte un mécanisme de contrôle des versions, assurant que les classes servant à la reconstitution des objets sont cohérentes avec les classes dont ces objets étaient instances lors de leur sérialisation,

Le service rendu par Java lors de la sérialisation d'un objet est important, car la tâche est complexe. En effet, pour sauvegarder un objet il faut sauvegarder aussi les éventuels objets qui sont les valeurs de ses variables d'instance ; à leur tour, ces objets peuvent en référencer d'autres, qu'il faut sauver également, et ainsi de suite. Cela revient à parcourir un graphe, qui très souvent comporte des cycles, en évitant de sauvegarder plusieurs fois un même objet et sans sombrer dans des boucles infinies.

A titre d'exemple, voici un programme purement démonstratif qui crée une liste chaînée circulaire et la sauvegarde dans un fichier. Chaque maillon porte une `date` (l'instant de sa création) dont on a supposé que la sauvegarde était sans intérêt ; pour cette raison, cette variable a été déclarée `transient` :

```
class Maillon implements Serializable {
    String info;
    transient Date date;
    Maillon suivant;

    Maillon(String i, Maillon s) {
        info = i;
        suivant = s;
        date = new Date();
    }
    public String toString() {
        return info + " " + date;
    }
}

class TestSerialisation {
    static String[] jour = { "Dimanche", "Lundi", "Mardi", ... "Samedi" };

    public static void main(String[] args) {
        Maillon tete, queue;
        tete = queue = new Maillon(jour[6], null);
        for (int i = 5; i >= 0; i--)
            tete = new Maillon(jour[i], tete);
        queue.suivant = tete;
        try {
            ObjectOutputStream sortie = new ObjectOutputStream(
                                              new FileOutputStream("Liste.dat"));
            sortie.writeObject(tete);
            sortie.close();
        }
        catch (Exception e) {
            System.out.println("problème fichier: " + e.getMessage());
        }
    }
}
```

Et voici un programme qui reconstruit la liste chaînée (bien entendu, dans la liste reconstituée les maillons n'ont pas de `date`) :

```
class TestDeserialisation {
    public static void main(String[] args) {
        Maillon tete = null;
        try {
            ObjectInputStream entree = new ObjectInputStream(
                                                new FileInputStream("Liste.dat"));
            tete = (Maillon) entree.readObject();
            entree.close();
        }
```

```
            catch (Exception e) {
                System.out.println("problème fichier: " + e.getMessage());
            }

                // affichage de contrôle
            for (int i = 0; i < 7; i++, tete = tete.suivant)
                System.out.println(tete);
        }
    }
```

9.5 Expressions régulières

9.5.1 Principe

Le lecteur est supposé connaître par ailleurs la notion d'*expression régulière*, dont l'explication, même succincte, dépasserait le cadre de ce cours. Disons simplement que les expressions régulières considérées ici sont *grosso modo* celles du langage Perl[64], et donnons quelques exemples montrant leur utilisation en Java.

Le paquetage concerné, `java.util.regex`, se compose des deux classes `Pattern` et `Matcher` :

`Pattern` - Un objet de cette classe est la représentation « compilée »[65] d'une expression régulière.

Cette classe n'a pas de constructeur. On crée un objet `Pattern` par une expresion de la forme :

`Pattern` *motif* `= Pattern.compile(`*texteExpReg* [`,` *flags*]`);`

où *texteExpReg* est une chaîne de caractères contenant l'expression régulière. Si cette dernière est incorrecte, l'instruction ci-dessus lance une exception `PatternSyntaxException`, qui est une sous-classe de `RuntimeException`[66].

`Matcher` - Un objet de cette classe se charge d'appliquer une expression régulière sur un texte, pour effectuer les opérations de reconnaissance, de recherche ou de remplacement souhaitées.

On crée un objet `Matcher` à partir d'un objet `Pattern`, par une expression de la forme :

`Matcher` *reconnaisseur* `=` *motif*`.matcher(`*texteAExaminer*`);`

L'objet *reconnaisseur* ainsi créé dispose des méthodes :

`boolean matches()` : l'expression régulière reconnaît-elle la totalité du texte à examiner ?

`boolean lookingAt()` : l'expression régulière reconnaît-elle le début du texte à examiner ?

`boolean find(`[`int start`]`)` : l'expression régulière reconnaît-elle un morceau du texte à examiner ?

A la suite d'un appel d'une des trois méthodes ci-dessus, les méthodes `int start()` et `int end()` de l'objet `Matcher` renvoient le début et la fin de la sous-chaîne reconnue, tandis que la méthode `String group()` renvoie la chaîne reconnue elle-même.

On se reportera à la documentation de l'*API* pour des explications sur les (nombreuses) autres possibilités de la classe `Matcher`.

9.5.2 Exemples

1. Découper un texte. Pour les opérations les plus simples il n'y a pas besoin de créer des objets `Matcher`, des méthodes ordinaires (exemple 1) ou statiques (exemple 2) de la classe `Pattern` suffisent. Par exemple, le programme suivant prend une chaîne donnée en argument et l'affiche à raison d'un mot par ligne, en considérant que les séparateurs de mots sont la virgule (« `,` »), le point-virgule (« `;` ») et les caractères blancs de toute sorte (collectivement représentés par « `\s` ») :

64. La syntaxe précise des expressions régulières acceptées par Java est donnée dans la documentation de l'API, au début de l'explication concernant la classe `java.util.regex.Pattern`

65. Un objet `Pattern` consiste essentiellement en l'encapsulation des tables qui définissent l'automate d'états fini correspondant à l'expression régulière donnée.

66. Rappelons que les `RuntimeException` sont des exceptions non contrôlées. Par conséquent, si une méthode contient des appels de `Pattern.compile` il n'est pas nécessaire de lui adjoindre une clause « `throws PatternSyntaxException` ».


```
import java.util.regex.Pattern;

public class Mots {
    public static void main(String[] args) {
        Pattern motif = Pattern.compile("[,;\\s]+");

        // découpe de la chaîne args[0] en mots
        String[] tabMots = motif.split(args[0]);

        for (int i = 0; i < tabMots.length; i++)
            System.out.println(tabMots[i]);
    }
}
```

Exemple d'utilisation :

```
$ java  Mots  "André, Béatrice     Caroline,  Emile"
André
Béatrice
Caroline
Emile
$
```

2. Vérifier la correction d'une chaîne. Une autre opération simple qu'une méthode statique de la classe `Pattern` peut effectuer sans l'aide d'un objet `Matcher` : compiler une expression régulière et déterminer si elle correspond à la totalité d'une chaîne donnée.

Par exemple, la méthode suivante détermine si la chaîne passée en argument est l'écriture correcte d'un nombre décimal en virgule fixe :

```
boolean bienEcrit(String texte) {
    final String expr = "[+-]?[0-9]+(\\.[0-9]*)?";

    // texte appartient-il à l'ensemble de mots défini par expr ?
    return Pattern.matches(expr, texte);
}
```

Note 1. Cette méthode donne pour bonnes des chaînes comme `"12.3"`, `"123."` et `"123"`, mais rejette `".123"`. Pour corriger cela il suffit de définir `expr` comme ceci :

```
final String expr = "[+-]?(([0-9]+(\\.[0-9]*)?)|(\\.[0-9]+))";
```

Note 2. Dans le cas où la méthode précédente est susceptible d'être fréquemment appelée, il est manifestement maladroit de compiler chaque fois la même expression régulière. L'emploi d'un objet `Pattern` permet d'éviter cela :

```
Pattern pattern = Pattern.compile("[+-]?[0-9]+(\\.[0-9]*)?");

boolean bienEcrit(String texte) {
    return pattern.matcher(texte).matches();
}
```

3. Recherches multiples. Le programme suivant extrait les liens contenus dans une page html (représentés par des balises comme `<a href="http://www.luminy.univ-mrs.fr" ... >`) :

```
import java.io.*;
import java.util.regex.*;

public class TrouverURL {

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

        try {
            File fichier = new File(args[0]);
            char[] tampon = new char[(int) fichier.length()];
            Reader lecteur = new FileReader(fichier);
            lecteur.read(tampon);
            lecteur.close();
            texte = new String(tampon);
        } catch (IOException e) {
            e.printStackTrace();
            return;
        }

        String expReg = "<a[ \\t\\n]+href=\"[^\"]+\"";
        Pattern motif = Pattern.compile(expReg, Pattern.CASE_INSENSITIVE);
        Matcher recon = motif.matcher(texte);

        int pos = 0;
        while (recon.find(pos)) {
            String s = texte.substring(recon.start(), recon.end());
            System.out.println(s);
            pos = recon.end();
        }
    }
}
```

Ce programme affiche des lignes de la forme

```
<a href="MonSoft.html"
<a href="Polys/PolyJava.html"
...
<a href="http://www.luminy.univ-mrs.fr/"
<a href="mailto:garreta@univmed.fr"
```

On peut extraire des informations plus fines, en utilisant la notion de *groupe de capture* des expressions régulières. Voici les seules modifications à faire dans le programme précédent :

```
public class TrouverURL {
    ...
        String expReg = "<a[ \\t\\n]+href=\"([^\"]+)\"";
    ...                                      ↑      ↑
            String s = texte.substring(recon.start(1), recon.end(1));
    ...                                      ↑             ↑
}
```

L'affichage est maintenant comme ceci (magique, n'est-ce pas ?) :

```
MonSoft.html
Polys/PolyJava.html
...
http://www.luminy.univ-mrs.fr/
mailto:garreta@univmed.fr
```

4. Remplacer toutes les occurrences. Le programme suivant remplace les occurrences du mot *chat* par le mot *chien*[67]. Appelé avec l'argument *"un chat, deux chats, trois chats dans mon jardin"* il affiche donc la chaîne *un chien, deux chiens, trois chiens dans mon jardin.*

67. Utile, hein ?

Notez que c'est `recon`, l'objet `Matcher`, qui prend soin de recopier dans la chaîne `sortie` les caractères qui se trouvent *entre* les occurrences du motif cherché :

```
public static void main(String[] args) throws Exception {
    Pattern motif = Pattern.compile("chat");
    Matcher recon = motif.matcher(args[0]);

    StringBuffer sortie = new StringBuffer();
    while(recon.find())
        recon.appendReplacement(sortie, "chien");
    recon.appendTail(sortie);

    System.out.println(sortie.toString());
}
```

L'exemple ci-dessus a été rédigé comme cela pour illustrer les méthodes `appendReplacement` et `appendTail`. On notera cependant que, s'agissant de remplacer toutes les occurrences d'un motif par une chaîne, il y a un moyen plus simple :

```
public static void main(String[] args) throws Exception {
    Pattern motif = Pattern.compile("chat");
    Matcher recon = motif.matcher(args[0]);

    String sortie = recon.replaceAll("chien");

    System.out.println(sortie);
}
```


10 Threads

L'étude des threads est à sa place dans un cours sur la programmation parallèle et sort largement du cadre de ce polycopié. Nous nous limitons ici à donner quelques explications sur certaines notions basiques auxquelles on est confronté dans les applications les plus courantes, par exemple lorsqu'on programme des interfaces utilisateur graphiques.

Un *thread* ou *processus léger* (on dit aussi *fil d'exécution*) est l'entité constituée par un programme en cours d'exécution. Cela se matérialise par un couple (*code*, *données*) : le code en cours d'exécution et les données que ce code traite. Si on en parle ici c'est que Java supporte l'existence simultanée de plusieurs threads : à un instant donné plusieurs programmes peuvent *s'exécuter en même temps* et *en agissant sur les mêmes données*.

Ce que « en même temps » veut dire exactement dépend du matériel utilisé. Dans un système possédant plusieurs processeurs il est possible que divers threads s'exécutent chacun sur un processeur distinct et on est alors en présence d'un parallélisme *vrai*. Mais, plus couramment, les systèmes ne bénéficient que d'un ou deux processeurs et il faut se contenter d'un parallélisme *simulé* : les divers threads existant à un instant donné disposent du processeur *à tour de rôle*.

Différentes raisons peuvent faire qu'un thread qui est en train d'utiliser le processeur le cède à un autre thread qui patiente pour l'avoir :

1) le blocage du thread actif explicitement demandé par une commande placée dans le code, comme `wait` (attendre) ou `sleep` (dormir),
2) le blocage du thread actif par le fait qu'il entame une opération d'entrée-sortie [68],
3) le déblocage d'un thread ayant une priorité supérieure à celle du thread actif,
4) l'épuisement d'une tranche de temps allouée au thread actif.

Notez que les blocages des deux derniers cas ne découlent pas d'opérations exécutées par le code qui se bloque ; ils sont donc tout à fait imprévisibles.

Si tout thread cesse d'être actif dès qu'un thread de priorité supérieure est prêt (cas 3, ci-dessus) on dit qu'on a affaire à un *parallélisme préemptif*. La machine Java procède *généralement* ainsi.

Le fonctionnement par attribution de tranches de temps (cas 4) n'est pas mentionné par les spécifications de la machine Java. Celle-ci en bénéficie ou non, selon les caractéristiques du matériel sous-jacent.

10.1 Création et cycle de vie d'un thread

10.1.1 Déclaration, création et lancement de threads

Il y a deux manières, très proches, de créer un thread :
- en définissant et en instanciant une sous-classe de la classe `Thread` définie à cet effet, dans laquelle au moins la méthode `void run()` aura été redéfinie,
- en appelant le constructeur de `Thread` avec pour argument un objet `Runnable` [69]

Dans un cas comme dans l'autre, le thread créé est rendu « vivant » lors de l'appel de sa méthode `start()`. Java met alors en place tout le nécessaire pour gérer le nouveau thread, puis appelle sa méthode `run()`. Au moment de son démarrage, le thread nouvellement créé :
- a pour code la méthode `run()`,
- a pour données celles du thread dans lequel il vient d'être créé.

Exemple. Le thread purement démonstratif suivant est très simple : dix fois il affiche son nom et un entier croissant, puis s'endort pour une période comprise entre 0 et 1000 millisecondes :

```
public class ThreadSimple extends Thread {
    public ThreadSimple(String nom) {
        super(nom);
    }
```

68. C'est en considérant ce type de situations qu'on peut comprendre pourquoi le parallélisme fait gagner du temps même lorsqu'il est simulé : quand un thread se met en attente d'une entrée-sortie sur disque (opération beaucoup plus lente que le travail en mémoire centrale) ou, pire, en attente d'une frappe au clavier, d'un événement réseau, etc., le contrôle est cédé à un thread non bloqué ; ainsi, le processeur est moins souvent oisif.

69. Un objet `Runnable` – c.-à-d. implémentant l'interface `java.lang.Runnable` – est tout simplement un objet possédant une méthode `run()`.

```
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(getName() + " " + i);
            try {
                sleep((long) (Math.random() * 1000));
            } catch (InterruptedException e) {
            }
        }
        System.out.println(getName() + " terminé");
    }
}
```

NOTE. La méthode `sleep` doit toujours être appelée dans un bloc `try...catch` puisque c'est par le lancement d'une exception `InterruptedException` que le thread endormi est réveillé lorsque le temps indiqué est écoulé. Il en est de même, et pour la même raison, de la méthode `wait`.

Voici un programme qui fait tourner en parallèle deux exemplaires de ce thread :

```
public class DemoDeuxThreads {
    public static void main (String[] args) {
        new ThreadSimple("Seychelles").start();
        new ThreadSimple("Maurice").start();
    }
}
```

Affichage obtenu (c'est un exemple ; lors d'une autre exécution, l'imbrication des messages « `Seychelles` ... » parmi les messages « `Maurice` ... » peut être différente) :

```
Seychelles 0
Seychelles 1
Maurice 0
Seychelles 2
Seychelles 3
Maurice 1
Maurice 2
...
Seychelles terminé
Maurice 8
Maurice 9
Maurice terminé
```

Deuxième exemple (un peu prématuré, les interfaces graphiques ne sont étudiées qu'à la section suivante). Nous souhaitons disposer d'un cadre dont la barre de titre affiche constamment l'heure courante à la seconde près. Il suffira pour cela de créer, en même temps que le cadre, un deuxième thread qui dort [70] la plupart du temps, se réveillant chaque seconde pour mettre à jour le texte affiché dans la barre de titre :

```
import java.text.DateFormat;
import java.util.*;
import javax.swing.*;

public class CadreAvecHeure extends JFrame {
    private String titre;
    public CadreAvecHeure(String titre) {
        super(titre);
        this.titre = titre;
        new Thread(new Runnable() {
            public void run() {
                gererAffichageHeure();
            }
        }).start();
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(600, 400);
        setVisible(true);
    }
```

70. Quand un thread dort il ne consomme aucune ressource système.

```
    private void gererAffichageHeure() {
        while (true) {
            try {
                Thread.sleep(1000);
            } catch(InterruptedException e) {
            }
            Date h = Calendar.getInstance().getTime();
            String s = DateFormat.getTimeInstance().format(h);
            setTitle(titre + " " + s);
        }
    }
    public static void main(String[] args) {
        new CadreAvecHeure("Test");
    }
}
```

10.1.2 Terminaison d'un thread

Lorsqu'un thread atteint la fin de sa méthode `run()`, il se termine normalement et libère les ressources qui lui ont été allouées. Mais comment provoque-t-on la terminaison anticipée [71] d'un thread ?

La classe `Thread` comporte une méthode `stop()` dont le nom exprime bien la fonction. Mais elle est vigoureusement désapprouvée (*deprecated*), car elle laisse dans un état indéterminé certains des objets qui dépendent du thread stoppé.

La manière propre et fiable de terminer un thread consiste à modifier la valeur d'une variable que le thread consulte régulièrement. Par exemple, voici la classe `CadreAvecHeure` montrée précédemment, complétée par une méthode `arreterLaPendule()` qui stoppe le rafraîchissement de l'heure affichée :

```
public class CadreAvecHeure extends JFrame {
    String titre;
    boolean go;

    public CadreAvecHeure(String titre) {
        comme la version précédente
    }

    private void gererAffichageHeure() {
        go = true;
        while (go) {
            try {
                Thread.sleep(1000);
            } catch(InterruptedException e) {
            }
            Date d = Calendar.getInstance().getTime();
            String s = DateFormat.getTimeInstance().format(d);
            setTitle(titre + " " + s);
        }
    }

    public void arreterLaPendule() {
        go = false;
    }
    ...
}
```

10.2 Synchronisation

Lorsqu'il démarre, un thread travaille sur les données du thread dans lequel il a été créé. Par exemple, dans le programme `CadreAvecHeure` ci-dessus, le thread « fils » qui exécute la méthode `gererAffichageHeure` accède aux données `this` (atteinte lors de l'appel de `setTitle`) et `go` qui appartiennent aussi au thread « père » qui l'a créé. Cela pose le problème de l'accès concurrent aux données, source potentielle de blocages

71. On notera que si un thread se compose d'une boucle infinie, comme dans la méthode `gererAffichageHeure`, sa terminaison ne peut être qu'anticipée : sans intervention extérieure, cette méthode ne se terminera jamais.

et d'erreurs subtiles, un problème délicat qu'on règle souvent à l'aide de *verrous* protégeant les *sections critiques*.

10.2.1 Sections critiques

Imaginons la situation suivante : nous devons parcourir une collection, membre d'`UneCertaineClasse`, en effectuant une certaine action sur chacun de ses éléments. La collection est représentée par un objet `List`, l'action par un objet `Action` (une interface expressément définie à cet effet).

```
interface Action {
    void agir(Object obj);
}

public class UneCertaineClasse {
    List liste;
    ...
    void parcourir(Action action) {
        Iterator iter = liste.iterator();
        while (iter.hasNext())
            action.agir(iter.next());
    }
    ...
}
```

S'il n'y a pas plusieurs threads pouvant accéder à la `liste`, la méthode précédente est correcte. Mais s'il peut arriver que, pendant qu'un thread parcourt la `liste`, un autre thread la modifie en ajoutant ou en enlevant des éléments, alors le parcours précédent a des chances de devenir incohérent. Une première manière de résoudre ce problème consiste à rendre synchronisée la méthode `parcourir` :

```
public class UneCertaineClasse {
    List liste;
    ...
    synchronized void parcourir(Action action) {
        Iterator iter = liste.iterator();
        while (iter.hasNext())
            action.agir(iter.next());
    }
    ...
}
```

L'effet du qualifieur `synchronized` placé devant la méthode `parcourir` est le suivant : lors d'un appel de cette méthode, de la forme [72]

```
unObjet.parcourir(uneAction);
```

un verrou sera posé sur `unObjet` de sorte que tout autre thread qui tentera une opération synchronisée sur ce même objet (en appelant une autre méthode `synchronized` de cette même classe) sera bloqué jusqu'à ce que ce verrou soit enlevé.

Cela règle le problème de l'accès concurrent à la liste, mais peut-être pas de manière optimale. En effet, si le traitement que représente la méthode `agir` est long et complexe, alors la liste risque de se trouver verrouillé pendant une longue période et ralentir l'ensemble du programme.

D'où une solution bien plus légère : établir une section critique (i.e. protégée par un verrou) dans laquelle on ne fait que cloner la liste puis, le verrou étant levé, effectuer le parcours du clone, lequel ne craint pas les modifications que d'autres threads pourraient faire sur la liste originale :

```
synchronized static void parcourir(List liste, Action action) {
    List copie = new LinkedList();

    synchronized(liste) {
        Iterator iter = liste.iterator();
        while (iter.hasNext())
            copie.add(iter.next());
    }
```

72. Un appel de la forme « `parcourir(uneAction)` » n'est pas différent, car il équivaut à « `this.parcourir(uneAction)` ».

```
        Iterator iter = copie.iterator();
        while (iter.hasNext())
            action.agir(iter.next());
    }
```

10.2.2 Méthodes synchronisées

Pour terminer cette présentation des threads voici un grand classique de la programmation parallèle : une implémentation du modèle dit « des producteurs et consommateurs ».

Considérons la situation suivante : un certain nombre de fois – par exemple 10 – chaque producteur « fabrique » un produit (numéroté), le dépose dans un entrepôt qui ne peut en contenir qu'un, puis dort un temps variable :

```
public class Producteur extends Thread {
    private Entrepot entrepot;
    private String nom;

    public Producteur(Entrepot entrepot, String nom) {
        this.entrepot = entrepot;
        this.nom = nom;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            entrepot.deposer(i);
            System.out.println("Le producteur " + this.nom + " produit " + i);
            try {
                sleep((int) (Math.random() * 100));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Un certain nombre de fois – par exemple 10 encore – chaque consommateur prend le produit qui se trouve dans l'entrepôt et le « consomme » :

```
public class Consommateur extends Thread {
    private Entrepot entrepot;
    private String nom;

    public Consommateur(Entrepot entrepot, String nom) {
        this.entrepot = entrepot;
        this.nom = nom;
    }

    public void run() {
        int valeur = 0;
        for (int i = 0; i < 10; i++) {
            valeur = entrepot.prendre();
            System.out.println("Le consommateur " + this.nom + " consomme " + valeur);
        }
    }
}
```

Ce qui est remarquable dans ce système c'est que les producteurs et les consommateurs ne se connaissent pas et ne prennent aucune mesure pour prévenir les conflits. Or, lorsque l'entrepôt est vide les consommateurs ne peuvent pas consommer et lorsqu'il est plein les producteurs ne peuvent pas produire. C'est l'entrepôt qui gère ces conflits. En revanche, l'entrepôt est indépendant du nombre de producteurs et de consommateurs qui traitent avec lui :

```
public class Entrepot {
    private int contenu;
    private boolean disponible = false;
```



```
    public synchronized int prendre() {
        while (disponible == false) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        disponible = false;
        notifyAll();
        return contenu;
    }

    public synchronized void deposer(int valeur) {
        while (disponible == true) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        contenu = valeur;
        disponible = true;
        notifyAll();
    }
}
```

Ne vous y trompez pas : le fonctionnement de l'entrepôt est assez savant. Les méthodes `prendre` et `deposer` étant `synchronized`, un seul thread peut s'y trouver à un instant donné.

Lorsqu'un consommateur appelle la méthode `prendre`
- si un produit est disponible (i.e. `disponible == true`) alors `disponible` devient `false` et une notification est envoyée à tous les threads qui sont en attente d'une notification sur cet entrepôt ; en même temps, le consommateur obtient le produit,
- s'il n'y a pas de produit disponible (i.e. `disponible == false`) alors le thread appelle `wait` et se met donc en attente d'une notification sur cet entrepôt.

L'appel de `notifyAll` débloque tous les threads en attente. Si parmi eux il y a des producteurs, *au plus un* d'entre eux trouve `disponible == false` et peut donc produire, tous les autres (producteurs et consommateurs) se rebloquent (à cause du `while`) et attendent une nouvelle notification.

Le fonctionnement de l'entrepôt vu du côté de la méthode `deposer` est rigoureusement symétrique du précédent. Voici un programme principal qui lance deux de ces threads :

```
public class Essai {
    public static void main(String[] args) {
        Entrepot entrepot = new Entrepot();
        Producteur producteur = new Producteur(entrepot, "A");
        Consommateur consommateur = new Consommateur(entrepot, "B");
        producteur.start();
        consommateur.start();
    }
}
```

Affichage obtenu :

```
Le producteur A depose 0
Le consommateur B obtient 0
Le producteur A depose 1
Le consommateur B obtient 1
```
etc.

11 Interfaces graphiques

Plus encore que pour les sections précédentes, il faut rappeler que ces notes ne cherchent pas à être un manuel de référence, même pas succinct. Quels que soient les langages et les systèmes utilisés, les bibliothèques pour réaliser des interfaces graphiques sont toujours extrêmement copieuses. Celles de Java ne font pas exception ; nous n'avons pas la place ici pour expliquer les très nombreuses classes et méthodes qui les constituent.

Nous nous contenterons donc d'expliquer les principes de fonctionnement des éléments d'*AWT* et de *Swing* les plus importants et d'en montrer le mode d'emploi à travers des exemples de taille raisonnable.

Outre un certain nombre de livres plus ou moins exhaustifs, dont l'ouvrage (en français, à part le titre) :

> David Flanagan
> *Java Foundation Classes in a Nutshell*
> O'Reilly, 2000

deux références absolument indispensables pour programmer des interfaces graphiques en Java sont les deux hypertextes déjà cités :

- la documentation en ligne de l'API Java : `http://java.sun.com/javase/6/docs/api/`, notamment tout ce qui concerne les paquets dont les noms commencent par `java.awt` et `javax.swing`,
- le tutoriel en ligne : `http://java.sun.com/tutorial/`, spécialement sa section *Creating a GUI with JFC/Swing*.

11.1 *JFC = AWT + Swing*

Une caractéristique remarquable du langage Java est de permettre l'écriture de programmes portables avec des interfaces-utilisateur graphiques, alors que partout ailleurs de telles interfaces sont réalisées au moyen de bibliothèques séparées du langage et très dépendantes du systèmes d'exploitation sur lequel l'application doit tourner, ou du moins de la couche graphique employée.

La première bibliothèque pour réaliser des interfaces-utilisateur graphiques [73] en Java a été *AWT*, acronyme de *Abstract Windowing Toolkit*. Fondamentalement, l'ensemble des composants de *AWT* est défini comme la partie commune des ensembles des composants graphiques existant sur les différents systèmes sur lesquels la machine Java tourne. Ainsi, chaque composant d'*AWT* est implémenté par un composant *homologue* (*peer*) appartenant au système hôte, qui en assure le fonctionnement.

AWT est une bibliothèque graphique originale et pratique, qui a beaucoup contribué au succès de Java, mais elle a deux inconvénients : puisque ses composants sont censés exister sur tous les environnements visés, leur nombre est forcément réduit ; d'autre part, même réputés identiques, des composants appartenant à des systèmes différents présentent malgré tout des différences de fonctionnement qui finissent par limiter la portabilité des programmes qui les utilisent.

C'est la raison pour laquelle *AWT* a été complétée par une bibliothèque plus puissante, *Swing*. Apparue comme une extension (c.-à-d. une bibliothèque optionnelle) dans d'anciennes versions de Java, elle appartient à la bibliothèque standard depuis l'apparition de « Java 2 ». En *Swing*, un petit nombre de composants de haut niveau (les *cadres* et les *boîtes de dialogue*) sont appariés à des composants homologues de l'environnement graphique sous-jacent, mais tous les autres composants – qui, par définition, n'existent qu'inclus dans un des précédents – sont écrits en « pur Java » [74] et donc indépendants du système hôte.

Au final, *Swing* va plus loin que *AWT* mais ne la remplace pas ; au contraire, les principes de base et un grand nombre d'éléments de *Swing* sont ceux de *AWT*. Ensemble, *AWT* et *Swing* constituent la bibliothèque officiellement nommée *JFC* (pour *Java Foundation Classes* [75]).

L'objet de la description qui va suivre est *Swing* mais, comme on vient de le dire, cela ne nous dispensera pas d'expliquer beaucoup de concepts de *AWT* qui jouent un rôle important dans *Swing*, comme le système des événements (*EventListener*), les gestionnaires de disposition (*LayoutManager*), etc.

Les classes de *AWT* constituent le paquet `java.awt` et un ensemble de paquets dont les noms commencent par *java.awt* : `java.awt.event`, `java.awt.font`, `java.awt.image`, etc.

Les classes de *Swing* forment le paquet `javax.swing` et un ensemble de paquets dont les noms commencent par *javax.swing* : `javax.swing.border`, `javax.swing.filechooser`, `javax.swing.tree`, etc.

73. Pour nommer les interfaces-utilisateur graphiques vous trouverez souvent l'acronyme *GUI*, pour *Graphics User Interface*.

74. La documentation officielle exprime cela en disant que les composants de *AWT*, chacun apparié avec un composant du système sous-jacent, sont « lourds », tandis que ceux de *Swing*, écrits en pur Java, sont « légers ». Ah, la belle langue technicommerciale... !

75. *JFC* est une sorte de citation de la célèbre *MFC* ou *Microsoft Foundation Classes*, une bibliothèque de classes C++ de chez Microsoft pour programmer les interfaces-utilisateur graphiques des applications destinées à Windows.

11.2 Le haut de la hiérarchie

Commençons par donner quelques indications sur la manière dont le travail se partage entre les classes qui constituent le haut de la hiérarchie d'héritage, c'est-à-dire les classes dont toutes les autres héritent.

Cette section 11.2 est entièrement « culturelle » (elle ne contient pas d'informations techniques précises), mais il vaut mieux avoir compris les concepts qu'elle expose avant de commencer à utiliser les composants graphiques.

Composants (`Component`)

La classe `java.awt.Component` est le sommet de la hiérarchie qui nous intéresse ici, c'est-à-dire la superclasse de toutes les classes qui représentent des composants graphiques, aussi bien *AWT* que *Swing*. Le comportement d'un objet `Component` est très riche, mais si on se limite aux fonctionnalités les plus représentatives du rôle d'un composant il faut mentionner :

• *Se dessiner*. Puisqu'un composant est un objet graphique, la plus prévisible de ses méthodes est celle qui en produit l'exposition sur un écran ou un autre organe d'affichage. Cette méthode se nomme `paint` et elle est toute prête pour les composants prédéfinis, qui prennent soin de leur apparence graphique, mais on doit la redéfinir lorsqu'on crée des composants au dessin spécifique.

Comme on le verra, cette méthode obligatoire n'est *jamais* appelée par les programmes ; au lieu de cela, c'est la machine Java qui l'appelle, chaque fois que l'apparence graphique d'un composant a été endommagée et doit être réparée.

Davantage d'informations au sujet de la méthode `paint` et des opérations graphiques sont données dans la section 11.6.

• *Obtenir la taille et la position du composant*. Une autre propriété fondamentale de tout composant est que, une fois dessiné, il occupe un rectangle[76] sur l'écran ou l'organe d'affichage. Des méthodes nommées `getBounds`, `getSize`, `getLocation`, etc., permettent d'obtenir les coordonnées de ce rectangle.

• *Définir la taille et la position du composant*. On pourrait penser que puisqu'il y a des méthodes pour obtenir la taille et la position d'un composant, il y en a également pour fixer ces paramètres.

Ces méthodes existent (elles se nomment `setSize`, `setMinimumSize`, `setMaximumSize`, `setPreferredSize`, etc.) mais la question est plus compliquée qu'on ne pourrait le penser d'abord car, sauf exception, la taille et la position d'un composant découlent d'une négociation permanente avec le *gestionnaire de disposition* (`LayoutManager`) du conteneur dans lequel le composant a été placé.

Des explications sur les gestionnaires de disposition sont données à la section 11.7

• *Définir et obtenir les propriétés graphiques.* Puisque le propre d'un composant est d'être dessiné il est normal qu'il comporte tout un ensemble de méthodes pour définir ou consulter ses propriétés graphiques, comme la couleur du fond (`setBackground`, `getBackground`), la couleur de ce qui est dessiné par-dessus (`setForeground`, `getForeground`), la police courante (`setFont`, `getFont`), etc.

Les propriétés graphiques d'un composant définissent les valeurs initiales du *contexte graphique* (objet `Graphics`) utilisé par toute opérations de « peinture » (i.e. l'appel par la machine Java de la méthode `paint`). Les contextes graphiques et la méthode `paint` sont expliqués à la section 11.6.

• *Être source d'événements*. Puisqu'un composant est visible sur un écran, il est naturellement prédestiné à être la cible d'actions faites par l'utilisateur à l'aide de la souris, du clavier ou de tout autre organe de saisie disponible.

Lorsqu'une telle action se produit, Java crée un *événement*, un objet décrivant le type et les circonstances de l'action produite ; on dit que le composant sur lequel l'action s'est produite est la *source de l'événement*. Java notifie alors cet événement à un ou plusieurs objets, appelés les *auditeurs de l'événement*, qui sont censés déclencher la réaction requise.

Ce mécanisme se manifeste au programmeur à travers des méthodes nommées `add`*`Xxx`*`Listener` (la partie *`Xxx`* dépend de l'événement en question), qui permettent d'attacher des auditeurs des divers types d'événements aux objets qui peuvent en être des sources : `addMouseListener`, `addActionListener`, etc.

La question des événements et de leurs auditeurs est reprise en détail à la section 11.5.

Conteneurs (`Container`)

La classe `java.awt.Container` est une sous-classe de `java.awt.Component`. Un conteneur est un composant qui peut en contenir d'autres. Cela se traduit par deux méthodes fondamentales :

• *Ajouter un composant*. C'est la méthode `add`, sous la forme « *leConteneur*`.add(`*unComposant*`)` » qui effectue de tels ajouts. Selon le gestionnaire de disposition en vigueur, elle peut requérir d'autres arguments.

76. On considère presque toujours qu'un composant occupe un rectangle, même lorsqu'il ne semble pas rectangulaire.

• *Associer au conteneur un gestionnaire de disposition.* Un gestionnaire de disposition, ou *layout manager*, est un objet (invisible) qui « connaît » le conteneur et la liste de ses composants et qui commande la taille et la position de ces derniers, cela aussi bien lors de l'affichage initial du conteneur, au moment de sa création que, par la suite, chaque fois que la taille ou la forme du conteneur sont modifiées.

Des explications sur les gestionnaires de disposition sont données à la section 11.7.

Fenêtres (`Window`)

La classe `java.awt.Window` est une sous-classe de `java.awt.Container`. La caractéristique d'une fenêtre est la possibilité d'être visible sans nécessiter d'être incluse dans un autre composant, contrairement à tous les composants qui ne sont pas des fenêtres.

La classe `Window` apporte donc la machinerie nécessaire pour gérer l'existence d'un composant sur l'organe d'affichage, ce qui demande, parmi d'autres choses, de gérer la coexistence avec les autres fenêtres – dont beaucoup n'ont pas de lien avec l'application en cours, ni même avec Java – qui sont visibles sur l'organe d'affichage en même temps. Cette machinerie est mise en route à travers des méthodes comme `setVisible` (ou `show`) et `pack`, que nous expliquerons plus en détail à la section 11.4.

D'une certaine manière, les fenêtres sont le siège de la « vie » des interfaces graphiques. Lorsqu'une fenêtre est rendue visible, un nouveau thread est créé dans lequel s'exécute le programme qui détecte les actions de l'utilisateur sur la fenêtre et sur chacun des composants qu'elle contient. Cela permet à la fenêtre d'être réactive même lorsque l'application est occupée à d'autres tâches.

Les diverses fenêtres appartenant à une même application et existant à un instant donné forment une arborescence : sauf le cadre de l'application, chaque fenêtre en référence une autre, sa *propriétaire* ; ensemble, les fenêtres constituent donc un arbre ayant pour racine le cadre de l'application. Cela assure, par exemple, que la destruction du cadre principal entraînera bien la destruction de toutes les fenêtres créées par l'application.

Malgré l'importance de la classe `Window` il est rare qu'elle soit explicitement mentionnée dans les applications courantes : les fonctionnalités de cette classe sont presque toujours exploitées à travers les sous-classes `Frame` et `Dialog`.

Cadres et boîtes de dialogue (`Frame`, `Dialog`)

Les classes `java.awt.Frame` (cadre) et `java.awt.Dialog` (boîte de dialogue) sont des sous-classes de `Window`. Les instances de ces deux classes sont des objets bien connus de quiconque a vu on ordinateur (en marche), car de nos jours on les rencontre dans les interfaces graphiques de toutes les applications.

Un cadre est une fenêtre munie d'un bord, un bandeau de titre et, éventuellement, une barre de menus. Sur un système moderne chaque application ordinaire [77] a un cadre et la plupart des applications n'en ont qu'un, si bien que l'application et son cadre finissent par se confondre dans l'esprit de l'utilisateur ; ce n'est pas fâcheux, c'est même un effet recherché.

Du point de vue de l'utilisateur, le cadre d'une application remplit au moins les trois fonctions suivantes :
– le cadre est l'outil standard pour modifier la taille et la forme de l'interface,
– le cadre porte la barre des menus de l'application,
– la fermeture du cadre produit la terminaison de l'application.

Du point de vue du programmeur, un cadre est la seule fenêtre qui n'a pas besoin de posséder une fenêtre propriétaire. Il en découle que le point de départ d'une interface graphique est toujours un cadre, et que tout composant est « rattaché » à un cadre [78] :
– un composant qui n'est pas une fenêtre (`Window`) doit être inclus dans un conteneur,
– un composant qui est une fenêtre mais pas un cadre doit référencer une autre fenêtre, sa propriétaire.

Les boîtes de dialogue sont elles aussi des fenêtres munies d'un bord et d'une barre de titre mais, contrairement aux cadres, elles sont par nature nombreuses, diversifiées et éphémères.

Les boîtes de dialogue sont l'outil standard pour afficher un message, poser une question ou demander des informations. Leur apparition est commandée par le programme, au moment requis, et elles permettent à l'utilisateur de lire le message ou la question posée et, le cas échéant, de faire les actions demandées (cocher des cases, remplir des champs de texte, faire des choix dans des listes, etc.) ; il y a toujours un ou plusieurs

77. Par *application ordinaire* nous entendons une application visible à l'écran qui interagit avec l'utilisateur. Cela exclut les applications invisibles, les « services », les « démons », etc.

78. En particulier, la destruction d'un cadre produit celle de tous les composants qui lui sont rattachés ; cela résout les problèmes de fenêtres orphelines survivant à la terminaison de l'application qui les a créées, de bouts de composant oubliés et, plus généralement, de certains types de fuites de mémoire qui ont empoisonné les premières bibliothèques graphiques.

boutons (« OK », « Oui », « Non », « Open », etc.) dont la pression produit la disparition de la boîte de dialogue et, s'il y a lieu, l'acquisition par le programme des informations que l'utilisateur a données en agissant sur la boîte.

Les boîtes de dialogue peuvent être *modales* et *non modales*. Aussi longtemps qu'elle est visible, une boîte modale bloque (c'est-à-dire rend insensibles aux actions de l'utilisateur) toutes les autres fenêtres de l'application, sauf celles dont la boîte en question est propriétaire, directement ou indirectement.

Les boîtes non modales n'ont pas cet effet bloquant : une boîte de dialogue non modale et les autres fenêtres de l'application « vivent » en parallèle, ce qui peut être une cause de trouble de l'utilisateur. Les boîtes de dialogue sont très majoritairement modales.

Le cas des composants de *Swing*

Swing est venu après *AWT* et, quand il est arrivé, tous les noms intéressants étaient déjà pris : `Component`, `Frame`, `Dialog`, `Button`, `Panel`, etc. Plutôt que d'inventer des noms entièrement nouveaux, forcément peu significatifs, les concepteurs de *Swing* ont préféré nommer les classes de *Swing* par les mêmes noms que *AWT*, avec un signe distinctif constant, un *J* en début du nom : `JComponent`, `JFrame`, `JDialog`, `JButton`, `JPanel`, etc.

Attention, l'oubli de ce *J* initial change le sens de ce qu'on écrit, mais d'une manière que le compilateur ne peut pas toujours signaler comme une erreur ; cela introduit des dysfonctionnements plus ou moins importants dans les programmes.

Les sections précédentes traitant des classes `java.awt.Component`, `java.awt.Container`, `java.awt.Window`, `java.awt.Frame`, `java.awt.Dialog`, etc. on pourrait penser qu'elles ne concernent pas *Swing*. C'est tout le contraire, car toutes les classes des composants de *Swing* sont sous-classes d'une ou plusieurs de celles-là : `javax.swing.JComponent` est (pour des raisons techniques) sous-classe de `java.awt.Container`, donc de `java.awt.Component`, et la majorité des composants de *Swing* sont sous-classes de `java.awt.JComponent` ; `javax.swing.JFrame` et `javax.swing.JDialog` sont sous-classes de `java.awt.Frame` et `java.awt.Dialog` respectivement.

Enfin, nos avons déjà dit que certaines des fonctions les plus fondamentales de *Swing* sont assurées par des éléments de *AWT* qui n'ont pas été étendus lors de la conception de *Swing* : les événements, les gestionnaires de disposition, etc.

11.3 Le point de départ : `JFrame` et une application minimale.

Venons-en à des choses plus pratiques. Pour commencer, voici une application parmi les plus réduites qu'on puisse écrire :

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

public class Bonjour {
    public static void main(String[] args) {
        JFrame cadre = new JFrame("Respect des traditions");
        JLabel etiquette = new JLabel("Bonjour à tous!", JLabel.CENTER);
        cadre.getContentPane().add(etiquette);
        cadre.getContentPane().setBackground(Color.WHITE);
        cadre.setSize(250, 100);
        cadre.setVisible(true);
    }
}
```

L'exécution de ce programme affiche le cadre montré à la figure 9. Il est extrêmement simple, mais il est « vivant » : l'utilisateur peut en changer la taille et la position, le maximiser ou l'*iconifier*, le faire passer devant ou derrière une autre fenêtre, etc.

Attention. Contrairement aux apparences, si – comme ici – le programmeur n'a pas pris les dispositions utiles, cliquer sur la case de fermeture (le bouton avec une croix à l'extrême droite du bandeau de titre) fera disparaître le cadre mais *ne terminera pas* l'application, qui continuera à tourner, et à consommer des ressources, alors qu'elle n'aura plus aucun élément visible !

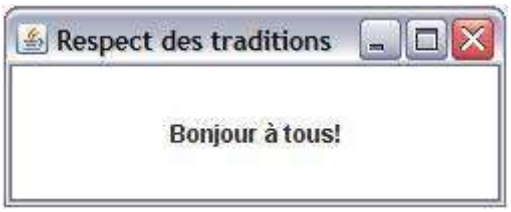


FIGURE 9 – Une version *Swing* de l'incontournable « `PRINT "HELLO"` »

En attendant une meilleure idée [79] on peut évacuer ce problème agaçant en ajoutant, après la création du `cadre`, l'instruction

```
cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Notez que, contrairement à ce qu'on aurait pu penser (et contrairement à ce qui se passe pour les `Frame` de `AWT`), un `JFrame` ne contient pas *directement* les composants qu'on souhaite voir dans le cadre. Un `JFrame` est un objet complexe dont un élément, le « panneau de contenu » (*content pane*) est le conteneur auquel il faut ajouter des composants. Sur la figure 9, le panneau de contenu est le rectangle blanc contenant le texte « Bonjour à tous ! ». On accède au panneau de contenu d'un objet `JFrame` par l'expression `getContentPane()` [80].

En réalité, le programme précédent n'est pas tout à fait minimal. Si on se contentait d'un cadre vide on pourrait faire encore plus simple en supprimant les trois lignes à partir de « `JLabel etiquette =` ... ». Mais les trois instructions restant alors sont obligatoires :

– appel de `new JFrame(...)` pour créer l'objet `JFrame`.

  Si on était, comme c'est souvent le cas, en train de définir une sous-classe de `JFrame`, l'appel du constructeur de `JFrame` serait caché dans une instruction `super(...)` implicite ou explicite,

– appel de `setSize(...)` pour donner une taille au cadre, sinon il sera réduit à quelques pixels.

  A la place de `setSize(...)` on peut appeler la méthode `pack()`, qui donne au cadre une taille juste suffisante pour contenir les composantes qui lui sont ajoutés, mais il faut alors que ces composants aient une taille minimum significative ; de plus, il faut appeler `pack` *après* avoir ajouté ces composants,

– appel de `setVisible()` pour faire apparaître le cadre parmi les autres éléments graphiques visibles à l'écran et pour initier le thread chargé de détecter et traiter les actions de l'utilisateur sur le cadre.

En pratique les cadres des applications sont destinés à être fortement personnalisés, en recevant des composants divers et nombreux. C'est pourquoi il est plus commode de définir une sous-classe plutôt qu'une instance de `JFrame`. Voici une nouvelle version du programme précédent, plus en accord avec ce qui se fait habituellement :

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

public class Bonjour extends JFrame {
    public Bonjour() {
        super("Respect des traditions");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JLabel etiquette = new JLabel("Bonjour à tous!", JLabel.CENTER);
        getContentPane().add(etiquette);
        getContentPane().setBackground(Color.WHITE);
        setSize(250, 100);
        setVisible(true);
    }
    public static void main(String[] args) {
        JFrame cadre = new Bonjour();
    }
}
```

79. Une meilleure idée serait un de ces dialogues qui s'instaurent dans beaucoup d'applications lorsque l'utilisateur manifeste des envies de départ, genre « Attention, votre travail n'est pas enregistré, vous voulez vraiment quitter l'application ? ».

80. Cette complication dans l'ajout des composants aux cadres de *Swing* a disparu dans la version 5 du *JDK*, car la méthode `add` de `JFrame` y a été surchargée afin qu'on puisse écrire *unCadre*`.add(`*unComposant*`)`.

NOTE. Puisqu'elle n'est plus mentionnée par la suite, la variable locale `cadre` est tout à fait superflue. La méthode précédente peut aussi bien s'écrire :

```
public static void main(String[] args) {
    new Bonjour();
}
```

11.4 Le *thread* de gestion des événements

On peut être surpris par le programme précédent : le thread[81] principal de l'application – celui qui se crée quand on lance le programme, et dont la tâche est d'appeler la méthode `main` – exécute une unique instruction, la création d'une instance de `Bonjour`, puis se termine.

L'exécution de ce programme ne devrait donc durer qu'un instant. Pourtant, l'expérience montre qu'une interface graphique est mise en place et reste après la terminaison du thread principal.

C'est donc qu'un deuxième thread a été créé et se charge de faire vivre l'interface graphique, ce qui veut dire détecter les actions de l'utilisateur, notamment à l'aide de la souris, et réagir en conséquence, par exemple en modifiant l'aspect de l'interface. Tant que ce deuxième thread ne sera pas terminé, l'interface graphique existera.

Ce thread est appelé « thread de traitement des événements » (*event-dispatching thread*). Il faut surtout savoir deux choses à son sujet : à quel moment il est créé et qu'il est le seul habilité à modifier l'interface graphique.

CRÉATION. Le thread de traitement des événements est créé lorsque l'interface graphique est *réalisée*, c'est-à-dire effectivement affichée ou prête à l'être :
- un composant de haut niveau (`JFrame`, `JDialog` ou `JApplet`) est réalisé lors de l'appel d'une de ses méthodes `setVisible(true)`, `show()` ou `pack()`,
- la réalisation d'un composant de haut niveau produit la réalisation de tous les composants qu'il contient à ce moment-là,
- l'ajout d'un composant à un conteneur déjà réalisé entraîne la réalisation du composant ajouté.

ACCÈS CONCURRENTS À L'INTERFACE. Le thread de traitement des événements est le seul habilité à modifier l'interface. Imaginez une application dans laquelle le thread principal ne se termine pas immédiatement après avoir réalisé le cadre de l'application ; ce serait une mauvaise idée que de faire que ce thread, ou un autre créé par la suite, ajoute ou modifie ultérieurement des composants de l'interface car plusieurs thread feraient alors des accès concurrents aux mêmes objets. Il pourrait en résulter diverses sortes de situations d'erreur et de blocage. La règle, appelée « règle du thread unique », est la suivante :

> *Lorsqu'un composant* Swing *a été réalisé, tout code qui peut affecter l'état de ce composant ou en dépendre doit être exécuté dans le thread de traitement des événements.*

Qu'un code soit « exécuté dans le thread de traitement des événements » signifie en pratique qu'il est écrit dans un gestionnaire d'événements (i.e. une méthode d'un `EventListener`, destinée à être appelée par Java en réaction à la survenue de l'événement) ou dans une méthode appelée par un tel gestionnaire.

Il y a des exceptions à cette règle, c'est-à-dire des opérations sur des composants de l'interface qui peuvent être appelées sans danger depuis n'importe quel thread. Elles sont toujours signalées dans la documentation par le texte "*This method is thread safe, although most Swing methods are not.*"

Par exemple, les zones de texte `JTextArea` possèdent des méthodes `setText(String t)` et `append(String t)` permettant de remplacer ou d'allonger le texte affiché dans la zone de texte. Ces opérations sont *thread safe*, car le contraire serait très malcommode.

D'autres opérations *thread safe* bien connues sont les méthodes `repaint(...)`, `revalidate(...)`, etc. de la classe `JComponent`. Ces méthodes sont censées mettre à jour l'affichage des composants, mais en réalité elles ne font que « poster » des requêtes dans la file d'attente des événements (c'est le thread de traitement des événements qui les en extraira).

A RETENIR. Conséquence pratique de la règle du thread unique : l'appel de `setVisible(true)` ou de `show()` doit être la *dernière instruction* du processus de construction d'un cadre ou d'une boîte de dialogue.

81. Un *thread* (cf. § 10, page 94), on dit aussi *fil d'exécution* ou *processus léger*, est l'entité constituée par *un programme en cours d'exécution*. Les éléments principaux de cette entité sont le code qui s'exécute et les données que ce code manipule.

Le langage Java est *multithread* : le lancement d'une application crée un *thread principal*, dont le code est déterminé par la méthode `main`, mais ce thread peut en créer un ou plusieurs autres, qui s'exécuteront en parallèle (parallélisme *vrai* si votre machine comporte plusieurs processeurs, parallélisme *simulé* si elle n'en comporte qu'un).

On dit que les thread sont des processus légers surtout pour indiquer qu'ils partagent les données : lorsqu'un thread est créé il accède à toutes les variables du thread qui l'a créé.

S'il y a un appel de `pack()` alors celui-ci doit se trouver immédiatement avant[82] `setVisible(true)` ou `show()`.

La plus petite application avec interface graphique revisitée

Respecter la règle du thread unique (ci-dessus) n'est pas toujours commode, ni même possible. Il existe des composants complexes dont la mise en place requiert qu'on agisse sur eux après qu'ils ont été réalisés et rendus visibles, ce qui introduit la possibilité d'inter-blocages.

Une manière évidemment correcte de résoudre en bloc tous ces problèmes consiste à exécuter *toutes* les opérations concernant l'interface graphique dans un seul thread, forcément le thread de gestion des événements.

C'est pourquoi l'équipe de développement de Java promeut désormais une nouvelle manière de créer et lancer les interfaces graphiques. Le programme *correct* montré à la section 11.3 était, jusqu'en décembre 2003, la version canonique d'une petite application avec interface graphique, mais la forme « officielle » d'un tel programme est désormais la suivante :

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

public class Bonjour {

    private static void creerEtMontrerInterface() {
        JFrame cadre = new JFrame("Respect des traditions");
        JLabel etiquette = new JLabel("Bonjour à tous!", JLabel.CENTER);
        cadre.getContentPane().add(etiquette);
        cadre.getContentPane().setBackground(Color.WHITE);
        cadre.setSize(250, 100);
        cadre.setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                creerEtMontrerInterface();
            }
        });
    }
}
```

Ce programme peut sembler obscur. Il se lit comme ceci : la méthode `SwingUtilities.invokeLater(`*obj*`)` prend pour argument un objet `Runnable` (c'est-à-dire un objet possédant une méthode `run()`) et elle produit l'appel de *obj*`.run()` dans le thread de gestion des événements[83].

Nous obtenons un objet *runnable* en créant une instance d'une classe anonyme définie comme implémentation de l'interface `Runnable` et ayant une méthode `run()` qui se réduit à un appel d'une méthode `creerEtMontrerInterface` qui, d'évidence, remplace ce qui était précédemment la méthode `main`.

11.5 Événements

Puisqu'un composant est visible sur un écran, il est naturellement prédestiné à devenir la cible d'actions faites par l'utilisateur à l'aide de la souris, du clavier ou de tout autre organe de saisie disponible.

Au moment où elle est détectée par l'ordinateur, une action de l'utilisateur provoque la création par la machine Java d'un *événement*, un objet qu'on peut imaginer comme un « rapport » décrivant la nature et les circonstances de cette action ; on dit que le composant qui en a été la cible est la *source*[84] de l'événement en question.

Lorsqu'un événement est créé, le composant qui en est la source a la charge de notifier ce fait à tous les objets qui ont été enregistré auprès de lui comme étant « concernés » par ce type d'événements et souhaitant

82. En théorie, la séquence « `pack(); show();` » n'est pas en accord avec la règle du thread unique, puisque l'appel de `show` est un accès, dans le thread principal, à un composant déjà réalisé par l'appel de `pack`. En pratique cela passe, car il y a bien peu de chances qu'un troisième thread fasse des accès aux composants d'une interface qui n'a pas encore été rendue visible.

83. Cela nous concerne peu ici, mais il faut savoir que la méthode `invokeLater(`*obj*`)` produit l'appel de *obj*`.run()` uniquement *après* que tous les événements en attente ont été traités (c'est là l'aspect *later* de cette méthode).

84. Attention, le mot est un peu trompeur : la vraie source de l'événement est l'utilisateur (ou sa main qui tient la souris), mais cela n'intéresse notre programme qu'à partir du moment où l'action est ressentie par la machine ; à ce moment, c'est bien la cible de l'action de l'utilisateur qui devient le point d'apparition, ou source, de l'événement.

donc être prévenus lorsqu'ils se produiraient. Bien entendu, « notifier un événement » c'est appeler une certaine méthode, spécifique de l'événement.

Les objets qui demandent à recevoir les notifications des événements d'un certain type doivent donc posséder les méthodes correspondantes ; on garantit cela en imposant que ces objets soient des implémentations d'interfaces *ad hoc*, nommées *XxxListener*, où *Xxx* caractérise le type d'événement considéré : `MouseListener`, `WindowListener`, `ActionListener`, etc.

En résumé : pour qu'un objet puisse recevoir les notifications d'une catégorie *Xxx* d'événements il faut que sa classe implémente l'interface *XxxListener* ; cet objet peut alors être enregistré auprès d'une source d'événements *Xxx* en tant qu'auditeur (*listener*) des événements *Xxx*.

Par conséquent, si un objet est source d'événements *Xxx* alors il doit posséder la méthode *addXxxListener*[85] grâce à laquelle d'autres objets lui sont enregistrés en qualité d'auditeurs de ces événements.

11.5.1 Exemple : détecter les clics de la souris.

Donnons-nous le problème suivant : détecter les clics de la souris au-dessus d'un panneau et en afficher les coordonnées.

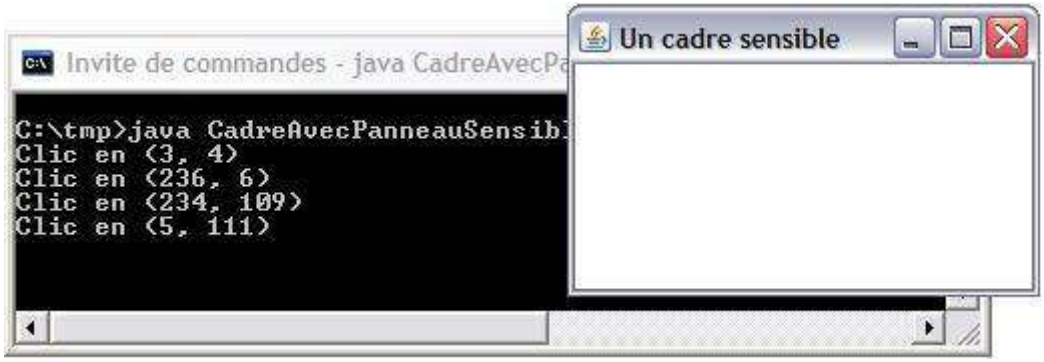


FIGURE 10 – Détecter les clics

La figure 10 montre notre application et l'état de la console d'où on l'a lancée, après qu'on a cliqué près des quatre coins du panneau. Voici une première version de ce programme (des explications sont données après le listing) :

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

public class CadreAvecPanneauSensible extends JFrame {
    CadreAvecPanneauSensible() {
        super("Un cadre sensible");
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        JPanel panneau = new JPanel();
        panneau.setBackground(Color.WHITE);
        GuetteurSouris felix = new GuetteurSouris();
        panneau.addMouseListener(felix);

        getContentPane().add(panneau);
        setSize(250, 150);
        setVisible(true);
    }

    public static void main(String[] args) {
        new CadreAvecPanneauSensible();
    }
}
```

85. On peut en déduire un moyen de découvrir l'ensemble des événements dont un composant peut être la source : chercher, dans la documentation de sa classe, les méthodes nommées *addXxxListener*.

```
class GuetteurSouris implements MouseListener {
    public void mousePressed(MouseEvent e) {
        System.out.println("Clic en (" + e.getX() + ", " + e.getY() + ")");
    }
    public void mouseReleased(MouseEvent e) {
    }
    public void mouseClicked(MouseEvent e) {
    }
    public void mouseEntered(MouseEvent e) {
    }
    public void mouseExited(MouseEvent e) {
    }
}
```

Le code précédent met en évidence les trois éléments clés dans tout processus d'exploitation d'actions de l'utilisateur :

- *les événements* qu'il faut détecter et traiter ; ici, il s'agit des instances de la classe `MouseEvent`. Notez que le programme ne montre pas la création de ces objets, pas plus que l'appel des méthodes pour les exploiter, cela est l'affaire de méthodes (héritées de la classe `Component`) qui ne sont appelées que par la machine Java ;
- *la source* de ces événements, une instance indirecte de la classe `java.awt.Component`. Ici, la source des événements est l'obje `JPanel` qui est la valeur de la variable `panneau` ;
- *le gestionnaire* de ces événements, un auditeur dûment enregistré auprès de la source. Ici, cet auditeur est une instance, nommée `felix`, de notre classe `GuetteurSouris`.

On constate donc que l'expression

```
panneau.addMouseListener(felix)
```

joue un rôle central dans cette affaire : elle indique que `panneau` est considéré ici comme une source d'événements souris [86] et que ces événements, lorsqu'ils surviennent, doivent être notifiés à l'objet `felix`.

Les quatre méthodes vides de la classe `GuetteSouris` peuvent surprendre. Il faut comprendre qu'elles découlent de contraintes successives :
- d'une part, pour pouvoir être enregistré comme auditeur d'événements souris, un objet doit appartenir à une classe qui implémente l'interface `MouseListener`, cela est imposé par la déclaration de la méthode `addMouseListener`,
- d'autre part, l'interface `MouseListener` se compose de cinq méthodes (abstraites), correspondant aux cinq événements qu'on peut déclencher avec une souris.

NOTE 1. Puisque la classe `GuetteSouris` n'est utilisée que dans des méthodes de la classe `CadreAvecPanneauSensible`, on aurait pu déclarer la première à l'intérieur de la deuxième. De cette manière, `GuetteSouris` serait devenue un *classe interne* (cf. section 6.6) à la classe `CadreAvecPanneauSensible`.

NOTE 2. Rien n'oblige à ce que le rôle d'auditeur d'une sorte d'événements soit tenu par des objets d'une classe spécifiquement définie à cet effet. N'importe quel objet peut tenir ce rôle – y compris celui qui est la source des événements en question – à la condition qu'on en ait fait une implémentation de l'interface *XxxListener* correspondante. Par exemple, voici une autre écriture de notre exemple, dans laquelle c'est le cadre qui exploite les événements souris :

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

public class CadreAvecPanneauSensible extends JFrame implements MouseListener {
    CadreAvecPanneauSensible() {
        super("Un cadre sensible");
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        JPanel panneau = new JPanel();
        panneau.setBackground(Color.WHITE);
        panneau.addMouseListener(this);
```

86. Entendons-nous bien : un `JPanel` est *toujours* une source d'événements souris. Mais ces événements ne sont exploités par l'application que si un ou plusieurs auditeurs ont été enregistrés.


```
        getContentPane().add(panneau);
        setSize(250, 150);
        setVisible(true);
    }

    public static void main(String[] args) {
        new CadreAvecPanneauSensible();
    }

    public void mousePressed(MouseEvent e) {
        System.out.println("Clic en (" + e.getX() + ", " + e.getY() + ")");
    }

    public void mouseReleased(MouseEvent e) {
    }
    public void mouseClicked(MouseEvent e) {
    }
    public void mouseEntered(MouseEvent e) {
    }
    public void mouseExited(MouseEvent e) {
    }
}
```

11.5.2 Adaptateurs et classes anonymes

ADAPTATEURS. Quand on écrit l'implémentation d'une interface *Xxx*`Listener` il est regrettable de devoir écrire toutes les méthodes de l'interface alors que seul un petit nombre d'événements possibles, voire un seul, nous intéresse.

Les *adaptateurs* sont des implémentations toutes prêtes des interfaces d'auditeurs d'événements, entièrement faites de méthodes vides. Chaque interface[87] *Xxx*`Listener` possède une classe *Xxx*`Adapter` correspondante. Pour écrire le traitement des événements qui nous intéressent il suffit alors de définir une sous classe de l'adaptateur concerné, dans laquelle seules les méthodes pertinentes sont définies.

Par exemple, voici notre programme précédent – première version – repris dans cet esprit :

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

public class CadreAvecPanneauSensible extends JFrame {
    CadreAvecPanneauSensible() {
        super("Un cadre sensible");
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        JPanel panneau = new JPanel();
        panneau.setBackground(Color.WHITE);
        GuetteurSouris felix = new GuetteurSouris();
        panneau.addMouseListener(felix);

        getContentPane().add(panneau);
        setSize(250, 150);
        setVisible(true);
    }

    public static void main(String[] args) {
        new CadreAvecPanneauSensible();
    }
}
```

87. Sauf les interfaces constituées d'une seule méthode, comme `ActionListener`, pour lesquelles un adaptateur n'aurait aucun intérêt.

```
class GuetteurSouris extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        System.out.println("Clic en (" + e.getX() + ", " + e.getY() + ")");
    }
}
```

Classes anonymes. On peut faire encore plus simple. Dans l'exemple précédent nous définissons une sous-classe de `MouseAdapter` dans le but exclusif d'en créer une unique instance. Dans un tel cas de figure, Java permet qu'on définisse la sous-classe *dans la même expression qui crée l'instance*. La sous-classe est alors sans nom, mais cela n'a pas d'importance puisqu'on n'envisage pas de lui faire d'autres instances. Cela s'écrit :

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

public class CadreAvecPanneauSensible extends JFrame {
    CadreAvecPanneauSensible() {
        super("Un cadre sensible");
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        JPanel panneau = new JPanel();
        panneau.setBackground(Color.WHITE);

        MouseListener felix = new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                System.out.println("Clic en (" + e.getX() + ", " + e.getY() + ")");
            }
        };
        panneau.addMouseListener(felix);

        getContentPane().add(panneau);
        setSize(250, 150);
        setVisible(true);
    }

    public static void main(String[] args) {
        new CadreAvecPanneauSensible();
    }
}
```

Note. Si on ne craint pas les écritures compactes on peut même se passer de la variable `felix` :

```
...
panneau.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        System.out.println("Clic en (" + e.getX() + ", " + e.getY() + ")");
    }
});
...
```

11.5.3 Principaux types d'événements

Chaque type d'événement correspond à une interface d'auditeurs. Les principales (et les seules en *AWT*) sont les suivantes :

MouseListener

L'interface `MouseListener` concerne les événements *isolés*[88] générés à l'aide de la souris. Il s'agit d'événements de bas niveau, c'est-à-dire des actions avec la souris sur des composants qui ne sont pas équipés pour traduire ces gestes élémentaires en informations de plus haut niveau, comme c'est le cas pour les boutons, les menus, etc. Les méthodes de cette interface sont :

88. Par événements « isolés » nous entendons les événements autres que les *trains d'événements* que produisent les déplacements de la souris, voyez *Mouse motion events*, plus loin.

`void mousePressed(MouseEvent e)` Appelée lorsqu'un bouton de la souris a été pressé, le pointeur se trouvant sur le composant dont on guette les événements.

`void mouseReleased(MouseEvent e)` Appelée lorsqu'un bouton de la souris a été relâché, alors qu'il avait été pressé quand le pointeur se trouvait sur le composant dont on guette les événements.

`void mouseClicked(MouseEvent e)` Appelée lorsque la souris a été « cliquée » (pressée puis relâchée *au même endroit*), le curseur se trouvant sur le composant dont on guette les événements. De par sa définition, cet événement est toujours précédé d'un événement `mouseReleased`. La réciproque n'est pas vraie : un événement `mouseReleased` n'est suivi d'un événement `mouseClicked` que si la position de la souris n'a pas changé depuis le dernier événement `mousePressed`.

`void mouseEntered(MouseEvent e)` Appelée lorsque le pointeur de la souris commence à « survoler » le composant dont on guette les événements.

`void mouseExited(MouseEvent e)` Appelée lorsque le pointeur de la souris cesse de « survoler » le composant dont on guette les événements.

L'argument de ces méthodes est un objet `MouseEvent`. Les méthodes les plus intéressantes d'un tel objet sont :

`getX(), getY()` qui renvoient les coordonnées[89] du pointeur au moment où l'événement a été notifié,

`getButton()` qui permet de savoir quel bouton a été pressé.

MouseMotionListener

Cette interface concerne les mouvements de la souris. Lorsqu'on fait bouger la souris sans appuyer sur un de ses boutons on dit qu'on la « déplace » (*move*), lorsqu'on la fait bouger tout en maintenant un bouton pressé on dit qu'on la « traîne » (*drag*). Cette interface comporte donc deux méthodes :

`void mouseMoved(MouseEvent e)` Appelée lorsque la souris change de position sans qu'aucun bouton ne soit pressé.

`void mouseDragged(MouseEvent e)` Appelée lorsque la souris change de position avec un de ses boutons pressé.

Il faut être plutôt sobre en implémentant les méthodes précédentes, car elles ont à traiter des événements qui arrivent par trains. Selon les performances du système utilisé, un grand nombre d'appels de ces méthodes est généré pendant un seul déplacement de la souris. Il faut que la réponse soit courte pour ne pas introduire un asynchronisme désagréable.

KeyListener

Il se produit un *événement clavier* chaque fois qu'une touche est pressée ou relâchée. Tous les composants peuvent être sources de tels événements, il suffit qu'ils « aient le focus ». Cette interface se compose de trois méthodes :

`void keyPressed(KeyEvent e)` Appelée lorsqu'une touche a été pressée.

`void keyReleased(KeyEvent e)` Appelée lorsqu'une touche a été relâchée.

`void keyTyped(KeyEvent e)` Appelée lorsqu'un caractère a été saisi au clavier.

Les événements `keyPressed` et `keyReleased` sont de bas niveau. Chacun correspond à une touche unique, et fournit un code numérique (le *code de touche virtuelle*, ou *VK code*). En particulier, `keyPressed` est le seul moyen de détecter la pression d'une touche ne produisant pas de caractère (comme « Shift », « Ctrl », etc.).

L'événement `keyTyped`, au contraire, est assez élaboré, puisqu'il est capable d'agréger plusieurs touches pressées ensemble pour former un unique caractère : A majuscule, Control-C, etc.

L'argument de ces méthodes est un objet `KeyEvent`, dont les méthodes les plus intéressantes sont :

`char getKeyChar()` Renvoie le caractère associé à l'événement clavier (compte tenu des éventuelles touches de modification) Dans le cas de `keyPressed` et `keyReleased` il peut n'exister aucun caractère Unicode représentant la touche concernée, la valeur `KeyEvent.CHAR_UNDEFINED` est alors renvoyée.

`int getKeyCode()` Dans le cas de `keyPressed` et `keyReleased`, renvoie le code de touche virtuelle correspondant à la touche pressée ou relâchée. Dans le cas de `keyTyped`, le résultat est `KeyEvent.VK_UNDEFINED`.

Sur beaucoup de systèmes, si l'utilisateur maintient une touche pressée, un flot d'événements `keyPressed` est généré.

89. On notera que, dans le cas des événements `MouseExited` et, surtout, `MouseReleased`, ces coordonnées peuvent être négatives.

FocusListener

Les événements qui nous intéressent ici concernent l'acquisition ou la perte du *focus* par un composant. A tout moment, un seul composant a le focus ; par définition, c'est à lui que sont adressés les événements du clavier. Généralement, un composant acquiert le focus lorsqu'il subit un clic de la souris ; le composant qui avait le focus à ce moment-là le perd. La pression successive de la touche tabulation fait également circuler le focus parmi les composants placés sur un même conteneur.

`void focusGained(FocusEvent e)` Appelée lorsque le composant en question acquiert le focus (c'est-à-dire que les actions sur le clavier lui seront désormais adressées).

`void focusLost(FocusEvent e)` Appelée lorsque le composant perd le focus (un autre composant l'a pris).

ActionListener

Cette interface concerne des actions (avec la souris ou le clavier) sur un composant qui les traduit en une commande de plus haut niveau, comme un choix dans une liste ou un menu, une pression sur un bouton, etc. L'interface se compose d'une seule méthode :

`void actionPerformed(ActionEvent e)` Appelée lorsqu'une action a été faite (ce que « action » veut dire dépend du composant : presser un bouton, cocher ou décocher une case à cocher, faire un choix dans un menu, etc.).

L'argument de cette méthode est un objet `ActionEvent`, muni de la méthode

`String getActionCommand()` Renvoie une chaîne qui identifie l'action en question. En règle générale, il s'agit d'une chaîne écrite sur le composant sur lequel l'action de l'utilisateur s'est effectuée : le titre d'un bouton pressé ou d'une case cochée, le texte de l'élément de liste ou de l'item de menu choisi, etc.

FIGURE 11 – Trois boutons

Exemple : voici la définition d'une classe `Panneau` très naïve qui est un conteneur portant trois boutons (voyez la figure 11) et, en même temps, l'auditrice des actions dont ces boutons sont la source :

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

class TroisBoutons extends JPanel implements ActionListener {

    public TroisBoutons() {
        JButton bouton = new JButton("Oui");
        bouton.addActionListener(this);
        add(bouton);

        bouton = new JButton("Non");
        bouton.addActionListener(this);
        add(bouton);

        bouton = new JButton("Euh...");
        bouton.addActionListener(this);
        add(bouton);
    }
```



```
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Oui"))
            System.out.println("Il a dit oui");
        else if (e.getActionCommand().equals("Non"))
            System.out.println("Il a dit non");
        else
            System.out.println("Il ne sait pas");
    }

    public static void main(String[] args) {
        JFrame cadre = new JFrame("Boutons");
        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cadre.getContentPane().add(new TroisBoutons());
        cadre.pack();
        cadre.setVisible(true);
    }
}
```

Voici une autre manière d'écrire une classe fonctionnellement identique à la précédente, en utilisant les références des boutons plutôt que leurs chaînes *action command*[90] :

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

class TroisBoutons extends JPanel implements ActionListener {
    JButton bOui, bNon;

    public TroisBoutons() {
        bOui = new JButton("Oui");
        bOui.addActionListener(this);
        add(bOui);

        bNon = new JButton("Non");
        bNon.addActionListener(this);
        add(bNon);

        JButton bouton = new JButton("Euh...");
        bouton.addActionListener(this);
        add(bouton);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == bOui)
            System.out.println("Il dit Oui");
        else if (e.getSource() == bNon)
            System.out.println("Il dit Non");
        else
            System.out.println("Il ne sait pas");
    }

    public static void main(String[] args) {
        JFrame cadre = new JFrame("Boutons");
        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cadre.getContentPane().add(new TroisBoutons());
        cadre.pack();
        cadre.setVisible(true);
    }
}
```

90. Cette manière de faire est plus efficace, puisqu'on remplace les comparaisons « `equals` » par des comparaisons « `==` », mais cela oblige à déclarer des collections de variables d'instance.

ItemListener

Il se produit un « événement d'item » lorsqu'un item est sélectionné ou déselectionné. Par *item* nous entendons ici un élément d'une liste déroulante, d'une liste de choix, une case à cocher, etc. Une seule méthode dans cette interface :

`void itemStateChanged(ItemEvent e)` Appelée lorsqu'un item a été sélectionné ou déselectionné.

AdjustmentListener

Cette interface concerne les actions faites sur des barres de défilement. Une seule méthode :

`void adjustmentValueChanged(AdjustmentEvent e)` Appelée lorsque le curseur de la barre de défilement a été déplacé (quelle qu'en soit la raison : on a tiré le curseur, on a cliqué sur une des flèches de la barre ou bien on a cliqué sur la barre en dehors du curseur).

En général, à la suite d'un tel événement on interroge la barre de défilement à l'aide de la méthode `getValue` de la classe `Scrollbar`. Dans le cas où l'on dispose de plusieurs barres de défilement on peut utiliser la méthode `Object getSource()` pour identifier celle qui est la source de l'événement.

Ces événements ne sont pas d'un usage aussi fréquent qu'on pourrait le penser car, dans la plupart des applications, les barres de défilement des fenêtres sont gérées par la machinerie interne des objets `JScrollPane`.

TextListener

Les événements de cette interface notifient les modifications du texte en cours de saisie dans un champ ou une zone de texte. Une seule méthode :

`void textValueChanged(TextEvent e)` Appelée lorsque le texte concerné a changé.

On notera que les champs de texte (`JTextField` et `JPasswordField`) produisent également un événement action lorsque l'utilisateur presse la touche « Entrée », ce qui est souvent l'événement réellement guetté.

WindowListener

Les événements que cette interface concerne sont les actions sur une fenêtre : ouverture et fermeture et, dans le cas d'un cadre ou d'un dialogue, les actions sur les éléments de la barre de titre. Ce qui donne les méthodes :

`void windowClosing(WindowEvent e)` Appelée lorsque l'utilisateur essaye de fermer la fenêtre en agissant sur son « menu système » ou sa case de fermeture. Dans le cas du cadre principal d'une application c'est ici qu'on doit mettre l'éventuel code de confirmation avant terminaison, voir l'exemple ci-dessous.

`void windowActivated(WindowEvent e)` Appelée lorsque la fenêtre est rendue active.

`void windowDeactivated(WindowEvent e)` Appelée lorsqu'une fenêtre cesse d'être la fenêtre active.

`void windowClosed(WindowEvent e)` Appelée lorsque la fenêtre a été fermée, suite à un appel de sa méthode `dispose`. Ne pas confondre cet événement, qui n'a de sens que pour une fenêtre fille (après la fermeture du cadre principal il n'y a plus d'application) avec l'événement `windowClosing`.

`void windowOpened(WindowEvent e)` Appelée lorsqu'une fenêtre est rendue visible pour la première fois.

`void windowIconified(WindowEvent e)` Appelée lorsqu'une fenêtre est minimisée (iconifiée).

`void windowDeiconified(WindowEvent e)` Appelée lorsqu'une fenêtre qui avait été minimisée (iconifiée) retrouve une taille normale.

11.5.4 Exemple : fermeture "prudente" du cadre principal

Voici une illustration classique de la manière de détecter et traiter les événements *Window* : demander à l'utilisateur une confirmation avant de fermer le cadre principal et mettre fin à l'application (voyez la figure 12). Bien sûr, dans une vraie application, au lieu des quatre lignes à partir de `JPanel panneau = new JPanel();` on verrait l'ajout au cadre de choses plus intéressantes qu'un simple panneau vide.

Très intéressant, cet exemple montre également l'emploi des très utiles « boîtes de question » (`JOptionPane`).

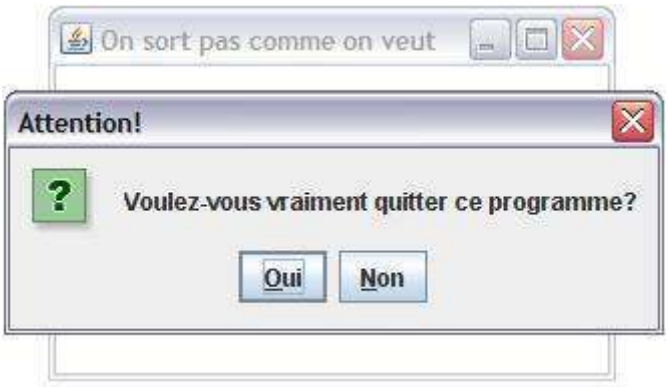


FIGURE 12 – Confirmation avant fermeture

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

public class CadreAvecSortieGardee extends JFrame {

    public CadreAvecSortieGardee(String titre) {
        super(titre);
        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                sortiePrudente();
            }
        });

        JPanel panneau = new JPanel();
        panneau.setPreferredSize(new Dimension(200, 150));
        panneau.setBackground(Color.WHITE);
        getContentPane().add(panneau);

        pack();
        setVisible(true);
    }

    void sortiePrudente() {
        if (JOptionPane.showConfirmDialog(this,
                "Voulez-vous vraiment quitter ce programme?",
                "Attention!",
                JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION)
            System.exit(0);
    }

    public static void main(String[] args) {
        new CadreAvecSortieGardee("On sort pas comme on veut");
    }
}
```

11.6 Peindre et repeindre

11.6.1 La méthode `paint`

Les composants graphiques doivent être peints (sur l'écran) : une première fois lors de leur affichage initial, ensuite chaque fois qu'ils ont été totalement ou partiellement modifiés, effacés, agrandis, etc.

Tout composant qui n'est pas une fenêtre est forcément inclus dans un conteneur qui, s'il n'est pas lui-même une fenêtre, doit à son tour être inclus dans un autre conteneur, lui-même peut-être inclus dans un troisième, etc. Cette chaîne d'inclusions se poursuit jusqu'à une fenêtre, le seul conteneur qui n'est pas obligé d'être inclus dans un autre. Or, la gestion que fait la machine Java des fenêtres (en compétition avec les autres

fenêtres, Java ou non, visibles à l'écran) lui permet de détecter des situations comme celle-ci : une fenêtre qui était totalement ou partiellement masquée par une autre, vient d'être découverte et, par conséquent, tous ou certains des composants placés dans cette fenêtre doivent être redessinés. La machine Java met alors dans une certaine « file d'attente de choses à faire » une requête indiquant qu'il faut repeindre les composants qui se trouvent dans la partie endommagée, qu'il faut restaurer [91].

Aussi longtemps que l'apparence standard d'un composant vous convient, vous n'avez pas à vous préoccuper de sa peinture : le composant prend soin de lui même. Là où cette question devient intéressante c'est lorsqu'on considère des composants « personnalisés », dont l'apparence graphique souhaitée n'est pas celle que leur classe prévoit par défaut. Souvent cela veut dire que le composant est un panneau (`JPanel`) et qu'on doit y montrer un dessin formé de tracés élémentaires (segments, arcs, rectangles et ovales pleins, etc.) ou bien une image obtenue à partir d'un fichier d'un format reconnu par Java, comme *GIF* ou *JPEG*.

C'est la méthode `paint` qui a la responsabilité de peindre un composant. Tous les composants en ont une version, ne serait-ce que celle qu'ils ont héritée de la classe `Component`. Pour qu'un composant ait un dessin personnalisé il faut et il suffit de redéfinir cette méthode dans la classe du composant [92].

La méthode `paint` n'est jamais explicitement appelée par le programme ; au lieu de cela, c'est la machine Java qui se charge de l'appeler, chaque fois que l'apparence du composant doit être refaite. Cela arrive principalement dans trois sortes de situations, dont seule la troisième est à notre portée :

- le composant doit être repeint suite à un événement extérieur à notre application, voire même à la machine Java (exemple : une autre fenêtre est apparue devant le composant, puis a disparu),
- le composant doit être repeint à cause d'un événement qui s'adresse bien à l'interface graphique de notre application mais qui est pris en charge par un élément ou une super-classe du composant (exemple : l'utilisateur a changé la taille de la fenêtre, en agissant sur son bord),
- le composant doit être repeint car les données dont il exhibe une représentation ont changé (la machine Java ne peut pas deviner cela, nous sommes les seuls à le savoir).

L'argument de la méthode `paint` étant un objet de type `Graphics`, il nous fait pour commencer expliquer ce qu'est un tel objet.

11.6.2 Les classes `Graphics` et `Graphics2D`

Un *contexte graphique* est un objet qui encapsule l'ensemble des informations et des outils nécessaires pour effectuer des opérations graphiques. Les contextes graphiques sont instances de la classe `Graphics`, ou d'une de ses sous-classes comme `Graphics2D`.

• L'*état* d'un tel objet se manifeste à travers un ensemble de méthodes *get* (les méthodes *set* correspondantes existent) dont les plus importantes sont :

`Shape getClip()` donne la *zone de coupe* courante, c'est-à-dire la région en dehors de laquelle les opérations graphiques sont sans effet,

`Rectangle getClipRect()` donne l'enveloppe rectangulaire de la zone de coupe (voir ci-dessus),

`Color getColor()` donne la couleur qui sera employée par les opérations graphiques, aussi bien pour tracer des traits que pour remplir des figures ou pour « dessiner » des textes,

`Font getFont()` donne la police de caractères employée pour écrire des textes,

`FontMetrics getFontMetrics()` un objet FontMetrics qui représente un ensemble de mesures à propos de la police de caractères courante.

• Le *comportement* d'un objet `Graphics` est constitué par ses opérations graphiques. Parmi les principales :

`void clearRect(int x, int y, int width, int height)` efface le rectangle spécifié en le peignant avec la couleur de fond (background) du composant,

`void copyArea(int x, int y, int width, int height, int dx, int dy)` copie une portion de l'image en la plaçant à un endroit défini par `dx` et `dy`,

`Graphics create()` crée un contexte graphique qui est un clone de celui sur lequel cette méthode est appelée,

91. Ces requêtes sont affectées de la priorité la plus basse possible, si bien que la machine ne s'occupera de repeindre les composants que lorsque elle n'aura vraiment rien d'autre à faire ; de plus, Java ne conserve dans la file d'attente qu'un exemplaire au plus d'une telle requête. On évite ainsi les séries de peintures successives que provoquerait la prise en charge prioritaire de ces requêtes, entraînant un travail inutile et des clignotements désagréables.

92. Conséquence pratique : si un composant doit avoir un dessin personnalisé, alors il devra être instance d'une classe spécialement définie à cet effet.

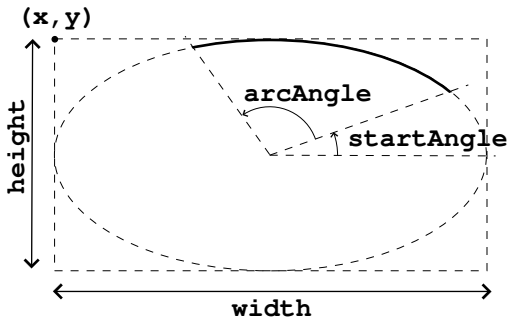


FIGURE 13 – Tracé d'un arc

`void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)`
trace un arc d'ellipse mesurant `arcAngle` degrés et commençant au point défini par `startAngle` degrés (l'origine est « à 3 heures »). Cette ellipse est inscrite dans le rectangle de largeur `width` et de hauteur `height` dont le coin supérieur gauche est le point de coordonnées `x` et `y` (cf. figure 13).

`boolean drawImage(Image img, int x, int y, ImageObserver observer)` dessine l'image `img` en plaçant son coin supérieur gauche au point de coordonnées `x`, `y`.
`observer` est un objet destinataire des notifications sur l'état d'avancement de l'image qui – par exemple dans le cas d'images obtenues sur le web – peut se charger très lentement ; si la question est sans intérêt (par exemple s'il s'agit d'images qui n'arrivent pas par le réseau) on peut mettre ici n'importe quel composant.

`void drawLine(int x1, int y1, int x2, int y2)` trace un segment de droite joignant le point (x_1, y_1) au point (x_2, y_2),

`void drawRect(int x, int y, int width, int height)` trace le bord d'un rectangle de largeur `width` et de hauteur `height` dont le coin supérieur gauche est placé au point de coordonnées `x` et `y`.

`void drawOval(int x, int y, int width, int height)` trace le bord d'une ellipse définie par son enveloppe rectangulaire (voyez `drawRect` ci-dessus),

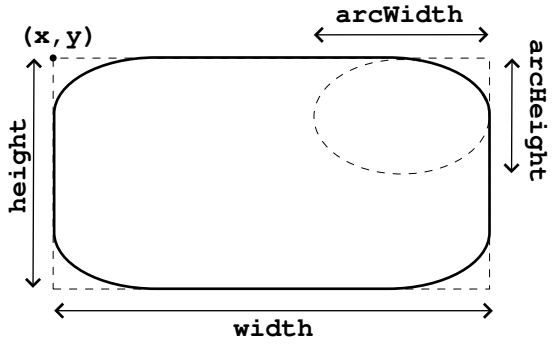


FIGURE 14 – Un rectangle aux coins arrondis

`void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`
trace le bord d'un rectangle (`x`,`y`,`width`,`height`) dont les coins sont des quarts de l'ellipse inscrite dans un rectangle de largeur `arcWidth` et de hauteur `arcHeight` (cf. figure 14).

`void drawString(String str, int x, int y)` trace la chaîne `str` de telle manière que le coin *inférieur* gauche de l'enveloppe rectangulaire du premier caractère soit sur le point de coordonnées `x` et `y`,

`void fillRect(int x, int y, int width, int height)` peint l'intérieur d'un rectangle (cf. `drawRect`) en utilisant la couleur courante,

`void fillOval(int x, int y, int width, int height)` peint l'intérieur d'une ellipse (cf. `drawOval`) en utilisant la couleur courante,

`void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`
peint l'intérieur d'un rectangle aux coins arrondis (cf. `drawRoundRect`) en utilisant la couleur courante,

`void translate(int x, int y)` place l'origine des coordonnées sur le point de coordonnées `x` et `y` (relatives à l'origine précédente).

A propos de Graphics2D

La classe `Graphics2D` étend la classe `Graphics` en permettant un contrôle beaucoup plus sophistiqué de la géométrie, les systèmes de coordonnées, les transformations graphiques, la couleur, l'aspect des textes, etc.

Il faut savoir que la valeur du paramètre de type `Graphics` passé à la méthode `paint` est en réalité un objet `Graphics2D`. Lorsque les opérations « améliorées » de la classe `Graphics2D` sont nécessaires, la méthode `paint` commence donc de la manière suivante, qui est donc légitime :

```
public void paint(Graphics g) {
    super.paint(g);
    Graphics2D g2d = (Graphics2D) g;
    ...
    opérations graphiques mettant en œuvre l'objet g2d
    (toutes les opérations des classes Graphics et Graphics2D sont donc légitimes)
    ...
}
```

11.6.3 Exemple : la *mauvaise* manière de dessiner

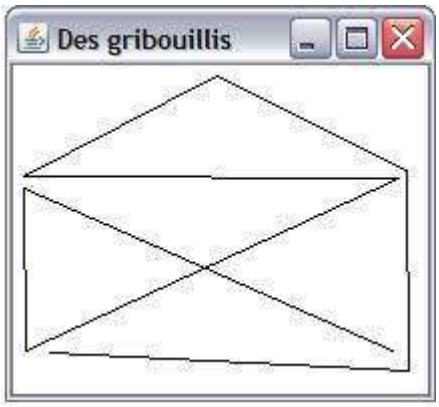


FIGURE 15 – Gribouillages...

Donnons-nous le problème suivant (voyez la figure 15) : dessiner une ligne polygonale que l'utilisateur définit en cliquant successivement sur les points qu'il souhaite placer comme sommets de la construction.

Première version, nous expliquons plus loin en quoi elle n'est pas bonne :

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

public class UnGribouillis extends JPanel {

    UnGribouillis() {
        setPreferredSize(new Dimension(600, 400));
        setBackground(Color.WHITE);
        addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                clic(e.getX(), e.getY());
            }
        });
    }
```



```
    private int xPrec = -1;     // coordonnées du
    private int yPrec;          // clic précédent

    void clic(int x, int y) {
        if (xPrec >= 0) {
            Graphics g = getGraphics();
            g.drawLine(xPrec, yPrec, x, y);
        }
        xPrec = x;
        yPrec = y;
    }

    public static void main(String[] args) {
        JFrame cadre = new JFrame("Des gribouillis");
        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cadre.getContentPane().add(new UnGribouillis());
        cadre.pack();
        cadre.setVisible(true);
    }
}
```

Le principe de ce programme est simple : à partir du second, chaque clic détecté provoque le tracé d'un segment joignant le point où il s'est produit au point où s'est produit le clic précédent. On dessine donc « à fonds perdu », ce qui est le défaut de cette manière de faire : si la ligne polygonale est endommagée, notre application ne sera pas capable de la redessiner [93].

11.6.4 Exemple : la *bonne* manière de dessiner.

On l'aura compris, la bonne manière de dessiner ne consiste pas à tracer les éléments graphiques au fur et à mesure que leurs paramètres sont connus, mais au contraire à mémoriser l'information requise pour tout (re-)dessiner chaque fois que cela est nécessaire :

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

public class UnAutreGribouillis extends JPanel {

    UnAutreGribouillis() {
        setPreferredSize(new Dimension(600, 400));
        setBackground(Color.WHITE);
        addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                points.add(new Point(e.getX(), e.getY()));
                repaint();
            }
        });
    }

    private Vector points = new Vector();
```

93. Dans certains cas, le système sous-jacent prend sur lui de mémoriser le contenu des fenêtres afin d'être en mesure de les restaurer lorsqu'elles auront été partiellement ou totalement masquées. C'est gentil mais naïf car cela ignore le fait que l'image à montrer n'est peut-être pas la même que celle qui a été sauvegardée.

```
    public void paint(Graphics g) {
        super.paint(g);

        Iterator it = points.iterator();
        if (it.hasNext()) {
            Point p = (Point) it.next();
            while (it.hasNext()) {
                Point q = (Point) it.next();
                g.drawLine(p.x, p.y, q.x, q.y);
                p = q;
            }
        }
    }

    public static void main(String[] args) {
        JFrame cadre = new JFrame("Des gribouillis");
        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cadre.getContentPane().add(new UnAutreGribouillis());
        cadre.pack();
        cadre.setVisible(true);
    }
}
```

Dans ce second programme, la réaction à un clic de la souris n'est plus le dessin d'un segment, mais uniquement la mémorisation des coordonnées du clic suivie d'un appel de la méthode `repaint`[94] pour indiquer que le dessin du composant n'est plus valide (puisque la ligne polygonale possède désormais un sommet de plus, que la figure ne montre pas encore).

Le dessin effectif est l'affaire de la méthode `paint`, dont le programmeur doit écrire la définition mais *jamais l'appel*, car la plupart des appels de cette méthode sont imprévisibles. La création d'un sommet de la ligne polygonale est une exception : `repaint` ayant été appelée, l'appel de `paint` est pour une fois prévisible, mais les autres situations qui requièrent le travail de `paint` (masquage de la fenêtre, redimensionnement, etc.) ne peuvent pas être explicitement prédites et placées parmi les instructions du programme.

NOTE 1. Presque toujours la version redéfinie de la méthode `paint` commence, comme ici, par appeler la version héritée :

```
super.paint(g);
```

Cela est est nécessaire car, sinon, le fond de l'écran ne serait pas repeint et de curieux dysfonctionnements apparaîtraient.

NOTE 2. La méthode `paint` est la seule (bonne) méthode pour peindre les composants de *AWT*. Pour les composants de *Swing*, cependant, la méthode `paint` consiste en des appels successifs de trois méthodes déléguées `paintComponent`, `paintBorder` et `paintChildren`. C'est pourquoi, dans le cas où notre composant a un bord et comporte des sous-composants, il vaut souvent mieux redéfinir la méthode `paintComponent` au lieu de s'en prendre à `paint`. La différence n'est pas bien grande :

```
public class UnAutreGribouillis extends JPanel {
    ...
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        ...
        le reste de la méthode est identique à paint, ci-dessus
        ...
    }
    ...
}
```

11.7 Gestionnaires de disposition

Le placement des composants dans les conteneurs est l'affaire des gestionnaires de disposition, ou *layout managers*, qui se chargent :

94. Contrairement à ce qu'on pourrait penser, la méthode `repaint` ne provoque pas un appel immédiat de `paint`. Au lieu de cela, elle se limite à mettre dans la « file des choses à faire » une requête de peinture, s'il n'y en avait pas déjà une. Ce n'est que quand toutes les autres requêtes de cette file auront été traitées (c'est-à-dire quand la machine Java n'aura rien d'autre à faire) que la méthode `paint` sera effectivement appelée et le composant redessiné.

– du placement initial des composants, lors des appels de la méthode `add`,
– le cas échéant, de donner une taille et une forme à chaque composant, en fonction de sa « taille préférée », de son contenu et de sa disposition relativement aux autres composants dans le même conteneur,
– du repositionnement des composants lorsque la taille ou la forme du conteneur change.

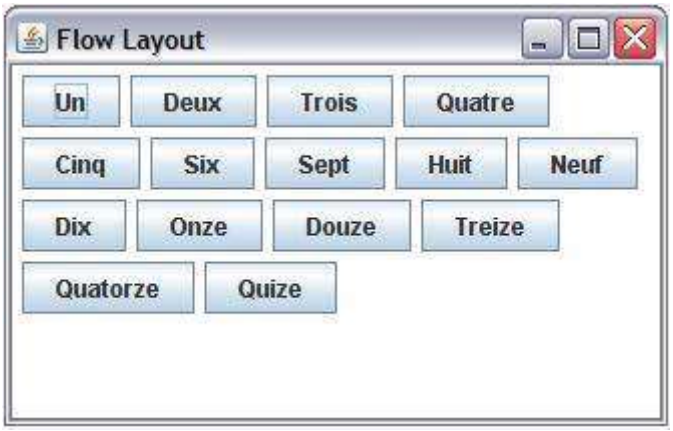


FIGURE 16 – Un panneau avec un gestionnaire `FlowLayout`

Absence de gestionnaire

Pour commencer, évacuons une question très secondaire : il est possible (mais non recommandé) de se passer complètement de gestionnaire de disposition. Cela doit être explicitement demandé, par l'instruction

conteneur`.setLayout(null);`

A la suite de cette commande, chaque composant devra être explicitement dimensionné et positionné lors de son ajout au conteneur (bonjour les calculs avec des pixels !). Les méthodes pour cela sont

`void setLocation(int x, int y)`, `void setLocation(Point p)` définit la position du coin supérieur gauche du composant (ou plutôt de son enveloppe rectagulaire), relativement à une origine qui est placée au coin supérieur gauche du conteneur,

`void setSize(int width, int height)`, `void setSize(Dimension d)` définit les dimensions du composant (ou plutôt de son enveloppe rectangulaire),

`void setBounds(int x, int y, int width, int height)`, `void setBounds(Rectangle r)` remplit simultanément les fonctions de `setLocation` et celles de `setSize`.

Lorsque les composants ont été placés manuellement il y a intérêt à interdire les changements de taille et de forme du conteneur. Cela s'obtient par l'expression

conteneur`.setResizable(false);`

ATTENTION. Certains programmeurs débutants, rebutés par la difficulté qu'il y a parfois à obtenir d'un gestionnaire de disposition qu'il fasse ce qu'on veut, choisissent de placer manuellement les composants. L'expérience montre que, sauf de rares exceptions, *se passer de gestionnaire de disposition n'est pas la manière la plus simple de réaliser une interface graphique.*

11.7.1 `FlowLayout`

Un gestionnaire `FlowLayout` (voir figure 16) dispose les composants par lignes, de la gauche vers la droite et du haut vers le bas.

Par défaut les composants sont centrés horizontalement et écartés entre eux de 5 pixels, mais on peut changer cela au moment de l'instanciation. Par exemple, le panneau de la figure 16 a été construit par la séquence suivante :

```
...
JPanel panneau = new JPanel();
panneau.setLayout(new FlowLayout(FlowLayout.LEFT));
panneau.add(new JButton("Un"));
panneau.add(new JButton("Deux"));
...
```

Le gestionnaire de disposition par défaut d'un `JPanel` est `FlowLayout`.

11.7.2 `BorderLayout`

Un BorderLayout (voir figure 17) distingue cinq zones dans le conteneur auquel il est attaché : le nord, le sud, l'est, l'ouest et le centre.

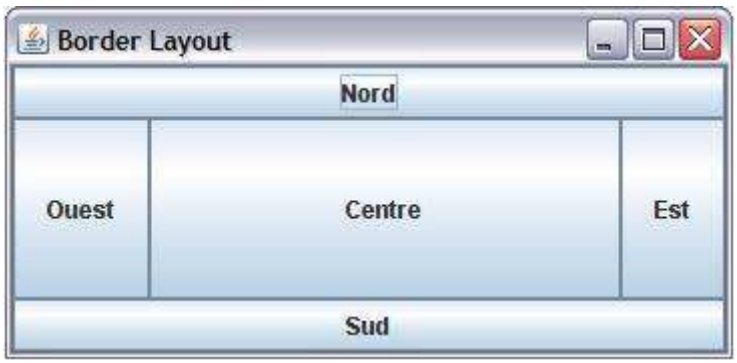


FIGURE 17 – Un panneau avec un gestionnaire `BorderLayout`

Le nord et le sud, lorsqu'ils sont présents, prennent toute la largeur du conteneur et ont la plus petite hauteur qui convient aux composants qu'ils contiennent. L'est et l'ouest, lorsqu'ils sont présents, prennent toute la hauteur du conteneur moins les parties éventuellement occupées par le nord et le sud, et ont la largeur minimale qui respecte les composants qu'ils contiennent. Enfin, le centre prend toute la place restante.

Par exemple, le panneau de la figure 17 a été construit par la séquence

```
...
JPanel panneau = new JPanel();
panneau.setLayout(new BorderLayout());
panneau.add(new JButton("Haut"),   BorderLayout.NORTH);
panneau.add(new JButton("Gauche"), BorderLayout.EAST);
panneau.add(new JButton("Bas"),    BorderLayout.SOUTH);
panneau.add(new JButton("Droite"), BorderLayout.WEST);
panneau.add(new JButton("Centre"), BorderLayout.CENTER);
...
```

Le gestionnaire de disposition par défaut du panneau de contenu d'un `JFrame` ou d'un `JDialog` est `BorderLayout`.

11.7.3 `GridLayout`

Un gestionnaire `GridLayout` (voir figure 18) organise les composants selon une grille rectangulaire ayant un nombre de lignes et de colonnes convenus lors de la création du gestionnaire. Toutes les cases de cette grille ont les mêmes dimensions.

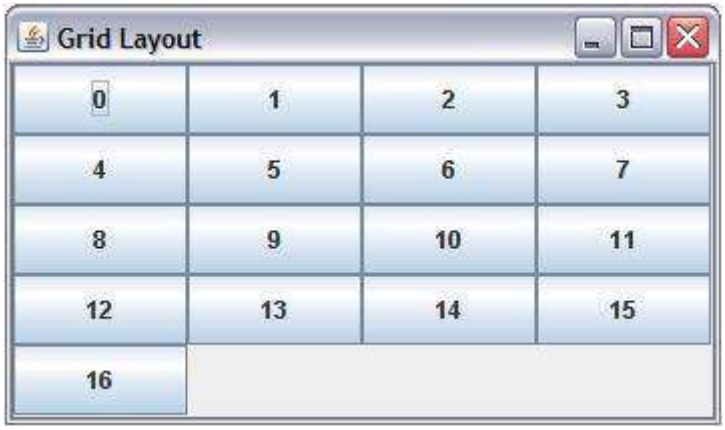


FIGURE 18 – Un panneau avec un gestionnaire `GridLayout` à 4 colonnes

Par exemple, le panneau de la figure 18 a été construit par la séquence suivante :

```
...
JPanel panneau = new JPanel();
panneau.setLayout(new GridLayout(5, 4));
for (int i = 0; i <= 16; i++)
    panneau.add(new JButton("" + i));
...
```

11.7.4 `GridBagLayout`

Un gestionnaire `GridBagLayout` est plus puissant, mais bien plus complexe que les précédents. Il se base sur un quadrillage *imaginaire* du conteneur tel que l'espace occupé par chaque composant soit une région rectangulaire formée par la réunion d'un certain nombre de cases de ce quadrillage.

Lors de son ajout au conteneur, chaque composant est associé à une certaine *contrainte* qui spécifie quelles cases du quadrillage imaginaire le composant occupe et comment il les occupe. Ensemble, toutes ces contraintes définissent les dimensions des lignes et des colonnes du quadrillage.

Les *contraintes* sont des instances de la classe `GridBagConstraints`, dont les principaux champs sont :

`gridx` : numéro de la colonne du quadrillage (la première colonne porte le numéro 0) où se place l'angle supérieur gauche du composant.

`gridy` : numéro de la ligne du quadrillage (la première ligne porte le numéro 0) où se place l'angle supérieur gauche du composant.

`gridwidth` : nombre de colonnes du quadrillage sur lesquelles le composant s'étend.

`gridheight` : nombre de lignes du quadrillage sur lesquelles le composant s'étend.

`weightx` : nombre exprimant la largeur de la colonne que le composant occupe ; peu importe l'unité, pourvu que ce soit la même pour toutes les colonnes.

Mettez 0 si ce composant s'étend sur plusieurs colonnes ou si la largeur peut se déduire de la contrainte d'un autre composant de cette colonne.

`weighty` : nombre exprimant la hauteur de la ligne que le composant occupe ; peu importe l'unité, pourvu que ce soit la même pour toutes les lignes.

Mettez 0 si ce composant s'étend sur plusieurs lignes ou si la hauteur peut se déduire de la contrainte d'un autre composant de cette ligne.

`fill` : indication de la manière dont le composant doit remplir sa zone. Les valeurs possibles sont :
- `GridBagConstraints.NONE` : le composant doit garder sa taille propre,
- `GridBagConstraints.HORIZONTAL` : le composant doit remplir toute la largeur de sa zone,
- `GridBagConstraints.VERTICAL` : le composant doit remplir toute la hauteur de sa zone,
- `GridBagConstraints.BOTH` : le composant doit remplir toute sa zone.

`anchor` : position du composant dans sa zone, s'il ne la remplit pas entièrement. Les valeurs possibles sont : `GridBagConstraints.NORTH` (au nord), `GridBagConstraints.NORTHEAST` (au nord-est), `GridBagConstraints.EAST` (à l'est), `GridBagConstraints.SOUTHEAST` (au sud-est), etc.

11.7.5 Exemple : un panneau muni d'un `GridBagLayout`

A titre d'exemple [95], proposons-nous de construire un panneau contenant les huit composants montrés sur la figure 19.

La figure 20 montre le quadrillage (qui n'existe que dans la tête du programmeur) par rapport auquel les composants sont placés. Pour chacun on doit construire une contrainte ; cela est rapidement pénible, à moins de se donner une méthode auxiliaire comme notre méthode `ajout` :

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

public class TestGridBagLayout extends JFrame {
```

95. Cet exemple est démonstratif mais inutilisable. D'une part, ce n'est pas réaliste de mettre ce type de composants dans le cadre principal d'une application ; cet exemple correspond plus à une boîte de dialogue (`JDialog`) qu'à un cadre (`JFrame`).
D'autre part, cet exemple est construit avec des composants anonymes, ce qui rend impossible la récupération des informations que l'utilisateur donne en saisissant des textes dans les champs de texte ou en pressant les boutons.

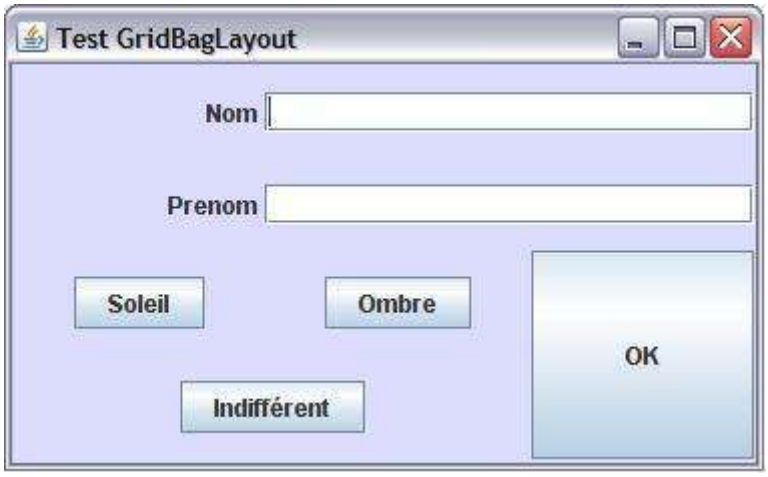


FIGURE 19 – Un panneau avec un gestionnaire `GridBagLayout`

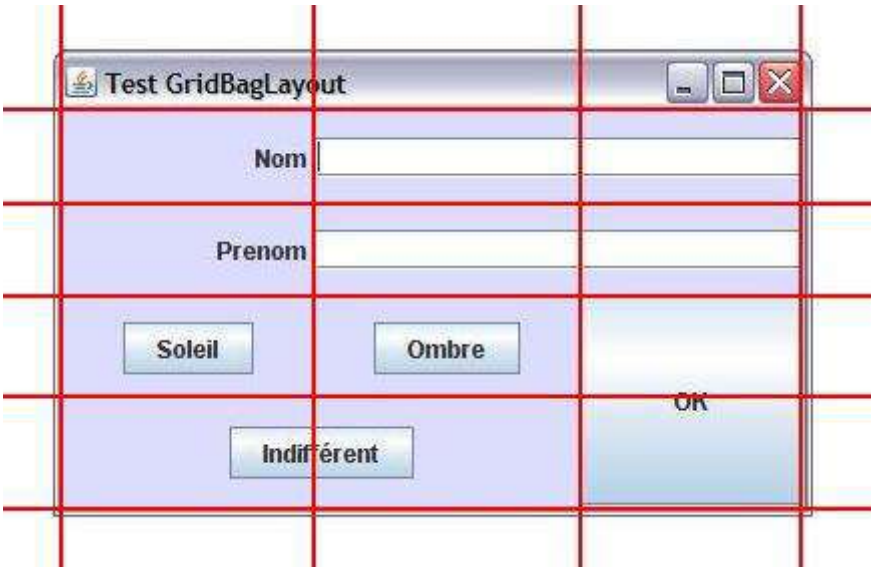


FIGURE 20 – Le quadrillage sous-jacent au gestionnaire `GridBagLayout` de la figure 19

```
TestGridBagLayout() {
    super("Test GridBagLayout");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JPanel panneau = new JPanel();
    panneau.setBackground(new Color(220, 220, 255));
    panneau.setPreferredSize(new Dimension(400, 300));
    panneau.setLayout(new GridBagLayout());
    ajout(panneau, new JLabel("Nom "),
        0, 0, 1, 1, 10, 10, GridBagConstraints.NONE, GridBagConstraints.EAST);
    ajout(panneau, new JLabel("Prenom "),
        0, 1, 1, 1, 0, 10, GridBagConstraints.NONE, GridBagConstraints.EAST);
    ajout(panneau, new JTextField(),
        1, 0, 2, 1, 0, 0, GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    ajout(panneau, new JTextField(),
        1, 1, 2, 1, 0, 0, GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
```

```
            ajout(panneau, new JButton("Soleil"),
                0, 2, 1, 1, 0, 10, GridBagConstraints.NONE, GridBagConstraints.CENTER);
            ajout(panneau, new JButton("Ombre"),
                1, 2, 1, 1, 10, 0, GridBagConstraints.NONE, GridBagConstraints.CENTER);
            ajout(panneau, new JButton("Indifférent"),
                0, 3, 2, 1, 0, 10, GridBagConstraints.NONE, GridBagConstraints.CENTER);
            ajout(panneau, new JButton("OK"),
                2, 2, 1, 2, 10, 0, GridBagConstraints.BOTH, GridBagConstraints.CENTER);

            getContentPane().add(panneau);
            pack();
            setVisible(true);
        }

        private void ajout(Container conteneur, Component composant,
                int gridx, int gridy, int gridwidth, int gridheight,
                int weightx, int weighty, int fill, int anchor) {
            GridBagConstraints contrainte = new GridBagConstraints();
            contrainte.gridx      = gridx;
            contrainte.gridy      = gridy;
            contrainte.gridwidth  = gridwidth;
            contrainte.gridheight = gridheight;
            contrainte.weightx    = weightx;
            contrainte.weighty    = weighty;
            contrainte.fill       = fill;
            contrainte.anchor     = anchor;
            conteneur.add(composant, contrainte);
        }

        public static void main(String[] args) {
            new TestGridBagLayout();
        }
    }
```

Ci-dessus le programmeur a fixé à 10 la largeur de chaque colonne et la hauteur de chaque ligne, le panneau tout entier ayant une largeur de 30 (cela se déduit des composants *Nom*, *Soleil* et *OK*) et une hauteur de 40 (cela découle de *Nom*, *Prénom*, *Soleil* et *Indifférent*). L'unité de mesure est sans importance, la seule information qu'on a voulu donner ainsi est que la largeur de chaque colonne est le tiers de la largeur totale et la hauteur de chaque ligne est le quart de la hauteur totale.

11.7.6 Exemple : imbrication de gestionnaires de disposition

Les `GridBagLayout` ne sont pas les seuls outils pour la construction de gestionnaires d'interfaces graphiques complexes. Une autre manière de monter de telles interfaces, plus souple et – avec de l'expérience – plus simple, consiste à imbriquer plusieurs panneaux (objets `JPanel`) les uns dans les autres, chacun muni d'un gestionnaire adéquat.

A ce propos on consultera avec profit l'exemple montré à la section 11.8.2, *Construire une boîte de dialogue*.

11.7.7 Bordures

Il est possible de créer une bordure autour d'un composant. La manière la plus simple est de la faire fabriquer par « l'usine de bordures » (`BorderFactory`) selon le schéma :

unComposant`.setBorder(BorderFactory.create`*Xxx*`Border(`*paramètres de la bordure*`))`

Ci-dessus, Xxx représente des mots qui identifient le type de bordure souhaité. La figure 21 montre diverses sortes de bordures, et le jargon avec lequel elles sont nommées dans la classe `BorderFactory`.

11.8 Composants prédéfinis

Nous ne ferons pas ici une présentation systématique des différentes, et nombreuses, classes de composants disponibles dans *AWT* et *Swing*. Cela est très bien fait dans le tutoriel Java (`http://java.sun.com/`

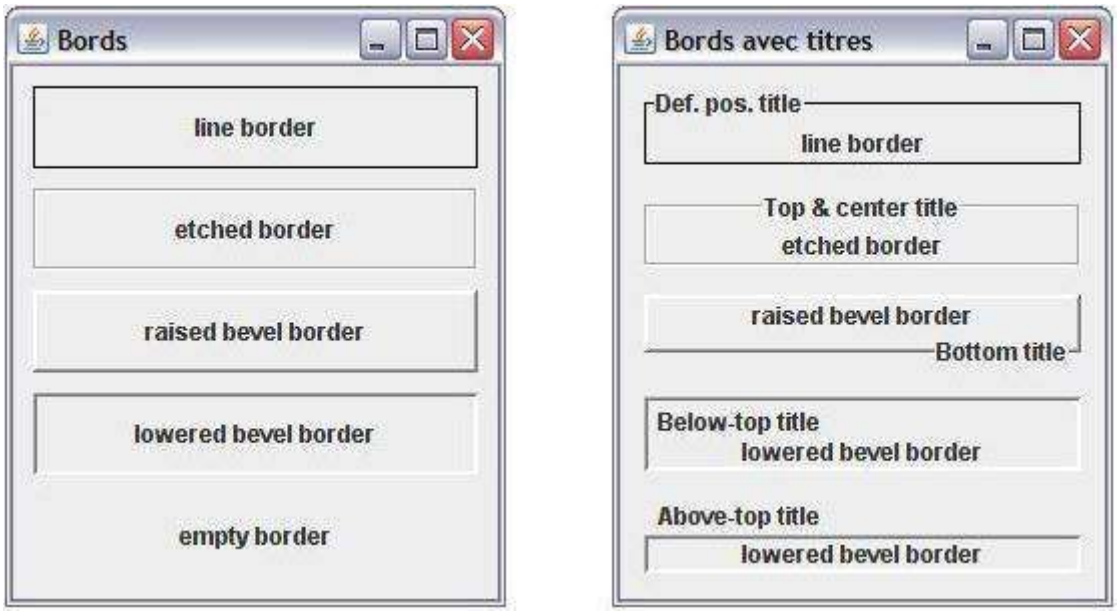


FIGURE 21 – Des bordures, sans et avec titres

`tutorial/`). En particulier, à la section *A Visual Index to the Swing Components* (leçon *Using Swing Components* de la séquence *Creating a GUI with JFC/Swing*) on trouve une présentation extrêmement parlante de tous les composants disponibles.

Au lieu de cela nous allons montrer le mode d'emploi de certains des composants les plus utilisés, à travers une application très naïve gérant le carnet d'adresses d'un club sportif.

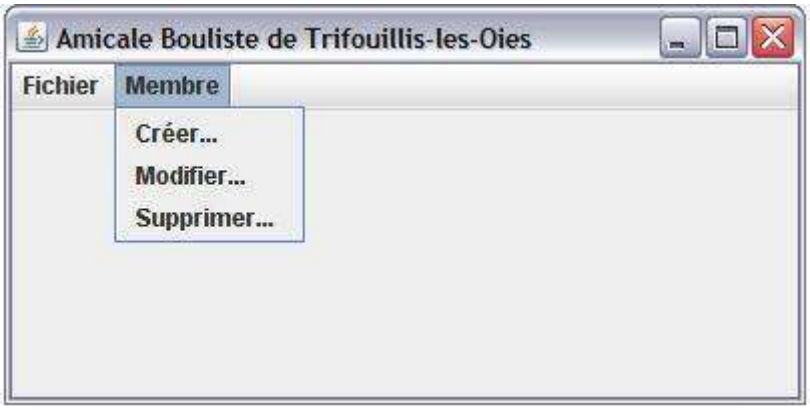


FIGURE 22 – Cadre avec menus

11.8.1 Exemple : mise en place et emploi de menus

La première version de notre programme illustre la mise en place des menus, voyez la figure 22. Les actions sur les menus sont détectées, mais les réactions en sont, pour le moment, des appels de méthodes vides :

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

public class ClubSportif extends JFrame {

    ClubSportif() {
        super("Amicale Bouliste de Trifouillis-les-Oies");
        setSize(400, 200);
        initialiserLesMenus();
        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
```



```
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                quitter();
            }
        });
        setVisible(true);
    }

    private void initialiserLesMenus() {
        JMenuBar barre = new JMenuBar();
        setJMenuBar(barre);
        JMenu menu = new JMenu("Fichier");
        barre.add(menu);
        JMenuItem item = new JMenuItem("Ouvrir...");
        menu.add(item);
        item.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                ouvrirFichier();
            }
        });
        item = new JMenuItem("Enregistrer...");
        menu.add(item);
        item.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                enregistrerFichier();
            }
        });
        menu.addSeparator();
        item = new JMenuItem("Quitter");
        menu.add(item);
        item.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                quitter();
            }
        });

        menu = new JMenu("Membre");
        barre.add(menu);
        item = new JMenuItem("Créer...");
        menu.add(item);
        item.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                creerMembre();
            }
        });
        item = new JMenuItem("Modifier...");
        menu.add(item);
        item.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                modifierMembre();
            }
        });
        item = new JMenuItem("Supprimer...");
        menu.add(item);
        item.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                supprimerMembre();
            }
        });
    }
```

```
    private void creerMembre() {
    }

    private void modifierMembre() {
    }

    private void supprimerMembre() {
    }

    private void ouvrirFichier() {
    }

    private void enregistrerFichier() {
    }

    private void quitter() {
        System.exit(0);
    }

    public static void main(String[] args) {
        new ClubSportif();
    }
}
```

NOTE. Pour réagir aux actions faites sur les menus nous avons choisi de donner un auditeur distinct à chaque item de menu. Une autre solution aurait consisté à donner le même auditeur à tous les item. Cela nous aurait obligé à écrire une méthode `actionPerformed` ressemblant à ceci [96] :

```
...
public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("Ouvrir..."))
        ouvrirFichier();
    else if (e.getActionCommand().equals("Enregistrer..."))
        enregistrerFichier();
    else if (e.getActionCommand().equals("Quitter"))
        quitter();
    else if (e.getActionCommand().equals("Créer..."))
        creerMembre();
    else if (e.getActionCommand().equals("Modifier..."))
        modifierMembre();
    else if (e.getActionCommand().equals("Supprimer"))
        supprimerMembre();
    else
        throw new RuntimeException("Action command inattendue: "
                                                    + e.getActionCommand());
}
...
```

11.8.2 Exemple : construire une boite de dialogue

Concevoir et mettre en place les boîtes de dialogue (dans d'autres milieux on les appelle *masques de saisie*) est certainement la partie la plus ennuyeuse du développement d'une interface-utilisateur graphique. En Java la chose est rendue encore plus pénible par la présence des gestionnaires de disposition qui, tout en étant très utiles pour gérer les changements de taille et de forme des conteneurs, manifestent souvent une grande réticence à placer les composants selon la volonté du programmeur.

Pour illustrer cette question nous allons ajouter à notre application une boîte de dialogue pour saisir les informations qui caractérisent un membre du club. Pour commencer, nous ne nous occupons ici que d'assembler les composants, nous verrons à la section suivante comment utiliser la boîte pour acquérir des informations.

96. Cette manière de détecter les actions sur les menus est certainement moins efficace que la précédente. Mais l'efficacité a-t-elle une quelconque importance, quand il s'agit de réagir à des gestes de l'utilisateur ?

FIGURE 23 – Boîte de dialogue de saisie

La figure 23 montre l'aspect de la boîte lors de son utilisation. La figure 24 montre les différents panneaux (`JPanel`), munis de gestionnaires de dispositions distincts, que nous avons dû imbriquer les uns dans les autres pour obtenir le placement des composants que montre la figure 23.

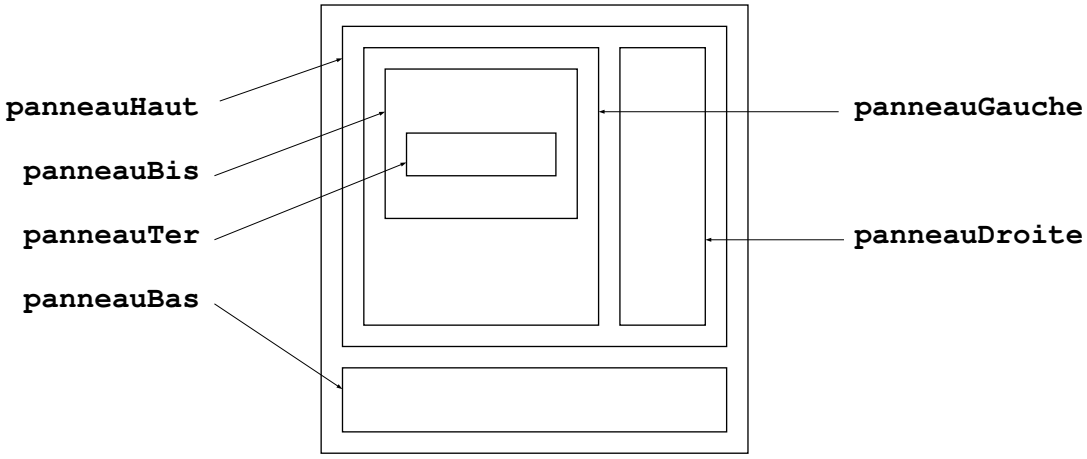


FIGURE 24 – Imbrication de panneaux pour réaliser la boite de dialogue de la figure 23

Voici le constructeur de notre classe :

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

public class BoiteDialogueMembre extends JDialog implements Sports {
    JTextField    champNom, champPrenom;
    JRadioButton  sexeMasculin, sexeFeminin;
    JTextArea     zoneAdresse;
    JCheckBox[]   casesSports;
    JButton       boutonOK, boutonAnnuler;
    boolean       donneesAcquises;
```

```
    BoiteDialogueMembre(JFrame cadre) {
        super(cadre, "Saisie d'un membre", true);

        zoneAdresse = new JTextArea(5, 20);

        JPanel panneauTer = new JPanel();
        panneauTer.add(new JLabel("Sexe"));
        panneauTer.add(sexeMasculin = new JRadioButton("Homme"));
        panneauTer.add(sexeFeminin  = new JRadioButton("Femme"));
        ButtonGroup groupe = new ButtonGroup();
        groupe.add(sexeMasculin);
        groupe.add(sexeFeminin);

        JPanel panneauBis = new JPanel();
        panneauBis.setLayout(new GridLayout(6, 1));
        panneauBis.add(new JLabel("Nom"));
        panneauBis.add(champNom = new JTextField(20));
        panneauBis.add(new JLabel("Prénom"));
        panneauBis.add(champPrenom = new JTextField(20));
        panneauBis.add(panneauTer);
        panneauBis.add(new JLabel("Adresse"));

        JPanel panneauGauche = new JPanel();
        panneauGauche.setLayout(new BorderLayout());
        panneauGauche.add(panneauBis, BorderLayout.CENTER);
        panneauGauche.add(new JScrollPane(zoneAdresse), BorderLayout.SOUTH);

        JPanel panneauDroite = new JPanel();
        panneauDroite.setLayout(new GridLayout(nomSport.length, 1));
        casesSports = new JCheckBox[nomSport.length];
        for (int i = 0; i < nomSport.length; i++)
            panneauDroite.add(casesSports[i] = new JCheckBox(nomSport[i]));

        JPanel panneauHaut = new JPanel();
        panneauHaut.setLayout(new BorderLayout());
        panneauHaut.add(panneauGauche, BorderLayout.CENTER);
        panneauHaut.add(panneauDroite, BorderLayout.EAST);
        panneauHaut.setBorder(BorderFactory.createEmptyBorder(0, 10, 0, 10));

        JPanel panneauBas = new JPanel();
        panneauBas.add(boutonOK = new JButton("   OK   "));
        panneauBas.add(boutonAnnuler = new JButton("Annuler"));
        boutonOK.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                finDuDialogue(true);
            }
        });
        boutonAnnuler.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                finDuDialogue(false);
            }
        });

        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(panneauHaut, BorderLayout.CENTER);
        getContentPane().add(panneauBas, BorderLayout.SOUTH);
    }
    ...
    d'autres membres de cette classe sont montrés dans les sections suivantes
    ...
}
```

Le constructeur de `BoiteDialogueMembre` ne se termine pas par le couple « `pack(); setVisible(true);` » habituel. Ces opérations sont à la charge de celui qui crée une boîte de dialogue, on verra pourquoi.

Le troisième argument, `true`, de l'appel du constructeur de `JDialog`

```
super(cadre, "Saisie d'un membre", true);
```

fait que la boîte de dialogue est *modale* : aussi longtemps qu'elle est visible à l'écran, l'utilisateur ne peut pas agir sur une autre partie de l'interface graphique de l'application.

L'interface `Sports`, mentionnée précédemment, sert juste à mettre en place un ensemble de constantes[97] communes à plusieurs classes :

```
public interface Sports {
    long TENNIS =      1 << 0;   // ou  1
    long SQUASH =      1 << 1;   // ou  2
    long NATATION =    1 << 2;   // ou  4
    long ATHLETISME =  1 << 3;   // ou  8
    long RANDONNEE =   1 << 4;   // ou  16
    long FOOT =        1 << 5;   // ou  32
    long BASKET =      1 << 6;   // ou  64
    long VOLLEY =      1 << 7;   // ou  128
    long PETANQUE =    1 << 8;   // ou  256

    String[] nomSport = { "Tennis", "Squash", "Natation", "Athlétisme",
                          "Randonnée", "Foot", "Basket", "Volley", "Pétanque" };
    long[] valSport = { TENNIS, SQUASH, NATATION, ATHLETISME,
                        RANDONNEE, FOOT, BASKET, VOLLEY, PETANQUE };
}
```

11.8.3 Exemple : saisie de données

Dans la boîte de dialogue que nous développons ici les seuls composants dont les événements sont détectés et traités sont les deux boutons *OK* et *Annuler*[98]. La pression du bouton *OK* déclenche la validation des informations portées par les champs de la boîte de dialogue et, en cas de succès, la fermeture de cette boîte ; le bouton *Annuler* la ferme dans tous les cas. Voici le reste de notre classe `BoiteDialogueMembre` :

```
public class BoiteDialogueMembre extends JDialog implements Sports {
    ...
    private void finDuDialogue(boolean sortieParBoutonOK) {
        donneesAcquises = false;
        if (sortieParBoutonOK) {
            donneesAcquises = validerDonnees();
            if (donneesAcquises)
                dispose();
        }
        else
            dispose();
    }

    private boolean validerDonnees() {
        if (champNom.getText().length() == 0)
            JOptionPane.showMessageDialog(this, "Le champ nom est vide",
                    "Donnée manquante", JOptionPane.ERROR_MESSAGE);
        else if (champPrenom.getText().length() == 0)
            JOptionPane.showMessageDialog(this, "Le champ prénom est vide",
                    "Donnée manquante", JOptionPane.ERROR_MESSAGE);
        else if ( ! sexeMasculin.isSelected() && ! sexeFeminin.isSelected())
            JOptionPane.showMessageDialog(this, "Le sexe n'est pas indiqué",
                    "Donnée manquante", JOptionPane.ERROR_MESSAGE);
        else if (zoneAdresse.getText().length() == 0)
            JOptionPane.showMessageDialog(this, "L'adresse n'est pas indiquée",
                    "Donnée manquante", JOptionPane.ERROR_MESSAGE);
```

97. Ne pas oublier que toutes les variables d'une interface sont, par définition, statiques et finales (i.e. des constantes de classe).

98. Nous l'avons déjà dit, nous développons ici un exemple naïf. Dans une application plus sérieuse nous pourrions détecter les frappes au clavier afin de vérifier les éventuelles contraintes sur les textes tapés.

```
            else
                return true;
            return false;
        }
    }
```

La validation des données faite ici est sommaire : nous nous contentons de vérifier que tous les champs ont été renseignés. Dans les cas réels il y a souvent à faire des validations plus fines.

Il est important de bien comprendre le fonctionnement de la méthode `finDuDialogue`. Lorsque l'utilisateur presse le bouton *Annuler*, ou bien lorsqu'il presse le bouton *OK* et que les données sont trouvées valides, alors l'appel de `dispose()` détruit l'objet graphique correspondant à la boîte de dialogue (i.e. tout ce qui se voit à l'écran), mais l'objet Java (i.e. l'instance de la classe `BoiteDialogueMembre`) n'est pas encore détruit car, comme nous allons le voir, il est la valeur d'une certaine variable (la variable locale `dial` de la méthode `creerMembre` de la classe `ClubSportif`).

Notez que si l'utilisateur appuie sur le bouton *OK* et que les données ne sont pas valides alors il ne se passe rien : la méthode `finDuDialogue` retourne sans voir appelé `dispose`, donc la boîte de dialogue est toujours visible.

Il faut voir enfin que, la boîte étant modale, c'est l'appel de `dispose` fait dans `finDuDialogue` qui met fin au « blocage » de l'application qui était en place depuis l'appel de `setVisible(true)` fait par le constructeur de `BoiteDialogueMembre`.

Voyons maintenant comment les instances de `BoiteDialogueMembre` rendent service à la classe `ClubSportif`, dont voici la partie nouvelle :

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

public class ClubSportif extends JFrame implements Sports {
    Vector membres = new Vector();     // ce vecteur contient les membres du club
                                       // ses éléments sont des objets Membre
    ...

    private void creerMembre() {
        BoiteDialogueMembre dial = new BoiteDialogueMembre(this);
    α
        dial.pack()
        dial.setVisible(true)
    β
        if (dial.donneesAcquises) {
            Membre unMembre = new Membre();
            unMembre.nom = dial.champNom.getText();
            unMembre.prenom = dial.champPrenom.getText();
            unMembre.sexeMasculin = dial.sexeMasculin.isSelected();
            unMembre.adresse = dial.zoneAdresse.getText();

            unMembre.sportsPratiqués = 0;
            for (int i = 0; i < valSport.length; i++)
                if (dial.casesSports[i].isSelected())
                    unMembre.sportsPratiqués |= valSport[i];

            membres.add(unMembre);
        }
    }
    ...
}
```

La méthode `creerMembre` mérite une explication. Elle commence par la déclaration d'une variable locale `dial` et la construction d'un objet `BoiteDialogueMembre` ; ensuite, elle appelle les méthodes `pack()` et `setVisible(true)`. Or, on va attendre très longtemps le retour de ce dernier appel, car la boîte de dialogue étant modale, cette expression qui a pour effet de la rendre visible a également pour effet de bloquer le thread dans lequel la méthode `creerMembre` était en train de s'exécuter. Ainsi, le contrôle n'atteindra le point β que lorsque la méthode `dispose` de la boîte de dialogue aura été appelée – nous avons vu plus haut dans quelles conditions cela se produit.

Lorsque l'utilisateur aura mis fin au dialogue, l'objet qui est la valeur de `dial` ne sera plus visible mais il existera toujours (ce ne sera plus le cas lorsque, à la fin de la méthode `creerMembre`, la variable `dial` sera détruite) et il n'y a aucune difficulté à récupérer les valeurs de ses membres.

Notons au passage l'endroit, signalé par le repère α, où il faut donner aux champs de texte et aux cases à cocher des valeurs initiales lorsque la saisie ne doit pas se faire à partir de zéro, par exemple dans le cas de la modification d'un membre (voyez la section 11.8.6).

Est apparue également dans notre application une petite classe auxiliaire nommée `Membre`[99]. Le rôle de cette classe est évident : mémoriser les informations qui caractérisent un membre de notre club. Notez que, comme la méthode `creerMembre` le montre, les sports pratiqués sont stockés chacun sur un bit de l'entier long `sportsPratiqués` (cela limite à 64 le nombre de sports possibles, on peut penser que c'est suffisant...) :

```
public class Membre implements Serializable {
    String   nom;
    String   prenom;
    boolean  sexeMasculin;
    String   adresse;
    long     sportsPratiqués;
}
```

11.8.4 Exemple : choisir un fichier

FIGURE 25 – Boite de dialogue d'enregistrement de fichier

Pour montrer d'autres boîtes de dialogue, en particulier les boîtes prêtes à l'emploi offertes par *Swing* pour choisir un fichier (voir la figure 25), nous allons ajouter à notre application la possibilité de sauver et restaurer les membres du club dans un fichier.

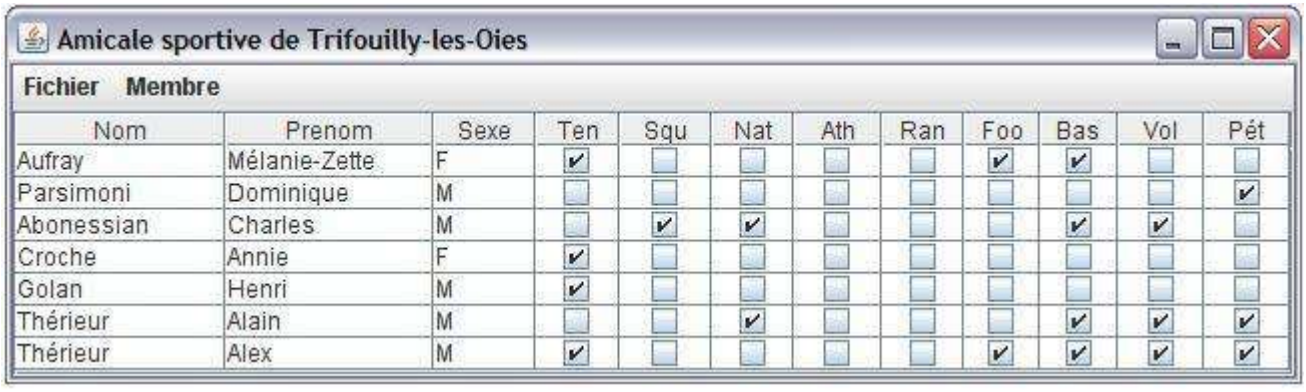
C'est pour avoir le droit d'en écrire les instances dans un fichier (on dit plutôt *sérialiser*) que nous avons déclaré que la classe `Membre` implémente l'interface `java.io.Serializable`. Il s'agit d'une interface vide, qui n'entraîne donc aucune obligation sur la classe qui l'implémente ; l'énoncé « `implements Serializable` » sert uniquement à indiquer que le programmeur donne le feu vert à la sérialisation des instances de la classe.

Pour obtenir les fonctionnalités indiquées il nous suffit d'écrire les deux méthodes `enregistrerFichier` et `ouvrirFichier` :

99. Une fois de plus, ne pas oublier que nous développons un exemple naïf, dans lequel seule l'interface utilisateur nous intéresse. Dans un cas réel, la classe `Membre` serait bien plus complexe et comporterait probablement des méthodes pour garantir la cohérence de l'information.

```
public class ClubSportif extends JFrame implements Sports {
    ...
    private void enregistrerFichier() {
        JFileChooser dial = new JFileChooser();
        dial.setDialogTitle("Enregistrer les membres");
        d'autres personnalisations de la boîte de dialogue pourraient apparaître ici
        if (dial.showSaveDialog(this) != JFileChooser.APPROVE_OPTION)
            return;
        File nomFichier = dial.getSelectedFile();

        try {
            FileOutputStream fos = new FileOutputStream(nomFichier);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(membres);
            fos.close();

        } catch (IOException ex) {
            JOptionPane.showMessageDialog(this, ex.toString(),
                                "Problème fichier", JOptionPane.ERROR_MESSAGE);
        }
    }

    private void ouvrirFichier() {
        JFileChooser dial = new JFileChooser();
        dial.setDialogTitle("Restaurer les membres");
        d'autres personnalisations de la boîte de dialogue pourraient apparaître ici

        if (dial.showOpenDialog(this) != JFileChooser.APPROVE_OPTION)
            return;
        File nomFichier = dial.getSelectedFile();

        try {
            FileInputStream fis = new FileInputStream(nomFichier);
            ObjectInputStream ois = new ObjectInputStream(fis);
            membres = (Vector) ois.readObject();
            fis.close();
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(this, ex.toString(),
                                "Problème fichier", JOptionPane.ERROR_MESSAGE);
        }
    }
    ...
}
```

11.8.5 Exemple : une table pour afficher des données



Nom	Prenom	Sexe	Ten	Squ	Nat	Ath	Ran	Foo	Bas	Vol	Pét
Aufray	Mélanie-Zette	F	✔					✔	✔		
Parsimoni	Dominique	M									✔
Abonessian	Charles	M		✔	✔				✔	✔	
Croche	Annie	F	✔								
Golan	Henri	M	✔								
Thérieur	Alain	M			✔				✔	✔	✔
Thérieur	Alex	M	✔					✔	✔	✔	✔

FIGURE 26 – Le tableau des membres du club

Pour en finir avec notre exemple nous allons lui ajouter une table affichant la liste des membres du club avec certaines informations associées, comme montré sur la figure 26.

Une table (classe `JTable`) est un composant relativement complexe obéissant au modèle de conception MVC dont le principe est de diviser le travail entre deux ojets : le *modèle* et la *vue*. Le premier détient les données, la seconde en fait une présentation graphique. Le modele MVC est commenté avec plus de soin à section 11.9.

Ici c'est surtout sur le modèle que nous allons travailler ; la vue sera un objet `JTable` dont le comportement par défaut nous suffira entièrement.

Les devoirs d'un modèle de table sont définis par l'interface `TableModel`, dont une implémentation partielle très utile est `AbstractTableModel`. D'où notre classe `ModeleTableClub` :

```
import javax.swing.table.*;

public class ModeleTableClub extends AbstractTableModel implements Sports {
    Vector membres;

    ModeleTableClub(Vector membres) {
        this.membres = membres;
    }

    public int getRowCount() {
        return membres.size();
    }

    public int getColumnCount() {
        return 3 + valSport.length;
    }

    public String getColumnName(int j) {
        if (j == 0)
            return "Nom";
        else if (j == 1)
            return "Prenom";
        else if (j == 2)
            return "Sexe";
        else
            return nomSport[j - 3].substring(0, 3);
    }

    public Class getColumnClass(int j) {
        if (j < 2)
            return String.class;
        else if (j == 2)
            return Character.class;
        else
            return Boolean.class;
    }

    public Object getValueAt(int i, int j) {
        Membre unMembre = (Membre) membres.elementAt(i);
        if (j == 0)
            return unMembre.nom;
        else if (j == 1)
            return unMembre.prenom;
        else if (j == 2)
            return new Character(unMembre.sexeMasculin ? 'M' : 'F');
        else
            return new Boolean((unMembre.sportsPratiqués & valSport[j - 3]) != 0);
    }
}
```

Dans la classe `ModeleTableClub` nous [re-]définissons celles des méthodes imposées par l'interface `TableModel` que la classe abstraite `AbstractTableModel` ne définit pas ou ne définit pas comme il nous faut.

Les méthodes précédentes sont assez faciles à comprendre :

`ModeleTableClub` : le constructeur sert à mémoriser une référence sur le vecteur de membres que le tableau est censé représenter ;

`getRowCount` : le nombre de lignes du tableau n'est autre que le nombre de membres du club ;

`getColumnCount` : le nombre de colonnes du tableau dépend que des champs que nous avons décidé de représenter (ici, le nom, le prénom, le sexe et les sports pratiqués) ;

`getColumnName` : donne le nom de chaque colonne, c'est-à-dire ce qu'il faut afficher dans la première ligne ;

`getColumnClass` : donne le type des informations de chaque colonne (supposée homogène), cela permet d'en donner une présentation adaptée comme, pour un booléen, une case à cocher ;

`getValueAt` : cette méthode est la plus importante : `getValueAt(i, j)` renvoie l'élément qui se trouve à la ligne `i` et la colonne `j` du tableau.

En plus de l'ajout de la classe précédente, nous devons modifier notre programme à plusieurs endroits : d'une part les variables d'instance et le constructeur de la classe principale :

```
public class ClubSportif extends JFrame implements Sports {

    Vector membres = new Vector();     // ce vecteur contient les membres du club
                                       // ses éléments sont des objets Membre
    AbstractTableModel modeleTable;    // le "modèle" de la table
    JTable table;                      // la "vue" de la table

    ClubSportif() {
        précédent contenu du constructeur

        modeleTable = new ModeleTableClub(membres);
        table = new JTable(modeleTable);
        getContentPane().add(new JScrollPane(table));

        setVisible(true);
    }
```

D'autre part, il faut faire en sorte que les changements dans la table des membres se répercutent dans le modèle. Deux manières de faire cela : pour de petits changement, comme l'ajout d'un unique membre, il suffit de le notifier au modèle. Par exemple, la méthode `creerMembre` se terminera maintenant ainsi :

```
...
membres.add(unMembre);
modeleTable.fireTableRowsInserted(membres.size() - 1, membres.size() - 1);
```

Pour de plus grands changements, comme lors du chargement d'un fichier, on peut carrément remplacer le modèle par un modèle neuf. Ainsi, dans la méthode `ouvrirFichier` on trouvera maintenant

```
...
membres = (Vector) ois.readObject();
modeleTable = new ModeleTableClub(membres);
table.setModel(modeleTable);
```

11.8.6 Exemple : sélection et modification dans une table

Pour en finir avec notre exemple, voici la méthode `modifierMembre` qui prend en charge les changements des informations concernant un membre existant. Ce que cette méthode illustre surtout, par comparaison avec `creerMembre`, est la mise en place d'une boîte de dialogue dont les composants contiennent des informations initiales.


```
    private void modifierMembre() {
        int rang = table.getSelectedRow();
        if (rang < 0) {
            JOptionPane.showMessageDialog(this,
                "Il faut d'abord sélectionner un membre",
                "Erreur", JOptionPane.ERROR_MESSAGE);
            return;
        }
        Membre unMembre = (Membre) membres.elementAt(rang);

        BoiteDialogueMembre dial = new BoiteDialogueMembre(this);
        dial.champNom.setText(unMembre.nom);
        dial.champPrenom.setText(unMembre.prenom);
        dial.sexeMasculin.setSelected(unMembre.sexeMasculin);
        dial.sexeFeminin.setSelected(!unMembre.sexeMasculin);
        dial.zoneAdresse.setText(unMembre.adresse);

        for (int i = 0; i < valSport.length; i++) {
            boolean b = (unMembre.sportsPratiqués & valSport[i]) != 0;
            dial.casesSports[i].setSelected(b);
        }
        dial.pack();
        dial.setVisible(true);

        if (dial.donneesAcquises) {
            unMembre.nom = dial.champNom.getText();
            unMembre.prenom = dial.champPrenom.getText();
            unMembre.sexeMasculin = dial.sexeMasculin.isSelected();
            unMembre.adresse = dial.zoneAdresse.getText();

            unMembre.sportsPratiqués = 0;
            for (int i = 0; i < valSport.length; i++)
                if (dial.casesSports[i].isSelected())
                    unMembre.sportsPratiqués |= valSport[i];

            modeleTable.fireTableRowsUpdated(rang, rang);
        }
    }
```

11.9 Le modèle MVC (Modèle-Vue-Controleur)

Le modèle MVC est un modèle de conception, ou *design pattern*, qui consiste à maintenir une séparation nette entre les données manipulées par l'application (le *modèle*), les présentations qui en sont faites (les *vues*) et les dispositifs par lesquels les actions de l'utilisateur sont détectées (les *contrôleurs*).

Puisque l'utilisateur a une tendance naturelle à agir sur les composants qu'il voit, les vues sont le plus souvent associées à des contrôleurs. L'aspect le plus intéressant de cette méthodologie est la *séparation de la vue et du modèle*, qui garantit que le codage interne et le traitement des données ne sont pas pollués par des questions de présentation ; cela permet, par exemple, que des présentations différentes d'un même modèle (textuelle, graphique, sonore, etc.) soient données, consécutivement ou simultanément.

Bien entendu, séparation n'est pas totale indépendance : chaque changement du modèle doit être notifié à la vue afin que celle-ci puisse être mise à jour et reflète constamment l'état actuel du modèle.

Les composants de *Swing* s'accordent au modèle MVC : chaque composant est considéré comme une vue (ou plutôt comme un couple vue-contrôleur), et il doit être associé à un modèle qui joue le rôle de source des données pour le composant.

Les méthodes qu'un objet doit posséder pour être le modèle d'un composant sont définies par une interface associée à ce composant : `ButtonModel`, `ListModel`, `TableModel`, `TreeModel`, etc.

Pour les composants les plus élémentaires, ces modèles sont très simples. Le programmeur peut presque toujours ignorer cette question, car il a la possibilité d'utiliser, souvent sans le savoir, des modèles par défaut. Par exemple, c'est un tel modèle qui est mis en place lorsqu'on crée un bouton par une instruction simple, comme « `new JButton("Oui")` ».

La section suivante développe un exemple avec un composant bien plus complexe, nommé `JTree`, dans lequel l'existence du modèle et sa séparation d'avec la vue ne peut être ignorée.

11.9.1 Exemple : les vues arborescentes (`JTree`)

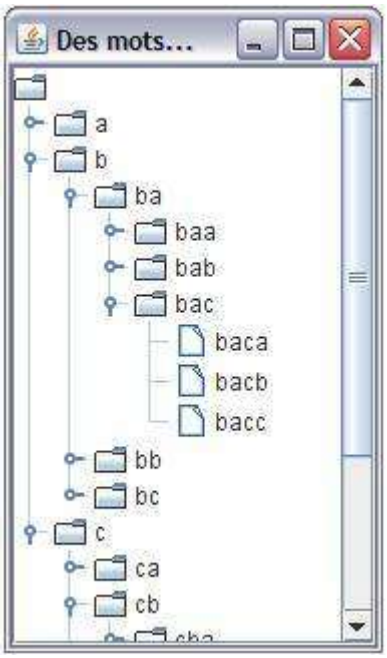


FIGURE 27 – Représentation arborescente (`JTree`) d'un ensemble de mots

Donnons-nous le problème suivant (cf. figure 27) : afficher l'arborescence de tous les mots d'au plus quatre lettres formés avec les trois lettres a, b et c[100]. Nous allons le traiter de quatre manières différentes, correspondant à divers choix que nous pouvons faire pour le modèle (la structure d'arbre sous-jacente à la vue `JTree`) et pour les nœuds dont cet arbre se compose :

- Souvent on n'a aucun prérequis à propos des nœuds implémentant l'arbre, lesquels n'ont pas besoin d'être très sophistiqués. On peut alors utiliser des objets de la classe prédéfinie `DefaultMutableTreeNode` pour les nœuds et un objet `DefaultTreeModel` pour le modèle. Ci-après, c'est la version 3, celle où on travaille le moins.
- Parfois on a déjà une définition de la classe des nœuds, découlant d'autres parties de l'application, mais on peut l'augmenter et notamment lui ajouter ce qui lui manque pour être une implémentation de l'interface `TreeNode`. Il nous suffit alors de prendre pour modèle un objet `DefaultTreeModel`, associé à de tels nœuds enrichis. C'est notre version 1 ci-après.
- Il peut arriver qu'on nous impose par ailleurs une définition précise de la classe des nœuds, à laquelle on nous interdit d'ajouter quoi que ce soit. On doit alors définir notre propre classe du modèle, implémentant l'interface `TreeModel`, sachant manipuler ces nœuds spécifiques. Nous avons fait cela à la version 2.
- Enfin, lorsqu'on décide comme dans la situation précédente de définir notre propre classe du modèle, il peut arriver que cette classe puisse être purement « calculatoire », c'est-à-dire qu'elle ne repose sur aucune structure de données sous-jacente. C'est le cas de notre version 4.

CONSTRUCTION D'UNE STRUCTURE DE DONNÉES SOUS-JACENTE. Pour commencer, supposons que les nœuds de l'arborescence nous soient *imposés* par ailleurs : ce sont les instances d'une certaine classe `Arbuste`. Chacun comporte une chaîne de caractères, un vecteur mémorisant l'ensemble des fils et une référence sur le pere :

100. Oui, bon, ce n'est pas très utile. Mais c'est un exemple, non ?

```
public class Arbuste {
    private String info;
    private Vector fils;
    private Arbuste pere;

    public Arbuste(String info) {
        this.info = info;
        fils = new Vector();
        pere = null;
    }

    public String getInfo() {
        return info;
    }

    public Arbuste getPere() {
        return pere;
    }

    public int nombreFils() {
        return fils.size();
    }

    public Arbuste getFils(int i) {
        return (Arbuste) fils.elementAt(i);
    }

    public void ajouterFils(Arbuste unFils) {
        fils.add(unFils);
        unFils.pere = this;
    }

    public String toString() {
        return info;
    }
}
```

Voici un programme pour essayer cette classe. L'appel `creerMots("", m, k)` construit l'arborescence des mots de k lettres formées avec les m lettres a, b, c...

```
public class Essai {
    public static Arbuste creerMots(String prefixe, int large, int profond) {
        Arbuste res = new Arbuste(prefixe);
        if (profond > 0)
            for (int i = 0; i < large; i++)
                res.ajouterFils(creerMots(prefixe + (char)('a' + i), large, profond - 1));
        return res;
    }

    public static void montrer(Arbuste arbre) {
        System.out.print(arbre + " ");
        int n = arbre.nombreFils();
        for (int i = 0; i < n; i++)
            montrer(arbre.getFils(i));
    }

    public static void main(String[] args) {
        Arbuste racine = creerMots("", 3, 4);
        montrer(racine);
    }
}
```

Voici (tronqué) l'affichage produit par ce programme :

```
a aa aaa aaaa aaab aaac aab aaba aabb aabc aac aaca aacb aacc ab
aba abaa abab abac abb abba abbb abbc abc abca abcb abcc ac aca
acaa acab acac acb ... cbca cbcb cbcc cc cca ccaa ccab ccac ccb
ccba ccbb ccbc ccc ccca cccb cccc
```

VUE ARBORESCENTE, VERSION 1. Pour afficher notre arborescence nous allons créer un composant `JTree`. Un tel composant constitue une *vue* ; pour le créer il faut lui associer un *modèle*, qui doit être une implémentation de l'interface `TreeModel`. Or il existe une implémentation de `TreeModel` toute prête : la classe `DefaultTreeModel`, qui n'a besoin pour fonctionner que d'une arborescence faite de nœuds implémentant l'interface `TreeNode`.

Notre première approche consiste donc à faire en sorte que nos objets `Arbuste` implémentent l'interface `TreeNode`. Pour cela il nous faut ajouter à notre classe `Arbuste` les méthodes de l'interface `TreeNode` (notez que certaines sont de simples renommages des méthodes qui existent déjà dans notre classe) :

```
import java.util.*;
import javax.swing.tree.*;

public class Arbuste implements TreeNode {

        ici apparaissent les membres de la version précédente de Arbuste,
        auxquels il faut ajouter les méthodes « imposées » suivantes :

    public Enumeration children() {
        return fils.elements();
    }

    public boolean getAllowsChildren() {
        return ! isLeaf();
    }

    public TreeNode getChildAt(int childIndex) {
        return getFils(childIndex);
    }

    public int getChildCount() {
        return nombreFils();
    }

    public int getIndex(TreeNode node) {
        return fils.indexOf(node);
    }

    public TreeNode getParent() {
        return pere;
    }

    public boolean isLeaf() {
        return fils.isEmpty();
    }
}
```

Voici la nouvelle classe de test, elle affiche le cadre représenté à la figure 27 :

```
import javax.swing.*;
import javax.swing.tree.*;

public class Essai {

    public static Arbuste creerMots(String prefixe, int large, int profond) {
        Arbuste res = new Arbuste(prefixe);
        if (profond > 0)
            for (int i = 0; i < large; i++)
                res.ajouterFils(creerMots(prefixe + (char)('a' + i), large, profond - 1));
        return res;
    }
```



```
    public static void main(String[] args) {
        JFrame cadre = new JFrame("Des mots...");
        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cadre.setSize(300, 500);

        Arbuste racine = creerMots("", 3, 4);
        JTree vue = new JTree(new DefaultTreeModel(racine));

        cadre.getContentPane().add(new JScrollPane(vue));
        cadre.setVisible(true);
    }
}
```

NOTE. Si nous avions envisagé que l'arbre puisse être modifié à la suite de son affichage, par exemple en réaction à des actions de l'utilisateur, alors il aurait fallu que notre classe `Arbuste` implémente l'interface `MutableTreeNode`, une sous-interface de `TreeNode` qui spécifie des méthodes supplémentaires pour ajouter et enlever des nœuds aux arbres.

VUE ARBORESCENTE, VERSION 2. Imaginons maintenant qu'il nous soit difficile ou impossible de modifier la classe `Arbuste` pour en faire une implémentation de `TreeNode`. Cela nous obligera à adopter une deuxième manière de construire la vue arborescente : ne pas utiliser un modèle par défaut (classe `DefaultTreeModel`) mais un modèle que nous aurons défini exprès pour travailler avec des objets `Arbuste`. C'est notre classe `ModeleArbuste`, entièrement faite de méthodes imposées par l'interface `TreeModel` :

```
import javax.swing.tree.*;
import javax.swing.event.*;
import java.util.*;

public class ModeleArbuste implements TreeModel {
    private Arbuste racine;
    private Vector auditeurs;

    ModeleArbuste(Arbuste racine) {
        this.racine = racine;
        auditeurs = new Vector();
    }
    public Object getRoot() {
        return racine;
    }
    public int getChildCount(Object parent) {
        return ((Arbuste) parent).nombreFils();
    }
    public Object getChild(Object parent, int index) {
        return ((Arbuste) parent).getFils(index);
    }
    public int getIndexOfChild(Object parent, Object child) {
        int n = ((Arbuste) parent).nombreFils();
        for (int i = 0; i < n; i++)
            if (((Arbuste) parent).getFils(i) == child)
                return i;
        return -1;
    }
    public boolean isLeaf(Object node) {
        return ((Arbuste) node).nombreFils() == 0;
    }
    public void addTreeModelListener(TreeModelListener l) {
        auditeurs.add(l);
    }
    public void removeTreeModelListener(TreeModelListener l) {
        auditeurs.remove(l);
    }
```

```
        public void valueForPathChanged(TreePath path, Object newValue) {
            /* Inutile */
        }
    }
```

Une seule ligne de la nouvelle classe `Essai` diffère d'avec la version précédente. Pour construire la vue, au lieu de `new JTree(new DefaultTreeModel(racine))` il faut faire :

```
        ...
        JTree vue = new JTree(new ModeleArbuste(racine));
        ...
```

VUE ARBORESCENTE, VERSION 3. A l'opposé de la précédente, une autre manière d'aborder ce problème consiste à penser que des nœuds aussi banals que les nôtres peuvent être réalisés avec le matériel disponible dans la bibliothèque, ce qui devrait nous dispenser d'écrire la classe `Arbuste`.

En effet, si tout ce qu'on demande à un nœud est de porter une information (peut importe son type, pourvu que ce soit un objet), d'avoir des fils et éventuellement d'avoir un père, alors la classe `DefaultMutableTreeNode` convient parfaitement. Bien entendu, cela suppose que nous ayons le droit de renoncer à la classe `Arbuste`.

Voici ce que devient, dans cette optique, la *totalité* de notre programme (il est maintenant beaucoup plus court) :

```
    import javax.swing.*;
    import javax.swing.tree.*;

    public class Essai {

        public static DefaultMutableTreeNode creerMots(String prefixe, int large, int profond) {
            DefaultMutableTreeNode res = new DefaultMutableTreeNode();
            res.setUserObject(prefixe);
            if (profond > 0)
                for (int i = 0; i < large; i++)
                    res.insert(creerMots(prefixe + (char)('a' + i), large, profond - 1), i);
            return res;
        }

        public static void main(String[] args) {
            JFrame cadre = new JFrame("Des mots...");
            cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            cadre.setSize(300, 500);

            DefaultMutableTreeNode racine = creerMots("", 3, 4);
            JTree vue = new JTree(new DefaultTreeModel(racine));

            cadre.getContentPane().add(new JScrollPane(vue));
            cadre.setVisible(true);
        }
    }
```

VUE ARBORESCENTE, VERSION 4. Il y a encore une autre manière de traiter ce problème ; elle consiste à examiner le modèle défini à la version 2, censé manipuler un arbre fait de nœuds `Arbuste`, et se dire qu'un tel modèle pourrait aussi bien « faire semblant » de manipuler un arbre qui, en tant que structure de données, *n'existe pas*.

Voici la classe qui réalise un tel modèle. Puisqu'il n'y a pas d'arbre, les nœuds se confondent avec les informations associées :


```
import javax.swing.tree.*;
import javax.swing.event.*;
import java.util.*;

public class ModeleArbuste implements TreeModel {
    private int large;
    private int profond;
    private Vector auditeurs;

    ModeleArbuste(int large, int profond) {
        this.large = large;
        this.profond = profond;
        auditeurs = new Vector();
    }

    public Object getRoot() {
        return "";
    }

    public int getChildCount(Object parent) {
        if (((String) parent).length() < profond)
            return large;
        else
            return 0;
    }

    public Object getChild(Object parent, int index) {
        return ((String) parent) + (char)('a' + index);
    }

    public int getIndexOfChild(Object parent, Object child) {
        String s = (String) child;
        return s.charAt(s.length() - 1) - 'a';
    }

    public boolean isLeaf(Object node) {
        return ! (((String) node).length() < profond);
    }

    public void addTreeModelListener(TreeModelListener l) {
        auditeurs.add(l);
    }

    public void removeTreeModelListener(TreeModelListener l) {
        auditeurs.remove(l);
    }

    public void valueForPathChanged(TreePath path, Object newValue) {
        /* inutile */
    }
}
```

11.10 Images

Dans les premières versions de Java la manipulation des images obéit à un modèle de *production* et *consommation*. Ayant pour objet premier les images provenant du Web, ce modèle suppose que l'exploitation d'une image est un processus, éventuellement de longue durée, impliquant un *producteur*, comme un fichier sur un serveur éloigné, un *consommateur*, par exemple l'interface graphique d'une application qui affiche l'image, et un *observateur*, c'est-à-dire un objet qui « supervise » l'opération et qui reçoit les notifications concernant l'état d'avancement du transfert de l'image.

A côté de ce modèle, passablement complexe, l'équipe de développement de Java fait maintenant la promotion d'un modèle dit *en mode immédiat*, centré sur la notion d'image en mémoire : fondamentalement, on s'intéresse moins au transfert de l'image (pour lequel le modèle producteur/consommateur reste pertinent) qu'à ce qu'on peut faire avec l'image lorsque, une fois transférée, elle est stockée dans la mémoire du système.

11.10.1 Exemple : utiliser des icônes

FIGURE 28 – Six boutons et une (grande) étiquette décorés d'icônes

Les icônes sont des images de taille fixe, souvent petites, généralement utilisées pour décorer certains composants comme les étiquettes et les boutons.

L'application très simple montrée[101] à la figure 28 est faite d'une grande étiquette portant un texte (« *Hôtesse de l'air à la Bardaf* ») et une icône (Natacha, un personnage de BD des années 70). Le texte peut se placer à gauche, dans l'alignement ou à droite de l'icône, ainsi qu'en haut, à la même hauteur ou en bas de l'icône (les composants étiquettes de *Swing* – objets `JLabel` – permettent de faire cela). L'application comporte en outre une barre d'outils détachable avec six boutons, disposés en deux groupes de trois boutons mutuellement exclusifs, qui commandent la position du texte par rapport à l'icône. Les boutons sont blancs quand ils sont au repos (non sélectionnés) et rouges quand ils sont sélectionnés.

Ce programme utilise treize fichiers contenant chacun une icône : six icônes de boutons blancs (fichiers `bhgauche.gif`, `bhcentre.gif`, etc.), six icônes de boutons rouges, dont les noms sont de légères altérations des précédents (fichiers `bhgaucheR.gif`, `bhcentreR.gif`, etc.) et le dessin central (fichier `natacha.jpg`). Tous ces fichiers sont placés dans un répertoire ayant le nom relatif `images`.

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

public class Natacha extends JFrame implements ActionListener {
    JLabel etiquette;
    JToolBar barreOutils;

    static String chemin = "images/";
    static String nomFic[] = { "bhgauche", "bhcentre", "bhdroite",
                   "bvhaut", "bvcentre", "bvbas" };
    static int position[] = { JLabel.LEFT, JLabel.CENTER, JLabel.RIGHT,
                   JLabel.TOP, JLabel.CENTER, JLabel.BOTTOM };
```

101. L'imperfection des moyens de reprographie habituellement infligés à ce genre de polycopiés nous dissuade de montrer ici de vraies photographies, qui auraient pourtant été tout à fait adaptées à l'illustration de notre propos.

```
    Natacha() {
        super("Natacha");
        creerBarreOutils();
        creerEtiquette();
        getContentPane().add(barreOutils, BorderLayout.NORTH);
        getContentPane().add(etiquette, BorderLayout.CENTER);
        pack();
        setVisible(true);
    }

    void creerBarreOutils() {
        barreOutils = new JToolBar();
        ButtonGroup[] groupe = { new ButtonGroup(), new ButtonGroup() };
        for (int i = 0; i < 6; i++) {
            Icon iconeBoutonAuRepos = new ImageIcon(chemin + nomFic[i] + ".gif");
            Icon iconeBoutonSelect = new ImageIcon(chemin + nomFic[i] + "R.gif");
            JRadioButton radioBouton = new JRadioButton(iconeBoutonAuRepos);
            radioBouton.setSelectedIcon(iconeBoutonSelect);
            radioBouton.setActionCommand(nomFic[i]);
            radioBouton.addActionListener(this);
            groupe[i / 3].add(radioBouton);
            barreOutils.add(radioBouton);
            if (i == 1 || i == 3)
                radioBouton.setSelected(true);
        }
    }

    void creerEtiquette() {
        ImageIcon icone = new ImageIcon(chemin + "natacha.jpg");
        etiquette = new JLabel("Hôtesse de l'air à la Bardaf", icone, JLabel.CENTER);
        etiquette.setIconTextGap(20);
        etiquette.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
        etiquette.setHorizontalTextPosition(JLabel.CENTER);
        etiquette.setVerticalTextPosition(JLabel.TOP);
    }

    public void actionPerformed(ActionEvent e) {
        for (int i = 0; i < 6; i++)
            if (e.getActionCommand().equals(nomFic[i]))
                break;
        if (i < 3)
            etiquette.setHorizontalTextPosition(position[i]);
        else
            etiquette.setVerticalTextPosition(position[i]);
    }

    public static void main(String[] args) {
        new Natacha();
    }
}
```

NOTE. Dans l'exemple ci-dessus, les boutons de la barre d'outils n'ont pas un texte affiché. Il faut donc explicitement leur associer des chaînes *action command* afin de pouvoir les distinguer dans la méthode `actionPerformed`. Comme n'importe quel ensemble de six chaînes distinctes fait l'affaire, nous avons utilisé les noms des fichiers images, qui ont le mérite d'exister et d'être deux à deux différents.

11.10.2 Exemple : charger une image depuis un fichier

Après les icônes, qui sont des images simplifiées qu'on ne souhaite pas transformer, intéressons-nous aux images (quelconques) en mémoire, ou *buffered images*.

De telles images ont deux principales raisons d'être [102] : soit ce sont des images « calculées », c'est-à-

102. Une bonne raison de s'intéresser aux images en mémoire est le *double buffering*. Lorsqu'on dessine directement sur l'écran

dire construites pixel à pixel par le programme (les fameuses images de synthèse), soit ce sont des images « traitées », c'est-à-dire produites à l'extérieur du programme (par exemple des photographies) mais devant subir des transformations géométriques, des modifications des couleurs, etc. Bien entendu, dans un cas comme dans l'autre, il est en général nécessaire que ces images, une fois construites en mémoire, puissent être affichées sur un écran.

FIGURE 29 – Afficher une image

La figure 29 correspond à une application très simple qui montre une première manière d'obtenir une image en mémoire : la charger depuis un fichier (qui est nommé [103] ici `XGorce_020613.gif` et est placé dans un certain répertoire `images`) :

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

public class AfficheurdImages extends JPanel {
    Image uneImage;

    AfficheurdImages() {
        uneImage = Toolkit.getDefaultToolkit().getImage("images/XGorce_020613.gif");
        setPreferredSize(new Dimension(500, 150));
    }

    public void paint(Graphics g) {
        super.paint(g);
        g.drawImage(uneImage, 5, 5, this);
    }

    public static void main(String[] args) {
        JFrame cadre = new JFrame("Afficher une image");
        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cadre.getContentPane().add(new AfficheurdImages());
        cadre.pack();
        cadre.setVisible(true);
    }
}
```

Dans la classe ci-dessus les méthodes intéressantes sont le constructeur et la méthode `paint` : le premier obtient l'image à partir d'un fichier, la seconde l'affiche.

REMARQUE POINTUE. Ce programme fait ce qu'on lui demande de faire et il semble sans mystère car, à moins d'avoir une vue exceptionnelle, l'utilisateur ne voit pas que dès le démarrage l'image est (re-)dessinée un grand nombre de fois. On peut s'en apercevoir en ajoutant dans la méthode `paint` la ligne

```
System.out.println(compteur++);
```

une image très complexe, les étapes de la construction sont perceptibles et l'affichage devient désagréable pour l'utilisateur, particulièrement dans le cas d'animations. Le *double buffering* est la technique consistant à dessiner non pas sur l'écran, mais sur une portion de mémoire qu'on met à l'écran en une seule opération, donc de manière instantanée, lorsque le tracé est entièrement fini.

103. Xavier Gorce dessine un *strip* quotidien dans *LeMonde.fr*, dans lequel les animaux de la forêt – et, plus récemment, les manchots de la banquise – commentent l'actualité ou font des remarques philosophiques.

(`compteur` est une variable d'instance privée). Le programme affiche l'image et, en plus, il affiche des nombres qui permettent de constater que `paint` a été appelée plus de vingt fois avant que le programme ne se calme !

Cela montre le caractère asynchrone du modèle producteur/consommateur, dont relève la méthode `getImage` de la classe `Toolkit`. L'appel qui en est fait dans le constructeur

```
uneImage = Toolkit.getDefaultToolkit().getImage(nom de fichier);
```

n'attend pas la fin du chargement de l'image ; au lieu de cela, cet appel initie le processus et retourne immédiatement, avant que les données qui constituent l'image ne soient acquises [104]. Par conséquent, l'appel de `paint` qui est fait lors de l'affichage initial de l'interface ne dessine pratiquement rien.

Ce qui sauve ce programme est le quatrième argument (`this`) de l'appel de `drawImage` :

```
g.drawImage(uneImage, 5, 5, this);
```

il indique que l'*observateur* de cette image est le panneau `AfficherUneImage` lui-même, qui devient ainsi le destinataire des notifications concernant la progression du chargement. Chacune de ces notifications provoque un appel de `repaint`, donc l'affichage d'un nouvel état de l'image, jusqu'à ce que cette dernière soit entièrement chargée ; cela se passe assez vite et l'utilisateur croit voir l'affichage en une fois d'une image unique [105].

Charger une image rangée avec les classes

Dans l'exemple précédent on charge une image à partir d'un fichier spécifié par son chemin dans un système de fichiers donné (probablement celui qui entoure le développement) :

```
uneImage = Toolkit.getDefaultToolkit().getImage("images/XGorce.081102.gif");
```

Se pose alors la question : comment donner le chemin du fichier image dans le cas – le plus fréquent – où l'application doit s'exécuter dans un environnement inconnu, distinct de celui de son développement ?

Si le fichier de l'image est un fichier variable appartenant au système dans lequel l'application s'exécute il n'y a pas de problème : le chemin sera saisi durant l'exécution, soit sous forme de texte, soit à l'aide d'un `FileChooser` (cf. 11.8.4).

Mais comment indiquer un fichier « constant », fourni par le développeur de l'application, rangé avec les classes de celle-ci, par exemple dans l'archive *jar* utilisée pour distribuer l'application ? La solution consiste à obtenir l'adresse du fichier par la méthode `getResource` de la classe (nous disons bien la *classe*) en cours d'exécution. Le travail est alors délégué à l'objet *class loader* qui a chargé cette classe, en quelque sorte cela revient à chercher le fichier « là où on a trouvé la classe ».

Par exemple, si nous avons archivé les classes de notre application avec un répertoire `images` contenant le fichier en question, il suffira de remplacer la ligne montrée précédemment par :

```
URL url = AfficheurdImages.class.getResource("images/XGorce.021108.gif");
uneImage = Toolkit.getDefaultToolkit().getImage(url);
```

NOTE. Dans une méthode d'instance (c.-à-d. non statique), la première des lignes ci-dessus peut s'écrire également

```
URL url = getClass().getResource("images/XGorce.021108.gif");
```

11.10.3 Exemple : construire une image pixel par pixel

La méthode `createImage(ImageProducer producteur)` de la classe `java.awt.Component` renvoie une image correspondant aux données (les *pixels*) fournies par l'objet `producteur`. Le type de l'image (i.e. le mode de codage des pixels) est celui qui correspond à l'environnement graphique utilisé, logiciel et matériel.

Comme producteur on peut prendre un objet `MemoryImageSource`, consistant essentiellement en un tableau de nombres, chacun décrivant un pixel selon le codage *RGB*.

Par exemple, le programme suivant dessine une image en noir et blanc calculée pixel par pixel (voyez la figure 30) : le centre de l'image est blanc, puis les pixels s'assombrissent au fur et à mesure qu'on s'éloigne du centre, proportionnellement au carré de la distance au centre :

104. C'est la raison pour laquelle, dans le constructeur, nous ne pouvons pas donner à notre panneau la taille effective requise par l'image (ce qui aurait été bien pratique !) mais uniquement une taille convenue comme 500×150. C'est que, immédiatement après l'appel de `createImage`, l'image n'est qu'amorcée et elle n'a pas de taille définie.

105. Pour s'en convaincre il suffit de remplacer `this` par `null` dans l'appel de `drawImage` : les notifications sur l'avancement du chargement seront perdues, il y aura un seul appel de `paint` et ce programme n'affichera presque rien.

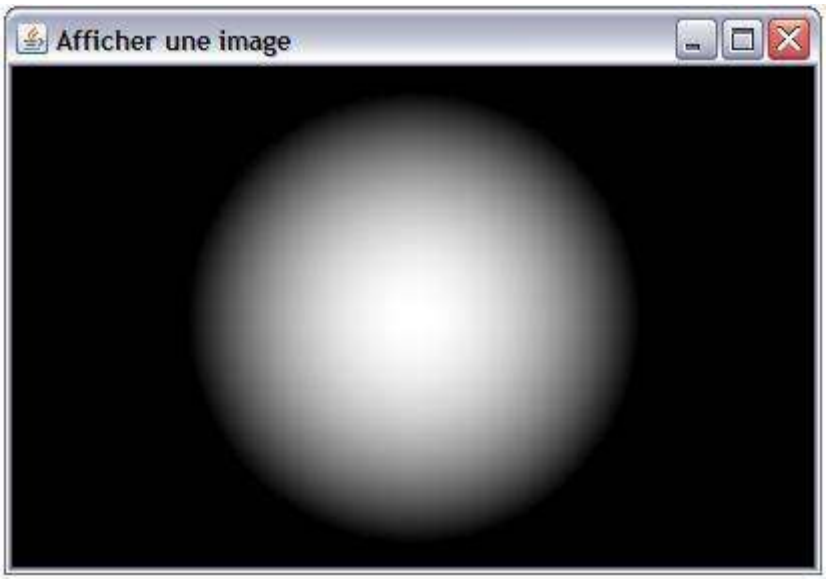


FIGURE 30 – Une image de synthèse !

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

class AfficheurdImages extends JPanel {
    static final int L = 400;
    static final int H = 250;
    Image uneImage;

    AfficheurdImages() {
        setPreferredSize(new Dimension(L, H));
        int tabPixels[] = new int[L * H];
        int i = 0;
        for (int y = 0; y < H; y++) {
            for (int x = 0; x < L; x++) {
                int dx = x - L / 2;
                int dy = y - H / 2;
                int r = Math.max(255 - (dx * dx + dy * dy) / 50, 0);
                tabPixels[i++] = (255 << 24) | (r << 16) | (r << 8) | r;
            }
        }
        uneImage = createImage(new MemoryImageSource(L, H, tabPixels, 0, L));
    }

    public void paint(Graphics g) {
        super.paint(g);
        g.drawImage(uneImage, 0, 0, this);
    }

    public static void main(String[] args) {
        JFrame cadre = new JFrame("Afficher une image");
        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cadre.getContentPane().add(new AfficheurdImages());
        cadre.pack();
        cadre.setVisible(true);
    }
}
```

12 Java 5

Si chaque version de Java a apporté au langage un lot d'éléments nouveaux, Java 5 a été une avancée particulièrement significative. Il y aurait beaucoup à dire à propos des nouveautés de cette version, mais nous nous limiterons ici aux points suffisamment importants pour mériter de figurer dans un polycopié succinct comme celui-ci.

Au moment où ce document est produit la version courante de Java est la version 6, officiellement dénommée Java™ Platform Standard Edition 6 (ou, entre développeurs, *JDK 1.6*) mais les versions 5 et 1.4 sont encore assez répandues. Vous pouvez spécifier quelles extensions vous autorisez dans votre programme en lançant la compilation avec l'option `-source` :

`javac -source` { *1.4* | *1.5 ou 5* | *1.6 ou 6* } *Classe*`.java`

Par défaut, la compilation se fait en accord avec la version 6. Si vous compilez avec l'option `-source 1.4` alors les extensions expliquées ci-après ne seront pas admises ; cela peut s'avérer utile, certaines prescriptions de la version 5 provoquant parfois des messages d'avertissement copieux et obscurs lors de la compilation de sources d'origine 1.4.

12.1 Types énumérés

On a souvent besoin d'ensembles finis de données conventionnelles, comme { *nord*, *sud*, *est*, *ouest* }, { *lundi*, *mardi*, *mercredi*, *jeudi*, *vendredi*, *samedi*, *dimanche* }, etc. Quel que soit le domaine d'application, ces sortes de données partagent certains traits caractéristiques :

1. Elles sont *symboliques* et naturellement munies de noms significatifs. C'est pourquoi il est regrettable de les représenter à l'aide de simples nombres, comme on le fait quand on n'a pas d'autre moyen : 0 pour *nord*, 1 pour *sud*, 2 pour *est*, etc.
2. Ce sont des données *atomiques* – comme des nombres –, ce qui devrait permettre un faible encombrement et un traitement très rapide. C'est pourquoi il est regrettable de les représenter par des chaînes de caractères, comme on le fait parfois : `"nord"` pour *nord*, `"sud"` pour *sud*, etc.
3. Ces données constituent des ensembles *finis*, souvent de taille réduite, ce qui devrait permettre certaines opérations et certaines optimisations – par exemple, pour la représentation de leurs *sous-ensembles* – auxquelles il faut renoncer si on représente ces données par des nombres ou des chaînes de caractères.
4. Enfin, et surtout, elles constituent des ensembles *disjoints* d'avec les autres types de données : un point cardinal, par exemple, n'est pas interchangeable avec un jour de la semaine ! Cela devrait permettre, de la part du compilateur, un puissant contrôle de type qu'on n'a pas si on représente ces données par des nombres ou des chaînes.

En Java, jusqu'à la version 1.4, on implémentait presque toujours les données symboliques par des suites de constantes numériques :

```
public static final int NORD = 0;
public static final int SUD = 1;
public static final int EST = 2;
public static final int OUEST = 3;
```

Cette manière de faire respecte les deux premières caractéristiques ci-dessus, mais pas les deux suivantes. En effet, rien ne dit au compilateur que les constantes `NORD`, `SUD`, `EST`, `OUEST` constituent la *totalité* d'un type et, surtout, ces constantes ne forment même pas un type : le compilateur ne pourra pas nous aider si par inadvertance nous affectons la valeur `NORD` à une variable `jourDeLaSemaine` (dont les valeurs attendues `LUNDI`, `MARDI`, etc., sont également représentées par des constantes entières).

Java 5 corrige ces défauts en introduisant les nouveaux *types énumérés* ou *enums*, expliqués ci-après.

12.1.1 Enums

La déclaration d'un type énuméré se compose au moins des éléments suivants (cela s'enrichira par la suite) :

qualifieurs `enum` *identificateur* `{` *identificateur*`,` *identificateur*`,` *... identificateur* `}`

Deux exemples :

```
public enum PointCardinal { NORD, SUD, EST, OUEST }
public enum JourSemaine { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE }
```

NOTE. Les deux déclarations précédentes définissant des types publics, elles doivent être écrites dans deux fichiers séparés, respectivement nommés `PointCardinal.java` et `JourSemaine.java`.

Exemples d'utilisation ultérieure :

```
PointCardinal direction = PointCardinal.NORD;
...
JourSemaine jourDePresence = JourSemaine.MERCREDI;
```

Notez bien que la déclaration d'un type `enum` n'est pas qu'un artifice pour donner des noms expressifs à une poignée de nombres entiers. Les déclarations précédentes créent bien deux types nouveaux, `PointCardinal` et `JourSemaine`, distincts entre eux et distincts de tous les autres types, *dont l'emploi pourra être finement contrôlé par le compilateur.*

Pour faire comprendre la nature des éléments d'un type `enum` disons pour commencer (car en fait c'est plus riche que cela) que la déclaration

```
public enum PointCardinal { NORD, SUD, EST, OUEST }
```

a déjà *tout l'effet* qu'aurait eu la déclaration suivante :

```
public class PointCardinal {
    public static final PointCardinal NORD = new PointCardinal();
    public static final PointCardinal SUD = new PointCardinal();
    public static final PointCardinal EST = new PointCardinal();
    public static final PointCardinal OUEST = new PointCardinal();

    private PointCardinal() {} // pour interdire la création d'autres instances
}
```

12.1.2 Méthodes des types énumérés

Un type énuméré apparaît donc comme une classe dont les seules instances sont les valeurs du type énuméré. En réalité, tout type énuméré est sous-classe de la classe `java.lang.Enum`, qui introduit quelques méthodes bien utiles :

`String name()`, `String toString()`
: Ces deux méthodes renvoient la valeur en question, exprimée sous forme de chaîne de caractères ; par exemple `PointCardinal.NORD.name()` est la chaîne `"NORD"`. La seconde est plus intéressante car elle est implicitement appelée par Java pour convertir une valeur en chaîne ; de plus, elle peut être redéfinie. Cela s'écrit comme ceci :

```
public enum PointCardinal {
    NORD, SUD, EST, OUEST;

    public String toString() {
        return "<" + super.toString().toLowerCase() + ">";
    }
}
```

Dans ces conditions, l'instruction

```
System.out.println(PointCardinal.OUEST);
```

produit l'affichage de

```
<ouest>
```

`static` *énumération* `valueOf(String nom)`
: Renvoie la valeur du type énuméré dont le nom est donné. Ce nom doit être écrit *exactement* comme dans la définition du type énuméré. Par exemple, `PointCardinal.valueOf("EST")` vaut `PointCardinal.EST`.

`static` *énumération*`[] values()`
: Renvoie un tableau contenant *toutes* les valeurs du type énuméré. Par exemple, le code

```
PointCardinal[] directions = PointCardinal.values();
for (int i = 0; i < directions.length; i++)
    System.out.print(directions[i] + " ");
```

affiche le texte « `NORD SUD EST OUEST` ».

`int ordinal()`
: Renvoie un entier qui exprime le rang de la valeur dans le type : `PointCardinal.NORD.ordinal()` vaut 0, `PointCardinal.SUD.ordinal()` vaut 1, etc.

  La transformation réciproque est facile à obtenir : `PointCardinal.values()[0]` vaut `PointCardinal.NORD`, `PointCardinal.values()[1]` vaut `PointCardinal.SUD`, etc.

12.1.3 Aiguillages commandés par des types énumérés

Bien qu'elles ne soient pas des nombres ou des caractères, les valeurs des types énumérés peuvent servir à piloter des aiguillages. Par exemple, supposons avoir la déclaration

```
public enum CategorieVehicule {
    BERLINE, COUPE, FAMILIALE;
}
```

voici un tel aiguillage :

```
...
CategorieVehicule categorie;
...
switch (categorie) {
    case BERLINE:
        code correspondant au cas où categorie vaut CategorieVehicule.BERLINE
        break;
    case COUPE:
        code correspondant au cas où categorie vaut CategorieVehicule.COUPE
        break;
    case FAMILIALE:
        code correspondant au cas où categorie vaut CategorieVehicule.FAMILIALE
        break;
    default:
        throw new AssertionError("\nErreur de programmation: cas "
                + categorie + " non traité dans switch");
}
```

Notez que, puisque le `switch` est commandé par une expression de type `CategorieVehicule`, à l'intérieur de celui-ci le compilateur laisse écrire `BERLINE`, `COUPE` ou `FAMILIALE` au lieu de `CategorieVehicule.BERLINE`, `CategorieVehicule.COUPE` ou `CategorieVehicule.FAMILIALE`.

Notez également que l'utilisation de la clause `default`, comme ci-dessus, est recommandée. En procédant ainsi, si plus tard on ajoute des valeurs au type énuméré en oubliant d'ajouter les `case` correspondants dans les `switch` concernés, on en sera prévenu (hélas, on ne sera prévenu qu'à l'exécution, on aurait sans doute préféré l'être durant la compilation...).

12.1.4 Dictionnaires et ensembles basés sur des types énumérés

Dictionnaires. Le nouveau type `EnumMap` (en réalité c'est une classe paramétrée mais pour commencer nous passerons cet aspect sous silence – voyez la fin de cette section) a été introduit pour permettre la construction de tables associatives dont les clés sont les valeurs d'un type énuméré, avec optimisation de l'espace et du temps d'accès. Cela s'emploie comme ceci :

```
Map permanence = new EnumMap(JourSemaine.class);
...
permanence.put(JourSemaine.MARDI, "M. Jack Palmer");
permanence.put(JourSemaine.JEUDI, "Mme Raymonde Bidochon");
...
JourSemaine jour;
...
String qui = (String) permanence.get(jour);
if (qui != null)
    System.out.println("le permanent de " + jour + " est " + qui);
...
```

ENSEMBLES. De manière analogue à `EnumMap`, le type `EnumSet` (encore une classe paramétrée, voyez la fin de cette section) a été introduit en vue de la définition d'ensembles compacts et efficaces [106] dont les éléments sont les valeurs d'un type énumération donné.

Les ensembles `EnumSet` supportent, comme toutes les collections, les opérations fondamentales `contains` (test d'appartenance), `add` et `remove` (ajout et suppression d'un élément), plus quelques opérations de *fabrication* particulièrement commodes :

`static EnumSet of(Object premierElement, Object... autresElements)`
: Construction d'un ensemble formé des éléments indiqués (il s'agit d'une méthode avec nombre variable d'arguments, cf. § 12.3.3).

`static EnumSet allOf(Class typeElements)`
: Construction d'un ensemble formé de tous éléments du type énuméré indiqué.

`static EnumSet complementOf(EnumSet s)`
: Construction d'un ensemble formé de tous les éléments du type énuméré en question qui ne sont pas dans l'ensemble indiqué.

Exemple :

```
...
EnumSet joursDePresence
        = EnumSet.of(JourSemaine.LUNDI, JourSemaine.MERCREDI, JourSemaine.VENDREDI);
...
joursDePresence.add(JourSemaine.MARDI);
joursDePresence.remove(JourSemaine.VENDREDI);
...
JourSemaine jour;
...
if (joursDePresence.contains(jour))
    System.out.println("ok pour " + jour);
...
```

NOTE. Nous ne l'avons pas fait apparaître dans les lignes précédentes, pour ne pas les alourdir, mais en réalité `EnumMap` et `EnumSet` sont des classes paramétrées (cf. § 12.5) et leurs déclarations complètes prennent les formes – assez troublantes – suivantes :

```
class EnumSet<E extends Enum<E>> { ... }
```

```
class EnumMap<K extends Enum<K>, V> { ... }
```

12.2 Emballage et déballage automatiques

12.2.1 Principe

En Java, les données de types primitifs, `byte`, `short`, `int`, `long`, `float`, `double`, `char` et `boolean`, ne sont pas des objets. Cela est ainsi pour des raisons évidentes d'efficacité, car traiter ces types de données comme le font les langages les plus simples permet d'utiliser des opérations « câblées » (c'est-à-dire directement prises en charge par le matériel) extrêmement optimisées. Cependant, d'un point de vue méthodologique, ce n'est pas satisfaisant :

- c'est regrettable pour la clarté des idées, car cela introduit deux poids et deux mesures (ce qui est vrai pour les objets ne l'est pas forcément pour les données primitives, et réciproquement), multiplie les cas particuliers, alourdit les spécifications, etc.,
- d'un point de vue pratique c'est encore plus regrettable, car cela interdit d'employer sur des données de types primitifs les puissants outils de la bibliothèque Java comme les collections, qui ne travaillent qu'avec des objets.

Ce second défaut étant considérable, il a bien fallu lui trouver un remède. Cela a consisté (cf. § 9.1.1) en la définition de huit classes `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character` et `Boolean`, en correspondance avec les huit types primitifs ; chaque instance d'une de ces classes se compose d'une unique donnée membre qui est une valeur du type primitif que l'instance « enveloppe ».

L'opération consistant à construire un tel objet enveloppant une valeur v d'un type primitif s'appelle « emballage de la valeur v » ; l'opération réciproque s'appelle « déballage de la valeur v ». Ainsi, par exemple, pour des valeurs de type `int`, ces opérations *peuvent* (et, jusqu'à Java 1.4, *doivent*) s'écrire :

106. On nous assure que ces ensembles sont réalisés avec des tables de bits, ils sont donc aussi peu encombrants et d'un traitement aussi efficace que les ensembles qu'on peut réaliser soi-même avec des éléments représentés par des puissances de 2 et les opérateurs de bits & et |.


```
int unInt;
...
Integer unInteger = new Integer(unInt);        // emballage de unInt
...
unInt = unInteger.intValue();                  // déballage de unInt
```

La bonne nouvelle est qu'en Java 5 ces deux opérations sont devenues *automatiques*, c'est-à-dire insérées par le compilateur là où elles sont nécessaires, sans que le programmeur n'ait à s'en occuper explicitement. Plus précisément :

- dans un endroit où un objet est attendu on peut mettre une expression d'un type primitif ; à l'exécution, sa valeur sera convertie en objet par construction d'une instance de la classe enveloppe correspondant à ce type primitif,
- dans un endroit où une valeur d'un type primitif est attendue on peut mettre une expression dont le type est la classe-enveloppe correspondante ; à l'exécution, l'objet sera converti dans le type primitif requis par appel de la méthode *type*`Value()`, où *type* est le nom de ce type primitif.

Exemple. Un important domaine d'application de l'emballage et déballage automatiques est la mise en œuvre de collections contenant des données primitives. Par exemple, voici comment on peut réaliser simplement une *file d'attente* d'entiers en Java 5 :

Déclaration et création :

```
List file = new LinkedList();       // ou Vector, ou toute autre implémentation de List
```

Opération « introduire un entier v dans la file » (v a été déclarée `int`) :

```
file.add(v);                        // add ajoute à la fin de la liste
```

Opération « extraire un entier u de la file » :

```
int u = (Integer) file.remove(0);   // extraction de l'élément de tête
```

NOTE. Avec les collections génériques expliquées plus loin (cf. § 12.5) les opérations précédentes deviennent encore plus simples et, surtout, plus *fiables* :

```
List<Integer> file = new LinkedList<Integer>();
...
file.add(v);                        // ici il est vérifié que la conversion de v
...                                 // donne un Integer
int u = file.remove(0);             // ce qui sort de la file est un Integer
```

12.2.2 Opérations dérivées et autres conséquences

Le mécanisme de l'emballage et déballage automatiques est simple et puissant, mais il faut prendre garde qu'il a de nombreuses conséquences, dont certaines subtiles.

ARITHMÉTIQUE. Par exemple, avec la déclaration

```
Integer nombre;
```

le code suivant est légitime

```
nombre = nombre + 1;
```

le compilateur le traduira en :

```
nombre = new Integer(nombre.intValue() + 1);
```

De même, il est possible d'écrire

```
nombre++
```

cette expression ayant la valeur et l'effet qu'aurait l'appel d'une fonction *fictive* ressemblant à ceci :

```
Integer nombre++() {
    Integer tmp = nombre;
    nombre = new Integer(nombre.intValue() + 1);
    return tmp;
}
```

COMPARAISON. Le cas de la comparaison est plus délicat. Donnons-nous la situation suivante :

```
Integer a = 123;
Integer b = 123;
Integer c = 12345;
Integer d = 12345;
```

et examinons l'affichage produit alors par les expressions suivantes :

```
System.out.println( a == 123 );
System.out.println( c == 12345 );
System.out.println( a == b );
System.out.println( c == d );
```

Sans surprise, la quatrième expression (`c == d`) s'affiche `false`. En effet, `c` et `d` sont des objets construits séparément et on a déjà expliqué (cf. § 3.6.3) que, pour les objets, `==` exprime l'identité (égalité des références), non l'équivalence. Pour tester l'équivalence (égalité des valeurs) il aurait fallu écrire `c.equals(d)`.

En revanche, on sera peut-être étonné en constatant que les trois autres expressions affichent `true`. Pour les deux premières (`a == 123` et `c == 12345`) la question est la même : il s'agit de savoir si la comparaison `a == 123` porte sur deux valeurs primitives ([`a` converti en `int`] et 123) ou bien sur deux objets (`a` et [123 transformé en `Integer`]). C'est le premier qui se produit : la comparaison des valeurs étant en général plus intéressante que la comparaison des références, le compilateur préfère travailler dans le type primitif. D'où le résultat `true` : `a` et `c` convertis en `int` sont bien respectivement égaux aux nombres auxquels on les compare.

Il reste le troisième cas, `a == b`, qui s'affiche `true` faisant penser que, bien que construits séparément, `a` et `b` sont le même objet. Et c'est bien ce qui se passe ! En effet, dans certains *cas réputés fréquents*, avant de provoquer un appel de `new`, Java examine si la valeur en question n'a pas déjà été emballée, auquel cas l'objet-enveloppe précédemment construit est réutilisé. Ainsi, les deux lignes

```
Integer a = 123;
Integer b = 123;
```

ont le même effet que

```
Integer a = new Integer(123);
Integer b = a;
```

ce qui explique l'égalité trouvée.

Les « cas réputés fréquents », c'est-à-dire les valeurs pour lesquelles la machine cherche à savoir si elles ont déjà été emballées avant de créer un nouvel objet, sont les petites valeurs ; plus précisément :

- les booléens `false` et `true`,
- les valeurs de type `byte`, `short` et `int` comprises entre -128 et 127,
- les caractères dont le code est compris entre `\u0000` (0) et `\u007F` (127)

12.2.3 La résolution de la surcharge en Java 5

Dans quelle mesure le mécanisme de l'emballage et déballage automatiques complique-t-il celui de la résolution de la surcharge que Java emploie par ailleurs ? Supposons, par exemple, disposer d'une certaine méthode `traitement` surchargée :

```
void traitement(double x) { ... }
...
void traitement(Integer y) { ... }
```

a l'occasion d'un appel comme

```
int z;
...
traitement(z);
```

laquelle des deux méthodes doit-on activer ? Autrement dit, le type `Integer` est-il considéré plus proche du type `int` que le type `double`, ou le contraire ? Les concepteurs de Java 5 ont décidé de privilégier une certaine forme de compatibilité avec Java 1.4, ainsi la conversion vers le type primitif (ici `double`) est préférée. La règle précise est la suivante :

En Java 5 la résolution de la surcharge se fait en trois temps :

1. Le compilateur cherche d'abord à résoudre l'appel d'une méthode surchargée sans utiliser l'emballage et déballage automatiques ni les arguments variables (cf. § 12.3.3). Si l'appel peut être résolu ainsi, alors les règles appliquées auront été les même qu'en Java 1.4.
2. Si la première étape a échoué, alors le compilateur cherche à résoudre l'appel en se permettant des emballages et des déballages, mais sans considérer les méthodes avec des arguments variables.
3. Enfin, si l'étape 2 a échoué aussi, alors le compilateur cherche à résoudre l'appel en considérant les méthodes avec des arguments variables.

12.3 Quelques outils pour simplifier la vie du programmeur

12.3.1 Importation de membres statiques

Ou : *comment raccourcir encore les noms de certaines entités ?* La directive `import` que nous connaissons permet de mentionner une classe publique sans avoir à préfixer son nom par celui de son paquetage. C'est bien, mais cela n'empêche pas que les autres entités manipulées dans les programmes, comme les membres des classes, ont tendance à avoir des noms à rallonges. Ne pourrait-on pas faire d'autres simplifications du même genre ?

Pour les membres ordinaires – non statiques – on ne peut pas grand chose : ils doivent être préfixés par un objet car cela est dans leur nature même de *membres d'instance*, c'est-à-dire nécessairement liés à un objet. Seule simplification, bien connue : quand cet objet est `this`, la plupart du temps il peut être omis.

Il n'en est pas de même pour les *membres de classe* (membres qualifiés `static`), cela a permis de simplifier leur emploi en Java 5. Jusqu'à Java 1.4, ces membres doivent être écrits préfixés par le nom de leur classe ; il s'agit d'éviter d'éventuelles collisions de noms, ce qui est un risque potentiel qui n'est pas toujours effectif : les membres de classes différentes ont une petite tendance à avoir des noms différents. Ne pourrait-on pas, lorsqu'il n'y a pas de collision, permettre que les noms des membres statiques soient utilisés sans préfixe ?

C'est la nouvelle directive `import static` qui rend ce service. Elle s'emploie sous deux formes :

`import static` *nomCompletDeClasse*`.`*nomDeMembreStatique*`;`
`import static` *nomCompletDeClasse*`.*;`

Dans la première forme, le nom du membre statique indiqué pourra être utilisé seul, c'est-à-dire non prefixé par le nom de la classe. Dans la deuxième forme, *tous* les membres statiques de la classe indiquée pourront être utilisés seuls.

Exemple. Le programme `Sinus` affiche le sinus d'un angle donné en degrés. Version traditionnelle :

```
public class Sinus {
    public static void main(String[] args) {
        double x = Double.parseDouble(args[0]) * Math.PI / 180;
        double y = Math.sin(x);
        System.out.println("sin(" + x + ") = " + y);
    }
}
```

La version avec des membres statiques importés est plus légère (en faisant abstraction des directives `import`) :

```
import static java.lang.Math.*;
import static java.lang.Double.parseDouble;
import static java.lang.System.out;

public class Sinus {
    public static void main(String[] args) {
        double x = parseDouble(args[0]) * PI / 180;
        double y = sin(x);
        out.println("sin(" + x + ") = " + y);
    }
}
```

Note 1. La directive `import static` se révèle précieuse pour simplifier l'emploi des types énumérés (cf. § 12.1.1), puisque les valeurs de ces derniers sont des membres (implicitement) statiques. Par exemple, étant donné le type

```
public enum JourSemaine { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE }
```

dans un programme où on doit utiliser ces valeurs, l'écriture de la directive

```
import static JourSemaine.*;
```

permettra d'écrire `LUNDI`, `MARDI`, etc. au lieu de `JourSemaine.LUNDI`, `JourSemaine.MARDI`, etc.

Note 2. Bien entendu, l'emploi de `import static` fait réapparaître le risque de conflits de noms que l'écriture explicite des noms des classes cherchait à éviter. Il faut signaler cependant que ce risque est minimisé par le fait que le compilateur ne se déclare en situation de conflit que lorsqu'il est effectivement placé devant un accès à une variable ou un appel de méthode représentant pour lui une ambiguïté insoluble.

Pour illustrer cette bonne volonté du compilateur, imaginez deux classes `A` et `B` ainsi définies

```
public class A {
    public static void fon(int i) { ... }
    ...
}
public class B {
    public static void fon(float x) { ... }
    ...
}
```

Il est remarquable de constater que dans un fichier qui comporte les deux lignes suivantes

```
import static A.fon;      // ou import static A.*
import static B.fon;      // ou import static B.*
...
```

la méthode `fon` sera traitée comme une méthode surchargée ordinaire : sur la rencontre d'un appel comme `fon(u)`, l'une ou l'autre définition sera choisie selon le type de `u`.

12.3.2 Boucle *for* améliorée

Une nouvelle construction, dite *boucle for améliorée*, a été ajoutée au langage dans le but d'alléger l'écriture de deux sortes de boucles extrêmement fréquentes : le parcours d'un tableau et le parcours d'une collection.

1° Si `tableau` est un tableau d'éléments d'un certain type *TypeElement*, qui peut être un type primitif, un tableau ou une classe, la boucle

```
for (int i = 0; i < tableau.length; i++)
    exploiter( tableau[i] )
```

peut s'écrire plus simplement :

```
for (TypeElement elt : tableau)
    exploiter( elt )
```

2° Si `liste` est une structure de données susceptible d'être parcourue au moyen d'un *itérateur* – c'est-à-dire si *liste* implémente la nouvelle interface `java.lang.Iterable` – la boucle

```
for (Iterator iter = liste.iterator(); iter.hasNext(); )
    exploiter( (TypeElement)iter.next() )
```

peut s'écrire plus simplement :

```
for (TypeElement elt : liste)
    exploiter(elt)
```

Exemple 1. Pour illustrer le parcours d'un tableau, voici un bout de code qui affiche une matrice :

```
double[][] matrice = new double[NL][NC];
...
acquisition des valeurs de la matrice
...
for (double[] ligne : matrice) {
    for (double x : ligne)
        System.out.print(x + " "); // ou, plus malin : System.out.printf("%8.3f ", x);
    System.out.println();
}
```

Exemple 2. Pour illustrer le parcours d'une collection, voici un bout de code qui effectue un certain *traitement* sur chaque élément d'un vecteur donné, en prenant soin de parcourir non pas le vecteur en question mais le résultat du clonage [107] de ce dernier (notez que la spécification de la boucle *for* améliorée garantit que l'opération `v.clone()` sera effectuée une seule fois, en commençant la boucle) :

```
Vector v = new Vector();
...
for (Object o : (Vector) v.clone())
    traitement(o);
```

107. Parcourir un clone au lieu de parcourir la structure elle-même paraît bizarre, mais c'est ce qu'on doit faire s'il est possible que le *traitement* modifie le vecteur – par exemple, en lui ajoutant ou en lui supprimant des éléments.

MISE EN GARDE. On peut trouver beaucoup de qualités à la nouvelle boucle *for* améliorée, mais il faut réaliser qu'elle ne peut pas remplacer en toute circonstance la boucle *for* ordinaire, tout simplement parce qu'elle n'offre pas un accès (indice, pointeur, etc.) à la position dans la structure de l'élément en cours d'exploitation.

Ainsi, un code aussi simple que le suivant ne gagne pas en concision ni en efficacité lorsqu'on le transforme afin de l'écrire en utilisant la boucle *for* améliorée :

```
int i;
...
for (i = 0; i < tab.length; i++)
    if ( condition( tab[i] ) )
        break;
utilisation de la valeur de i
```

Objets itérables

Peut-on parcourir ses propres structures avec la nouvelle boucle *for* améliorée ? Autrement dit, quelle propriété doit avoir un objet `U` pour pouvoir figurer dans une expression « `for ( déclaration : U )` » ? La réponse est simple, il suffit que notre objet implémente l'interface `java.lang.Iterable` :

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

Par exemple, voici la définition d'une classe `Intervalle`[108] dont les instances représentent des intervalles de nombres entiers et peuvent, à ce titre, être utilisées dans des boucles *for* améliorées[109] :

```
public class Intervalle implements Iterable<Integer> {
    private int inf, sup;

    public Intervalle(int inf, int sup) {
        this.inf = inf;
        this.sup = sup;
    }

    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            private int pos = inf;

            public boolean hasNext() {
                return pos <= sup;
            }
            public Integer next() {
                return pos++;   // emballage automatique
            }
            public void remove() {
            }
        };
    }
}
```

Exemple d'emploi[110] (notez qu'il y a déballage automatique des valeurs successivement produites par l'itérateur, puisque `i` est déclarée `int`) :

```
...
Intervalle plageDeNombres = new Intervalle(-5, 15);
for (int i : plageDeNombres)
    System.out.print(i + " ");
...
```

Affichage obtenu :

```
-5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

108. Les amateurs de serpents reconnaîtront l'objet `xrange` du langage *Python*

109. Nous utilisons ici les types génériques – cf. § 12.5 – `Iterable<Integer>` et `Iterator<Integer>` qui, avec le déballage automatique qu'ils induisent, rendent cet exemple plus intéressant.

110. Il s'agit ici d'un exemple purement démonstratif. Pour parcourir les entiers compris entre deux bornes `inf` et `sup` il est sans doute plus efficace d'utiliser une boucle *for* classique : `for (int i = inf; i <= sup; i++) ...`

12.3.3 Méthodes avec une liste variable d'arguments

Java 5 permet d'appeler une même méthode [111] avec un nombre d'arguments effectifs qui varie d'un appel à un autre. Ces arguments variables doivent partager un type commun, qui peut être une super-classe de leurs types effectifs. Dans la déclaration de la méthode, ce type commun doit apparaître suivi de trois points « `...` » et un identificateur. Exemple :

```
void afficherScores(int dossard, String nom, Number... scores) {
    corps de la méthode
}
```

A l'intérieur de la méthode, l'identificateur associé aux arguments variables (ici `scores`) est le nom d'un tableau dont les éléments sont les arguments effectifs figurant dans l'appel. Voici une écriture possible de `afficherScores` :

```
void afficherScores(int dossard, String nom, Number... scores) {
    System.out.println(nom + " dossard n° " + dossard);
    for (int i = 0; i < scores.length; i++)
        System.out.println("    " + scores[i] + " points");
}
```

Cette méthode devra être appelée avec pour arguments un entier, une chaîne de caractères et un nombre quelconque d'instances de sous-classes de la classe `Number`. Par exemple :

```
afficherScores(100, "Durand", new Integer(10));
afficherScores(101, "Dupond", new Integer(10), new Integer(15));
afficherScores(102, "Dubois", new Integer(10), new Long(15), new Double(20.5));
```

On peut écrire les appels de cette méthode plus simplement, en mettant à profit l'emballage automatique (cf. § 12.2) :

```
afficherScores(102, "Dubois", 10, 15, 20.5);
```

D'autre part, ce mécanisme peut aussi être utilisé avec des listes variables d'arguments de types primitifs. La déclaration de la méthode `afficherScores` aurait pu être

```
void afficherScores(int dossard, String nom, double... scores) {
    le corps de la méthode reste le même
}
```

Remarque 1. Quand une méthode comporte une liste variable d'arguments elle peut être appelée de deux manières :

- en faisant correspondre à l'argument variable un certain nombre d'arguments individuels distincts (ayant des types compatibles avec la déclaration),
- en faisant correspondre à l'argument variable un unique tableau (d'éléments ayant un type compatible).

Ainsi, les deux appels suivants sont équivalents :

```
int[] tab = { 10, 20, 30, 25, 15 };
...
afficherScores(102, "Dubois", 10, 20, 30, 25, 15);
afficherScores(102, "Dubois", tab);
```

A l'intérieur de la méthode, rien ne permet de distinguer le premier appel du second.

Remarque 2. La possibilité d'avoir des listes variables d'arguments interfère manifestement avec le mécanisme de la surcharge. Imaginez deux méthodes distinctes avec les prototypes suivants :

```
void calcul(int x, int y) { ... }

void calcul(int... z) { ... }
```

La question est : comment est résolu un appel tel que que `calcul(a, b)` ? La réponse à été donnée au § 12.2.3 : le compilateur essaie d'abord de résoudre la surcharge *sans* prendre en compte les méthodes avec liste variable d'arguments ; il ne considère ces méthodes que lorsqu'une telle résolution échoue. Ici, l'appel `calcul(a, b)` activera donc la première méthode, tandis que des appels comme `calcul(a)` ou `calcul(a, b, c)` activeraient la deuxième.

111. Attention, nous ne parlons pas ici de surcharge, c'est-à-dire de la possibilité d'appeler des méthodes *distinctes* ayant le même nom, mais bien d'appeler *une même méthode* avec un nombre d'arguments qui diffère d'un appel à un autre.

12.3.4 Entrées-sorties simplifiées : `printf` et `Scanner`

La méthode `printf`

Les vieux routiers de la programmation en C n'en croiront pas leurs yeux : à l'occasion de la version 5 on a incorporé à Java leur chère fonction `printf` ! Mais oui ! Exemple (inutile mais démonstratif) :

```
...
double x = 123456;
for (int i = 0; i < 8; i++) {
    System.out.printf("%03d | %12.4f |%n", i, x);
    x /= 10;
}
...
```

Affichage obtenu :

```
000 |  123456,0000 |
001 |   12345,6000 |
002 |    1234,5600 |
003 |     123,4560 |
004 |      12,3456 |
005 |       1,2346 |
006 |       0,1235 |
007 |       0,0123 |
```

La méthode `printf` appartient aux classes `PrintStream` (flux d'*octets* en sortie représentant des données mises en forme) et `PrintWriter` (flux de *caractères* en sortie représentant des données mises en forme). Dans les deux cas il y en a deux versions :

```
public fluxEnQuestion printf(Locale locale, String format, Object... args);
public fluxEnQuestion printf(String format, Object... args);
```

La première forme permet de préciser quelles particularités locales doivent être employées, la deuxième utilise les particularités locales courantes. Dans tous les cas la méthode renvoie comme résultat le flux sur lequel elle a été appelée (cela permet les cascades, genre : *flux*`.printf(...).printf(...)`).

L'argument `format` est une chaîne composée de *caractères ordinaires*, qui seront simplement recopiés dans le flux de sortie, parmi lesquels se trouvent un certain nombre de *spécifications de format* qui indiquent comment faut-il mettre en forme les données, représentées par l'argument variable `args`, placées à la suite de l'argument `format`.

Les spécifications de format se présentent ainsi (les crochets expriment le caractère facultatif de ce qu'ils encadrent) :

$$\texttt{\%}[\textit{drapeau}][\textit{largeur}][\texttt{.}\,\textit{precision}]\textit{type}$$

Les principaux *drapeaux* sont

`-` : commande le cadrage à gauche,

`+` : précise que les données numériques doivent toujours avoir un signe,

espace : indique que les nombres positifs doivent être précédés d'un blanc,

`0` : spécifie que les espaces libres précédant les nombres doivent être remplis avec 0,

Largeur et *précision* sont des entiers : le premier précise le nombre total de caractères que la donnée doit occuper une fois mise en forme ; le second le nombre de chiffres après la virgule.

Les principales indications de *type* sont :

`%%` : pour indiquer le caractère `%`

`%b, %B` : pour afficher un booléen sous la forme `false`, `true` [resp. `FALSE`, `TRUE`].

`%c, %C` : pour afficher un caractère dont la donnée correspondante (qui doit être de type `Byte`, `Short`, `Character` ou `Integer`) exprime le code interne.

`%d` : pour afficher un entier écrit en base 10.

`%e, %E` : pour afficher un nombre flottant en notation exponentielle.

`%f` : pour afficher un nombre flottant sans utiliser la notation exponentielle.

`%g, %G` : pour afficher un nombre flottant avec la notation qui convient le mieux à sa valeur.

`%n` : pour afficher l'indicateur de fin de ligne de la plate-forme utilisée (c'est plus fiable que `\n`).

`%s, %S` : pour afficher une chaîne de caractères.

`%x, %X` : pour afficher un nombre en hexadécimal.

Il y a aussi de nombreuses – pas moins de 32 ! – spécifications de format concernant la date et l'heure. Vous trouverez toutes les informations sur la méthode `printf` dans la documentation de l'API, à l'occasion de la classe `java.util.Formatter`.

La classe `Scanner`

La classe `java.util.Scanner` est faite de méthodes pour lire simplement un flux de données formatées. Ensemble, ces méthodes rendent le même genre de services que la fonction `scanf` bien connue des programmeurs C.

On construit un objet `Scanner` au-dessus d'un flux d'octets ou de caractères préalablement ouvert ; dans le cas – le plus simple – de la lecture à la console, ce flux est `System.in`.

Les méthodes de la classe `Scanner` les plus intéressantes sont certainement :

`byte nextByte()`
: Lecture de la prochaine valeur de type `byte` (c'est-à-dire : consommation de tous les caractères se présentant dans le flux d'entrée qui peuvent faire partie d'une données de ce type, construction et renvoi de la valeur correspondante).

`short nextShort()`
: Lecture de la prochaine valeur de type `short`.

`int nextInt()`
: Lecture de la prochaine valeur de type `int`.

`long nextLong()`
: Lecture de la prochaine valeur de type `long`.

`float nextFloat()`
: Lecture de la prochaine valeur de type `float`.

`double nextDouble()`
: Lecture de la prochaine valeur de type `double`.

`String nextLine()`
: Lecture de tous les caractères se présentant dans le flux d'entrée jusqu'à une marque de fin de ligne et renvoi de la chaîne ainsi construite. La marque de fin de ligne est consommée, mais n'est pas incorporée à la chaîne produite.

`BigInteger nextBigInteger()`
: Lecture du prochain grand entier.

`BigDecimal nextBigDecimal()`
: Lecture du prochain grand décimal.

Exemple très simple :

```
public class TestScanner {
    public static void main(String[] args) {
        Scanner entree = new Scanner(System.in);

        System.out.print("nom et prénom? ");
        String nom = entree.nextLine();
        System.out.print("âge? ");
        int age = entree.nextInt();
        System.out.print("taille (en m)? ");
        float taille = entree.nextFloat();

        System.out.println("lu: " + nom + ", " + age + " ans, " + taille + " m");
    }
}
```

Exécution de ce programme :


```
nom et prénom? Tombal Pierre
âge? 32
taille (en m)? 1,72
lu: Tombal Pierre, 32 ans, 1.72 m
```

Erreurs de lecture. Les erreurs lors de la lecture de données sont des fautes que le programmeur ne peut pas *éviter*, quel que soit le soin apporté à la conception de son programme. Il doit donc se contenter de faire le nécessaire pour les *détecter* lors de l'exécution et y réagir en conséquence. En Java cela passe par le mécanisme des exceptions :

```
...
double x = 0;
for (;;) {
    System.out.print("Donnez x: ");
    try {
        x = entree.nextDouble();
        break;
    } catch (InputMismatchException ime) {
        entree.nextLine();
        System.out.println("Erreur de lecture - Recommencez");
    }
}
System.out.print("x = " + x);
...
```

Remarquez, dans l'exemple précédent, comment dans le cas d'une erreur de lecture il convient de « nettoyer » le tampon d'entrée (c'est le rôle de l'instruction `entree.nextLine();`) afin que la tentative suivante puisse se faire avec un tampon vide.

Tests de présence de données. La classe `Scanner` offre aussi toute une série de méthodes booléennes pour tester le type de la prochaine donnée disponible, *sans lire* cette dernière :

`boolean hasNext()`
: Y a-t-il une donnée disponible pour la lecture ?

`boolean hasNextLine()`
: Y a-t-il une ligne disponible pour la lecture ?

`boolean hasNextByte()`
: La prochaine donnée à lire est-elle de type `byte` ?

`boolean hasNextShort()`
: La prochaine donnée à lire est-elle de type `short` ?

`boolean hasNextInt()`
: La prochaine donnée à lire est-elle de type `int` ?

`boolean hasNextLong()`
: La prochaine donnée à lire est-elle de type `long` ?

`boolean hasNextFloat()`
: La prochaine donnée à lire est-elle de type `float` ?

`boolean hasNextDouble()`
: La prochaine donnée à lire est-elle de type `double` ?

`boolean hasNextBigInteger()`
: La prochaine donnée à lire est-elle un grand entier ?

`boolean hasNextBigDecimal()`
: La prochaine donnée à lire est-elle un grand décimal ?

12.4 Annotations

12.4.1 Principe

Les *annotations*, on dit aussi *méta-données*, sont un mécanisme permettant d'accrocher des informations « périphériques » aux classes et méthodes. Ces informations ne modifient pas la sémantique des programmes, mais s'adressent à des outils qui auraient à manipuler ces derniers. Pour cette raison, les annotations ne disparaissent pas lors de la compilation [112] ; au contraire, elles sont reconnues et traitées par le compilateur

112. Cela est une différence notable entre les annotations et, par exemple, les *commentaires de documentation* (adressés à l'outil `javadoc`) dont il ne reste aucune trace après la compilation.

et sont conservées avec les classes produites – sauf instruction contraire. En contrepartie, cela les oblige à obéir à une syntaxe aussi contraignante que celle des programmes.

Pour pouvoir être utilisées, les annotations doivent d'abord être déclarées par un texte, définissant ce qu'on appelle un *type annotation*, qui est bizarrement voisin d'une déclaration d'interface :

```
public @interface nomTypeAnnotation {
    déclaration des éléments de l'annotation
}
```

Les éléments d'une annotation sont des données, analogues à des variables finales (i.e. des constantes), mais il faut les déclarer par des expressions qui ressemblent à des déclarations de méthodes :

```
type nomElement();
```

avec la particularité qu'on peut indiquer une valeur par défaut :

```
type nomElement() default valeur;
```

Par exemple, voici un type annotation à quatre éléments destinée à signaler les méthodes d'une classe dont on estime qu'elles peuvent être améliorées :

```
public @interface AmeliorationRequise {
    int identification();
    String synopsis();
    String auteur() default "(plusieurs)";
    String date() default "non indiquée";
}
```

Lorsqu'un type annotation a un seul élément, celui-ci *doit* se nommer `value` :

```
public @interface Copyright {
    String value();
}
```

Enfin, un type annotation peut aussi n'avoir aucun élément ; elle est alors dite *annotation de marquage* (*marker annotation*) :

```
public @interface Test { }
```

Voici une classe dont les méthodes ont été annotées en utilisant les trois types annotations précédents. Notez que

- dans le cas d'une annotation à un seul élément, on peut écrire « *annotation*`(`*valeur*`)` » au lieu de « *annotation*`(value =` *valeur*`)` »
- dans le cas d'une annotation sans éléments, on peut écrire « *annotation* » au lieu de « *annotation*`()` »

```
public class ObjetAnnote {
    @Test public void unePremiereMethode(arguments) {
        corps de la méthode
    }

    @Copyright("Editions Dupuis, 2004")
    public void uneAutreMethode(arguments) {
        corps de la méthode
    }

    @AmeliorationRequise(identification = 80091,
            synopsis = "Devrait être interruptible",
            date = "11/09/2004")          // ici auteur aura la valeur "(plusieurs)"
    public void uneTroisiemeMethode(arguments) {
        corps de la méthode
    }
    autres méthodes
}
```

12.4.2 Exploitation des annotations

A l'exécution, l'accès aux annotations portées par un exécutable (ensemble de classes) se fait à travers les méthodes de l'interface `java.lang.AnnotatedElement`, dont voici une définition succincte :

```
package java.lang.reflect;

public interface AnnotatedElement {

        // renvoie un tableau contenant toutes les annotations portées par cet élément :
    Annotation[] getAnnotations();

        // renvoie le tableau des annotations faites directement (non héritées) sur cet élément :
    Annotation[] getDeclaredAnnotations();

        // renvoie true si et seulement si cet élément porte une annotation ayant le type indiqué :
    boolean isAnnotationPresent(Class<? extends Annotation> annotationType);

        // renvoie l'annotation de cet élément ayant le type indiqué, ou null si elle n'existe pas
    <T extends Annotation> T getAnnotation(Class<T> annotationType);
}
```

Bien entendu, cette interface est désormais implémentée par la plupart des classes qui constituent le mécanisme de la réflexion de Java (cf. § 9.3) : `Class`, `Constructor`, `Field`, `Method`, `Package`, etc. Un exemple est montré à la fin de la section 12.4.1.

Voyons cela à travers deux exemples simples :

Exemple 1. Le programme suivant essaie toutes les méthodes portant l'annotation `Test` dans une classe quelconque, donnée par son nom (en supposant que toutes ces méthodes sont statiques et sans argument) ; le point important dans cet exemple est la méthode `isAnnotationPresent` :

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

public class ExecuterTests {

    public static void main(String[] args) throws ClassNotFoundException {
        int nbrSucces = 0;
        int nbrEchecs = 0;

        for (Method uneMethode : Class.forName(args[0]).getMethods()) {
            if (uneMethode.isAnnotationPresent(Test.class)) {
                try {
                    uneMethode.invoke(null);
                    nbrSucces++;
                } catch (Throwable ex) {
                    System.out.println("Test " + uneMethode
                                                + " échec: " + ex.getCause());
                    nbrEchecs++;
                }
            }
        }
        System.out.println(nbrSucces + " succès et " + nbrEchecs + " échecs");
    }
}
```

Exemple 2. Le programme suivant affiche les valeurs de toutes les propriétés de toutes les annotations attachées à toutes les méthodes d'une classe donnée par son nom :

```
import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
```

```
public class MethodesAnnotees {
    public static void main(String[] args) throws Exception {
        Method[] methodes = Class.forName(args[0]).getMethods();
        for (Method uneMethode : methodes) {
            Annotation[] annotations = uneMethode.getAnnotations();
            if (annotations.length > 0) {
                System.out.println(uneMethode.getName());
                for (Annotation uneAnnotation : annotations) {
                    Class type = uneAnnotation.annotationType();
                    System.out.println("  " + type.getName());
                    for (Method propr : type.getDeclaredMethods())
                        System.out.println("    " + propr.getName()
                                + " : " + propr.invoke(uneAnnotation));
                }
            }
        }
    }
}
```

Par exemple, s'il était appelé avec pour argument le nom de la classe `ObjetAnnote` montrée à la page 162, ce programme afficherait :

```
unePremiereMethode
  Test
uneAutreMethode
  Copyright
    value : Editions Dupuis, 2004
uneTroisiemeMethode
  AmeliorationRequise
    identification : 80091
    synopsis : Devrait être interruptible
    auteur : (plusieurs)
    date : 11/09/2004
```

12.4.3 Annotations prédéfinies

Un petit nombre (appelé à grandir ?) d'annotations sont prédéfinies et peuvent être utilisées sans autre précaution :

`@Deprecated` – Le compilateur affichera un avertissement lors de tout emploi d'un membre ainsi annoté [113].

`@Override` – Indique que la méthode ainsi annotée *doit* redéfinir une méthode héritée : si tel n'est pas le cas, un message d'erreur sera produit par le compilateur.

`@SuppressWarnings("`*type d'avertissement*`")` – Demande au compilateur de ne pas produire une certaine sorte de messages d'avertissement durant la compilation de l'élément de code ainsi annoté.

Par exemple, cette annotation permet que le clonage d'un vecteur générique puisse se faire « en silence » [114] :

```
Vector<String> a = new Vector<String>();
...
@SuppressWarnings("unchecked")
Vector<String> b = (Vector<String>) a.clone();
...
```

12.4.4 Méta-annotations

Les annotations définies par l'utilisateur peuvent être annotées à leur tour, par des annotations appelées alors *méta-annotations*. Les types des méta-annotations sont prédéfinis ; à l'heure actuelle il y en a quatre :

113. L'annotation `@Deprecated` rend donc un service qui était assuré précédemment par la marque `@deprecated` écrite dans un commentaire « `/** ... */` » à l'adresse de l'outil `javadoc`.
114. A propos des conversions « `unchecked` » voir l'explication des *types bruts* à la section 12.5.2

`@Documented` – Indique que l'annotation que cette méta-annotation annote doit être prise en compte par l'outil `javadoc` et d'autres outils similaires, comme cela est fait pour les autres éléments (classes, interfaces, etc.) qui composent le programme source.

`@Inherited` – Indique que l'annotation concernée est automatiquement héritée. Si un membre est ainsi annoté et ses descendants (sous-classes, méthodes redéfinies, etc.) ne le sont pas, alors ces derniers prendront automatiquement cette annotation.

`@Retention` – Indique jusqu'à quel point du processus de développement l'annotation concernée doit être conservée. Les valeurs possibles sont
- `RetentionPolicy.SOURCE` : l'annotation sera éliminée par le compilateur (lequel aura cependant « vu » l'annotation, contrairement à ce qui arrive pour un commentaire),
- `RetentionPolicy.CLASS` : le compilateur laissera l'annotation dans la classe compilée mais la machine virtuelle l'éliminera lors du chargement de la classe,
- `RetentionPolicy.RUNTIME` : l'annotation sera conservée dans la classe compilée et retenue par la machine virtuelle, si bien qu'elle sera accessible – par les mécanisme de la *réflexion* – pendant l'exécution du programme.

La valeur par défaut est `RetentionPolicy.CLASS`.

`@Target` – Indique sur quel type d'élément l'annotation porte. Plusieurs types peuvent être indiqués en même temps (la valeur de cette méta-annotation est un tableau de tels types). Les types possibles sont
- `ElementType.ANNOTATION_TYPE` : type annotation,
- `ElementType.CONSTRUCTOR` : constructeur,
- `ElementType.FIELD` : champ (variable d'instance),
- `ElementType.LOCAL_VARIABLE` : variable locale,
- `ElementType.METHOD` : méthode,
- `ElementType.PACKAGE` : paquetage,
- `ElementType.PARAMETER` : paramètre d'une méthode,
- `ElementType.TYPE` : classe, interface ou type énumération (`enum`).

La valeur par défaut est *tous les types*.

12.5 Généricité

Emblématique de Java 5, la généricité est peut-être la plus importante des extensions du langage faites à l'occasion de cette version. Il s'agit d'obtenir en Java rien moins que les services que rendent les *templates* en C++, c'est-à-dire la possibilité de définir et d'employer des classes et des méthodes *paramétrées par des types* qui interviennent dans leur définition.

Par exemple, une classe `Truc` étant définie par ailleurs, cela nous permettra de déclarer une collection `v` comme étant de type *liste de trucs* (cela s'écrira `List<Truc>`) au lieu du simple type `List`, c'est-à-dire *liste d'objets quelconques*. Les bénéfices obtenus sont faciles à entrevoir :
- lors d'ajouts d'éléments à `v`, la vérification que ceux-ci sont bien de type `Truc`,
- lors d'accès à des éléments de `v`, la certitude que les objets obtenus sont de type `Truc`.

Voici à quoi cela ressemblera :

```
// déclaration et création de la liste
List<Truc> liste = new Vector<Truc>();
...
// remplissage de la liste
for (int i = 0; i < n; i++) {
    liste.add(expression);       // ici il est contrôlé que expression est un Truc
}
...
// exploitation des éléments de la liste
for (int i = liste.size() - 1; i >= 0; i--) {
    Truc q = liste.get(i);     // pas besoin de conversion, ce qui sort de la liste
                               // est connu comme étant de type Truc
    exploitation du truc q
}
```

En permettant aux compilateurs d'effectuer des contrôles de type nombreux et fins, la généricité augmente significativement la qualité et la fiabilité des programmes. Hélas, dans certains langages elle en augmente aussi la complication, parfois de manière considérable. On apprendra donc avec soulagement qu'en Java :

– la généricité est totalement prise en charge au moment de la compilation et n'entraîne *aucune modification de la machine java*,
– la généricité est totalement compatible avec le langage préexistant : là où un objet « à l'ancienne » est attendu on peut mettre une instance d'une classe générique, et réciproquement [115].

12.5.1 Classes et types paramétrés

Pour commencer il nous faut trois nouveaux concepts :

1° Une *classe paramétrée* est une déclaration de classe dans laquelle le nom de la classe est suivi des signes `< >` encadrant une liste de *types paramètres (éventuellement) contraints* séparés, lorsqu'il y en a plusieurs, par des virgules. Exemple :

```
class Machin<A, B extends Number, C extends Collection & Cloneable> {
...
}
```

Comme on le devine ci-dessus, les contraintes, lorsqu'elles existent, sont de la forme (les crochets signalent le caractère facultatif de ce qu'ils encadrent) :

`extends` *classeOuInterface* [`&` *interface* ... `&` *interface*]

cette expression doit être comprise, selon que *classseOuInterface* est une classe ou une interface, respectivement comme

`extends` *classe* `implements` *interface*, ... *interface*

ou bien comme

`implements` *interface*, *interface*, ... *interface*

2° Les *types variables* (ou *types paramètres*) sont les identificateurs constituant la liste des types paramètres d'une classe paramétrée (`A`, `B` et `C` [116] dans l'exemple précédent). A l'intérieur de la classe, c'est-à-dire dans les définitions de ses membres, ces identificateurs jouent le rôle de types. Exemple :

```
public class Machin<A, B extends Number, C extends Collection & Cloneable > {
    A descripteur;
    B valeur;
    C listeTransactions;

    public Machin(A descripteur, B valeur) {
        this.descripteur = descripteur;
        this.valeur = valeur;
        ...
    }
    ...
}
```

3° Un *type paramétré* est le nom d'une classe paramétrée suivi des signes `< >` encadrant une liste de « vrais » types. Cela représente donc le choix d'un élément dans l'ensemble (infini) de types défini par la classe paramétrée. De telles formules apparaissent le plus souvent dans des déclarations de variables et de méthodes. Exemple [117] :

```
Machin<String, Integer, Vector<Truc>> unMachin =
                    new Machin<String, Integer, Vector<Truc>>("un exemple", 100);
```

Attention. L'opération consistant à produire un type paramétré à partir d'une classe paramétrée (en remplaçant les types paramètres par des types effectifs) est souvent appelée *instanciation* de la classe paramétrée. On prendra garde à l'ambiguïté ainsi introduite, ce terme désignant aussi la création d'un objet à partir [d'un constructeur] de sa classe :

115. Assez souvent cela entraînera un avertissement de la part du compilateur (il vous dira que votre programme « uses *unchecked* or *unsafe* operations »), mais pas un diagnostic d'erreur.
116. La recommandation officielle est de donner à ces types paramètres des noms très courts, en majuscules et *sans signification* : `A`, `B`, `C` ou `T1`, `T2`, `T3`, etc. Cela afin qu'à l'intérieur d'une classe paramétrée on puisse distinguer du premier coup d'œil les types paramètres des « vrais » types, qui ont généralement des noms significatifs, et des membres, qui sont écrits plutôt en minuscules.
117. Les programmeurs C++ noteront avec plaisir que le compilateur Java ne prend pas toujours deux `>` consécutifs pour une occurrence de l'opérateur `>>`. Mais oui, l'humanité progresse !

`new Integer(1000)` : instanciation de la classe `Integer` ; le résultat est un objet,

`Vector<Integer>` : instanciation de la classe paramétrée `Vector<E>` ; le résultat est une classe.

Ce discours devient critique lorsque ces deux opérations sont invoquées dans la même expression :

`new Vector<Integer>()` : instanciation de la classe paramétrée `Vector<E>` et instanciation de la classe ainsi obtenue.

Utilisation des classes paramétrées

On l'aura compris, le plus beau domaine d'utilisation des classes et types paramétrés est celui des collections. On ne s'étonnera donc pas, en consultant la documentation de l'API, de découvrir que toutes les interfaces et les classes collections de la bibliothèque ont été remplacées par une nouvelle forme paramétrée, `Collection<E>`, `Vector<E>`, `Iterator<E>`, etc.

Ces nouvelles classes se combinent très bien avec l'emballage/déballage automatiques (cf. section 12.2) et permettent l'écriture de programmes plus simples et plus fiables que par le passé. Voici un exemple que nous avons déjà montré : le programme suivant compte le nombre d'apparitions de chaque mot donné dans la ligne de commande. En introduisant une table associative (objet `Map`) dont les clés sont nécessairement des chaînes de caractères et les valeurs nécessairement des `Integer` le programmer augmente sensiblement la fiabilité de son programme :

```
import java.util.*;

public class Frequence {
   public static void main(String[] args) {
      Map<String, Integer> tableAssoc = new TreeMap<String, Integer>();
      for (String mot : args) {
          Integer freq = tableAssoc.get(mot);
          tableAssoc.put(mot, freq == null ? 1 : freq + 1);
      }
      System.out.println(tableAssoc);
   }
}
```

Pour montrer la définition d'une classe paramétrée, voici celle de la classe `Paire` dont les instances représentent des paires de choses :

```
public class Paire<P, S> {
    private P premier;
    private S second;

    public Paire(P premier, S second) {
        this.premier = premier;
        this.second = second;
    }

    public P getPremier() {
        return premier;
    }

    public S getSecond() {
        return second;
    }

    public String toString() {
        return "<" + premier + "," + second + ">";
    }

    public Paire<S, P> swap() {
        return new Paire<S, P>(second, premier);
    }
```

```
    public static <A, B> Paire<A, B> makeInstance(A premier, B second) {
        return new Paire<A, B>(premier, second);
    }
    ...
}
```

12.5.2 Types bruts

Comme il a été dit, pour ce qui est de la généricité, Java 5 est compatible avec les versions précédentes du langage. Il faut donc que les nouvelles classes paramétrées puissent être utilisées par des programmes qui ne pratiquent pas la généricité et, inversement, il faut que d'anciennes classes déjà écrites puissent être employées dans des programmes qui exploitent massivement la généricité.

C'est pourquoi toute classe paramétrée est considérée par le compilateur comme compatible avec sa version sans paramètres, appelée le *type brut* (*raw type*) correspondant à la classe paramétrée. Par exemple, `Paire` est le type brut associé à la classe paramétrée `Paire<P, S>`.

Ainsi, avec la classe définie ci-dessus, les quatre lignes suivantes sont acceptées par le compilateur

```
...
Paire<String, Number> p1 = new Paire<String, Number>("un", 1);
Paire<String, Number> p2 = new Paire("deux", 2);
Paire p3 = new Paire<String, Number>("trois", 3);
Paire p4 = new Paire("quatre", 4);
...
```

A l'essai on constate que les lignes concernant `p2` et `p4` donnent lieu à un avertissement signalant un appel sans contrôle du constructeur de la classe `Paire<P, S>`. Dans le cas de `p2`, le compilateur est également chagriné en constatant une affectation sans contrôle[118] de `p2` (déclaré de type `Paire<P, S>`) par un objet du type brut `Paire`.

On peut supprimer ces messages d'avertissement (après avoir acquis la certitude qu'ils ne signalent pas une erreur) en utilisant une annotation `@SuppressWarnings` comme expliqué à la section 12.4 :

```
...
@SuppressWarnings("unchecked")
Paire<String, Number> p2 = new Paire("deux", 2);
...
```

NOTE. *A l'exécution* tous les types paramétrés construits au-dessus d'un même type brut sont considérés comme définissant la même classe. Par exemple, le code suivant affiche `true` :

```
...
Paire<String, Number> a = new Paire<String, Number>(...);
Paire<Point, Rectangle> b = new Paire<Point, Rectangle>(...);
...
System.out.println(a.getClass() == b.getClass());
...
```

12.5.3 Types paramétrés et *jokers*

La généricité est une notion plus complexe qu'il n'y paraît, il faut de la prudence y compris dans des situations qui paraissent pourtant évidentes. Par exemple, si la classe `Chat` est une sous-classe de la classe `Mammifere`, il semble naturel de penser que `List<Chat>` est une sous-classe de `List<Mammifere>` (si un chat est une sorte de mammifère alors une liste de chats est une sorte de liste de mammifères, cela paraît clair !).

C'est pourtant complètement erroné. Pour le voir il suffit de se rappeler que « A sous-classe de B » signifie « tout ce qu'on peut demander à un B on peut le demander à un A ». Or, il y a des choses qu'on peut demander à une liste de mammifères et pas à une liste de chats : par exemple, l'insertion d'un chien.

Pour être plus précis, supposons que des classes `Animal`, `Mammifere`, `Chat` et `Chien` aient été définies, avec les relations d'héritage que leurs noms suggèrent :

```
class Animal {
    ...
}
```

118. Cette affectation est sans contrôle parce que les concepteurs de Java 5 ont voulu que la généricité soit traitée durant la compilation et qu'elle ne modifie ni le code produit ni la machine virtuelle. Si la généricité avait été traitée à l'exécution il aurait été facile de contrôler de telles opérations.

```
class Mammifere extends Animal {
    ...
}
class Chat extends Mammifere {
    ...
}
class Chien extends Mammifere {
    ...
}
```

Considérons d'autre part une méthode qui prend pour argument une liste de mammifères :

```
void uneMethode(List<Mammifere> menagerie) {
    ...
}
```

et envisageons divers appels possibles de cette méthode :

```
void uneAutreMethode() {
    List<Animal>    listeAnimaux    = new Vector<Animal>();
    List<Mammifere> listeMammiferes = new Vector<Mammifere>();
    List<Chat>      listeChats      = new Vector<Chat>();

    code qui insère des éléments dans les listes précédentes

    uneMethode(listeMammiferes);    (a)    OK
    uneMethode(listeAnimaux);       (b)    ERREUR
    uneMethode(listeChats);         (c)    ERREUR
}
```

L'appel *(a)* est correct. L'appel *(b)* est erroné et ce n'est pas une surprise : une liste d'animaux – pouvant contenir des poissons ou des oiseaux – ne peut pas prendre la place d'une liste de mammifères. Ce qui est peut-être surprenant est que l'appel *(c)* soit incorrect également : pour comprendre pourquoi il suffit de réaliser que `uneMethode` peut contenir l'instruction (légitime, d'après la déclaration de `menagerie`) :

```
menagerie.add(new Chien());
```

Ce serait tout à fait incohérent de laisser faire l'ajout d'un chien dans une liste de chats !

Jokers

Bon, d'accord, une liste de chats n'est pas une liste de mammifères. Mais alors, comment faire pour qu'une méthode puisse prendre pour argument aussi bien une liste de mammifères qu'une liste de chats, ou une liste de chiens, etc. ? Ou bien, comment signifier que l'argument d'une méthode est, par exemple, une liste d'*une sorte de* mammifères ?

Les syntaxes nouvelles « `?` » (signifiant une classe quelconque) et « `? extends` *uneClasse* » (signifiant « une sous-classe quelconque de *uneClasse* ») ont été introduites à cet effet. Exemple :

```
void uneMethode(List<? extends Mammifere> menagerie) {
    ...
}
```

Maintenant, l'appel *(c)* qui posait problème est devenu légal :

```
void uneAutreMethode() {
    ...
    uneMethode(listeMammiferes);          // OK
    uneMethode(listeAnimaux);             // ERREUR
    uneMethode(listeChats);               // OK
    ...
}
```

Evidemment, si l'appel ci-dessus est accepté c'est que la notation `<? extends Mammifere>` impose des contraintes additionnelles. Ainsi, à l'intérieur de `uneMethode`, la liste `menagerie` sera considérée comme un objet en *lecture seule* et toute modification de la liste sera interdite :

```
void uneMethode(List<? extends Mammifere> menagerie) {
    ...
    Mammifere mam = menagerie.get(i);       // Pas de problème : ce qui sort de la liste peut
    ...                                     // certainemnt être vu comme un Mammifere

    menagerie.add(new Chien());             // ERREUR (ben oui, menagerie est peut-être
    ...                                     // une liste de chats... !)
}
```

Symétrique de la précédente, la syntaxe nouvelle « `? super` *uneClasse* » (signifiant « une super-classe quelconque de *uneClasse* ») permet, par exemple, de définir des listes qu'on pourra modifier à l'aide d'objets de type *uneClasse* :

```
void uneTroisiemeMethode(List<? super Mammifere> menagerie) {
    ...
}
```

Les appels légitimes ne seront pas les mêmes que précédemment :

```
void uneAutreMethode() {
    ...
    uneTroisiemeMethode(listeMammiferes);   // OK
    uneTroisiemeMethode(listeAnimaux);      // OK
    uneTroisiemeMethode(listeChats);        // ERREUR
    ...
}
```

Ce qu'il est permis de faire à l'intérieur de la méthode change aussi :

```
void uneTroisiemeMethode(List<? super Mammifere> menagerie) {
    ...
    Mammifere mam = menagerie.get(i);       // ERREUR (menagerie est peut-être
    ...                                     // une liste de simples animaux)
    ...
    menagerie.add(new Chien());             // Pas de problème : un chien peut certainement
    ...                                     // prendre la place d'un élément de menagerie
}
```

12.5.4 Limitations de la généricité

Cette section obscure et incertaine peut être ignorée en première lecture.

Contrairement à d'autres langages, comme C++, où les classes génériques sont instanciées d'abord et compilées ensuite, en Java les classes génériques sont compilées d'abord et – pour autant que cela veuille dire quelque chose – instanciées plus tard. D'autre part, on nous assure que la machine virtuelle Java n'a pas subi la moindre modification lors de l'introduction de la généricité dans le langage.

Cela explique pourquoi certaines opérations impliquant des types paramètres sont possibles, et d'autres non. Pour le dire simplement : un type paramètre `T` ne peut être employé que pour écrire des expressions qui peuvent être compilées sans connaître les détails précis (structure, taille, méthodes...) du type que `T` représente [119]. Ainsi, un type paramètre peut être utilisé :

- dans des déclarations de membres, de variables locales ou d'arguments formels,
- dans des conversions de type (*casts*).

En revanche, un type paramètre *ne peut pas être employé*

- en représentation d'un type primitif,
- comme type des éléments dans la définition d'un tableau,
- intervenant dans la définition d'un membre statique,
- en position de constructeur d'une classe, comme dans « `new T()` »,
- dans une comparaison comme « `T ==` *une classe* »
- dans le rôle d'un objet (instance de la classe `Class`), comme dans « `T.getName()` »

A titre d'exemple, voici quelques bons et mauvais usages des types paramètres :

119. Si cela vous aide, dites-vous que le code produit par la compilation d'une classe paramétrée est le même que celui qu'on aurait obtenu si on avait remplacé tous les types paramètres par `Object` – en faisant abstraction des avertissements et erreurs à propos des types que cela aurait soulevé.

```
public class Zarbi<T> {

    T x;                                    // Pas de problème

    static T y;                             // *** ERREUR ***

    Zarbi<Integer> z;                       // OK

    Zarbi<int> t;                           // *** ERREUR ***

    public T getX() {                       // Très bien
        return x;
    }

    public Zarbi(Object o) {
        x = (T) o;                          // Parfait (du moins pour la compilation)
    }

    public Zarbi() {
        x = new T();                        // *** ERREUR ***
    }

    public boolean estOk(Object x) {
        return x.getClass() == T;           // *** ERREUR ***
    }
}
```

12.5.5 Méthodes génériques

Comme les classes paramétrées, les méthodes peuvent elles aussi dépendre d'une liste de types paramètres. Ces paramètres, encadrés par les signes `< >`, doivent être placés immédiatement devant le type du résultat de la méthode, selon le schéma suivant :

qualifieurs `<` *types-paramètres* `>` *type-du-résultat* *nom-méthode* `(` *arguments* `)`

Par exemple, la méthode suivante prend pour arguments une liste d'objets d'un certain type `T`, un type inconnu durant la compilation, et renvoie l'élément qui est au milieu de la liste ; l'argument est donc de type `List<T>` et le résultat de type `T` :

```
public static <T> T elementCentral(List<T> liste) {
    int n = liste.size();
    if (n > 0)
        return liste.get(n / 2);
    else
        return null;
}
```

Très souvent [120], les types paramètres des méthodes génériques apparaissent dans les types de leurs arguments formels (c'est le cas dans la méthode ci-dessus). Elles ont alors cette particularité : pour les instancier il suffit de les appeler ; le compilateur *déduit* les types paramètres des arguments effectifs figurant dans l'appel.

Exemples d'appels de la méthode précédente (les variables `i` et `s` ont été déclarées par ailleurs) :

120. Il n'est pas formellement requis que les types paramètres d'une méthode paramétrée apparaissent dans les déclarations des arguments. Cependant, à l'usage, on s'aperçoit que cela est une obligation de fait pour obtenir des méthodes correctes *et utiles*.

```
Vector<Integer> v = new Vector<Integer>();
LinkedList<String> ll = new LinkedList<String>();
...
ajout de valeurs aux listes v et ll
...
i = elementCentral(v);
s = elementCentral(ll);
...
```

Sans qu'il ait fallu l'indiquer explicitement, lors du premier appel de la méthode `elementCentral` le compilateur aura pris `T` ≡ `Integer` et, lors du deuxième, `T` ≡ `String`.

Un autre exemple de méthode générique a été donnée par la méthode `makeInstance` de la classe `Paire` (cf. page 167) :

```
public class Paire<P, S> {
    private P premier;
    private S second;

    public Paire(P premier, S second) {
        this.premier = premier;
        this.second = second;
    }

    public static <A, B> Paire<A, B> makeInstance(A premier, B second) {
        return new Paire<A, B>(premier, second);
    }
    ...
}
```

Avec cela, nous avons deux manières de construire des paires d'objets : soit en explicitant les types des membres de la paire (la variable `p` a été déclarée par ailleurs) :

```
p = new Paire<Double, String>(246.5, "Km");
```

soit en laissant le compilateur déduire ces types des arguments d'un appel de `makeInstance` :

```
p = Paire.makeInstance(246.5, "Km");
```

• • •

Index