Langage Java

Claude GUÉGANNO
http://claude.gueganno.free.fr

10 novembre 2004

Table des matières

1 Origine et objectifs de la technologie Java **3**
1.1 Un peu d'histoire. . . 3
1.2 Aperçu de la technologie `Java` 5

2 Création d'applets ou d'applications **9**
2.1 Application `Java` 10
2.2 Applet `Java` 12

3 Éléments de base du langage **13**
3.1 Les types de données 13
3.2 Les variables 14
3.3 Les opérateurs 15

4 Les structures de commande **20**
4.1 Les blocs d'instructions 20
4.2 Les répétitions 21
4.3 Les choix 24

5 Classes et objets **27**
5.1 Généralités 27
5.2 Conception de classe et création d'objets 29
5.3 Exemple plus approfondi 33
5.4 Héritage 38
5.5 Les tableaux 40

6 Définition et gestion des exceptions Java **42**
6.1 Introduction 42
6.2 Les exceptions `Java` 43
6.3 Gestion des exceptions 44
6.4 Déclaration de méthodes pouvant lever des exceptions 45
6.5 Création d'exceptions 46

7 Les opérations d'entrées sorties sur les fichiers **49**
7.1 Lecture de fichier
Classes `FileInputStream` et `DataInputStream` 49

7.2 Cas particulier de l'entrée standard (clavier)
Classe `BufferedReader` 51
7.3 Lecture formatée
Classes `FileReader` et `StreamTokenizer` 54
7.4 Écriture de fichiers : `FileOutputStream` 56
7.5 Application 57

8 Communication par réseau 58
8.1 Le serveur (classe `ServerSocket`) 58
8.2 Le client (Classe `Socket`) 60
8.3 Serveur multisession 62

9 Création d'interfaces graphiques 67
9.1 Traçage de textes et de dessins 67
9.2 Traitement des événements liés à la souris 68
9.3 Gestion des événements liés aux touches du clavier 72
9.4 Les composants évolués 74
9.5 Interface homme machine et zone de traçage 79
9.6 Application `Java` 83

10 Les *packages* 85
10.1 Introduction 85
10.2 Utilisation des *packages* 85
10.3 Création de *packages* 86

11 Classes abstraites et interfaces 88
11.1 Exemple d'héritage, à partir d'une classe abstraite 88
11.2 Avantages et inconvénients 90
11.3 Définition des *interfaces* de Java 90
11.4 Mise en œuvre sur un exemple 90

12 Les Threads 95
12.1 Les bases 95
12.2 Solutions `Java` aux problèmes du parallélisme 97

13 Méthodes natives 102
13.1 Généralités 102
13.2 Mise en œuvre 102
13.3 Bilan des phases de fabrication 107

Chapitre 1

Origine et objectifs de la technologie Java

1.1 Un peu d'histoire...

En décembre 1990, Sun lança le «Project Green» dont l'objet était de créer un prototype d'ordinateur autonome. La réalisation fut un petit ordinateur autonome baptisé le «Star7» construit avec des composants du marché parmi lesquels :

- un écran LCD 5″ couleur 240×240 ;
- un contrôleur graphique ;
- un écran tactile ;
- un processeur RISC SPARC^{TM} ;
- une connexion réseau sans fils (200 kbps) ;
- un équipement multimédia miniature ;
- émetteur et récepteur infrarouge.

Pour faire vivre cette petite plateforme électronique, une version spéciale de UNIX pouvant être logée dans moins de 1 MB fut écrite. Un système de fichier fut embarqué dans la mémoire flash du Star7. Parmi les fonctionnalités intéressantes citons :

- communication avec un autre Star7 ;
- communication avec un ordinateur ;
- aide en ligne avec un personnage animé : *Duke* ;
- logiciel de télécommande à infrarouge ;
- possibilité de télécharger de nouvelles applications *on the fly* [1] par le réseau sans fil ;

1.1.1 Langage de développement

Le langage choisi pour le projet initial était le `C++` . Mains de nombreuses difficultés furent rencontrées : évolution et suivi du logiciel, carences des

[1] Pas de procédure de redémarrage de la machine pour prendre en compte l'application . . ..

outils deconception, *bugs* difficiles à détecter ...Le portage de l'application devint cependant le problème majeur à résoudre. À cette fin, James Gosling initiateur de la technologie `Java` , réalisa qu'il fallait apporter une solution définitive à tout cela. C'est pourquoi, il commença à travailler sur un langage appelé `Oak` en lui fixant les objectifs suivants :

- intégration des aspects «réseaux»
- sécurité
- fiabilité
- indépendant de la plateforme
- intégration du *multithreading*
- dynamique (pour permettre le chargement d'applications)
- taille de code réduite (pour être embarqué)
- simple et familier (dans la suite de `C` et `C++`)

Selon Gosling, le langage `Oak` est la fusion de plusieurs styles de programmation :

orientée objet : comme `Smalltalk`.

calcul : comme `FORTRAN`.

système : comme le `C`.

distribué[2] : comme rien d'autre.

1.1.2 `Oak` et le *World Wide Web*

Au moment ou le Star7 devenait opérationnel en mettant en avant des concepts novateurs, le *World Wide Web* envahissait la planète. Mais le *Web* ne véhiculait alors que des pages statiques. Toutes les opérations devant être effectuée sur le serveur, les possibilités étaient donc limitées. Pour y pallier, on imagina alors de pouvoir faire exécuter des programmes par le navigateur. Il apparut que les contraintes de sécurité, de portabilité, d'intégration du réseau, d'ouverture, de dynamique ...étaient justement les mêmes qui s'étaient révélées pour la réalisation du logiciel du Star7.

En 1994 un navigateur expérimental[3] qui prenait en compte les programmes `Oak` fut développé. Il devait s'appeler plus tard HotJava^{TM}. Pour cette occasion, le langage `Oak` fut rebaptisé `Java` .

En avril 1995, Hotjava et l'environnement `Java` étaient officiellement annoncés. Le mois suivant Netscape obtenait la licence de la technologie `Java` et l'intégrait à Netscape 2.0. En janvier 1996, Sun distribuait le kit de développement `Java` (JDK).

[3] WebRunner

	Java	PersonalJava	EmbeddedJava	JavaCard
ROM	4...8 MB	$<$ 2 MB	$<$ 512 KB	16 KB
RAM	$>$ 4 MB	1 MB	$<$ 512 KB	512 B
μProcesseur	32 bit $>$ 100 MHz	32 bit $>$ 50 MHz	32 bit $>$ 25 MHz	8 bit 300KIPS

TAB. 1.1 – *Les cibles JAE*

1.2 Aperçu de la technologie `Java`

1.2.1 Les API

Afin de s'adapter à la demande, quatre API [4] ont été réalisés, correspondant à quatre catégories de cibles matérielles associées chacune à un environnement Java (JAE). Cette stratégie permet d'utiliser `Java` pour des applications d'importances variées (allant de l'application graphique la plus complexe au logiciel embarqué de quelques KB). On distingue :

Java AE pour les développements sur station de travail ou ordinateur personnel. Applications : serveurs d'entreprises, environnement graphique pour OS, logiciels, ...

PersonalJava AE pour les développements de matériel mobile. Applications : téléphones mobiles haut de gamme, terminaux internet, téléphone vidéo, ...

EmbeddedJava AE pour les applications embarquées. Applications :Appareillage industriel, instrumentation, imprimantes, téléphones mobiles de milieu de gamme, ...

JavaCard AE pour les applications ou le coût du produit ne permet que quelques kilo-octets de mémoire. Application : horloges, ...

Le tableau 1.1 résume les caractéristiques essentielles des quatre JAEs :

1.2.2 La machine virtuelle

La machine virtuelle `Java` se décompose en cinq parties fondamentales :

1. Un ensemble d'instructions en pseudo-code. Les instructions portent essentiellement sur les valeurs empilées. (Ex : `bipush, iadd, imul, iand, i2l, ...`
2. Un ensemble de registres. Ils mémorisent l'état de la machine et influencent ses opérations. Ils sont remis à jour après chaque exécution d'une instruction de pseudo-code. On y trouve : `pc, optop, frame, vars`.
3. Une pile. La machine virtuelle Java est basée sur une pile. Elle est utilisée pour fournir les opérandes aux instructions pseudo-codes et aux méthodes, mais aussi pour en recevoir le résultat.

[4] Application Programming Interface

4. Un segment de mémoire. C'est la partie de mémoire dans laquelle sont assignés les objets nouvellement créés.
5. Une zone de stockage des méthodes. Elle stocke les pseudo-codes qui implémentent les méthodes.

Inplémenter `Java` sur une cible consiste donc d'une part à réaliser la dernière étape d'un compilateur c'est à dire traduire le pseudo-code Java en instruction pour le processeur visé ; et d'autre part à gérer les zones de données à l'exécution et à réaliser les algorithmes de récupération de mémoire.

1.2.3 Portabilité

La raison de la portabilité totale d'un programme Java est la stabilité du jeu d'instructions du *byte code*. La portabilité d'une application écrite dans un langage comme `C++` dépend du compilateur utilisé pour la machine cible et de l'implémentation des bibliothèques graphiques, réseau, ...(voir figure 1.1).

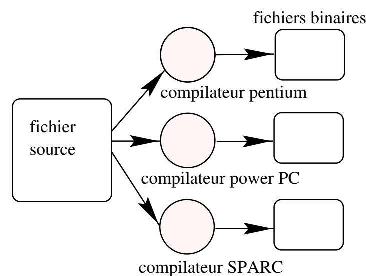

FIG. 1.1 – *Compilation classique*

Pour Java tout se résume dans le slogan déposé par Sun pour la technologie Java : «*Write Once, Run Anywhere*»^{TM}.

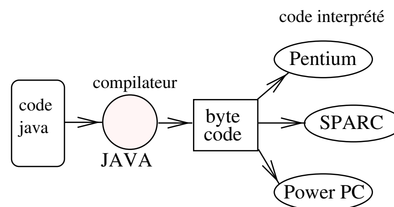

FIG. 1.2 – *Compilation `Java`*

Multithreading

Le langage Java est original dans ce sens ou il supporte directement les *threads* [5]. Les problèmes de multitâche peuvent donc être résolus indépen-

[5]pour *threads of control.*

demment du système d'exploitation. Java ne prend pas en compte les interruptions matérielles pour des raisons de portabilité. En effet, les systèmes d'interruptions matérielles sont très différents d'un système à l'autre.

Taille des mots

Selon les machines, la taille des entiers varie. Elle est liée à la taille des registres de donnée du microprocesseur. D'autre part, les types entiers doivent être qualifiés *signés* ou *non signés*. Avec Java, un type donné a toujours la même taille, quelque soit la machine, de plus, les entiers sont toujours signés de façon à éliminer le plus possible d'erreurs.

1.2.4 Fiabilité

Java apporte un certain nombre de concepts qui sont des solutions au manque de fiabilité des logiciels écrits avec son prédécesseur `C++` .

1. Il n'y a aucun mécanisme de prévu pour l'accès direct aux adresses de la mémoire. Il n'y a pas d'arithmétique possible sur les pointeurs. Les accès obligatoire à la mémoire pour des applications industrielles particulières devront faire appel à des méthodes natives écrites en `C` .
2. Java définit le mot clé `goto`, mais il est réservé et non utilisé.
3. L'inclusion de fichiers [6] n'est pas autorisée. Le `typedef` est supprimé. En effet, les logiciels qui utilisent beaucoup ces possibilités sont difficiles à maintenir.
4. La confusion entre booléen et entier n'est plus possible comme en `C` ou en `C++` . Ainsi, les erreurs fréquentes sur les opérateurs `&` et `&&` sont évitées.
5. Les accès au tableaux sont vérifiés. Il n'est plus possible de lire ou d'écrire en dehors de l'espace déclaré pour un tableau. Un tableau est un objet de Java qui mémorise sa propre taille dans une variable `length`.

   ```
   for (int i=0 ; i<T.length ; i++)
   ```

6. Les exceptions (...les erreurs) ne se produisent pas, elles sont levées (`thrown`) Java lève une exception en réponse à une situation inhabituelle. Le programmeur peut lui aussi lever ses propres exceptions, ou capturer (`catch`) une exception pour traiter les erreurs possibles de manière optimale. Notons que le `try ...catch ...` est un apport de `C++` , mais l'aspect facultatif de son utilisation n'en a pas fait un outil valable de sécurisation des programmes.

[6] avec une directive du style de *#include*

1.2.5 Le Kit de Développement Java JDK^{TM}

C'est un ensemble d'outils et de classes disponibles gratuitement sur le site de Sun (http ://java.sun.com). On y trouve au moins :

javac c'est le «compilateur» `Java` . En réalité, le code généré est le *byte-code* destiné à être interprété par une machine virtuelle Java. La ligne de commande :

```
javac test.java
```

compile le fichier *test.java* et donne en sortie un fichier «exécutable» (en codage *bye-code*).

java lance une machine virtuelle `Java` pour exécuter sans navigateur un programme `Java` compilé avec l'outil précédent.

```
java test
```

exécute la méthode *main* de la classe `test`.

javah créée un fichier *header* utile dans le cas des méthodes natives.

appletviewer pour visualiser l'exécution d'une ou plusieurs *applets* mentionnée dans un fichier HTML. Un navigateur quelconque peut être substitué à cet outil.

javap donne une documentation succinte sur une classe donnée en argument. La commande

```
javap java.lang.String
```

donne une documentation sur la classe `String` du *package* `java.lang`.

Il convient de télécharger l'aide sur les classes prédéfinies de Java : *JavaDoc*.

Chapitre 2

Création d'applets ou d'applications

Les *applets* sont des programmes appelés à partir d'une page HTML. Le *byte-code* de l'applet est chargé du serveur vers la machine cliente. Le navigateur du client interprète alors le *byte-code* , à condition, bien entendu, qu'il intègre une machine virtuelle `Java` .

Les applications sont des programmes presque identiques aux *applets* , mais il n'y a pas besoins de navigateur pour les exécuter. Il suffit de lancer une machine virtuelle `Java` en lui donnant en argument le nom du fichier *.class* qui contient le *byte-code* Les deux démarches sont résumées dans le tableau de la figure 2.1.

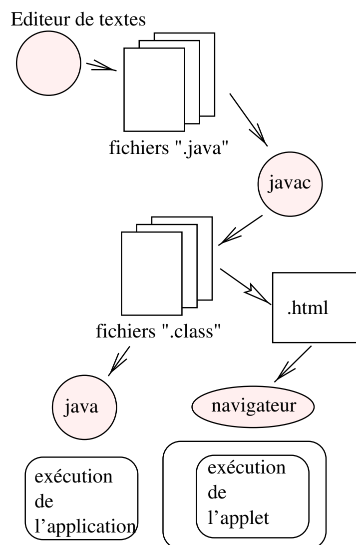

FIG. 2.1 – *Schémas de développement*

2.1 Application Java

L'application suivante permet de présenter, d'une part quelques aspects du langage `Java` ; et d'autre part la démarche de fabrication du programme.

```
/* Fichier: ex1.java */
import java.io.*;

class ex1  {

    /* sous programme */
    static int somme(int x, int y, int z) {
        return x+y+z;
    }

    /* programme principal */
    public static void main(String args[]) {
        System.out.println("Résultat = " + somme(1,2,3));
    }
}
```

Quelques remarques :

- le programme est *encapsulé* dans une **classe**,
- le **nom de la classe** et le **nom du fichier** sont identiques (obligatoire),
- il est conseillé de faire figurer **une seule classe** par fichier,

Une application `Java` est donc un programme autonome. On y trouve une méthode `main`. Un programme en `C` ordinaire pourrait , sans trop de modifications être transformé en application `Java` .

2.1.1 Fabrication du programme

Le fichier source *ex*1*.java* peut être créé avec n'importe quel éditeur de texte (*emacs, xemacs, edit, notepad, kedit, vi, xcoral, nedit* . . .). Une fois le fichier source `Java` créé, il faut le «compiler» pour obtenir le *byte-code* :

```
javac ex1.java
```

Le résultat, s'il n'y a pas d'erreur, est un fichier appelé *ex*1*.class*.

Pour exécuter l'application :

```
java  ex1
```

2.1.2 Les arguments de la ligne de commande

Ils sont contenus dans le paramètre `args` de la fonction `main`. Ce paramètre est un objet de type «tableau d'objets» `String` ($\Leftrightarrow$ tableau de chaîne de caractères).

L'exemple suivant permet d'afficher les arguments reçus sur la ligne de commande :

```
import java.io.*;

class argument {

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

La compilation se fait en exécutant la commande :

```
javac argument.java
```

L'exécution de

```
java argument un deux "trois et quatre"
```

donne la trace d'exécution suivante :

```
% java argument un deux "trois et quatre"
  arg[0]= un
  arg[1]= deux
  arg[2]= trois et quatre
%
```

Remarques

→ `args.length` est le nombre d'arguments reçus.

→ `args[i]` représente le $(i-1)^{\text{ème}}$ argument (ils sont numérotés à partir de 0)

→ Un argument composé de plusieurs mots doit être entouré des doubles-quotes (`"`).

2.2 Applet Java

L'*applet* est destinée à être exécutée par un navigateur. À ce titre, elle aura beaucoup de restrictions par rapport à une application, en particulier en ce qui concerne l'accès aux ressources (programmes, fichiers, ...) de la machine sur laquelle elle s'exécute. Ceci mis à part, il y a très peu de différences entre une *applet* et une application. Au niveau de la conception, une *applet* ne dispose pas de méthode `main` , ce que nous y avons écrit précédemment sera simplement transféré dans la méthode *init*.

Pour exécuter une applet, son *byte-code* contenu dans le fichier *.class* doit être chargé par le navigateur. Un «tag» HTML est prévu à cet effet. Supposons que l'applet s'appelle *ex2.class*, le fichier HTML doit contenir au moins les lignes :

```
<html>
<head>
<title>Applet Java </title>
</head>
<body>
 <applet CODE="ex2.class" WIDTH=200 HEIGHT=50>
 Texte visible pour les navigateurs
 non compatibles...
</applet>
</body>
</html>
```

Le fichier `ex2.class` est fabriqué à partir d'un fichier source `ex2.java`, par le même procédé de compilation que pour une application.

```
// fichier ex2.java
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;
import java.applet.*;

public class ex2 extends Applet {
    // variable de la classe 'ex2'
    Font f = new Font("Times Roman", Font.BOLD, 16);

    public void paint(Graphics g) {
        g.setFont(f);
        g.setColor(Color.red);
        g.drawString("Hello World!",10,40);
    }
}
```

Chapitre 3

Éléments de base du langage

3.1 Les types de données

Les types de bases sont donnés dans le tableau suivant :

<table>
<tr><th>type</th><th>Intervalle de valeurs</th><th>Opérations</th></tr>
<tr><td>float</td><td>NaN, $-(2^{24})^{-149}..(2^{24})^{104}$</td><td>==,!=, <,>, <=, >= +, -, *, /, ++, --</td></tr>
<tr><td>double</td><td>NaN, $-(2^{53})^{-1075}..(2^{53})^{970}$</td><td>voir float</td></tr>
<tr><td>boolean</td><td>true/false</td><td>==,!=,!, &,^, |, &&</td></tr>
<tr><td>byte</td><td>-128..127</td><td>==,!=,<,>, <=, >=, +, -, *, /, %, ++, − ,<<, >>,>>>, &, |, ^ ,? :</td></tr>
<tr><td>short</td><td>-32768..32767</td><td>voir byte</td></tr>
<tr><td>int</td><td>-2147483648 .. 2147483647</td><td>voir byte</td></tr>
<tr><td>long</td><td>-9223372036854775808 .. 9223372036854775807</td><td>voir byte</td></tr>
<tr><td>char</td><td>'\u0000'..'\uffff'</td><td>==,!=</td></tr>
</table>

Les types non signés (⇒ variable toujours positive) n'existent pas.

3.1.1 Application

Voici un exemple de programme utilisant ces types :

```
// Fichier ex3.java
class   ex3 {

    final int CONSTANT=1;

    public static void main(String[] args){
        boolean test=true ;
        byte octet;
```

```
            short petit;
            int ent;
            long ent_l;
            char car;
            float X;
            double X_l;

            System.out.println("B : " +test+" /B : "+!test );
            petit = 10;
            ent = 1000 *(int)petit - 10 + 3/4;
            ent_l = 10000 *(long)ent;
            System.out.println("Entiers: " + petit +" " + ent + " " + ent_l);
            X = (float)1e308;
            X_l = 10.0*(double)X;
            System.out.println("Dép: " + X*10);
            System.out.println("Pi : "+ Math.PI);
            System.out.println("Div. par  0 :   " + X/Math.sin(0.0));
            car = 'a';
            System.out.println("Caractère: "+car) ;
        }
    }
```

Trace d'exécution :

```
B : true /B : false
Entiers: 10 9990 99900000
Dép: Infinity
Pi : 3.141592653589793
Div. par  0 :   Infinity
Caractère: a
```

3.1.2 Constantes

Une constante est déclarée comme une variable avec le mot clé *final* en en-tête. Elles sont généralement écrites en majuscule. De façon générale, tout élément déclaré *final* ne pourra être redéfini ou modifié par la suite.

3.2 Les variables

`Java` possède trois sortes de variables :

Les variables d'instances : déterminent les attributs ou l'état d'un objet donné. Ces variables sont connues par l'ensemble des méthodes de la classe.

les variables de classe : elles s'appliquent à tous les objets instanciés à partir d'une classe donnée. Les variables d'instances et de classes sont déclarées au même niveau. Une variable de classe sera précédée du préfixe *static*.

les variables locales : elles sont déclarées et utilisées dans les définitions des méthodes (compteurs de boucles, ...).

Les variables globales n'existent pas.

3.3 Les opérateurs

Les opérateurs suivants agissent sur tous les types ***réel*** ou ***entier*** , à l'exeption de l'opérateur `%` qui ne prend que des opérandes ***entier*** .

3.3.1 Les opérateurs arithmétiques

Opérateur	Opération
`-`	moins unaire
`+`	plus unaire
`*`	multiplication
`/`	division
`+`	addition
`-`	soustraction
`%`	*modulo* (reste de la division entière)

Tab. 3.1 – Les opérateurs arithmétiques.

Remarques :

† L'opérateur `%` ne s'applique qu'aux entiers. Si le premier opérateur est négatif, le signe du résultat est négatif. Il est positif sinon. (`15%2 = 1, -15%2 = -1, 15%(-2) = 1`).

† Il n'existe pas d'opérateurs d'exponentiation.

† La division (`/`) est une division entière lorsque les deux opérandes sont entiers.
Ainsi, `5/2` donnera `2` pour résultat alors que `5./2.` donnera `2.5`.

3.3.2 Les opérateurs sur les *bits*

Les opérateurs suivants opèrent *bit* à *bit* et s'appliquent à des opérandes de type ***entier*** (ou ***caractère***).

négation *bit* **à** *bit* (`~`) c'est un opérateur unaire. Il inverse un à un tous les bits de son opérande.
Exemple : `~0x5f` est une expression qui vaut `0xa0` (soit `160`). En effet,

```
 0x5f  →  0101 1111
~0x5f  →  1010 0000  →  0xa0
```

Opérateur	**Opération**
`~`	négation *bit* à *bit* (unaire)
`<<`	décalage à gauche
`>>`	décalage à droite
`>>>`	décalage à droite (introduction de 0)
`&`	et *bit* à *bit*
`^`	ou exclusif *bit* à *bit*
`|`	ou inclusif *bit* à *bit*

TAB. 3.2 – Les opérateurs sur les *bits*.

décalage à gauche (`<<`) `n << i` décale l'entier `n` de `i` *bits* vers la gauche. Les *bits* sortants à gauche sont perdus et des `0` sont introduits à droite. Cet opérateur permet d'effectuer des multiplications d'entiers par des puissances de 2. Si la variable `n` est signée, le bit de signe est conservé.

décalage à droite (`>>`) `n >> i` décale l'entier `n` de `i` *bits* vers la droite. Les *bits* sortants à droite sont perdus et des `0` sont introduits à gauche.Cet opérateur permet d'effectuer des divisions d'entiers par des puissances de 2. Si la variable `n` est signée, le bit de signe est conservé et propagé.

et ***bit*** **à** ***bit*** (`&`) un «et» est effectué *bit* à *bit* sur les deux opérandes.

Exemple : `0xb6 & 0x53` vaut `0x12` :

```
0xb6  →  1011 0110
0x53  →  0101 0011
         ---------
         0001 0010  → 0x12
```

ou exclusif ***bit*** **à** ***bit*** (`^`) un «ou» exclusif est effectué *bit* à *bit* sur les deux opérandes.

Exemple : `0xb6 ^ 0x53` vaut `0xe5` :

```
0xb6  →  1011 0110
0x53  →  0101 0011
         ---------
         1110 0101  → 0xe5
```

ou inclusif ***bit*** **à** ***bit*** (`|`) un «ou» inclusif est effectué *bit* à *bit* sur les deux opérandes.

Exemple : `0xb6 | 0x53` vaut `0xf7` :

```
0xb6  →  1011 0110
0x53  →  0101 0011
         ---------
         1111 0111  → 0xf7
```

3.3.3 Les opérateurs d'affectations

Notion de *lvalue*

Une *lvalue* est une expression qui peut apparaître à gauche de l'opérateur d'affectation. Une *lvalue* possède un adresse en mémoire.

Exemple : dans l'expression `i = 7 ;` la *lvalue* est la variable entière `i`.

L'opérateur d'affectation `=`

En `Java` , l'affectation est une opération comme une autre. L'objet à gauche du `=` (la *lvalue*) se voit affecter la valeur retournée par l'expression de droite.

Associativité. On peut écrire :

```
int i, j;
i = j = 0;
```

L'instruction `i = j = 0;` est équivalente à `i = (j = 0);`

L'expression `(j = 0)` vaut 0. Elle peut être regardée de deux points de vue :

1. la variable `j` se voit affecter la valeur 0,
2. c'est une expression de valeur 0

L'affectation combinée

Il s'agit de la traduction en `Java` des instructions de «lecture-modification-écriture» souvent présentes dans les jeux d'instructions des micro-processeurs.

La syntaxe générale est

lvalue `op= expression ;`

avec `op` $\in$ { `*, /, %, +, -, <<, >>, &, ^, |` }

`X op= Y ;` est équivalent à `X = X op Y ;`

Exemple : `k += 2 ;` est équivalent à `k = k + 2 ;`

3.3.4 Les opérateurs d'incrémentation et de décrémentation

Dans un programme, il est fréquent de devoir incrémenter ou décrémenter une variable. `Java` propose deux opérateurs unaires pour effectuer ces deux opérations.

`++` incrémentation de 1

`--` décrémentation de 1

Pré-incrémentation et pré-décrémentation

Les opérateurs d'incrémentation [de décrémentation] sont placés devant la variable. L'incrémentation [la décrémentation] est effectuée puis l'utilisation de la variable est faite.

Exemple : `a = ++b ;` $\Leftrightarrow$ { `b = b+1 ;` `a = b ;`

Post-incrémentation et post-décrémentation

Les opérateurs d'incrémentation [de décrémentation] sont placés après la variable. L'utilisation de la variable est effectuée avant l'incrémentation [la décrémentation]

Exemple : `a = b++ ;` $\Leftrightarrow$ { `a = b ;` `b = b+1 ;`

La valeur de l'expression `b++ ;` est la valeur de `b` avant l'incrémentation.

3.3.5 Les opérateurs relationnels

Le résultat de la comparaison entre deux expressions est de type `boolean`

`false` si le résultat de la comparaison est faux,

`true` si le résultat de la comparaison est vrai.

Opérateur	Opération
==	test si égalité
!=	test si différent
<	test si inférieur
<=	test si inférieur ou égal
>	test si supérieur
>=	test si supérieur ou égal

Tab. 3.3 – Les opérateurs relationnels.

Exemple :

`pair = (n % 2 == 0) ;` `pair` vaut `true` si `n` est pair et `false` sinon.

`if (k != 100)` teste si `k` est différent de 100

3.3.6 Les opérateurs logiques

Exemple : l'expression $10 \leq x < 100$ se traduira en `Java` par :

```
10<=x && x<100
```

dans ce cas, si $x < 10$, l'évaluation s'arrête et l'expression vaut `false`.

ou par

```
10<=x & x<100
```

Opérateur	Opération
`&&`	***et*** logique (évaluation minimum)
`&`	***et*** logique (évaluation complète)
`||`	***ou*** logique (évaluation minimum)
`|`	***ou*** logique (évaluation complète)
`!`	réalise le ***non*** logique (opérateur unaire)

TAB. 3.4 – Les opérateurs booléens.

cette fois, les deux expressions `10<=x` et `x<100` sont systématiquement évaluées.

L'évaluation des expresssions booléennes s'effectue de gauche à droite (sauf pour l'opérateur `!`) et elle est stoppée dès que le résultat de l'expression devient définitif (*short evaluation*).

3.3.7 L'opérateur conditionnel

Il permet d'écrire des expressions dont le résultat est fonction de certaines conditions.

Syntaxe : `(expression) ? expression1 : expression2 ;`
Si `expression` est *Vrai* , le résultat est `expression1`, sinon, le résultat est `expression2`.

Exemple : cet opérateur permet de réaliser simplement la fonction «valeur absolue» :

```
b = (a<0) ? -a : a;
```

Chapitre 4

Les structures de commande

4.1 Les blocs d'instructions

Un bloc d'instructions est une séquence d'instructions enfermée entre deux accolades. Un bloc peut posséder ses propres variables locales. Dans l'exemple suivant, deux blocs apparaissent :

- le bloc de la fonction principale `main`.
- un bloc isolé dans lequel est réalisée la permutation des valeurs des entiers `i` et `j`. Ce bloc interne possède une variable privée `m` qui n'est pas reconnue en dehors.

```
/* fichier: bloc.java */
import java.io.*;

class bloc {
    public static void  main(String[] args) {
        int i=1,j=2;
        System.out.println("i=" + i + "  j=" + j);
        {
            int m;
            m=i;
            i=j;
            j=m;
        }
        System.out.println("i=" + i + "  j=" + j);
    }
}
```

Remarque : `i` et `j` sont dites *variables locales* de la fonction `main`.

4.2 Les répétitions

4.2.1 L'instruction `while`

C'est la traduction en `C` de la structure ***tant que*** …***répéter*** …Elle permet de répéter un bloc d'instructions tant qu'une condition reste vraie.

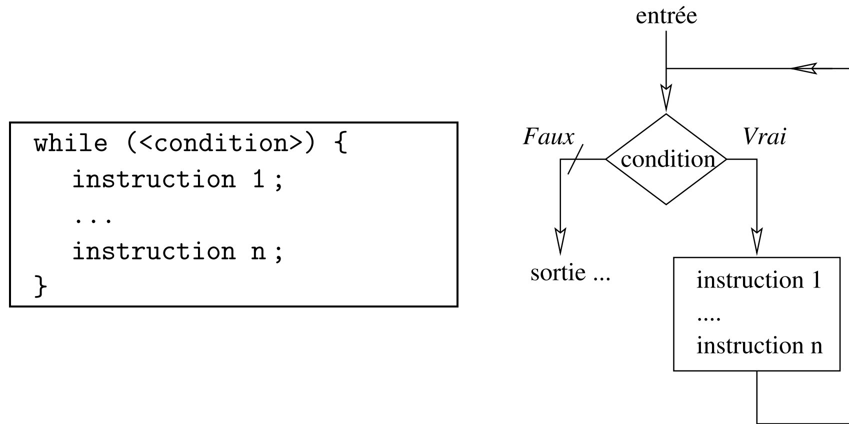

FIG. 4.1 – *Instruction* `while` .

Exemple : On désire concevoir un programme qui calcule la somme des carrés des n premiers entiers.

```
/* fichier: rep1.java */
import java.io.*;

class rep1 {
  public static void  main(String[] args){
    int n, S;
    System.out.print("n = ");
    n = terminal.lireEntier();
    S = 0;
    while (n > 0) {
         S += n*n;
         n--;
    }
    System.out.println("S = " + S);
  }
}
```

Remarques :

– Lorsque l'instruction `while` ne porte que sur une seule instruction, les accolades sont facultatives. Ainsi, le bloc `while` de l'exemple peut être remplacé par :

```
while (n > 0)  S += n*(n--);
```

– Pour l'exemple, on suppose que la classe `terminal` soit présente dans le répertoire courant. Pour les détails sur cette classe, voir le paragraphe 7.2.

4.2.2 L'instruction `do ...while`

C'est l'expression en `Java` de la structure de commande ***répéter*** . . .***tant que*** .

Cette instruction est similaire à la précédente, mais les instructions du bloc à répéter sont exécutées au moins une fois.

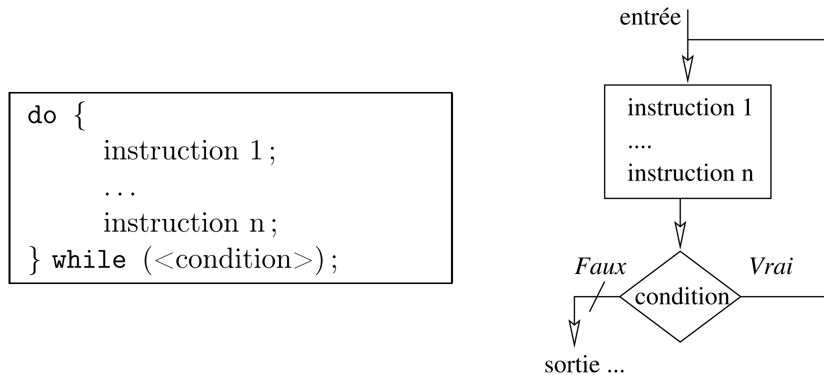

FIG. 4.2 – *Instruction* `do ...while` .

Exemple : On souhaite qu'un utilisateur entre un code à quatre chiffres compris entre 1000 et 9999. Un code erroné est refusé. Il s'agit donc de lire des valeurs jusqu'à ce que l'une d'entre elles répond au critère d'acceptation.

```
/* fichier: bloc.java */
import java.io.*;

class code {
    public static void  main(String[] args){
        int code;
        do {
            System.out.print(" Code = ");
            code = terminal.lireEntier();;
        } while (code<1000 ||  code>9999);
        System.out.println("Code accepté");
    }
}
```

Remarque : Comme pour l'instruction `while` , l'usage des accolades est facultatif lorsque le bloc n'est composé que d'une seule instruction.

4.2.3 L'instruction `for`

L'instruction `for` de `Java` permet non seulement de coder la répétition ***pour*** ...de l'algoritmique, mais en plus, elle est une généralisation des répétions.

Utilisation simple

La répétition ***pour*** est utilisée pour exécuter une séquence d'instructions un nombre de fois préalablement connu :

pour <indice> ***de*** <valeur initiale> ***à*** <valeur finale> ***par pas de*** <pas>
  répéter
    début
      instruction 1 ;
      ...
      instruction n ;
    fin

Le codage `Java` est alors :

```
for (instruction1; <condition>; instruction2 ) {
    instructions3
}
```

Le comportement de cette instruction est décrit par l'organigramme de la figure 4.3.

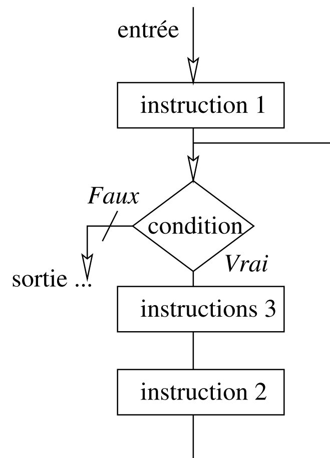

FIG. 4.3 – *Organigramme de l'instruction* `for` .

La réalisation d'une structure de commande ***pour*** peut donc de faire en appliquant les règles suivantes :

→ la valeur initiale de l'indice sera affectée dans `instruction1`,

→ la valeur finale sera traduite dans la `condition`,

→ l'incrément sera appliqué à l'indice lors de l'exécution de l'`instruction2`.

Exemple : le programme suivant affiche les 100 premiers carrés :

```
import java.io.*;

class carre {
    public static void main(String[] args){
        int i;
        for (i=1; i<=100; i++) {
            System.out.print(i*i + " ");
            if (i%10 == 0) System.out.println();
        }
    }
}
```

4.3 Les choix

4.3.1 Le choix simple `if ...else`

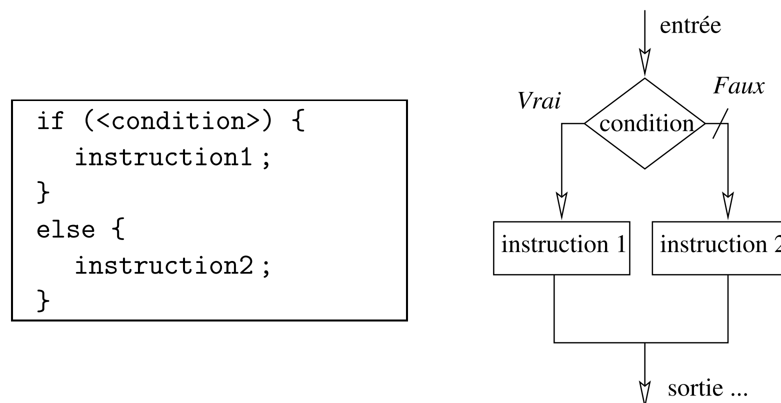

FIG. 4.4 – *Choix simple (alternative).*

Les accolades sont facultatives lorsqu'il n'y a qu'une seule instruction dans le bloc.

Les choix simples répétés

Une instruction quelconque du `if` peut être elle même une instruction `if` . Ceci permet d'imbriquer les structures `if` pour donner le codage d'un choix multiple.

Exemple : le programme suivant calcule une réduction en fonction de l'âge de l'usager :

→ 50% pour les moins de 12 ans,

→ 20% entre 12 et 25 ans,

→ 40% pour les plus de 60 ans.

```
/* fichier: reduction.java */
import java.io.*;

class reduction {
    public static void main(String[] args){
        int age, reduction;
        System.out.print("Quel est votre âge ? ");
        age = terminal.lireEntier();

        if (age < 12) reduction = 50;
        else if (age <25) reduction = 20;
        else if (age <60) reduction = 0;
        else reduction = 40;

        System.out.println("Votre réduction est de " + reduction + "%");
    }
}
```

Étudier le comportement de ce programme en l'absence de `else` ...

4.3.2 Le choix multiple `switch ...case`

Le choix multiple de `Java` est limité à la comparaison d'une variable d'un type simple (***caractère*** , ***entier*** ou ***réel***) à plusieurs constantes. Les différents cas traités ne font pas l'objet de la création d'un bloc d'instructions. C'est l'instruction `break` qui détermine la fin du treitement. Elle renvoie systématiquement à l'accolade fermante de l'instruction `switch` . L'étiquette `default` indique le traitement à réaliser lorsqu'il n'y a pas de correspondance entre la variable et les différentes constantes. Ce traitement est facultatif.

```
switch (variable) {
   case valeur1 :
      instructions1 ;
      break ;
   case valeur2 :
      instructions2 ;
      break ;    ...
   case valeurn :
      instructionsn ;
      break ;
   default :
      instructions0 ;
      break ;
}
```

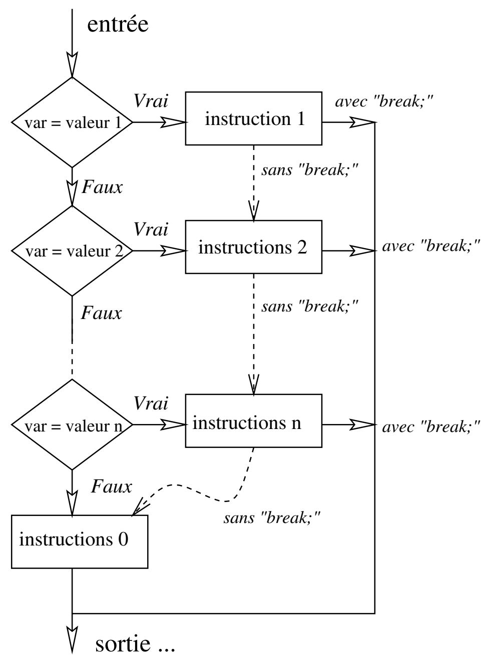

FIG. 4.5 – *Choix multiple.*

Chapitre 5

Classes et objets

5.1 Généralités

Classes

Les **classes** sont des type composés définis par l'utilisateur. Elles peuvent contenir aussi bien des données membres représentant le type que des fonctions membres implémentant des opération sur ce type. Les données membres s'appellent des **attributs**. Les fonctions membres s'appellent des **méthodes**. On peut les représenter graphiquement (figure 5.1).

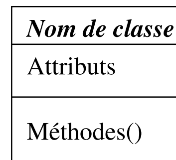

FIG. 5.1 – *Représentation graphique d'un objet.*

Objets

Les **objets** sont créés à partir d'une classe. Pour simplifier, on peut dire que la classe joue le rôle du type, et l'objet le rôle d'une variable. Un objet est créé à partir d'une classe par l'opérateur `new`.

Exemple : Considérons les déclarations suivantes :

```
int i= 9;
String S = new String("Hello World");
```

1. `String` est une classe et `int` est un type.
2. `S` est un identificateur ou référence d'objet ou variable de type `String` et `i` est une variable de type `int`.

3. `i` est initialisé à `9`. Par contre, pour obtenir un nouvel objet de la classe `String`, il faut avoir recours à l'opérateur `new`.

`public`, `private` et `protected`

On peut à volonté cacher ou rendre disponibles des parties de la classe en les plaçant dans des sections *privées* ou *publiques* introduites par les mots clés respectifs `private` ou `public`. Les membres qui sont déclarés `private` ne sont accessibles qu'aux fonctions membres de la classe en question.

Les membres de la classe déclarés `protected` seront accessibles à partir des sous classes (voir l'héritage au paragraphe 5.4).

Accès

Accès aux valeurs des variables d'instance ou de classe La seule façon d'y accéder est l'opérateur *point* :

```
Objet.variable;
```

Méthodes de classe Les méthodes de classes s'appliquent à la classe toute entière et non pas à ses instances. Elles sont utilisées pour des méthodes à vocation générale. Par exemple, la classe `String` contient une méthode appelée `valueOf()` qui reçoit l'un des nombreux types d'argument (`int`, `float`, ...).`valueOf()` et qui retourne ensuite une instance de `String`.

Exemple :

```
String s;
s = String.valueOf(32535);
 /* s devient l'objet "32535" */
```

Appel des méthodes Les méthodes sont appelées par l'opérateur *point*.

```
Objet.Methode(arg1, arg2,arg3);
```

Références aux objets L'utilisation de l'opérateur "=" permet d'obtenir une référence à un objet. Exemple :

```
Point P1, P2;
P1 = new Point(100,120);
P2 = P1;
```

Les variables $P1$ et $P2$ «pointent» le même objet.

Polymorphisme Le polymorphisme n'existe pas en java. Ainsi, il est impossible de surcharger les opérateurs. Attention aux confusions pouvant provenir de l'utilisation de l'opérateur "==". Exemple :

```
String S1, S2, S3;
S1 = "Hello";
S2 = S1;
S3 = new String(S1);
System.out.println("S2==S1"+S1==S2);
System.out.println("S3==S1"+S3==S1);
```

Ce programme affichera `true` puis `false`. L'opérateur `"=="` indique s'il s'agit de deux références à un même objet.

Bibliothèque de classes `Java` En standard, `Java` fourni un certain nombre de familles de classes. Dans tout environnement intégrant `Java` (navigateur, ...), on est sûr de les trouver. Il existe d'autres bibliothèques commercialisées par des éditeurs de logiciel, mais, il faut veiller dans ce cas à les fournir à un utilisateur du programme sur une autre plateforme. On distingue, dans les bibliothèques standard :

- `java.lang` :concerne le langage lui même. Exemples : `Object`, `String`, `Integer`, `Float`, `Character` et `System` en font partie.
- `java.util` : classes utilitaires : `Date, Vector`, ...
- `java.io` : pour les opérations de lecture écriture dans les flux d'entrées/sorties.
- `java.net` : ce sont les classes de gestion de réseau `Socket, URL` ...
- `java.awt` : pour *Abstract Windowing Toolkit* ; elles permettent d'implémenter un interface graphique. `Window, Menu`, `Button`, `Font`, `Checkbox`, ...
- `java.applet` : pour implémenter les *Applets* Java.

On accède aux membres ou aux méthodes d'une classe ou d'un objet avec l'opérateur *point* :

```
Objet.variable;
```

5.2 Conception de classe et création d'objets

5.2.1 Abstraction de type

La classe décrit le domaine de définition d'un ensemble d'objets. Chaque objet appartient à une classe. La classe décrit ce qui est commun à tout un ensemble d'objets. Un objet est contruit à partir d'une classe.

Considérons par exemple le cas des fractions rationnelles. Il serait utile de disposer d'un *type fraction*. Ce type n'existant pas d'origine, il faut donc le créer. Le but étant, à terme d'utiliser des variables de type `fraction` dans un programme, aussi simplement que des variables `int` ou `double`.

L'utilisation du type `fraction` passe par la création d'une *classe* `fraction`. La définition de cette classe repose tout d'abord par l'observation du «monde réel» :

Quelles sont les «données internes» d'une fraction ?

Une fraction est composée d'un *numérateur entier* et d'un *dénominateur entier* non nul.

Traduction informatique : la classe `fraction` dispose de deux **attributs** :

```
int numerateur;
int denominateur;
```

Dans quelles occasions sont-elles «créées» ?

Considérons les fragments d'expressions mathématiques illustrant l'utilisation des fractions :

$$\begin{cases} a = \frac{2}{3} \\ b = a \\ \text{Soit la fraction } c \ldots \end{cases}$$

Il est possible de créer une fraction a à partir de deux entiers, une fraction b à partir d'une fraction déjà existante. On a parfois besoins d'utiliser une fraction sans pour autant lui affecter une valeur particulière. C'est le cas de la fraction c.

Traduction informatique : la classe `fraction` disposera de trois **constructeurs** (au moins). Ils permettent de **construire** (on dit aussi **instancier**) des objets de la classe `fraction`. Ils portent le nom de cette classe. Lorsqu'il y en a plusieurs, ils se distinguent par le(s) type(s) de paramètre(s) reçus.

```
fraction(int n, int d) { /* avec 2 entiers */
     numerateur = n;
     denominateur = d;
}

fraction(fraction F) { /* par duplication */
     numerateur = F.numerateur;
     denominateur = F.denominateur;
}

fraction() {} /* constructeur "vide" */
```

Instanciation d'objets `fraction` ces trois constructeurs permettent de construire des objets de type `fraction` dans des programmes extérieurs à cette classe :

```
fraction a = new fraction(2,3); /* 1er constructeur */
fraction b = new fraction(a);   /* 2ème constructeur */
fraction c = new fraction();    /* 3ème constructeur */
```

Quelles sont les opérations possibles sur une fraction donnée ?

Pour notre exemple, nous en retiendrons trois :
- l'affichage, ou plus simplement, la conversion en chaîne de caractères.
- la simplification,
- l'ajout (à un objet donné, on ajoute une fraction)

Traduction informatique : la classe `fraction` disposera de deux **méthodes** : `toString` et `simplifie`. Pour écrire la fonction de simplification il s'est avéré utile d'écrire une fonction de calcul du *pgcd* de deux entiers. Cette fonction est ici qualifiée par le mot clé `private`, ce qui signifie qu'elle reste interne à la classe. Elle ne fait pas partie de l'interface.

Un codage de ces méthodes est :

```
public String toString() {
    return "+numerateur + "/" + denominateur;
}

private  static int pgcd(int a, int b) {
    if (b==0) return a;
    return pgcd(b, a%b);
}

void simplifie() {
    int p;
    if (denominateur>numerateur)
         p = pgcd(denominateur, numerateur);
    else
         p = pgcd(numerateur, denominateur);
    denominateur /= p;
    numerateur /= p;
}

void ajoute(fraction F) {
    numerateur = numerateur*F.denominateur +
                                 denominateur*F.numerateur;
    denominateur = denominateur*F.denominateur;
    simplifie();
}
```

Quelles sont les fonctions générales aux fractions ? Ce sont les opérateurs classiques appliqués aux fractions : addition, multiplication .... Ces opérateurs concerne l'ensemble de la classe.

Pour l'exemple, intéressons nous à la somme de deux fractions. La somme de deux fractions conduit à la création d'une nouvelle fraction :

$$c = a + b$$

Traduction informatique : la classe `fraction` possède une fonction *statique* qui créée un nouvel objet à partir de deux objets reçus en argument.

```
static fraction somme(fraction f1, fraction f2) {
    int n,d;
    n = f1.numerateur*f2.denominateur +  f1.denominateur*f2.numerateur;
    d = f1.denominateur*f2.denominateur;
```

```
        fraction R = new fraction(n,d); // nouvelle fraction
        R.simplifie();
        return R;
    }
```

Avec les définitions précédentes (résumées graphiquement par la figure 5.2), le code complet de la classe `fraction` est le suivant :

```
import java.io.*;

class fraction {
    int numerateur, denominateur;

    fraction(int n, int d) { /* avec 2 entiers */
        numerateur = n;
        denominateur = d;
    }

    fraction(fraction F) { /* par duplication */
        numerateur = F.numerateur;
        denominateur = F.denominateur;
    }

    fraction() {} /* constructeur "vide" */

    public String toString() {
        return "" + numerateur + "/" + denominateur;
    }

    private  static int pgcd(int a, int b) {
        if (b==0) return a;
        return pgcd(b, a%b);
    }

    void simplifie() {
        int p;
        if (denominateur>numerateur)
            p = pgcd(denominateur, numerateur);
        else
            p = pgcd(numerateur, denominateur);
        denominateur /= p;
        numerateur /= p;
    }


    void ajoute(fraction F) {
        numerateur = numerateur*F.denominateur +
            denominateur*F.numerateur;
        denominateur = denominateur*F.denominateur;
        simplifie();
    }

    static fraction somme(fraction f1, fraction f2) {
        int n,d;
        n = f1.numerateur*f2.denominateur +  f1.denominateur*f2.numerateur;
        d = f1.denominateur*f2.denominateur;
```

```
            fraction R = new fraction(n,d);
            R.simplifie();
            return R;
        }
    }
```

fraction
+numerateur +denominateur
+fraction() +fraction(int, int) +fraction(fraction) +lireDenom() +lireNum() −pgcd(int,int) +simplifie() +ajoute(fraction) +somme(fraction, fraction) +toString()

FIG. 5.2 – *Représentation graphique de la classe* `fraction`. *Signe* `+` ⇔ *public ; signe* `-` ⇔ *privé*

5.3 Exemple plus approfondi

5.3.1 Définition d'une classe `vecteur3d`

Le langage `Java` n'implémente pas directement le type `vecteur3d`. On décide de l'implémenter avec les données et fonctions suivantes :

donnée statique :
- *N : entier*

données :
- *X, Y, Y : réels*

constructeurs :
- ***vecteur3d****()*
- ***vecteur3d****(a,b,c :réels)*
- ***vecteur3d****(vecteur3d V)*

fonctions :
- ***somme***
- ***produitScalaire***
- ***norme***
- ***affiche***

La variable N comptabilisera le nombre d'objets créés.

5.3.2 Traduction en `Java` et test de la classe `vecteur3d`

Le type `vecteur3d` est implémenté dans une classe écrite dans le fichier *vecteur3d.java*. Il est recommendé d'écrire une seule classe par fichier et de

faire en sorte que le fichier et la classe portent le même nom. On obtient ainsi un «composant logiciel» réutilisable et facile à identifier.

```
 1: import java.io.*;
 2: import java.lang.*;
 3:
 4: class vecteur3d {
 5:    public static int N; /* compte les vecteurs créés */
 6:    public double X,Y,Z;
 7:
 8:    /* constructeurs */
 9:    public vecteur3d(){
10:         X=Y=Z=0.;
11:         N++;
12:    }
13:
14:     public vecteur3d(double a, double b, double c){
15:         X=a; Y=b; Z=c;
16:         N++;
17:     }
18:
19:     public vecteur3d(vecteur3d V) {
20:         X = V.X;  Y = V.Y;  Z = V.Z;
21:         N++;
22:     }
23:
24:     /* somme: création d'un 'vecteur3d' comme somme de v et
25:        du 'vecteur3d' courant (this)
26:     */
27:     public vecteur3d somme(vecteur3d v){
28:         vecteur3d w;
29:         w = new vecteur3d(X + v.X, Y + v.Y, Z + v.Z);
30:         return w;
31:     }
32:
33:     /* produitScalaire:
34:        renvoie le produit scalaire de v1 et v2
35:        méthode statique
36:     */
37:     public static double produitScalaire(vecteur3d v1, vecteur3d v2){
38:         return v1.X*v2.X + v1.Y*v2.Y + v1.Z*v2.Z;
39:     }
40:
41:     /* norme */
42:     public double norme(){ return Math.sqrt(X*X+Y*Y+Z*Z); }
43:
44:     /* affiche */
45:     public void affiche(){ System.out.print("("+X+", "+Y+", "+Z+")"); }
46:
47:     static {
48:         N = 0;
49:     }
```

```
50: }
```

Programme de test Il est lui aussi écrit dans un fichier à part (*vecteurT.java*).

```
 1: import java.io.*;
 2:
 3: class vecteurT {
 4:    public static void main(String[] args) {
 5:        vecteur3d v1 = new vecteur3d(1., 2., 3.);
 6:        vecteur3d v2 = new vecteur3d(v1);
 7:        vecteur3d v3 = new vecteur3d();
 8:        vecteur3d v4;
 9:
10:        System.out.print("V1"); v1.affiche();
11:        System.out.print("\nV2"); v2.affiche();
12:        v4 = v1.somme(v2);
13:        System.out.print("\nV4"); v4.affiche();
14:        System.out.print("\n||v1|| = " + v1.norme());
15:        double P = vecteur3d.produitScalaire(v1,v2);
16:        System.out.println("\nV1*V2 = " + P);
17:        System.out.print("Ce programme a créé ");
18:        System.out.println( vecteur3d.N + " objets 'vecteur3d'");
19:    }
20: }
```

Exécution

```
% java vecteurT
V1(1.0, 2.0, 3.0)
V2(1.0, 2.0, 3.0)
V4(2.0, 4.0, 6.0)
||v1|| = 3.7416573867739413
V1*V2 = 14.0
Ce programme a créé 4 objets 'vecteur3d'
```

5.3.3 Étude ligne par ligne du programme

Constructeurs Ici, il y en a trois :

1. Un constructeur «vide» permettant de créer un objet sans aucune initialisation de ses données internes *(lignes 9-12 du fichier vecteur3d.java)*, ce constructeur est indispensable lorsqu'on crée un tableau d'objets.
2. Un constructeur paramétré par des variables de même types respectifs que les données internes à la classe *(lignes 14-17)*.
3. Un constructeur qui construit un objet par duplication d'un autre objet du même type *(lignes 19-22)*.

Création d'objets (ou instanciation d'objets) Dans notre exemple, les objets sont créés dans la fonction principale (`main`) de la classe `vecteurT`. L'instanciation d'un objet d'une classe donnée se fait avec l'opérateur `new`. Par exemple, l'instruction :

```
vecteur3d v1 = new vecteur3d(1., 2., 3.);
```

permet d'instancier un nouvel objet de classe `vecteur3d`, et d'affecter celui-ci à l'identificateur `v1`.

Par contre, la ligne

```
vecteur3d v4;
```

ne donne pas d'objet. C'est une simple déclaration. `v4` est un identificateur non initialisé. Ainsi, une instruction comme `v4.affiche()` ; provoquerait une erreur.

L'initialisation de la variable `v4` se fait à la ligne 12 de *vecteurT.java* :

```
v4 = v1.somme(v2);
```

On peut vérifier que la fonction `somme` appelle effectivement un `new` dans son code. Un constructeur est donc exécuté.

Variables d'instance Dans la classe `vecteur3d`, ce sont les réels `X`, `Y` et `Z`. On les appelle ainsi car chaque objet créé aura son propre triplet `(X,Y,Z)`. De la mémoire est allouée pour ces variables à chaque exécution d'un constructeur de `vecteur3d`.

Variable de classe Une variable de classe est partagée par tous les objets de cette classe. Elle existe même lorsqu'aucun objet n'a été créé.

Pour déclarer une variable de classe, il suffit de faire précéder sa déclaration du mot clé `static`.

La classe `vecteur3d` n'en possède qu'une : l'entier `N` *(ligne 5 du fichier vecteur3d.java)* Elle est utilisée dans notre exemple pour comptabiliser le nombre d'objets `vecteur3d` créés.

Code statique Il s'agit des lignes

```
47:     static {
48:         N = 0;
49:     }
```

Ces lignes sont exécutées une seule fois à la première référence à la classe `vecteur3d`. C'est un outil commode utilisé pour initialiser les variables de classe, pour charger une librairie dynamique ...

Méthodes statiques Une méthode statique est déclarée avec le mot clé `static`. Elle ne peut pas accéder aux variables non statiques ni appeler une méthode non statique. C'est en fait une fonction à usage général qui ne nécessite pas qu'un objet de la classe ait été instancié pour qu'on puisse l'utiliser. On peut considérer une méthode statique comme une «fonction de bibliothèque» se rattachant à la classe.

Dans l'exemple, la méthode `produitScalaire` *(lignes 37-39)* est statique. Elle ne touche pas aux données `(X,Y,Z)` propres à chaques instances. Elle ne travaille qu'avec les paramètres reçus.

Un exemple d'appel à cette méthode se trouve à la ligne 15 du programme de test :

```
double P = vecteur3d.produitScalaire(v1,v2);
```

L'appel à la méthode n'est pas préfixé par un objet précis, mais par le nom de la classe.

Méthodes non statiques Elles peuvent opérer sur les variables d'instances (ici `(X,Y,Z)`). Elles doivent être préfixées par un objet initialisé de la classe.

Exemple :

```
vecteur3d v1 = new vecteur3d(1., 2., 3.);
  .
  .
double M = v1.norme();
```

Remarque : On peut toujours passer d'une méthode «normale» à une méthode statique, ainsi, on peut écrire une deuxième version de la norme d'un vecteur de la manière suivante :

```
/* norme (version statique) */
public static double norme2(vecteur3d V){
    return Math.sqrt(V.X*V.X+V.Y*V.Y+V.Z*V.Z);
}
```

Dans ce cas, l'appel se fera de la manière suivante :

```
vecteur3d v1 = new vecteur3d(1., 2., 3.);
  .
  .
double M = vecteur3d.norme2(v1);
```

5.4 Héritage

Une classe peut *hétiter* des données ou des méthodes d'une autre classe grâce à la dérivation de classe. Lorsqu'une classe `A` *hérite* d'une classe `B`, elle accède automatiquement à toutes les données et toutes les méthodes `public` ou `protected` de `A` sans avoir à les recopier.

extends utilisé dans la déclaration d'une classe pour indiquer de quelle classe supérieure elle hérite.

this est une référence à la classe courante (utilisée pour les passages d'argument)

super est une référence à la classe supérieure.

Exemple : Retour sur la classe `fraction`. On souhaite donner un nom aux fractions. Ceci permettra d'avoir un affichage plus facile. Cette nouvelle classe qui **étend** les attributs et méthodes de `fraction` s'appellera `fractionN`[1] (voir figure 5.3).

Il faut donc

- ajouter une variable de type `String`,
- créer au moins un constructeur,
- redéfinir la méthode `affiche`.

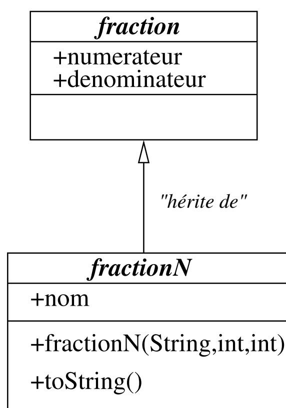

FIG. 5.3 – *`fractionN` hérite de `fraction`.*

Le code de la classe `fractionN` est :

```
1: import java.io.*;
2:
3: class fractionN extends fraction {
4:
5:    String nom;
6:
7:    fractionN() { nom=null; }
```

[1] pour *fraction nommée*

```
 8:
 9:    fractionN(String S, int n, int d) {
10:        super(n,d);
11:        nom = S;
12:    }
13:
14:    void affiche() {
15:        System.out.print(nom + " = ");
16:        super.affiche();
17:    }
18: }
```

Programme de test

```
 1: import java.io.*;
 2:
 3:
 4: class fractionT {
 5:
 6:     public static void main(String[] args) {
 7:
 8:         /* une fraction A ... */
 9:         fraction A = new fraction(2, 4);
10:         System.out.print("A = ");
11:         A.affiche();
12:
13:         /* une fraction B ... */
14:         fraction B = new fraction(2, 3);
15:         System.out.print("B = ");
16:         B.affiche();
17:
18:         A.ajoute(B);
19:         System.out.print("A = ");
20:         A.affiche();
21:
22:         fraction C = fraction.somme(A,B);
23:         System.out.print("C = ");
24:         C.affiche();
25:
26:         fractionN E = new fractionN("E", 5, 7);
27:         E.affiche();
28:     }
29: }
```

Exécution

```
% java fractionT
A = 2/4
B = 2/3
A = 7/6
C = 11/6
E = 5/7
```

Exemple :

5.5 Les tableaux

5.5.1 Déclaration d'une variable de type tableau

On peut simplement déclarer une variable de type tableau, sans pour autant créer l'objet correspondant.

Exemple :

```
point3D Tp[];
String S[];
```

Une autre manière de déclarer un tableau est de placer les crochets après le type et non pas après la variable.

```
point3D[] Tp;
String[] S;
```

5.5.2 Création d'un objet tableau

Exemple :

```
Tp = new point3D[200];
S = new String[12];
point2D Tp2 = new point2D[512];
String[] S2 = {"Java", "C++", "Fortran", "Ada", "Prolog", "Scheme"};
```

Dans la troisième ligne, on réalise la déclaration et la création.
Dans la quatrième ligne, le tableau est déclaré, créé et initialisé.

Remarques : L'accès aux membres d'un tableau se fait comme en C. La numérotation des indices commence également à 0.
Dans `Java` un tableau ne contient pas d'objet, mais des références à ces objets. Lors de la recopie d'un tableau, il n'y a donc pas de recopie d'objet, mais simplement de leurs références.

Exemple :

```
...
int[]  T1 = {1, 3, 5, 7, 9};
float[] T2 = new float[T1.length];

for (int i=0; i<T1.length; i++) {
```

```
        T2[i] = (float)T1[i];
    }
```

Chapitre 6

Définition et gestion des exceptions Java

6.1 Introduction

Quelque soit le langage utilisé, l'usage est de traiter le maximum de cas possibles d'erreurs pouvant occasionner l'arrêt du programme[1] Les origines des erreurs sont multiples :

- toutes les situations dans lesquelles le code peut se retrouver n'ont pas été prévues ;
- entrées erronées de l'utilisateur ;
- fichiers contenant des «mauvaises» données ;
- connexions au réseau qui n'aboutissent pas ;
- des problèmes de matériel ;
- ...

En Java, toutes ces différents événements s'appellent des *exceptions*. Le traitement des exceptions permet dans certains cas de les contourner.

Traitement «traditionnel» des exceptions L'exemple suivant illustre la méthode habituellement mise en œuvre pour traiter les erreurs.

```
int status = FonctionDelicate();

switch(status) {
    case ERREUR_MIN : ...
        break;
    case ERREUR_MAX :
        break;
    // beaucoup d'autres cas ...
    default:
        break;
}
```

Le traitement des erreurs fait de la sorte est bien entendu tout à fait acceptable. Mais la difficulté reste de prévoir tous les cas possibles pouvant

[1] Et dans certains cas l'arrêt du système. . .

entraîner un «plantage» du programme. Un autre problème sous jacent est l'harmonisation des traitements d'un programmeur à l'autre[2]
Pour toutes ces raisons, le dispositif d'exception de Java est là pour gérer, créer et s'attendre à des erreurs, ou faire face à des situations exceptionnelles. Grâce à une combinaison de fonctions spéciales du langage, de vérifications de cohérence lors de la compilation, et d'un jeu de classes extensibles d'exceptions, un programme Java parvient à maîtriser beaucoup mieux les erreurs et autres situations inhabituelles.

6.2 Les exceptions `Java`

Une exception peut être «levée» par le système (`throw`) ou «capturée» par le programme (`catch`). Dans les deux cas, elle est traitée.
Une exception est gérée par Java comme un objet qui hérite de la classe `Throwable` («jetable» ou «levable»). Lorsqu'une exception est levée, une instance de la classe `Throwable` est créée. La figure 6.1 donne une vue partielle de la hiérarchie des classes pour les exceptions. Les instances d' `Error` sont

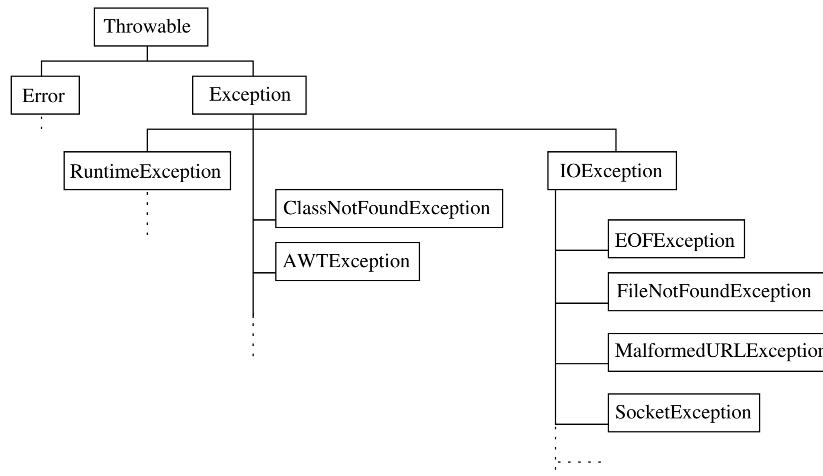

FIG. 6.1 – *Hiérarchie des classes pour les exceptions*

les erreurs internes de l'environnement d'exécution[3]. Ces erreurs sont rares, habituellement fatales et on ne peut rien y faire au niveau de la programmation.
La classe `Exception` est plus intéressante. Les sous-classes d'`Exception` sont réparties en deux catégories :

1. les exceptions d'exécution (*runtime exceptions*), sous-classes de la classe `RuntimeException`, telles que'`ArrayIndexOutofBounds` ou `NullPointerException`.
2. d'autres exceptions telles que `EOFException`.

[2]Maintenabilité du logiciel
[3]C'est à dire de la machine virtuelle Java

Les exceptions du premier groupe correspondent à la prise en compte du manque de robustesse du code. Les deux exceptions citées en ewemple ne doivent jamais se produirent dans un code bien écrit.
Les exceptions du second groupe correspondent à des événements anormaux, échappant au contrôle du programme.

Définition des exceptions : elles font, pour la plupart, partie du *package* `java.lang`. Mais un grand nombre d'autres *packages* définissent d'autres exceptions . Ainsi, `java.io` défini la classe `IOException`, `java.net` défini des sous-classes de `IOException` comme par exemple `MalformedURLException`

6.3 Gestion des exceptions

6.3.1 Vérification de cohérence

Le compilateur `Java` impose dans de nombreux cas la gestion des exceptions . Si une exception n'est pas traitée, le compilateur ne génère pas le *byte-code*. On dit qu'il y a «vérification de cohérence». Le message suivant est caractéristique d'une vérification qui conclut à la non cohérence :

```
test.java:54: Exception java.lang.InterruptedException
must be caught or it must be declared in the throws clause
of this method.
```

6.3.2 Protection du code et capture des erreurs

La capture d'une exception se fait en deux temps :

1. il faut protéger le bloc du code qui pourrait lever une exception à l'intérieur d'un bloc `try` ;
2. il faut traiter l'exception à l'intérieur d'un bloc `catch`.

```
try {
   // appel de la méthode
}
catch (Exception e) {
   // traitement des erreurs
}
```

Il va de soit que si l'exécution du code «sous surveillance» se passe correctement, le bloc `catch` n'est pas exécuté.
L'exemple suivant est un extrait du code qui avait été admis dans le chapitre précédent. On remarque que si le fichier *p3d.txt* n'est pas trouvé, l'objet *x3D* de type *objet3D* n'est pas créé.

```
class fenG extends Frame {
  FileReader f = null;
  objet3D x3D;
```

```
  fenG(String titre) {
    try {
      f = new FileReader(new File("p3d.txt"));
      x3D = new objet3D(f);
    }
    catch (Exception e){
      System.out.println("Erreur d'ouverture du fichier "+ e.getMessage());
    }
  }
}
```

Prise en compte de plusieurs exceptions : lorsqu'un code est susceptible de provoquer différentes exceptions non reliées par l'héritage, on utilise plusieurs clauses *catch* pour une clause *try*. Exemple :

```
try {
  // code à surveiller
}
catch (RuntimeException r) { .... }
catch (IOException o) { .... }
catch (Exception e) { .... }
catch (Throwable t) { .... } // traitement de tout ce qui n'a
                             // pas été capturé
```

Il faut évidemment placer les sous-classes avant leur classe parente. Un seul bloc sera exécuté.

La clause *finally* Cette clause est réservée pour les actions qui doivent être exécutées qu'il y ait exception ou pas.

Exemple :

```
File f;
if (f.open("data.txt")) {
   try {
      // ....
   }
   catch {
      // ....
   }
   finally {
      f.close();
   }
}
```

6.4 Déclaration de méthodes pouvant lever des exceptions

La clause `throws` : pour déclarer une méthode qui permet de lever des exceptions , il faut ajouter le mot clé `throws` dans la signature de la méthode. L'exception levée devra être traitée dans une clause `catch`. Dans l'exemple

suivant, le traitement des exceptions levées se fait dans la méthode appelante (citée dans le paragraphe 6.3.2 :

```
objet3D(FileReader fi)  throws IOException {
  N = 0;
  listeP = new point3D[200];

  StreamTokenizer st = null;
  st = new StreamTokenizer(fi);
  st.whitespaceChars(32,44); // inclut les ',' et les <SP>
  st.eolIsSignificant(true);
  st.commentChar('#'); // ligne  à passer
  // Le premier entier est le nombre de points :
    st.nextToken();
    N =(int) st.nval;
    for (int i=0; i<N; i++) {
      double x,y,z;
      String Str;
      st.nextToken(); // fin de ligne
      st.nextToken(); Str = st.sval;
      st.nextToken(); x = st.nval;
      st.nextToken(); y = st.nval;
      st.nextToken(); z = st.nval;
      listeP[i] = new point3D(Str, x, y, z);
    }
}
```

Ceci signifie que la séquence `try ... catch ...` ne figure pas dans la définition de cette méthode, mais par contre, elle sera appelée elle même par une méthode qui réalise le traitement.

6.5 Création d'exceptions

L'exemple suivant illustre la manière de créer des exceptions spécifiques.

Nouvelle classe dérivée de `Exception` La nouvelle exception s'appelle `MonException`. Elle doit être définie dans un fichier particulier de même nom avec l'extension `.java` :

```
// fichier MonException.java
import java.lang.*;
import java.io.*;

public class MonException extends RuntimeException {

  public  MonException(){
    System.out.println("Création de l'objet exception <VIDE>!");
  }
  public  MonException(String s){
    super(s);
    System.out.println("Création de l'objet exception !"+s);
  }
```

```
}
```

Le levage de cette exception doit être effectué de manière explicite dans le code qui l'utilise. Dans l'exemple qui suit, c'est la méthode `Method` de la classe `objet3D` qui lève `MonException` :

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


/* --------------  objet3D ------------------------- */
class objet3D {
  objet3D(){}
  public void Method(int i)  {
    if (i == 13) throw new MonException("ex13");
    else if (i == 100) throw new MonException();
    else System.out.println(i+ "... çà va !");
  }
}


/* -----------  fenêtre graphique ------------------------- */
class fenG extends Frame {
  objet3D y3D;

  fenG(String titre) {
    y3D = new objet3D();
    try {
      y3D.Method(13);
    }
    catch (MonException m) {
      System.out.println("Mon erreur ! " + m +" "+  m.getMessage());
    }
    try {
      y3D.Method(10);
    }
    catch (MonException m) {
      System.out.println("Mon erreur ! " + m +" " + m.getMessage());
    }
    try {
      y3D.Method(100);
    }
    catch (MonException m) {
      System.out.println("Mon erreur ! " + m +" "+ m.getMessage());
    }
    // y3D.Method(13);
    System.out.println("Fin normale du programme");
  }
}
```

```
/* ---------- Programme principal ------------------------- */
class myexcp{
  static fenG xG;
  public static void main(String args[]){
    xG = new fenG("Exception");
  }
}
```

Trace d'exécution avec l'instruction non protégée :

```
>java myexcp
Création de l'objet exception !ex13
Mon erreur ! MonException: ex13 ex13
10... çà va !
Création de l'objet exception <VIDE>!
Mon erreur ! MonException null
Création de l'objet exception !ex13
MonException: ex13
        at objet3D.Method(myexcp.java:10)
        at fenG.<init>(myexcp.java:41)
        at myexcp.main(myexcp.java:51)
```

C'est le processeur virtuel Java qui récupère l'erreur : le programme est arrêté.

Exécution sans la ligne non protégée :

```
>java myexcp
Création de l'objet exception !ex13
Mon erreur ! MonException: ex13 ex13
10... çà va !
Création de l'objet exception <VIDE>!
Mon erreur ! MonException null
Fin normale du programme
```

Cette fois le programme continue normalement.

Chapitre 7

Les opérations d'entrées sorties sur les fichiers

Les opérations sur les fichiers utilisent la notion de flux (*stream*). Cette notion provient des *pipes* de UNIX.

Un flux est un chemin de communication entre la source d'une information et sa destination. Ce paragraphe limite l'étude aux opérations sur les fichiers.

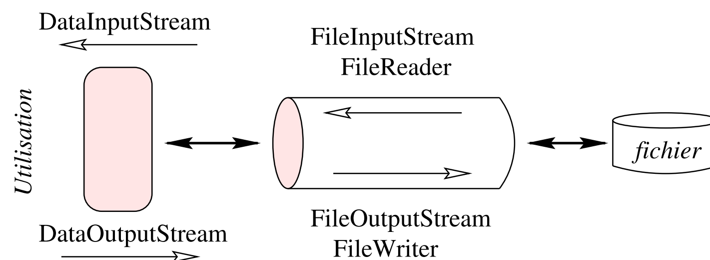

Fig. 7.1 – *Les objets associés*

7.1 Lecture de fichier Classes `FileInputStream` et `DataInputStream`

La classe `FileInputStream` permet de lier un flux d'entrée à un fichier. (`FileInputStream` hérite de `InputStream`). La création d'un objet `FileInputStream` se fait simplement par une ligne du type :

```
FileInputStream fis = new FileInputStream("p3d.txt"));
```

Après exécution de cette ligne, le fichier qui a pour nom `p3d.txt` est lié au flux verb+fis+. Il s'agit d'un flux d'octets. Utilisé directement, on ne peut pas faire grand chose avec la variable verb+fis+, mais ce qui compte c'est que

le flux est créé et qu'il va pouvoir être associé à un interface d'interprétation des données octets reçus. C'est ce qui est réalisé dans l'exemple qui suit : le flux créé sert directement d'argument à un interface de lecture de type `DataInputStream` .

Pour associer le flux à un interface évolué d'interprétation il faut écrire :

```
DataInputStream dis = new DataInputStream(new FileInputStream("p3d.txt"));
```

La classe `DataInputStream` dispose des méthodes (utiliser la commande `javap java.io.DataInputStream` pour les avoir toutes) :

- `boolean readBoolean()`
- `byte readByte()`
- `int readUnsignedByte()`
- `short readShort()`
- `int readUnsignedShort()`
- `char readChar()`
- `int readInt()`
- `long readLong()`
- `float readFloat()`
- `double readDouble()`
- `String readString()`
- ...

Attention.. Le fichier doit contenir les valeurs au format lu. Par exemple, si on lit un entier dans le fichier avec une instruction du type `int i = dis.readInt()`, c'est vraiment un entier qui est recherché dans le flux *et non pas un entier codé ASCII.*

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

/* --------objet3D ------*/
class objet3D {

    objet3D()  throws IOException {
        try {
            DataInputStream dis =
                new DataInputStream(new FileInputStream("test.txt"));
            String line;
            int x;
            while ((line = dis.readLine()) != null) {
                System.out.println(line);
            }
            dis.close();
        } catch (EOFException e)  { }
    }
}
```

```
/* ----- Programme principal ----- */
class principal {

    public static void main(String args[]){
        try {
            x3D = new objet3D();
        }
        catch (Exception e){
            System.out.println("Echec à la création de object3D");
        }
    }
}
```

7.2 Cas particulier de l'entrée standard (clavier) Classe `BufferedReader`

`Java` ne dispose pas de fonctions prédéfinies performantes pour traiter le cas de la lecture d'entrées formatées à partir du clavier. On peut y palier en créant une classe `terminal` qui exploite un objet `BufferedReader` dédié au clavier. Cet objet doit être créé en prenant pour argument `System.in`. L'exemple suivant met en œuvre le type `terminal` :

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


class terminal {

    static private BufferedReader br;

    terminal() {}

    /* Entrées ------------------------------------------- */
    public static int lireEntier() {
        int Retour;
        try {
            Retour = (new Integer(br.readLine())).intValue();
        }
        catch (Exception e) {
            System.out.println("Entrée d'entier incorrecte !");
            Retour = 0;
        }
        return Retour;
    }

    public static float lireReel() {
        float Retour;
```

```
    try {
        Retour = (new Float(br.readLine())).floatValue();
    }
    catch (Exception e) {
        System.out.println("Entrée de réel incorrecte !");
        Retour = 0;
    }
    return Retour;
}



public static char lireCar() {
    char Retour;
    try { // caractère <=> 1er caractère de la chaîne lue...
        Retour = (br.readLine()).charAt(0);
    }
    catch (Exception e) {
        System.out.println("Entrée de caractère incorrecte !");
        Retour = 0;
    }
    return Retour;
}

public static String lireChaine() {
    String Retour;
    try {
        Retour = br.readLine();
    }
    catch (Exception e) {
        System.out.println("Entrée de chaîne incorrecte !");
        Retour = null;
    }
    return Retour;
}

/* Sorties -------------------------------------------- */
public static void efface () {
    System.out.println("\u001b[2J");
}

// sortie formatée d'un entier 'i' avec 'n' caractères
public static void ecrireEntierF(int i, int n) {
    String S = String.valueOf(i);
    int nbsp = n - S.length(); // nombre d'espaces à ajouter
    if (nbsp>0) for (int k=0; k<nbsp; k++) System.out.print(' ');
    System.out.print(i);
}

static {
    try {
        br = new BufferedReader(new FileReader(FileDescriptor.in));
```

```
        }
        catch (Exception e) {
            System.out.println("Erreur d'ouverture du terminal");
        }
    }
}
```

Dans la classe `terminal`, toutes les fonctions sont `static` : il n'y a donc pas besoins d'insatancier un objet de type `terminal` pour les utiliser.

La classe suivante teste la classe `terminal`.

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


public class es {

    public static void main (String[] args){
        System.out.println("Début du programme...");


        terminal.efface();

        /* Lecture d'un entier */
        System.out.print("i = ");
        int i = terminal.lireEntier();
        System.out.println("Vous avez entré " + i +';');

        /* Lecture d'un réel */
        System.out.print("x = ");
        float x = terminal.lireReel();
        System.out.println("Vous avez entré " + x +';');

        /* Lecture d'un caractère */
        System.out.print("c = ");
        char c  = terminal.lireCar();
        System.out.println("Vous avez entré " + c +';');

        /* Lecture d'une chaîne */
        System.out.print("S = ");
        String S   = terminal.lireChaine();
        System.out.println("Vous avez entré " + S +';');

        System.out.println("Sorties formatées :");
        terminal.ecrireEntierF(1,5);
        terminal.ecrireEntierF(12,5);
        terminal.ecrireEntierF(123,5);
        terminal.ecrireEntierF(1234,5);
        System.out.println();
```

```
        terminal.ecrireEntierF(1234,5);
        terminal.ecrireEntierF(123,5);
        terminal.ecrireEntierF(12,5);
        terminal.ecrireEntierF(1,5);
        System.out.println();

        System.out.println("Fin normale  du programme.");
    }
}
```

Remarque : la sortie écran ne pose pas de problème car les fonctions de Java `System.out.print` et `System.out.println` assurent des affichages pour tous les types de base.

7.3 Lecture formatée Classes `FileReader` et `StreamTokenizer`

Ces deux objets associés permettent de réaliser des opérations de haut niveau sur des fichiers au format complexe pouvant contenir différents séparateurs, des commentaires, des mots, des nombres ...

Supposons par exemple que l'on désire lire le fichier suivant (*p3d.txt*), dans lequel le caractère `'#'` indique une ligne de commentaire, et ou les séparateurs peuvent être les caractères «espace» (code ASCII 32) ou le caractère `','` (code ASCII 44) :

```
# commentaire ...
5
A,0,0,1
B 1 0 0
E 2.7 -6.5 8.9
# fin de l'information
```

Le programme suivant permet de «trier» les informations au cours de la lecture du fichier.

```
import java.io.*;

/* ------------  objet3D ----------------------------------- */
class objet3D {

    objet3D(FileReader fi)  throws IOException {

        StreamTokenizer st = null;
        st = new StreamTokenizer(fi);
        st.whitespaceChars(32,44); // inclut les ',' et les <SP>
        st.eolIsSignificant(true);
        st.commentChar('#');  // ligne à passer
        boolean Fin  = false;
        do  {
```

```
            switch (st.nextToken()) {
            case  StreamTokenizer.TT_EOF:
                Fin = true;
                System.out.println("Fin de fichier");
                break;
            case  StreamTokenizer.TT_EOL:
                System.out.println("Saut de ligne");
                break;
            case StreamTokenizer.TT_NUMBER:
                System.out.println("Nombre " + st.nval);
                break;
            case StreamTokenizer.TT_WORD:
                System.out.println("Message trouvé!");
                break;
            default:
                System.out.println("Je ne sais pas !");
                break;

            }
        } while (! Fin);
    }
}

/* ------------- Programme principal ------------------------- */
class fich2 {
    public static void main(String args[]){
        FileReader f = null;
        objet3D x3D;
        try {
            f = new FileReader(new File("p3d.txt"));
            x3D = new objet3D(f);
        }
        catch (Exception e){}
    }
}
```

Lorsqu'on exécute la commande

```
java fich2
```

la trace d'exécution est :

```
Saut de ligne
Nombre 5.0
Saut de ligne
Message trouvé!
Nombre 0.0
Nombre 0.0
Nombre 1.0
Saut de ligne
Message trouvé!
Nombre 1.0
Nombre 0.0
Nombre 0.0
Saut de ligne
Message trouvé!
Nombre 2.7
Nombre -6.5
Nombre 8.9
Saut de ligne
Saut de ligne
Fin de fichier
Message trouvé!
```

Remarques :

- Les lignes de commentaires n'ont donné lieu à aucun affichage.
- On peut cumuler plusieurs instructions `st.whitespaceChars(..,..)` ; lorsque les séparateurs sont plus nombreux que 2. On les inscrit 2 par 2 (pour un seul ⇒ deux fois le même ...).

7.4 Écriture de fichiers : `FileOutputStream`

L'écriture d'informations obéit aux mêmes principes généraux. L'exemple suivant créée un fichier contenant une chaîne de caractères et un entier sur 32 *bits*.

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

/* ---------  objet3D ----------------------------------- */
class objet3D {

    objet3D(){}

    public void Sauve(FileOutputStream fw)  throws IOException {
        DataOutputStream dos = new DataOutputStream(fw);
        dos.writeChars("Première ligne\n");
        dos.writeInt(0x1234789a);
        dos.close();
    }
}

/* --------------- Programme principal ------------------------- */
class fich3 {

    public static void main(String args[]){
        //      FileWriter f = null;
        FileOutputStream  f = null;
        objet3D x3D;
        x3D = new objet3D();
        try {
            f = new FileOutputStream("p3d1.txt");
            x3D.Sauve(f);
```

```
            f.close();
        }
        catch (Exception e){}
    }
}
```

Le tableau suivant reproduit la sortie d'un éditeur «hexa» montrant le contenu du fichier *p3d1.txt* créé :

Adresse	Valeur hexadécimale	Valeur ASCII
0000000	0050 0072 0065 006d 0069 00e8 0072 0065	.P.r.e.m.i.è.r.e
0000010	0020 006c 0069 0067 006e 0065 000a 1234	. .l.i.g.n.e...4
0000020	789a	x

Remarques :

- Avec `writeChars`, les caractères sont stockés sur deux octets. Les codes ASCII en constituent les poids faibles.
- L'entier est bien produit en binaire et non pas avec un format ASCII. Dans le fichier, les poids faibles et forts ne sont pas inversés.
- La taille du fichier est de 34 octets.
- Pour écrire des caractères sur 8 *bits*, il faut utiliser la méthode `writeBytes`.

7.5 Application

Exercice 1 Compléter la classe `terminal` en y ajoutant une méthode :

`ecrireReelF(float X, int n)`

qui affiche le réel X avec n chiffres après la virgule. Pour tester ce programme, écrire un programme principal (dans une classe différente) qui lit trois entiers n_1, n_2 et n_3 (entrés au clavier) et qui affiche $\frac{n_1}{n_2}$ avec n_3 chiffres après la virgule.

Exercice 2 Écrire un programme qui compte le nombre de boucles `for` et de boucles `while` d'un source écrit en langage `C`.

Chapitre 8

Communication par réseau

Ce chapitre ne donne qu'un aperçu sommaire des objets à mettre en œuvre pour établir une liaison entre deux machines. Il se compose d'un exemple qui peut être exécuté sur deux machines reliées par le réseau.

8.1 Le serveur (classe `ServerSocket`)

Le serveur est construit en instanciant un objet `ServerSocket`. Le constructeur prend comme argument le numéro de port sur lequel se fait l'écoute (attente de connexion d'un client). La démarche à suivre est :

1. Lancer le serveur.

```
ServerSocket V = new ServerSocket(1237);
```

2. Attendre qu'un client se connecte :

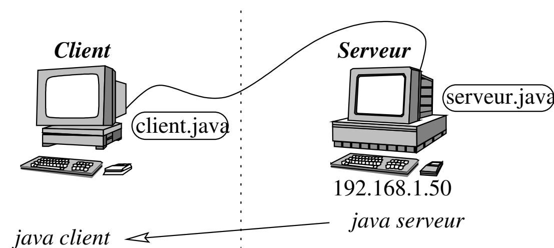

FIG. 8.1 – *Exemple de connexion réseau. Le fichier serveur.java est compilé sur une machine d'adresse IP 192.168.1.50. Le fichier client.java est compilé sur une machine connectée à la première. L'application 'java serveur' est lancée sur la machine hôte avant l'application 'java client'.*

```
Socket C = S.accept();
```

3. Ouvrir les flux de communication en entrée et en sortie pour permettre le dialogue (voir dans l'exemple).
4. Communiquer en utilisant les méthodes de `DataInputStream` et `DataOutputStream`.
5. Fermer les flux ainsi que la connexion en fin de session.

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

class serveur {
    public static void main(String[] args){

        /* les objets du serveur : */
        ServerSocket Serveur; /* le serveur lui même */
        Socket Client;  /* un par client */
        DataInputStream dis;
        DataOutputStream dos;

        try {
            /* 1 - démarrage du serveur (sur le port 1237) */
            Serveur = new ServerSocket(1237);

            /* 2 - attente d'une connexion d'un client */
            Client = Serveur.accept();

            /* 3 - ouverture des flux de communiction */
            dis = new DataInputStream(
                    new BufferedInputStream(Client.getInputStream()));
            dos = new DataOutputStream(
                   new DataOutputStream(Client.getOutputStream()));

            /* 4 - envoi d'un message au client */
            dos.writeByte(85);
            dos.flush();

            /* 5 - attente de la réponse */
            byte B =  dis.readByte();
            System.out.println("Reçu : " + B);

            /* 6 - çà suffit ! */
            dis.close();
            dos.close();
            Client.close();
        }
        catch(Exception e) {
            System.out.println("Erreur");
            System.exit(1);
        }
    }
}
```

8.2 Le client (Classe `Socket`)

La classe `Socket` fourni une interface de communication réseau, vue du côté client. La démarche à suivre est la suivante :

1. Ouvrir une connexion vers un serveur. Ceci se fait en créant un objet `Socket`. Le constructeur de `Socket` prend comme argument le nom du serveur et le numéro de port.

   ```
   Socket C = new Socket("192.168.1.50", 1237);
   ```

2. Ouvrir les flux de communication en entrée et en sortie pour permettre le dialogue (voir dans l'exemple).
3. Communiquer en utilisant les méthodes de `DataInputStream` et `DataOutputStream`.
4. Fermer les flux ainsi que la connexion en fin de session.

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

class client {
    public static void main(String[] args){

        /* Les objets du client */
        Socket Client;
        DataInputStream dis;
        DataOutputStream dos;

        try {
            /* 1 - connexion au serveur */
            Client = new Socket("192.168.1.50", 1237);

            /* 2 - ouverture des flux d'E/S du client */
            dis = new DataInputStream(
                   new BufferedInputStream(Client.getInputStream()));
            dos = new DataOutputStream(
                   new DataOutputStream(Client.getOutputStream()));

            /* 3 - attente d'un message du serveur */
            byte S = dis.readByte();
            System.out.println("Reçu S = " + S);

            /* 4 - écho du message après modification */
            dos.writeByte(S+1);
            dos.flush();

            /* 5 - fin normale */
            dos.close();
            dis.close();
            Client.close();
        }
        catch (IOException e) {
            System.out.println("Erreur client");
            System.exit(1);
        }
    }
}
```

Exercice 3 Modifier l'exemple précédent pour permettre au client d'envoyer des chaînes de caractères (terminées par `'\n'`) au serveur. Le serveur renvoie alors au client le nombre de mots contenus dans le message. L'échange se termine lorsque le client envoie la chaîne de caractère `"fin\n"`.

8.3 Serveur multisession

Dans l'exemple suivant, la classe `Serveur` écoute sur le port `9010`. Dès qu'un client se connecte, un *thread* est créé (instance de la classe `Lien`.

8.3.1 Classes du serveur

Classe `Serveur`

Cette classe contient également le programme principal de l'application serveur.

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

class Serveur implements Runnable{

    /* les objets du serveur : */
    ServerSocket leServeur; /* le serveur lui même */
    Vector listeClient = new Vector();
    boolean debug=true;

    Thread th;
    int Id=1;

    public Serveur(){
        /** Démarrage du serveur (sur le port 1237) */
        try {
            leServeur = new ServerSocket(9010);
        } catch(Exception e){
            System.out.println("Impossible de lancer le serveur");
            System.exit(0);
        }
        th = new Thread(this);
    }

    public void run(){
        while (true) {
            try {
                /** Attente d'une connexion d'un client */
                Socket S = leServeur.accept();
                System.out.println("Connexion acceptée");
                Lien L = new Lien(S, Id++);
                listeClient.addElement(L);
                System.out.println(listeClient);
            } catch(Exception e) {
                System.out.println("Erreur de connexion d'un client");
```

```
            }
        }
    }

    public void go(){
        th.start();
    }

    public static void main(String[] args){
        Serveur S = new Serveur();
        S.go();
    }

}
```

Classe `Lien`

Une instance de cette classe est créée à chaque nouvelle connexion. L'application cliente consiste ici à envoyer au serveur des chaînes de caractères saisies au clavier. Le serveur renvoie à chaque fois un prompt.

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

class Lien implements Runnable {
    DataInputStream dis;
    DataOutputStream dos;
    Socket socketClient;
    int Id; // identificateur pour la connexion

    Thread th;
    byte[] tOK;

    public Lien(Socket S, int n){
        socketClient = S;
        Id = n;
        /** Ouverture des flux de communication */
        try {
            dis = new DataInputStream(
                        new BufferedInputStream(S.getInputStream()));
            dos = new DataOutputStream(
                        new DataOutputStream(S.getOutputStream()));
        }catch(Exception e){}
        th = new Thread(this);
        String Sprompt = (new Integer(Id))+" - OK> ";
```

```
        tOK = Sprompt.getBytes();
        th.start();
    }

    protected byte[] lireTrame(){
        byte[] b = null;
        do {
            try {
                try {Thread.sleep(30);} catch(Exception e){}
                int N = dis.available(); /** nombre d'octets disponibles */
                if (N!=0) {
                    b = new byte[N];
                    dis.read(b);
                }
            } catch (Exception e){
                System.out.println("Erreur de réception");
            }
        } while (b==null);
        return b;
    }

    protected void ecrireTrame(byte[] b){
        try {
            dos.write(b);
        } catch (Exception e){}
    }

    public void run(){
        ecrireTrame(tOK); // trame d'accueil
        while (true) {
            byte[] b = lireTrame();
            System.out.println("Trame reçue (" + b.length + ")");
            ecrireTrame(tOK); //réponse
        }
    }

    public void close(){
        try {
            dis.close();
            dos.close();
        } catch (Exception e){}
    }

    public String toString(){
        return socketClient.toString();
    }
```

```
}
```

8.3.2 Classe du client

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

class Client {
    public static void main(String[] args){

        /* Les objets du client */
        Socket Client;
        DataInputStream dis;
        DataOutputStream dos;

        try {
            /* 1 - connexion au serveur */
            Client = new Socket("172.16.16.245", 9010);

            /* 2 - ouverture des flux d'E/S du client */
            dis = new DataInputStream(
                   new BufferedInputStream(Client.getInputStream()));
            dos = new DataOutputStream(
                   new DataOutputStream(Client.getOutputStream()));

            /* 3 - attente d'un message du serveur */
            int n = dis.available();
            byte[] b = new byte[n];
            dis.read(b);
            System.out.println(new String(b));
            String S;
            do {
                S = terminal.lireChaine();
                b = S.getBytes();
                dos.write(b);
                dos.flush();
                do {
                    n = dis.available();
                    b = new byte[n];
                    dis.read(b);
                    System.out.print(new String(b));
                    if (n==0) {
                        try {Thread.sleep(20);} catch(Exception e){}
                    }
```

```
            } while (n==0);
        } while (!S.equals("quit"));


        /* 5 - fin normale */
        dos.close();
        dis.close();
        Client.close();
    }
    catch (IOException e) {
        System.out.println("Erreur client");
        System.exit(1);
    }
  }
}
```

Chapitre 9

Création d'interfaces graphiques

Il s'agit dans ce chapitre de se familiariser avec la mise en œuvre des outils d'interfaces usuels :

- traçage de textes et d'éléments graphiques
- formulaires (boutons, cases à cocher, boîtes de défilement ...)
- traitement des événements de la souris et du clavier.

On procèdera de manière graduelle en introduisant progressivement les éléments dans des *applets* de plus en plus complètes.

La dernière section du chapitre est un exemple qui montre comment procéder pour une application `Java` .

9.1 Traçage de textes et de dessins

9.1.1 Les *applets*

Les *applets* («petites applications») sont des programmes appelés à partir d'une page HTML. Une page HTML (*HyperText Markup Language*) est un document interprété par un navigateur. Les *applets* servent à rendre une page *internet* plus attractive. Elles ne peuvent être exécutées que si le navigateur dispose d'un interpréteur `Java` (c'est en géneral le cas) et que cet interpréteur est validé (ce n'est pas toujours le cas : avec `Netscape`, aller voir dans *Edit* → *préferences* → *advanced* → *Enable Java*).

Exemple. Soit le fichier *app1.html* suivant

```html
<html>
<head>

<title>Textes et dessins</title>
</head>

<body BGCOLOR="#a0e0e0" >
<center>
<font color="blue">
<h1>Traçage de textes et de dessins</h1>
</font>

<font color="red">
<h2>Exemple d'Applet</h2>
</font>

<applet code=app1.class width=300 height=300>
 <b>Votre navigateur n'est pas compatible <i>Java</i></b>
</applet>
</body>
</center>
</html>
```

Le code de l'*applet* est contenu dans le fichier *app1.java* :

```java
import java.applet.*;
import java.awt.*;
import java.io.*;

public class app1 extends Applet {

    public void paint(Graphics g) {
        g.setColor(Color.blue);
        g.fillRect(10, 20, 120, 50);
        g.setColor(Color.black);
        g.drawString("Hello World !",30,50);
    }

}
```

9.2 Traitement des événements liés à la souris

Les événements produits par la souris peuvent être gérés dans une application ou une *applet* `Java` . Ces événements sont :

- un *click* sur l'un des boutons (enfoncement puis relachement rapide) : *mouse clicked* ;
- un bouton a été enfoncé (mais pas relaché immédiatement) : *mouse pressed* ;
- un bouton est relaché : *mouse released* ;

– le curseur de souris entre dans la fenêtre de l'application : *mouse entered* ;
– le curseur de souris sort de la fenêtre de l'application : *mouse exited* ;
– la souris est déplacée avec un bouton enfoncé : *mouse dragged* ;
– la souris est déplacée sans action sur les boutons : *mouse moved*.

Pour `Java` , ces événements sont séparés en deux groupes :

MouseListener
MouseListener `mouseClicked`
`mousePressed`
`mouseReleased`
`mouseEntered`
`mouseExited`

MouseMotionListener
`mouseDragged`
`mouseMoved`

Ainsi, une *applet* traitant des événements pris dans chacun de ces groupes devra être déclarée :

```
public class app2 extends Applet implements MouseListener,MouseMotionListener {
   ...
}
```

Ce qui signifie que l'*applet* `app2` **implémente** les méthodes des **interfaces** `MouseListener` et `MouseMotionListener` .

9.2.1 Exemple

L'exemple suivant reprend l'essentiel des principes exposés précédemmnent.

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

public class app2 extends Applet implements MouseListener,MouseMotionListener
{

    int debutX, debutY;
    int finX, finY;
    int X, Y;
    String P;

    /* Constructeur */
    public app2(){
        setBackground(Color.white);
        addMouseMotionListener(this);
        addMouseListener(this);
        debutX=-1;
        debutY=-1;
    }
```

```
public void mousePressed(MouseEvent e) {
    e.consume();
    debutX = e.getX();
    debutY = e.getY();
    X =  Y = -1;
    finX = finY = -1;
    int b =  e.getModifiers();
    if (b == InputEvent.BUTTON1_MASK)  P = new String("Bouton 1");
    else if (b==0) P = new String("Bouton 1"); // netscape !!
    else if (b == InputEvent.BUTTON2_MASK)  P = new String("Bouton 2");
    else if (b == InputEvent.BUTTON3_MASK)  P = new String("Bouton 3");
    else P = new String("Bouton ???");
            repaint();
}

public void mouseReleased(MouseEvent e) {
    e.consume();
    finX = e.getX();
    finY = e.getY();
    repaint();
}

public void mouseDragged(MouseEvent e) {
    e.consume();
    X = e.getX();
    Y = e.getY();
    repaint();
}

/* Méthodes inutilisées, mais devant être définies */
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mouseClicked(MouseEvent e) {}
public void mouseMoved(MouseEvent e) {}

/* Dessin de l'applet */
public void paint(Graphics g) {
    if (debutX>0) {
        g.setColor(Color.yellow);
        g.drawString(P,debutX,debutY);
    }
    if (X>0) {
        g.setColor(Color.red);
        g.drawLine(debutX, debutY, X, Y);
    }
    if (finX>0) {
        g.setColor(Color.cyan);
        g.drawLine(debutX, debutY, finX, finY);
        g.setColor(Color.green);
        g.drawString(P, finX, finY);
    }
```

```
    }
}
```

La lecture du code de cette *applet* conduit aux remarques suivantes :

1. Lorsqu'une *applet* implémente une interface, toutes les méthodes de cet interface doivent être définies, quitte à les laisser vide, comme `mouseEntered` ou `mouseMoved` par exemple.
2. Les fonctions (ou méthodes) de traitement reçoivent un argument unique de type `MouseEvent`.
3. Le traitement d'un événement de la souris commence toujours par un appel `e.consume()` . En effet, les événements reçus sont nombreux et mis dans une file de traitement. Un tel appel permet d'informer le système qu'un événement est en cours de traitement et peut donc être retiré de la file.
4. L'argument «`MouseEvent e`» possède deux autres méthodes intéressantes : `getX()` et `getY()` qui renvoient respectivement l'abscisse et l'ordonnée de l'endroit ou s'est produit l'événement.
5. L'applet `app2` possède une méthode particulière portant le nom de la classe : c'est un constructeur. Cette fonction particulière est exécutée à la création de l'*applet*. On y trouve en particulier les routines d'installation du traitement des événements de la souris.

```
addMouseMotionListener(this);
addMouseListener(this);
```

Comment distinguer les boutons ... . Il faut pour cela utiliser la méthode `getModifiers()` de `MouseEvent`. Exemple :

```
int b =  e.getModifiers(); /* lecture du masque */
if (b == InputEvent.BUTTON1_MASK) {  /* Bouton 1 */  }
else if (b==0)  {  /* Bouton 1 pour Netscape !! */  }
else if (b == InputEvent.BUTTON2_MASK) { /* Bouton 2 */ }
else if (b == InputEvent.BUTTON3_MASK) { /* Bouton 3 */ }
```

Remarque : avec `Netscape`, il y a un problème de compatibilité pour le traitement du bouton 1.

9.2.2 Exécution de l'*applet* `app2.java`

L'exécution se fait en lisant le fichier `index.html` avec un navigateur, avec l'URL :

`file:///...chemin.../index.html`

ou en lançant la commande

`appletviewer index.html &`

Dans les deux cas, le fichier `app2.class` doit se trouver dans le même répertoire que `index.html`.

Remarque : avec `Netscape`, il semble que pour prendre en compte une modification de l'*applet*, il faille relancer le navigateur. Avec `appletviewer`, il suffit de la recharger avec le menu `Applet` → `Reload` (d'où l'intérêt du `&`).

9.3 Gestion des événements liés aux touches du clavier

Le principe est le même que pour les événements liés à la souris. L'interface à implémenter est : `KeyListener`.

KeyListener
`keyPressed`
`keyReleased`
`keyTyped`

Exemple de mise en œuvre : L'exemple déplace un disque rouge dans la zone de traçage de la fenêtre et affiche le code de chaque touche enfoncée.

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

public class appkey extends Applet implements KeyListener {

    int X,Y;  /* position courante */
    int code; /* code de touche enfoncé */
    int D;    /* Déplacement */

    public appkey() {
        X=Y=150;
        D=1;
    }

    public void init() {
        setBackground(Color.white);
        addKeyListener(this);
    }


    /* touche enfoncée  */
    public void keyPressed(KeyEvent e){
        e.consume();
        if (e.isAltDown()) D=10; else D=1;
        code = e.getKeyCode();
        switch (code) {
            case KeyEvent.VK_UP: Y-=D; break;
            case KeyEvent.VK_DOWN: Y+=D; break;
            case KeyEvent.VK_LEFT: X-=D; break;
```

```
                case KeyEvent.VK_RIGHT: X+=D; break;
            }
            repaint();
        }

        /* touche relachée */
        public void keyReleased(KeyEvent e){}

        /* touche enfoncée puis relachée */
        public void keyTyped(KeyEvent e){}

        public void paint(Graphics g){
            g.setColor(Color.blue);
            g.drawString(Integer.toString(code), 10, 20);
            g.setColor(Color.red);
            g.fillOval(X-10,Y-10,20,20);
        }
    }
```

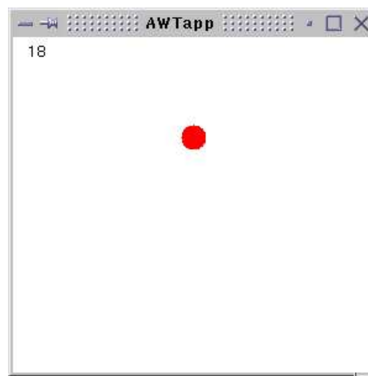

FIG. 9.1 – *Rendu de l'applet* ***appkey****. Le code de la dernière touche enfoncée apparaît en haut à droite. Le disque est déplacé par les touches «flèches». L'appui simultané sur la touche* ***Alt*** *multiplie la vitesse du déplacement par 10.*

Remarques :

- exemple testé avec `Netscape`, résiste obstinément à `appletviewer`. Fonctionne également en application (`Frame` au lieu de `Applet`).
- Quelques constantes sont prédéfinies dans la classe `java.awt.event.KeyEvent`. Elles représentent les touches du clavier :

`VK_0` à `VK_9`	Touches des chiffres
`VK_A` à `VK_Z`	Touches des caractères
`VK_CLEAR`	Touche d'effacement
`VK_DOWN`	Flèche vers le bas
`VK_END`	Touche fin
`VK_ESCAPE`	Touche d'échappement
`VK_F1` à `VK_F12`	Touche de fonctions
`VK_HELP`	Touche d'aide
`VK_HOME`	Touche début
`VK_INSERT`	Touche d'insertion
`VK_LEFT`	Flèche vers la gauche
`VK_NUMPAD0` à `VK_NUMPAD9`	Touches du pavé numérique
`VK_PAGE_DOWN`	Page suivante
`VK_PAGE_UP`	Page précédente
`VK_PAUSE`	Touche pause
`VK_PRINTSCREEN`	Touche d'impression d'écran
`VK_RIGHT`	Flèche vers la droite
`VK_UP`	Flèche vers le haut

Il y en a d'autres, pour les obtenir toutes :

```
javap java.awt.event.KeyEvent
```

Il est possible de savoir si les touches de modification (`Alt`, `Control` et `Shift`) sont enfoncées avec les méthodes `isAltDown()`, `isControlDown()` et `isMetaDown()` de `KeyEvent`.

Exemple

```
if (e.isMetaDown()) { ... }
```

9.4 Les composants évolués

`Java` propose quelques composants standards pour réaliser des interfaces de haut niveau :

Étiquettes : c'est le composant le plus simple, il s'agit d'une zone de texte. Elles servent en général à informer sur l'interface. La différence entre une étiquette et un texte affiché par `drawString` est importante : un étiquette n'a pas besoins d'être réaffichée dans le `paint`. La prise en compte est automatique. (Classe `Label`)

Boutons : composant également très simple et très répandu. Un bouton sélectionné déclenche une action. (Classe `Button`)

Cases à cocher : ce sont des composants d'interface caractérisés par deux états : sélectionné ou non. (Classe `Checkbox`).

Boutons radio : ce sont des cases à cocher fonctionnant en groupe. Un seul bouton peut être coché à la fois. (Classes `Checkbox` et `CheckboxGroup`).

Menus de sélection : ce sont des listes déroulantes (ou *popup*) d'articles. (Classe `Choice`)

Listes à défilement : fournit un menu d'articles dont un ou plusieurs peuvent être sélectionnés ou désélectionnés. C'est un composant un peu plus évolué que le précédent. (Classe `List`)

Champs de textes : ils permettent de saisir un texte. (Classe `TextField`)

9.4.1 Utilisation

Pour qu'un composant soit pris en compte, il faut :

1. le créer,
2. l'ajouter dans la fenêtre,
3. l'associer à un écouteur d'événements,
4. traiter les événements dans une méthode de l'écouteur associé.

Exemple Considérons un bouton (classe `Button`). C'est une classe qui comporte une méthode `addActionListener`. Ceci signifie que l'application ou l'*applet* doit implémenter l'interface `ActionListener` pour permettre la gestion des événements liés à ce bouton. Ainsi l'*applet* doit être déclarée :

```
public class appIhm extends Applet implements ActionListener {
  Button B;
  ...
}
```

Dans le constructeur de l'application, ou dans la méthode *init* de l'*applet*, on doit trouver les lignes

```
B = new Button("Nom du bouton");
add(B);
B.addActionListener(this);
```

Enfin, la méthode `actionPerformed` de l'interface `ActionListener` doit être écrite :

```
public void actionPerformed(ActionEvent e){
    if (e.getSource() == B) { /* action à exécuter */};
}
```

Il en est de même pour les autres composants. Le tableau suivant regroupe les composants et l'interface écouteur associé.

Composant	interface	Méthode à implémenter
`Button`	`ActionListener`	`actionPerformed`
`TextField`	`ActionListener`	`actionPerformed`
`List`	`ActionListener`	`actionPerformed`
`Checkbox`	`ItemListener`	`itemStateChanged`
`Choice`	`ItemListener`	`itemStateChanged`

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

public class appIhm extends Applet
      implements ActionListener, ItemListener {

    /* Elements d'interface */
    Button Commander;
    Checkbox Fromage, Dessert, Vin, Eau, Jdf;
    Choice Choix;
    List Entree;
    TextField Remarque;

    /* Variables d'état de l'application */
    boolean fromage, dessert, eau, vin, jdf;
    String plat, remarque, entree;

    public appIhm() {
        fromage=true; dessert=false;  eau=vin=jdf=false;
        plat = new String("Viande");
        remarque = new String("");
        entree = new String("Radis");
    }

    public void init() {
        setLayout(new FlowLayout());

        /* Label */
        add (new Label("Menu"));

        /* Menu déroulant */
        Choix = new Choice();
        Choix.add("Viande"); Choix.add("Poisson");
        add(Choix);
        Choix.addItemListener(this);

        /* Liste */
        Entree = new List();
        Entree.add("Radis"); Entree.add("Salade");
        Entree.add("Pamplemousse"); Entree.add("Jambon");
        Entree.add("Rien");
        add(Entree);
        Entree.addActionListener(this);

        /* Case à cocher */
        Eau = new Checkbox("Eau");
        Jdf = new Checkbox("Jus de fruit");
        Vin = new Checkbox("Vin");
        add(Eau); add(Vin); add(Jdf);
        Eau.addItemListener(this);
        Vin.addItemListener(this);
```

```
        Jdf.addItemListener(this);

        /* Boutons radio */
        CheckboxGroup CbG = new CheckboxGroup();
        Fromage =  new Checkbox("Fromage", true, CbG);
        Dessert =  new Checkbox("Dessert", false, CbG);
        add(Fromage);
        add(Dessert);
        Fromage.addItemListener(this);
        Dessert.addItemListener(this);

        /* Bouton */
        Commander = new Button("Commander");
        add(Commander);
        Commander.addActionListener(this);

        /* Zone de texte */
        Remarque = new TextField("Remarque", 30);
        add(Remarque);
        Remarque.addActionListener(this);
    }

    /* Ecouteur pour les boutons */
    public void actionPerformed(ActionEvent e){
        if (e.getSource() == Commander) repaint();
        else if (e.getSource() == Remarque) remarque = Remarque.getText();
        else if (e.getSource() == Entree) entree = Entree.getSelectedItem();
    }

    /* Ecouteur pour les cases à cocher */
    public void itemStateChanged(ItemEvent e) {
        if (e.getSource() ==  Choix) plat = Choix.getSelectedItem();
        else if (e.getSource() == Fromage) { fromage=true; dessert=false;}
        else if (e.getSource() == Dessert) { dessert=true; fromage=false;}
        else if (e.getSource() == Jdf) jdf= Jdf.getState();
        else if (e.getSource() == Vin)  vin = Vin.getState();
        else if (e.getSource() == Eau) eau= Eau.getState();
    }

    public void paint(Graphics g) {
        g.setColor(Color.blue);
        g.drawString(remarque, 10, 245);
        g.setColor(Color.red);
        g.drawString(entree, 10, 185);
        g.drawString(plat, 10,200);
        if (jdf) g.drawString("Jus de fruit", 10,215);
        if (vin) g.drawString("Vin", 90,215);
        if (eau) g.drawString("Eau", 140,215);
        if (dessert) g.drawString("Dessert", 10, 230);
        else g.drawString("Fromage", 10, 230);
    }
}
```

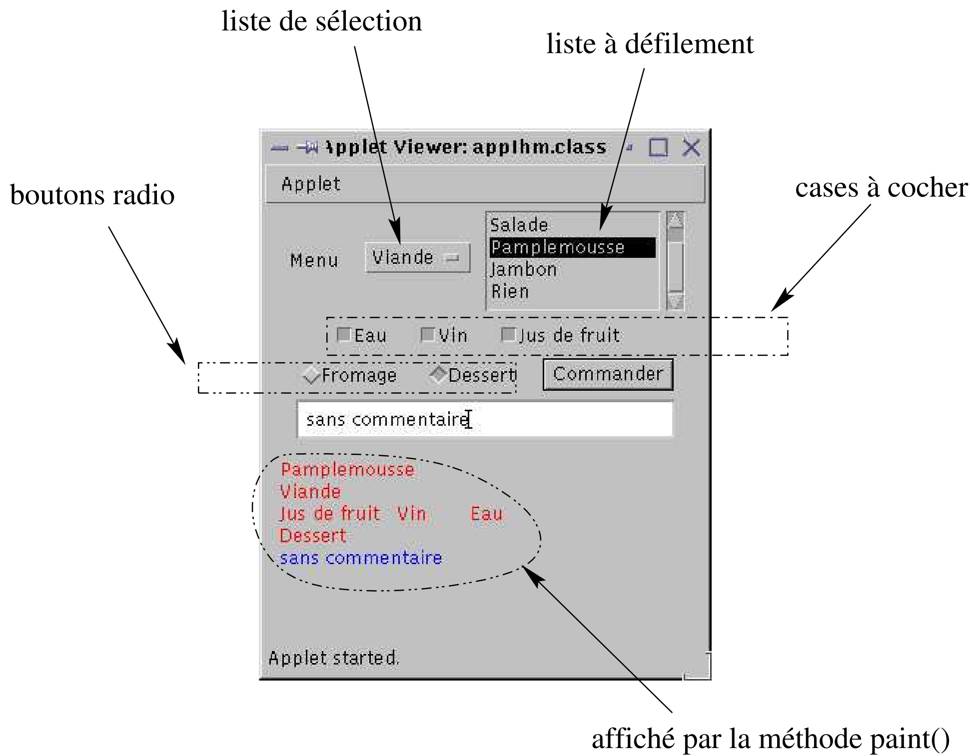

FIG. 9.2 – *Rendu de l'applet* `appIhm`

9.4.2 Placement des composants

Dans l'exemple précédent, les composants sont placés à la volée en fonction de l'emplacement disponible et de leur encombrement , et ceci dans l'ordre ou ils sont insérés dans le programme. Ce comportement est imposé par la ligne

```
setLayout(new FlowLayout());
```

de la méthode `init()`. Notons au passage que pour une *applet*, c'est le comportement par défaut (cette instruction est donc facultative), mais par contre pour une application, elle devient le minimum obligatoire, sans quoi, chaque composant occupe toute la fenêtre, ce qui fait qu'on ne voit que le dernier inséré.

Les autres possibilités sont :

`GridLayout` : la page est divisée en lignes et en colonnes. Les composants occupent les cases dans l'ordre d'insertion.

`BorderLayout` : cette mise en page place les composants selon des critères géographiques : sud, nord, est, ouest ou centre. Elle ne gère efficacement que 5 composants.

`CardLayout` : une mise en page de type *card* permet de visualiser les composants par groupe. Cette technique s'apparente aux menus avec onglets. La classe `CardLayout` possède des méthodes permettant de passer d'une carte à l'autre. On peut alors associer un bouton à chaque carte, de façon à permettre la sélection du menu.

`GridBagLayout` : permet de faire une présentation de manière très précise. Cette solution utilise un concept de grille sur laquelle les composants trouvent leur place.

9.5 Interface homme machine et zone de traçage

Beaucoup d'applications mélangent les genres : il faut non seulement proposer une interface conviviale (boutons, menus déroulants ...) ; mais il faut en plus tracer des graphiques, voire accéder à une interface purement graphique.

La solution `Java` est donnée par la classe `Canvas` qui est simplement une zone de traçage. Il suffit de l'insérer (avec `add`) dans la fenêtre principale de l'application.

9.5.1 Méthode conseillée

1. On réalise l'interface homme machine «comme d'habitude», la fenêtre principale d'application (qui hérite de `Applet` pour une *applet* et de `Frame` pour une application) implémente tous les interfaces nécessaires à ses composants (`ActionListener`, `ItemListener`).
2. On fait hériter une classe à la classe `Canvas`. Cette nouvelle classe implémente les interfaces `MouseListener`, `MouseMotionListener`, et `KeyListener`.
3. Un objet instancié de la classe précédente est ajouté à la fenêtre principale.

9.5.2 Exemple

L'exemple est proposé sous forme d'*applet*. Une interface de commande est composée de deux boutons.

- un bouton `HG` qui renvoie le curseur graphique en haut à gauche de la zone de traçage,
- un bouton `BD` qui renvoie le curseur graphique en bas à droite de la zone de traçage.

La zone de traçage capture les événements de la souris et le curseur graphique peut être déplacé. Un regtangle rouge est dessiné à la position courante du curseur graphique.

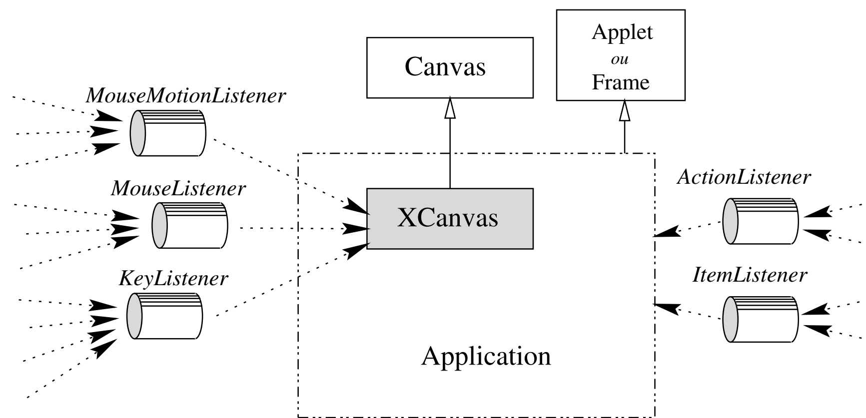

FIG. 9.3 – *L'application traite directement les événements liés aux composants de l'interface. Les événéments liés à la souris et au clavier sont «sous–traités» par un objet graphique* `Xcanvas`

Fichier `appTrace.java`

On y trouve le traitement des événements liés aux composants.

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


/* Applet : appTrace.java
            ---------
   * Crée 2 boutons (HG et BD)  + une zone de traçage. (X)
   * L'applet prend en compte les événements liés aux
     boutons.
   * Les événements liés à la souris sont laissés à la zone
     de traçage.
*/

public class appTrace extends Applet  implements ActionListener {
    XCanvas X;
    Button HG;
    Button BD;

    public appTrace (){
        setLayout( new FlowLayout());
        HG = new Button("Haut-Gauche");
        BD  = new Button("Bas-Droite");
        add(HG);
        add(BD);
        HG.addActionListener(this);
        BD.addActionListener(this);
        X = new XCanvas();
        add(X);
```

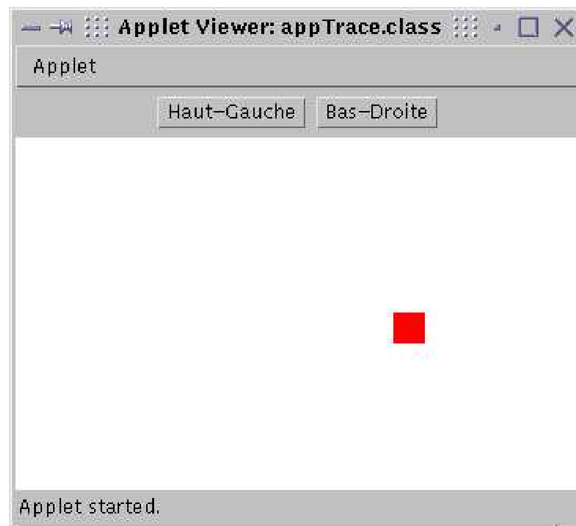

FIG. 9.4 – *Rendu de l'applet* ***appTrace***.

```
    }


    public void actionPerformed(ActionEvent e){
        if (e.getSource() == HG) {
            System.out.println("en haut à gauche ...");
            X.setXY(10,10);
            X.repaint();
        }
        else {
            System.out.println("en bas à droite ...");
            X.setXY(400,300);
            X.repaint();
        }
    }

}
```

Fichier `XCanvas.java`

C'est la partie graphique. Les événements liés à la souris sont traités.

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

/*
   classe XCanvas.java
          ------------
```

```
   Zone de traçage de l'applet appTrace
   Traite les événements de la souris
*/

public class XCanvas extends Canvas
    implements MouseListener,MouseMotionListener {

    int X,Y;

    /* Constructeur */
    public XCanvas() {
        super();
        setBackground(Color.white);
        setSize(new Dimension(512,384));
        X=Y=100;
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    public void paint(Graphics g) {
        g.drawString("Zone de traçage", X,Y);
    }

    public void setXY(int x, int y){
        X=x;
        Y=y;
    }

    public void mousePressed(MouseEvent e) {
        e.consume();
        X = e.getX();
        Y = e.getY();
        repaint();
    }

    public void mouseDragged(MouseEvent e) {
        e.consume();
        X = e.getX();
        Y = e.getY();
        repaint();
    }

    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mouseClicked(MouseEvent e) {}
    public void mouseMoved(MouseEvent e) {}
}
```

9.6 Application Java

Il y a peu de différences avec les *applets*. L'application contient une méthode `main`, et la fenêtre de l'application hérite de la classe `java.awt.Frame`. Voici un extrait d'application avec fenêtre :

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

class Xscope extends Frame {


    private Button BoutonCH = new Button("CH");
    private Button BoutonSauver = new Button("Sauver (.fig)");
    private Button BoutonQuitter = new Button("Quitter");


    class ActionQuitter extends WindowAdapter implements ActionListener {
        private void fin(){
            // instructions à exécuter avant de partir ...
            System.exit(0);
        }
        public void actionPerformed(ActionEvent e) { fin(); }
        public void windowOpened(WindowEvent e) {}
        public void windowClosing(WindowEvent e){ fin(); }
    }

    class ActionCH implements ActionListener {
        public void actionPerformed(ActionEvent e){
            // actions de du bouton CH1
        }
    }

    class ActionSauver implements ActionListener {
        public void actionPerformed(ActionEvent e){
            FileDialog fd;
            fd = new FileDialog(fen, "Sauvegarde");
            fd.setMode(FileDialog.SAVE);
            fd.setDirectory(".");
            fd.show();
            // nom du fichier à sauver ==  'fd.getFile()'
        }
    }

    public Xscope() {
        super("EXEMPLE");
        addWindowListener(new ActionQuitter());
        setLayout(new BorderLayout());

        Panel PB = new Panel();
        BoutonCH.addActionListener(new ActionCH());
```

```
        PB.add(BoutonCH);
        BoutonSauver.addActionListener(new ActionSauver());
        PB.add(BoutonSauver);
        BoutonQuitter.addActionListener(new ActionQuitter());
        PB.add(BoutonQuitter);

        setTitle(titre);
        add(PB,"South");
        pack();
        show();
    }

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

Reparque : cet exemple apporte une autre solution pour traiter les événements des boutons. Ici, chaque bouton est associé à une classe `Listener` spécialisée. Le code est ainsi plus rationnel et facile à maintenir.

L'exécution de l'application se fait avec la commande :

```
java Xscope
```

Chapitre 10

Les *packages*

10.1 Introduction

Les *packages* ont pour objet d'organiser le logiciel lorsqu'il prend des proportions importantes en volume de code. Ils organisent les classes en groupes. Un *package* contient un nombre quelconque de classes reliées entre elles par leur objectif, ou simplement par l'héritage. D'une manière générale, les *packages*

- permettent d'organiser les classes en unités,
- réduisent les problèmes de conflits de noms,
- peuvent servir à identifier les classes.

Un *package* peut contenir d'autres *packages*. Ceci permet de gérer des niveaux d'organisation.

10.2 Utilisation des *packages*

Accès avec le nom complet Avec cette méthode, le nom de la classe doit être précédé du ou des noms de *packages* auquel elle appartient. Exemple :

```
java.awt.Font f = new java.awt.Font();
```

Cette méthode d'accès est adaptée lorsqu'une classe est utilisée peu de fois dans un programme.

La commande `import` permet d'importer des classes depuis un *package*. Une classe peut être importée individuellement :

```
import java.awt.Font;
```

On peut également importer tout un *packages* :

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

remarques

– L'instruction importe toutes les classes du *packages* `java.awt` qui ont été déclarées `public`
– Seules les classes auquelles se réfèrent le code sont importées.
– Elle n'importe pas les «sous»-*packages* Ainsi, pour utiliser le *package* `image`, on trouvera en général les deux lignes :

```
import java.awt.*;
import java.awt.image.*;
```

La commande Java `import` est différente de la directive `#include` du C. La directive du C est appliquée au début de la compilation et conduit à la création d'un fichier unique qui est entièrement compilé pour produire l' «exécutable». La commande `Java import` informe l'interpréteur des répertoires ou les liens pourront être effectués au cours de l'interprétation du programme.

10.3 Création de *packages*

La création de *packages* est liée à une arborescence de fichiers sur le disque ou elles seront utilisées. Pour créer ses propres *packages*, il convient donc dans un premier temps de construire les répertoires respectant la hiérarchie voulue pour les classes.

Par exemple, le fichier suivant se trouve dans le répertoire `geo` figurant lui même dans un répertoire connu par `Java` grâce à la variable $CLASSPATH$.

```
/* fichier point.java */
package geo;

public class point {
  int  X,Y;

  public point(int xx, int yy){
    X = xx;
    Y = yy;
  }

  public void affiche() {
    System.out.print("("+X+","+Y+")");
  }
}
```

Exemple : sous `UNIX`, pour fixer la variable `CLASSPATH`, on écrira les commandes successives (*shell* `bash`) :

```
$ CLASSPATH=$CLASSPATH:~/java/geo
$ export CLASSPATH
```

Exécution un programme appartenant à un *package*. Supposons que le fichier `poinT.java` du *package* `geo` soit exécutable, pour le lancer :

```
$ java geo.pointT
```

Utilisation d'une classe d'un package. Pour utiliser la classe `point` du package `geo` dans un programme, il faut écrire au préalable :

```
import geo.*;
```

Chapitre 11

Classes abstraites et interfaces

Les *interfaces* de Java fournissent des modèles de comportement pouvant être adaptés à différentes classes. L'héritage (*extends*) permet déjà à une classe de récupérer les propriétés et les méthodes (les comportements) de la classe de laquelle elle hérite.

Pour définir un comportement, la première idée qui vient en programmation objet est de définir une classe abstraite qui défini les méthodes du comportement, voire qui en implémente certaines d'entre elles. C'est la première solution que nous envisagerons dans un exemple simple.

L'interdiction en `Java` de procéder à l'héritage multiple limite les possibilités des classes abstraites. La solution Java est apportée par le concept d' $interface$.

11.1 Exemple d'héritage, à partir d'une classe abstraite

On désire implémenter une hiérarchie conforme au schéma de la figure 11.1. On se limite aux méthodes suivantes :

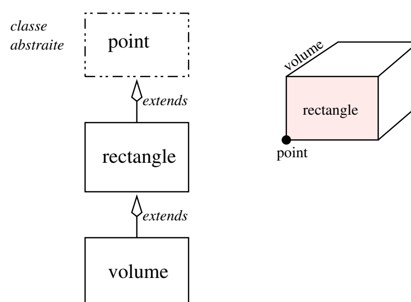

FIG. 11.1 – *Exemple d'héritage d'une classe abstraite.*

- `affiche` affichage de l'objet avec ses coordonnées courantes.
- `efface` effacement de l'objet.

– `déplacexy` déplacement de l'objet dans le plan (xy)

L'analyse de ces méthodes conduit aux conclusions :

1. les méthodes `affiche` et `efface` sont spécifiques à chacune des classes *rectangle* ou *volume*.
2. la méthode `effacexy` est indépendante de l'objet, à condition de faire appel aux deux méthodes précédentes.

On peut donc écrire la méthode `effacexy` sans savoir sur quel type d'objet elle sera appliquée. Pour cela, il faut pouvoir l'écrire sans que `affiche` et `efface` soient implémentées. C'est l'objectif des **classes abstraites** : permettre de décrire un comportement abstrait pour un ensemble de classes liées par l'héritage.

Dans l'exemple suivant, la classe `point` est déclarée **abstraite**, c'est à dire qu'elle **ne sera pas instanciée**, mais qu'elle est destinée à servir de modèle dans un processus d'héritage. Les classes `rectangle` et `volume` n'implémentent pas la méthode `deplacexy`, mais elles peuvent évidemment utiliser la méthode prévue dans la classe abstraite.

Classes

```
import java.io.*;

abstract class point {
  int X,Y;
  point(int x, int y) {X=x; Y=y;}
  abstract void dessine();
  abstract void efface();
  void deplacexy(int x, int y) {
    efface();
    X+=x; Y+=y;
    dessine();
  }
}

class rectangle extends point{
  int L, H;
  rectangle(int x, int y, int l, int h) {super(x,y); L=l; H=h;}
  void dessine() {
    System.out.println("Rectangle : ("+X+","+Y+","+L+","+H+")");
  }
  void efface() {System.out.println("Efface rectangle");}
}

class volume extends rectangle {
  int E;
  volume (int x, int y, int l, int h, int e) { super(x,y,h,l); E=e;}
  void dessine(){
    System.out.println("Volume :("+X+","+Y+","+L+","+H+","+E+")");
  }
  void efface() {System.out.println("Efface volume");}
}
```

Programme d'application

```
import java.io.*;

public class volumeT {

  public static void main (String[] argv){
    volume  V = new volume(1,1,2,2,4);
    V.dessine();
    V.efface();
    V.deplacexy(10, 20);
  }
}
```

Trace d'exécution :

```
> java volumeT
Volume :(1,1,2,2,4)
Efface rectangle
Efface rectangle
Volume :(11,21,2,2,4)
```

11.2 Avantages et inconvénients

Cette solution présente comme avantage, au moins :
- la simplicité de mise en oeuvre,
- la possibilité d'avoir des données dans la classe racine abstraite,
- les méthodes spécifiques aux classes héritées n'ont pas besoins d'être implémentées : elles sont abstraites,
- les méthodes déduites des méthodes abstraites peuvent être implémentées dans la classe abstraite.

…mais présente également un inconvénient majeur : en Java, une classe ne peut hériter que d'une «super-classe» unique[1]. Or, il pourrait être intéressant de gérer une collection d'objets `volume` en pile dans une liste chaînée, ou dans n'importe quelle autre structure de données abstraite. Il y a donc un choix de spécification à faire dans une telle situation : hériter d'une classe `pile` ou d'une classe `point`.

Le concept d'$interface$ est un élément de réponse à cette question.

11.3 Définition des *interfaces* de Java

11.4 Mise en œuvre sur un exemple

11.4.1 Gestion d'une pile en liste chaînée.

Pile. Une pile est une structure de donnée linéaire organisée selon un principe de séquence, mais les opérations d'ajout et de retrait se font d'un seul bout de la séquence (figure 11.2).

[1] Pas d'héritage multiple

On utilise une pile dès qu'un traitement doit être fait ultérieurement, par exemple dans les algorithmes de recherche dans un arbre, d'évaluation d'expressions, ...

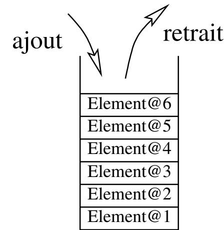

FIG. 11.2 – *Modèle de pile*

Les opérations à réaliser sur une pile sont :

1. Construire une pile initialement vide.
2. Vérifier qu'une pile est vide ou non.
3. Vérifier si une pile est pleine ou non (facultatif pour une gestion en liste chaînée).
4. Ajouter un élément au sommet de la pile (*push*).
5. Retirer l'élément en haut de la pile (précondition : la pile n'est pas vide).
6. Consulter l'élément en haut de la pile (précondition : la pile n'est pas vide).

Liste chaînée. La programmation d'une pile peut être faite, soit dans un tableau (mais l'existence de la taille du tableau oblige à gérer le fait que la pile puisse être pleine), soit dans une liste chaînée (vue au chapitre précédent).

La représentation chaînée oblige à définir les éléments de chaînes. On y trouve une partie «contenu» et un indicateur «suivant» qui marque l'élément suivant dans la pile.

La chaîne sera alors entièrement définie par l'identificateur «sommet» de l'élément de chaîne qui se trouve en haut de la pile.

11.4.2 Implémentation de l'interface

Les différentes méthodes spécifiées précédemment sont définies dans l'$interface$. Il n'y a aucune implémentation.

Fichier *Pile.class*

```
// interface Pile
// défini les primitives d'utilisation d'une liste gérée en pile.
interface  Pile {
  void push (Object element);
  Object pop();
```

```
    Object top();
    boolean estVide();
    boolean estPleine();
  }
```

On ne peut, à ce niveau savoir sur quel type d'élément de chaîne les méthodes vont agir. C'est pourquoi, le type imposé est `Object`, c'est à dire la classe racine dont toutes les autres héritent implicitement.

11.4.3 Implémentation de la pile

Le fichier suivant déclare et défini les classes `Element` et `PileChainee`. Le mot clé *implements* annonce que la classe `PileChainee` implémente les méthodes de l'interface dont le nom est ici `Pile`.

```
import java.lang.*;

class Element {
  public String contenu;
  public Element  suivant;
  public Element(String s, Element n) {
    contenu=s;
    suivant=n;
  }
}

public class PileChainee implements Pile {
  private Element tete;
  public PileChainee() { tete = null;}
  public void push(Object X) {tete = new Element((String)X, tete); }
  public Object pop() {
    Noeud X = tete; // sauve le "contenu"
    tete = tete.suivant; // avant de perdre l'"Element"
    return (String)X.contenu;
  }
  public Object top() { return (String)tete.contenu; }
  public boolean estVide(){ return tete==null;}
  public boolean estPleine() {return true; }
}
```

Application de la pile à la programmation d'une calculatrice. On désire réaliser un programme de calculatrice comparable (en plus modeste !) à celui que l'on rencontre sur certaines calculatrices de poche qui utilisent la notation dite *polonaise*.

La séquence de touche

```
+ 2 - 5 * 2 2
```

est analysée comme :

```
2 + ( 5 - ( 2 * 2)) = 3
```

ou, en notation fonctionnelle :

```
(+ 2 (- 5 (* 2 2)))
```

Le programme suivant réalise l'interprétation de l'opération demandée, en exploitant les propriétés de la pile.

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

public class PileChaineeT {

  static PileChainee stringVersPile(String[] args){
    PileChainee pc = new PileChainee();
    for (int i=args.length - 1; i>=0; i--) pc.push(args[i]);
    return pc;
  }

  static void affichePile(PileChainee pile) {
    if (pile.estVide()) return;
    String S = (String)pile.pop();
    System.out.print(S + " , ");
    affichePile(pile);
    pile.push(S);
  }


  static int evalExpr(PileChainee pc){
    String X = (String)pc.pop();
    if (X.equals("+")) return evalExpr(pc) + evalExpr(pc);
    else if (X.equals("-")) return evalExpr(pc) - evalExpr(pc);
    else if (X.equals("x")) return evalExpr(pc) * evalExpr(pc);
    else return Integer.parseInt(X);
  }

  static void  afficheOperation(PileChainee pc){
    if (pc.estVide()) return;
    String X = (String)pc.pop();
    String Y = null;
    if (X.equals("+") || X.equals("-") || X.equals("x")) {
      Y =(String)pc.pop();
      System.out.print(" ( "+ X + " " + Y);

    }
    else System.out.print(" "+ X+ " ) ");
    afficheOperation(pc);
    if (Y != null) pc.push(Y);
    pc.push(X);
  }

  public static void main(String[] arguments){
      PileChainee pileOp = new PileChainee();
      pileOp = stringVersPile(arguments) ;
```

```
            affichePile(pileOp); System.out.println();
            afficheOperation(pileOp);
            System.out.println("= " + evalExpr(pileOp));
            System.out.println("\nFin normale.") ;
        }
    }
```

Trace d'exécution

```
> java PileChaineeT +  1 x 3 -  2 + 8 9
+ , 1 , x , 3 , - , 2 , + , 8 , 9 ,
 ( + 1 ( x 3 ( - 2 ( + 8 9 ) = -44

Fin normale.
```

On pourra analyser les fonctions d'appoint du programme principal, en particulier celles d'entre elles qui sont récursives.

Chapitre 12

Les Threads

12.1 Les bases

Le support du *multithreading* en Java est associé à la notion de *thread*. Un *thread* est un flot d'exécution unique à l'intérieur d'un processus. Un processus est une unité d'exécution qui possède un domaine d'adressage qui lui est spécifique.

Le *thread* s'exécute à l'intérieur d'un processus[1], Il doit être géré par un processus parent.

La première mise en œuvre qui suit, montre les mécanismes de base pour créer un *thread* en Java .

Le premier fichier source est celui de la classe `tache` qui est le support d'exécution choisi pour les *threads*.

```
import java.io.*;

class tache implements Runnable {
  String Nom;
  static int Commun = 10;

  public tache(String nom){
    Nom = nom;
  }

  public void run() {
    int i=0;
    while (true) {
      Commun = Commun + 10;
      System.out.println(Nom + " : " + ++i + " --> Commun = " + Commun);
      try {Thread.sleep(1000);} catch (Exception e) {}
    }
  }
}
```

La variable `Commun` est commune à tous les *threads* qui utiliseront cette classe comme support.

[1] c'est à dire dans le contexte de ce processus : même allocation mémoire ...

La classe «lanceur» crée deux objets de type `classe` et trois *threads* à partir de ces deux classes.

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


public class tacheT {

  public static void main(String[] arg) {

    tache ta1 = new tache("T1");
    Thread t1 = new Thread(ta1);
    tache ta2 = new tache("T2");
    Thread t2 = new Thread(ta2);
    Thread t11 = new Thread(ta1);

    t1.start();
    t2.start();
    t11.start();
    try {Thread.sleep(5000);} catch (Exception e) {}
    t1.stop();
    t2.stop();
    t11.stop();
  }
}
```

Le lancement de l'application permet de mettre en évidence la différence entre :

1. variable de classe (`Commun`),
2. variable d'instance (`Nom`),
3. variable locale à une méthode (`i`).

En effet, la trace d'exécution donne :

```
> java tacheT
T1 : 1 --> Commun = 20
T2 : 1 --> Commun = 30
T1 : 1 --> Commun = 40
T1 : 2 --> Commun = 50
T2 : 2 --> Commun = 60
T1 : 2 --> Commun = 70
T1 : 3 --> Commun = 80
T2 : 3 --> Commun = 90
T1 : 3 --> Commun = 100
T1 : 4 --> Commun = 110
T2 : 4 --> Commun = 120
T1 : 4 --> Commun = 130
T1 : 5 --> Commun = 140
```

```
T2 : 5 --> Commun = 150
T1 : 5 --> Commun = 160
>
```

Remarques

- La classe `tache` doit déclarer l'utilisation de l'interface `Runnable`.
- Elle doit, pour faire quelque chose, implémenter la méthode `run`.
- Pour créer un *thread*, il faut au préalable lui créer son support d'exécution. Ici, c'est un objet de type `tache`.
- Des *threads* peuvent partager un même objet, (c'est à dire être créés à partir d'un identificateur unique comme t_1 et t_{11}), ou bien être construits à partir d'objets différents (comme t_1 et t_2).
- La classe `Thread` possède des méthodes de classe comme par exemple `sleep()`. En plus de permettre la création d'objets, elle fournit donc une bibliothèque pour le multitâche.
- Le programme lanceur peut jouer le rôle d'ordonnanceur de tâches vis à vis des *threads*. Ainsi, on peut les créer, les lancer, les stopper, ...

12.2 Solutions Java aux problèmes du parallélisme

12.2.1 La synchronisation

Le moyen prévu par `Java` pour traiter le problème de la synchronisation est la méthode `join()` qui permet d'attendre qu'un *thread* donné ait terminé son exécution. Pour le relancer, il faut le recréer. Il doit donc y avoir dans la classe qui le supporte une variable qui mémorise l'état de la tâche mise en œvre par les lancements successifs du *thread*.

L'exemple suivant montre la synchronisation d'un *thread* recréé deux fois. Le code du *thread* est écrit dans le fichier `tache1.java` :

```
import java.io.*;

class tache1 implements Runnable {
  String Nom;
  static int etape;

  public tache1(String nom){
    Nom = nom;
  }

  public void run() {
    switch (etape) {
    case 1 :
      System.out.println(Nom + " --> Traitement " + etape++);
      break;
    case 2:
      System.out.println(Nom + " --> Traitement " + etape++);
      break;
    default:
```

```
        System.out.println("Autre");
        break;
      }
    }

    static {
      etape = 1;
    }
  }
```

L'exécution est conditionnée par les valeurs successives que prend la variable `etape`.

La synchronisation est faite dans la fonction principale de la classe `tache1T` :

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


public class  tache1T {

  public static void main(String[] arg) {

    tache1 ta1 = new tache1("T1");
    Thread t1 = new Thread(ta1);

    int i=0;
    System.out.println("Main "+ ++i +" (Lancement de T1)");
    t1.start();
    try {t1.join();} catch (Exception e) {}
    System.out.println("Main " + ++i);
    t1 = new Thread(ta1);
    t1.start();
    try {t1.join();} catch (Exception e) {}
    System.out.println("Main " + ++i);
    t1 = new Thread(ta1);
    t1.start();
    try {t1.join();} catch (Exception e) {}
    System.out.println("Main " + ++i);
  }
}
```

Les trois `start` lancent le *thread* dans une nouvelle étape de son exécution. Le `join` fait que le programme principal «attend» que le *thread* ait terminé l'exécution de sa méthode `run`

La trace d'exécution donne l'image exacte du synchronisme ainsi réalisé.

```
> java tache1T
Main 1 (Lancement de T1)
T1 --> Traitement 1
Main 2
T1 --> Traitement 2
Main 3
```

```
Autre
Main 4
>
```

12.2.2 L'exclusion mutuelle

Lorsque plusieurs lignes de codes sont indissociables (constituant une «région critique», il est nécessaire de les protéger si elles peuvent être exécutées par plusieurs tâches.

Considérons par exemple le programme suivant, qui exécute en concurrence deux tâches t_1 et t_2 :

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


public class  tache2T {

  public static void main(String[] arg) {

    tache2 ta1 = new tache2("T1");
    tache2 ta2 = new tache2("T2");
    Thread t1 = new Thread(ta1);
    Thread t2 = new Thread(ta2);

    t1.start();
    t2.start();

    try {t1.join();} catch (Exception e) {}
    try {t2.join();} catch (Exception e) {}
  }
}
```

Les *threads* lancés exécutent le même code, limité à 4 affichages temporisés :

```
import java.io.*;

class tache2 implements Runnable {

  String Nom;

  tache2(String nom) {
    Nom = nom;
  }

  public void run() {
    for (int i=0; i<4; i++) {
      try {Thread.sleep(500);} catch (Exception e) {}
      System.out.println(Nom + " : "+ i);
    }
  }
}
```

L'exécution, est :

```
> java tache2T
T1 : 0
T2 : 0
T1 : 1
T2 : 1
T1 : 2
T2 : 2
T1 : 3
T2 : 3
>
```

Il n'y a pas ici, d'exclusion sur la boucle `for`. Pour qu'il en soit ainsi, il est nécessaire de rendre le code de la fonction `run()` soit rendu insécable. Ceci est réalisable en Java en insérant le mot clé `synchronised`. Le code de la classe `tache2` devient alors :

```
import java.io.*;

class tache2 implements Runnable {

  String Nom;

  tache2(String nom) {
    Nom = nom;
  }

  public static synchronized void regionCritique(String S){
    for (int i=0; i<4; i++) {
      try {Thread.sleep(500);} catch (Exception e) {}
      System.out.println(S + " : "+ i);
    }
  }

  public void run() {
    regionCritique(Nom);
  }
}
```

et la trace d'exécution ...

```
> java tache2T
T1 : 0
T1 : 1
T1 : 2
T1 : 3
T2 : 0
T2 : 1
T2 : 2
T2 : 3
>
```

La région critique a bien été respectée.

Remarque : Pour que la région critique soit effectivement prise en compte, il y a en fait deux possibilités :

1. les *threads* sont créés à partir d'un objet unique, et la région critique est une méthode `synchronized` quelconque (`static` ou non).
2. les *threads* sont créés à partir d'objets différents, et la région critique est un méthode de classe (déclarée `static`).

Chapitre 13

Méthodes natives

13.1 Généralités

L'objectif de ce chapitre est de montrer comment, à partir du langage `Java` , accéder à des fonctions écrites en langage `C`.

Les motivations pour cela peuvent être diverses :

- accès au matériel, ce qui n'est pas directement autorisé par le langage `Java` ,
- appel de fonctions d'une librairie existante
- ...

Précaution. L'écriture d'une méthode native détourne évidemment tout l'aspect sécurisé de la programmation en `Java` au bénéfice de la programmation classique. Aussi, il conviendra de réserver l'usage de méthodes natives pour les cas d'absolue nécessité.

Une approche convenable de l'interfaçage des deux langages doit permettre de répondre au moins aux questions suivantes :

- Comment dans une fonction `C` récupérer un paramètre passé dans la méthode correspondante `Java` ?
- Comment retourner une valeur du `C` vers la méthode `Java` ?

13.2 Mise en œuvre

On décide d'interfacer avec `Java` les deux fonctions suivantes (fichier *fonctions.h*)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* Transforme la chaîne reçue en majuscule */
void majuscule(char *str){
    int N = strlen(str);
    int i;
    for (i=0; i<N; i++) str[i] = toupper(str[i]);
```

```
}

/* Renvoie un réel aléatoire de l'intervalle [n, n+1] */
float aleatoire(int n){
    long int r = random();
    float X = (float)r/(float)RAND_MAX;
    return (float)n+X;
}
```

13.2.1 Définition de la classe «relais»

Du point de vue du langage `Java` , il est nécessaire que le logiciel «étranger» (écrit en langage `C`) soit vu à partir d'une classe ordinaire. Le mot clé `native` indique que le code d'une méthode est extérieur à la classe.

```
class essai {
    /* Attributs */
    private String nom;
    private float valeur;

    /* méthodes natives */
    private native float alea(int n);
    private native void majuscule(byte[] b);

    private byte[] string2byte(String S){
        int i;
        byte[] retour = new byte[S.length() + 1];
        for (i=0; i<S.length(); i++){
            retour[i] = (byte)S.charAt(i);
        }
        retour[i]=0;
        return retour;
    }

    /* constructeur */
    public essai(String S, int I) {
        nom = S;
        valeur = /*(float)I; */ alea(I);
    }

    /* interface */
    public void transforme() {
        byte[] b = string2byte(nom);
        majuscule(b);
        nom = new String(b);
    }

    public void affiche() {
        System.out.print("Nom = " + nom
                         + "  Valeur = " + valeur);
    }
```

```
    /* exécuté au départ */
    static {
        System.loadLibrary("fonctions");
    }
}
```

Remarques :

1. Le prototype standard de chaque méthode native est précédé du mot clé `native`.
2. Une méthode native est simplement déclarée dans le fichier `Java` de la classe à laquelle elle appartient.
3. La classe possédant des méthodes natives possède des lignes de code dites «de classe» (déclarées `static`). Elles sont exécutées une seule fois à la première référence à la classe `essai`.
4. L'instruction `System.loadLibrary("fonctions");` est naturellement le chargement d'une librairie dynamique. Sous UNIX, ceci correspondra à une librairie appelée `libfonctions.so`. Une erreur courante est l'absence de cette librairie dans le chemin reconnu par le système pour les librairies dynamiques[1].

13.2.2 Définition des méthodes natives

Les méthodes natives ont été déclarées mais n'ont pas été définies. Elles le sont dans un fichier en langage C que nous appellerons

`essai.c`

Le choix du nom n'a aucune importance (à la différence des fichiers `Java`).

Le prototype des fonctions natives à écrire est produit par l'outil `javah` de JDK. Dans notre cas, les lignes de commande à exécuter sont :

```
> javac essai.java
> javah -jni essai
```

La première ligne produit le fichier `essai.class`
La seconde, produit un fichier `essai.h` à partir de `essai.class`.

Le fichier `essai.h` contient les prototypes des fonctions C qui devront être écrites. Il ne doit pas être modifié. Pour gagner du temps, on peut cependant le recopier vers le fichier `essai.c`[2].

```
> cp essai.h essai.c
```

Pour illustrer la démarche de développement, prenons le cas de la méthode native `alea`. Le fichier `essai.h` contient les lignes suivantes (extrait) :

---

[1] Ce chemin est différent du *PATH* pour la recherche des exécutables.

[2] Compte tenu de la complexité syntaxique des prototypes, cette opération est même recommandée...

```
/*
 * Class:     essai
 * Method:    alea
 * Signature: (I)F
 */
JNIEXPORT jfloat JNICALL Java_essai_alea
  (JNIEnv *, jobject, jint);
```

À partir de ces lignes – que nous retrouvons dans `essai.c` après la copie – nous pouvons écrire le code de la méthode native :

```
#include <jni.h>

/* Déclarations des fonctions externes
   (important pour éviter de grosses erreurs d'exécution)
*/

extern float aleatoire(int);
extern void majuscule(char*);

/*
 * Class:     essai
 * Method:    alea
 * Signature: (I)F
 */
JNIEXPORT jfloat JNICALL Java_essai_alea
(JNIEnv *e, jobject this, jint n) {
  return (jfloat) aleatoire((int)n);
}

/*
 * Class:     essai
 * Method:    majuscule
 * Signature: ([B)V
 */
JNIEXPORT void JNICALL Java_essai_majuscule
(JNIEnv *e, jobject this, jbyteArray buf){
  jbyte *jbuf = (*e)->GetByteArrayElements(e, buf, NULL);
  majuscule( (char*)jbuf );
  (*e)->ReleaseByteArrayElements(e,buf,jbuf,0);
}
```

Les paramètres `JNIEnv *e` et `jobject this` permettent la communication entre le monde natif et le monde `Java` (accès à des classes ou à des variables du programme `Java`).

Le type *jint* est une image pour l'implémentation du compilateur C du type *int* de `Java` . En effet, `Java` normalise les types de base pour toutes les machines alors que pour la plupart des autres langages (dont le C) la taille d'un type donné dépend de l'architecture de la machine considérée.

13.2.3 Librairie dynamique

La librairie dynamique est obtenue à partir des sources : `fonctions.c` et `essai.c`.

Compilations :

```
gcc -c fonctions.c

gcc -I/usr/lib/jdk1.1.7/include\
            -I/usr/lib/jdk1.1.7/include/genunix -c essai.c
```

Les options `-I` servent à indiquer au compilateur les endroits où se trouvent les fichiers *header* (*.h*) de JDK. Le résultat de cette compilation est un fichier `essai.o` et un fichier `fonctions.o`

Édition de liens

```
gcc -shared -Wl,-soname,libfonctions.so\
            -o libfonctions.so essai.o fonctions.o
\
```

L'option `-Wl` permet de passer des arguments à l'éditeur de liens. Il reçoit pour consigne de fabriquer un fichier *.so*. L'option `-o` permet de fixer le nom du fichier de sortie. Le nom stipulé dans la classe Java `essai.class` étant `fonctions`, il faut ici donner comme nom : `'lib'` + `'fonctions'` + `'.so'`.

13.2.4 Programme principal et exécution

```
import java.io.*;


class essaiT {

    public static void main(String[] args){
        essai E = new essai("langage", 9);
        System.out.println("   Avant l'appel :");
        E.affiche();

        E.transforme();
```

```
            System.out.println("\n   Après l'appel :");
            E.affiche();
            System.out.println("\nbye...");
        }
    }
```

13.3 Bilan des phases de fabrication

L'ensemble des opérations de conception sont résumées dans le schéma de la figure 13.1.

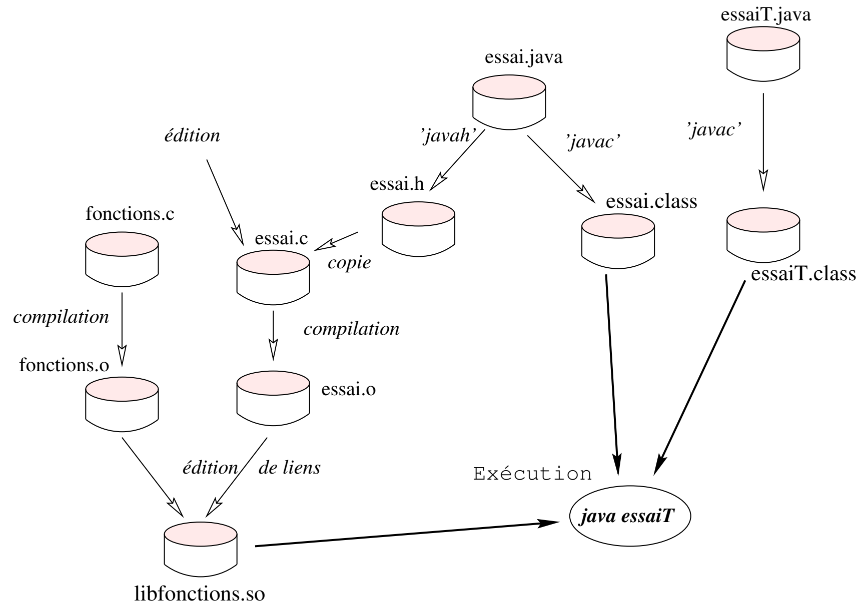

FIG. 13.1 – *Bilan des développements du programme en C et du programme Java.*

Index

étiquettes, 74

ActionListener, 75
actionPerformed, 75
addLayout, 78
applet java, 12
appletviewer, 8
application java, 10
arguments, 11

bloc d'instructions, 20
boutons, 74
boutons radio, 74
BufferedReader, 51
Button, 74

Canvas, 79
cases à cocher, 74
catch, 43
Checkbox, 74
CheckboxGroup, 74
Choice, 75
choix, 24, 75
  multiple, 25
  simple, 24
  simple répété, 25
classe, 27
  abstraite, 87
classe terminal, 51
CLASSPATH, 85
clavier, 51, 72
client, 60
constantes, 14
constructeur, 35

DataInputStream, 50
default, 25
do ...while, 22

exception, 42
exclusion, 98

Fenêtre principale d'application, 79
fichier
  écriture, 56
  lecture, 49
fichiers, 49
FileInputStream, 49
FileOutputSream, 56
FileReader, 54
final, 14
FlowLayout, 78
for (... ; ... ; ...), 23
Frame, 79

goto, 7

héritage, 38
HTML, 12

if ...else, 24
import, 84
instanciation, 36
interface, 87
isAltDown, 74
isControlDown, 74
isMetaDown, 74

java (commande), 8
javac, 10
javah, 8
javap, 8

KeyListener, 72
keyPressed, 72
keyReleased, 72
keyTyped, 72

Label, 74
List, 75

listes de défilement, 75
lvalue, 16

méthodes natives, 101
machine virtuelle, 5
méthodes, 37
méthode statique, 37
`mouseClicked`, 69
`mouseDragged`, 69
`mouseEntered`, 69
`mouseExited`, 69
`MouseMotionListener`, 69
`mouseMoved`, 69
`mousePressed`, 69
`mouseReleased`, 69

`native`, 102
`new`, 27

`Oak`, 4
opérateur
    conditionnel, 19
opérateurs, 15
    arithmétiques, 15
    d'affectation, 16
    d'incrémentation, 17
    de décrémentation, 17
    logiques, 18
    relationnels, 18
    sur les bits, 15

package, 84
pile, 91
polymorphisme, 28
`private`, 28
`public`, 28

répétitions, 21
réseau, 58

`ServerSocket`, 58
serveur, 58
`Socket`, 60
souris, 68
star7, 3
`static`, 36
`StreamTokenizer`, 54
structures de commandes, 20
`switch ...case`, 25
synchronisation, 96
System.in, 51

tableaux, 40
`terminal`, 51
`TextField`, 75
`Thread`, 94
`throw`, 43
`throws`, 45
`try`, 44
`typedef`, 7
types, 13

variable
    d'instance, 36
    de classe, 36

`while`, 21

zones de textes, 75