www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Le langage Java

Touraivane
Tél : (33) 4 91 8 85 38
Fax : (33) 4 91 8 85 18 - (33) 4 91 8 85 10
Touraivane@gbm.esil.univ-mrs.fr
http ://gbm.esil.univ-mrs.fr/˜tourai

Le langage Java

Version V.0

Mise à jour : 5 juin 1998

ESIL
Ecole Supérieure d'Ingénieurs de Luminy
Département Génie Bio Médical

Table des matières

I Java : Le langage 9

1 Des procédures aux objets 15

2 C'est quoi Java ? 21

3 Eléments de base 23

4 Opérations et expressions 29

5 Les structures de contrôle 39

6 Classes et Objets 45

7 Héritage 55

8 Les interfaces 63

9 Packages 67

10 Exceptions 73

11 Tableaux et chaînes de caractères 79

12 Conversions et promotions 87

13 Classes imbriquées 91

14 La hiérarchie des classes 95

15 Entrées-Sorties 97

16 Serialization 117

17 Compression des données 123

18 Le package java.util 125

19 Les types 133

20 Programmation système 141

21 Programmation dynamique 147

22 Threads 155

II Java : Programmation graphique 171

23 Applets et applications autonomes 175

24 Découvrir la programmation graphique 185

25 Gestion des évènements 191

26 Les widgets **215**

27 Ranger les widgets **247**

28 Dessiner sur une fenêtre graphique **263**

29 Couleurs et Fontes **271**

30 Images **277**

31 Le son **289**

III Java : JFC 291

32 Java Foundation Class **295**

33 Les composants Swing **305**

34 Gestion des évènements swing **307**

35 Transfert de données **309**

36 Java2D **317**

37 Java3D **319**

38 Quelques classes Swing **321**

IV Java : Programmation avancée 331

39 Programmation réseau **335**

40 Appel de méthodes distantes (RMI) **353**

41 Introduction aux bases de données **357**

42 Introduction à JDBC **359**

43 Code natif (JNI) **365**

44 Corba **379**

V Java : Beans 381

45 Introduction à Java Beans **385**

VI Java : Annexes 387

46 javadoc **391**

47 Le langage SQL **395**

48 Fromat jar et zip **397**

VII Java : Travaux dirigés 399

VIII Java : Corrigés 421

IX Java : Travaux pratiques 461

X Java : Corrigés des travaux pratiques 471

Avant propos

Ce document est une série de notes de cours sur le langage *Java*. Il s'adresse aux étdudiants ayant une bonne pratique d'un langage de programmation procédurale. La connaissance d'un langage orienté objet n'est pas nécessaire.

Pour réaliser ces notes de cours, je me suis basé sur les documents suivants :

- Spécification du langage Java (voir 39.3)
- la documentation de jdk (voir 39.3) qui contiennent les descriptions des API et de nombreuses spécifications
- le tutorial de jdk (voir 39.3)
- le langage Java (voir 39.3)
- Java in a NutShell (voir 39.3)
- Java distributed Computing (voir 39.3)
- Exploring Java (voir 39.3)
- JDBC (voir 39.3)
- certains article de JavaWorld (voir 39.3) et Java Connection (voir 39.3)

La présentation de ces notes de cours occupent environ 100 heures de cours/TD/TP :

Le langage	30 heures de cours et 20 heures de TD
La programmation graphique	30 heures de TP
La programmation avancée	15 heures de TP

Première partie

Java : Le langage

Table des Matières

1 Des procédures aux objets . . . 15
1.1 La programmation procédurale . . . 15
1.2 Programmation modulaire . . . 16
1.3 Flexibilité . . . 16
1.4 Abstraction de données . . . 17
1.5 Les concepts de la programmation par objets . . . 18
2 C'est quoi Java ? . . . 21
2.1 Le langage Java est simple, familier et orienté objet . . . 21
2.2 Le langage Java est distribué . . . 22
2.3 Le langage Java est interprété . . . 22
2.4 Le langage Java est robuste et sûr . . . 22
2.5 Le langage Java est portable et indépendant des plates-formes : . . . 22
2.6 Le langage Java est dynamique et multithread . . . 22
3 Eléments de base . . . 23
3.1 Le jeu de caractères Unicode . . . 23
3.2 Les commentaires . . . 23
3.3 Les identificateurs . . . 24
3.4 Les types de données élémentaires . . . 24
3.5 Les constantes littérales . . . 25
3.6 Les variables . . . 26
4 Opérations et expressions . . . 29
4.1 Généralités sur les expressions . . . 29
4.2 Affectation . . . 30
4.3 Expressions arithmétiques . . . 31
4.4 Expressions de comparaison . . . 33
4.5 Concaténation des chaînes de caractères . . . 33
4.6 Expressions logiques . . . 33
4.7 Manipulation de bits . . . 34
4.8 Autres opérateurs binaires d'affectation . . . 35
4.9 Expression conditionnelle . . . 35
4.10 Changement de type . . . 35
4.11 Création des objets . . . 36
4.12 Récapitulatif . . . 36
5 Les structures de contrôle . . . 39
5.1 Instructions et blocs . . . 39
5.2 Instruction conditionnelle (if) . . . 39
5.3 Etude de cas (switch) . . . 40
5.4 Itérations . . . 41

5.5 Etiquette, break, continue et return . . . 42
5.6 Structure d'un programme autonome Java . . . 43
6 Classes et Objets . . . 45
6.1 Déclaration des classes . . . 45
6.2 Définitions de champs . . . 48
6.3 Définitions de méthodes . . . 50
6.4 Bloc d'initialisation static . . . 52
6.5 Conversion des types primitifs en objets et inversement . . . 52
7 Héritage . . . 55
7.1 Introduction . . . 55
7.2 Constructeur de la sous classe . . . 56
7.3 Redéfinition des méthodes . . . 57
7.4 Destructeurs . . . 58
7.5 Méthodes et classes finales . . . 59
7.6 Conversion entre classes et sous classes . . . 59
7.7 Classes et méthodes abstraites . . . 59
7.8 La classe Object . . . 60
8 Les interfaces . . . 63
8.1 Déclaration des interfaces . . . 63
8.2 Implanter des interfaces . . . 64
8.3 Utiliser des interfaces . . . 64
9 Packages . . . 67
9.1 Importer des packages . . . 67
9.2 Accessibilité . . . 69
9.3 Convention de nommage de packages . . . 70
10 Exceptions . . . 73
10.1 Qu'est-ce qu'une exception . . . 75
10.2 Définir de nouveaux types d'exception . . . 75
10.3 Déclencher des exceptions . . . 75
10.4 Capturer les exceptions . . . 76
10.5 Les classes Error, Exception et RuntimeException . . . 77
11 Tableaux et chaînes de caractères . . . 79
11.1 Tableaux . . . 79
11.2 Chaînes de caractères . . . 80
12 Conversions et promotions . . . 87
12.1 Type de conversion . . . 87
12.2 Affectation . . . 89
12.3 Invocation de méthodes . . . 89
12.4 Chaînes de caractères . . . 89
12.5 Changement de type . . . 89
12.6 Promotion numérique . . . 90
13 Classes imbriquées . . . 91
13.1 Nested class et Inner class . . . 91
13.2 Un exemple . . . 91
13.3 Classes locales . . . 93
13.4 Classes anonymes . . . 93

13.5 Classes imbriquées final, private, protected, ou static . . . 94
13.6 Classes imbriquées et accessibilité . . . 94
14 La hiérarchie des classes . . . 95
15 Entrées-Sorties . . . 97
15.1 Introduction . . . 97
15.2 Hiérarchie des classes et interfaces de java.io . . . 100
15.3 Les entrées/sorties standard (terminal) . . . 104
15.4 Les entrées/sorties structurées . . . 104
15.5 Les tableaux et les chaînes . . . 109
15.6 Les fichiers . . . 110
15.7 Les tubes (pipes) . . . 113
15.8 StreamTokenizer . . . 114
15.9 Conversion des types de flots . . . 116
15.10Séquence de flots . . . 116
16 Serialization . . . 117
16.1 Lire et écrire des objets . . . 117
16.2 Contrôle de la "serialization" . . . 118
16.3 Compression . . . 121
16.4 L'interface Externalizable . . . 121
16.5 Gestion des versions . . . 122
17 Compression des données . . . 123
18 Le package java.util . . . 125
18.1 L'interface Enumeration . . . 126
18.2 La classe Vector . . . 126
18.3 La classe Stack . . . 127
18.4 La classe Dictionary . . . 128
18.5 La classe Hashtable . . . 128
18.6 La classe BitSet . . . 129
18.7 La classe StringTokenizer . . . 129
18.8 La classe Random . . . 130
18.9 La classe Date . . . 130
18.10Le package java.util jdk 1.2 . . . 130
19 Les types . . . 133
19.1 Le package java.lang . . . 133
19.2 Classes et types primitifs . . . 134
19.3 La classe java.lang.Boolean . . . 134
19.4 La classe java.lang.Number . . . 135
19.5 La classe java.lang.Integer . . . 135
19.6 La classe java.lang.Byte . . . 135
19.7 La classe java.lang.Short . . . 136
19.8 La classe java.lang.Long . . . 136
19.9 La classe java.lang.Float . . . 137
19.10La classe java.lang.Double . . . 137
19.11La classe java.lang.Character . . . 138
19.12La classe java.lang.Math . . . 138
19.13Jdk 1.2beta3 . . . 139

20 Programmation système 141
20.1 La classe java.lang.System 141
20.2 La classe java.lang.Runtime 143
20.3 La classe java.lang.Process 144
20.4 La classe java.lang.SecurityManager 144
20.5 La classe java.lang.ClassLoader 146
20.6 La classe java.lang.Compiler 146
21 Programmation dynamique 147
21.1 Ausculter une classe 147
21.2 Manipuler dynamiquement des objets 150
21.3 Jdk 1.2 154
22 Threads 155
22.1 Introduction 155
22.2 La classe thread et l'interface Runnable 156
22.3 Gestion des threads 159
22.4 Priorité des threads 167
22.5 Interblocages 168
22.6 Démons 168
22.7 La classe Thread 169
22.8 La classe ThreadGroup 169

1. Des procédures aux objets

Sommaire

1.1 La programmation procédurale 15
1.2 Programmation modulaire 16
1.3 Flexibilité 16
1.4 Abstraction de données 17
1.5 Les concepts de la programmation par objets 18
1.5.1 Qu'est-ce qu'un objet ? 18
1.5.2 Qu'est-ce qu'un message ? 18
1.5.3 Qu'est-ce qu'une classe ? 19
1.5.4 Qu'est ce que l'héritage ? 19

La programmation par objets est une technique de programmation. Un langage de programmation qui supporte un style de programmation particulier fournit des facilités qui permettent d'utiliser ce style de manière aisée ainsi que des garde-fous nécessaires au contrôle du style de programmation. Ainsi, *Java* est un langage adapté à la programmation objet, même si (au prix d'un plus gros effort), les langages procéduraux comme C ou Pascal permettent également d'utiliser ce style de programmation.

1.1 La programmation procédurale

La programmation procédurale est celle que vous connaissez. Dans ce style de programmation, l'accent est mis sur l'exécution du programme. Les facilités de base offertes par les langages qui supportent ce style de programmation sont les fonctions qui admettent des arguments et qui renvoient des valeurs. Comme exemple, considérons la fonction qui calcule le *pgcd* de deux entiers positifs

```
long pgcd(long x, long y) {
    unsigned long r ;
    if (x < y) { r = x ; x = y ; y = x ; }
    do {
        r = x % y     // r est le reste de la division entière de x et y
        x = y ;
        y = r
    }                 // réitérer ce processus en échangeant (x, y) par (y, r)
    while (r != 0) ;  // tant que le reste de la division entière x et y est non nul
    return x ;        // retourne le pgcd
}
```

Une fois cette fonction donnée, il est possible à présent d'utiliser cette fonction dans toute autre fonction.

```
void une_fonction(void) {
    ...
    x = pgcd(4356, y) ;
    ...
}
```

1.2 Programmation modulaire

Avec l'augmentation de la taille des programmes, la programmation procédurale atteint ses limites ; la programmation modulaire voit alors le jour. Ce style de programmation utilise les principes d'encapsulation des données. L'ensemble des procédures ou fonctions dépendantes et les données qu'elles manipulent sont regroupés dans un module. Un programme est alors constitué de plusieurs modules et la communication inter-modules se fait à travers une interface ; les détails d'implantation de chaque module sont cachés aux autres modules. Imaginons qu'il faille réaliser un programme dans lequel on utilise des piles. Un des modules de ce programme se charge de l'implantation d'une pile et les détails de cette implantation devant être ignorés par les autres modules. L'interface que fournit ce module est constituée d'un fichier (par exemple " pile.h ")

```
// Interface du module PILE de caractères (pile.h)
void empiler(char) ;
char depiler(void) ;
// Fin de l'interface
```

Les utilisateurs de ce module ignorent parfaitement les détails d'implantation de cette pile. En particulier, s'agit-il d'un tableau ou d'une liste chaînée ? Seul le programmeur de ce module le sait. L'utilisation de ce module par d'autres se fait alors de la manière suivante :

```
// Module XXX utilisant une pile de caractère
#include "pile.h"
void une_fonction() {
        short c1, c2 ;
        ...
        empiler('a') ;
        empiler('b') ;
        ...
        c1 = depiler() ;
        if (c1 == ERREUR) erreur(" la pile est vide ") ;
        c2 = depiler() ;
        if (c2 == ERREUR) erreur(" la pile est vide ") ;
        ...
 }
// Fin du module XXX
```

Cette première tentative de *modularité* ne permet l'utilisation que d'une seule pile. Il est souvent nécessaire de disposer de plusieurs piles dans un même programme. La solution consiste à fournir une interface plus sophistiquée dans laquelle un nouveau type de donnée appelé **id_de_pile** sera défini.

```
// Interface du module PILE de caractères
typedef id_de_pile ...
id_de_pile creer_pile() ;
void empiler(id_de_pile, char) ;
char depiler(id_de_pile) ;
void detruire_pile(id_de_pile) ;
// Fin de l'interface
```

Idéalement, on aurait souhaiter que le type abstrait **id_de_pile** se comporte comme un type prédéfini. Or, cela n'est pas le cas : il appartient à l'utilisateur de ce module de s'assurer de la bonne utilisation. En particulier, celui-ci devra allouer les variables nécessaires pour manipuler ces piles, s'assurer qu'elles sont désallouées. Il n'existe pas de convention claire pour les passages des arguments de type définis par un module particulier. Bref, on ne dispose pas d'aide à la programmation.

```
// Module XXX utilisant une pile de caractère
#include "pile.h"
void une_fonction(void) {
        id_de_pile p1, p2 ;
        char c1, c2 ;
        ...
        p1 = creer_pile() ;             // p2 non créée
        empiler(p1, 'a') ;
        c1 = depiler(p1) ;
        if (c1 == EOF)
                erreur(" la pile est vide ") ;
        detruire_pile(p2) ;
        p1 = p2 ;                                       // p2 utilisée après sa destruction
                                                                // p1 non détruit
        ...
}
// Fin du module XXX
```

1.3 Flexibilité

Une fois un type de donnée abstrait défini, il interfère très peu avec le reste de programme ; pourtant toute modification ou enrichissement de ce type de donnée entraîne une refonte complète du programme. Imaginons que l'on définit un type de donnée forme.

```
enum type {cercle, triangle, carree} ;
typedef struct forme {
        point centre ;
        type t ;
} forme ;
```

Le programmeur devra connaître toutes les formes manipulées pour implanter la fonction dessiner :

```
void dessiner(forme f) {
        switch(t.type) {
                case cercle :
                        ...
                        break ;
                case triangle :
                        ...
                        break ;
                case carree :
                        ...
                        break ;
        }
}
```

Ainsi, la suppression d'une forme ou l'ajout d'une nouvelle forme force le programmeur à reprendre l'ensemble des fonctions et à les adapter. Programmation par objets : l'héritage Le mécanisme d'héritage de la programmation objet apporte une solution élégante au problème soulevé dans la section précédante. On définit tout d'abord une classe qui possède les propriétés générales de toutes les formes :

```
class forme {
        private point centre ;
                ...
        public point position() { return center ; }
        public void deplacer(point vers) { center = vers ; dessiner() ; }
        public abstract void dessiner() ;
        public abstract void tourner(int) ;
                ...
}
```

Les fonctions dont l'implantation (ou la mise en oeuvre) est spécifique pour chaque forme sont marquées par le mot clé `abstract`. Pour définir effectivement ces fonctions virtuelles, on commencera par définir une classe dérivée de la classe forme :

```
class cercle extends forme {
        private int rayon ;
        public void dessiner() { ... }
        public void tourner(int) { }
}
```

1.4 Abstraction de données

Le support pour la programmation par abstraction de données consiste à fournir des facilités pour définir des opérations pour un type défini par l'utilisateur. Contrairement aux types prédéfinis, aucun système ne peut définir, par défaut, les fonctions d'initialisation et de destruction des types définis par le programmeur ; il lui appartient donc de fournir ces fonctions.

Considérons un type nouvellement défini (que l'on appellera tableau) qui est constitué de sa taille et d'un pointeur vers les éléments du tableau. La création d'un objet de type tableau se fait en définissant une variable de la classe tableau. Que doit-on créer à l'initialisation ? Faut-il allouer la place nécessaire pour coder le tableau ? etc. Les réponses à ces questions ne peuvent être fournies que par l'utilisateur ayant défini le type tableau.

Une première solution consiste à se définir une fonction `init` que l'utilisateur se forcera à appeler avant toute utilisation d'un objet d'un type utilisateur.

```
class tableau { private int [] t ; public void init(int taille) ;
        ...
} ;

void une_fonction() {
tableau t ;
        ...
        t.init(2) ;     // n'utiliser t qu'après son initialisation
}
```

Cette solution est peu agréable et *Java* fournit un mécanisme plus astucieux pour faire l'initialisation : l'appel de la fonction d'initialisation se fait automatiquement à la définition de la variable. La fonction d'initialisation se nomme *constructeur*. Par contre, contrairement aux langages de programmation objet habituels, il n'est pas nécessaire de fournir un *destructeur* pour les objets d'un type défini par l'utilisateur. Cette destruction se fera automatiquement et une récupération mémoire se charge de restituer au système la place mémoire rendue libre.

```
class tableau {
        private int [] t ;
        tableau(int s) {
                if (s<=0) erreur(...) ;
                t = new int[n] ;
        } ;
}
```

1.5 Les concepts de la programmation par objets

Les concepts clés de la programmation orientée objets sont : les objets qui sont des collections d'attributs et méthodes associées. les messages qui servent de moyen de communication entre objets les classes qui servent à définir des modèles d'objets. l'héritage qui fournit un mécanisme puissant et naturel pour organiser et structurer les programmes.

1.5.1 Qu'est-ce qu'un objet ?

Comme leur nom l'indique, les objets sont au c.ur de la programmation par objet. Dans la vie courante, le monde réel qui nous entoure est constitué d'objets. Ces objets possèdent deux caractéristiques : ils sont dans un certain état et possède un comportement.

Les objets dont nous allons étudier sont similaires aux objets du monde réel qui nous entoure en ce sens, qu'eux également, sont dans un certain état et possèdent un certain nombre de comportements. L'état d'un objet est représenté par des attributs et les comportements sont définis par des méthodes.

Un objet est une collection d'attributs munis de méthodes

On peut représenter les objets du monde réel par des objets informatiques comme c'est le cas dans un programme d'animation. On peut également représenter des concepts abstraits ; une manipulation graphique est un objet dans un environnement graphique.

Un objet qui modélise un objet du monde réel (un vélo, par exemple) possède des attributs qui indiquent l'état de celui-ci (la vitesse, la cadence de pédalage etc.). Le comportement de cet objet est constitué par des méthodes (accélération, freinage, changement de vitesse, etc.). Les seules actions possibles sur les objets sont celles définies par les méthodes. Les changements d'état (modification des variables d'attributs) ne sont permis qu'à travers les méthodes. Cette manière de protéger les attributs par des méthodes s'appelle encapsulation.

L'encapsulation des données a pour objectif de cacher les détails d'implantation d'un objet pour les autres objets. Pour reprendre l'exemple du vélo, lorsqu'on change de vitesse, il n'est nullement nécessaire de connaître le mécanisme du levier de vitesse ; la seule chose qui nous intéresse, c'est de savoir comment changer de vitesse. De même, il n'est pas utile de connaître comme un objet ou une classe d'objets est implanté, il nous suffit juste de savoir quelles méthodes utiliser.

Les avantages de l'encapsulation Les avantages de l'encapsulation sont de deux sortes :

- Modularité. L'implantation d'un objet peut être réalisée et maintenue indépendamment de la réalisation et de l'implantation des autres objets.
- Masquage de l'information. Les objets communiquent entre eux à travers une interface publique. Mais chaque objet peut conserver des caractéristiques privées (attributs ou méthodes) qui peuvent être changées à tout moment sans perturber les autres objets.

Quoique idéale, le concept d'objets dont les attributs ne sont accessibles qu'à travers les méthodes est un concept beaucoup trop rigide. Pour des raisons d'efficacité, les objets informatiques ne sont composés uniquement d'attributs accessibles par d'autres objets à travers des méthodes (attributs privés). Certains attributs pourront être accédés directement par les autres objets.

Dans beaucoup de langages objets, un objet peut décider de rendre publics (donc accessibles sans passer par des méthodes) certains de ses attributs. Ces attributs peuvent alors être "visibles" de l'extérieur et (plus grave) peuvent être modifiés.

A l'opposé, toutes les méthodes ne sont pas forcément utilisables par les autres objets. Un objet peut choisir de cacher certaines méthodes i.e. les rendre privées. Bref, dans les langages de type C++, les objets peuvent rendre publiques des attributs et rendre privées des méthodes.

1.5.2 Qu'est-ce qu'un message ?

Un objet isolé dans l'univers n'a que peu d'intérêt ; il ne devient intéressant que lorsqu'il réagit avec d'autres objets. Il en est de même pour les objets informatiques qui communiquent entre eux en s'envoyant des messages. Lorsqu'un objet A veut faire effectuer à l'objet B une de ses méthodes (méthodes de B), il envoie à B un message.

Certains messages doivent contenir des informations complémentaires pour réaliser la tâche demandée. Le message changer de vitesse à notre fameux vélo doit s'accompagner de l'information lui spécifiant s'il faut passer à la vitesse supérieure ou inférieure. Ces informations complémentaires accompagnent un message grâce aux paramètres de la méthode.

Un message est donc constitué de trois parties :

- l'objet auquel on s'adresse
- le nom de la méthode à réaliser
- les éventuels paramètres nécessaires à la méthode.
- les pré-conditions et les post-conditions.

1.5.3 Qu'est-ce qu'une classe ?

Dans la vie courante, il existe plusieurs objets de même nature ; votre vélo n'est qu'un vélo parmi tant d'autres. Dans la terminologie de la programmation orientée objet, on dira que votre vélo est une instance de la classe des objets connus sous le nom de vélo. Tous les vélos ont des caractéristiques communes mais peuvent être dans des états différents.

De même, dans la programmation orientée objet, on pourra avoir plusieurs objets de même nature, un moule commun permettant de tirer partie des caractéristiques communes des ces objets. Ce moule est appelé classe.

Une classe est un prototype qui définit des attributs et des méthodes communes à tous les objets d'une certaine nature.

Les valeurs des attributs d'instances (attributs d'une instance d'une classe d'objets) sont distinguées. Ainsi, une fois la classe créée, il faut créer une instance de cette classe pour en faire un objet effectivement utilisable. Lors de la création d'une instance d'une certaine classe, on crée un objet de la nature spécifiée et le système alloue de la mémoire pour les attributs d'instances définis dans la classe. C'est alors que l'on pourra faire appel aux méthodes de cet objet pour réaliser quelque chose.

Contrairement aux attributs d'instance, les objets d'une même classe utilisent la même méthode d'instance. Il n'y a pas un exemplaire de chaque méthode pour chaque instance d'une classe.

Outre les attributs d'instance et des méthodes, les classes peuvent également définir des attributs de classes et de méthodes de classes. Ces attributs et méthodes de classes peuvent être accessibles soit par l'intermédiaire d'une instance de classe, soit directement par l'intermédiaire de la classe elle-même (il n'est pas nécessaire de disposer d'une instance d'une classe pour utiliser des attributs et méthodes de classes).

Le système ne crée qu'un seul exemplaire des attributs de classe et toutes les instances d'une même classe partagent ces attributs. Pour en revenir à notre fameux vélo, supposons que tous les vélos aient un même nombre de vitesse. Il serait alors inefficace de définir un attribut par instance de vélo codant cette information sur le nombre de vitesses (chaque instance disposerait d'une copie de cette variable d'attribut). Dans de ce cas, il serait naturel (et surtout plus efficace) de définir un attribut de classe qui contient ce nombre de vitesses ; toutes les instances partageant la même variable. Une modification par une instance de cette classe d'un attribut de classe, entraînerait la modification pour toutes les autres instances de cette classe.

L'intérêt des classes. Comme nous l'avons dit plus haut, les objets permettent une programmation modulaire et le masquage d'informations. Les classes permettent la réutilisabilité. Une même classe (pour peu qu'elle soit bien conçue) sera utilisée très souvent par d'autres objets, dans d'autres programmes.

1.5.4 Qu'est ce que l'héritage ?

Les objets sont définis à partir des classes ; la connaissance de la classe, à laquelle appartient un objet, permet de connaître beaucoup de ses caractéristiques. Même si nous ne connaissons pas avec une précision absolue un vélo particulier, le simple fait qu'il appartienne à la classe des vélos nous permet de savoir qu'il possède deux roues, un guidon, etc.

DESSIN !!!

Dans les langages orientés objets, il est permis de définir des classes en fonction d'autres classes. Par exemple, les vélos de course, les vélos tout terrain et les tandems sont des classes d'objets qui partagent des caractéristiques communes ; ce sont tous des vélos avec quelques caractéristiques supplémentaires. On dira que les vélos de course, les vélos tout terrain et les tandems sont sous-classes de la classe des vélos et que la classe des vélos est une classe de base des classes de vélos de course, de vélos tout terrain et des tandems.

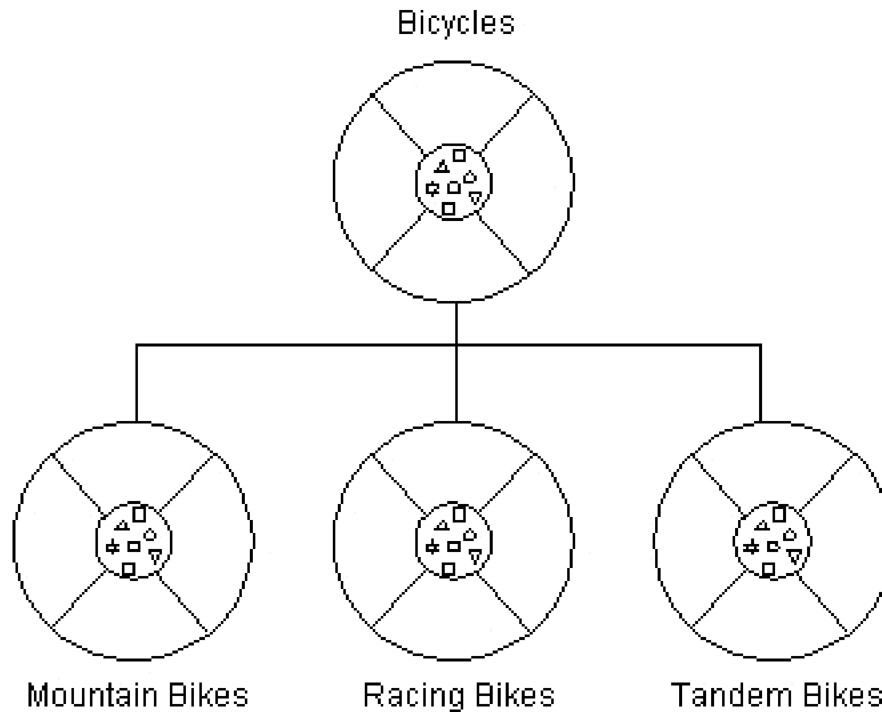

FIG. 1.1: Héritage

Chaque sous-classe hérite de toutes les caractéristiques de la classe père : les attributs et les méthodes. Les sous-classes ne sont condamnées à ne contenir que les seules caractéristiques de la classe père ; elles peuvent s'enrichir de nouveaux attributs et de nouvelles méthodes. Ces sous classes sont donc définies par les attributs et les méthodes de la classe père, mais également des attributs et méthodes propres à la sous classe.

Les sous-classes peuvent également modifier les méthodes héritées d'une classe père et proposer une version spécialisée de cette méthode. L'héritage (comme dans la vie courante) se propage de génération en génération. Une classe père peut elle-même être une sous-classe d'une autre classe et ainsi de suite... La hiérarchie des classes peut être aussi profond que l'on souhaite. Les attributs et les méthodes sont transmis vers d'une classe vers la sous-classe et ce autant de fois que nécessaire.

Les avantages de l'héritage. Les sous-classes proposent des comportements spécialisés dans certains cas, tout en conservant certains autres hérités de la classe père. A travers l'héritage, le programmeur peut réutiliser un même code dans des circonstances diverses.

Le programmeur peut définir des super classes abstraites qui définissent des comportements génériques. Ces comportements peuvent être implantés, partiellement implantés ou même pas du tout implantés. Lorsque des comportements génériques ne sont pas implantés, il appartient à d'autres de définir des sous-classes de cette super classe dans lesquelles des comportements spécialisés seront spécifiés.

2. C'est quoi Java ?

Sommaire

2.1 Le langage Java est simple, familier et orienté objet 21
2.2 Le langage Java est distribué 22
2.3 Le langage Java est interprété 22
2.4 Le langage Java est robuste et sûr 22
2.5 Le langage Java est portable et indépendant des plates-formes : 22
2.6 Le langage Java est dynamique et multithread 22

Vous avez sûrement déjà entendu parler du langage *Java*. Le tapage médiatique autour du langage *Java* et d'Internet laisse à penser que ce langage est uniquement conçu pour réaliser des pages *WEB* interactives et sophistiquées. *Java* est souvent associé aux fameux applets que l'on voit sur des pages *WEB* ; applets qui sont généralement des animations graphiques. Cette présentation finit par classer le langage *Java* parmi les divers langages dédiés aux pages WEB tels VBScript et JavaScript. Ces derniers sont conçus pour réaliser des programmes devant être encapsulés dans du code *HTML* de manière à réaliser des pages interactives, et ce, sans avoir à utiliser les *CGI*. *VBScript* et *JavaScript* sont intimement liés aux pages *WEB* et aux browser de pages *HTML* : ils ne permettent pas de créer des applications autonomes i.e. indépendantes des pages *WEB*.

Quant à Java, il s'agit d'un langage de programmation à part entière. Il a été inventé dans les laboratoire de *SUN* par Bill Joy et James Gosling. L'objectif premier de ce langage était définir un langage de programmation portable sur toutes les plates-formes existante (1990). Avec l'intérêt grandissant d'Internet dans les années 1993, ce langage, de par sa portabilité, se métamorphose en langage dédié à Internet. Dans les années 1993, *SUN* diffuse le premier browser *HotJava* qui permet d'exécuter des applets Java i.e. des programmes Java encapsulés dans des pages *WEB*. L'attrait des applets *Java*, conduit *NetScape* et *MicroSoft* à inclure dans leur propre browser, les composants permettant d'exécuter les applets. Et c'est par le biais d'Internet, que le langage *Java* a connu un succès qu'on peut qualifier de foudroyant. En effet, bien peu de langages informatiques ont connu un tel succès en si peu de temps.

Les concepteurs de *Java* qualifient leur langage de simple, orienté objet, familier, distribué, interprété, robuste, sûre, portable, dynamique et multithread.

2.1 Le langage Java est simple, familier et orienté objet

Java est un langage simple car il n'utilise qu'un nombre restreint de nouveaux concepts. Sa syntaxe est très proches du langage *C*, ce qui rend familier aux programmeurs *C* et *C++*. Java épure le langage *C* et *C++* de toutes les scories sur lesquels se blessent bien des programmeurs et qui occupent une partie non négligeable du temps de développement. Par exemple,

- les `struct` et `union` n'existent plus, seul le concept de classes existe en *Java*.
- Bonne nouvelle pour les programmeurs débutants : il n'y a plus de pointeurs et des manipulations les concernant ; un gain en temps important en découle lors des développement : plus de pointeurs adressant des emplacements farfelus !
- Plus de problème de gestion de la mémoire, *Java* se charge (presque) de restituer au système les zones mémoire inaccessibles et ce sans l'intervention du programmeur
- etc.

Il n'a plus de préprocesseur dans *Java* ; par exemple,

- le code conditionnel pour debboguer se fait avec des tests sur des variables static et final. Lors de la production du code, les instructions inutiles seront éliminées.
- Du fait que Java est indépendant des plates-formes, il n'est plus nécessaire décrire du code dépendant des diverses architectures.

– Les fameux fichiers " .h " n'ont plus de raison d'être ; le code produit contient toutes les informations sur les types de données manipulés.

Java est un langage orienté objet : un programme Java est centré complètement sur les objets et fournit un ensemble prédéfini de classes facilitant la manipulation des entrées-sorties, la programmation réseau, système, graphique. Excepté les types de données fondamentaux, tout est objet ! Seule l'héritage simple existe en Java, pour utiliser l'héritage multiple, on utilisera les interfaces.

2.2 Le langage Java est distribué

Conçu pour développer des applications en réseaux, les manipulations des objets distants ou locaux se font de la même manière. Par exemple, l'ouverture d'un fichier local ou distant se programme de manière identique. La classe Socket permet la programmation d'applications Client/Serveur de manière aisée.

2.3 Le langage Java est interprété

Pour être portable, un programme Java n'est pas compilé en code machine ; il est transformé en code intermédiaire interprété. De plus, il n'y pas plus de phase d'édition de liens ; les classes sont chargées en fonction des besoins lors de l'exécution et ce de manière incrémentale.

2.4 Le langage Java est robuste et sûr

Le langage Java est fortement typé ; il élimine bien des erreurs d'incohérence de type à la compilation. La suppression de la manipulation des pointeurs permet également de réduire de manière importante les erreurs. Un glaneur de mémoire rendu libre permet de décharger le programmeur d'une gestion fastidieuse de la mémoire.

Destiné pour des applications réseaux, la sécurité dans Java est un aspect primordial. Le fait de pas pouvoir manipuler les pointeurs et donc d'accéder à des zones mémoire sensibles diminue fortement l'introduction des virus informatiques.

Java ne supprime pas tous les problèmes de sécurité mais les réduit fortement. Un ensemble de garde fous sont placés entre l'interprétation du code Java et la réalisation de l'instruction (Verifier, Class Loader, Security Manager) pour contrôle au mieux l'exécution du code Java.

2.5 Le langage Java est portable et indépendant des plates-formes :

Le code intermédiaire produit est indépendant des plates-formes : il pourra être exécuté sur tous types de machines et systèmes pour peu qu'ils possèdent l'interpréteur de code Java. Même la programmation graphique, réseau et système est totalement indépendant des machines et systèmes. Les problèmes de " portage " qui occupe une partie non négligeable du temps de développement des logiciels disparaît ainsi.

2.6 Le langage Java est dynamique et multithread

Le langage Java est dynamique et s'adapte à l'évolution du système sur lequel il s'exécute. Les classes sont chargées en fur et à mesure des besoins, à travers le réseau s'il le faut. Les mises à jour des applications peuvent se faire classe par classe sans avoir à recompiler le tout en un exécutable final.

De nos jours, les applications possèdent un haut degré de parallélisme : il faut pouvoir écouter une musique, tout en voyant une animation graphique etc. Java permet le mutlithreading de manière simple.

3. Eléments de base

Sommaire

3.1 Le jeu de caractères Unicode 23
3.2 Les commentaires 23
3.2.1 Les commentaires multilignes 23
3.2.2 Les commentaires lignes 24
3.2.3 La documentation 24
3.3 Les identificateurs 24
3.4 Les types de données élémentaires 24
3.5 Les constantes littérales 25
3.5.1 Constantes booléennes 25
3.5.2 Constantes entières 25
3.5.3 Constantes flottantes 25
3.5.4 Constantes de type caractères 26
3.5.5 Constantes de type chaîne 26
3.5.6 La constante null 26
3.6 Les variables 26
3.6.1 Nature des variables 27
3.6.2 Variables locales 27
3.6.3 Variables globales ? 27

3.1 Le jeu de caractères Unicode

Le langage *Java* est destiné à être utilisé sur *Internet*. Il est donc naturel que celui se préoccupe des jeux de caractères internationaux. Le jeu de caractères utilisé dans *Java* est celui connu sous le nom de *Unicode*. Ce jeu de caractère est codé sur 16 bits (au lieu de 7 bits du code ASCII, et 8 bits pour Latin-1), les 256 premiers caractères sont ceux du jeu de caractères Latin-1. Ce jeu de caractères étendu est prévu pour coder l'ensemble de tous alphabets internationaux (*japonais*, *chinois*, *arabe*, *hindi*, etc.)

Pour l'heure, ne disposant pas d'éditeurs de texte permettant de visualiser le jeu de caractère Unicode, les caractères non Latin-1 seront notés par des séquences d'échappement de la forme `\u`*nnnn*. Ces caractères peuvent être utilisés dans presque toutes les parties d'un programme *Java* : commentaire, nom de variables ou méthodes, nom de classes etc.

3.2 Les commentaires

3.2.1 Les commentaires multilignes

Tout programme (grand ou petit, simple ou complexe) contient (ou devrait contenir) des commentaires. Ils ont pour but d'expliquer ce qu'est sensé faire le programme, les conventions adoptées et tout autre information rendant le programme lisible à soi même et surtout à autrui. *Java* dispose de trois types de commentaires : les *commentaires multilignes*, les *commentaires lignes* et les *commentaires de type documentation*. On peut utiliser n'importe quel caractère Unicode dans ces trois types de commentaires.

Début	Fin	Alphabet
0020	007E	Latin (US)
00A0	00FF	Latin-1 (Supplément)
⋮	⋮	⋮
05D0	05EA	Hébreu
⋮	⋮	⋮
0B80	0BFF	Tamoul
⋮	⋮	⋮
4E00	9FFF	Les idéogrammes unifiés (chinois, japonais, coréen, .)

Tab. 3.1: Le standard Unicode

3.2.2 Les commentaires lignes

Les *commentaires lignes* débutent avec les symboles "`//`" et qui se terminent à la fin de la ligne. Ils sont utilisés pour des commentaires courts qui tiennent sur une ligne.

```
// Ce programme imprime la chaîne
//  de caractères " bonjour " à l'écran
...
```

Un *commentaire multiligne* commence par les caractères "`/*`" et se terminent par "`*/`". A l'intérieur de ces délimiteurs toute suite de caractères est valide (sauf évidemment "`*/`").

```
/* Ce programme imprime la chaîne
de caractères "bonjour" à l'écran
*/
```

3.2.3 La documentation

Enfin, un texte encadré entre "`/**`" et "`*/`" servira à produire automatiquement (avec l'outil `javadoc`) la documentation sous forme *HTML* à l'image de la documentation officielle de *SUN*. Ces commentaires servent à documenter les classes que l'on définit. *Java* exige donc qu'ils figurent avant la définition de la classe, d'un membre de la classe ou d'un constructeur. Vous trouverez en annexe (46), des informations plus détaillées sur `javadoc`.

```
/** Documentation de la classe .
*/
```

3.3 Les identificateurs

Comme dans tout langage de programmation, les *identificateurs* servent à désigner des *variables*, des *constantes*, des noms de méthodes etc. On appelle identificateur toute suite de lettres ou chiffres commençant par une lettre ; une lettre est un caractère alphabétique ou le caractère '_' ou le caractère '$'. Voici des exemples d'identificateurs :

```
toto     toto123     _toto     $toto     \u5678toto
```

Parmi tous les identificateurs, il en existe certains qui sont réservés (voir figure 3.2) *i.e.* qu'ils ont un sens prédéfini pour le langage *Java* et que le programmeur ne peut utiliser comme identificateurs.

Outre ces mots clés, il existe trois identificateurs (qui sont des constantes symboliques) sont également réservés : **`null`**, **`true`** et **`false`**.

3.4 Les types de données élémentaires

Le langage *Java* est un langage fortement typé ; chaque variable et chaque expression possède un type bien défini, et ce, dès l'écriture du programme. La donnée de ces types permet de restreindre les valeurs qu'une variable est susceptible de contenir et de donner une signification aux opérations. Les types de données de *Java* sont divisés en deux grands groupes : les *types primitifs* (ou de base) et le type *référence*. Les types primitifs sont constitués des types numériques (type entier et type flottant) et du type booléen.

Le type entier se compose des types suivants :

`byte :` les entiers codés sur 8 bits (de -128 à 127)

abstract	double	int	super
boolean	else	interface	switch
break	extends	long	synchronized
byte	final	native	this
case	finally	new	throw
catch	float	package	throws
char	for	private	transient
class	goto	protected	try
const	if	public	void
continue	implements	return	volatile
default	import	short	while
do	instanceof	static	

TAB. 3.2: Mots clés réservés

char : les caractères Unicode codés sur 16 bits (de `\u0000` à `\uffff`)

short : les entiers codés sur 16 bits (de -32768 à 32767)

int : les entiers codés sur 32 bits (de -2147483648 à 2147483647)

long : les entiers codés sur 64 bits (de -9223372036854775808 à 9223372036854775807)

Le type flottant se compose des types suivants :

float : les nombres en virgule flottante sur 32 bits (IEEE 754-1985)

double : les nombres en virgule flottante sur 64 bits (IEEE 754-1985)

Le type *booléen* (`boolean`) est constitué des constantes littérales `true` ou `false`.

Le type *référence* se décompose en trois sous types : le type *classe*, le type *interface* et le type *tableau*.

3.5 Les constantes littérales

Nous avons présenté les divers types de données élémentaires du langage *Java* sans dire comment écrire une constante de l'un de ces types dans un programme *Java*. Nous allons à présent donner la syntaxe utilisée dans le langage *Java* pour désigner des constantes littérales .

3.5.1 Constantes booléennes

Les seules constantes possibles sont les identificateurs `true` et `false`.

3.5.2 Constantes entières

Les constantes entières peuvent s'exprimer

- En notation décimale : `123`, `-123`, etc. Excepté le zéro, aucune constante en notation décimale ne commence par `0`.
- En notation octale avec un `0` en première position : `0123`
- En notation hexadécimale avec les caractères `0x` ou `0X` en première position : `0x1b`, `0X2c`, `0X1B`, `0X2C`, etc.

Le type d'une constante entière est toujours de type int. Les suffixes `L` et `l` permettent de préciser qu'une constante entière est de type `long`.

```
1L, 0x7FFL, 0x0fffffffffffffffL, 16L, etc.
```

représentent des constantes entières de type long.

3.5.3 Constantes flottantes

Une constante flottante se présente sous la forme d'une suite de chiffres (*partie entière*), un point qui joue le rôle de virgule, une suite de chiffres (*partie fractionnaire*), une des deux lettres `e` ou `E`, éventuellement le signe `+` ou `-` et suivi d'une suite de chiffres (valeur absolue de l'*exposant*). La partie entière ou la partie fractionnaire peut être omise (pas les deux) ; de même le point ou l'exposant peut être omis (pas les deux).

Une constante flottante est supposée être de type `double`. Le suffixe `F` ou `f` permet de transformer le type de la constante en type `float`. Le suffixe `D` ou d indique qu'elle est de type `double`. Le plus grand nombre flottant (`float`) positif est `3.40282347e+38f`. Le plus petit nombre flottant (`float`) positif (non nul) est `1.40239846e-45f`. Le plus grand nombre flottant (`double`) positif est `1.79769313486231570e+308`. Le plus petit nombre flottant (`double`) positif (non nul) est `4.94065645841246544e-324`.

```
.5e7, 5.e6, 5e6, 1e1f
```

Une constante `double` ne peut être affectée à une variable de type `float` que si elle est convertie explicitement en `float`.

3.5.4 Constantes de type caractères

Les constantes de type caractère sont constituées d'un caractère ou d'une séquence d'échappement encadré par des apostrophes ("'"). Le caractère `'` se note `\'` et le caractère `\` se note `\\`. Les caractères non *ISO-Latin-1* se note à l'aide de séquences d'échappement .

```
'a'  '%'   '\\t'   '\\\\'   '\\''   '\\u03a9'   '\\uFFFF'   '\\177'
```

Voici une liste non exhaustive de séquences d'échappement : Séquence

`\n`	nouvelle ligne
`\t`	tabulation horizontale
`\b`	retour d'un caractère en arrière
`\r`	retour chariot
`\f`	saut de page
`\a`	beep
`\'`	apostrophe
`\"`	guillemets
`\\`	anti-slash
`\ddd`	en notation octale
`\udddd`	en notation unicode

Tab. 3.3: Séquences d'échapemment

3.5.5 Constantes de type chaîne

Une chaîne de caractères est une suite de caractères (éventuellement vide) entre guillemets. Il en découle que l'on est autorisé à utiliser les séquences d'échappement dans les chaînes. La fonction

```
System.out.print("Bonjour\n\tComment ca va ? "\n) ;
```

produit la sortie suivante :

```
Bonjour
          Comment ca va ?
```

Une constante chaîne de caractère doit être écrite sur une seule ligne. Lorsqu'il est trop long pour tenir sur une même ligne, on découpe celle-ci en deux et on les concatène avec l'opérateur `+` qui est l'opérateur de concaténation de chaînes. Les deux instructions suivantes sont équivalentes.

```
x = "abcdefghijklmnopqrstuvwxyz" + "ABCDEFGHIJKLMNOPQRSTUVWXYZ" ;
x = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ" ;
```

3.5.6 La constante null

Le type `null` admet une constante `null` servant à désigner une référence non définie.

3.6 Les variables

En *Java*, toute variable utilisée dans un programme doit auparavant être définie. La définition d'une variable consiste à la nommer, lui donner un type et éventuellement lui donner une valeur initiale (on dira initialiser). C'est cette définition qui réserve (on dira alloue) la place mémoire nécessaire et ce en fonction du type. Une variable n'est rien d'autre qu'un emplacement mémoire d'une taille suffisamment grande pour contenir une valeur d'un type donné.

Initialiser une variable consiste à remplir, avec une constante, l'emplacement réservé à cette variable. Cette opération s'effectue avec l'opérateur `=`. On pourra également (de manière optionnelle) donner un "qualifier" à une variable : le mot clé `final`, par exemple, précise la valeur de la variable ne sera pas modifiée dans le programme. Il existe, bien entendu, d'autres qualifier que nous verrons plus tard.

```
int x = 2 ;
char c = 'c' ;
double d = 1.3 ;
final float f = 1.2f ;
```

Le type d'une variable est soit un type primitif soit une référence. La valeur d'une variable sera modifiée par une affectation. Une variable de type primitif contient toujours une valeur de son type. Une variable de type référence contient soit une référence `null` soit une référence vers un objet dont le type est compatible avec le type de la variable.

Les déclarations de variables peuvent figurer n'importe où dans le corps d'une méthode. Il n'est pas nécessaire de placer ces définitions de variables au début d'une méthode, d'une classe ou d'un bloc d'instructions.

Chaque variable possède une visibilité et une durée de vie.

3.6.1 Nature des variables

On distingue 7 natures de variables :

- les variables d'instance (6)
- les variables de classe (6.2)
- les variables de type tableau (11)
- les paramètres des méthodes (6.1.2)
- les paramètres des constructeurs (6.1.4)
- les variables de type exception (10)
- les variables locales

3.6.2 Variables locales

Une variable locale est une variable qui est définie à l'intérieur d'une méthode. Il n'est pas nécessaire que cette déclaration figure en début de bloc. Comme en C++, les variables locales sont visibles à partir de leur définition jusqu'à la fin du bloc le plus interne qui contient cette définition. Leur durée de vie est limitée à au bloc qui contient la déclaration

```
...
for (int i = 0 ; i != 0 ; i = i + 1) {
        ...             // variable i visible dans tout ce bloc
}
...
```

3.6.3 Variables globales ?

Toute définition de variables figure forcément à l'intérieur d'une déclaration de classe. Contrairement au langages C et C++, le concept de variables globales n'existe pas. Il n'y a que des variables statiques ou de classe qui sont l'équivalent (sous certaines conditions) de nos habituelles variables globales. Toutefois, leur visibilité dépend des qualifiers associés à la variable et à la classe dans laquelle elle est définie. Par contre, sa durée de vie commence au chargement de la classe qui contient sa définition et se termine à la fin de l'exécution du programme.

```
public class X {
    ...
    public static int varGlobale=100 ;
    ...
}
```

Dans ce exemple, la variable varGlobale est l'équivalent d'une variable globale. Elle est visible de n'importe quelle partie du programme. Ce sont les qualifiers public qui permettent d'étendre ainsi leur visibilité. Nous verrons tout cela en détail dans les chapitres consacrés aux classes, à l'héritage, aux interfaces et aux packages.

4. Opérations et expressions

Sommaire

4.1 Généralités sur les expressions . . . 29
4.1.1 Priorité des opérateurs . . . *30*
4.1.2 Ordre d'évaluation . . . *30*
4.1.3 Type d'une expression . . . *30*
4.1.4 Erreur d'évaluation d'une expression . . . *30*
4.2 Affectation . . . 30
4.3 Expressions arithmétiques . . . 31
4.3.1 Expressions multiplicatifs . . . *31*
4.3.2 Expressions additifs . . . *32*
4.3.3 Opérateurs unaires . . . *32*
4.3.4 Pré et post incrément et décrément . . . *33*
4.4 Expressions de comparaison . . . 33
4.5 Concaténation des chaînes de caractères . . . 33
4.6 Expressions logiques . . . 33
4.7 Manipulation de bits . . . 34
4.7.1 Opérations bits à bits . . . *34*
4.7.2 Décalage de bits . . . *34*
4.8 Autres opérateurs binaires d'affectation . . . 35
4.9 Expression conditionnelle . . . 35
4.10 Changement de type . . . 35
4.10.1 Les conversions implicites . . . *35*
4.10.2 Les conversions explicites . . . *36*
4.11 Création des objets . . . 36
4.11.1 Cas des objets . . . *36*
4.11.2 Cas des tableaux . . . *36*
4.12 Récapitulatif . . . 36

4.1 Généralités sur les expressions

Une expression est un objet syntaxique obtenu en assemblant "correctement" des constantes, des variables et des opérateurs. Par exemple, $x + 3$ est une expression construite en appliquant l'opérateur $+$ sur les opérandes x et 3. Dans le langage Java, il y a bien d'autres opérateurs que les opérateurs arithmétiques qu'on a l'habitude de manipuler.

Un programme passe la quasi totalité de son temps à évaluer des expressions pour produire des effets de bord ou pour calculer des valeurs. Une expression avec un *effet de bord* est une expression qui lorsqu'elle s'évalue, produit un changement de l'état du système. Par exemple, l'affectation est une expression qui a pour effet de bord la modification du contenu de l'opérande gauche. L'invocation d'une méthode est une expression qui lorsqu'elle s'évalue produit comme effet de bord, l'appel de la méthode après évaluation des paramètres de la méthode.

Le résultat de l'évaluation d'une expression est soit une valeur, soit une variable (en C, une *lvalue*) soit void. Le résultat de l'évaluation d'une expression est void dans l'unique cas de l'invocation d'une méthode qui ne retourne rien (une procédure).

4.1.1 Priorité des opérateurs

Nous avons l'habitude de manipuler des expressions (par exemple arithmétiques) et il nous est relativement aisé de préciser exactement le sens des expressions comme : $2 + 3 * 4 * 5 - 2$, $2 - 3 - 4$ etc. On sait que ces expressions sont équivalentes à $(2 + (3 * (4 * 5))) - 2$, $(2 - 3) - 4$. Introduire les parenthèses permet de définir sans ambiguïté l'expression que l'on manipule.

A priori, c'est notre culture mathématique qui nous permet de parenthéser ces expressions. Pour éviter l'usage des parenthèses qui alourdissent la lecture, et lorsque cela est possible, les mathématiciens ont fixé des règles pour que tout le monde parenthèse (dans sa tête) de la même manière toute expression ambiguë.

Par exemple, dans l'expression $2 + 3 * 4$, la sous expression $3 * 4$ est évaluée en premier et le résultat obtenu est ajouté à la valeur 2 (forme parenthésée : $2 + (3 * 4)$). On dit que l'opérateur $*$ possède une priorité supérieure à la priorité de l'opérateur $+$. De même, dans l'expression $2 - 3 - 4$, la sous expression $2 - 3$ est évaluée en premier et, au résultat obtenu, on soustrait la valeur 4 (forme parenthésée : $(2 - 3) - 4$). On dit que l'ordre d'évaluation de l'opérateur $-$ est de gauche à droite.

4.1.2 Ordre d'évaluation

Comme nous le verrons plus loin, outre les expressions arithmétiques, le langage *Java* dispose de beaucoup d'autres sortes d'expressions. La donnée d'une priorité et d'un ordre d'évaluation permet de fixer des règles communes d'évaluation des expressions. Ces priorités et ordre d'évaluation ne permettent évidemment pas de se dispenser des parenthèses dans tous les cas. En effet, on utilise les parenthèses lorsqu'on veut évaluer une expression d'une manière autre que celle définie à l'aide de la priorité et de l'ordre d'évaluation. Par exemple, si l'on veut faire $2+3$ et multiplier le résultat par 4, on sera contraint de noter $(2+3)*4$ et il n'est pas possible de l'écrire sans les parenthèses avec la notation infixée.

Les opérandes de chaque opération sont complètement évalués avant d'effectuer l'opération. Il existe, cependant trois exceptions : il s'agit des opérateurs `&&`, `||` et `? :`. Java garantit également que les opérandes sont évalués de gauche à droite. Par exemple, dans l'expression $x+y$, on est sûr que x est évalué avant y. Cette connaissance de l'ordre d'évaluation est particulièrement importantes lorsqu'il s'agit d'expressions avec des effets de bord.

4.1.3 Type d'une expression

Toutes les expressions, syntaxiquement correctes, ont un *type*. Le *type* des expressions sont toujours connus dès la "compilation". Une expression de type T peut être affectée à une variable du même type ou d'un *type compatible*.

Le type d'expression est donnée par le type des opérandes et de la sémantique des opérateurs. Tous types entiers autre que `long` sont convertis en `int` avant d'être évalués. Le résultat est de type `int` à moins que l'un des opérandes soit de type `long`.

4.1.4 Erreur d'évaluation d'une expression

L'évaluation d'une expression peut conduire à une erreur. Dans ces cas, Java lance une *exception*(voir 10) précisant la raison exacte de l'erreur.

- `OutOfMemoryError` : Cette erreur est produite lorsque l'espace mémoire requise est insuffisante. Cette erreur est générée lors de la création (dynamique) des objets d'une classe, des tableaux, des chaînes de caractères.
- `ArrayNegativeSizeException` : Une des dimensions d'un tableau est négative
- `NullPointerException` : La valeur d'une référence d'un objet est null.
- `IndexOutOfBoundsException` : la valeur de l'indice d'un tableau est hors des bornes du tableau.
- `ClassCastException` : l'opération de cast n'est pas permise.
- `ArithmeticException` : l'opérande droit d'une division ou d'un modulo est nul.
- `ArrayStoreException` : la référence que l'on veut affecter à un élément d'un tableau n'est pas du bon type.
- Les exceptions générées par l'invocation d'une méthode.
- Les exceptions générés par les constructeurs lors de la création d'objets (voir 6 et 10).
- etc.

4.2 Affectation

L'opération la plus importante dans un langage de programmation est celle qui consiste à donner une valeur à une variable. Cette opération appelée *affectation* est désignée par le symbole $=$.

L'*affectation* range une valeur dans une variable (une zone mémoire), il est impératif que le membre gauche d'une affectation représente une zone mémoire : c'est ce qu'on appelle une *lvalue* en *C*. Une constante n'est pas une *lvalue* car il ne désigne pas l'adresse d'une zone mémoire. Elle ne peut donc pas figurer en membre gauche d'une affectation.

Le membre droit d'une affectation peut désigner soit une constante soit une zone mémoire soit une expression quelconque : l'affectation `x = 2` range la valeur `2` dans la variable `x`, l'affectation `x = y` range dans la variable `x` le contenu de la variable `y` et l'affectation `x = y + 1` range dans la variable `x` le contenu de la variable `y` incrémenté de 1.

L'affectation est une expression.

La valeur d'une affectation est la valeur de son membre gauche après exécution de l'affectation. Si la variable `y` contient la valeur 100 alors l'affectation `x=y+20` a pour valeur 120.

Une affectation peut figurer en membre droit d'une autre affectation.

L'affectation `x=y=1` est parfaitement valide car elle représente l'affectation `x=(y=1)`. Puisque `y=1` est une expression, elle peut figurer en membre droit d'une affectation. Puisque elle est syntaxiquement correcte, quel sens donner à cette affectation ? L'expression `y=1` a pour valeur celle de son membre droit (ici la valeur entière 1), l'affectation`x=y=1` range dans la variable `x` la valeur 1 après avoir rangé dans `y` cette même valeur 1. Ainsi, l'affectation `x1=x2=x3=...=x20=0` est équivalent à la suite d'affectations (dans l'ordre indiqué) `x20=0`, `x19=0`, `x18=0`, ..., `x1=0`.

Une erreur de compilation est générée lorsque le type de l'expression membre droit de l'affectation ne peut être converti en le type du membre gauche.

A l'exécution, si l'opérande gauche n'est pas un élément d'un tableau, l'opérande gauche est tout d'abord évalué, ensuite l'opérande droit est évalué et enfin la résultat de cette dernière est convertie au type de l'opérande gauche. Si durant l'une des ces trois phases, une erreur survient, l'évaluation de l'affectation est immédiatement arrêtée et l'erreur générée pour l'affectation est cette première erreur. Si, par contre, l'opérande gauche est une élément d'un tableau, les opérations effectuées sont les suivants (dans l'ordre) :

- Evaluation de la sous expression désignant le tableau et en cas d'erreur d'évaluation, renvoi de cette dernière
- Evaluation de la sous expression désignant l'indice dans ce tableau et en cas d'erreur d'évaluation, renvoi de cette dernière
- Evaluation de l'opérande droit et en cas d'erreur d'évaluation, renvoi de cette dernière
- Si le tableau est null, renvoi de l'erreur NullPointerException
- Si l'indice est en dehors des bornes du tableau, renvoi de l'erreur IndexOutOfBoundsException
- Conversion du type de l'opérande gauche vers le type de l'opérande droit et affectation. En cas d'erreur, l'erreur ArrayStoreException est renvoyée.

TAB. 4.1: Opérateur d'affectation

Préc	Ordre	Opérateur	Type	Description
13	droite gauche	=	Variable, qcq	Affectation

4.3 Expressions arithmétiques

Les opérateurs `+`, `-` et `*` fonctionnent comme on s'y attend. Comme en *C*et *C++*, les caractères sont un type particulier d'entier. On peut donc appliquer les opérations arithmétiques sur les caractères. Les expressions `'A' - '0'`, `'A' + 1` sont syntaxiquement corrects et s'évaluent parfaitement.

Cas des entiers L'arithmétique entière de Java est une arithmétique complément à deux, modulaire. Cette arithmétique ne produit jamais de débordement car le résultat est toujours réduite modulo le domaine de valeurs du type des opérandes.

Cas des flottants Java utilise le standard IEEE 754-1985 pour les nombres à virgules flottantes. Une opération arithmétique peut conduire soit à un *underflow* (valeur trop petite) ou un *overflow* (valeur top grande). Une opération sur les flottants peut produire un résultat pour lequel il n'existe pas une approximation permettant de le représenter par un nombre flottant. C'est, par exemple, le cas du résultat d'une division par zéro. Ce type de résultat est représenté par une valeur particulière appelée `NaN` (*Not a Number*).

L'arithmétique flottante Java est un sous ensemble du standard IEEE-754-1985.

4.3.1 Expressions multiplicatifs

Tous les opérateurs multiplicatifs ont la même priorité et l'ordre d'évaluation est gauche-droite. Le type des opérandes doit être de type numérique. Le type de l'opération effectué dépend du type (après conversion) de ses opérandes. Le type d'une opération multiplicatif est soit de type entier (`int` ou `long`) soit flottant (`float` ou `double`).

Les opérations multiplicatifs sont la multiplication (`*`) , la division (`/`) et le reste de la division (`%`). Si l'un des opérande est `NaN`, le résultat est `NaN`.

La multiplication

La multiplication, notée avec le symbole `*`, est une opération commutative sans effet de bord. Elle est associative sur les entiers et ne l'est pas sur les flottants. Lorsque le résultat de la multiplication de deux entiers dépasse des plages de valeurs permises selon le type (`int` ou `long`), le résultat est ramené au modulo du type.

a	b	$a \times b$	a/b	$a\%b$
Fini	± 0.0	± 0.0	$\pm\infty$	NaN
Fini	$\pm\infty$	$\pm\infty$	± 0.0	a
± 0.0	± 0.0	$\pm\infty$	NaN	NaN
$\pm\infty$	Fini	$\pm\infty$	$\pm\infty$	NaN
$\pm\infty$	$\pm\infty$	$\pm\infty$	NaN	NaN
$\pm\infty$	± 0.0	NaN	NaN	NaN

TAB. 4.2: Arithmétique virgule flottante

La division

Le résultat de la division, notée avec le symbole `/`, de deux entiers est le quotient de la division entière de ces deux nombres. Le résultat de la division de deux flottants est le quotient de la division de ces deux flottants.

Le reste de la division

Le résultat de la division de deux entiers est le reste de la division entière de ces deux nombres. Contrairement à *C*et *C++*, le reste de la division de deux flottants est également défini pour les nombres à virgules flottantes. Le résultat du reste de la division de a par b est définie par $r = a - (b \times q)$ où q est le quotient de a par b.

TAB. 4.3: Opérateurs multiplicatifs

Préc	Ordre	Opérateur	Type	Description
2	gauche droite	*, /, %	Arithmétique	Multiplication, division et reste

4.3.2 Expressions additifs

Les opérations additifs, qui sont l'addition (`+`) et la soustraction (`-`), ont la même priorité et l'ordre d'évaluation est gauche-droite. Le type des opérandes doit être de type numérique. Le type de l'opération effectué dépend du type (après conversion) de ses opérandes. Le type d'une opération multiplicatif est soit de type entier (`int` ou `long`) soit flottant (`float` ou `double`).

Les opérations additifs entières, tout comme les opérations multiplicatifs, ne produit aucune erreur. Il n'y pas de débordement possible : l'arithmétique est une arithmétique e complément à 2.

Quant aux opération additifs sur les flottants, la somme de deux infinis produit un NaN si les signes différent et l'infini sinon. La différence de deux infinis de même signe est un NaN. ; sinon le résultat est celui attendu.

TAB. 4.4: Opérateurs additifs

Préc	Ordre	Opérateur	Type	Description
3	gauche droite	+, -	Arithmétique	Addition et soustraction

4.3.3 Opérateurs unaires

Les opérateurs unaires (`-` ou `+`) s'appliquent sur un type numérique. Pour les flottants, si x est 0.0 alors $-x$ est égal à -0.0 alors $0.0 - x = 0.0$.

TAB. 4.5: Opérateurs arithmétiques unaires

Préc	Ordre	Opérateur	Type	Description
1	droite gauche	+, -	Arithmétique	Plus et moins unaire

4.3.4 Pré et post incrément et décrément

Il existe deux opérateurs ++ et deux opérateurs - : une pour la forme préfixée et l'autre pour la forme postfixée. Ce sont des opérateurs unaires qui s'appliquent sur un opérande de type numérique.

Opérateur post-incrément et post-décrément (++ et –)

La valeur d'une expression de post-incrément (post décrément) est la valeur de l'opérande et a pour effet de bord, le stockage de la valeur de l'opérande incrémenté (resp. décrémenté) de 1.

Opérateur pré-incrément et pré-décrément(++ et –)

La valeur d'une expression de pré-incrément (pré-décrément) est la valeur de l'opérande incrémenté (resp. décrémenté) de 1 et a pour effet de bord, le stockage de cette valeur.

Tab. 4.6: Opérateurs d'incrément et de décrément

Préc	Ordre	Opérateur	Type	Description
1	droite gauche	++, –	Arithmétique	Incrémént et décrément

4.4 Expressions de comparaison

Le résultat d'une compraison est une valeur boolénne (vrai ou faux). Dans le langage Java, le résultat d'une comparaison est **true** ou **false** selon que cette comparaison est vraie ou fausse.

Tab. 4.7: Opérateurs de comparaison

Préc	Ordre	Opérateur	Type	Description
5	gauche droite	<,<=, >, >=	Arithmétique	Comparaison arithmétique
6	gauche droite	==, !=	objet et type primitif	egalité et différent

4.5 Concaténation des chaînes de caractères

Si l'un des opérandes de l'opérateur + est un objet de type **String**, alors la sémantique de cette opération est la concaténation de chaînes. Si besoin, l'autre opérande est converti en **String**.

```
x = 2 ; y = 3 ;
System.out.println("Le produit de " +  x  +  " et de " +  y +  " est : " +  x*y) ;
```

produit la sortie

```
Le produit de 2 et de 3 est : 6
```

Tab. 4.8: Opérateur de concaténation de chaînes

Préc	Ordre	Opérateur	Type	Description
3	gauche droite	+	Chaînes de caractères	Concaténation de chaînes

4.6 Expressions logiques

On dispose des connecteurs logiques (sous une syntaxe particulière) et on peut fabriquer des expressions avec celles-ci. La valeur d'une expression booléenne est, comme le résultat des comparaisons, une valeur booléenne.

Comme d'habitude, l'évaluation d'une expression booléenne obéit aux règles de priorité et à l'ordre d'évaluation. Java propose deux types d'opérateurs logiques : les opérateurs classiques et les opérateurs conditionnels.

L'évaluation des opérations classiques se fait par l'évaluation complète de chaque opérande et l'application de l'opération sur les résultats obtenus.

L'évaluation des opérateurs conditionnels se fait par évaluation successives des opérandes. Cette évaluation s'arrête dès que l'on est capable de donner la valeur de l'expression. Par exemple, en supposant que la variable `x` contient la valeur 5, l'évaluation de l'expression `(x >= 2) || ( x >= y)` s'arrête avant même d'avoir évalué l'expression `x >= y`, puisque `(x >= 2)` est vrai d'où on peut conclure que la valeur de l'expression `(x >= 2) || ( x >= y)` est également vraie.

Cette remarque a son importance dans deux cas :

- pour les tests d'arrêts
- pour les expressions avec effet de bord. Par exemple, selon la valeur de x, l'évaluation de l'expression (x >= 2) || (x++ == y) aura pour effet de bord l'incrément de x ou pas.

TAB. 4.9: Opérateur booléens

<table>
<tr><th>Préc</th><th>Ordre</th><th>Opérateur</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>droite gauche</td><td>!</td><td>Booléen</td><td>non booléen</td></tr>
<tr><td>7</td><td>gauche droite</td><td>&</td><td>Booléen</td><td>ET booléen</td></tr>
<tr><td>8</td><td>gauche droite</td><td>^</td><td>Booléen</td><td>OU exclusif booléen</td></tr>
<tr><td>9</td><td>gauche droite</td><td>|</td><td>Booléen</td><td>OU booléen</td></tr>
<tr><td>11</td><td>gauche droite</td><td>&&</td><td>Booléen</td><td>ET conditionnel booléen</td></tr>
<tr><td>11</td><td>gauche droite</td><td>||</td><td>Booléen</td><td>OU conditionnel booléen</td></tr>
</table>

4.7 Manipulation de bits

4.7.1 Opérations bits à bits

On dispose des opérateurs suivants :

- & (ET bit à bit),
- | (OU bit à bit)
- ^ (OU Exclusif bit à bit)
- ˜(Complément bit à bit)

TAB. 4.10: Opérateurs bit à bit

<table>
<tr><th>Préc</th><th>Ordre</th><th>Opérateur</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>droite gauche</td><td>~</td><td>Entier</td><td>complément bit à bit</td></tr>
<tr><td>7</td><td>gauche droite</td><td>&</td><td>Entier</td><td>ET bit à bit</td></tr>
<tr><td>8</td><td>gauche droite</td><td>^</td><td>Entier</td><td>OU exclusif bit à bit</td></tr>
<tr><td>9</td><td>gauche droite</td><td>|</td><td>Entier</td><td>Ou bit à bit</td></tr>
</table>

4.7.2 Décalage de bits

L'opérateur « décale les bits vers la gauche en complétant par des bits à zéro sur la partie droite. L'opérateur »> décale les bits vers la droite en complétant par des bits à zéro sur la partie gauche. L'opérateur » décale les bits vers la droite en complétant par des bits à la valeur du bit de poids fort sur la partie gauche.

Précédence Ordre Opérateur Type Description

TAB. 4.11: Opérateurs de décalage de bits

Préc	Ordre	Opérateur	Type	Description
4	gauche droite	«, », »>	Entier	Décalage de bits

4.8 Autres opérateurs binaires d'affectation

Les opérateurs suivants ne sont que des raccourcis de notation : les expressions `y += x`, `y -= x`, `y *= x`, `y /= x`, `y %= x`, `y »= x`, `y «= x`, `y »>= x`, `y &= x`, `y ^= x` et `y |= x`, correspondent aux raccourcis pour `y = y+x`, `y = y-x`, `y = y*x`, `y = y/x`, `y = y%x`, `y = y»x`, `y = y«x`, `y = y»>x`, `y = y&x`, `y = y^x` et `y = y|x`.

TAB. 4.12: Autres opérateurs d'affectation

Préc	Ordre	Opérateur	Type	Description
13	droite gauche	=, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	Variable, qcq	Affectation simple et avec opération

4.9 Expression conditionnelle

Voici l'opérateur ternaire (le seul). Cette expression est une sorte de *si-alors-sinon* sous forme d'expression : si la condition e est vraie alors cette expression vaut x sinon elle vaut y.

Exemple : L'expression `a = (v ==2) ? 1 : 2` affecte la variable `a` à la valeur 1 si `v` vaut 2, sinon affecte la variable `a` à la valeur 2.

TAB. 4.13: opérateur conditionnel

Préc	Ordre	Opérateur	Type	Description
12	droite gauche	? :	Booléen, qcq	Si-alors-sinon

4.10 Changement de type

Cet opérateur permet de convertir explicitement le type d'une donnée en un autre type. L'opérateur de parenthèsage `()` permet de définir l'ordre d'évaluation d'une expression. C'est l'opérateur que l'on utilise traditionnellement. C'est également ce même opérateur qui est utilisé pour encapsuler les paramètres des fonctions. Même lorsqu'une fonction n'a pas d'arguments, ces parenthèses sont requises.

Le problème de la conversion des types se pose lorsqu'une expression est composée de données de nature différentes. Par exemple, quel sens donner à une addition d'un entier et d'un nombre flottant ? Que se passe-t-il lorsqu'on affecte un variable de type entier à une valeur de type caractère ou flottant. La langage Java définit précisément quelles sont les types de données compatibles et quel type de conversion est effectué.

La conversion explicite d'un opérande se fait en le préfixant du type choisi encadré par des parenthèses.

```
double d = 2.5 ;
long l = (long) d ;
```

4.10.1 Les conversions implicites

Les *conversions* implicites sont celles faites automatiquement par un compilateur lors de l'évaluation d'une expression (et donc également d'une affectation). Comme nous le verrons plus loin, il y a conversion implicite lors de l'invocation d'une méthode. Il y a deux type de conversion implicite :

1. Les valeurs entières peuvent être affectées à toute variable numérique dont le type support un domaine de valeur plus grand. Un char est peut utilisé partout où une valeur de type int est permis. Les valeurs de type flottant peuvent être affectées à toute variable de type flottant de précision supérieure ou égale. Les données de type entières peuvent être converties en données de type flottant.

```
long i = 0x7effffffffffffffL ;          // 9151314442816847871
float f = i ;                           // 9.15131e+18
long l = (long)f ;                      // 9151314442816847872
```

2. Le second type de conversion implicite concerne les référence vers des objets : partout une référence vers un objet d'un type T est requise, une référence vers un objet d'une sous classe de T peut être fournie. La référence `null` peut être fournie pour tout type de référence y compris les tableaux.

4.10.2 Les conversions explicites

On utilise la conversion explicite pour changer le type d'une donnée. Ce changement ne peut être arbitraire : seules certaines conversions sont permises. Par exemple, on ne peut convertir un boolean en int mais un double peut être converti en un long.

- Un double est convertible en un float.
- Un type entier est convertible en un type entier plus petit
- Un char est convertible en n'importe quel type entier et inversement.
- Une référence vers un objet d'une classe C est convertible en une référence vers une de ses super-classes.
- Le mot clé instanceof

Pour savoir si un objet o d'une certaine classe est convertible en un objet d'une autre classe C, on utilisera un test de la forme :

```
if (o instanceof C)
```

TAB. 4.14: Opérateur de changement de type

Préc	Ordre	Opérateur	Type	Description
1	droite gauche	(*type*)	qcq	Changement de type
5	gauche droite	`instanceof`	objet	Comporaison de type

Nous verrons plus loin, en détails, les changements de type permis et ceux interdits.

4.11 Création des objets

4.11.1 Cas des objets

Comme nous le verrons plus loin, les objets complexes et structurés à l'aide des classes. Ces classes sont "une sorte de `struct` *C*". Les classes possèdes des *champs* et des *méthodes*. L'opérateur d'accès à ces membres des classes est le "`.`".

4.11.2 Cas des tableaux

Nous verrons également que les *tableaux* sont d'un type particulier d'objets. Les éléments des tableaux tab sont notés `tab[i]`. L'opérateur `[]` sert d'accès d'opérateur d'accès aux éléments d'un tableau.

TAB. 4.15: Opérateur d'accès aux membres

Préc	Ordre	Opérateur	Type	Description
1	droite gauche	., []	objet	Accès aux membres

4.12 Récapitulatif

TAB. 4.16: Récapitulatif

Préc	Ordre	Opérateur	Type	Description
1	droite gauche	++, −	Arithmétique	Incrémént et décrément
1	droite gauche	+, -	Arithmétique	Plus et moins unaire
1	droite gauche		Entier	complément bit à bit
1	droite gauche	!	Booléen	non booléen
1	droite gauche	(type)	qcq	Changement de type
1	droite gauche	., []	objet	Accès aux membres
1	droite gauche	(type)	qcq	Changement de type
2	gauche droite	*, /, %	Arithmétique	Multiplication, division et reste
3	gauche droite	+, -	Arithmétique	Addition et soustraction
3	gauche droite	+	Chaînes de caractères	Concaténation de chaînes
4	gauche droite	«, », »>	Entier	Décalage de bits
5	gauche droite	<,<=, >, >=	Arithmétique	Comparaison arithmétique
5	gauche droite	instanceof	objet	Comparaison de type
6	gauche droite	==, !=	objet et type primitif	égalité et différent
7	gauche droite		Entier et booléen	ET bit à bit et booléen
8	gauche droite	ˆ	Entier et booléen	OU exclusif bit à bit et booléen
9	gauche droite	|	Entier et booléen	Ou bit à bit et booléen
11	gauche droite	&&	Booléen	ET conditionnel
11	gauche droite	||	Booléen	OU conditionnel
12	droite gauche	? :	Booléen, qcq, qcq	Si-alors-sinon
13	droite gauche	=, *=, /=, %=,		
		+=, -=,	Variable, qcq	Affectation simple et
		<<=, >>=, >>>=,		avec opérations
		&=, ˆ=, |=		

5. Les structures de contrôle

Sommaire

5.1 Instructions et blocs . . . 39
5.2 Instruction conditionnelle (if) . . . 39
5.3 Etude de cas (switch) . . . 40
5.4 Itérations . . . 41
  5.4.1 Instruction while . . . *41*
  5.4.2 Instruction do ... while . . . *41*
  5.4.3 Instruction for . . . *42*
5.5 Etiquette, break, continue et return . . . 42
  5.5.1 Les étiquettes . . . *42*
  5.5.2 L'instruction break . . . *42*
  5.5.3 L'instruction continue . . . *42*
  5.5.4 L'instruction return . . . *42*
  5.5.5 Quid ? Point de goto ? . . . *43*
5.6 Structure d'un programme autonome Java . . . 43

5.1 Instructions et blocs

Un programme (voir 5.6) *Java* est constitué de déclarations de *classes* dans lesquelles figurent des méthodes. Les méthodes sont construites à l'aide d'*instructions* combinées entre elles avec des *structures de contrôles*. Tout d'abord qu'entend-on par instruction ? Une expression telle que `x=0`, `i++` ou `printf("coucou\n")` sont des instructions lorsqu'elles sont suivies du caractère point virgule (" ;").

```
x=0 ;
i++ ;
printf("coucou\n") ;
```

Le point virgule est appelé terminateur d'instruction. Même si n'importe quelle expression peut être utilisée par écrire une instruction, les instructions intéressantes sont celles qui provoquent un effet de bord *i.e.* qui permettent, outre le fait d'évaluer cette expression, de modifier l'état du système. C'est le cas de l'affectation qui modifie le contenu de l'opérande gauche.

Les instructions composées ou blocs sont des suites d'instructions simples ou composées regroupées à l'aide des accolades "`{`" et "`}`". L'accolade fermante n'est pas suivie d'un point virgule.

```
int i ;
i = 32 ;
printf("coucou\n") ;
```

5.2 Instruction conditionnelle (if)

Cette instruction conditionnelle permet d'exécuter des instructions de manière sélective en fonction du résultat d'un test. La syntaxe de l'instruction est :

```
if (expression)  instruction1
if (expression) instruction1 else instruction2
```

Si l'expression est vraie, l'instruction1 s'exécute ; sinon, dans le deuxième cas, c'est l'instruction2 qui s'exécute. Rappelons que, contrairement au langage C, une valeur booléenne n'est pas un entier. En particulier, la fameuse erreur, que tout le monde a au moins fait une fois dans sa vie de programmeur C, et qui sont si difficile à trouver, du style

```
if (x = 2) ...          au lieu de       if (x == 2) ...
```

ne peuvent plus se produire en Java. De plus, les instructions suivantes ne plus équivalentes :

```
if (expression) ...    et      if (expression != 0) ...
```

D'après la syntaxe de l'instruction **if**, la partie **else** est facultative ; il en découle une ambiguïté lorsqu'il y a des instructions **if** imbriquées. Dans l'exemple suivant, comme le suggère l'indentation du programme

```
if (condition1)
    if (condition2)
        instruction1
    else
        instruction2
```

L'**instruction2** correspond au **else** de **l'expression2**. L'écriture suivante est strictement identique à la première.

```
if (condition1)
    if (condition2)
        instruction1
else
    instruction2
```

Visiblement, le programmeur veut faire correspondre l'**instruction2** au **else** de l'**expression1**. Comment faire ? Il suffit de transformer l'instruction imbriquée en un bloc d'instruction(s) de la manière suivante :

```
if (condition1)  {
    if (condition2)
        instruction1
    }
else
    instruction2
```

Voici une construction, que l'on rencontre fréquemment, qui permet de faire une étude de cas :

```
if (condition1)
    instruction1
else if (condition2)
    instruction2
else if (condition3)
    instruction3
else
    instruction4
```

Cette instruction effectue successivement chacun des tests et aucun des tests n'est vraie, l'instruction4 sera exécutée (instruction par défaut).

5.3 Etude de cas (switch)

On dispose d'une instruction pour faire une étude de cas : c'est l'instruction **switch**.

```
switch (expression)  {
    case constante1 :
        instruction1
    case constante2 :
        instruction2
    ...
    case constanteN :
        instructionN
    default :
        instruction
}
```

Si la valeur de expression vaut $constante_i$, on exécute la suite des instructions commençant à $instruction_i$. Attention, cette instruction ne se contente pas d'exécuter les instructions comprises entre $instruction_i$ et $instruction_{i+1}$; elle exécute toutes les instructions $instruction_i$ ainsi que toutes celles qui suivent $instruction_i$ jusqu'à la rencontre de l'instruction **break** ou de la fin de l'instruction **switch**.

Lorsque la valeur de l'expression est égale à aucune des constantes mentionnées, ce sont les instructions étiquetées par **default** qui seront exécutées.

Exemple.

```
        char c ;
        ...
        switch(c) {
            case '1' :
            case '2' :                                  // notez l'absence d'instruction ici
            case '3' :
            case '5' :
            case '7' :
                System.out.println("%c est un nombre premier\n", c) ;
                break ;                                 // notez l'instruction break
            case '6' :
                System.out.println ("%c est un multiple de 3\n", c) ;
                                                        // notez l'absence d'instruction break
            case '4' :
            case '8' :
                System.out.println ("%c est un multiple de 2\n", c) ;
                break ;
            case '9' :
                System.out.println ("%c est un multiple de 3\n", c) ;
                break ;
            default :                                   // instruction par défaut
                System.out.println ("%c n'est pas un chiffre\n", c) ;
            }
    ...
```

Dans le programme ci-dessus, on a regroupé les cas 1, 2, 3, 5 et 7 pour n'écrire qu'une seule fois l'instruction d'affichage. L'instruction `break` arrête l'exécution de ce `switch`. Le chiffre 6 est à la fois un multiple de 2 et de 3. On factorise une partie de l'affichage avec les cas 4 et 8 ; d'où l'absence de l'instruction `break`.

5.4 Itérations

Les instructions itératives ou boucles sont réalisées à l'aide d'une des trois structures de contrôle suivantes :

```
while (condition)
    instruction

do
    instruction
while (condition) ;

for (expression1 ; condition2 ; expression3)
    instruction
```

5.4.1 Instruction while

La structure de contrôle `while` évalue la condition et exécute l'instruction tant que cette condition est vraie. Le programme suivant affiche à l'écran (dans l'ordre) tous les nombres de 10 à 0.

```
x = 10 ;
while (x >= 0) {                    // tant que x est positif ou nul
   printf("%d", x) ;                // imprimer x
    x = x -1 ;                      // décrémenter x
}
```

Autre version de ce même programme

```
x = 10 ;
while (x >= 0)                      // tant que x est positif
    printf("%d", x--) ;             // imprimer et décrémenter x
```

5.4.2 Instruction do ... while

Une variante de cette instruction `while` est l'instruction `do ... while`. Contrairement à l'instruction `while`, l'instruction

```
do
    instruction
while (condition) ;
```

est exécutée au moins une fois. L'itération s'arrête lorsque l'expression est fausse. Pour les habitués du langage *Pascal*, cette instruction n'est pas l'équivalent de l'instruction `repeat ... until`.

5.4.3 Instruction for

L'instruction **for**, comme l'instruction **while**, n'exécute l'instruction que si la condition2 est vraie.

```
for (expression1 ;                    // Initialisation
    condition2 ;                      // Conditions d'arrêt
       expression3)                   // Fin du corps de boucle
    instruction
```

Cette instruction qui est composée de trois parties séparées par des point virgules : expression1 constitue les initialisations nécessaires avant l'entrée dans la boucle. condition2 constitue la ou les conditions de boucle et expression3 constitue les instructions de fin de boucle. L'instruction for

```
for (expression1 ; condition2 ; expression3)
   instruction
```

peut se traduire avec l'instruction while de la manière suivante :

```
expression1 ;
while (condition2) {
    instruction
    expression3 ;
}
```

Les parties expression1, condition2 et expression3 peuvent être éventuellement être vides. Par exemple, on effectue une boucle infinie avec l'instruction

```
for ( ; ; )
    instruction
```

L'opérateur virgule " , " est généralement utilisé pour combiner plusieurs initialisations et plusieurs instructions de fin du corps de boucle.

Exemple

```
for (i=1, j=10 ; i < j ; i++, j--) {
        ...
}
```

5.5 Etiquette, break, continue et return

5.5.1 Les étiquettes

Comme tous les langages, n'importe quelle instruction peut être étiqutées. La syntaxe d'une étiquette est

```
etiquette :     instruction
```

5.5.2 L'instruction break

L'instruction **break** que l'on a déjà vu lors de la description de l'instruction **switch**. En fait, l'instruction **break** est utilisée dans les structures de contrôle : **switch**, **while**, **do ... while** et **for**. Dans une boucle, cette instruction provoque la sortie immédiate de la boucle sans tenir compte des conditions d'arrêt de la boucle.

Une variante de l'instruction **break** permet de lui associer une étiquette.

```
etiquette :
while (...) {
    ...
    break etiquette ;
    ...
}
```

5.5.3 L'instruction continue

Dans une structure de contrôle **while**, **do ... while** et **for**, l'instruction continue produit l'abandon de l'itération courante et, si le condition d'arrêt l'autorise, le démarrage de l'itération suivante.

5.5.4 L'instruction return

L'instruction **return** ou **return expression** provoque l'abandon de la fonction en cours et le retour à la fonction appelante. Le résultat est la valeur que la fonction appelée renvoie à la fonction appelante. Comme dans le cas des passage d'arguments d'une fonction, la valeur retournée par la fonction subit les conversions habituelles. Lorsque la fonction ne retourne aucune valeur (cas d'une procédure), on utilisera l'instruction **return** sans argument.

5.5.5 Quid ? Point de goto ?

Eh bien non, pas de **goto** ; et on s'en passe très bien avec d'une part les break étiquetés et d'autre part le bloc de code **finally**.

5.6 Structure d'un programme autonome Java

Un programme Java est constitué d'une ou de plusieurs classes. Parmi toutes ces classes, il doit exister au moins une classe qui contient la méthode statique et publique **main** qui est le point d'entrée de l'exécution du programme. Comme d'habitude, commençant une tout petit exemple :

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

Cette classe définit une classe **Bonjour** qui ne possède qu'une seule méthode. La méthode main doit être déclarée **static** et **public** pour qu'elle puisse être invoquée par l'interpréteur Java. L'argument **args** est un tableau de **String** qui correspond aux arguments de la ligne de commande lors du lancement du programme.

Avant de pouvoir exécuter ce programme, il faut tout d'abord le compiler avec la commande **javac**.

```
javac Bonjour.java
```

La commande **javac** traduit le code source en code intermédiaire java. Ce code est évidemment indépendant de la plate forme sur laquelle il a été compilé. Le compilateur Java produit alors autant de fichiers que classes qui ont été définies dans le fichier source. Les fichiers compilés ont l'extension **.class**.

Enfin, pour exécuter ce programme, il faut utiliser l'interpréteur de code Java et lui fournir le nom de la classe public que l'on veut utiliser comme point de départ de notre programme.

```
java Bonjour
```

6. Classes et Objets

Sommaire

6.1 Déclaration des classes 45
6.1.1 Champs *45*
6.1.2 Méthodes *46*
6.1.3 Objet et méthodes associées *47*
6.1.4 Constructeurs *47*
6.1.5 Destructeur *48*
6.2 Définitions de champs 48
6.2.1 Initialisation des champs static *49*
6.2.2 Initialisation des champs non-static *49*
6.2.3 Le mot clé this *49*
6.2.4 Champs final *50*
6.3 Définitions de méthodes 50
6.3.1 Passage de paramètres *50*
6.3.2 Signature et polymorphisme *50*
6.3.3 Variables locales *51*
6.3.4 Méthodes static *51*
6.4 Bloc d'initialisation static 52
6.5 Conversion des types primitifs en objets et inversement 52

Dans le langage *C*, on a l'habitude créer des objets complexes à l'aide de *structures*. Dans les langages orientés objets, on créera plutôt des *classes*. Les *classes* constituent le concept de base de le programmation objet. Elles permettent de définir des nouveaux types de données qui doivent se comporter "comme" des types pré définis et dont les détails d'implantation sont cachés aux utilisateurs de ces classes Seule l'*interface* fournie par son concepteur pourra être utilisée. Comme nous le verrons plus loin, contrairement aux types prédéfinis, les classes peuvent être créées de manière hiérarchique : une classe est souvent une sous-classe d'une autre classe.

Un *objet* est une instance d'une certaine classe ; au lieu de parler d'une variable d'une certaine classe, on dira plutôt un objet d'une certaine classe. En *Java*, on ne peut accéder aux objets qu'à travers une *référence* vers celui-ci. Une *référence* est, en quelque sorte, un pointeur vers la structure de donnée ; la différence entre une référence et un pointeur est qu'il n'est pas permis de manipuler les références comme les pointeurs de *C*ou *C++*. On ne peut connaître la valeur de la référence et on ne peut évidemment pas effectuer d'opérations arithmétiques sur les références. La seule chose permise est de changer la valeur de la référence pour pouvoir "faire référence" à un autre objet.

Une classe définit généralement deux choses :

- les structures de données associées aux objets de la classe ; les variables désignant ses données sont appelés champs.
- les services que peuvent rendre les objets de cette classe qui sont les méthodes définies dans la classe.

6.1 Déclaration des classes

6.1.1 Champs

Java (comme *C++*) possède trois mots clés pour l'*encapsulation des données* : `public`, `private` et `protected`. Les données et méthodes déclarées `private` ne sont accessibles que par les méthodes de sa propre classe. Inversement les informations déclarées `public` sont accessibles par toutes les classes. Nous verrons plus loin la signification du mot clé `protected`.

Imaginons que l'on veuille déclarer une structure de donnée `Date` constituée de trois entiers codant le jour, le mois et l'année. Nous allons pour ce faire, définir une classe `Date` de la manière suivante :

```
class Date {
    private int mois ;
    private int jour ;
    private int  année ;
    ...
}
```

Les données `mois`, `jour` et `année` sont des données privées. Elles ne sont accessibles que par les seules méthodes de cette classe. Pour que l'on puisse modifier ces champs, il faut que l'on fournisse les méthodes permettant de manipuler ces données privées.

6.1.2 Méthodes

Comme en *C++*, les méthodes sont définies par :

- un **nom** constitué par un **`identificateur`**
- des **paramètre formels** : ceux-ci sont séparés par des ",". Lorsque la méthode n'a pas de paramètre, contrairement au langage *C*, il ne faut pas préciser `void`. Le nombre de paramètres est fixe : il n'est pas possible de définir des méthodes à arguments variables.
- du **type** du retour est soit `void` (si la méthode ne retourne aucune valeur), soit un type primitif ou une référence vers un objet.
- du **corps de la méthode**.

Comme les champs d'une classe, les méthodes doivent être qualifiées de `public`, `private` ou `protected`. Les méthodes `private` ne peuvent être invoquées que par les seules méthodes de cette classe.

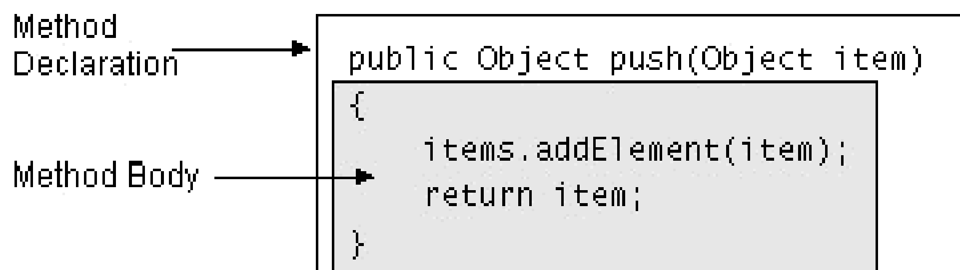

Fig. 6.1: Méthodes

```
class Date {
    private int mois ;
    private int jour ;
    private int  année ;
    ...
    public void affecter(int m, int j, int a) {
       mois = m ; jour = j ; année = a ;
    }
}
```

La méthode **`affecter`** fait partie de la classe `Date` ; il lui est donc permis d'accéder aux champs privés `mois`, `jour` et `année`. Grâce à cette méthode affecter, puisque elle est déclarée `public`, on pourra désormais affecter les champs `mois`, `jour` et `année` d'un objet de type `Date`. Les méthodes publiques de la classe Date constituent l'*interface publique* de cette classe.

```
class Date {
    private int mois ;
    private int jour ;
    private int  année ;
    public void affecter(int m, int j, int a) {
        mois = m ; jour = j ; année = a ;
    }
    public int QuelJour() { return jour ; }
    public int QuelMois() { return mois ; }
    public int QuelleAnnée() { return année ; }
    public void JourSuivant() { ... }
    public void imprimer() { ... }
}
```

Contrairement au langage $C++$, la définition effective des méthodes de la classe doit se faire dans la définition de la classe elle-même.

```
class Date {
    ...
    public void imprimer() { // imprimer la date
        System.out.println(jour + "/" + mois + "/" + année) ;
    }
}
```

6.1.3 Objet et méthodes associées

Une fois la classe déclarée, pour pouvoir utiliser un objet de cette classe, il faut définir une *instance* (ou un *objet*) de cette classe. Nous avons dit que les objets ne sont accessibles qu'à travers des *références*. Une définition qui se contente de définir un objet comme "*une variable ayant le type de la classe choisie*" ne fait que définir une référence vers un éventuel objet de cette classe.

```
Date d ;
```

La variable `d` représente une référence vers un objet de type `Date` ; Si l'on veut un objet effectif, il faut la créer explicitement avec le mot clé `new` et le constructeur de la classe `Date`.

```
Date d ;
d =  new Date() ;
```

Comme on l'a déjà dit, une méthode est un message que l'on envoie à un objet. Ainsi, pour afficher la date contenue dans l'objet `d`, on lui envoie le message `imprimer` :

```
d.imprimer() ;
```

De telles méthodes sont appelées *méthodes d'instance* ; nous verrons plus loin qu'il existe un autre type de méthode qu'on appelle *méthode de classe*.

Cette méthode n'est utilisable (ailleurs que les méthodes de la classe `Date`) que parce qu'elle fait partie de l'interface de cette classe *i.e.* qu'elle fait partie des méthodes publiques. Par contre, il ne sera pas possible d'accéder aux champs `d.jour`, `d.mois` et `d.annee` : ce sont des *données privées*.

Les structures de données d'une classe donnée, sont dupliqués pour chaque objet de cette classe. Si l'on définit deux objets de type `Date`, ils possèdent, tous les deux, leur propre exemplaire des champs `jour`, `mois` et `année`. La modification d'un de ces champs pour un objet n'affecte évidemment pas la valeur du même champ pour l'autre objet. De tels champs sont appelés **`variables d'instance`**. Nous verrons plus loin que nous pourront définir des champs où toutes les instances d'une même classe partagent le même champ (champs `static`).

Quant aux méthodes, elles ne sont évidemment pas dupliquées pour chaque objet. Un seul exemplaire de toutes les méthodes définies dans une classe suffit. On remarquera également (pour le moment du moins) que les méthodes ne peuvent être invoquées qu'en utilisant un objet. Nous verrons plus loin que des méthodes particulières n'obéissent pas à cette restriction (méthodes `static`).

6.1.4 Constructeurs

Lorsque l'on définit un objet d'une classe, il est souvent utile de pouvoir initialiser cet objet. Avec la définition de notre classe `Date`, il est évidemment possible, avec la méthode `affecter`, d'affecter les champs `jour`, `mois` et `année`.

```
Date d ;
d =  new Date() ;
d.affecter(10, 3, 87) ;
```

Mais cette façon de faire, n'est pas la plus agréable. Une meilleure façon de faire consiste à définir une méthode spécifique d'initialisation des champs qui sera automatiquement appelée lors de la création d'un objet. Cette fonction s'appelle constructeur.

```
class Date  {        ...
    public Date(int j, int m, int a) { jour = j ; mois = m ; année = a ; }
}
```

Notez que le constructeur se reconnaît par le fait qu'il porte le même nom que la classe et qu'il n'a pas de valeur de retour (pas même `void`). Voici, à présent, des exemples d'utilisation correcte et incorrecte :

```
Date noel97, dateDeNaissance ;
noel97 =  new Date(25, 12, 97) ;                // Correct
dateDeNaissance = new Date() ;                 // Incorrect
```

La création de l'objet référencé par `dateDeNaissance` est incorrecte. En effet, le constructeur de la classe `Date` dont nous disposons requiert trois arguments ; il n'est donc pas possible de créer un objet de la classe `Date` sans donner le jour, le mois et l'année en argument. On doit contourner ce problème en fournissant soit plusieurs constructeurs (à 0, 1, 2 et 3 arguments) :

```
class Date  {
    ...
    public Date(int j, int m, int a) { jour = j ; mois = m ; année = a ; }
    public Date(int j, int m) { jour = j ; mois = m ; année = 57 ; }
    public Date(int j) { jour = j ; mois = 9 ; année = 57 ; }
    public Date() { jour = 15 ; mois = 9 ; année = 57 ; }
}
```

ou plus simplement par :

```
class Date  {
    ...
    public Date(int j, int m, int a) { jour = j ; mois = m ; année = a ; }
    public Date(int j, int m) { this(j, m, 75) }
    public Date(int j) this(j, 9, 57) ; }
    public Date() { this(15, 9, 57) ; }
}
```

Pourquoi alors, avant la définition de nos propres constructeurs, nous avons pu créer un objet de type `Date` sans lui passer de paramètres ? Il ne vous rester plus qu'arriver à la section concernant les constructeurs *par défaut* (voir 7.2.2).

Plus que pour l'initialisation des membres de la classe, le constructeur est particulièrement utile lorsque l'objet que l'on veut créer requiert d'autres structures de données alloués dynamiquement. Par exemple, une fenêtre graphique est un objet constitué de la fenêtre elle-même et d'autres objets tels que des boutons, des menus et autres gadgets graphiques. Le constructeur d'un tel objet se chargera donc de créer tous les autres objets dont il a besoin.

Imaginons que l'on veuille compléter la classe `Date` avec une chaîne de caractères qui précise un fait marquant associé à un objet de la classe `Date`. L'initialisation des variables d'instances se fait dans l'ordre suivant :

- Initialisation des valeur par défaut en fonction des types des variables d'instances
- Initialisation des valeurs explicitement fournies lors de la définition de la classe
- Appel au constructeur de la classe.

6.1.5 Destructeur

Contrairement aux langage *C++*, nous allons pouvoir, dans beaucoup de cas, ne plus nous soucier de la restitution de l'espace mémoire consommée. *Java* dispose d'un système de *récupération de mémoire* automatique. *Java* estime que l'espace occupé par objet peut être restitué au système quand il n'y a plus aucune référence vers cet objet.

Par défaut, le *récupérateur de mémoire* fonctionne en arrière plan pendant l'exécution d'un programme *Java*. Il est possible de supprimer cette récupération en donnant l'option `-noasyngc` sur la ligne de commande du lancement de la machine virtuelle. La récupération de mémoire peut alors être invoquée explicitement pas le programmeur à des moments bien précis avec la méthode `System.gc()`.

Avant l'appel effective à la récupération d'un objet, la machine virtuelle *Java* fait appel à la méthode `finalize`. A quoi donc peut servir cette méthode puisque la récupération de mémoire se charge de tout ? La raison en est simple : *Java* peut s'occuper de la récupération des objets *Java* et rien d'autre.

Par exemple, un programme qui utilise des ressources systèmes (les descripteurs de fichiers, les *sockets*, etc.), c'est lors de l'invocation de la méthode finalize que le programmeur se chargera de libérer ses ressources ; *Java* ne pourra pas le faire tout seul. Une classe qui contient la méthode `finalize` devra avoir le squelette suivant :

```
protected void finalize() throw Throwable {
    super.finalize() ;
    // Code propre aux objets de cette classe.
    ...
}
```

Prenez ce bout de code tel quel, même s'il y a beaucoup de choses mystérieuses. Après la lecture des chapitres sur l'héritage et les exceptions, tout ceci devrait s'éclaircir.

6.2 Définitions de champs

Les objets des classes que nous avons définis avaient leur propre jeu de données *privées* et *publiques*. Par exemple, les objets de la classe `Date` possèdent chacun leur propre champ `jour`, `mois` et `année`.

Il existe des cas où il est souhaitable d'avoir une même donnée qui soit commune à tous les objets d'une classe. Un champ d'une classe est dit `static` lorsqu'il n'y a qu'un exemplaire de ce champ pour l'ensemble des objets de cette classe. Ce champ existe même s'il n'existe aucun objet de cette classe. Les champs `static` sont parfois appelés *variables de classe* commence à exister à partir du moment ou une classe est initialisée.

```
class Date {
    private int mois ;
    private int jour ;
    private int  année ;
    public static int nbDate = 0 ;
```

```
        public Date(int m, int j, int a) {
            mois = m ; jour = j ; année = a ;
            nbDate++ ;
        }
        ...

        public static void main(String args[]) {
            Date noel97 = new Date(25, 12, 97) ;
            Date dateDeNaissance = new Date(15, 9, 57) ;
            noel97.imprimer() ;
            dateDeNaissance.imprimer() ;
            System.out.println(noel97.nbDate) ;
            System.out.println(dateDeNaissance.nbDate) ;
        }
    }
```

Dans cet exemple, Les champs **jour**, **mois** et **année** des objets **noel97** et **dateDeNaissance** sont des *variables d'instance* ; autrement dit, chacun de ces objets possède leur propre instance de ces champs. La modification d'un de ces champs pour un objet donné n'influe pas sur ce même champ de l'autre objet. Par contre, le champ **nbDate** est déclaré **static**, c'est donc une *variable de classe*. Les deux objets **noel97** et **dateDeNaissance** partagent alors la même structure de donnée. Si l'on modifie la valeur de ce champ à partir d'un objet, cette modification est valide pour l'autre objet. C'est ainsi que dans cet exemple, on peut compter le nombre d'objets de type **Date** que l'on crée en incrémentant la valeur du champ **nbDate** dans le constructeur de la classe **Date**. On obtient donc le résultat suivant :

```
25/12/97                          noel97.imprimer()
15/9/57                           dateDeNaissance.imprimer()
2                                 System.out.println(noel97.nbDate) ;
2                                 System.out.println(dateDeNaissance.nbDate) ;
```

6.2.1 Initialisation des champs static

Les champs **static** sont initialisés une fois lors de l'initialisation de la classe qui les contient. Une erreur de compilation se produit lorsque

- une variable de classe est initialisée par référence à une variable de classe définie plus loin (textuellement) dans la définition de la classe.

```
class X {
static int x  = y  + 1 ;         // Erreur ! y est déclaré après x
static int y = 0 ;               // O.K.
static int z = z+1 ;             // Erreur !
}
```

- une variable de classe est initialisée par référence à une variable d'instance de la classe.

```
class X {
    public int x = 120 ;
    static int y  = x + 10  ;    // x est une variable d'instance
}
```

6.2.2 Initialisation des champs non-static

Les champs non **static** (variables d'instance) sont initialisés lors de la création des objets (des instances) de la classe. Contrairement aux champs **static**, chaque création d'objets provoque l'initialisation des variables d'instances de cet objet. Une erreur de compilation se produit lorsque une variable d'instance est initialisée par référence à une variable d'instance définie plus loin (textuellement) dans la définition de la classe. On pourra utiliser les valeurs des variables de classe pour initialiser les variables d'instance puisque la création et l'initialisation de la classe se fait bien avant la création des objets de la classe.

```
class X {
     int x  = y  + 1 ;             // Erreur !
     int y = 0 ;                   // O.K. !
     int z = z+1 ;                 // Erreur !
}
```

6.2.3 Le mot clé this

Le mot clé **this** désigne l'objet sur lequel la méthode est invoquée. Par exemple, la méthode **affecter** peut se réécrire de la manière suivante :

```
public void affecter(int m, int j, int a) {
    this.mois = m ; this.jour = j ; this.annee = a ;
}
```


www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

L'intérêt du mot clé **this** n'est évident pas dans ce cas là ; par contre si l'on voulait créer une liste de toutes les objets de type **Date** crées, on ne pourra se passer de ce mot clé **this** pour créer le chaînage.

```
class Date {
    private int mois ;
    private int jour ;
    private int  année ;
    private Date suivant ;
    public static Date listeDates = null ;
    public Date(int m, int j, int a) {
        mois = m ; jour = j ; année = a ;
        suivant = listeDates ;
        listDates = this ;
    }
    ...
}

class Test {
    public static void main(String args[]) {
        Date noel97 = new Date(25, 12, 97) ;
        Date dateDeNaissance = new Date(15, 9, 57) ;
        for (Date d = Date.listeDates ; d != null ; d = d.suivant)
            d.imprimer() ;
    }
}
```

6.2.4 Champs final

Un champ d'une classe peut être qualifié de **final**. Ce qualifier indique au compilateur que ce champ ne peut être modifié et gardera tout au long de son existence une valeur constante. Le compilateur produira donc une erreur lorsqu'il y aura une tentative de modification de la valeur de ce champ. L'intérêt de ce qualifier est triple :

- C'est une aide à la programmation. En précisant que la valeur de cette donnée ne peut changer, on se prémunit de certaines erreurs de programmation.
- C'est une aide pour le compilateur. En effet, sachant que le champ conservera une valeur constante tout au long du programme, le compilateur peut effectuer toutes les optimisations pour produire un code efficace.
- Les champs **final** sont en fait des constantes et peuvent apparaître partout où une constante est attendue. *********

Si l'on ne peut modifier la valeur de champ, comment lui donner une valeur initiale ? Et bien, en l'initialisant ! En effet, seule l'initialisation de ce champ est permise ; s'il s'agit d'une donnée primitive c'est une initialisation classique et s'il s'agit d'une référence alors l'initialisation lui affectera une référence un objet. Une référence qualifiée de **final** n'interdit pas la modification de l'objet référencé ; l'objet pourra être modifié mais la référence désignera toujours le même objet.

Tous les champs peuvent être qualifiés de **final**, que ce soit des variables de classes ou des variables d'instance.

6.3 Définitions de méthodes

6.3.1 Passage de paramètres

Lors des appels aux méthodes, tous les paramètres sont passés *par valeur*. Le concept de *passage par adresse* n'existe pas. Rappelons que les seuls types possibles de paramètres sont les types primitifs et les références. Autrement dit,

- les types primitifs (les entiers, les booléens et les flottants) sont passées toujours par valeur. C'est la valeur du paramètre (i.e. la copie d'une constante ou du contenu d'une variable) qui est passée en paramètre à la méthode invoquée. Une méthode ne peut donc jamais modifier la valeur d'une variable de type primitif du code appelant.
- les références également sont passés par valeur. Ce qui est passé ici en paramètre, c'est la valeur de la référence et jamais l'objet lui-même. Une méthode peut donc modifier cette copie de la valeur de la référence sans que cela modifie la valeur de la référence du code appelant. Par contre, si la méthode modifie un champ de l'objet référencé par cette valeur, c'est l'objet (qui lui n'est peut être passé par valeur) qui est modifié pour tout référence vers cet objet. Le code appelant se voit donc l'objet référencé modifié.

6.3.2 Signature et polymorphisme

Contrairement aux langage *C*, un même identificateur peut être utilisé par désigner deux méthodes à condition que leur *signature* soit différente. On appelle *signature* d'une méthode, la donnée de son nom, du nombre de ses paramètres formels et de leur type.

```
int une_méthode(int i)  { ... }          // Erreur ! Le retour de la méthode
float une_méthode(int i)  { ... }        // ne fait pas partie de la signature

int une_méthode(int i)  { ... }          // O.K. !
float une_méthode(float f)  { ... }      // Ces deux méthodes ont des signatures distinctes

int une_méthode(int i)  { ... }          // O.K. !
int une_méthode(int i, int j)  { ... }  // Ces deux méthodes ont des signatures distinctes
```

6.3.3 Variables locales

Les *variables locales* sont allouées lors de l'invocation de la méthode et sont détruites à la fin de celle-ci. Ces variables ne sont visibles qu'à l'intérieur de la méthode.

Les variables locales doivent avoir été affectées avec leur utilisation ; dans le cas contraire, une erreur de compilation est engendrée. Cette valeur peut être donnée par initialisation (à la définition de la variable) ou par affectation. Ces exigences, loin d'être des contraintes, sont des aides précieuses pour le programmeur !

```
void une_méthode() {
    int i  ; j , k ;
    j = i ;                      // Erreur de compilation !
    if (...) {
         k = 1 ;
    }
    j = k ;                      // Erreur de compilation !
}
```

Lorsqu'une méthode est invoquée par différents *threads* (voir 22), chaque thread possède ses propres variables locales et paramètres.

Un objet référencé par une variable locale, peut continuer à exister après la fin de la méthode, même si cet objet a été crée dans cette méthode. Cet objet sera restitué au système que lorsqu'il n'y a plus aucune référence vers cet objet.

6.3.4 Méthodes static

Jusqu'à présent, les méthodes que nous avons vues s'appliquent toujours sur un objet ou plus exactement sur une référence vers un objet. Les méthodes qu'on qualifie de `static` sont celles qui n'ont pas besoin d'un objet pour être invoquée. Ces méthodes se rapprochent des fonctions classiques du langage *C* et sont appelés *méthodes de classe*.

Comme toutes les méthodes, une méthode `static` est toujours membre d'une classe ; elle est invoquée en lui associant, non pas un objet, mais la classe à laquelle elle appartient. Par exemple, la méthode `sqrt` qui calcule la racine carrée d'un nombre, appartient à la classe `Math`. Pour invoquer cette méthode, on utilisera la syntaxe suivante :

```
Math.sqrt(x) ;          // Math désigne non pas un objet, mais une classe
```

Une méthode `static`, puisqu'elle ne s'applique pas sur un objet, ne peut accéder aux variables d'instances (sauf de celles passées en paramètre). De même, le mot clé `this` n'a pas de sens dans une méthode `static`.

```
class Date {
    private int mois ;
    private int jour ;
    private int  année ;
    private Date suivant ;
    private static Date listeDates = null ;
    public Date(int m, int j, int a) {
        mois = m ; jour = j ; année = a ;
        suivant = listeDates ;
        listDates = this ;
    }
    ...
    public static void listerDates()  {
        for (Date d = Date.listeDates ; d != null ; d = d.suivant)
            d.imprimer() ;
    }
}

class Test {
    public static void main(String args[]) {
        Date noel97 = new Date(25, 12, 97) ;
        Date dateDeNaissance = new Date(15, 9, 57) ;
        Date.listerDates() ;
    }
}
```

6.4 Bloc d'initialisation static

Les variables statiques peuvent être initialisées lors de leur déclaration. Il est parfois utile de disposer, non pas de simples initialisations, mais d'un ensemble d'instructions plus complexes pour réaliser l'initialisation des champs statiques. On peut faire le parallèle avec les variables d'instances. Celles-ci disposent du constructeur pour effectuer des initialisations complexes. De même, les *blocs d'initialisation statique* permettent d'effectuer des initialisations complexes sur les champs statiques.

Contrairement aux constructeurs, les blocs d'initialisation statique n'ont pas une syntaxe proche d'une méthode. En effet, ces initialisations ont lieu au moment du **`chargement d'une classe`** et c'est le système qui se charge d'invoquer ces initialisations automatiquement au chargement. Il n'est donc pas question de passer des paramètres à ces initialisations. De plus, comme pour les constructeurs, les initialisations ne retournent aucune valeur. Bref, les *blocs d'initialisation* peuvent être comparées à des méthodes sans paramètres et sans valeur de retour ; il n'est donc pas utile de les nommer. C'est ainsi que la syntaxe des initialisations statiques ne ressemble en rien à des méthodes ; il ne s'agit que de blocs d'instructions préfixés par le mot clé **`static`**.

```
class A {
   ...
   static {
     ... // code d'initialisation static
   }
   ...
}
```

6.5 Conversion des types primitifs en objets et inversement

Comme nous le verrons plus en détails dans le chapitre 12, il n'est pas possible de transformer les types primitifs en objets par une opération de changement de type (*cast*). Par contre, *Java* définit pour des classes spéciales pour un certain nombre de types primitifs (**`Integer`**, **`Float`**, **`Boolean`**, etc.. Par exemple, on créera une instance de la classe **`Integer`** ayant pour valeur 10 de la manière suivante :

```
Integer instanceInteger = new Integer(10);
```

Inversement la classe **`Integer`** dispose de méthode qui permettent d'obtenir la valeur du champ entier d'une instance de cette classe.

```
int i = instanceInteger.intValue();    // retourne 10
```

Une description détaillée de ces classes est donnée en 19.2.

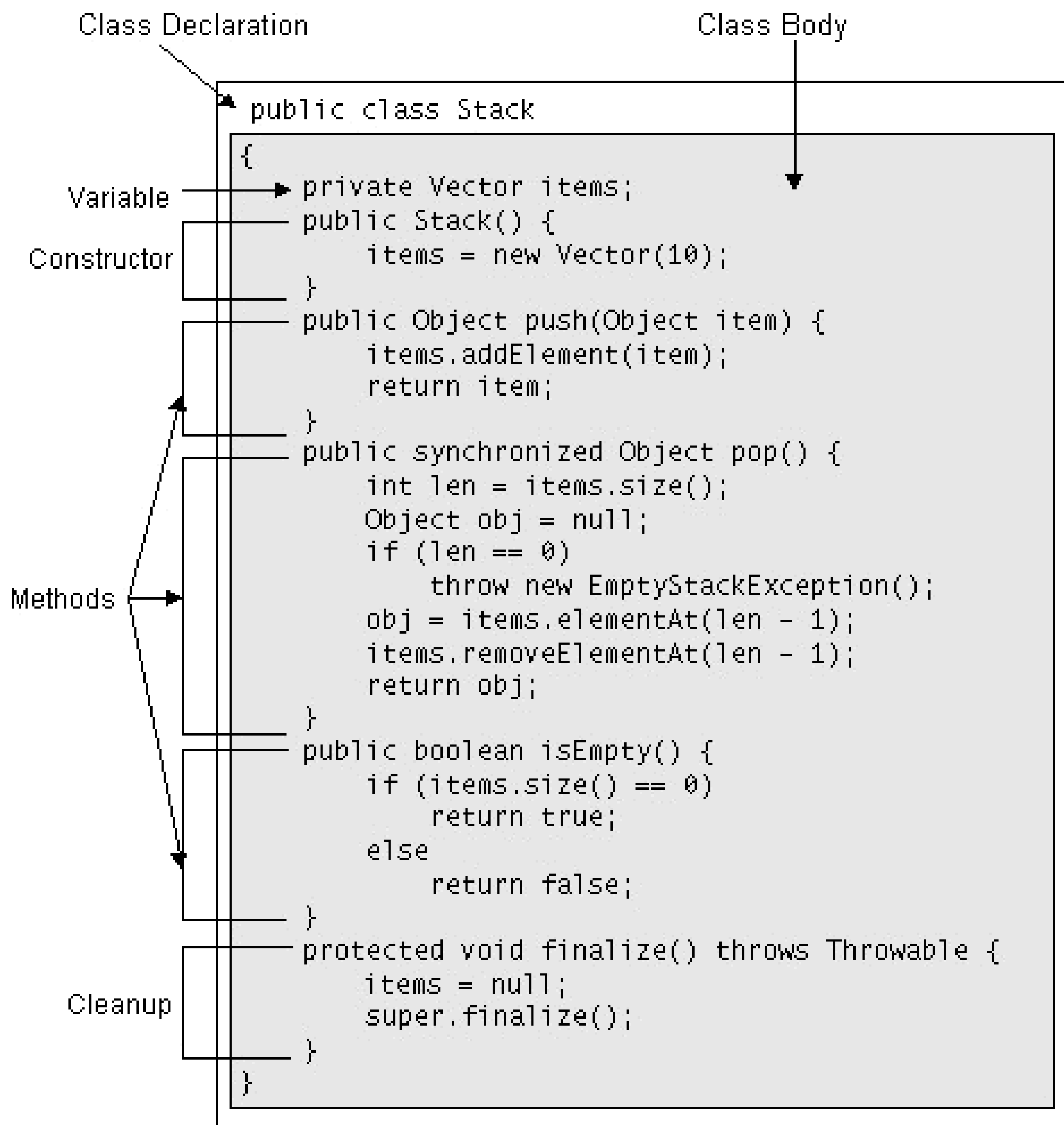

Fig. 6.2: Exemple de classe

7. Héritage

Sommaire

7.1 Introduction . . . 55
7.2 Constructeur de la sous classe . . . 56
7.2.1 Invocation du constructeur de la classe de base . . . 56
7.2.2 Constructeur par défaut . . . 56
7.2.3 L'enchaînement des constructeurs . . . 57
7.2.4 Redéfinition des champs . . . 57
7.3 Redéfinition des méthodes . . . 57
7.3.1 Méthodes d'instances . . . 57
7.3.2 Méthodes de la classe de base . . . 58
7.3.3 Méthodes static . . . 58
7.4 Destructeurs . . . 58
7.5 Méthodes et classes finales . . . 59
7.6 Conversion entre classes et sous classes . . . 59
7.7 Classes et méthodes abstraites . . . 59
7.8 La classe Object . . . 60

7.1 Introduction

Un des grands intérêts des langages orienté objet, c'est de pouvoir définir des dépendances entres classes. Cela permet, en particulier, de réaliser des programmes parfaitement modulaires en disposant de modules réutilisatsables.

En reprenant l'exemple de notre classe `Date`, supposons que l'on veuille définir une classe `DateEvénement` qui associe à une date donnée, un évènement qui la caractérise. La solution triviale serait de définir la classe `DateEvénement` en redéfinition entièrement cette classe et ce en prenant exemple sur la classe `Date` et en rajoutant les nouvelles fonctionnalités que l'on juge utile.

```
classe DateEvenement {
    private int jour, mois, année ;
    private String event = null ;
    public DateEvenement(int m, int m, int j, String e) {
        jour = j ; mois = m ; année = a ; event = e ;
    }
    public affecter(int m, int m, int j, String e) {
        jour = j ; mois = m ; année = a ; event = e ;
    }
    ...
    public void imprimer() {
        System.out.println(jour + "/" + mois + "/" + année + "->" + event) ;
    }
}
```

Cette approche est à l'opposé de l'esprit de la programmation objet et du génie logiciel. Dans la mesure où l'on dispose déjà de la classe, les langages orientés objets offre un moyen bien plus simple pour définir cette nouvelle classe. Il s'agit de *l'héritage*.

L'*héritage* est une caractéristique des langages orientés objet. Une classe obtenue *par héritage* possèdent la totalité des membres de la classe de base ainsi toutes ses méthodes. Une classe `B` peut donc se définir par rapport une autre classe `A`. On

dira que la sous classe `B` hérite des attributs et fonctionnalités de la *classe de base* `A`. L'ensemble des classes sont organisés de manière hiérarchique permettant de structurer l'ensemble des informations manipulées par un programme.

Limiter une *sous classe* aux caractéristiques de la classe de base n'a évidemment aucun intérêt. Une *sous classe* complètent, en quelque sorte, la classe de base. Elle se doit de définir des attributs et des services supplémentaires. Elle pourra modifier, éventuellement, le comportement d'une des méthodes de la classe de base.

Reprenons notre exemple de la classe `DateEvenement` : cette classe possède beaucoup de caractéristiques de la classe. En ce qui concerne les champs, seul un champ supplémentaire `event` est rajoutée à cette classe. Quant aux méthodes, le constructeur et les autres méthodes de cette nouvelle classe doivent être complétés ou adaptées en fonction de l'ajout de ce nouveau champ. Une partie importante du code écrit pour la classe `Date` reste inchangée. On peut donc définir la classe `DateEvenement` par dérivation ou extension de la classe `Date` et ce avec le mot clé `extends`.

```
class DateEvenement extends Date {
   ...
}
```

Avec de cette définition, les objets de la classe `DateEvenement` possèdent les champs et méthodes de la classe `Date`. Il appartient au concepteur de cette nouvelle de définir les champs et méthodes propre à cette classe. Dans notre exemple, la classe `DateEvenement` doit définir un champ `event` et les méthodes d'accès et de modification de ce champ.

```
class DateEvenement extends Date {
    private String event = null ;
    ...
    public String quelEvent() { return event ; }
    ...
}
```

7.2 Constructeur de la sous classe

7.2.1 Invocation du constructeur de la classe de base

Lorsqu'on définit une *classe dérivée*, il faut s'assurer que, lors de la création des objets de cette nouvelle classe, les champs propres à cette classe dérivée ainsi que les champs de la classe de base soient initialisés correctement. Les champs de la classe `Date` sont des champs privés, la sous classe ne peut, en aucun cas, se charger toute seule de l'initialisation de ces membres. Les constructeurs d'une classe dérivée devront forcément utiliser, pour les champs qui ne sont propres à cette classe, les constructeurs de la classe de base. Pour invoquer le constructeur de la classe de base, on fera appel à l'instruction `super(...)`. Un constructeur d'une classe dérivée se compose généralement deux parties : celle concernant les champs de la classe de base et celle concernant les champs propres de la classe dérivée.

L'invocation de `super(...)` doit être la première instruction du constructeur de la classe dérivée.

```
class DateEvenement extends Date {
    ...
    public DateEvenement(int j, int m, int année, int e) {
        super(j, m, a) ;        // appel au constructeur de la classe de base
        event = e ;             // Initialisation des champs propres de la classe dérivée.
    }
    ...
}
```

L'ordre dans lequel les différents constructeurs et initialisations sont effectués est le suivant :

- appel au constructeur de la classe de base
- initialisation des champs en fonction de celles définies dans la déclaration des champs
- exécution du corps du constructeur de la classe dérivée

7.2.2 Constructeur par défaut

Si le constructeur de la classe dérivée n'invoque pas le constructeur de la classe de base explicitement avec l'instruction super(...), *Java* fait quand même appel au constructeur, sans argument, de la classe de base : super(). Un constructeur définit comme suit

```
public DateEvenement(String e) {event = e ; }
```

est automatiquement transformé en

```
public DateEvenement(String e) { super() ; event = e ; }
```

Dans le cas où un tel constructeur n'existe pas dans la classe de base, une erreur de compilation est générée. Il existe un cas où l'absence de l'instruction `super(...)` ne conduit pas cet appel implicite : celui où le corps du constructeur commence par l'instruction `this(...)`.

Si aucun constructeur n'est défini dans la classe dérivée, un constructeur sans argument est quant même invoqué. Tout se passe comme si un constructeur implicite était défini ; constructeur de la forme :

```
public DateEvenement() { super() ; }
```

7.2.3 L'enchaînement des constructeurs

Pour tout objet crée, le constructeur de la classe de base est invoqué qui lui a son tour invoque le constructeur de sa classe de base et ainsi de suite. Il existe donc en enchaînement d'invocation de constructeurs. Cette cascade d'appels aux constructeurs s'arrête dès que l'on atteint le constructeur de la classe `Object`.

La classe `Object` est la mère de toutes les classes ; toute classe est dérivée directement ou indirectement de la classe `Object`. Ainsi, lors de la création d'un objet, le premier constructeur invoqué est celui de la classe `Object` suivi des autres constructeurs dans l'ordre de la hiérarchie de dérivation des classes.

7.2.4 Redéfinition des champs

Les champs déclarés dans la classes dérivée sont toujours des champs supplémentaires. Si l'on définit un champ dans la sous classe ayant le même nom qu'un champ de la classe de base, il existera deux champs de même noms. le nom de champ désignera toujours le champ déclaré dans la classe dérivée. Pour avoir accès au champ de la classe de base, il faudra changer le type de la référence pointant sur l'objet ou en utilisant le mot clé `super`.

```
class A {
    public int i ;
    ...
}
class B extends A {
    public int i ;
    ...
    public void uneMethode() {
        i = 0 ;                     // i est le champ défini dans la classe B
        this.i = 0 ;                // i est le champ défini dans la classe B
        super.i = 1 ;               // i est le champ défini dans la classe A
        ( (A) this ).i  = 1         // i est le champ défini dans la classe A
        ...
        }
}
```

Cette technique peut s'appliquer en cascade de la manière suivante :

```
class C extends B {
    public int i ;
    ...
    public void uneMethode() {
        i = 0 ;                      // i est le champ défini dans la classe C
        this.i = 0 ;                 // i est le champ défini dans la classe C
        super.i = 1 ;                // i est le champ défini dans la classe B
        ( (B) this ).i  = 1          // i est le champ défini dans la classe B
        ( (A) this ).i  = 1          // i est le champ défini dans la classe A
        ...
    }
}
```

Par contre, l'instruction suivante est incorrecte :

```
super.super.i = 1 ;                   //  Incorrect syntaxiquement !
```

Tout comme l'utilisation du mot clé this, le mot clé super ne peut être utilisé dans les méthodes static.

7.3 Redéfinition des méthodes

7.3.1 Méthodes d'instances

On n'est, évidemment pas, tenu de déclarer des nouveaux champs dans une classe dérivée : il est tout possible que l'on dérive une classe pour uniquement modifier les méthodes de la classe de base. Par exemple, si l'on voulait une classe nouvelle classe `DateAnglais` qui ne diffère de la classe Date que par le format d'impression de la date (impression du mois avant celui du jour), il suffirait de définir une classe dérivée de la classe Date et de redéfinir la méthode imprimer pour cette nouvelle classe.

La redéfinition d'une méthode consiste à fournir une implantation différente de la méthode de même signature fournie par la classe de base. Dans cet exemple, la méthode imprimer des classes Date et `DateAnglais` ont la même signature ; celle de la classe `DateAnglais` redéfinit celle de la classe `Date`.

```
class DateAnglais extends Date {
    public void imprimer() {
        System.out.println(quelMois() + "/" + quelJour() + "/" + quelAnnée() + "/" + event) ;
        }
}
```

La rédéfinition d'une méthode ne concerne que méthodes ayant la même signature dans la classe de base et la sous classe.

La redéfinition des méthodes est un mécanisme puissant : le parallèle avec la redéfinition des champs peut être trompeur. En effet, lorsqu'une méthode est redéfinie, on ne peut invoquer la méthode définie dans la classe de base par un simple changement de type de la référence.

```
class Fruit {
    public void quiEsTu() { System.out.println("Je suis un fruit") ;
}
class Pomme extends Fruit {
    public void quiEsTu() { System.out.println("Je suis une pomme") ;
}
class poire extends fruit {
    public void quiEsTu() { System.out.println("Je suis une poire") ;
}
class test {
    public void main(String args[]) {
        Pomme pm = new Pomme() ;
        Poire pr = new Poire() ;
        Fruit f ;
        f.quiEsTu() ;                   // Je suis un fruit
        f = (Fruit)pm ;
        f.quiEsTu() ;                   // Je suis une pomme
        f = (Fruit)pr ;
        f.quiEsTu() ;                   // Je suis une poire
    }
}
```

Dans cet exemple, même en changement de le type de la référence de l'objet `Pomme` et `Poire` en une référence vers un `Fruit`, la méthode `quiEsTu` invoqué est toujours celle de l'objet référencé. Ce comportement peut paraître surprenant de prime abord ; mais on se rend vite compte ce comportement est tout à fait souhaitable. Par exemple, si l'on dispose d'un tableau de fruits (des pommes et des poires), les éléments de ce tableau sont des pommes et des poires mais sont définis dans la définition du tableau, comme des fruits. Pourtant, sans ce comportement, il nous serait impossible de connaître la nature exacte des fruits rangés dans le tableau.

7.3.2 Méthodes de la classe de base

Pour avoir accès à une méthode redéfinie de la classe de base, à l'intérieur d'une méthode de la classe dérivée, il faudra utiliser le mot clé `super`. Comme pour les champs redéfinis, il suffit de préfixer le nom de méthode par le mot clé `super` pour invoquer la méthode de la classe de base.

```
class DateEvenement extends Date {
    private String event = null ;
    public DateEvenement(int j, int m, int a, int e) {super(j, m, a) ; event = e ;   }
    public void imprimer() { super.imprimer();  System.out.println(e); }
    ...
}
```

7.3.3 Méthodes static

Une méthode `static` peut également être redéfinie par une autre méthode `static`. Par contre, une static ne peut être redéfinie en une méthode non `static`.

7.4 Destructeurs

Contrairement aux constructeurs, les destructeurs ne sont invoqués en chaîne. Il appartient au programmeur, s'il le juge utile, de réaliser cette chaîne de destructeur lui-même et ce à l'aide du mot clé `super`.

```
class B extends A {
    public finalize() {
        super.finalize() ;
        // Code de finalize pour la classe dérivée....
        ...
    }
}
```

Rappelons que dans la majorité des acs, on ne se souciera pas des destructeurs puisque *Java* dispose d'un système de *récupération de mémoire* (voir 6.1.5) automatique.

7.5 Méthodes et classes finales

Une méthode est **final** si elle ne peut être redéfinie dans aucune des sous classes. Ainsi, le concepteur de la classe de base exprime son souhait de figer définitivement l'implantation de cette méthode.

```
class Date {
    private int jour, mois, année ;
    ...
    public final quelJour() { return jour ; }
    ...
}
```

On peut également décider figer la définition d'une classe entière en la déclarant **final**. Cela implique qu'il ne sera plus possible de dériver une nouvelle classe à partir de cette dernière. Et comme on peut dériver cette classe, il est évidemment plus possible de rédéfinir les méthodes de cette classe dans une sous classe.

```
final class DateEvenement {  ... }
```

L'intérêt de déclarer **final** les classes et les méthodes est fournir un module sûr. On est assuré que le comportement ne peut être modifié. Une classe finale est une classe en qui on peut avoir confiance. Par contre, celui réduit la souplesse et l'extensibilité des modules proposés. Une meilleure approche est parfois de définir toutes les méthodes finales sans déclarer **final** la classe elle-même. Ainsi, on pourra dériver de nouvelles classes mais changer avoir la possibilités de changer le comportement des méthodes de la classe de base.

Un second intérêt des méthodes et classes finales est le souci d'efficacité. Les méthodes et classes finales permettent au compilateur de produire du code optimisé. En particulier, la détermination statique des méthodes peut être effectuée dans ce cas et parfois même des évaluations partielles de votre programme.

7.6 Conversion entre classes et sous classes

Une opération de cast permet de modifier le type d'une référence. Les changement de nature ne permise que dans des cas bien précis.

Un opération de *cast* a le droit d'affiner le type d'une référence. Par exemple, une référence vers un objet de type **Date** peut être changé en une référence vers un objet de type **DateEvénement**. Ce dont il est question ici, ce n'est que la perception que la machine *Java* a de la référence. L'objet référencé est toujours le même ; on essaye tout juste de faire croire à la machine *Java* que l'objet référencé est d'une autre nature. En aucun cas, l'objet lui ne subit de modification par cette opération de **cast**.

On ne peut changer le type d'une référence en une référence vers un objet d'une classe dérivée que si l'on est sûr que l'objet référencé est bien du type prétendu. Une référence vers un objet de type **DateEvénement** peut être changé en une référence vers un objet de type **Date** que si l'on est assuré que l'objet référencé est effectivement un objet de la classe **DateEvénement**. Sinon, l'erreur **ClassCastException** est générée.

```
Date d;
DateEvenement de;
...
de = d ;                          // Erreur de comilation !
d = de ;                          // O.K.
de = (DateEvenement)d ;           // O.K. d contient une référence vers de
                                  // un objet de type DateEvenement
```

On trouvera de amples détails, dans le chapitre 12, sur l'ensemble des conversions permises par le langage *Java*.

7.7 Classes et méthodes abstraites

On peut parfois utiliser les classes pour définir, non pas un type d'objet bien précis, mais un concept. On pourrait, par exemple, définira une classe **Forme** avec des méthodes abstraites comme **superficie**, **dessiner**, **tourner**, etc. On ne sait pas implanter la méthode **superficie** dans le cas d'une forme générale. Par contre, une fois connue une forme précise (un carré, un cercle, etc.), on sait implanter cette méthode pour cette forme. Autrement dit, un objet de type forme n'a aucun intérêt en soi. Les objets utiles sont les carrés, les cercles, etc. Par contre, il existe souvent des caractéristiques et comportements communs à toutes les formes.

Le concepteur de la classe **Forme** définira donc cette classe en la qualifiant d'*abstraite* (**abstract**). Il regroupera dans cette classe tous les champs communs à toutes les formes ainsi que les méthodes pour lesquelles une implantation est possible. Quant aux méthodes qui ne peuvent être implantées et qui doivent absolument l'être pour des formes bien précises, il les qualifiera d'abstraites.

Une **méthode abstraite** est une méthode dont la définition est supposée être donnée par redéfinition dans les classes dérivées. On considère dans ce cas là que la classe de base ne peut fournir une méthode par défaut.

Il ne sera jamais possible créer un objet de type **Forme**. Les objets qui sont susceptible d'exister sont des formes bien précises : des carrés, des cercles, des lignes, etc. Ces formes effectives, seront des objets des classes obtenus en dérivant la classe **Forme**. Les classe **Carrée**, **Cercle**, **Ligne** etc. seront des classes dérivées de la classe **Forme**.

Une classe abstraite est une classe partiellement implantée i.e. que certaines des méthodes sont abstraites. Le langage *Java* impose de qualifier la classe d'abstraite lorsqu'une de ses méthode est abstraite.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Tout ceci permet de regrouper des données et méthodes communes dans une classe et de spécifier les méthodes qu'une classe dérivée de celle-ci doit absolument implanter. Si l'on définit une sous classes sans implanter toutes les méthodes abstraites de la classe de base, une erreur de compilation est générée.

```
abstract class Forme {
    ...
    public abstract void superficie() ;
    ...
    }
class Carrée extends Forme {
    ...
    public void superficie () { ... }
    ...
}
```

La redondance de ce qualifier **abstract**, à la fois pour la méthode et pour la classe, est là pour permettre au programmeur de déterminer au premier coup d'oeil, en voyant la classe de savoir s'il existe une méthode abstraite dans cette classe. En effet, il ne sera pas nécessaire de parcourir l'ensemble des méthodes d'une classe pour savoir s'il s'agit d'une classe abstraite. Le mot clé **abstract** devant obligatoirement figurer dans l'entête de la classe, il sera donc facile de repérer les classes abstraites. Il n'est pas permis de créer des objets d'une classe abstraite.

7.8 La classe Object

La classe **Object** est la classe de base de toutes les autres classes. Autrement dit, toutes les classes que l'on définit sont implicitement des sous classes de la classe **Object**. Les méthodes définies dans cette classe **Object** peuvent donc être utilisées ou redéfinies.

On peut classer les méthodes publiques de la classe **Object** en deux catégories :

– Les méthodes utilitaires.

```
package java.lang;
public class Object {
    public String toString()
    public boolean equals(Object obj)
    public final native Class getClass()
    public native int hashCode()
    protected native Object clone()
    protected void finalize()  throws Throwable
```

– Les méthodes pour le support des *threads* (**notify**, **notifyAll** et **wait**). Nous les verrons en détail dans le chapitre consacré aux **threads**(voir 22).

```
    public final native void  notify()
    public final native void notifyAll()
    public final native void wait(long timeout)throws InterruptedException
    public final void wait(long timeout, int nanos)throws InterruptedException
    public final void wait()throws InterruptedException
}
```

`public String toString ()`

La méthode **toString** est utilisée pour donner une représentation textuelle d'un objet. Pour reprendre notre exemple de la classe **Date**, au lieu d'avoir une méthode spécifique pour afficher une date avec un format bien particulier avec la méthode **afficher**, on aurait plutôt intérêt à redéfinir la méthode **toString** pour la classe **Date**. Ainsi, on pourra afficher un objet de type Date comme tout autre objet avec la méthode **System.out.print**.

```
public String toString() { return jour  + " /" + mois + " /" +année; }
```

`public boolean  equals ()`

Comme la méthode **toString**, la méthode **equals** rend la valeur true si deux l'objet sur lequel la méthode equals est invoquée est égale à l'objet passé en paramètre. Qu'est ce que l'égalité ? La notion de l'égalité dépend des objets concernés. Par défaut, la sémantique de la méthode **equals** fournie dans la classe **Object** est l'égalité entre les valeurs des références : les deux références désignent le même objet. Cette méthode **equals** est redondante par rapport au simple test d'égalité que l'on effectue avec l'opérateur **==**. Par exemple, on pourrait vouloir définir une méthode **equals** pour la classe **Date** dont la sémantique porte, non pas sur les valeurs des références mais, sur les valeurs des champs des objets. Il suffit alors de redéfinir la méthode **equals** dans la classe **Date** comme suit :

```
public boolean equals(Object  d) {
    if  ( d != null && d instanceof Date) {
        return this.jour == d.jour && this.mois == d.mois && this.année == d.année;
    }
    else return false;
}
```

Avec cette redéfinition de la méthode `equals` appliquée à deux objets de type `Date` rend la valeur `true` lorsque les champs `jour`, `mois` et `année` ont respectivement les mêmes valeurs.

`public final native Class getClass ()`

La méthode `getClass` est une méthode `final` qui ne peut donc pas être redéfinie. Elle retourne la classe à laquelle appartient l'objet sur lequel elle s'applique. La valeur retournée est un objet de la classe `java.lang.Class` (voir 21.1) qui dispose d'un certain nombre de méthode pour examiner et extraire des informations sur les classes.

```
class test {
  public static void main(String args[]) {
    Date d = new DateEvenement(25,12,1997,"Noel 97");
    System.out.println(d.getClass());
  }
}
produit l'affichage suivant~:
class DateEvenement
```

On remarquera qu'il affiche bien `DateEvenement` et non pas `Date`.

`public native int hashCode ()`

La méthode `hashCode` rend un entier représentant le code de hachage de l'objet sur lequel s'applique la méthode. Cet entier est généralement unique pour chaque objet. A priori, il n'est pas nécessaire de redéfinir cette méthode pour les classes que l'on définit. Il existe certaines applications où l'on aimerait faire correspondre un même entier à un ensemble d'objets d'une classe : il s'agit une relation d'équivalence sur les objets. Nous verrons plus loin les détails d'implantation des *hashcodes* avec la classe `HashTable`classe.HashTable(voir 18.5).

`protected native Object  clone () throws CloneNotSupportedException`

La méthode `clone` retourne une copie de l'objet sur lequel la méthode est invoquée. L'implantation de la méthode `clone` pour les objets d'une classe que l'on définit. Lorsqu'on définit une classe, on peut choisir de :

- Permettre le clonage en implantant l'interface `Cloneable` qui qui est constituée d'unique méthode `clone`. Cette méthode peut, si nécessaire, interdire certains clonages en lançant l'exception `CloneNotSupportedException`[1].
- Interdire le clonage en n'implantant pas l'interface `Cloneable` et en implantant la métohde `clone` qui se contente de lancer l'exception `CloneNotSupportedException`.

[1] le chapitre 10 est entièrement conscaré aux exceptions

8. Les interfaces

Sommaire

8.1 Déclaration des interfaces 63

8.1.1 Interface publique 64

8.1.2 Membres des interfaces 64

8.2 Implanter des interfaces 64

8.3 Utiliser des interfaces 64

8.3.1 Interface dérivée 65

8.3.2 Redéfinition des champs 65

8.3.3 Héritage diamant 65

Le langage *Java* ne permet pas de l'*héritage multiple*. Il pallie ce manque par l'introduction des *interfaces*. Le choix délibéré de *Java* de supprimer l'héritage multiple est dicté par un souci de simplicité.

A TERMINER héritage diamant

Pour supprimer les problèmes liés à l'héritage multiple, les *interfaces* sont des "sortes de classes" qui ne possèdent que des champs `static final` (autant dire des constantes) et des méthodes abstraites.

En fait, les interfaces sont un moyen de préciser les services qu'une classe peut rendre. On dira qu'une classe implante une interface `Z` si cette classe fournit les implantations des méthodes déclarées dans l'interface `Z`. Autrement dit, la définition d'une interface consiste se donner une collection de méthodes abstraites et de constantes. L'implantation de ces méthodes devra évidemment être fournie par les classes qui se réclament de cette interface.

Bref, une interface est une classe abstraite dans laquelle, il n'y a ni variables d'instances et ni méthodes non abstraites.

Les interfaces ne fournit pas l'héritage multiple car :

- on ne peut hériter de variables d'instances
- on ne peut hériter d'implantation de méthodes
- la hiérarchie des interfaces est totalement indépendante de celle des classes.

8.1 Déclaration des interfaces

Comme les classes, les interfaces sont constituées de champs et de méthodes ; mais, comme nous l'avons déjà dit, il existe de très fortes contraintes sur la nature des membres d'une interface :

- Toutes les méthodes qui sont déclarées dans cette interface sont abstraites ; aucune implantation n'est donnée dans la définition de l'interface. Toutes les méthodes étant publiques et abstraites, les mots clés `public` et `abstract` n'apparaissent pas : ils sont implicites.
- Toutes les méthodes d'une interfaces sont toujours publiques et non statiques.
- Tous les champs d'une interface sont `public`, `static` et `final`. Ils sont là pour définir des constantes qui sont parfois utilisées dans les méthodes de l'interface. Les mots clés `static` et `final` ne figurent pas dans la définition des champs ; ils sont implicites.

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

```
interface Service {
        int MAX = 1024 ;
        ...
        int une_méthode(...) ;
        ...
}
```

Par convention les noms des interfaces commencent par une lettre majuscule et se terminent par “able” ou “ible”. Dans nos exemples, on n’appliquera pas ces conventions.

8.1.1 Interface publique

Une interface peut être qualifiée de `public`. Une interface `public` peut être utilisée par n’importe quelle classe. En l’absence de ce qualifier, elle ne peut être utilisée que par les seules classes appartenant au même *package* (voir 9) que l’interface.

Contrairement aux classes, on ne peut qualifier une interface de `private` ou `protected`.

8.1.2 Membres des interfaces

Les champs d’une interface sont des champs `static`. Toutes les règles d’initialisation des champs statiques s’appliquent ici.

Les qualifiers `transcient`, `volatile` ou `synchronized` ne peuvent être utilisés pour les membres des interfaces. De même, les qualifiers `private` et `protected` ne doivent être utilisés pour les membres des interfaces.

8.2 Implanter des interfaces

Les interfaces définissent des “promesses de services”. Mais seule une classe peut rendre effectivement les services qu’une interface promet. Autrement dit, l’interface toute seule ne sert à rien : il nous faut une classe qui implante l’interface. Une classe qui implante une interface le déclare dans son entête

```
class X implements Service {
        ...
        int une_méthode(...) {
                ...
        }
        ...
}
```

Par cette déclaration, la classe `X` promet d’implanter toutes les méthodes déclarées dans l’interface `Service`. La classe `X` doit donc fournir l’implantation des méthodes précisées dans l’interface `Service` ; on devra donc trouver dans le définition de cette classe, l’implantation de la la méthode `une_méthode`.

La signature de la méthode doit évidemment être la même que celle promise par l’interface. Dans le cas contraire, la méthode est considérée comme une méthode de la classe et non une implantation de l’interface.

Si une méthode de même signature existe dans la classe mais avec un type de retour différent une erreur de compilation est générée.

8.3 Utiliser des interfaces

Comme pour les classes, on peut définir des références ayant le type d’une interface. Par contre, il n’est pas possible de définir un objet de ce type : *les interfaces sont une sorte de classes abstraites* !

```
Service s ;
```

Qu’est-ce qu’une référence de type interface ? Et bien, c’est une référence contient

- la valeur null, ou
- une référence à une instance d’une classe qui doit forcément implanter cette interface.

Moyennant ces précisions, une référence vers une interface peut être utilisée partout où une référence vers un objet peut être utilisée.

Pourquoi vouloir désigner un objet, non pas par rapport à sa classe, mais par rapport à l’interface qu’elle cette implante ? Voici un exemple d’utilisation. Soit l’interface suivante :

```
interface Représentable {
    void dessiner();
    void tourner(int angle);
    void translater(int depl);
    }
```

permettant de *dessiner* et effectuer une *rotation* ou une *translation* sur des formes géométriques. Chaque forme (un carrée, un cercle, un polygone, etc.) devra implanter cette interface pour faire partie de l’ensemble de formes représentatable.

Imaginons, à présent, une classe `Dessin` veuille gérer un tableau de formes représentables. Quel est le type de ce tableau ?

La première solution consiste à définir un tableau `Object []` en précisant que les éléments sont des instances de la classe `Object` ; ce qui sera toujours vrai puisque la classe `Object` est la classe ancêtre de toute les classes. Cette solution est évidemment à éviter car ce tableau pourra, certes, contenir des formes représentables mais aussi n’importe quelle autre instance. Cette définition est bien trop large !

Comment faire pour restreindre le type des objets, que ce tableau peut contenir, aux seuls instances des classes ayant implanté l’interface `Représentables` ? Pour cela, il suffit de définir un tableau de type interface :

```
class Dessin  {
    private Représentable[] listeFormes;
    ...
}
```

Ainsi, cette classe pour, par exemple, implanter une méthode `dessinerTout` qui dessine l'ensemble des formes :

```
class Dessin  {
    private Représentable[] listeFormes;
    ...
    public void dessinerTout() {
        for (i = 0; i < sommet; i++) listeFormes[i].dessiner()
    }
}
```

8.3.1 Interface dérivée

Tout comme les classes, les interfaces peuvent être organisées de manière hiérarchique à l'aide de l'héritage. Une classe ne peut être dérivée que d'une autre classe ; de même, une interface ne peut être dérivée que d'une autre interface. Mais, contrairement aux classes, une interface peut étendre plusieurs interfaces.

```
interface A extends X {
    ...
}

interface A extends X, Y, Z {
    ...
}
```

Une interface dérivée hérite de toutes les constantes et méthodes des interfaces ancêtres ; à moins qu'un autre champ de même nom ou une autre méthode de même signature soit redéfinie dans l'interface dérivée.

8.3.2 Redéfinition des champs

Tout comme pour les classes, les champs des interface peut être redéfinis dans une interface dérivée.

```
interface A { int info = 1; }
interface B extends A { int info = 2; }
```

La définition du champ `info` dans l'interface `B` masque la définition du champ `info` de l'interface `A`. Pour parler du champ `info` de l'interface `A`, on le notera `A.info`.

8.3.3 Héritage diamant

Un même champ peut être hérité de plusieurs manière pour une même interface :

```
interface A { char infoA = 'A'; }
interface B extends A { char infoB = 'B'; }
interface C extends A { char infoB = 'C'; }
interface D extends B, C { char infoD = 'D'; }
```

Le champ `infoA` est hérité par l'interface `D` de deux manières : une fois par l'interface `B` et une autre fois par celle de `C`. Mais il n'existera pas deux champs `infoA` dans l'interface `D` ; il n'y en aura qu'un.

9. Packages

Sommaire

9.1 Importer des packages . . . 67
9.1.1 Importer une classe . . . 67
9.1.2 Importation toutes les classes d'un package . . . 68
9.1.3 La variable d'environnement CLASSPATH . . . 68
9.1.4 Packages et systèmes de fichiers . . . 68
9.2 Accessibilité . . . 69
9.2.1 Accessibilité des classes . . . 69
9.2.2 Accessibilité des membres d'une classe . . . 69
9.2.3 Accessibilité des membres de sous classes . . . 70
9.3 Convention de nommage de packages . . . 70

Un *package* est un ensemble de *classes*, d'*interfaces* et d'autres *packages* regroupés sous un nom. Il correspondent, en quelque sorte, au concept de *librairies* adapté au langage Java. Un fichier source destiné à faire partie du package `monpackage` doit préciser son appartenance par la déclaration :

```
package monpackage ;
```

Cette déclaration précise que toutes les classes et interfaces définies dans ce fichier font partie du package `monpackage`. Elle doit apparaître avant toute déclaration de classes et interfaces. Le nom d'un package sert de *convention de nommage* : il constitue en sorte de préfixe pour chaque nom figurant dans le fichier source.

Dans le langage *Java*, les variables statiques font forcément partie d'une classe ; le concept de variables globales n'existe pas. De même, toutes les classes font toujours partie d'un package.

9.1 Importer des packages

9.1.1 Importer une classe

Lorsqu'on veut utiliser une classe d'un package, le moyen le plus direct est de nommer cette classes par son absolu (*fully qualified name*) :

```
CoursJava.monpackage.Date d ;
d = new CoursJava.monpackage.Date(15, 9, 57) ;
```

On aura compris que cette manière de faire est bien trop fastidieuse pour qu'on s'en contente. Pour éviter cette lourdeur, *Java*dispose de la directive `import`. Il s'agit de prévenir le compilateur que l'on risque d'utiliser des noms simplifiés pour nos classes et qu'il devra préfixé tout seul les noms de classes quand c'est nécessaire.

```
import CoursJava.monpackage.Date ;
```

Une fois cette classe importée, on pourra désormais utiliser les noms simplifiés pour les classes importées.

```
Date d = new  Date(15, 9, 57) ;
```

Que se passe-t-il quand dans deux packages importés que l'on veut importer, les noms des classes sont identiques. Il faudra alors prendre sont courage à deux mains et utiliser les noms pleinement qualifiés pour résoudre ce conflit.

9.1.2 Importation toutes les classes d'un package

Dans notre exemple `Date`, nous n'avions besoin d'importer qu'une seule classe. Dans des situations plus réalistes, il est fréquent qu'on soit obliger d'importer plusieurs classes. Il serait fastidieux d'importer une à une toutes les classes qui nous intéressent. Par exemple, pour réaliser une application graphique, il serait bien trop fastidieux d'importer, une à une, toutes les classes qui correspondent aux divers *widgets* que l'on veut utiliser. *Java*propose d'importer en bloc, un ensemble de packages par la directive :

```
import FR.univ-mrs.esil.gbm.touraivane.CoursJava.monpackage.* ;
```

Avec cette *directive*, toutes classes du package `monpackage` sont importées et on pourra utiliser les noms simplifiés pour toutes ses classes.

La directive import se décline également sous la forme :

```
import CoursJava.monpackage ;
```

Cette forme permet de nommer les classes de ce package en utilisant uniquement sa dernière composante : la classe

```
CoursJava.monpackage.X
```

peut désormais être nommée par monpackage.X.

9.1.3 La variable d'environnement CLASSPATH

La variable d'environnement `CLASSPATH` contient la liste des répertoires, dans le machine locale, où résident les classes, interfaces et packages. Le chargement des classes accessibles par la variable d'environnement `CLASSPATH` est effectué en prenant quelques libertés par rapport à la sécurité. Ces classes sont chargées une et une seule fois et ne peuvent être modifiées ou remplacées dynamiquement.

Ces classes sont considérées comme "dignes de confiance" car résidant dans la machine locale. Le chargement de ces classes se fait sans faire appel au vérificateur de code ; ceci permet de réaliser des classes avec des optimisations qui contre-viennent au contrôle de sécurité du code *Java*.

Une autre manière de spécifier les répertoires où sont stockés les classes consiste à les donner sur la ligne de commande lors de l'invocation de l'interpréteur *Java*. avec l'option `-classpath`.

9.1.4 Packages et systèmes de fichiers

Les packages peuvent être organisés de manière hiérarchique. Le package `java` de *jdk* est en fait une collection d'autres packages : `lang`, `awt`, etc. Ce package ne contient aucune classe ni interface. Le package `java.lang` est également une collection d'autres packages : `Math`, `String`, etc. Enfin le `java.lang.io` est un package qui contient les classes pour la gestion des entrées/sorties.

Le nom absolu d'un package définit également la structure du système de fichier. Par exemple, la classe `java.lang.Math` se trouve dans le répertoire `java/lang/Math` ou `java`
`lang`
`Math` selon la nature du système d'exploitation utilisé.

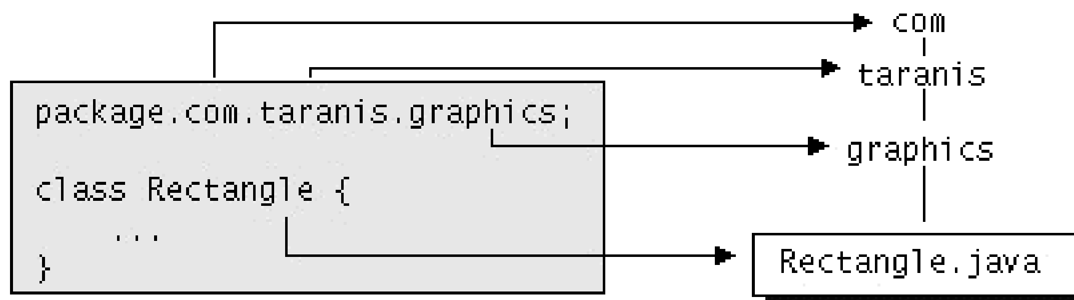

Fig. 9.1: Package

Packages sans nom

Lorsque la directive `package` ne figure pas dans le fichier source, les classes et interfaces font partie d'un package "sans nom" par défaut.

9.2 Accessibilité

. .

Nous avons déjà évoqué les qualifiers `public` et `private` pour les membres des classes ainsi que pour les classes elle-même. Maintenant que nous savons ce que sont les classes, les classes dérivées, les interfaces et packages, nous allons revoir et compléter ces concepts.

9.2.1 Accessibilité des classes

Les classes sans qualifier, comme la classe `Date` que nous avons définie, sont des classes qui accessibles de toutes les classes du package auquel il appartient. Cette appartenance doit se comprendre au sens strict, c'est à dire que la classe `Date` est accessible à toutes les classes de son package mais n'est pas accessible aux classes des packages contenus celui-ci. Pour rendre accessible notre classe `Date` à toutes les autres classes, il faut qualifier de `public` la classe `Date` :

```
public class Date { .... }
```

*Java*impose une restriction sur les nombre de classes publiques que l'on est autorisé à définir dans une unité de compilation : Il ne doit y avoir qu'une seule classe publique par unité de compilation et le nom de ce dernier doit être le même que celui la classe publique suffixé par l'extension "`.java`".

Les classes peuvent être qualifiés de `public` ou ne pas être qualifiée.

Classes non qualifiées : elles peuvent être référencées de n'importe quelle classe du même package.

Classes public : elles peuvent être référencées de n'importe quelle partie du programme : c'est la visibilité maximale.

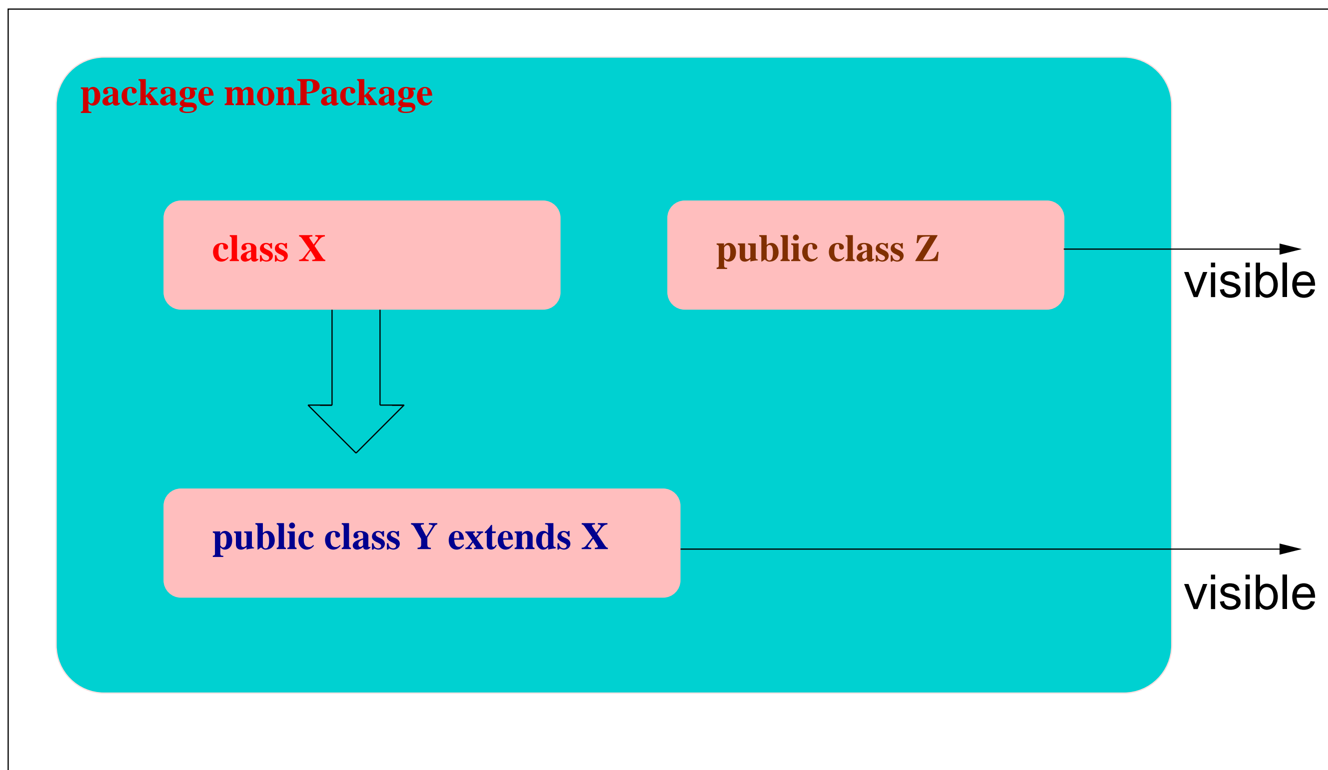

Fig. 9.2: Visibilité des classes

9.2.2 Accessibilité des membres d'une classe

Les membres d'une classes peuvent être qualifiés `public`, `private` ou `protected`. Il peuvent également ne pas être qualifiés.

Membres private : ils ne sont accessibles qu'à l'intérieur de sa propre classe : c'est l'accessibilité minimale.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Membres protected : ils sont accessibles à partir de sa propre classe, de toutes ses classes dérivées et toutes les classes appartenant au même package. Il n'est pas nécessaire que les classes dérivées appartienne au même package que la classe de base.

Membres non qualifié : ils accessibles à partir de toutes les classes du package.

Membres public : ils sont accessibles à partir de n'importe quelle classe de n'importe quel package : c'est l'accessibilité maximale.

Fig. 9.3: Visibilité des classes

9.2.3 Accessibilité des membres de sous classes

Une méthode privée d'une classe de base peut être redéfinie dans une sous classe en un méthode publique. L'inverse n'est pas permis. Par contre, cette restriction n'existe pas pour les variables d'instances et de classe.

9.3 Convention de nommage de packages

La notion de package a pour finalité de résoudre les conflits des noms des classes et interfaces utilisées dans une application *Java*. Il faut garder à l'esprit qu'une application *Java*peut être réalisée par un nombre important de programmeurs répartis à travers le monde. Dans ce contexte, trouver un moyen d'éviter les *collisions de noms* devient vital.

La convention adoptée consiste à préfixer le package par le nom de domaine *Internet* où il a été réalisé. Pour avoir réaliser le package `monpackage` à l'ESIL, on conviendra de la nommer :

```
package FR.univ-mrs.esil.monpackage ;
```

On remarquera que le domaine de plus haut niveau (ici `FR`) est noté en majuscule. Cette convention ne résout évidemment pas tous les collisions de noms ; il ne règle le problème qu'entre les divers domaines *Internet*. Il reste en encore à résoudre les problèmes internes au domaine de l'*ESIL*. Il suffit alors de raffiner cette convention en subdivisant le domaine *ESIL* en sous

domaines. Par exemple, pour s'assurer que le package `monpackage` n'est pas en conflit avec d'autres package de l'*ESIL*, on pourrait décider que celui-ci serait nommé :

```
package FR.univ-mrs.esil.gbm.touraivane.CoursJava.monpackage ;
```

10. Exceptions

Sommaire

10.1 Qu'est-ce qu'une exception 75
10.2 Définir de nouveaux types d'exception 75
10.3 Déclencher des exceptions 75
10.4 Capturer les exceptions 76
10.4.1 Le bloc try 76
10.4.2 Les clauses catch 76
10.4.3 Les clauses finally 76
10.4.4 Les exceptions non capturées 76
10.5 Les classes Error, Exception et RuntimeException 77

La gestion des erreurs sans un programme est une activité importante et souvent complexe dans les langages classiques. Bien des erreurs dans les programmes proviennent d'une mauvaise gestion des erreurs. La méthode utilisée en *C*, par exemple, pour la gestion des erreurs est de tester l'état d'une variable ou d'un retour de fonction pour savoir si tout s'est bien passé. Rien n'oblige un programmeur du langage *C* à tester rigoureusement les conditions d'erreurs. Et bien des programmes ne le font pas de manière exhaustive. Il n'y a aucune aide pour structurer cette gestion d'erreurs.

Avec les *exceptions*, le langage *Java* propose une approche radicalement différente de l'approche traditionnelle. Qu'est ce qu'une *exception* ? Une exception est une sorte de signal qui indique qu'une erreur ou une situation remarquable est intervenue. On dira qu'une méthode ayant détecté une situation anormale déclenche (`throw`) une exception. Cette exception sera capturée (`catch`) par le code.

On distingue deux types d'erreurs : les *exceptions* et les *erreurs*. Les *erreurs* sont généralement des erreurs fatales et le programme s'arrête après déclenchement de ce type d'erreurs. Les *exceptions* ne sont pas constitués uniquement des erreurs produites pas des "erreurs systèmes". Le concepteur d'une classe peut définir des erreurs (non fatales) pour assurer la robustesse du code. Par exemple, le débordement d'un tableau est une erreur qui est déclenchée et qui n'est pas une erreur fatale.

Il existe encore quelques méthodes qui rendent des valeurs que l'on doit interpréter comme des cas d'erreurs (comme en *C*), mais il s'agit généralement des situation ou l'erreur n'est pas très grave.

Lorsqu'une méthode déclenche une exception, la machine *Java* remonte la suite des invocations de méthodes jusqu'à atteindre une méthode qui capture cette exception. Si aucune méthode capturant cette exception n'est trouvée, alors l'exécution s'arrête.

L'utilisation des exceptions permet de

- Séparer le code concernant le fonctionnement normale d'un programme de celui concernant la gestion des erreurs. Dans les langages classiques, la gestion des erreurs et le code concernant le fonctionnement normal d'un programme sont imbriqués. Avec les exceptions, la gestion des exception est complètement séparé. Tiré du tutorial de JDK, voici un exemple qui montre la séparation du code de gestion des exceptions comparé à une programmation sans exception

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
           allocate that much memory;
           if (gotEnoughMemory) {
               read the file into memory;
               if (readFailed) {
                   errorCode = -1;
               }
```

```
            }
            else errorCode = -2;
        }
        else errorCode = -3;
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        }
        else errorCode = errorCode and -4;
    }
    else errorCode = -5;
    return errorCode;
}
```

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    }
    catch (fileOpenFailed) {
        doSomething;
    }
    catch (sizeDeterminationFailed) {
        doSomething;
    }
    catch (memoryAllocationFailed) {
        doSomething;
    }
    catch (readFailed) {
        doSomething;
    }
    catch (fileCloseFailed) {
        doSomething;
    }
}
```

– Propager de proche en proche les exceptions d'une méthode à la méthode appelante jusqu'à atteindre une méthode capable de gérer l'exception. Il n'est pas nécessaire que la gestion d'une exception figure dans la méthode qui effectue l'action ayant conduit au déclenchement de celui-ci. Une méthode peut ignorer la gestion d'une exception à condition qu'elle "retransmette" l'exception à la méthode appelante.

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error) doErrorProcessing;
    else proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error) return error;
    else proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error) return error;
    else proceed;
}
```

```
method1 {
    try { call method2; }
    catch (exception) {  doErrorProcessing; }
}

method2 throws exception { call method3; }

method3 throws exception { call readFile; }
```

– Regrouper par type la gestion des exceptions. Lex exceptions sont regroupées par catégories pour permettre une factorisation de la gestion de ces exceptions.

10.1 Qu'est-ce qu'une exception

Une exception ou erreur est un objet de la classe `java.lang.throwable` et toutes les exceptions et erreurs sont des sous classes de la classe `java.lang.throwable` :

– `java.lang.error` : pour les erreurs

– `java.lang.exception` : pour les exceptions

```
public class Throwable extends Object {
    // Constructeurs publiques
    public Throwable();
    public Throwable(String message);
    // Méthodes publiques
    public Throwable fillnStackTrace();
    public String getMessage();
    public void printStackTrace();
    public void printStackTrace(PrintStream s);
    public String toString();
}

public class Error extends Throwable {
    // Constructeurs publiques
    public Error();
    public Error(String s);
}

public class Exception extends Throwable {
    // Constructeurs publiques
    public Exception ();
    public Exception (String s);
}
```

A chaque objet des classes est associé un message qui peut être obtenu avec la méthode `getMessage()` de la classe `java.lang.throwable`.

10.2 Définir de nouveaux types d'exception

Les exceptions sont des objets d'une classe dérivée de la classe `Exception`. Imaginons que l'on veuille fournir un code sûr pour notre classe `Date`. Par exemple, la méthode `affecter` devrait produire une exception si les entiers fournis en paramètres de cette méthode n'est pas cohérent avec une date. On va, à présent, définir une exception `DateNonConforme`.

```
public class DateNonConforme extends java.lang.Exception {
    public DateNonConforme(String e) {
        super("Date non Conforme:" + e);
        ...
    }
}
```

Pourquoi donc définir une nouvelle classe alors qu'on pourrait pu tout simplement créer un objet de la classe `Exception` ? Les raisons pour lesquelles on préférera créer une nouvelle classe dérivée sont les suivantes :

– Il est parfois utile de disposer d'informations complémentaires sur l'exception. La définition d'une nouvelle classe permet alors de définir des champs pour coder ces informations complémentaires.

– Les exceptions sont filtrées (en vue de leur gestion) en fonction de leur type. Le choix d'une nouvelle classe permet de filtrer un type bien précis d'exception et non pas toutes les exceptions de la classe `Java.lang.Exception`.

– Lorsque que, pour chaque type d'exception, on crée une nouvelle classe, la gestion des exceptions est bien plus simple. Il n'est pas nécessaire de faire une étude de cas pour déterminer quel est le type de l'exception.

10.3 Déclencher des exceptions

Chaque méthode qui est susceptible d'émettre une exception doit le déclarer dans l'entête son entête. Il s'agit des exceptions qui lui sont propres et non pas toutes celles déclenchées par les invocations des méthodes qu'elle contient. Pour reprendre l'exemple de notre classe `Date`, la méthode `affecter` devrait déclencher une exception lorsque ses paramètres ne correspondent pas à une date.

```
public void affecter(int j, int m, int a) throws DateNonConforme {
    ...
    if (m >= 1 && m <= 12) this.mois = m;
    else throw new DateNonConforme(" mois " + m);
    ...
}
```

Le ou les exceptions levées par une méthode doivent absolument être déclarées dans l'entête. Si les exceptions à déclarer dans l'entête sont oubliées ou déclarées de manière incomplète, il y aura une erreur à la compilation. Lorsqu'une méthode lève plusieurs exceptions, elles seront séparées par des virgules dans l'entête de la méthode.

```
void une_methode(...) throws Except1, ..., Exceptn { ... }
```

10.4 Capturer les exceptions

10.4.1 Le bloc try

Les exceptions levées par une méthode doivent être capturées par la méthode appelante. Pour ce faire, on aura pris soin d'encapsuler l'appel de la méthode dans un bloc **try**.

```
...
try {
    ...
    une_methode(...)
    ...
}
catch (typeException1 identificateur1) {
    ...
    }
...
catch (typeExceptionn identificateurn) {
    ...
    }
finally  {
    ...
}
```

La fin du bloc **try** contient toute la gestion des exceptions à l'aide des clauses **catch**. Elle ne sont pas obligatoires ; il peut y avoir un nombre arbitraire de clauses **catch** (éventuellement nul).

10.4.2 Les clauses catch

Les instructions du corps du bloc **try** sont exécutées jusqu'à la fin de ce bloc si aucune exception n'est levée. Dans le cas contraire, le contrôle est transféré à l'une des clauses **catch**. Si aucune clause **catch** n'est prévue pour traiter cette exception, le contrôle est transféré à la dernière des méthodes appelantes englobées dans un bloc **try** et ainsi de suite.

Si une clause **finally** existe à la fin d'un bloc **try**, le contrôle est passé à cette portion du code dans tous les cas :

- Fin normale du corps du bloc **try**
- Fin anormale du bloc **try**
- Sortie du corps du bloc **try** par un **return** ou un **break**

10.4.3 Les clauses finally

La clause **finally** est utilisée lorsque l'état d'un système doit être mise à jour quelque soit la manière dont les instructions se sont exécutées. Par exemple, un bloc de code qui crée un fichier doit fermer ce fichier à la fin du traitement.

Le code contenu dans une clause finally ne peut être ignoré : il n'existe aucun moyen de quitter un bloc **try** sans exécuter le bloc **finally**.

10.4.4 Les exceptions non capturées

Comme nous l'avons déjà dit, une méthode peut ignorer la gestion d'une exception à condition qu'elle "retransmette" l'exception à la méthode appelante. Pour cela, il suffit que la méthode qui reçoit une exception déclare dans son entête qu'elle la retransmet.

```
...
public void une_methode(...) throws DateNonConforme {
    ...
    Date d = new Date(200, 56, -2);
    ...
}
```

10.5 Les classes Error, Exception et RuntimeException

Toutes les erreurs et exception sont des sous classes de la classe `Throwable`. La figure 10.1 donne la hiérarchie des ces classes.
`Throwable` Les seules exceptions qu'il n'est pas nécessaire de préciser dans la clause `throws` sont les `RuntimeException` et `Error` et leurs extensions.

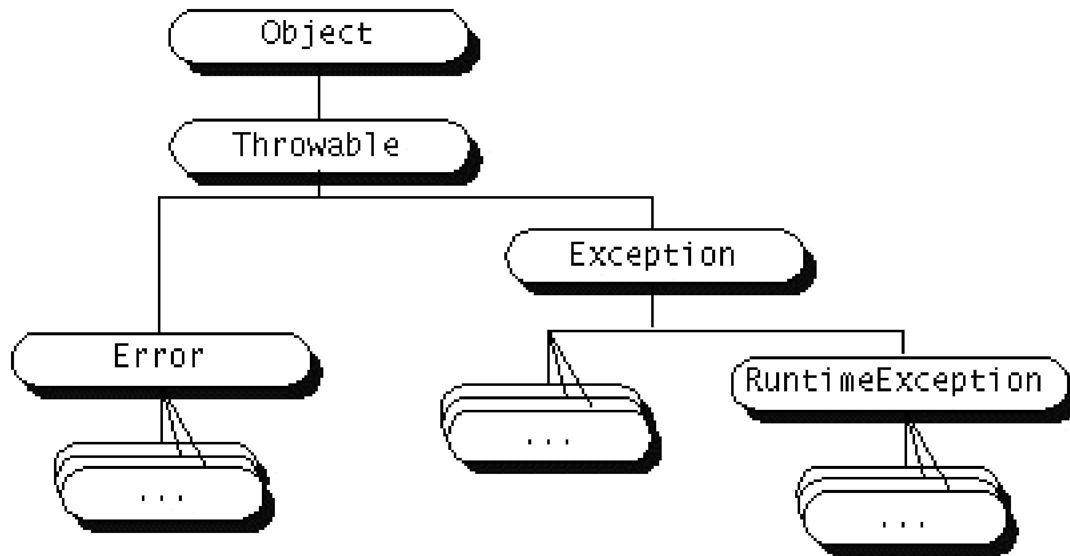

Fig. 10.1: La classe Throwables et ses classes dérivées

Les objets de la classe `Error` sont des erreurs relativement grave et une application n'est pas sensée la capturée. De même, une application Java n'est sensée lancer ce type d'erreurs.

Une application *Java* ne travaille généralement qu'avec des objets de la classe `Exception`. Dans ce cas, il s'agit "d'erreurs" au sens large, qu'une application est susceptible de lancer. Comme nous l'avons vu, on crée des sous classes pour définir des exceptions propres à chaque application.

Parmi ces sous classe, il en existe une prédéfinie qui est la classe `RuntimeException`. Ce type d'exception à la particularité de pouvoir être ignoré par les applications. Contrairement aux exceptions pures, il n'y aura pas d'erreurs de compilation si ce genre d'exception ne sont pas capturées.

11. Tableaux et chaînes de caractères

Sommaire

11.1 Tableaux 79
11.1.1 Type des éléments *79*
11.1.2 Accès aux éléments *79*
11.1.3 Taille des tableaux *80*
11.1.4 Initialisation *80*
11.1.5 Tableaux multidimentionels *80*
11.2 Chaînes de caractères 80
11.2.1 La classe String *80*
11.2.2 La classe StringBuffer *84*

11.1 Tableaux

Les *tableaux* sont des suites d'objets de même type. Le nombre d'éléments de cette est fixe et est appelé *taille* du tableau. Les éléments des tableaux sont soit de type numérique soit des références. Les *tableaux* sont des objets ; un objet de type tableau se définit donc en définissant une variable de type référence.

```
int [ ] tab1;
int tab1[ ];
```

Ces variables sont des références ; l'espace mémoire nécessaire pour coder la suite des objets des tableaux se réserve avec le mot clé `new` et l'opérateur `[]`.

```
tab1 = new int [ 200 ] ;
```

Contrairement au langage *C*, il n'est pas nécessaire que la taille du tableau soit textuellement une constante. Comme il s'agit d'une allocation dynamique, la taille peut être une expression dont la valeur est un entier positif ou nulle.

```
tab2 = new int [ 2 * nbre + 3];
```

11.1.1 Type des éléments

Les éléments d'un tableau peuvent être de n'importe quel type : type primitif ou référence. En particulier, on peut très bien définir un tableau de références vers une classe abstraite. De même, les éléments d'un tableau peuvent être des objets qui implante une interface.

```
Forme [] tab1 = new Forme[10];
Runnable [] tab2 = new Runnable[10];
```

Dans le premier cas, on initialisera le tableau avec des références vers objets d'une classe dérivée de la classe Forme. Quant aux deuxième cas, n'importe quel objet d'une classe implantant l'interface `Runnable` peut être affecté aux éléments du tableau.

11.1.2 Accès aux éléments

On accède aux éliment d'un tableau avec l'opérateur `[ ]`. On notera `tab[0]`, `tab[1]`, ..., `tab[n-1]` les n premiers éléments du tableau `tab`.

Les éléments d'un tableau peuvent être indicés par un `int`, `short`, `byte`, ou `char`. Par contre, l'indice ne peut être un `long` ; si c'est le cas, une erreur de compilation est engendrée.

Lors de l'accès aux éléments d'un tableau, *Java* fait la vérification de débordement. Lorsque l'indice du tableau est en dehors des limites des bornes du tableau, l'erreur `IndexOutOfBoundsException` est lancée.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

11.1.3 Taille des tableaux

Par contre, comme il n'existe pas d'arithmétique sur les pointeurs, à chaque tableau est associée sa taille qui est un champ **public final** (**length**) de la classe des tableaux. On connaît donc la taille du tableau en examinant ce champ.

```
for (int i =0; i < tab1.length ; i++)  tab1[ i ] = i ;
```

Il n'est évidemment pas possible de modifier arbitrairement la valeur de ce champ : c'est un champ **final** qui s'initialise à la création du tableau.

11.1.4 Initialisation

Lors de la création d'un tableau, ses éléments sont initialisés à leur valeur par défaut en fonction du type des éléments. Pour les tableaux de nombres (entiers et flottants), la valeur initiale des éléments est zéro ; quant aux tableaux de référence, les éléments sont initialisées à la valeur **null**.

Attention ! Définir un tableau d'objets ne définit qu'un tableau de références. Les objets devront être alloués ultérieurement.

```
Date [ ] tabDate = new Date [ 3 ];
tabDate[ 0 ] = new Date(15,9,57);
tabDate[ 1 ] = new Date(28,5,57);
tabDate[ 2 ] = new Date(19,3,91);
```

L'initialisation d'un tableau peut évidement se faire lors de sa définition et ce, comme en *C*, à l'aide des accolades :

```
int [ ] tab            = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 } ;
Date [ ] tabDate       = { new Date(15,9,57), new Date(28,5,57), new Date(19,3,91) } ;
```

11.1.5 Tableaux multidimentionels

Les *tableau !multidimentionel* sont des tableaux dont les éléments sont eux-mêmes des tableaux. La syntaxe pour la définition d'une matrice 5x5 d'entiers est :

```
int [ ] [ ] mat = new int [ 5 ] [ 5 ];
Date [ ] [ ] matDate = new Date [ 5 ] [ 5 ];
```

Comme pour les tableaux unidimentionnels, il est possible de créer un tableau multidimentionels par initialisation :

```
int [ ] [ ] mat = { {11,12,13,14,15}, {21,22,23,24,25}, {31,32,33,34,35}, {41,42,43,44,45}, {51,52,53,54,55} }
```

11.2 Chaînes de caractères

Une chaîne de caractères en *Java* est un objet de la classe **java.lang.String**. Les objets de type **String** sont des objets figés ; il n'est pas possible de modifier le contenu de tels objets. Il s'agit de constantes de type chaîne de caractère. C'est pourquoi, *Java* fournit une classe **StringBuffer** qui implante les chaînes de caractères modifiables et à taille variable.

11.2.1 La classe String

Dans beaucoup de cas, les chaînes de caractères que l'on crée sont des objets constants. Le compilateur *Java* transforme automatiquement les constantes de type chaînes (voir 3.5.5) de caractères en objet de type **String**. On peut également créer explicitement un objet de type **String** avec un des constructeurs de cette classe.

```
String x = "coucou";
String y = new String("Coucou");
String z = x + y;
```

Outre les constructeurs, la classe **String** fournit les méthodes pour la

- comparaisons des chaînes,
- recherche de sous chaînes,
- extraction de sous chaînes,
- copier avec transformation en majuscule ou minuscule.

```
public final class String extends Object implements Serializable, Comparable {
   public String()
   public String(String value)
   public String(char[] value)
   public String(char[] value, int offset, int count)
   public String(byte[] ascii, int hibyte, int offset, int count)
   public String(byte[] ascii, int hibyte)
   public String(byte[] bytes, int offset, int length, String enc) throws UnsupportedEncodingException
   public String(byte[] bytes, String enc) throws UnsupportedEncodingException
```

```
    public String(byte[] bytes, int offset, int length)
    public String(byte[] bytes)
    public String(StringBuffer buffer)
    public int length()
    public char charAt(int index)
    public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
    public void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)
    public byte[] getBytes(String enc) throws UnsupportedEncodingException
    public byte[] getBytes()
    public boolean equals(Object anObject)
    public boolean equalsIgnoreCase(String anotherString)
    public int compareTo(String anotherString)
    public int compareTo(Object o)
    public boolean regionMatches(int toffset, String other, int ooffset, int len)
    public boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)
    public boolean startsWith(String prefix, int toffset)
    public boolean startsWith(String prefix)
    public boolean endsWith(String suffix)
    public int hashCode()
    public int indexOf(int ch)
    public int indexOf(int ch, int fromIndex)
    public int lastIndexOf(int ch)
    public int lastIndexOf(int ch, int fromIndex)
    public int indexOf(String str)
    public int indexOf(String str, int fromIndex)
    public int lastIndexOf(String str)
    public int lastIndexOf(String str, int fromIndex)
    public String substring(int beginIndex)
    public String substring(int beginIndex, int endIndex)
    public String concat(String str)
    public String replace(char oldChar, char newChar)
    public String toLowerCase(Locale locale)
    public String toLowerCase()
    public String toUpperCase(Locale locale)
    public String toUpperCase()
    public String trim()
    public String toString()
    public char[] toCharArray()
    public static String valueOf(Object obj)
    public static String valueOf(char[] data)
    public static String valueOf(char[] data, int offset, int count)
    public static String copyValueOf(char[] data, int offset, int count)
    public static String copyValueOf(char[] data)
    public static String valueOf(boolean b)
    public static String valueOf(char c)
    public static String valueOf(int i)
    public static String valueOf(long l)
    public static String valueOf(float f)
    public static String valueOf(double d)
    public String intern()
}
```

Constructeurs

`public String ()`

Ce constructeur permet de créer un objet de type **String** vide.

`public String (String valeur)`

Ce constructeur permet de créer un nouvel objet de type **String** qui est une copie de l'objet **String** passé en argument.

`public String (char[ ] valeur)` *throws NullPointerException*

Ce constructeur construit un nouvel objet de type **String** par recopie des éléments du tableau de caractères passé en argument. Comme il s'agit de recopie, une modification des éléments du tableau postérieure à la création de cet objet **String** ne le modifiera pas.

Si le tableau est **null** alors l'exception **NullPointerException** est lancée.

`public String (char[ ] valeur, int offset, int nombre)` *throws NullPointerException, IndexOutOfBoundsException*

Ce constructeur construit un nouvel objet de type **String** par recopie des éléments d'un sous tableau du tableau de caractères passé en argument. Le sous tableau à recopier commence à l'indice **offset** et est de taille **nombre**. Comme il s'agit de recopie, une modification des éléments du tableau postérieure à la création de cet objet **String** ne le modifiera pas.

Si le tableau est `null` alors l'exception `NullPointerException` est lancée.
L'exception `IndexOutOfBoundsException` est lancée quand

- `offset` est négatif,
- `nombre` est négatif,
- `offset+nombre` est plus grand que `valeur.length`.

`public String (byte[ ] valeur, int hibyte)` *throws NullPointerException*

Ce constructeur construit un nouvel objet de type `String` par recopie des éléments du tableau de caractères passé en argument. La transformation d'un élément b du tableau de `byte` en caractère c est faite en complétant les 8 bits de poids fort l'argument `hibyte` d'après la formule :

```
c == ((hibyte & 0xff}) << 8) | (b & 0xff)
```

Si le tableau est `null` alors l'exception `NullPointerException` est lancée.

`public String (byte[ ] valeur, int hibyte, int offset, int nombre)` *throws NullPointerException, IndexOutOfBoundsException*

Ce constructeur construit un nouvel objet de type `String` par recopie des éléments d'un sous tableau du tableau de caractères passé en argument. Le sous tableau à recopier commence à l'indice `offset` et est de taille `nombre`. La transformation d'un élément b du tableau de `byte` en caractère c est faite en complétant les 8 bits de poids fort l'argument `hibyte` d'après la formule :

```
c == ((hibyte & 0xff}) << 8) | (b & 0xff)
```

Si le tableau est `null` alors l'exception `NullPointerException` est lancée.
L'exception `IndexOutOfBoundsException` est lancée quand

- `offset` est négatif,
- `nombre` est négatif,
- `offset+nombre` est plus grand que `valeur.length`.

`public String (StringBuffer buffer)` *throws NullPointerException*

Ce constructeur construit un nouvel objet de type `String` par recopie des caractères de l'objet `StringBuffer` passé en argument.
Si l'objet `StringBuffer` est `null` alors l'exception `NullPointerException` est lancée.

Comparaison

`public int compareTo (String s)` *throws NullPointerException*

Cette méthode rend

- 0 : si les chaînes sont identiques
- un entier négatif : si l'argument `s` est lexicographiquement plus plus petit que l'objet sur lequel s'applique la méthode.
- un entier positif : si l'argument `s` est lexicographiquement plus plus grand que l'objet sur lequel s'applique la méthode.

Si `s` est `null`, l'exception `NullPointerException` est lancée.

`public boolean regionMatches (int toffset, String other, int ooffset, int len)`

Retourne *vrai* si la "région" donnée de cet objet `String` est égales à la région de de la chaîne `other`. La comparaison démarre à `ooffset` pour `other` et `toffset` pour l'objet sur lequel s'applique cette méthode. La comparaison ne concerne que les `len` premiers caractères.

`public boolean regionMatches (boolean ignoreCase, int toffset, String other, int ooffset, int len)`

Même comportement que la méthode précédante mais permet de spécifier si l'on veut différencier les majuscules des minuscules.

`public boolean startsWith (String suffix)` *throws NullPointerException*

Retourne *true* si la chaîne commence par la chaîne `suffixe`.

`public boolean startsWith (String suffix, int toffset)` *throws NullPointerException*

Retourne *true* si la sous chaîne débutant à l'indice `toffset` commence par la chaîne `suffixe`.

`public boolean endsWith (String suffix)` *throws NullPointerException*

Retourne *true* si la chaîne se termine par la chaîne `suffixe`.

`public boolean equals (Object o)`

Retourne `true` si et seulement si

- `o` n'est pas `null`
- l'objet sur lequel s'applique cette méthode est une chaîne identique à `o`.

Cette méthode est une redéfinition de la méthode `equals` de la classe `Object`.

`public boolean equalsIgnoreCase (String s)`

Retourne `true` si et seulement si

- `s` n'est pas `null`
- l'objet sur lequel s'applique cette méthode est une chaîne identique (aux majuscules/minuscules près) à `s`.

Extraction

`public void getChars (int srcBegin, int srcEnd, char dst[], int dstBegin)` *`throws NullPointerException, IndexOutOfBoundsExceptio`*

Recopie la sous chaîne comprise **`srcBegin`** et **`srcEnd`** dans le tableau **`dst`** à partir de l'indice **`dstBegin`**.

`public String substring (int beginIndex)`

retourne la sous chaîne commençant à l'indice **`beginIndex`**.

`public String substring (int beginIndex, int endIndex)`

retourne la sous chaîne compris entre les indices **`beginIndex`** et **`endIndex`**.

`public char charAt (int index)`

Retourne le caractère à l'indice **`index`**.

Recherche

`public int indexOf (int ch)`

Retourne l'indice de la première occurrence du caractère **`ch`**.

`public int indexOf (int ch, int fromIndex)`

Retourne l'indice supérieure à **`fromIndex`** de la première occurrence du caractère **`ch`**.

`public int indexOf (String str)` *`throws NullPointerException`*

Retourne l'indice de la première occurrence de la chaîne **`str`**.

`public int indexOf (String str, int fromIndex)` *`throws NullPointerException`*

Retourne l'indice supérieure à **`fromIndex`** de la première occurrence de la chaîne **`str`**.

`public int lastIndexOf (int ch)`

Retourne l'indice de la dernière occurrence du caractère **`ch`**.

`public int lastIndexOf (int ch, int fromIndex)`

Retourne l'indice supérieure à **`fromIndex`** de la dernière occurrence du caractère **`ch`**.

`public int lastIndexOf (String str)` *`throws NullPointerException`*

Retourne l'indice de la dernière occurrence de la chaîne **`str`**.

`public int lastIndexOf (String str, int fromIndex)` *`throws NullPointerException`*

Retourne l'indice supérieure à **`fromIndex`** de la dernière occurrence de la chaîne **`str`**.

Conversion

`public String toLowerCase ()`

Convertit une chaîne de caractères en minuscules.

`public String toUpperCase ()`

Convertit une chaîne de caractères en majuscules.

`public char[] toCharArray ()`

Convertit un **`String`** en un tableau de caractères.

`public static String valueOf (type obj)`

Retourne une représentation sous forme de **`String`** de l'objet **`obj`**.

Autes méthodes

`public String replace (char oldChar, char newChar)`

Retourne une instance de **`String`** où chaque occurence de **`oldChar`** est remplacée par **`newChar`**.

`public String concat (String str)` *`throws NullPointerException`*

Retoune un nouveau **`String`** résultat de la concaténation de **`str`** à l'objet sur lequel s'applique cette méthode.

`public String trim ()`

Supprime les espaces en début et en fin de chaîne.

11.2.2 La classe StringBuffer

Les objets de la classe **String** sont des objets constants ; la chaîne de caractères qu'elles codent ne peuvent être modifiées. Si l'on veut utiliser des chaînes de caractères modifiables, on devra utiliser des objets de la classe **StringBuffer**.

Un objet de type **StringBuffer** a un espace de stockage à la création qui est automatiquement redimensionné en fonction des besoins.

```
public final class StringBuffer extends Object implements Serializable {
   public StringBuffer()
   public StringBuffer(int length)
   public StringBuffer(String str)
   public int length()
   public int capacity()
   public void ensureCapacity(int minimumCapacity)
   public void setLength(int newLength)
   public char charAt(int index)
   public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
   public void setCharAt(int index, char ch)
   public StringBuffer append(Object obj)
   public StringBuffer append(String str)
   public StringBuffer append(char[] str)
   public StringBuffer append(char[] str, int offset, int len)
   public StringBuffer append(boolean b)
   public StringBuffer append(char c)
   public StringBuffer append(int i)
   public StringBuffer append(long l)
   public StringBuffer append(float f)
   public StringBuffer append(double d)
   public StringBuffer insert(int offset, Object obj)
   public StringBuffer insert(int offset, String str)
   public StringBuffer insert(int offset, char[] str)
   public StringBuffer insert(int offset, boolean b)
   public StringBuffer insert(int offset, char c)
   public StringBuffer insert(int offset, int i)
   public StringBuffer insert(int offset, long l)
   public StringBuffer insert(int offset, float f)
   public StringBuffer insert(int offset, double d)
   public StringBuffer reverse()
   public String toString()
}
```

Constructeurs

`public StringBuffer ()`

Construit une instance de **StringBuffer** d'une capacité de 16 caractères et qui ne contient aucun caractère.

`public StringBuffer (int length)` *throws NegativeArraySizeException*

Construit une instance de **StringBuffer** d'une capacité de **length** caractères et qui ne contient aucun caractère.

`public StringBuffer (String str)`

Construit une instance de **StringBuffer** d'une capacité de 16+taille(str) caractères et qui est rempli avec les caractères de l'objet **str**. C'est la conversion d'un **String** en **StringBuffer**.

Extraction

`public void getChars (int srcBegin, int srcEnd, char[] dst, int dstBegin)` *throws NullPointerException, IndexOutOfBoundsExceptio*

Recopie la sous chaîne comprise **srcBegin** et **srcEnd** dans le tableau **dst** à partir de l'indice **dstBegin**.

Modification

`public void setCharAt (int index, char ch)` *throws IndexOutOfBoundsException*

Le caractère **ch** devient le caractère d'indice **index**.

```
public StringBuffer  append (boolean obj)
public StringBuffer  append (char obj)
public StringBuffer  append (int obj)
public StringBuffer  append (long obj)
public StringBuffer  append (float obj)
public StringBuffer  append (double obj)
public StringBuffer  append (Object obj)
```

Concatène la représentation textuelle de l'objet `obj`.

`public StringBuffer append (String str)`

Concatène `str` à l'instance de `StringBuffer` sur laquelle s'applique cette méthode.

`public StringBuffer append (char[] str)`

Concatène la représentation textuelle du tableau de caractères `str`.

`public StringBuffer append (char[] str, int offset, int len)` *`throws NullPointerException, IndexOutOfBoundsException`*

Concatène la sous chaîne de caractères `str`.

Recopie la sous chaîne comprise `srcBegin` et `srcEnd` dans le tableau `dst` à partir de l'indice `dstBegin`.

```
public StringBuffer  insert (int offset, boolean obj)  throws StringIndexOutOfBoundsException
public StringBuffer  insert (int offset, char obj)  throws StringIndexOutOfBoundsException
public StringBuffer  insert (int offset, int obj)  throws StringIndexOutOfBoundsException
public StringBuffer  insert (int offset, long obj)  throws StringIndexOutOfBoundsException
public StringBuffer  insert (int offset, float obj)  throws StringIndexOutOfBoundsException
public StringBuffer  insert (int offset, double obj)  throws StringIndexOutOfBoundsException
public StringBuffer  insert (int offset, String obj)  throws StringIndexOutOfBoundsException
public StringBuffer  insert (int offset, Object obj)  throws StringIndexOutOfBoundsException
```

Insère la representation textuelle de `obj` à l'indice `offset`.

`public StringBuffer insert (int offset, char[] str)` *`throws StringIndexOutOfBoundsException`*

Insère la représentation textuelle du tableau de caractères `str` à l'indice `offset`.

`public StringBuffer reverse ()`

Remplace la suite des caractères du `StringBuffer` par sa représentation miroir.

Gestion de la taille

`public int length ()`

Retourne le nombre de caractères présents dans l'objet `StringBuffer`.

`public void setLength (int l)` *`throws IndexOutOfBoundsException`*

Etend en remplissant du caractères '$\breve{0}$00' ou tronque le `StringBuffer` de sorte que sa taille soit égale à `l`.

`public int capacity ()`

Retourne la taille de la zone allouée pour l'objet.

`public void ensureCapacity (int minimumCapacity)`

A TERMINER

`public void setCharAt (int index, char ch)` *`throws IndexOutOfBoundsException`*

Le caractère `ch` devient le caractère d'indice `index`.

12. Conversions et promotions

Sommaire

12.1 Type de conversion . . . 87
12.1.1 Conversion vers le même type . . . 87
12.1.2 Conversion vers un type primitif plus grand . . . 88
12.1.3 Conversion vers un type primitif plus petit . . . 88
12.1.4 Conversion vers un objet plus grand . . . 88
12.1.5 Conversion vers un objet plus petit . . . 88
12.1.6 Conversion de chaînes . . . 89
12.1.7 Conversion interdites . . . 89
12.2 Affectation . . . 89
12.3 Invocation de méthodes . . . 89
12.4 Chaînes de caractères . . . 89
12.5 Changement de type . . . 89
12.6 Promotion numérique . . . 90
12.6.1 Opérations unaires . . . 90

A revoir et compléter

Nous avons dit (4.1.3) que toute expression possède un type. Ce type peut être déduit en examinant les constantes, les variables et les méthodes qui figurent dans une expression. On peut être amené à écrire des expressions dont le type n'est pas correct. Parmi ces expressions "incorrectes", certaines sont des vraies erreurs et le compilateurs se charge de le signaler par un message d'erreur à la compilation. Mais il en existe d'autre pour lesquelles *Java* décide d'effectuer automatiquement les *conversions* nécessaires (quand cela est possible) pour rendre l'expression correcte.

Il existe plusieurs types de *conversions* :

- Les conversions effectuées à l'exécution. La conversion d'un objet de la classe `Object` en un objet de la classe `Thread` nécessite une vérification à l'exécution pour s'assurer que l'objet référencé est bien un objet de la classe `Thread` ou d'une de ses sous classes. Si ce n'est pas le cas, une exception est déclenchée.
- Les conversions effectuées à la compilation. La conversion d'un objet de la classe `Thread` en un objet de la classe `Object` ne nécessite aucune vérification à l'exécution. Cette conversion est toujours valide.
- Les conversions avec perte d'information. La conversion d'un `double` en un `long` peut éventuellement conduire une perte d'information.
- Les conversions sans perte d'information. La conversion d'un `int` en `long` peut toujours se faire sans perte d'information.

12.1 Type de conversion

12.1.1 Conversion vers le même type

La conversion d'un type en lui même est permise. Ceci peut sembler trivial voir inutile mais cela permet d'autoriser les opérations de changement de type (`cast`) redondantes.

La seule conversion permise pour le type `boolean` est la conversion vers le type `boolean` lui-même.

12.1.2 Conversion vers un type primitif plus grand

Il existe 19 conversions possibles dans cette catégorie :

- **byte** vers **short**, **int**, **long**, **float**, ou **double**
- **short** vers **int**, **long**, **float**, ou **double**
- **char** vers **int**, **long**, **float**, ou **double**
- **int** vers **long**, **float**, ou **double**
- **long** vers **float** ou **double**
- **float** vers **double**

La conversion d'un int ou long vers un float ou double peut conduire à une perte d'information. La value approchée résultat est obtenu choisissant le flottant le plus de l'entier (*IEEE 754 round-to-nearest*).

Quant aux conversions des entiers (plus petit en taille) vers un autre entier (plus grand en taille) se fait évidemment sans perte de précision et en complétant les bits de poids fort à 0.

Toutes ces conversions ne produisent jamais d'exception lors de l'exécution.

12.1.3 Conversion vers un type primitif plus petit

Il existe 23 conversions possibles dans cette catégorie :

- **byte** vers char
- **short** vers **byte** ou char
- char vers **byte** ou **short**
- int vers **byte**, **short**, ou char
- long vers **byte**, **short**, char, ou int
- float vers **byte**, **short**, char, int, ou long
- double vers **byte**, **short**, char, **int**, long, ou float

Ces conversions peuvent évidemment conduire à des pertes d'informations. Les conversions des entiers (plus grand en taille) vers un autre entier (plus petit en taille) se fait en supprimant les bits de poids fort.

Conversions des flottants vers des entiers

12.1.4 Conversion vers un objet plus grand

- Conversion d'un type de sous classe en un type d'une classe ancêtre.
- Conversion d'un objet d'une classe implantant l'interface en un objet de cette interface.
- Conversion du type null en n'importe type de classe, d'interface ou tableau.
- Conversion d'un type de sous interface en un type d'une interface ancêtre.
- Conversion d'un type interface en type `Object`.
- Conversion d'un tableau en type `Object`.
- Conversion d'un tableau en type `Cloneable`.
- Conversion d'un tableau d'objets SC[] en un tableau d'objets TC[] à condition qu'il existe une conversion de même nature de SC vers TC.

Toutes ces conversions ne produisent jamais d'exception lors de l'exécution.

12.1.5 Conversion vers un objet plus petit

- Conversion d'un type de classe vers un type de sous classe.
- Conversion d'un type de classe S vers un type interface à condition que S ne soit pas final et qu'il n'implante pas l'interface.
- Conversion du type `Object` en un tableau.
- Conversion du type `Object` en une interface.
- Conversion du type interface en un type classe non **final**.
- Conversion du type interface en un type classe **final** à condition qu'il implante l'interface.
- Conversion du type interface I en un type interface J à condition que I ne soit pas une sous interface de J et qu'il n'existe pas de méthode de même signature et de type différent dans I et J.
- Conversion d'un tableau d'objets SC[] en un tableau d'objets TC[] à condition qu'il existe une conversion de même nature de SC vers TC.

Toutes ces conversions effectue des test à l'exécution et peuvent produire l'exception `ClassCastException` lors de l'exécution.

12.1.6 Conversion de chaînes

IL y a une conversion d'un type quelconque (même le type `null`) en type `String`.

12.1.7 Conversion interdites

Il n'est pas permis de convertir
- un objet en type primitif.
- un type primitif en objet sauf pour les `String`.
- le type `null` en type primitif.
- le type `null` en autre que lui même.
- en type `boolean` qu'un autre type `boolean`.
- le type `boolean` en autre que lui même ou en `String`
- un type de classe en un autre type de classe qui ne soit pas une sous classe ou classe ancêtre.
- un type de classe en un type interface si la classe est `final` et qu'elle n'implante pas cette interface.
- un type de classe (autre que `Object`) en un type tableau.
- un type interface en type classe (autre que `String`) si cette dernière est `final` et qu'elle n'implante pas cette interface.
- un type interface en type interface si dans ces deux interfaces il n'existe pas de méthode de même signature et de type différent.
- un type tableau en un type de classe autre que `String` ou `Object`.
- un type tableau en un type interface autre que `Cloneable`
- le type SC[] en type TC[] si la conversion de Sc en TC est interdite.

12.2 Affectation

Lors d'une affectation, le type de l'opérande droit est converti en le type de l'opérande gauche. L'affectation

Array Store Exception

```
byte x = 5;
```

est valide car la constante 5 (qui est de type int) est converti automatiquement en byte. Il n'est pas nécessaire d'écrire

```
byte x = (byte)5;
```

Lorsque la conversion requise pour une affectation est interdite alors une erreur de compilation est générée.

12.3 Invocation de méthodes

Lors de l'invocation d'une méthode, les paramètres effectifs de la méthode sont convertis en accord avec le type des paramètres formels.

Contrairement à l'affectation, la conversion vers un type plus restreint n'est pas effectuée et ce pour simplifier la surcharge des méthodes :

```
class Test {
    static int m(byte a, int b) { return a+b; }
    static int m(short a, short b) { return a-b; }
    public static void main(String[] args) {
        System.out.println(m(12, 2));           // Erreur de compilation
        }
}
```

12.4 Chaînes de caractères

La conversion de chaînes de caractères ne concerne que l'opérateur de concaténation "+".

N'importe quel type peut être converti en `String`. Pour être converti en un objet de type `String`, une valeur d'un type primitif est tout d'abord converti en un objet de son type (par exemple, un float en un objet de type Float). Ainsi, on se ramène au cas général de la conversion d'un objet d'un classe en un objet de type `String` qui utilise la méthode `toString`.

12.5 Changement de type

Le changement de type s'obtient en appliquant explicitement l'opération de *cast*. Par cette opération, il est permis d'effectuer n'importe quel opération de conversion permise.

Certaines conversions non permises peuvent être détectées à la compilation et auquel cas, une erreur de compilation est générée.

détailler voir spec

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

12.6 Promotion numérique

Dans le contexte d'opération arithmétiques, les conversions automatiques effectuées sont appelées `promotions numériques`.

12.6.1 Opérations unaires

La promotion numérique sont effectuées dans les cas suivants :

13. Classes imbriquées

Sommaire

13.1 Nested class et Inner class . . . 91
13.2 Un exemple . . . 91
13.3 Classes locales . . . 93
13.4 Classes anonymes . . . 93
13.5 Classes imbriquées final, private, protected, ou static . . . 94
13.6 Classes imbriquées et accessibilité . . . 94

13.1 Nested class et Inner class

Dans tout ce que nous vennons de voir, les classes *Java* étaient toutes au même niveau. On peut faire un parallèle avec les fonctions du langage *C* où toutes les fonctions sont au même niveau et il n'est pas possible d'imbriquer des fonctions dans d'autres fonctions (comme dans le langage *Pascal*).

Depuis la version 1.1, le langage *Java* permet de définir des classes à l'intérieur d'autres classes.

```
class A {
    ...
    class B {
        ....
    }
}
```

L'intérêt de ces classes qu'on imbrique dans d'autres classes est

- que leur visibilité est réduite à la seule classe qui les contient.
- que ces dernières peuvent faire référence aux membres de la classe englobante ainsi que des variables locales du bloc qui les contient.

Tout comme les champs et méthodes, une classe imbriquée peut être qualifiée de **static**. Et comme on pouvait s'y attendre, les classes **static** ne peuvent faire référence aux champs et méthodes non **static** de la classe englobante.

Les classes définies à l'intérieur d'une classe ont accès à tous les champs (publiques ou privés) de la classe qui la contient. Autrement dit, tous les champs et méthodes d'une classe sont accessibles à partir de ses champs, méthodes et classes imbriquées.

Les classes **static** sont appelées *classes imbriquées statiques* (*static nested class*) et les autres seront appelés *classes intérieures* (*inner class*) Les classes intérieures ne peuvent définir des champs **static**.

Une instance d'une classe intérieure n'existe que s'il existe une instance de la classe qui la contient. Alors qu'une classe imbriquée **static** se comporte exactement comme une classe non imbriquée.

Les classes imbriquées (static ou pas) peuvent être déclarées **abstract** ou **final**. De plus, une classe imbriquée peut être **public**, **private** ou **protected**.

13.2 Un exemple

Dans la programmation graphique, il est très fréquent d'utiliser les classes imbriquées pour la gestions des évènements. Sans entrer dans les détails de la programmation graphique que nous verrons plus loin (voir 25), signalons que *Java* fournit l'interface **java.awt.event.MouseListener** pour la gestion des évènements de la souris.

```
interface java.awt.event.MouseListener {
  public void mousePressed(java.awt.event.MouseEvent e);
  public void mouseClicked(java.awt.event.MouseEvent e);
  public void mouseReleased(java.awt.event.MouseEvent e);
  public void mouseEntered(java.awt.event.MouseEvent e);
  public void mouseExited(java.awt.event.MouseEvent e);
}
```

Le programmeur capture les évènements en implantant la méthode correspondante. Dans l'exemple qui suit, on définit une applet avec un bouton étiquetté par une chaîne de caractères qui se retourne à chaque clic.

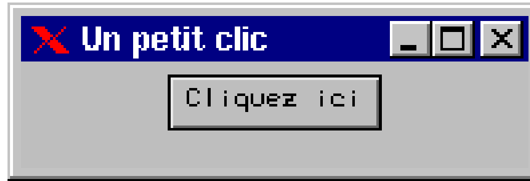

FIG. 13.1: Exemple de gestion d'évènements graphiques

```
public class Click extends java.applet.Applet implements java.awt.event.MouseListener {
   java.awt.Button bouton;
   boolean b = true;
   public void init() {
      bouton = new java.awt.Button("Cliquez ici");
      add("Center", bouton);
      bouton.addMouseListener(this);
   }
   public void mousePressed(java.awt.event.MouseEvent e) {
      b = !b;
      bouton.setLabel(b ? "Cliquez ici" : "ici zeuqilC");
   }
   public void mouseClicked(java.awt.event.MouseEvent e) {}
   public void mouseReleased(java.awt.event.MouseEvent e) {}
   public void mouseEntered(java.awt.event.MouseEvent e) {}
   public void mouseExited(java.awt.event.MouseEvent e) {}
}
```

Dans cet exemple, le programmeur ne s'intéresse pas à tous les évènements possibles mais qu'à un nombre limité d'entre eux. Par exemple, un bouton peut s'intéresser qu'au fait de cliquer. Dans ces cas, le composant qui implante l'interface `MouseListener` doit fournir l'implantation de toutes les méthodes de cette interface. Ce qui conduit le programmeur à fournir tout un ensemble de méthode vide.

Pour éviter cette inflation de méthodes vides, *Java* fournit, pour certaines des ces interfaces, une classe qui l'implante avec des méthodes vides. Par exemple, la classe `MouseAdapter` est définie comme suit :

```
public class MouseAdapter implements MouseListener {
  public  void mouseClicked(MouseEvent e) {}
  public  void mousePressed(MouseEvent e) {}
  public  void mouseReleased(MouseEvent e) {}
  public  void mouseEntered(MouseEvent e) {}
  public  void mouseExited(MouseEvent e) {}
}
```

Il suffit alors de se définir une sous classe de l'une de celles-ci et de ne redéfinir que les méthodes qui nous intéressent.

```
class MyMouseAdapter extends java.awt.event.MouseAdapter {
   private boolean b = true;
   private java.awt.Button bouton;
   public (MyMouseAdapter(java.awt.Button bouton) { this.bouton = bouton; }
   public void mousePressed(java.awt.event.MouseEvent e) {
      b = !b;
      bouton.setLabel(b ? "Cliquez ici" : "ici zeuqilC");
   }
}

public class Click extends java.applet.Applet {
   private java.awt.Button bouton;
   public void init() {
      bouton = new java.awt.Button("Cliquez ici");
      add("Center", bouton);
```

```
        MyMouseAdapter ma = new MyMouseAdapter(bouton);
        bouton.addMouseListener(ma);
    }
}
```

Avec cette manière de faire, on a réduit considérablement le nombre de méthodes vides que l'on est amené à programmer. Cependant, on vient d'augmenter le nombre de classes que l'on doit programmer et ce d'autant que la classe `MyMouseAdapter` n'a d'intérêt que pour la seule classe `Click`.

Une première approche consiste à utiliser une classe imbriquée pour encapsuler la classe `MyMouseAdapter` dans la classe `Click`.

```
public class Click extends java.applet.Applet {
    java.awt.Button bouton;
    public void init() {
        bouton = new java.awt.Button("Cliquez ici");
        add("Center", bouton);
        MyMouseAdapter ma = new MyMouseAdapter();
        bouton.addMouseListener(ma);
    }
    class MyMouseAdapter extends java.awt.event.MouseAdapter {
        boolean b = true;
        public void mousePressed(java.awt.event.MouseEvent e) {
            b = !b;
            bouton.setLabel(b ? "Cliquez ici" : "ici zeuqilC");
        }
    }
}
```

Avec la classe imbriquée, il n'est plus nécessaire de passer d'argument dans constructeur de la classe imbriquée : les champs de la classe englobante sont accessible par la classe imbriquée.

13.3 Classes locales

Une classe imbriquée peut être définie, non seulement à l'emplacement d'un champ d'une autre classe, mais également partout une instruction peut figurer. Dans ce cas, toutes les variables (arguments des méthodes, variables locales) accessibles pour une instruction quelconque l'est également pour les méthodes de la classe imbriquée et ce **à condition qu'elles soit déclarées `final`**.

```
public class Click extends java.applet.Applet {
    public void init() {
    final java.awt.Button bouton = new java.awt.Button("Cliquez ici");
        add("Center", bouton);
        class MyMouseAdapter extends java.awt.event.MouseAdapter {
            boolean b = true;
            public void mousePressed(java.awt.event.MouseEvent e) {
                b = !b;
                bouton.setLabel(b ? "Cliquez ici" : "ici zeuqilC");
            }
        }
        MyMouseAdapter ma = new MyMouseAdapter();
        bouton.addMouseListener(ma);
    }
}
```

Ce sont les éventuels problèmes de synchronisation qui dicte cette restriction. Un moyen de contourner l'impossiblite d'accéder et/ou de modifier les varibales locales est d'utiliser un tableau contenant l'élément qu'on veut manipuler.

13.4 Classes anonymes

Les classes anonymes permettent de définir des classes locales sans devoir les nommer.

```
public class Click extends java.applet.Applet {
    public void init() {
    final java.awt.Button bouton = new java.awt.Button("Cliquez ici");
        add("Center", bouton);
        MouseAdapter ma = new java.awt.event.MouseAdapter {
            boolean b = true;
            public void mousePressed(java.awt.event.MouseEvent e) {
                b = !b;
                bouton.setLabel(b ? "Cliquez ici" : "ici zeuqilC");
            }
        };
        bouton.addMouseListener(ma);
```

```
    }
  }
```

13.5 Classes imbriquées final, private, protected, ou static

Une classe locale ne peut être qualifiée `public`, `private`, `protected`, ou `static`. En effet, en tant que classe locale, elle est privée et elle n'est pas visible en dehors du bloc qui le définit.

Les classes non locales peuvent être, comme les classes ordinaires, qualifées de `private` ou `protected`.

Les classes nommées peuvent être `final` ou `abstract`.

13.6 Classes imbriquées et accessibilité

Une classe imbriquée a accès à tous les champs de la classe qui la contient ; mem les champs privés.

```
class A {
   int i;
   public static void main(String args[]) {
      pp p = new pp();
      qq q = p.new qq();
      System.out.println(p.i);
      System.out.println(q.j);
   }
   public static class qq {
       public static void affichier(String args[]) {
   static int jj;
   int j = 5;
   }
}
```

Une classe imbriquée `static` ne peut accéder (directement) aux varibales d'instance de la la classe englobante.

Seules les classes imbriquée `static` peuvent contenir des champs `static`.

14. La hiérarchie des classes

Comme tous les langages de programmation objet, *Java* fournit un ensemble de classes au programmeurs. L'ensemble de ces classes sont regroupés en packaqes. La version 1.1 de *Java* est constitué des packages suivants :

`java.applet`	programmation des applets
`java.awt`	programmation graphique
`java.awt.datatransfer`	transfert de données inter applications ou entre plusieurs applications.
`java.awt.event`	pgestion des évènements (en particulier graphqiue)
`java.awt.image`	gestion des images
`java.beans`	
`java.io`	entrées sorties
`java.lang`	noyau du langage
`java.lang.reflect`	
`java.math`	
`java.net`	programmtion réseau

`java.rmi`
`java.rmi.dgc`
`java.rmi.registry`
`java.rmi.server`
	programmation d'applications distribuées

`java.security`
`java.security.acl`
`java.security.interfaces`
	gestion de sécurité d'exécution

`java.sql`	gestions des bases de données
`java.text`	représentation textuelle des données.
`java.util`	Divers classes utiles (piles, dictionnaires, etc.)
`java.util.zip`	compression et décompression des données

La version 1.2 du langage est bien plus étoffée et contient :

java.applet	programmation des applets
java.awt	programmation graphique
java.awt.accessibility java.awt.color java.awt.datatransfer	transfert de données inter applications ou entre plusieurs applications.
java.awt.dnd java.awt.event	Drag and Drop gestion des évènements (en particulier graphqiue)
java.awt.font	
java.awt.geom	
java.awt.im	
java.awt.image	gestion des images
java.awt.image.codec	
java.awt.image.renderable	
java.awt.print	
java.awt.swing	
java.awt.swing.border	
java.awt.swing.event	
java.awt.swing.table	
java.awt.swing.text	
java.awt.swing.text.html	
java.awt.swing.text.rtf	
java.awt.swing.tree	
java.awt.swing.undo	
java.beans	
java.beans.beancontext	
java.io	entrées sorties
java.lang	noyau du langage
java.lang.ref	
java.lang.reflect	
java.math	
java.net	programmtion réseau
java.rmi java.rmi.activation	
java.rmi.dgc java.rmi.registry java.rmi.server	programmation d'applications distribuées
java.security java.security.acl java.security.cert	gestion de sécurité d'exécution
java.security.interfaces	
java.security.spec	
java.sql	gestions des bases de données
java.text	représentation textuelle des données.
java.util	Divers classes utiles (piles, dictionnaires, etc.)
java.util.zip	compression et décompression des données
java.util.jar	
java.util.mime	
java.util.zip	
javax.servlet	
javax.servlet.http	
org.omg.CORBA	
org.omg.CORBA.ORBPackage	
org.omg.CORBA.TypeCodePackage	
org.omg.CORBA.portable	
org.omg.CosNaming	
org.omg.CosNaming.NamingContextPackage	

15. Entrées-Sorties

Sommaire

15.1 Introduction . . . 97
15.2 Hiérarchie des classes et interfaces de java.io . . . 100
15.2.1 Les classes de base . . . *100*
15.2.2 Les classes dérivées de bas niveau . . . *103*
15.2.3 Les entrées sorties structurées . . . *103*
15.2.4 Autres classes . . . *103*
15.2.5 Les interfaces . . . *104*
15.3 Les entrées/sorties standard (terminal) . . . 104
15.4 Les entrées/sorties structurées . . . 104
15.4.1 Le type Filter . . . *104*
15.4.2 Le type Data . . . *105*
15.4.3 Le type Buffered . . . *106*
15.4.4 Le type LineNumber . . . *106*
15.4.5 Le type PushBack . . . *107*
15.4.6 Le type Print . . . *107*
15.4.7 Définir ses propres filtres . . . *109*
15.5 Les tableaux et les chaînes . . . 109
15.5.1 Les tableaux d'octets . . . *109*
15.5.2 Les chaînes . . . *109*
15.6 Les fichiers . . . 110
15.6.1 La classe FileDescriptor . . . *110*
15.6.2 La classe File . . . *110*
15.6.3 Les classes FileInputStream, FileReader, FileOutputStream et FileWriter . . . *111*
15.6.4 L'interface FilenameFilter . . . *112*
15.6.5 RandomAccessFile . . . *112*
15.7 Les tubes (pipes) . . . 113
15.7.1 PipedInputStream, PipedReader . . . *113*
15.7.2 PipedOutputStream, PipedWriter . . . *113*
15.8 StreamTokenizer . . . 114
15.9 Conversion des types de flots . . . 116
15.10Séquence de flots . . . 116

15.1 Introduction

Les entrées/sorties sont basés sur les *flots* (*streams*). Un *flot de données* est un canal de communication constitué d'une suite ordonnée de données ayant deux extrémités : une *entrée* et une *sortie*. *Java* fournit un moyen de manipuler des flots de données de manière identique quelque soit le système d'exploitation sous-jacent. L'ensemble des classes permettant d'utiliser les entrées/sorties est regroupé dans le package `java.io`.

Le package `java.io` contient les classes `InputStream` et `OutputStream`. Les classes `InputStream` et `OutputStream` sont des classes abstraites qui définissent les méthodes d'entrée/sortie de base. Elles manipulent des suites d'octets.

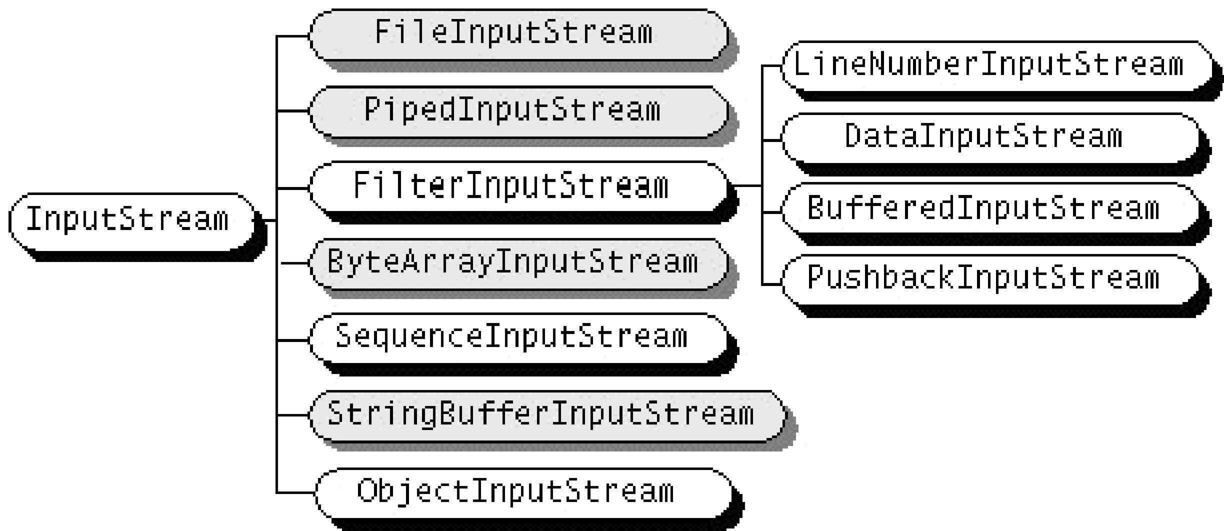

FIG. 15.1: Les flots d'entrée orienté octets

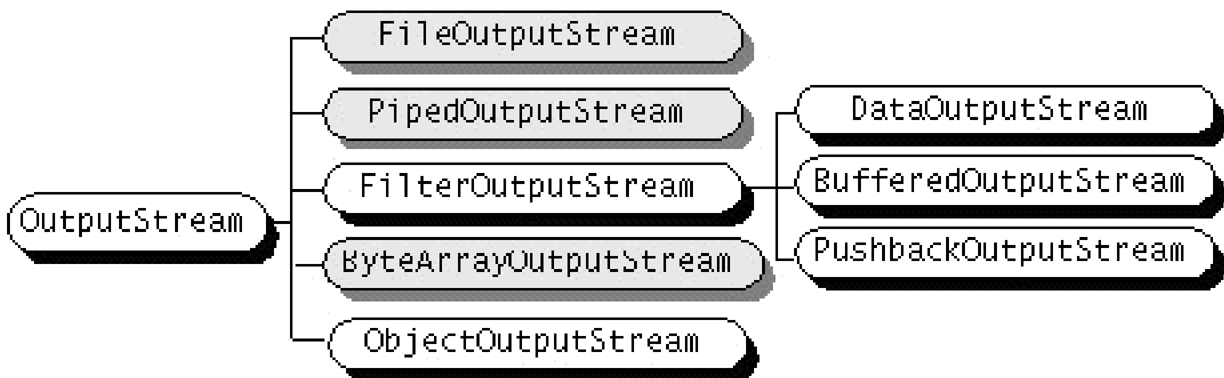

FIG. 15.2: Les flots de sortie orienté octets

Depuis la version 1.1, les classes `Reader` et `Writer` ont été rajoutées au package `java.io`. La hiérarchie des nouvelles classes suit de près celle des existantes déjà dans la version 1.0. La principale raison de cette modification est le besoin de gérer les caractères internationaux sur 16 bits aux lieu des 8 bits de l'ancienne version.

Ces classes `Reader` et `Writer` sont également des classes abstraites et manipulent comme unité de base le caractère (et non pas l'octet).

Les classes `Reader` et `Writer`, tout comme les classes `InputStream` et `OutputStream`, sont les classes de base à partir desquelles toutes les autres classes sont dérivées.

On trouve également dans ce package les classes `ObjectInputStream` et `ObjectOutputStream` pour lire et écrire des objets (voir 16).

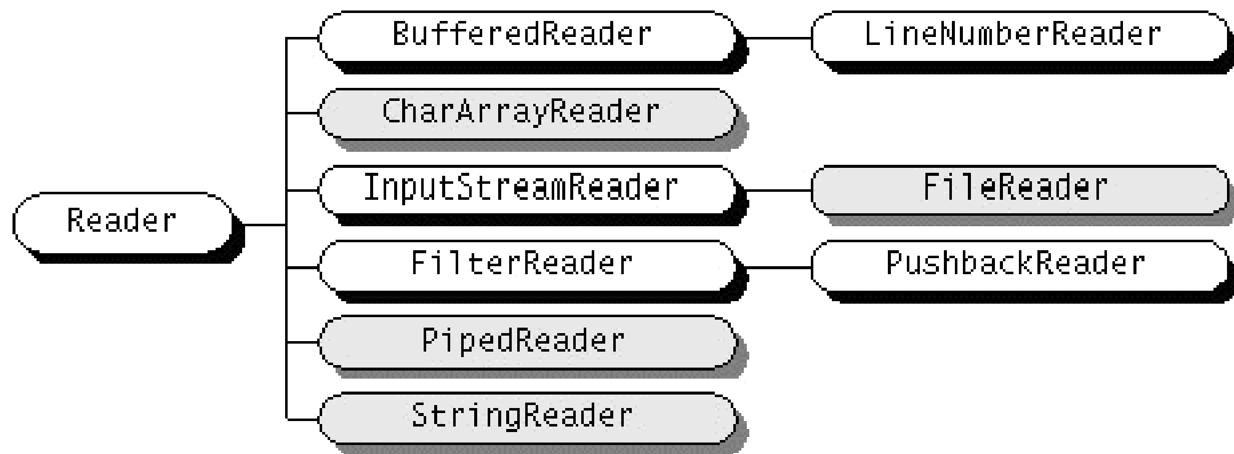

FIG. 15.3: Les flots d'entrée orienté octets

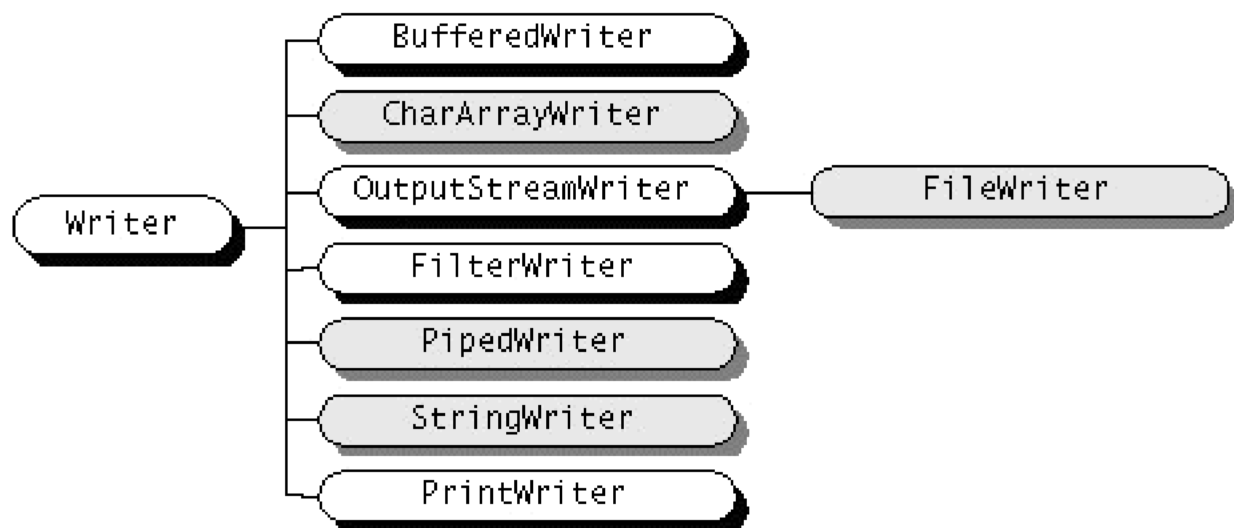

FIG. 15.4: Les flots de sortie orienté octets

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

15.2 Hiérarchie des classes et interfaces de java.io

La version 1.1 du package `java.io` est constitué des packages suivants :

Les interfaces :
DataInput
DataOutput
Externalizable
FilenameFilter
ObjectInput
ObjectInputValidation
ObjectOutput
Serializable

Les Classes :
BufferedInputStream
BufferedOutputStream
BufferedReader
BufferedWriter
ByteArrayInputStream
ByteArrayOutputStream
CharArrayReader
CharArrayWriter
DataInputStream
DataOutputStream
File
FileDescriptor
FileInputStream
FileOutputStream
FileReader
FileWriter
FilterInputStream
FilterOutputStream
FilterReader
FilterWriter
InputStream
InputStreamReader
LineNumberInputStream
LineNumberReader
ObjectInputStream
ObjectOutputStream
ObjectStreamClass
OutputStream
OutputStreamWriter
PipedInputStream
PipedOutputStream
PipedReader
PipedWriter
PrintStream
PrintWriter
PushbackInputStream
PushbackReader
RandomAccessFile
Reader
SequenceInputStream
StreamTokenizer
StringBufferInputStream
StringReader
StringWriter
Writer

15.2.1 Les classes de base

Classes de base pour beaucoup d'autres classes, les classes `InputStream` et `OutputStream` définissent les méthodes pour lire et écrire dans un flot de données. Ces classes ignorent la structure de données manipulées et se contentent de le “voir” comme des suites d'octets. Comme ce sont des classes abstraites, il n'est pas possible de créer des objets de cette classe.

InputStream et Reader

```
public abstract class InputStream {
    public InputStream()
    public int available() throws IOException
    public void close() throws IOException
    public synchronized void mark(int limite)
    public boolean markSupported(int limite)
    public abstract int read() throws IOException
    public int read(byte[ ] tampon) throws IOException
    public int read(byte[ ] tampon, int debut, int nbre) throws IOException
    public synchronized void reset()throws IOException
    public long skip(long nbre) throws IOException

}
```

`public abstract int  read () throws IOException`

Cette méthode lit un octet sur le flot spécifié et retourne cette valeur sous la forme d'un entier. Il s'agit d'une lecture bloquante. Si la valeur retournée est -1, cela indique que l'on atteint la fin normale du flot. On peut ainsi tester cette valeur pour gérer l'erreur "fin de flot de données". Cela étant, en cas d'une quelconque erreur lors de la lecture, une exception `IOException` est levée. Il convient alors de capturée cette exception et gérer toutes les erreurs possibles. Une utilisation standard de cette méthode ressemble à ce qui suit :

```
try {
    int intLu = System.in.read();
    ...
}
catch (IOException e) {
    ...
}
```

La valeur retournée est un `int` et non pas un `byte`. Un octet lu (et non pas le code d'erreur) est un entier compris entre 0 et 255. Pour pouvoir l'utiliser comme un `byte`, il faut le transformer en `byte` :

```
byte  byteLu = (byte) intLu;
```

`public int  read (byte[ ] tampon)  throws IOException`

Cette méthode lit dans le flot d'entrée une suite d'octets et les range dans le tableau tampon. Le nombre d'octets lus correspond à la taille du tableau (qui est donnée par `tampon.length`). Cette méthode retourne -1 en cas de fin de flot et le nombre d'octets lus autrement. Ces deux méthodes sont également bloquantes jusqu'à ce que l'entrée soit disponible.

```
byte [ ] tampon = new byte[ 200 ];
int nb_car_lus = System.in.read(tampon);
```

`public int  read (byte[ ] tampon, int debut, int nbre)  throws IOException`

Cette méthode lit dans le flot d'entrée une suite d'octets et les range dans le tableau tampon à partir de l'indice debut. Le nombre d'octets à lire est donné par la variable nbre. Cette méthode retourne -1 en cas de fin de flot et le nombre d'octets lus autrement.

```
byte [ ] tampon = new byte[ 200 ];
int nb_car_lus = System.in.read(tampon, 10, 20);
```

`public int  available ()  throws IOException`

La méthode available permet de déterminer le nombre d'octets disponible sur l'entrée. Ceci permet, par exemple, d'allouer dynamiquement le tableau dans lequel on va ranger les octets lus.

```
int disponible = System.in.available();
if (disponible > 0) {
    byte [ ] tampon = new byte[ disponible ];
    nb_car_lus = System.in.read(tampon);
    ...
}
```

Cette méthode ne s'applique pas à l'entrée standard. Le nombre d'octets disponible n'a de sens que sur les fichiers. Pour l'entrée standard, on aura toujours pour valeur 0.

`public long  skip (long nbre)  throws IOException`

La méthode skip permet de sauter le nombre nbre d'octets dans le flot d'entrée. Cette méthode n'est là que pour des problèmes de performances. Si la fin du flot survient avant les nbre octets, le flot est positionné à sa fin.

`public void  close ()  throws IOException`

Cette méthode referme le flot d'entrée et libère les ressources systèmes.

`public void  mark (int readlimit)`

Marque la position courante du flot d'entrée ce qui permet de repositionner le flot à cette marque lors de l'invocation de la méthode `reset`. L'argument `readlimit` donne le nombre d'octets pouvant être lus avant la marque devienne invalide.

`public void reset () throws IOException`

Repositionne le flot à la marque repérée par la méthode `mark`. L'exception `IOException` est levée si la marque est invalidée ou inexistante.

`public boolean markSupported ()`

Retourne *vrai* si le flot implante les méthode `mark` et `reset`.

La classe `Reader` est similaire joue le même rôle que la classe `InputStream` mais elle agit sur les caractères.

```
public abstract class Reader {
    protected Reader(Object lock)
    protected Reader()
    public int available() throws IOException
    public void close() throws IOException
    public synchronized void mark(int limite)
    public boolean markSupported(int limite)
    public abstract int read() throws IOException
    public int read(char[ ] tampon) throws IOException
    public int read(char[ ] tampon, int debut, int nbre) throws IOException
    public boolean ready() throws IOException
    public synchronized void reset()throws IOException
    public long skip(long nbre) throws IOException
}
```

OutputStream et Writer

Toutes les méthodes, excepté le constructeur, lève une exception en cas d'erreur.

```
public class OutputStream {
   public OutputStream()
   public abstract void write(int b) throws IOException
   public void write(byte b[]) throws IOException
   public void write(byte b[], int off, int len) throws IOException
   public void flush() throws IOException
   public void close() throws IOException
}
```

`public abstract void write (int b) throws IOException`

Cette méthode écrit un octet sur le flot de sortie. L'octet à écrire est passé en argument sous la forme d'une int et non pas sous la forme d'un byte. Rappelons que les expressions qui manipulent les bytes sont de type int ; il en découle qu'il n'est pas nécessaire faire une conversion explicite lors du passage de ce paramètre. Les bits de poids forts de cet int sont évidemment perdus lors de l'écriture. Cette méthode est bloquante jusqu'à la réalisation de l'écriture.

`public void write (byte b[ ] tampon) throws IOException`

Cette méthode écrit un tableau d'octets sur le flot de sortie.

`public void write (byte b[ ] tampon, int debut, int nbre) throws IOException`

Cette méthode écrit les octets du tableau tampon à partir de l'indice debut. Le nombre d'octets à écrire est donné par le paramètre nbre.

`public void flush () throws IOException`

Vide le tampon de sortie dans le flot de sortie.

`public void close () throws IOException`

Cette méthode referme le flot de sortie et libère les ressources système relatives à ce flot.

La classe `Writer` est similaire joue le même rôle que la classe `OutputStream` mais elle agit sur les caractères.

```
class abstract  Writer
  protected Writer()
  protected Writer(Object lock)
  public void write(int c) throws IOException
  public void write(char cbuf[]) throws IOException
  public abstract void write(char cbuf[], int off, int len) throws IOException
  public void write(String str) throws IOException
  public void write(String str, int off, int len) throws IOException
  public abstract void flush() throws IOException
  public abstract void close() throws IOException
```

15.2.2 Les classes dérivées de bas niveau

Les classes dérivées de `Reader` (resp. `Writer`) et celles dérivées de `InputStream` (resp. `OutputStream`) ne se distinguent que par le fait que les premiers manipulent des caractères et les seconds, des octets. Elles sont construites selon l'origine ou la destination du flot utilisé :

- les tableaux d'octets ou de caractères
  Les classes `ByteArrayInputStream`, `ByteArrayOutputStream`, `CharArrayReader` et `CharArrayWriter` sont dérivées respectivement des classes `InputStream`, `OutputStream`, `Reader` et `Writer` elles permettent de travailler sur un tableau d'octets ou caractères comme s'il s'agissait d'un flot d'entrée/sortie.
- les fichiers : Les classes `FileInputStream`, `FileOutputStream`, `FileReader` et `FileWriter` sont dérivées respectivement des classes `InputStream`, `OutputStream`, `Reader` et `Writer` permettent de lire et écrire dans les fichiers.
- les tubes (*pipe*) :
  Les classes `PipedInputStream`, `PipedOutputStream`, `PipedReader` et `PipedWriter` sont dérivées respectivement des classes `InputStream`, `OutputStream`, `Reader` et `Writer` sont utiles pour la programmation des *tubes* (*pipes*).
- une succession d'autres flots : La classe `SequenceInputStream` est dérivée de la classe `InputStream`, elle permet de transformer plusieurs sources de données en un seul flot continu.
- les chaînes de caractères : `StringBufferInputStream/StringReader/StringWriter` : Dérivée également de la classe `InputStream`, `Reader` et `Writer`, elle permet de lire des chaînes de caractères.
- etc.

De plus, les classes `FilterInputStream` et `FilterOutputStream` est définie pour servir de classe de base pour manipuler, non pas des octets ou des caractères, des objets de plus haut niveau.

15.2.3 Les entrées sorties structurées

Contrairement aux classes que nous venons d'évoquer, celles qui suivent permettent de manipuler des données structurées.

- `BufferedInputStream/BufferedOutputStream/BufferedReader/BufferedWriter` : Dérivées respectivement des classes `FilterInputStream`, `FilterOutputStream`, `Reader` et `Writer` elles permettent d'effectuer les opérations d'entrées/sorties à travers un tampon, et ce pour des besoins de performances.
- `FilterInputStream/FilterOutputStream/FilterReader/FilterWriter` : Dérivées respectivement des classes `InputStream`, `OutputStream`, `Reader` et `Writer`, ces classes filtrent les entrées ou sorties. Ces classes ne sont pas directement utiles. Elles sont elle-même dérivées en fonction du type de filtrage désiré.
- `DataInputStream/DataOutputStream` : Dérivées également des classes `FilterInputStream` et `FilterOutputStream`, elles permettent de lire ou écrire, non pas des octets, par des données plus sophistiquées : des données de types de base de Java ainsi que les chaînes de caractères (`String`).
- `LineNumberInputStream/LineNumberReader` : Dérivée également des classes `FilterInputStream` et `Reader`, elles permettent la gestion des numéros de lignes.
- `PushBackInputStream/PushBackReader` : Dérivée également des classes `FilterInputStream` et `Reader`, elles permettent de remettre un octet ou caractère lu ; classe utile pour l'écriture d'analyseur.
- `PrintStream/PrintWriter` : Dérivée également des classes `FilterOutputStream` et `Writer`, elles permettent la représentation textuelle des données de type primitifs et des chaînes. La variable `System.out` est un objet de la classe `PrintStream` et nous avons déjà utilisé la méthode print et println de cette classe pour afficher des chaînes de caractères et d'autres données primitives.

15.2.4 Autres classes

- `File` : Avec cette classes, *Java* permet de nommer les fichiers selon la même convention quelque soit la système d'exploitation. Elle permet également des méthodes pour manipuler les répertoires, "lister" les fichiers d'un répertoire, examiner les propriétés d'un fichier, etc.
- `FileDescriptor` : Permet de créer un nouveau flot de type `File` ou `RandomAccessFile` vers un autre flot sans avoir à connaître le nom du fichier.
- `InputStreamReader/OutputStreamWriter` : Classes dérivées des classes `Reader` et `Writer` qui permettent la conversion des flots d'octets en flots de caractères.
- `RandomAccessFile` : Classe permettant la lecture et l'écriture en accès direct.
- `ObjectInputStream/ObjectOutputStream` : Classes dérivées des classes `InputStream` et `OutputStream`, elle permettent d'écrire et de lire des objets quelconques.
- `StreamTokenizer` : Dérivée également de la classe `InputStream`, elle effectue une analyse syntaxique des données en entrée.
- `ObjectStreamClass` : voir 16

15.2.5 Les interfaces

- `DataOutput/DataInput` : Ce sont des interfaces qui définissent les méthodes de filtrage des entrées/sorites. Ce sont ces interfaces qui sont implantées dans les classes `DataInputStream`, `DataOutputStream` et `randomAccessFile` etc. Les méthodes de filtrage permettent de lire ou écrire les données de type primitif de manière indépendante de l'architecture utilisée.
- `ObjectInput/ObjectOutput` :
- `FilenameFilter` : C'est une interface qui définit une unique méthode `accept`.
- `Externalizable` : voir 16
- `ObjectInput/ObjectOutput` : voir 16
- `Serializable` : voir 16

15.3 Les entrées/sorties standard (terminal)

L'interface avec le système d'exploitation est gérée par la classe `java.lang.System`. La gestion des entrées/sorties standard passe par les objets `System.in` (*entrée standard*, équivalent de `stdin`), `System.out` (*sortie standard*, équivalent de `stdout`) et `System.err` (*sortie erreur*, équivalent de `stderr`).

```
public final class System extends Object {
        ...
        public static PrintStream err, out;
        public static InputStream in;
        ...
}
```

Les sorties ne sont pas des références vers des objets de la classe `OutputStream` mais plutôt de la classe `PrintStream` qui donne une représentation textuelle des données. Nous verrons plus loin et en détails la classe `PrintStream`. Pour le moment, contentons nous de remarquer que c'est une sous classe de la classe `OutputStream`.

De même, l'entrée est déclaré comme un objet de la classe `InputStream`. Cette classe est une classe abstraite et l'objet `in` est en réalité une instance de la classe `BufferedInputStream`.

```
class Cat {
   public static void main(String[] args)  throws java.io.IOException {
      int c, count=0;
      while ((c = System.in.read()) != -1) {
         count++;
         System.out.write(c);
      }
      System.out.println("Input has " + count + " chars.");
   }
}
```

Noter que le retour de la méthode `read` est un `int` ; ce qui permet de tester une fin de fichier avec la valeur -1.

15.4 Les entrées/sorties structurées

15.4.1 Le type Filter

,

Les flots de type `filter` sont conçus pour une faciliter la manipulation des données d'un flot en appliquant un certain type de filtrage. Par exemple, les classes `FilterInputStream` et `FilterReader` proposent des méthodes qui filtrent les octets ou caractères reçus, à travers un filtre défini dans l'interface `InputFilter`, et revoient une valeur sous la forme d'un objet un peu plus plus structuré qu'un octet.

Les constructeurs de ses classes prennent, selon le cas, en argument un flot de type `InputStream` ou `Reader`.

En fait, cette classe redéfinit toutes les méthodes de la classe `InputStream` et `Reader`.

De même, les classes `FilterOutputStream` et `FilterWriter` proposent des méthodes qui structurent les données à rendre.

Les classes `FilterInputStream`, `FilterReader`, `FilterOutputStream` et `FilterWriter` ne sont pas intéressantes pour elles-mêmes : elles servent de classe de base pour des classes plus utilisées.

De la classe `FilterInputStream`, on dérive les classes

- DataInputStream : pour la lecture des types de bases de *Java*.
- BufferedInputStream : pour la lecture de données associée un tampon de lecture pour minimiser les opérations de lecture système.
- PushbackInputStream : pour une lecture de données permettant de "revenir" sur une ou plusieurs lectures effectuées. Le nombre de retour arrière dépendant du tampon disponible.
- CheckedInputStream : voir package `java.util.zip`
- DigestInputStream : voir package `java.util.zip`

- InflaterInputStream : voir package `java.util.zip`
- ProgressMonitorInputStream : voir package `java.util.zip`

De la classe `FilterOutputStream`, on dérive les classes

- DataOutputStream : pour l'écriture des types de bases de *Java*.
- PrintStream : pour l'écriture des types primitifs et objets en utilisant le jeu de caractère du système utilisé. Cette classe peut être ignorée et devrait être remplacée, dans toutes les applications nouvelles, par la classe `PrintWriter`.
- BufferedOutputStream : pour l'écriture de données associée un tampon de lecture pour minimiser les opérations de lecture système.
- CheckedOutputStream : voir package `java.util.zip`
- DeflaterOutputStream : voir package `java.util.zip`
- DigestOutputStream : voir package `java.util.zip`

De la classe `FilterReader`, on dérive les classes

- PushbackReader : pour une lecture de données permettant de "revenir" sur une ou plusieurs lectures effectuées. Le nombre de retour arrière dépendant du tampon disponible.

15.4.2 Le type Data

Les classes `DataOutputStream` et `DataInputStream` sont des flots de type filtre spécialisés pour la reconnaissance des types primitifs de *Java* ainsi que des chaînes de caractères. C'est généralement ces classes qui sont les plus utilisées dans les applications standards. Cette classe décharge donc le programmeur de la nécessité de réorganiser les données reçues (entrée) ou envoyer (sortie) en données primitives.

```
public class DataInputStream extends FilterInputStream implements DataInput {
   public DataInputStream(InputStream in)
   public final int read(byte b[]) throws IOException
   public final int read(byte b[], int off, int len) throws IOException
   public final void readFully(byte b[]) throws IOException
   public final void readFully(byte b[], int off, int len) throws IOException
   public final int skipBytes(int n) throws IOException
   public final boolean readBoolean() throws IOException
   public final byte readByte() throws IOException
   public final int readUnsignedByte() throws IOException
   public final short readShort() throws IOException
   public final int readUnsignedShort() throws IOException
   public final char readChar() throws IOException
   public final int readInt() throws IOException
   public final long readLong() throws IOException
   public final float readFloat() throws IOException
   public final double readDouble() throws IOException
   public final String readLine() throws IOException
   public final String readUTF() throws IOException
   public final static String readUTF(DataInput in) throws IOException
}
```

Le constructeur de la classe `DataInputStream` prend en argument un objet de type `InputStream`. Pour chaque type primitive, on dispose d'une méthode : `readBoolean`, `readByte`, `readChar`, `readDouble`, `readFloat`, `readInt`, `readLong`, `readShort`. De même les chaînes de caractères peuvent être lues par la méthode `readLine`. Le format *UTF* (*Unicode Transformation Format*) est une format de conversion d'une chaîne de caractères *Unicode* en une chaîne de caractères *ASCII*.

Un objet `DataInputStream` peut être utilisé pour les flots de type *fichier*, *socket* ou *entrée standard*.

```
DataInputStream entrée = new DataInputStream(System.in);
int entier = entrée.readInt();
String chaîne = entrée.readString();
...
```

La classe `DataOutputStream` écrit sur le flot de sortie spécifié des données primitives et des chaînes de caractères.

```
public class DataOutputStream extends FilterOutputStream implements DataOutput {
   protected int written
   public DataOutputStream(OutputStream out)
   public void flush() throws IOException
   public int size() throws IOException
   public synchronized void write(int b) throws IOException
   public synchronized void write(byte b[ ], int debut, int nbre) throws IOException
   public final void writeBoolean(boolean v) throws IOException
   public final void writeByte(int v) throws IOException
   public final void writeBytes(String s) throws IOException
   public final void writeChar(int v) throws IOException
   public final void writeChars(String s) throws IOException
```

```
    public final void writeDouble(double v) throws IOException
    public final void writeFloat(float v) throws IOException
    public final void writeInt(int v) throws IOException
    public final void writeLong(long v) throws IOException
    public final void writeShort(int v) throws IOException
    public final void writeUTF(String s) throws IOException
  }
```

Le constructeur de la classe `DataOutputStream` prend en argument un objet de type `OutputStream`. Pour chaque type primitive, on dispose d'une méthode : `writeBoolean`, `writeByte`, `writeChar`, `writeDouble`, `writeFloat`, `writeInt`, `writeLong`, `writeShort`. De même les chaînes de caractères peuvent être écrites par la méthode `writeChars` ou `writeBytes`.

Un objet `DataOutputStream` peut être utilisé pour les flots de type *fichier*, *socket* ou *sortie standard* et *sortie erreur*.

```
DataOutputStream sortie = new DataOutputStream(System.out);
sortie.writeInt(254);
sortie.writeString("Coucou");
...
```

15.4.3 Le type Buffered

Chacune de ces classes disposent de deux constructeurs :

```
BufferedInputStream(InputStream in)
BufferedOutputStream(InputStream in, int size)

BufferedReader(Reader in)
BufferedReader(Reader in, int size)

BufferedOutputStream(OutputStream out)
BufferedOutputStream(OutputStream out, int size)

BufferedWriter(Writer out)
BufferedWriter(Writer out, int size)
```

L'argument `size` définit la taille de la zone d'entrée sortie tampon que l'on veut associer au flot. Lorsque cet argument n'est pas donné, une taille par défaut est fixée (512 octets).

Ces classes redéfinissent certaines des méthodes de leur classe père et n'introduisent aucune méthode supplémentaire.

```
import java.io.*;

// Affiche deux fois le contenu d'un fichier
class Main {
  public static void main(String[] args) {
    try {
      FileInputStream in = new FileInputStream("fichier1");
      BufferedInputStream bufin = new BufferedInputStream(in);
      if (bufin.markSupported()) {
        int limit;
        bufin.mark(limit=bufin.available());

        for (int i = 0; i < limit; i++)
          System.out.print((char)(bufin.read()));
        bufin.reset();
      }
      int c;
      while ((c=bufin.read()) >= 0) System.out.print((char)c);
      bufin.close();
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

15.4.4 Le type LineNumber

Depuis la version 1.1, la classe `LineNumberReader` remplace la classe `LineNumberInputStream`.

Les classes `LineNumberInputStream` et `LineNumberReader` conserve l'information sur les numéros de lignes du flot. Une ligne est une suite de caractères se terminant par `'`
`r'` (*carriage return*) ou par `'`
`n'` (*line feed*).

Il existe un unique constructeur pour la classe `LineNumberInputStream`

```
LineNumberInputStream(InputStream in)
```

et deux constructeurs pour la classe `LineNumberReader`

```
LineNumberReader(Reader in)
LineNumberReader(Reader in, int sz)
```

Ces deux classes contiennent les méthodes qui permettent la gestion des lignes.

```
int getLineNumber()
void setLineNumber(int lineNumber)
```

Les méthode `mark` et `reset` gèrent également les numéros de lignes, comme on peut s'y attendre.

```
...
LineNumberInputStream in = new LineNumberInputStream(i);
PrintStream out = new PrintStream(o);
int c, oldLineNumber = 0, newLineNumber = 0;
boolean writePrefix = true;

while((c = in.read()) > -1) {
  if (writePrefix) {
    out.print(newLineNumber+1);
    out.write('\t');
  }
  out.write(c);
  if (writePrefix =
      ((newLineNumber = in.getLineNumber()) != oldLineNumber))
    oldLineNumber = newLineNumber;
}
...
```

15.4.5 Le type PushBack

Avec les classes `PushBackInputStream` et `PushBackReader`, il est possible d'effectuer une lecture de données permettant de "revenir" sur la dernière lecture. Les analyseurs lexicaux ont souvent besoin de cette possibilité.

```
public class PushbackInputStream extends FilterInputStream {
   byte[]   buf
   int   pos
   PushbackInputStream(InputStream in, int size)
   PushbackInputStream(InputStream in)
   int   available()
   void   close()
   boolean   markSupported()
   int   read()
   int   read(byte[] b, int off, int len)
   long   skip(long n)
   void   unread(int b)
   void   unread(byte[] b, int off, int len)
   void   unread(byte[] b)
}
```

```
private static boolean readNumber(PushbackInputStream in, Vector vec) {
  StringBuffer sb = new StringBuffer();
  int c = -1;
  try {
    for (c = in.read(); c >= 0; c = in.read()) {
      if (Character.isDigit((char)c)) {
        sb.append((char)c);
      } else {
        in.unread(c);
        break;
      }
    }
  } catch (IOException e) {
  }
...
}
```

15.4.6 Le type Print

Autre extension de la classe `FilterOutputStream`, la classe `PrintStream` est utilisée chaque fois que l'on désire obtenir un représentation textuelle des données de type primitif ou des chaînes de caractères. Les méthodes utilisées sont `println` et `print` selon que l'on veuille passer ou pas à la ligne suivante après l'écriture des données.

La classe `PrintStream` ne sait pas gérer les caractères *Unicode*. Sur les 16 bits utilisés pour coder un caractère *Unicode*, les 8 bits de poids forts seront perdus lors de l'écriture.

La représentation textuelle d'un objet quelconque est réalisée s'il existe une méthode `toString` dans la définition de sa classe. Cette méthode fait partie des méthode de la classe `Object`.

```
public class PrintStream extends FilterOutputStream {
    public PrintfStream(OutputStream out)
    public PrintfStream(OutputStream out, boolean autoflush)
    public boolean checkError()
    public void close()
    public void print(Object obj)
    public synchronized void print(String s)
    public synchronized void print(char [ ] s)
    public void print(char c)
    public void print(int i)
    public void print(long  l)
    public void print(float f)
    public void print(double d)
    public void print(boolean b)
    public void println()
    public synchronized void println(Object obj)
    public synchronized void println(String s)
    public synchronized void println(char [ ] s)
    public synchronized void println(char c)
    public synchronized void println(int i)
    public synchronized void println(long  l)
    public synchronized void println(float f)
    public synchronized void println(double d)
    public synchronized void println(boolean b)
    public void write(int b) throws IOException
    public void write(byte[ ]  b, int début, int nbre) throws IOException
}
```

Pour pouvoir écrire l'ensemble des caractères *Unicode*, on utilisera la classe `PrintWriter`. Depuis la version 1.1, on utilisera plutôt la classe `PrintWriter`

```
public class PrintWriter extends Writer {
  PrintWriter(Writer out)
  PrintWriter(Writer out, boolean autoFlush)
  PrintWriter(OutputStream out)
  PrintWriter(OutputStream out, boolean autoFlush)
  boolean checkError()
  void close()
  void flush()
  void print(boolean b)
  void print(char c)
  void print(int i)
  void print(long l)
  void print(float f)
  void print(double d)
  void print(char[] s)
  void print(String s)
  void print(Object obj)
  void println()
  void println(boolean x)
  void println(char x)
  void println(int x)
  void println(long x)
  void println(float x)
  void println(double x)
  void println(char[] x)
  void println(String x)
  void println(Object x)
  void setError()
  void write(int c)
  void write(char[] buf, int off, int len)
  void write(char[] buf)
  void write(String s, int off, int len)
  void write(String s)
}
```

On notera que cette classe n'est pas une classe dérivée de la classe `FilterWriter` mais est une classe directement dérivée de la classe `Writer`.

Nous avons déjà utilisé ce type d'objet ; la variable `static System.out` est une instance de la classe `PrintWriter`.

15.4.7 Définir ses propres filtres

A TERMINER

15.5 Les tableaux et les chaînes

15.5.1 Les tableaux d'octets

Ces classes permettent d'utiliser un tableau pour flot de données.

La classe **ByteArrayInputStream** possède deux constructeurs :

```
ByteArrayInputStream(byte[] buf)
ByteArrayInputStream(byte[] buf, int offset, int length)
```

L'argument **buf** est le flot d'entrée dans les deux cas ; dans le deuxième cas, seul la partie du tableau comprise entre les indices **offset** et **offset+length-1** constitue le flot d'entrée.

Toutes les autres méthodes (**available**, **close**, **mark**, **markSupported**, **read**, **reset**, **skip**) sont héritées de la classe **InputStream**.

La classe **ByteArrayOutputStream** possède deux constructeurs :

```
public ByteArrayOutputStream()
public ByteArrayOutputStream(int size)
```

Le premier constructeur crée un tableau d'octets dont la taille initiale est de 32 octets ; si un tableau plus grand est nécessaire, *Java* redimenssionne automatiquement celui-ci. Le deuxième constructeur fixe la taille initiale du tableau.

`public void writeTo (OutputStream out)` *throws IOException*

Ecrit le contenu du tableau dans l'argument **out**. Equivalent à write(buf, 0, count).

`public void reset ()`

Remet à zéro la valeur du champ **count** de manière de supprimer tout ce qui a été rangé dans le tableau.

`public byte[] toByteArray ()`

Crée une copie du tableau et retourne ce nouveau tableau.

`public String toString (String enc)` *throws UnsupportedEncodingException*

Converti le tableau en un **String** et retourne ce dernier.

A TERMINER encoding

Toutes les autres méthodes (**close**, **write**) sont héritées de la classe **OutputStream**.

```
import java.io.*;

public class ByteArray {
   public static void main(String[] args) {
      byte[] tabEntrée = { (byte)'a', (byte)'b', (byte)'c', (byte)'d', (byte)'e'};
      ByteArrayInputStream entrée = new ByteArrayInputStream(tabEntrée);
      ByteArrayOutputStream sortie = new ByteArrayOutputStream();
      int b;
      while ((b=entrée.read()) >= 0) sortie.write(b);
      entrée.reset();
      entrée.skip(3);
      while ((b=entrée.read()) >= 0) sortie.write(b);
      System.out.println(sortie);
   }
}
```

15.5.2 Les chaînes

Les classes **StringBufferInputStream**, **StringReader** et **StringWriter** permettent d'utiliser un objet de type **String** pour flot de données. Depuis la version 1.1, la classe **StringBufferInputStream** est remplacé par la classe **StringReader**.

```
public class StringReader extends Reader {
   StringReader(String s) {}
   void close() {}
   void mark(int readAheadLimit)  {}
   boolean markSupported() {}
   int read() {}
   int read(char[] cbuf, int off, int len) {}
   boolean ready() {}
   void reset() {}
   long skip(long ns) {}
}
```

```
public class StringWriter extends Writer {
   StringWriter()
   StringWriter(int initialSize)
   void  close()
   void flush()
```

```
      StringBuffer getBuffer()
      String toString()
      void write(int c)
      void write(char[] cbuf, int off, int len)
      void write(String str)
      void write(String str, int off, int len)
   }
```

```
String s = ...;
InputStream in = new StringBufferInputStream(s);
try {
   int c;
   while ((c = in.read()) >= 0) { ... }
   }
   catch (IOException e) { ... }
```

15.6 Les fichiers

Avec la classe **File**, *Java* permet de manipuler les fichiers selon la même convention quelque soit la système d'exploitation. Elle permet également des méthodes pour manipuler les répertoires, "lister" les fichiers d'un répertoire, examiner les propriétés d'un fichier, etc.

15.6.1 La classe FileDescriptor

A TERMINER

15.6.2 La classe File

```
public class File extends Object {
   public final static String pathSeparator;
   public final static String pathSeparatorChar;
   public final static String separator;
   public final static String separatorChar;
   public File(String chemin) throws NullPointerException;
   public File(String chemin, String nom);
   public File(String repertoire, String nom);

   public boolean canRead();
   public boolean canWrite();
   public boolean delete();
   public boolean equals(Object obj);
   public boolean exists();
   public String getAbsolutePath();
   public String getName();
   public String getParent();
   public String getPath();
   public String getPath();
   public int hashCode();
   public boolean isAbsolute();
   public boolean isDirectory();
   public boolean isFile();
   public long lastModified();
   public long length();
   public String [ ] list();
   public String [ ] list(FilenameFilter filtre);
   public boolean mkdir();
   public boolean mkdirs();
   public boolean renameTo();
   public String toString();
}
```

Les objets de type **File** désignent des répertoires ou des fichiers. Pour créer un tel objet, on utilise évidemment un des constructeurs de cette classe qui prend en argument selon le cas :

- le nom absolu du fichier
- le nom relatif
- le nom du répertoire suivi du nom (nom relatif au répertoire spécifié sous forme de String ou de File)

```
File fichier1   = new File("/tmp/toto.txt");      // fichier avec nom absolu
File répertoire = new File("/tmp");               // répertoire avec nom absolu
File fichier2   = new File("toto.txt");           // nom relatif
File fichier3   = new File("/tmp","toto.txt");
File fichier4   = new File(répertoire,"toto.txt");
```

La création d'un objet de type File n'engendre pas d'erreurs ; il n'est pas nécessaire que le fichier ou répertoire spécifié existe. Comme on peut le remarquer, les méthodes de cette classe ne permettent pas d'écrire ou de lire dans le fichier spécifié. Pour pouvoir le faire, on passera par la création d'un objet de la classe `FileInputStream` ou `FileOutputStream`.

La classe `File` dispose de méthodes pour

- créer (`mkdir`, `mkdirs`), supprimer (`delete`) , renommer (renameTo) des fichiers et des répertoires,
- lister (`list`) des répertoires,
- consulter les attributs d'un objet de type `File` (`canRead`, `canWrite`, `exists`, `getName`, `getPath`, `isDirectory`, etc.).

15.6.3 Les classes FileInputStream, FileReader, FileOutputStream et FileWriter

La classe `FileInputStream` permet lire des données à partir d'un fichier désigné par un objet de type `File` ou `FileDescriptor`.

```
public class FileInputStream extends InputStream {
   public FileInputStream(String nom) throws FileNotFoundException, IOException;
   public FileInputStream(File f) throws FileNotFoundException, IOException;
   public FileInputStream(FileDescriptor f);
   public int available() throws IOException;
   public void close() throws IOException;
   public final FileDescriptor getFD() throws IOException;
   public int read() throws IOException;
   public int read(byte[ ] tampon) throws IOException;
   public int read(byte[ ] tampon, int debut, int nbre) throws IOException;
   public long skip(long nbre) throws IOException;
   protected void finalize()throws IOException;
}
```

De même, la classe `FileReader` permet lire des fichiers de caractères ; fichiers désignés par un objet de type `File` ou `FileDescriptor`.

```
public class FileReader  extends InputStreamRaeder {
  FileReader(String fileName)
  FileReader(File file)
  FileReader(FileDescriptor fd)
}
```

La classe `FileWriter` permet d'écrire dans des fichiers, des caractères. Ces fichiers désignés par un objet de type `File` ou `FileDescriptor`.

```
public class FileWriter extends OutputStreamWriter {
   public FileWriter(String fileName)
   public FileWriter(String fileName, boolean append)
   public FileWriter(File file)
   public FileWriter(FileDescriptor fd)
}
```

Voici une exemple de programme qui recopie un fichier dans un répertoire ou sous un autre nom.

```
import java.io.*;

public class CopieFichier {
   public static void main(String[] args) throws IOException {
      if (args.length != 2)
         System.err.println("Usage: java CopieFichier FichierSource FichierDestination");
      else {
         File src = new File(args[0]), dest = new File(args[1]);
         FileInputStream fsrc = null;
         FileOutputStream fdest = null;
         if (!src.exists()) erreur(args[0] + "Introuvable");
         if (!src.isFile()) erreur(args[0] + " n'est pas un fichier");
         if (!src.canRead()) erreur(args[0] + " Interdit de lecture");
         if (dest.isDirectory()) dest = new File(dest, src.getName());
         if (dest.exists()) {
            if (!dest.canWrite()) erreur(args[1]+ " Interdit d'écriture");
            System.out.print("Remplacer le fichier " + args[1] + " existant? (O/N): ");
            System.out.flush();
            BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
            String response = in.readLine();
            if (!response.equals("O") && !response.equals("o")) erreur("Copie annulée");
         }
         else {
            String parent = dest.getParent();
            if (parent == null) parent = System.getProperty("user.dir");
            File dir = new File(parent);
            if (!dir.exists()) erreur("FileCopy: destination directory doesn't exist: " + parent);
```

```
            if (dir.isFile()) erreur("FileCopy: destination is not a directory: " + parent);
            if (!dir.canWrite()) erreur("FileCopy: destination directory is unwriteable: " + parent);
         }
         fsrc = new FileInputStream(src);
         fdest = new FileOutputStream(dest);
         byte[] buffer = new byte[4096];
         int bytes_read;
         while((bytes_read = fsrc.read(buffer)) != -1)
            fdest.write(buffer, 0, bytes_read);

         if (fsrc != null)  fsrc.close();
         if (fdest != null)  fdest.close();
      }
   }
   private static void erreur(String msg) throws IOException {
      throw new IOException(msg);
   }
}
```

15.6.4 L'interface FilenameFilter

Cette interface permet, comme son nom l'indique, de choisir un sous ensemble des fichiers disponibles en les filtrant selon un critère qu'implante la méthode **accept** ; méthode unique de cette interface.

```
public interface FilenameFilter {
    public boolean accept(File dir, String name)
}
```

Voici une exemple de programme qui affiche les fichiers sources *Java* du répertoire courant.

```
import java.io.*;

public class FiltreFichierJava implements FilenameFilter {
   public static void main(String args[]) throws IOException {
      File f = new File(".");
      FilenameFilter filter = new FiltreFichierJava();
      System.out.println("Fichiers Java : " );
      String[] noms = f.list(filter);
      for (int i = 0;  noms != null && i < noms.length; i++)
         System.out.println("\t" + noms[i]);
   }

   public boolean accept(File dir, String name) {
      return (name.endsWith(".java"));
   }
}
```

15.6.5 RandomAccessFile

Les fichiers dont nous avons parlé jusqu'à présent étaient des fichiers séquentiels. *Java* fournit une gestion pour les fichiers à accès non séquentiel. Par exemple, ce type de fichier est intéressant pour les fihiers **zip** ; l'extraction d'un fichier contenu dans un fichier **zip** sera bien plus efficace avec un **RandomAccessFile** qu'un fichier séquentiel standard.

```
public class RandomAccessFile extends Object implements DataOutput, DataInput {
      public RandomAccessFile(String name, String mode) throws IOException
      public RandomAccessFile(File file, String mode) throws IOException
      public final FileDescriptor getFD() throws IOException
      public int read() throws IOException
      public int read(byte b[], int off, int len) throws IOException
      public int read(byte b[]) throws IOException
      public final void readFully(byte b[]) throws IOException
      public final void readFully(byte b[], int off, int len) throws IOException
      public int skipBytes(int n) throws IOException
      public void write(int b) throws IOException
      public void write(byte b[]) throws IOException
      public void write(byte b[], int off, int len) throws IOException
      public long getFilePointer() throws IOException
      public void seek(long pos) throws IOException
      public long length() throws IOException
      public void close() throws IOException
      public final boolean readBoolean() throws IOException
```

```
    public final byte readByte() throws IOException
    public final int readUnsignedByte() throws IOException
    public final short readShort() throws IOException
    public final int readUnsignedShort() throws IOException
    public final char readChar() throws IOException
    public final int readInt() throws IOException
    public final long readLong() throws IOException
    public final float readFloat() throws IOException
    public final double readDouble() throws IOException
    public final String readLine() throws IOException
    public final String readUTF() throws IOException
    public final void writeBoolean(boolean v) throws IOException
    public final void writeByte(int v) throws IOException
    public final void writeShort(int v) throws IOException
    public final void writeChar(int v) throws IOException
    public final void writeInt(int v) throws IOException
    public final void writeLong(long v) throws IOException
    public final void writeFloat(float v) throws IOException
    public final void writeDouble(double v) throws IOException
    public final void writeBytes(String s) throws IOException
    public final void writeChars(String s) throws IOException
    public final void writeUTF(String str) throws IOException
}
```

A TER-
MINER

15.7 Les tubes (pipes)

Les classes `PipedInputStream`, `PipedReader`, `PipedOutputStream` et `PipedWriter` implante les entrées sorties pour les tubes (*pipes*).

15.7.1 PipedInputStream, PipedReader

```
public class PipedInputStream extends InputStream {
    byte[]   buffer;
    int in;
    int out;
    static int PIPE_SIZE;
    PipedInputStream(PipedOutputStream src)
    PipedInputStream()
    int available()
    void close()
    void connect(PipedOutputStream src)
    int read()
    int read(byte[] b, int off, int len)
    void receive(int b)
}
```

```
public class PipedReader extends Reader {
    PipedReader()
    PipedReader(PipedWriter src)
    void   close()
    void   connect(PipedWriter src)
    int   read(char[] cbuf, int off, int len)
}
```

15.7.2 PipedOutputStream, PipedWriter

```
public class PipedOutputStream extends OutputStream {
    PipedOutputStream(PipedInputStream snk)
    PipedOutputStream()
    void close()
    void connect(PipedInputStream snk)
    void flush()
    void write(int b)
    void write(byte[] b, int off, int len)
}
```

```
public class PipedWriter extends Writer {
    PipedWriter()  {}
    PipedWriter(PipedReader sink)  {}
    void   close()   {}
```

```
      void   connect(PipedReader sink)  {}
      void   flush()   {}
      void   write(char[] cbuf, int off, int len)  {}
   }
```

Le squelette d'un programme type utlisant les tubes pour connecter les sorties d'un premier thread à l'entrée d'un deuxième thread est de la forme suivante :

```
...
public static void main(String[] args) throws IOException {
    PipedOutputStream prod = new PipedOutputStream();
    PipedInputStream cons = new PipedInputStream();

    consumer.connect(producer);

    PremierThread th1  = new PremierThread (cons);
    DeuxiemeThread th2 = new DeuxiemeThread(prod);

    th1.start();
    th2.start();

    try {  Thread.sleep(5000); }
    catch (InterruptedException e) {}

    th1.stop();
    th2.stop();

    prod.close();
    cons.close();
}
...
```

15.8 StreamTokenizer

Java fournit, avec la classe **StreamTokenizer**, des facilités pour faire des analyses relativement simples. Le processus d'analyse est contrôlé par une table d'analyse et un certain nombre de drapeaux (*flags*). Les données en entrée sont découpées en unités lexicales et chaque unité lexicale est lue d'un coup. La lecture d'une unité lexicale se fait grâce à la méthode **nextToken**.

```
public class StreamTokenizer {
   public int ttype
   public final static int TT_EOF
   public final static int TT_EOL
   public final static int TT_NUMBER
   public final static int TT_WORD
   public String sval
   public double nval
   public StreamTokenizer(InputStream is)
   public StreamTokenizer(Reader r)
   public void resetSyntax()
   public void wordChars(int low, int hi)
   public void whitespaceChars(int low, int hi)
   public void ordinaryChars(int low, int hi)
   public void ordinaryChar(int ch)
   public void commentChar(int ch)
   public void quoteChar(int ch)
   public void parseNumbers()
   public void eolIsSignificant(boolean flag)
   public void slashStarComments(boolean flag)
   public void slashSlashComments(boolean flag)
   public void lowerCaseMode(boolean fl)
   public int nextToken() throws IOException
   public void pushBack()
   public int lineno()
   public String toString()
}
```

`public int  nextToken () throws IOException`

Lit l'unité lexicale suivante et retourne le type de l'unité lexicale reconnue ; type stocké également dans le champ **ttype** de cette classe. Les champs **nval** et **sval** contiennent des informations complémentaires sur l'unité lexicale lue.

Les champs

- `nval` contient la valeur du nombre lu.
- `sval` contient le mot lu sous forme d'une chaîne de caractères.

Les constantes

- `TT_EOF` indique une fin de flot.
- `TT_EOL` indique une fin de ligne.
- `TT_NUMBER` indique la lecture d'un nombre
- `TT_WORD` indique la lecture d'un mot.

Le flot d'entrée est composé de caractères "spéciaux" (qui vont avoir un sens particulier) et des caractères ordinaires :

- Les caractères de '`A`' à '`Z`', de '`a`' à '`z`' et de '
  `u00A0`' à '
  `u00FF`' sont les caractères alphabétiques
- Les séparateurs sont les caractères compris entre '
  `u0000`' à '
  `u0020`'.
- Le caractères '`/`' est le caractère de début de commentaire ligne.
- Les apostrophes et les guillemets délimitent des chaînes.

Les programmes qui utilisent la classe `StreamTokenizer` ressemble très souvent à :

```
import java.io.*;

public class Analyse {
   public static void main(String args[]) throws Exception {
      if (args.length != 1) {
         System.out.println("usage: java Analyse <fichier>");
         System.exit(0);
      }
      StreamTokenizer in = new StreamTokenizer(new FileReader(args[0]));
      while (in.nextToken() != StreamTokenizer.TT_EOF) {

         switch (in.ttype) {
            case StreamTokenizer.TT_EOL :
               System.out.println("Fin de ligne");
               break;
            case StreamTokenizer.TT_NUMBER :
               System.out.println("Nombre      " + in.nval);
               break;
            case StreamTokenizer.TT_WORD :
               System.out.println("Mot         " + in.sval);
               break;
            case '\'' :
            case '"' :
               System.out.println("Chaine      " +  in.sval);
               break;
         default:
               System.out.println("Caractère   " + (char)in.ttype);
         }
      }
   }
}
```

Il existe des méthodes pour redéfinir les caractères spéciaux, ordinaires, etc :

`public void  wordChars (int low, int hi)`

Les caractères compris dans l'intervalle [`low`, `hi`] deviennent des caractères qui composent un mot.

`public void  whitespaceChars (int low, int hi)`

Les caractères compris dans l'intervalle [`low`, `hi`] deviennent des séparateurs.

`public void  ordinaryChars (int low, int hi)`

Les caractères compris dans l'intervalle [`low`, `hi`] deviennent des caractères oridinaires.

`public void  ordinaryChar (int ch)`

Le caractère `ch` devient un caractère oridinaire.

`public void  commentChar (int ch)`

Le caractère `ch` marque le début d'un commentaire.

`public void  quoteChar (int ch)`

Les chaînes de caractères peuvent être délimités par le caractère `ch`.

`public void eolIsSignificant (boolean flag))`

Si `flag` est faux, une fin de ligne est traité comme le caractère *espace* i.e. comme un séparateur ; *faux* est la valeur par défaut.

`public void slashStarComments (boolean flag))`

Si `flag` est vrai, un texte encadré par `/*` et `*/` est considéré comme un commentaire ; *faux* est la valeur par défaut.

`public void slashSlashComments (boolean flag))`

Si `flag` est vrai, un texte encadré commençant `//` est considéré comme un commentaire ligne ; *faux* est la valeur par défaut.

`public void lowerCaseMode (boolean flag))`

Si `flag` est vrai, tous les caractères des unités lexicales sont convertis en minuscules ; *faux* est la valeur par défaut.

`public void pushBack ())`

Remet dans le flot la dernière unité lexicale lue. Une seule unité lexicale peut être remis ; il est équivalent de faire plusieurs appels ou un seul appel de cette méthode.

`public int lineno ())`

Retourne le numéro de ligne courante ; utile pour préciser les erreurs.

`public void resetSyntax ()`

Supprime toutes les définitions de syntaxe. Tous les caractères deviennent des caractères oridinaires ; une unité lexicale devient tout simplement le prochain caractère.

A TERMINER

`public void parseNumbers ()`

15.9 Conversion des types de flots

Les classes `InputStreamReader` et `InputStreamReader` permettent la conversion des flots d'octets en flots de caractères et inversement.

Supponsons que l'on veuille transformer l'entrée standard (qui est un `InputStream`) en un `BufferedReader`. Les constructeurs de la classe `BufferedReader` prennent un objet de type `Reader` en argument et non pas un `InsputStream`. Il faut donc convertir `System.in` en un `Reader` :

```
Reader in = new InputStreamRaeder(System.in);
```

On peut alors obtenir un BufferedReader à partir de ce nouvel objet :

```
BufferedReader bin = new BufferedReader(in);
```

15.10 Séquence de flots

A TERMINER

16. Serialization

Sommaire

16.1 Lire et écrire des objets 117
16.2 Contrôle de la “serialization” 118
16.2.1 Implanter l’interface Serializable *118*
16.2.2 Redéfinir les méthodes readObject et writeObject *119*
16.2.3 L’interface ObjectOutput *119*
16.2.4 L’interface ObjectInput *120*
16.2.5 La classe ObjectInputStream *120*
16.2.6 La classe ObjectOutputStream *120*
16.2.7 Les champs transitoire (transient) *121*
16.3 Compression 121
16.4 L’interface Externalizable 121
16.5 Gestion des versions 122

16.1 Lire et écrire des objets

Le concept de `Object Serialization` a été introduit à partir de version 1.1. Il permet de lire ou d’écrire des objets (instance d’une classe) ; il ne s’agit pas, lorsqu’il s’agit d’écrire un objet, de donner une représentation textuelle d’un objet, mais de donner une représentation binaire. Cette représentation devra évidemment être indépendante de la plateforme utilisée.

L’intérêt de pouvoir donner une représentation binaire d’un objet est partciulièrement utile dans les cas suivants :

- Programmer le couper/coller
- transférer des objets à travers les sockets dans une application orienté réseau (RMI)
- d’archiver des objets sur des supports non volatiles pour une utilisation ultérieure (Java Beans).

L’écriture d’un objet est relativement simple; par exemple, l’exemple suivant permet d’écrire l’état d’un objet dans un fichier :

```
FileOutputStream sortie = new FileOutputStream("resultat");
ObjectOutputStream s = new ObjectOutputStream(sortie);
s.writeObject(new Date());
s.flush();
```

De même, la lecture d’un objet devrait correspondre à :

```
FileInputStream entree = new FileInputStream("resultat");
ObjectInputStream s = new ObjectInputStream(in);
169
Date date = (Date)s.readObject();
```

On aura compris que, comme pour l’écriture, à travers l’interface `Serializable`, le concepteur de la classe `Date` devra fournir la méthode `readObject`[1].

Sauf contre indication, il existe une implantation par défaut pour les méthodes `writeObject` et `readObject`.

La méthode `writeObject` :

- les champs de type primitif non statique sont écrits sauf ceux qualifiés de *transient* (voir **??**)

[1] Il existe un comportement par défaut que permet dans beaucoup de cas de se passer de la redéfinition de la méthode `readObject`

– les champs de type non primitif et non statiques sont écrits récursivement. Si un même objet est référencé plusieurs fois, une seule écriture est réalisée.

La méthode `readObject` :

– crée tout d'abord une instance de l'objet à lire en utilisant le constructeur par défaut (sans argument).
– lit les données un à un dans l'ordre inverse de celui de l'écriture.

16.2 Contrôle de la “serialization”

16.2.1 Implanter l'interface Serializable

Un premier niveau de contôle, le concepteur d'une classe peut ou non autoriser la lecture et l'écritutre des instances de sa classe.

Pour ce faire, *Java* définit une interface `Serializable`. Ainsi, un objet ne sera écrit par la méthode `writeObject` que si la classe à laquelle elle appartient implante cette interface.
Un objet est dit *Serializable* que si elle implante l'interface `Serializable`. L'interface `Serializable` définie par *Java* est une interface vide.

```
package java.io;
public interface Serializable
```

Autrement dit, pour qu'un objet puisse être écrit ou lu, il n'est pas nécessaire de définir quoique ce soit. Il suffit juste de signaler, lors de la définition de la classe, que celle-ci implante l'interface `Serializable`. Il existe une implantation par défaut de la méthode `readObject` et `writeObject`.

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

class UneDate implements Serializable {
  static UneDate liste = null;
  int jour, mois, année;
  UneDate suiv = null;
  public UneDate(int j, int m, int a) {
    jour = j; mois = m; année = a;
    suiv = liste;
    liste = this;
  }

  public static void main(String args[]) throws Exception {
    FileOutputStream f = new FileOutputStream("Resultat");
    ObjectOutput s = new ObjectOutputStream(f);
    s.writeObject("Des dates");
    UneDate [] dd = {new UneDate(25, 12, 98), new UneDate(15, 9, 57)};
    s.writeObject(UneDate.liste);
    s.flush();
    UneDate.liste = null;

    FileInputStream ff = new FileInputStream("Resultat");
    ObjectInput t = new ObjectInputStream(ff);
    String str = (String) t.readObject();
    UneDate.liste = (UneDate) t.readObject();
    System.out.println(str +"\n");

    for (UneDate d = UneDate.liste; d!=null; d = d.suiv)
      System.out.println(d.jour + " " + d.mois + " " + d.année );
  }
}
```

Une classe n'a pas besoin d'implanter l'interface `Serializable` si elle dérive d'une classe qui le fait.

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

class  UneDateEvent extends UneDate {
  String event;
  public UneDateEvent(int j, int m, int a, String s) {
    super(j, m, a);
    event = s;
  }
  public static void main(String args[]) throws Exception {
    FileOutputStream f = new FileOutputStream("Resultat");
    ObjectOutput s = new ObjectOutputStream(f);
    s.writeObject("Des dates Evènements");
```

```
    UneDate [] dd = {new UneDateEvent(25, 12, 98, "Noel 98"),
                     new UneDateEvent(15, 9, 57, "Anniversaire")};
    s.writeObject(UneDateEvent.liste);
    s.flush();
    UneDateEvent.liste = null;

    FileInputStream ff = new FileInputStream("Resultat");
    ObjectInput t = new ObjectInputStream(ff);
    String str = (String) t.readObject();
    UneDateEvent.liste = (UneDateEvent) t.readObject();
    System.out.println(str +"\n");

    for (UneDateEvent d = (UneDateEvent)  UneDateEvent.liste; d!=null; d = (UneDateEvent)d.suiv)
      System.out.println(d.event + ": " + d.jour + " " + d.mois + " " + d.année );
  }
}
```

16.2.2 Redéfinir les méthodes readObject et writeObject

Dans certains cas, l'implantation par défaut ne convient pas et il faut alors redéfinir les méthodes **readObject** et **writeObject**. Voici un squelette de ces deux méthodes :

```
private void writeObject(ObjectOutputStream s) throws IOException {
   s.defaultWriteObject();
   // code particularisé
}

private void readObject(ObjectInputStream s) throws IOException  {
   s.defaultReadObject();
   // code particularisé
   ...
   // suivi d'éventuelles mises à jour de l'objet
}
```

Supposons que, de notre classe **Date** ci-dessus, nous ne voulions sauvegarder que le jour et le mois et non pas l'année. Il faut alors redéfinir les méthodes **writeObject** et **readObject**.

La classe **ObjectOutputStream** implante l'interface **DataOutput** qui permet d'écrire des données primitives grâce aux méthodes **writeInt**, **writeFloat**, etc.

```
class UneDate98 implements Serializable {
   static int ii = 2;
   int jour, mois, année;
   public UneDate98(int j, int m, int a) {
      jour = j; mois = m; année = a;
   }

   private void writeObject(ObjectOutputStream s) throws IOException {
      s.defaultWriteObject();
      // code particularisé
      s.writeInt(jour);
      s.writeInt(mois);
   }
   private void readObject(ObjectInputStream s)
                 throws OptionalDataException, ClassNotFoundException, IOException {
      s.defaultReadObject();
      // code particularisé
      jour = s.readInt();
      mois = s.readInt();
      // suivi d'éventuelles mises à jour de l'objet
      année = 1998;
   }

}
```

16.2.3 L'interface ObjectOutput

L'interface **ObjectOutput** étend l'interface **DataOutput** pour permettre de lire des données de type primitifs ou pas.

```
public interface ObjectOutput extends DataOutput {
   public void writeObject(Object obj) throws IOException;
   public void write(int b) throws IOException;
   public void write(byte b[]) throws IOException;
```

```
      public void write(byte b[], int off, int len) throws IOException;
      public void flush() throws IOException;
      public void close() throws IOException;
   }
```

16.2.4 L'interface ObjectInput

L'interface `ObjectInput` étend l'interface `DataInput` pour permettre de lire des données de type primitifs ou pas.

```
public interface ObjectInput extends DataInput{
   public Object readObject() throws ClassNotFoundException, IOException;
   public int read() throws IOException;
   public int read(byte b[]) throws IOException;
   public int read(byte b[], int off, int len) throws IOException;
   public long skip(long n) throws IOException;
   public int available() throws IOException;
   public void close() throws IOException;
}
```

16.2.5 La classe ObjectInputStream

La classe `ObjectOutputStream` implante l'interface `ObjectOutput`.

```
public class ObjectInputStream extends InputStream {
   public ObjectInputStream(InputStream in) throws IOException, StreamCorruptedException
   public final Object readObject()
                       throws OptionalDataException, ClassNotFoundException, IOException
   public final void defaultReadObject()
                       throws IOException, ClassNotFoundException, NotActiveException
   public synchronized void registerValidation(ObjectInputValidation obj, int prio)
                       throws NotActiveException, InvalidObjectException
   protected Class resolveClass(ObjectStreamClass v) throws IOException, ClassNotFoundException
   protected Object resolveObject(Object obj) throws IOException
   protected final boolean enableResolveObject(boolean enable) throws SecurityException
   protected void readStreamHeader() throws IOException, StreamCorruptedException
   public int read() throws IOException
   public int read(byte data[], int offset, int length) throws IOException
   public int available() throws IOException
   public void close() throws IOException
   public boolean readBoolean() throws IOException
   public byte readByte() throws IOException
   public int readUnsignedByte() throws IOException
   public short readShort() throws IOException
   public int readUnsignedShort() throws IOException
   public char readChar() throws IOException
   public int readInt() throws IOException
   public long readLong() throws IOException
   public float readFloat() throws IOException
   public double readDouble() throws IOException
   public void readFully(byte data[]) throws IOException
   public void readFully(byte data[], int offset, int size) throws IOException
   public int skipBytes(int len) throws IOException
   public String readLine() throws IOException
   public String readUTF() throws IOException
}
```

A TERMINER

16.2.6 La classe ObjectOutputStream

```
public class ObjectOutputStream extends OutputStream
                             implements ObjectOutput, ObjectStreamConstants {
   public ObjectOutputStream(OutputStream out) throws IOException
   public final void writeObject(Object obj) throws IOException
   public final void defaultWriteObject() throws IOException
   public void reset() throws IOException
   protected void annotateClass(Class cl) throws IOException
   protected Object replaceObject(Object obj) throws IOException
   protected final boolean enableReplaceObject(boolean enable) throws SecurityException
   protected void writeStreamHeader() throws IOException
   public void write(byte b[]) throws IOException
   public void write(byte b[], int off, int len) throws IOException
   public void flush() throws IOException
```

```
    protected void drain() throws IOException
    public void close() throws IOException
    public void writeBoolean(boolean data) throws IOException
    public void writeByte(int data) throws IOException
    public void writeShort(int data) throws IOException
    public void writeChar(int data) throws IOException
    public void writeInt(int data) throws IOException
    public void writeLong(long data) throws IOException
    public void writeFloat(float data) throws IOException
    public void writeDouble(double data) throws IOException
    public void writeBytes(String data) throws IOException
    public void writeChars(String data) throws IOException
    public void writeUTF(String data) throws IOException
 }
```

A TERMINER

16.2.7 Les champs transitoire (transient)

Certains champs dont on ne veut pas permettre la sauvegarde ou la restauration pour diverses raisons (économie, sécurité etc.) doivent être qualifés de **transient**. Dans l'exemple précédant, au lieu de redéfinir les méthodes **writeObject** et **readObject**, pour ne pas sauvegarder et restaurer le champ **année**, il aurait suffit de le qualifier de transient :

```
transient int année;
```

16.3 Compression

Avec la serialisation, on dispose d'une représentation binaire des objets. Il est possible avec les classes du package **java.util.zip** de donner une représentation binaire compressée. La compression pourra utiliser le format *Zip* ou *Gzip*.

```
try {
   GZIPOutputStream zip =
      new GZIPOutputStream(new FileOutputStream(fichier));
   ObjectOutputStream out = new ObjectOutputStream(zip);
   ...
}

try {
   GZIPInputStream zip =
      new GZIPInputStream(new FileInputStream(filename));
   ObjectInputStream in = new ObjectInputStream(zip);
   ...

}
```

16.4 L'interface Externalizable

```
public interface Externalizable extends Serializable {
      public void writeExternal(ObjectOutput out) throws IOException;
      public void readExternal(ObjectInput in) throws IOException, java.lang.ClassNotFoundException;
   }
```

L'interface **Externalizable** permet de gérer “à la main” le processus de *serialization*. Cette interface étend l'interface **Serializable** et définit deux méthodes **writeExternal** et **readExternal** qui se chargent du contrôle des la représentation binaires de tous les champs (même les champs hérités). Il n'y pas, dans ce cas, de comportement par défaut ; tout doit être précisément défini dans les méthodes **writeExternal** et **readExternal**.

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


class UneDateExtern  implements Externalizable {
  static UneDateExtern liste = null;
  int jour, mois, année;
  UneDateExtern suiv = null;
  public UneDateExtern() { }
  public UneDateExtern(int j, int m, int a) {
    jour = j; mois = m; année = a;
    suiv = liste;
    liste = this;
  }
```

```
    public void writeExternal(ObjectOutput s) throws IOException {
      s.writeInt(jour);
      s.writeInt(mois);
      s.writeInt(année);
      s.writeObject(suiv);
    }
    public void readExternal(ObjectInput s) throws IOException, ClassNotFoundException   {
      jour = s.readInt();
      mois = s.readInt();
      année = s.readInt();
      suiv = (UneDateExtern) s.readObject();
    }
  }

  class  External extends UneDateExtern  implements Externalizable {
    String event;
    public External() { super(); }
    public External(int j, int m, int a, String s) {
      super(j, m, a);
      event = s;
    }
    public void writeExternal(ObjectOutput s) throws IOException {
      s.writeObject(event);
      super.writeExternal(s);
    }

    public void readExternal(ObjectInput s) throws IOException, ClassNotFoundException  {
      event = (String) s.readObject();
      super.readExternal(s);
    }

    public static void main(String args[]) throws Exception {
      FileOutputStream f = new FileOutputStream("Resultat");
      ObjectOutput s = new ObjectOutputStream(f);
      s.writeObject("Des dates Evênements");
      UneDateExtern [] dd = {new External(25, 12, 98, "Noel 98"),
                       new External(15, 9, 57, "Anniversaire")};
      s.writeObject(External.liste);
      s.flush();
      External.liste = null;

      FileInputStream ff = new FileInputStream("Resultat");
      ObjectInput t = new ObjectInputStream(ff);
      String str = (String) t.readObject();
      System.out.println(str +"\n");
      External.liste = (External) t.readObject();

      for (External d = (External)  External.liste; d!=null; d = (External)d.suiv)
        System.out.println(d.event + ": " + d.jour + " " + d.mois + " " + d.année );
    }
  }
```

16.5 Gestion des versions

A terminer

La sérialisation des objets pose problème lorsque l'état physique de l'objet écrit est incohérent avec l'état physique de l'objet attendu en lecture. Imaginons un objet sérialisé et stocké sur un support persistant et que la désérialisation intervient bien plus tard alors que la machine virtuelle a évolué. Les données binaires stockées peuvent être non valides pour la nouvelle machine virtuelle !!!

17. Compression des données

Sommaire

18. Le package java.util

Sommaire

18.1 L'interface Enumeration 126
18.2 La classe Vector 126
18.3 La classe Stack 127
18.4 La classe Dictionary 128
18.5 La classe Hashtable 128
18.6 La classe BitSet 129
18.7 La classe StringTokenizer 129
18.8 La classe Random 130
18.9 La classe Date 130
18.10 Le package java.util jdk 1.2 130

Java, comme tout langage de programmation orienté objets, offre un certain nombre de classes et interfaces utiles au programmeur. L'ensemble de ces classes et interfaces sont regroupés dans le package **java.util**. La version de jdk1.1 fournit :

Les Classes :
BitSet :
Calendar :
Date :
Dictionary :
EventObject :
GregorianCalendar :
Hashtable :
ListResourceBundle :
Locale :
Observable :
Properties :
PropertyResourceBundle :
Random :
ResourceBundle :
SimpleTimeZone :
Stack :
StringTokenizer :
TimeZone :
Vector :

Les interfaces :
Enumeration :
EventListener :
Iterator :

18.1 L'interface Enumeration

La plupart des classes utilisent l'interface **Enumeration** pour itérer sur les éléments d'une suite.

```
public Interface Enumeration {
   public abstract boolean hasMoreElements()
   public abstract Object nextElement()
}
```

Ainsi, si l'on définit un nouveau type de données comme une vecteur, une suite, etc., on implantera cette interface pour permettre de balayer les éléments de cette nouvelle structure de données.

Voici une utilisation type des instance de classes qui implantent l'interface **Enumeration** :

```
Enumeration e = vecteur.elements();
while (e.hasMoreElements()) {
        x = e.nextElement());
        ...
}
```

Rien ne garantit l'ordre dans lequel les éléments sont fournis ; tout ce dont dont est sûr, c'est que tous les éléments seront fournis dans un ordre non prévisible. De plus, il n'est pas possible, avec cette interface, de revenir en arrière. Les éléments sont fournis une et une seule fois.

```
Enumeration e = vecteur.elements();
while (e.hasMoreElements()) {
        x = e.nextElement());
        ...
}

Enumeration e = vecteur.elements();
while (e.hasMoreElements()) {
        y = e.nextElement());
        ...


}
```

18.2 La classe Vector

Les instances de la classe **Vector** sont des tableaux qui est automatiquement agrandis en fonctions des besoins.

```
public class Vector implements Cloneable, Serializable {
   protected Object elementData[]
   protected int elementCount
   protected int capacityIncrement

   public Vector(int initialCapacity, int capacityIncrement)
   public Vector(int initialCapacity)
   public Vector()
   public final synchronized void copyInto(Object anArray[])
   public final synchronized void trimToSize()
   public final synchronized void ensureCapacity(int minCapacity)
   public final synchronized void setSize(int newSize)
   public final int capacity()
   public final int size()
   public final boolean isEmpty()
   public final synchronized Enumeration elements()
   public final boolean contains(Object elem)
   public final int indexOf(Object elem)
   public final synchronized int indexOf(Object elem, int index)
   public final int lastIndexOf(Object elem)
   public final synchronized int lastIndexOf(Object elem, int index)
   public final synchronized Object elementAt(int index)
   public final synchronized Object firstElement()
   public final synchronized Object lastElement()
   public final synchronized void setElementAt(Object obj, int index)
   public final synchronized void removeElementAt(int index)
   public final synchronized void insertElementAt(Object obj, int index)
   public final synchronized void addElement(Object obj)
   public final synchronized boolean removeElement(Object obj)
   public final synchronized void removeAllElements()
   public synchronized Object clone()
   public final synchronized String toString()
}
```

Les variables membres de cette classe sont

- `elementData` qui contient les éléments du vecteur,
- `elementCount` qui précise le nombre d'élément valide dans ce tableau
- `capacityIncrement` qui fixe de combien le vecteur doit être étendu quand nécessaire. Si cette valeur vaut 0, le vecteur est doublé à chaque redimensionnement.

La classe `Vector` fournit trois constructeurs

```
public Vector(int initialCapacity, int capacityIncrement)
public Vector(int initialCapacity)
public Vector()
```

où initialCapacity est la taille initiale du tableau et `capacityIncrement` est le taux croissance du tableau. Le troisième constructeur crée un vecteur avec deux valeurs par défaut.

Les méthodes

- `setElementAt(Object o, int i)` ajoute l'objet `o` à l'indice `i`.
- `insertElementAt(Object o, int i)` insère l'objet `o` à l'indice `i` après avoir décalé vers le sommet les éléments vides d'indice supérieur ou égal à `i`.
- `addElement(Object o)` ajoute l'objet `o` en fin de vecteur.
- `copyInto`(Object anArray[])) Recopie un tableau dans un vecteur.
- `removeElement(Object o)` supprime la première occurence de l'objet `o`.
- `removeElementAt(int i)` supprime l'élément figurant à l'indice `i` et tasse le vecteur.
- `removeAllElements()` supprime tous les éléments d'un vecteur.
- `elementAt(int i)` retourne l'élément rangé à indice `i`. L'exception `ArrayIndexOutOfBoundsException` est levée si `i` est une valeur invalide.
- `firstElement` retourne le premier élément du vecteur. L'exception `ArrayIndexOutOfBoundsException` est levée le vecteur est vide.
- `lastElement()` retourne le premier élément du vecteur. L'exception `ArrayIndexOutOfBoundsException` est levée le vecteur est vide.
- `elements()` retourne une énumération des éléments du vecteur.
- `indexOf(Object o)` retourne l'indice de la première occurrence de `o`.
- `indexOf(Object o), int i` retourne l'indice de la première occurrence de `o` à partir de l'indice `i`.
- `lastIndexOf(Object o)` retourne l'indice de la dernière occurrence de `o`.
- `lastIndexOf(Object o, int i)` retourne l'indice de la dernière occurrence de `o` à partir de l'indice `i`.
- `contains(Object o)` retourne vrai si l'objet `o` figure dans le vecteur.
- `size()` retourne la taille du vecteur.
- `setSize(int t)` fixe la taille du vecteur. Si la nouvelle taille est plus grande que la taille courante, le vecteur est complété par `null`. Si la nouvelle taille est plus petite que la taille courante, le vecteur est tronqué.
- `trimToSize()` Réduit la taille du vecteur de manière à ne contenir que les seuls éléments valides.
- `capacity` : retourne taille du tableau codant le vecteur.
- `ensureCapacity(int t)` augmente la taille du tableau codant le vecteur (si mécessaire).
- `isEmpty` retourne vrai si le vecteur est vide.

18.3 La classe Stack

Sous classe de la classe `Vector`, la classe `Stack` implante les piles (listes LIFO).

```
public class Stack extends Vector
   public Stack()
   public Object push(Object item)
   public synchronized Object pop()
   public synchronized Object peek()
   public boolean empty()
   public synchronized int search(Object o)
}
```

Cette classe fournit

- un unique constructeur sans argument et les
- la méthode `pop` qui dépile l'objet en sommet de pile
- la méthode `push` qui empile l'objet passé en argument en sommet de pile
- la méthode `peek` qui retourne l'objet en sommet de pile sans dépiler
- la méthode `empty` qui rend vrai si la pile est vide.
- la méthode `search` retourne la position de la première occurrence de l'objet passé en argument ; -1 s'il n'y figure pas.

18.4 La classe Dictionary

C'est une classe abstraite qui permet d'implanter les tableaux associatifs.

```
public abstract class Dictionary {
   public Dictionary()
   public abstract int size()
   public abstract boolean isEmpty()
   public abstract Enumeration keys()
   public abstract Enumeration elements()
   public abstract Object get(Object key)
   public abstract Object put(Object key, Object value)
   public abstract Object remove(Object key)
}
```

Vous l'aurez remarquer, cette classe ressemble plutôt à une interface qu'à une classe abstraite.
Les sous classes de **Dictionary** doivent implanter les méthodes :

- **size** qui rend le nombre de clés
- **isEmpty** qui rend vrai si il n'y aucune association (clé, valeur).
- **keys** retourne une énumération des clés.
- **elements** retourne une énumération des valeurs.
- **get** qui rend la valeur associée à une clé
- **put** qui permet d'associer une valeur à une clé. Cette méthode retourne l'ancienne valeur associée à la clé ; **null** si l'ancienne association n'existe pas.
- **remove**

18.5 La classe Hashtable

Sous classe de la classe **Dictionary**, la table de hachage est une manière de stocker des couple (clé, valeur). Dans une table de hachage, les objets utilisés comme clé doivent implanter les méthodes **hoshCode** et **equals**. En effet, un facteur important d'efficacité des tables de hachage est la génération des codes de hachage pour les clés.

```
public class Hashtable extends Dictionary implements Cloneable, Serializable {
   public Hashtable(int initialCapacity, float loadFactor)
   public Hashtable(int initialCapacity)
   public Hashtable()
   public int size()
   public boolean isEmpty()
   public synchronized Enumeration keys()
   public synchronized Enumeration elements()
   public synchronized boolean contains(Object value)
   public synchronized boolean containsKey(Object key)
   public synchronized Object get(Object key)
   protected void rehash()
   public synchronized Object put(Object key, Object value)
   public synchronized Object remove(Object key)
   public synchronized void clear()
   public synchronized Object clone()
   public synchronized String toString()
}
```

Une table de hachage est redimensionnée lorsque le rapport du nombre d'éléments dans la table sur la capacité totale de la table dépasse une certaine valeur qu'on appelle *facteur de charge*.

La classe **Hashtable** définit trois constructeurs :

- **Hashtable()** construit une table de hachage avec un capacité et un facteur de charge par défaut.
- **Hashtable(int t)** construit une table de hachage avec un capacité de **t** et un facteur de charge par défaut.
- **Hashtable(int t, float fdc)** construit une table de hachage avec un capacité de **t** et un facteur de charge **fdc**.

Outre les méthodes définies dans la classe **Dictionary**, la classe **Hashtable** définit les méthodes :

- **containsKey(Object Key)** qui retourne vrai si **key** est une clé de cette table.
- **contains(Object value)** qui retourne vrai si **value** est une valeur de cette table.
- **rehash** qui recalcule la table des codes avec une capacité supérieure.
- **clear** qui vide une table de hachage.
- **clone** qui crée une copie de la table des codes. Attention ! Seuls la table est copiée, pas les éléments et les clés.

Voici une exemple d'utilisation que l'on trouve dans la documentation de *jdk*.

```
Hashtable numbers = new Hashtable();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));
```

Pour aller recher un élément dans cette table, on fera :

```
Integer n = (Integer)numbers.get("two");
if (n != null) { System.out.println("two = " + n); }
```

18.6 La classe BitSet

La classe `BitSet` permet de manipuler des vecteurs de bits ; vecteurs redimensionable.

```
public final class BitSet implements Cloneable, Serializable {
   public BitSet()
   public BitSet(int nbits)
   public void set(int bit)
   public void clear(int bit)
   public boolean get(int bit)
   public void and(BitSet set)
   public void or(BitSet set)
   public void xor(BitSet set)
   public int hashCode()
   public int size()
   public boolean equals(Object obj)
   public Object clone()
   public String toString()
}
```

Les noms des méthodes sont assez explicites et il n'y a donc pas besoin de commenter cette classe.

18.7 La classe StringTokenizer

Cette classe fournit les outils pour décomposer une chaîne de caractères en unité lexicale. Contrairement à la classe `StreamTokenizer` (voir 15.8), `StringTokenizer` ignore tout de la syntaxe des nombres, des identificateurs, des chaînes, commentaires, etc.

A la création d'une instance d'un `StringTokenizer`, un ensemble de délimiteurs est défini qui permet de découper une chaîne de caractères en unité lexicale.

```
public class StringTokenizer implements Enumeration {
   public StringTokenizer(String str, String delim, boolean returnTokens)
   public StringTokenizer(String str, String delim)
   public StringTokenizer(String str)
   public boolean hasMoreTokens()
   public String nextToken()
   public String nextToken(String delim)
   public boolean hasMoreElements()
   public Object nextElement()
   public int countTokens()
}
```

Cette classe possède trois constructeurs :

```
public StringTokenizer(String str, String delim, boolean returnTokens)
public StringTokenizer(String str, String delim)
public StringTokenizer(String str)
```

`str` est la chaîne à découper en unité lexicale, `delim` est l'ensemble des délimiteurs et le boolean `returnTokens` à vrai rend également les délimiteurs comme unité lexicale. Par défaut, `delim` est définie par la chaîne `"\t\n\r\f"` et `returnTokens` est à faux.

Les méthodes

- `countTokens()` rend le nombre d'unité lexicale restant
- `hasMoreElements()` et `hasMoreToken()` rendent *vrai* s'il reste encore des unités lexicalesrend vrai s'il reste encore des unités lexicales dans la chaîne.
- `nextElement()` rend la prochaine unité sous la forme d'une instance d'un `Object`.
- `nextTocken()` rend la prochaine unité sous la forme d'une instance d'un `String`.
- `nextTocken(String delim)` rend la prochaine unité sous la forme d'une instance d'un `String` et ce en prenant `delim` comme nouvel ensemble de délimiteurs.

18.8 La classe Random

La classe **Random** permet de générer des séquences de nombres pseudo-aléatoires. Pourquoi ses séquences sont elles qualifiées de pseudo-aléatoire et non pas, tout simplement, aléatoires ? La raison en est que les quences engendrées sont dépendante d'une valeur initiale. A une valeur initiale, correspond une séquence bien précise qui est, tout à fait, reproductible. C'est cette valeur initiale que la constructeur de cette classe prend en argument. Le constructeur sans argument prend comme valeur initiale, une valeur basée sur le le temps courant.

```
public class Random implements Serializable {
   public Random()
   public Random(long seed)
   public synchronized void setSeed(long seed)
   protected synchronized int next(int bits)
   public void nextBytes(byte bytes[])
   public int nextInt()
   public long nextLong()
   public float nextFloat()
   public double nextDouble()
   public synchronized double nextGaussian()
}
```

Les méthodes **nextInt**, **nextLong**, **nextFloat** et **nextDouble** retournent une valeur (du type voulu) pseudo-aléatoire. La méthode **nextGaussian** retourne une valeur **double** pseudo-aléatoire distribuée selon avec une moyenne de 0 et un écart type de 1.

18.9 La classe Date

La classe **Date** permet de manipuler les dates et les instants du système utilisé. Sachez que le temps que vous obtenez sur votre ordinateur est généralement imprécis et n'utilise pas le système de temps universel (UTC : Universel Time Coordiante).

```
public class Date {
   public Date()
   public Date(long date)
   public Date(int year, int month, int date)
   public Date(int year, int month, int date, int hrs, int min)
   public Date(int year, int month, int date, int hrs, int min, int sec)
   public Date(String s)
   public static long UTC(int year, int month, int date, int hrs, int min, int sec)
   public static long parse(String s)
   public int getYear()
   public void setYear(int year)
   public int getMonth()
   public void setMonth(int month)
   public int getDate()
   public void setDate(int date)
   public int getDay()
   public int getHours()
   public void setHours(int hours)
   public int getMinutes()
   public void setMinutes(int minutes)
   public int getSeconds()
   public void setSeconds(int seconds)
   public long getTime()
   public void setTime(long time)
   public boolean before(Date when)
   public boolean after(Date when)
   public boolean equals(Object obj)
   public int compareTo(Date anotherDate)
   public int compareTo(Object o)
   public int hashCode()
   public String toString()
   public String toLocaleString()
   public String toGMTString()
   public int getTimezoneOffset()
}
```

18.10 Le package java.util jdk 1.2

La version de jdk1.2Beta3 fournit :

Les interfaces :
`Collection` :
`Comparator` :
`Enumeration` :
`EventListener` :
`Iterator` :
`List` :
`ListIterator` :
`Map` :
`Observer` :
`Set` :
`SortedMap` :
`SortedSet` :

Les Classes :
`ArrayList` :
`Arrays` :
`BitSet` :
`Calendar` :
`Collections` :
`Date` :
`Dictionary` :
`EventObject` :
`GregorianCalendar` :
`HashMap` :
`HashSet` :
`Hashtable` :
`LinkedList` :
`ListResourceBundle` :
`Locale` :
`Observable` :
`Properties` :
`PropertyPermission` :
`PropertyResourceBundle` :
`Random` :
`ResourceBundle` :
`SimpleTimeZone` :
`Stack` :
`StringTokenizer` :
`TimeZone` :
`TreeMap` :
`TreeSet` :
`Vector` :

A TERMINER

19. Les types

Sommaire

19.1 Le package java.lang 133
19.2 Classes et types primitifs 134
19.3 La classe java.lang.Boolean 134
19.4 La classe java.lang.Number 135
19.5 La classe java.lang.Integer 135
19.6 La classe java.lang.Byte 135
19.7 La classe java.lang.Short 136
19.8 La classe java.lang.Long 136
19.9 La classe java.lang.Float 137
19.10La classe java.lang.Double 137
19.11La classe java.lang.Character 138
19.12La classe java.lang.Math 138
19.13Jdk 1.2beta3 139

19.1 Le package java.lang

Le package `java.lang` définit un ensemble de classes et interfaces qui consitue le noyau du langage *Java*.

Java, comme tout langage de programmation orienté objets, offre un certain nombre de classes et interfaces utiles au programmeur. L'ensemble de ces classes et interfaces sont regroupés dans le package `java.util`.

La version de jdk1.1 fournit pour ce package :

Les interfaces :
`Cloneable` :
`Runnable` :

Les Classes :
Boolean
Byte
Character
Class
ClassLoader
Compiler
Double
Float
Integer
Long
Math
Number
Object
Process
Runtime
SecurityManager
Short
String
StringBuffer
System
Thread
ThreadGroup
Throwable
Void

19.2 Classes et types primitifs

Rappelons qu'il n'est pas possible de transformer les types primitifs en objets par une opération de changement de type (*cast*). Par contre, *Java* définit pour des classes spéciales pour un certain nombre de types primitifs (`Integer`, `Float`, `Boolean`, etc. Par exemple, on créera une instance de la classe `Integer` (qui n'est pas un `int`) ayant pour valeur 10 de la manière suivante :

```
Integer instanceInteger = new Integer(10);
```

Inversement la classe `Integer` dispose de méthode qui permettent d'obtenir la valeur du champ entier d'une instance de cette classe.

```
int i = instanceInteger.intValue();    // retourne 10
```

Les classes définies par *Java* pour les type primitifs sont les classes `Boolean`, `Character`, `Number`, `Integer`, `Long`, `Float` et `Double`.

19.3 La classe java.lang.Boolean

```
public final class Boolean {
    public static final Boolean TRUE = new Boolean(true)
    public static final Boolean FALSE = new Boolean(false)
    public Boolean(boolean value)
    public Boolean(String s)
    public String toString()
    public boolean equals(Object obj)
    public int  hashCode()
    public boolean booleanValue()
    public static Boolean  valueOf(String s)
    public static boolean getBoolean(String name) ;
}
```

`public Boolean (boolean value)`

Constructeur de la classe `Boolean` qui prend en argument l'un des deux valeurs : `true` ou `false`.

`public Boolean (String s)`

Constructeur de la classe `Boolean` qui prend en argument une chaîne de caractères. La chaîne de caractères `"true"` est transformée en instance de la classe `Boolean` contenant la valeur `true` et ce quelque soit la combinaison majuscule/minuscule. Tout autre chaîne de caractères sera considérée comme représentant `false`.

`public String toString ()`

Redéfinition de la méthode `toString` de la classe `Object`.

`public boolean equals (Object obj)`

Redéfinition de la méthode `equals` de la classe `Object`. Des instances de la classe `Boolean` sont égales si elles contient toutes deux la même valeur booléenne.

`public int hashCode ()`

Retourne le hash code de l'objet.

`public boolean booleanValue ()`

Retourne la valeur booléenne de l'objet.

`public static Boolean valueOf (String s)`

Retourne la valeur booléenne représentée par la chaîne passé en argument. Equivalent à `new Boolean(s).booleanValue()`.

`public static boolean getBoolean (String name)`

Retourne `true` si `name` est la chaîne `"true"` avec n'importe quelle combinaison majuscule/miniscule.

19.4 La classe java.lang.Number

La classe `Number` est une classe abstraite qui est déribée pour implaneter les classes `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`.

```
public final class Number {
   public Number()
   public abstrct int intValue()
   public abstrct short shortValue()
   public abstrct int intValue()
   public abstrct long longValue()
   public abstrct float floatValue()
   public abstrct double doubleValue()
}
```

19.5 La classe java.lang.Integer

Cette classe permet de représenter un `int` sous forme de classe.

```
public final class Integer extends Number implements Comparable {
   public static final int MIN_VALUE
   public static final int MAX_VALUE
   public static final Class TYPE
   public Integer(int value)
   public Integer(String s) throws NumberFormatException
   public static String toString(int i, int radix)
   public static String toHexString(int i)
   public static String toOctalString(int i)
   public static String toBinaryString(int i)
   public static String toString(int i)
   public static int parseInt(String s, int radix) throws NumberFormatException
   public static int parseInt(String s) throws NumberFormatException
   public static Integer valueOf(String s, int radix) throws NumberFormatException
   public static Integer valueOf(String s) throws NumberFormatException
   public byte byteValue()
   public short shortValue()
   public int intValue()
   public long longValue()
   public float floatValue()
   public double doubleValue()
   public String toString()
   public int hashCode()
   public boolean equals(Object obj)
   public static Integer getInteger(String nm)
   public static Integer getInteger(String nm, int val)
   public static Integer getInteger(String nm, Integer val)
   public static Integer decode(String nm) throws NumberFormatException
   public int compareTo(Integer anotherInteger)
   public int compareTo(Object o)
}
```

Les méthodes `parseInt` renvoie la valeur du nombre représenté de chaîne passée en argument. Une exception `NumberFormatException` est levée en cas d'erreur de syntaxe.

19.6 La classe java.lang.Byte

```
public final class Byte extends Number implements Comparable {
   public static final byte MIN_VALUE
   public static final byte MAX_VALUE
   public static final Class TYPE
```

```
   public Byte(byte value)
   public Byte(String s) throws NumberFormatException
   public static String toString(byte b)
   public static byte parseByte(String s) throws NumberFormatException
   public static byte parseByte(String s, int radix) throws NumberFormatException
   public static Byte valueOf(String s, int radix) throws NumberFormatException
   public static Byte valueOf(String s) throws NumberFormatException
   public static Byte decode(String nm) throws NumberFormatException
   public byte byteValue()
   public short shortValue()
   public int intValue()
   public long longValue()
   public float floatValue()
   public double doubleValue()
   public String toString()
   public int hashCode()
   public boolean equals(Object obj)
   public int compareTo(Byte anotherByte)
   public int compareTo(Object o)
}
```

19.7 La classe java.lang.Short

```
public final class Short extends Number implements Comparable {
   public static final short MIN_VALUE
   public static final short MAX_VALUE
   public static final Class TYPE
   public Short(short value)
   public Short(String s) throws NumberFormatException
   public static String toString(short s)
   public static short parseShort(String s) throws NumberFormatException
   public static short parseShort(String s, int radix) throws NumberFormatException
   public static Short valueOf(String s, int radix) throws NumberFormatException
   public static Short valueOf(String s) throws NumberFormatException
   public static Short decode(String nm) throws NumberFormatException
   public byte byteValue()
   public short shortValue()
   public int intValue()
   public long longValue()
   public float floatValue()
   public double doubleValue()
   public String toString()
   public int hashCode()
   public boolean equals(Object obj)
   public int compareTo(Short anotherShort)
   public int compareTo(Object o)
}
```

19.8 La classe java.lang.Long

```
public final class Long extends Number implements Comparable {
   public static final long MIN_VALUE
   public static final long MAX_VALUE
   public static final Class TYPE
   public Long(long value)
   public Long(String s) throws NumberFormatException
   public static String toString(long i, int radix)
   public static String toHexString(long i)
   public static String toOctalString(long i)
   public static String toBinaryString(long i)
   public static String toString(long i)
   public static long parseLong(String s, int radix) throws NumberFormatException
   public static long parseLong(String s) throws NumberFormatException
   public static Long valueOf(String s, int radix) throws NumberFormatException
   public static Long valueOf(String s) throws NumberFormatException
   public byte byteValue()
   public short shortValue()
   public int intValue()
   public long longValue()
```

```
    public float floatValue()
    public double doubleValue()
    public String toString()
    public int hashCode()
    public boolean equals(Object obj)
    public static Long getLong(String nm)
    public static Long getLong(String nm, long val)
    public static Long getLong(String nm, Long val)
    public int compareTo(Long anotherLong)
    public int compareTo(Object o)
}
```

19.9 La classe java.lang.Float

```
public final class Float extends Number implements Comparable {
    public static float MAX_VALUE
    public static float MIN_VALUE
    public static float NaN
    public static float NEGATIVE_INFINITY
    public static float POSITIVE_INFINITY
    public static Class TYPE
    public Float(float value)
    public Float(double value)
    public Float(String s)
    public byte byteValue()
    public int compareTo(Float anotherFloat)
    public int compareTo(Object o)
    public double doubleValue()
    public boolean equals(Object obj)
    public static int floatToIntBits(float value)
    public float floatValue()
    public int hashCode()
    public static float intBitsToFloat(int bits)
    public int intValue()
    public static boolean isInfinite(float v)
    public boolean isInfinite()
    public static boolean isNaN(float v)
    public boolean isNaN()
    public long longValue()
    public short shortValue()
    public static String toString(float f)
    public String toString()
    public static Float valueOf(String s)
}
```

19.10 La classe java.lang.Double

```
public final class Double extends Number implements Comparable {
    public static double MAX_VALUE
    public static double MIN_VALUE
    public static double NaN
    public static double NEGATIVE_INFINITY
    public static double POSITIVE_INFINITY
    public static Class TYPE
    public Double(double value)
    public Double(String s)
    public byte byteValue()
    public int compareTo(Double anotherFloat)
    public int compareTo(Object o)
    public static long doubleToLongBits(double value)
    public double doubleValue()
    public boolean equals(Object obj)
    public float floatValue()
    public int hashCode()

    public int intValue()
    public static boolean isInfinite(double v)
    public boolean isInfinite()
    public static boolean isNaN(double v)
```

```
    public boolean isNaN()
    public static double longBitsToDouble(long bits)
    public long longValue()
    public short shortValue()
    public static String toString(double f)
    public String toString()
    public static Float valueOf(String s)
}
```

19.11 La classe java.lang.Character

Cette classe représente un caractère sous la forme de classe.

```
public final class Number {
    public Character(char c)
    public int hashCode()
    public boolean equals(Object o)
    public String toString()
    public static boolean isLowerCase(char ch)
    public static boolean isTitleCase(char ch)
    public static boolean isDigit(char ch)
    public static boolean isDefined(char ch)
    public static boolean isLetter(char ch)
    public static boolean isLetterOrDigit(char ch)
    public static boolean isJavaIdentifierPart(char ch)
    public static boolean isJavaIdentifierStart(char ch)
    public static boolean isUnicodeIdentifierPart(char ch)
    public static boolean isUnicodeIdentifierStart(char ch)
    public static boolean isIdentifierIgnorable(char ch)
    public static char toLowerCase(char ch)
    public static char toUpperCase(char ch)
    public static char toTitleCase(char ch)
    public static int digit(char ch, int radix)
    public static int getNumericValue(char ch, int radix)
    public static boolean isSpaceChar(char ch)
    public static boolean isWhiteChar(char ch)
    public static boolean isISOControl(char ch)
    public static int getType(char ch)
    public static char forDigit(char ch)
}
```

19.12 La classe java.lang.Math

La classe `Math` constitue la bibliothèque mathématique de *Java*. Toutes les méthodes de cette classe sont publiques et statiques.

```
public final class Math {
    public static final double E
    public static final double PI
    public static double sin(double a)
    public static double cos(double a)
    public static double tan(double a)
    public static double asin(double a)
    public static double acos(double a)
    public static double atan(double a)
    public static double exp(double a)
    public static double log(double a)
    public static double sqrt(double a)
    public static double IEEEremainder(double f1, double f2)
    public static double ceil(double a)
    public static double floor(double a)
    public static double rint(double a)
    public static double atan2(double a, double b)
    public static double pow(double a, double b)
    public static int round(float a)
    public static long round(double a)
    public static double random()
    public static int abs(int a)
    public static long abs(long a)
    public static float abs(float a)
```

```
        public static double abs(double a)
        public static int max(int a, int b)
        public static long max(long a, long b)
        public static float max(float a, float b)
        public static double max(double a, double b)
        public static int min(int a, int b)
        public static long min(long a, long b)
        public static float min(float a, float b)
        public static double min(double a, double b)
    }
```

19.13 Jdk 1.2beta3

La version 1.2 de ce même package comporte :

Les interfaces :
`Cloneable` :
`Comparable` : :
`Runnable` :
`Runtime.MemoryAdvice` : :

Les Classes :
`Boolean`
`Byte`
`Character`
`Character.Subset`
`Class`
`ClassLoader`
`Compiler`
`Double`
`Float`
`Integer`
`Long`
`Math`
`Number`
`Object`
`Package`
`Process`
`Runtime`
`RuntimePermission`
`SecurityManager`
`Short`
`String`
`StringBuffer`
`System`
`Thread`
`ThreadGroup`
`ThreadLocal`
`Throwable`
`Void`

20. Programmation système

Sommaire

20.1 La classe java.lang.System 141
20.1.1 Les entrées sorties 142
20.1.2 System Property 142
20.1.3 La classe Properties 142
20.1.4 Récupération de mémoire 143
20.2 La classe java.lang.Runtime 143
20.3 La classe java.lang.Process 144
20.4 La classe java.lang.SecurityManager 144
20.4.1 Installer un gestionnaire de sécurité 145
20.4.2 Créer une politique de sécurité 145
20.5 La classe java.lang.ClassLoader 146
20.6 La classe java.lang.Compiler 146

20.1 La classe java.lang.System

La classe `System` définit un ensemble de champs et méthodes statiques ; ses contructeurs sont privés et il n'est donc pas possible de créer des instance de cette classe. Comme toutes les méthodes sont statiques, on pourra donc les utilisersans avoir à créer une instance de cette classe.

```
class System {
   public static final InputStream in
   public static final PrintStream out
   public static final PrintStream err

   public static void setIn(InputStream in)
   public static void setOut(PrintStream out)
   public static void setErr(PrintStream err)
   public static void setSecurityManager(SecurityManager s)
   public static SecurityManager getSecurityManager()
   public static native long currentTimeMillis()
   public static native void arraycopy(Object src, int src_position, Object dst, int dst_position, int length)
   public static native int identityHashCode(Object x)
   public static Properties getProperties()
   public static void setProperties(Properties props)
   public static String getProperty(String key)
   public static String getProperty(String key, String def)
   public static String getenv(String name)
   public static void exit(int status)
   public static void gc()
   public static void runFinalization()
   public static void runFinalizersOnExit(boolean value)
   public static void load(String filename)
   public static void loadLibrary(String libname)
}
```

20.1.1 Les entrées sorties

En particulier, cette classe définit les entrése sorties standard : `System.in`, `System.out` et `System.err`. Les méthodes `print`, `println` et `write` de classe `PrintStream` sont utilisées pour effectuer des sorties. Pour plus de précisions, se reporter au chapitre sur les entres sorties (voir **??**)

20.1.2 System Property

Cette classe permet également de gérer une suite de caractéristiques du système (*System Propreties* à l'image des variables d'environnement. Une proriété est constitué d'un doublet clé/valeur ; uu démarrage de la machine virtuelle *Java*, un certain nombre de caratéristiques sont disponibles :

`"file.separator"`	Sérateur de réperoires ("/" sur Unix ou "\" sur Windows)
`"java.class.path"`	Classpath de *Java*.
`"java.class.version"`	Version des classes *Java*.
`"java.home"`	Répertoire d'installation de *Java*.
`"java.vendor"`	Chaîne de caractères caractérisant le founisseur de *Java*.
`"java.vendor.url"`	URL du founisseur de *Java*.
`"java.version"`	Version de *Java*.
`"line.separator"`	Caractère de passage à la ligne suivante
`"os.arch"`	Architecture du système
`"os.name"`	Nom du système d'exploitation
`"os.version"`	Version du système
`"path.separator"`	Séprarteur de chemin (" :" par défaut)
`"user.dir"`	Répertoire courant
`"user.home"`	Répertoire racine de l'utilisateur
`"user.name"`	Nom du compte utilisateur

Les programmes *Java* peuvent

- consulter ces propriétés avec les méthodes `getProperty` et `getProperties`. Par exemple, on pourra connaître le nom du système d'exploitation utilisé avec

```
System.getProperty("os.name");
```

  Pour avoir la listes de toutes propriétés qui ont été définies, on utlisera la méthode `getPropreties`

```
Properties p = System.getProperties();
for (Enumeration enum = p.propertyNames(); enum.hasMoreElements(); ) {
    String cle = (String)enum.nextElement();
    System.out.println(cle + " = " + (String)(p.get(key)));
}
```

- ajouter de nouvelles propriétés ou modifier les valeurs des propriétés

```
Properties p = System.getProperties();
p.put("une.cle", "/tmp");
System.setProperties(p);
System.out.println("une.cle: " + System.getProperty("une.cle"));
```

20.1.3 La classe Properties

Une liste de propriétes est donc une liste de couple (clé/valeur) ; c'est une sorte de dictionnaire dont les clés sont des chaînes de caractères.

```
public class Properties extends Hashtable {
   public Properties()
   public Properties(Properties defaults)
   public synchronized void load(InputStream in) throws IOException
   public synchronized void save(OutputStream out, String header)
   public String getProperty(String key)
   public String getProperty(String key, String defaultValue)
   public Enumeration propertyNames()
   public void list(PrintStream out)
   public void list(PrintWriter out)
}
```

Les méthodes `getProprety` et `setProprety` recherchent et affectent la valeur à partir d'une clé. Les méthodes `save` et `load` sauvegardent ou rechargent une liste de propriétés à partir d'un flot. La méthode `PropretyNames` donne une enumation des clés dipsonibles et la méthode `list` affiche la liste des couples clé/valeur disponibles.

20.1.4 Récupération de mémoire

Autre tâche dédiée à la classe `System`, la gestion de la mémoire. A priori, la récupération de mémoire fonctionne constament en arrière plan et les objets *Java* qui ne sont plus référencés sont récupérés. Mais avant d'effectuer la récupération de mémoire, la machine virtuelle *Java* invoque la méthode `finalize` sur les objets devenus inutiles. Tant qu'il s'agit d'objets *Java*, il n'est pas utile d'implanter cette méthode ; elle n'est nécessaire que dans le cas où l'on utilise des ressources non *Java*.

La classe `System` fournit la méthode `unFinalization` afin de contraindre la machine virtuelle à invoquer les méthodes `finalise` sur les objets à récupérer.

Le moment où cette récupération a lieu est non prévisible. Dans certaines applications, il peut s'avérer utile d'effectuer une récupération de mémoire à la demande. Ainsi, la méthode `gc` de la classe `System` permet de lancer la récupération de mémoire à la demande du programmeur.

20.2 La classe java.lang.Runtime

```
public class Runtime {
   public static Runtime getRuntime()
   public void exit(int status)
   public static void runFinalizersOnExit(boolean value)
   public Process exec(String command) throws IOException
   public Process exec(String command, String envp[]) throws IOException
   public Process exec(String cmdarray[]) throws IOException
   public Process exec(String cmdarray[], String envp[]) throws IOException
   public native long freeMemory()
   public native long totalMemory()
   public native void gc()
   public native void runFinalization()
   public native void traceInstructions(boolean on)
   public native void traceMethodCalls(boolean on)
   public synchronized void load(String filename)
   public synchronized void loadLibrary(String libname)
   public InputStream getLocalizedInputStream(InputStream in)
   public OutputStream getLocalizedOutputStream(OutputStream out)
}
```

Un objet de type `Runtime` code l'état du système et les opérations qu'il est en droit d'effectuer. La méthode statique `Runtime.getRuntime()` permet de récupérer l'objet `Runtime` de l'état courant.

Bien des méthodes que l'on trouve dans la classe `System` se contente de faire appel aux méthodes de même nom de la classe `Runtime`. C'est la cas, par exemple, des méthodes `exit`, `gc`, `runFinalization`, `loadLibrary`, etc.

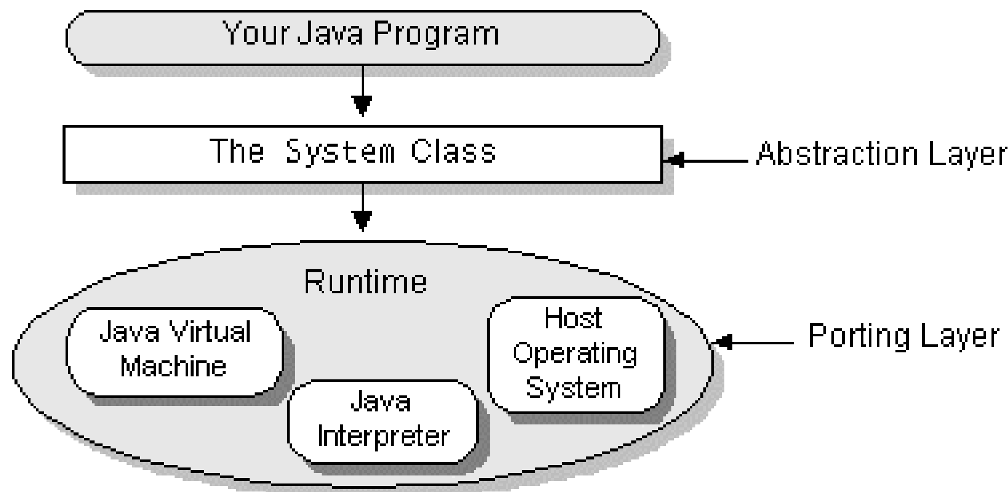

FIG. 20.1: Runtime

Un objet `Runtime` est intimmement lié au système utilisé ; ce qui n'est pas le cas de la classe `System`. Par exemple, on pourra, selon les systèmes, trouver des définitions différente de cette classe. La classe `System` est une abstraction de la classe `Runtime` et qui a la propriété d'être indépendant de l'architecture utilisée.

Cette classe définit les méthodes `exec` que nous verrons dans la section suivante.

20.3 La classe java.lang.Process

```
public abstract class Process {
   public Process()
   public abstract OutputStream getOutputStream()
   public abstract InputStream getInputStream()
   public abstract InputStream getErrorStream()
   public abstract int waitFor() throws InterruptedException
   public abstract int exitValue()
   public abstract void destroy()
}
```

La plupart des systèmes d'exploitation actuels sont capable d'exécuter plusieurs processus en meme temps. La classe **Process** de *Java* permet d'utiliser cette facilter pour lancer plusieurs exécution en "parallèle"[1]. On ne rentrera pas dans le détail de la gestions des process ici.

Les processus que l'on exécuter ne sont évidemment pas réduits aux applications *Java* ; toute application résidant sur le système hôte peut être lancé à partir de *Java*grâce à la méthode **exec** de la classe **Runtime**.

```
public Process exec(String[] cmdarray) throws IOException
public Process exec(String cmd) throws IOException
```

Ces méthodes renvoie un objet de type **Process** pour chaque process fils crée. Cet objet premet d'avoir accès aux flots d'entrées sorties du process crée (**getInputStream**, **getOutputStream** et **getErrorStream**).

```
class TestProc {
  public static void main(String args[]) throws Exception  {
    Process p = Runtime.getRuntime().exec("/bin/ls");
    String ligne;
    InputStreamReader in = new InputStreamReader(p.getInputStream());
    BufferedReader bin = new BufferedReader(in);
    while ((ligne = bin.readLine()) != null)
      System.out.println(ligne);
  }
}
```

On dispose également, dans cette classe, des méthodes pour contrôler l'exécution des process :

- La méthode **waitFor** attend la fin de l'exécution du process fils rt retourne sa valeur de sortie.
- La méthode **exitValue** renvoie la valeur de sortie du process
- La méthode **destroy** tue le process.

20.4 La classe java.lang.SecurityManager

Le gestionnaire de sécurité (*Security Manager*) est en charge des problèmes de sécurité au niveau de l'application. Clui-ci est consulté à chaque fois qu'une application accède aux ressources du système (fichier, ports réseaux, processus, etc.).

Le langage *Java* permet de définir, pour chaque application, sa propre politique de sécurité. Une application *Java* ne installer un gestionnaire de sécurité qu'une et une seule fois. Une fois le gestionnaire installé, toute utilisation es ressources systèmes sont filtrés par celui-ci. Pour les applications que ne possèdent pas de politique de sécurité particulière, il n'existe pas de politique par défaut. Autrement dit, sans gestionnaire de sécurité, une application *Java* a exactement les mêmes privilèges que n'importe quelle autre application et les seules protections sont celles mises en place le système utilisé.

La classe en charge de gestion de la sécurité est la classe **SecurityManager** du package **java.lang**. La classe **Système**, avec la méthode **getSecurityManager**, rend l'objet gestionnaire de sécurité. Cette méthode rend la valeur **null** si aucune politique de sécurité n'est définie.

La classe **SecurityManager** est une classe abstraite qui conteint les méthodes pour le contrôle de l'accès aux ressources du système. En implantant les méthodes de cette interface, chaque application définit les permissions d'utilisation de ses ressources.

```
public abstract class SecurityManager {
      protected SecurityManager()
      public boolean getInCheck()
      protected native Class[] getClassContext()
      protected native ClassLoader currentClassLoader()
      protected Class currentLoadedClass()
      protected native int classDepth(String name)
      protected native int classLoaderDepth()
      protected boolean inClass(String name)
      protected boolean inClassLoader()
      public Object getSecurityContext()
      public void checkCreateClassLoader()
      public void checkAccess(Thread g)
      public void checkAccess(ThreadGroup g)
```

[1]Nous verrons au chapiter refthreads que ce n'est pas la seule manière d'exécuter des activités en parallèle.

```
    public void checkExit(int status)
    public void checkExec(String cmd)
    public void checkLink(String lib)
    public void checkRead(FileDescriptor fd)
    public void checkRead(String file)
    public void checkRead(String file, Object context)
    public void checkWrite(FileDescriptor fd)
    public void checkWrite(String file)
    public void checkDelete(String file)
    public void checkConnect(String host, int port)
    public void checkConnect(String host, int port, Object context)
    public void checkListen(int port)
    public void checkAccept(String host, int port)
    public void checkMulticast(InetAddress maddr)
    public void checkMulticast(InetAddress maddr, byte ttl)
    public void checkPropertiesAccess()
    public void checkPropertyAccess(String key)
    public boolean checkTopLevelWindow(Object window)
    public void checkPrintJobAccess()
    public void checkAwtEventQueueAccess()
    public void checkPackageAccess(String pkg)
    public void checkPackageDefinition(String pkg)
    public void checkSetFactory()
    public void checkMemberAccess(Class clazz, int which)
    public ThreadGroup getThreadGroup()
  }
```

20.4.1 Installer un gestionnaire de sécurité

Pour installer un gestionnaire de sécurité, il suffit de

- créer une sous classe de la classe `SecurityManager`
- créer une instance de cette sous classe
- invoquer la méthode `setSecurityManager` de la classe `System`.

```
class BloquerTout extends SecurityManager { }

class Exemple {
  public static void main(String args[]) {
    System.setSecurityManager(new BloquerTout());
    ...
  }
}
```

Bien qu'abstraite, la classe `SecuirtyManager` implante toutes les méthodes : elles se contentent de lancer l'exception `SecurityException`. Ainsi, notre sous classe `BloquerTout` met en place une politique très restrictive : on peut accéder à aucune ressource du système.

Une fois ce gestionnaire installer, il n'est plus possible de le modifier en cours de route. Le gesionnaire de sécurité ne s'installe qu'une et une seule fois !!!

La méthoe `setSecurityManager` engendre l'exception `SecurityException` en cas de tentative d'installation d'un nouveau gestionnaire :

```
try { System.setSecurityManager(new BloquerTout()); }
catch (SecurityException e) {
    System.out.println("Vous n'avez pas le droit de changer de gestionnaire");
}
```

20.4.2 Créer une politique de sécurité

Comme nous vennons de la dire, pour créer une politique de sécurité, il tout d'abord créer une sous classe de la classe `SecurityManager` qui se charge de permettre l'accès aux différentes ressources du système.

Domaine	Méthode	Permission
Sockets	checkAccess	Modification des threads
	checkConnect	Ouverture d'un socket
	checkListen	Attente de connexion (socket)
	checkAccept	Acceptation d'un connexion (socket)
Réseau	checkMulticast	IP multicast
Interpréteur	checkExit	Quitter le programme avec `System.exit`
Système	checkExec	Lancer un sous processus
	checkPrintJobAccess	Accès à l'impression
	checkSystemClipboardAccess	Accès au presse papier
Fichiers	checkLink	Chargement dynamique de code natif
	checkRead	Accès en lecture d'un fichier
	checkWrite	Accès en écriture d'une fichier
	checkDelete	Supprimer un fichier
Propriétés	checkPropertiesAccess	Consultation et modification des propriétés
Graphique	checkTopLevelWindow	AUtoriastion d'ouvrir la fenetre principale
Evènements	checkAwtEventQueueAccess	Accès à la file d'évènements
Package	checkPackageAccess	Accès au package spécifié
	checkPackageDefinition	Créer des classe dans le package
...	...	...

Un gestionnaire qui choisirait de ne permettre que le lecture de fichier et ce restreint aux fichiers `.html` devrait implanter ceci :

```
class LectureHtml extends SecurityManager {
   public void checkRead(String s) {
      if (! s.endsWith(".html"))
         throw new SecurityException("Interdiction d'accéder aux fichiers non HTML");
   }
}
```

20.5 La classe java.lang.ClassLoader

L'environnement d'exécution de *Java* (*Runtime*) se charge du chargement des classes, de leur verification et de leur exécution à partir du code intermédiaire *Java*. Comme nous l'avons déjà dit (voir 9.1.3), la variable d'environnement `CLASSPATH` permet de préciser à la machine virtuelle *Java* où se trouvent les classes utilisées dans le système local de fichiers.

Parallèlement à cette manière de rechercher et charger les classes, *Java* permet au programmeur, grâce à la classe `java.lang.CLassLoader`, de définir sa propre stratégie de recherche et chargement des classes. C'est le cas du browser *HotJava*, écrit en *Java*, qui est capable de charger les applets à travers le réseau.

```
public abstract class ClassLoader {
   protected ClassLoader()
   public Class loadClass(String name) throws ClassNotFoundException
   protected abstract Class loadClass(String name, boolean resolve) throws ClassNotFoundException
   protected final Class defineClass(byte data[], int offset, int length)
   protected final Class defineClass(String name, byte data[], int offset, int length)
   protected final void resolveClass(Class c)
   protected final Class findSystemClass(String name) throws ClassNotFoundException
   protected final void setSigners(Class cl, Object signers[])
   protected final Class findLoadedClass(String name)
   public static final InputStream getSystemResourceAsStream(String name)
   public static final URL getSystemResource(String name)
   public InputStream getResourceAsStream(String name)
   public URL getResource(String name)
}
```

20.6 La classe java.lang.Compiler

```
public final class Compiler {
    public static native boolean compileClass(Class clazz)
    public static native boolean compileClasses(String string)
    public static native Object command(Object any)
    public static native void enable()
    public static native void disable()
}
```

A TERMINER

21. Programmation dynamique

Sommaire

21.1 Ausculter une classe . . . 147
21.2 Manipuler dynamiquement des objets . . . 150
21.2.1 Constructeurs . . . 150
21.2.2 Les champs . . . 151
21.2.3 Les méthodes . . . 152
21.2.4 Les modifiers . . . 153
21.2.5 Les membres . . . 153
21.2.6 Les tableaux . . . 153
21.3 Jdk 1.2 . . . 154

Java désigne par le terme de `Core Reflection` l'API en charge de l'introspection des classes et objets de la machine virtuelle *Java*.

Avec la classe `java.lang.Class`, il est possible de déterminer la classe à laquelle appartient un objet, obtenir des informations sur les méthodes, les champs, les constructeurs, les super classes etc. Il permet également de créer des objets alors que l'on ne connaît la classe à laquelle ils appartiennent qu'à l'exécution.

21.1 Ausculter une classe

Les instances de la classe `Class` représente les classes et interfaces existants dans une application *Java*. Chaque classe, interface, tableau d'un type particulier sont représentés par une instance de la classe `Class`. Il en est de même des types primitifs et du type `void`. Cette classe ne fournit pas de constructeur, les instances de cette classe sont créées par la machine *Java*.

```
public final class Class implements java.io.Serializable {
   private Class()
   public String toString()
   public static native Class forName(String className) throws ClassNotFoundException;
   public native Object newInstance() throws InstantiationException, IllegalAccessException;
   public native boolean isInstance(Object obj)
   public native boolean isAssignableFrom(Class cls)
   public native boolean isInterface()
   public native boolean isArray()
   public native boolean isPrimitive()
   public native String getName()
   public native ClassLoader getClassLoader()
   public native Class getSuperclass()
   public native Class[] getInterfaces()
   public native Class getComponentType()
   public native int getModifiers()
   public native Object[] getSigners()
   native void setSigners(Object[] signers)
   public Class getDeclaringClass()
   public Class[] getClasses()
   public Field[] getFields() throws SecurityException
   public Method[] getMethods() throws SecurityException
   public Constructor[] getConstructors() throws SecurityException
```

```
    public Field getField(String name) throws NoSuchFieldException, SecurityException
    public Method getMethod(String name, Class[] parameterTypes) throws NoSuchMethodException, SecurityException
    public Constructor getConstructor(Class[] parameterTypes) throws NoSuchMethodException, SecurityException
    public Class[] getDeclaredClasses() throws SecurityException
    public Field[] getDeclaredFields() throws SecurityException
    public Method[] getDeclaredMethods() throws SecurityException
    public Constructor[] getDeclaredConstructors() throws SecurityException
    public Field getDeclaredField(String name) throws NoSuchFieldException, SecurityException
    public Method getDeclaredMethod(String name, Class[] parameterTypes) throws NoSuchMethodException, SecurityException
    public Constructor getDeclaredConstructor(Class[] parameterTypes) throws NoSuchMethodException, SecurityException
    public InputStream getResourceAsStream(String name)
    public java.net.URL getResource(String name)
}
```

On ne peut qu'obtenir des instances de cette classe qu'à travers les différentes méthodes de cette classe, de la classe `Object`, de la classe `ClassLoader`, etc.

Lorsqu'un nom de la classe C est connu à la compilation, il est possible de créer l'instance de la classe `Class` qui représente C en suffixant le nom complet de la classe par "*.class*" ou en utilisant la méthode `forName` de la classe `Class`

```
Class c = java.util.Date.class;
Class c = Class.forName("java.util.Date");
```

Si le nom de la classe C n'est connu qu'à l'exécution, la méthode `getCalss` appliqué à un objet permet d'obtenir une instance de la classe `Class`.

```
Class c = obj.getClass();
```

Ainsi, avec une chaine de caractères, on peut parler de toutes les classes. Comment faire pour parler des tableaux, matrices, etc. Pour ce faire, *Java* définit représentation textuelle permet de parler d'un type de donnée sans devoir créer une instance de cet objet.

Un tableau d'éléments de type `T` est représenté par la chaîne "`[T`". Les types primitifs sont représentés par une chaîne de caractères d'un caractère (`B` pour `byte`, `C` pour `char`, `D` pour `double`, `F` pour `float`,Member `I` pour `int`, `J` pour `long`, `S` pour `short`, `Z` pour `boolean`). Une classe ou interface est représentée par la chaîne "`L`*`NomDeClasse ;`*".

Par exemple, un objet de la classe `Date` est représenté par la chaîne `"java.util.Date"` et une matrice d'objets de classe `Date` se représente par la chaîne "[[Ljava.util.Date ;".

`public String getName ()`

Retourne la représentation textuelle d'une classe.

```
(new Object[3]).getClass().getName()
(new Date())..getClass().getName()
```

retournent les chaînes

```
"[Ljava.lang.Object;"
"java.util.Date"
```

`public static native Class forName (String className)` *`throws ClassNotFoundException`*

Retourne un objet de type `Class` qui code la classe désigné par la chaîne `className`.

```
Class t = Class.forName("java.lang.Thread");
```

`public Object  newInstance ()` *`throws InstantiationException, IllegalAccessException`*

Créé une instance d'une classe

```
Class c = Class.forName("java.util.Date");
c.newInstance();   // Création d'une instance de java.util.Date
```

`public native int getModifiers ()` *`throws InstantiationException, IllegalAccessException`*

Retourne les modifiers de la classe ou de l'interface sou la forme d'un entier. Un modifier est un de ces mots clés qui précède la la définition d'une classe, d'une méthode ou d'un champ : *public*, `protected`, `private`, `final`, etc. Ces modifiers sont représentés par des entiers dont les valeurs sont définies dans la classe `Modifier` qui correspondent à un champ de bits.

```
Class c = Class.forName("java.util.Date");
int m = c.getModifiers();
if (Modifier.isPublic(m)) System.out.print("Public ");
...
if (Modifier.isabstract(m)) System.out.print("Abstract ");
...
```

`public native Class getSuperclass ()`

Retourne l'instance de la classe `Class` qui représente la super classe. Si l'instance sur laquelle cette méthode s'applique est un objet de classe `Object`, la valeur `null` est renvoyée.

```
Class c = Class.forName("java.util.Stack");
Class supclass = c.getSuperclass();
String supername = (supclass == null) ? "Aucun" : supclass.getName();
```

`public class[] getInterfaces ()`

Retourne un tableau d'instance de la classe `Class` qui représente toutes les interfaces qu'implante l'objet sur lequel cette méthode s'applique.

```
Class c = Class.forName("java.util.Stack");
Class ifs[] = s.getInterfaces();
for (int i = 0; i < ifs.length; i++)
   System.out.println("Interface[" + i + "] = " + ifs[i]);
```

`public Field[] getFields ()` *`throws SecurityException`*

Retourne un tableau d'instance de la classe `Field` qui représente tous les champs **publiques** de l'objet sur lequel cette méthode s'applique.

`public Field[] getDeclaredFields ()` *`throws SecurityException`*

Retourne que les champs (public, protected, private) définis dans cette classe uniquement ; pas les champs hérités des super classes.

```
Class c = Class.forName("java.util.Stack");
Field [] ch = c.getDeclaredFields();
for (int i = 0; i < ch.length; i++)
   System.out.println("Champs[" + i + "] = " + ch[i]);
```

`public Field[] getMethods ()` *`throws SecurityException`*

Retourne un tableau d'instance de la classe `Method` qui représente toutes les méthodes **publiques** de l'objet sur lequel cette méthode s'applique.

`public Field[] getDeclaredMethods ()` *`throws SecurityException`*

Retourne toutes les méthodes (public, protected, private) définies dans cette classe uniquement ; pas les méthodes héritées des super classes.

```
Class c = Class.forName("java.util.Stack");
Field [] mt = c.getDeclaredMethods();
for (int i = 0; i < mt.length; i++)
   System.out.println("Méthodes[" + i + "] = " + mt[i]);
```

`public Constructor[] getConstructors ()` *`throws SecurityException`*

Retourne un tableau d'instance de la classe `Constructor` qui représente tous les constructeurs **publiques** de la classe de l'objet sur lequel cette méthode s'applique.

`public Field[] getDeclaredConstructors ()` *`throws SecurityException`*

Retourne tous les constructeurs définis dans cette classe (public ; private, protected).

```
Class c = Class.forName("java.util.Stack");
Field [] mt = c.getDeclaredMethods();
for (int i = 0; i < mt.length; i++)
   System.out.println("Méthodes[" + i + "] = " + mt[i]);
```

```
import java.lang.Class;
import java.lang.Boolean;
import java.io.*;
import java.lang.reflect.*;

class InfoClass {
   public static void main (String args[]) throws Exception {
      Class s = Class.forName(args[0]);
      Class supclass = s.getSuperclass();
      String supername = (supclass == null) ? "Aucun" : supclass.getName();
      Class ifs[] = s.getInterfaces();
      ClassLoader loader = s.getClassLoader();

      Field [] ch = s.getDeclaredFields();
      Method [] m = s.getDeclaredMethods();
      imprimerModifier(s.getModifiers());
      System.out.println("Nom de la classe : " + s);
      System.out.println("Nom de la super classe : " + supername);
      System.out.println("Class Loader : " + loader);

      for (int i = 0; i < ifs.length; i++)
         System.out.println("Interface[" + i + "] = " + ifs[i]);
      for (int i = 0; i < ch.length; i++)
         System.out.println("Champs[" + i + "] = " + ch[i]);
      for (int i = 0; i < m.length; i++)
         System.out.println("Méthodes[" + i + "] = " + m[i]);

      if (s.isInterface())
```

```
            System.out.println(s+ " est une Interface");
         else
            System.out.println(s+ " n'est pas une Interface");
      }
      static void imprimerModifier(int c) {
         System.out.print("Modifier(s) : ");
         if ((c & Modifier.PUBLIC) != 0) System.out.print("PUBLIC ");
         if ((c & Modifier.PRIVATE) != 0) System.out.print("PRIVATE ");
         if ((c & Modifier.PROTECTED) != 0) System.out.print("PROTECTED ");
         if ((c & Modifier.STATIC) != 0) System.out.print("STATIC ");
         if ((c & Modifier.FINAL) != 0) System.out.print("FINAL ");
         if ((c & Modifier.SYNCHRONIZED) != 0) System.out.print("SYNCHRONIZED ");
         if ((c & Modifier.VOLATILE) != 0) System.out.print("VOLATILE ");
         if ((c & Modifier.TRANSIENT) != 0) System.out.print("TRANSIENT ");
         if ((c & Modifier.NATIVE) != 0) System.out.print("NATIVE ");
         if ((c & Modifier.INTERFACE) != 0) System.out.print("INTERFACE ");
         if ((c & Modifier.ABSTRACT) != 0) System.out.print("ABSTRACT ");
         System.out.println();
      }
   }
```

Voici les affichages produit par l'exécution suivante :

```
% java InfoClass java.util.Stack
Modifier(s) : PUBLIC
Nom de la classe : class java.util.Stack
Nom de la super classe : java.util.Vector
Class Loader : null
Champs[0] = private static final long java.util.Stack.serialVersionUID
Champs[0] = public java.lang.Object java.util.Stack.push(java.lang.Object)
Champs[1] = public synchronized java.lang.Object java.util.Stack.pop()
Champs[2] = public synchronized java.lang.Object java.util.Stack.peek()
Champs[3] = public boolean java.util.Stack.empty()
Champs[4] = public synchronized int java.util.Stack.search(java.lang.Object)
class java.util.Stack n'est pas une Interface
%
```

Dans cet exemple, nous nous sommes contenté d'afficher ce que les méthodes ont retournées comme valeurs. Nous n'avons pas du tout parler des différents types de données qui interviennent dans ce programme. Par exemple, la méthode **getMethods** retourne un tableau d'objet de la classe `Method`. Le package **java.lang.reflect** définit un ensemble de classes pour manipuler ce type d'objets.

La version de jdk1.1 fournit pour ce package :

Les Classes :
`Array` :
`Constructor` :
`Field` :
`Method` :
`Modifier` :

Les interfaces :
`Membre` :

21.2 Manipuler dynamiquement des objets

21.2.1 Constructeurs

Lorsque le type de l'objet à créer n'est connu qu'à l'exécution, il n'est pas possible d'utiliser les constructeurs de manière standard :

```
Vector v = new Vector();
```

Il faut alors passer par la méthode **newInstance** :

```
try {
   Class c = Class.forName("java.util.vector");
   v = c.newInstance();
}
catch (InstantiationException e) { System.out.println(e); }
catch (IllegalAccessException e) { System.out.println(e); }
catch (ClassNotFoundException e) { System.out.println(e); }
```

Il reste toutefois un point à éclaircir : comment créer dynamiquement un objet en utilisant un constructeur avec arguments :

```
Vector v = new Vector(100);
```

Remarquons l'argument que l'on passe dans ce constructeur est de type primitif (c'est un `int`). Or la méthode `newInstance` n'admet qu'un tableau d'objets ; il faut donc convertir notre `int` en un objet de type `Integer`.

```
try {
   Class c = Class.forName("java.util.Vector");
   Class [] argsClass = int.class;
   Object[] args =  new Integer(100) ;
   constr = c.getConstructor(argsClass);
   v = constr.newInstance(args);
}
catch (InstantiationException e) { System.out.println(e); }
catch (IllegalAccessException e) { System.out.println(e); }
catch (ClassNotFoundException e) { System.out.println(e); }
catch (InvocationTargetException e) { System.out.println(e); }
```

```
   public final class Constructor implements Member {
      public Class getDeclaringClass()
      public String getName()
      public native int getModifiers()
      public Class[] getParameterTypes()
      public Class[] getExceptionTypes()
      public boolean equals(Object obj)
      public int hashCode()
      public String toString()
      public native Object newInstance(Object initargs[]) throws InstantiationException, IllegalAccessException,
                                                                 IllegalArgumentException, InvocationTargetException
   }
```

A partir d'une instance de `Constructor`, les méthodes `getDeclaringClass`, `getName`, `getModifier` `getParameterTypes` et `getExceptionTypes` permettent de récupérer la classe à laquelle ce constructeur appartient, le nom du du constructeur, *les qualifiers* du constructeur et les types des arguments et les exceptions que ce constructeur peut engendrer.

21.2.2 Les champs

Il est, dans certaisn cas, intéressant de récupérer les insformations sur les champs d'une classe ; à cette fin , la classe `Field` propose un ensemble de méthodes dans la classe `Field`.

```
   public final class Field implements Member {
      public Class getDeclaringClass()
      public String getName()
      public Class getType()
      public boolean equals(Object obj)
      public int hashCode()
      public String toString()
      public native Object get(Object obj) throws IllegalArgumentException, IllegalAccessException
      public native boolean getBoolean(Object obj) throws IllegalArgumentException, IllegalAccessException
      public native byte getByte(Object obj) throws IllegalArgumentException, IllegalAccessException
      public native char getChar(Object obj) throws IllegalArgumentException, IllegalAccessException
      public native short getShort(Object obj) throws IllegalArgumentException, IllegalAccessException
      public native int getInt(Object obj) throws IllegalArgumentException, IllegalAccessException
      public native long getLong(Object obj) throws IllegalArgumentException, IllegalAccessException
      public native float getFloat(Object obj) throws IllegalArgumentException, IllegalAccessException
      public native double getDouble(Object obj) throws IllegalArgumentException, IllegalAccessException
      public native void set(Object obj, Object value) throws IllegalArgumentException, IllegalAccessException
      public native void setBoolean(Object obj, boolean z) throws IllegalArgumentException, IllegalAccessException
      public native void setByte(Object obj, byte b) throws IllegalArgumentException, IllegalAccessException
      public native void setChar(Object obj, char c) throws IllegalArgumentException, IllegalAccessException
      public native void setShort(Object obj, short s) throws IllegalArgumentException, IllegalAccessException
      public native void setInt(Object obj, int i) throws IllegalArgumentException, IllegalAccessException
      public native void setLong(Object obj, long l) throws IllegalArgumentException, IllegalAccessException
      public native void setFloat(Object obj, float f) throws IllegalArgumentException, IllegalAccessException
      public native void setDouble(Object obj, double d) throws IllegalArgumentException, IllegalAccessException
   }
```

Le méthode `get` générique ne revoie que des instances de classes, jamais des types primitifs. Avec cette méthodes, les types primitifs sont transformés en leur équivalent *class wrapper*. Les autres méthodes `get` sont là pour récupérer des type primitifs.

```
class X {
   int i;
   Integer I;
   int [] t = ...
}
```

```
class Y {
   ...
   public printFields(Object o) {
      ...
      try {
         Integer I;
         int i;
         Array t;
         Class c = o.getClass();
         Field fi = c.getField("i");
         I = c.getField("i").get(o);
         i = c.getField("i").getInt(o);
         I = c.getField("I").get(o);
         i =  c.getField("I").getInt(o);       // Erreur !!!
         t = c.getField("t").get(o);
       }
      catch (...) ...
      ...
    }
}
```

Les méthodes **set** et **set*type*** permettent affecter des champs.

A partir d'une instance de `Field`, les méthodes `getDeclaringClass`, `getName`, `getModifier` et `getType` permettent de récupérer la classe à laquelle ce champ appartient, le nom du champ, *les qualifiers* du champ et le type (sous la forme d'un objet de type `Class`) du champ.

```
try {
   Class c = Class.forName("...");
   Object o = c.newInstance();
   Field f = c.getField("...");
   Interger nouveau = new Integer(200);
   f.set(o, nouveau);
\
catch(...) { ...}
...
}
```

21.2.3 Les méthodes

Le sméthodes peuvent également se manipuler dynamiquement. Outre la manipulation des attributs d'une méthode, il est possible d'invoquer dynamiquement une méthode.

```
public final class Method implements Member {
   public Class getDeclaringClass()
   public String getName()
   public native int getModifiers()
   public Class getReturnType()
   public Class[] getParameterTypes()
   public Class[] getExceptionTypes()
   public boolean equals(Object obj)
   public int hashCode()
   public String toString()
   public native Object invoke(Object obj, Object args[])
                          throws  IllegalAccessException,
                                  IllegalArgumentException,
                                  InvocationTargetException
}
```

Voici un exmple d'invocation dynamique d'une méthode avec la méthode `invoque` de la classe `Method`.

```
try \{
   Class c = Class.forName("...");
   Object o = c.newInstance();
   Class[] argsClass =  {...};
   Method m = c.getMethod("Nom", argsClass);
   res = (type) m.invoke(...);
\}
catch (NoSuchMethodException e) \{ System.out.println(e); \}
catch (IllegalAccessException e) \{ System.out.println(e); \}
catch (InvocationTargetException e) \{ System.out.println(e); \}
```

A partir d'une instance de `Method`, les méthodes `getDeclaringClass`, `getName`, `getModifier` `getParameterTypes` `getReturnType` et `getExceptionTypes` permettent de récupérer la classe à laquelle cette méthode appartient, son nom, ses *qualifiers* et le type des arguments et de la valeur retournée ainsi que les exceptions que cette méthode peut engendrer.

21.2.4 Les modifiers

```
public final class Modifier {
   public static final int PUBLIC
   public static final int PRIVATE
   public static final int PROTECTED
   public static final int STATIC
   public static final int FINAL
   public static final int SYNCHRONIZED
   public static final int VOLATILE
   public static final int TRANSIENT
   public static final int NATIVE
   public static final int INTERFACE
   public static final int ABSTRACT
   public Modifier()
   public static boolean isPublic(int mod)
   public static boolean isPrivate(int mod)
   public static boolean isProtected(int mod)
   public static boolean isStatic(int mod)
   public static boolean isFinal(int mod)
   public static boolean isSynchronized(int mod)
   public static boolean isVolatile(int mod)
   public static boolean isTransient(int mod)
   public static boolean isNative(int mod)
   public static boolean isInterface(int mod)
   public static boolean isAbstract(int mod)
   public static String toString(int mod)
}
```

A TER-MINER

21.2.5 Les membres

```
public interface Member {
   public static final int PUBLIC
   public static final int DECLARED
   public abstract Class getDeclaringClass()
   public abstract String getName()
   public abstract int getModifiers()
}
```

A TER-MINER

21.2.6 Les tableaux

Comme nous l'avons déjà présenté plus haut (voir 21.1), les tableaux d'objets ont une représentation textuelle de la forme `"[[Ljava.util.Date ;"` pour désigner les matrices d'objets de type `java.util.Date`.

```
public final class Array {
   public static Object newInstance(Class componentType, int length)
                          throws NegativeArraySizeException
   public static Object newInstance(Class componentType, int dimensions[])
                          throws IllegalArgumentException, NegativeArraySizeException
   public static native int getLength(Object array) throws IllegalArgumentException
   public static native Object get(Object array, int index)
                          throws IllegalArgumentException, ArrayIndexOutOfBoundsException
   public static native boolean getBoolean(Object array, int index)
                          throws IllegalArgumentException, ArrayIndexOutOfBoundsException
   public static native byte getByte(Object array, int index)
                          throws IllegalArgumentException, ArrayIndexOutOfBoundsException
   public static native char getChar(Object array, int index)
                          throws IllegalArgumentException, ArrayIndexOutOfBoundsException
   public static native short getShort(Object array, int index)
                          throws IllegalArgumentException, ArrayIndexOutOfBoundsException
   public static native int getInt(Object array, int index)
                          throws IllegalArgumentException, ArrayIndexOutOfBoundsException
   public static native long getLong(Object array, int index)
                          throws IllegalArgumentException, ArrayIndexOutOfBoundsException
   public static native float getFloat(Object array, int index)
                          throws IllegalArgumentException, ArrayIndexOutOfBoundsException
   public static native double getDouble(Object array, int index)
                          throws IllegalArgumentException, ArrayIndexOutOfBoundsException
   public static native void set(Object array, int index, Object value)
                          throws IllegalArgumentException, ArrayIndexOutOfBoundsException
   public static native void setBoolean(Object array, int index, boolean z)
```

```
                                throws IllegalArgumentException, ArrayIndexOutOfBoundsException
    public static native void setByte(Object array, int index, byte b)
                                throws IllegalArgumentException, ArrayIndexOutOfBoundsException
    public static native void setChar(Object array, int index, char c)
                                throws IllegalArgumentException, ArrayIndexOutOfBoundsException
    public static native void setShort(Object array, int index, short s)
                                throws IllegalArgumentException, ArrayIndexOutOfBoundsException
    public static native void setInt(Object array, int index, int i)
                                throws IllegalArgumentException, ArrayIndexOutOfBoundsException
    public static native void setLong(Object array, int index, long l)
                                throws IllegalArgumentException, ArrayIndexOutOfBoundsException
    public static native void setFloat(Object array, int index, float f)
                                throws IllegalArgumentException, ArrayIndexOutOfBoundsException
    public static native void setDouble(Object array, int index, double d)
                                throws IllegalArgumentException, ArrayIndexOutOfBoundsException
}
```

La classe `Array` permet un certain nombre de manipulation sur les tableaux. Les méthodes `get` (resp. `get`*`Type`*) et `set` (resp. `set`*`Type`*) de récupérer et d'affecter un élément du tableau.

La création dynamique d'un tableau peut se faire avec la méthode `newInstance` de cette classe, la méthode `getLength` donne la taille du tableau.

A TERMINER

21.3 Jdk 1.2

La version de jdk1.2 fournit pour ce package :

Les Classes :
`AccessibleObject` :
`Array` :
`Constructor` :
`Field` :
`Method` :
`Modifier` :
`ReflectPermission` :

Les interfaces :
`Membre` :

A TERMINER

22. Threads

Sommaire

22.1 Introduction . . . 155
22.2 La classe thread et l'interface Runnable . . . 156
22.2.1 Le corps du code . . . 156
22.2.2 Créer et démarrer un thread . . . 157
22.2.3 Une mauvaise surprise . . . 157
22.2.4 Suspendre et redémarrer un thread . . . 158
22.2.5 Autre manière de lancer un thread . . . 159
22.3 Gestion des threads . . . 159
22.3.1 La synchronisation . . . 159
22.3.2 La communication entre threads . . . 161
22.3.3 La scission de l'assemblée . . . 163
22.3.4 Arrêter un thread . . . 164
22.3.5 Applets et threads . . . 164
22.3.6 Organisation et sécurité des threads . . . 166
22.4 Priorité des threads . . . 167
22.5 Interblocages . . . 168
22.6 Démons . . . 168
22.7 La classe Thread . . . 169
22.8 La classe ThreadGroup . . . 169

22.1 Introduction

Les programmes que nous avons l'habitue d'écrire sont conçus pour être exécutés de manière *séquentielle*. A partir d'un point de départ (par exemple, la méthode main pour une application autonome), tout ce que la machine exécute peut être suivi à la trace, pas à pas.

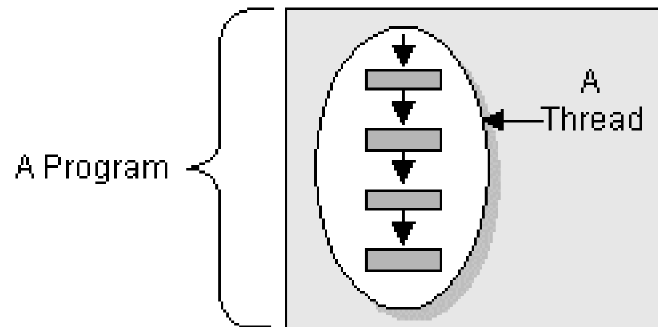

Fig. 22.1: Programme mono thread

Les programmes dont nous allons parler ici sont d'une tout autre nature : il s'agit d'applications dans lesquelles, il n'y a pas un unique "*fil d'exécution*" mais plusieurs fils d'exécutions qui se déroulent en parallèle, ou du moins *qui laisse à l'utilisateur l'impression qu'ils se déroulent en parallèle.*

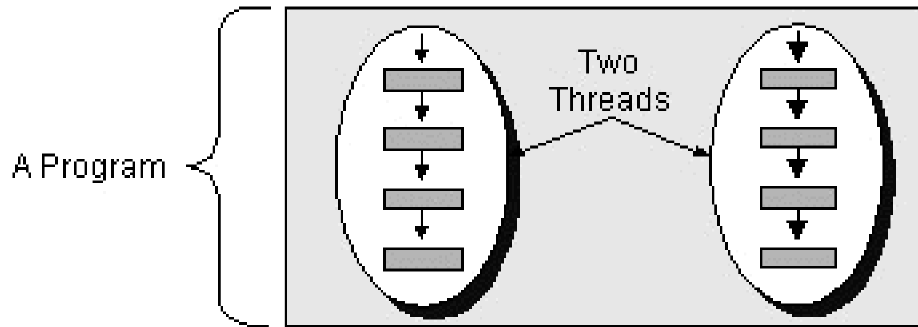

FIG. 22.2: Programme multi thread

Dans les langages habituels, la programmation des plusieurs fils d'exécution qu'on appelle *threads* est généralement un domaine réservé du spécialiste du système utilisé. Bien peu de programmeurs s'y risquent ! Avec *Java*, la programmation des *threads* et si simple que l'on peut l'évoquer très tôt dans la présentation du langage.

Ceux qui ont déjà vus des applets ont pu se rendre compte qu'ils utilisent généralement de multiples threads. Par exemple, pendant que l'applet réalise une tâche précise, on y voit également des animations graphiques, de la musique etc. ; et toutes ces tâches se déroulent en "parallèle".

Pour qui connaissent les processus (*UNIX* par exemple), on peut dire qu'un thread est un processus léger qui partagent le même espace de travail. Excepté les variables locales et les paramètres des méthodes qui sont dupliqués dans un espace propre à chaque thread, tout le reste est mis en commun pour tous les threads d'un même programme (mémoire, code et ressources partagés).

Comme les processus, les threads se trouvent, à tout instant, dans un état particulier. Ces divers états sont :

- Crée
- Actif
- Endormi
- Mort

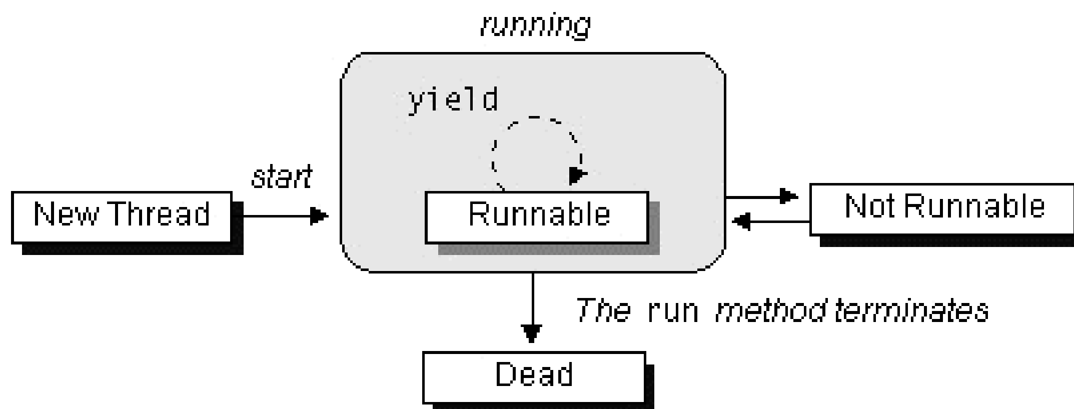

FIG. 22.3: Cycle de vide d'un thread

22.2 La classe thread et l'interface Runnable

22.2.1 Le corps du code

La première chose que l'on doit définir un filet d'exécution en parallèle, c'est évidemment les instructions que l'on veut faire exécuter dans ce filet. Java ne disposant pas de concept de pointeurs vers les fonctions, on effectue un petit détour pour arriver au résultat souhaité : on utilise une interface java.lang.Runnable qui définit une méthode unique run qui contient ces fameuses instructions que l'on veut faire exécuter au thread.

```
public abstract interface Runnable {
        public abstract void run();
}
```

La création du filet d'exécution en parallèle consiste donc à exécuter la méthode **run** sur un objet particulier ; tout ce qui doit être exécuté doit être contenu cette méthode. La méthode publique **run** n'a pas d'argument et ne retourne rien. Il ne peut non plus lancer des exceptions. N'importe quelle classe peut donc contenir la méthode **run** et déclarant qu'elle implante l'interface **Runnable**.

```
class TestThread implements Runnable {
    ...
    public void run() {
    ...
    }
}
```

22.2.2 Créer et démarrer un thread

Une fois définies les instructions que l'on veut exécuter, il faut arriver à créer ce fameux filet d'exécution. Pour ce faire, on doit créer, comme on l'a déjà dit, un objet de la classe **java.lang.thread**. Il existe plusieurs constructeurs pour cette classe : en particulier, il en existe une qui prend en argument un objet qui implante la classe **Runnable**. La création d'un thread n'implique pas le démarrage de l'exécution des instructions qui lui sont associées : il existe mais n'est pas encore actif.

```
TestThread  test = new TestThread ();
Thread monThread = new Thread(test);
```

Une fois le thread crée, il faut le rendre *actif* explicitement pour pouvoir démarrer l'exécution de ce thread ; ce qui se fait en invoquant la méthode **start** de la classe **Thread**.

```
monThread.start();
```

La méthode **start** est une méthode de la classe **Thread** qui alloue les diverses ressources nécessaires à l'exécution d'un thread et invoque la méthode **run** de l'objet passé en argument du constructeur de notre thread. Notre thread passe alors dans un état actif. Rendre actif un thread ne signifie pas qu'il s'exécute en continu jusqu'à la fin de son exécution : il ne fait que rejoindre les autres existants pour partager avec eux le temps d'exécution. Le système se charge de lui allouera régulièrement une tranche de temps pour qu'il puisse exécuter ces instructions. Ces tranches de temps alloués à tous les threads actifs sont assez fines pour que l'utilisateur ait l'impression que tous les threads s'exécutent en même temps.

22.2.3 Une mauvaise surprise

Nous en savons assez pour commencer un premier exemple simple où deux threads s'exécutent en parallèle. Pour se convaincre que ces deux threads s'exécutent en parallèle, nous allons faire en sorte que chaque thread affiche à l'écran une chaîne de caractères qui le caractérise (par exemple, ils afficherons respectivement les chaînes **TIC** et **TAC**).

```
class ThreadTest implements Runnable {
    java.lang.String s;
    ThreadTest(String s)    { this.s = s; }
    public void run() {
        while (true)  System.out.println(s);
    }
}

public class TicTac {
    public static void main(String argv[]) {
        ThreadTest tic = new ThreadTest("TIC");
        ThreadTest tac = new ThreadTest("TAC");
        Thread ThrTic = new Thread(tic);
        Thread ThrTac = new Thread(tac);
        ThrTic.start();
        ThrTac.start();
    }
}
```

Une version bien plus agréable de ce programme peut être écrite en remarquant qu'il faut systématiquement créer un thread et le démarrer à chaque création d'un objet **ThreadTest**. Il est alors tout à fait judicieux ceci fasse partie du constructeur de la classe **ThreadTest**.

```
class ThreadTest implements Runnable {
    java.lang.String s;
    Thread t ;
    ThreadTest(String s)    {
        this.s = s;
        t = new Thread(this) ; t.start() ;
    }
    public void run() {
        while (true)  System.out.println(s);
    }
}

public class TicTac {
    public static void main(String argv[]) {
        ThreadTest tic = new ThreadTest("TIC");
        ThreadTest tac = new ThreadTest("TAC");
    }
}
```

Voilà un programme qui devrait, à priori, afficher des `TIC` et des `TAC`. Si vous exécutez ce programme et selon le système d'exploitation que vous utilisez, vous aurez la désagréable surprise de ne voir que des `TIC` et jamais des `TAC` ! Est-ce à dire que la programmation avec les threads n'est pas portable ? Oui, un peu mais nous verrons plus loin pourquoi il en est ainsi et comment, malgré tout, arriver à écrire des programmes portables avec les threads. En attendant, nous allons modifier ce programme pour qu'il fasse ce que l'on attend de lui.

22.2.4 Suspendre et redémarrer un thread

Il existe des situations où il est utile de suspendre provisoirement un thread : c'est, par exemple, le cas des animations graphique que l'on voit sur les applets. Imaginons qu'une page *HTML* contienne une applet qui affiche une animation graphique. Celle-ci n'a pas besoin de s'exécuter lorsque la partie de la page *HTML* où apparaît l'applet n'est plus visible temporairement. Une solution simple pour ne pas effectuer les opérations inutiles d'affichages de l'animation consiste à détruire le thread qui gère cette animation et à la recréer une fois que cette partie de la page redevient visible. Ces opérations de destruction et de création des threads peuvent être coûteuses : une solution bien plus efficace consiste à suspendre l'animation le temps que l'animation sorte du champ de vision de l'utilisateur et à reprendre son exécution quand nécessaire. Pour ce faire, on dispose des méthodes `suspend` et `resume` de la classe `Thread`.

Les méthodes suspend et resume

L'utilisation de la méthode *suspend* pour endormir un thread doit se faire avec précaution : il faut s'assurer qu'on finira par invoquer la méthode `resume` ou `stop` pour rendre actif ou pour tuer ce thread.

La méthode sleep

Une autre manière de suspendre un thread consiste à l'endormir pendant une certain durée. Encore une fois, si l'on réalise une animation graphique, pour éviter que les différentes images qui constituent l'animation ne s'affichent de manière trop rapide, on utilisera la méthode `sleep` pour suspendre l'exécution pendant un laps de temps que l'on donnera en argument de cette méthode. Cet argument correspond au nombre de millisecondes durant lesquelles on veut suspendre l'exécution. La méthode `sleep` lance l'exception `InterruptedException` si le thread est stoppé pendant son sommeil. L'invocation de la méthode `sleep` doit donc gérer cette exception.

```
try {
    monThread.sleep(500) ;
}
catch (InterruptedException e) { ... }
```

Il existe, en fait, deux méthodes `sleep` dans la classe `Thread`. La première prend en argument un entier de type `long` donnant la durée en milli secondes du temps d'interruption. Quant à la deuxième, elle prend un argument supplémentaire (de type `int`) ; la durée de suspension est alors augmentée d'un nombre de nano secondes donné par ce dernier argument.

```
public static native void sleep(long ms) throws InterruptedException
public static void sleep(long ms, int ns) throws InterruptedException
```

Le sommeil réparateur

On va, à présent, modifier l'exemple précédant pour réussir à faire exécuter en parallèle les deux threads. Puisque le premier thread qui s'exécute ne fait aucun cas de celui en attente d'exécution, on va forcer chacun de ces threads à s'endormir régulièrement de manière à permettre à l'autre de s'exécuter un peu. La méthode `sleep` va nous permettre de réaliser cela :

```
class ThreadTest implements Runnable {
    java.lang.String s;
    Thread t ;
    ThreadTest(String s)    { this.s = s;  t = new Thread(this) ; t.start() ; }
    public void run() {
        while (true)  {
            System.out.println(s);
            try { t.sleep(100) ; }
            catch (InterruptedException e) {  }
        }
    }
}
```

Avec cette modification, on voit enfin apparaître les `TIC` et Les `TAC` entrelacés ! Preuve que ces deux threads s'exécutent bien en parallèle.

22.2.5 Autre manière de lancer un thread

L'interface **`Runnable`** permet à tout objet d'une classe qui l'implantant d'être la cible d'un thread. Une autre manière de lancer un thread est de définir une sous classe de la classe **`Thread`**. En fait, la classe **`Thread`** implante elle-même l'interface **`Runnable`**. La méthode qu'elle définit est une méthode **`run`** vide. En redéfinissant la méthode **`run`** dans la sous classe, il est possible de définir les actions que l'on veut faire exécuter à notre thread.

```
class ThreadTest extends Thread {
    java.lang.String s;
    ThreadTest(String s)    { this.s = s; }
    public void run() {
        while (true)  {
            System.out.println(s);
            try { this.sleep(100) ; }
            catch (InterruptedException e) {  }
        }
    }
}

public class TicTac {
    public static void main(String argv[]) {
        ThreadTest tic = new ThreadTest("TIC");
        ThreadTest tac = new ThreadTest("TAC");
        tic.start();
        tac.start();
    }
}
```

La méthode **`start`** de la classe **`Thread`** invoque immédiatement, après initialisation du thread, la méthode **`run`**. En redéfinissant cette dernière dans la classe dérivée, on obtient le même résultat qu'avec le programme donné en 22.2.4 ci-dessus.

22.3 Gestion des threads

22.3.1 La synchronisation

Comme nous l'avons déjà dit, les threads partagent quasiment toutes les données entre eux. L'utilisation des threads est bien plus simple et efficace que l'utilisation des processus. Le prix à payer pour cette simplicité et cette efficacité est que le programmeur est tenu d'être vigilant dans l'utilisation de ces objets partagés. C'est à lui qu'incombe la responsabilité de la *synchronisation* des différents threads.

Le mot est lâché : *synchronisation* ! Partager des données entre différents threads impliquent un minimum d'entente entre les threads pour garantir l'intégrité des données manipulées. Nous verrons plus loin, dans ce chapitre, pourquoi et comment on peut réaliser cette synchronisation. Imaginons que deux thread exécutent la même séquence d'instructions suivante :

```
int a ;
a = d.quelAnnée() ;
a = a + 1 ;
d.affecterAnnée(d) ;
```

La variable **`a`** est une variable locale et existe donc en deux exemplaires, un pour chaque thread. Si l'objet **`d`** n'est pas un objet locale, il est alors partagé par les deux threads. Chaque thread récupère dans la variable a la valeur du champ année de l'objet **`d`**. Puis, cette valeur est incrémentée de 1 et est rangée dans le champ année de l'objet **`d`**. Si ces threads s'exécutent en “même temps”, la valeur rangée dans le champ année, après exécution des deux threads, ne sera incrémentée que de 1, alors que l'on s'attend à ce qu'il soit incrémenté de 2 (incrémenté de 1, une fois par chaque thread).

Pourquoi en est-il ainsi ? Un thread peut être interrompu à tout moment pour permettre à un autre thread de pouvoir s'exécuter. Il est tout à fait possible que les threads exécutant la séquence d'instructions donnée plus haut soit interrompus entre l'instruction d'incrément et celle du stockage de la valeur dans le champ année.

```
THREAD 1                                    THREAD 1

a = d.quelAnnée() ;                         Suspendu
a = a + 1 ;                                 Suspendu
Suspendu                                    a = d.quelAnnée() ;
Suspendu                                    a = a + 1 ;
d.affecterAnnée(d) ;                        Suspendu
Suspendu                                    d.affecterAnnée(d) ;
...                                             ...
```

Les méthodes synchronized

La solution à ce problème consiste à interdire l'exécution de ce bout de code par plusieurs threads simultanément. En fait, Java ne permet pas de protéger n'importe quel partie du code contre les exécutions concurrentes. Il offre uniquement la possibilité de verrouiller un objet (pas une variable de type primitif) pour empêcher les accès concurrents. Lorsqu'une méthode qualifiée de `synchronized` est invoquée sur un objet par un thread, ce dernier est interdit pour toutes les méthodes qualifiées de `synchronized` s'appliquant sur le même objet par d'autres threads.

Imaginons que l'on dispose d'un unique mégaphone et de plusieurs orateurs qui veulent s'exprimer ; il va bien falloir coordonner la prise de parole de chaque orateur. Lorsqu'un orateur prend la parole, on attend qu'il ait fini pour passer la parole au suivant.

```
class MegaPhone {
    synchronized void parler(String qui, String dit, Thread t) {
        for (int i = 0 ; i<=10 ; i++ ) {
            System.out.println(qui + " affirme : "+ dit) ;
            try { t.sleep(200); }
            catch (InterruptedException e) {};
        }
    }
}

class Orateur extends Thread {
    String nom, discours ; MegaPhone mph;
    public Orateur(String n, String d, MegaPhone m) {
        nom = n; discours = d ; mph = m; start();
    }
    public void run() { mph.parler(nom, discours, this); }
}

class Assemblee {
    public static void main(String args[]) {
        MegaPhone mph = new MegaPhone() ;
        new Orateur("Orateur 1", "Je suis 1", mph) ;
        new Orateur("Orateur 2", "Je suis 2", mph) ;
        new Orateur("Orateur 3", "Je suis 3",  mph) ;
    }
}
```

Dans cet exemple, les trois orateurs utilisent un même mégaphone. Chaque orateur est constitué par un thread. La méthode parler est qualifiée de `synchronized`. Il en découle que l'objet de la classe `MegaPhone` est verrouillée dès lors qu'il est pris par un orateur même si celui-ci ne l'utilise pas à "plein temps" (`sleep`). Les autres orateurs ne pourront parler que lorsque celui qui détient le mégaphone a fini.

Synchronisation des classes

Une méthode `synchronized` pose un verrou sur l'objet sur lequel elle s'applique. Une méthode statique (méthodes de classes) peut elle être qualifiée de `synchronized` ? Eh bien, oui ! Les méthodes `synchronized` statiques pose un verrou sur la classe plutôt que sur un objet. Deux méthodes statiques et `synchronized` d'une même classe ne s'exécuter en même temps. En cela, ces méthodes sont proches des méthodes d'instance `synchronized`.

Par contre, il n'y aucun lien entre les *verrous de classe* et les *verrous des objets*. Une classe verrouillée (par une méthode `synchronized static`) n'empêche pas l'exécution d'une méthode d'instance `synchronized` et inversement. Les verrous de classe (resp. d'instance) ne synchronisent que les méthodes de classes (resp. d'instance)

Redéfinition des méthodes synchronized

A TERMINER

L'instruction synchronized

Avec le mot clé `synchronized`, nous avons pu verrouiller un objet sur tout une méthode. Il existe une instruction `synchronized` qui permet de verrouiller une partie d'une méthode. Cette instruction est constitué de deux parties : l'objet à verrouiller et le ou les instructions à exécuter.

```
synchronized (objet)
    instruction-simple-ou-bloc-d-instructions
```

En reprenant l'exemple de la modification de la date, on pourrait écrire le code suivant :

```
synchronized (d) {
    a = d.quelAnnée() ;
    a = a + 1 ;
    d.affecterAnnée(d) ;
}
```

Une méthode qualifiée de synchronized est équivalente à la méthode non synchronized et composée de l'instruction synchronized.

```
synchronized type méthode (...) {
    corps de la méthode
}

type méthode (...) {
    synchronized (this) {
        corps de la méthode
    }
}
```

22.3.2 La communication entre threads

Les méthodes et instructions `synchronized` permettent de contrôler l'interférence entre les threads. Nous allons à présent voir comment faire coopérer divers threads.

La méthode `wait` de la classe thread suspend l'exécution d'un thread en attendant qu'une certaine condition se réalise. La réalisation de cette condition est signalé par un autre thread par la méthode `notify` ou `notifyAll`. Toutes ces méthodes font partie de la classe `Object` et donc disponible par héritage à tous les classes.

La méthode yield

Une méthode peut décider de s'interrompre pour laisser la place à un autre thread de s'exécuter. La méthode `yield` est celle qui permet à un thread de ne pas être égoïste. Reprenons l'exemple ci-dessus (voir 22.2.4) : nous avons dit que, selon l'architecture du système utilisée, il était possible que les threads s'exécutent de manière séquentielle et non concurrente. Il a fallu ajouter des appels à la méthode `sleep` pour suspendre, pendant une durée fixe, le thread en cours d'exécution. Cette solution simple n'est évidemment pas très agréable puisqu'il faut décider d'une durée de sommeil arbitraire alors que l'on aurait voulu que les ressources soient partagées au mieux. Avec cette approche, les deux threads que l'on voulait exécuter pouvaient être en sommeil en même temps et les ressources CPU sont perdus pour tout le monde.

La méthode `yield` indique que le thread en cours d'exécution accepte de s'interrompre pour laisser la possibilité aux autres threads de s'exécuter. Le temps de sommeil sera laissé à l'appréciation du *scheduler*.

```
class ThreadTest implements Runnable {
    java.lang.String s;
    ThreadTest(String s)    { this.s = s; }
    public void run() {
        while (true)  {
            System.out.println(s);
            yield() ;
        }
    }
}
```

Avec cette modification, on devrait obtenir le même résultat qu'avec l'exemple de la section 22.2.4 avec toutefois une meilleur répartition du temps CPU.

La méthode join

Un thread peut décider d'attendre la fin d'un autre thread pour continuer son exécution. Les méthodes `join` de la classe `Thread` permettent cela. La méthode `join` lance l'exception `InterruptedException` si le thread est stoppé pendant son attente. L'invocation de la méthode join doit donc gérer cette exception.

```
try {
    UnThread.join(500) ;
}
catch (InterruptedException e) { ... }
```

Il existe, en fait, trois méthodes `join` dans la classe `Thread`. La première est sans argument et attend la fin de l'exécution du thread sur lequel cette méthode s'applique. La deuxième prend en argument un entier de type `long` donnant la durée en milli secondes du temps d'attente maximale. Si cet argument vaut 0, cette méthode est équivalent à la première forme. Quant à la troisième, elle prend un argument supplémentaire (de type `int`) ; la durée d'attente est alors augmentée d'un nombre de nano secondes donné par ce dernier argument.

```
public final synchronized void join(long millis) throws InterruptedException
public final synchronized void join(long millis, int nanos) throws InterruptedException
public final void join() throws InterruptedException
```

Illustrons ceci avec l'exemple de la section 22.2.4 ci-dessus et ajoutons un nouvel orateur qui doit clôturer la séance. Il faut qu'il attende que tous les orateurs se soient exprimés pour prononcer son allocution de fin de séance.

```
class Assemblee {
    public static void main(String args[]) {
        MegaPhone mph = new MegaPhone() ;
        Orateur o1 = new Orateur("Orateur 1", "Je suis 1", mph) ;
        Orateur o2 = new Orateur("Orateur 2", "Je suis 2", mph) ;
        Orateur o3 = new Orateur("Orateur 3", "Je suis 3",  mph) ;
        try { o1.join(); o2.join() ; o3.join();  }
        catch (InterruptedException e) {};
        new Orateur("Président", "Je suis le président et je clotûre la séance",  mph) ;
    }
}
```

La méthode wait

Lorsque la méthode **wait** est invoquée à partir d'une méthode **synchronized**, en même temps que l'exécution est suspendue, le verrou posé sur l'objet à partir de laquelle cette méthode à été invoquée est relâché. Dès que la condition de réveil survient, le thread attend de pouvoir reprendre le verrou et continuer l'exécution.

```
public final void wait() throws InterruptedException
```

Une autre version de **wait** prend en argument un entier de type **long**. Cet argument définit la durée d'attente maximale (en millisecondes)de la survenue de l'évènement. Lorsque ce temps d'attente est dépassée, même si la condition espérée ne survient pas.

```
public final native void wait(long ms) throws InterruptedException
```

Une dernière version de wait prend en argument un entier de type **long** et un entier de type **int**. Dans ce cas, la durée d'attente est de ms millisecondes plus ns nanosecondes.

```
public final void wait(long ms, int ns) throws InterruptedException
```

Une dernière version de wait prend en argument un entier de type **long** et un entier de type **int**. Dans ce cas, la durée d'attente est de ms millisecondes plus ns nanosecondes.

La méthode notify et notifyAll

La méthode **notify** réveille un et un seul thread qui est attente. Si plusieurs threads sont en attente, le thread réveillé est celui qui a été suspendu le plus longtemps. La notification n'est pas conditionnée par un évènement : les threads ne savent pas quelle condition à été satisfaite. C'est au code qui à invoqué la méthode **wait** de vérifier après coup s'il s'agit bien de la condition qu'il attendait. Tout ceci pour signaler que la mise en attente d'un thread doit tester la condition de réveil.

```
while ( ! condition) wait() ;
...                                                   // Code a exécuter quand la condition sera vraie
```

Lorsque plusieurs threads sont en attente et que l'on veut les réveiller tous, on utilisera la méthode **notifyAll**. Dans la plupart des cas, c'est plutôt cette méthode que l'on utilisera.

Les orateurs sympas

Notre assemblée est constitué d'orateurs "sympas" : au lieu d'accaparer le mégaphone jusqu'à la fin de leur discours, ils acceptent d'interrompre leur discours de temps en temps, de libérer le mégaphone et de se mettre en attente de la disponibilité du mégaphone.

```
class MegaPhone {
    boolean libre = true;
    synchronized void parler(String qui, String dit, Thread t) {
        for (int i = 0 ; i<=10 ; i++ ) {
            System.out.println(qui + " affirme : "+ dit) ;
            notifyAll() ;                   // Libère le mégaphone
            try { if (! libre) wait() ; }
            catch (InterruptedException e) {};// se met en attente du mégaphone
         }
    }
}
```

On remarquera que cette méthode est qualifiée de **synchronized** et le verrou posé est libéré lors de l'appel de **wait** pour permettre l'exécution des autres threads.

22.3.3 La scission de l'assemblée

Supposons à présent que, suite à des conflits internes, une scission de cette assemblée se produit : les protagonistes se retrouvent à présent avec deux mégaphones.

```
class LesMegaPhones {
    boolean estLibre[]={true, true};
```

Et malgré le conflit, les orateurs reste courtois et se partagent les mégaphones. La méthode disponible donne ne numéro du mégaphone libre. Si les deux mégaphones sont pris, elle retourne -1.

```
    synchronized int disponible() {
        for (int i=0; i<estLibre.length; i++)
            if (estLibre[i]) return i;
        return -1;
    }
```

Un orateur qui vent prendre possession d'un mégaphone, recherche un mégaphone libre. Si aucun n'est disponible, il se met en attente.

```
    synchronized int accaparer() {
        int i;
        while ((i =disponible()) == -1 ) {
            try { wait(); }
            catch (InterruptedException e) {};
        }
        estLibre[i]=false;
        return i;
    }
```

Lorsque l'orateur a fini de s'exprimer, il libère le mégaphone et le signale à tous les autres orateurs.

```
    synchronized void liberer(int i) {
        estLibre[i]=true;
        notifyAll();
    }
}
```

Comme dans l'exemple précédent, la classe **`Orateur`** est une classe dérivée de la classe **`Thread`**.

```
class Orateur extends Thread {
    String nom, discours ;
    LesMegaPhones m;
    public Orateur(String n, String d, LesMegaPhones m) {
        nom = n; discours = d ; this.m = m; start();
    }
    public void run() {
        parler();
    }
```

Pour parler, l'orateur prend un mégaphone (si possible), dit ce qu'il a dire, libère le mégaphone et répète ces opérations plusieurs fois.

```
    synchronized void parler() {
        int numMega;
        for (int j = 0; j < 5; j++) {
            numMega = m.accaparer();                // prend un mégaphone ou se met en attente
            try {sleep(1000); }
            catch (InterruptedException e) {};
            System.out.println(nom + " affirme avec le megaphone " + numMega + " " + discours) ;
            m.liberer(numMega);                     // libère le mégaphone
        }
    }
}
```

Et enfin, la classe principale :

```
class Assemblee {
    public static void main(String args[]) {
        LesMegaPhones mph = new LesMegaPhones();
        new Orateur("Orateur 1", "Je suis 1", mph) ;
        new Orateur("Orateur 2", "Je suis 2", mph) ;
        new Orateur("Orateur 3", "Je suis 3", mph) ;
        new Orateur("Orateur 4", "Je suis 4", mph) ;
        new Orateur("Orateur 5", "Je suis 5", mph) ;
    }
}
```

22.3.4 Arrêter un thread

Lorsqu'une application *Java*démarre, un premier thread s'exécute : il s'agit du thread principal, celui qui démarre l'exécution de la méthode **main** dans le cas des applications autonomes. Lorsque la méthode **main** se termine et si aucun autre thread n'a été crée, l'application s'arrête. Par contre, si des threads ont été crées, même si la méthode **main** arrive à la fin de son exécution, l'application ne s'arrêtera que lorsque tous les autres threads arriveront en fin d'exécution[1].

La fin de l'exécution d'un thread survient lorsque toutes les instructions de la méthode **run** correspondante ont été exécutées. Que se passe-t-il si la méthode **run** est conçue pour fonctionner indéfiniment, comme c'est le cas, par exemple, pour les animations graphiques ? Notre thread continue à s'exécuter indéfiniment : et ce même si le code l'ayant lancer est terminé. Un programme *Java*ne se termine que lorsque tous les threads sont terminés. Un thread qui s'exécute indéfiniment force donc l'interpréteur *Java*à ne pas s'arrêter. Les threads orphelins continuent à mobiliser des ressources système et l'interpréteur.

Comment donc arrêter les threads en cours (actifs ou endormis) ? Il y plusieurs méthodes :

- Une solution simple consiste à utiliser un variable booléenne que l'on positionnera à vrai lorsque l'application désire s'arrêter et le corps de la méthode **run** sera modifié pour tester cette valeur avant de poursuivre l'exécution.

```
void run() {
    while ( ! stoperLesThreads) { ... }
}
```

- Une autre manière, plus directe, consiste à invoquer la méthode **stop** de la classe **Thread**. Cette méthode, tout comme la méthode **run**, n'a ni argument et ne retourne rien. Cette méthode lance une erreur **ThreadDeath** qui un objet de dérivé de la classe **Error**.
- Une autre forme de la méthode **stop** prend en argument une exception autre que **ThreadDeath**. Le code de la méthode **run** sera alors conçu de telle sorte qu'il capture cette exception et donc termine la méthode run. A titre d'exemple, modifions le fonctionnement de notre assemblée (voir 22.3.1) avec un président bien pressé qui veut mettre fin rapidement à la réunion que les orateurs ait fini ou pas.

```
class MegaPhone {
  synchronized void parler(String qui, String dit, Thread t) {
    for (int i = 0 ; i<=1 ; i++ ) {
      System.out.println(qui + " affirme : "+ dit) ;
      try { t.sleep(200); }
      catch (InterruptedException e) {};
    }
  }
}

class Orateur extends Thread {
  String nom, discours ; MegaPhone mph;
  public Orateur(String n, String d, MegaPhone m) {
    nom = n; discours = d ; mph = m; start();
  }
  public void run() {
    for (int i = 0 ; i<=1 ; i++ ) {
      mph.parler(nom, discours, this);
      yield();
    }
  }
}

class AssembleeJoin {
  public static void main(String args[]) {
    MegaPhone mph = new MegaPhone() ;
    Orateur o1 = new Orateur("Orateur 1", "Je suis 1", mph) ;
    Orateur o2 = new Orateur("Orateur 2", "Je suis 2", mph) ;
    Orateur o3 = new Orateur("Orateur 3", "Je suis 3",  mph) ;
    try { o1.join(200); o2.join(200); o3.join(200);   }
    catch (InterruptedException e) {};
    System.out.println("Je suis le président et je suis pressé") ;
    o1.stop();      o2.stop();      o3.stop();
    new Orateur("Président", "Je suis le président et je clotûre la séance",  mph) ;
  }
}
```

22.3.5 Applets et threads

[1]Ceci n'est pas complètement exact : il y a le cas particulier des thread démons (daemons, en anglais) que nous verrons plus loin (voir 22.6).

Comme nous l'avons déjà vu, les applets sont des applications qui sont capables de démarrer et de s'arrêter en fonction de l'action de l'utilisateur. Les méthodes **init**, **start** et **stop** permettent respectivement d'initialiser, de démarrer et d'arrêter. Contrairement aux threads, les applets sont susceptibles d'être démarrés et arrêtés autant de fois que nécessaire : le concept de suspension n'existe pas pour les applets. On peut calquer le comportement des threads que l'on veut gérer dans une applet avec celui de l'applet elle-même :

- Créer et démarrer un thread lors du démarrage de l'applet
- Arrêter le thread lorsque l'applet s'arrête.

Un premier exemple d'applet avec un thread

A titre d'exemple, créons une applet qui affiche toutes les secondes l'heure. Il s'agit tout d'abord de définir une classe dérivée de classe **java.applet.Applet** et qui implente l'interface **Runnable**. Cette classe possède pour champ privé un objet de la classe **Thread**.

```
public class clockApp extends java.applet.Applet implements Runnable {
    private Thread t;
    ...
}
```

Cette applet crée un nouveau thread à chaque fois que l'applet démarre et le détruit lorsque celle-ci s'arrête. On invoque la méthode **stop** sur ce thread et on affecte à **null** la variable qui référence ce thread pour que *Java*récupère la mémoire utilisée pour celui-ci.

```
public void start() { t = new Thread(this); t.start(); }

public void stop() {  t.stop(); t = null; }
```

Même si cela ne justifie pas ici, une bonne habitude de programmation lors de la programmation des threads consiste à tester les valeurs des threads avant leur utilisation : pour éviter de créer par inadvertance plusieurs threads inutilement ou tenter de détruire des threads inexistants :

```
public void start() {
    if (t == null) { t = new Thread(this); t.start(); }
}

public void stop() {
    if (t != null) { t.stop(); t = null; }
}
```

Le filet d'exécution confié au thread associé à cette applet consiste tout simplement à s'endormir pendant une seconde puis à invoquer la méthode **repaint** pour rafraîchir l'affichage de l'applet.

```
public void run() {
    while (true) {
        try { t.sleep(1000); }
        catch (InterruptedException e) {}
        repaint();
    }
}
```

La méthode repaint de l'applet se contente d'afficher la date donnée par le système.

```
    public void paint(java.awt.Graphics g) {
        g.drawString(new java.util.Date().toString(), 10, 20);
    }
}
```

Une fois compilé et créé le fichier HTML qui fait appel à cette classe clockApp, on devrait voir s'égrener les secondes sur notre page HTML.

```
<HTML>
    <HEAD>
        <TITLE>Et le temps passe !!!</TITLE>
    </HEAD>
    <BODY>
        ...
        <APPLET CODE="clockApp.class" WIDTH=250 HEIGHT=25> </APPLET>
        ...
    </BODY>
</HTML>
```

Améliorons cet exemple

Dans cet exemple, le thread est systématiquement détruit chaque fois que l'applet invoque la méthode `stop` ce qui implique que l'on est obligé de récréer un nouveau thread avec la méthode `start`. En imaginant que l'initialisation de notre thread implique des opérations coûteuses, il serait bien plus judicieux de suspendre le thread. Le problème qui se pose alors est de déterminer le moment où l'on détruira effectivement ce thread. Le comportement d'utilisateur devant sa page *HTML* n'étant pas prévisible, on ne pourra jamais être assuré de pouvoir détruire le thread. A moins que l'on remarque que la méthode `destroy` de la classe `Applet` est invoquée lorsque l'applet doit être détruite entièrement (généralement lors de sa disparition du cache). Cette méthode permet donc de libérer toutes les ressources mobilisées par une applet.

Grâce à cette méthode `destroy`, nous allons pouvoir suspendre et reprendre l'exécution du thread lorsque l'applet s'arrête et redémarre ; et on laissera le soin à la méthode `destroy` de tuer le thread.

```
public void start() {
    if (t == null)
       { t = new Thread(this); t.start(); }
    else
        t.resume() ;
}

public void stop() {
   if (t != null) t.suspend();
}
public void destroy() {  t.stop(); t = null; }
```

22.3.6 Organisation et sécurité des threads

Chaque thread appartient à un groupe de threads. Par défaut, les threads que l'on crée (sauf contre indication) font partie du même groupe que le thread qui l'a créé. Au démarrage d'une application *Java*, il existe un premier groupe de thread `main` qui contient le thread principal et le thread qui se charge de la récupération de mémoire.

Pour créer un nouveau thread appartenant un nouveau groupe de threads, on crée tout d'abord le groupe de thread qui est un objet de la classe `ThreadGroup` et ensuite on crée le thread en utilisant une des constructeurs qui prend en argument un objet de la classe `ThreadGroup`.

```
ThreadGroup monGroupe = new ThreadGroup("Mon groupe de threads") ;
Thread t = new Thread(monGroupe) ;
```

Les goupes de threads sont constitués de threads et d'autres groupes de threads. Du groupe principal appelé `main`, on peut alors créer une arborescence complète de threads et de groupes de threads. L'orgranisation des threads en groupes pour des raisons

– facilité de gestion des threads

– de sécurité et contrôler les actions qu'un thread peut réaliser sur un autre thread.

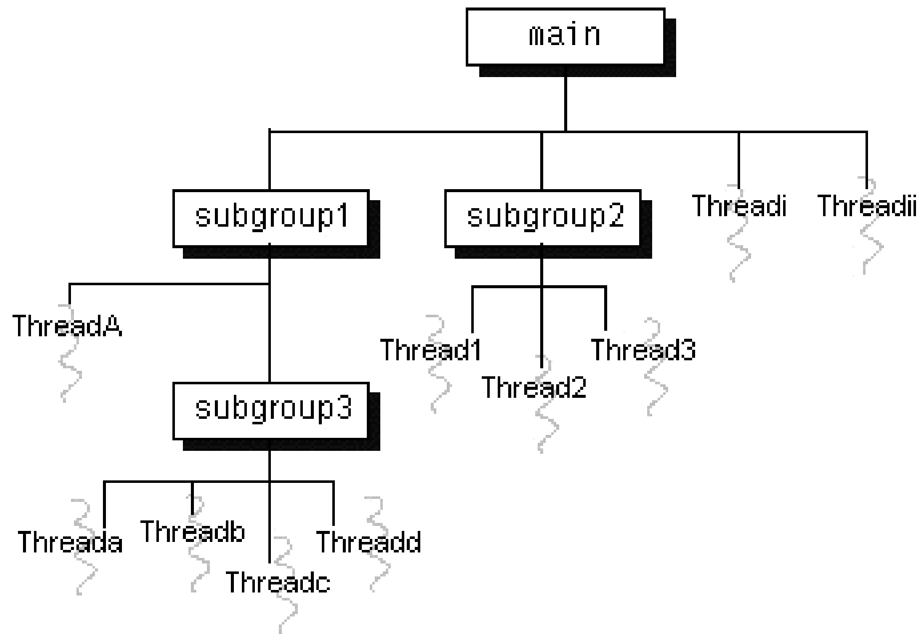

FIG. 22.4: Organisation des threads

La classe **ThreadGroup** fournit un ensemble de méthodes pour gérer les threads et les sous groupes de threads ainsi que pour obtenir les caractéritiques des groupes de threads. Par exemple, il existe une méthode **activeCount** qui donne le nombre de threads actifs dans un groupe. Il existe également une méthode **enumerate** qui donne la liste des threads actifs dans un groupe. Voici un exemple de programme qui remplit un tableau de threads actifs dans le groupe de thread courant.

```
class ListeDeThreads {
    void listeDesThreadsCourants() {
        ThreadGroup groupeCourant = Thread.currentThread().getThreadGroup();
        int nbreThreads;
        Thread[] listeDeThreads;

        nbreThreads = groupCourant.activeCount();
        listeDeThreads = new Thread[nbreThreads];
        GroupCourant.enumerate(listeDeThreads);
        for (int i = 0; i < numThreads; i++) {
            System.out.println("Thread numéro " + i + " = " + listeDeThreads[i].getName());
        }
    }
}
```

A FINIR

22.4 Priorité des threads

Comme nous l'avons déjà dit, les threads ne s'exécutent pas forcément en parallèle. En pratique, en particulier sur les machines mono processeur, chaque thread s'exécutent tout seul. C'est la répartition du temps CPU à chaque thread qui nous donne l'illusion qu'ils s'exécutent tous en même temps. Cette technique de distribution des ressources aux threads (ou processus) s'appelle l'ordonnancement (scheduling). Le langage Java dispose d'un algorithme déterministe simple basé sur la notion de priorité des threads.

A chaque thread est associé une priorité. Une priorité est un entier compris entre les valeurs `MIN_PRIORITY` and `MAX_PRIORITY` définies dans la classe **Thread**. Lorsque plusieurs threads sont démarrés, celui qui a la priorité la plus forte s'exécute en premier. C'est uniquement lorsque les threads de priorité le plus élevé ont fini leur exécution ou qu'ils sont suspendus pour une raison quelconque que les autres threads sont exécutés. S'il existe deux threads de même priorité en attente d'exécution, java choisit un des threads et l'exécute jusqu'à ce que l'un des évènements suivants se produisent :

- Un thread de priorité plus élevé est en attente d'exécution
- Le thread en cous d'exécution se termine
- Le thread en cous d'exécution demande de partager le temps d'exécution

– Sur les systèmes supportant le time-slicing, le temps imparti à ce thread touche à sa fin.

Lorsqu'on crée un nouveau thread, il hérite de la priorité du thread qui l'a crée. Cette priorité peut être modifié à tout moment grâce à la méthode **setPriority**.

L'algorithme d'ordonnancement de *Java*est *préemptif* : a tout moment, *Java*peut interrompre un thread en cous d'exécution pour donner la main à un thread de priorité plus forte.

Un président insupportable

```
class Président extends Orateur {
    private Orateurs [] listeOrateurs ;
    Président(String n, String d, MégaPhone m, Orateurs [] o) {
       super(n, d, m) ;
       listeOrateurs = o ;
       setPriority(MAX_PRIORITY) ;
    }

    public void run() {
       for (int i = 0 ; i < o.length ; i++) o[i].join(100) ;
           for (int i = 0 ; i<= 3 ; i++) {
               mph.parler(nom, discours, this);
               try { t.sleep(20); }
               catch (InterruptedException e) {}
           }
       for (int i = 0 ; i < o.length ; i++) o[i].stop(100) ;
    }
}
class Assemblée {
    public static void main(String args[]) {
        MegaPhone mph = new MegaPhone() ;
        Orateur [] lesOrateur =  { new Orateur("Orateur 1", "Je suis 1", mph),
                                   new Orateur("Orateur 2", "Je suis 2", mph),
                                   new Orateur("Orateur 3", "Je suis 3",  mph)
                                 } ;
        new Président("Président", "Je suis le président et je clotûre la séance",  mph, lesOrateurs) ;
    }
}
```

Time-slicing

Certains systèmes comme *95/NT*, implante la technique du *time-slicing* pour lutter contre les threads égoïstes qui accaparent les ressources CPU. Le *time-slicing* n'a d'intérêt que lorsque plusieurs threads de même priorité sont en activité. C'était le cas de notre exemple de la section 22.2.4 des threads **TIC** et **TAC**. Dans les systèmes ne possédant pas le *time-slicing*, cet exemple produit une succession de **TIC** et comme il s'agit d'une boucle infini, on aura jamais aucun **TAC** à l'écran. En effet, dans de tels systèmes, une fois un thread choisi pour être exécuté, celui conserve cette ressource jusqu'à ce que

– qu'il se termine
– qu'il décide volontairement de partager cette ressources (**sleep()**, **yield()**)
– qu'un thread de priorité supérieure réclame cette ressource.

Dans les systèmes supportant le *time-slicing*, cet exemple produit une succession de **TIC TAC** car le *time-slicing* s'arrange pour faire partager le temps CPU entre les threads de même priorité. Rappelons encore que s'il existe des threads de priorité supérieure à ces derniers, le système les ignorera tant qu'il existe des threads de priorité supérieure.

Si l'on veut réaliser des applications portable, il ne faut pas utiliser les possibilités du *time-slicing*.

22.5 Interblocages

A TERMINER

22.6 Démons

Nous avons dit plus haut qu'une application s'arrêtait que lorsque tous les threads avaient terminé leur exécution. C'est n'est vrai qu'en partie. En effet, il existe deux sortes de threads : les threads *utilisateurs* et les threads *démons* (de l'anglais *daemon*).

Les threads *daemon* sont conçus pour exécuter une tache en arrière plan aussi longtemps que le programme tourne. Autrement, ils s'arrêtent tout seul lorsque tous les threads utilisateurs arrivent en fin d'exécution.

Généralement les threads *daemons* fournissent un certain service pour un autre thread qui lui est un thread utilisateur. Il est par conséquent raisonnable les threads *daemons* s'arrêtent dès qu'il n'y a plus personne à qui offrir une service. C'est le cas, par exemple, du thread qui s'occupent de charger les images dans le browser *HotJava*.

Rien, dans la programmation, ne distingue les threads *daemons* des autres. La méthode **setDaemon(true)** transforme un *utilisateur* en thread *daemon*.

22.7 La classe Thread

```
public class Thread
   public static final int MIN_PRIORITY
   public static final int NORM_PRIORITY
   public static final int MAX_PRIORITY
   public Thread()
   public Thread(Runnable target)
   public Thread(ThreadGroup group, Runnable target)
   public Thread(String name)
   public Thread(ThreadGroup group, String name)
   public Thread(Runnable target, String name)
   public static native Thread currentThread()
   public static native void yield()
   public static native void sleep(long millis) throws InterruptedException
   public static void sleep(long millis, int nanos) throws InterruptedException
   public native synchronized void start()
   public void run()
   public final void stop()
   public final synchronized void stop(Throwable o)
   public void interrupt()
   public static boolean interrupted()
   public boolean isInterrupted()
   public void destroy()
   public final native boolean isAlive()
   public final void suspend()
   public final void resume()
   public final void setPriority(int newPriority)
   public final int getPriority()
   public final void setName(String name)
   public final String getName()
   public final ThreadGroup getThreadGroup()
   public static int activeCount()
   public static int enumerate(Thread tarray[])
   public native int countStackFrames()
   public final synchronized void join(long millis) throws InterruptedException
   public final synchronized void join(long millis, int nanos) throws InterruptedException
   public final void join() throws InterruptedException
   public static void dumpStack()
   public final void setDaemon(boolean on)
   public final boolean isDaemon()
   public void checkAccess()
   public String toString()
```

22.8 La classe ThreadGroup

```
public class ThreadGroup
   public ThreadGroup(String name)
   public ThreadGroup(ThreadGroup parent, String name)
   public final String getName()
   public final ThreadGroup getParent()
   public final int getMaxPriority()
   public final boolean isDaemon()
   public synchronized boolean isDestroyed()
   public final void setDaemon(boolean daemon)
   public final void setMaxPriority(int pri)
   public final boolean parentOf(ThreadGroup g)
   public final void checkAccess()
   public int activeCount()
   public int enumerate(Thread list[])
   public int enumerate(Thread list[], boolean recurse)
   public int enumerate(ThreadGroup list[])
   public int enumerate(ThreadGroup list[], boolean recurse)
   public final void stop()
   public final void suspend()
   public final void resume()
   public final void destroy()
   public void list()
   public void uncaughtException(Thread t, Throwable e)
   public boolean allowThreadSuspension(boolean b)
```

```
public String toString()
```

Deuxième partie

Java : Programmation graphique

Table des Matières

23 Applets et applications autonomes . . . 175
23.1 Application autonome . . . 175
23.2 Applets . . . 177
23.3 Java et HTML . . . 177
23.4 Cycle de vie d'une applet . . . 178
23.5 La balise APPLET . . . 179
23.6 Applets et restrictions . . . 181
23.7 Communication entre applets et browser . . . 181
23.8 La classe Applet . . . 181
23.9 L'interface AppletContext . . . 182
23.10 Communication entre applets . . . 183
24 Découvrir la programmation graphique . . . 185
24.1 Les composants . . . 185
24.2 Les layout manager . . . 186
24.3 La gestion des évènements . . . 186
24.4 Des widgets personnalisés . . . 187
24.5 Les différents widgets . . . 188
25 Gestion des évènements . . . 191
25.1 Le modèle évènementiel de Java . . . 191
25.2 La gestion des évènements graphiques . . . 192
25.3 Récapitulatif des évènements associés aux composants . . . 214
26 Les widgets . . . 215
26.1 Introduction . . . 215
26.2 Button . . . 222
26.3 Frame . . . 223
26.4 Panel . . . 224
26.5 Label . . . 225
26.6 TextField et TextArea . . . 226
26.7 Checkbox et CheckboxGroup . . . 229
26.8 Choice . . . 232
26.9 List . . . 233
26.10 Canvas . . . 235
26.11 Menu . . . 235
26.12 Scrollbar . . . 239
26.13 Dialog . . . 240
26.14 FileDialog . . . 242
26.15 ScrollPane . . . 244
26.16 Composants légers (Ligthweight) . . . 244

27 Ranger les widgets **247**
27.1 Gestion "manuelle" des placements 247
27.2 Généralités sur les Layouts 249
27.3 FlowLayout 250
27.4 BorderLayout 252
27.5 CardLayout 254
27.6 GridLayout 256
27.7 GridBagLayout 258
27.8 L'exemple du tutorail 260
27.9 Créer ses propres layouts 261
28 Dessiner sur une fenêtre graphique **263**
28.1 Introduction 263
28.2 Dessiner des formes géométriques 265
28.3 Les méthodes repaint, paint et update 265
28.4 Redessiner tout ou pas ? 266
28.5 Redéfinir la méthode update 268
28.6 Le "double buffering" 268
29 Couleurs et Fontes **271**
29.1 Dessiner du texte 271
29.2 Gestion des couleurs 273
30 Images **277**
30.1 Introduction 277
30.2 Manipulation des images 278
30.3 Contrôle du chargement des images 279
30.4 Les animations graphiques 283
30.5 Producteurs, consommateurs et filtres d'images 284
30.6 Fabriquer des images "à la main" 287
31 Le son **289**

23. Applets et applications autonomes

Sommaire

23.1 Application autonome 175
23.1.1 Les arguments de la ligne de commande 175
23.1.2 Les System Properties 176
23.2 Applets 177
23.3 Java et HTML 177
23.4 Cycle de vie d'une applet 178
23.5 La balise APPLET 179
23.6 Applets et restrictions 181
23.7 Communication entre applets et browser 181
23.8 La classe Applet 181
23.9 L'interface AppletContext 182
23.10Communication entre applets 183

23.1 Application autonome

Un programme *Java*est constitué d'une ou de plusieurs classes. Parmi toutes ces classes, il doit exister au moins une classe qui contient la méthode statique et publique `main` qui est le point d'entrée de l'exécution du programme. Comme d'habitude, commençant une tout petit exemple :

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

Cette classe définit une classe `Bonjour` qui ne possède qu'une seule méthode. La méthode `main` doit être déclarée `static` et `public` pour qu'elle puisse être invoquée par l'interpréteur *Java*. L'argument `args` est un tableau de `String` qui correspond aux arguments de la ligne de commande lors du lancement du programme.

Avant de pouvoir exécuter ce programme, il faut tout d'abord le compiler avec la commande `javac`.

```
javac Bonjour.java
```

La commande `javac` traduit le code source en code intermédiaire *Java*. Ce code est évidemment indépendant de la plate forme sur laquelle il a été compilé. Le compilateur *Java*produit alors autant de fichiers que classes qui ont été définies dans le fichier source. Les fichiers compilés ont l'extension `.class`.

Enfin, pour exécuter ce programme, il faut utiliser l'interpréteur de code *Java* et lui fournir le nom de la classe publique que l'on veut utiliser comme point de départ de notre programme.

```
java Bonjour
```

23.1.1 Les arguments de la ligne de commande

Les arguments de la ligne de commande permettent de passer des options, des valeurs etc. lors de l'exécution d'un programme. Les arguments de ligne de commandes sont passés à la méthode `main` dans un tableau de chaînes de caractères.

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

A titre d'exemple, le programme suivant :

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

affiche tous les arguments de la ligne de commande

```
% java Echo Bonjour et Bonne Année
Bonjour
et
Bonne
Année
```

Dans les langages *C* et *C++*, les arguments de la ligne de commande sont définis par

- **argc** : qui donne le nombre d'arguments
- **argv** : qui est pointeur vers un tableau de chaînes de caractères

Quant aux applications *Java*, le système ne passe qu'un seul paramètre qui est un tableau de **String**. On connaît le nombre d'arguments en examinant le champ **length** du tableau.

Contrairement à *C* et *C++*, *Java* n'inclut pas le nom de l'application dans les arguments.

```
class ParseCmdLine {
    public static void main(String[] args) {
        int i = 0, j;
        String arg;
        char flag;
        boolean vflag = false;
        String outputfile = "";

        while (i < args.length && args[i].startsWith("-")) {
            arg = args[i++];

            if (arg.equals("-verbose")) {
            System.out.println("mode verbose"); vflag = true;
            }
            else if (arg.equals("-output")) {
            if (i < args.length)
                outputfile = args[i++];
            else
                System.err.println("-output doit etre suivi d'un nom de fichier");
            if (vflag)
                System.out.println("output file = " + outputfile);
            }
            else {
                for (j = 1; j < arg.length(); j++) {
                flag = arg.charAt(j);
                switch (flag) {
                    case 'x':
                        if (vflag) System.out.println("Option x");
                        break;
                    case 'n':
                        if (vflag) System.out.println("Option n");
                        break;
                    default:
                        System.err.println("ParseCmdLine: option inconnue " + flag);
                        break;
                }
               }
            }
        }
        if (i == args.length)
            System.err.println("Usage: ParseCmdLine [-verbose] [-xn] [-output afile] filename");
        else
            System.out.println("Success!");
    }
}
```

23.1.2 Les System Properties

A TERMINER

23.2 Applets

Les *applets* ne sont pas des *applications java*. Cette affirmation a de quoi surprendre puisque vous avez déjà sûrement vu, en naviguant sur *Internet*, de très jolies exemples qui "tournent". Ce qui distingue une application autonome d'une *applet*, c'est :

- Une application *Java* s'exécute directement sous le contrôle de l'interpréteur de code *Java*, alors qu'une applet s'exécute sous le contrôle un *browser* ou d'un *appletviewer*.
- Une application peut accéder à toutes les ressources du système alors qu'une applet ne peut accéder qu'un ensemble limité de ressources. Par exemple, il ne sera pas possible d'ouvrir n'importe quel fichier qui réside sur votre système. Il est interdit aux applets d'ouvrir les résidant localement. On est presque rassuré par ces restrictions : il s'agit d'assurer la sécurité.

Une applet qui est une sorte d'application graphique est un objet de la classe `java.applet.Applet`.

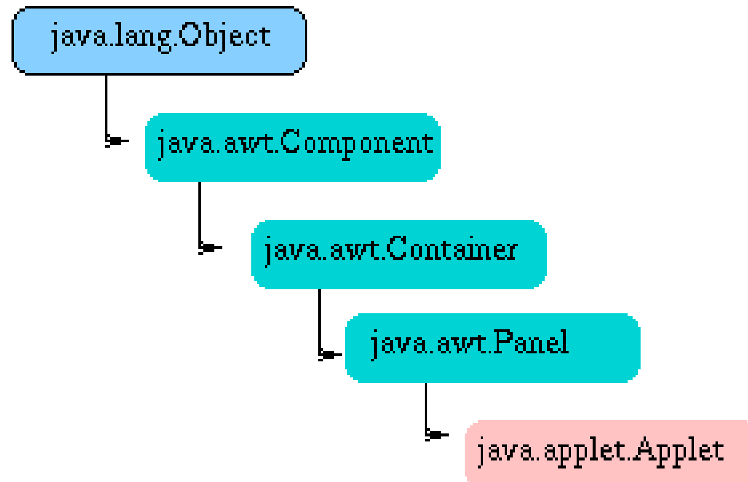

Fig. 23.1: La classe Applet

Du point du *browser* chargé d'exécuter du code *Java* une applet est tout simplement un type d'objet particulier à visualiser. Par exemple, les *browsers* sont, depuis bien longtemps, capable de reconnaître et d'interpréter un clip vidéo inséré dans une page *HTML*. Généralement ce n'est pas le *browser* qui est capable d'interpréter un clip vidéo . Par contre, il est capable de faire appel à un programme externe pour l'interpréter : ce sont les *helper applications*.

Cette technique peut évidemment s'utiliser pour exécuter du code *Java*. Il suffit de disposer d'un programme externe (fourni avec *JDK*) capable d'exécuter les applets *Java* et configurer le *browser* pour qu'il fasse appel à ce programme. Un tel programme existe est c'est le fameux *appletviewer*.

Cette approche n'est pas la plus agréable du point de vue visuel. En effet, l'*appletviewer*, en tant programme autonome, ouvre sa propre fenêtre pour exécuter le code *Java*. C'est pourquoi, *SUN* avec *HotJava*, *NetScape* et *MicroSoft* avec *Internet Explorer* ont implanté dans leur *browser* leur propre *appletviewer* de manière à insérer les sorties (généralement graphiques) au sein même de leur *browser*.

23.3 Java et HTML

Pour insérer une applet *Java*dans une page HTML, il faut : Une classe public (par exemple, `Bonjour.class`), point de départ de l'applet Un fichier *HTML* contenant la balise :

```
<applet
        code = Bonjour.class
        width = 200,
        height = 200>
</applet>
```

Il suffit alors d'utiliser l'*appletviewer* ou n'importe quel *browser HTML* qui reconnaît la balise applet pour visualiser cette page. Les balises applet indiquent qu'il faut exécuter du code *Java*lors de l'ouverture de la page. L'argument code de cette balise *HTML* précise la classe *Java*, point de départ de l'applet. Les arguments `width` et `height` définissent les dimensions en *pixels* du cadre réservé à l'applet dans la page *HTML*. Si vous essayez cet exemple, vous aurez une bien mauvaise surprise : la chaîne de caractères `"Bonjour"` s'affiche, non pas dans la page `HTML`, mais dans la console *Java*. C'est décevant !

Pourquoi les sorties ne s'affichent pas sur la page `HTML` ? Contrairement aux applications autonomes, une applet est d'emblée une application graphique. Par conséquent, les entrées/sorties standard s'effectue sur une console. Le comportement de note applet est tout à fait normal.

Comment faire pour afficher ces sorties sur la page HTML ? La partie de la page HTML réservée à l'applet *Java* doit être vue comme une fenêtre graphique. Il faut donc, non pas utiliser les entrées/sorties standard, mais plutôt dessiner du texte dans la partie de la page *HTML* réservée à l'applet.

Nous détaillerons, plus tard, les possibilités graphiques de *Java*. En attendant, contentons nous simplement du fait que la méthode **`paint`** est invoquée (redéfinition de la méthode **`paint`** de la classe de base de l'applet) chaque fois que l'applet passe à l'état "visible". Il appartient donc à l'applet de définir de qui doit être fait dans cette méthode.

```
public void paint(Graphics g)
```

Sans rentrer dans les détails, signalons que l'objet g de la classe **`Graphics`** est celui rattaché à notre applet et c'est sur cet objet que les modifications graphiques seront effectuées. Parmi les multiples méthodes de la classe **`Graphics`**, la méthode

```
public void drawString(String s, int x, int y)
```

permet de dessiner le texte contenue dans la chaîne **`s`** à partir de la position donnée par les coordonnées `(x, y)` relativement au coin supérieur gauche de la zone réservée à l'applet dans la page.

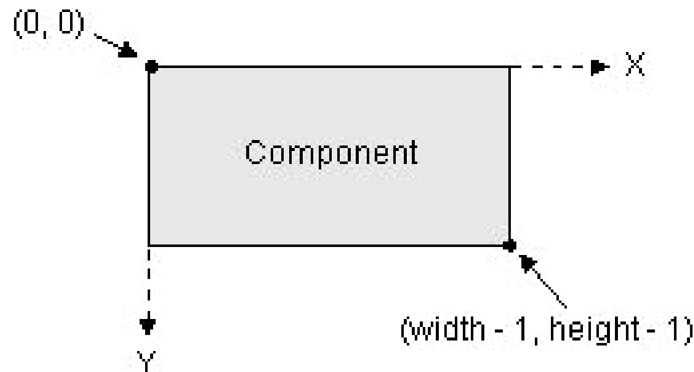

FIG. 23.2: Le système de coordonnées

Nous avons, à présent, tout ce qu'il nous faut pour faire en sorte que l'applet écrive le texte `"Bonjour"` directement sur la page *HTML*.

```
/** Classe Bonjour qui se contente d'écrire la chaîne Bonjour
 */
public class Bonjour extends java.applet.Applet {
    /** La méthode paint est une redéfinition de la méthode paint de la classe Applet.
        Elle a pour rôle de dessiner la chaîne Bonjour chaque fois que nécessaire
    */
    public void paint(java.awt.Graphics g) {
        g.drawString("Bonjour", 100, 100) ;
    }
}
```

On associe un fichier *HTML* qui devrait permettre de visualiser l'applet :

```
<html>
    <head>
    <title>Bonjour</title>
    </head>
    <body>
        Vous devriez voir la chaîne de caractères "Bonjour" ci-dessous
        <applet
            code=Bonjour.class
            width=200,
            height=200>
        </applet>
    </body>
</html>
```

Ce petit exemple illustre bien des concepts que nous avons détaillés (langage *Java*) ou que nous détaillerons plus tard (création des programmes *Java*avec environnement graphique : *Abstract Windowing Toolkit - AWT*).

23.4 Cycle de vie d'une applet

Une applet hérite beaucoup de propriétés des objets graphiques. En particulier, il se découpe en quatre parties :

Initialisation : c'est la phase où l'appletviewer charge l'applet et lui demande d'effectuer les initialisations prévues dans le code *Java*.

Visible : chaque fois que l'applet devient ou redevient visible à l'utilisateur, il est demandé à l'applet d'effectuer les opérations nécessaires pour se dessiner de nouveau.

Invisible : chaque fois que l'applet devient ou redevient invisible à l'utilisateur, il est demander à l'applet d'effectuer les opérations nécessaires pour éventuellement libérer les ressources ou interrompre certaines de ces activités. Par exemple, si l'applet consiste une animation graphique, pendant qu'elle est invisible à l'utilisateur, l'animation peut être suspendue.

Arrêt définitif : lorsque l'applet doit s'arrêter (changement de page HTML, par exemple), il est demander à l'applet de se terminer proprement.

A chacun de ses états correspond une méthode de la classe `Applet`.

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

public class Simple extends Applet {
  public void init() {         // Initialisation
    System.out.println("Initialisation");
  }
  public void start() {        // Visible
    System.out.println("Start");
  }
  public void stop() {         // Invisible
    System.out.println("Stop");
  }
  public void destroy() {      // Arret définitif
    System.out.println("destroy");
  }
  public boolean mouseDown(Event event, int x, int y) {
    System.out.println("Mouse Down");
    return true;
  }
}
```

Essayer cet exemple et voyez les affichages produits sur la sortie standard pour voir les différents états par lesquels passe cet applet.

23.5 La balise APPLET

Les applets, contrairement aux applications autonomes, ne peuvent utiliser les variables d'environnement du système. Par contre, en vue de configurer l'applet, les paramètres données dans la balise *HTML* `<applet>` peuvent être récupérés par la méthode `Applet.getParmeter`.

La forme générale de la balise APPLET est :

```
< APPLET
    CODE = appletFile
    WIDTH = pixels
    HEIGHT = pixels
    [CODEBASE = codebaseURL]
    [ALT = alternateText]
    [NAME = appletInstanceName]
    [ALIGN = alignment]
    [VSPACE = pixels]
    [HSPACE = pixels]
>
[< PARAM NAME = appletParameter1 VALUE = value >]
[< PARAM NAME = appletParameter2 VALUE = value >]
. . .
[alternateHTML]
</APPLET>
```

`CODEBASE=` *codebaseURL*

Attribut optionnel précisant la base URL de la classe de l'applet. Si cet attribut n'est pas spécifié, la base du document HTML est pris comme base URL de la classe.

`CODE =` *nom de fichier .class*

Attribut obligatoire qui donne le nom de la classe qui contient l'applet. Ce nom de classe est toujours relatif à l'attribut `CODEBASE` ; il n'est pas possible de donner un nom absolu.

`ALT =` *Texte*

Permet d'afficher un message pour les browser qui comprennent la balise `applet` mais qui sont incapable d'exécuter les applets.

`NAME = appletInstanceName`
: Attribut optionnel donne l'instance d'une applet dans la même page *HTML* de manière à établir une communication entre applets.

`WIDTH = pixels et HEIGHT = pixels`
: Attributs obligatoire fixant la taille initiale de l'applet en pixels.

`ALIGN = alignment`
: Positionnement de l'applet dans une page *HTML* ; les valeurs permises sont **left**, **right**, **top**, **texttop**, **middle**, **absmiddle**, **baseline**, **bottom**, **absbottom**. Ce sont les mêmes valeurs que celles que l'on utilise pour la balise *HTML* **IMG**.

`VSPACE = pixels et HSPACE = pixels`
: Nombre de pixels entre l'applet et le reste de la page **HTML** comme pour la balise **IMG**.

`< PARAM NAME = appletParameter1 VALUE = value >`
: Ces balises *PARAM* sont les seuls moyens de passer des arguments à une applet. C'est l'équivalent des arguments de la ligne de commande pour les applets. C'est le code l'applet qui se charge de récupérer ces arguments avec les méthodes `getParameter`.

`alternateHTML`
: Texte *HTML* affiché dans les browsers qui ne reconnaissent pas les balises `APPLET`. Ce texte est complètement ignoré par les autres browsers.

```
<html>
  <head>
    <title>Les balises HTML</title>
  </head>
  <body>
    <h1>Les balises HTML</h1>
        <applet
            code=Balise.class
            width=200
            height=100
            alt = "Votre browser ne peut exécuter cet applet"
            align = top
            vspace = 10
            hspace = 10
        >
        <param name=chaine1 value=Bonjour>
        <param name=chaine2 value="tout le monde">
Votre browser ne peut exécuter les applets Java
        </applet>
  </body>
</html>
```

```
public class Balise extends java.applet.Applet {
  String s1, s2;
  public void init() {
    s1 = getParameter("chaine1");
    s2 = getParameter("chaine2");
  }
  public void paint(java.awt.Graphics g) {
    g.drawString(s1 + "\n" + s2 + "\n", 50, 50) ;
  }
}
```

Comme toute application graphique, une applet peut contenir les composants graphiques suivantes :

- des boutons (java.awt.Button (voir 26.2))
- des “Checkboxes” (java.awt.Checkbox (voir 26.7))
- des lignes de texte (java.awt.TextField (voir 26.6))
- des zones de texte éditables (java.awt.TextArea (voir 26.6))
- des étiquettes (java.awt.Label (voir 26.5))
- des listes défilants (java.awt.List (voir 26.9))
- des listes déroulants (java.awt.Choice (voir 26.8))
- des glisières et ascenceurs (java.awt.Scrollbar (voir 26.12))
- des zones de dessin (java.awt.Canvas (voir 26.10))
- des menus (java.awt.Menu (voir 26.11), java.awt.MenuItem (voir 26.11), java.awt.CheckBox (voir 26.7), java.awt.MenuItem (voir 26.11))

– des containers (java.awt.Panel (voir 26.4), java.awt.Window (voir 26.3) et leurs classes dérivées)

Une applet, comme toute application graphique, peut également réagir aux actions de l'utilisateur :

– gestion de la souris
– gestion du clavier
– etc.

23.6 Applets et restrictions

L'exécution d'une applet est une opération "sensible" puisqu'une machine cliente exécute un code provenant d'une autre machine. Pour des raisons de sécurité, les applets ne peuvent accéder à toutes les ressources de la machine cliente. Chaque *browser* adopte sa propre politique de sécurité et sont susceptibles d'évoluer. Quoiqu'il en soit, ils ont en commun un certain nombre de caractéristiques. Une applet

– ne peut charger de bibliothèques ou définir des méthodes natives.
– ne peut avoir accès aux fichiers (en lecture ou écriture) de la machine cliente.
– ne peut établir de connections avec une machine différente de la machine hébergeant l'applet.
– ne peut démarrer aucune programme sur la machine cliente.
– ne peut lire toutes les propriétés du système client (System Properties).
– possède un "look" différent de celui d'une application.

23.7 Communication entre applets et browser

La classe **Applet** et l'interface **AppletContexte** permettent d'établir la communication entre une applet et le browser qui l'exécute.

Par exemple, la méthode **showStatus** de la classe **Applet** permet d'afficher une chaîne de caractères dans le coin du *browser*. La méthode **showDocument**de l'interface **AppletContext** affiche sur le *browser* le document spécifié en argument.

```
public void showDocument(java.net.URL url)
public void showDocument(java.net.URL url, String targetWindow)
```

23.8 La classe Applet

`public final void  setStub (AppletStub stub)`

A TERMINER

`public boolean  isActive ()`

Détermine si l'applet est dans un état actif.

`public URL  getDocumentBase ()`

Retourne l'*URL* du document qui contient l'applet.

`public URL  getCodeBase ()`

Retourne l'*URL* de l'applet.

`public String  getParameter (String name)`

Retourne la valeur du paramètre (voir 23.5) de l'applet.

`public AppletContext  getAppletContext ()`

Retourne le contexte (voir 23.7) de l'applet.

`public void  resize (int width, int height)`

Redimensionne l'applet selon les valeurs données en arguments.

`public void  resize (Dimension d)`

Idem avec en argument un objet de type Dimension.

`public void  showStatus (String msg)`

Ecrit la chaîne spécifiée en argument dans la fenêtre d'état des browsers.

`public Image  getImage (URL url)`

Retourne un objet de type **Image** pour l'image spécifié en argument. Le paramètre **url** est une URL absolue. Si l'image spécifié n'est pas trouvé, il n'y aucun message d'erreur.

`public Image  getImage (URL url, String name)`

Retourne un objet de type `Image` pour l'image spécifié en argument. Le paramètre `url` est une URL absolue et le fichier image est donnée par l'argument `name` relativement à `url`. Si l'image spécifié n'est pas trouvé, il n'y aucun message d'erreur.

`public AudioClip getAudioClip (URL url)`

Retourne un objet de type `AudioClip` pour le clip audio spécifié en argument. Si clip audio spécifié n'est pas trouvé, il n'y aucun message d'erreur.

`public AudioClip getAudioClip (URL url, String name)`

Retourne un objet de type `AudioClip` pour le clip audio spécifié en argument. Le paramètre `url` est une URL absolue et le fichier clip audio est donnée par l'argument `name` relativement à `url`. Si clip audio spécifié n'est pas trouvé, il n'y aucun message d'erreur.

`public String getAppletInfo ()`

Retourne les informations concernant l'applet : auteur, version et copyright.

A terminer

`public String[][] getParameterInfo ()`

`public void play (URL url)`

Déclenche la diffusion du clip audio.

`public void play (URL url, String name)`

Déclenche la diffusion du clip audio.

`public void init ()`

La méthode `init` est invoquée lors de l'l'initialisation (voir 23.4) de l'applet.

`public void start ()`

La méthode `start` est invoquée lors du démarrage (voir 23.4) de l'applet.

`public void stop ()`

La méthode `stop` est invoquée lors du arrêt (voir 23.4) de l'applet.

`public void destroy ()`

La méthode `destroy` est invoquée lors du destruction (voir 23.4) de l'applet.

23.9 L'interface AppletContext

`public abstract AudioClip getAudioClip (URL url)`

Retourne l'objet `AudioClip` correspondant au clip audio spécifié par `url`. Le paramètre `url` est une URL absolue.

`public abstract Image getImage (URL url)`

Retourne un objet de type `Image` pour l'image spécifié en argument. Le paramètre `url` est une URL absolue. Si l'image spécifié n'est pas trouvé, il n'y aucun message d'erreur.

`public abstract Applet getApplet (String name)`

Retourne l'objet `Applet` représentant l'applet en cours d'exécution sur la page *HTML*. Le paramètre `name` est une chaîne figurant soit dans la balise `applet` soit dans la balise `param`. Si aucune applet de ce nom n'existe, une référence `null` est retourné.

`public abstract Enumeration getApplets ()`

Retourne la liste de tous les applets figurant dans le document du contexte de l'applet.

`public abstract void showDocument (URL url)`

Remplace la page *HTML* courante par celle spécifiée par le paramètre `url`. Le paramètre `url` est une URL absolue.

`public abstract void showDocument (URL url, String target)`

Affiche la page *HTML* spécifiée par le paramètre `url` dans le *frame* spécifié par le paramètre `target`. Le paramètre `url` est une URL absolue. Le paramètre `target` est l'une des chaînes suivantes :

"_self"	frame courant
"_parent"	frame père
"_top"	frame racine
"_blank"	nouvelle fenêtre sans frame et sans nom
nom	nouvelle fenêtre sans frame mais avec un nom

`public abstract void showStatus (String status)`

Ecrit la chaîne spécifiée en argument dans la fenêtre d'état des browsers.

23.10 Communication entre applets

Deux applets d'une même page HTML peuvent communiquer entre elles. Pour cela il suffit de :

– Donner un nom à l'applet avec qui on veut communiquer ; ce qui peut se faire soit en l'ajoutant dans la balise *HTML* **applet**

```
<applet codebase=Programmes/awt/applet CODE=Recepteur.class WIDTH=450 HEIGHT=200 NAME="Recepteur">
</applet>
```

soit en rajoutant un paramètre supplémentaire dans la liste des paramètres d'une applet.

```
<applet codebase=Programmes/awt/applet CODE=Recepteur.class WIDTH=450 HEIGHT=200>
<param name="nom" value="Recepteur">
</applet>
```

– De récupérer le contexte de l'applet cible :

```
Applet r = getAppletContext().getApplet("Recepteur");
```

– D'exécuter une action que l'applet cible peut faire :

```
((Recepteur)r).faireQuelqueChose("Message de l'émetteur");
```

Ne marche (pour le moment) que sur HotJava.

```
public class Emetteur extends java.applet.Applet {
  String s = "coucou !";
  public void init() {
    //    java.applet.Applet r = null;
    Recepteur    r = (Recepteur) getAppletContext().getApplet("Recepteur");
    r.faireQuelqueChose("Message de l'émetteur !! ");
  }

  public void paint(java.awt.Graphics g) {
    g.drawString(s, 50, 50) ;
  }
}
```

```
public class Recepteur extends java.applet.Applet {
  String s = "J'attends quelque chose";
  public void paint(java.awt.Graphics g) {
    g.drawString(s, 50, 50) ;
  }
  public void faireQuelqueChose(String s) {
    this.s = s;  repaint();
  }
}
```

24. Découvrir la programmation graphique

Sommaire

24.1 Les composants 185
24.2 Les layout manager 186
24.3 La gestion des évènements 186
24.4 Des widgets personnalisés 187
24.5 Les différents widgets 188

Une application graphique *Java* est une application possédant une certain nombre de fenêtre graphiques et qui va interagir avec l'utilisateur de cette application. Le concept de base d'une application graphique est le **widget**. Un *widget* est l'entité de base d'une interface graphique qui réagit aux actions d'un utilisateur : manipulation de la souris, du clavier et tout autre évènement du système graphique.

Les *widgets* sont de forme rectangulaires sont organisés hiérarchiquement. Un *widget* particulier appelé *widget principal widget racine* ou se trouve au sommet de cette hiérarchie.

Du point de vue du programmeur, une application graphique *Java* est un objet d'une classe contenant un ou plusieurs composants graphiques. L'ensemble des composants graphiques sont disponibles à travers les classes du package **java.awt**.

Généralement, à partir de ses composants standards et des comportements prédéfinis aux actions de l'utilisateurs, le programmeur, à travers l'héritage, crée de nouvelles classes pour son application en redéfinissant l'apparence et le comportement prédéfinis.

Nous verrons, plus loin, en détail les différents type de composants (voir 26) que l'on dispose en *Java*, ainsi que de la gestion des évènements (voir 25).

24.1 Les composants

Commençons par une toute petite applet. Une applet est une application graphique dans lequel on peut ajouter toutes sortes de *widgets*. Nous allons insérer dans notre applet, deux boutons qui sont des objets de la classe **java.awt.Button**

```
Button b1 = new Button("Coucou !");
Button b2 = new Button("! uocuoc");
add(b);
```

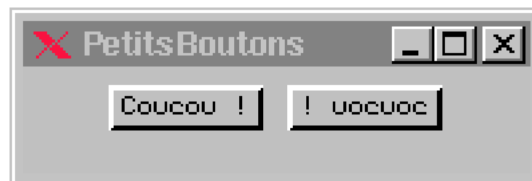

Fig. 24.1: Un premier exemple

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

public class PetitsBoutons extends Applet  {
  public void init() {
    Button b1 = new Button("Coucou !");
    Button b2 = new Button("! uocuoc");
    add(b1);
    add(b2);
    }
public static void main(String args[]) {
  PetitsBoutons m = new PetitsBoutons();
  m.init();
  Frame f = new Frame("PetitsBoutons");
  f.setSize(200,70);
  f.pack();
  f.show();
  f.add(m);
  }
}
```

Une applet est un container(*Containers* en anglais). Les containers sont constitués de composants qui s'affiche à l'écran dès la portion de la fenêtre dans laquelle ils sont placés devient visible à l'utilisateur. Le fait d'ajouter un composant avec la méthode `add` conduit le système à dessiner les objets devenus visibles.

24.2 Les layout manager

Nous venons de parler de la position des composants dans une fenêtre. Le code que nous avons écrit ne contient aucune précision quant à ces positions. Comment cela marche ? Par défaut, les containers se débrouillent pour placer leurs composants selon une configuration prédéfinie. Ainsi, à chaque container est associé un gestionnaire de placement (*Layout manager* (voir 27)).

Il existe plusieurs manière de placer des composants dans une fenêtre : il existe donc plusieurs type de gestionnaires dans *Java*. Nous verrons cela en détail dans le chapitre 27 entièrement consacré aux (*Layout* (voir 27).)`Layout`.

Contentons nous de remarquer que le gestionnaire par défaut, pour certains containers, place les objet les un à la suite des autres, de gauche à droit, selon l'ordre dans lequel ils ont été insérés. Si la place vient à manquer sur une même ligne, les objets restants sont placés dans les lignes suivantes. Lorsqu'on redimentionne la fenêtre, c'est le rôle de ce gestionnaire de replacer correctement les objets.

24.3 La gestion des évènements

Comme vous avez pu le remarquer, l'action de l'utilisateur sur le bouton de notre applet ne produit aucun effet ou presque. Le bouton s'enfonce bien (comme tout bouton qui se respecte) mais aucune autre action ne survient. Comment associer à la pression sur le bouton une action particulière ?

Tout objet peut décider d'être à l'écoute d'évènements. En particulier, s'il décide de réagir aux actions de la souris, la classe à laquelle appartient ce objet devra implanter l'interface `MouseListener`.

```
public  void mouseClicked(MouseEvent e)
public  void mousePressed(MouseEvent e)
public  void mouseReleased(MouseEvent e)
public  void mouseEntered(MouseEvent e)
public  void mouseExited(MouseEvent e)
```

Supposons que notre applet décide d'être à l'écoute de la souris.

```
public class UnPetitBoutonPression extends Applet   implements MouseListener { ... }
```

La classe `UnPetitBoutonPression` devra donc implanter toutes les méthode de l'interface `MouseListener`. L'implantation de ces méthodes va définir les actions à réaliser lorsqu'un des cinq évènements se produit :

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

public class PetitsBoutonsPression extends Applet implements MouseListener {
  Button b1, b2;
  public void init() {
    b1 = new Button("Coucou !");
    b2 = new Button("! uocuoc");
    add(b1);
    add(b2);
    b1.addMouseListener(this);
    b2.addMouseListener(this);
```

```
    }
    public void mousePressed(MouseEvent e) {
      String s = b1.getLabel(); b1.setLabel(b2.getLabel()); b2.setLabel(s);
      }
    public void mouseReleased(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void mouseClicked(MouseEvent e) { }

  public static void main(String args[]) {
    PetitsBoutonsPression m = new PetitsBoutonsPression();
    m.init();
    Frame f = new Frame("PetitsBoutonsPression");
    f.setSize(200,70);
    f.pack();
    f.show();
    f.add(m);
    }
  }
```

24.4 Des widgets personnalisés

L'applet que nous venons de voir est extrêmement réduit et les évènements à gérer sont en petit nombre. Dans la pratique, il y peut avoir un nombre incalculable (ou presque) d'action à gérer. Il est alors fastidieux de programmer toute cette gestion dans le code des méthodes de l'interface `MouseListener`. Il est évidement bien plus agréable de "décentraliser" le code de gestion des évènements de la souris. Chaque composant définit sa propre gestion facilitant ainsi une programmation bien plus modulaire. Notre bouton, par exemple, devrait gérer toute seule la gestion des évènements sans se reposer sur celle de l'applet. Pour ce faire, il convient de définir, pour notre bouton, une classe dérivée de la classe `java.awt.button` dans laquelle, on redéfinira les méthodes de l'interface `MouseListener`.

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

public class MeilleursPetitsBoutons extends Applet {
  MonBouton b1, b2;
  public void init() {
    b1 = new MonBouton("Coucou !", this);
    b2 = new MonBouton("! uocuoc", this);
  }
  public void echanger() {
    String s = b1.getLabel(); b1.setLabel(b2.getLabel()); b2.setLabel(s);
  }
public static void main(String args[]) {
  MeilleursPetitsBoutons m = new MeilleursPetitsBoutons();
  m.init();
  Frame f = new Frame("MeilleursPetitsBoutons");
  f.setSize(200,70);
  f.pack();
  f.show();
  f.add(m);
  }
}

class MonBouton extends Button implements MouseListener {
  MeilleursPetitsBoutons pere;
  public MonBouton(String s, MeilleursPetitsBoutons pere) {
    super(s);
    this.pere = pere;
    setForeground(Color.red);
    setBackground(Color.blue);
    pere.add(this);
    addMouseListener(this);
  }
  public void mousePressed(MouseEvent e) { pere.echanger(); }
  public void mouseReleased(MouseEvent e) {  }
  public void mouseEntered(MouseEvent e) { }
  public void mouseExited(MouseEvent e) { }
  public void mouseClicked(MouseEvent e) { }
}
```

Déporter la gestion des évènements sur les boutons permet de maîtriser mieux la programmation de ces évènements. Cette remarque est également vraie pour l'apparence et autres propriétés des composants. Si l'on désire, donner une apparence particulière à nos boutons, plutôt que de le faire dans la classe principale, on déportera également ce code dans la nouvelle classe que l'on vient de se créer :

```
class MonBouton extends Button implements MouseListener {
  MeilleursPetitsBoutons pere;
  public MonBouton(String s, MeilleursPetitsBoutons pere) {
    super(s);
    this.pere = pere;
    setForeground(Color.red);
    setBackground(Color.blue);
    pere.add(this);
    addMouseListener(this);
  }
  public void mousePressed(MouseEvent e) { pere.echanger(); }
  public void mouseReleased(MouseEvent e) {  }
  public void mouseEntered(MouseEvent e) { }
  public void mouseExited(MouseEvent e) { }
  public void mouseClicked(MouseEvent e) { }
}
```

24.5 Les différents widgets

Comme nous l'avons déjà dit, un widget est un composant graphique. Il existe plusieurs natures de widgets. Tous ces widgets sont dérivées de la classe **Component**.

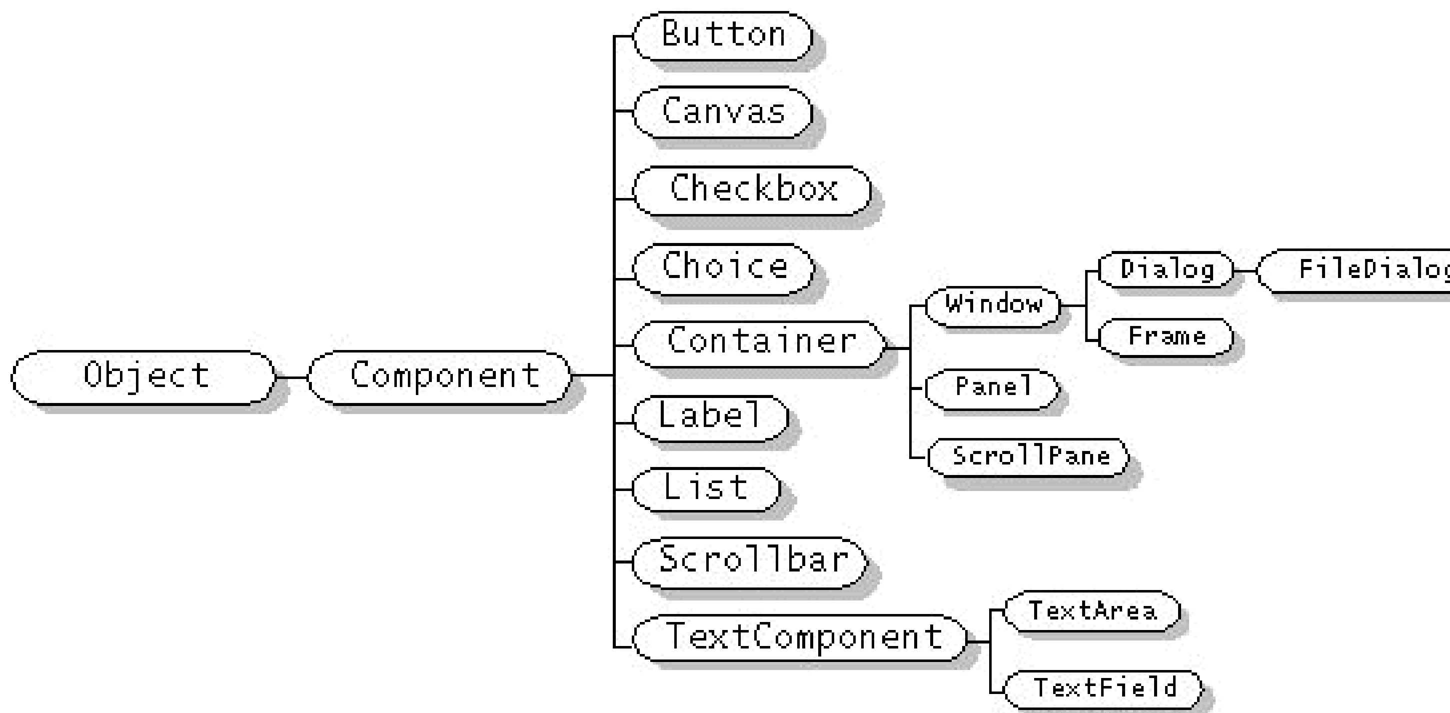

Fig. 24.2: Les différents widgets

Nous consacrerons un chapitre entier à ces widgets (voir 26). En attendant, voici un exemple, provenant du tutorial de *jdk*, réunissant une certain nombre de widgets.

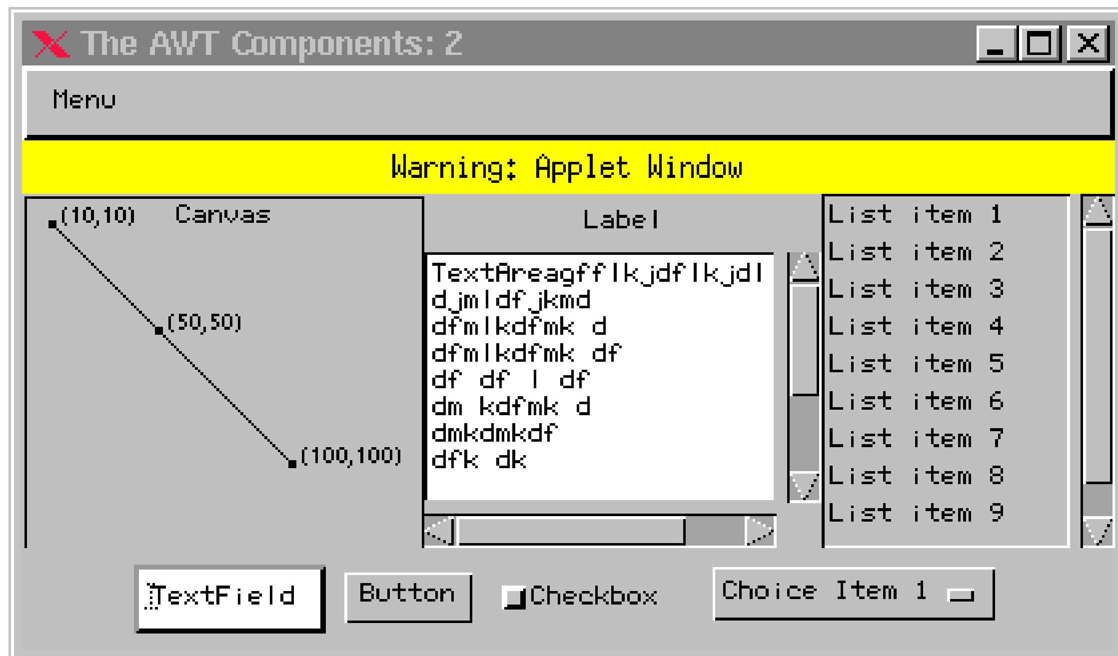

Fig. 24.3: Quelques widgets

25. Gestion des évènements

Sommaire

25.1 Le modèle évènementiel de Java 191
25.1.1 Les objets évènement 191
25.1.2 Les objets récepteurs 192
25.1.3 Les objets émetteurs ou sources 192
25.1.4 Un premier exemple 192
25.2 La gestion des évènements graphiques 192
25.2.1 Les évènements prédéfinis 193
25.2.2 Les classes adaptateurs 195
25.2.3 Les classes imbriquées 196
25.2.4 Gestion d'évènements pour composants étendus 196
25.2.5 L'interface ActionListener et la classe ActionEvent 198
25.2.6 Les interfaces MouseListener, MouseMotionListener et la classe MouseEvent 198
25.2.7 L'interface KeyListener et la classe KeyEvent 201
25.2.8 L'interface WindowListener et la classe WindowEvent 203
25.2.9 L'interface ComponentListener et la classe ComponentEvent 205
25.2.10 L'interface ContainerListener et la classe ContainerEvent 206
25.2.11 L'interface FocusListener et la classe FocusEvent 207
25.2.12 L'interface TextListener et la classe TextEvent 209
25.2.13 L'interface ItemListner et la classe ItemEvent 210
25.2.14 L'interface AdjustmentListener et la classe AdjustmentEvent 212
25.2.15 L'interface InputMethodListener et la classe InputMethodEvent 213
25.3 Récapitulatif des évènements associés aux composants 214
25.3.1 Suppression de la gestion des évènements 214
25.3.2 La queue d'évènements 214

25.1 Le modèle évènementiel de Java

Un évènement est une message qu'un objet envoie à un autre objet lui signalant la survenue de quelque chose de "remarquable". Il y a donc un émetteur et un récepteur. Dans une application graphique, il y a généralement une quantité importante d'objets et d'évènements possibles. Tous les objets ne sont forcément intéressés par l'ensemble des évènements. Par exemple, un bouton graphique ne s'intéresse qu'aux seules évènements souris ; les évènements clavier ne l'intéressent pas. Aussi, pour être récepteur d'un évènement, un objet devra signaler son intérêt pour cet évènement. Un même évènement peut intéresser plusieurs objets et inversement un même objet peut s'intéresser à plusieurs évènements.

Le modèle évènementiel de *Java* est basé sur les objets évènements, les objets récepteurs et sources d'évènements.

25.1.1 Les objets évènement

La classe de base pour les évènements est la classe `java.util.EventObject`

```
public class EventObject implements java.io.Serializable {
    protected transient Object  source;
    public EventObject(Object source);
    public Object getSource();
    public String toString();
}
```

Le champ `source` code l'objet source de l'évènement.

Un évènement est alors une instance de cette classe ou d'une de ses sous classes. Par exemple, l'ensemble des évènements graphiques sont des objets des classes dérivées de `java.util.EventObject`.

```
class TestEvent extends java.util.EventObject {
  public TestEvent(Object o) { super(o); }
}
```

25.1.2 Les objets récepteurs

Un récepteur est un objet à qui le système signale la survenue d'un évènement et ce chaque fois qu'il se produit. L'émetteur de l'évènement doit savoir quelle méthode il faut invoquer lorsqu'un évènement particulier survient. Les récepteurs sont implantés à l'aide d'interfaces et l'interface de base de tous les récepteurs est `java.util.EventListener`. Le nombre de méthodes définies dans un récepteur est variable.

```
interface TestListener extends java.util.EventListener {
  void TestEvènement1(TestEvent e)
  void TestEvènement2(TestEvent e)
}
```

La signature des méthodes de ces interfaces est toujours de la forme `void nom(TypeEvénement e)`.

25.1.3 Les objets émetteurs ou sources

Les émetteurs sont les objets qui déclenchent les évènements. Ce sont des objets d'une classe qui doit implanter les méthodes pour permettre à d'autres objets se s'abonner pour recevoir un évènement et éventuellement se désabonner.

```
public void addTestListener(TestListener l);
public void removeTestListener(TestListener l);
```

25.1.4 Un premier exemple

```
class TestEmetteur {
  private TestListener AlEcoute = new TestListener[3];
  int sommet = 0;
  public synchronized void addTestListener(TestListener l) {
    if (sommet < AlEcoute.length)  AlEcoute[sommet++] = l;
  }
  public synchronized void removeTestListener(TestListener l) {
    for (int i = 0;  i <= sommet && AlEcoute[i] != l; i++);
    if (i <= sommet)
        for (; i < sommet; AlEcoute[i] = AlEcoute[i+1], i++);
    sommet--;
  }
  public void notifyTestEvènement1() {
    TestEvent e = new TestEvent(this);
// clone
    for (int i = 0; i < AlEcoute.length; i++) {
      TestListener l = AlEcoute[i];
      l.TestEvènement(e);
      }
  }

}
```

25.2 La gestion des évènements graphiques

La gestion des évènements graphiques suit le schémas général que nous venons d'exposer :

- Les évènements sont engendrés par des émetteurs
- Les récepteurs s'inscrivent auprès d'un ou plusieurs émetteurs pour signaler leur intérêt pour les évènements.
- L'action à réaliser lorsque le ou les évènements attendus surviennent.

Les objets qui veulent gérer les évènements graphiques sont des instances d'une classe qui implante l'interface `ActionListener`. On devra donc trouver dans tout programme qui veut gérer les évènements les trois choses suivantes :

- une classe qui implante une interface `ActionListener`

  ```
  public class X implements ActionListener { ... }
  ```

- une composant qui s'inscrit auprès d'un ou plusieurs émetteurs

  ```
  un_composant.addActionListener(objet_de_la_class_X);
  ```

– l'implantation des méthodes de l'interface `ActionListener` i.e. l'action a réaliser

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

Voici un exemple d'une application graphique qui réagit lorsqu'on appuie sur le bouton.

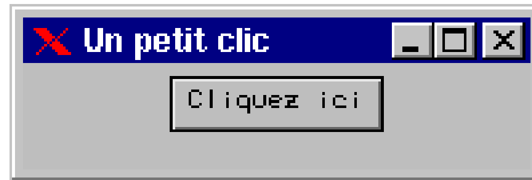

Fig. 25.1: Exemple de gestion d'évènements graphiques

```
public class Click extends java.applet.Applet
                   implements java.awt.event.ActionListener {
  java.awt.Button bouton;
  boolean b = true;
  public void init() {
    bouton = new java.awt.Button("Cliquez ici");
    add("Center", bouton);
    bouton.addActionListener(this);
  }
  public void actionPerformed(java.awt.event.ActionEvent e) {
    b = !b;
    bouton.setLabel(b ? "Cliquez ici" : "ici zeuqilC");
  }
}
```

25.2.1 Les évènements prédéfinis

Java définit un ensemble d'interfaces pour la gestion des évènements graphiques ; toutes ces interfaces sont dérivées de l'interface vide `java.util.EventListener`. Chacune de ces interfaces est composée d'une ou de plusieurs méthodes. Par exemple, comme nous l'avons déjà vu précédemment (voir 24.3), l'interface `MouseListener` permet la gestion des actions de la souris sur un composant graphique. Cette interface contient les méthodes suivantes :

```
public  void mouseClicked(MouseEvent e)
public  void mousePressed(MouseEvent e)
public  void mouseReleased(MouseEvent e)
public  void mouseEntered(MouseEvent e)
public  void mouseExited(MouseEvent e)
```

Un composant graphique qui est à l'écoute des évènements de la souris doit implanter cette interface `MouseListener`. En implantant cette interface, on définit les opérations que l'on veut voir effectuer à chacune des actions de l'utilisateur sur la souris.

Comme pour la souris, d'autres interfaces permettent la gestion de l'ensemble des évènements qui peuvent survenir dans une application graphique.

Interface	Méthodes
ActionListener (voir 25.2.5)	actionPerformed
AdjustmentListener (voir 25.2.14)	adjustmentValueChanged
ComponentListener (voir 25.2.9)	componentHidden componentMoved componentResized componentShown
ContainerListener (voir 25.2.10)	componentAdded componentRemoved
FocusListener (voir 25.2.11)	focusGained focusLost
ItemListener (voir 25.2.13)	itemStateChanged
KeyListener (voir 25.2.7)	keyPressed keyReleased keyTyped
MouseListener (voir 25.2.6)	mouseClicked mouseEntered mouseExited mousePressed mouseReleased
MouseMotionListener (voir 25.2.6)	mouseDragged mouseMoved
TextListener (voir 25.2.12)	textValueChanged
WindowListener (voir 25.2.8)	windowActivated windowClosed windowClosing windowDeactivated windowDeiconified windowIconified windowOpened

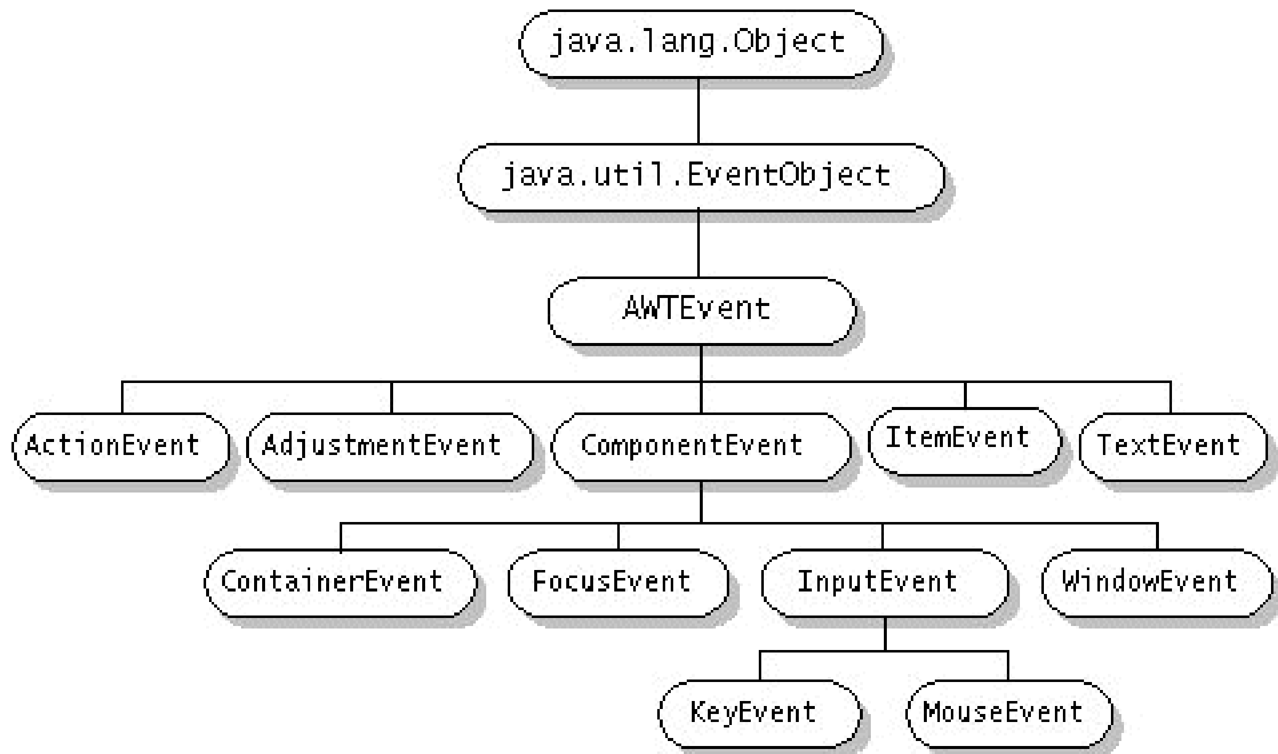

FIG. 25.2: Les objets évènements

25.2.2 Les classes adaptateurs

Un composant graphique qui est à l'écoute d'un évènement de la souris doit implanter l'interface `MouseListener` i.e. donner une implantation pour les toutes les méthodes de l'interface. Or, bien souvent, un composant ne s'intéresse pas à tous les évènements possibles mais qu'à un nombre limité d'entre eux. Par exemple, un bouton peut s'intéresser qu'au fait de cliquer. Dans ces cas, le composant qui implante l'interface `MouseListener` doit fournir l'implantation de toutes les méthodes de cette interface. Ce qui conduit le programmeur à fournir tout un ensemble de méthode vide.

```
public class Click1 extends java.applet.Applet implements java.awt.event.MouseListener {
  java.awt.Button bouton;
  boolean b = true;
  public void init() {
    bouton = new java.awt.Button("Cliquez ici");
    add("Center", bouton);
    bouton.addMouseListener(this);
  }
  public void mousePressed(java.awt.event.MouseEvent e) {
    b = !b;
    bouton.setLabel(b ? "Cliquez ici" : "ici zeuqilC");
  }
 public void mouseClicked(java.awt.event.MouseEvent e) {}
 public void mouseReleased(java.awt.event.MouseEvent e) {}
 public void mouseEntered(java.awt.event.MouseEvent e) {}
 public void mouseExited(java.awt.event.MouseEvent e) {}
}
```

Pour éviter cette inflation de méthodes vides, *Java* fournit, pour certaines des ces interfaces, une classe qui l'implante avec des méthodes vides. Par exemple, la classe `MouseAdapter` est définie comme suit :

```
public class MouseAdapter implements MouseListener {
  public  void mouseClicked(MouseEvent e) {}
  public  void mousePressed(MouseEvent e) {}
  public  void mouseReleased(MouseEvent e) {}
  public  void mouseEntered(MouseEvent e) {}
  public  void mouseExited(MouseEvent e) {}
```

```
}
```

Il suffit alors de se définir une sous classe de l'une de celles-ci et de ne redéfinir que les méthodes qui nous intéressent.

Interface	Classe adaptateur
ActionListener (voir 25.2.5)	aucune
AdjustmentListener (voir 25.2.14)	aucune
ComponentListener (voir 25.2.9)	ComponentAdapter
ContainerListener (voir 25.2.10)	ContainerAdapter
FocusListener (voir 25.2.11)	FocusAdapter
ItemListener (voir 25.2.13)	aucune
KeyListener (voir 25.2.7)	KeyAdapter
MouseListener (voir 25.2.6)	MouseAdapter
MouseMotionListener (voir 25.2.6)	MouseMotionAdapter
TextListener (voir 25.2.12)	aucune
WindowListener (voir 25.2.8)	WindowAdapter

25.2.3 Les classes imbriquées

Que si l'on veut pas hériter d'une classe adaptateur. N'oublions pas que *Java* ne permet que l'héritage simple.

```
class SourisAdaptateur extends java.awt.event.MouseAdapter {
  Click2 c;
  public SourisAdaptateur(Click2 c) {
    super();
    this.c = c;
  }
  public void mousePressed(java.awt.event.MouseEvent e) {
    c.b = !c.b;
    c.bouton.setLabel(c.b ? "Cliquez ici" : "ici zeuqilC");
  }
}

public class Click2 extends java.applet.Applet  {
  java.awt.Button bouton;
  boolean b = true;
  public void init() {
    bouton = new java.awt.Button("Cliquez ici");
    add("Center", bouton);
    bouton.addMouseListener(new SourisAdaptateur(this));
  }
}
```

Cette approche est hélas lourde et une meilleur approche consiste à utiliser les classes imbriquées (voir 13).

```
public class Click3 extends java.applet.Applet  {
  java.awt.Button bouton;
  boolean b = true;
  public void init() {
    bouton = new java.awt.Button("Cliquez ici");
    add("Center", bouton);
    bouton.addMouseListener(new SourisAdaptateur());
  }
  class SourisAdaptateur extends java.awt.event.MouseAdapter {
    public void mousePressed(java.awt.event.MouseEvent e) {
      b = !b; bouton.setLabel(b ? "Cliquez ici" : "ici zeuqilC");
    }
  }
}
```

25.2.4 Gestion d'évènements pour composants étendus

Une autre manière de gérer les évènement est fournie dans *AWT*. Il s'agit de de gérer de manière globale les évènements :

- une méthode `protected void processEvent(AWTEvent e)` de la classe `Component` qui en fonction de l'argument `e` capture les évènements.
- des méthodes spécifique selon le type d'évènements :
  - protected void processActionEvent(ActionEvent e)
  - protected void processAdjustmentEvent(AdjustmentEvent e)
  - protected void processComponentEvent(ComponentEvent e)

- protected void processComponentKeyEvent(KeyEvent e)
- protected void processContainerEvent(ContainerEvent e)
- protected void processFocusEvent(FocusEvent e)
- protected void processInputMethodEvent(InputMethodEvent e)
- protected void processItemEvent(ItemEvent e)
- protected void processKeyEvent(KeyEvent e)
- protected void processMouseEvent(MouseEvent e)
- protected void processTextEvent(TextEvent e)
- protected void processWindowEvent(WindowEvent e)

Il est important de noter lorsqu'on redéfinit l'une de ses méthodes dans un container, il ne faut pas oublier d'invoquer la méthode de même nom dans la classe père.

```
protected void processActionEvent(ActionEvent e) {
  // Faire ce qu'il faut
  ...
  super.processActionEvent(e);
}
```

Comme pour les *EventListener*, il faut pour pouvoir gérer les évènements par cette méthode, être à l'écoute des évènements qui nous intéressent ; ce qui se fait en invoquant la méthode `protected final void enableEvents(long typeEvent)`. Le paramètre `typeEvent` est un masque de bits :

- `ACTION_EVENT_MASK`
- `ADJUSTMENT_EVENT_MASK`
- `COMPONENT_EVENT_MASK`
- `CONTAINER_EVENT_MASK`
- `FOCUS_EVENT_MASK`
- `ITEM_EVENT_MASK`
- `KEY_EVENT_MASK`
- `MOUSE_EVENT_MASK`
- `MOUSE_MOTION_EVENT_MASK`
- `TEXT_EVENT_MASK`
- `WINDOW_EVENT_MASK`

```
public class Click4 extends java.applet.Applet  {
   UnButton bouton1;
   UnButton bouton2;
   public void init() {
      bouton1 = new UnButton();
      bouton2 = new UnButton();
      add(bouton1);
      add(bouton2);
   }
}

class UnButton extends java.awt.Button {
   boolean b = true;
   public UnButton() {
      super("Cliquez ici");
      enableEvents(java.awt.AWTEvent.MOUSE_EVENT_MASK);
   }
   protected void processMouseEvent(java.awt.event.MouseEvent e) {
      if (e.getID() == java.awt.event.MouseEvent.MOUSE_CLICKED)  {
         b = !b;
         setLabel(b ? "Cliquez ici" : "ici zeuqilC");
         System.out.println("kjh");
      }
   }
}
```

A quelques rares exceptions près, il n'est généralement pas nécessaire de gérer les évènements de cette manière. Vous trouverez un exemple d'utilisation avec les `PopupMenu` (voir popupEx.

25.2.5 L'interface ActionListener et la classe ActionEvent

L'interface `ActionListener` est l'interface la plus simple d'utilisation. Elle est composée d'une unique méthode `actionPerformed`.

```
Interface java.awt.event.ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```

Cette méthode est invoquée lorsque l'un des évènements suivants survient :

- clic sur un bouton (java.awt.Button)
- double clic sur un élément d'une liste (java.awt.List)
- choix d'un élément d'un menu (java.awt.MenuItem)
- retour chariot sur un champ texte (java.awt.TextField)

Les instances de la classe `java.awt.event.ActionEvent` contiennent des informations sur l'évènement.

```
public class java.awt.event.ActionEvent extends java.awt.AWTEvent {
  public static final int SHIFT_MASK
  public static final int CTRL_MASK
  public static final int META_MASK
  public static final int ALT_MASK
  public static final int ACTION_FIRST
  public static final int ACTION_LAST
  public static final int ACTION_PERFORMED
  public ActionEvent(Object source, int id, String command)
  public ActionEvent(Object source, int id, String command, int modifiers)
  public String getActionCommand()
  public int getModifiers()
  public String paramString()
}
```

`public String getActionCommand ()`

Retourne la chaîne de caractères associées à l'évènement.

`public String getModifiers ()`

Retourne un entier représentant le ou les “modifier keys” enfoncés lors de la survenue de l'évènement.

A terminer

`public String paramString ()`

25.2.6 Les interfaces MouseListener, MouseMotionListener et la classe MouseEvent

Les interfaces `MouseListener` et `MouseMotionListener` permettent la gestion de la souris pour un composant donnée. Pour des raisons d'efficacité (la gestion du déplacement de la souris étant plus lourde que la simple gestion des clics), la gestion de la souris est divisée en deux :

- l'interface `MouseListener` s'occupe de la position par rapport au composant et des actions simples de la souris (clic)

```
Interface java.awt.event.MouseListener extends EventListener {
 public void mouseClicked(MouseEvent e)
 public void mousePressed(MouseEvent e)
 public void mouseReleased(MouseEvent e)
 public void mouseEntered(MouseEvent e)
 public void mouseExited(MouseEvent e)
}
```

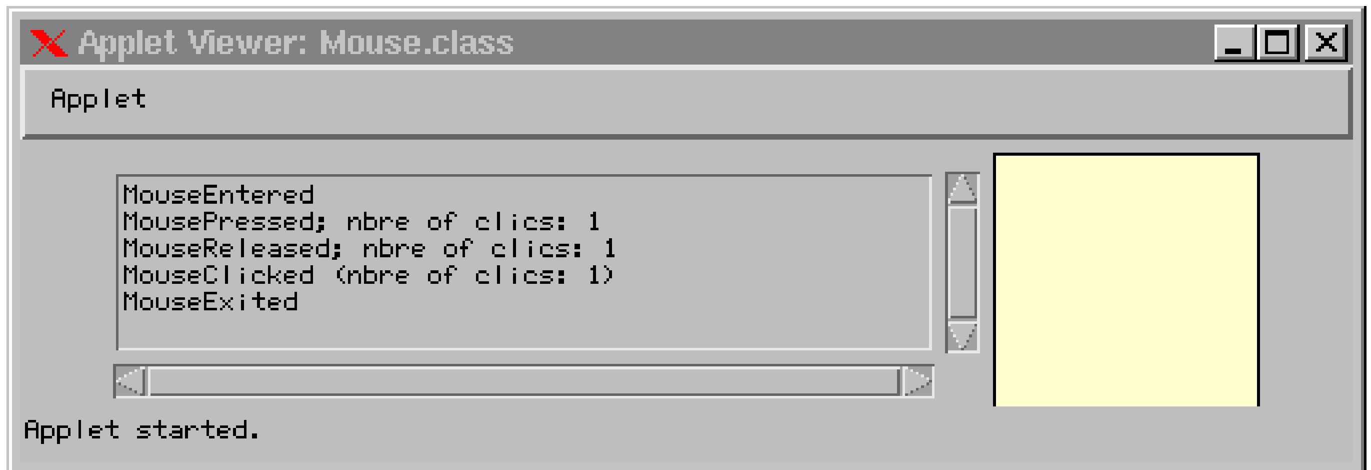

FIG. 25.3: MouseListener

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;

public class Mouse extends Applet implements MouseListener {
  TextArea textArea;
  ZoneVierge blankArea;
  public void init() {
    textArea = new TextArea(5, 50);
    textArea.setEditable(false);
    add(textArea);
    blankArea = new ZoneVierge(new Color(0.98f, 0.97f, 0.85f));
    add(blankArea);
    blankArea.addMouseListener(this);
  }
  public void mousePressed(MouseEvent e) {
    afficher("MousePressed; nbre of clics: " + e.getClickCount(), e);
  }
  public void mouseReleased(MouseEvent e) {
    afficher("MouseReleased; nbre of clics: " + e.getClickCount(), e);
  }
  public void mouseEntered(MouseEvent e) {
    afficher("MouseEntered", e);
  }
  public void mouseExited(MouseEvent e) {
    afficher("MouseExited", e);
  }
  public void mouseClicked(MouseEvent e) {
    afficher("MouseClicked (nbre of clics: " + e.getClickCount() + ")", e);
  }
  void afficher(String eventDescription, MouseEvent e) {
    textArea.append(eventDescription + System.getProperty("line.separator"));
    textArea.setCaretPosition(2000);
  }
}
```

- l'interface `MouseMotionListener` s'occupe la gestion du déplacement de la souris

```
Interface java.awt.event.MouseMotionListener extends EventListener {
    public void mouseDragged(MouseEvent e)
    public void mouseMoved(MouseEvent e)
}
```

Toutes ces méthodes ont un unique paramètre qui une instance de la classe `MouseEvent`.

```
public class java.awt.event.MouseEvent extends  java.awt.InputEvent {
    public static final int MOUSE_FIRST
    public static final int MOUSE_LAST
```


www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

```
        public static final int MOUSE_CLICKED
        public static final int MOUSE_PRESSED
        public static final int MOUSE_RELEASED
        public static final int MOUSE_MOVED
        public static final int MOUSE_ENTERED
        public static final int MOUSE_EXITED
        public static final int MOUSE_DRAGGED

        public MouseEvent(Component source, int id, long when, int modifiers,
                      int x, int y, int clickCount, boolean popupTrigger)

        public int getX()
        public int getY()
        public Point getPoint()
        public void translatePoint(int x, int y)
        public int getClickCount()
        public boolean isPopupTrigger()
        public String paramString()
    }
```

`public int getX ()`

Retourne l'abscisse de la position (relativement au composant) de l'évènement.

`public int getY ()`

Retourne l'ordonnée de la position (relativement au composant) de l'évènement.

`public Point getPoint ()`

Retourne la position (relativement au composant) de l'évènement.

A terminer `public void translatePoint (int x, int y)`

`public int getClickCount ()`

Retourne le nombre de clics survenus.

A terminer `public boolean isPopupTrigger ()`

Cette applet est une feuille de dessin simplifiée pour faire des dessins à main levée avec la souris. On reparlera de cet applet dans le chapitre 28.

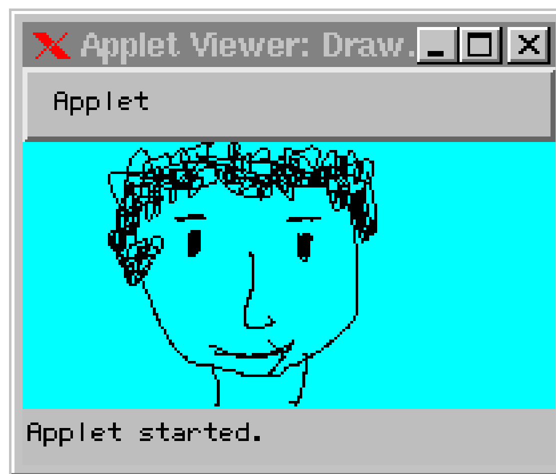

FIG. 25.4: Feuille de dessin

```
public class Draw extends java.applet.Applet {
  int x=0, y=0;
  public Draw() {
    super();
    setBackground(java.awt.Color.cyan);
    addMouseListener(new java.awt.event.MouseAdapter() {
      public void mousePressed(java.awt.event.MouseEvent e) {
```

```
        x = e.getX(); y = e.getY();
      }
    });
    addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
      public void mouseDragged(java.awt.event.MouseEvent e) {
        java.awt.Graphics g = getGraphics();
        g.drawLine(x, y, e.getX(), e.getY());
        x = e.getX(); y = e.getY();
      }
    });
  }
}
```

Remarque sur le DnD

25.2.7 L'interface KeyListener et la classe KeyEvent

L'interface `KeyListener` permette la gestion du clavier pour un composant donnée. Les évènements clavier surviennent lorsque l'utilisateur effectue une frappe clavier. L'évènement est reçu par le composant en premier plan (celui qui a le *focus*).

```
Interface java.awt.event.KeyListener extends EventListener {
    public void keyTyped(KeyEvent e)
    public void keyPressed(KeyEvent e)
    public void keyReleased(KeyEvent e)}
```

Toutes ces méthodes ont un unique paramètre qui une instance de la classe `KeyEvent`.

```
public class java.awt.event.KeyEvent extends  java.awt.InputEvent {
    public static final int KEY_FIRST, KEY_LAST, KEY_TYPED, KEY_PRESSED
    public static final int KEY_RELEASED, VK_ENTER, VK_BACK_SPACE, VK_TAB
    public static final int VK_CANCEL, VK_CLEAR, VK_SHIFT, VK_CONTROL, VK_ALT
    public static final int VK_PAUSE, VK_CAPS_LOCK, VK_ESCAPE, VK_SPACE
    public static final int VK_PAGE_UP, VK_PAGE_DOWN, VK_END, VK_HOME
    public static final int VK_LEFT, VK_UP, VK_RIGHT, VK_DOWN
    public static final int VK_COMMA, VK_PERIOD, VK_SLASH
    public static final int VK_0, ... VK_9;
    public static final int VK_SEMICOLON, VK_EQUALS
    public static final int VK_A, ..., VK_Z
    public static final int VK_OPEN_BRACKET, VK_BACK_SLASH, VK_CLOSE_BRACKET
    public static final int VK_NUMPAD0, ..., VK_NUMPAD9
    public static final int VK_MULTIPLY, VK_ADD, VK_SEPARATER, VK_SUBTRACT, VK_DECIMAL, VK_DIVIDE
    public static final int VK_F1, ..., VK_F12
    public static final int VK_DELETE, VK_NUM_LOCK, VK_SCROLL_LOCK
    public static final int VK_PRINTSCREEN, VK_INSERT, VK_HELP, VK_META
    public static final int VK_BACK_QUOTE, VK_QUOTE, VK_KP_UP, VK_KP_DOWN
    public static final int VK_KP_LEFT, VK_KP_RIGHT, VK_DEAD_GRAVE, VK_DEAD_ACUTE
    public static final int VK_DEAD_CIRCUMFLEX, VK_DEAD_TILDE, VK_DEAD_MACRON, VK_DEAD_BREVE
    public static final int VK_DEAD_ABOVEDOT, VK_DEAD_DIAERESIS, VK_DEAD_ABOVERING
    public static final int VK_DEAD_DOUBLEACUTE, VK_DEAD_CARON, VK_DEAD_CEDILLA
    public static final int VK_DEAD_OGONEK, VK_DEAD_IOTA, VK_DEAD_VOICED_SOUND
    public static final int VK_DEAD_SEMIVOICED_SOUND, VK_AMPERSAND, VK_ASTERISK
    public static final int VK_QUOTEDBL, VK_LESS, VK_GREATER, VK_BRACELEFT
    public static final int VK_BRACERIGHT, VK_FINAL, VK_CONVERT, VK_NONCONVERT
    public static final int VK_ACCEPT, VK_MODECHANGE, VK_KANA, VK_KANJI, VK_UNDEFINED
    public static final char CHAR_UNDEFINED

    public KeyEvent(Component source, int id, int modifiers, int keyCode, char keyChar)
    public KeyEvent(Component source, int id, long when, int modifiers, int keyCode)

    public int getKeyCode()
    public void setKeyCode(int keyCode)
    public void setKeyChar(char keyChar)
    public void setModifiers(int modifiers)
    public char getKeyChar()
    public static String getKeyText(int keyCode)
    public static String getKeyModifiersText(int modifiers)
    public boolean isActionKey()
    public String paramString()
}
```

`public char  getKeyChar ()`

A terminer

`public static String  getKeyText (int keyCode)`

A terminer

`public static String getKeyModifiersText (int modifiers)` A terminer

A terminer `public boolean isActionKey ()`

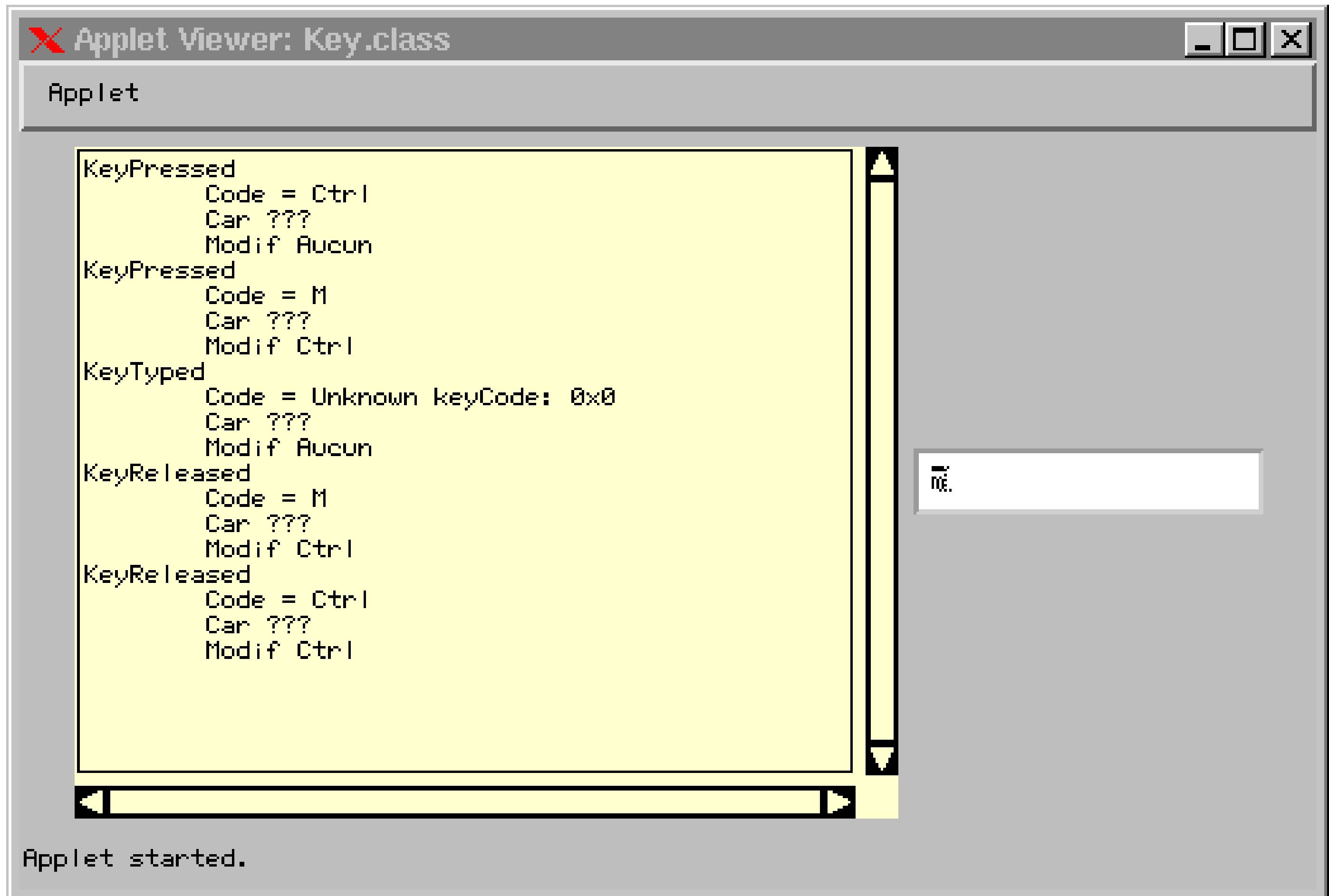

FIG. 25.5: KeyListener

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;

public class Key extends Applet implements KeyListener {
  TextArea textArea;
  TextField textField;
  String rc = System.getProperty("line.separator");
  public void init() {
    textArea = new TextArea(20, 50);
    textArea.setEditable(false);
    textArea.setBackground(new Color(0.98f, 0.97f, 0.85f));
    add(textArea);
    textField = new TextField(20);
    add(textField);
    textField.addKeyListener(this);
  }
  public void keyTyped(KeyEvent e) { afficher("KeyTyped " + rc + decoder(e)); }
```

```
    public void keyPressed(KeyEvent e) { afficher("KeyPressed " + rc + decoder(e)); }
    public void keyReleased(KeyEvent e) { afficher("KeyReleased " + rc + decoder(e)); }
    String  decoder(KeyEvent e) {
      char car = e.getKeyChar();
      String carStr = Character.isISOControl(car) ? "???" : "'"+ car + "'";
      int code  = e.getKeyCode();
      String codeStr = KeyEvent.getKeyText(code);
      int modif = e.getModifiers();
      String modifStr = KeyEvent.getKeyModifiersText(modif);
      if (modifStr.length() == 0)  modifStr = "Aucun";
      return  "\tCode = " + codeStr  + rc + "\tCar " + carStr  + rc + "\tModif " + modifStr ;
    }
    void afficher(String s) {
      textArea.append(s + rc);
      textArea.setCaretPosition(2000);
    }
  }
```

25.2.8 L'interface WindowListener et la classe WindowEvent

Les évènements de l'interface `WindowListener` sont générés lorsqu'une fenêtre s'ouvre (`windowOpened`), se ferme (`windowClosing` et `windowClosed`), s'iconifie (`windowIconified`), se déiconifie (`windowDeiconified`), devient active (`windowActivated`) ou inactive (`windowDeactivated`).

```
Interface java.awt.event.WindowListener extends EventListener {
    public void windowOpened(WindowEvent e)
    public void windowClosing(WindowEvent e)
    public void windowClosed(WindowEvent e)
    public void windowIconified(WindowEvent e)
    public void windowDeiconified(WindowEvent e)
    public void windowActivated(WindowEvent e)
    public void windowDeactivated(WindowEvent e)
}
```

Toutes ces méthodes ont un unique paramètre qui une instance de la classe `WindowEvent`.

```
public class java.awt.event.WindowEvent extends  java.awt.ComponentEvent {
    public static final int WINDOW_FIRST
    public static final int WINDOW_LAST
    public static final int WINDOW_OPENED
    public static final int WINDOW_CLOSING
    public static final int WINDOW_CLOSED
    public static final int WINDOW_ICONIFIED
    public static final int WINDOW_DEICONIFIED
    public static final int WINDOW_ACTIVATED
    public static final int WINDOW_DEACTIVATED

    public WindowEvent(Window source, int id)
    public Window getWindow()
    public String paramString()
}
```

`public  WindowEvent ()`

A terminer

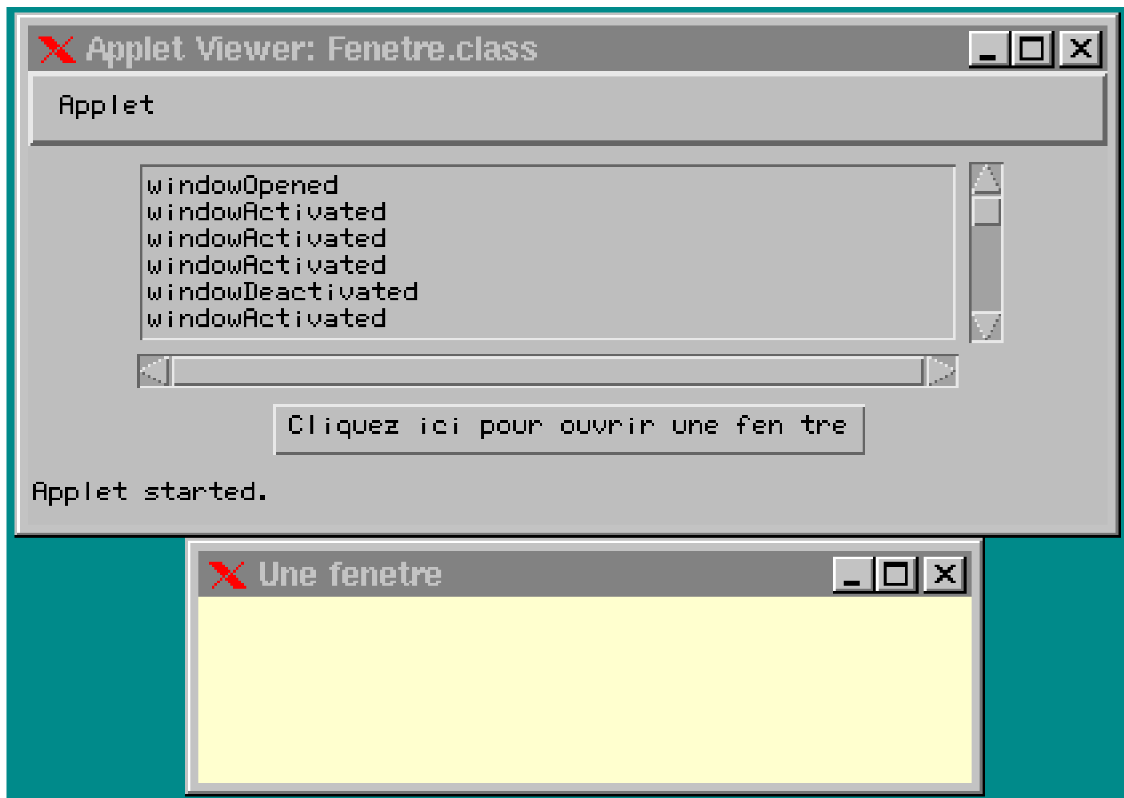

Fig. 25.6: WindowListener

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Fenetre extends Applet implements  WindowListener, ActionListener {
  java.awt.Button b;
  TextArea textArea;
  Frame fenetre;
  String rc = System.getProperty("line.separator");
  public void init() {
    textArea = new TextArea(5, 50);
    textArea.setEditable(false);
    add(textArea);
    b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
    add(b);
    b.addActionListener(this);

    fenetre = new Frame("Une fenetre");
    fenetre.setSize(300, 100);
    fenetre.setBackground(new Color(0.98f, 0.97f, 0.85f));
    fenetre.addWindowListener(this);
  }
  void afficher(String s) {
    textArea.append(s + rc);
    textArea.setCaretPosition(2000);
  }
  public void actionPerformed(ActionEvent e) {
    if (! fenetre.isShowing()) fenetre.setVisible(true);
    else fenetre.setVisible(false);
  }
  public void windowClosing(WindowEvent e) {
    afficher("windowClosing");
    fenetre.setVisible(false);
  }
```

```
    public void windowOpened(WindowEvent e)      { afficher("windowOpened");      }
    public void windowIconified(WindowEvent e)   { afficher("windowIconified");   }
    public void windowDeiconified(WindowEvent e) { afficher("windowDeIconified"); }
    public void windowActivated(WindowEvent e)   { afficher("windowActivated");   }
    public void windowDeactivated(WindowEvent e) { afficher("windowDeactivated"); }
    public void windowClosed(WindowEvent e)      {  }
}
```

25.2.9 L'interface ComponentListener et la classe ComponentEvent

L'interface `ComponentListener` et la classe `ComponentAdapter` permettent de gérer les évènements concernant les caractéristiques d'un composant : taille, déplacement et visibilité. Comme d'habitude, un composant qui veut gérer ces évènements s'inscrit avec la méthode **`addComponentListener`**.

```
Interface java.awt.event.ComponentListener extends EventListener {
  public  void componentResized(ComponentEvent e)
  public  void componentMoved(ComponentEvent e)
  public  void componentShown(ComponentEvent e)
  public  void componentHidden(ComponentEvent e)
}
```

Toutes ces méthodes ont un unique paramètre qui une instance de la classe `ComponentEvent`.

```
public class java.awt.event.ComponentEvent extends  java.awt.AWTEvent {
  public static final int COMPONENT_FIRST
  public static final int COMPONENT_LAST
  public static final int COMPONENT_MOVED
  public static final int COMPONENT_RESIZED
  public static final int COMPONENT_SHOWN
  public static final int COMPONENT_HIDDEN
  public ComponentEvent(Component source, int id)
  public Component getComponent()
  public String paramString()
}
```

`public  ComponentEvent ()`

A terminer

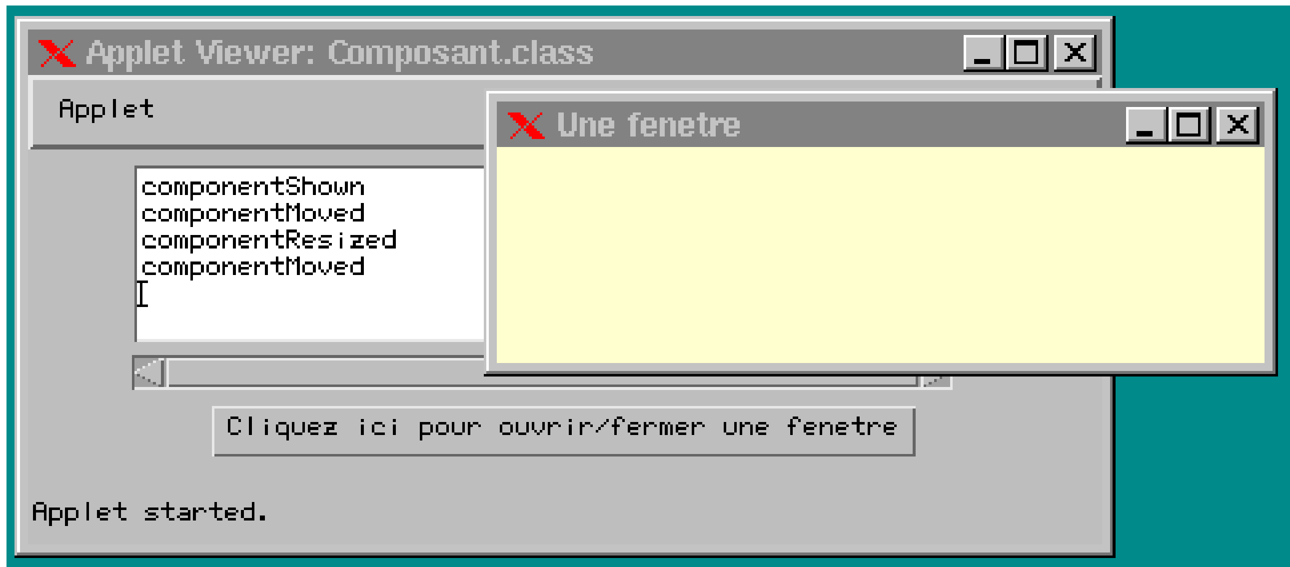

FIG. 25.7: ComponentListener

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Composant extends Applet implements   WindowListener, ComponentListener, ActionListener {
```

```
  java.awt.Button b;
  TextArea textArea;
  Frame fenetre;
  String rc = System.getProperty("line.separator");
  public void init() {
    textArea = new TextArea(5, 50);
    textArea.setEditable(true);
    add(textArea);
    b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
    add(b);
    b.addActionListener(this);
    fenetre = new Frame("Une fenetre");
    fenetre.setSize(300, 100);
    fenetre.setBackground(new Color(0.98f, 0.97f, 0.85f));
    fenetre.addWindowListener(this);
    fenetre.addComponentListener(this);
  }
  void afficher(String s) {
    textArea.append(s + rc);
    textArea.setCaretPosition(2000);
  }
  public void actionPerformed(ActionEvent e) {
    if (! fenetre.isShowing()) fenetre.setVisible(true);
    else fenetre.setVisible(false);
  }
  public void windowClosing(WindowEvent e) { fenetre.setVisible(false); }
  public void windowOpened(WindowEvent e)      { }
  public void windowIconified(WindowEvent e)   { }
  public void windowDeiconified(WindowEvent e) { }
  public void windowActivated(WindowEvent e)   { }
  public void windowDeactivated(WindowEvent e) { }
  public void windowClosed(WindowEvent e)      { }

  public  void componentResized(ComponentEvent e)  { afficher("componentResized");  }
  public  void componentMoved(ComponentEvent e)    { afficher("componentMoved");    }
  public  void componentShown(ComponentEvent e)    { afficher("componentShown");    }
  public  void componentHidden(ComponentEvent e)   { afficher("componentHidden");   }
}
```

25.2.10 L'interface ContainerListener et la classe ContainerEvent

Un composant peut être à l'écoute de l'ajout (`add`) et/ou de la suppression (`remove`) d'autre composants.

```
Interface java.awt.event.ContainerListener extends EventListener {
    public void componentAdded(ContainerEvent e)
    public void componentRemoved(ContainerEvent e)
}

public class java.awt.event.ContainerEvent extends  java.awt.ComponentEvent {
    public static final int CONTAINER_FIRST
    public static final int CONTAINER_LAST
    public static final int COMPONENT_ADDED
    public static final int COMPONENT_REMOVED

    public ContainerEvent(Component source, int id, Component child)
    public Container getContainer()
    public Component getChild()
    public String paramString()
}
```

A terminer

```
public  ContainerEvent ()
```

FIG. 25.8: ContainerListener

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Container extends Applet implements ActionListener, ContainerListener {
  private static final int MAX = 10;
  java.awt.Button ajout, suppression;
  TextArea textArea;
  String rc = System.getProperty("line.separator");
  Button [] tab = new Button[MAX];
  int sommet = 0;
  public void init() {
    textArea = new TextArea(5, 50);
    textArea.setEditable(false);
    add(textArea);
    ajout = new Button("Ajouter un composant");
    ajout.setActionCommand("Ajouter");
    add(ajout);
    ajout.addActionListener(this);
    suppression = new Button("Supprimer un composant");
    suppression.setActionCommand("Supprmier");
    add(suppression);
    suppression.addActionListener(this);
    addContainerListener(this);
  }
  void afficher(String s) {
    textArea.append(s + rc);
    textArea.setCaretPosition(2000);
  }
  public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand() == "Ajouter") {
      if (sommet < MAX) { add(tab[sommet++] = new Button("B"+(sommet+1))); validate(); }
    }
    else if (sommet > 0) { remove(tab[--sommet]); tab[sommet] = null; validate(); }
  }
  public void componentAdded(ContainerEvent e) { afficher("On ajoute le bouton " + e.getChild().getName()); }
  public void componentRemoved(ContainerEvent e) { afficher("On supprime le bouton " + e.getChild().getName()); }
}
```

25.2.11 L'interface FocusListener et la classe FocusEvent

Ces évènements sont générés lorsqu'un composant devient actif (focus) ou passif.

```
Interface java.awt.event.FocusListener extends EventListener {
    public void focusGained(FocusEvent e)
    public void focusLost(FocusEvent e)
}
```

```
public class java.awt.event.FocusEvent extends  java.awt.ComponentEvent {
   public static final int FOCUS_FIRST
   public static final int FOCUS_LAST
   public static final int FOCUS_GAINED
   public static final int FOCUS_LOST
   public FocusEvent(Component source, int id, boolean temporary)
   public FocusEvent(Component source, int id)
   public boolean isTemporary()
   public String paramString()
}
```

A terminer

```
public  FocusEvent ()
```

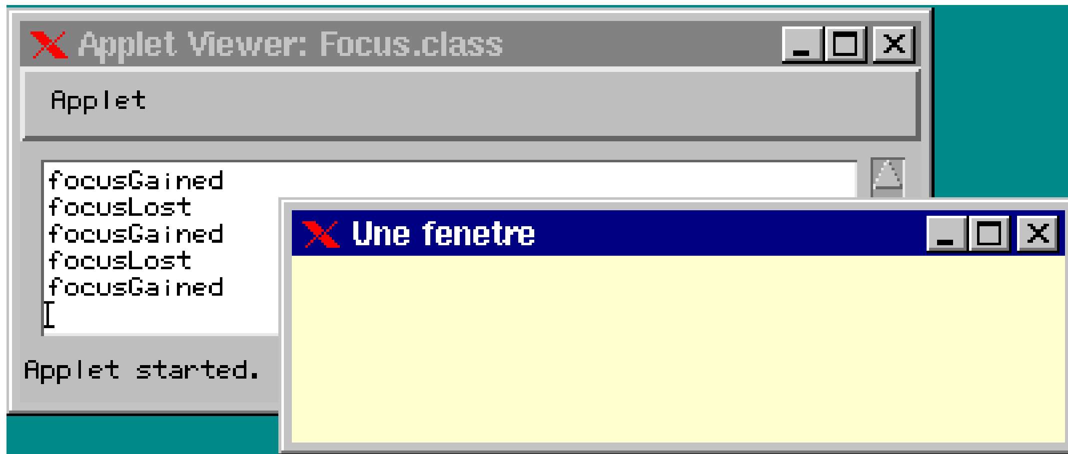

Fig. 25.9: FocusListener

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Focus extends Applet implements FocusListener, ActionListener {
  TextArea textArea;
  Button b;
  Frame fenetre;
  String rc = System.getProperty("line.separator");
  public void init() {
    textArea = new TextArea(5, 50);
    textArea.setEditable(true);
    add(textArea);
    b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
    add(b);
    b.addActionListener(this);
    fenetre = new Frame("Une fenetre");
    fenetre.setSize(300, 100);
    fenetre.setBackground(new Color(0.98f, 0.97f, 0.85f));
    fenetre.setVisible(false);
    fenetre.addFocusListener(this);
    fenetre.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { fenetre.setVisible(false); }
    });
  }
  public void actionPerformed(ActionEvent e) {
    if (! fenetre.isShowing()) fenetre.setVisible(true);
    else fenetre.setVisible(false);
  }
  void afficher(String s) {
```

```
      textArea.append(s + rc);
      textArea.setCaretPosition(2000);
    }
    public void focusGained(FocusEvent e) { afficher("focusGained "); }
    public void focusLost(FocusEvent e) { afficher("focusLost "); }
  }
```

25.2.12 L'interface TextListener et la classe TextEvent

Les évènements de l'interface **TextListener** sont générés lorsqu'un composant de type texte (**TextArea** et **TextField**) subissent des changements.

```
Interface java.awt.event.TextListener extends EventListener {
public void textValueChanged(TextEvent e)
}
```

```
public class java.awt.event.TextEvent extends  java.awt.AWTEvent {
public static final int TEXT_FIRST
public static final int TEXT_LAST
public static final int TEXT_VALUE_CHANGED
public TextEvent(Object source, int id)
public String paramString()
}
```

`public  TextEvent ()`

A terminer

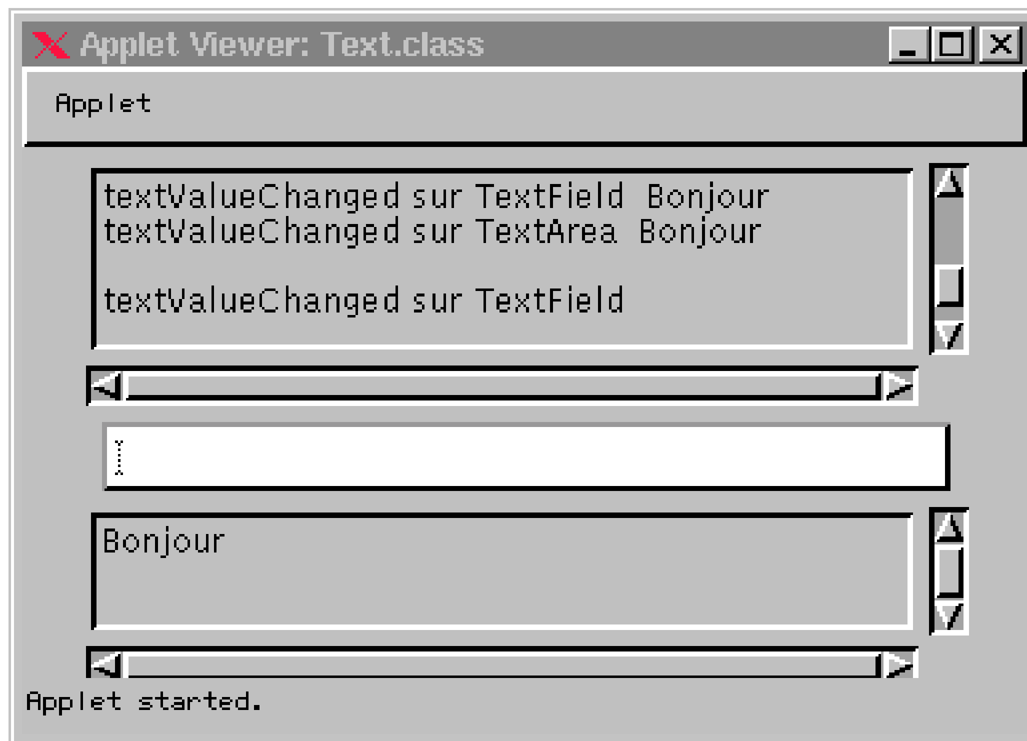

FIG. 25.10: TextListener

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Text extends Applet implements TextListener, ActionListener  {
  TextArea textArea;
  TextArea TA;
  TextField TF;
  String rc = System.getProperty("line.separator");
```

```
  public void init() {
    textArea = new TextArea(5, 50);
    textArea.setEditable(false);
    add(textArea);
    TF = new TextField(50);
    TF.setEditable(true);
    add(TF);
    TA = new TextArea(3, 50);
    TA.setEditable(false);
    add(TA);
    TA.addTextListener(this);
    TF.addTextListener(this);
    TF.addActionListener(this);
  }
  void afficher(String s) {
    textArea.append(s + rc);
    textArea.setCaretPosition(2000);
  }
  public void textValueChanged(TextEvent e)  {
    TextComponent tc = (TextComponent)e.getSource();
    String s = tc.getText();
    afficher("textValueChanged sur " + ((tc==TA) ? "TextArea " : "TextField ") + " " + s);
  }
  public void actionPerformed(ActionEvent e) {
    TextComponent tc = (TextComponent)e.getSource();
    String s = tc.getText();
    if (s.equals("")) return;
    TA.append(s + rc);
    TA.setCaretPosition(2000);
    TF.setText("");
  }
}
```

25.2.13 L'interface ItemListner et la classe ItemEvent

Les évènements de l'interface `ItemListener` sont générés par les composant qui implantent l'interface `ItemSelectable` (`Checkbox`, `MenuItem`, `Choice` et `list`)

```
Interface java.awt.event.ItemListener  {
public void itemStateChanged(ItemEvent e)
}
```

```
public class java.awt.event.ItemEvent extends  java.awt.AWTEvent {
public static final int ITEM_FIRST
public static final int ITEM_LAST
public static final int ITEM_STATE_CHANGED
public static final int SELECTED
public static final int DESELECTED
public ItemEvent(ItemSelectable source, int id, Object item, int stateChange)
public ItemSelectable getItemSelectable()
public Object getItem()
public int getStateChange()
public String paramString()
}
```

A terminer

```
public  ItemEvent ()
```

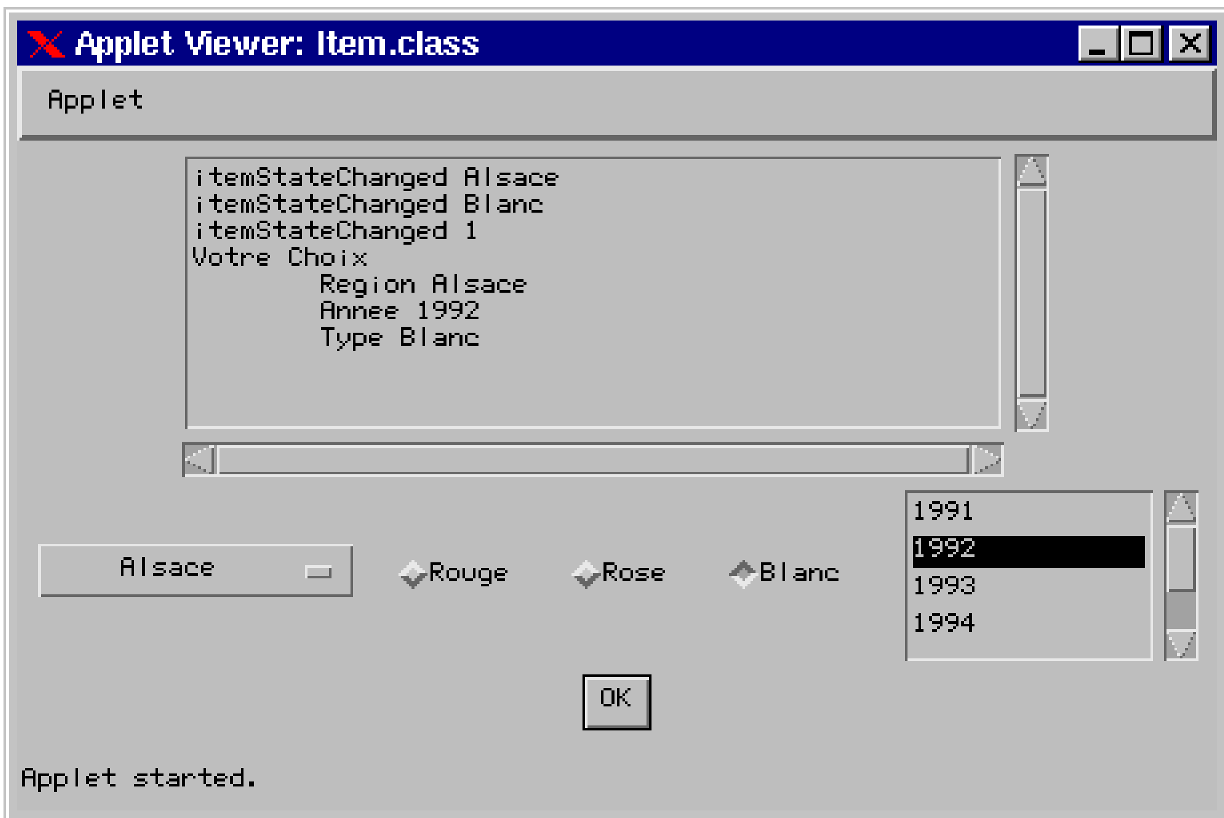

Fig. 25.11: ItemListener

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Item extends Applet implements ActionListener  {
  TextArea textArea;
  String rc = System.getProperty("line.separator");
  MesChoix meschoix;
  MesCases mescases;  MaListe maliste;
  public void init() {
    textArea = new TextArea(8, 50);
    textArea.setEditable(false);
    add(textArea);
    meschoix = new  MesChoix(this);
    mescases = new MesCases(this);
    maliste = new MaListe(this);
    Button b = new Button("OK");
    add(meschoix); add(mescases);  add(maliste); add(b);
    b.addActionListener(this);
  }
  void afficher(String s) { textArea.append(s + rc); textArea.setCaretPosition(2000); }
  public void actionPerformed(ActionEvent e) {
    afficher("Votre Choix ");
    afficher("\tRegion " + meschoix.getSelectedItem());
    afficher("\tAnnee " + maliste.getSelectedItem());
    afficher("\tType " + mescases.getSelectedCheckbox());
  }
}
class MesChoix extends Choice implements ItemListener  {
  Item t;
  public MesChoix(Item t) {
    super();
    this.t = t;
```

```
      addItem("Bordeaux"); addItem("Bourgogne"); addItem("Alsace"); addItem("Cotes du Rhones");
      addItemListener(this);
    }
    public void itemStateChanged(ItemEvent e)  { t.afficher("itemStateChanged " + e.getItem()); }
  }
  class MaListe extends List implements ItemListener  {
    Item t;
    MaListe(Item t) {
      super();
      this.t = t;
      add("1991"); add("1992"); add("1993"); add("1994"); add("1995"); add("1996");
      addItemListener(this);
    }
    public void itemStateChanged(ItemEvent e)  { t.afficher("itemStateChanged " + e.getItem()); }
  }
  class MesCases extends Container implements ItemListener {
    Item t;
    CheckboxGroup cbg;
    public MesCases(Item t) {
      super();
      this.t = t;
      cbg = new CheckboxGroup();
      Checkbox c1 = new  Checkbox("Rouge", cbg, true);
      Checkbox c2 = new  Checkbox("Rose", cbg, false);
      Checkbox c3 = new  Checkbox("Blanc", cbg, false);
      add(c1); add(c2); add(c3);
      c1.addItemListener(this); c2.addItemListener(this); c3.addItemListener(this);
    }
    public String getSelectedCheckbox() { return cbg.getSelectedCheckbox().getLabel(); }
    public void itemStateChanged(ItemEvent e)  { t.afficher("itemStateChanged " + e.getItem()); }
  }
```

25.2.14 L'interface AdjustmentListener et la classe AdjustmentEvent

L'interface **AdjustmentListener** est celle qui permet la gestion des ascenseurs (java.awt.ScrollBar).

```
Interface java.awt.event.AdjustmentListener {
public  void adjustmentValueChanged(AdjustmentEvent e)
}

public class java.awt.event.ActionEvent extends java.awt.AWTEvent {
  public static final int ADJUSTMENT_FIRST
  public static final int ADJUSTMENT_LAST
  public static final int ADJUSTMENT_VALUE_CHANGED
  public static final int UNIT_INCREMENT
  public static final int UNIT_DECREMENT
  public static final int BLOCK_DECREMENT
  public static final int BLOCK_INCREMENT
  public static final int TRACK
  public AdjustmentEvent(Adjustable source, int id, int type, int value)
  public Adjustable getAdjustable()
  public int getValue()
  public int getAdjustmentType()
  public String paramString()
}
```

A terminer `public  AdjustmentEvent (Adjustable source, int id, int type, int value)`

`public Adjustable  getAdjustable ()`

Retourne la source de l'évènement ; remplace la méthode **getSource**.

`public int  getAdjustmentType ()`

Retourne le type de l'évènement qui est l'un des constantes : **UNIT_INCREMENT**, **UNIT_DECREMENT**, **BLOCK_INCREMENT**, **BLOCK_DECREMENT**, **TRACK**.

`public int  getValue ()`

Rtourne la valeur du composant après que l'évènement se soit produit.

A terminer `public String  paramString ()`

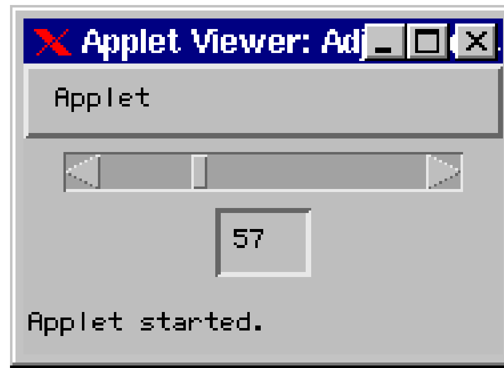

FIG. 25.12: AdjustmentListener

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Adjustment extends Applet implements AdjustmentListener {
  TextField tf;
  SC scb;
  public void init() {
    scb = new SC();
    tf = new TextField(3);
    add(scb); add(tf);
    scb.addAdjustmentListener(this);
    tf.setEditable(false);
    tf.setText("50");
  }
  public  void adjustmentValueChanged(AdjustmentEvent e) { tf.setText(Integer.toString(e.getValue())); }
}

class SC extends Scrollbar {
  Dimension minSize;
  public SC() {
    super(Scrollbar.HORIZONTAL, 50, 1, 0, 200);
    minSize = new Dimension(150, 15);
  }
  public Dimension getPreferredSize() { return getMinimumSize(); }
  public synchronized Dimension getMinimumSize() { return minSize; }
}
```

25.2.15 L'interface InputMethodListener et la classe InputMethodEvent

a partir de 1.2

L'interface **InputMethodListener** se charge de la gestion de l'édition des composants de type texte (**TextField** et **TextArea**).

```
Interface java.awt.event.InputMethodListener extends EventListener {
    public void inputMethodTextChanged(InputMethodEvent event)
    public void caretPositionChanged(InputMethodEvent event)
}
```

```
public class java.awt.event.InputMethodEvent extends  java.awt.AWTEvent {
public static final int INPUT_METHOD_FIRST
public static final int INPUT_METHOD_TEXT_CHANGED
public static final int CARET_POSITION_CHANGED
public static final int INPUT_METHOD_LAST
public static final int ALL_CHARACTERS_COMMITTED
public InputMethodEvent(Component source, int id,
                        AttributedCharacterIterator text, int committedCharacterCount,
                        TextHitInfo caret, TextHitInfo visiblePosition)
public InputMethodEvent(Component source, int id, TextHitInfo caret,
                        TextHitInfo visiblePosition)
public AttributedCharacterIterator getText()
public int getCommittedCharacterCount()
public TextHitInfo getCaret()
public TextHitInfo getVisiblePosition()
public void consume()
```

```
public boolean isConsumed()
}
```

A terminer

```
public  InputMethodEvent ()
```

25.3 Récapitulatif des évènements associés aux composants

	Action	Adjust.	Compon.	Contain.	Focus	Item	Key	Mouse	MouseMot.	Text	Window
Button	X		X		X		X	X	X		
Canvas			X		X		X	X	X		
Checkbox			X		X	X	X	X	X		
CheckboxMenuItem						X					
Choice			X		X	X	X	X	X		
Component			X		X		X	X	X		
Container			X	X	X		X	X	X		
Dialog			X	X	X		X	X	X		X
Frame			X	X	X		X	X	X		X
Label			X		X		X	X	X		
List	X		X		X	X	X	X	X		
MenuItem	X										
Panel			X	X	X		X	X	X		
Scrollbar		X	X		X		X	X	X		
ScrollPane			X	X	X		X	X	X		
TextArea			X		X		X	X	X	X	
TextComponent			X		X		X	X	X	X	
TextField	X		X		X		X	X	X	X	
Window			X	X	X		X	X	X		X

A TERMINER

25.3.1 Suppression de la gestion des évènements

A TERMINER

25.3.2 La queue d'évènements

26. Les widgets

Sommaire

26.1 Introduction . . . 215
26.1.1 Component et Container . . . 216
26.1.2 Affichage . . . 217
26.1.3 Gestion de la position et des dimensions . . . 217
26.1.4 Un petit exemple pour commencer . . . 218
26.1.5 Applets et applications autonomes . . . 219
26.1.6 Architecture générale . . . 219
26.1.7 La classe Component . . . 220
26.1.8 La classe Container . . . 222
26.2 Button . . . 222
26.3 Frame . . . 223
26.4 Panel . . . 224
26.5 Label . . . 225
26.6 TextField et TextArea . . . 226
26.7 Checkbox et CheckboxGroup . . . 229
26.7.1 Checkbox . . . 229
26.7.2 CheckboxGroup . . . 231
26.8 Choice . . . 232
26.9 List . . . 233
26.10Canvas . . . 235
26.11Menu . . . 235
26.12Scrollbar . . . 239
26.13Dialog . . . 240
26.14FileDialog . . . 242
26.15ScrollPane . . . 244
26.16Composants légers (Ligthweight) . . . 244

26.1 Introduction

Java possède plusieurs types de widgets :

Button : un bouton à cliquer Nous avons déjà rencontré ce type de widget ; il s'agit d'un bouton etiquetté. Un bouton émet le signal lorsque l'utilisateur presse sur le bouton.

Checkbox : bouton à cocher etiquetté.

Choices : menu déroulant.

List : menu défilant.

MenuItem : items de la barre de menu.

Canvas : zone vierge sur laquelle on peut dessiner ou inclure d'autres widgets.

Scrollbar : Asscenceurs et glissières

TextField : zone de texte sur une ligne

TextArea : zone de texte multi lignes

etc.

Avant d'examiner en détail chacun de ses widgets, voici un exemple, provenant du tutorial de *jdk*, réunissant une certain nombre de widgets.

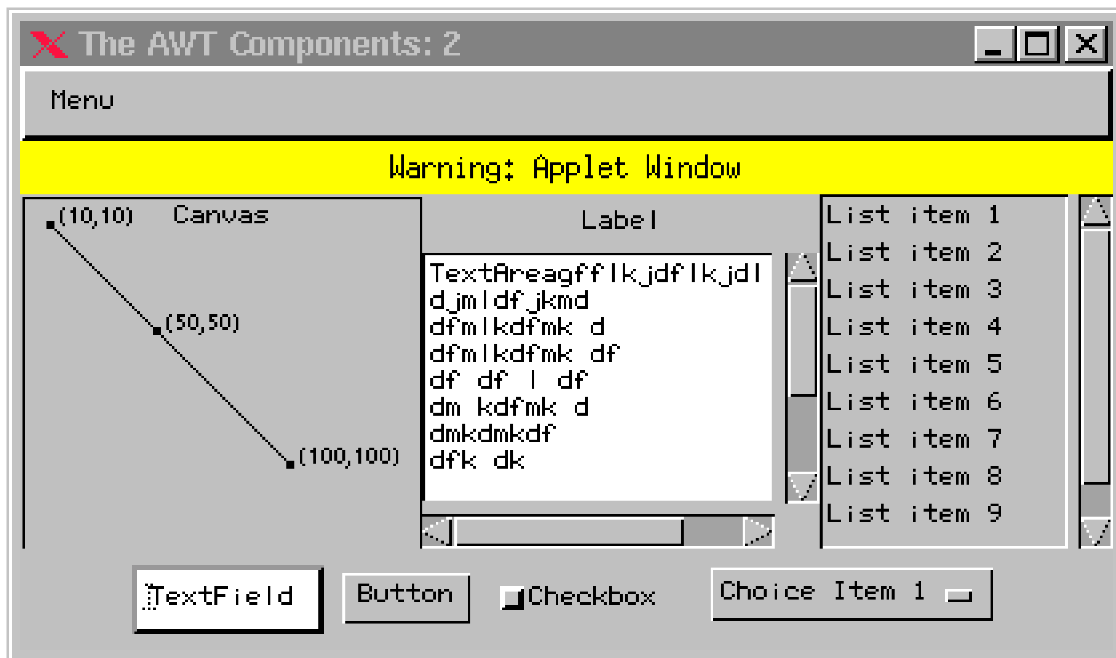

Fig. 26.1: Quelques widgets

26.1.1 Component et Container

Tous les widgets que nous pouvons utiliser sont toutes dérivée de la classe abstraite `Component`. Par exemple, la classe `Button` est une classe dérivée de la classe `Component`.

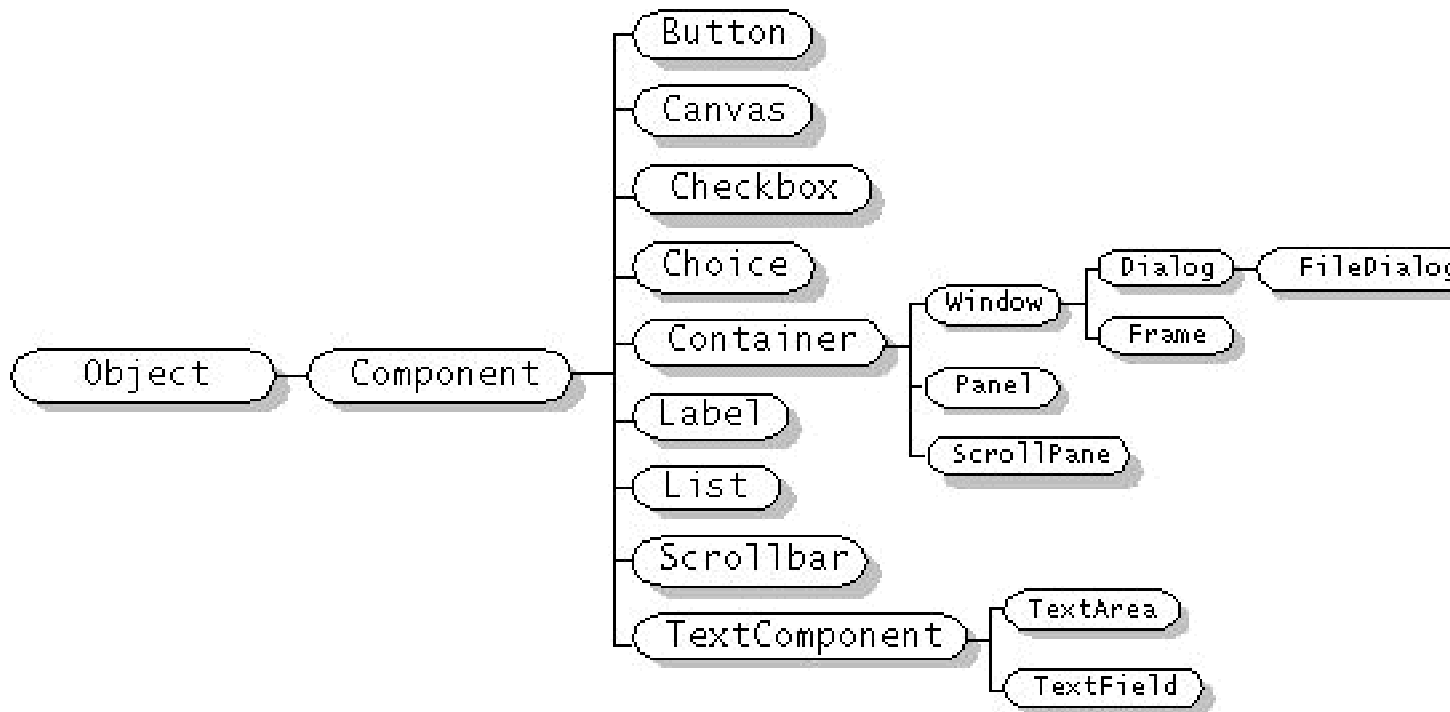

FIG. 26.2: Les différents widgets

Les composants graphiques sont généralement organisés de manière hiérarchique : un composant principal contient d'autres composant qui, à leur tout, peuvent contenir d'autres composants et ainsi de suite.

Parmis tous les composants graphiques, seuls les composants de type `Container` sont susceptibles de contenir d'autres composants. La classe `Container` est elle-même dérivée de la classe `Component` comme le montre la figure 26.2

Un composant s'ajoute dans un container avec la méthode `add` de la classe `Component`.

```
public Component add(Component comp)
public Component add(String name, Component comp)
public Component add(Component comp, int index)
public void add(Component comp, Object constraints)
public void add(Component comp, Object constraints, int index)
protected void addImpl(Component comp, Object constraints, int index)
```

En tant que classe abstraite, il n'est pas permis de créer des instances de cette classe. La classe `Component` factorise à un certain nombre de propriétés et comportements communs à tous les widgets :

- la gestion de la position et des dimensions
- la gestion de l'affichage (voir 28)
- La gestion d'évènements (voir 25)
- Le contrôle des couleurs (voir 29) et fontes (voir 29)
- La gestion des images (voir 30)

26.1.2 Affichage

La posibilité de se dessiner à l'écran est une fonctionnalité commune à tous les composants.

Les composants des applications graphiques doivent pouvoir se dessiner et se redessiner. Lorsqu'un composant nécessite de se dessiner ou redessiner, le composant principal commence par se dessiner. Puis, il dessine un à un tous les composants qu'il contient et ce de manière récursive et sans aucune interruption.

Nous consacrerons un chapitre complet (voir 28) aux problèmes liés à l'affichage.

26.1.3 Gestion de la position et des dimensions

La position et des dimensions d'un composant dans une fenêtre sont généralement gérés par le gestionnaire de placement (*Layout Manager*). Les méthodes `getPreferredSize` et `getMinimumSize` informent ce gestionnaire des contraintes que l'on se fixe sur la taille du composant. En l'abscence de ces précisions, le gestionnaire décidera tout seul de la configuration des composants.

Il existe plusieurs stratégie de placement possible ; à chacune d'elle correspond un gestionnaire particulier. Nous verrons ces différents gestionnaire dans un chapitre suivant (voir 27). En attendant, dans les exemples à venir, on utilisera toujours le gestionnaire `FlowLayout`.

26.1.4 Un petit exemple pour commencer

Commençons une première applet qui contient un bouton pemettant d'ouvrir une fenêtre dans laquelle se trouve trois boutons.

– création de l'applet avec un premier bouton

```
public class Exemple1 extends Applet implements ActionListener {
  Button b;
  DesBoutons fenetre;
```

– création du bouton pour ouvrir une nouvelle fenêtre

```
  public void init() {
    b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
    add(b);
```

– création de la fenêtre dans laquelle se trouve trois boutons

```
    fenetre = new DesBoutons("Des boutons");
```

– Attente du clic sur ce bouton

```
    b.addActionListener(this);
  }
```

– gestion du clic

```
  public void actionPerformed(ActionEvent e) {
    if (! fenetre.isShowing()) fenetre.setVisible(true);
    else fenetre.setVisible(false);
  }
}
```

– Définition de fenêtre avec les trois boutons

```
class DesBoutons extends Frame {
  public DesBoutons(String s) {
    super(s);
    setLayout(new FlowLayout());
    Button b[] = {new Button("Bouton 1"), new Button("Bouton 2"), new Button("Bouton 3")};
    add(b[0]); add(b[1]); add(b[2]);
    pack();
```

– Gestion de la fermeture de la fenêtre

```
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { setVisible(false); }
    });
  }
}
```

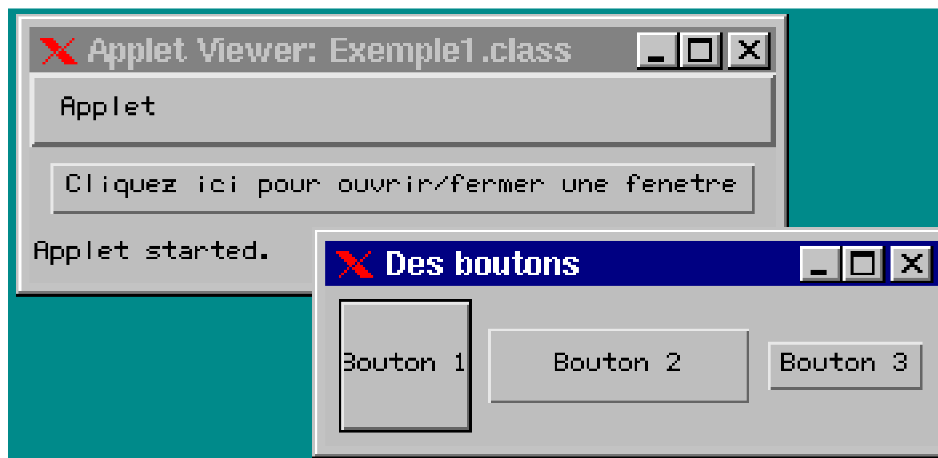

FIG. 26.3: Un premier exemple

26.1.5 Applets et applications autonomes

Comme nous l'avons déjà dit, une applet est une application graphique et bien de initialisations nécessaires pour une application graphique autonome sont faites par défaut dans la classe **Applet**.

Voyons comment faire pour transformaer cette applet en application autonome. Ce qui nous manque c'est la fenêtre principale dans laquelle s'inscrit notre applet. Ainsi, pour transformer notre applet en application autonome, il nous faut créer cette fenêtre principale :

```
import java.awt.*;
import java.awt.event.*;
public class Appli extends Frame implements ActionListener {
  DesBoutons fenetre;
  public Appli() {
    setLayout(new FlowLayout());
    setTitle("Application Autonome");
    Button b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
    b.addActionListener(this);
    add(b);
    fenetre = new DesBoutons("Des boutons");
    pack();
    show();
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
   }
  public static void main(String args[]) {
    Appli window = new Appli();
  }
  public void actionPerformed(ActionEvent e) {
    if (! fenetre.isShowing()) fenetre.setVisible(true);
    else fenetre.setVisible(false);
  }
}
```

26.1.6 Architecture générale

Les composants de *AWT* (Abstract Window Toolkit) sont conçus pour être indépendant des plateformes ; i.e. le même code tourne sur tous les plates formes. Un bouton *AWT* sun *Windows* ressemble à un bouton *Windows* et ce même code exécuter sur un *Macintosh* donne au bouton l'apparence d'un bouton *Macintosh*.

Autrement dit, si le code du programmeur reste le même quelque soit la plate forme, l'implantation fournie pour *AWT* ne l'est pas. Ainsi, à chaque composant *AWT* correspond un objet natif spécifique à chaque plate forme. Ce dernier est désigné par le terme de *Peers* et correspond à une classe *Java*. A la classe des boutons *Java* (java.awt.Button) est associée la classe (java.awt.peer.PeerButton) et il en va de même pour rous les composants de *AWT*.

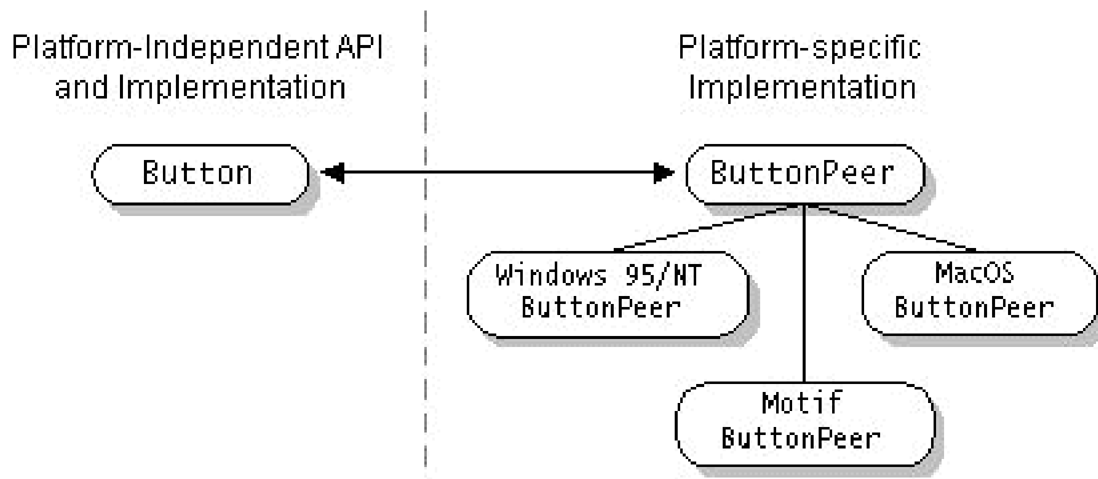

Fig. 26.4: Implantation des composants AWT

A priori, le programmeur n'a pas à se soucier des objets *Peer*. Tout ceci est bien caché derrière la classe `java.awt.component`. Mais la clarification du mécanisme des objets *Peer* permet de comprendre certaines limitations imposées dans *AWT*.

A la création d'un composant (un bouton par exemple), le constructeur de la classe `Component` demande à la classe `Toolkit` de créer l'objet natif correspondant.

La classe `Toolkit` est une espèce d'atelier qui se charge de la gestion de la création des objets natifs. Cette classe `Toolkit` sert en quelque sorte de barrière de séparation entre le monde idylique de *Java* et le monde réel où beaucoup de choses dépendent de l'architecture, du système d'exploitation, du gestionnaire graphique etc.

Au moment d'afficher un container, tous les composants sont crées (*Java*et *Peer*). Si un nouveau composant est ajouté au container après qu'il ait été rendu visible, il faut explicitement faire appel à la méthode `validate` pour que le composant *Peer* soit créé. On peut faire appel à la méthode `validate`

- de la classe du composant que l'on veut ajouter
- ou de la classe `Container` qui contient le composant.

26.1.7 La classe Component

```
public abstract class Component extends Object
                implements ImageObserver, MenuContainer, Serializable {
    public static final float TOP_ALIGNMENT
    public static final float CENTER_ALIGNMENT
    public static final float BOTTOM_ALIGNMENT
    public static final float LEFT_ALIGNMENT
    public static final float RIGHT_ALIGNMENT
    protected Component()
    public String getName()
    public void setName(String name)
    public Container getParent()
    public void setDropTarget(DropTarget dt) throws IllegalArgumentException, SecurityException
    public DropTarget getDropTarget()
    public final Object getTreeLock()
    public Toolkit getToolkit()
    public boolean isValid()
    public boolean isVisible()
    public boolean isShowing()
    public boolean isEnabled()
    public void setEnabled(boolean b)
    public void setVisible(boolean b)
    public void setForeground(Color c)
    public Color getBackground()
    public void setBackground(Color c)
    public Font getFont()
    public void setFont(Font f)
    public Locale getLocale()
    public void setLocale(Locale l)
    public ColorModel getColorModel()
    public Point getLocation()
    public Point getLocationOnScreen()
    public void setLocation(int x, int y)
    public void setLocation(Point p)
    public Dimension getSize()
    public void setSize(int width, int height)
    public void resize(Dimension d)
    public Rectangle getBounds()
    public void setBounds(int x, int y, int width, int height)
    public void setBounds(Rectangle r)
    public Dimension getPreferredSize()
    public Dimension getMinimumSize()
    public Dimension getMaximumSize()
    public float getAlignmentX()
    public float getAlignmentY()
    public void doLayout()
    public void validate()
    public void invalidate()
    public Graphics getGraphics()
    public FontMetrics getFontMetrics(Font font)
    public void setCursor(Cursor cursor)
    public Cursor getCursor()
    public void paint(Graphics g)
    public void update(Graphics g)
    public void paintAll(Graphics g)
    public void repaint()
```

```
        public void repaint(long tm)
        public void repaint(int x, int y, int width, int height)
        public void repaint(long tm, int x, int y, int width, int height)
        public void print(Graphics g)
        public void printAll(Graphics g)
        public boolean imageUpdate(Image img, int flags, int x, int y, int w, int h)
        public Image createImage(ImageProducer producer)
        public Image createImage(int width, int height)
        public boolean prepareImage(Image image, ImageObserver observer)
        public boolean prepareImage(Image image, int width, int height, ImageObserver observer)
        public int checkImage(Image image, ImageObserver observer)
        public int checkImage(Image image, int width, int height, ImageObserver observer)
        public boolean contains(int x, int y)
        public boolean contains(Point p)
        public Component getComponentAt(int x, int y)
        public Component getComponentAt(Point p)
        public final void dispatchEvent(AWTEvent e)
        public void addComponentListener(ComponentListener l)
        public void removeComponentListener(ComponentListener l)
        public void addFocusListener(FocusListener l)
        public void removeFocusListener(FocusListener l)
        public void addKeyListener(KeyListener l)
        public void removeKeyListener(KeyListener l)
        public void addMouseListener(MouseListener l)
        public void removeMouseListener(MouseListener l)
        public void addMouseMotionListener(MouseMotionListener l)
        public void removeMouseMotionListener(MouseMotionListener l)
        public void addInputMethodListener(InputMethodListener l)
        public void removeInputMethodListener(InputMethodListener l)
        public InputMethodRequests getInputMethodRequests()
        public InputContext getInputContext()
        protected final void enableEvents(long eventsToEnable)
        protected final void disableEvents(long eventsToDisable)
        protected void processEvent(AWTEvent e)
        protected void processComponentEvent(ComponentEvent e)
        protected void processFocusEvent(FocusEvent e)
        protected void processKeyEvent(KeyEvent e)
        protected void processMouseEvent(MouseEvent e)
        protected void processMouseMotionEvent(MouseEvent e)
        protected void processInputMethodEvent(InputMethodEvent e)
        public void addNotify()
        public void removeNotify()
        public void requestFocus()
        public void transferFocus()
        public void add(PopupMenu popup)
        public void remove(MenuComponent popup)
        protected String paramString()
        public String toString()
        public void list()
        public void list(PrintStream out)
        public void list(PrintStream out, int indent)
        public void list(PrintWriter out)
        public void list(PrintWriter out, int indent)
        public void addPropertyChangeListener(PropertyChangeListener listener)
        public void removePropertyChangeListener(PropertyChangeListener listener)
        public void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)
        public void removePropertyChangeListener(String propertyName, PropertyChangeListener listener)
        protected void firePropertyChange(String propertyName, Object oldValue, Object newValue)
    }
```

On ne décrira pas, ici, la totalité des méthodes de cette classe. Nous en verrons un certain nombre au fur et à mesure des exemples qui suivent.

`public boolean isValid ()`

Vrai si le composant est valide.

`public void validate ()`

S

`public boolean isVisible ()`

Vrai si le composant est visible.

`public boolean isShowing ()`

Vrai si le composant est visible et le container qui le contient est visible et l'affiche.

26.1.8 La classe Container

```
public abstract class Container extends Component {
   protected Container()
   public int getComponentCount()
   public Component getComponent(int n)
   public Component[] getComponents()
   public Insets getInsets()
   public Component add(Component comp)
   public Component add(String name, Component comp)
   public Component add(Component comp, int index)
   public void add(Component comp, Object constraints)
   public void add(Component comp, Object constraints, int index)
   protected void addImpl(Component comp, Object constraints, int index)
   public void remove(int index)
   public void remove(Component comp)
   public void removeAll()
   public LayoutManager getLayout()
   public void setLayout(LayoutManager mgr)
   public void doLayout()
   public void invalidate()
   public void validate()
   protected void validateTree()
   public Dimension getPreferredSize()
   public Dimension getMinimumSize()
   public Dimension getMaximumSize()
   public float getAlignmentX()
   public float getAlignmentY()
   public void paint(Graphics g)
   public void update(Graphics g)
   public void print(Graphics g)
   public void paintComponents(Graphics g)
   public void printComponents(Graphics g)
   public void addContainerListener(ContainerListener l)
   public void removeContainerListener(ContainerListener l)
   protected void processEvent(AWTEvent e)
   protected void processContainerEvent(ContainerEvent e)
   public Component getComponentAt(int x, int y)
   public Component getComponentAt(Point p)
   public void addNotify()
   public void removeNotify()
   public boolean isAncestorOf(Component c)
   protected String paramString()
   public void list(PrintStream out, int indent)
   public void list(PrintWriter out, int indent)
}
```

26.2 Button

Nous avons déjà rencontré ce type de widget ; il s'agit d'un bouton etiquetté.

```
public class Button extends Component {
   public Button()
   public Button(String label)
   public void addNotify()
   public String getLabel()
   public void setLabel(String label)
   public void setActionCommand(String command)
   public String getActionCommand()
   public void addActionListener(ActionListener l)
   public void removeActionListener(ActionListener l)
   protected void processEvent(AWTEvent e)
   protected void processActionEvent(ActionEvent e)
   protected String paramString()
}
```

FIG. 26.5: Des boutons

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class WBouton extends Applet implements ActionListener {
  int milieu = 1;
  Button b[] = { new Button("Desactiver bouton " + milieu        ),
   new Button("Tourner"),
   new Button("Activer bouton " + milieu)
                 };
  public void init() {
    for (int i = 0; i < b.length; i++) b[i].addActionListener(this);
    for (int i = 0; i < b.length; i++) add(b[i]);

    b[milieu].setActionCommand("Tourner");
    b[(milieu-1)%3].setActionCommand("Désactiver");
    b[(milieu+1)%3].setActionCommand("Activer");

  }
  public void actionPerformed(ActionEvent e) {
    String s;
      System.out.println(((milieu-1)%3) +  " " + milieu + " " + ((milieu+1)%3));
    s = e.getActionCommand();
      System.out.println(s);
    if ("Désactiver".equals(s))
      b[milieu].setEnabled(false);
    else if ("Activer".equals(s))
      b[milieu].setEnabled(true);
    else {
      milieu = (milieu+1)%3;
      System.out.println(((milieu-1+3)%3) +  " " + milieu + " " + ((milieu+1)%3));
      b[milieu].setActionCommand("Tourner");
      b[(milieu-1+3)%3].setActionCommand("Désactiver");
      b[(milieu+1)%3].setActionCommand("Activer");
      b[milieu].setLabel("Tourner");
      b[(milieu-1+3)%3].setLabel("Desactiver bouton " + milieu);
      b[(milieu+1)%3].setLabel("Activer bouton " + milieu);
      doLayout();
    }
  }
  public static void main(String[] args) {
    Frame f = new Frame("Des boutons ...");
    WBouton p =  new WBouton();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
    f.add(p);
    f.setSize(350, 170);
    f.pack();
    f.show();
  }
}
```

26.3 Frame

Les objets de la classe **Frame** sont des fenêtres graphiques que l'on utilise dans les applets et les applications. Comme nous l'avons déjà dit, toute application graphique possède au moins un objet de type **Frame**.

```
public class Frame extends Window implements MenuContainer {
```

```
    public Frame()
    public Frame(String title)
    public void addNotify()
    public String getTitle()
    public void setTitle(String title)
    public Image getIconImage()
    public void setIconImage(Image image)
    public MenuBar getMenuBar()
    public void setMenuBar(MenuBar mb)
    public boolean isResizable()
    public void setResizable(boolean resizable)
    public void remove(MenuComponent m)
    public void dispose()
    protected String paramString()
}
```

26.4 Panel

Les objets de type `Panel` sont des containers. Une applet est un `Panel` avec quelques particularités pour pouvoir être exécutée dans un *Browser*.

```
public class Panel extends Container {
    public Panel()
    public Panel(LayoutManager layout)
    public void addNotify()
}
```

Par défaut, le gestion de placement `FlowLayout` est associé aux `Panel`. On peut décider d'utiliser un autre gestionnaire en le fournissant en argument du constructeur ou en utilisant la méthode `setLayout` de la classe `Container`.

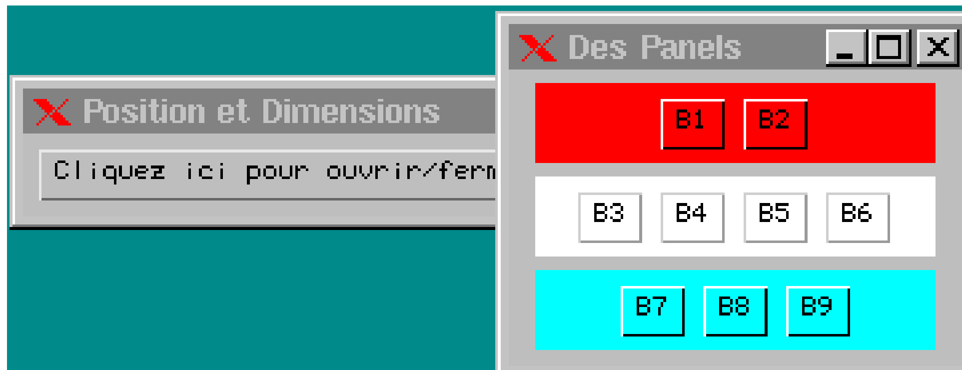

Fig. 26.6: Des Panels

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class WPanel extends Applet implements ActionListener {
  Button b;
  DesPanels fenetre;
  public void init() {
    b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
    add(b);
    b.addActionListener(this);
    fenetre = new DesPanels("Des Panels");
  }
  public void actionPerformed(ActionEvent e) {
    if (! fenetre.isShowing()) fenetre.setVisible(true);
    else fenetre.setVisible(false);
  }
  public static void main(String[] args) {
    Frame f = new Frame("Position et Dimensions");
    WPanel p =  new WPanel();
    f.addWindowListener(new WindowAdapter() {
```

```
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
    f.add(p);
    f.pack();
    f.show();
  }
}

class DesPanels extends Frame {
  Dimension d = new Dimension(180, 150);
  public DesPanels(String s) {
    super(s);
    setLayout(new FlowLayout());
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { setVisible(false); }
    });
    MesPanels [] p = new MesPanels[3];
    Button b0 [] =  {new Button("B1"), new Button("B2")};
    Button b1 [] =  {new Button("B3"), new Button("B4"), new Button("B5"), new Button("B6")};
    Button b2 [] =  {new Button("B7"), new Button("B8"), new Button("B9")};
    p[0] = new MesPanels(b0, Color.red);
    p[1] = new MesPanels(b1, Color.white);
    p[2] = new MesPanels(b2, Color.cyan);
    add(p[0]); add(p[1]); add(p[2]);
    pack();
  }
  public Dimension getMinimumSize() {return d;}
  public Dimension getPreferredSize() { return d;}
}

class MesPanels extends Panel  {
  Dimension d = new Dimension(150, 30);
  public MesPanels(Button [] b, Color c) {
    setBackground(c);
    for (int i = 0; i<b.length; i++) add(b[i]);
  }
  public Dimension getMinimumSize() {return d;}
  public Dimension getPreferredSize() { return d;}
}
```

26.5 Label

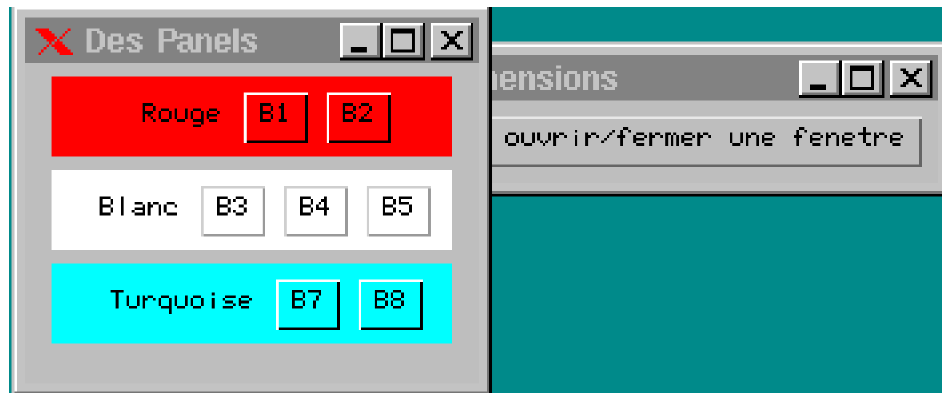

Fig. 26.7: Label

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class WLabel extends Applet implements ActionListener {
  Button b;
```

```
  DesPanelsL fenetre;
  public void init() {
    b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
    add(b);
    b.addActionListener(this);
    fenetre = new DesPanelsL("Des Panels");
  }
  public void actionPerformed(ActionEvent e) {
    if (! fenetre.isShowing()) fenetre.setVisible(true);
    else fenetre.setVisible(false);
  }
  public static void main(String[] args) {
    Frame f = new Frame("Position et Dimensions");
    WLabel p =  new WLabel();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
    f.add(p);
    f.pack();
    f.show();
  }
}

class DesPanelsL extends Frame {
  Dimension d = new Dimension(180, 150);
  public DesPanelsL(String s) {
    super(s);
    setLayout(new FlowLayout());
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { setVisible(false); }
    });
    MesPanelsL [] p = new MesPanelsL[3];
    Button b0 [] =  {new Button("B1"), new Button("B2")};
    Button b1 [] =  {new Button("B3"), new Button("B4"), new Button("B5"), new Button("B6")};
    Button b2 [] =  {new Button("B7"), new Button("B8"), new Button("B9")};
    p[0] = new MesPanelsL(b0, Color.red, "Rouge");
    p[1] = new MesPanelsL(b1, Color.white, "Blanc");
    p[2] = new MesPanelsL(b2, Color.cyan, "Turquoise");
    add(p[0]); add(p[1]); add(p[2]);
    pack();
  }
  public Dimension getMinimumSize() {return d;}
  public Dimension getPreferredSize() { return d;}
}

class MesPanelsL extends Panel  {
  Dimension d = new Dimension(150, 30);
  public MesPanelsL(Button [] b, Color c, String s) {
    Label l;
    setBackground(c);
    add(l=new Label(s));
    l.setAlignment(Label.RIGHT);
    for (int i = 0; i<b.length; i++) add(b[i]);
  }
  public Dimension getMinimumSize() {return d;}
  public Dimension getPreferredSize() { return d;}
}
```

26.6 TextField et TextArea

zone de texte sur une ligne et multi ligne

```
public class TextComponent extends Component {
  protected transient TextListener textListener
  public void removeNotify()
  public void setText(String t)
  public String getText()
  public String getSelectedText()
  public boolean isEditable()
  public void setEditable(boolean b)
```

```
  public int getSelectionStart()
  public void setSelectionStart(int selectionStart)
  public int getSelectionEnd()
  public void setSelectionEnd(int selectionEnd)
  public void select(int selectionStart, int selectionEnd)
  public void selectAll()
  public void setCaretPosition(int position)
  public int getCaretPosition()
  public void addTextListener(TextListener l)
  public void removeTextListener(TextListener l)
  protected void processEvent(AWTEvent e)
  protected void processTextEvent(TextEvent e)
  protected String paramString()
}
```

```
class TextField extends TextComponent {
  public TextField()
  public TextField(String text)
  public TextField(int columns)
  public TextField(String text, int columns)
  public void addNotify()
  public char getEchoChar()
  public void setEchoChar(char c)
  public boolean echoCharIsSet()
  public int getColumns()
  public void setColumns(int columns)
  public Dimension getPreferredSize(int columns)
  public Dimension getPreferredSize()
  public Dimension getMinimumSize(int columns)
  public Dimension getMinimumSize()
  public void addActionListener(ActionListener l)
  public void removeActionListener(ActionListener l)
  protected void processEvent(AWTEvent e)
  protected void processActionEvent(ActionEvent e)
  protected String paramString()
}
```

```
public class TextArea extends TextComponent {
  public static final int SCROLLBARS_BOTH
  public static final int SCROLLBARS_VERTICAL_ONLY
  public static final int SCROLLBARS_HORIZONTAL_ONLY
  public static final int SCROLLBARS_NONE
  public TextArea()
  public TextArea(String text)
  public TextArea(int rows, int columns)
  public TextArea(String text, int rows, int columns)
  public TextArea(String text, int rows, int columns, int scrollbars)
  public void addNotify()
  public void insert(String str, int pos)
  public void append(String str)
  public void replaceRange(String str, int start, int end)
  public int getRows()
  public void setRows(int rows)
  public int getColumns()
  public void setColumns(int columns)
  public int getScrollbarVisibility()
  public Dimension getPreferredSize(int rows, int columns)
  public Dimension getPreferredSize()
  public Dimension getMinimumSize(int rows, int columns)
  public Dimension getMinimumSize()
  protected String paramString()
}
```

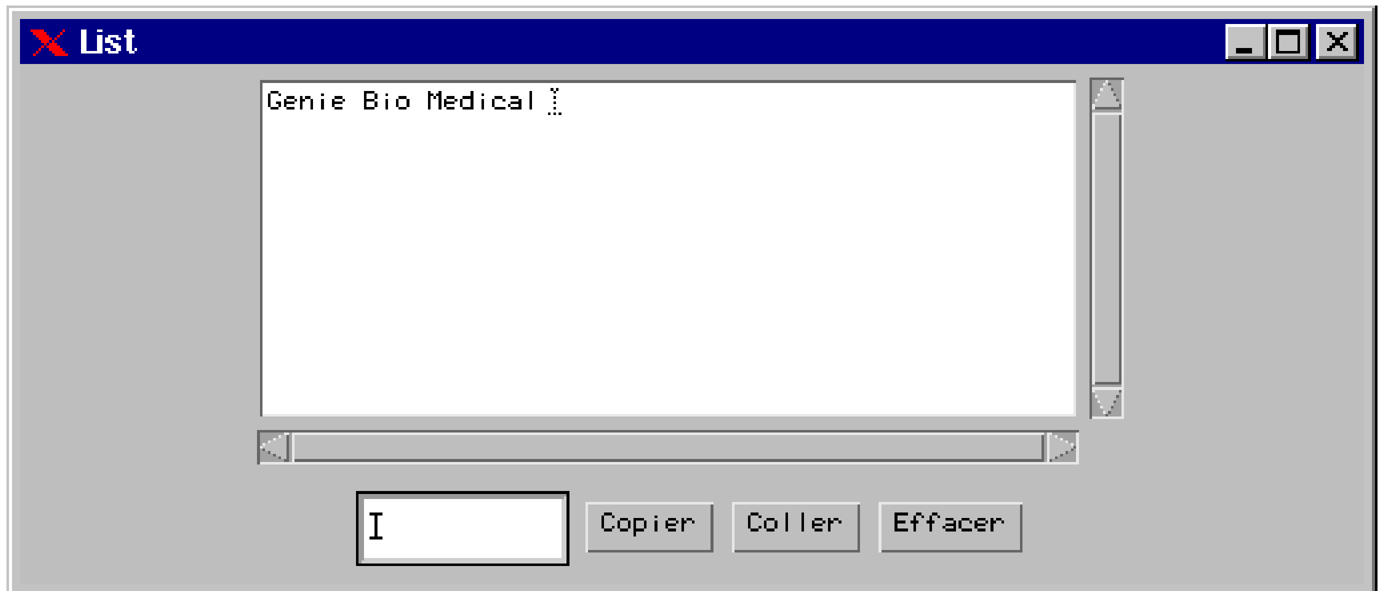

Fig. 26.8: TextField et TextArea

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class WText extends Applet implements TextListener, ActionListener  {
  TextArea TA;
  TextField TF;
  String rc = System.getProperty("line.separator");
  public void init() {
    TA = new TextArea("", 10, 50, TextArea.SCROLLBARS_BOTH);
    TA.setEditable(true);
    add(TA);
    TF = new TextField(10);
    TF.setEditable(true);
    TA.addTextListener(this);
    TF.addTextListener(this);
    TF.addActionListener(this);
    add(new UnPanel(TF, TA));
  }
  public void textValueChanged(TextEvent e)  {
    TextComponent tc = (TextComponent)e.getSource();
    String s = tc.getText();
  }
  public void actionPerformed(ActionEvent e) {
    TextComponent tc = (TextComponent)e.getSource();
    String s = tc.getText();
    if (s.equals("gbm")) s = "Genie Bio Medical ";
    else if (s.equals("es2i")) s = "Informatique ";
    else if (s.equals("gbma")) s = "Genie Biologique et Micro Biologie appliquée ";
    TA.append(s);
    TA.setCaretPosition(2000);
    TF.setText("");
  }
  public static void main(String[] args) {
    Frame f = new Frame("Text");
    WText p =  new WText();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
    f.add(p);
    f.pack();
    f.show();
```

```
  }
}

class UnPanel extends Panel implements  ActionListener  {
  TextField tf; TextArea ta;
  String selection = "";
  public UnPanel(TextField TF, TextArea TA) {
    tf = TF; ta = TA;
    add(tf);
    Button copier = new Button("Copier");
    add(copier);
    Button coller = new Button("Coller");
    add(coller);
    Button effacer = new Button("Effacer");
    add(effacer);
    copier.addActionListener(this);
    coller.addActionListener(this);
    effacer.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    String tc = e.getActionCommand();
    System.out.println(tc + " ... " + selection);
    if (tc.equals("Coller")) { ta.insert(selection, ta.getCaretPosition()); }
    else if (tc.equals("Copier"))  selection  = ta.getSelectedText();
    else {
      selection  = ta.getSelectedText();
      ta.replaceText("", ta.getCaretPosition() - selection.length(), ta.getCaretPosition() );
    }
    System.out.println(tc + " " + selection);
  }

}
```

26.7 Checkbox et CheckboxGroup

Bouton à cocher etiquetté.

26.7.1 Checkbox

```
public class Checkbox extends Component implements ItemSelectable {
   public Checkbox()
   public Checkbox(String label)
   public Checkbox(String label, boolean state)
   public Checkbox(String label, boolean state, CheckboxGroup group)
   public Checkbox(String label, CheckboxGroup group, boolean state)
   public void addNotify()
   public String getLabel()
   public void setLabel(String label)
   public boolean getState()
   public void setState(boolean state)
   public Object[] getSelectedObjects()
   public CheckboxGroup getCheckboxGroup()
   public void setCheckboxGroup(CheckboxGroup g)
   public void addItemListener(ItemListener l)
   public void removeItemListener(ItemListener l)
   protected void processEvent(AWTEvent e)
   protected void processItemEvent(ItemEvent e)
   protected String paramString()
}
```

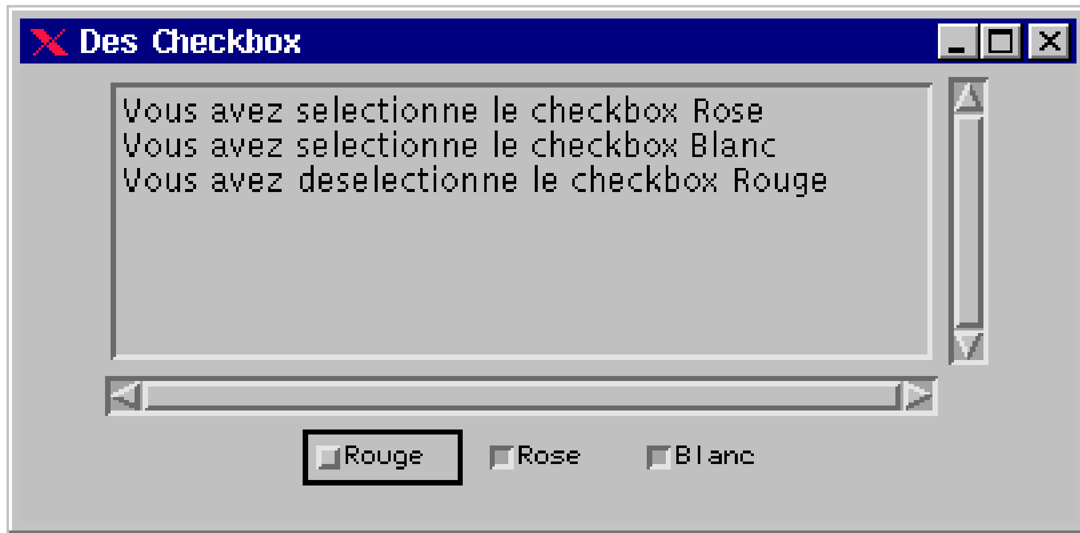

FIG. 26.9: Des Checkbox

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class WCheckbox extends Applet implements ActionListener, ItemListener {
  TextArea textArea;
  String rc = System.getProperty("line.separator");
  Button b;
  public void init() {
    textArea = new TextArea(8, 50);
    textArea.setEditable(false);
    add(textArea);

    Checkbox c1 = new  Checkbox("Rouge", true);
    Checkbox c2 = new  Checkbox("Rose", false);
    Checkbox c3 = new  Checkbox("Blanc", false);
    add(c1); add(c2); add(c3);
    c1.addItemListener(this);
    c2.addItemListener(this);
    c3.addItemListener(this);
    b = new Button("O.K."); b.addActionListener(this);
  }
  void afficher(String s) { textArea.append(s + rc); textArea.setCaretPosition(2000); }
  public void actionPerformed(ActionEvent e) { afficher(e.toString()); }
  public void itemStateChanged(ItemEvent e)  {
    String sel;
    sel = (e.getStateChange()==e.SELECTED) ? "selectionne" : "deselectionne";
    afficher("Vous avez " + sel +
     " le checkbox " + e.getItem());
  }
  public static void main(String[] args) {
    Frame f = new Frame("Des Checkbox");
    WCheckbox p =  new WCheckbox();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
    f.add(p);
    f.pack();
    f.show();
  }

}
```

26.7.2 CheckboxGroup

```
public class CheckboxGroup implements Serializable {
   public CheckboxGroup()
   public Checkbox getSelectedCheckbox()
   public void setSelectedCheckbox(Checkbox box)
   public String toString()
}
```

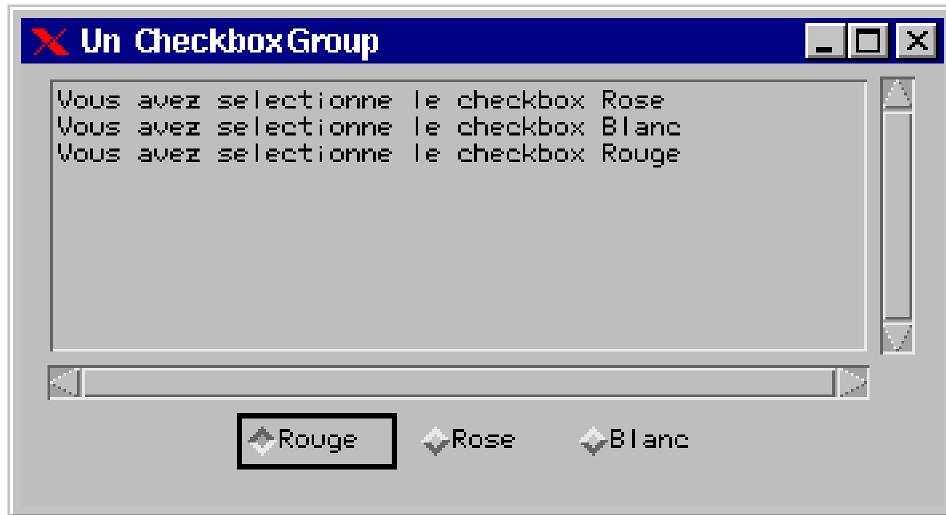

Fig. 26.10: Des CheckboxGroup

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class WCheckboxGroup extends Applet implements ActionListener, ItemListener {
  TextArea textArea;
  String rc = System.getProperty("line.separator");
  Button b;
  public void init() {
    textArea = new TextArea(8, 50);
    textArea.setEditable(false);
    add(textArea);

    CheckboxGroup cbg = new CheckboxGroup();
    Checkbox c1 = new  Checkbox("Rouge", cbg, true);
    Checkbox c2 = new  Checkbox("Rose", cbg, false);
    Checkbox c3 = new  Checkbox("Blanc", cbg, false);
    add(c1); add(c2); add(c3);
    c1.addItemListener(this);
    c2.addItemListener(this);
    c3.addItemListener(this);
    b = new Button("O.K."); b.addActionListener(this);
  }
  void afficher(String s) { textArea.append(s + rc); textArea.setCaretPosition(2000); }
  public void actionPerformed(ActionEvent e) { afficher(e.toString()); }
  public void itemStateChanged(ItemEvent e)  {
    String sel;
    sel = (e.getStateChange()==e.SELECTED) ? "selectionne" : "deselectionne";
    afficher("Vous avez " + sel +
     " le checkbox " + e.getItem());
  }
  public static void main(String[] args) {
    Frame f = new Frame("Un CheckboxGroup");
    WCheckboxGroup p =  new WCheckboxGroup();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
```

```
    p.init();
    f.add(p);
    f.pack();
    f.show();
  }

}
```

26.8 Choice

menu déroulant.

```
class Choice extends Component implements ItemSelectable {
  public Choice()
  public void addNotify()
  public int getItemCount()
  public String getItem(int index)
  public void add(String item)
  public void addItem(String item)
  public void insert(String item, int index)
  public void remove(String item)
  public void remove(int position)
  public void removeAll()
  public String getSelectedItem()
  public Object[] getSelectedObjects()
  public int getSelectedIndex()
  public void select(int pos)
  public void select(String str)
  public void addItemListener(ItemListener l)
  public void removeItemListener(ItemListener l)
  protected void processEvent(AWTEvent e)
  protected void processItemEvent(ItemEvent e)
  protected String paramString()
}
```

Fig. 26.11: Choice

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class WChoice extends Applet  implements ItemListener {
  TextArea textArea;
  String rc = System.getProperty("line.separator");
  Choice meschoix;
  public void init() {
```

```
    textArea = new TextArea(8, 50);
    textArea.setEditable(false);
    add(textArea);
    meschoix = new  Choice();
    meschoix.addItem("Bordeaux");
    meschoix.addItem("Bourgogne");
    meschoix.addItem("Alsace");
    meschoix.addItem("Cotes du Rhones");
    meschoix.addItemListener(this);
    add(meschoix);
  }
  public void itemStateChanged(ItemEvent e)  { afficher("itemStateChanged " + e.getItem()); }
  void afficher(String s) { textArea.append(s + rc); textArea.setCaretPosition(2000); }
  public static void main(String[] args) {
    Frame f = new Frame("Choice");
    WChoice p =  new WChoice();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
    f.add(p);
    f.pack();
    f.show();
  }

}
```

26.9 List

menu défilant.

```
public class List extends Component implements ItemSelectable {
   public List()
   public List(int rows)
   public List(int rows, boolean multipleMode)
   public void addNotify()
   public void removeNotify()
   public int getItemCount()
   public String getItem(int index)
   public String[] getItems()
   public void add(String item)
   public void add(String item, int index)
   public void replaceItem(String newValue, int index)
   public void removeAll()
   public void remove(String item)
   public void remove(int position)
   public int getSelectedIndex()
   public int[] getSelectedIndexes()
   public String getSelectedItem()
   public String[] getSelectedItems()
   public Object[] getSelectedObjects()
   public void select(int index)
   public void deselect(int index)
   public boolean isIndexSelected(int index)
   public int getRows()
   public boolean isMultipleMode()
   public void setMultipleMode(boolean b)
   public int getVisibleIndex()
   public void makeVisible(int index)
   public Dimension getPreferredSize(int rows)
   public Dimension getPreferredSize()
   public Dimension getMinimumSize(int rows)
   public Dimension getMinimumSize()
   public void addItemListener(ItemListener l)
   public void removeItemListener(ItemListener l)
   public void addActionListener(ActionListener l)
   public void removeActionListener(ActionListener l)
   protected void processEvent(AWTEvent e)
   protected void processItemEvent(ItemEvent e)
   protected void processActionEvent(ActionEvent e)
   protected String paramString()
```

```
}
```

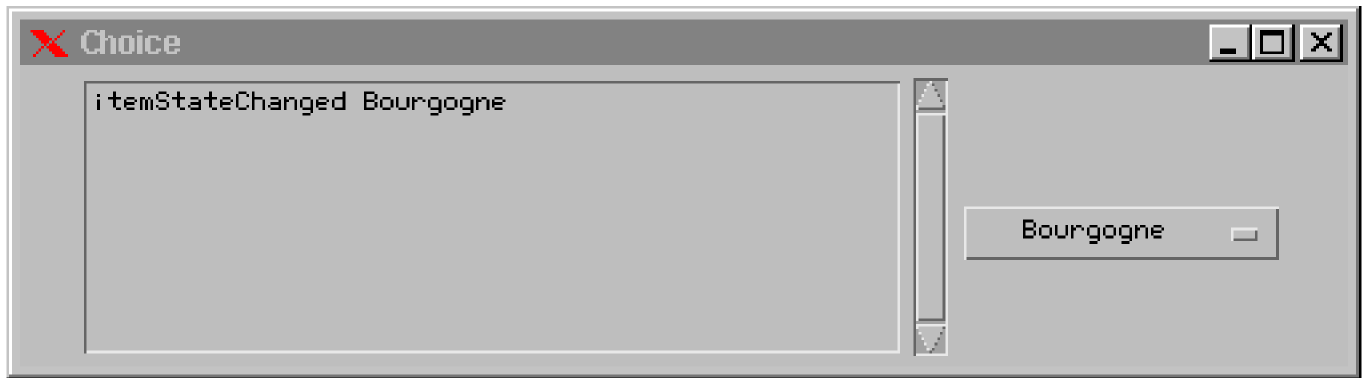

FIG. 26.12: List

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class WList extends Applet implements ItemListener  {
  TextArea textArea;
  List l1, l2, l3;
  String rc = System.getProperty("line.separator");
  public void init() {
    textArea = new TextArea(8, 50);
    textArea.setEditable(false);
    add(textArea);
    l1 = new List(3, true);
    l2 = new List(1, true);
    l3 = new List(2, false);
    add(l1); add(l2); add(l3);
    l1.add("1991"); l1.add("1992"); l1.add("1993"); l1.add("1994"); l1.add("1995"); l1.add("1996");
    l2.add("Rouge"); l2.add("Rose"); l2.add("Blanc");
    l3.add("Bordeaux"); l3.add("Bourgogne"); l3.add("Alsace"); l3.add("Rhone");
    l1.addItemListener(this);
    l2.addItemListener(this);
    l3.addItemListener(this);

  }
  public void itemStateChanged(ItemEvent e)  {
    List l = (List)(e.getSource());
    String sl;
    String s[];
    if (l == l1) sl = "Annee";
    else if (l == l2) sl = "Type";
    else sl = "Region";
    sl += "-> votre choix: ";
    int i;
    for (s=l.getSelectedItems(), i=0; i<s.length; sl += s[i++] + " ");
    afficher(sl);
  }
  void afficher(String s) { textArea.append(s + rc); textArea.setCaretPosition(2000); }
  public static void main(String[] args) {
    Frame f = new Frame("List");
    WList p =  new WList();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
    f.add(p);
    f.pack();
    f.show();
  }
}
```

26.10 Canvas

Zone vierge sur laquelle on peut insérer des dessins et des immages et capturer les évènements utilisateur. C'est, en particulier, utile pour implenter des composants non standard. Par exemple, si l'on veut définir un bouton ayant une image en fond, on utilisera un objet de type **Canvas**. Nous verrons au chapitre 30 la manipulation des images.

```
public class Canvas extends Component {
   public Canvas()
   public void addNotify()
   public void paint(Graphics g)
}
```

FIG. 26.13: Canvas

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

public class WCanvas extends Applet {
  public void init() {
    Image img1 = getImage(getCodeBase(), "tourai.gif");
    MonCanvas mc = new MonCanvas(img1);
    add(mc);
  }
}

class MonCanvas extends Panel {
  Image image;
  Dimension size = new Dimension(150, 150);
  public MonCanvas(Image image) { this.image = image; }
  public Dimension getPreferredSize() { return getMinimumSize(); }
  public Dimension getMinimumSize() { return size; }
  public void paint (Graphics g) {
    g.drawImage(image, 0, 0, this);
  }
}
```

26.11 Menu

Menu.

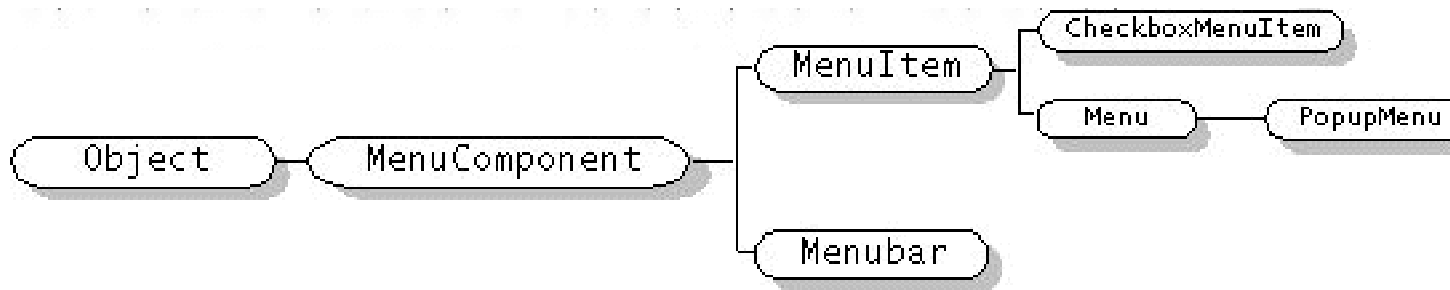

FIG. 26.14: Les menus

```
public class Menu extends MenuItem implements MenuContainer {
   public Menu()
   public Menu(String label)
   public Menu(String label, boolean tearOff)
   public void addNotify()
   public void removeNotify()
   public boolean isTearOff()
   public int getItemCount()
   public MenuItem getItem(int index)
   public MenuItem add(MenuItem mi)
   public void add(String label)
   public void insert(MenuItem menuitem, int index)
   public void insert(String label, int index)
   public void addSeparator()
   public void insertSeparator(int index)
   public void remove(int index)
   public void remove(MenuComponent item)
   public void removeAll()
   public String paramString()
}
```

```
public class MenuItem extends MenuComponent {
   public MenuItem()
   public MenuItem(String label)
   public MenuItem(String label, MenuShortcut s)
   public void addNotify()
   public String getLabel()
   public void setLabel(String label)
   public boolean isEnabled()
   public void setEnabled(boolean b)
   public MenuShortcut getShortcut()
   public void setShortcut(MenuShortcut s)
   public void deleteShortcut()
   protected final void enableEvents(long eventsToEnable)
   protected final void disableEvents(long eventsToDisable)
   public void setActionCommand(String command)
   public String getActionCommand()
   public void addActionListener(ActionListener l)
   public void removeActionListener(ActionListener l)
   protected void processEvent(AWTEvent e)
   protected void processActionEvent(ActionEvent e)
   public String paramString()
}
```

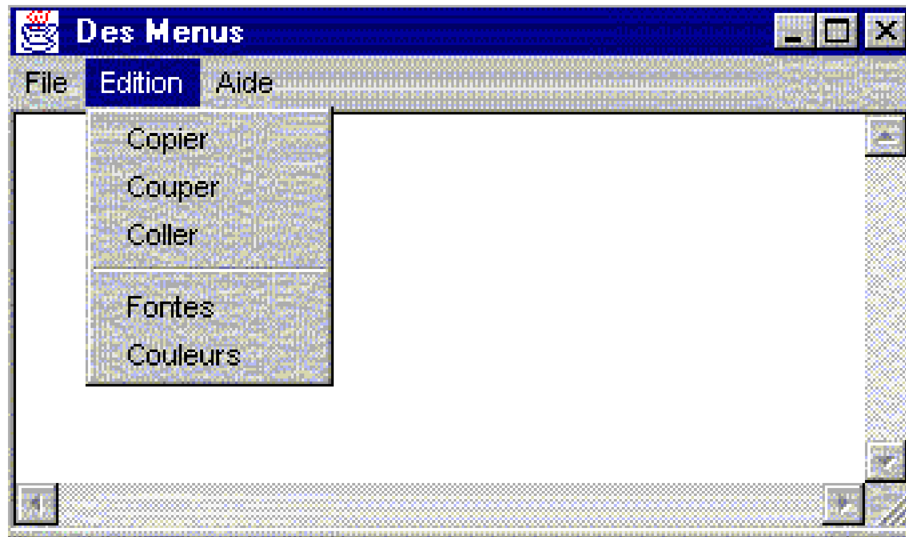

Fig. 26.15: Menu

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

public class WMenu extends Applet implements ActionListener  {
   Button b;
   FenetreMenu fenetre;
   public void init() {
      b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
      add(b);
      b.addActionListener(this);
      fenetre = new FenetreMenu(" Des Menus");
   }
   public void actionPerformed(ActionEvent e) {
      if (! fenetre.isShowing())
         fenetre.setVisible(true);
      else
         fenetre.setVisible(false);
   }
   public static void main(String[] args) {
      Frame f = new Frame("Des menus");
      WMenu p =  new WMenu();
      f.addWindowListener(
                               new WindowAdapter() {
                                  public void windowClosing(WindowEvent e) {
                                     System.exit(0); }
                               });
      p.init();
      f.add(p);
      f.pack();
      f.show();
   }
}

class MonTextArea extends TextArea  implements ActionListener {
   PopupMenu popup;
   String rc = System.getProperty("line.separator");
   public MonTextArea(int l, int c) {
      super(l,c);
      setEditable(false);
      popup = new PopupMenu("Un Popup Menu");
      add(popup);
      popup.add(new MenuItem("item 1"));
      popup.add(new MenuItem("item 2"));
      popup.addActionListener(this);
```

```
      enableEvents(AWTEvent.MOUSE_EVENT_MASK);
   }
   public void processMouseEvent(MouseEvent e) {
      if (e.isPopupTrigger()) {
         popup.show(e.getComponent(), e.getX(), e.getY());
      }
      super.processMouseEvent(e);
   }
   public void actionPerformed(ActionEvent e) {
      afficher(e.getActionCommand());
   }
   public void afficher(String s) {
      append(s + rc);
      setCaretPosition(2000);
   }
}

class FenetreMenu extends Frame  implements ActionListener {
   MonTextArea textarea;
   String rc = System.getProperty("line.separator");
   public FenetreMenu(String s) {
      super(s);
      setSize(200, 150);
      textarea = new MonTextArea(8, 50);
      add(textarea);

      MenuBar mb = new MenuBar();
      setMenuBar(mb);

      Menu file;
      mb.add(file =  new Menu("File"));

      file.add(new MenuItem("Nouveau"));
      file.add(new MenuItem("Ouvrir"));
      file.add(new MenuItem("Enregistrer"));
      file.addSeparator();
      file.add(new MenuItem("Imprimer"));
      file.add(new MenuItem("Quitter"));
      file.addActionListener(this);


      Menu edition;
      mb.add(edition =  new Menu("Edition"));

      edition.add(new MenuItem("Copier"));
      edition.add(new MenuItem("Couper"));
      edition.add(new MenuItem("Coller"));
      edition.addSeparator();
      edition.add(new MenuItem("Fontes"));
      edition.add(new MenuItem("Couleurs"));
      edition.addActionListener(this);

      Menu aide;
      mb.add(aide =  new Menu("Aide"));
      mb.setHelpMenu(aide);
      MenuItem ap;
      aide.add(ap=new MenuItem("A propos"));
      ap.setShortcut(new MenuShortcut(KeyEvent.VK_5));
      aide.add(new MenuItem("Aide en ligne"));
      aide.add(new MenuItem("Mise a jour"));
      aide.addActionListener(this);
   }

   public void actionPerformed(ActionEvent e) {
      afficher(e.getActionCommand());
   }
   void afficher(String s) {
      textarea.append(s + rc);
      textarea.setCaretPosition(2000);
   }
}
```

26.12 Scrollbar

Les **`Scrollbar`** servent dans deux cas de figure :

– comme glissière pour modifier une valeur graphiquement

```
public class Scrollbar extends Component implements Adjustable {
  public static final int HORIZONTAL
  public static final int VERTICAL
  public Scrollbar()
  public Scrollbar(int orientation)
  public Scrollbar(int orientation, int value, int visible, int minimum, int maximum)
  public void addNotify()
  public int getOrientation()
  public void setOrientation(int orientation)
  public int getValue()
  public void setValue(int newValue)
  public int getMinimum()
  public void setMinimum(int newMinimum)
  public int getMaximum()
  public void setMaximum(int newMaximum)
  public int getVisibleAmount()
  public void setVisibleAmount(int newAmount)
  public void setUnitIncrement(int v)
  public int getUnitIncrement()
  public void setBlockIncrement(int v)
  public int getBlockIncrement()
  public void setValues(int value, int visible, int minimum, int maximum)
  public void addAdjustmentListener(AdjustmentListener l)
  public void removeAdjustmentListener(AdjustmentListener l)
  protected void processEvent(AWTEvent e)
  protected void processAdjustmentEvent(AdjustmentEvent e)
  protected String paramString()
}
```

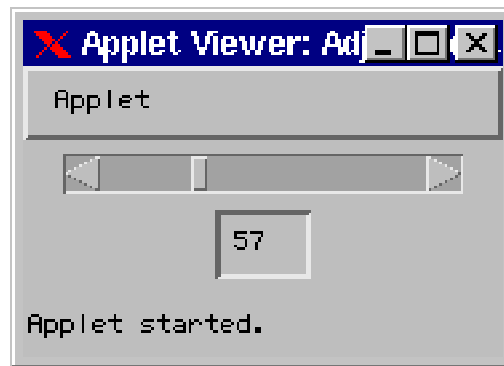

Fig. 26.16: Scrollbar

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Adjustment extends Applet implements AdjustmentListener {
  TextField tf;
  SC scb;
  public void init() {
    scb = new SC();
    tf = new TextField(3);
    add(scb); add(tf);
    scb.addAdjustmentListener(this);
    tf.setEditable(false);
    tf.setText("50");
  }
  public  void adjustmentValueChanged(AdjustmentEvent e) { tf.setText(Integer.toString(e.getValue())); }
}
```

```
class SC extends Scrollbar {
  Dimension minSize;
  public SC() {
    super(Scrollbar.HORIZONTAL, 50, 1, 0, 200);
    minSize = new Dimension(150, 15);
  }
  public Dimension getPreferredSize() { return getMinimumSize(); }
  public synchronized Dimension getMinimumSize() { return minSize; }
}
```

– comme ascenseurs pour contrôler la visiblité d'un composant (`ScrollPane`).

```
public class ScrollPane extends Container {
  public static final int SCROLLBARS_AS_NEEDED
  public static final int SCROLLBARS_ALWAYS
  public static final int SCROLLBARS_NEVER
  public ScrollPane()
  public ScrollPane(int scrollbarDisplayPolicy)
  protected final void addImpl(Component comp, Object constraints, int index)
  public int getScrollbarDisplayPolicy()
  public Dimension getViewportSize()
  public int getHScrollbarHeight()
  public int getVScrollbarWidth()
  public Adjustable getVAdjustable()
  public Adjustable getHAdjustable()
  public void setScrollPosition(int x, int y)
  public void setScrollPosition(Point p)
  public Point getScrollPosition()
  public final void setLayout(LayoutManager mgr)
  public void doLayout()
  public void printComponents(Graphics g)
  public void addNotify()
  public String paramString()
}
```

26.13 Dialog

Les objets de type `Dialog` sont des fenêtres rattachées à une autre fenêtre. Les fenêtres `Dialog` doivent s'iconifier et se fermer en même temps que la fenêtre à laquelle elle est rattachée.

Par défaut, le gestionnaire de placement est le `BorderLayout`.

```
public class Dialog extends Window {
      public Dialog(Frame owner)
      public Dialog(Frame owner, boolean modal)
      public Dialog(Frame owner, String title)
      public Dialog(Frame owner, String title, boolean modal)
      public Dialog(Dialog owner)
      public Dialog(Dialog owner, String title)
      public Dialog(Dialog owner, String title, boolean modal)
      public void addNotify()
      public boolean isModal()
      public void setModal(boolean b)
      public String getTitle()
      public void setTitle(String title)
      public void show()
      public boolean isResizable()
      public void setResizable(boolean resizable)
      protected String paramString()
   }
```

Fig. 26.17: Dialog

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class WDialog extends Applet implements ActionListener {
  Button b;
  DesDialogs fenetre;
  public void init() {
    b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
    add(b);
    b.addActionListener(this);
    fenetre = new DesDialogs("Des boutons");
  }
  public void actionPerformed(ActionEvent e) {
    if (! fenetre.isShowing()) fenetre.setVisible(true);
    else fenetre.setVisible(false);
  }
  public static void main(String[] args) {
    Frame f = new Frame("Des Dialog");
    WDialog p =  new WDialog();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
    f.add(p);
    f.setSize(350, 170);
    f.pack();
    f.show();
  }
}

class DesDialogs extends Frame  implements ActionListener {
  Dialog d1, d2;
  public DesDialogs(String s) {
    super(s);
    setLayout(new FlowLayout());
    Button b = new Button("Fenetre modale");
    b.addActionListener(this);
    add(b);
    setSize(200,200);
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { setVisible(false); }
    });
  }
  public void actionPerformed(ActionEvent e) {
    if (d1 == null) {
      d1 = new FModale(this, "Modal", true);
      d1.setBounds(getLocationOnScreen().x, getLocationOnScreen().y, 400, 100);
    }
    d1.setVisible(true);
```

```
  }
}

class FModale extends Dialog  implements ActionListener {
  public FModale(Frame owner, String title, boolean modal) {
    super(owner, title, modal);
    setLayout(new FlowLayout());
    Button b;
    add(new Label("Message d'alerte"));
    add(b = new Button("O. K ? "));
    b.addActionListener(this);
  }
  public void actionPerformed(ActionEvent e) { setVisible(false); }
}
```

26.14 FileDialog

```
public class FileDialog extends Dialog {
   public static final int LOAD
   public static final int SAVE
   public FileDialog(Frame parent)
   public FileDialog(Frame parent, String title)
   public FileDialog(Frame parent, String title, int mode)
   public void addNotify()
   public int getMode()
   public void setMode(int mode)
   public String getDirectory()
   public void setDirectory(String dir)
   public String getFile()
   public void setFile(String file)
   public FilenameFilter getFilenameFilter()
   public void setFilenameFilter(FilenameFilter filter)
   protected String paramString()
}
```

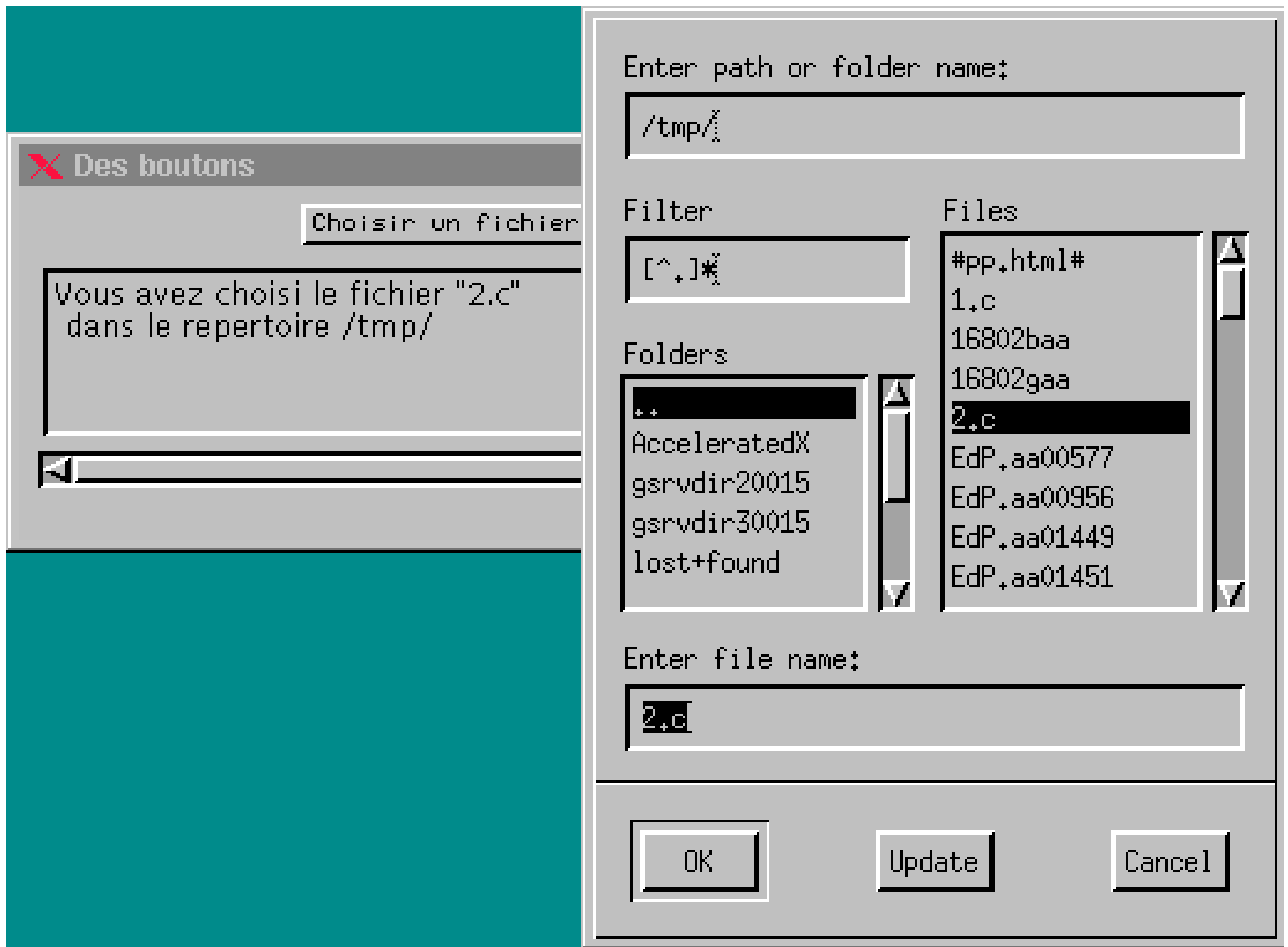

Fig. 26.18: Des FileDialogs

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class WFileDialog extends Applet implements ActionListener {
  Button b;
  DesFDialogs fenetre;
  public void init() {
    b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
    add(b);
    b.addActionListener(this);
    fenetre = new DesFDialogs("Des boutons");
  }
  public void actionPerformed(ActionEvent e) {
    if (! fenetre.isShowing()) fenetre.setVisible(true);
    else fenetre.setVisible(false);
  }
  public static void main(String[] args) {
    Frame f = new Frame("Des FileDialog");
    WFileDialog p =  new WFileDialog();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
```

```
      f.add(p);
      f.setSize(350, 170);
      f.pack();
      f.show();
    }
  }

  class DesFDialogs extends Frame  implements ActionListener {
    FileDialog d1, d2;
    String rc = System.getProperty("line.separator");
    TextArea ta;
    public DesFDialogs(String s) {
      super(s);
      setLayout(new FlowLayout());
      Button b = new Button("Choisir un fichier");
      b.addActionListener(this);
      add(b);

      ta = new TextArea(5,50);
      ta.setEditable(false);
      add(ta);
      pack();
      addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) { setVisible(false); }
      });
    }
    void afficher(String s) { ta.append(s + rc); ta.setCaretPosition(2000); }
    public void actionPerformed(ActionEvent e) {
      if (e.getSource() == d1)
        System.out.println(e);
      else {
        if (d1 == null) d1 = new FileDialog(this, "File Dialog", FileDialog.LOAD);
        d1.setDirectory("/tmp");
        d1.setVisible(true);
        afficher("Vous avez choisi le fichier \"" + d1.getFile() +
         "\"\n dans le repertoire " + d1.getDirectory());
      }
    }
  }
```

26.15 ScrollPane

26.16 Composants légers (Ligthweight)

Lorsque l'on crée un nouveau composant, celui-ci est souvent dérivé de `Canvas` ou `Panel`. Ces composants sont intimement liés aux fenêtrex du système utilisé (fenêtres natives). Il en résulte :

- les fenêtres natives sont des composants lourds et il est préférable de minimiser leur nombre.
- les fenêtres natives sont opaques et il n'est donc pas possible de définir des composants qui utilisent la transparence
- les fenêtres natives se comportent parfois différemment selon l'architecture du système utilisé.

Les composants légers ont été introduits pour pallier ces défauts. Il s'agit d'objets dérivés directement de la classe `Component` et `Container`. Il n'y donc pas de zone opaque associée à ces composants. Ces composants gardent toutes les propriétés classiques des composants : affichage, gestion de configuration (layout) et gestion d'évènements. Ils ont en outre les avantage suivants :

- La transparence est assurée par le fait de ne pas définir une couleur pour le fond du composant
- Ces composants sont légers au sens où il ne nécessite pas un objet `peer`.
- Ces composants sont entièrement programmés en Java et donc parfaitement portables.

Composants légers et lourds Par opposition aux composants légers, nous appellerons *composants lourds* (*Heavyweight Components*) les composants classiques que nous avons décrit tout au long de ce chapitre.

Une application peut faire cohabiter les composants légers et lourds. Un container lourd peut contenir des composants légers, un composant lourd peut être contenu dans un container léger. La seule restriction est que le container racine soit un container lourd.

Affichage et gestion d'évènements L'affichage et la gestion des évènements des composants légers se fait par l'intermédiaire d'un `Container`. Pour gérer l'affichage, il suffit d'implanter correctement la méthode `paint`. En effet, si le container est un objet dérivé de la classe `Container` et que la méthode `paint()` du container est redéfinie, il ne faut surtout pas oublier de faire appel à la méthode `super.paint()` sous peine de ne pas voir afficher vos composants légers.

Optimiser l'affichage Comme vous le savez, à présent, les composants légers sont affichés en les dessinant dans une méthode `paint`. Aussi, nous retrouverons pour l'affichage des composants légers, les mêmes problèmes que nous avons rencontré pour dessiner dans une fenêtre graphique (voir 28). Une bonne habitude de programmation consiste à utiliser le double buffering (voir 28.6) pour afficher les composants légers.

Voici l'exemple fourni dans la documentation officielle de *JDK* :

```
public class DoubleBufferPanel extends Panel {
  Image offscreen;
  public void invalidate() {
    super.invalidate();
    offscreen = null;
  }

  public void update(Graphics g) { paint(g); }
  public void paint(Graphics g) {
    if(offscreen == null)
      offscreen = createImage(getSize().width, getSize().height);
    Graphics og = offscreen.getGraphics();
    og.setClip(0,0,getSize().width, getSize().height);
    super.paint(og);
    g.drawImage(offscreen, 0, 0, null);      og.dispose();
  }
}
```

Un exemple de bouton

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

class LWButton extends Component {
   String label;
   protected boolean pressed = false;
   public LWButton(String label) {
      this.label = label;
      enableEvents(AWTEvent.MOUSE_EVENT_MASK);
   }

   public void paint(Graphics g) {
         FontMetrics fm = getFontMetrics(getFont());
      int s = Math.min(getSize().width - 1, getSize().height - 1);
      if(pressed) g.setColor(getBackground().darker().darker());
      else g.setColor(getBackground());

      g.fillRect(0, 0, fm.stringWidth(label), fm.getHeight() );
      g.setColor(getBackground().darker().darker().darker());
      g.drawRect(0, 0, fm.stringWidth(label), fm.getHeight() );
      Font f = getFont();
      if (f != null) {
         fm = getFontMetrics(getFont());
         g.setColor(getForeground());
         g.drawString(label, s/2 - fm.stringWidth(label)/2, s/2 + fm.getMaxDescent());
      }
   }

   public Dimension getPreferredSize() {
      Font f = getFont();
      if(f != null) {
         FontMetrics fm = getFontMetrics(getFont());
         int max = Math.max(fm.stringWidth(label) + 40, fm.getHeight() + 40);
//          return new Dimension(max, max);
         return new Dimension(fm.stringWidth(label) + 40, fm.getHeight() + 40);
      }
      else return new Dimension(100, 100);
   }
```

```
      public Dimension getMinimumSize() {
         return new Dimension(100, 100);
      }

      public void processMouseEvent(MouseEvent e) {
         Graphics g;
         switch(e.getID()) {
            case MouseEvent.MOUSE_PRESSED:
               pressed = true;
               repaint();
               break;
            case MouseEvent.MOUSE_RELEASED:
               if(pressed == true) {
                  pressed = false;

                  repaint();
               }
               break;
/*
            case MouseEvent.MOUSE_EXITED:
               if(pressed == true) {
                  pressed = false;
                  repaint();
               }
               break;
*/
         }
         super.processMouseEvent(e);
      }
   }

   public class Leger extends Applet {
      public void init() {
         add(new LWButton("cccccccccccc"));
      }
   }
```

A TERMINER

Un exemple de bouton animé

27. Ranger les widgets

Sommaire

27.1 Gestion “manuelle” des placements 247
27.2 Généralités sur les Layouts 249
27.3 FlowLayout 250
27.4 BorderLayout 252
27.5 CardLayout 254
27.6 GridLayout 256
27.7 GridBagLayout 258
27.8 L’exemple du tutorail 260
27.9 Créer ses propres layouts 261

Nous avons, volontairement, jusqu’ici négligé le problème de la gestion de la disposition des composants dans un container. Comme vous avez pu le constater, dans tous les exemples que nous avons vus jusqu’ici, nous nous sommes contenté d’ajouter des composants dans un container sans préciser quoi ce soit, quant à la position de ce composant dans le container.

Comme nous l’avons déjà signalé rapidement, certains containers disposent par défaut une gestion de placement (*Layout Manager*) qui se charge de placer les composants “correctement”.

Qu’entend-t-on pas correctement ? Le problème essentiel que doit résoudre un gestionnaire de placement est de déterminer la position et la taille des composants les uns par rapport aux autres et ce quelle que soit la taille de la fenêtre dans laquelle ils s’inscrivent.

Cour ce faire, il n’existe pas une seule manière de gérer le placement des composants ; il existe plusieurs politiques de placement des composants dans un container. Aussi *Java* propose plusieurs type de gestionnaires.

Par exemple, le gestionnaire

- `FlowLayout` est associée aux `Panel`.
- `BorderLayout` est associée aux `Window`.

27.1 Gestion “manuelle” des placements

Avant d’entrer dans la description et l’utilisation des divers gestionnaires de placement, remarquons qu’il est tout à fait possible de se passer d’un gestionnaire. En effet, si l’on est très courageux, il suffira de placer “à la main”les différents composants : en donnant la taille et la position que chaque composant doit avoir dans un container.

Cette manière de faire présente deux défauts majeurs :

- elle oblige de déterminer de manière précise la configuration des différents composants ; tâche souvent très fastidieuse.
- la taille par défaut des composants peut varier d’un système à l’autre et la belle configuration sur un système peut avoir une allure catastrophique sur un autre système.

Aussi, il est toujours conseillé d’utiliser un gestionnaire de placement. Si malgré tout, on veut se passer de gestionnaire de placement, il suffira de préciser qu’un container ne possède pas de gestionnaire de placement en invoquant la méthode

```
setLayout(null);
```

Avec cet appel, le programmeur prendre la responsabilité de placer correctement les composants. Il lui appartient de définir précisément où les composant s’inscrivent dans un container en précisant avec la méthode `setBounds`.

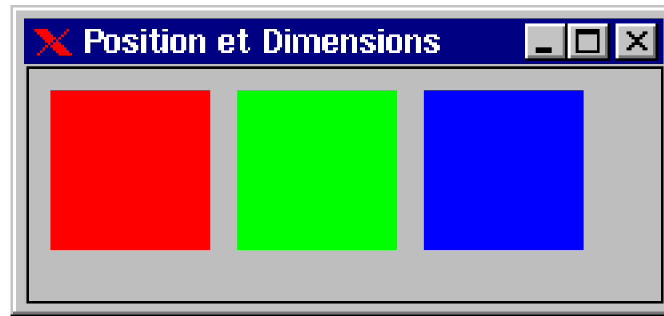

FIG. 27.1: Sans layout

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Null extends Applet  {
  Panel p1, p2, p3;
  public void init() {
    setLayout(null);
    add(p1 = new Panel());
    add(p2 = new Panel());
    add(p3 = new Panel());
    p1.setBackground(Color.red);
    p2.setBackground(Color.green);
    p3.setBackground(Color.blue);
    p1.setBounds(10, 10, 60, 60);
    p2.setBounds(80, 10, 60, 60);
    p3.setBounds(150, 10, 60, 60);
  }
  public void paint(Graphics g) {
    Dimension d = getSize();
    g.drawRect(1,1, d.width-2, d.height-2);
  }
  public static void main(String[] args) {
    Frame f = new Frame("Position et Dimensions");
    Null p =  new Null();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
    f.add(p);
    f.pack();
    f.setSize(250, 120);
    f.show();
  }
}
```

Reprenons l’exemple (voir 26.2) précédemment vu et supprimons le gestionnaire de placement que nous avons utilisé. Rappelons que ce gestionnaire se chargeait et du placement des boutons et la gestion de la taille de ces dernier en fonction de la modification de leur étiquette. En supprimant ce gestionnaire, il appartient au programmeur d’effectuer toutes ses taches.

```
public class TroisBoutons extends Applet implements ActionListener {
  ...
  public void init() {
    ...
    setLayout(null);
  }
  public void paint(Graphics g) {
    int p0, p1, p2;
    int x0, x1, x2;
    p0 = 5*(b[0].getLabel().length()+4); x0 = 10;
    p1 = 5*(b[1].getLabel().length()+4); x1 = x0 + 10 + p0 + 10;
    p2 = 5*(b[2].getLabel().length()+4); x2 = x1 + 10 + p1 + 10;
    b[0].setBounds(x0, 10, p0, 30);
    b[1].setBounds(x1, 10, p1, 30);
    b[2].setBounds(x2, 10, p2, 30);
  }
```

```
    public void actionPerformed(ActionEvent e) {
      ...
      repaint();
      }
    ...
  }
```

Vous aurez compris que cette façon de faire est évidemment fastidieuse : il faut calculer “à la main” la taille des *widgets*, leur position, leur redimensionnement, etc. Cette programmation peut s’avérer relativement délicate si l’on tient compte des problèmes de redimentionnement de la fenêtre, de la tailles des fontes, etc.

Pour simplifier cette programmation, *Java* offre la possibilité de spécifier la disposition globale que l’on désire obtenir et laisser le soin au gestionnaire de placement le soin de gérer leur position et leur redimensionnement.

27.2 Généralités sur les Layouts

Un gestionnaire de placement est une instance d’une classe qui implante l’un des deux interfaces `LayoutManager` ou `LayoutManager2`.

```
public interface LayoutManager {
   public void addLayoutComponent(String name, Component comp)
   public void removeLayoutComponent(Component comp)
   public Dimension preferredLayoutSize(Container parent)
   public Dimension minimumLayoutSize(Container parent)
   public void layoutContainer(Container parent)
}

public interface LayoutManager2 extends LayoutManager {
   public void addLayoutComponent(Component comp, Object constraints)
   public Dimension maximumLayoutSize(Container target)
   public float getLayoutAlignmentX(Container target)
   public float getLayoutAlignmentY(Container target)
   public void invalidateLayout(Container target)
}
```

Sauf contre indication, lors de la création d’un container, il lui est automatiquement associé un `LayoutManger` par défaut. *Java* fournit plusieurs type de gestionnaires :

- `FlowLayout`
- `BorderLayout`
- `CardLayout`
- `GridLayout`
- `GridBagLayout`

Si l’on veut associer un `LayoutManager` particulier à un container, on invoque la méthode `setLayout`.

```
FlowLayout gestionnaire = new FlowLayout();
unContainer.setLayout(gestionnaire);
```

Chaque container dispose d’une ou de plusieurs méthodes **`add`** pour ajouter un composant dans un container. Cette méthode fait appel au `LayoutMagener` associé au container pour la gestion de la configuration des composants. De même, les méthodes **`remove`** et **`removeAll`** permettent de supprimer des composants d’un container.

Les méthodes `getPreferredSize` et `getMinimumSize` retournent la taille idéale et la taille minimale d’un composant. Lorsqu’on choisit de donner à un composant une taille non standard, il faut préciser, en rédéfinissant ces méthodes, la taille idéale et celle minimale que l’on veut voir attribuer à notre composant.

```
class Bouton extends Button {
  Dimension d;
  public Bouton(String s, int x, int y) { super(s); d = new Dimension(x, y); }
  public Dimension getMinimumSize() { return d; }
  public Dimension getPreferredSize() { return getMinimumSize(); }
}
```

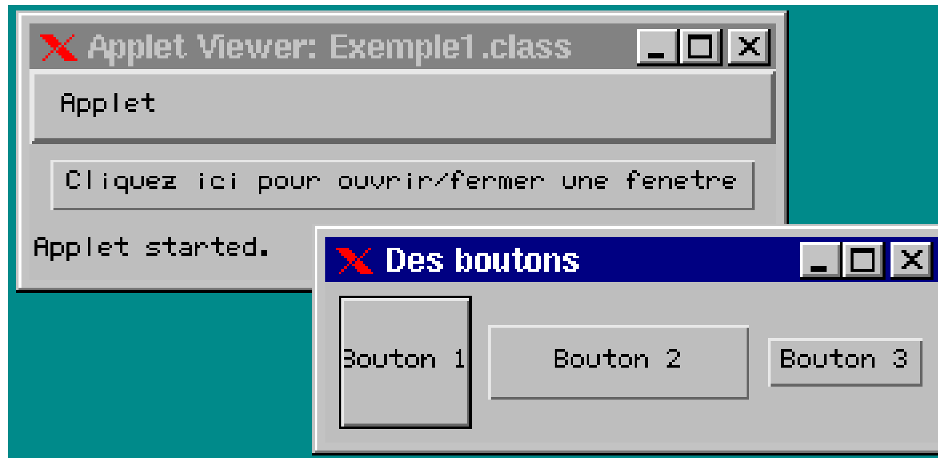

Fig. 27.2: Composants de taille non standard

La configuration choisie par un `LayoutManager` peut devenir obsolète dans les situations suivantes :

- Modification de la taille du container
- Modification de la taille ou de la position d'un des composants d'un container
- Ajout, suppression, affichage ou masquage d'un des composants d'un container

Il convient alors de demander au `LayoutManager` de trouver une nouvelle configuration ; ce qui se fait en invoquant la méthode `validate` (ou `doLayout`) du container. La méthode `validate` (ou `doLayout`) se charge de disposer "correctement" tous les composants contenus dans le container.

27.3 FlowLayout

Le gestionnaire `FlowLayout` dispose les composants de gauche à droite dans l'ordre de leur insertion dans le container. Lorsque la taille du container est trop petite pour contenir tous les composants sur une même ligne, le gestionnaire passe à la ligne suivante ; chaque ligne est centrée.

```
public class FlowLayout extends Object implements LayoutManager, Serializable {
   public static final int LEFT
   public static final int CENTER
   public static final int RIGHT
   public FlowLayout()
   public FlowLayout(int align)
   public FlowLayout(int align, int hgap, int vgap)
   public int getAlignment()
   public void setAlignment(int align)
   public int getHgap()
   public void setHgap(int hgap)
   public int getVgap()
   public void setVgap(int vgap)
   public void addLayoutComponent(String name, Component comp)
   public void removeLayoutComponent(Component comp)
   public Dimension preferredLayoutSize(Container target)
   public Dimension minimumLayoutSize(Container target)
   public void layoutContainer(Container target)
   public String toString()
}
```

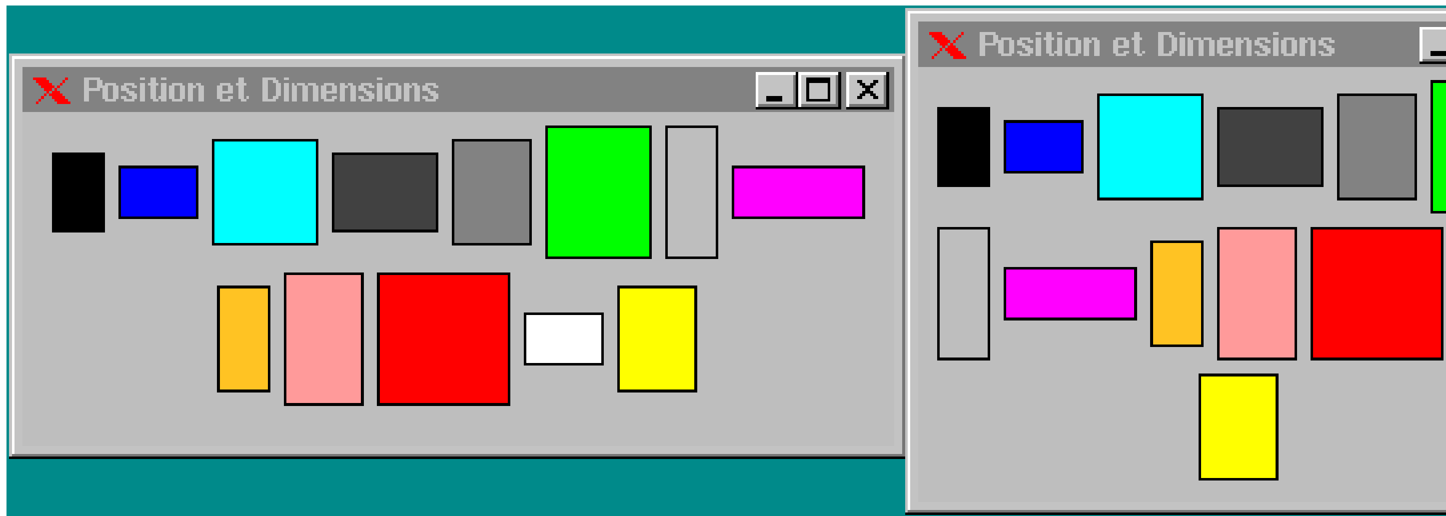

Fig. 27.3: FlowLayout

Redimensionner la fenêtre contenant les divers panels pour voir les composants se placer “correctement”.

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

public class LFlow extends Applet implements ActionListener {
  Button b;
  FenetreFlow fenetre;
  public void init() {
    b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
    add(b);
    b.addActionListener(this);
    fenetre = new FenetreFlow("FlowLayout Manager");
  }
  public void actionPerformed(ActionEvent e) {
    if (! fenetre.isShowing()) fenetre.setVisible(true);
    else fenetre.setVisible(false);
  }
  public static void main(String[] args) {
    Frame f = new Frame("Position et Dimensions");
    LFlow p =  new LFlow();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
    f.add(p);
    f.pack();
    f.show();
  }
}

class FenetreFlow extends Frame  {
  static Color [] couleurs = {Color.black, Color.blue, Color.cyan,
       Color.darkGray, Color.gray, Color.green,
       Color.lightGray, Color.magenta, Color.orange,
       Color.pink, Color.red, Color.white, Color.yellow};
  static Dimension [] dim = {new Dimension(20, 30), new Dimension(30, 20),
                             new Dimension(40, 40), new Dimension(40, 30),
     new Dimension(30, 40), new Dimension(40, 50),
     new Dimension(20, 50), new Dimension(50, 20),
     new Dimension(20, 40), new Dimension(30, 50),
     new Dimension(50, 50), new Dimension(30, 20),
     new Dimension(30, 40)};
  public  FenetreFlow(String s) {
    super(s);
```

```
    setLayout(new FlowLayout());
    for (int i = 0; i<couleurs.length; i++) add(new MonPanel(couleurs[i], dim[i]));
    setSize(250, 200);
  }
}
class MonPanel extends Panel {
  Dimension d;
  public MonPanel(Color c, Dimension dim) { setBackground(c);  d = dim; }
  public Dimension getMinimumSize() { return d; }
  public Dimension getPreferredSize() { return getMinimumSize(); }
  public void paint(Graphics g) {
    Dimension d = getSize(); g.drawRect(0,0, d.width-1, d.height-1);
  }
}
```

27.4 BorderLayout

```
public class BorderLayout extends Object implements LayoutManager2, Serializable {
   public static final String NORTH
   public static final String SOUTH
   public static final String EAST
   public static final String WEST
   public static final String CENTER
   public BorderLayout()
   public BorderLayout(int hgap, int vgap)
   public int getHgap()
   public void setHgap(int hgap)
   public int getVgap()
   public void setVgap(int vgap)
   public void addLayoutComponent(Component comp, Object constraints)
   public void addLayoutComponent(String name, Component comp)
   public void removeLayoutComponent(Component comp)
   public Dimension minimumLayoutSize(Container target)
   public Dimension preferredLayoutSize(Container target)
   public Dimension maximumLayoutSize(Container target)
   public float getLayoutAlignmentX(Container parent)
   public float getLayoutAlignmentY(Container parent)
   public void invalidateLayout(Container target)
   public void layoutContainer(Container target)
   public String toString()
}
```

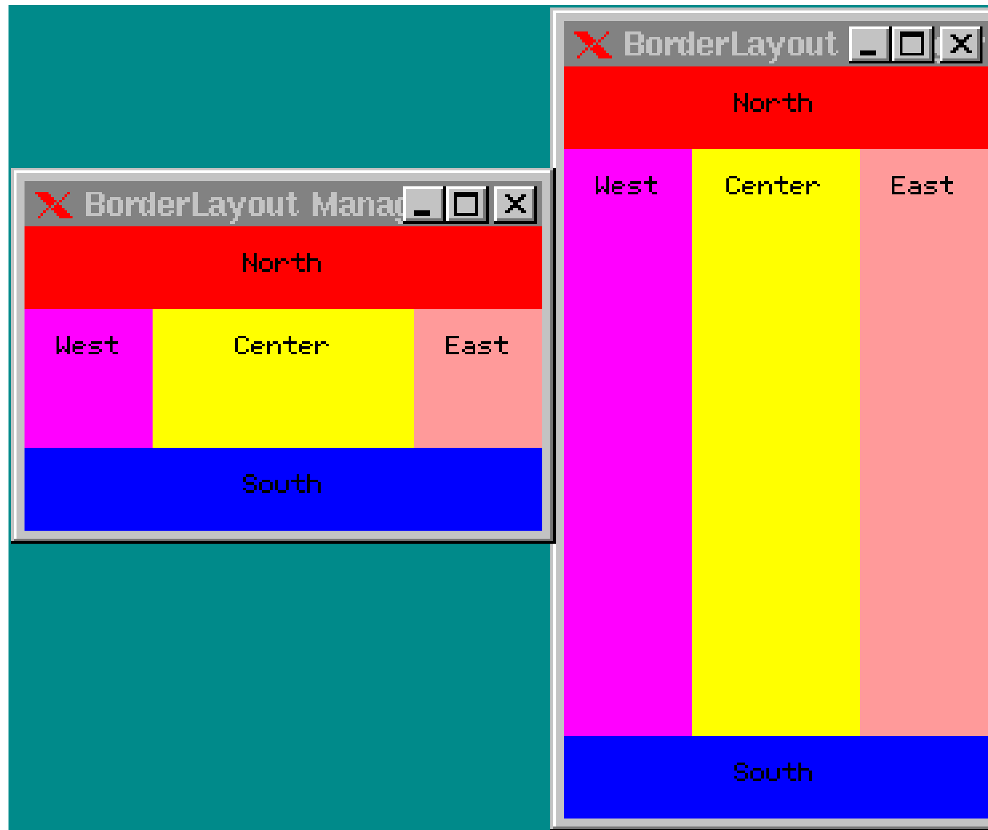

Fig. 27.4: BorderLayout

Redimentionner la fenêtre contenant les divers “panels” pour voir les composants se placer “correctement”.

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

public class LBorder extends Applet implements ActionListener {
  Button b;
  FenetreBorder fenetre;
  public void init() {
    b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
    add(b);
    b.addActionListener(this);
    fenetre = new FenetreBorder("BorderLayout Manager");
  }
  public void actionPerformed(ActionEvent e) {
    if (! fenetre.isShowing()) fenetre.setVisible(true);
    else fenetre.setVisible(false);
  }
  public static void main(String[] args) {
    Frame f = new Frame("Position et Dimensions");
    LBorder p =  new LBorder();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
    f.add(p);
    f.pack();
    f.show();
  }
}
```

```
class FenetreBorder extends Frame  {
  public  FenetreBorder(String  s) {
    super(s);
    setSize(200, 150);
    setLayout(new BorderLayout());

    add(new MonPanel("1/5", Color.blue), "North");
    add(new MonPanel("2/5", Color.red), "East");
    add(new MonPanel("3/5", Color.pink), "South");
    add(new MonPanel("4/5", Color.magenta), "West");
    add(new MonPanel("5/5", Color.yellow), "Center");
  }
}


class MonPanel extends Panel {
  public MonPanel(String s, Color c) {
    super(new BorderLayout());
    setBackground(c);
    add("Center", new Label(s, Label.CENTER));
  }
}
```

27.5 CardLayout

Le gestionnaire **CardLayout** range les composants comme on pourrait le faire avec un paquet de cartes. Un seul composant est visible à la fois.

```
public class CardLayout extends Object implements LayoutManager2, Serializable {
   public CardLayout()
   public CardLayout(int hgap, int vgap)
   public int getHgap()
   public void setHgap(int hgap)
   public int getVgap()
   public void setVgap(int vgap)
   public void addLayoutComponent(Component comp, Object constraints)
   public void addLayoutComponent(String name, Component comp)
   public void removeLayoutComponent(Component comp)
   public Dimension preferredLayoutSize(Container parent)
   public Dimension minimumLayoutSize(Container parent)
   public Dimension maximumLayoutSize(Container target)
   public float getLayoutAlignmentX(Container parent)
   public float getLayoutAlignmentY(Container parent)
   public void invalidateLayout(Container target)
   public void layoutContainer(Container parent)
   public void first(Container parent)
   public void next(Container parent)
   public void previous(Container parent)
   public void last(Container parent)
   public void show(Container parent, String name)
   public String toString()
}
```

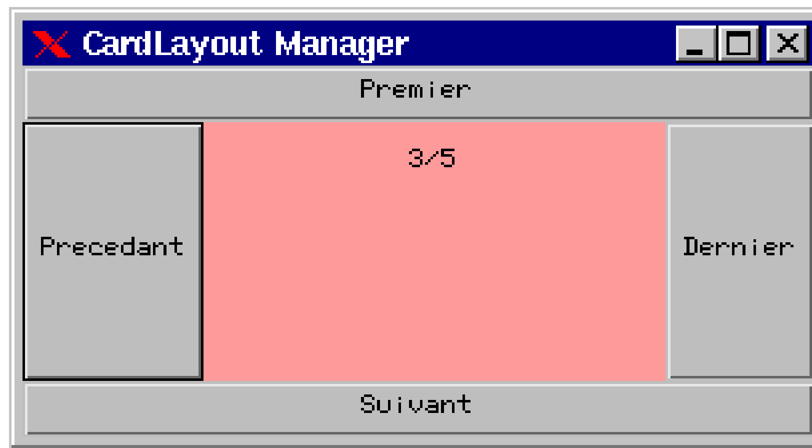

Fig. 27.5: CardLayout

Cliquez sur les boutons pour voir rendre visible le composant souhaité.

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

public class LCard extends Applet implements ActionListener {
  Button b;
  FenetreCard fenetre;
  public void init() {
    b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
    add(b);
    b.addActionListener(this);
    fenetre = new FenetreCard("CardLayout Manager");
  }
  public void actionPerformed(ActionEvent e) {
    if (! fenetre.isShowing()) fenetre.setVisible(true);
    else fenetre.setVisible(false);
  }
  public static void main(String[] args) {
    Frame f = new Frame("Position et Dimensions");
    LCard p =  new LCard();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
    f.add(p);
    f.pack();
    f.show();
  }
}

class FenetreCard extends Frame  implements ActionListener {
  CardLayout card;
  FenetreCard1 f;
  Button b1, b2, b3, b4;
  public  FenetreCard(String s) {
    super(s);
    card = new CardLayout();
    setLayout(new BorderLayout());
    add("Center", f = new FenetreCard1(card));
    add("North", b1 = new Button("Premier"));
    add("East", b2 = new Button("Dernier"));
    add("South", b3 = new Button("Suivant"));
    add("West", b4 = new Button("Precedant"));
    b1.addActionListener(this);
    b2.addActionListener(this);
    b3.addActionListener(this);
    b4.addActionListener(this);
```

```
    setSize(250, 200);
  }
  public void actionPerformed(ActionEvent e) {
System.out.println(e.getActionCommand());
    if ("Premier".equals(e.getActionCommand()))  card.first(f);
    else if ("Dernier".equals(e.getActionCommand())) card.last(f);
    else if ("Suivant".equals(e.getActionCommand())) card.next(f);
    else card.previous(f);
  }

}
class FenetreCard1 extends Panel  {
  public  FenetreCard1(CardLayout c) {
    setLayout(c);
    Panel p0=new Panel(), p1=new Panel(), p2=new Panel(), p3=new Panel(), p4=new Panel();
    add("1", p0); p0.setBackground(Color.blue);  p0.add(new Label("1/5", Label.CENTER));
    add("2", p1); p1.setBackground(Color.red);  p1.add(new Label("2/5", Label.CENTER));
    add("3", p2); p2.setBackground(Color.pink);  p2.add(new Label("3/5", Label.CENTER));
    add("4", p3); p3.setBackground(Color.magenta);  p3.add(new Label("4/5", Label.CENTER));
    add("5", p4); p4.setBackground(Color.yellow);  p4.add(new Label("5/5", Label.CENTER));
  }
}
```

27.6 GridLayout

Le gestionnaire `GridLayout` dispose les composants à l'intérieure d'une grille dont chaque case est de taille égale et de telle sorte que chaque composant occupe la plus grande place possible dans une case de la grille.

```
public class GridBagLayout extends Object implements LayoutManager2, Serializable {
      protected static final int MAXGRIDSIZE
      protected static final int MINSIZE
      protected static final int PREFERREDSIZE
      protected Hashtable comptable
      protected GridBagConstraints defaultConstraints
      protected GridBagLayoutInfo layoutInfo
      public int[] columnWidths
      public int[] rowHeights
      public double[] columnWeights
      public double[] rowWeights
      public GridBagLayout()
      public void setConstraints(Component comp, GridBagConstraints constraints)
      public GridBagConstraints getConstraints(Component comp)
      protected GridBagConstraints lookupConstraints(Component comp)
      public Point getLayoutOrigin()
      public int[][] getLayoutDimensions()
      public double[][] getLayoutWeights()
      public Point location(int x, int y)
      public void addLayoutComponent(String name, Component comp)
      public void addLayoutComponent(Component comp, Object constraints)
      public void removeLayoutComponent(Component comp)
      public Dimension preferredLayoutSize(Container parent)
      public Dimension minimumLayoutSize(Container parent)
      public Dimension maximumLayoutSize(Container target)
      public float getLayoutAlignmentX(Container parent)
      public float getLayoutAlignmentY(Container parent)
      public void invalidateLayout(Container target)
      public void layoutContainer(Container parent)
      public String toString()
      protected GridBagLayoutInfo GetLayoutInfo(Container parent, int sizeflag)
      protected void AdjustForGravity(GridBagConstraints constraints, Rectangle r)
      protected Dimension GetMinSize(Container parent, GridBagLayoutInfo info)
      protected void ArrangeGrid(Container parent)
   }
```

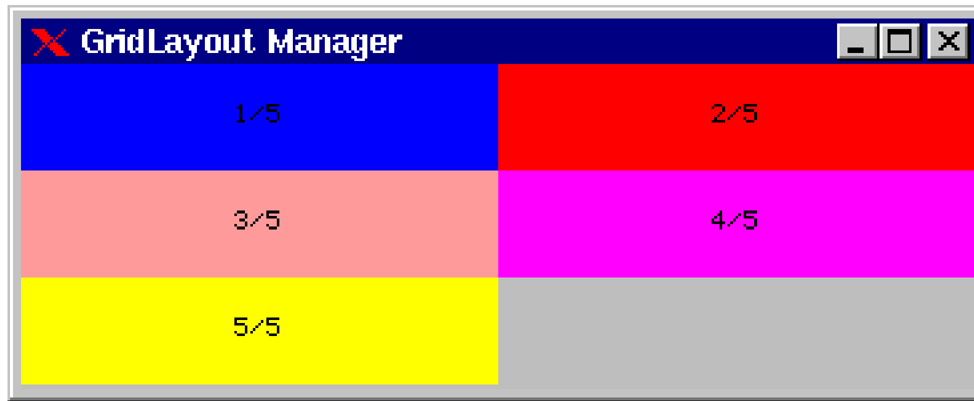

Fig. 27.6: GridLayout

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

public class LGrid extends Applet implements ActionListener {
  Button b;
  FenetreGrid fenetre;
  public void init() {
    b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
    add(b);
    b.addActionListener(this);
    fenetre = new FenetreGrid("GridLayout Manager");
  }
  public void actionPerformed(ActionEvent e) {
    if (! fenetre.isShowing()) fenetre.setVisible(true);
    else fenetre.setVisible(false);
  }
  public static void main(String[] args) {
    Frame f = new Frame("Position et Dimensions");
    LGrid p =  new LGrid();
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    p.init();
    f.add(p);
    f.pack();
    f.show();
  }
}

class FenetreGrid extends Frame  {
  public  FenetreGrid(String s) {
    super(s);
    setLayout(new GridLayout(0,2));
    add(new MonPanel("1/5", Color.blue));
    add(new MonPanel("2/5", Color.red));
    add(new MonPanel("3/5", Color.pink));
    add(new MonPanel("4/5", Color.magenta));
    add(new MonPanel("5/5", Color.yellow));
    setSize(250, 200);
  }
}


class MonPanel extends Panel {
  public MonPanel(String s, Color c) {
    super(new BorderLayout());
    setBackground(c);
    add("Center", new Label(s, Label.CENTER));
  }
```

```
}
```

27.7 GridBagLayout

Le gestionnaire **GridBagLayout** permet une gestion plus fine du placement des composants. Il permet d'aligner les composants à l'intérieure d'une grille sans imposer que chaque case de la grille soit de même taille et où une composant peut éventuellement occuper plusieurs cases.

A chaque composant géré par le **GridBagLayout** est associé un ensemble de contraintes d'occupation sur la grille. La spécification des contraintes associées à un composant sont données par une instance de la classe **GridBagConstraints**.

```
...
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints c = new GridBagConstraints();
setLayout(gridbag);
...
```

La disposition des composants par un **GridBagLayout** dépend des contraintes données par le **GridBagConstraints** et par les tailles minimales (**getMinimumSize**) et préférées (**getPreferredSize**).

```
...
gridbag.setConstraints(unComposant, c);
add(unComposant);
...
```

La classe **GridBagConstraints** définit un ensemble de variables d'instance qui permettent de spécifier les contraintes de placement des composants.

```
public class GridBagConstraints implements Cloneable, Serializable {
   public static final int RELATIVE
   public static final int REMAINDER
   public static final int NONE
   public static final int BOTH
   public static final int HORIZONTAL
   public static final int VERTICAL
   public static final int CENTER
   public static final int NORTH
   public static final int NORTHEAST
   public static final int EAST
   public static final int SOUTHEAST
   public static final int SOUTH
   public static final int SOUTHWEST
   public static final int WEST
   public static final int NORTHWEST
   public int gridx
   public int gridy
   public int gridwidth
   public int gridheight
   public double weightx
   public double weighty
   public int fill
   public Insets insets
   public int anchor
   public int ipadx
   public int ipady
   public GridBagConstraints()
   public Object clone()
}
```

`public int gridx, gridy`

éfinit la ligne et la colonne de la case supérieure gauche occupé par le composant (numérotaion de 0 à n-1). On pourra utiliser la valeur **GridBagContraints.RELATIVE** pour **gridx** (resp. **gridy**) pour placer un composant à droite (resp. en bas) du composant précédemment ajouté.

`public int gridwidth, gridheight`

éfinit le nombre de colonnes et de lignes occupé par le composant. Par défaut, **gridwidth** et **gridheight** valent 1. On pourra utiliser la valeur

- **GridBagContraints.REMAINDER** pour **gridwidth** (resp. **gridheight**) pour que le composant soit le dernier la ligne (resp. de la colonne).
- **GridBagContraints.RELATIVE** pour **gridwidth** (resp. **gridheight**) pour que le composant soit le suivant du dernier la ligne (resp. de la colonne).

`public int fill`

a varibale d'instance `fill` précise ce qu'il faut faire lorsque l'espace associé à un composant est plus grand que le composant. Les valeurs permises pour cette variable sont

- `GridBagConstraints.NONE` (valeur par défaut)
- `GridBagConstraints.HORIZONTAL` : le composant occupe toute la largeur de l'espace réservé.
- `GridBagConstraints.VERTICAL` : le composant occupe toute la hauteur de l'espace réservé.
- `GridBagConstraints.BOTH` : le composant occupe tout l'espace réservé.

`public Insets insets`

éfinit l'espace entre le composant et le bord des cases de la grille

`public int anchor`

éfinit, lorsque le composant est plus petit que la case de la grille qui le contient, la position du composant dans l'espace qui est associé. Les valeurs permises sont :
`GridBagConstraints.CENTER` (valeur par défaut),
`GridBagConstraints.NORTH`,
`GridBagConstraints.NORTHEAST`,
`GridBagConstraints.EAST`,
`GridBagConstraints.SOUTHEAST`,
`GridBagConstraints.SOUTH`,
`GridBagConstraints.SOUTHWEST`,
`GridBagConstraints.WEST`,
`GridBagConstraints.NORTHWEST`
Les identificateurs sont assez parlants pour se passer d'explication.

`public int ipadx, ipady`

es valeurs de `ipadx` et `ipady` définissent l'espace, autour d'un composant, à ajouter à la taille minimale du composant.

`public int weightx, weighty`

es valeurs de `weightx` et `weighty` permettent de spécifier la manière de disposer l'espace libre autour d'un composant dans une case de la grille.

Voici, pour finir, un exemple pour illustrer ce gestionnaire.

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

public class BagLayout extends Applet implements ActionListener {
   MonFrame fenetre;
   public void init() {
      Button b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
      add(b);
      b.addActionListener(this);
      fenetre = new MonFrame("Une fenetre");
      fenetre.setSize(300, 100);
      fenetre.setVisible(false);
      fenetre.addWindowListener(
            new WindowAdapter() {
               public void windowClosing(WindowEvent e) {
                  fenetre.setVisible(false); }
            });
   }
   public void actionPerformed(ActionEvent e) {
      if (! fenetre.isShowing()) fenetre.setVisible(true);
      else fenetre.setVisible(false);
   }
}

class UnComposant extends Label {
   public UnComposant(String s, GridBagLayout gridbag, GridBagConstraints cstr, Color c) {
      super(s); gridbag.setConstraints(this, cstr); setBackground(c);
      setAlignment(Label.CENTER);
  }
}
```

```
   class MonFrame extends Frame {
      public MonFrame(String s) {
         super("GridBagLayout");
         setLayout(new GridBagLayout());
         GridBagLayout gridbag = new GridBagLayout();
         GridBagConstraints c = new GridBagConstraints();
         setLayout(gridbag);

         c.fill = GridBagConstraints.BOTH;
         c.weightx = 1;
         add(new UnComposant("Button1",  gridbag, c, Color.red));
         add(new UnComposant("Button2",  gridbag, c, Color.blue));
         add(new UnComposant("Button3",  gridbag, c, Color.green));

         c.gridx = 0;
         c.gridy = 1;
         c.fill = GridBagConstraints.NONE;
         add(new UnComposant("Button4",  gridbag, c, Color.cyan));

         c.gridx = 1;
         c.gridwidth = 2;
         c.gridheight = 2;
         c.fill = GridBagConstraints.HORIZONTAL;
         add(new UnComposant("Button5",  gridbag, c, Color.magenta));

//         c.gridx = GridBagConstraints.RELATIVE;
         c.gridwidth = 1;
         c.gridheight = 1;
         c.gridy = GridBagConstraints.RELATIVE;
         add(new UnComposant("Button6",  gridbag, c, Color.yellow));
      }
   }
```

27.8 L'exemple du tutorail

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

public class GridBagWindow extends Frame {
    boolean inAnApplet = true;

    protected void makebutton(String name,
                              GridBagLayout gridbag,
                              GridBagConstraints c) {
        Button button = new Button(name);
        gridbag.setConstraints(button, c);
        add(button);
    }

    public GridBagWindow() {
        GridBagLayout gridbag = new GridBagLayout();
        GridBagConstraints c = new GridBagConstraints();

        setFont(new Font("SansSerif", Font.PLAIN, 14));
        setLayout(gridbag);

        c.fill = GridBagConstraints.BOTH;
        c.weightx = 1.0;
        makebutton("Button1", gridbag, c);
        makebutton("Button2", gridbag, c);
        makebutton("Button3", gridbag, c);

        c.gridwidth = GridBagConstraints.REMAINDER; //end of row
        makebutton("Button4", gridbag, c);
```

```
        c.weightx = 0.0;                   //reset to the default
        makebutton("Button5", gridbag, c); //another row

        c.gridwidth = GridBagConstraints.RELATIVE; //next-to-last
        makebutton("Button6", gridbag, c);

        c.gridwidth = GridBagConstraints.REMAINDER; //end of row
        makebutton("Button7", gridbag, c);

        c.gridwidth = 1;                   //reset to the default
        c.gridheight = 2;
        c.weighty = 1.0;
        makebutton("Button8", gridbag, c);

        c.weighty = 0.0;                   //reset to the default
        c.gridwidth = GridBagConstraints.REMAINDER; //end of row
        c.gridheight = 1;                  //reset to the default
        makebutton("Button9", gridbag, c);
        makebutton("Button10", gridbag, c);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                if (inAnApplet) {
                    dispose();
                } else {
                    System.exit(0);
                }
            }
        });
    }

    public static void main(String args[]) {
        GridBagWindow window = new GridBagWindow();
        window.inAnApplet = false;

        window.setTitle("GridBagWindow Application");
        window.pack();
        window.setVisible(true);
    }
}
```

27.9 Créer ses propres layouts

A TERMINER

28. Dessiner sur une fenêtre graphique

Sommaire

28.1 Introduction 263
28.2 Dessiner des formes géométriques 265
28.3 Les méthodes repaint, paint et update 265
28.4 Redessiner tout ou pas ? 266
28.5 Redéfinir la méthode update 268
28.6 Le “double buffering” 268

28.1 Introduction

La classe **java.awt.Graphics** fournit un ensemble de méthodes pour dessiner des formes géométriques, du texte ainsi que des images sur une fenêtre graphique :

- des traits (**drawLine**) qui dessine un trait dans un objet de type **Graphics** avec la couleur par défaut. Cette couleur peut être modifiée en changement la couleur par défaut (*foreground*).
- des rectangles (**drawRect**, **fillRect** et **clearRect**).
- des rectangles en 2D (**draw3DRect** et **fill3DRect**)
- des rectangles aux bords arrondis (**drawRoundRect** et **fillRoundRect**)
- des ovales (**drawOval** et **fillOval**)
- des arcs (**drawArc** and **fillArc**)
- des polygones (**drawPolygon** and **fillPolygon**)

```
public abstract class Graphics {
   public Graphics()
   public void clearRect(int x, int y, int width, int height)
   public void clipRect(int x, int y, int width, int height)
   public void copyArea(int x, int y, int width, int height, int dx, int dy)
   public Graphics create()
   public Graphics create(int x, int y, int width, int height)
   public void dispose()
   public void draw3DRect(int x, int y, int width, int height, boolean raised)
   publicvoid drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
   public void drawBytes(byte[] data, int offset, int length, int x, int y)
   public void drawChars(char[] data, int offset, int length, int x, int y)
   public boolean drawImage(Image img, int x, int y, ImageObserver observer)
   public boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)
   public boolean drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer)
   public boolean drawImage(Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver observer)
   public boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer)
   public boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color bgcolor, ImageObser
   public void drawLine(int x1, int y1, int x2, int y2)
   public void drawOval(int x, int y, int width, int height)
   public void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)
   public void drawPolygon(Polygon p)
```

```
        public void drawPolyline(int[] xPoints, int[] yPoints, int nPoints)
        public void drawRect(int x, int y, int width, int height)
        public void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)
        public void drawString(String str, int x, int y)
        public void fill3DRect(int x, int y, int width, int height, boolean raised)
        public void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)
        public void fillOval(int x, int y, int width, int height)
        public void fillPolygon(int[] xPoints, int[] yPoints, int nPoints)
        public void fillPolygon(Polygon p)
        public void fillRect(int x, int y, int width, int height)
        public void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)
        public void finalize()
        public Shape getClip()
        public Rectangle getClipBounds()
        public Rectangle getClipBounds(Rectangle r)
        public Rectangle getClipRect()
        public Color getColor()
        public Font getFont()
        public FontMetrics getFontMetrics()
        public FontMetrics getFontMetrics(Font f)
        public boolean hitClip(int x, int y, int width, int height)
        public void setClip(int x, int y, int width, int height)
        public void setClip(Shape clip)
        public void setColor(Color c)
        public void setFont(Font font)
        public void setPaintMode()
        public void setXORMode(Color c1)
        public String toString()
        public void translate(int x, int y)
    }
```

Pour effetuer une opération graphique sur une composant, il nous faut disposer d'une instance de la classe `Graphics`. Cette instance est créée par la classe `Component` (qui crée un instance d'une classe dérivée de `Graphics`) et est passée en argument des méthode `paint` et `update`.

Chaque composant possède son propre système de coordonnées : le coin gauche supérieur est en (0,0) et le coin droit inférieur est en (`width`-1, `height`-1). L'échelle de mesure est en pixels.

Voici une exemple simple qui illustre l'utilisation des objets de type Graphics et de la méthode `drawLine`. Cette applet est une feuille de dessin simplifiée pour faire des dessins à main levée avec la souris.

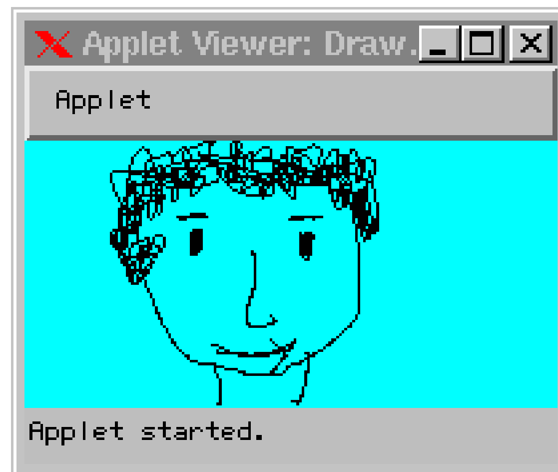

FIG. 28.1: Feuille de dessin

```
public class Draw extends java.applet.Applet {
  int x=0, y=0;
  public Draw() {
    super();
    setBackground(java.awt.Color.cyan);
    addMouseListener(new java.awt.event.MouseAdapter() {
      public void mousePressed(java.awt.event.MouseEvent e) {
        x = e.getX(); y = e.getY();
```

```
      }
    });
    addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
      public void mouseDragged(java.awt.event.MouseEvent e) {
        java.awt.Graphics g = getGraphics();
        g.drawLine(x, y, e.getX(), e.getY());
        x = e.getX(); y = e.getY();
      }
    });
  }
}
```

28.2 Dessiner des formes géométriques

Des lignes La méthode `drawLine` dessiner une ligne d'un pixel de large entre les points (x0, y0) et (x1, y1).

```
void drawLine(int x0, int y0, int x1, int y1)
```

Cette ligne n'est dessiné que dans les limites du composant sur lequel cette méthode s'applique. La couleur du dessin est désignée par le terme *foregound color* et peut être modifiée par la méthode `setForeground`.

Des rectangles

```
void drawRect(int x, int y, int w, int h)
void fillRect(int x, int y, int w, int h)
void drawRoundRect(int x, int y, int w, int h, int arcWidth, int arcHeight)
void fillRoundRect(int x, int y, int w, int h, int arcWidth, int arcHeight)
void draw3DRect(int x, int y, int w, int h, boolean raised)
void fill3DRect(int x, int y, int w, int h, boolean raised)
```

Des ovals et des arcs

```
void drawOval(int x, int y, int w, int h)
void fillOval(int x, int y, int w, int h)
void drawArc(int x, int y, int w, int h, int startAngle, int arcAngle)
void fillArc(int x, int y, int w, int h, int startAngle, int arcAngle)
```

Des polygônes

```
void drawPolygon(int xPoints[], int yPoints[], int nPoints)
void drawPolygon(Polygon p)
void fillPolygon(int xPoints[], int yPoints[], int nPoints)
void fillPolygon(Polygon p)
```

28.3 Les méthodes repaint, paint et update

Chaque fois qu'une application graphique a besoin de se dessiner ou se redessiner, le composant principal commence par se dessiner. Puis, il dessine un à un tous les composants qu'il contient et ce de manière récursive et sans aucune interruption.

Outre les moments "naturels" (démarrage de l'application et composants redevenus visibles), on peut demander à un composant de se redessiner par un appel explicite à l'une des méthodes `repaint`.

```
public void repaint()
public void repaint(long time)
public void repaint(int x, int y, int width, int height)
public void repaint(long time, int x, int y, int width, int height)
```

Lorsque l'application décide de se redessiner (automatiquement ou par un appel explicite à la méthode `repaint`), les composants se redessinent en faisant appel à la méthode `update`. La classe `Component` (classe mère de toutes les objets graphiques) fournit une implantation de la méthode `update` qui se consiste à

- Effacer le fond (*background*) du composant *i.e.* remplir le fond avec la couleur du fond,
- Invoquer la méthode `paint`. L'implantation de la méthode `paint` de la classe `Component` est vide. Lorsque le fait de dessiner un composant nécessite des opérations particulières, il suffit de redéfinir la méthode `paint`.

La méthode `paint` possède un argument qui est un objet de type `java.awt.Graphics` :

```
public void paint(java.awt.Graphics g) { ... }
```

et toutes opérations de dessin que l'on veut réaliser devra se faire sur cet objet.

```
g.drawString("Bonjour !", 10, 10);
```

Dans l'exemple qui suit, l'applet décide d'afficher une chaîne de caractères ; il faut donc redéfinir la méthode `paint`.

```
public class Bonjour extends java.applet.Applet {
    public void paint(java.awt.Graphics g) {
        g.drawString("Bonjour", 50, 50) ;
    }
}
```

Reprenons l'exemple de la feuille de dessin 28.1. Si l'on exécute ce programme, on voit apparaître une fenêtre dans un fond bleu turquoise (**setBackground(java.awt.Color.cyan)**) et à l'aide la souris on arrive à dessiner des formes.

A présent, si l'on cache cette fenêtre par une autre fenêtre ou qu'on "l'iconifie" et qu'on la fait fait réapparaître, tous nos dessins ont disparus. Par contre, le fond de la fenêtre est toujours bleu turquoise.

Que s'est-il passé ? N'ayant pas implanté la méthode **paint**, *Java* ne sait pas ce qu'il faut redessiner. Tout ce qu'il est capable de faire (comportement par défaut) consiste à redonner à la fenêtre l'apparence qu'il avait avant que l'on commence à dessiner : il a bien mis un fond bleu turquoise (et non pas gris comme tous les widgets par défaut).

Pour corriger ceci, il faut absolument définir la méthode **paint** pour que, à tout moment, on puisse être capable de refaire le dessin réalisé par l'utilisateur. Une manière simple (voire simpliste) consiste à mémoriser tous les points par lesquels la souris est passée. On pourra alors refaire le dessin réalisé par l'utilisateur chaque fois que le contenu de la fenêtre doit être redessiné.

On prendra une implantation simpliste : Deux tableaux d'entiers pouvant contenir les coordonnées des points ; tableaux qu'on limitera à ne contenir qu'un millier de points.

```
public class Persistant extends java.applet.Applet {
  private final int maxpts = 1000;
  int [] x = new int[maxpts];
  int [] y = new int[maxpts];
  int sommet;
  public Persistant() {
    super();
    sommet = 0;
    setBackground(java.awt.Color.cyan);
    addMouseListener(new java.awt.event.MouseAdapter() {
      public void mousePressed(java.awt.event.MouseEvent e) {
        empiler(-1, -1);
        empiler(e.getX(), e.getY());
      }
      public void mouseReleased(java.awt.event.MouseEvent e) {
        java.awt.Graphics g = getGraphics();
        empiler(e.getX(), e.getY());
        System.out.println(x[sommet-2] + " " + y[sommet-2]  + " " + x[sommet-1]  + " " + y[sommet-1]);
        g.drawLine(x[sommet-2], y[sommet-2], x[sommet-1], y[sommet-1]);
      }
    });
    addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
      public void mouseDragged(java.awt.event.MouseEvent e) {
        java.awt.Graphics g = getGraphics();
        empiler(e.getX(), e.getY());
        g.drawLine(x[sommet-2], y[sommet-2], x[sommet-1], y[sommet-1]);
      }
    });
  }
  void empiler(int i, int j) {
    if (sommet < maxpts) { x[sommet] = i; y[sommet++] = j; }
  }
  public void paint(java.awt.Graphics g) {
    for (int i=0; i<sommet; )
      if (x[i]==-1) i+=2;
      else  { g.drawLine(x[i-1], y[i-1], x[i], y[i]); i++; }
  }
}
```

28.4 Redessiner tout ou pas ?

Nous avons dit que la méthode **update** est utilisée chaque fois que la méthode **update** ; et elles le sont soit automatiquement par *Java* soit explicitement par le programmeur. La méthode **update** efface tout le contenu du widget et le redessine entièrement (en utilisant la méthode **paint**).

Dans beaucoup de cas, l'implanatation par défaut de la méthode **update** est suffisante. Mais il existe des cas où cette manière de faire peut conduire un appleffet de clignotement désagréable. Voici un exemple qui permet de visualiser cet effet de clignotement.

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

```
public class Cligno extends Applet implements Runnable, ActionListener {
  Thread t;
  Button b;
  Panel p;
  static public int crayon;
  static public int pas;
  public void init() {
    b = new Button("Demarrer"); p = new MonPanel(); add(b);
    add(p); b.addActionListener(this);
  }
  public void stop() { t.stop(); }
  public void run() {
    while (true) {
      crayon = (crayon + 10) % p.getSize().width;
      pas = (pas + 5) % p.getSize().width;;
      try {t.sleep(500);}
      catch (InterruptedException e){}
      p.repaint();
    }
  }
  public void actionPerformed(ActionEvent e) {
    if ("Demarrer".equals(e.getActionCommand())) {
      b.setLabel("Arreter");
      if (t!=null) t.resume();
      else { t = new Thread(this); t.start(); }
    } else {
      b.setLabel("Demarrer");
      if (t != null) t.suspend();
    }
  }
}

class MonPanel extends Panel {
  int pave = 5;
  Font f = new Font("Courier", Font.BOLD, 24);
  static Color [] couleurs = {Color.black, Color.blue, Color.cyan,
                     Color.darkGray, Color.gray, Color.green,
                     Color.lightGray, Color.magenta, Color.orange,
                     Color.pink, Color.red, Color.white, Color.yellow};
  static Vector fond ;
  Dimension minSize = new Dimension(200, 200);
  public MonPanel() {
    fond = new Vector();
    for (int j=0; j<200; j+= pave)
      for (int i=0; i<200; i+=pave)
fond.addElement(new Color(i, j, (i+j)%256));
  }
  public boolean blanc = true;
  public Dimension getPreferredSize() { return getMinimumSize(); }
  public synchronized Dimension getMinimumSize() { return minSize; }
  public void paint(java.awt.Graphics g) {
    int c = 0;
    for (int j=0; j<200; j+= pave)
      for (int i=0; i<200; i+=pave) {
g.setColor((Color)fond.elementAt(c++));
g.fillRect(j,i, pave, pave);
      }
    c = Cligno.crayon;
    g.setColor(couleurs[c++%couleurs.length]);
    g.setFont(f);
    g.drawString("Coucou !", Cligno.pas, Cligno.pas);
  }
}
```

Dans cet exemple, l'effet de clignotement provient du fait que on efface complètement le fond, qu'on le remplit avec la couleur de fond (*Background*) fenêtre pour redessiner.

Or, le fond d'écran ne change pas ; seul les emplacements des chaînes de caractères diffèrent à chaque rafraîchissement. Il est alors indiqué de restreindre la zone à redessiner à l'aide de la méthode **repaint** :

```
public void repaint(int x, int y, int width, int height)
```

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

public class NonCligno extends Applet implements Runnable, ActionListener {
  Thread t;
  Button b;
  Panel p;
  static public int blanc;
  public void init() {
    b = new Button("Demarrer"); add(b); b.addActionListener(this);
    p = new MonPanel(); add(p);
  }
  public void stop() { t.stop(); }
  public void run() {
    while (true) {
      int x = blanc;
      blanc = (blanc + 10) % p.getSize().width;
      try {t.sleep(300);}
      catch (InterruptedException e){}
      p.repaint(x, x, 70, 30);
    }
  }
  public void actionPerformed(ActionEvent e) {
    if ("Demarrer".equals(e.getActionCommand())) {
      b.setLabel("Arreter");
      if (t!=null) t.resume();
      else { t = new Thread(this); t.start(); }
    } else {
      b.setLabel("Demarrer");
      if (t != null) t.suspend();
    }
  }
}

class MonPanel extends Panel {
  Dimension minSize = new Dimension(200, 200);;
  public boolean blanc = true;
  public Dimension getPreferredSize() { return getMinimumSize(); }
  public synchronized Dimension getMinimumSize() { return minSize; }
  public void paint(java.awt.Graphics g) {
    for (int j=0; j<200; j+= 20)
      for (int i=0; i<200; i+=20) {
g.setColor(new Color(i, j, 25)); g.fillRect(j,i, 20, 20);
      }
    g.drawString("Coucou !", NonCligno.blanc, NonCligno.blanc);
  }
}
```

Cette techniwue qui consiste à restreindre à la zone de rafraîssissement est souvent appelé *cliping*.

28.5 Redéfinir la méthode update

L'exemple précédant est parfaitement adapté pour l'utilisation du *cliping*. Est-on réduit, dans les cas ou le *clipping* ne se prête pas, à supporter le clignotement ?

Pas forcément ! Comme on l'a déjà dit, le clignotement est dû à l'implantation par défaut de la méthode `update`. Une autre manière d'éviter le clignotment, consiste à redéfinir la méthode `update` pour l'empêcher de réinitialiser la zone à afficher avant l'appel à la méthode `paint`.

```
public void update(Graphics g) {
  paint(g);
}
```

28.6 Le “double buffering”

On diminue encore le clignotement en utilisant la technique du `double buffering`. Il s'agit de conserver une copie de l'image visible en mémoire.

A TERMIER

```
Image offScreenImage;
...
public void update(java.awt.Graphics g) {
  if (offScreenImage == null)
    offScreenImage = createImage(getSize().width, getSize().height);
  Graphics offGr = offScreenImage.getGraphics();
  paint(offGr);
  g.drawImage(offScreenImage, 0, 0, this);
}
```

29. Couleurs et Fontes

Sommaire

29.1 Dessiner du texte . . . 271
29.1.1 La classe Graphics . . . *271*
29.1.2 La classe java.awt.Font . . . *272*
29.1.3 La classe java.awt.FontMetrics . . . *272*
29.2 Gestion des couleurs . . . 273

29.1 Dessiner du texte

Dans cette section, nous allons voir comment écrire (ou plus exactement dessiner du texte) sur une fenêtre graphique en choisissant les fontes, les couleurs etc. Comme nous l'avons déjà vu, la classe **Graphics** fournit les les méthodes pour dessiner du texte avec la méthode **drawString**. Les classes **java.awt.Font** et **java.awt.FontMetrics** met à la disposition du programmeur un ensemble d'outils pour la gestion des caractères. Rappelons que la classe **Graphics** maintient ce que nous avons appelé contexte graphique i.e. les informations sur les opérations graphiques : couleurs, fontes, emplacement et dimension du rectangle de *clipping*. Le contexte graphique définit également la destination des opérations graphiques.

29.1.1 La classe Graphics

La classe **Graphics** définit trois méthodes pour dessiner du texte dans une fenêtre graphique.

```
void drawString(String str, int x, int y)
void drawChars(char [] data, int offset, int length, int x, int y)
void drawBytes(byte [] data, int offset, int length, int x, int y)
```

Dans la méthode **drawString(String str, int x, int y)**, **str** est la chaîne de caractères que l'on veut dessiner et (**x**, **y**) les coordonnées de la localisation du début du texte.

```
public void paint(Graphics g) { g.drawString("coucou !", 25, 25); }
```

Dans la méthode **void drawChars(char [] data, int offset, int length, int x, int y)**, **data** est un tableau de caractères, **offset** d'indice du début du texte à dessiner, **length** le nombre de carcatère à dessiner et (**x**, **y**) les coordonnées de la localisation du début du texte.

```
public void paint(Graphics g) {
  char [] texte = {'c', 'o', 'u', 'c', 'o', 'u', ' ', '!'};
  g.drawString(texte, 0, 3, 25, 25);
  g.drawString(texte, 3, 5, 25, 50);
}
```

Dans la méthode **void drawBytes(byte [] data, int offset, int length, int x, int y)**, **data** est un tableau d'octets, **offset** d'indice du début du texte à dessiner, **length** le nombre de carcatère à dessiner et (**x**, **y**) les coordonnées de la localisation du début du texte.

```
public void paint(Graphics g) {
  byte[] texte = {'c', 'o', 'u', 'c', 'o', 'u', ' ', '!'};
  g.drawByte(texte, 0, 3, 25, 25);
  g.drawByte(texte, 3, 5, 25, 50);
}
```

29.1.2 La classe java.awt.Font

La classe **java.awt.Font** permet de définir et manipuler les fontes : famille, style, taille, etc. Pour écrire avec une fonte particulière, il faut associer la fonte choisie avec l'objet **Graphics** :

```
Font f = new Font("Dialog", Font.PLAIN, 12);
g.setFont(f);
```

Les différentes familles de fontes sont **Dialog**, **Helvetica**, **TimesRoman**, **Courier**, **Symbol**, etc.

A TERMINER

Le style d'une fonte est donnée par l'une des valeurs suivantes : **PLAIN**, **BOLD** et **ITALIC**.

29.1.3 La classe java.awt.FontMetrics

L'écriture d'un texte dans une fenêtre graphique n'est qu'un cas particulier de dessin. Un texte (comme une quelconque dessin) peut être positionné n'importe où dans la zone dédiée. La position du texte dans une fenêtre ne peut être le fait du hasard ; particulièrement lorsque le texte tient sur plusieurs lignes. Pour pouvoir positionner correctement du texte, il nous faut connaître les caractéristiques graphique de la fonte utilisée :

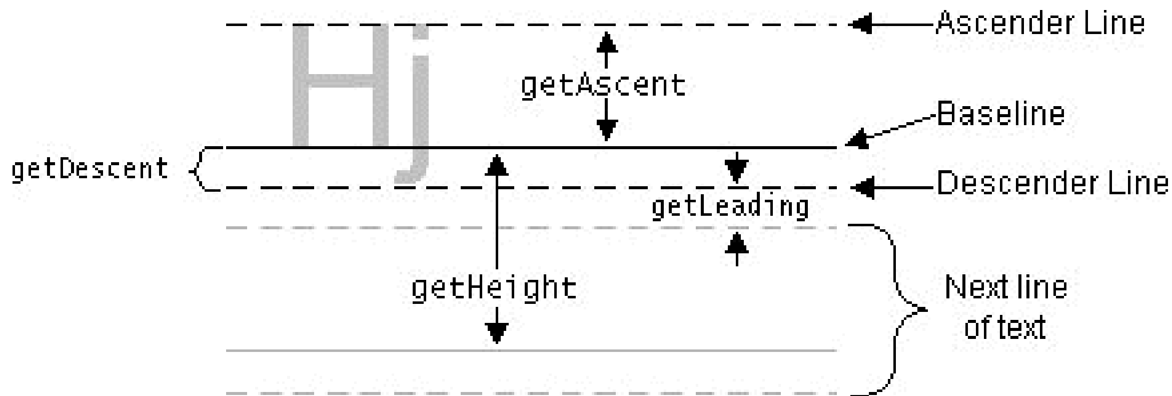

FIG. 29.1: Lignes d'écriture

Nous avons dit que les arguments (**x**, **y**) des méthodes **drawString** **drawChars** **drawBytes** définissent la position du début du texte. On peut, à présent, être plus précis : les arguments (**x**, **y**) de ces méthodes définissent la position du *baseline* du début du texte.

```
public abstract class FontMetrics extends Object implements Serializable {
   protected Font font
   protected FontMetrics(Font font)
   public Font getFont()
   public int getLeading()
   public int getAscent()
   public int getDescent()
   public int getHeight()
   public int getMaxAscent()
   public int getMaxDescent()
   public int getMaxDecent()
   public int getMaxAdvance()
   public int charWidth(int ch)
   public int charWidth(char ch)
   public int stringWidth(String str)
   public int charsWidth(char[] data, int off, int len)
   public int bytesWidth(byte[] data, int off, int len)
   public int[] getWidths()
   public String toString()
}
```

La classe **java.awt.FontMetrics** fournit les caractéristiques de la fonte utilisée. Par exemple, si l'on veut encadrer un texte, on pourra utiliser les méthodes **getAscent**, **stringWidth** et **getHeight** :

```
import java.awt.*;
public class Fontes extends java.applet.Applet {
   String str = "Coucou !!!";
   public void paint(java.awt.Graphics g) {
      g.setFont(new Font("Dialog", Font.BOLD, 32));
      FontMetrics fm = g.getFontMetrics(g.getFont());
```

```
        g.drawString(str, 20+3, 20+fm.getAscent()+3);
        g.drawRect(20, 20, fm.stringWidth(str) + 3, fm.getHeight() + 3);
    }
}
```

29.2 Gestion des couleurs

Il existe plusieurs manières de coder les couleurs :

- le système *RGB* (***R****ed*, ***G****reen*, ***B****lue*) qui représente une couleur par un mélange des couleurs rouge, vert et blue.
- le système *HSV* (***H****ue*, ***S****aturation*, ***V****alue*)
- le système *HLS* (***H****ue*, ***L****igthness*, ***S****aturation*)
- etc.

De plus, l'information sur une couleur peut être donnée en spécifiant la couleur de chaque pixel

- selon l'un des système de couleurs précédants (*direct color model*)
- par un indice dans une table de couleurs (*indexed color model*)

Par défaut, AWT utilise le modèle de couleurs (*color model*) nommée `ARGB` où ***A*** désigne la transparence (*Alpha*) et *RGB* désigne le système évoqué précédemment.

Dans ce modèle, chaque est représenté par un entier de 4 octets ; chaque octet désignant la quantité d'un des quatre composants.

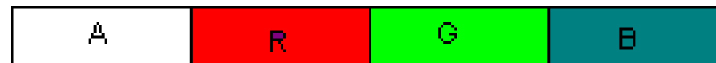

Fig. 29.2: Codage ARGB

```
public class Color extends Object implements Paint, Serializable {
    public static final Color white
    public static final Color lightGray
    public static final Color gray
    public static final Color darkGray
    public static final Color black
    public static final Color red
    public static final Color pink
    public static final Color orange
    public static final Color yellow
    public static final Color green
    public static final Color magenta
    public static final Color cyan
    public static final Color blue
    public Color(int r, int g, int b)
    public Color(int r, int g, int b, int a)
    public Color(int rgb)
    public Color(int rgba, boolean hasalpha)
    public Color(float r, float g, float b)
    public Color(float r, float g, float b, float a)
    public Color(ColorSpace cspace, float[] components, float alpha)
    public int getRed()
    public int getGreen()
    public int getBlue()
    public int getAlpha()
    public int getRGB()
    public int getRGBA()
    public Color brighter()
    public Color darker()
    public int hashCode()
    public boolean equals(Object obj)
    public String toString()
    public static Color decode(String nm) throws NumberFormatException
    public static Color getColor(String nm)
    public static Color getColor(String nm, Color v)
    public static Color getColor(String nm, int v)
    public static int HSBtoRGB(float hue, float saturation, float brightness)
    public static float[] RGBtoHSB(int r, int g, int b, float[] hsbvals)
    public static Color getHSBColor(float h, float s, float b)
```

```
   public float[] getRGBComponents(float[] compArray)
   public float[] getRGBColorComponents(float[] compArray)
   public float[] getComponents(float[] compArray)
   public float[] getColorComponents(float[] compArray)
   public float[] getComponents(ColorSpace cspace, float[] compArray)
   public float[] getColorComponents(ColorSpace cspace, float[] compArray)
   public ColorSpace getColorSpace()
   public PaintContext createContext(ColorModel cm, Rectangle r, Rectangle2D r2d, AffineTransform xform)
   public int getTransparency()
}
```

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

public class Couleurs extends Applet  implements AdjustmentListener {
   int r = 255;
   int g = 255;
   int b = 255;
   Scrollbar sbRed = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 1, 257);
   Scrollbar sbGreen = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 1, 257);
   Scrollbar sbBlue = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 1, 257);
   Scrollbar sbH = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 1, 1002);
   Scrollbar sbS = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 1, 1002);
   Scrollbar sbB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 1, 1002);
   TextField status = new TextField();
   float[] hsb;

   public void init() {
      sbRed.addAdjustmentListener(this);
      sbGreen.addAdjustmentListener(this);
      sbBlue.addAdjustmentListener(this);
      sbH.addAdjustmentListener(this);
      sbS.addAdjustmentListener(this);
      sbB.addAdjustmentListener(this);

      add(makeScrollbars("R", sbRed, "G", sbGreen, "B", sbBlue),
         BorderLayout.WEST);
      add(makeScrollbars("H", sbH, "S", sbS, "B", sbB), BorderLayout.EAST);
      status.setEditable(false);
      add(status, BorderLayout.SOUTH);
      setSize(300, 300);
      show();
      setBackground(adjustHSBScrollbars());
      adjustRGBScrollbars();
   }

   Panel makeScrollbars(String l1, Scrollbar sb1,
   String l2, Scrollbar sb2, String l3, Scrollbar sb3) {
      double[] rowWeights = {0.0, 1.0};
      GridBagLayout gbl = new GridBagLayout();
      Panel p = new Panel(gbl);

      gbl.rowWeights = rowWeights;
      p.setLayout(gbl);
      add(p, gbl, new Label(l1, Label.CENTER),
         0, 0, GridBagConstraints.NONE);
      add(p, gbl, sb1, 0, 1, GridBagConstraints.VERTICAL);
      add(p, gbl, new Label(l2, Label.CENTER),
         1, 0, GridBagConstraints.NONE);
      add(p, gbl, sb2, 1, 1, GridBagConstraints.VERTICAL);
      add(p, gbl, new Label(l3, Label.CENTER),
         2, 0, GridBagConstraints.NONE);
      add(p, gbl, sb3, 2, 1, GridBagConstraints.VERTICAL);
      return p;
   }

   void add(Panel p, GridBagLayout gbl, Component comp, int x, int y, int fill) {
      GridBagConstraints gbc = new GridBagConstraints();
      gbc.gridx = x;
      gbc.gridy = y;
      gbc.fill = fill;
      gbl.setConstraints(comp, gbc);
```

```
      p.add(comp);
   }

   public void adjustmentValueChanged(AdjustmentEvent evt) {
      Color c = getBackground();
      Object src = evt.getSource();

     // Value from scrollbar is one greater than actual RGB or HSB value

      if (src == sbRed) {
         r = 255-(sbRed.getValue()-1);
         c = adjustHSBScrollbars();
      }
      else if (src == sbGreen) {
         g = 255-(sbGreen.getValue()-1);
         c = adjustHSBScrollbars();
      }
      else if (src == sbBlue) {
         b = 255-(sbBlue.getValue()-1);
         c = adjustHSBScrollbars();
      }
      else if (src == sbH) {
         hsb[0] = (1000-(sbH.getValue()-1)) / 1000.0f;
         c = adjustRGBScrollbars();
      }
      else if (src == sbS) {
         hsb[1] = (1000-(sbS.getValue()-1)) / 1000.0f;
         c = adjustRGBScrollbars();
      }
      else if (src == sbB) {
         hsb[2] = (1000-(sbB.getValue()-1)) / 1000.0f;
         c = adjustRGBScrollbars();
      }
      setBackground(c);
      repaint();
      status.setText("RGB("+r+","+g+","+b+"  /  "+
         "#"+Integer.toString(c.getRGB()&0xffffff, 16)+")"+
         "    HSB("+hsb[0]+","+hsb[1]+","+hsb[2]+")");
   }

   Color adjustHSBScrollbars() {
      hsb = Color.RGBtoHSB(r, g, b, null);
     // need to offset values by 1 to account for 'visible' part
     // of scrollbar
      sbH.setValue(1001-(int)(hsb[0] * 1000));
      sbS.setValue(1001-(int)(hsb[1] * 1000));
      sbB.setValue(1001-(int)(hsb[2] * 1000));
      return new Color(r, g, b);
   }
   Color adjustRGBScrollbars() {
      Color c = Color.getHSBColor(hsb[0], hsb[1], hsb[2]);

     // An alternate way of converting the HSB values to RGB:
     // Color c = new Color(Color.HSBtoRGB(hsb[0], hsb[1], hsb[2]));
     // need to offset values by 1 to account for 'visible' part
     // of scrollbar
      sbRed.setValue(256-(r = c.getRed()));
      sbGreen.setValue(256-(g = c.getGreen()));
      sbBlue.setValue(256-(b = c.getBlue()));
      return c;
   }
}
```

A TERMINER

30. Images

Sommaire

30.1 Introduction 277
30.2 Manipulation des images 278
30.3 Contrôle du chargement des images 279
30.3.1 La méthode checkImage 280
30.3.2 Observateur d'images 280
30.4 Les animations graphiques 283
30.4.1 Un premier exemple d'animation 283
30.4.2 Un exemple plus agréable 284
30.5 Producteurs, consommateurs et filtres d'images 284
30.5.1 Filtres d'images 285
30.5.2 Un exemple 285
30.6 Fabriquer des images "à la main" 287

30.1 Introduction

Java fournit des facilités pour manipuler des images au format *GIF* et *JPEG*. On trouvera dans les packages **java.awt** et **java.awt.image** tout un ensemble de classes pour le faire. Une image est un objet de la classe **java.awt.Image**. On crée généralement un objet de type **Image** en invoquant la méthode **getImage** de la classe **java.applet.Applet** ou **java.awt.Toolkit**

```
uneImage = uneApplet.getImage(URL);
uneImage = uneApplet.getImage(URL, String);

uneImage = unComposant.getToolkit().getImage(URL);
uneImage = unComposant.getToolkit().getImage(String);
uneImage = unComposant.getToolkit().getImage(URL);

uneImage = Toolkit.getDefaultToolkit().getImage(URL);
uneImage = ToolKit.getDefaultToolkit().getImage(String);
uneImage = Toolkit.getDefaultToolkit().getImage(URL);
```

La méthode **getImage** ne se préoccupe pas du chargement de l'image ; elle se termine dès que le fichier image est trouvé. Le chargement de l'image ne se fait que lorsque celle-ci doit être réellement nécessaire : par exemple, lors de l'invocation de la méthode **drawImage**. Ce chargement se fait de manière asynchrone. Ceci permet de ne pas attendre la fin du chargement de l'image pour continuer la suite du programme. Cette approche est évidemment essentielle pour pouvoir télécharger des images sur le réseau sans bloquer l'application. C'est, par exemple, des browser *WEB* qui peuvent afficher les images au fur et mesure de leur chargement tout en permettant de continuer à exécuter d'autres tâches.

Dans certaines applications, il faut donc contrôler le chargement effectif de l'image ; ce que l'on fera en utilisant un **MediaTracker** ou en implantant la méthode **imageUpdate** de l'interface **ImageObserver**.

L'exemple qui suit affiche une image à l'écran. Par souci de simplicité, l'implantation est plutôt sommaire. Nous verrons un peu plus loin les diverses manières de contrôler le chargement des images.

Fig. 30.1: Afficher une image

```
public class UneImage extends java.applet.Applet {
  java.awt.Image image;
  public void init() {
    image = getImage(getCodeBase(), "tourai.gif");
    prepareImage(image, this);
  }
  public void paint(java.awt.Graphics g) { g.drawImage(image, 0, 0, this); }
}
```

L'affichage d'un objet de type `Image` se fait en invoquant la méthode **`drawImage`** sur le contexte graphique

```
drawImage(Image, int, int, ImageObserver)
```

30.2 Manipulation des images

La méthode **`drawImage`** se décline sous les formes suivantes :

```
drawImage(Image img, int x, int y, Color bgc, ImageObserver o)
drawImage(Image img, int x, int y, ImageObserver o)
drawImage(Image img, int x, int y, int l, int h, Color bgc, ImageObserver o)
drawImage(Image img, int x, int y, int l, int h, ImageObserver o)
drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color bgc, ImageObserver o)
drawImage(Image img, int, int, int, int, int, int, int, int, ImageObserver o)
```

où

- **`img`** est l'image à afficher
- (**`x`**, **`y`**) la position du coin supérieur gauche dans le composant
- **`l`** et **`h`** sont la largeur et la hauteur de la portion de image
- **`bgc`** est la couleur du fond de l'image pour les zones transparentes
- **`o`** est l'objet qui reçoit la notification du chargement de l'image
- (dx1, dy1) et (dx2, dy2) sont les coins (supérieur gauche et inférieur droit) de l'image destination
- (sx1, sy1) et (sx2, sy2) sont les coins (supérieur gauche et inférieur droit) de l'image source

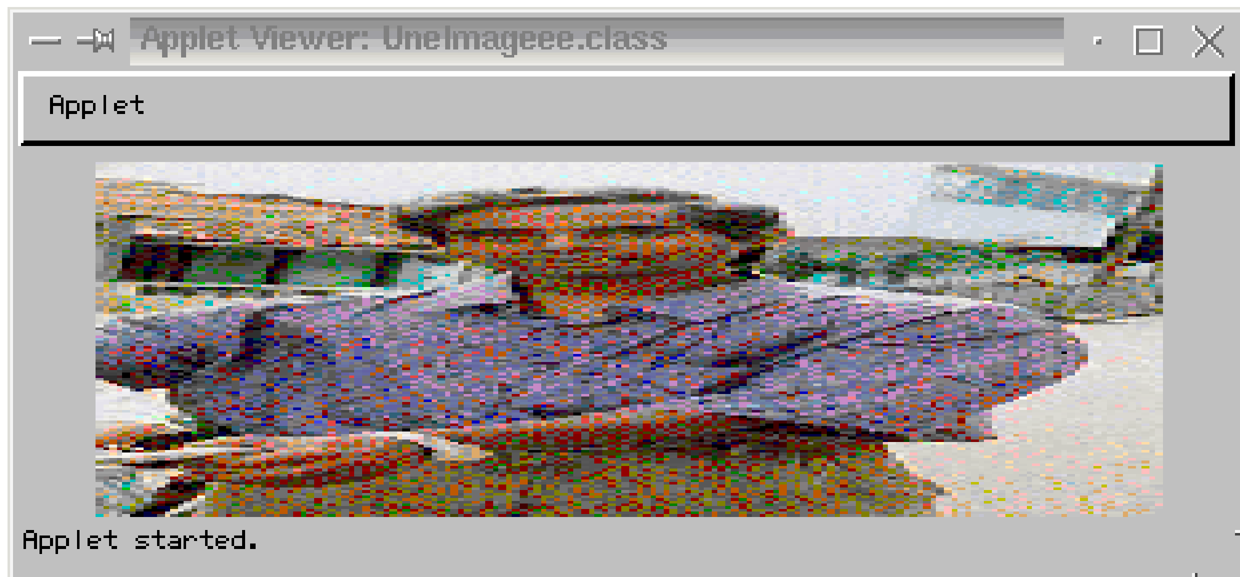

FIG. 30.2: Déformer une image

```
public class UneImageee extends java.applet.Applet {
  java.awt.Image image;
  public void init() {
    image = getImage(getCodeBase(), "photo.jpg");
  }
  public void paint (java.awt.Graphics g) {
    g.drawImage(image, 0, 0, 400, 150, this);
    g.drawImage(image, 0, 0, 80, 80, this);
  }
}
```

Comme nous l'avons dit, le chargement d'une image ne se fait que lors qu'on en a réellement besoin :

- Affichage de l'image (`drawImage`)
- Détermination des propriétés de l'image `getWidth(ImageObserver o)` et `getHeight(ImageObserver o)`). Si ces dimensions ne sont pas connus, la valeur -1 est retounée et au fur et à mesure du chargement de l'image, l'observateur d'image reçoit une notification.
- Demande explicite de chargement (`prepareImage(ImageObserver o)`)

```
public void init(){
   im = getImage(getDocumentBase(), "tourai.gif");
   prepareImage(im, this);
}
```

  Au fur et à mesure du chargement, l'observateur d'image reçoit notification.

30.3 Contrôle du chargement des images

Les objets qui manipulent les images se rangent dans trois catégories :

- Les *producteurs d'images* qui implantent l'interface `ImageProducer`. Ceux-ci créent des pixels et le distribuent à des consommateurs d'images.
- Les *consommateurs d'images* qui implantent l'interface `ImageConsumer`. Ils récupèrent des pixels et les utilisent pour les afficher ou les analyser.
- Les *observateurs d'images* qui implantent l'interface `ImageObserver`. Les observateurs d'images prennent connaissance de l'état de chargement du ou des images qu'ils observent. Le dernier argument de la méthode `drawImage` est un observateur d'image. Dans les exemples que nous venons de voir, l'observateur d'image que nous avons fournit est un objet de type `Applet`. En effet, la classe `Component` implante l'interface `ImageObserver` et fournit une implantation de la méthode `ImageUpdate`.

30.3.1 La méthode checkImage

Lorsqu'on utilise des images (particulièrement avec les applets pour lesquelles leur chargement est généralement long), il convient de vérifier si l'image est effectivement disponible pour affichage ou pas.

```
public void paint(Graphics g) {
   int n = checkImage(im, this) & ImageObserver.ALLBITS;
   if (n == ImageObserver.ALLBITS) g.drawImage(im, 50, 50, this);
   else g.drawString("Chargement en cours ...")
}
```

Nous verrons dans la section suivante tous les champs de la classe `ImageObserver`.

30.3.2 Observateur d'images

L'interface `ImageObserver` définit une seule méthode `imageUpdate` qui est invoquée chaque fois qu'une portion d'image est chargée. En implantant cette méthode, un composant peut définir ce qu'il convient de faire à chaque acquisition d'une partie de l'image (l'afficher ou le transformer etc.)

```
public interface ImageObserver {
   public static final int WIDTH
   public static final int HEIGHT
   public static final int PROPERTIES
   public static final int SOMEBITS
   public static final int FRAMEBITS
   public static final int ALLBITS
   public static final int ERROR
   public static final int ABORT
   public abstract boolean imageUpdate(Image img, int infoflags, int x, int y, int width, int height)
}
```

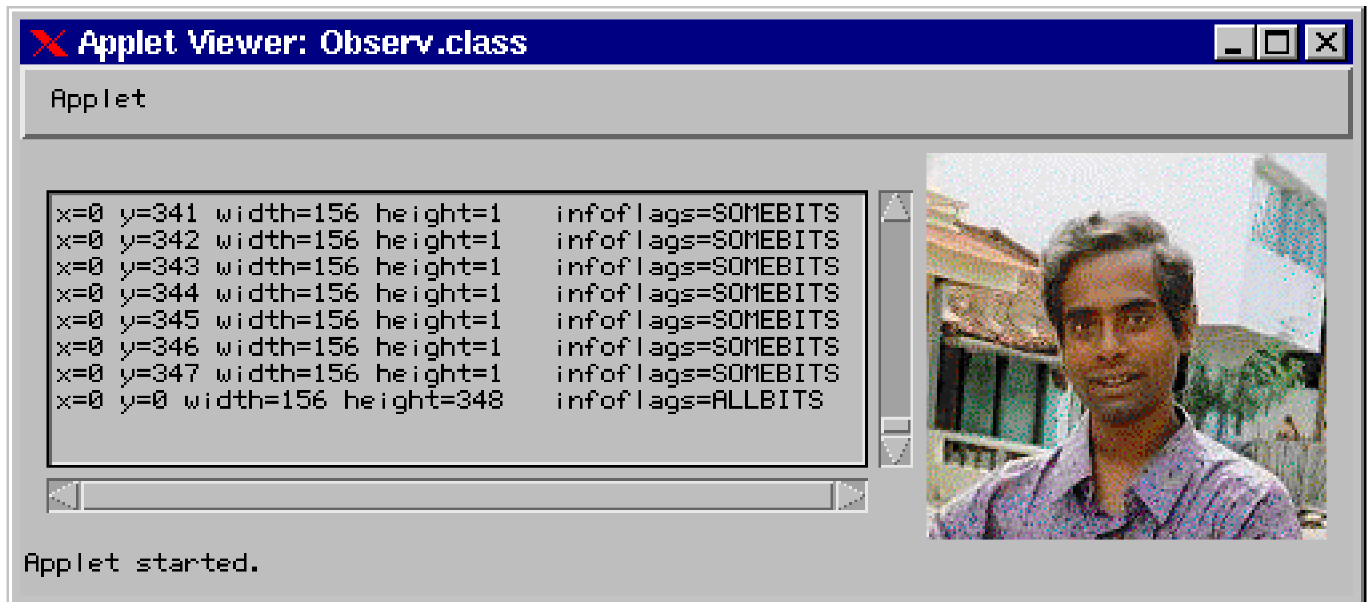

FIG. 30.3: ImageObserver

```
public class Observ extends java.applet.Applet {
  java.awt.TextArea textArea;
  java.awt.Image image;
  Trombine panel;
  String rc = System.getProperty("line.separator");
  public void init() {
    textArea = new java.awt.TextArea(8, 50);
    textArea.setEditable(false);
    add(textArea);
    image = getImage(getCodeBase(), "photo.jpg");
```

```
    panel = new Trombine(image, this);
    add(panel);
    prepareImage(image, panel);
  }
  public void afficher(String s) { textArea.append(s + rc); textArea.setCaretPosition(2000); }
}


class Trombine extends java.awt.Panel {
  java.awt.Dimension d = new java.awt.Dimension(150, 150);
  java.awt.Image image;
  Observ app;
  boolean tout = false;
  public Trombine(java.awt.Image image, Observ app) { this.image = image; this.app = app; }
  public void paint(java.awt.Graphics g) { if (tout) g.drawImage(image, 0, 0, this); }
  public java.awt.Dimension getMinimumSize() { return d; }
  public java.awt.Dimension getPreferredSize() { return getMinimumSize(); }
  public boolean imageUpdate(java.awt.Image img, int infoflags, int x, int y, int width, int height) {
    String str = "x="+x+" ";
    str += "y="+y+" ";
    str += "width="+width+" ";
    str += "height="+height+" ";
    str += "  infoflags=";
    if ((infoflags & ABORT) != 0) str += "ABORT ";
    if ((infoflags & ALLBITS) != 0)
      { str += "ALLBITS "; tout = true; repaint(); }
    if ((infoflags & ERROR) != 0) str += "ERROR ";
    if ((infoflags & FRAMEBITS) != 0) str += "FRAMEBITS ";
    if ((infoflags & HEIGHT) != 0) str += "HEIGHT ";
    if ((infoflags & PROPERTIES) != 0) str += "PROPERTIES ";
    if ((infoflags & SOMEBITS) != 0) str += "SOMEBITS ";
    if ((infoflags & WIDTH)!= 0) str += "WIDTH ";
    app.afficher(str);
    return true;
  }
}
```

On a rarement besoin d'un contrôle si fin du chargement des images ; en général, on veut juste savoir si notre image est disponible ou pas. Dans ces cas, on pourra utiliser la classe **MediaTracker** qui fournit les méthodes pour charger des images (`chechID`, `checkAll`) et attendre que les images soit chargées (`waitForID`, `waitAll`).

```
public class MediaTracker  implements Serializable {
   public static final int LOADING
   public static final int ABORTED
   public static final int ERRORED
   public static final int COMPLETE
   public MediaTracker(Component comp)
   public void addImage(Image image, int id)
   public void addImage(Image image, int id, int w, int h)
   public boolean checkAll()
   public boolean checkAll(boolean load)
   public boolean isErrorAny()
   public Object[] getErrorsAny()
   public void waitForAll() throws InterruptedException
   public boolean waitForAll(long ms) throws InterruptedException
   public int statusAll(boolean load)
   public boolean checkID(int id)
   public boolean checkID(int id, boolean load)
   public boolean isErrorID(int id)
   public Object[] getErrorsID(int id)
   public void waitForID(int id) throws InterruptedException
   public boolean waitForID(int id, long ms) throws InterruptedException
   public int statusID(int id, boolean load)
   public void removeImage(Image image)
   public void removeImage(Image image, int id)
   public void removeImage(Image image, int id, int width, int height)
 }
```

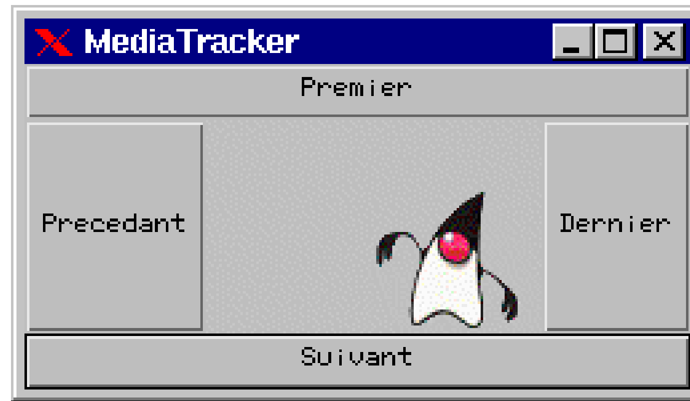

FIG. 30.4: MediaTracker

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

public class MTracker extends Applet implements ActionListener {
   Button b;
   Fenetre fenetre;
   Image[] images;
   MediaTracker tracker;
   public void init() {
      b = new Button("Cliquez ici pour ouvrir/fermer une fenetre");
      add(b);
      b.addActionListener(this);
      images = new Image[16];
      tracker = new MediaTracker(this);
      for (int i = 1; i <= 16; i++) {
         images[i-1] = getImage(getCodeBase(), "tumble/t"+i+".gif");
         tracker.addImage(images[i-1], 0);
      }
      fenetre = new Fenetre("MediaTracker", images, tracker);
   }
   public void actionPerformed(ActionEvent e) {
      if (! fenetre.isShowing()) fenetre.setVisible(true);
      else fenetre.setVisible(false);
   }
   public static void main(String[] args) {
      Frame f = new Frame("Position et Dimensions");
      MTracker p =  new MTracker();
      f.addWindowListener(
                           new WindowAdapter() {
                              public void windowClosing(WindowEvent e) {
                                 System.exit(0); }
                           });
      p.init();
      f.add(p);
      f.pack();
      f.show();
   }
}

class Fenetre extends Frame  implements ActionListener {
   MCanvas f;
   Button b1, b2, b3, b4;
   public  Fenetre(String s, Image [] images, MediaTracker tracker) {
      super(s);
      setLayout(new BorderLayout());
      add("Center", f = new MCanvas(images, tracker));
      add("North", b1 = new Button("Premier"));
      add("East", b2 = new Button("Dernier"));
      add("South", b3 = new Button("Suivant"));
      add("West", b4 = new Button("Precedant"));
```

```
      b1.addActionListener(this);
      b2.addActionListener(this);
      b3.addActionListener(this);
      b4.addActionListener(this);
      setSize(250, 100);
   }
   public void actionPerformed(ActionEvent e) {
      String s = e.getActionCommand();
      if ("Premier".equals(s)) f.num = 0;
      else if ("Dernier".equals(s)) f.num = 15;
      else if ("Suivant".equals(s)) f.num = (f.num+1)%16 ;
      else if ("Precedant".equals(s)) f.num = (f.num+15)%16;
      f.repaint();
   }

   class MCanvas extends Canvas {
      Image[] images;
      MediaTracker tracker;
      int num = 0;
      Dimension d = new Dimension(130, 80);
      public MCanvas(Image[] images, MediaTracker tracker) {
         setBackground(Color.white);
         this.images = images;
         this.tracker = tracker;
         try {
            tracker.waitForAll();  // Attendre le chargement
         }
            catch (InterruptedException e) {
            }
         setSize(130, 80);
      }
      public void paint(Graphics g) {
         if (! tracker.checkAll()) { g.drawString("Please wait...", 10, 10); }
         else g.drawImage(images[num], 0, 0, this);
      }
      public Dimension getMinimumSize() { return d;}
      public Dimension getPreferredSize() { return d;}
   }
}
```

30.4 Les animations graphiques

30.4.1 Un premier exemple d'animation

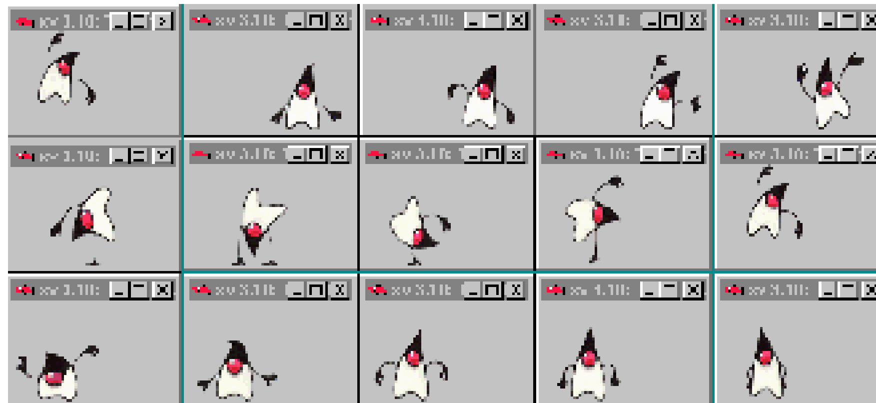

Fig. 30.5: Animation graphique

```
package td.threads;

import java.applet.*;
import java.awt.*;

public class Tumble extends Applet implements Runnable {

  Image im[] = new Image[32];
  Image  imCourant;
  Image  Ivir;
  Graphics  Gvir;
  Thread th;

  public void init() {
    Ivir = createImage(300,300);
    Gvir = Ivir.getGraphics();
    for (int i=0; i<16; i++) {
      im[i] = getImage(getCodeBase(), "tumble/T"+(i+1)+".gif");
      im[31-i] = getImage(getCodeBase(), "tumble/T"+(i+1)+".gif");
      imCourant = im[0];
    }
  }

  public void start() { if (th == null)  { th = new Thread(this);  th.start(); } }

  public void stop() { if (th != null) { th.stop(); th = null; } }
  public void run() {
    while (true) {
      for (int i=0; i<32; i++) {
try { th.sleep(100); }
catch (InterruptedException e) {}
imCourant = im[i];
repaint();
      }
      try { th.sleep(1000); }
      catch (InterruptedException e) {}
    }
  }
  public void paint(Graphics g) {
    Gvir.drawImage(imCourant, 15, 50, this);
    //g.drawImage(imCourant, 15, 50, this);
    g.drawImage(Ivir, 0, 0, this);
  }
}
```

30.4.2 Un exemple plus agréable

A TERMINER

30.5 Producteurs, consommateurs et filtres d'images

Un producteur d'images est un objet d'une classe qui implante l'interface `ImageProducer`. Il produit des images pour un ou plusieurs consommateurs. Dans beaucoup de cas, il n'est pas nécessaire de soucier des producteurs et consommateurs d'images. La production et la consommation d'images se fait de manière transparente pour le programmeur.

Décrivons un peu la face caché d'une utilisation standard d'une image. De manière simplifiée, un code comme celui suit devrait exister dans une telle application :

```
public class UneApplet extends Applet {
   private Image img = null;
   public void init() {
      img = getImage(getDocumentBase(), "image.gif");
      prepareImage(img, this);
   }
   public void paint(Graphics g) { g.drawImage(img, 50, 50, this); }
}
```

La création d'une image nécessite un producteur d'images. Ici, ce producteur est caché profondément dans l'implantation de la classe `Component`.

La préparation de l'image et son affichage requiert un consommateur d'images qui, lui aussi, est caché profondément dans l'implantation de la classe `Component`.

Enfin, au fur et à mesure du chargement de l'image, l'observateur d'images reçoit une notification de l'état du chargement. Cet observateur est également caché l'implantation de la classe `Component`.

30.5.1 Filtres d'images

Entre la production d'images et la consommation d'images, il est possible de placer un filtre de transformation.

Fig. 30.6: Filtre d'images

Un filtre de transformation est un objet la classe `ImageFilter` ou de l'une de ses classes dérivées :

– `ReplicateScaleFilter` : qui permet de redimensionner une image selon un algorithme relativement simpliste. On utilise généralement une instance de cette classe en conjonction avec une instance de la classe `FilteredImageSource` pour produire une image redimensionnée.

```
Image src = getImage(...);
ImageFilter colorfilter = new UnFiltreQuelconque();
Image img = createImage(new FilteredImageSource(src.getSource(), colorfilter);
```

– `AreaAveragingScaleFilter` : sous classe de la classe `ReplicateScaleFilter` qui également de redimessioner une image avec un algorithme différent du précédant et donne un rendu plus attenué.

```
public class AreaAveragingScaleFilter extends ReplicateScaleFilter
```

– `BufferedImageFilter` :

JDK-1.2 A TERMINER

– `CropImageFilter` : qui permet d'extraire une partie rectangulaire d'une image et le fournit une source ne contenant que la partie de l'image selectonnée. On utilise généralement une instance de cette classe en conjonction avec une instance de la classe `FilteredImageSource`.

```
ImageFilter cropfilter = new CropImageFilter(x0, y0, x1-x0, y1-y0);
ImageProducer prod = im.getSource();
prod = new FilteredImageSource(prod, cropfilter);
imm =  createImage(prod);
```

– `RGBImageFilter` : qui permet de créer un filtre de couleurs. On utilise généralement une instance de cette classe en conjonction avec une instance de la classe `FilteredImageSource` pour produire une image dont les couleurs sont altérées selon ce que la méthode `filterRGB` décide de faire. C'est une classe abstraite et il appartient au programmeur de définir une classe dérivée spécifiant le type d'altération qu'il désire en implantant la méthode `filterRGB`.

```
class EchangerRougeEtBleuFilter extends RGBImageFilter {
  public int filterRGB(int x, int y, int rgb) {
    return ((rgb & 0xff00ff00) | ((rgb & 0xff0000) >> 16) | ((rgb & 0xff) << 16));
  }
}
```

30.5.2 Un exemple

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

public class Filtre extends Applet  implements AdjustmentListener  {
   Image im ;
   int x0=0, y0=0;
   int x1=0, y1=0;
   Image imm;
   Button b1, b2;
   CopieImage cp;
   PlusOuMoinsRouge colorfilter ;
   ImageProducer prodCrop;
   public void init() {
      cp = new CopieImage(this);
      im = getImage(getCodeBase(), "pp.gif");
      setForeground(Color.white);
      addMouseMotionListener(
                              new MouseMotionAdapter() {
                                 public void mouseDragged(MouseEvent e) {
```

```
                                x1 = e.getX(); y1 = e.getY(); repaint();
                             }
                          }
                       );
      addMouseListener(
                         new MouseAdapter() {
                            public void mousePressed(MouseEvent e) {
                               x1 = x0 = e.getX(); y1 = y0 = e.getY();
                            }
                            public void mouseReleased(MouseEvent e) {
                               x1 = e.getX(); y1 = e.getY(); BoutDImage(); repaint();
                            }
                         }
                      );
   }
   public void update(Graphics g) {
      paint(g);
      cp.redessiner(imm);
   }
   public void paint(Graphics g) {
      g.drawImage(im, 0, 0, this);
      g.drawRect((x0 < x1) ? x0 : x1, (y0 < y1) ? y0 : y1, Math.abs(x1-x0), Math.abs(y1-y0));
   }
   void BoutDImage() {
      ImageFilter cropfilter = new CropImageFilter((x0 < x1) ? x0 : x1, (y0 < y1) ? y0 : y1, Math.abs(x1-x0), Math.abs(y1-y0));
      ImageProducer prod = im.getSource();
      prodCrop = new FilteredImageSource(prod, cropfilter);
      colorfilter = new PlusOuMoinsRouge();
      prod = new FilteredImageSource(prodCrop, colorfilter);
      imm =  createImage(prod);
   }

   public static int i = 100;

   public  void adjustmentValueChanged(AdjustmentEvent e) {
      i = e.getValue();
      cp.redessiner(imm);
   }
}

class CopieImage extends Frame {
   Copie c;
   ImageFilter colorfilter;
   FilteredImageSource prod;
   Filtre f;
   Scrollbar scb;
   public CopieImage(Filtre f) {
      this.f = f;
      setLayout(new BorderLayout());
      c = new Copie(f);
      add(c, "Center");
      scb = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 360);
      add(scb, "South");
      scb.addAdjustmentListener(f);


      setVisible(true);
      addWindowListener(
                          new WindowAdapter() {
                             public void windowClosing(WindowEvent e) { setVisible(false); }
                          }
                       );
   }
   void redessiner(Image imm) {
      setVisible(true);
      c.redessiner(f.imm);
   }
}

class Copie extends Canvas {
   Filtre f;
   int x0 = 50, y0 = 50;
```

```
      public Copie(Filtre f) {
         this.f = f;
         addKeyListener(
                    new KeyAdapter() {
                       public void keyTyped(KeyEvent e) {
                          switch (e.getKeyChar()) {
                             case 'a' : System.out.println("agrandir");
                                break;
                             case 'r' : System.out.println("reduire");
                                break;
                             case 'g' :
                                if (--x0 < 0) x0=0;
                                repaint();
                                break;
                             case 'd' :
                                x0++;
                                repaint();
                                break;
                             case 'h' :
                                if (--y0 < 0) y0=0;
                                repaint();
                                break;
                             case 'b' :
                                y0++;
                                repaint();
                                break;
                          }
                       }
                    }
                 );
      }
      void redessiner(Image imm) { repaint(); }
      public void paint(Graphics g) {
         if (f.imm != null) g.drawImage(f.imm, x0, y0, this);
         else g.drawString("Aucune portion d'image selectionnée", 10, 10);
      }
   }

   class PlusOuMoinsRouge extends RGBImageFilter {
      public int filterRGB(int x, int y, int rgb) {
         return (rgb & 0xff000000)  | ((rgb & 0x000000ff) << 16)
                              |  (rgb & 0x0000ff00) | ((rgb & 0x00ff0000) >> 16) ;
      }
   }
```

30.6 Fabriquer des images "à la main"

A TERMINER

31. Le son

Sommaire

Troisième partie

Java : JFC

Table des Matières

32 Java Foundation Class 295
32.1 Swing 296
32.2 Les composants Swing 296
32.3 L'architecture Model-View-Controller 301
32.4 Les composants Swing 303
32.5 Créer une application Swing 303
33 Les composants Swing 305
33.1 JButton 305
34 Gestion des évènements swing 307
35 Transfert de données 309
35.1 Introduction 309
35.2 l'API java.awt.datatransfer 310
35.3 Transfert de chaînes de caractères 311
35.4 Personaliser ses propres données de transfert 313
35.5 Formats Multiples 314
35.6 Transfert d'images 314
35.7 Drag and Drop 315
35.8 l'API Drag and Drop 315
36 Java2D 317
36.1 Introduction 317
36.2 Java 2D et le rendu 317
36.3 Textes et fontes 317
36.4 Gestion des couleurs 318
36.5 Imaging 318
36.6 Graphics Devices 318
37 Java3D 319
38 Quelques classes Swing 321
38.1 JComponent 321
38.2 JButton 323
38.3 JRadioButton 323
38.4 JCheckBox 323
38.5 JComboBox 323
38.6 JDialog 324
38.7 JProgressBar 325
38.8 JTable 325
38.9 JTabbedPane 328
38.10JTree 329

32. Java Foundation Class

Sommaire

32.1 Swing . . . 296
32.1.1 Les packages . . . *296*
32.1.2 Look and Feel . . . *296*
32.2 Les composants Swing . . . 296
32.3 L'architecture Model-View-Controller . . . 301
32.3.1 Introduction . . . *301*
32.3.2 MVC et JFC . . . *301*
32.4 Les composants Swing . . . 303
32.5 Créer une application Swing . . . 303
32.5.1 Application standard . . . *303*
32.5.2 Application MVC . . . *303*
32.5.3 Application complexe . . . *303*

L'objectif des *JFC* (*Java Foundation Class*) est de fournir, à l'image des `MFC` (*Microsoft Foundation classes*) un ensemble de classes de plus haut niveau que les classes de *AWT*.

Pour être précis, les *Java Foundation Classes* sont constitués de

- ***AWT*** :
- ***Swing*** :
- ***Java 2D*** :
- ***Accessibility*** :
- ***Drag and Drop*** :

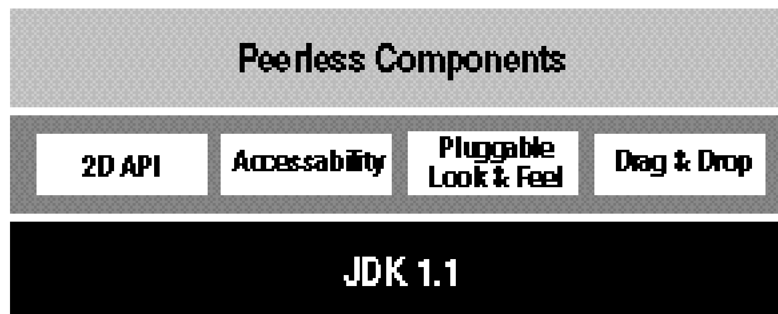

Fig. 32.1: Java Foundation Class

32.1 Swing

Les packages *Swing* sont intégrés à la version 1.2 de *JDK* et se nomment `java.awt.swing`, `java.awt.swing.event`, etc.

Cette partie du document est conforme à la version Java 1.2beta3

Comme *JFC* s'inspire des `IFC`, les widgets *JFC-Swing* ont généralement un air de famille avec les widgets *IFC*. Les widgets *AWT* et les widgets de *JFC-Swing* coexistent dans *JDK*. Le nom des composants *Swing* commence par la lettre J (`JButton`, `JLabel`, etc.) On retrouve donc dans les *Swing* et des concepts identiques que ceux rencontrés dans *AWT* et des widgets et des fonctionnalités supplémentaires.

Par contre, contrairement aux composants de *AWT*, les composants *Swing* n'implante le modèle évènementiel de `jdk 1.1` et quelques autres fonctionnalités propres à *Swing*

Rappellons que les objets de la classe `java.awt.Container` de AWT sont conçus pour pouvoir contenir des composants graphiques (`java.awt.Component`). Tous les composants *Swing* sont dérivées de la classe `JComponent`. Cette dernière est une classe dérivée de la classe `java.awt.component`. Autrement dit, n'importe quel composant *Swing* peut contenir un ou plusieurs autres composants *AWT* ou *Swing*. Cette approche permet de définir des composants beaucoup plus riches (souvenons nous qu'un bouton *AWT* ne pouvait contenir une image!!!)

32.1.1 Les packages

JFC-Swing est composé des packages suivants :

- `java.awt.swing` qui contient les composants, adaptateurs et autres classes et interfaces pour les composants.
- `java.awt.swing.event` qui les évènements spécifiques aux composants *Swing*.
- `java.awt.swing.table` qui contient les outils pour la gestion des tables.
- `java.awt.swing.text` qui contient les outils pour la gestion du texte.
- `java.awt.swing.text.html` qui contient les outils pour la gestion des liens HTML.
- `java.awt.swing.text.rtf`
- `java.awt.swing.tree` qui contient les extensions des widgets `JTree`.
- `java.awt.swing.undo`
- `java.awt.swing.border` qui contient le rendu des bords des composants.

32.1.2 Look and Feel

Sauf contre indication, le “look” des widgets sont prédéfinis et correspondnet au “look” *Java*. Il est possible de donner un look plus personnalisé en fonction de la plate forme utilisée. A cette fin, *Swing* définit la classe `UIManager` qui se charge de configuration du “look” des widgets.

32.2 Les composants Swing

Swing possède plusieurs types de widgets :

- JApplet
- JButton
- JCheckBox
- JCheckBoxMenuItem
- JComboBox
- JComponent
- JDesktopPane
- JDialog
- JEditorPane
- JFrame
- JInternalFrame
- JLabel
- JLayeredPane
- JList
- JMenu
- JMenuBar
- JMenuItem

- JOptionPane
- JPanel
- JPasswordField
- JPopupMenu
- JProgressBar
- JRadioButton
- JRadioButtonMenuItem
- JRootPane
- JScrollBar
- JScrollPane
- JSeparator
- JSlider
- JSplitPane
- JTabbedPane
- JTable
- JTextArea
- JTextField
- JTextPane
- JToggleButton
- JToolBar
- JToolTip
- JTree
- JViewport
- JWindow

FIG. 32.2: JButton

FIG. 32.3: JLabel

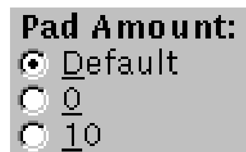

FIG. 32.4: JRadioButton

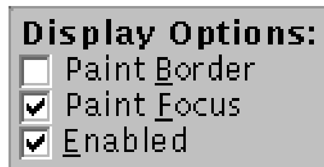

Fig. 32.5: JCheckbox

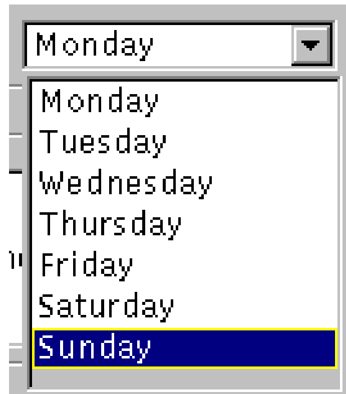

Fig. 32.6: JComboBox

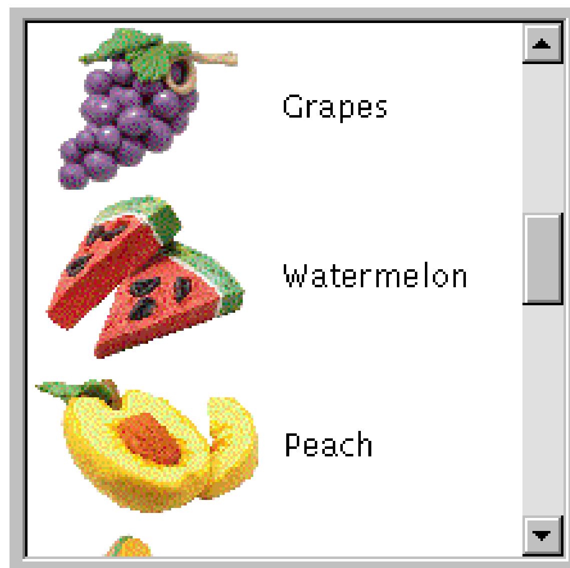

Fig. 32.7: JList

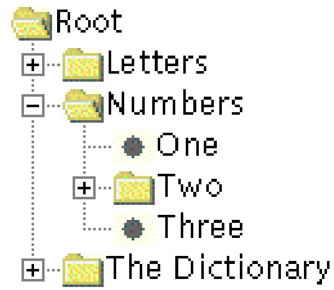

Fig. 32.8: JTree

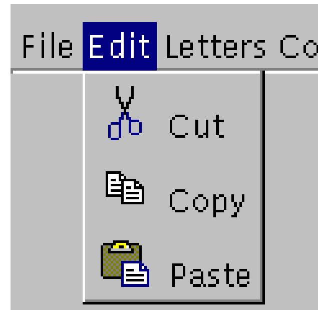

Fig. 32.9: JMenu

Fig. 32.10: JProgressBar

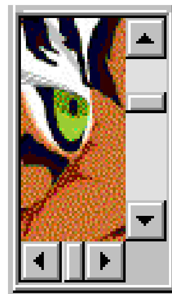

Fig. 32.11: JScrollPane

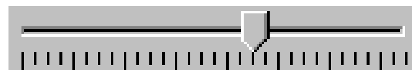

Fig. 32.12: JSlider

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

First Name	Last Name	Favorite Color	Favorite Number	Vegetar
Tim	Prinzing	Blue	22	☐
Chester	Rose	Black	0	☐
Ray	Ryan	Gray	77	☐
Georges	Saab	Red	4	☐
Kathy	Walrath	Blue	8	☐
Arnaud	Weber	Green	44	☐

Fig. 32.13: JTable

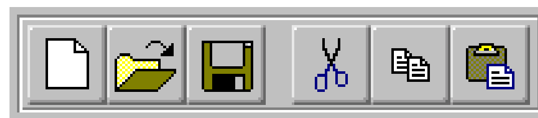

Fig. 32.14: JToolBar

Click this button to disable the middle button.

Fig. 32.15: JTooltip

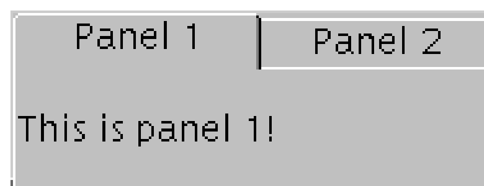

Fig. 32.16: JTabs

32.3 L'architecture Model-View-Controller

32.3.1 Introduction

Swing ne se contente pas de définir des composants et fonctionnalités supplémentaires par rapport à *AWT* ; elle introduit une approche nouvelle de la programmation graphique : l'architecture *Model-View-Controller* (MVC) qui permet une meilleure gestion de

- l'apparence des composants
- des évènements
- de la représentation des données

Une interface utilisateur obéissant à l'architecture MVC est composé de trois type d'objets qui communiquent entre-elles :

- un *modèle* qui une représentation logique de l'interface utilisateur
- une *vue* qui est la représentaion "visuelle" du modèle
- un *contrôleur* qui gère l'interaction avec l'utilisateur.

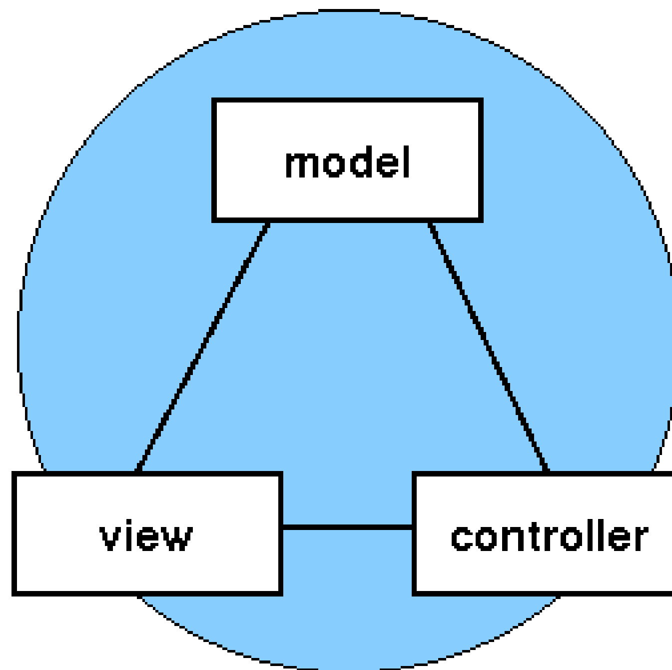

Fig. 32.17: Architecture Model/View/Controller

Cette approche permet

- De dissocier la représentation visuelle de sa représentation logique ; ce qui permet de définir plusieurs représentations visuelles pour un même mdoèle (Ex. Histogramme, Camembert)
- de changer le ou les vues sans rien modifier à la représentation logique.
- L'association modèle/représentation visuelle est effectuée dynamiquement i.e. intervient à l'exécution

Cette approche n'est pas forcément lié à la programmation des interfaces graphiques même si elle s'applique parfaitement à de telles applications.

32.3.2 MVC et JFC

La modèle *Swing* effectue une adaptation du modèle MVC dans lequel la vue et le contrôleur sont combinés en un même objet appelé *delegate*. Cette adaptation est relativement classique et permet de simplifier la communication entre la vue et le contrôleur.

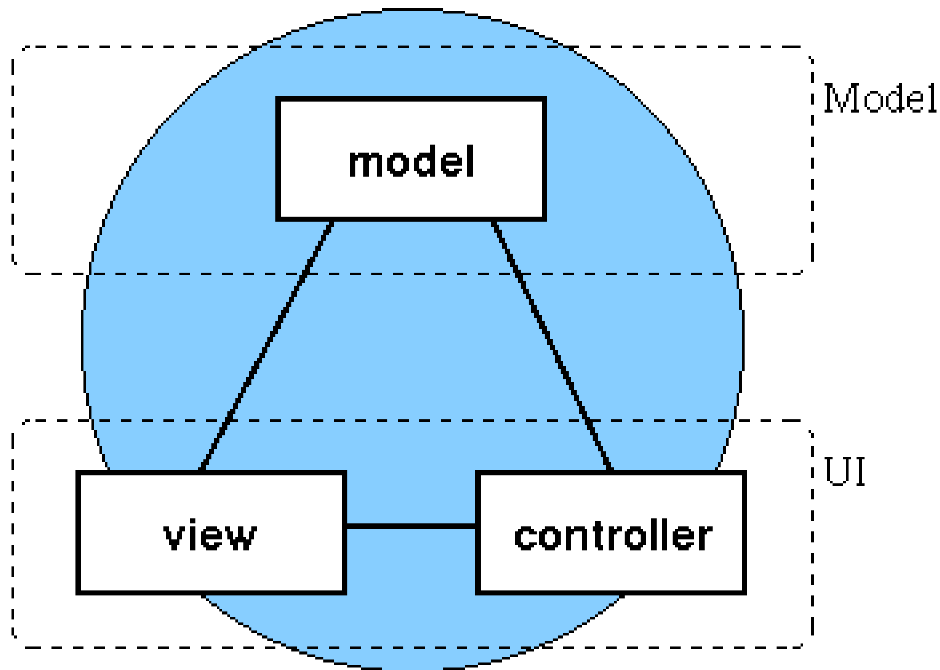

FIG. 32.18: Composant graphique JFC

Voici pour éclairer le modèle MVC, l'exemple d'un bouton.

Le modèle Le modèle d'un bouton est illustré par l'interface `ButtonModel` qui est implantée par la classe `DefaultButtonModel`. Ce modèle schématise l'état interne d'un bouton et ses comportements :

– Consultation de l'état interne

```
String   getActionCommand()
int   getMnemonic()
boolean   isArmed()
boolean   isEnabled()
boolean   isPressed()
boolean   isRollover()
boolean   isSelected()
```

– Modification de l'état interne

```
void   setActionCommand(String s)
void   setArmed(boolean b)
void   setEnabled(boolean b)
void   setGroup(ButtonGroup group)
void   setMnemonic(int key)
void   setPressed(boolean b)
void   setRollover(boolean b)
void   setSelected(boolean b)
```

– Ajout et suppression de l'écoute des évènements

```
void   addActionListener(ActionListener l)
void   addChangeListener(ChangeListener l)
void   addItemListener(ItemListener l)
void   removeActionListener(ActionListener l)
void   removeChangeListener(ChangeListener l)
void   removeItemListener(ItemListener l)
```

– Déclencher la gestion des évènements

```
void   fireActionPerformed(ActionEvent e)
void   fireItemStateChanged(ItemEvent e)
void   fireStateChanged()
```

Le contrôleur et la vue graphique Le comportement de la vue et du contrôler est défini par l'interface `ButtonUI`. Les classes qui implante cette interface sont responsables de la représentation visuelle et de la gestion des évènements :

– Affichage du bouton
– Consultation de la position du bouton
– gestion des évènements clavier et souris

Un exemple

A TERMINER

32.4 Les composants Swing

32.5 Créer une application Swing

Bien des points que nous vennons d'évoquer peuvent être ignorés lorsqu'on définit une application *Swing* de base. Il y a en fait plusieurs manières d'aborder la création d'une application *Swing* :

– Utiliser la composants *Swing* comme un composant *AWT* en lui ajoutant quelques fonctionnalités supplémentaires qui ne disponible que dans *Swing* ; par exmple, le Look and feel.
– Réaliser une application plus complexe qui implante une architecture MVC où un composant joue le rôle de contrôleur en réagissant aux actions de l'utilisation pour modifier un modèle de données ; modification qui en retour la vue des autres composants.
– Réaliser des applications plus complexes à l'aides des composants complexes de *Swing* tels que les `Jlist`, `Jtree` et autres composants texte.

32.5.1 Application standard

32.5.2 Application MVC

32.5.3 Application complexe

33. Les composants Swing

Sommaire

33.0.1 JComponent *305*
33.0.2 Un exemple *305*
33.1 JButton 305

33.0.1 JComponent

Les widgets disponibles dans *Swing* sont toutes dérivée de la classe abstraite `JComponent`. Par exemple, la classe `JButton` est une classe dérivée de la classe `JComponent`.

- Gestion statique et dynamique de l'apparence des composants
- Facilité de créer de nouveaux composants en combinant des composants standards.
-

33.0.2 Un exemple

33.1 JButton

Fig. 33.1: Des boutons Swing

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

public class SwingBouton extends Applet implements ActionListener {
   int milieu = 1;
   Image imActiver = getToolkit().getImage("activer.gif");
   Image imTourner = getToolkit().getImage("tourner.gif");
   Image imDesactiver = getToolkit().getImage("desactiver.gif");
   Image imPressed = getToolkit().getImage("pressed.gif");
   Icon iconActiver, iconTourner, iconDesactiver, iconPressed = new ImageIcon(imPressed);
```

```
   JButton b[] = {
      new JButton("Desactiver bouton " + milieu, iconDesactiver=new ImageIcon(imDesactiver)),
      new JButton("Tourner", iconTourner = new ImageIcon(imTourner)),
      new JButton("Activer bouton " + milieu, iconActiver = new ImageIcon(imActiver))
   };

   public void init() {
      for (int i = 0; i < b.length; i++) {
         b[i].addActionListener(this);
         add(b[i]);
         b[i].setPressedIcon(iconPressed);
      }
      b[milieu].setActionCommand("Tourner");
      b[(milieu-1)%3].setActionCommand("Désactiver");
      b[(milieu+1)%3].setActionCommand("Activer");
   }
   public void actionPerformed(ActionEvent e) {
      String s = e.getActionCommand();
      if ("Désactiver".equals(s)) b[milieu].setEnabled(false);
      else if ("Activer".equals(s)) b[milieu].setEnabled(true);
      else {
         milieu = (milieu+1)%3;
         b[milieu].setActionCommand("Tourner");
         b[(milieu-1+3)%3].setActionCommand("Désactiver");
         b[(milieu+1)%3].setActionCommand("Activer");
         b[milieu].setText("Tourner");
         b[(milieu-1+3)%3].setText("Desactiver bouton " + milieu);
         b[(milieu+1)%3].setText("Activer bouton " + milieu);
         b[milieu].setIcon(iconTourner);
         b[(milieu-1+3)%3].setIcon(iconDesactiver);
         b[(milieu+1)%3].setIcon(iconActiver);
         validate();
      }
   }
   public static void main(String[] args) {
      Frame f = new Frame("Des menus");
      SwingBouton p =  new SwingBouton();
      f.addWindowListener(
            new WindowAdapter() {
               public void windowClosing(WindowEvent e) { System.exit(0); }
            });
      p.init();
      f.add(p);
      f.pack();
      f.show();
   }
}
```

34. Gestion des évènements swing

Sommaire

35. Transfert de données

Sommaire

35.1 Introduction . . . 309
35.1.1 Données et type de données . . . 309
35.1.2 Objets transférables . . . 310
35.1.3 Le presse papier . . . 310
35.2 l'API java.awt.datatransfer . . . 310
35.3 Transfert de chaînes de caractères . . . 311
35.4 Personaliser ses propres données de transfert . . . 313
35.5 Formats Multiples . . . 314
35.6 Transfert d'images . . . 314
35.7 Drag and Drop . . . 315
35.7.1 La source . . . 315
35.7.2 La destination . . . 315
35.7.3 Les actions associées au Drag and Drop . . . 315
35.7.4 Transfert de données et Drag and Drop . . . 315
35.7.5 Drag and Drop multiple . . . 315
35.8 l'API Drag and Drop . . . 315

35.1 Introduction

Java fournit avec le package `Java.awt.datatransfer` des facilités pour le transfert de données au sein d'une même application ou entre différentes applications. Par exemple, une application *Java* peut utiliser le *presse papier* (*Clipboard*) en exporter ou importer des données.

L'architecture retenu pour *Java* obéit aux exigences suivantes :

- Disposer d'un mécanisme assez général pour manipuler toutes sortes de données transférable
- Permettre au programmeur de créer et transférer de nouveaux types de données
- Permettre le transfert de données entre applications *Java* et application non-*Java*.

35.1.1 Données et type de données

Tranférer des données nécessite de connaître de type de données. Les données peuvent apparaître sous diverses formes :

- texte simple
- images
- etc.

Et chacune de ces données peut apparaître sous différents formats ; *Java* utilise le terme de *flavor* pour désigner les formats.

Un exemple typique de formats différents pour un même type de données concerne les formats reconnus par un formateur de texte comme *Word* :

- Format Word 7
- Format Word 6
- Format Word 2

- Format RTF
- Format texte brut

Un format est représenté par un objet de type `java.awt.datatransfer.DataFlavor` qui se compose des informations suivantes :

- Un nom parlant pour l'humain
- Un nom logique qui identifie sans ambiguïté le format de données. Ce nom utilise la notation standard MIME (voir RFC 1521) de la forme `type/soustype`
  - text/html
  - text/plain
  - Videa/mpeg
  - text/jpeg
  - text/gif
  - etc.
- Une classe qui code l'objet à transférer. Les données transférer sont toujours des instances d'une certaine classe.

Il existe deux formats de données prédéfinis dans *JDK 1.1* :

- format représentant une classe *Java* :

```
application/x-java-serialized-object; class=[Java class]
```

- format représentant un type MIME :

```
application/<mime-subtype>; class=java.io.InputStream
```

A TERMINER

35.1.2 Objets transférables

Un objet *transférable* est une instance d'une classe qui implante l'interface `java.awt.datatransfer.Transferable`. Cette interface doit définir une liste de formats (`DataFlavor`) pour des données. Ces formats sont ordonnés du plus complexe au plus simple. De plus, un tel objet doit pouvoir fournir une donnée selon un format spécifié ou signaler que le format n'est pas connu pour cet objet.

Il existe, pour des facilité la vie du programmeur, des classes prédéfinies. Un exemple d'une telle classe est la classe `java.awt.datatransfer.StringSelection`.

35.1.3 Le presse papier

Le presse papier ((*Clipboard*) définit un mécanisme standard pour implanter les opérations *couper/copier/coller* (`cut/copy/paste`). On peut soit créer des presses papier privées ou utiliser le presse papier du système hôte. Ce dernier permet de tranférer des données entre applications Java et non-Java.

Un `Clipboard` est un objet, partagé par plusieurs clients, susceptible de contenir un objet `Transferable`. A tout moment, un seul client est propriétaire du `Clipboard` : c'est le dernier client à y avoir stocké une donnée. Lorsqu'un client perd la propriété du `Clipboard`, celui-ci reçoit une notification par l'invocation de la méthode `lostOwnership` de l'interface `java.awt.datatransfert.ClipboardOwner`.

35.2 l'API java.awt.datatransfer

Ce package `java.awt.datatransfer` comprend

- l'interface `ClipboardOwner` : cette interface est utilisée pour définir l'objet à qui appartient le *presse papier* (*clipboard*) et qui en détient le contrôle.

```
Interface ClipboardOwner {
  public void lostOwnership(Clipboard cb, Transferable contenu);
}
```

- l'interface `Transferable` : Cette interface permet de stocker le contenu d'un objet dans le presse papier. Tout objet susceptible d'être stocké dans le presse papier doit être l'instance d'une classe qui implante cette interface. La méthode `getTransferData` de cette interface est utilisée pour convertir l'objet en un format qui puisse être mis dans le presse papier.

```
Interface Transferable {
  Object   getTransferData(DataFlavor flavor);
  DataFlavor[]   getTransferDataFlavors();
  boolean   isDataFlavorSupported(DataFlavor flavor);
}
```

- la classe `Clipboard` :
  Cette classe implante les fonctionnalités liés au *effacer/copier/coller/* (*cut/copy/paste*).

```
public class Clipboard {
  protected ClipboardOwner owner
  protected Transferable contents
  public Clipboard(String name)
  public String getName()
  public void setContents(Transferable contents,
  public Transferable getContents(Object requestor)
}
```

– la classe `DataFlavor` : Cette classe est utilisée par l'interface `Transferable` pour spécifier le format des données tel qu'il est stocké dans le presse papier.

```
public class DataFlavor {
   public static final DataFlavor stringFlavor
   public static final DataFlavor plainTextFlavor
   public DataFlavor(String primaryType, String subType, MimeTypeParameterList params,
                Class representationClass, String humanPresentableName)
   public DataFlavor(Class representationClass, String humanPresentableName)
   public DataFlavor(String mimeType, String humanPresentableName)
   public DataFlavor(String mimeType) throws MimeTypeParseException, ClassNotFoundException
   public DataFlavor()
   public String getMimeType()
   public Class getRepresentationClass()
   public String getHumanPresentableName()
   public String getPrimaryType()
   public String getSubType()
   public String getParameter(String paramName)
   public void setHumanPresentableName(String humanPresentableName)
   public boolean equals(Object o)
   public boolean equals(MimeType mt)
   public boolean equals(DataFlavor dataFlavor)
   public boolean equals(String s)
   public boolean isMimeTypeEqual(String mimeType)
   public final boolean isMimeTypeEqual(DataFlavor dataFlavor)
   public boolean isMimeTypeEqual(MimeType mtype)
   public boolean isMimeTypeSerializedObject()
   public boolean isRepresentationClassInputStream()
   public boolean isRepresentationClassSerializable()
   public void writeObject(ObjectOutputStream oos) throws IOException
   public void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException
   public Object clone() throws CloneNotSupportedException
   protected String normalizeMimeTypeParameter(String parameterName, String parameterValue)
   protected String normalizeMimeType(String mimeType)
}
```

– la classe `StringSelection` : Les instances de cette classe représentent des objets `String` qui sont susceptible d'être mis dans le presse papier. Elle implante les interface `ClipboardOwner` et `Transferable`.

```
public class StringSelection {
implements Transferable, ClipboardOwner
  public StringSelection(String data)
  public DataFlavor[] getTransferDataFlavors()
  public boolean isDataFlavorSupported(DataFlavor flavor)
  public Object getTransferData(DataFlavor flavor) throws UnsupportedFlavorException, IOException
  public void lostOwnership(Clipboard clipboard, Transferable contents)
}
```

35.3 Transfert de chaînes de caractères

La classe `Clipboard` contient deux méthodes `getContents` et `setContents` qui permettent d'interagir avec le presse papier. La classe `Toolkit` possède la méthode `getSystemClipboard` qui retourne une référence vers le presse papier du système utilisé. Dans l'applet qui suit, vous devrez pouvoir importer et exporter le contenu du presse papier de votre système.

FIG. 35.1: Transfert du et vers le presse papier

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

```
import java.awt.event.*;
import java.applet.*;
public class AppletClip extends Applet {
   TextField textField = new TextField();
   TextDisplay textDisplay = new TextDisplay(textField);
   public void init() {
      setLayout(new BorderLayout());
      add(textDisplay, BorderLayout.CENTER);
      add(textField, BorderLayout.SOUTH);
   }
}

class TextDisplay extends Canvas implements TextListener {
   String text = "";
   TextField textField;
   Font f = new Font("Monospaced", Font.BOLD, 24);
   Clipboard cb = getToolkit().getSystemClipboard();

   TextDisplay(TextField textField) {
      this.textField = textField;
      setFont(new Font("Monospaced", Font.BOLD, 24));
      addMouseListener(new MouseEventHandler());
      textField.addTextListener(this);
   }

   public void paint(Graphics g) {
      FontMetrics fm = getFontMetrics(f);
      g.setFont(f);
      g.drawString(text, (getSize().width-fm.stringWidth(text))/2,
         (getSize().height-fm.getHeight())/2+fm.getAscent());
   }

   public void textValueChanged(TextEvent e)  {
      StringSelection c = new StringSelection(textField.getText());
      cb.setContents(c, c);
   }

   class MouseEventHandler extends MouseAdapter {
      public void mousePressed(MouseEvent evt) {
         Transferable t = cb.getContents(this);
         try {
            if (t != null && t.isDataFlavorSupported(DataFlavor.stringFlavor))
               text =  (String)t.getTransferData(DataFlavor.stringFlavor);
            else text = "Presse papier vide ou ne contient pas une chaîne";
            textField.setText(text);
         }
         catch (UnsupportedFlavorException e)  { e.printStackTrace(); }
         catch (IOException e)  { e.printStackTrace(); }
         repaint();
      }
   }
}
```

Récupérer des données du presse papier

- Récupérer une instance de type **Clipboard** à partir du **Toolkit**

```
Clipboard cb = getToolkit().getSystemClipboard();
```

- Récupérer le contenu du presse papier qui est un objet qui implante l'interface **Transferable**.

```
Transferable t = cb.getContents(this);
```

- Convertir le contenu du presse papier en objet reconnu par *Java*. Dans notre exemple, on désire récupérer des chaînes de caractères (**String**) et on spécifie cette conversion en passant en argument de la méthode **getTransferData**, la valeur **DataFlavor.stringFlavor**.

```
try {
  text =  (String)t.getTransferData(DataFlavor.stringFlavor);
}
catch (UnsupportedFlavorException e)  { ... }
catch (IOException e)  { ... }
```

Stocker des données dans le presse papier

- Récupérer une instance de type `Clipboard` à partir du `Toolkit`

```
Clipboard cb = getToolkit().getSystemClipboard();
```

- Créer une instance de `StringSelection` qui une chaîne de caractères transferable à partir du texte contenu dans le `textField`.

```
StringSelection c = new StringSelection(textField.getText());
```

- Transférer cette instance de `StringSelection` dans le presse papier. La méthode `setContents` admet pour argument un objet `Transferable` et un objet `ClipboardOwner`. La classe `StringSelection` implante ces deux interfaces.

```
cb.setContents(c, c);
```

35.4 Personaliser ses propres données de transfert

Fig. 35.2: Transfert d'un tableau de chaînes

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

public class TabChaines extends Applet implements ActionListener {
   static Clipboard cb = new Clipboard("Clipboard Principal");
   List lst ;
   TextArea textArea;
   String rc = System.getProperty("line.separator");

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

      Button copy = new Button("Copy");
      add(copy, "South");
      copy.addActionListener(this);

      Button coller = new Button("Coller");
      add(coller, "North");
      coller.addActionListener(this);

      Panel p = new Panel();
      add(p, "Center");

      textArea = new TextArea();
      p.add(textArea);

      lst = new List(4, true);
      lst.add("Lune");
      lst.add("Mars");
      lst.add("Mercure");
      lst.add("Jupiter");
      lst.add("Venus");
      lst.add("Saturne");
      lst.add("Soleil");
      lst.add("Terre");
      lst.add("Pluton");
      p.add(lst);
   }

   public void actionPerformed(ActionEvent e) {
      if (e.getActionCommand().equals("Copy"))
         copier();
      else  if (e.getActionCommand().equals("Coller"))
         coller();
   }
```

```
   private void coller() {
      Transferable t = cb.getContents(this);

     // Make sure the clipboard is not empty.
      if (t == null) {
         System.out.println("The clipboard is empty.");
         return;
      }
      try {
         String[] strs = (String[])t.getTransferData(
                         IntegersSelection.arrayFlavor);

         System.out.println(strs);
         for (int i=0; i<strs.length; i++)
            afficher(strs[i] + " ");
         afficher(rc + "-------------------" + rc);
      }
         catch (IOException e) {
            afficher("Données non disponible");
         }
         catch (UnsupportedFlavorException e) {
            afficher("DataFlower de mauvais type");
         }
   }

   private void copier() {
      cb.setContents(new IntegersSelection(lst.getSelectedItems()), null);
   }
   void afficher(String s) {
      textArea.append(s);
      textArea.setCaretPosition(2000);
   }
}

class IntegersSelection implements Transferable {
   static DataFlavor arrayFlavor;
   String[] strs;
   static {
      try {
         arrayFlavor = new DataFlavor(Class.forName("[Ljava.lang.String;"), "Tableau de String");
      }
         catch (ClassNotFoundException e) {
         }
   }
   DataFlavor[] flavors = {
      arrayFlavor};

   IntegersSelection(String[] strs) {
      this.strs = strs; }

   public Object getTransferData(DataFlavor flavor) throws UnsupportedFlavorException, IOException {
      if (flavor.equals(arrayFlavor))
         return strs;
      else
         throw new UnsupportedFlavorException(flavor);
   }

   public DataFlavor[] getTransferDataFlavors() {
      return flavors; }

   public boolean isDataFlavorSupported(DataFlavor flavor) {
      return flavor.equals(arrayFlavor); }
}
```

35.5 Formats Multiples

A TERMINER

35.6 Transfert d'images

A TERMINER

35.7 Drag and Drop

L'objectif du *Drag and Drop* est permettre de faire glisser des objets à l'intérieur d'une même application ou mieux encore d'une application à une autre application – application *Java* ou native.

Une opération classique de *Drag and Drop* se décompose en les trois états suivants :

- Une source (*DragSource*) associée à un composant graphique et des données transferables.
- Une cible (ou plusieurs cibles) associée à un composant graphique et capable de recevoir des données transferables.
- Une action de l'utilisateur sur un composant graphique

35.7.1 La source

La source est un composant à partir de laquelle une opération de *Drag* peut être effectué. Pour qu'un composant puisse permettre cette opération, il faut qu'il implante l'interface `java.awt.dnd.DragSource`.

Cette interface permet de préciser :

-
-
-

35.7.2 La destination

35.7.3 Les actions associées au Drag and Drop

35.7.4 Transfert de données et Drag and Drop

35.7.5 Drag and Drop multiple

35.8 l'API Drag and Drop

La gestion du *Drag and Drop* se fait l'aide du pachage `java.awt.dnd`. On trouve dans ce package les

- les interfaces
  - Autoscroll
  - DragSourceListener
  - DropTargetListener
  - FlavorMap
- Classes
  - DnDConstants
  - DragSource
  - DragSourceContext
  - DragSourceDragEvent
  - DragSourceDropEvent
  - DragSourceEvent
  - DropTarget
  - DropTargetContext
  - DropTargetContext.TransferableProxy
  - DropTargetDragEvent
  - DropTargetDropEvent
  - DropTargetEvent
- Exceptions
  - InvalidDnDOperationException

36. Java2D

Sommaire

36.1 Introduction . . . 317
36.1.1 Java 2D et formes géométriques . . . 317
36.1.2 Java 2D et gestion du texte . . . 317
36.1.3 Java 2D et images . . . 317
36.2 Java 2D et le rendu . . . 317
36.2.1 Transformations 2D . . . 317
36.2.2 Formes nouvelles . . . 317
36.2.3 Tracé . . . 317
36.2.4 Dégradés et Textures . . . 317
36.2.5 Composer des images . . . 317
36.3 Textes et fontes . . . 317
36.3.1 Gestion du texte . . . 318
36.3.2 Layout . . . 318
36.4 Gestion des couleurs . . . 318
36.5 Imaging . . . 318
36.6 Graphics Devices . . . 318

36.1 Introduction

36.1.1 Java 2D et formes géométriques

36.1.2 Java 2D et gestion du texte

36.1.3 Java 2D et images

36.2 Java 2D et le rendu

36.2.1 Transformations 2D

36.2.2 Formes nouvelles

36.2.3 Tracé

36.2.4 Dégradés et Textures

36.2.5 Composer des images

36.3 Textes et fontes

36.3.1 Gestion du texte

36.3.2 Layout

36.4 Gestion des couleurs

36.5 Imaging

36.6 Graphics Devices

37. Java3D

Sommaire

38. Quelques classes Swing

Sommaire

38.1 JComponent 321
38.2 JButton 323
38.3 JRadioButton 323
38.4 JCheckBox 323
38.5 JComboBox 323
38.6 JDialog 324
38.7 JProgressBar 325
38.8 JTable 325
38.9 JTabbedPane 328
38.10 JTree 329

38.1 JComponent

```
public abstract class JComponent  extends Container  implements Serializable {
 protected ComponentUI ui
 protected EventListenerList listenerList
 public static final int WHEN_FOCUSED
 public static final int WHEN_ANCESTOR_OF_FOCUSED_COMPONENT
 public static final int WHEN_IN_FOCUSED_WINDOW
 public static final int UNDEFINED_CONDITION
 public static final String TOOL_TIP_TEXT_KEY
 protected AccessibleContext accessibleContext

 public JComponent()
 public void updateUI()
 protected void setUI(ComponentUI newUI)
 public String getUIClassID()
 protected Graphics getComponentGraphics(Graphics g)
 protected void paintComponent(Graphics g)
 protected void paintChildren(Graphics g)
 protected void paintBorder(Graphics g)
 public void update(Graphics g)
 public void paint(Graphics g)
 public boolean isPaintingTile()
 public boolean isFocusCycleRoot()
 public boolean isManagingFocus()
 public void setNextFocusableComponent(Component aComponent)
 public void setRequestFocusEnabled(boolean aFlag)
 public void requestFocus()
 public void setPreferredSize(Dimension preferredSize)
 public void setMaximumSize(Dimension maximumSize)
 public Dimension getMaximumSize()
 public void setMinimumSize(Dimension minimumSize)
 public Dimension getMinimumSize()
```

```
public boolean contains(int x, int y)
public void setBorder(Border border)
public Border getBorder()
public Insets getInsets()
public float getAlignmentY()
public void setAlignmentY(float alignmentY)
public float getAlignmentX()
public void setAlignmentX(float alignmentX)
public Graphics getGraphics()
public void setDebugGraphicsOptions(int debugOptions)
public int getDebugGraphicsOptions()
public void registerKeyboardAction(ActionListener anAction, String aCommand,
           KeyStroke aKeyStroke, int aCondition)
public void registerKeyboardAction(ActionListener anAction,KeyStroke aKeyStroke,
                                   int aCondition)
public void unregisterKeyboardAction(KeyStroke aKeyStroke)
public KeyStroke[] getRegisteredKeyStrokes()
public int getConditionForKeyStroke(KeyStroke aKeyStroke)
public ActionListener getActionForKeyStroke(KeyStroke aKeyStroke)
public void resetKeyboardActions()
public boolean requestDefaultFocus()
public void setVisible(boolean aFlag)
public boolean isFocusTraversable()
protected void processFocusEvent(FocusEvent e)
protected void processComponentKeyEvent(KeyEvent e)
protected void processKeyEvent(KeyEvent e)
public void setToolTipText(String text)
public String getToolTipText()
public String getToolTipText(MouseEvent event)
public Point getToolTipLocation(MouseEvent event)
public JToolTip createToolTip()
public void scrollRectToVisible(Rectangle aRect)
public void setAutoscrolls(boolean autoscrolls)
public boolean getAutoscrolls()
protected void processMouseMotionEvent(MouseEvent e)
public AccessibleContext getAccessibleContext()
public final Object getClientProperty(Object key)
public final void putClientProperty(Object key, Object value)
public static boolean isLightweightComponent(Component c)
public void reshape(int x, int y, int w, int h)
public void setBounds(Rectangle r)
public Rectangle getBounds(Rectangle rv)
public Dimension getSize(Dimension rv)
public Point getLocation(Point rv)
public int getX()
public int getY()
public int getWidth()
public int getHeight()
public boolean hasFocus()
public boolean isOpaque()
public void setOpaque(boolean isOpaque)
public void computeVisibleRect(Rectangle visibleRect)
public Rectangle getVisibleRect()
protected void firePropertyChange(String propertyName, Object oldValue, Object newValue)
public void firePropertyChange(String propertyName, byte oldValue, byte newValue)
public void firePropertyChange(String propertyName, char oldValue, char newValue)
public void firePropertyChange(String propertyName, short oldValue, short newValue)
public void firePropertyChange(String propertyName, int oldValue, int newValue)
public void firePropertyChange(String propertyName, long oldValue, long newValue)
public void firePropertyChange(String propertyName, float oldValue, float newValue)
public void firePropertyChange(String propertyName, double oldValue, double newValue)
public void firePropertyChange(String propertyName, boolean oldValue, boolean newValue)
public synchronized void addPropertyChangeListener(PropertyChangeListener listener)
public synchronized void removePropertyChangeListener(PropertyChangeListener listener)
protected void fireVetoableChange(String propertyName, Object oldValue,
          Object newValue) throws PropertyVetoException
public synchronized void addVetoableChangeListener(VetoableChangeListener listener)
public synchronized void removeVetoableChangeListener(VetoableChangeListener listener)
public Container getTopLevelAncestor()
public void addAncestorListener(AncestorListener listener)
public void removeAncestorListener(AncestorListener listener)
public void addNotify()
```

```
 public void removeNotify()
 public void repaint(long tm, int x, int y, int width, int height)
 public void repaint(Rectangle r)
 public void revalidate()
 public boolean isValidateRoot()
 public boolean isOptimizedDrawingEnabled()
 public void paintImmediately(int x, int y, int w, int h)
 public void paintImmediately(Rectangle r)
 public void setDoubleBuffered(boolean aFlag)
 public boolean isDoubleBuffered()
 public JRootPane getRootPane()
}
```

38.2 JButton

```
public class JButton extends AbstractButton implements Accessible {
   public JButton()
   public JButton(Icon icon)
   public JButton(String text)
   public JButton(String text, Icon icon)
   public void updateUI()
   public String getUIClassID()
   public AccessibleContext getAccessibleContext()
}
```

38.3 JRadioButton

```
public class JRadioButton extends JToggleButton implements Accessible {
 public JRadioButton()
 public JRadioButton(Icon icon)
 public JRadioButton(Icon icon, boolean selected)
 public JRadioButton(String text)
 public JRadioButton(String text, boolean selected)
 public JRadioButton(String text, Icon icon)
 public JRadioButton(String text, Icon icon, boolean selected)
 public void updateUI()
 public String getUIClassID()
 public AccessibleContext getAccessibleContext()
}
```

38.4 JCheckBox

```
public class JCheckBox extends JToggleButton implements Accessible {
 public JCheckBox()
 public JCheckBox(Icon icon)
 public JCheckBox(Icon icon, boolean selected)
 public JCheckBox(String text)
 public JCheckBox(String text, boolean selected)
 public JCheckBox(String text, Icon icon)
 public JCheckBox(String text, Icon icon, boolean selected)
 public void updateUI()
 public String getUIClassID()
 public AccessibleContext getAccessibleContext()
}
```

38.5 JComboBox

```
public class JComboBox  extends JComponent
                      implements ItemSelectable, ListDataListener,
                                 ActionListener, Accessible {

 protected ComboBoxModel dataModel
 protected ListCellRenderer renderer
 protected ComboBoxEditor editor
 protected int maximumRowCount
```

```
 protected boolean isEditable
 protected Object selectedItemReminder
 protected JComboBox. KeySelectionManager keySelectionManager
 protected String actionCommand
 protected boolean lightWeightPopupEnabled

 public JComboBox(ComboBoxModel aModel)
 public JComboBox(Object items[])
 public JComboBox(Vector items)
 public JComboBox()
 public void setUI(ComboBoxUI ui)
 public void updateUI()
 public String getUIClassID()
 public ComboBoxUI getUI()
 public void setModel(ComboBoxModel aModel)
 public ComboBoxModel getModel()
 public void setLightWeightPopupEnabled(boolean aFlag)
 public boolean isLightWeightPopupEnabled()
 public void setEditable(boolean aFlag)
 public boolean isEditable()
 public void setMaximumRowCount(int count)
 public int getMaximumRowCount()
 public void setRenderer(ListCellRenderer aRenderer)
 public ListCellRenderer getRenderer()
 public void setEditor(ComboBoxEditor anEditor)
 public ComboBoxEditor getEditor()
 public void setSelectedItem(Object anObject)
 public Object getSelectedItem()
 public void setSelectedIndex(int anIndex)
 public int getSelectedIndex()
 public void addItem(Object anObject)
 public void insertItemAt(Object anObject, int index)
 public void removeItem(Object anObject)
 public void removeItemAt(int anIndex)
 public void removeAllItems()
 public void showPopup()
 public void hidePopup()
 public void addItemListener(ItemListener aListener)
 public void removeItemListener(ItemListener aListener)
 public void addActionListener(ActionListener l)
 public void removeActionListener(ActionListener l)
 public void setActionCommand(String aCommand)
 public String getActionCommand()
 protected void fireItemStateChanged(ItemEvent e)
 protected void fireActionEvent()
 protected void selectedItemChanged()
 public Object[] getSelectedObjects()
 public void actionPerformed(ActionEvent e)
 public void contentsChanged(ListDataEvent e)
 public boolean selectWithKeyChar(char keyChar)
 public void intervalAdded(ListDataEvent e)
 public void intervalRemoved(ListDataEvent e)
 public void setEnabled(boolean b)
 public void configureEditor(ComboBoxEditor anEditor, Object anItem)
 public void processKeyEvent(KeyEvent e)
 public boolean isFocusTraversable()
 public void setKeySelectionManager(JComboBox. KeySelectionManager aManager)
 public JComboBox. KeySelectionManager getKeySelectionManager()
 public int getItemCount()
 public Object getItemAt(int index)
 public boolean isOpaque()
 public AccessibleContext getAccessibleContext()
}
```

38.6 JDialog

```
public class JDialog extends Dialog
                     implements WindowConstants, Accessible, RootPaneContainer {
 protected JRootPane rootPane
 protected boolean rootPaneCheckingEnabled
 protected AccessibleContext accessibleContext
```

```
 public JDialog()
 public JDialog(Frame owner)
 public JDialog(Frame owner, boolean modal)
 public JDialog(Frame owner, String title)
 public JDialog(Frame owner, String title, boolean modal)

 protected void dialogInit()
 protected JRootPane createRootPane()
 protected void processWindowEvent(WindowEvent e)
 public void setDefaultCloseOperation(int operation)
 public int getDefaultCloseOperation()
 public void update(Graphics g)
 public void setJMenuBar(JMenuBar menu)
 public JMenuBar getJMenuBar()
 protected boolean isRootPaneCheckingEnabled()
 protected void setRootPaneCheckingEnabled(boolean enabled)
 protected void addImpl(Component comp, Object constraints, int index)
 public void setLayout(LayoutManager manager)
 public JRootPane getRootPane()
 protected void setRootPane(JRootPane root)
 public Container getContentPane()
 public void setContentPane(Container contentPane)
 public JLayeredPane getLayeredPane()
 public void setLayeredPane(JLayeredPane layeredPane)
 public Component getGlassPane()
 public void setGlassPane(Component glassPane)
 public void setLocationRelativeTo(Component c)
 public AccessibleContext getAccessibleContext()
}
```

38.7 JProgressBar

```
public class JProgressBar extends JComponent implements SwingConstants, Accessible {
 protected int orientation
 protected boolean paintBorder
 protected BoundedRangeModel barModel
 protected transient ChangeEvent changeEvent
 protected ChangeListener changeListener

 public JProgressBar()

 public void update(Graphics g)
 public int getOrientation()
 public void setOrientation(int newOrientation)
 public boolean isBorderPainted()
 public void setBorderPainted(boolean b)
 protected void paintBorder(Graphics g)
 public ProgressBarUI getUI()
 public void setUI(ProgressBarUI ui)
 public void updateUI()
 public String getUIClassID()
 protected ChangeListener createChangeListener()
 public void addChangeListener(ChangeListener l)
 public void removeChangeListener(ChangeListener l)
 protected void fireStateChanged()
 public BoundedRangeModel getModel()
 public void setModel(BoundedRangeModel newModel)
 public int getValue()
 public int getMinimum()
 public int getMaximum()
 public void setValue(int n)
 public void setMinimum(int n)
 public void setMaximum(int n)
 public AccessibleContext getAccessibleContext()
}
```

38.8 JTable

```
public class JTable extends JComponent
                   implements TableModelListener, Scrollable, TableColumnModelListener,
                              ListSelectionListener, CellEditorListener, Accessible {
 public static final int AUTO_RESIZE_OFF
 public static final int AUTO_RESIZE_LAST_COLUMN
 public static final int AUTO_RESIZE_ALL_COLUMNS
 protected TableModel dataModel
 protected TableColumnModel columnModel
 protected ListSelectionModel selectionModel
 protected JTableHeader tableHeader
 protected int rowHeight
 protected int rowMargin
 protected Color gridColor
 protected boolean showHorizontalLines
 protected boolean showVerticalLines
 protected int autoResizeMode
 protected boolean autoCreateColumnsFromModel
 protected Dimension preferredViewportSize
 protected boolean rowSelectionAllowed
 protected boolean cellSelectionEnabled
 protected transient Component editorComp
 protected transient TableCellEditor cellEditor
 protected transient int editingColumn
 protected transient int editingRow
 protected transient Hashtable defaultRenderersByColumnClass
 protected transient Hashtable defaultEditorsByColumnClass
 protected Color selectionForeground
 protected Color selectionBackground

 public JTable()
 public JTable(TableModel dm)
 public JTable(TableModel dm, TableColumnModel cm)
 public JTable(TableModel dm, TableColumnModel cm, ListSelectionModel sm)
 public JTable(int numColumns, int numRows)
 public JTable(Vector data, Vector columnNames)
 public JTable(Object data[][], Object columnNames[])

 public static JScrollPane createScrollPaneForTable(JTable aTable)
 public void setTableHeader(JTableHeader newHeader)
 public JTableHeader getTableHeader()
 public void setRowHeight(int newHeight)
 public int getRowHeight()
 public void setIntercellSpacing(Dimension newSpacing)
 public Dimension getIntercellSpacing()
 public void setGridColor(Color newColor)
 public Color getGridColor()
 public void setShowGrid(boolean b)
 public void setShowHorizontalLines(boolean b)
 public void setShowVerticalLines(boolean b)
 public boolean getShowHorizontalLines()
 public boolean getShowVerticalLines()
 public void setAutoResizeMode(int mode)
 public int getAutoResizeMode()
 public void setAutoCreateColumnsFromModel(boolean createColumns)
 public boolean getAutoCreateColumnsFromModel()
 public void createDefaultColumnsFromModel()
 public void setDefaultRenderer(Class columnClass, TableCellRenderer renderer)
 public TableCellRenderer getDefaultRenderer(Class columnClass)
 public void setDefaultEditor(Class columnClass, TableCellEditor editor)
 public TableCellEditor getDefaultEditor(Class columnClass)
 public void setSelectionMode(int selectionMode)
 public void setRowSelectionAllowed(boolean flag)
 public boolean getRowSelectionAllowed()
 public void setColumnSelectionAllowed(boolean flag)
 public boolean getColumnSelectionAllowed()
 public void setCellSelectionEnabled(boolean flag)
 public boolean getCellSelectionEnabled()
 public void selectAll()
 public void clearSelection()
 public void setRowSelectionInterval(int index0, int index1)
 public void setColumnSelectionInterval(int index0, int index1)
 public void addRowSelectionInterval(int index0, int index1)
```

```
public void addColumnSelectionInterval(int index0, int index1)
public void removeRowSelectionInterval(int index0, int index1)
public void removeColumnSelectionInterval(int index0, int index1)
public int getSelectedRow()
public int getSelectedColumn()
public int[] getSelectedRows()
public int[] getSelectedColumns()
public int getSelectedRowCount()
public int getSelectedColumnCount()
public boolean isRowSelected(int row)
public boolean isColumnSelected(int column)
public boolean isCellSelected(int row, int column)
public Color getSelectionForeground()
public void setSelectionForeground(Color selectionForeground)
public Color getSelectionBackground()
public void setSelectionBackground(Color selectionBackground)
public TableColumn getColumn(Object identifier)
public int convertColumnIndexToModel(int viewColumnIndex)
public int convertColumnIndexToView(int modelColumnIndex)
public int getRowCount()
public int getColumnCount()
public String getColumnName(int column)
public Class getColumnClass(int column)
public Object getValueAt(int row, int column)
public void setValueAt(Object aValue, int row, int column)
public boolean isCellEditable(int row, int column)
public void addColumn(TableColumn aColumn)
public void removeColumn(TableColumn aColumn)
public void moveColumn(int column, int targetColumn)
public int columnAtPoint(Point point)
public int rowAtPoint(Point point)
public Rectangle getCellRect(int row, int column, boolean includeSpacing)
public void sizeColumnsToFit(boolean lastColumnOnly)
public String getToolTipText(MouseEvent event)
public boolean editCellAt(int row, int column)
public boolean editCellAt(int row, int column, EventObject e)
public boolean isEditing()
public Component getEditorComponent()
public int getEditingColumn()
public int getEditingRow()
public TableUI getUI()
public void setUI(TableUI ui)
public void updateUI()
public String getUIClassID()
public void setModel(TableModel newModel)
public TableModel getModel()
public void setColumnModel(TableColumnModel newModel)
public TableColumnModel getColumnModel()
public void setSelectionModel(ListSelectionModel newModel)
public ListSelectionModel getSelectionModel()
public void tableChanged(TableModelEvent e)
public void columnAdded(TableColumnModelEvent e)
public void columnRemoved(TableColumnModelEvent e)
public void columnMoved(TableColumnModelEvent e)
public void columnMarginChanged(ChangeEvent e)
public void columnSelectionChanged(ListSelectionEvent e)
public void valueChanged(ListSelectionEvent e)
public void editingStopped(ChangeEvent e)
public void editingCanceled(ChangeEvent e)
public void setPreferredScrollableViewportSize(Dimension size)
public Dimension getPreferredScrollableViewportSize()
public int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int dir)
public int getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int dir)
public boolean getScrollableTracksViewportWidth()
public boolean getScrollableTracksViewportHeight()
protected void createDefaultRenderers()
protected void createDefaultEditors()
protected void initializeLocalVars()
protected TableModel createDefaultDataModel()
protected TableColumnModel createDefaultColumnModel()
protected ListSelectionModel createDefaultSelectionModel()
protected JTableHeader createDefaultTableHeader()
```

```
 protected void resizeAndRepaint()
 public TableCellEditor getCellEditor()
 public void setCellEditor(TableCellEditor anEditor)
 public void setEditingColumn(int aColumn)
 public void setEditingRow(int aRow)
 public boolean isOpaque()
 public Component prepareEditor(TableCellEditor editor, int row, int column)
 public void removeEditor()
 public AccessibleContext getAccessibleContext()
}
```

38.9 JTabbedPane

```
public class JTabbedPane extends JComponent
                         implements Serializable, Accessible, SwingConstants  {
 protected int tabPlacement
 protected SingleSelectionModel model
 protected ChangeListener changeListener
 protected transient ChangeEvent changeEvent

 public JTabbedPane()
 public JTabbedPane(int tabPlacement)

 public TabbedPaneUI getUI()
 public void setUI(TabbedPaneUI ui)
 public void updateUI()
 public String getUIClassID()
 protected ChangeListener createChangeListener()
 public void addChangeListener(ChangeListener l)
 public void removeChangeListener(ChangeListener l)
 protected void fireStateChanged()
 public SingleSelectionModel getModel()
 public void setModel(SingleSelectionModel model)
 public int getTabPlacement()
 public void setTabPlacement(int tabPlacement)
 public int getSelectedIndex()
 public void setSelectedIndex(int index)
 public Component getSelectedComponent()
 public void setSelectedComponent(Component c)
 public void insertTab(String title, Icon icon, Component component, String tip, int index)
 public void addTab(String title, Icon icon, Component component, String tip)
 public void addTab(String title, Icon icon, Component component)
 public void addTab(String title, Component component)
 public void removeTabAt(int index)
 public int getTabCount()
 public int getTabRunCount()
 public String getTitleAt(int index)
 public Icon getIconAt(int index)
 public Icon getDisabledIconAt(int index)
 public Color getBackgroundAt(int index)
 public Color getForegroundAt(int index)
 public boolean isEnabledAt(int index)
 public Component getComponentAt(int index)
 public Rectangle getBoundsAt(int index)
 public void setTitleAt(int index, String title)
 public void setIconAt(int index, Icon icon)
 public void setDisabledIconAt(int index, Icon disabledIcon)
 public void setBackgroundAt(int index, Color background)
 public void setForegroundAt(int index, Color foreground)
 public void setEnabledAt(int index, boolean enabled)
 public void setComponentAt(int index, Component component)
 public int indexOfTab(String title)
 public int indexOfTab(Icon icon)
 public int indexOfComponent(Component component)
 public String getToolTipText(MouseEvent event)
 public AccessibleContext getAccessibleContext()
}
```

38.10 JTree

```
public class JTree  extends JComponent  implements Scrollable, Accessible {
 protected transient TreeModel treeModel
 protected transient TreeSelectionModel selectionModel
 protected boolean rootVisible
 protected transient TreeCellRenderer cellRenderer
 protected int rowHeight
 protected boolean showsRootHandles
 protected JTree. TreeSelectionRedirector selectionRedirector
 protected transient TreeCellEditor cellEditor
 protected boolean editable
 protected boolean largeModel
 protected int visibleRowCount
 protected boolean invokesStopCellEditing
 public static final String CELL_RENDERER_PROPERTY
 public static final String TREE_MODEL_PROPERTY
 public static final String ROOT_VISIBLE_PROPERTY
 public static final String SHOWS_ROOT_HANDLES_PROPERTY
 public static final String ROW_HEIGHT_PROPERTY
 public static final String CELL_EDITOR_PROPERTY
 public static final String EDITABLE_PROPERTY
 public static final String LARGE_MODEL_PROPERTY
 public static final String SELECTION_MODEL_PROPERTY
 public static final String VISIBLE_ROW_COUNT_PROPERTY
 public static final String INVOKES_STOP_CELL_EDITING_PROPERTY

 public JTree()
 public JTree(Object value[])
 public JTree(Vector value)
 public JTree(Hashtable value)
 public JTree(TreeNode root)
 public JTree(TreeNode root, boolean asksAllowsChildren)
 public JTree(TreeModel newModel)

 protected static TreeModel getDefaultTreeModel()
 protected static TreeModel createTreeModel(Object value)
 public TreeUI getUI()
 public void setUI(TreeUI ui)
 public void updateUI()
 public String getUIClassID()
 public TreeCellRenderer getCellRenderer()
 public void setCellRenderer(TreeCellRenderer x)
 public void setEditable(boolean flag)
 public boolean isEditable()
 public void setCellEditor(TreeCellEditor cellEditor)
 public TreeCellEditor getCellEditor()
 public TreeModel getModel()
 public void setModel(TreeModel newModel)
 public boolean isRootVisible()
 public void setRootVisible(boolean rootVisible)
 public void setShowsRootHandles(boolean newValue)
 public boolean getShowsRootHandles()
 public void setRowHeight(int rowHeight)
 public int getRowHeight()
 public boolean isFixedRowHeight()
 public void setLargeModel(boolean newValue)
 public boolean isLargeModel()
 public void setInvokesStopCellEditing(boolean newValue)
 public boolean getInvokesStopCellEditing()
 public boolean isPathEditable(TreePath path)
 public String getToolTipText(MouseEvent event)
 public String convertValueToText(Object value, boolean selected, boolean expanded,
                                               boolean leaf, int row, boolean hasFocus)
 public int getRowCount()
 public void setSelectionPath(TreePath path)
 public void setSelectionPaths(TreePath paths[])
 public void setSelectionRow(int row)
 public void setSelectionRows(int rows[])
 public void addSelectionPath(TreePath path)
 public void addSelectionPaths(TreePath paths[])
```

```
 public void addSelectionRow(int row)
 public void addSelectionRows(int rows[])
 public Object getLastSelectedPathComponent()
 public TreePath getSelectionPath()
 public TreePath[] getSelectionPaths()
 public int[] getSelectionRows()
 public int getSelectionCount()
 public int getMinSelectionRow()
 public int getMaxSelectionRow()
 public int getLeadSelectionRow()
 public TreePath getLeadSelectionPath()
 public boolean isPathSelected(TreePath path)
 public boolean isRowSelected(int row)
 public boolean isExpanded(TreePath path)
 public boolean isExpanded(int row)
 public boolean isCollapsed(TreePath path)
 public boolean isCollapsed(int row)
 public void makeVisible(TreePath path)
 public boolean isVisible(TreePath path)
 public Rectangle getPathBounds(TreePath path)
 public void scrollPathToVisible(TreePath path)
 public void scrollRowToVisible(int row)
 public TreePath getPathForRow(int row)
 public int getRowForPath(TreePath path)
 public void expandPath(TreePath path)
 public void expandRow(int row)
 public void collapsePath(TreePath path)
 public void collapseRow(int row)
 public TreePath getPathForLocation(int x, int y)
 public int getRowForLocation(int x, int y)
 public TreePath getClosestPathForLocation(int x, int y)
 public int getClosestRowForLocation(int x, int y)
 public boolean isEditing()
 public boolean stopEditing()
 public void startEditingAtPath(TreePath path)
 public TreePath getEditingPath()
 public void setSelectionModel(TreeSelectionModel selectionModel)
 public TreeSelectionModel getSelectionModel()
 protected TreePath[] getPathBetweenRows(int index0, int index1)
 public void setSelectionInterval(int index0, int index1)
 public void addSelectionInterval(int index0, int index1)
 public void removeSelectionInterval(int index0, int index1)
 public void removeSelectionPath(TreePath path)
 public void removeSelectionPaths(TreePath paths[])
 public void removeSelectionRow(int row)
 public void removeSelectionRows(int rows[])
 public void clearSelection()
 public boolean isSelectionEmpty()
 public void addTreeExpansionListener(TreeExpansionListener tel)
 public void removeTreeExpansionListener(TreeExpansionListener tel)
 public void fireTreeExpanded(TreePath path)
 public void fireTreeCollapsed(TreePath path)
 public void addTreeSelectionListener(TreeSelectionListener tsl)
 public void removeTreeSelectionListener(TreeSelectionListener tsl)
 protected void fireValueChanged(TreeSelectionEvent e)
 public void treeDidChange()
 public void setVisibleRowCount(int newCount)
 public int getVisibleRowCount()
 public boolean isOpaque()
 public Dimension getPreferredScrollableViewportSize()
 public int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)
 public int getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction)
 public boolean getScrollableTracksViewportWidth()
 public boolean getScrollableTracksViewportHeight()
 public AccessibleContext getAccessibleContext()
}
```

Quatrième partie

Java : Programmation avancée

Table des Matières

39 Programmation réseau . . . 335
39.1 Introduction aux réseaux . . . 335
39.2 Adressage IP . . . 340
39.3 Les URLs . . . 340
39.4 URLEncoder . . . 342
39.5 Les sockets . . . 342
39.6 Connexion TCP et sockets . . . 344
39.7 Datgrammes UDP et sockets . . . 346
39.8 La classe DatagramPacket . . . 346
39.9 La classe DatagramSocket . . . 347
39.10 Le client UDP . . . 347
39.11 Le serveur UDP . . . 348
39.12 URLConnection . . . 349
39.13 Un petit serveur HTTP . . . 351
39.14 Un CGI en Java . . . 351
39.15 Gestion de protocoles . . . 351
39.16 ContentHandler . . . 351
39.17 Gestion de protocoles . . . 351
39.18 Sockets multipoints . . . 351
40 Appel de méthodes distantes (RMI) . . . 353
40.1 Introduction . . . 353
40.2 Un exemple . . . 354
41 Introduction aux bases de données . . . 357
41.1 Introduction . . . 357
41.2 Le modèle relationnel . . . 357
42 Introduction à JDBC . . . 359
42.1 Introduction . . . 359
42.2 Se connecter à une base de données . . . 360
42.3 Envoi de requêtes SQL . . . 361
42.4 Un premier exemple complet . . . 362
43 Code natif (JNI) . . . 365
43.1 Un exemple . . . 366
43.2 Conversion des caractères unicode . . . 368
43.3 Conversion des données . . . 368
43.4 Créer et modifier des objets . . . 371
43.5 Signature . . . 371
43.6 Accès aux champs . . . 371
43.7 Invocation de méthodes Java . . . 373

43.8 Exceptions et code natif . . . 374
43.9 Capturer une exception . . . 374
43.10 Lancer une exception . . . 374
43.11 Code natif et threads . . . 375
43.12 Code natif et C++ . . . 375
43.13 Démarrer la machine viruelle Java . . . 375
43.14 La commande javap . . . 376
43.15 Récapitulatif des fonctions JNI . . . 376
44 Corba . . . 379

39. Programmation réseau

Sommaire

39.1 Introduction aux réseaux . . . 335
39.1.1 Généralités . . . 335
39.2 Adressage IP . . . 340
39.3 Les URLs . . . 340
39.4 URLEncoder . . . 342
39.5 Les sockets . . . 342
39.6 Connexion TCP et sockets . . . 344
39.6.1 Le client TCP . . . 344
39.6.2 Le serveur TCP . . . 344
39.6.3 Connexions multiples . . . 345
39.7 Datgrammes UDP et sockets . . . 346
39.8 La classe DatagramPacket . . . 346
39.9 La classe DatagramSocket . . . 347
39.10Le client UDP . . . 347
39.11Le serveur UDP . . . 348
39.12URLConnection . . . 349
39.13Un petit serveur HTTP . . . 351
39.14Un CGI en Java . . . 351
39.15Gestion de protocoles . . . 351
39.16ContentHandler . . . 351
39.17Gestion de protocoles . . . 351
39.18Sockets multipoints . . . 351

39.1 Introduction aux réseaux

Ce chapitre passe en revue, très rapidement, les quelques notions nécessaires pour la programmation réseau.

39.1.1 Généralités

Un *réseau* informatique est constitué d'un ensemble d'ordinateurs et de périphériques interconnectés. Un *nœud* du réseau qui correspond à un ordinateur est généralement appelé *hôte*.

TCP/IP est l'architecture réseau la plus répandue. Contrairement à ce que son nom laisse croire, TCP/IP n'est pas une architecture réduite à deux protocoles : IP (Internet Protocol) et TCP (Transmission Control Protocol). L'architecture TCP/IP inclut de nombreux "services d'application", des protocoles élaborés et complexes, sur lesquels les applications distribuées peuvent s'appuyer. Sans rentrer dans les détails, le schéma suivant précise les divers couches du réseau où interviennent les protocoles IP, TCP, et UDP.

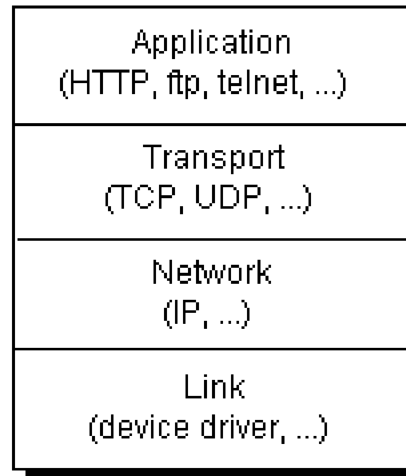

FIG. 39.1: Les couches réseau

Les couches les plus basses, Ligne, Physique, et Réseau ne sont pas incluses dans l'architecture TCP/IP. En fait, TCP/IP s'appuie sur n'importe quelles couches basses existantes :

- réseaux locaux IEEE 802.3 ou 802.5 et le protocole LLC IEEE 802.2,
- réseau local Ethernet (Ethernet Specification II de Xerox),
- réseaux longue distance propriétaires (SNA, DECNet),
- réseaux longue distance normalisés (X.25, Relais de Trames, ATM),
- liaisons spécialisées (fixes) ou commutées (RTC, RNIS) utilisant divers protocoles de niveau ligne,
- liaisons radio, satellite,
- etc.

Protocole IP

Le protocole IP ne se préoccupe que de routage et contrôle dans un environnement "inter-réseau". Le protocole IP permet l'échange de datagrammes (en mode non connecté), entre des hôtes connectés à des réseaux physiques divers. Le protocole IP (*Internet Protocol*) est un protocole ouvert et indépendant du système. Des nœud d'un réseau sont potentiellement reliés par plusieurs chemins. Chaque paquet suit un chemin particulier mais une succession de paquets provenant un même bloc de données ne suivent pas forcément le même chemin.

Le protocole IP comporte des limitations notables :

- adressage au niveau des hôtes, et non au niveau des applications ;
- livraison non garantie ;
- erreurs possibles sur les données transportées ;
- déséquencement possible.

Les applications désirant communiquer via IP disposent de deux alternatives :

- utiliser un protocole de transport fiable (TCP : Transmission Control Protocol) ;
- utiliser un protocole de transport rapide (UDP : User Datagram Protocol).

Protocole TCP

Le protocole *TCP* est un protocole plus sûr bâti au dessus du protocole IP. Le rôle de TCP est de sécuriser l'envoi et la réception des données par un système d'accusé de réception. . C'est un protocole de type "orienté connexion", c'est à dire que les échanges de données ont lieu de façon ordonnée, fiable, après négociation préliminaire des caractéristiques de transfert. En particulier, le protocole TCP garde de lui-même une copie des données émises tant que celles-ci n'ont pas été acquittées. TCP utilise également un mécanisme de checksum pour garantir l'intégrité des données, ainsi que des numéros de séquence assurant la réception ordonnée des données Lorsque des paquets sont perdus, TCP demande la réexpédition des ces derniers. Il reconstitue également les paquets suivant l'ordre de leur expédition. TCP est donc un protocole plus fiable mais bien moins économique.

Protocole UDP

Le protocole UDP (User Datagram Protocol) est un protocole de la couche de transport au dessus de IP. Le protocole UDP est essentiellement utilisable lorsque les applications désirant échanger des données privilégient la rapidité et la simplicité par rapport à la fiabilité. Le protocole UDP ne numérote pas les données envoyées, n'acquitte pas les données reçues, et propose un checksum optionnel portant sur les données Avec ce protocole, rien ne garantit la livraison effectif d'un paquet. De plus, lorsque les différents paquets d'un même bloc de données arrivent à destination rien ne garantit l'ordre d'arrivée de ces paquets.

Il existe des applications toutes les données sont vitales et aucune perte de données n'est tolérable ; dans ces cas, on choisira d'utiliser le protocole TCP. Par contre, il existe d'autres applications ou un perte de quelques données est sans conséquence : dans ces, on utilisera plutôt le protocole UDP.

Adresse internet

Chaque nœud du réseau possède une *adresse internet* composée d'une série d'octets. L'attribution d'une adresse à un nœud du réseau dépend du type de réseau. Pour ce qui nous concerne, nous pouvons ignorer la manière dont ces adresses sont attribués aux nœuds du réseau. Par exemple, l'adresse de la machine du serveur du département GBM de l'ESIL est constitué des octets `139`, `124`, `32` et `249` que l'on a l'habitude de noter sous la forme `139.124.32.249`.

Parallèlment aux adresses, les nœuds possèdent un nom plus parlant pour l'être humain. Le serveur du département GBM de l'ESIL ne nomme `gbm.esil.univ-mrs.fr`. Ce nom est totalement indépendant de l'adresse internet ce qui permet de changer l'adresse internet d'une machine sans devoir forcément changer son nom.

DNS

Le DNS (Domain Name System) permet la mise en correspondance des adresses internet et des noms de nœuds.

Les données qui circulent dans un réseau sont découpés en tronçons d'une certaine taille ; ces tronçon sont appelés *paquets*.

Les ordinateurs communiquent entre eux à travers un protocole : il s'agit d'une convention que reconnaît les machines émettrices et les machines réceptrices. Un exemple typique d'une protocole est le protocole *HTTP* qui définit les modalités de communication entre un navigateur *WEB* et un serveur de pages `HTML`. Il existe bien d'autres protocoles : `file`, `ftp`, `gopher`, `news`, `telnet`, `WAIS`, etc.

Architecture Client/Serveur

Le mode de communication qu'un hôte établit avec un autre hôte qui fournit un service quelconque est désigné par le terme *Client/Serveur*. Les notions de base de l'architecture client/serveur sont :

- Client : Processus demandant l'exécution d'une opération à un autre processus par un envoi de message et attendant la réponse à cette question par un message en retour.
- Serveur : Processus accomplissant une opération sur demande d'un client et transmettant la réponse à ce client.
- Requête : Message transmis par un client à un serveur décrivant l'opération à exécuter pour le compte du client.
- Réponse : Message transmis par un serveur à un client suite à l'exécution d'une opération contenant les paramètres de retour de l'opération

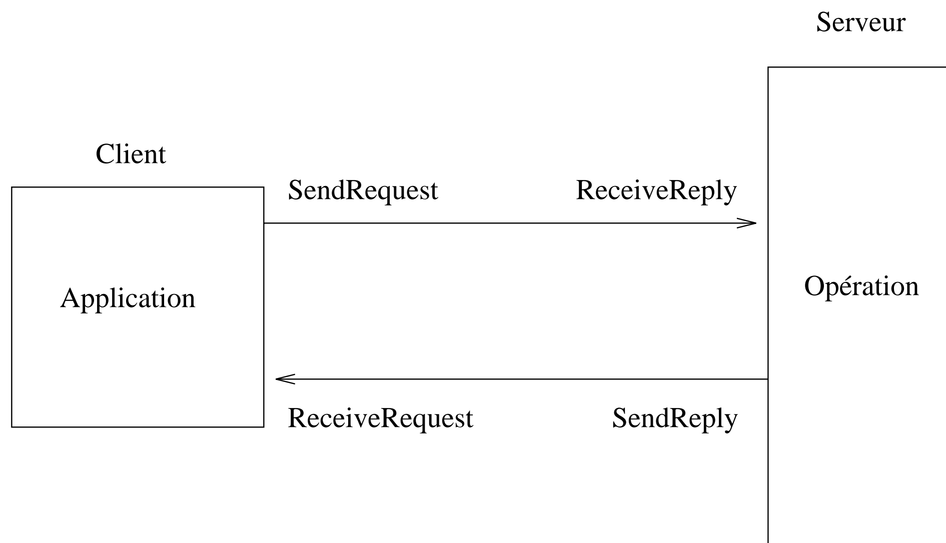

Fig. 39.2: Dialogue Client/Serveur

Port UDP et TCP

Chaque hôte est susceptible de fournir plusieurs services simultanément : serveur HTTP, FTP, MAIL, etc. Selon le type de service que l'on veut utiliser, une machine cliente s'adressera à une machine serveur sur un canal dédié ; on appelle *ports logiques* ces canaux.

Chaque machine sous IP possède quelques 65 535 ports ; par exemple, le service HTTP est lié au port 80, le service SMTP (courrier électronique) est relié au port 25, etc.

Une machine cliente qui s'adresse à une machine serveur le fait en spécifiant son adresse internet et le numéro de port qui l'intéresse.

Une différence importante au niveau de l'utilisation des ports TCP et UDP est la suivante

- tous les messages UDP destinés à un numéro de port UDP seront livrés en un même " point" ; il y a une seule file d'attente de messages pour le port.
- alors que les messages TCP sur des connexions TCP vers un même port peuvent être discriminés : il y a une file d'attente de messages par connexion

Avec TCP, il sera aisé de concevoir des applications serveur "multi-processus" : chaque processus sera en charge de gérer une relation client-serveur, via une connexion TCP ; chaque processus possède une file d'attente propre. Avec UDP, c'est impossible : la même file d'attente sera utilisée par tous les processus.

Avec UDP, les seules applications serveur qui pourront fonctionner en version "multi-processus" seront celles conçues pour accepter une requête et renvoyer immédiatement une réponse, et une seule. C'est le cas de nombreuses applications dites " transactionnelles" : les requêtes sont sérialisées par le port UDP ; chacune est traitée par un processus, la réponse est ensuite envoyée. Les requêtes consécutives ne doivent pas être corrélées.

Ports Réservés

Les applications serveurs nécessitent des ports connus de façon globale : un port de serveur est équivalent à un numéro de téléphone publié dans un annuaire.

Avec TCP et UDP, un certain nombre de ports sont réservés à des services bien connus. Par exemple, sur TCP :

- Serveur FTP, toujours en attente sur le port 21 ;
- Serveur Telnet, toujours en attente sur le port 23 ;
- Serveur SMTP, toujours en attente sur le port 25.

Sur UDP :

- Agent SNMP, sur le port 161 (voir chapitre 7),
- Logger SNMP, sur le port 162 (voir chapitre 7),
- Serveur rwhod, sur le port 513,
- etc.

Ces ports alloués statiquement, aussi appelés *Well Known Ports*, correspondant à des *Well Known Services (WKS)*, dans un espace de port réservé au serveur :

- de 1 à 1024, ce sont des services fondamentaux (gérés par des administrateurs) ;
- de 1025 à 5000, ils sont disponibles aux utilisateurs pour implanter des serveurs de nature quelconque.

Sous Unix, les ports réservés pour les "services bien connus" sont visibles dans le fichier `/etc/services`. Par exemple :

```
tcpmux          1/tcp           # TCP port multiplexer (RFC 1078)
echo            7/tcp
echo            7/udp
ftp-data        20/tcp
ftp             21/tcp
telnet          23/tcp
smtp            25/tcp          mail
domain          53/tcp          nameserver      # name-domain server
domain          53/udp          nameserver
bootp           67/udp          bootps      # bootp server
bootpc          68/udp                      # bootp client
tftp            69/udp
finger          79/tcp
iso-tsap        102/tcp
snmp            161/udp
snmp-trap       162/udp         snmptrap
xdmcp           177/udp         # X Display Mgr. Control Prot.
route           520/udp         router routed
```

Les URLs

On appelle URL (*Uniform Resource Location*) la manière de nommer une ressource sur le réseau Internet. Par exemple, `http ://gbm.esil.univ-mrs.fr/eleves/index.html` désigne la page WEB d'accueil des élèves du département GBM de l'E-SIL. Un URL se compose de trois parties : le protocole, l'hôte et le document. La syntaxe générale d'un URL est :

```
<protocole>://<nom_hote>[:<port>]/<chemin>/<nom_fichier>#<section>
```

Tout comme pour les fichiers dans un système de fichiers, une ressource internet peut être spécifié soit par son URL absolu soit par son URL relatif. Un URL relatif désigne une ressource qui peut être désigné relativement au document courant. Les URL relatifs ne peuvent référencer que des ressources qui se trouvent dans le même hôte que le document où figure l'URL.

Le protocole HTTP

Le protocole de communication HHTP consiste :

1. établir une connexion TCP avec le serveur sur le port dédié (80 par défaut, ou bien le port spécifié dans l'URL).
2. émettre la requête d'extraction du document spécifié par l'URL
3. Réception de la réponse
4. clôture de connexion

A TERMINER

Le protocole MIME

A TERMINER

CGI

Les *CGI* (*Commun Gateway Interface*) sont des programmes qui résident et s'exécutent sur le serveur. ces programmes sont déclenchés en réponse à la demande d'un client à travers une page HTML. C'est une manière de réaliser des pages WEB dynamiques.

Comme ce sont des programmes qui résident sur le serveur et qui s'exécutent sur le serveur, il n'y a aucune restriction quant au choix du langage utilisé. Dans la pratique, on constante que les CGI sont, le plus souvent, des programmes réalisés en C, Perl ou AppleScript ; même s'il est tout à fait possible de réaliser des CGI en Java, C++, Pascal, Prolog, Lisp, et tout autre langage de programmation.

Le package java.net

Tout ce qui concerne la programmation réseau est contenu dans le package `java.net` excepté les manipulations spécifiques aux applets que l'on trouve dans le package `java.applet`.

la classe `java.net` contient les classes

- Authenticator
- ContentHandler
- DatagramPacket
- DatagramSocket
- DatagramSocketImpl
- HttpURLConnection
- InetAddress
- JarURLConnection
- MulticastSocket
- NetPermission
- PasswordAuthentication
- ServerSocket
- Socket
- SocketImpl
- SocketPermission
- URL
- URLClassLoader
- URLConnection
- URLEncoder
- URLStreamHandler

et les interfaces

- ContentHandlerFactory
- FileNameMap
- SocketImplFactory
- URLStreamHandlerFactory

39.2 Adressage IP

Comme nous l'avons dit, un hôte sur le réseau peut être nommé soit par son adresse internet soit son nom DNS; leur manipulation se fait par la classe `java.net.InetAddress`.

```
public final class InetAddress extends Object implements Serializable {
public boolean isMulticastAddress()
public String getHostName()
public byte[] getAddress()
public String getHostAddress()
public int hashCode()
public boolean equals(Object obj)
public String toString()
public static InetAddress getByName(String host) throws UnknownHostException
public static InetAddress[] getAllByName(String host) throws UnknownHostException
public static InetAddress getLocalHost() throws UnknownHostException
}
```

Cette classe ne possède pas de constructeur. En général, les instances de `InetAddress` s'obtiennent comme résultat des méthodes `static` : `getByName`, `getAllByName` et `getLocalHost`.

`public static InetAddress getByName (String host)` *`throws UnknownHostException`*

Retourne un objet de la classe InetAddress codant l'adresse IP d'un hôte. L'argument `host` est le nom DNS de l'hôte ou son adresse IP.

`public static InetAddress[] getAllByName (String host)` *`throws UnknownHostException`*

Retourne un tableau d'objets de la classe InetAddress codant toutes les adresses IP d'un hôte. L'argument `host` est le nom DNS de l'hôte ou son adresse IP.

`public static InetAddress getLocalHost ()` *`throws UnknownHostException`*

Retourne un objet de la classe InetAddress codant l'adresse IP du hôte local.

`public String getHostName ()`

Retourne une chaîne de caractères contenant le nom d'hôte de l'objet `InetAddress` ou son adresse IP (si le nom n'existe pas).

`public byte[] getAddress ()`

Retourne un tableau d'octet contenant les 4 octets de l'adresse IP de l'objet `InetAddress`.

`public String getHostAddress ()`

Retourne une chaîne de caractères contenant les 4 octets de l'adresse IP de l'objet `InetAddress` ; cette chaîne est de la forme `"%d.%d.%d.%d"`.

```
import java.net.*;

public class Adresses {
   public static void main(String[] args) {
      for(int i=0 ; i< args.length; i++) {
         try {
            InetAddress[] inetadrs = InetAddress.getAllByName(args[i]);
            System.out.println(args[i] + ":");
            for(int j=0 ; j<inetadrs.length ; j++) System.out.println(inetadrs[j]);
         }
            catch (UnknownHostException e) { System.out.println(args[i] + " inconnu !!! "); }
      }
   }
}
```

Applet Demo

39.3 Les URLs

Comme nous l'avons déjà dit, une URL désigne une ressource dans une réseau sous le format suivant :

`protocole://nom_hote[:port]/chemin/nom_fichier#section`

En *Java*, l'URL n'est pas représenté par une chaîne de caractères mais par une instance de la classe `java.net.URL`

```
public final class URL extends Object implements Serializable, Comparable {
   public URL(String protocol, String host, int port, String file) throws MalformedURLException
   public URL(String protocol, String host, String file) throws MalformedURLException
   public URL(String spec) throws MalformedURLException
   public URL(URL context, String spec) throws MalformedURLException
   protected void set(String protocol, String host, int port, String file, String ref)
   public int getPort()
   public String getProtocol()
```

```
    public String getHost()
    public String getFile()
    public String getRef()
    public boolean equals(Object obj)
    public int hashCode()
    public int compareTo(Object url)
    public boolean sameFile(URL other)
    public String toString()
    public String toExternalForm()
    public URLConnection openConnection() throws IOException
    public final InputStream openStream() throws IOException
    public final Object getContent() throws IOException
    public static void setURLStreamHandlerFactory(URLStreamHandlerFactory fac)
}
```

La classe `URL` dispose de quatre constructeurs :

- `public URL (String prot, String h, int p, String f)` *throws MalformedURLException*

  Crée une instance de **URL** à partir du protocole **prot**, de l'hôte **h**, du port **p** et du fichier **f**.

  ```
  URL url = new URL("http", "gbm.esil.univ-mrs.fr", 80, "eleves/index.html");
  ```

- `public URL (String protocol, String host, String file)` *throws MalformedURLException*

  Crée une instance de **URL** à partir du protocole **prot**, de l'hôte **h** et du fichier **f**. Le port choisi est le port par défaut du protocole spécifié.

  ```
  URL url = new URL("http", "gbm.esil.univ-mrs.fr", "eleves/index.html");
  ```

- `public URL (String spec)` *throws MalformedURLException*

  Crée une instance de **URL** à partir de sa représentation textuelle.

  ```
  URL url = new URL("http://gbm.esil.univ-mrs.fr/eleves/index.html");
  ```

- `public URL (URL context, String spec)` *throws MalformedURLException*

  ```
  URL gbm = new URL("http://gbm.esil.univ-mrs.fr");
  URL eleves = new URL(gbm, "eleves/index.html");
  ```

Tous ces constructeurs lève l'exception **MalformedURLException** lorsque le protocole spécifié est inconnu. Vous aurez noté qu'une instance de **URL** est une sorte de constante puisque la classe **URL** ne fournit aucune méthode pour modifier le protocole ou l'hôte, etc.

Par contre, il est possible de consulter les attributs d'une **URL**. Les méthodes **getPort**, **getProtocol**, **getHost**, **getFile** et **getRef** permettent de récupérer le port, le protocole, l'hôte, le fichier et la section du fichier.

Pour illustrer tout ceci, voici un exemple provenant du tutorial de JDK :

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

public class ParseURL {
    public static void main(String[] args) throws Exception {
        URL aURL = new URL("http://java.sun.com:80/docs/books/tutorial/intro.html#DOWNLOADING");
        System.out.println("protocol = " + aURL.getProtocol());
        System.out.println("host = " + aURL.getHost());
        System.out.println("filename = " + aURL.getFile());
        System.out.println("port = " + aURL.getPort());
        System.out.println("ref = " + aURL.getRef());
    }
}
```

et voici le résultat de l'exécution de ce programme :

```
protocol = http
host = java.sun.com
filename = /docs/books/tutorial/intro.html
port = 80
ref = DOWNLOADING
```

Pour récupérer des données à partir d'une **URL**, la classe **URL** fournit les méthodes **openConnection**, **openStream** et **getContent** :

`public URLConnection openConnection ()` *throws IOException*

Retourne une instance de la classe **URLConection** qui représente une connexion vers vers l'objet désigné par l'url. Ceci peut sembler un peu mystérieux pour le moment ; tout ceci devrai s'éclaircir un peu plus loin (voir 39.12).

`public final Object getContent ()` *throws IOException*

Retourne le contenu de l'url. C'est un raccourci pour **openConnection().getContent()**

`public final InputStream openStream ()` *throws IOException*

Cette méthode établit la communication entre le client et le serveur en ouvrant un *InputStream*. Equivalent à **openConnection().openStream()**.

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

public class OpenURL {
   public static void main(String[] args) {
      BufferedReader in  = null;
      try {
         if (args.length != 1) {
            System.out.println("usage: java OpenURL <URL>");
            System.exit(0);
         }
         URL url = new URL(args[0]);
         in = new BufferedReader(new InputStreamReader(url.openStream()));

         String s;
         while ( (s=in.readLine()) != null) System.out.println(s);
      }
         catch (IOException e) {System.out.println("Erreur1");}
         catch (Exception e) {System.out.println("Erreur2");}
         finally {
            try { if (in != null) in.close();  }
               catch (Exception e) {System.out.println("Erreur3");}
         }
   }
}
```

et voici une partie de l'affichage produit :

```
Java OpenURL http://gbm.esil.univ-mrs.fr/~tourai/index.html
<HTML>
<HEAD>
<TITLE> Toura&iuml;vane
</TITLE>
<!-- Changed by: Utilisateur Extra, 25-Jun-1996 -->
<!-- Changed by: Touraivane , 15-Jul-1996 -->
<BASE HREF="http://gbm.esil.univ-mrs.fr/Staff/Enseignants/tourai/"></BASE>
</HEAD>
<BODY background="fonds/white.gif" link="#FF0000" vlink="#FF00FF" alink="#FF00FF
">
...
```

A TERMINER `public boolean  sameFile (URL other)`

A TERMINER `public String  toString ()`

A TERMINER `public String  toExternalForm ()`

A TERMINER `public static void  setURLStreamHandlerFactory (URLStreamHandlerFactory fac)`

39.4 URLEncoder

39.5 Les sockets

TCP/IP offre, en particulier, une interface dite *sockets* pour l'écriture d'applications communicantes. Les *sockets* a été introduit avec le système d'exploitation UNIX de Berkeley. L'idée des sockets est de faire en sorte que deux programmes sur des hôtes différents puissent communiquer l'un avec l'autre à travers sans se soucier des détails de bas niveau de la communication réseau.

Les primitives TCP pour les sockets permettent :

- l'attente passive de connexions TCP initiées par des applications distantes ;
- l'initiation de connexions "actives" vers des applications distantes en attente passive ;
- l'acceptation ou le refus d'une connexion initiée par une application distante ;
- l'envoi et la réception de données sur une connexion établie ;
- la terminaison brutale ou ordonnée d'une connexion établie ;
- le test d'une connexion.

Les sockets ne sont pas des objets spécifiquement réseaux, et encore moins spécifiquement TCP/IP. Ce sont des objets génériques qui doivent être paramétrés lors de leur création selon l'utilisation prévue :

- socket pour communication inter-processus classique
- socket pour communication réseau en mode datagramme
- socket pour communication réseau en mode connecté

Dans la communication par sockets, il existe toujours une machine qui joue le rôle du serveur et l'autre (ou les autres) qui joue le rôle de client.

Le serveur est une application qui tourne sur un hôte et qui est à l'écoute des requêtes d'un ou de plusieurs clients sur un port particulier.

Le client est une application qui tourne un hôte (pas forcément différent de celui sur lequel tourne le serveur). Il doit connaître l'hôte et le port sur lequel le serveur est à l'écoute ; le client tente alors une connexion au serveur sur l'hôte et le port approprié.

Lorsque le serveur est conçu pour communiquer avec plusieurs clients, quand tout se passe bien,

- il connecte le client sur un nouveau numéro de port
- il se remet en attente de connexion sur le port original

Le client et le serveur peuvent alors communiquer l'un avec l'autre en écrivant et en lisant sur les sockets.

Les programmes serveurs doivent toujours être actifs lorsque des clients initient une connexion. Il existe deux façons d'assurer que ces serveurs soient activés :

- lancement "manuel" des programmes serveurs,
- utilisation du "super serveur INETD".

Activation "manuelle"

Le lancement manuel signifie que le programme serveur s'exécute en permanence, même s'il n'est pas appelé par des programmes clients. Compte tenu du grand nombre de services réseaux disponibles sur une machine donnée, de nombreux processus, oisifs en attente mais consommant du temps CPU et des ressources, existent sur le système. Cette solution est peu recommandée.

Utilisation de INETD sous UNIX

La deuxième solution est l'utilisation d'un serveur particulier, le démon `inetd`, qui réalise les attentes pour le compte d'autres programmes serveurs. De plus, l'utilisation du réseau par un démon lancé par `inetd` est cachée : le programmeur utilise les descripteurs de fichiers correspondant à l'entrée et à la sortie standard du programme. La redirection de ces deux descripteurs vers un socket est réalisée de façon implicite et systématique par le processus `inetd` lui-même.

Le fichier /etc/inetd.conf

Le démon `inetd`, lors de son activation, consulte un fichier de configuration décrivant les services à assurer sur le système. Le fichier s'appelle `/etc/inetd.conf`, et contient un ensemble de lignes au format suivant :

```
service       stream  tcp      nowait  root    /usr/etc/progd  progd
```

Une telle ligne signifie que service est implanté sur cette machine, que les sockets utilisés sont de type `stream`, que le protocole utilisé est `tcp`, que le programme à activer lors d'une demande de service est `/usr/etc/progd`, que le nom à donner au programme sera `progd`, que le programme doit s'exécuter avec comme propriétaire root.

Le paramètre `nowait` indique que plusieurs instances de ce programme peuvent être activées sans attendre la terminaison des instances précédantes.

Le fichier /etc/services

Les ports à utiliser pour les sockets (ici les sockets TCP) sont définis dans un fichier particulier, `/etc/services`, qui contiendra la ligne suivante :

```
service       78/tcp   alias_service
```

Cette ligne indique que pour le service défini dans le fichier `inetd.conf`, il faut utiliser le port 78 du protocole `tcp`. Le reste de la ligne est un alias possible pour le nom du service. Une fois ces informations récupérées, le serveur `inetd` va créer un socket, lui associer le numéro de port, et se mettre en attente d'une indication d'établissement de connexion à ce port. Comme de nombreux services existent, le superserveur `inetd` crée de nombreux sockets, assigne les numéros de port adéquats, se met en attente, etc..

Cas de services UDP

Dans le cas de service utilisant udp, le fichier `inetd.conf` contient des informations légèrement différentes :

```
service       dgram   udp     wait     root    /usr/etc/progd        progd
```

Le paramètre `dgram` identifie un socket de type datagramme, `udp` indique que le protocole utilisé est `UDP`, `wait` indique qu'il faut attendre la terminaison d'une instance du service avant de pouvoir en lancer la suivante. Notons que si les programmes serveurs sont conçus pour s'exécuter en parallèle (voir chapitre 5), on peut trouver la ligne suivante dans le fichier /etc/inetd.conf :

```
service       dgram   udp     nowait   root    /usr/etc/progd        progd
```

Dans les deux cas, on devra trouver la ligne correspondante dans le fichier `/etc/services` :

```
service       78/udp   alias_service
```

39.6 Connexion TCP et sockets

Java fournit deux classes pour la gestion des sockets :

- la classe `Socket` qui permet d'initialiser la connexion avec un serveur
- la classe `ServerSocket` qui permet de créer un serveur à l'écoute des connexions à venir.

39.6.1 Le client TCP

```
public class Socket {
   protected Socket()
   protected Socket(SocketImpl impl) throws SocketException
   public Socket(String host, int port) throws UnknownHostException, IOException
   public Socket(InetAddress address, int port) throws IOException
   public Socket(String host, int port, InetAddress localAddr, int localPort) throws IOException
   public Socket(InetAddress address, int port, InetAddress localAddr, int localPort) throws IOException
   public Socket(String host, int port, boolean stream) throws IOException
   public Socket(InetAddress host, int port, boolean stream) throws IOException
   public InetAddress getInetAddress()
   public InetAddress getLocalAddress()
   public int getPort()
   public int getLocalPort()
   public InputStream getInputStream() throws IOException
   public OutputStream getOutputStream() throws IOException
   public void setTcpNoDelay(boolean on) throws SocketException
   public boolean getTcpNoDelay() throws SocketException
   public void setSoLinger(boolean on, int val) throws SocketException
   public int getSoLinger() throws SocketException
   public void setSoTimeout(int timeout) throws SocketException
   public int getSoTimeout() throws SocketException
   public void close() throws IOException
   public String toString()
   public static void setSocketImplFactory(SocketImplFactory fac) throws IOException
}
```

Une application cliente doit ouvrir une connexion sur en créant un `socket`.

```
try {
   socket sock = new Socket("gbm.esil.univ-mrs.fr", 2000);
}
catch (UnknoxnHostException e) { System.out.println("Hôte inconnu"); System.exit(-1); }
catch (IOException e) { System.out.println("Erreur lors de connexion"); System.exit(-1);}
```

La connexion établie, les canaux d'entrée/sortie s'obtiennent grâce aux méthode `getInputStream` et `getOutputStream`.

Pour tester le client, mous allons utiliser une application serveur qui existe sur tous les systèmes UNIX et qui est à l'écoute du port 7 : il s'agit du serveur `echo` qui se contente de renvoyer au client la chaîne que ce dernier lui envoie.

```
import java.io.*;
import java.net.*;
public class Client {
   public static void main(String[] args) throws Exception, NumberFormatException {
      Socket serveur = new Socket("gbm.esil.univ-mrs.fr", Integer.parseInt(args[0]));
      PrintWriter Sout = new PrintWriter(serveur.getOutputStream());
      BufferedReader Sin = new BufferedReader(new InputStreamReader(serveur.getInputStream()));
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      String jEnvoie, jeReçois;
      do  {
         jEnvoie = in.readLine();
         Sout.println(jEnvoie);
         Sout.flush();
         jeReçois = Sin.readLine();
         System.out.println(jeReçois);
      }
      while (! jEnvoie.equals("fin"));
      Sin.close();
      Sout.close();
      serveur.close();
   }
}
```

39.6.2 Le serveur TCP

```
public class ServerSocket {
```

```
    public ServerSocket(int port) throws IOException
    public ServerSocket(int port, int backlog) throws IOException
    public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException
    public InetAddress getInetAddress()
    public int getLocalPort()
    public Socket accept() throws IOException
    protected final void implAccept(Socket s) throws IOException
    public void close() throws IOException
    public void setSoTimeout(int timeout) throws SocketException
    public int getSoTimeout() throws IOException
    public String toString()
    public static void setSocketFactory(SocketImplFactory fac) throws IOException
}
```

Une application serveur doit créer un instance **ServerSocket** :

```
try {
   ServerSocket sock = new ServerSocket(2000);
}
catch (IOException e) {
   System.out.println("Erreur de création du serveur sur le port 2000");
   System.exit(-1);
}
```

Une fois le serveur crée, il va se mettre en attente d'une connexion cliente :

```
Socket sockClient;
try {
   sockClient= serverSocket.accept();
}
catch (IOException e) {
   System.out.println("Echec de la mise en attente sur le port 2000");
   System.exit(-1);
}
```

La méthode **accept** permet au serveur d'être à l'écoute du port 2000 et d'attendre une future connexion cliente. Lorsqu'une telle connexion s'établit, la varibale **sockClient** permet de récupérer le canal de communication établit avec le client grâce aux méthodes **getInputStream** et **getOutputStream**.

Pour illustrer le fonctionnement du serveur, programmons le fameux serveur **echo**.

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

public class TestEcho {
   public static void main(String[] args) throws Exception {
      ServerSocket serveur = new ServerSocket(2000);
      while (true) {
         Socket client = serveur.accept();
         PrintWriter Cout = new PrintWriter(client.getOutputStream());
         BufferedReader Cin = new BufferedReader(new InputStreamReader(client.getInputStream()));
         String jeReçois;
         do  {
            jeReçois = Cin.readLine();
            Cout.println("Vous avez dit " + jeReçois);
            Cout.flush();
         }
         while (! jeReçois.equals("fin"));
         Cin.close();
         Cout.close();
         client.close();
      }
   }
}
```

39.6.3 Connexions multiples

Le serveur **TestEcho** est bien trop simpliciste. En effet, ce serveur n'accepte qu'un client à la fois. Que se passe-t-il si plusieurs clients essayent de se connecter en même temps ? Le premier qui se connecte monopolise la connexion et les autres clients doivent attendre la fin de communication du premier pour pouvoir dialoguer avec le serveur à tour de rôle.

Cette solution est très restrictive. Nous allons modifier le serveur pour qu'il puisse répondre à plusieurs clients à la fois. Pour ce faire, le serveur devra créer un thread par client connecté de manière à pouvoir se remettre en attente d'un éventuel futur client.

Le squelette d'un programme qui réalise ceci est de la forme :

```
while (true) {
   attente d'un connexion
   créer un thread pour dialoguer avec le client qui s'est connecté
}
```

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

public class TestEchoMultiple {
   public static void main(String[] args) throws Exception {
      ServerSocket serveur = new ServerSocket(2000);
      while (true) {
         Socket client = serveur.accept(); System.out.println("C'est par");
         new EncoreUnClient(client).start();
      }
   }
}

class EncoreUnClient extends Thread {
   Socket client;
   public EncoreUnClient(Socket client) {
      super("Clients");
      this.client = client;
   }

   public void run()  {
      PrintWriter Cout = null;
      BufferedReader Cin = null;
      try {
      System.out.println("C'est parti");
         Cout = new PrintWriter(client.getOutputStream());
         Cin = new BufferedReader(new InputStreamReader(client.getInputStream()));
         String jeReçois;
         do  {
            jeReçois = Cin.readLine();
            Cout.println("Vous avez dit " + jeReçois);
            Cout.flush();
         }
         while (! jeReçois.equals("fin"));

         if (Cin != null) Cin.close();
         if (Cout != null) Cout.close();
         client.close();
      }
         catch (IOException e) { e.printStackTrace(); }
   }
}
```

39.7 Datgrammes UDP et sockets

On dispose de deux classes pour la communication par UDP :

- la class **DatagramPacket** qui en charge de l'emballage et du déballage des données en paquets.
- la class **DatagramSocket** qui se charge d'envoyer et de recevoir les paquets frabriqués par un **DatagramPacket**.

Ainsi, l'envoi de données par UDP consite à les confier à un **DatagramPacket** en vue de leur emballage sous forme de paquets et à envoyer ce **DatagramPacket** à un **DatagramSocket** pour leur expédition. Un **DatagramSocket** se charge de la récéption ; les paquets reçus doivent être mis dans un **DatagramPacket** en vue de leur déballage.

39.8 La classe DatagramPacket

```
public final class DatagramPacket {
   public DatagramPacket(byte ibuf[], int ilength)
   public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int iport)
   public synchronized InetAddress getAddress()
   public synchronized int getPort()
   public synchronized byte[] getData()
   public synchronized int getLength()
   public synchronized void setAddress(InetAddress iaddr)
   public synchronized void setPort(int iport)
```

```
      public synchronized void setData(byte ibuf[])
      public synchronized void setLength(int ilength)
   }
```

La classe `DatagramPacket` dispose de deux constructeurs

```
public DatagramPacket(byte ibuf[], int ilength)
public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int iport)
```

où

- **ibuf** désigne le tampon,
- **ilength** désigne la taille du paquet à envoyer (forcément inférieure à la taille du tampon),
- **iaddr** l'adresse de destination,
- **iport** le port de destination.

Le premier constructeur est utiliser pour recevoir les paquets et le deuxième pour en envoyer.

Les méthodes **getAddress**, **getPort**, **getLength** et **getData** permettent de récupérer respectivement l'adresse de destination, le port de destination, la taille du paquet et les données contenues dans le paquet.

Les méthodes **setAddress**, **setPort**, **setLength** et **setData** assigne respectivement l'adresse de destination, le port de destination, la taille du paquet et les données contenues dans le paquet.

39.9 La classe DatagramSocket

```
Class java.net.DatagramSocket {
   public DatagramSocket() throws SocketException
   public DatagramSocket(int port) throws SocketException
   public DatagramSocket(int port, InetAddress laddr) throws SocketException
   public void send(DatagramPacket p) throws IOException
   public synchronized void receive(DatagramPacket p) throws IOException
   public InetAddress getLocalAddress()
   public int getLocalPort()
   public synchronized void setSoTimeout(int timeout) throws SocketException
   public synchronized int getSoTimeout() throws SocketException
   public void close()
}
```

La classe `DatagramSocket` possède trois constructeurs :

- `public DatagramSocket() throws SocketException`
  Crée un socket datagramme et l'associe à un port quelconque de la machine locale.
- `public DatagramSocket(int numport) throws SocketException`
  Crée un socket datagramme et l'associe à un port **numport** de la machine locale.
- `public DatagramSocket(int numport, InetAddress laddr) throws SocketException`
  Crée un socket datagramme et l'associe à un port **numport** de la machine d'adresse **laddr**.

Les méthodes **send** et **receive** de cette classe permettent d'envoyer et de recevoir les packets UDP.

La méthode **send** lève une exception lorsque le **SecurityManager** refuse l'envoi d'un paquet vers l'hôte spécifié. C'est le cas ou une erreur peut se produire.

La méthode **receive** (tout comme la méthode **accept** de la classe **SocketServer**) se met en attente et lève également une exception lorsqu'il y une erreur dans la reception du paquet UDP. La méthode receive doit disposer d'un tampon assez grand pour le packet à recevoir. Dans le cas contraire, le paquet est tronqué à la taille du tampon.

39.10 Le client UDP

Une application cliente doit ouvrir une connexion sur en créant un **socket**.

```
try {
   socket sock = new Socket("gbm.esil.univ-mrs.fr", 2000);
}
catch (UnknoxnHostException e) { System.out.println("Hôte inconnu"); System.exit(-1); }
catch (IOException e) { System.out.println("Erreur lors de connexion"); System.exit(-1);}
```

La connexion établie, les canaux d'entrée/sortie s'obtiennent grâce aux méthode **getInputStream** et **getOutputStream**.

Pour tester le client, mous allons utiliser une application serveur qui existe sur tous les systèmes UNIX et qui est à l'écoute du port 7 : il s'agit du serveur **echo** qui se contente de renvoyer au client la chaîne que ce dernier lui envoie.

```
import java.io.*;
import java.net.*;
public class Client {
   public static void main(String[] args) throws Exception, NumberFormatException {
      Socket serveur = new Socket("gbm.esil.univ-mrs.fr", Integer.parseInt(args[0]));
```

```
         PrintWriter Sout = new PrintWriter(serveur.getOutputStream());
         BufferedReader Sin = new BufferedReader(new InputStreamReader(serveur.getInputStream()));
         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
         String jEnvoie, jeReçois;
         do  {
            jEnvoie = in.readLine();
            Sout.println(jEnvoie);
            Sout.flush();
            jeReçois = Sin.readLine();
            System.out.println(jeReçois);
         }
         while (! jEnvoie.equals("fin"));
         Sin.close();
         Sout.close();
         serveur.close();
      }
   }
```

39.11 Le serveur UDP

Et voici à présent un serveur UDP à l'image du serveur TCP que nous vu. Il faut donc :

– Créer un **DatagramSocket**

```
DatagramSocket socket = new DatagramSocket(2001);
```

– Créer un paquet pour recevoir les données

```
DatagramPacket paquet  = new DatagramPacket(buf, buf.length);
```

– Se mettre en attente d'un paquet

```
socket.receive(paquet);
```

– Récupérer l'expéditeur (hôte et port) et le contenu du message

```
InetAddress address = paquet.getAddress();
int port = paquet.getPort();
jeReçois = new String(paquet.getData(), 0, paquet.getLength());
```

– Renvoyer un message à l'expéditeur dans un paquet

```
paquet = new DatagramPacket(buf, buf.length, address, port);
socket.send(paquet);
```

Et voici le programme complet.

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

public class UdpTestEcho {
   public static void main(String[] args) throws Exception {

      String jeReçois;
      DatagramSocket socket = new DatagramSocket(2001);
      while (true) {

         byte[] buf = new byte[56];
         DatagramPacket paquet  = new DatagramPacket(buf, buf.length);
         socket.receive(paquet);
         jeReçois = new String(paquet.getData(), 0, paquet.getLength());
         System.out.println("j'ai recu : " + jeReçois + " " + jeReçois.length());

         InetAddress address = paquet.getAddress();
         int port = paquet.getPort();
         paquet = new DatagramPacket(buf, buf.length, address, port);
         socket.send(paquet);
      }
   }
}
```

39.12 URLConnection

La classe `URLConection` est une une classe abstraite qui permet une interaction de plus haut niveau que l'utilisation brute de la classe `URL`. En fait, cette classe permet de manipuler le contenu d'une *URL* bien plus simplement : entete MIME, téléchargement, requete CGI, etc.

On entend par gestion de protocole la prise en charge du protocole, le traitement des données spécifiques et leur présentation à l'utilisateur. La classe `URLConnection` sert également à cette gestion ; ce que nous verrons dans la section suivante.

```
public abstract class URLConnection  {
   protected URL url
   protected boolean doInput
   protected boolean doOutput
   protected boolean allowUserInteraction
   protected boolean useCaches
   protected long ifModifiedSince
   protected boolean connected

   protected URLConnection(URL url)

   public static FileNameMap getFileNameMap()
   public static void setFileNameMap(FileNameMap map)
   public abstract void connect() throws IOException
   public URL getURL()
   public int getContentLength()
   public String getContentType()
   public String getContentEncoding()
   public long getExpiration()
   public long getDate()
   public long getLastModified()
   public String getHeaderField(String name)
   public int getHeaderFieldInt(String name, int Default)
   public long getHeaderFieldDate(String name, long Default)
   public String getHeaderFieldKey(int n)
   public String getHeaderField(int n)
   public Object getContent() throws IOException
   public InputStream getInputStream() throws IOException
   public OutputStream getOutputStream() throws IOException
   public String toString()
   public void setDoInput(boolean doinput)
   public boolean getDoInput()
   public void setDoOutput(boolean dooutput)
   public boolean getDoOutput()
   public void setAllowUserInteraction(boolean allowuserinteraction)
   public boolean getAllowUserInteraction()
   public static void setDefaultAllowUserInteraction(boolean defaultallowuserinteraction)
   public static boolean getDefaultAllowUserInteraction()
   public void setUseCaches(boolean usecaches)
   public boolean getUseCaches()
   public void setIfModifiedSince(long ifmodifiedsince)
   public long getIfModifiedSince()
   public boolean getDefaultUseCaches()
   public void setDefaultUseCaches(boolean defaultusecaches)
   public void setRequestProperty(String key, String value)
   public String getRequestProperty(String key)
   public static void setDefaultRequestProperty(String key, String value)
   public static String getDefaultRequestProperty(String key)
   public static synchronized void setContentHandlerFactory(ContentHandlerFactory fac)
   protected static String guessContentTypeFromName(String fname)
   public static String guessContentTypeFromStream(InputStream is) throws IOException
}
```

On obtient généralement une instance de la classe `URLConection` en retour de l'invocation de la méthode `openConnection` de la classe `URL`.

```
...
URL u = new URL("http://gbm.esil.univ-mrs.fr/index.html");
URLConnection uc = u.openConnection();
...
```

La seule méthode abstraite de cette classe est la méthode `connect` ; toutes les autres méthodes sont implantées.

Les méthodes principales de cette classe sont `connect`, `getContent`, `getInputStream` et `getOutputStream`.

La méthode abstraite `connect` permet d'ouvrir une connection vers un url. Cette méthode est redéfinie par les sous classes de la classe `URLConnection` ; c'est le cas de la classe `HttpURLConnection`. La méthode `connect` est rarement utilisé, c'est la méthode `openConnection` qui invoque cette méthode pour la bonne version de protocole.

La méthode **getContent** est celle invoquée par la méthode de même nom de la classe **URL** : elle retoune le contenu d'une url. L'exception **UnknownServiceException** est levée si le protocole et/ou le type MIME de l'url n'est pas reconnu.

La méthode **getInputStream** peut être utilisé lorsque **getContent** ne peut l'être. L'utilisation de cette méthode est similaire à la méthode **openStream** de la classe **URL**.

```
...
URL = u = new URL(...);
URLConnection uc = u.openConnection();
BufferedReader in = new BufferedReader(new InputStreamReader(url.openInputStream()));
...
String s = in.readLine();
...
```

La méthode **getOutputStream** permet d'envoyer des données à une url (méthode POST des CGI par exemple). Avant d'utiliser cette méthode, il faut invoquer la méthode **setDoOutput** car, par défaut, les **URLConnection** ne permettent pas d'envoyer des données.

```
...
URL = u = new URL("http://gbm.esil.univ-mrs/cgi_bin/programme);
URLConnection uc = u.openConnection();
uc.setDoOutput(true);
PrintWriter out = new PrintWriter(url.openOutputStream());
out.writeln(...);
...
```

Les méthodes **getContentType**, **getContentLength**,**getContentEncoding** permettent de consulter le type MIME, la longueur et le type d'encodage.

Les méthodes **getHeaderField**, **getHeaderFieldKey**, **getHeaderFieldKeyDate** et **getHeaderFieldKeyInt** permettent de récupérer les informations insérées dans les entêtes MIME.

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

public class TestContent {
   public static void main(String[] args) {
      BufferedReader in  = null;
      try {
         if (args.length != 1) {
            System.out.println("usage: java OpenURL <URL>");
            System.exit(0);
         }
         URL url = new URL(args[0]);
         URLConnection uc = url.openConnection();
         System.out.println(uc.getContentType());
         System.out.println(uc.getContentLength());
         System.out.println(uc.getContentEncoding());

         System.out.println(new Date(uc.getDate()));
         System.out.println(new Date(uc.getExpiration()));
         System.out.println(new Date(uc.getLastModified()));

         int i=0;
         String nom=null;
         do {
            System.out.println(uc.getHeaderFieldKey(i) + ": " + (nom=uc.getHeaderField(i++)));
         }
         while (nom != null);

         System.out.println(uc.getHeaderField("content-type"));
         System.out.println(uc.getHeaderField("content-length"));
         System.out.println(uc.getHeaderField("date"));
      }
         catch (IOException e) {
            System.out.println("Erreur1");}
         catch (Exception e) {
            System.out.println("Erreur2");}
         finally {
            try {
               if (in != null) in.close();  }
               catch (Exception e) {
                  System.out.println("Erreur3");}
         }
   }
}
```

A priori, si toutes les conventions de nommage sont respectées, le type MIME peut se déduire aisément de l'extension du nom du la ressource url spécifié.

Extension	Type MIME
.pdf	application/pdf
.zip	application/zip
.tar	application/x-tar
.wav	audio/x-wav
.gif	image/gif
...	...

En exécutant l'exemple précédant, on obtient bien le type des fihciers spécifiés :

```
% java TestContent http://gbm.esil.univ-mrs.fr/~tourai/images/photo.jpg
Content-type: text/html
```

Par contre si l'extension n'est pas correct, on alors des surprises ; le fichier `photo` est une copie renommée du fichier `photo.jpg`

```
% java TestContent http://gbm.esil.univ-mrs.fr/~tourai/images/photo
Content-type: text/plain
```

Aussi, *Java* dispose de deux méthodes pour deviner le type MIME associé à un fichier :

- `protectd Static String guessContentTypeFromName(String)`
  qui tente d'associer un type MIME en fonction de l'extension du nom
- `protectd Static String guessContentTypeFromStream(InputStream is)`
  qui tente d'associer un type MIME en analysant les premier octets.

39.13 Un petit serveur HTTP

A TERMINER

39.14 Un CGI en Java

A TERMINER

39.15 Gestion de protocoles

A TERMINER

39.16 ContentHandler

A TERMINER

39.17 Gestion de protocoles

A TERMINER

39.18 Sockets multipoints

40. Appel de méthodes distantes (RMI)

Sommaire

40.1 Introduction 353
40.2 Un exemple 354
40.2.1 Le serveur *354*
40.2.2 Le client *356*
40.2.3 La compilation *356*
40.2.4 L'exécution *356*

40.1 Introduction

Les systèmes distribuées sont des applications sur des espaces d'adressage différents : ou sur un même hôte ou des hôtes différents. Le problème central de ce type d'applications est la communication entre les différentes applications.

Pour le moment, nous nous sommes contenté de tranférer des données à travers le réseau. Nous l'avons fait les divers protocoles existants : FTP, HTTP, etc. Nous avons vu, qu'avec les sockets, il existe un moyen simple de communication en une application serveur et une application cliente pour transférer des données. Jusqu'à présent, nous ne connaisons que les CGI qui permettent de lancer un programme sur une machine distante.

Les *appels de procédures distantes* (*RPC : **R**emote **P**rocedure **C**all*) est un autre moyen pour exécuter des programmes distants. Le but du service RPC est de permettre la programmation aisée de programmes client-serveur sous une forme transparente aux utilisateurs.

L'idée de base est que le paradigme d'appel de procédure est bien connu des programmeurs. Le RPC va permettre de distribuer (ou répartie) le programme principal et les procédures sur des systèmes distants :

- le programme principal devient un programme client,
- les procédures appelées constituent un programme serveur.

Ainsi, au lieu de manipuler directement des sockets, avec le RPC, le programmeur va avoir l'illusion de faire des appels à des procédures locales alors même que ces appels impliquent une communication avec un hôte distant pour envoyer et recevoir des données.

La difficulté essentielle à résoudre pour offrir un service d'appel de procédures à distance est le maintien des propriétés implicitement associées à un appel de procédure local :

- Codage et décodage des arguments des procédures et de la valeur retournée par les fonctions,
- ininterruptibilité (le programme appelant est suspendu),
- effets de bord permanents,
- fiabilité (la procédure retourne toujours).

Le client

Le RPC maintient une certaine transparence : du point de vue du client, la procédure appelée apparaît comme une procédure locale. En fait, la procédure distante appelée est représentée localement par une procédure particulière appelée *stub client* ou *souche cliente* : c'est cette procédure locale que le client invoque réellement. Cette souche cliente utilise ensuite le réseau pour contacter la procédure distante implantée par le serveur. Le rôle de la souche cliente est multiple :

- prendre en charge les communications nécessaires pour l'échange des paramètres et résultats de la procédure,
- prendre en charge les mécanismes de détection et reprise sur erreur,
- assurer le formatage correct des paramètres et résultats.

Le serveur

Au niveau du serveur, on définit la notion de programme serveur : il est identifié par un numéro de programme non ambigu, un numéro de version qui permet éventuellement la coexistence de versions successives du même programme serveur, et il est constitué d'un ensemble de procédures. Un module particulier contrôle l'accès aux procédures du serveur, le "stub serveur".

Communication stub client/Serveur

La communication entre le *stub client* et le *stub serveur* se fait indirectement via deux services :

- le service XDR (eXternal Data Representation) qui assure la mise en forme des paramètres échangés (codage, élimination des pointeurs, etc.) ;
- le service RPC qui envoie les requêtes, les réponses, reprend les erreurs, etc.

Les échanges à travers le réseau peuvent se faire soit via TCP, soit via UDP

RPC et RMI

RPC a été conçu pour la programmation procédurale et ne prévoit rien en ce qui concerne la programmation objet. L'adaptation de RPC à la programmation objet a pour nom RMI (***R**emote **M**ethod **I**nvocation*). S'il existe des RMI qui peuvent s'adapter aux objets *Java*, le langage *Java* fournit un RMI qui est propre aux objets *Java*. Il s'agit ici de travailler dans un monde homogène *Java* ; contrairement à CORBA qui présuppose une environnement hétérogène, des langages différents.

Java Remote Invocation Method

On appelle *objets distants* tout objet dont les méthodes sont susceptibles d'être invoquées par une machine virtuelle *Java* autre que celle sur laquelle réside cet objet. Il s'agit généralement d'une machine distante.

Lors de l'invocation d'une méthode distante, il reste un problème délicat posé par le passage de ses arguments et l'objet éventuellement retourné par celle-ci.

A TERMINER

L'ensemble des classes qui prennent en charge la gestion des *RMI* sont regroupées dans les packages `java.rmi`, `java.rmi.server`, `java.rmi.registry` et `java.rmi.dgc`.

Package	
`java.rmi`	package pour la gestion client
`java.rmi.server`	package pour la gestion du serveur
`java.rmi.registry`	package pour localiser les objets distants
`java.rmi.dgc`	récupération de mémoire

40.2 Un exemple

Avant d'entrer dans les détails, voici l'exemple fournit dans le tutorial de jdk.

40.2.1 Le serveur

L'interface java.rmi.Remote

Un objet distant se doit d'implanter une ou plusieurs *interfaces distantes* ; une interface distante est une interface qui étend directement ou indirectement l'interface `java.rmi.Remote`.

L'interface `java.rmi.Remote` ne définit aucune méthode propre, c'est une interface vide qui ne sert qu'à identifer les objets suscpetibles d'être des objets distants.

L'ensemble des exceptions que l'invocation d'une méthode distante peut générée est regroupé dans la classe `java.rmi.RemoteException` et toutes les méthodes implantant cette interface se doivent de renvoyer cette exception.

```
public interface Hello extends java.rmi.Remote {
        String sayHello() throws java.rmi.RemoteException;
}
```

L'implantation de l'interface

A présent, nous allons définir une classe qui implante cette interface ce qui va nous permettre de définir (plus tard) des objets distants qui seront des instances de cette classe. Cette classe doit être une sous classe de la classe `java.rmi.server.UnicastRemoteObject`.

```
public class HelloImpl extends java.rmi.server.UnicastRemoteObject implements Hello {
```

Sans plus de détails, disons que cette classe `java.rmi.server.UnicastRemoteObject` fournit un ensemble de méthodes qui vont permettre l'invocation des méthodes distantes.

Il nous faut, maintenant, implanter la méthod `sayHello` de l'interface `Hello` définit précédemment.

```
public String sayHello() throws java.rmi.RemoteException { return  "Coucou !"; }
```

C'est la seule méthode (d'après notre interface) qui peut être invoqué par le client. Elle se contente de retourner au client la chaîne de caractères `"Coucou !"`. A priori, rien ne distingue cette méthode des autres méthodes que l'on pourrait fournir avec cette classe. Seule son appartenance à l'interface `Hello` lui confère un statut particulier.

Il nous reste à terminer l'implantation de la classe en fournissant la méthode `main`, et le constructeur de notre classe.

Le constructeur de cette classe ne fait rien d'autre que d'appeler le constructeur de la classe mère ; ce qui est le comportement par défaut si l'on ne définit aucun constructeur. Pourtant, ici, il va falloir impérativement, la définir à cause de l'exception qu'elle peut engendrer.

```
public HelloImpl() throws RemoteException { super(); }
```

La méthode `main` se charge des opérations à effectuer sur le serveur. On crée tout d'objet une instance de cette classe que l'on enregistre le *registre de nommage* avec la méthode statique `rebind` de la classe `java.rmi.Naming`.

```
HelloImpl obj = new HelloImpl("Hello");
java.rmi.Naming.rebind("Hello", obj);
System.out.println("Le serveur RMI Hello est actif");
\end{alltt}\end{formatprog}

C'est cet enregistrement qui met à la disposition du client ces objets.
Les clients peuvent alors accès à cet objet ou à la liste des objets disponibles.


Voici le code complet de cette classe.

\begin{formatprog}
\verbatimfile{Programme/rmi/HelloImpl.java}
\end{formatprog}




\formatsection\subsubsection{Fabriquer les stubs et squelettes}\formattext



La partie du codage en \Java\ de notre serveur est à présent fini. IL nous reste à fabriquer les \emph{stubs} et \texttt{squelettes} indisp



L'utilitaire \texttt{rmic} prend la fichier \texttt{.class} et produit
ces \emph{stubs} et \emph{squelettes}:

\begin{formatprog}\begin{alltt}
\emph{% javac HelloImpl.java}
\emph{% rmic HelloImpl}
\end{alltt}\end{formatprog}

Cette dernière commande produit les deux fichiers \texttt{HelloImpl\_Skel.class}
 et \texttt{HelloImpl\_Stub.class}





\formatsection\subsubsection{Mise en route du serveur}\formattext

A présent, tout est prêt pour la mise en route du serveur. Avant toute chose, il faut lancer le programme \texttt{rmiregistry}
\begin{formatprog}\begin{alltt}
\emph{% rmigestry 2000  &     \emph{// sous unix}}
\emph{% start rmirestry 2000 \emph{// sous Windows}}
\end{alltt}\end{formatprog}
```

```
Ce programme s'exécute en tâche de fond et se connect par défaut au port 1099.
Si ce port est pris, il suffit de spécifier le port que l'on veut utiliser.
Ce numéro de port est celui utilisé par le programme \texttt{HelloImpl}:

\begin{formatprog}\begin{verbatim}
        java.rmi.Naming.rebind("//gbm.esil.univ-mrs.fr:2048", obj);
```

Le lancement du serveur proprement dit se fait en excéutant la programme `HelloImpl` :

```
% java HelloImpl &
Le serveur RMI Hello est actif
%
```

A présent tout est prêt pour recevoir des appels distants !!!

40.2.2 Le client

Le client doit définir l'objet distant dont il veut obtenir une référence. Comment un client peut-il désigné un objet distant ? La syntaxe utilisé est propre de la syntaxe des URL HTTP. Le protocole utilisé est `rmi`, l'objet réside sur la machine `gbm.esil.univ-mrs.fr` et le serveur est à l'écoute sur le port `2048` et l'objet recherché s'est fait inscrire sous le nom de `Hello`. Le client désigne alors cet objet par :

```
rmi://gbm.esil.univ-mrs.fr/Hello
```

La classe `Naming` dispose d'une méthode statique `lookup` qui permet de récupérer cet objet :

```
Objet o = Naming.lookup("rmi://gbm.esil.univ-mrs.fr:2048/Hello");
```

Une fois extrait cet objet, il suffit d'invoquer la méthode `sayHello` sur cet objet.

```
Objet o = Naming.lookup("rmi://gbm.esil.univ-mrs.fr/Hello");
```

```
\verbatimfile{Programmes/rmi/CallHello.java}
```

40.2.3 La compilation

```
javac Hello.java HelloImpl.java
```

```
rmic HelloImpl
```

40.2.4 L'exécution

```
start rmiregistry  2048     // Windows
rmiregistry  2048 &         //  Unix
```

```
java HelloImpl &
```

```
java CallHello
```

41. Introduction aux bases de données

Sommaire

41.1 Introduction 357
41.2 Le modèle relationnel 357

41.1 Introduction

41.2 Le modèle relationnel

Une realtion R est un sous enseble du produit cartésien de n ensembles $D_1, D_2, \dots D_n$

42. Introduction à JDBC

Sommaire

42.1 Introduction 359
42.2 Se connecter à une base de données 360
42.3 Envoi de requêtes SQL 361
42.4 Un premier exemple complet 362
42.4.1 Création d'une base 362
42.4.2 Consulter d'une base 362
42.4.3 Modifier d'une base 363
42.4.4 Ajouter un pilote ODBC sur Windows 363
42.4.5 Ajouter un pilote ODBC sur Unix 363

42.1 Introduction

JDBC (***J**ava **D**ata**B**ase **C**onnectivity*) est un ensemble de classes servant à manipuler les bases de données. Avec `JDBC`, *Java* est capable de communiquer avec des bases de variées. Il faut entendre par base de données aussi bien les très grosses bases de données industrielles (tels que *Oracle*, *Informix*, *Sybase*, etc.) que les des bases de données plus modestes tels que *FoxPro*, *MS Access*, *mSQL*. Il permet de manipuler également les fichiers textes ou les feuilles de calculs *Excel*.

Avec *JDBC*, *Java* communique avec potentiellement toutes les bases de données ; autrement dit, une même application peut interagir avec un fichier texte, une feuille de calcul *Excel*, une base de données *Access*, *Oracle*, *Informix*, *Sybase*, etc. Il n'est pas nécessaire d'écrire une application par type de base de données.

L'interaction avec une base de données se déroulent principalement en ces quatre phases successives :

- Chargement et configuration du client qui veut interroger une base de donnée
- Connexion à la base de données
- Exécution des commandes *SQL*.
- Inspection des résultats (si disponible).

A chacune de ces phases, correspond une classe *Java* que l'on trouvera dans le package `java.sql` :

- **`DriverManager`**
- **`Connection`**
- **`Statement`**
- **`ResultSet`**

JDBC fournit un moyen de communication de bas niveau avec les bases de données : *Java* se contente d'envoyer des commandes *SQL* à la base de données.

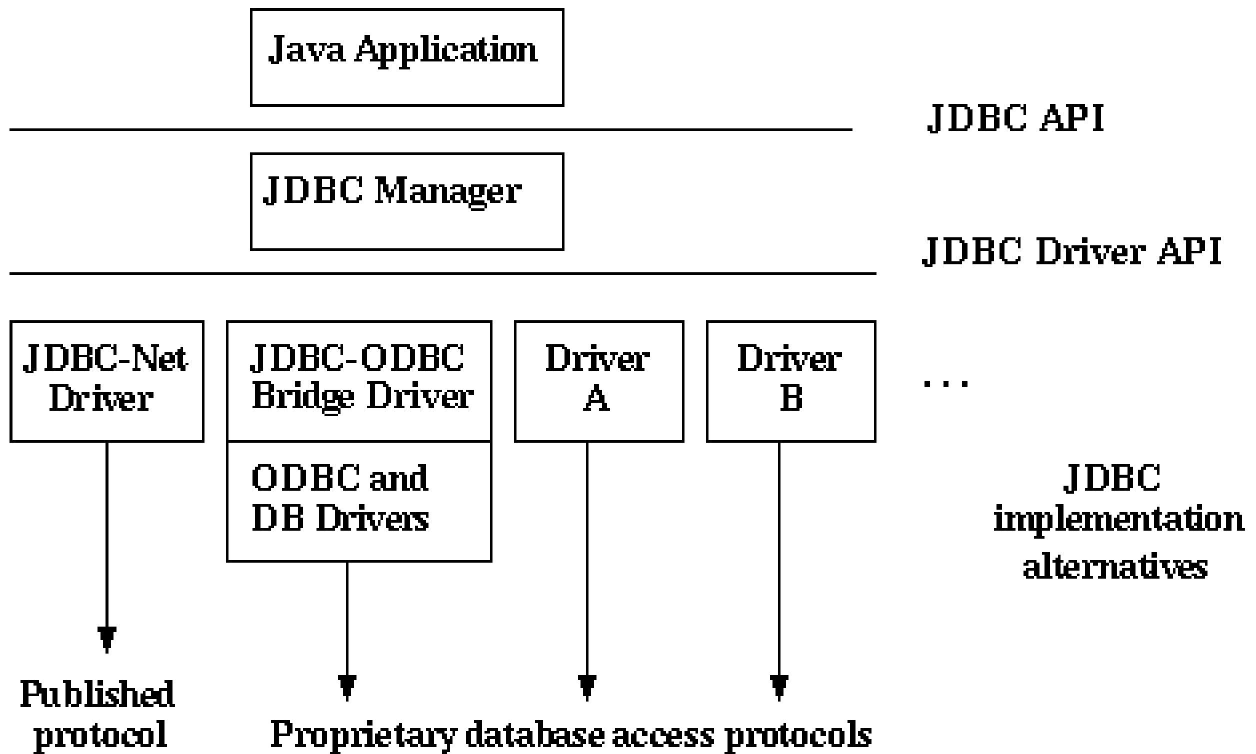

FIG. 42.1: Interface JDBC

JDBC, ODBC, et Cie

Microsoft propose une *API* permettant d'accéder à diverses bases de données. Cette *API*, appelé *ODBC* (***O**pen **D**ata**B**ase **C**onnectivity*) est très certainement l'*API* la plus répandue dans le monde.

Pourquoi avoir alors créé une *API* particulière pour *Java* ? *ODBC* est orienté langage C et cela obligerait à écrire des méthodes natives en *Java*ce que l'on veut absolument éviter pour des raisons de portabilité, sécurité, etc. De plus, *ODBC* est relativement complexe et *JDBC* se veut bien plus simple.

Au lieu de réinventer la roue, *JDBC* est bâti au dessus de ODBC[1] et permet une programmation relativement simple des applications orientées bases de données.

SQL

A TERMINER

Applets, applications et bases de données

A TERMINER

42.2 Se connecter à une base de données

Une connexion est constituée des commandes *SQL* qui sont exécutées et des résultats qui sont retournés. Une même application peut établir plusieurs connexions avec une même base de données et/ou avoir des connexions avec plusieurs bases de données.

La manière standard d'établir une connexion consiste à invoquer la méthode `DriverManager.getConnection`. Cette méthode retourne, lorsque la connexion a pu s'établir avec la base de données, un objet de type `Connection`.

[1] *MicroSoft* semble également s'orienter vers des *API*s plus simple à appréhender avec *RDO*, *ADO* et *OLE DB* qui sont également bâtis au dessus *ODBC*

Voic à titre d'exemple, un bout de code classique qui établit une connexion vers une base de données désigné par `jdbc :odbc :Test` avec pour nom d'utilisateur `"CoursJava"` et pour mot de passe `"ȷ !a-v a"`

```
String url = "jdbc:odbc:Test";
Connection con = DriverManager.getConnection(url, "CoursJava", "\j!a-v~a");
```

Divers types de pilotes JDBC

A TER-MINER

URL JDBC

Une base de données est donc désignée par une *URL*. Rappelons qu'on appelle *URL* (*Uniform Resource Location*) la manière de nommer une ressource sur le réseau *Internet*. Par exemple, `http ://gbm.esil.univ-mrs.fr/eleves/index.html` désigne la page *WEB* d'accueil des élèves du département *GBM* de l'*ESIL*. Un *URL* se compose de trois parties : le protocole, l'hôte et le document. La syntaxe générale d'un *URL* est :

```
<protocole>://<nom_hote>[:<port>]/<chemin>/<nom_fichier>#<section>
```

La syntaxe des *URL*s *JDBC* suivent le même schéma pour identifier une base de données et le pilote approprié pour établir une connexion avec la base :

```
jdbc:<sous protocol>:<nom>
```

La première partie est le protocole utilisé : avec *Java* ce protocole est toujours `jdbc`.

Le sous protocole est le nom du pilote ou le nom de la connectivité pour la base choisie. Dans ce qui va suivre, nous utiliserons le pilote `odbc`.

A TER-MINER dcenaming

Le nom est le moyen d'identifier la base proprement dite. Selon le pilote utilisé, ce nom pourra être subdivisé en plusieurs parties.

Par exemple, si l'on dispose du sous protocole générique `dbnet`, l'URL

```
jdbc:dbnet:\\//gbm:500/eleves
```

identifie la base de données `eleves` qui est accessible à travers le port 500 sur l'hôte `gbm`

Le sous protocole odbc

Le sous protocole `odbc` admet des noms de base suivant la syntaxe :

```
jdbc:odbc:<data-source-name>[;<attribute-name>=<attribute-value>]*
```

Voici des exemples d'URL JDBC :

```
jdbc:odbc:qeor7
jdbc:odbc:wombat
jdbc:odbc:wombat;CacheSize=20;ExtensionCase=LOWER
jdbc:odbc:qeora;UID=kgh;PWD=fooey
```

42.3 Envoi de requêtes SQL

Une fois la connexion établie, pour pouvoir envoyer des requêtes SQL, il faut obtenir une instance de `Statement` sur laquelle on invoque une des méthodes `executeQuery`, `executeUpdate`, etc.

```
Statement stmt = con.createStatement();
ResultSet resultat = stmt.executeQuery("SELECT Nom, Age FROM Etudiant;");
```

Les résultats sont retournés dans une instance de la classe `ResultSet`. Ce résultat est constitué d'une suite de ligne, le passage d'une ligne à la suivante se fait par la méthode `next` de la la classe `ResultSet`. En outre, cette classe fournit un ensemble de méthodes `getXX` pour extraire les champs.

```
while (resultat.next()) {
   String nom = result.getString("Nom");
   int age = result.getInt("Age");
   System.out.println("Nom: " + nom + " Age: ", age);
}
```

42.4 Un premier exemple complet

42.4.1 Création d'une base

```
import java.sql.*;

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

      try { Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); }
         catch (Exception e) {
            System.out.println("Erreur de chargement du pilote JDBC/ODBD.");
            return;
         }
      Statement stmt = null;
      Connection con=null;

      try {
         con = DriverManager.getConnection ("jdbc:odbc:"+args[0], "", "");
         stmt = con.createStatement();
      }
         catch (Exception e) { System.err.println("Erreur de connexion à jdbc:odbc:ElevesJdbc"); }

      try {
         stmt.execute("create table toto (Nom varchar (10), Sexe varchar (1), ExamMars integer, ExamJuin integer;");
         stmt.execute("insert into toto values ('Jean', 'M', 10, 12);");
         stmt.execute("insert into toto values ('Pierre', 'M', 10, 10);");
         stmt.execute("insert into toto values ('Julie', 'F', 16, 12);");
         stmt.execute("insert into toto values ('Paul', 'M', 16, 14);");
         stmt.execute("insert into toto values ('Nathalie', 'F', 12, 12);");
         stmt.execute("insert into toto values ('Jacques', 'M', 8, 6);");
         stmt.execute("insert into toto values ('Renée', 'F', 8, 8);");
         stmt.execute("insert into toto values ('Jean', 'M', 5, 0);");
         stmt.execute("insert into toto values ('Nicole', 'F', 9, 18);");
         stmt.execute("insert into toto values ('Claude', 'M', 11, 13);");
         stmt.execute("insert into toto values ('Marie', 'F', 12, 6);");
         stmt.execute("insert into toto values ('Dominique', 'M', 14, 14);");
         stmt.execute("insert into toto values ('Sylvie', 'F', 6, 7);");
         con.close();
      }
         catch (Exception e) { e.printStackTrace(); }
   }
}
```

42.4.2 Consulter d'une base

```
import java.sql.*;

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

      try { Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); }
         catch (Exception e) {
            System.out.println("Erreur de chargement du pilote JDBC/ODBD.");
            return;
         }
      Statement stmt = null;
      Connection con=null;

      try {
         con = DriverManager.getConnection ("jdbc:odbc:"+args[0], "", "");
         stmt = con.createStatement();
      }
         catch (Exception e) { System.err.println("Erreur de connexion à jdbc:odbc:ElevesJdbc"); }

      try {
         ResultSet result = stmt.executeQuery("SELECT Nom, ExamJuin, ExamMars FROM toto ;");
         while (result.next()) {
            String name = result.getString("Nom");
            double m = result.getInt("ExamMars");
            double j = result.getInt("ExamJuin");
```

```
                System.out.println("La moyenne de "+name+" est : " + ((m+j)/2));
            }
            con.close();
        }
            catch (Exception e) { e.printStackTrace(); }
    }
}
```

42.4.3 Modifier d'une base

42.4.4 Ajouter un pilote ODBC sur Windows

42.4.5 Ajouter un pilote ODBC sur Unix

43. Code natif (JNI)

Sommaire

43.1 Un exemple 366
43.1.1 Déclarer une méthode native 366
43.1.2 Charger la librarie C 366
43.1.3 Créer le fichier .h 366
43.1.4 Implanter la méthode native 367
43.1.5 Créer la bibliothèque dynamique 367
43.1.6 La touche final 367
43.2 Conversion des caractères unicode 368
43.3 Conversion des données 368
43.3.1 Passage de paramètres 368
43.3.2 Types primitifs 368
43.3.3 Les objets 368
43.3.4 Les objets String 369
43.3.5 Les tableaux 370
43.4 Créer et modifier des objets 371
43.5 Signature 371
43.6 Accès aux champs 371
43.7 Invocation de méthodes Java 373
43.7.1 Méthode d'instance 373
43.7.2 Méthode de classe 374
43.8 Exceptions et code natif 374
43.9 Capturer une exception 374
43.10 Lancer une exception 374
43.11 Code natif et threads 375
43.12 Code natif et C++ 375
43.13 Démarrer la machine viruelle Java 375
43.14 La commande javap 376
43.15 Récapitulatif des fonctions JNI 376
43.15.1 Version 376
43.15.2 Opération sur les classes 376
43.15.3 Exceptions 376
43.15.4 Référencs globales et locales 376
43.15.5 Opérations sur les objets 376
43.15.6 Accès aux variables d'instance 376
43.15.7 Invocation de méthodes d'instance 376
43.15.8 Accès aux variables de classe 377
43.15.9 Invocation de méthodes de classe 377
43.15.10 Chaînes de caractères 377
43.15.11 Tableaux 377
43.15.12 Registering Native Methods 377
43.15.13 Monitor Operations 377
43.15.14 Java VM Interface 377

Dans ce chapitre, nous allons voir comment on peut mélanger du code écrit dans un autre langage à l'intérieur du code *Java*.

Précisons tout de suite que cette possibilité ne doit être utilisé que dans des cas exceptionnels. En effet, le fait d'introduire du code natif dans *Java* enlève un certain nombre d'attraits de ce langage : la portabilité, l'indépenance de la plate forme utilisé, possibilité de téléchargement etc...

Pourquoi donc vouloir mélanger *Java* avec un autre langage ?

Tout d'abord, *Java* ne permet d'accéder aux fonctionnalités spécifique d'un système. Par exemple, il n'est pas possible avec *Java* de manipuler des cartes d'acquisition. Toutes ces manipulations proches du système se fera donc dans un autre langage et sera interfacer avec *Java* pour le reste de l'application.

Ensuite, il peut être intéressant d'interfacer un code récent *Java* avec une application existante écrite dans un autre langage. La récriture complète serait bien trop fastidieuse.

Enfin, il existe des application dans lesquelles des bouts critiques doivent s'exécuter de manière extrèmement efficace. On pourra alors récrire ces parties cruciales dans un langage plus efficace (plus proche de la machine).

L'interface *Java* permettant de lier du code natif au code *Java* est désigné par le terme *JNT (**J**ava **N**ative **I**nterface)* Dans ce qui va suivre nous allons plus particulièrement nous intéresser à la programmation des méthodes natives en C et la fin du chapitre présentera les particularité de ce type de lien avec C++.

Dans la communication entre *Java* et un autre langage comme *C*qui n'est pas un langage orienté objet, il y a plusieurs problèmes à résoudre :

- Mise en correspondance des identificateurs *Java*(*Unicode*) en identificateurs *C* ou *C++*.
- Mise en correspondance des méthodes : *C* ne dispose ni d'objets ni de packages, il faut trouver, malgré tout un moyen de faire correpondre une méthode *Java* (qui est sous la forme `package.classe.méthode`) à une fonction *C*.
- Les méthodes (non statiques) s'applique sur des objets (implicitement `this`)
- Lier l'invocation d'une méthode à l'exécution d'une fonction, par exemple
- transmettre les paramètres entre la méthode *Java* et la focntion C
- Manipulation des champs des classes
- Gestion des erreurs avec des exceptions ; C ne dispose d'un système d'exception.
- Fiabilité du code : par exemple, les problèmes de gestion mémoire, de sécurité d'accès aux élements d'un tableaux, etc.
- Création d'objets *Java* depuis C.
- etc.

Applet et natif

43.1 Un exemple

43.1.1 Déclarer une méthode native

Une méthode native est une méthode dont l'implantation est réalisée dans un autre langage que *Java* généralement du C ou C++. Une méthode est qualifiée de native, lorsque il comporte le qualifier (ou *modifier*) `natif`.

```
public native void afficherHello();
```

Il n'est évidemment pas question de donner une quelconque implantation de cette méthode puisqu'elle censé être faite dans un autre langage que *Java*.

43.1.2 Charger la librarie C

Après implantation du code natif C, on devra la copiler et produire une bibliothèque dynamique (voir 43.1.5). Cette bibliothèque devra être chargée au chargement de la classe `Hello`. Le chargement d'une bibliothèque se fait avec la méthode `System.load(...)` ou `System.loadLibrary(...)`. En supposons que cette bibliothèque se nomme `libhello.so` (pour *UNIX*) ou `libhello.dll` (pour *Windows*), voici le code complet de la classe `Hello`.

```
public class Hello
   public native void afficherHello();
   static  System.load("Hello");
```

43.1.3 Créer le fichier .h

Avant d'implanter la fonction *C* qui sera mis en correspondance avec la méthode `afficherHello`, il faut

- compiler le fichier `Hello.java`
- créer la *glue* qui va permettre de lier le code C et le code *Java*.

La *glue* est constituée par un fichier `.h` qui sera automatiquement produit par la commande `javah`.

```
% javac Hello
% javah -jni Hello
```

Cette commande produit le fichier `Hello.h`

```
#include <native.h>
/* Header for class Hello */
#ifndef _Included_Hello
#define _Included_Hello

#pragma pack(4)

typedef struct ClassHello
    char PAD;   /* ANSI C requires structures to have a least one member */
 ClassHello;
HandleTo(Hello);

#pragma pack()

#ifdef __cplusplus
extern "C"
#endif
extern void Hello_afficherHello(struct HHello *);
#ifdef __cplusplus

#endif
#endif
```

vous aurez noté que ce fichier convient également pour interfacer *Java* avec *C++*.

43.1.4 Implanter la méthode native

Ce fichier va servir de glue en *Java* et *C*. A présent, on va pouvoir implanter la méthode native ; il faudra respecter un certain nombre de conventions dans cette implantation :

- Inclure le fichier `jni.h` fourni avec la distribution de jdk ainsi le fichier `Hello.h` précédemment produit.
- Chacune des méthodes à implanter devra être de la forme

```
JNIEXPORT type JNICALL Java_NomDePackage_NomDeClasse_NomDeLaMethode(JNIEnv *env, jobject obj, ...)
```

Lorsque la classe appartient au package sans, comme dans notre exemple, le nom du package et le caractère `_` le précédant sont omis. Pour cet exemple, le prototype de fonction *C* sera donc de la forme

```
JNIEXPORT void JNICALL Java_Hello_afficherHello(JNIEnv *env, jobject obj)
```

On remarquera que toutes ces méthodes admenttent au moins deux arguments : `JNIEnv *env` et `jobject obj`.

La variable `env` est un pointeur vers l'environnment *Java*et c'est cette dernière qui permet de récupérer les arguments de la méthode et tout autre information provenant de l'environnement *Java*.

La variable `obj` est une référence vers l'objet sur lequel la méthode s'applique. Lorsqu'il s'agit d'une méthode static, la valeur de cette référence est nulle.

```
#include <jni.h>
#include "Hello.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_Hello_afficherHello(JNIEnv *env, jclass obj) {
   printf("Coucou !!!\n");
}
```

43.1.5 Créer la bibliothèque dynamique

Il faut à présent compiler le fichier *C* et produire une bibliothèque dynamique

```
gcc -shared -I/usr/include/jdk-1.1.5/ -I/usr/include/jdk-1.1.5/genunix/ HelloImpl.c -o libHello.so
```

Cette commande est évidemment dépendante du système et de la configuration du système utilisée.

43.1.6 La touche final

A présent, tous les ingrédients sont là ; il ne nous reste plus qu'à programmer un exemple d'utilisation.

```
public class Test
   public static void main(String args[]) {
      Hello h = new Hello();
      h.afficherHello();
   }
}
```

Une fois compilé cette classe, il ne reste plus qu' exécuter :

```
% javac Test.java
% java Test
Hello !!!
%
```

43.2 Conversion des caractères unicode

Les caractères *Unicode* compris entre `\u0000` et `\u007f` (caractères *ASCII*) ne sont pas modifiés. Quant aux caractères *Unicode* qui ne sont pas dans cette plage, le caractère `\udddd` est transformer en la suite de caractères `_0dddd`.

43.3 Conversion des données

43.3.1 Passage de paramètres

Pour implanter correctement les méthodes, il faut établir un moyen de convertir les données *Java* en données et inversement. A chaque classe *Java* correspond une structure (`struct`) *C* et chaque membre de la classe garde le même qu'en *Java*, à la conversion des caractères *Unicode* près.

43.3.2 Types primitifs

A chaque type primitif, il existe un type *C* :

Type Java	Type C	Taille (en octet)
boolean	jboolean	1
byte	jbyte	1
char	jchar	2
short	jshort	2
int	jint	4
long	jlong	8
float	jfloat	4
double	jdouble	8

long ?

Lorsq'un type primitif est passé en argument d'une méthode native, ce passage sa fait par valeur.

Param.java

```
class Param {
    public static native int somme(int x, int y);
    static {
        System.loadLibrary("Param");
    }
}
```

ParamEx.java

```
class ParamEx {
    public static void main(String[] args) {
System.out.println(Param.somme(3, 5));
    }
}
```

ParamImpl.java

```
#include <jni.h>
#include "Param.h"
#include <stdio.h>

JNIEXPORT jint JNICALL Java_Param_somme(JNIEnv *env, jobject obj, jint X, jint Y) {
  return X + Y;
}
```

43.3.3 Les objets

Les objets *Java* ou les tableaux sont désigné en *C* par un `jobject`. Lorsq'une instance d'une classe est passée en argument d'une méthode native, ce passage sa fait par référence.

Java met également à notre disposition les types plus fin pour minimiser les erreurs de programmation.

`jobject` tous les objets.

- `jclass` Les instances de classes.
- `jstring` Les chaines

`jarray` Les tableaux.

- `jobjectArray` Tableaux d'objets.
- `jbooleanArray` Tableaux de `boolean`.
- `jbyteArray` Tableaux de `byte`
- `jcharArray` Tableaux de `char`
- `jshortArray` Tableaux de `short`
- `jintArray` Tableaux de `int`
- `jlongArray` Tableaux de `long`
- `jfloatArray` Tableaux de `float`
- `jdoubleArray` Tableaux de `double`

`jthrowable` Les exceptions

43.3.4 Les objets String

Les objets de type `String` ont besoin d'une conversion avant leur utilisation en C ; et pour ce faire, on dipose de la méthode `GetStringUTFChars`. Inversement, la méthode `NewStringUTF` transforme une chaîne *C* en `jstring`. La méthode `ReleaseStringUTFChars` informe la machine *Java* qu'il peut récupérer la place consommer par la transformation de la chaîne.

Str.java

```
class Str {
    public static native Str conc(String s1, String s2);
    static {
        System.loadLibrary("Str");
    }
}
```

StrEx.java

```
class StrEx {
    public static void main(String[] args) {
System.out.println(Str.conc("Coucou, ", "Tout le monde"));
    }
}
```

StrImpl.java

```
#include <jni.h>
#include "Str.h"
#include <string.h>

JNIEXPORT jobject JNICALL Java_Str_conc(JNIEnv * env, jclass obj, jstring js1, jstring js2)  {
   jstring js ;
   const char *s1 = (*env)->GetStringUTFChars(env, js1, 0);
   const char *s2 = (*env)->GetStringUTFChars(env, js2, 0);
   char *s = (char *)malloc(strlen(s1)+strlen(s2));
   s = strcat(strcpy(s, s1), s2);
   (*env)->ReleaseStringUTFChars(env, js1, s1);
   (*env)->ReleaseStringUTFChars(env, js2, s2);
   js = (*env)->NewStringUTF(env, s);
   free(s);
   return js;
}
```

`NewString`

`GetStringLength`

`GetStringChars`

`ReleaseStringChars`

`NewStringUTF`

`GetStringUTFLength`

`GetStringUTFChars`

`ReleaseStringUTFChars`

43.3.5 Les tableaux

Comme les **jstring**, les tableaux *Java* ne peuvent se manipuler directement : il faut passer par une phase de conversion.

A TERMINER

- GetArrayLength
- GetIntArrayElements
- ReleaseIntArrayElements
- GetBooleanArrayElements accesses elements in a Java boolean array.
- GetByteArrayElements accesses elements in a Java byte array.
- GetCharArrayElements accesses elements in a char array.
- GetShortArrayElements accesses elements in a short array.
- GetIntArrayElements accesses elements in an int array.
- GetLongArrayElements accesses elements in a long array.
- GetFloatArrayElements accesses elements in a float array.
- GetDoubleArrayElements accesses elements in a double array.
- Get/Set<type>ArrayRegion
- GetObjectArrayElement returns the object element at a given index.
- SetObjectArrayElement updates the object element at a given index.

Tab.java

```
class Tab {
    public static native int max(int [] t);
    static {
        System.loadLibrary("Tab");
    }
}
```

TabEx.java

```
class TabEx {
    public static void main(String[] args) {
        int [] tab = {4, 7, 5, 9, 2, 0, 1};
System.out.println(Tab.max(tab));
    }
}
```

TabImpl.java

```
#include <jni.h>
#include "Tab.h"
#include <string.h>

JNIEXPORT jint JNICALL Java_Tab_max(JNIEnv * env, jclass obj, jintArray t)  {
  int i;
  jsize len = (*env)->GetArrayLength(env, t), max = -1;
  jint *body = (*env)->GetIntArrayElements(env, t, 0);
  for (max = body[0], i=1; i<len; i++)
    if (max < body[i]) max = body[i];
 (*env)->ReleaseIntArrayElements(env, arr, body, 0);
  return max;
}
```

`New`***`type`***`Array`

`Get`***`type`***`ArrayElements`

`Release`***`type`***`ArrayElements`

`Get`***`type`***`ArrayRegion`

`Set`***`type`***`ArrayRegion`

43.4 Créer et modifier des objets

A TERMINER

`NewGlobalRef`

`NewObject`

`NewObjectV`

`NewObjectA`

`NewString`

`NewStringUTF`

`NewObjectArray`

`New`*`type`*`Array`

43.5 Signature

JNI utilise la représentation e la machine virtuelle *Java* pour les signatures :

Signature	Type Java
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L*fully-qualified-class* **;**	nom absolu de la classe *fully-qualified-class*
[*type*	*type*[]
(*arg-types***)** *type-retour*	type de la méthode

Par exemple, la méthode

```
long f (int n, String s, int[] arr);
```

a pour signature

```
(ILjava/lang/String;[I)J
```

43.6 Accès aux champs

Un programme *C* peut avoir accès aux champs (**`static`** ou pas) d'une classe *Java*et ce à travers les deux méthodes **`GetStaticFieldID`** et **`GetFieldID`**. Ces fonctions retournent un **`jfieldID`**.

```
jfieldID fid1 = (*env)->GetStaticFieldID(env, cls, "ii", "I");
jfieldID fid2 = (*env)->GetFieldID(env, cls, "ss", "Ljava/lang/String;");
jfieldID fid3 = (*env)->GetFieldID(env, cls, "tab", "[I");
```

Dans cet exemple, **`fid1`**, **`fid2`** et **`fid2`** sont respectivement des pointeurs vers les champs **`ii`** de type **`int`**, **`ss`** de type **`String`** et **`tab`** de type tableau d'**`int`**.

Avec ces **`jFieldID`**, il est alors poosible d'accéder aux valeurs de ces champs et/ou de les modifier. Les fonctions d'accès aux champs sont

```
jTypePrimitif GetTypePrimitifField(jobject obj, jfieldID fid)
jobject GetObjectField(jobject obj, jfieldID fieldID)
jTypePrimitif GetStaticTypePrimitifField(jobject obj, fid fid)
jobject GetStaticObjectField(jclass clazz, jfieldID fieldID)
```

Quant à la modification des valeures de champs, on devra utiliser le sméthodes suivantes :

```
void SetTypePrimitifField(jobject obj, jfieldID fieldID, jTypePrimitif val)
void SetObjectArrayElement(jobjectArray array, jsize index, jobject val)
void SetStaticObjectField(jclass clazz, jfieldID fieldID, jobject value)
void SetStaticTypePrimitifField(jobject obj, jfieldID fieldID, jTypePrimitif val)
```

Obj.java

```
class Obj {
   static int ii = 2222;
  int iii = 2222;
   float ff = 234.34F;
   String ss = "Coucou !";
   int [] tab;
   static {
       System.loadLibrary("Obj");
   }
   public Obj(int s) {
      tab = new int[s];
      for (int i = 0; i < s ; i++) tab[i]  = i;
   }
   public String toString()  {
      String s = null;
      s += "int : " + ii + "\n";
      s += "float : " + ff + "\n";
      s += "String : " + ss + "\n";
      s += "int [] : [" ;
      for (int i = 0; i < tab.length ; i++) s += tab[i]+" ";
      s += "]\n";
      return s;
   }
   public native void afficher() ;
}
```

ObjEx.java

```
class ObjEx {
   public static void main(String[] args) {
      Obj t = new Obj(10);
      t.afficher();
      System.out.println(t);
    }
}
```

ObjImpl.java

```
#include <jni.h>
#include "Obj.h"
#include <string.h>

JNIEXPORT void JNICALL Java_Obj_afficher(JNIEnv * env, jobject obj) {
  int i;
  jint *tab, ii;
  jsize len;
  jfloat ff;
  jstring jstr;
  const char *ss;
  jobject  jtab;
  jfieldID fid;

  jclass cls = (*env)->GetObjectClass(env, obj);
  fid = (*env)->GetStaticFieldID(env, cls, "ii", "I");
  if (fid == 0) return;
  ii = (*env)->GetStaticIntField(env, cls, fid);
  printf("int :  %d \n", ii);
  (*env)->SetStaticIntField(env, cls, fid, 444);

  fid = (*env)->GetFieldID(env, cls, "ff", "F");
  if (fid == 0) return;
  ff = (*env)->GetFloatField(env, obj, fid);
  printf("float :  %f \n", ff);
  (*env)->SetFloatField(env, obj, fid, 0.5f);

  fid = (*env)->GetFieldID(env, cls, "ss", "Ljava/lang/String;");
  if (fid == 0) return;
  jstr = (*env)->GetObjectField(env, obj, fid);
  ss = (*env)->GetStringUTFChars(env, jstr, 0);
  printf("String :  %s \n", ss);
  (*env)->ReleaseStringUTFChars(env, jstr, ss);
  jstr = (*env)->NewStringUTF(env, "Bonjour !");
```

```
  (*env)->SetObjectField(env, obj, fid, jstr);


  fid = (*env)->GetFieldID(env, cls, "tab", "[I");
  if (fid == 0) return;
  jtab = (*env)->GetObjectField(env, obj, fid);
  tab = (*env)->GetIntArrayElements(env, jtab, 0);
  len = (*env)->GetArrayLength(env, jtab);
  printf("int [] : [");
  for (i = 0; i < len ; i++) printf("%d  ", tab[i]);
  printf("]\n");
}
```

43.7 Invocation de méthodes Java

Cette section présente la manière d'invoquer une méthode *Java* à partir d'une fonction *C*.

43.7.1 Méthode d'instance

Pour l'invocation d'une méthode d'instance (non statique), il faut

- Récupérer la classe à laquelle on s'intéresse avec la fonction `GetObjectClass`

  ```
  jclass GetObjectClass(JNIEnv *env, jobject obj);
  ```

- Récupérer la méthode que l'on veut invoquer avec la fonction `GetMethodID`. Cette fonction retourne la valeur 0 et lance l'exception `NoSuchMethodError` si la méthode spécifiée n'existe pas.

  ```
  jmethodID GetMethodID(JNIEnv *env, jclass clazz,  const char *name, const char *sig);
  ```

  `clazz` désigne la classe, `name` le nom de la méthode, `sig` la signature de la méthode sous la forme d'une chaîne de caractères selon les conventions exposées en 43.5.
- invoquer la méthode avec la fonction `call`*`XX`*`Method`.

  ```
  NativeType Call<type>Method(JNIEnv *env, jobject obj, jmethodID methodID, ...);
  ```

CtoJava.java

```
class CtoJava {
   static int ii = 2222;
  int iii = 2222;
   float ff = 234.34F;
   String ss = "Coucou !";
   int [] tab;
   static {
       System.loadLibrary("CtoJava");
   }
   public CtoJava(int s) {
      tab = new int[s];
      for (int i = 0; i < s ; i++) tab[i]  = i;
   }
   public String toString()  {
      String s = new String();
      s += "int : " + ii + "\n";
      s += "float : " + ff + "\n";
      s += "String : " + ss + "\n";
      s += "int [] : [" ;
      for (int i = 0; i < tab.length ; i++) s += tab[i]+" ";
      s += "]\n";
      return s;
   }
   public native void jafficher() ;
}
```

CtoJavaEx.java

```
class CtoJavaEx {
   public static void main(String[] args) {
      CtoJava t = new CtoJava(10);
      t.jafficher();
      System.out.println(t);
    }
}
```

CtoJavaImpl.java

```
#include <jni.h>
#include "CtoJava.h"
#include <string.h>

JNIEXPORT void JNICALL Java_CtoJava_jafficher(JNIEnv * env, jobject obj) {
  const char *str;
  jclass cls = (*env)->GetObjectClass(env, obj);
  jstring s;
  jmethodID mid = (*env)->GetMethodID(env, cls, "toString", "()Ljava/lang/String;");
  if (mid == 0) return;
  printf("------------------------------------\n");
  s = (*env)->CallObjectMethod(env, obj, mid);
  str = (*env)->GetStringUTFChars(env, s, 0);
  printf("%s", str);
  (*env)->ReleaseStringUTFChars(env, s, str);
  printf("------------------------------------\n");
}
```

43.7.2 Méthode de classe

Pour l'invocation d'une méthode de classe (statique), il faut

- Récupérer la classe à laquelle on s'intéresse avec la fonction `GetObjectClass`
- Récupérer la méthode que l'on veut invoquer avec la fonction `GetStaticMethodID`. Cette fonction retourne la valeur 0 et lance l'exception `NoSuchMethodError` si la méthode spécifiée n'existe pas.

  ```
  jmethodID GetStaticMethodID(JNIEnv *env, jclass clazz,  const char *name, const char *sig);
  ```

  `clazz` désigne la classe, `name` le nom de la méthode, `sig` la signature de la méthode sous la forme d'une chaîne de caractères selon les conventions exposées en 43.5.
- invoquer la méthode avec la fonction `callStatic`*XX*`Method`.

  ```
  NativeType CallStatic<type>Method(JNIEnv *env, jobject obj, jmethodID methodID, ...);
  ```

43.8 Exceptions et code natif

Des méthodes *Java* sont suceptibles de générer des exceptions ; il en est de même de certaines fonctions de l'interface `JNI`. Par exemple, la méthode `getFieldID` peut lever l'exception `NoSuchFieldError`. La pluspart des fonction *JNI* à la fois retourne un code d'erreur et lancent un exception.

Asynchronous Exceptions

43.9 Capturer une exception

Lorsqu'une exception est lancée, le code natif peut choisir

- de terminer l'exécution ce qui a pour effet de renvoyer l'exception au code *Java* appellant
- de prendre en charge la gestion de l'exception. Lorsqu'une exception est levée, les seules fonctions *JNI* qui peuvent être appelées sont celles concernant la gestion des exception : `ExceptionOccurred`, `ExceptionDescribe` et `ExceptionClear`. Aussi, la première opération à effectuer, lorsque le code natif décide de gérer l'exception, est d'annuler l'exception de manière à pouvoir utiliser toutes les fonctions `JNI` nécessaires pour la suite du traitement.

La structure de donnée `jthrowable` code une exception en *C*. La méthode`ExceptionClear` annule l'exception et laméthode `ExceptionOccurred` retourne

```
jthrowable exception;
... // méthode ou fonction JNI engendrant une exception
exception = (*env)->ExceptionOccurred(env);
if (exception) {
   (*env)->ExceptionDescribe(env);
   (*env)->ExceptionClear(env);
```

43.10 Lancer une exception

Une fonction native peut lever une exception avec les fonctions `ThrowNew` ou `Throw` dont l'un des arguments est une structure de type `jclass`.

```
jint Throw(JNIEnv *env, jthrowable obj);
jint ThrowNew(JNIEnv *env, jclass clazz, const char *message);
```

```
    jclass newExceptionCls = (*env)->FindClass(env, "java/lang/IllegalArgumentException");
    if (newExceptionCls == 0)  return;
    (*env)->ThrowNew(env, newExceptionCls, "thrown from C code");
}
```

43.11 Code natif et threads

A TERMINER

43.12 Code natif et C++

A TERMINER

43.13 Démarrer la machine viruelle Java

```
public class Prog {
   public static void main(String[] args) {
      System.out.println("Hello World" + args[0]);
   }
}
```

```
#include <jni.h>

main() {
    JNIEnv *env;
    JavaVM *jvm;
    JDK1_1InitArgs vm_args;
    jint res;
    jclass cls;
    jmethodID mid;
    jstring jstr;
    jobjectArray args;


    /* IMPORTANT: specify vm_args version # if you use JDK1.1.2 and beyond */
    vm_args.version = 0x00010001;

    JNI_GetDefaultJavaVMInitArgs(&vm_args);
    vm_args.classpath = "/disk2/linux/tourai/Java/Perso/JavaPoly/Programmes/jni:\
/usr/lib/jdk-1.1.5/classes.zip";

    res = JNI_CreateJavaVM(&jvm,&env,&vm_args);
    if (res < 0) {
        fprintf(stderr, "Can't create Java VM\n");
        exit(1);
    }

    cls = (*env)->FindClass(env, "Prog");
    if (cls == 0) {
        fprintf(stderr, "Can't find Prog class\n");
        exit(1);
    }

    mid = (*env)->GetStaticMethodID(env, cls, "main", "([Ljava/lang/String;)V");
    if (mid == 0) {
        fprintf(stderr, "Can't find Prog.main\n");
        exit(1);
    }

    jstr = (*env)->NewStringUTF(env, " from C!");
    if (jstr == 0) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    args = (*env)->NewObjectArray(env, 1,
                        (*env)->FindClass(env, "java/lang/String"), jstr);
    if (args == 0) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    (*env)->CallStaticVoidMethod(env, cls, mid, args);

    (*jvm)->DestroyJavaVM(jvm);
```

```
}

unix:     cc -I<where jni.h is> -L<where libjava.so is> -ljava invoke.c
windows:  cl -I<where jni.h is> -MT invoke.c -link <where javai.lib is>\lb
```

43.14 La commande javap

A TERMINER

43.15 Récapitulatif des fonctions JNI

43.15.1 Version

```
jint GetVersion(JNIEnv *env);
```

43.15.2 Opération sur les classes

```
jclass DefineClass(JNIEnv *env, jobject loader, const jbyte *buf, jsize bufLen);
jclass FindClass(JNIEnv *env, const char *name);
jclass GetSuperclass(JNIEnv *env, jclass clazz);
jboolean IsAssignableFrom(JNIEnv *env, jclass clazz1,  jclass clazz2);
```

43.15.3 Exceptions

```
jint Throw(JNIEnv *env, jthrowable obj);
jint ThrowNew(JNIEnv *env, jclass clazz, const char *message);
jthrowable ExceptionOccurred(JNIEnv *env);
void ExceptionDescribe(JNIEnv *env);
void ExceptionClear(JNIEnv *env);
void FatalError(JNIEnv *env, const char *msg);
```

43.15.4 Référencs globales et locales

```
jobject NewGlobalRef(JNIEnv *env, jobject obj);
void DeleteGlobalRef(JNIEnv *env, jobject globalRef);
void DeleteLocalRef(JNIEnv *env, jobject localRef);
```

43.15.5 Opérations sur les objets

```
jobject AllocObject(JNIEnv *env, jclass clazz);
jobject NewObject(JNIEnv *env, jclass clazz, jmethodID methodID, ...);
jobject NewObjectA(JNIEnv *env, jclass clazz, jmethodID methodID, jvalue *args);
jobject NewObjectV(JNIEnv *env, jclass clazz, jmethodID methodID, va_list args);
jclass GetObjectClass(JNIEnv *env, jobject obj);
jboolean IsInstanceOf(JNIEnv *env, jobject obj,  jclass clazz);
jboolean IsSameObject(JNIEnv *env, jobject ref1,  jobject ref2);
```

43.15.6 Accès aux variables d'instance

```
jfieldID GetFieldID(JNIEnv *env, jclass clazz,  const char *name, const char *sig);
NativeType Get<type>Field(JNIEnv *env, jobject obj, jfieldID fieldID);
void Set<type>Field(JNIEnv *env, jobject obj, jfieldID fieldID,
NativeType value);
```

43.15.7 Invocation de méthodes d'instance

```
jmethodID GetMethodID(JNIEnv *env, jclass clazz,  const char *name, const char *sig);
NativeType Call<type>Method(JNIEnv *env, jobject obj, jmethodID methodID, ...);
NativeType Call<type>MethodA(JNIEnv *env, jobject obj, jmethodID methodID, jvalue *args);
NativeType Call<type>MethodV(JNIEnv *env, jobject obj, jmethodID methodID, va_list args);
NativeType CallNonvirtual<type>Method(JNIEnv *env, jobject obj, jclass clazz, jmethodID methodID, ...);
NativeType CallNonvirtual<type>MethodA(JNIEnv *env, jobject obj, jclass clazz, jmethodID methodID, jvalue *args);
NativeType CallNonvirtual<type>MethodV(JNIEnv *env, jobject obj, jclass clazz, jmethodID methodID, va_list args);
```

43.15.8 Accès aux variables de classe

```
jfieldID GetStaticFieldID(JNIEnv *env, jclass clazz, const char *name, const char *sig);
NativeType GetStatic<type>Field(JNIEnv *env, jclass clazz, jfieldID fieldID);
void SetStatic<type>Field(JNIEnv *env, jclass clazz, jfieldID fieldID, NativeType value);
```

43.15.9 Invocation de méthodes de classe

```
jmethodID GetStaticMethodID(JNIEnv *env, jclass clazz, const char *name, const char *sig);
NativeType CallStatic<type>Method(JNIEnv *env, jclass clazz, jmethodID methodID, ...);
NativeType CallStatic<type>MethodA(JNIEnv *env, jclass clazz, jmethodID methodID, jvalue *args);
NativeType CallStatic<type>MethodV(JNIEnv *env, jclass clazz, jmethodID methodID, va_list args);
```

43.15.10 Chaînes de caractères

```
jstring NewString(JNIEnv *env, const jchar *unicodeChars, jsize len);
jsize GetStringLength(JNIEnv *env, jstring string);
const jchar * GetStringChars(JNIEnv *env, jstring string, jboolean *isCopy);
void ReleaseStringChars(JNIEnv *env, jstring string, const jchar *chars);
jstring NewStringUTF(JNIEnv *env, const char *bytes);
jsize GetStringUTFLength(JNIEnv *env, jstring string);
const char* GetStringUTFChars(JNIEnv *env, jstring string,  jboolean *isCopy);
void ReleaseStringUTFChars(JNIEnv *env, jstring string, const char *utf);
```

43.15.11 Tableaux

```
jsize GetArrayLength(JNIEnv *env, jarray array);
jarray NewObjectArray(JNIEnv *env, jsize length,  jclass elementClass, jobject initialElement);
jobject GetObjectArrayElement(JNIEnv *env,  jobjectArray array, jsize index);
void SetObjectArrayElement(JNIEnv *env, jobjectArray array,  jsize index, jobject value);
ArrayType New<PrimitiveType>Array(JNIEnv *env, jsize length);
NativeType *Get<PrimitiveType>ArrayElements(JNIEnv *env,  ArrayType array, jboolean *isCopy);
void Release<PrimitiveType>ArrayElements(JNIEnv *env, ArrayType array, NativeType *elems, jint mode);
void Get<PrimitiveType>ArrayRegion(JNIEnv *env, ArrayType array,jsize start, jsize len, NativeType *buf);
void Set<PrimitiveType>ArrayRegion(JNIEnv *env, ArrayType array, jsize start, jsize len, NativeType *buf);
```

43.15.12 Registering Native Methods

```
jint RegisterNatives(JNIEnv *env, jclass clazz,
const JNINativeMethod *methods, jint nMethods);
jint UnregisterNatives(JNIEnv *env, jclass clazz);
```

43.15.13 Monitor Operations

```
jint MonitorEnter(JNIEnv *env, jobject obj);
jint MonitorExit(JNIEnv *env, jobject obj);
```

43.15.14 Java VM Interface

```
jint GetJavaVM(JNIEnv *env, JavaVM **vm);
```

44. Corba

Sommaire

Cinquième partie

Java : Beans

Table des Matières

45 Introduction à Java Beans 385

45. Introduction à Java Beans

Sommaire

Sixième partie

Java : Annexes

Table des Matières

46 javadoc **391**
46.1 Les balises HTML 391
46.2 Résumé et description détaillée 391
46.3 Les balises javadoc 392
46.4 La balise @see 392
46.5 Résumé 393
46.6 La commande javadoc 394
47 Le langage SQL **395**
48 Fromat jar et zip **397**

46. javadoc

Sommaire

46.1 Les balises HTML 391
46.2 Résumé et description détaillée 391
46.3 Les balises javadoc 392
46.4 La balise @see 392
46.4.1 La balise @author 392
46.4.2 La balise @version 392
46.4.3 La balise @param 392
46.4.4 La balise @return 392
46.4.5 Le balise @exception 392
46.4.6 La balise @since 393
46.4.7 La balise @deprecated 393
46.5 Résumé 393
46.5.1 Documentation des classes et interfaces 393
46.5.2 Documentation des méthodes et constructeurs 393
46.5.3 Documentation des champs 393
46.6 La commande javadoc 394

Comme nous l'avons déjà dit (voir 3.2.3) , *Java* permet d'inclure dans un programme sous forme de commentaires toute la documentation que l'on veut voir donner sur les classes et interfaces que l'on programme. L'utilitaire `javadoc` extrait du code *Java* ces commentaires pour les transformer en fichier *HTML* à l'image de la document des *API* de *SUN*.

Ces commentaires de documentation peuvent contenir des balises HTML et des balises javadoc facilitant la présentation de la documentation.

La documentation est constitué des caractères compris entre "/**" et "*/". De plus, si les premiers caractères non blancs (*espace* et *tabulation*) sont des "*", alors ils sont ignorés. Tous les caractères blanc précédants le caractère "*" sont également ignorés. Ces commentaires peuvent apparaître avant chaque définition de *classes*, d'*interface* ainsi qu'avant les *méthodes*, les *constructeurs* ou *champs*.

46.1 Les balises HTML

Ces commentaires peuvent contenir des balises HTML. Il convient toutefois d'éviter les balises : `<H1>`, `<H2>`, `<H3>`, `<H4>`, `<H5>`, `<H6>`, et `<HR>`. Ces balises peuvent entrer en conflit avec celles générées par l'outil `javadoc`.

```
/**
 * Ceci est un commentaire de <b>documentation</b>.
 */
```

46.2 Résumé et description détaillée

La première phrase figurant dans ce type de commentaire servira de résumé de la documentation de l'entité commenté. Il convient donc de donner une description succincte et complète de l'entité. La fin de la première phrase est déterminée par le caractère "." suivi d'un *espace*, d'une *tabulation* ou d'une *retour chariot*.

Attention aux phrases du genre : `This is a simulation of Prof. Knuth's MIX computer.`

Les phrases qui suivent et qui sont placées avant toute balise `javadoc` serviront de texte pour la description détaillée.

46.3 Les balises javadoc

Les commentaires de documentation ne analysés par l'utilitaire javadoc. Celui-ci reconnaît un certain nombre de balises qui lui sont propres. Une ligne de commentaire commençant par un des mots clés suivant : `@see`, `@author`, `@version`, `@param`, `@return` et `@exception`. Ces balises sont utiles pour mettre en forme la documentation et pour générer les références croisées.

46.4 La balise @see

La balise `@see` permet de définir des références vers d'autres *classes*, *interfaces*, *méthodes*, *constructeurs*, *champs* ou *URL*.

```
@see java.lang.String
@see String
@see java.io.InputStream;
@see String#equals
@see java.lang.Object#wait(int)
@see java.io.RandomAccessFile#RandomAccessFile(File, String)
@see Character#MAX_RADIX
@see <a href="spec.html">Java Spec</a>
```

Le caractère "`#`" est un séparateur entre le nom de classe et le nom d'un champ, d'une méthode ou d'un constructeur. Il peut y avoir autant de balises `@see` que nécessaires.

46.4.1 La balise @author

Les balises `@author` permettent de spécifier le ou les auteurs. Elles peuvent figurer dans les documentations de classes et des interfaces.

```
@author Mary Wollstonecraft
@author Hildegard von Bingen
@author Dorothy Parker
```

Il peut y avoir autant de balises `@author` que nécessaire. On peut également spécifier plusieurs auteurs par balise :

```
@author Mary Wollstonecraft, Hildegard von Bingen
        Dorothy Parker
```

Toutefois, les concepteurs du langage recommande d'utiliser une balise par auteur de manière permettre une mise en forme plus agréable.

46.4.2 La balise @version

Les balises `@version` peuvent figurer dans les documentations de classes et des interfaces.

```
@version 493.0.1beta
```

Il ne peut y avoir qu'au plus une balise `@version` par paragraphe.

46.4.3 La balise @param

Les balises `@param` permettent de commenter les paramètres des méthodes et des constructeurs. Elles ne peuvent figurer que dans les documentations de méthodes et des constructeurs.

```
@param fic le fichier dans lequel il faut effectuer la recherche
@param motif
        le motif à rechercher
@param nbre     le nombre de lignes à afficher à chaque correspondance
```

La structure d'un paragraphe de type `@param` est constitué du nom du paramètre suivi par une courte description.

La convention adoptée par *Java* veut que l'on choisisse soit de ne commenter aucun paramètre, soit de commenter tous les paramètres dans l'ordre où ils apparaissent.

46.4.4 La balise @return

Les balises `@return` permettent de documenter les valeurs retournées par les méthodes. Elles peuvent figurer dans les documentations des méthodes dont le type de retour est autre que `void`.

```
@return Le nombre de lignes contenant le motif recherché
```

Par convention, la balise `@return` est suivi d'une courte description de la valeur retournée. Il ne peut y avoir qu'au plus une balise `@return`.

46.4.5 Le balise @exception

Les balises `@exception` permettent de commenter les exceptions lancées par une méthode ou un constructeur. Elles ne peuvent donc figurer que dans les documentations de méthodes et des constructeurs.

```
@exception IndexOutOfBoundsException Débordement de la matrice
@exception java.io.FileNotFoundException le fichier
                                         n'existe pas
```

46.4.6 La balise @since

46.4.7 La balise @deprecated

46.5 Résumé

46.5.1 Documentation des classes et interfaces

- **@author** *nom de l'auteur*
- **@version** *identifiant de la version*
- **@see** *non de classe*
- **@since** *texte*
- **@deprecated** *texte "deprecated"*

```
/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 *    Window win = new Window(parent);
 *    win.show();
 * </pre>
 *
 * @author  Sami Shaio
 * @version %I%, %G%
 * @see     java.awt.BaseWindow
 * @see     java.awt.Button
 */
class Window extends BaseWindow {
    ...
}
```

46.5.2 Documentation des méthodes et constructeurs

@param *description des paramètres*

@return *description du retour de la méthode*

@exception *nom-complet-de-l'exception description*

@see *nom de la classe*

@since *texte*

@deprecated *texte "deprecated"*

```
/**
 * Returns the character at the specified index. An index
 * ranges from <code>0</code> to <code>length() - 1</code>.
 *
 * @param     index  the index of the desired character.
 * @return    the desired character.
 * @exception StringIndexOutOfRangeException
 *              if the index is not in the range <code>0</code>
 *              to <code>length()-1</code>.
 * @see       java.lang.Character#charValue()
 */
public char charAt(int index) {
    ...
}
```

46.5.3 Documentation des champs

A field comment can contain only the @see, @since and @deprecated tags (as described above).

```
/**
 * The X-coordinate of the window.
 *
 * @see window#1
 */
int x = 1263732;
```

46.6 La commande javadoc

La forme générale de la commande **javadoc** est de la forme

```
javadoc [ options ] [ package | source.java ]*
```

où les options sont les suivantes :

-classpath *chemin*

-public

-protected

-package

-private

-J *flags*

-encoding *nom*

-docencoding *nom*

-version

-author

-noindex

-notree

-d *répertoire*

-verbose

-sourcepath *chemin*

-nodeprecated

47. Le langage SQL

Sommaire

48. Fromat jar et zip

Sommaire

Septième partie

Java : Travaux dirigés

Fiche 1. Eleménts de base

Exercise 1.1

Déterminer l'ordre d'évaluation des expressions :

- `x + 2 * 5`
- `x - 2 - 3`
- `y = ++x + x++`
- `y = ++x + ++x`
- `x + 2 * 5` : `x + (2 * 5)`
- `x - 2 - 3` : `(x - 2) - 3`
- `y = ++x + x++` : `y = (++x) + (x++)` i.e. $y = 2 * (x+1) et x = x+2$
- `y = ++x + ++x` : `y = (++x) + (++x)` i.e. $y = 2 * (x+2) et x = x+2$

Exercise 1.2

Ecrire un programme qui affiche la somme et le produit de deux entiers données en argument de la ligne de commande.

Exercise 1.3

Ecrire un programme affiche l'inverse de l'entier donnée argument de la ligne de commande.

Exercise 1.4

Ecrire un programme qui converti les francs et écus, sachant qu'un écu vaut $7,22$ francs.

Exercise 1.5

Ecrire un programme qui affiche la décomposition en base 10 d'un entier. Par exemple, le nombre 123 sera affiché sous la forme $3 + 2 \times 10 + 1 \times 10^2$.

Exercise 1.6

Ecrire un programme qui calcule la n^{ime} valeur de la suite de Fibonacci qui définie par

$u_0 = 1$
$u_1 = 1$
$u_n = u_{n-1} + u_{n-2}$, pour n>2

Exercise 1.7

Ecrire un programme qui imprime le nombre d'indice i tels que $A_i = B_j$ où A et B sont deux tableaux d'entiers de longueur N.

Exercise 1.8

Ecrire un programme qui imprime le plus grand élément d'un tableau A de longueur N.

Exercise 1.9

Recherche Séquentielle.
Ecrire un programme qui recherche un entier x dans un tableau d'entiers A de longueur N et qui imprime le premier indice i tel que $A_i = x$.

Exercise 1.10

Tasser un tableau d'entiers A, tableau de taille N, en supprimant tous les zéros qui y figurent.

Exercise 1.11

Insertion Séquentielle.
Soit A un tableau ordonné d'entiers ; tableau de taille NA. Insérer un enter x (s'il n'existe pas déjà) dans ce tableau tout en conservant la relation d'ordre sur les éléments de ce tableau.

Exercise 1.12

Suppression Séquentielle.
Soit A un tableau ordonné d'entiers ; tableau de taille NA. Supprimer (s'il existe) l'entier x de ce tableau.

Exercise 1.13

Soient A et B deux tableaux d'entiers ordonné ; tableaux de taille NA et NB. Créer un troisième tableau ordonné C constitué de la fusion des deux tableaux précédents.

Exercise 1.14

Ecrire un programme qui lit un entier et qui affiche ce même nombre dans sa notation binaire, octale, décimale. Par rapport à l'exercice **??**, l'ordre dans lequel les chiffres doivent être affichés est inversé ; il faut donc utiliser un tableau pour stocker les valeurs et les afficher à la fin du calcul.

Exercise 1.15

Schéma de Horner.
Un polynôme $P(x) = a_0 X^n \dots a_{n_1} X + a_n$ est représenté par la suite de valeurs $(a_0, \dots, a_{n_1}, a_n)$ de ses coefficients. Ecrire un programme q ui calcule la valeur de $P(X)$ pour une valeur donnée de X en utilisant le schéma d'Horner suivant :

$$P(X) = (\dots((a_0 \times X + a_1) \times X) + a_2 \times X + a_{n-1}) \times X + a_n.$$

Estimer le nombre de multiplication effectué par ce programme et comparer avec des programmes plus simples.

Exercise 1.16

Recherche dicothomique.
Ecrire un programme qui recherche un entier x dans un tableau d'entiers A de longueur N et qui imprime tous les indices i tel que $A_i = x$.

Exercise 1.17

Calculer la somme et le produit de deux matrices carrés de taille N.

Exercise 1.18

Etant donné une matrice A de taille $N \times M$, on dit qsu'un couple d'indice (p, q) est un min-max de cette matrice si la valeur $a_{p,q}$ est un minimum de la ligne p et un maximum de la colonne q, i.e. :

$$a_{p,q} = min\{A_{p,1}, \dots, a_{p,M}\} = max\{A_{1,q}, \dots, a_{N,q}\}.$$

Ecrire un programme qui affiche l'ensemble de tels couples (p, q). La méthode pro posée consiste à effectuer le travail suivant pour chaque ligne p :

- trouver les minima de la ligne p et en mémoriser les numéros de colonne,
- pour chacun de ces rangs q, déterminer si $a_{p,q}$ est un maximum pour sa colonne.

Exercise 1.19

Ecrire un programme qui détermine tous les nombres premier compris en 1 et n en utilisant le « crible d'Erastothène ». L'algorithme proposé consiste à disposer d'un tableau des n valeurs consécutifs et on commence par cocher l'entier 1 du tableau. Puis on réitère le procédé suivant tant que le nombre premier que nous allons trouver est inférieur à la racine carrée de n :

> A partir du dernier nombre premier trouvé (au démarrage on supposera que c'est 1), chercher un entier non coché.
> On démontre que ce entier est bien premier. Cocher alors tous les multiples de ce nombre premier.

Exercise 1.20

Programmer le tri à bulles

Fiche 2. Classes et objets

Exercise 2.1

Compléter la classe `Date` en implantant toutes les méthodes de cette classe et donner un exemple d'utilisation.

```
class Date {
    private int jour, mois, annee;
    public void affecter(int j, int m, int a);
    public int quelJour();
    public int quelMois();
    public int quelleAnnee();
    public void lendemain() ;
    public void imprimer();
    public static void listerDates();
}
```

Exercise 2.2

Réaliser une classe `Point` permettant de manipuler un point du plan. On prévoira

- un constructeur
- les méthodes pour accéder aux coordonnés et pour les modifier,
- une méthode `deplacer` qui effectue une translation définie par les arguments.
- une méthode `afficher` qui affiche les coordonnées du point.

Donner un exemple d'utilisation.

Exercise 2.3

Définir une classe `PileEnt` d'entiers munis de son ou ses constructeurs, les méthodes `empile` et `depiler`.

Exercise 2.4

Définir une classe `Individu` composée d'un nom, d'une adresse et d'un numéro de téléphone. Donner le constructeur, les méthodes d'affectation et de consultation des noms, adresse et numéro de téléphone.

Exercise 2.5

Modifier le classe `Individu` pour pouvoir afficher le nombre d'objets créés.

Exercise 2.6

Modifier le classe `Individu` de manière à

1. gérer une liste des individus créés
2. afficher la liste des individus créés

Exercise 2.7

Définir une classe vecteur et une classe matrice carrée. Définir une fonction `multiplier` qui calcule le produit d'une matrice par un vecteur.

Exercise 2.8

Modifier le programme précédant pour que la fonction `multiplier` utilisent les champs privés des classes `Matrice` et `Vecteur`.

Exercise 2.9

Compléter les classes `Vecteur` et `Matrice` en surchargeant le méthode equals de la classe `Object` et de manière à pouvoir imprimer de la manière suivante :

```
Matrice m;
Vecteur v;
...
System.out.println(v + m);
```

Exercise 2.10

Créer une classe liste permettant de manipuler des listes chaînées dans laquelle la nature de l'information associé à chaque noeud n'est pas connu.

Exercise 2.11

Java fournit la classe java.util.Vector. En consultant la documentation de la cette classe, programmer cette dernière.

Fiche 3. Héritage, interfaces et packages

Exercise 3.1

En se basant sur la classe `Point`, créer une classe `PointA` comportant une méthode supplémentaire `distance` qui détermine la distance d'un point au centre.

- Les coordonnés sont des champs privés de la classe `Point`.
- Les coordonnés sont des champs privés de la classe `Protected`.

Exercise 3.2

Définir une classe `PointColore` destiné à contenir la couleur d'un point. On dotera cette classe d'une nouvelle fonction `imprimer` qui affiche également la couleur du point.

Exercise 3.3

Même question en n'utilisant pas une classe dérivée.

Exercise 3.4

Compléter la classe `Forme` suivante et définir les sous classes `Rectangle`, `Ovale` et `LigneBrisée`.

```
abstract class Forme {
  Point origine;
  protected Forme suiv;
  private static Forme listeDesFormes = null;
  public Forme(int x, int y) {  }
  public Forme(int x, int y, int fond, int contour) { ... }
  public static void afficherToutesLesFormes() { ... }
  public void supprimerToutesLesFormes() { ... }
  public void supprimer() { ... }
  abstract void deplacer(int x, int y);
  abstract void modifier();
  abstract void afficher();
}

final class Rectangle extends Forme  {  ... }
final class Oval extends Forme  { ... }
final class LigneBrisee extends Forme  { ... }

class Dessin {
  public static void main(String args[]) {
    Rectangle r1 = new Rectangle(0,0, 10, 10, 1, 1);
    Rectangle r2 = new Rectangle(5,5, 20, 10, 0, 2);
    Rectangle r3 = new Rectangle(10, 13, 25,100, 2, 2);
    Oval o1 = new Oval(1, 3, 5,10, 2, 2);
    int [] x =  {1, 3, 5}, y =  {5, 10, 3};
    LigneBrisee l1 = new LigneBrisee(x, y,  2);
    Forme.afficherToutesLesFormes();
    r2.supprimer();
    Forme.afficherToutesLesFormes();
  }
}
```

Exercise 3.5

Même question en utilisant une interface `Dessinable` au lieu des méthodes abstraites.

Exercise 3.6

Java fournit la classe `java.util.Stack` sous classe de la classe `java.util.Vector`. En consultant la documentation de la cette classe, programmer la classe `java.util.stack`.

Fiche 4. Chaînes de caractères

Exercise 4.1

Chercher le nombre d'occurrences d'une sous chaîne dans une chaîne.

Exercise 4.2

Ecrire un programme qui recopie en l'inversant une chaîne de caractères passé en argument de la ligne de commande.

Première solution

Autre solution

Exercise 4.3

Ecrire un programme qui inverse sans recopier une chaîne de caractères lue au clavier.

Exercise 4.4

Ecrire une fonction qui transforme recopie en transformant les lettres majuscules en lettres minuscules une chaîne de caractères passé en argument de la ligne de commande.

Fiche 5. Entrées-Sorties

Exercise 5.1

Programmer une course entre un lièvre et une tortue en faites en sorte que le lièvre gagne.

Fiche 6. Diverses petites choses

Exercise 6.1

Etant donnée la grammaire *BNF* suivante, ecrire un programme permettant de reconnaître si une chaîne de carctères suivie d'un point, lue à la console, est u n mot du langage engendré par cette grammaire.

expression	⟶	*terme operateur-additif express ion* | *terme*
terme	⟶	*facteur operateur-multiplicatif term e* | *facteur*
facteur	⟶	*nombre* | *variable* | '(' *expression* ')'
variable	⟶	'x' | 'y' | 'z'
operateur-additif	⟶	'+' | '-'
operateur-multiplicatif	⟶	'*' | '-'

Modifier l'analyseur précédent de telle sorte que le programme construise l'arbre syntaxique de l'expression analysée.

Ecrire une fonction qui imprime l'expression analysée à partir de l'arbre syntaxiq ue. Attention : pour imprimer correctement l'expression, avec les bons parenthésage s, il faudra utiliser la grammaire.

Exercise 6.2

Mêmes questions pour la grammaire :

arbre	⟶	*atome* '(' *liste-d-arbres* ') ' | *atome*
liste-d-arbres	⟶	*atome* ',' *liste-d-a rbres* | *arbre*
atome	⟶	'a' | 'b' | $\cdots$ |'z'

Exercise 6.3

Programmer le jeu du solitaire.

Exercise 6.4

On dispose de trois de piquets, numérotés 1, 2, et 3 et de n disques toutes de t alles différentes. Au départ, les disuaes sont empilés par taille décroissante sur le piquet numéro 1 . Le problème consiste à dépalcer tous les disques du piquet numéro 1 au piquet numé ro 3 el les empilant par taille décroissante, et ce en respectant les règles de dé placement suivantes :

- On ne peut déplacer qu'un disque à la fois
- un disque ne doit jamais être placé sur un disque de talle plus petite.

Fiche 7. Threads

Exercise 7.1

Programmer une course entre un lièvre et une tortue en faites en sorte que le lièvre gagne.

Exercise 7.2

Définir une classe Producteur qui empile, à intervalles réguliers, des messages et une classe Consommateur qui récupère dans la pile un message à intervalles réguliers. Le producteur devra interrompre la production si la pile est pleine et le consommateur devra patienter si le pile est vide. Donner une exemple d'utilisation avec un producteur et 3 consommateurs.

Exercise 7.3

Définir une classe Producteur qui empile, à intervalles réguliers, des messages et une classe Consommateur qui récupère dans la pile un message à intervalles réguliers. Le producteur devra interrompre la production si la pile est pleine et le consommateur devra patienter si le pile est vide. Donner une exemple d'utilisation avec un producteur et 3 consommateurs.

Exercise 7.4

Programmer une applet qui affiche une chaine de caractères en lui faisant changer la couleur régulièrement.

Exercise 7.5

Réaliser une animation graphique dans une applet à partir de la suite des 15 images que vous trouverez à `/home/users/public/Java/tumble`.
-

Fiche 8. Applet et Programmation graphique

Exercise 8.1

Réaliser une applet contenant trois boutons et associer une page HTML diiférente à chaque bouton de manière à pouvoir afficher la bonne page en fonction du blouton cliqué.

Exercise 8.2

Réaliser une applet qui affiche une image qui suit les mouvements de la souris et qui prend sa place définitive lorsqu'on la relâche.

Exercise 8.3

Réaliser une applet qui affiche un nouveau carré de couleur aléatoire chaque fois que l'on clique sur la souris. Si le clic survient sur un carré déjà crée, alors ce dernier suivra le déplacement de la souris tant qu'elle est appuyée.

Version avec une classe imbriquée

Exercise 8.4

Réaliser une applet qui affiche un clavier à l'écran et indique (sur une image de clavier) les touches enfoncées par l'utilisateur.

Exercise 8.5

Réaliser un petit éditeur de texte avec les fonctionnalités suivantes : *copier*, coller, effacer, insérer un élement d'un glossaire.

Exercise 8.6

Reprendre l'exemple **??** en utilisant ? ? ? pour stocker le dessin réalisé par l'utilisateur, une image.

Exercise 8.7

Réaliser une animation qui déplace dans une nuit étoilée un appareil volant.

Fiche 9. Programmation réseau

Exercise 9.1

Programmer une course entre un lièvre et une tortue en faites en sorte que le lièvre gagne.

Huitième partie

Java : Corrigés

Fiche 1. Eleménts de base

Solution 1.1

Déterminer l'ordre d'évaluation des expressions :

- `x + 2 * 5`
- `x - 2 - 3`
- `y = ++x + x++`
- `y = ++x + ++x`

- `x + 2 * 5` : `x + (2 * 5)`
- `x - 2 - 3` : `(x - 2) - 3`
- `y = ++x + x++` : `y = (++x) + (x++)` i.e. $y = 2 * (x + 1) etx = x + 2$
- `y = ++x + ++x` : `y = (++x) + (++x)` i.e. $y = 2 * (x + 2) etx = x + 2$

Solution 1.2

Ecrire un programme qui affiche la somme et le produit de deux entiers données en argument de la ligne de commande.

```
class Td2 {
  public static void main(String [] args) {
    try {
      int x = Integer.parseInt(args[0]);
      int y = Integer.parseInt(args[1]);
      System.out.println("La somme de " + x + " est de " + y + " est " + (x+y));
      System.out.println("Le produit de " + x + " est de " + y + " est " + (x*y));
    }
    catch (java.lang.ArrayIndexOutOfBoundsException e) {
      System.out.println("usage \"java Td1 <eniter> <entier>\"");
    }
    catch (java.lang.NumberFormatException e) {
      System.out.println("usage \"java Td1 <eniter> <entier>\"");
    }
  }
}

class Td2bis {
  public static void main(String [] args) {
    try {
    int x = Integer.valueOf(args[0]).intValue();
    int y = Integer.valueOf(args[1]).intValue();
    System.out.println("La somme de " + x + " est de " + y + " est " + (x+y));
    System.out.println("Le produit de " + x + " est de " + y + " est " + (x*y));
    }
    catch (java.lang.ArrayIndexOutOfBoundsException e) {
      System.out.println("usage \"java Td1 <eniter> <entier>\"");
    }
    catch (java.lang.NumberFormatException e) {
      System.out.println("usage \"java Td1 <eniter> <entier>\"");
    }
  }
}
```

Solution 1.3

Ecrire un programme affiche l'inverse de l'entier donnée argument de la ligne de commande.

```
class Td3 {
  public static void main(String [] args) {
    try {
    int x = Integer.parseInt(args[0]);
    double p = 1.0 / x ;
    System.out.println("L'inverse de " + x + " est " + p);
    }
    catch (java.lang.ArrayIndexOutOfBoundsException e) {
      System.out.println("usage \"java Td2 <eniter>\"");
    }
    catch (java.lang.NumberFormatException e) {
      System.out.println("usage \"java Td2 <eniter>\"");
    }
  }
}
```

Solution 1.4

Ecrire un programme qui converti les francs et écus, sachant qu'un écu vaut $7,22$ francs.

```
class Td4 {
  public static void main(String [] args) {
    try {
    float x = Float.valueOf(args[0]).floatValue();
    float p =  (float)(x / 7.22) ;
    System.out.println(x + " francs valent " + p + " écus");
    }
    catch (java.lang.ArrayIndexOutOfBoundsException e) {
      System.out.println("usage \"java Td4 <double>\"");
    }
    catch (java.lang.NumberFormatException e) {
      System.out.println("usage \"java Td4 <double>\"");
    }
  }
}
```

Solution 1.5

Ecrire un programme qui affiche la décomposition en base 10 d'un entier. Par exemple, le nombre 123 sera affiché sous la forme $3 + 2 \times 10 + 1 \times 10^2$.

```
class Td5 {
  public static void main(String [] args) {
    try {
    int entier = Integer.valueOf(args[0]).intValue();
    int i = 1;
    System.out.print(entier % 10);
    entier = entier / 10;
    while (entier != 0) {
      System.out.print(" + " + entier%10);
      if (i == 1) System.out.print("*10");
      else System.out.print("*10^" + i);
      entier = entier / 10;
      i = i+1;
    }
    System.out.print("\n");
  }
    catch (java.lang.ArrayIndexOutOfBoundsException e) {
      System.out.println("usage \"java Td4 <int>\"");
    }
    catch (java.lang.NumberFormatException e) {
      System.out.println("usage \"java Td4 <int>\"");
    }
  }
}


class Td5bis {
  public static void main(String [] args) {
    try {
```

```
    int entier = Integer.valueOf(args[0]).intValue(), i;
    System.out.print(entier % 10);
    for (i = 1, entier = entier / 10; entier != 0;  entier = entier / 10, i = i+1)
      System.out.print(" + " + (entier%10) + ((i==1) ? "*10" : "*10^" + i));
    System.out.print("\n");
  }
    catch (java.lang.ArrayIndexOutOfBoundsException e) {
      System.out.println("usage \"java Td4 <int>\"");
    }
    catch (java.lang.NumberFormatException e) {
      System.out.println("usage \"java Td4 <int>\"");
    }
  }
}
```

Solution 1.6

Ecrire un programme qui calcule la n^{ime} valeur de la suite de Fibonacci qui définie par

$u_0 = 1$
$u_1 = 1$
$u_n = u_{n-1} + u_{n-2}$, pour $n>2$

```
class Td6 {
  public static void main(String [] args) {
    int n = 0;
    try {
    n = Integer.valueOf(args[0]).intValue();
    }
    catch (java.lang.ArrayIndexOutOfBoundsException e) {
      System.out.println("usage \"java Td4 <int> >= 0\"");
    }
    catch (java.lang.NumberFormatException e) {
      System.out.println("usage \"java Td4 <int> >= 0\"");
    }
    if (n < 0)
      System.out.println("usage \"java Td4 <int> >= 0\"");

    int uNmoins2 = 1, uNmoins1 = 1, uN = 0, i;
    if (n>=2)
      for (i = 2; i++ <= n; ) {
        uN = uNmoins1 + uNmoins2;
        uNmoins2=uNmoins1;
        uNmoins1=uN;
      }
    else  uN = 1;
    System.out.println("Le " + n + "eme élément de la suite de Fibonacci est " + uN);
  }
}
```

Solution 1.7

Ecrire un programme qui imprime le nombre d'indice i tels que $A_i = B_j$ où A et B sont deux tableaux d'entiers de longueur N.

Solution 1.8

Ecrire un programme qui imprime le plus grand élément d'un tableau A de longueur N.

Solution 1.9

Recherche Séquentielle.
Ecrire un programme qui recherche un entier x dans un tableau d'entiers A de longueur N et qui imprime le premier indice i tel que $A_i = x$.

Solution 1.10

Tasser un tableau d'entiers A, tableau de taille N, en supprimant tous les zéros qui y figurent.

Solution 1.11

Insertion Séquentielle.
Soit A un tableau ordonné d'entiers ; tableau de taille NA. Insérer un enter x (s'il n'existe pas déjà) dans ce tableau tout en conservant la relation d'ordre sur les éléments de ce tableau.

Solution 1.12

Suppression Séquentielle.
Soit A un tableau ordonné d'entiers ; tableau de taille NA. Supprimer (s'il existe) l'entier x de ce tableau.

Solution 1.13

Soient A et B deux tableaux d'entiers ordonné ; tableaux de taille NA et NB. Créer un troisième tableau ordonné C constitué de la fusion des deux tableaux précédents.

Solution 1.14

Ecrire un programme qui lit un entier et qui affiche ce même nombre dans sa notation binaire, octale, décimale. Par rapport à l'exercice **??**, l'ordre dans lequel les chiffres doivent être affichés est inversé ; il faut donc utiliser un tableau pour stocker les valeurs et les afficher à la fin du calcul.

Solution 1.15

Schéma de Horner.
Un polynôme $P(x) = a_0 X^n \ldots a_{n_1} X + a_n$ est représenté par la suite de valeurs $(a_0, \ldots, a_{n_1}, a_n)$ de ses coefficients. Ecrire un programme q ui calcule la valeur de $P(X)$ pour une valeur donnée de X en utilisant le schéma d'Horner suivant :

$$P(X) = (\ldots((a_0 \times X + a_1) \times X) + a_2 \times X + a_{n-1}) \times X + a_n.$$

Estimer le nombre de multiplication effectué par ce programme et comparer avec des programmes plus simples.

Solution 1.16

Recherche dicothomique.
Ecrire un programme qui recherche un entier x dans un tableau d'entiers A de longueur N et qui imprime tous les indices i tel que $A_i = x$.

Solution 1.17

Calculer la somme et le produit de deux matrices carrés de taille N.

Solution 1.18

Etant donné une matrice A de taille $N \times M$, on dit qsu'un couple d'indice (p, q) est un min-max de cette matrice si la valeur $a_{p,q}$ est un minimum de la ligne p et un maximum de la colonne q, i.e. :

$$a_{p,q} = min\{A_{p,1}, \ldots, a_{p,M}\} = max\{A_{1,q}, \ldots, a_{N,q}\}.$$

Ecrire un programme qui affiche l'ensemble de tels couples (p, q). La méthode pro posée consiste à effectuer le travail suivant pour chaque ligne p :

- trouver les minima de la ligne p et en mémoriser les numéros de colonne,
- pour chacun de ces rangs q, déterminer si $a_{p,q}$ est un maximum pour sa colonne.

Solution 1.19

Ecrire un programme qui détermine tous les nombres premier compris en 1 et n en utilisant le « crible d'Erastothène ». L'algorithme proposé consiste à disposer d'un tableau des n valeurs consécutifs et on commence par cocher l'entier 1 du tableau. Puis on réitère le procédé suivant tant que le nombre premier que nous allons trouver est inférieur à la racine carrée de n :

A partir du dernier nombre premier trouvé (au démarrage on supposera que c'est 1), chercher un entier non coché.
On démontre que ce entier est bien premier. Cocher alors tous les multiples de ce nombre premier.

Solution 1.20

Programmer le tri à bulles

Fiche 2. Classes et objets

Solution 2.1

Compléter la classe Date en implantant toutes les méthodes de cette classe et donner un exemple d'utilisation.

```
class Date {
    private int jour, mois, annee;
    public void affecter(int j, int m, int a);
    public int quelJour();
    public int quelMois();
    public int quelleAnnee();
    public void lendemain() ;
    public void imprimer();
    public static void listerDates();
}
```

```
package td.classes;

class DateNonConforme extends java.lang.Exception {
  public DateNonConforme(String s) {
    super(s);
  }
}

class Date {
  private int mois, jour, annee;

  public Date(int j, int m, int a) throws DateNonConforme { affecter(j, m, a); }
  public Date(int j, int m) throws DateNonConforme { this(j, m, 97); }
  public Date(int j) throws DateNonConforme { this(j, 12, 97); }
  public Date() throws DateNonConforme { this(1, 12, 97); }
  public void affecter(int j, int m, int a) throws DateNonConforme {
    if (m < 1 || m > 12)
      throw new DateNonConforme("jours "+ j + " (mois " + m + ") annee " + a);
    switch(m) {
    case 4 : case 6 : case 9 : case 11 :
      if (j < 1 || j > 30)
        throw new DateNonConforme("(jours "+ j + ") mois " + m + " annee " + a);
      break;
    case 1 : case 3 : case 5 : case 7 : case 8 : case 10 : case 12 :
      if (j < 1 || j > 31)
        throw new DateNonConforme("(jours "+ j + ") mois " + m + " annee " + a);
      break;
    case 2:
      if ((j < 1 || j > 29) ||  ( (a % 4)!=0  && j==29))
        throw new DateNonConforme("(jours "+ j + ") mois " + m + " annee " + a);
    }
    mois = m; jour = j; annee = a;
  }

  public void lendemain() {
    switch(mois)  {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12 :
      if (++jour == 32) { jour = 1; mois++; break; }
      else return;
    case 4: case 6: case 9: case 11:
      if (++jour == 31) { jour = 1; mois++; break; }
      else return;
```

```
      case 2:
        if ((++jour == 29 && ((annee % 4) != 0)) ||
            (jour == 30 && ((annee % 4) == 0)) ) {
          jour = 1; mois++; break;
        }
        else return;
      }
      if (mois == 13) { mois = 1; annee++; }
    }

    public int QuelJour() { return jour; }
    public int QuelMois() { return mois; }
    public int QuelleAnnee() { return annee; }
    public void imprimer() { System.out.println(jour + "/" + mois + "/" + annee);}

    public static void main(String argv[]) {
      try {
        Date d = new Date();
        d.affecter(27,2,68);
        d.lendemain(); d.imprimer();
        d.lendemain(); d.imprimer();
        d.lendemain(); d.imprimer();
      }
      catch (DateNonConforme e) {System.out.println(e);};
    }
  }
```

Solution 2.2

Réaliser une classe `Point` permettant de manipuler un point du plan. On prévoira

- un constructeur
- les méthodes pour accéder aux coordonnés et pour les modifier,
- une méthode `deplacer` qui effectue une translation définie par les arguments.
- une méthode `afficher` qui affiche les coordonnées du point.

Donner un exemple d'utilisation.

```
package td.classes;

public class Point {
  private double x, y;                    // Coordonnées catésiennes du point
  public Point(double abs, double ord) { x = abs; y = ord; }
  public double abs() { return x; }
  public double ord() { return y; }
  public void deplacer(double dx, double dy) { x += dx; y += dy; }
  public void afficher() { System.out.println("(" + x + ", " + y + ")"); }
  public String toString() { return "(" + x + ", " + y + ")"; }
  public static void main(String argv[]) {
    Point p = new Point(10.5, -4.3);
    p.afficher();
    p.deplacer(-1.2, 50.7);
    p.afficher();
    System.out.println(p);
  }
}
```

Solution 2.3

Définir une classe `PileEnt` d'entiers munis de son ou ses constructeurs, les méthodes `empile` et `depiler`.

```
package td.classes;

class ErreurPileEntier extends java.lang.Exception {
  public ErreurPileEntier(String s) {
    super(s);
  }
}

class PileEntier  {
  private int [] tab = null;
```

```
    private int sommet;
      public PileEntier(int t) { sommet = 0; tab = new int [t]; }
      public int depiler()   throws ErreurPileEntier {
        if (sommet > 0) return tab[--sommet];
        else throw new ErreurPileEntier("Pile Vide");
      }
    public void empiler(int x)  throws ErreurPileEntier {
      if (sommet < tab.length)
        tab[sommet++] = x;
      else throw new ErreurPileEntier("Pile Pleine");
      }
    public static void main(String argv[]) {
      PileEntier p = new PileEntier(3);
      int i = 0;
      try { i = p.depiler(); System.out.println("depile: " + i); }
      catch (ErreurPileEntier e) { System.out.println(e); }
      try { p.empiler(1);  p.empiler(2);  p.empiler(3);  p.empiler(4); }
      catch (ErreurPileEntier e) {
        System.out.println(e);
        System.out.println("Voila le contenu de la pile:");
        for (i = 0; i < p.tab.length; i++) System.out.print(p.tab[i] + " ");
        System.out.println("");
      }
    }
  }
```

Solution 2.4

Définir une classe `Individu` composée d'un nom, d'une adresse et d'un numéro de téléphone. Donner le constructeur, les méthodes d'affectation et de consultation des noms, adresse et numéro de téléphone.

```
package td.classes;

class Individu1 {
  private String nom = null;
  private String adresse = null;
  private String telephone = null;
  public Individu1(String n, String a, String t) { nom = n; adresse = a; telephone = t; }

  // méthodes d'affectation des champs
  public void affecter_nom(String n) { nom = n; }
  public void affecter_adresse(String a) {adresse = a; }
  public void affecter_telephone(String t) {telephone = t; }

  // méthodes de consultation des champs
  public String consulter_nom() { return nom; }
  public String consulter_adresse() { return adresse; }
  public String consulter_telephone() { return telephone; }

  public void afficher() {
    System.out.println(consulter_nom() + " " + consulter_adresse() + " " + consulter_telephone());
  }
  public static void main(String argv[]) {
    Individu1 t = new Individu1("Touraivane", "ESIL-GBM", "0491828538");
    Individu1 g = new Individu1("Girard", "IUT-GTR", "0491828538");
    t.afficher();
    t.afficher();
  }
}
```

Solution 2.5

Modifier le classe `Individu` pour pouvoir afficher le nombre d'objets créés.

```
package td.classes;

class Individu2 {
  private static  int nbreIndividus = 0;
  private String nom = null;
  private String adresse = null;
  private String telephone = null;
  public Individu2(String n, String a, String t) { nom = n; adresse = a; telephone = t; nbreIndividus++; }
```

```
  // méthodes d'affectation des champs
  public void affecter_nom(String n) { nom = n; }
  public void affecter_adresse(String a) {adresse = a; }
  public void affecter_telephone(String t) {telephone = t; }

  // méthodes de consultation des champs
  public String consulter_nom() { return nom; }
  public String consulter_adresse() { return adresse; }
  public String consulter_telephone() { return telephone; }
  public int afficheNbreIndividu() { return nbreIndividus; }
  public void afficher() {
    System.out.println(consulter_nom() + " " + consulter_adresse() + " " + consulter_telephone());
  }

  public static void main(String argv[]) {
    Individu2 t = new Individu2("Touraivane", "ESIL-GBM", "0491828538");
    Individu2 g = new Individu2("Girard", "IUT-GTR", "0491828538");
    System.out.println("Le nombre d'individus crées est de " + nbreIndividus);
    System.out.print("\t");  t.afficher();
    System.out.print("\t");  g.afficher();
  }
}
```

```
// Version avec héritage
package td.classes;

class Individu3 extends Individu1 {
  private static  int nbreIndividus = 0;
  public Individu3(String n, String a, String t) { super(n, a, t); nbreIndividus++; }
  public int afficheNbreIndividu() { return nbreIndividus; }
  public static void main(String argv[]) {
    Individu1 t = new Individu3("Touraivane", "ESIL-GBM", "0491828538");
    Individu1 g = new Individu3("Girard", "IUT-GTR", "0491828538");

    System.out.println("Le nombre d'individus crées est de " + nbreIndividus);
    System.out.print("\t"); ((Individu1)t).afficher();
    System.out.print("\t"); ((Individu1)g).afficher();
  }
}
```

Solution 2.6

Modifier le classe `Individu` de manière à

1. gérer une liste des individus créés
2. afficher la liste des individus créés

```
package td.classes;

class Individu4 {
  private static  int nbreIndividus = 0;
  private static  Individu4 listIndividus = null;
  private Individu4 suiv;
  private String nom = null;
  private String adresse = null;
  private String telephone = null;
  public Individu4(String n, String a, String t) {
    nom = n; adresse = a; telephone = t;
    nbreIndividus++;
    suiv = listIndividus; listIndividus = this;
  }

  // méthodes d'affectation des champs
  public void affecter_nom(String n) { nom = n; }
  public void affecter_adresse(String a) {adresse = a; }
  public void affecter_telephone(String t) {telephone = t; }

  // méthodes de consultation des champs
  public String consulter_nom() { return nom; }
  public String consulter_adresse() { return adresse; }
  public String consulter_telephone() { return telephone; }
```

```
  public void afficher() {
    System.out.println(consulter_nom() + " " + consulter_adresse() + " " + consulter_telephone());
  }

  public static void afficheLesIndividu() {
    for (Individu4 crt = listIndividus; crt != null; crt = crt.suiv)
      crt.afficher();
  }
  public int afficheNbreIndividu() { return nbreIndividus; }
  public static void main(String argv[]) {
    Individu4 t = new Individu4("Touraivane", "ESIL-GBM", "0491828538");
    Individu4 g = new Individu4("Girard", "IUT-GTR", "0491828538");
    System.out.println("Le nombre d'individus crées est de " + nbreIndividus);
    afficheLesIndividu();
  }
}
```

```
// Version avec héritage

package td.classes;

class Individu5 extends Individu1 {
  private static  int nbreIndividus = 0;
  private static  Individu5 listIndividus = null;
   private Individu5 suiv;

  public Individu5(String n, String a, String t) {
    super(n, a, t);
    nbreIndividus++;
    suiv = listIndividus;
    listIndividus = this;
  }
  public static void afficheLesIndividu() {
    for (Individu5 crt = listIndividus; crt != null; crt = crt.suiv)
      ((Individu1)crt).afficher();
  }
  public int afficheNbreIndividu() { return nbreIndividus; }
  public static void main(String argv[]) {
    Individu5 t = new Individu5("Touraivane", "ESIL-GBM", "0491828538");
    Individu5 g = new Individu5("Girard", "IUT-GTR", "0491828538");

    System.out.println("Le nombre d'individus crées est de " + nbreIndividus);
    afficheLesIndividu();
  }
}
```

Solution 2.7

Définir une classe vecteur et une classe matrice carrée. Définir une fonction `multiplier` qui calcule le produit d'une matrice par un vecteur.

```
package td.classes;

class AccesVecteurException extends java.lang.Exception {
  public AccesVecteurException(String s) {
    super(s);
  }
}

class AccesMatriceException extends java.lang.Exception {
  public AccesMatriceException(String s) {
    super(s);
  }
}

class Vecteur  {
  private double [] t;
  public Vecteur(int n) { t = new double[n]; }
  public int taille()  { return t.length; }
  public double elt(int i) throws AccesVecteurException {
```

```
    if (i>=t.length || i<0) throw new AccesVecteurException("debordement");
    return t[i];
  }
  public void affecter(int i, double d) throws AccesVecteurException {
    if (i>=t.length || i<0) throw new AccesVecteurException("debordement");
    t[i] = d;
  }
  public void afficher() {
    for (int i = 0; i<t.length; i++)
      System.out.print(t[i]+ "\t");
    System.out.println("");
  }
}

class Matrice  {
  private Vecteur [] t;
  public Matrice(int n)  {
    t = new Vecteur[n];
    for (int i=0; i<t.length; i++) t[i] = new Vecteur(n);
  }
  public double elt(int i, int j) throws AccesMatriceException, AccesVecteurException {
    if (i>=t.length || i<0) throw new AccesMatriceException("debordement");
    if (i>=t.length || i<0) throw new AccesMatriceException("debordement");
    return t[i].elt(j);
  }
  public void affecter(int i, int j, double d) throws AccesMatriceException, AccesVecteurException {
    if (i>=t.length || i<0) throw new AccesMatriceException("debordement");
    if (i>=t.length || i<0) throw new AccesMatriceException("debordement");
    t[i].affecter(j, d);
  }
  public void afficher() {
    for (int i = 0; i<t.length; i++)
      t[i].afficher();
    System.out.println("");
  }

  public Vecteur multiplier(Vecteur v) throws AccesMatriceException, AccesVecteurException  {
    Vecteur r= new Vecteur(t.length);
    for (int i = 0; i<t.length; i++) {
      r.affecter(i, 0);
      for (int j = 0; j<t.length; j++) {
        r.affecter(i, r.elt(i)+ elt(i,j) * v.elt(j));
      }
    }
    return r;
  }
}


class ProduitMatrice {
  public static void main(String argv[]) {
    try {
      Vecteur v = new Vecteur(3);
      Vecteur r;
      Matrice m = new Matrice(3);
      for (int i = 0; i<v.taille(); i++)
        for (int j = 0; j<v.taille(); j++)
          m.affecter(i,j,(double)i+j);
      for (int i = 0; i<v.taille(); i++)
        v.affecter(i,(double)i);
      r = m.multiplier(v);
      v.afficher();
      m.afficher();
      r.afficher();
    }
    catch (AccesVecteurException e) {System.out.println(e);}
    catch (AccesMatriceException e) {System.out.println(e);}
  }
}
```

Solution 2.8

Modifier le programme précédant pour que la fonction `multiplier` utilisent les champs privés des classes `Matrice` et `Vecteur`.

```
package td.classes;

class AccesVecteurException extends java.lang.Exception {
  public AccesVecteurException(String s) {
    super(s);
  }
}

class AccesMatriceException extends java.lang.Exception {
  public AccesMatriceException(String s) {
    super(s);
  }
}

class Vecteur  {
  private double [] t;
  public Vecteur(int n) { t = new double[n]; }
  public int dimension()  { return t.length; }
  public double elt(int i) throws AccesVecteurException {
    if (i>=t.length || i<0) throw new AccesVecteurException("debordement");
    return t[i];
  }
  public void affecter(int i, double d) throws AccesVecteurException {
    if (i>=t.length || i<0) throw new AccesVecteurException("debordement");
    t[i] = d;
  }
  public void afficher() {
    for (int i = 0; i<t.length; i++)
      System.out.print(t[i]+ "\t");
    System.out.println("\n");
  }
  public boolean equals(Vecteur v) {
    if (t.length != v.dimension()) return false;
    else for (int i = 0; i<t.length; i++) {
      try {
        if (t[i] != v.elt(i)) return false;
      }
      catch (AccesVecteurException e) {}
    }
    return true;
  }
  public String toString() {
    String s = "";
    for (int i = 0; i<t.length; i++)
      s = s + t[i] + "\t";
    return s + "\n";
  }
}

class Matrice  {
  private Vecteur [] t;
  public Matrice(int n)  {
    t = new Vecteur[n];
    for (int i=0; i<t.length; i++) t[i] = new Vecteur(n);
  }
  public int dimension()  { return t.length; }
  public double elt(int i, int j) throws AccesMatriceException, AccesVecteurException {
    if (i>=t.length || i<0) throw new AccesMatriceException("debordement");
    if (i>=t.length || i<0) throw new AccesMatriceException("debordement");
    return t[i].elt(j);
  }
  public void affecter(int i, int j, double d) throws AccesMatriceException, AccesVecteurException {
    if (i>=t.length || i<0) throw new AccesMatriceException("debordement");
    if (i>=t.length || i<0) throw new AccesMatriceException("debordement");
    t[i].affecter(j, d);
  }
  public void afficher() {
    for (int i = 0; i<t.length; i++)
      t[i].afficher();
```

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

    public Vecteur multiplier(Vecteur v) throws AccesMatriceException, AccesVecteurException  {
      Vecteur r= new Vecteur(t.length);
      for (int i = 0; i<t.length; i++) {
        r.affecter(i, 0);
        for (int j = 0; j<t.length; j++) {
          r.affecter(i, r.elt(i)+ elt(i,j) * v.elt(j));
          }
      }
      return r;
    }
    public boolean equals(Matrice m) {
      if (t.length != m.dimension()) return false;
      else for (int i = 0; i<t.length; i++)
        // if (! t[i].equals(m.t[i])) return false;
        if (t[i] != m.t[i]) return false;
      return true;
    }
    public String toString() {
      String s = "";
      for (int i = 0; i<t.length; i++)
        s = s + t[i] + "\n";
      return s ;
    }
  }

  class ProduitMatrice2 {
    public static void main(String argv[]) {
      try {
        Vecteur v = new Vecteur(3);
        Vecteur r;
        Matrice m = new Matrice(3);
        for (int i = 0; i<v.dimension(); i++)
          for (int j = 0; j<v.dimension(); j++) m.affecter(i,j,(double)i+j);
        for (int i = 0; i<v.dimension(); i++) v.affecter(i,(double)i);
        r = m.multiplier(v);
        System.out.println(v + "\n" + m + "\n" + r);
      }
      catch (AccesVecteurException e) {System.out.println(e);}
      catch (AccesMatriceException e) {System.out.println(e);}
    }
  }
```

Solution 2.9

Compléter les classes `Vecteur` et `Matrice` en surchargeant le méthode equals de la classe `Object` et de manière à pouvoir imprimer de la manière suivante :

```
Matrice m;
Vecteur v;
...
System.out.println(v + m);
```

Solution 2.10

Créer une classe liste permettant de manipuler des listes chaînées dans laquelle la nature de l'information associé à chaque noeud n'est pas connu.

```
package td.heritage;

class ListesAccessException extends java.lang.Exception {
  public ListesAccessException(String s) {
    super(s);
  }
}

class Maillon {
  Object info;
  Maillon suiv;
```

```
  public Maillon(Object o, Maillon s) { info = o; suiv = s; }
}

class Listes {
  Maillon debut;
  Maillon crt;

  public Listes() { debut = crt = null; }

  public void ajouter( Object o) { debut = new Maillon(o, debut); }
  public void supprimer( Object o) {
    if (crt == debut) crt = debut.suiv;
    debut = debut.suiv;
  }
  public Object premier() throws ListesAccessException {
    if (crt == null) throw new ListesAccessException("Liste vide");
    crt = debut;
    return crt.info;
  }
  public Object prochain() throws ListesAccessException {
    if (crt == null) throw new ListesAccessException("Liste vide");
    crt = crt.suiv;
    if (crt == null) throw new ListesAccessException("Liste vide");
    return crt.info;
  }
  public void afficher() {
    Maillon m = debut;
    while (m != null) {
      System.out.print(m.info + "  ");
      m = m.suiv;
    }
  }
  public static void main(String argv[]) {
    Listes l = new Listes();
    l.ajouter(new Integer(1));
    l.ajouter(new Integer(2));
    l.ajouter(new Integer(3));
    l.ajouter(new Integer(4));
    l.afficher();
  }
}
```

Solution 2.11

Java fournit la classe java.util.Vector. En consultant la documentation de la cette classe, programmer cette dernière.

```
/*
 * @(#)Vector.java   1.39 98/04/22
 *
 * Copyright 1994-1997 by Sun Microsystems, Inc.,
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of Sun Microsystems, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * you entered into with Sun.
 */

   package java.util;

   public
   class Vector implements Cloneable, java.io.Serializable {
      protected Object elementData[];
      protected int elementCount;
      protected int capacityIncrement;
      private static final long serialVersionUID = -2767605614048989439L;
      public Vector(int initialCapacity, int capacityIncrement) {
         super();
         this.elementData = new Object[initialCapacity];
         this.capacityIncrement = capacityIncrement;
```

```
}
public Vector(int initialCapacity) {
   this(initialCapacity, 0); }
public Vector() {
   this(10); }
public final synchronized void copyInto(Object anArray[]) {
   int i = elementCount;
   while (i-- > 0) {
      anArray[i] = elementData[i];
   }
}
public final synchronized void trimToSize() {
   int oldCapacity = elementData.length;
   if (elementCount < oldCapacity) {
      Object oldData[] = elementData;
      elementData = new Object[elementCount];
      System.arraycopy(oldData, 0, elementData, 0, elementCount);
   }
}
public final synchronized void ensureCapacity(int minCapacity) {
   if (minCapacity > elementData.length) {
      ensureCapacityHelper(minCapacity);
   }
}
private void ensureCapacityHelper(int minCapacity) {
   int oldCapacity = elementData.length;
   Object oldData[] = elementData;
   int newCapacity = (capacityIncrement > 0) ?
   (oldCapacity + capacityIncrement) : (oldCapacity * 2);
   if (newCapacity < minCapacity) {
      newCapacity = minCapacity;
   }
   elementData = new Object[newCapacity];
   System.arraycopy(oldData, 0, elementData, 0, elementCount);
}
public final synchronized void setSize(int newSize) {
   if ((newSize > elementCount) && (newSize > elementData.length)) {
      ensureCapacityHelper(newSize);
   }
   else {
      for (int i = newSize ; i < elementCount ; i++) {
         elementData[i] = null;
      }
   }
   elementCount = newSize;
}
public final int capacity() {
   return elementData.length; }
public final int size() {
   return elementCount;}
public final boolean isEmpty() {
   return elementCount == 0;}
public final synchronized Enumeration elements() {
   return new VectorEnumerator(this);}
public final boolean contains(Object elem) {
   return indexOf(elem, 0) >= 0;}
public final int indexOf(Object elem) {
   return indexOf(elem, 0);
}
public final synchronized int indexOf(Object elem, int index) {
   for (int i = index ; i < elementCount ; i++) {
      if (elem.equals(elementData[i])) {
         return i;}
   }
   return -1;
}
public final int lastIndexOf(Object elem) {
   return lastIndexOf(elem, elementCount-1);
}
public final synchronized int lastIndexOf(Object elem, int index) {
   for (int i = index ; i >= 0 ; i--) {
      if (elem.equals(elementData[i])) {
```

```
            return i;
         }
      }
      return -1;
   }
   public final synchronized Object elementAt(int index) {
      if (index >= elementCount) {
         throw new ArrayIndexOutOfBoundsException(index + " >= " + elementCount);
      }
      try {
         return elementData[index];
      }
         catch (ArrayIndexOutOfBoundsException e) {
            throw new ArrayIndexOutOfBoundsException(index + " < 0");
         }
   }
   public final synchronized Object firstElement() {
      if (elementCount == 0) {
         throw new NoSuchElementException();
      }
      return elementData[0];
   }
   public final synchronized Object lastElement() {
      if (elementCount == 0) {
         throw new NoSuchElementException();
      }
      return elementData[elementCount - 1];
   }
   public final synchronized void setElementAt(Object obj, int index) {
      if (index >= elementCount) {
         throw new ArrayIndexOutOfBoundsException(index + " >= " +
            elementCount);
      }
      elementData[index] = obj;
   }
   public final synchronized void removeElementAt(int index) {
      if (index >= elementCount) {
         throw new ArrayIndexOutOfBoundsException(index + " >= " +
            elementCount);
      }
      else if (index < 0) {
         throw new ArrayIndexOutOfBoundsException(index);
      }
      int j = elementCount - index - 1;
      if (j > 0) {
         System.arraycopy(elementData, index + 1, elementData, index, j);
      }
      elementCount--;
      elementData[elementCount] = null; /* to let gc do its work */
   }
   public final synchronized void insertElementAt(Object obj, int index) {
      int newcount = elementCount + 1;
      if (index >= newcount) {
         throw new ArrayIndexOutOfBoundsException(index
            + " > " + elementCount);
      }
      if (newcount > elementData.length) {
         ensureCapacityHelper(newcount);
      }
      System.arraycopy(elementData, index, elementData, index + 1, elementCount - index);
      elementData[index] = obj;
      elementCount++;
   }
   public final synchronized void addElement(Object obj) {
      int newcount = elementCount + 1;
      if (newcount > elementData.length) {
         ensureCapacityHelper(newcount);
      }
      elementData[elementCount++] = obj;
   }
   public final synchronized boolean removeElement(Object obj) {
      int i = indexOf(obj);
```

```
      if (i >= 0) {
         removeElementAt(i);
         return true;
      }
      return false;
   }
   public final synchronized void removeAllElements() {
      for (int i = 0; i < elementCount; i++) {
         elementData[i] = null;
      }
      elementCount = 0;
   }
   public synchronized Object clone() {
      try {
         Vector v = (Vector)super.clone();
         v.elementData = new Object[elementCount];
         System.arraycopy(elementData, 0, v.elementData, 0, elementCount);
         return v;
      }
         catch (CloneNotSupportedException e) {
         // this shouldn't happen, since we are Cloneable
            throw new InternalError();
         }
   }
   public final synchronized String toString() {
      int max = size() - 1;
      StringBuffer buf = new StringBuffer();
      Enumeration e = elements();
      buf.append("[");

      for (int i = 0 ; i <= max ; i++) {
         String s = e.nextElement().toString();
         buf.append(s);
         if (i < max) {
            buf.append(", ");
         }
      }
      buf.append("]");
      return buf.toString();
   }
}

final
class VectorEnumerator implements Enumeration {
   Vector vector;
   int count;

   VectorEnumerator(Vector v) {
      vector = v;
      count = 0;
   }

   public boolean hasMoreElements() {
      return count < vector.elementCount;
   }

   public Object nextElement() {
      synchronized (vector) {
         if (count < vector.elementCount) {
            return vector.elementData[count++];
         }
      }
      throw new NoSuchElementException("VectorEnumerator");
   }
}
```

Fiche 3. Héritage, interfaces et packages

Solution 3.1

En se basant sur la classe `Point`, créer une classe `PointA` comportant une méthode supplémentaire `distance` qui détermine la distance d'un point au centre.

- Les coordonnés sont des champs privés de la classe `Point`.

```
package td.heritage;

import td.classes.Point;

class PointA extends Point {
  public PointA(double x, double y) { super(x, y); }
  public double distance() { return Math.sqrt(abs() * abs() + ord() * ord()); }
  public double distance(Point p) {
    return Math.sqrt((p.abs() - abs()) * (p.abs() - abs()) + (p.ord() - ord()) * (p.ord() - ord()));
  }
  public static void main(String args []) {
    PointA p1 = new PointA(10, -4);
    PointA p2 = new PointA(0, 52);
    System.out.println("La distance de " + p1 + " au centre est " + p1.distance());
    System.out.println("La distance de " + p2 + " au centre est " + p2.distance());
    System.out.println("La distance entre " + p1 + " et " + p2 + " est " + p1.distance(p2));
  }
}
```

- Les coordonnés sont des champs privés de la classe `Protected`.

```
package td.heritage;
import td.classes.Point2;

class PointA2 extends Point2 {
  public PointA2(double x, double y) { super(x, y); }
  public double distance() { return Math.sqrt(x*x + y*y); }
  public double distance(PointA2 p) { return Math.sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y)); }
  public static void main(String args []) {
    PointA2 p1 = new PointA2(10.5, -4.3);
    PointA2 p2 = new PointA2(-0.5, 52);
    System.out.println("La distance de " + p1 + " au centre est " + p1.distance());
    System.out.println("La distance de " + p2 + " au centre est " + p2.distance());
    System.out.println("La distance entre " + p1 + " et " + p2 + " est " + p1.distance(p2));
  }
}
```

Solution 3.2

Définir une classe `PointColore` destiné à contenir la couleur d'un point. On dotera cette classe d'une nouvelle fonction `imprimer` qui affiche également la couleur du point.

```
package td.heritage;
import td.classes.Point;

class PointColore extends Point {
  private int couleur;
  public PointColore(double x, double y, int c) { super(x, y); couleur = c; }
  public String toString() {
    return super.toString() + " couleur: " + couleur;
  }
  public static void main(String args []) {
```

```
    PointColore p1 = new PointColore(10, -4, 0);
    PointColore p2 = new PointColore(0, 52, 1);

    System.out.println( p2);
    System.out.println( p1);
  }
}
```

Solution 3.3

Même question en n'utilisant pas une classe dérivée.

```
package td.heritage;
import td.classes.Point;

class PointColore2 {
  private Point p;
  private int couleur;
  public PointColore2(double x, double y, int c) { p = new Point(x, y);  couleur = c; }
  public String toString() {
    return p.toString() + " couleur: " + couleur;
  }
  public static void main(String args []) {
    PointColore2 p1 = new PointColore2(10, -4, 0);
    PointColore2 p2 = new PointColore2(0, 52, 1);

    System.out.println( p2);
    System.out.println( p1);
  }
}
```

Solution 3.4

Compléter la classe `Forme` suivante et définir les sous classes `Rectangle`, `Ovale` et `LigneBrisée`.

```
abstract class Forme {
  Point origine;
  protected Forme suiv;
  private static Forme listeDesFormes = null;
  public Forme(int x, int y) {  }
  public Forme(int x, int y, int fond, int contour) { ... }
  public static void afficherToutesLesFormes() { ... }
  public void supprimerToutesLesFormes() { ... }
  public void supprimer() { ... }
  abstract void deplacer(int x, int y);
  abstract void modifier();
  abstract void afficher();
}

final class Rectangle extends Forme  {  ... }
final class Oval extends Forme  { ... }
final class LigneBrisee extends Forme  { ... }

class Dessin {
  public static void main(String args[]) {
    Rectangle r1 = new Rectangle(0,0, 10, 10, 1, 1);
    Rectangle r2 = new Rectangle(5,5, 20, 10, 0, 2);
    Rectangle r3 = new Rectangle(10, 13, 25,100, 2, 2);
    Oval o1 = new Oval(1, 3, 5,10, 2, 2);
    int [] x =  {1, 3, 5}, y =  {5, 10, 3};
    LigneBrisee l1 = new LigneBrisee(x, y,  2);
    Forme.afficherToutesLesFormes();
    r2.supprimer();
    Forme.afficherToutesLesFormes();
  }
}
```

```
package td.heritage;

 class Point {
  private int x, y;
  public Point(int abs, int ord) { x = abs; y = ord; }
  public double abs() { return x; }
```

```
  public double ord() { return y; }
  public String toString() { return "(" + x + ", " + y + ")"; }
  public void deplacer(int dx, int dy) { x += dx; y += dy; }
}

abstract class Forme {
  Point origine;
  protected int couleurFond;
  protected int couleurContour;
  protected Forme suiv;
  private static Forme listeDesFormes = null;

  public Forme(int x, int y) { this(x, y, 0, 1); }
  public Forme(int x, int y, int fond, int contour) {
    origine = new Point(x, y);  couleurFond = fond; couleurContour=contour;
    this.suiv = listeDesFormes; listeDesFormes = this;
  }
  public static void afficherToutesLesFormes() {
    for (Forme f = listeDesFormes; f != null; f = f.suiv)
      f.afficher();
    System.out.println("---------------------------");
  }
  public void supprimerToutesLesFormes() { listeDesFormes = null; }
  public void setFond(int i) { couleurFond = i; }
  public int getFond() { return couleurFond; }
  public void setContour(int i) { couleurContour = i; }
  public int getContour() { return couleurContour; }
  public void supprimer() {
    Forme f, prec;
    for (f = listeDesFormes, prec = null; f != this; prec = f, f = f.suiv);
    if (prec == null) listeDesFormes = listeDesFormes.suiv;
    else prec.suiv = this.suiv;
    this.suiv = null;
  }

  public abstract void deplacer(int x, int y);
  public abstract void modifier();
  public abstract void afficher();
}


final class Rectangle extends Forme  {
  Point fin;
  public Rectangle(int x0, int y0, int x1, int y1, int fond, int contour) {
    super(x0, y0, fond, contour);
    fin = new Point(x1, y1);
  }
  public void deplacer(int x, int y) { origine.deplacer(x, y); fin.deplacer(x, y); }
  public void modifier() { }
  public void afficher() { System.out.println("Rectangle["+origine+" ,"+fin+"]"); }

}


final class Oval extends Forme  {
  Point fin;
  public Oval(int x0, int y0, int x1, int y1, int fond, int contour) {
    super(x0, y0, fond, contour);
    fin = new Point(x1, y1);
  }
  public void deplacer(int x, int y) { origine.deplacer(x, y); fin.deplacer(x, y); }
  public void modifier() { }
  public void afficher() { System.out.println("Oval["+origine+" ,"+fin+"]"); }
}


final class LigneBrisee extends Forme  {
  Point [] listePoints;
  public LigneBrisee(int [] x, int [] y, int contour) {
    super(x[0], y[0], 0, contour);
    listePoints = new Point[x.length];
    for (int i=1; i < listePoints.length; listePoints[i]=new Point(x[i], y[i]), i++);
```

```
  }
  public void deplacer(int x, int y) {
    origine.deplacer(x, y);
    for (int i=1; i < listePoints.length; i++)
      listePoints[i].deplacer(x, y);
  }
  public void modifier() { }
  public void afficher() {
    System.out.print("Ligne["+origine);
    for (int i=1; i < listePoints.length; i++)
      System.out.print(", " + listePoints[i]);
    System.out.println("]");
  }
}

class Dessin {
  public static void main(String args[]) {
    Rectangle r1 = new Rectangle(0,0, 10, 10, 1, 1);
    Rectangle r2 = new Rectangle(5,5, 20, 10, 0, 2);
    Rectangle r3 = new Rectangle(10, 13, 25,100, 2, 2);
    Oval o1 = new Oval(1, 3, 5,10, 2, 2);
    int [] x =  {1, 3, 5};
    int [] y =  {5, 10, 3};
    LigneBrisee l1 = new LigneBrisee(x, y,  2);

    Forme.afficherToutesLesFormes();
    r2.supprimer();
    Forme.afficherToutesLesFormes();
  }
}
```

Solution 3.5

Même question en utilisant une interface `Dessinable` au lieu des méthodes abstraites.

```
package td.heritage;

class Point {
  private int x, y;
  public Point(int abs, int ord) { x = abs; y = ord; }
  public double abs() { return x; }
  public double ord() { return y; }
  public String toString() { return "(" + x + ", " + y + ")"; }
  public void deplacer(int dx, int dy) { x += dx; y += dy; }
}
interface Dessinnable {
  void deplacer(int x, int y);
  void supprimer();
  void modifier();
  void afficher();
}

class LesFormes {
  private Dessinnable forme;
  private LesFormes suiv;
  private static LesFormes listeDesFormes = null;
  public LesFormes(Dessinnable f) {
    forme = f;
    suiv = listeDesFormes;
    listeDesFormes = this;
  }
  public static void afficherToutesLesFormes() {
    for (LesFormes f = listeDesFormes; f != null; f = f.suiv)
      f.forme.afficher();
    System.out.println("---------------------------");
  }
  public void supprimerToutesLesFormes() { listeDesFormes = null; }
  public static void supprimer(Dessinnable d) {
    LesFormes f, prec;
    for (f = listeDesFormes, prec = null; f.forme != d; prec = f, f = f.suiv);
    if (prec == null) listeDesFormes = listeDesFormes.suiv;
    else prec.suiv = f.suiv;
```

```
  }
}

final class Rectangle implements Dessinnable {
  Point fin;
  Point origine;
  protected int couleurFond;
  protected int couleurContour;
  public Rectangle(int x0, int y0, int x1, int y1, int fond, int contour) {
    origine = new Point(x0, y0);
    fin = new Point(x1, y1);
    couleurFond = fond;
    couleurContour = contour;
    new LesFormes(this);
  }
  public void deplacer(int x, int y) { origine.deplacer(x, y); fin.deplacer(x, y); }
  public void supprimer() { LesFormes.supprimer(this); }
  public void modifier() { }
  public void afficher() { System.out.println("Rectangle["+origine+" ,"+fin+"]"); }

}

final class Oval  implements Dessinnable {
  Point fin;
  Point origine;
  protected int couleurFond;
  protected int couleurContour;
  public Oval(int x0, int y0, int x1, int y1, int fond, int contour) {
    origine = new Point(x0, y0);
    fin = new Point(x1, y1);
    couleurFond = fond;
    couleurContour = contour;
    new LesFormes(this);
  }
  public void deplacer(int x, int y) { origine.deplacer(x, y); fin.deplacer(x, y); }
  public void supprimer() { LesFormes.supprimer(this); }
  public void modifier() { }
  public void afficher() { System.out.println("Oval["+origine+" ,"+fin+"]"); }
}

final class LigneBrisee  implements Dessinnable {
  Point [] listePoints;
  protected int couleurFond;
  protected int couleurContour;
  public LigneBrisee(int [] x, int [] y,  int fond, int contour) {
    listePoints = new Point[x.length];
    for (int i=0; i < listePoints.length; listePoints[i]=new Point(x[i], y[i]), i++);
    couleurFond = fond;
    couleurContour = contour;
    new LesFormes(this);
  }
  public void deplacer(int x, int y) {
    for (int i=0; i < listePoints.length; i++)
      listePoints[i].deplacer(x, y);
  }
  public void supprimer() { LesFormes.supprimer(this); }
  public void modifier() { }
  public void afficher() {
    System.out.print("Ligne[");
    for (int i=0; i < listePoints.length; i++)
      System.out.print(", " + listePoints[i]);
    System.out.println("]");
  }
}

class Dessinner {
  public static void main(String args[]) {
    Rectangle r1 = new Rectangle(0,0, 10, 10, 1, 1);
    Rectangle r2 = new Rectangle(5,5, 20, 10, 0, 2);
    Rectangle r3 = new Rectangle(10, 13, 25,100, 2, 2);
    Oval o1 = new Oval(1, 3, 5,10, 2, 2);
    int [] x =  {1, 3, 5};
```

```
    int [] y =  {5, 10, 3};
    LigneBrisee l1 = new LigneBrisee(x, y, 0, 2);

    LesFormes.afficherToutesLesFormes();
    r2.supprimer();
    LesFormes.afficherToutesLesFormes();
  }
}
```

Solution 3.6

Java fournit la classe `java.util.Stack` sous classe de la classe `java.util.Vector`. En consultant la documentation de la cette classe, programmer la classe `java.util.stack`.

```
/*
 * @(#)Stack.java 1.17 98/04/22
 *
 * Copyright 1994-1997 by Sun Microsystems, Inc.,
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of Sun Microsystems, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * you entered into with Sun.
 */
  public
  class Stack extends Vector {

     public Object push(Object item) {  addElement(item);
        return item;
     }
     public synchronized Object pop() {
        Object   obj;
        int   len = size();
        obj = peek();
        removeElementAt(len - 1);
        return obj;
     }
     public synchronized Object peek() {
        int   len = size();
        if (len == 0)
           throw new EmptyStackException();
        return elementAt(len - 1);
     }
     public boolean empty() {
        return size() == 0;
     }
     public synchronized int search(Object o) {
        int i = lastIndexOf(o);

        if (i >= 0) {
           return size() - i;
        }
        return -1;
     }
     private static final long serialVersionUID = 1224463164541339165L;
  }
```

Fiche 4. Chaînes de caractères

Solution 4.1

Chercher le nombre d'occurrences d'une sous chaîne dans une chaîne.

Solution 4.2

Ecrire un programme qui recopie en l'inversant une chaîne de caractères passé en argument de la ligne de commande.

Première solution

```
public class InvCopie {
  public static void main(String[] args) {
    StringBuffer s = new StringBuffer(args[0].length());
    for(int j=args[0].length()-1; j>=0; j--)
     s.append(args[0].charAt(j));
    System.out.println(s);
  }
}
```

Autre solution

```
public class InvCopie2 {
  public static void main(String[] args) {
    StringBuffer s = new StringBuffer(args[0]);
    s = s.reverse();
    System.out.println(s);
  }
}
```

Solution 4.3

Ecrire un programme qui inverse sans recopier une chaîne de caractères lue au clavier.

```
import java.io.*;

public class InvCopie3 {
  public static void main(String[] args) throws IOException {
    BufferedReader din = new BufferedReader(new InputStreamReader(System.in));
    StringBuffer s = new StringBuffer(din.readLine());
    s.reverse();
    System.out.println(s);
  }
}
```

Solution 4.4

Ecrire une fonction qui transforme recopie en transformant les lettres majuscules en lettres minuscules une chaîne de caractères passé en argument de la ligne de commande.

```
import java.io.*;

public class MajMin {
  public static void main(String[] args) throws IOException {
    StringBuffer s = new StringBuffer(args[0]);
    char c;
    for (int i=0; i < s.length(); i++) {
```

```
      c = s.charAt(i);
      if ( (c >= 'A') && (c <= 'Z')) {
        c = (char) ('a' + c - 'A');
        s.setCharAt(i, c);
      }
    }
    System.out.println(s);
  }
}
```

Fiche 5. Entrées-Sorties

Solution 5.1

Programmer une course entre un lièvre et une tortue en faites en sorte que le lièvre gagne.

```
package td.threads;

class Animal extends Thread {
  String qui;
  int delay;
  public Animal(String n, int t) {
    qui = n; delay = t; start();
  }
  public void run() {
    for (int i = 0; i<=20; i++) {
      System.out.print(qui);
      try { sleep(delay); }
      catch ( InterruptedException e) {}
    }
    System.out.print(" " + qui + " est arrivé ");
  }
}

class Course {
  public static void main(String args[]) {
    new Animal("L", 100) ;
    new Animal("T", 500) ;
  }
}
```

Fiche 6. Diverses petites choses

Solution 6.1

Etant donnée la grammaire *BNF* suivante, ecrire un programme permettant de reconnaître si une chaîne de carctères suivie d'un point, lue à la console, est u n mot du langage engendré par cette grammaire.

expression	$\longrightarrow$	*terme operateur-additif express ion* | *terme*
terme	$\longrightarrow$	*facteur operateur-multiplicatif term e* | *facteur*
facteur	$\longrightarrow$	*nombre* | *variable* | '(' *expression* ')'
variable	$\longrightarrow$	'x' | 'y' | 'z'
operateur-additif	$\longrightarrow$	'+' | '-'
operateur-multiplicatif	$\longrightarrow$	'*' | '-'

Modifier l'analyseur précédent de telle sorte que le programme construise l'arbre syntaxique de l'expression analysée.

Ecrire une fonction qui imprime l'expression analysée à partir de l'arbre syntaxiq ue. Attention : pour imprimer correctement l'expression, avec les bons parenthésage s, il faudra utiliser la grammaire.

Solution 6.2

Mêmes questions pour la grammaire :

arbre	$\longrightarrow$	*atome* '(' *liste-d-arbres* ') ' | *atome*
liste-d-arbres	$\longrightarrow$	*atome* ',' *liste-d-a rbres* | *arbre*
atome	$\longrightarrow$	'a' | 'b' | $\cdots$ |'z'

Solution 6.3

Programmer le jeu du solitaire.

Solution 6.4

On dispose de trois de piquets, numérotés 1, 2, et 3 et de n disques toutes de t alles différentes. Au départ, les disuaes sont empilés par taille décroissante sur le piquet numéro 1 . Le problème consiste à dépalcer tous les disques du piquet numéro 1 au piquet numé ro 3 el les empilant par taille décroissante, et ce en respectant les règles de dé placement suivantes :

- On ne peut déplacer qu'un disque à la fois
- un disque ne doit jamais être placé sur un disque de talle plus petite.

Fiche 7. Threads

Solution 7.1

Programmer une course entre un lièvre et une tortue en faites en sorte que le lièvre gagne.

```
package td.threads;

class Animal extends Thread {
  String qui;
  int delay;
  public Animal(String n, int t) {
    qui = n; delay = t; start();
  }
  public void run() {
    for (int i = 0; i<=20; i++) {
      System.out.print(qui);
      try { sleep(delay); }
      catch ( InterruptedException e) {}
    }
    System.out.print(" " + qui + " est arrivé ");
  }
}

class Course {
  public static void main(String args[]) {
    new Animal("L", 100) ;
    new Animal("T", 500) ;
  }
}
```

Solution 7.2

Définir une classe Producteur qui empile, à intervalles réguliers, des messages et une classe Consommateur qui récupère dans la pile un message à intervalles réguliers. Le producteur devra interrompre la production si la pile est pleine et le consommateur devra patienter si le pile est vide. Donner une exemple d'utilisation avec un producteur et 3 consommateurs.

```
package td.threads;

import java.util.Vector;

class Producteur extends Thread {
  static final int MAXQUEUE = 5;
  private Vector messages = new Vector();

  public void run() {
    try {
      int i = 1;
      while (true) prodMessage(i++);
    }
    catch (InterruptedException e) {}
  }

  private synchronized void prodMessage(int i) throws InterruptedException  {
    while (messages.size() == MAXQUEUE)
      wait();
    messages.addElement("Message " + i);
    notify();
    sleep(200);
```

```
    }
    public synchronized String consMessage() throws InterruptedException {
      notify();
      while (messages.size() == 0)
        wait();
      String message = (String)messages.firstElement();
      messages.removeElement(message);
      return message;
    }
  }

  class Consommateur extends Thread {
    String Nom;
    Producteur producteur;
    Consommateur(Producteur p, String s)  { producteur = p; Nom = s;}
    public void run() {
      try {
        while (true) {
  String S = producteur.consMessage();
  System.out.println("Message recu par le " + Nom + " : " + S);
  sleep(300);
        }
      }
      catch (InterruptedException e) {}
    }

    public static void main(String arv[])  {
      Producteur p = new Producteur();
      p.start();
      new Consommateur(p, "Consommateur 1").start();
      new Consommateur(p, "Consommateur 2").start();
    }
  }
```

Solution 7.3

Définir une classe Producteur qui empile, à intervalles réguliers, des messages et une classe Consommateur qui récupère dans la pile un message à intervalles réguliers. Le producteur devra interrompre la production si la pile est pleine et le consommateur devra patienter si le pile est vide. Donner une exemple d'utilisation avec un producteur et 3 consommateurs.

```
package td.threads;

import java.util.Vector;

class Producteur extends Thread {
  static final int MAXQUEUE = 5;
  private Vector messages = new Vector();

  public void run() {
    try {
      int i = 1;
      while (true) prodMessage(i++);
    }
    catch (InterruptedException e) {}
  }

  private synchronized void prodMessage(int i) throws InterruptedException  {
    while (messages.size() == MAXQUEUE)
      wait();
    messages.addElement("Message " + i);
    notify();
    sleep(200);
  }
  public synchronized String consMessage() throws InterruptedException {
    notify();
    while (messages.size() == 0)
      wait();
    String message = (String)messages.firstElement();
    messages.removeElement(message);
    return message;
  }
}
```

```
class Consommateur extends Thread {
  String Nom;
  Producteur producteur;
  Consommateur(Producteur p, String s)  { producteur = p; Nom = s;}
  public void run() {
    try {
      while (true) {
String S = producteur.consMessage();
System.out.println("Message recu par le " + Nom + " : " + S);
sleep(300);
      }
    }
    catch (InterruptedException e) {}
  }

  public static void main(String arv[])  {
    Producteur p = new Producteur();
    p.start();
    new Consommateur(p, "Consommateur 1").start();
    new Consommateur(p, "Consommateur 2").start();
  }
}
```

Solution 7.4

Programmer une applet qui affiche une chaine de caractères en lui faisant changer la couleur régulièrement.

```
package td.threads;

import java.applet.*;
import java.awt.*;

public class ColorText extends Applet implements Runnable {

  Font f = new Font("TimesRoman", Font.BOLD, 48);
  Color  colors[] = new Color[50];
  Thread th;
  public void start() {
    if (th == null) {
      th = new Thread(this);
      th.start();
    }
  }
  public void stop() {
    if (th != null) { th.stop(); th = null; }
  }

  public void run() {
    float c = 0;
    for (int i=0; i<colors.length; i++) {
      colors[i] = Color.getHSBColor(c, (float)1.0, (float)1.0);
      c += .02;
    }

    int i = 0;
    while (true) {
      //      setForeground(colors[i]);
      setForeground(Color.getHSBColor((float)(Math.random()%1.0), (float)(Math.random()%1.0), (float)(Math.random()%1.0)));
      repaint();
      i = (i+1) % colors.length ;
      try { th.sleep(100); }
      catch (InterruptedException e) {}
    }
  }
  public void update(Graphics g) { paint(g); }
  public void paint(Graphics g) {
    g.setFont(f);
    g.drawString("Coucou !!!", 15, 50);
  }
}
```

Solution 7.5

Réaliser une animation graphique dans une applet à partir de la suite des 15 images que vous trouverez à `/home/users/public/Java/tumble`.

```
package td.threads;

import java.applet.*;
import java.awt.*;

public class Tumble extends Applet implements Runnable {

  Image im[] = new Image[32];
  Image  imCourant;
  Image  Ivir;
  Graphics  Gvir;
  Thread th;

  public void init() {
    Ivir = createImage(300,300);
    Gvir = Ivir.getGraphics();
    for (int i=0; i<16; i++) {
      im[i] = getImage(getCodeBase(), "tumble/T"+(i+1)+".gif");
      im[31-i] = getImage(getCodeBase(), "tumble/T"+(i+1)+".gif");
      imCourant = im[0];
    }
  }

  public void start() { if (th == null)  { th = new Thread(this);  th.start(); } }

  public void stop() { if (th != null) { th.stop(); th = null; } }
  public void run() {
    while (true) {
      for (int i=0; i<32; i++) {
try { th.sleep(100); }
catch (InterruptedException e) {}
imCourant = im[i];
repaint();
      }
      try { th.sleep(1000); }
      catch (InterruptedException e) {}
    }
  }
  public void paint(Graphics g) {
    Gvir.drawImage(imCourant, 15, 50, this);
    //g.drawImage(imCourant, 15, 50, this);
    g.drawImage(Ivir, 0, 0, this);
  }
}
```

-

Fiche 8. Applet et Programmation graphique

Solution 8.1

Réaliser une applet contenant trois boutons et associer une page HTML diiférente à chaque bouton de manière à pouvoir afficher la bonne page en fonction du blouton cliqué.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
public class Toc extends Applet implements ActionListener {
  public void init() {
    Button b1, b2, b3;
    add( b1 = new Button("Page1"));
    add( b2 = new Button("Page2"));
    add( b3 = new Button("Page3"));
    b1.addActionListener(this);
    b2.addActionListener(this);
    b3.addActionListener(this);
  }
  public void actionPerformed(ActionEvent e) {
    URL url = null;
    try {
      url = new URL(getCodeBase(), e.getActionCommand()+".html");
      System.out.println(">>> " + url);
      getAppletContext().showDocument(url);
      getAppletContext().showStatus(""+url);
    }
    catch ( MalformedURLException ex ) { System.out.println("??? " + url);}
  }
}
```

Solution 8.2

Réaliser une applet qui affiche une image qui suit les mouvements de la souris et qui prend sa place définitive lorsqu'on la relâche.

Solution 8.3

Réaliser une applet qui affiche un nouveau carré de couleur aléatoire chaque fois que l'on clique sur la souris. Si le clic survient sur un carré déjà crée, alors ce dernier suivra le déplacement de la souris tant qu'elle est appuyée.

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

public class Carre extends Applet {
  public Carre() {
    setLayout(null);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
   add(new PetitCarre(e.getX(), e.getY(), Color.red));
   repaint();
      }
```

```
    });
  }
  public static void main(String[] args) {
    Frame f = new Frame("Des petits carrés...");
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    f.add(new Carre());
    f.setSize(350, 170);
    f.show();
  }
}

final class PetitCarre extends Component {
  PetitCarre(int x, int y, Color color) {
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
   int bx = e.getX() + getLocation().x;
   int by = e.getY() + getLocation().y;
   setLocation(bx, by);
      }
    });
    setBounds(x, y, 32, 32);
    setBackground(color);
  }
  public void paint(Graphics g) {
    super.paint(g);
    g.setColor(getBackground());
    Dimension d = getSize();
    g.fill3DRect(0, 0, d.width-1, d.height-1, true);
  }
}
```

Version avec une classe imbriquée

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

public class Carre extends Applet {
  public Carre() {
    setLayout(null);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
   add(new PetitCarre(e.getX(), e.getY(), Color.red));
   repaint();
      }
    });
  }
  public static void main(String[] args) {
    Frame f = new Frame("Des petits carrés...");
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    f.add(new Carre());
    f.setSize(350, 170);
    f.show();
  }
}

final class PetitCarre extends Component {
  PetitCarre(int x, int y, Color color) {
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
   int bx = e.getX() + getLocation().x;
   int by = e.getY() + getLocation().y;
   setLocation(bx, by);
      }
    });
    setBounds(x, y, 32, 32);
    setBackground(color);
  }
  public void paint(Graphics g) {
    super.paint(g);
```

```
        g.setColor(getBackground());
        Dimension d = getSize();
        g.fill3DRect(0, 0, d.width-1, d.height-1, true);
      }
    }
```

Solution 8.4

Réaliser une applet qui affiche un clavier à l'écran et indique (sur une image de clavier) les touches enfoncées par l'utilisateur.

Solution 8.5

Réaliser un petit éditeur de texte avec les fonctionnalités suivantes : *copier*, coller, effacer, insérer un élement d'un glossaire.

Solution 8.6

Reprendre l'exemple **??** en utilisant ? ? ? pour stocker le dessin réalisé par l'utilisateur, une image.

Solution 8.7

Réaliser une animation qui déplace dans une nuit étoilée un appareil volant.

Fiche 9. Programmation réseau

Solution 9.1

Programmer une course entre un lièvre et une tortue en faites en sorte que le lièvre gagne.

```
package td.threads;

class Animal extends Thread {
  String qui;
  int delay;
  public Animal(String n, int t) {
    qui = n; delay = t; start();
  }
  public void run() {
    for (int i = 0; i<=20; i++) {
      System.out.print(qui);
      try { sleep(delay); }
      catch ( InterruptedException e) {}
    }
    System.out.print(" " + qui + " est arrivé ");
  }
}

class Course {
  public static void main(String args[]) {
    new Animal("L", 100) ;
    new Animal("T", 500) ;
  }
}
```

Neuvième partie

Java : Travaux pratiques

Problème 1. Résolution d'équations sur les arbres

lkjlkjdlkj lksj lksdj lkjsdf

Problème 2. Un petit BBS

lkjlkjdlkj lksj lksdj lkjsdf

Problème 3. Un petit browser HTTP

lkjlkjdlkj lksj lksdj lkjsdf

Problème 4. Accès aux bases de données

lkjlkjdlkj lksj lksdj lkjsdf

Dixième partie

Java : Corrigés des travaux pratiques

Problème 1. Résolution d'équations sur les arbres

Solution

lkjlkjdlkj lksj lksdj lkjsdf

```
\chapter*{Problème \thenumprob. Résolution d'équations sur les arbres}

\begin{sol}
lkjlkjdlkj lksj lksdj lkjsdf

\solution{Programmes/Projets/Unif.tex}
\end{sol}









%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%% Local Variables:
%%% mode: latex
%%% TeX-master: "../../poly"
%%% End:
```

Problème 2. Un petit BBS

Solution

lkjlkjdlkj lksj lksdj lkjsdf

```
\chapter*{Problème \thenumprob. Un petit BBS}

\begin{sol}
lkjlkjdlkj lksj lksdj lkjsdf

\solution{Programmes/Projets/Bbs.tex}
\end{sol}









%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%% Local Variables:
%%% mode: latex
%%% TeX-master: "../../poly"
%%% End:
```

Problème 3. Un petit browser HTTP

Solution

lkjlkjdlkj lksj lksdj lkjsdf

```
\chapter*{Problème \thenumprob. Un petit browser HTTP}

\begin{sol}
lkjlkjdlkj lksj lksdj lkjsdf

\solution{Programmes/Projets/Http.tex}
\end{sol}









%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%% Local Variables:
%%% mode: latex
%%% TeX-master: "../../poly"
%%% End:
```

Problème 4. Accès aux bases de données

Solution

lkjlkjdlkj lksj lksdj lkjsdf

```
\chapter*{Problème \thenumprob. Accès aux bases de données}

\begin{sol}
lkjlkjdlkj lksj lksdj lkjsdf

\solution{Programmes/Projets/Bd.tex}
\end{sol}









%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%% Local Variables:
%%% mode: latex
%%% TeX-master: "../../poly"
%%% End:
```

Index

- -classpath, 68
- *fully qualified name*, 67
- `CLASSPATH`, 68
- `StringBuffer`, 80
- `String`, 80
- `abstract`, 59
- `byte`, 24
- `char`, 24
- `double`, 24
- `false`, 24
- `float`, 24
- `implements`, 64
- `int`, 24
- `javac`, 43
- `java`, 43
- `long`, 24
- `null`, 26
- `private`
  - interface, 64
- `protected`
  - interface, 64
- `synchronized`
  - interface, 64
- `transcient`
  - interface, 64
- `true`, 24
- `volatile`
  - interface, 64
- évènement
  - émetteur, 192
  - modèle, 191
  - objets, 191
  - récepteur, 192
  - source, 192
- évènements, 191
- évènements graphiques, 186

- abstraite, 59
- ActionEvent, 192, 198
- ActionListener, 192, 193, 198
- AdjustmentEvent, 212
- AdjustmentListener, 193, 212
- adresse internet, 337
- affectation, 30
- allocation dynamique, 47
- applet, 185, 186
  - cycle de vie, 178
  - destroy, 178
  - html, 177, 179
  - init, 178
  - paramètre, 177, 179
  - start, 178
  - stop, 178

- bouton, 185

- caractère
  - `\"`, 26
  - `\'`, 26
  - `\\`, 26
  - `\a`, 26
  - `\b`, 26
  - `\f`, 26
  - `\n`, 26
  - `\r`, 26
  - `\t`, 26
  - constante, 26
  - Séquences d'échapemment, 26
- CGI, 339
- chaîne
  - concaténation, 26
  - constante, 26
- chaîne de caractères, 80
- champ
  - `private protected`, 69
  - `private`, 69
  - `protected`, 69
  - `public`, 69
  - accessibilité, 69
  - non qualifiée, 69
  - static, 63
- champs, 45
  - `protected`, 69
  - constructeur
    - redéfinition, 57
  - final, 63
  - initialisation, 48, 49
  - private, 45
  - protected, 45
  - public, 45
  - redéfinition, 57
  - static, 48
- classe
  - `abstract`, 59
  - `final`, 59
  - `private`, 69
  - `public`, 69
  - abstraite, 59
  - accessibilité, 69
  - adaptateur, 195
  - Adapter, 195
  - conversion, 59
  - imbriquée, 91, 196
  - inner, 91, 196
  - non qualifiée, 69
  - non static, 91
  - static, 91
- classes, 45

champs, 45
static, 48
méthodes, 46
variables d'instances, 45
variables de classes, 48
Client/Serveur, 337
collisions de noms, 70
commentaires, 23
documentation, 23
lignes, 23
multilignes, 24
ComponentAdapter, 196
ComponentEvent, 205
ComponentListener, 193, 205
composant graphique, 185
constante, 25
`false`, 25
`null`, 26
`true`, 25
booléenne, 25
caractère, 26
chaîne, 26
double, 25
entière, 25
float, 25
flottant, 25
String, 26
constructeur par défaut, 56
Constructeurs, 47
ContainerAdapter, 196
ContainerEvent, 206
ContainerListener, 193, 206
conversion, 87

décimale, 25
Destructeur, 48
DNS, 337
double
constante, 25

effet de bord, 29
encapsulation des données, 45
entrées, 97
Entrées-Sorties, 97
event, 186, 191
exception
`ClassCastException`, 59
expression, 29
erreur d'évaluation, 30
exception, 30
ordre d'évaluation, 30
priorité, 30
type, 30

FilterInputStream, 104
FilterOutputStream, 104
FilterReader, 104
FilterWriter, 104
finalize, 48
float
constante, 25
flots de données, 97
flottant
NaN, 31
overflow, 31
underflow, 31
constante, 25
FocusAdapter, 196
FocusEvent, 207
FocusListener, 193, 207

garbage collecteur, 48
gestion d'évènements, 186
Gestion de placement, 186
Gestion des évènements, 191

héritage multiple, 63
heritage, 55
hexadécimale, 25
HTTP, 339

identificateur, 24
identificateurs réservés, 24
InetAddress, 340
initialisation
champs
non static, 49
static, 49
inner class, 91
InputMethodEvent, 213
InputMethodListener, 213
instruction, 39
`break`, 40
`case`, 40
`default`, 40
`do ... while`, 41
`else`, 39
`for`, 42
`if`, 39
`switch`, 40
`while`, 41
étiquette, 42
étude de cas, 40
bloc, 39
break, 42
conditionelle, 39
continue, 42
itération, 41
return, 42
interface
`membre`, 64
`public`, 64
dérivée, 65
méthode, 63
membre, 63
redéfinition, 65
interfaces, 63
IP, 335
ItemEvent, 210
ItemListener, 193, 210

java.awt.button, 185
java.awt.event.ActionEvent, 192, 198
java.awt.event.ActionListener, 192, 193, 198
java.awt.event.AdjustmentEvent, 212
java.awt.event.AdjustmentListener, 193, 212
java.awt.event.ComponentAdapter, 196
java.awt.event.ComponentEvent, 205
java.awt.event.ComponentListener, 193, 205
java.awt.event.ContainerAdapter, 196

java.awt.event.ContainerEvent, 206
java.awt.event.ContainerListener, 193, 206
java.awt.event.FocusAdapter, 196
java.awt.event.FocusEvent, 207
java.awt.event.FocusListener, 193, 207
java.awt.event.InputMethodEvent, 213
java.awt.event.InputMethodListener, 213
java.awt.event.ItemEvent, 210
java.awt.event.ItemListener, 193, 210
java.awt.event.KeyAdapter, 196
java.awt.event.KeyEvent, 201
java.awt.event.KeyListener, 193, 201
java.awt.event.MouseAdapter, 196
java.awt.event.MouseEvent, 193, 198, 199
java.awt.event.MouseListener, 193, 198
java.awt.event.MouseMotionAdapter, 196
java.awt.event.MouseMotionListener, 193, 199
java.awt.event.TextEvent, 209
java.awt.event.TextListener, 193, 209
java.awt.event.WindowAdapter, 196
java.awt.event.WindowEvent, 203
java.awt.event.WindowListener, 193, 203
java.awt.textfield, 185
java.io, 97
java.io.InputStream, 97
java.io.OutputStream, 97
java.io.Reader, 97
java.io.Writer, 97
java.net.InetAddress, 340
java.net.URL, 340
java.util.EventObject, 191
jeu de caractères Unicode, 23

KeyAdapter, 196
KeyEvent, 201
KeyListener, 193, 201

layout, 186
layout manager, 186

méthode
  abstract, 59, 63
  private protected, 69
  private, 69
  protected, 69
  public, 63, 69
  abstraite, 59
  accessibilité, 69
  finalize, 48
  non qualifiée, 69
  passage de paramètres, 50
  polymorphisme, 50
  signature, 50
méthodes, 46
  final, 59
  destructeur
    redéfiniton, 58
  private, 46
  protected, 46
  public, 46
  redéfiniton, 57
  static
    redéfiniton, 58
MARGINPAR, 63, 85, 87–89, 109, 110, 113, 116, 120–122, 131, 146, 153, 154, 160, 167, 168, 176, 181, 182, 198, 200–203, 205, 206, 208–210, 212–214, 246, 261, 272, 275, 284, 285, 287, 303, 310, 314, 339, 340, 342, 351, 354, 360, 361, 366, 368, 370, 371, 374–376
membre
  private, 64
  protected, 64
mots clés, 24
  this, 49
MouseAdapter, 196
MouseEvent, 193, 198
MouseListener, 193, 198
MouseMotionAdapter, 196
MouseMotionEvent, 199
MouseMotionListener, 193, 199

new, 47
notation
  décimale, 25
  hexadécimale, 25
  octale, 25
numéro de port, 338

Object
  Serialization, 117
Objet
  écriture, 117
  lecture, 117
  Serialization, 117
objets, 45
  référence vers, 45
octale, 25
opérateur, 29
  cast, 35
  instanceof, 36
  affectation, 30
  affectation binaire, 35
  arithmétique, 31
    additif, 32
    multiplicatif, 31
    post décrément, 33
    post incrément, 33
    pré décrément, 33
    pré incrément, 33
  changement de type, 35
  comparaison, 33
  concaténation, 33
  conditionnel, 35
  logique, 33
  manipulation de bits, 34
  new, 47
  ordre d'évaluation, 30
  priorité, 30

package
  import, 67
  accessibilité, 69
  Convention de nommage, 70
  sans nom, 68
packages, 67
paquets, 337
polymorphisme, 50
port, 338
protocole, 339

récupération de mémoire, 48

référence, 45, 47
réseau, 335

Séquences d'échapemment, 26
Serialization, 117
signature, 50
socket, 342
sorties, 97
streams, 97
String
 concaténation, 26
 constante, 26
structures de contrôle, 39

tableau, 79
 `IndexOutOfBoundsException`, 79
 accès, 79
 initialisation, 80
 test de débordement, 79
TCP, 335
TextEvent, 209
TextListener, 193, 209
this, 49
type
 `byte`, 24
 `char`, 24
 `double`, 24
 `float`, 24
 `int`, 24
 `long`, 24
 `null`, 26
 booléen, 24
 conversion, 87
 entier, 24
 flottant, 24
 primitif, 24
 promotion, 87
 référence, 24

UDP, 335
unicode, 23
URL, 338, 340
URL JDBC, 361

variable, 26
 globale, 27
 initialisation, 26
 locale, 27
 référence, 27
 type, 27

widgets, 185, 188
WindowAdapter, 196
WindowEvent, 203
WindowListener, 193, 203