Eléments sur JAVA

Philippe Laroque
<laroque@u-cergy.fr>

$Id : java.lyx 1602 2008-11-26 10 :04 :49Z phil $

Résumé

Ce petit document ne se substitue pas à l'un des - nombreux - ouvrages disponibles sur JAVA. Il prétend simplement servir d'aide-mémoire à l'utilisateur pour retrouver rapidement le contexte d'utilisation d'une notion importante du langage.

1 Introduction

JAVA a vu le jour en 1995, comme résultat d'un projet interne SUN de création d'un langage de haut niveau. Le projet stipulait que le langage à créer devrait être portable, facile à apprendre et suffisamment ouvert pour permettre son extension aisée.

L'environnement JAVA disponible actuellement est la troisième version de ce langage. Sa richesse, en termes de domaines fonctionnels couverts, est telle qu'il est exclus de vouloir la couvrir en totalité dans un cours de quelques heures. Nous nous bornerons ici à faire ressortir les propriétés principales du langage :

- sa portabilité, avec le modèle de machine virtuelle et une brève description des outils qui l'entourent au sein du JDK (*Java Development Kit*) ;
- son orientation "objet", axe que nous développerons plus particulièrement puisqu'il est le socle sur lequel viennent s'appuyer tous les modules du langage ;
- sa facilité d'apprentissage, du moins si on connaît déjà les langages C et/ou C++

1.1 Un premier exemple de programme JAVA

Ce programme affiche simplement une chaîne de caractères sur l'écran :

```
/* FirstExample.java
 * un programme simple pour afficher une chaine de caracteres
 */
public class FirstExample {
  public static void main(String args[]) {
    System.out.println("premier programme JAVA") ;
    System.exit(0) ;
  }
}
```

Quelques remarques générales, avant d'entrer dans les détails dans les sections qui suivent :

- la ressemblance avec le langage C est importante, on verra qu'en fait JAVA est (presque) un sur-ensemble de C en ce qui concerne la syntaxe de base ;
- tout code doit appartenir à une classe (principe d'encapsulation, voir section 2.1.2 p. 6) ;
- toute classe peut définir ce qu'elle publie et ce qu'elle garde caché (mot réservé `public` de l'exemple) ;
- toute instruction (sauf les affectations) est un envoi de message (ici, à l'objet prédéfini `System`).

1.2 Quelques références

- Un tutorial de SUN, complet et assez bien fait :
  `http://java.sun.com/docs/books/tutorial`
- Des sites sur les langages objet (et plus spécialement JAVA) :
  `http://www.sigs.com`
  `http://www.javaworld.com`
- Un livre en ligne :
  `http://www.mindviewinc.com/`
- Un "How-To" très bien fait :
  `http://www.rgagnon.com/howto.html`
- Des exemples de code (en français) :
  `http://cui.unige.ch/java/`
- Le site "bible" de l'objet en général :
  `http://www.cetus-links.org`

1.3 L'environnement de travail

L'objectif principal du langage est la portabilité. Un programme JAVA doit donc être indépendant de la plate-forme matérielle et logicielle sur laquelle il s'exécute. Pour atteindre ce but, les concepteurs du langage ont réalisés deux outils qui constituent la base des environnements de développement JAVA :

- Une machine virtuelle, couche logicielle intermédiaire entre le système d'exploitation et les applications écrites en JAVA.

– Un compilateur (ou "semi-compilateur"), générant du code à destination de la machine virtuelle.

1.3.1 La machine virtuelle JAVA

Cette couche isole les programmes JAVA des contraintes de l'environnement local, assurant ainsi leur portabilité. Pour qu'un système puisse faire tourner des programmes JAVA (quels qu'ils soient), il faut et il suffit que cette machine virtuelle (que dans la suite nous nommerons JVM, pour Java Virtual Machine) ait été portée sur le système en question.

A l'heure actuelle, la JVM est portée sur la totalité des plate-formes "classiques" de développement, du mac au gros serveur UNIX en passant par les diverses déclinaisons de windows, linux etc.

La JVM est activée

– soit explicitement, à l'aide de la commande `java` suivie du programme qu'on souhaite exécuter (le nom d'une classe contenant une méthode `main`),
– soit implicitement dans un environnement plus riche qui la contient (par exemple, déclenchement automatique d'une applet dans un navigateur web).

1.3.2 Le compilateur

Il s'agit en fait d'un semi-compilateur, dans la mesure où il ne génère pas du langage machine (qui serait dépendant de la plate-forme, donc non portable), mais un code intermédiaire portable appelé *bytecode*.

Le compilateur est activé explicitement, en utilisant la commande `javac` suivie du ou des fichiers sources à compiler.

1.3.3 Le "documenteur"

Les concepteurs de JAVA ont eu l'initiative de proposer un mécanisme permettant d'extraire, *dans le code source lui-même*, certains des commentaires qui s'y trouvent, puis de les traiter afin d'en produire une documentation au format HTML, avec liens hypertextuels.

Tout constituant d'un programme (paquetage, classe, méthode, variable) peut être documenté. Pour les détails, nous renvoyons le lecteur à

`http ://java.sun.com/j2se/javadoc/index.html`

mais nous décrivons tout de même ici les grands principes d'utilisation de cet outil, appelé `javadoc`.

Les commentaires destinés à Javadoc commencent par "/**" et se terminent par "*/". Tout ce qui est situé à l'intérieur sera interprété comme du HTML, à l'exception de quelques mots réservés dont nous donnons plus loin un sous-ensemble fréquemment utilisé.

Tout mot réservé doit apparaître en début de ligne, exemple :

```
/**
 * This is a <em>doc</em> comment.
 * @see java.lang.Object
 */
```

Les commentaires relatifs à une classe (*resp.* une méthode, une variable etc.) doivent apparaître juste au-dessus de la définition de la classe (*resp.* méthode, variable, etc.)

Quelques mots réservés utiles sont réunis dans le tableau ci-dessous. Tous commencent par "@". Pour chacun d'eux, on a indiqué la version du JDK à partir de laquelle ils sont définis.

mot-clé	version	signification
`@author`	1.0	l'auteur du fichier
`@deprecated`	1.0	code appelé à disparaître
`@exception`	1.0	code lançant une exception
`@param`	1.0	paramètre de méthode
`@return`	1.0	valeur de retour
`@see`	1.0	lien vers un autre code
`@since`	1.1	ancienneté du code
`@version`	1.0	version du code/document

Si les sources sont organisées de manière arborescente pour respecter la hiérarchie des paquetages, il est possible de générer en une fois la totalité de la documentation de cette hiérarchie, avec la commande suivante (on suppose qu'on est placé à la racine de cette hiérarchie et que la documentation doit être placée dans $HOME/doc) :

```
javadoc -version -author -encoding UTF-8 \
-d $HOME/doc -sourcepath . pkg1 pkg2 ... pkgn
```

Les extraits de code Java qui suivent dans ce petit support fournissent des exemples d'utilisation des "tags" listés dans la tableau ci-dessus.

1.3.4 Autres outils

- `appletviewer` : outil de test permettant de lancer des applets sans navigateur web ;
- `jar` : outil permettant de placer dans un même fichier "archive" des fichiers concourant au fonctionnement d'une application, ce qui facilite leur distribution (et leur exécution distante) ;
- `jdb` : debugger assez basique, très nettement amélioré par les divers IDE du marché ;
- `javah` : générateur de fichiers en-tête C pour l'intégration de méthodes natives ;
- `javap` : désassembleur de programmes JAVA
- ...

1.4 Syntaxe de base

Nous ne ferons ici qu'un très bref passage sur la syntaxe de base de Java, dans la mesure où elle est identique à celle de C ou de C++ (en particulier, le langage est *case-sensitive*).

Dans la section 3 p. 10, on reprend en détail les éléments de syntaxe propres à JAVA.

1.4.1 Commentaires

Il y a trois types de commentaires :

- Les commentaires introduits par `//` se terminent implicitement en fin de ligne
- Les commentaires introduits par `/*` se terminent par `*/`. Ils peuvent donc s'étendre sur plusieurs lignes.
- Les commentaires introduits par `/**` sont identiques aux précédents, mais ils sont en outre récupérés par javadoc (voir 1.3.3 p. 3).

1.4.2 Types prédéfinis

Les seuls types prédéfinis en JAVA sont des types simples. Tous les autres types sont des classes, sur lesquelles on revient plus loin. Les types simples de JAVA sont les mêmes que ceux de C, à l'exception du type `boolean` qui n'existe pas en C. Le tableau suivant résume tout ceci :

Type	contenu	défaut[1]	taille	min/max
boolean	true/false	false	1 bit	
byte	signé	0	1 octet	$-128/127$
char	unicode	\u0000	2 octets	\u0000/\uFFFF
short	signé	0	2 octets	$-32768/32767$
int	signé	0	4 octets	$\approx 2 * 10^9$
long	signé	0	8 octets	$\approx 9 * 10^{18}$
float	IEEE754	0.0	4 octets	$\approx 3 * 10^{38}$
double	IEEE754	0.0	8 octets	$\approx 4 * 10^{308}$

1.4.3 Déclarations

Le typage de JAVA est statique : une variable doit être déclarée avant de pouvoir être accédée. Cette déclaration se fait en mentionnant son type et son nom (et en lui donnat éventuellement une valeur initiale).

ex :

```
int i ;
double d = 3.1415926535 ;
char c = 'A', d='\u0108' ;
```

Il y a une exception à cette règle : à l'intérieur d'une classe (voir 2.1.3 p. 7), une variable d'instance peut être utilisée dans une méthode déclarée avant la déclaration de la variable elle-même.

1.4.4 Les tableaux

Leur taille ne peut être fixée que dynamiquement. Pour déclarer un tableau dont les éléments sont de type `T`, on ajoute une paire de crochets après le nom de la variable. La création effective du tableau se fait par allocation dynamique de mémoire, grâce à l'opérateur `new` :

```
int monTableau[] = new int[100] ; // crée un tableau de 100 entiers
monTableau[0]=3 ;                 // les indices commencent à 0
```

On peut aussi créer un tableau en donnant son contenu en extension ; la taille du tableau (attribut `length`) est alors déterminée par le nombre d'éléments mentionnés :

```
int[] anArray = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000} ;
```

1.4.5 Les opérateurs

Tous ceux du C, à part les opérateurs sur les pointeurs. En effet, bien que JAVA ne manipule pratiquement que des pointeurs, ces derniers ne sont pas accessibles au programmeur. En particulier, il n'y a pas de moyen de récupérer l'adresse d'un objet.

Il y a trois grandes catégories d'opérateurs, celles de C :

- les opérateurs arithmétiques (`+, -, *, /, %, +=, -=, ++, --`, etc.)
- les opérateurs logiques (`!, &&, ||`)
- les opérateurs de bas niveau (`&,|, ^, ~`)

Un opérateur spécial, `instanceof`, permet de tester l'appartenance d'un objet à une classe (ou à une de ses sous-classes). L'objet constant `null` n'appartient à aucune classe, par construction.

1.4.6 Les structures de contrôle

Identiques à celles de C (`if, while, for, switch`). Un exemple :

```
for (int i = 0 ; i < n-1 ; i++) {
  for (int j = i+1 ; j < n ; j++ {
    if (tab[i] > tab[j]) {
      int tmp = tab[j] ;
      tab[j] = tab[i] ;
      tab[i] = tmp ;
    }
  }
}
```

1.4.7 Les méthodes

Là encore, syntaxe presque identique aux fonctions C : on fournit le type de retour, le nom, la liste des paramètres et le corps de la fonction. Exemple :

```
void push(Object o) {
  Node n = new Node(o,top) ;
  top = n ;
}
```

À signaler : contrairement à ce qui se fait en C, on peut donner le même nom à deux versions différentes de la même méthode, pourvu que ces deux versions diffèrent soit par le nombre de paramètres, soit par le type d'un des paramètres. On parle dans ce cas de *surcharge*.

2 L'approche objet

Cette approche a pour but de répondre aux reproches adressés à l'approche algorithmique descendante classique ("algorithme + structures de données = programme", *D. Knuth*). Elle prône un regroupement de l'aspect statique (données) et dynamique (comportement) de toute entité importante d'un système au sein d'un objet. Elle repose sur trois propriétés fondamentales, que doit posséder tout système (langage, BD, etc.) dit "à objets" :

- Abstraction et encapsulation de données
- Héritage de types (en général *via* des classes)
- Polymorphisme du comportement

2.1 Abstraction et encapsulation de données

2.1.1 Abstraction

On parle d'abstraction de données lorsqu'il est possible de manipuler des types de données sans en connaître la structure précise, uniquement en utilisant une liste documentée de fonctions et de procédures dédiées à ce type de données.

Tout langage de programmation permettant la définition de types structurés, et comprenant la notion de fonction, permet donc de faire de l'abstraction de données. C'est le cas en particulier de PASCAL, de C, d'ADA qui ne sont pas réellement des langages à objets.

La paternité de cette notion est à chercher dans les types abstraits, où seul le mode d'emploi du type est décrit (formellement), les choix d'implémentation n'étant pas indiqués. La définition d'un type abstrait se compose

- de la mention de types abstraits déjà définis sur lesquels repose la définition du nouveau type (constituants et/ou paramètres) ;
- d'une liste d'opérateurs avec leur profil (domaine de définition, type du résultat) ;
- d'une liste de conditions d'utilisation des opérateurs ;
- d'une liste d'axiomes donnant une sémantique à ce qui ne serait autrement qu'une définition syntaxique du type.

Exemple de définition d'un tel type abstrait :

```
Type Pile
paramètres élément
utilise booléen
opérateurs
        estVide : Pile -> booléen
        pileVide : -> Pile
        empiler : Pile X élément -> Pile
        dépiler : Pile -> Pile
        sommet : Pile -> élément
conditions :
        dépiler ssi pilevide = faux
        sommet ssi pilevide = faux
axiomes :
        depiler(empiler(p,e)) = p
        sommet(empiler(p,e)) = e
        estVide(pileVide) = vrai
        estVide(empiler(p,e)) = faux
```

Il est clair que les avantages de cette approche résident dans la souplesse et la maintenabilité : le concepteur d'un type peut modifier la structure interne (l'implémentation) du type sans que les programmes clients (ceux qui utilisent le type) ne nécessitent de modification. Ceci bien entendu si chacun respecte les règles du jeu, le concepteur en fournissant une liste consistante et complète d'opérateurs, l'utilisateur en ne cherchant pas à tirer parti des connaissances qu'il peut avoir des choix d'implémentation, par exemple à des fins de performance...

2.1.2 Encapsulation

La notion d'encapsulation est un peu plus forte que celle d'abstraction, puisqu'on dispose alors d'un moyen permettant au créateur du type d'empêcher l'accès aux choix d'implémentation, afin d'être certain

que seule l'interface proposée pour le type sera utilisée. On trouve encore des langages non objet qui supportent cette caractéristique, citons en particulier ADA 83.

2.1.3 Support d'implémentation : la classe

Le support de l'encapsulation dans un langage à objets est la classe. Le tableau suivant montre l'analogie (forcément réductrice) que l'on peut faire entre les concepts du monde des langages structurés et les concepts "objet" :

Monde structuré	Monde Objet
	objet
type	classe
variable	instance
appel fonctionnel	envoi de message
fonction	méthode

On remarque que le terme "objet" n'a pas de correspondant dans le monde structuré, ce qui est normal puisque c'est le concept fondateur du modèle. On peut faire une définition *a posteriori* de ce terme en disant que *tout objet est instance d'une classe.*

Une classe est définie par la donnée de deux composantes :

- Une composante statique, définissant la structure commune à tous les objets appartenant à la classe. Elle est composée d'attributs, éventuellement typés, appelés champs, variables d'instance ou données membres suivant le contexte. Ces trois termes seront synonymes dans la suite du document.
- Une composante dynamique, définissant l'ensemble des services pouvant être demandés aux instances. Ces services prennent le nom de méthodes, ou fonction membres, ou primitives. Là encore, ces termes seront synonymes dans la suite du document.

En vertu du principe d'encapsulation, les attributs ne seront en général pas directement accessibles des programmes clients : il faudra que ceux-ci utilisent les méthodes pour y accéder.

On distingue deux visions de la classe, en extension (comme l'ensemble des objets qui lui sont liés par la relation d'instanciation) et en intension (comme l'ensemble des propriétés partagées par ses instances).

2.2 Héritage

L'héritage est une relation entre classes (alors que l'instanciation est une relation entre une classe et un objet), c'est-à-dire en termes mathématiques un sous-ensemble du produit cartésien de l'ensemble des classes du système par lui-même.

Elle est définie par trois propriétés : on dit qu'une classe D hérite (ou dérive) d'une classe B si

1. l'ensemble des attributs de B est inclus dans l'ensemble des attributs de D
2. l'ensemble des méthodes de B est inclus dans l'ensemble des méthodes de D
3. le langage considère que D est *compatible avec* B, dans le sens où toute instance de D est admise comme étant un (cas particulier de) B dans un programme quelconque.

Si D dérive de B, on pourra dire de B qu'elle est la classe-mère, ou la superclasse, ou la surclasse, ou encore la classe de base de D. Tous ces termes seront synonymes dans la suite du document.

Une classe dérivée n'a pas à mentionner les attributs et méthodes de sa classe mère : ceux-ci sont implicitement présents. On fait donc une définition incrémentale de la nouvelle classe, en ne mentionnant que ce qui la différencie de sa (ses) classe(s) mère(s). Lorsqu'un objet reçoit une demande de service, si la méthode correspondante est définie dans sa classe elle est exécutée, sinon la classe mère est consultée, et ainsi de suite jusqu'à trouver la méthode dans l'une des classes du sur-graphe de la classe initiale. La méthode est alors exécutée. On peut aussi arriver sur une classe qui n'a pas de surclasse directe. Une procédure d'erreur dépendant du type de langage est alors déclenchée.

De plus, on ne peut faire d'héritage sélectif, la relation est de type "tout ou rien".

On parlera d'héritage simple quand toute classe du système a au plus une surclasse directe, sinon on parlera d'héritage multiple.

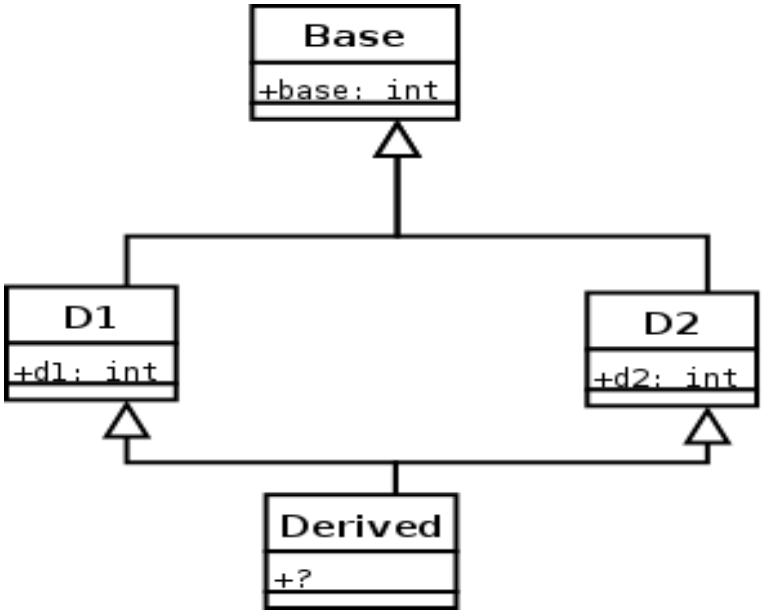


FIG. 1 – Le problème du "losange"

2.2.1 Propriétés mathématiques de l'héritage

La relation d'héritage est *réflexive* (toute classe est une extension triviale d'elle-même), *transitive* (si D dérive de C et si C dérive de B, alors on peut dire que D dérive de B) et *antisymétrique* (si A dérive de B et si B dérive de A, alors A et B sont une seule et même classe), c'est donc une relation d'ordre, ce que nous ne montrerons pas ici. C'est cependant une relation d'ordre partiel, car il existe des classes non comparables.

2.2.2 Héritage simple *vs* héritage multiple

Le graphe d'héritage est orienté (la relation n'est visiblement pas symétrique), sans cycle : c'est un treillis dans le cas général. Dans le cas de l'héritage simple, on se ramène à un (ensemble d') arbre(s). Ce cas est intéressant dans la mesure où le processus de recherche de la méthode dans le sur-graphe est alors simple : cette recherche n'est pas ambiguë. Dans le cas contraire, deux stratégies sont couramment utilisées :

- la première privilégie la sécurité du logiciel, elle interdit donc purement et simplement l'héritage multiple si un conflit (une ambiguïté) apparaît. C'est au concepteur de régler ces conflits, en général en redéfinissant dans la classe dérivée tous les services ambigus, pour que la classe puisse être définie ;
- la seconde privilégie la puissance de représentation du langage, elle consiste à fournir un processus partiellement automatique de linéarisation du graphe d'héritage en cas de conflit : on obtient alors un ordre total sur le sur-graphe, et l'ambiguïté disparaît.

La première approche concerne une grande majorité de langages, notamment tous les langages à objets de grande diffusion dans le monde industriel, y compris JAVA (dans une certaine mesure, voir la notion d'interface plus loin).

Les conflits de noms de service ne sont pas le seul problème lié à l'utilisation de l'héritage multiple : la sémantique de l'inclusion multiple d'une même classe de base pose également une question, dans la mesure où il n'y a pas unicité de la solution au problème : les langages qui implémentent l'héritage multiple ont en général leur propre "solution" à ce problème, parfois appelé "problème du losange" (voir figure 1) : Si la classe `Derived` hérite de `D1` et `D2`, qui sont deux classes dérivées de `Base`, quelle est la structure exacte de `Derived` ? Plus précisément, y a-t-il une ou deux occurrences de la structure de `Base` dans `Derived` ?

2.3 Polymorphisme

Littéralement, le polymorphisme désigne la capacité à prendre plusieurs formes. Ramené au contexte de l'objet, il s'agit d'une propriété qui va permettre à l'émetteur d'un message de l'envoyer à son destinataire sans connaître avec précision la nature de ce destinataire.

L'intérêt de cette notion vient de ce que l'héritage introduit dans un langage, même s'il est typé, des situations où l'on n'a pas à la compilation les informations nécessaires à la détermination de la classe d'appartenance d'un objet : les classes dérivées étant compatibles avec leur classe de base, une instance d'une classe dérivée peut être fournie lorsque le programme a besoin d'une instance de la classe de base. Dans ce cas, l'objet en question n'est plus traité que comme une instance de la classe de base, on a perdu l'information sur son type précis.

Pour parer à ce problème, on a recours à un mécanisme dit de liaison dynamique. Ce mécanisme est ainsi nommé parce qu'il retarde au moment de l'exécution le choix de la méthode à déclencher lors de la réception d'un message.

Prenons un exemple : de la classe `FormeGeometrique` on dérive les classes `Cercle` et `Rectangle`. On crée alors une classe Figure, qui contient différentes formes géométriques. On aura dans cette classe une méthode pour ajouter une nouvelle forme dans la figure. Cette méthode ressemblera, pour sa signature, à

```
ajouter : Figure x FormeGeometrique ->
```

C'est-à-dire que l'objet qu'on ajoute est officiellement une forme géométrique quelconque, même si dans la pratique (*i.e.* au moment de l'appel à cette méthode) on n'ajoutera que des cercles ou des rectangles. Si maintenant on définit dans la classe `Figure` une méthode `dessiner()`, qui demande à chaque forme de la figure de se dessiner, et si la liaison entre le message envoyé (dessiner) et la méthode déclenchée pour y répondre est faite à la compilation, on voit que c'est la version de la méthode dessiner de `FormeGeometrique` qui sera systématiquement utilisée, puisque la figure considère ses composantes comme des formes géométriques "standard". Si en revanche on applique la liaison dynamique, alors chaque forme se dessinera en fonction de sa nature réelle, cercle ou rectangle. C'est cela que recouvre, en programmation par objets, la notion de polymorphisme.

3 Implémentation objet de JAVA

Le but de cette section est de préciser ce qui a été implémenté du modèle précédemment décrit dans le langage JAVA, et comment cela a été fait.

Les sections suivantes vont donc aborder :
- la technique d'encapsulation, avec les paquetages ;
- l'implémentation de l'héritage, avec les interfaces ;
- d'autres aspects et extensions au modèle de base.

3.1 Encapsulation des données en JAVA

Le support principal de l'encapsulation est la classe. Tout type de donnée JAVA est soit un type simple, soit un tableau, soit une classe.

La classe peut contenir
- des attributs de type et en nombre quelconques ;
- des méthodes en nombre quelconque ;
- des classes internes (dont nous ne discuterons pas ici ; nous en verrons un exemple d'utilisation lors de l'étude de la programmation par événements).

L'ordre dans lequel les différents constituants de la classe sont déclarés n'a pas d'influence. Par convention, on déclare d'abord les attributs, puis les méthodes.

Tout attribut non simple (i.e. tableau ou objet) doit être alloué dynamiquement avant toute utilisation dans les méthodes. La déclaration d'un attribut objet le fait automatiquement pointer sur `null`, le seul pointeur universel prédéfini.

3.1.1 Contraintes et conventions

Toute classe doit être définie dans un fichier source qui porte son nom, suivi de l'extension `.java`.

Le compilateur `javac` créera à partir de ce fichier un fichier qui portera le même nom, avec l'extension `.class`.

La JVM exécutera ce code lorsqu'on lui passera le nom de la classe (et non celui du fichier !) en paramètre.

En général, les noms de classe commencent par une majuscule, les noms de paquetage, de méthode ou d'attribut par une minuscule. Quand un identifiant est la concaténation de plusieurs mots, chaque mot le composant débute par une majuscule, comme dans `UnNomDeClasseLong` ou `uneVariableLongue`.

3.1.2 Les paquetages

Lorsque plusieurs classes concourent à la réalisation d'une fonctionalité complexe donnée, ou plus généralement lorsqu'elles sont fortement corrélées entre elles, il est possible de les regrouper au sein d'une structure plus importante appelée *paquetage*.

L'intérêt de la définition d'un paquetage est de fournir une plus grande facilité de communication pour les classes qui se trouvent à l'intérieur du paquetage, et de ne publier qu'une interface restreinte pour les clients extérieurs.

Les paquetages font également office d'espace de nommage : ils permettent d'éviter, lorsqu'on construit un programme à l'aide d'élements de provenance diverse, de se retrouver avec deux classes portant le même nom : en effet, le nom complet d'une classe est l'ensemble constitué de son paquetage d'appartenance, d'un "." et du nom "court" de la classe, par exemple `java.lang.Object` représente de façon unique la classe `Object` du paquetage `java.lang` de la librairie de classes standard du langage.

Afin de s'assurer de l'unicité des noms des paquetages, il est d'usage d'utiliser les noms de domaines DNS inverses des entités au sein desquelles on travaille. Par exemple, si la société *Truc* possède un site `www.truc.com` et fabrique des paquetages JAVA, elle nommera un paquetage *bidule* du nom `com.truc.bidule`, ce qui en assure l'unicité.

Les paquetages n'ont pas de structure physique. Ils n'existent que logiquement, par inclusion d'une ou plusieurs classes dans un paquetage.

Lorsqu'on désire inclure une classe dans un paquetage, on utilise – au début du fichier de définition de la classe – l'instruction `package` suivie du nom du paquetage d'appartenance de la classe.

Lorsqu'on désire utiliser une classe `UneClasse` d'un paquetage `unPackage`, il faut
- soit mentionner le nom complet de la classe, `unPackage.UneClasse`

– soit importer la paquetage à l'aide d'une clause `import`. On peut alors utiliser la classe en ne mentionnant que son nom "court".

Exemple :

```
unPackage.MaClasse o = new unPackage.MaClasse() ;
```

ou bien

```
import unPackage.* ;
MaClasse o = new MaClasse() ;
```

Seul le paquetage `java.lang` n'a pas besoin d'être explicitement importé (il s'y trouve des classes "incontournables", comme `System` ou `Object`).

3.1.3 Niveaux d'accès

Tout attribut, méthode ou classe JAVA possède un niveau d'accessibilité. On distingue 4 niveaux :
– le niveau public (mot réservé `public`), qui rend l'accès possible de n'importe quel client ;
– le niveau protégé (mot réservé `protected`), qui limite l'accès aux membres des classes dérivées et aux classes du paquetage courant ;
– le niveau paquetage (par défaut), qui limite l'accès aux membres du paquetage courant ;
– le niveau privé (mot réservé `private`), qui limite l'accès aux membres de la classe.

Les niveaux existants et les accès qu'ils permettent sont résumés dans le tableau suivant :

visibilité	classe	attribut/méthode
`public`	tout le monde	tout le monde
`protected`	N/A	dérivées et classes du paquetage
	classes du paquetage	classes du paquetage
`private`	N/A	la classe

En général, il est recommandé de laisser un accès *paquetage* ou protégé aux attributs, et un accès public aux méthodes.

3.1.4 Construction et destruction d'instances

Dans la mesure où un programme qui utilise une classe n'a pas accès à la totalité des informations détenues par cette classe, il est nécessaire de fournir un moyen d'initialisation complète – et de destruction – des instances de cette classe.

En JAVA, pour l'initialisation, on utilise un constructeur. C'est une méthode particulière qui
– porte obligatoirement le nom de la classe ;
– est invoquée implicitement lors de l'allocation de mémoire faite par `new` ;
– peut avoir des paramètres. Dans ce cas, ces paramètres doivent être donnés à `new` pour lui permettre d'initialiser correctement l'instance créée.

Comme toute méthode, les constructeurs sont surchargeables. Il suffit qu'ils soient distinguables deux à deux par le nombre de paramètres, ou le type d'un paramètre. Comme ce sont des méthodes de la classe, ils ont bien entendu accès à la totalité des membres de cette classe, même les membres privés.

Tout constructeur est *a priori* public (sauf lorsqu'on souhaite explicitement contraindre l'initialisation *via* une méthode particulière).

La destruction des objets n'est pas à la charge du programmeur (comme ce serait le cas en C ou C++). Un processus spécial de la JVM, appelé "ramasse-miettes" (ou *garbage collector*), parcourt périodiquement la mémoire à la recherche de zones qui ne sont plus adressées par aucune variable et restitue ces zones comme disponibles. Néanmoins, pour certains objets complexes, on peut vouloir effectuer un traitement particulier lorsque ces objets disparaissent effectivement. La méthode `finalize()` peut alors être définie dans la classe, elle sera appelée avant passage du ramasse-miettes sur l'instance.

3.1.5 Exemple simple mais complet

Définissons la classe `Pile`, que nous placerons – avec la classe `Box`, une classe outil (donc non publique) nécessaire pour représenter la pile par une structure chaînée – dans un paquetage `pile`.

```
// $Id: Pile.1.java 1576 2008-11-06 11:09:02Z phil $

package pile;

// une classe non "publique" n'est pas exportee hors du paquetage
class Box {
  protected Object val;
  protected Box next;
  Box() { val=null; next=null; }
  Box(Object o) { val=o; next=null; }
  Box(Object o, Box n) { val=o; next=n; }
}

/**
 * on choisit pour implementer la pile une structure chainee.
 * Pas de gestion d'exceptions.
 * @author laroque@u-cergy.fr
 * @version $Id: Pile.1.java 1576 2008-11-06 11:09:02Z phil $
 */
public class Pile {
  protected Box top;

////////////////////////////////////////////////////////////
/**
 * Le constructeur. Assure que la pile est vide au depart.
 */
  public Pile() { top=null; }

////////////////////////////////////////////////////////////
/**
 * depile le sommet de pile (pas de gestion d'erreur en cas
 * de pile vide)
 * @return l'ancien sommet de pile
 */
  public Object pop() {
    Box tmp = top;
    top = top.next;
    return tmp.val;
  }

////////////////////////////////////////////////////////////
/**
 * empile un objet sur la pile
 * @param o l'objet a empiler
 * @return le receveur
 */
  public Pile push(Object o) {
    top = new Box(o,top);
    return this;
  }

////////////////////////////////////////////////////////////
/**
 * test de pile vide
 * @return true ssi la pile est vide
 */
  public boolean isEmpty() { return top == null; }
```

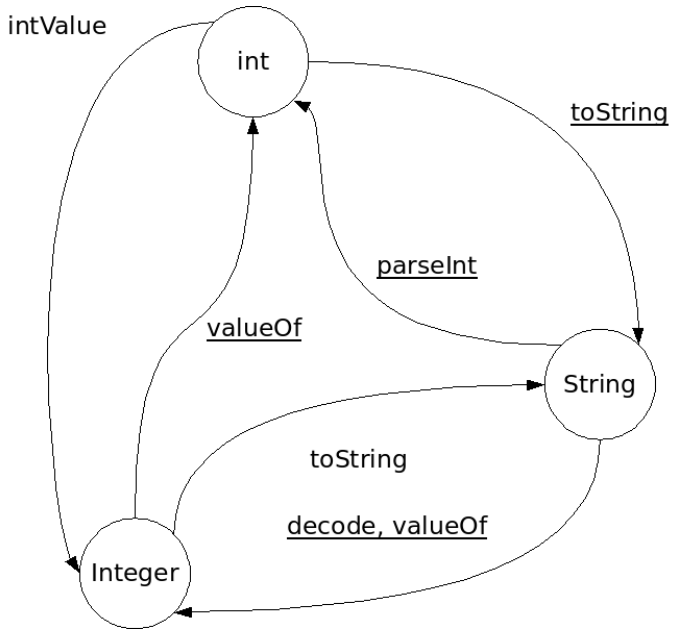
```
////////////////////////////////////////////////////////////
/**
 * acces au sommet de pile sans depiler. pas de gestion d'erreur
 * en cas de pile vide
 * @return le sommet
 */
  public Object top() { return top.val; }
}
```

3.2 Types simples, chaînes de caractères et wrappers

Chaque type simple possède une classe qui en est le reflet dans la bibliothèque JAVA. La raison de la présence de ces classes – appelées *wrappers* – est que les collections de la bibliothèque sont définies pour contenir des objets, elles ne peuvent donc pas contenir de variables de type simple. Lorsqu'on a besoin de placer des nombres dans une collection, on a donc recours à la classe qui correspond à ce type simple dans la bibliothèque :

type simple	wrapper
`byte`	`Byte`
`char`	`Character`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Les méthodes de conversion entre type simple et wrapper, ainsi que celles permettant d'obtenir une représentation sous forme de chaîne de caractères (classe `String`), sont résumées dans le dessin ci-dessous. On a illustré le cas des entiers ; les méthodes soulignées sont statiques, elles prennent un paramètre dont le type correspond au type d'arrivée de la flèche.



Ainsi, pour obtenir l'entier 3 à partir de l'objet `Integer` 3, on écrira :

```
Integer i = new Integer(3) ;
int j = Integer.valueOf(i) ;
String s = i.toString() ;
int k = Integer.parseInt(s) ;
```

L'affichage formaté de chaînes de caractères contenant des nombres peut se faire grâce à la méthode `format` de la classe `PrintStream`, proche du `printf` de C :

```
int i = 357 ;
System.out.format(``Valeur de i : %d\n'', i) ;
```

N.B. : la classe `String` possède également une méthode `format`, qui crée une `String` à partir du format spécifié. Elle est donc analogue au `sprintf` de C.

3.3 L'héritage en JAVA

3.3.1 Introduction

Les concepteurs de JAVA ont fait des choix d'implémentation de l'héritage qui tentent de conjuguer souplesse de représentation et non-ambiguïté. Pour ce faire, ils ont choisi de n'avoir qu'un héritage simple, mais sous une forme sophistiquée qui permet d'avoir un sous-ensemble des possibilités qu'offre l'héritage multiple, grâce à la notion d'interface.

Toute classe JAVA hérite, soit explicitement, soit implicitement, d'une même classe racine, `java.lang.Object`. Cette classe définit le comportement minimal commun à tout objet JAVA. L'importance de la présence de cette classe est énorme, puisqu'elle permet de créer des collections naturellement polymorphes : toute collection qui déclare contenir des `Object` peut en fait accueillir n'importe quel objet, qu'il soit d'une classe incluse dans le JDK, fournie par un vendeur ou construite "à la main". Ceci est d'autant plus naturel que la liaison est systématiquement dynamique (ce qui n'est pas le cas en C++ par exemple, avec la notion de fonction *virtuelle*).

Les sections qui suivent précise la syntaxe de la déclaration de relation d'héritage, les problèmes de gestion de la construction d'objets dérivés et le rôle des interfaces.

3.3.2 La relation `extends`

Une classe `B` qui dérive d'une classe `A` doit exprimer cette filiation par la mention d'une clause `extends` dans sa déclaration. Comme l'héritage est simple, il ne peut y avoir qu'un `extends` au plus par classe. Si aucun `extends` n'est mentionné, `extends java.lang.Object` est sous-entendu.

Exemple :

```
public class A { ... }
...
public class B extends A { ... }
```

Les méthodes de la classe `B` ont accès aux membres publics, protégés et *paquetage* de `A`, mais pas à ses membres privés.

3.3.3 Héritage et construction

Le constructeur de la classe dérivée ne peut pas accéder aux membres privés de la classe de base dont il hérite : il y a donc des attributs qu'il ne peut initialiser !

Pour résoudre ce problème, il doit se tourner vers le constructeur de la classe de base, qui lui a bien entendu libre accès à ces attributs. L'invocation du constructeur de la classe de base se fait au début du corps du constructeur de la classe dérivée (première instruction), par l'utilisation du mot réservé `super`. Elle est *obligatoire*, sauf dans le cas où le constructeur de la classe de base est sans paramètre, auquel cas elle est implicite.

Exemple : une classe `Carre` dérivée de `Rectangle` :

```
public class Rectangle {
  private double longueur, largeur ;
  public Rectangle (double L, double l) {
    longueur = L ; largeur = l ;
  }
  ...
}
...
public class Carre extends Rectangle {
  public Carre (double cote) {
```

```
    super(cote, cote);
  }
  ...
}
```

3.3.4 Classes et méthodes abstraites

Une méthode est dite *abstraite* si elle n'est que déclarée dans la classe, et non définie (*i.e.*, seul son prototype est fourni, pas sa définition). On explicite le caractère abstrait d'une méthode en faisant précéder son profil du mot réservé `abstract`.

L'avantage des méthodes abstraites est qu'elles peuvent déclarer un comportement global commun à un ensemble de classes, et ce très haut dans la hiérarchie de l'héritage.

Une *classe abstraite* est une classe qui possède au moins une méthode abstraite. Une telle classe est non instanciable. Là encore, on l'explicitera en plaçant le mot réservé `abstract` avant le nom de la classe dans sa déclaration.

Exemple :

```
public abstract class Algorithm {
  public abstract Object run();
  ...
}
```

Une classe qui hérite d'une classe abstraite doit implémenter toutes les méthodes abstraites de sa classe de base, ou bien elle est également une classe abstraite.

3.3.5 Les interfaces

L'introduction de la notion d'interface permet à JAVA de résoudre de façon élégante le problème du "losange" (voir 2.2.2 p. 8). En effet, ce problème survient dès qu'il y a inclusion multiple d'une classe de base commune : la structure de cette classe peut être dupliquée ou non.

Une interface JAVA est une classe abstraite particulière, qui possèdes les propriétés suivantes :
- une interface ne peut pas dériver d'une classe, seulement d'une autre interface (ou de plusieurs) ;
- les méthodes sont toutes abstraites ;
- les attributs sont tous des constantes de classe (voir 3.4.1 p. 15) ;
- attributs et méthodes sont `public`.

Avec ces contraintes, on voit que la structure d'une interface est forcément vide. Il est donc possible d'autoriser un héritage multiple sur les interfaces sans rencontrer le problème du losange...

Une interface étant par définition abstraite, il n'est pas nécessaire d'employer le mot réservé `abstract` dans ce cas.

Lorsqu'une classe souhaite "hériter" d'une interface, elle le déclare à l'aide du mot réservé `implements` (il y a donc une différence explicite avec l'héritage entre deux classes). Dans ce cas,
- soit elle implémente effectivement *toutes* les méthodes déclarées dans l'interface, et elle est alors utilisable comme toute autre classe ;
- soit elle laisse *au moins une* méthode non implémentée, et elle devient alors une classe abstraite.

Les interfaces sont extrêmement utiles dans le design des applications, et la librairie standard y fait abondamment usage. En particulier, elles permettent de définir des API (*Application Programming Interface*) standard, qui devront être implémentées par les vendeurs, qui assurent au code client une réelle portabilité. Un exemple concret simple est le paquetage java.sql, qui définit le protocole d'utilisation des bases de données dans un programme JAVA, et rend donc ce programme totalement indépendant de la base effectivement choisie (Oracle, MySQL, ACCESS,...)

En résumé, on peut dire que l'héritage en JAVA est simple pour les classes, multiple pour les interfaces : une classe donnée peut "étendre" (`extends`) une classe de base et "implémenter" (`implements`) un nombre quelconque d'interfaces.

3.4 Autres caractéristiques "objet"

3.4.1 Variables et méthodes "de classe"

Une variable de classe est un attribut qui, au lieu d'être dupliqué à chaque instanciation (chaque instance possède un exemplaire distinct de l'attribut, qu'elle peut donc valuer comme elle l'entend), est

partagé par l'ensemble des instances. On peut la voir comme une variable globale, mais de portée limitée aux instances de la classe.

De façon similaire, une méthode de classe est une méthode qui n'implémente pas un message envoyé à une instance particulière, mais qui définit un comportement global au niveau de la classe. En particulier, il n'y a pas de receveur pour une méthode de classe.

Un attribut ou une méthode de classe sont référencés en mentionnant le nom de la classe (notation pointée).

Exemple :

- `java.lang.System.out` est une variable de classe de la classe `System` du paquetage `java.lang` ;
- `java.lang.Math.random()` est une méthode de classe de la classe `Math` du même paquetage.

On se sert en général d'attributs de classe pour stocker une connaissance qui ne dépend pas d'une instance particulière de la classe, ni même d'un groupe d'instances identifié.

La déclaration d'un attribut (ou d'une méthode) de classe se fait en utilisant le mot réservé `static` :

Exemple :

```
public class MaClasse {
  protected static int nbInstances = 0 ;
  public int getNbInstances() { return nbInstances ; }
  ...
}
```

On remarque sur cet exemple que, contrairement aux attributs ordinaires qui sont initialisés dans le constructeur, les attributs de classe doivent être initialisés dès leur déclaration (puisqu'ils ont une valeur indépendamment de la présence ou non d'instances de la classe).

3.4.2 Classes "internes"

Il est possible (depuis le JDK 1.1) de définir des classes à l'intérieur de la définition d'une classe. Ces classes, dites *internes* (*inner classes*), ont les mêmes accès aux membres de la classe englobante que les méthodes de cette dernière. Bien qu'on puisse considérer cela comme une violation du principe d'encapsulation, ce mécanisme se révèle extêmement pratique dans certains cas (voir en particulier le cas des *adapters* graphiques, 8.2.2 p. 35).

4 La généricité

4.1 Principe

Reprenons l'exemple de la pile évoqué en 3.1.5. Pour assurer une réutilisation maximale de la structure, on a choisi de typer ses éléments comme des `Object`, ce qui permet de mettre dans la pile n'importe quel objet Java (puisque toutes les classes dérivent de `Object`).

Cette approche a cependant un inconvénient : aucune hypothèse n'étant faite sur le type des éléments, l'information sur le type précis de chaque élément est perdue par le compilateur :

```
Pile p = new Pile() ;
Compteur c = new Compteur() ;
p.push(c) ;
p.pop().incremente() ;
```

Cette dernière instruction va provoquer une erreur à la compilation : le type attendu en retour de la méthode `pop()` est `Object`, or aucune méthode `pop()` n'existe dans la classe `Object`. En d'autres termes, nous savons que nous avons mis un `Compteur` dans notre pile, et que c'est licite de l'incrémenter, mais le compilateur lui ne le sait pas. Il faut le lui indiquer explicitement, à l'aide d'un forçage de type (*type casting*) :

```
((Compteur)(p.pop())).incremente() ;
```

Cette fois le code va compiler sans problème, mais la notation est assez lourde. En outre, si par mégarde on a mis dans la pile autre chose qu'un `Compteur`, le problème n'apparaîtra qu'à l'exécution du programme.

Dans la plupart des cas, on a une idée du type des éléments que l'on va placer dans la pile (par exemple des `Compteur`). On a alors trois solutions :

1. ne rien changer, et accepter de mettre des *casts* partout où c'est nécessaire (on n'est pas à l'abri d'erreurs à l'exécution) ;
2. écrire une classe `Pile` qui prend spécifiquement des `Compteur`, par copier-coller de la classe initiale et remplacement de `Object` par `Compteur` (cela duplique du code, donc rend la maintenance plus difficile ; en outre, il faut lui trouver un autre nom que `Pile`, ou la placer dans un autre paquetage...) ;
3. rendre la classe `Pile` (presque) aussi générale et mieux informée en utilisant un *type paramétré* pour les éléments.

4.2 Syntaxe des types paramétrés

Le principe de définition d'un type paramétré consiste à mettre ce qui est variable dans le type (dans notre exemple, le type des éléments de la pile) comme paramètre de la classe en cours d'écriture. La syntaxe générale de la définition du type est

```
public class MaClasse<P1,...,PN> { ...}
```

On peut alors utiliser les paramètres `P1 ... PN` dans le code de la classe comme si c'étaient de "vrais" types de données.

De même, l'utilisation de ce type de classe paramétrée se fait très simplement, en donnant une valeur à chacun des paramètres de la classe lors de l'instanciation. Par exemple :

```
public class MaClasse<Type1,Type2> { ...}
...
MaClasse<int,float> m1 ;
MaClasse<String,int> m2 ;
```

4.3 Exemple : paramétrage de la classe `Pile`

Le paramétrage de la classe `Pile` implique celui de sa classe outil, `Box`, comme le montre le code ci-dessous :

```
// $Id: Pile.java 1576 2008-11-06 11:09:02Z phil $

package pile;
```

```
// une classe non "publique" n'est pas exportee hors du paquetage
class Box<E> {
  protected E val;
  protected Box<E> next;
  Box() { next=null; }
  Box(E o) { val=o; next=null; }
  Box(E o, Box<E> n) { val=o; next=n; }
}

/**
 * on choisit pour implementer la pile une structure chainee.
 * Pas de gestion d'exceptions.
 * @author laroque@u-cergy.fr
 * @version $Id: Pile.java 1576 2008-11-06 11:09:02Z phil $
 */
public class Pile<E> {
  protected Box<E> top;

  ////////////////////////////////////////////////////////////
  /**
   * Le constructeur. Assure que la pile est vide au depart.
   */
  public Pile() { top=null; }

  ////////////////////////////////////////////////////////////
  /**
   * depile le sommet de pile (pas de gestion d'erreur en cas
   * de pile vide)
   * @return l'ancien sommet de pile
   */
  public E pop() {
    Box<E> tmp = top;
    top = top.next;
    return tmp.val;
  }

  ////////////////////////////////////////////////////////////
  /**
   * empile un objet sur la pile
   * @param o l'objet a empiler
   * @return le receveur
   */
  public Pile push(E o) {
    top = new Box<E>(o,top);
    return this;
  }

  ////////////////////////////////////////////////////////////
  /**
   * test de pile vide
   * @return true ssi la pile est vide
   */
  public boolean isEmpty() { return top == null; }

  ////////////////////////////////////////////////////////////
  /**
   * acces au sommet de pile sans depiler. pas de gestion d'erreur
```

```
   * en cas de pile vide
   * @return le sommet
   */
  public E top() { return top.val; }
}
```

On peut alors utiliser cette pile pour y stocker des `Compteur`, sans réécrire de code (et sans *casts*) :

```
Pile<Compteur> p = new Pile<Compteur>() ;
Compteur c = new Compteur() ;
p.push(c) ;
p.pop().incremente() ;
```

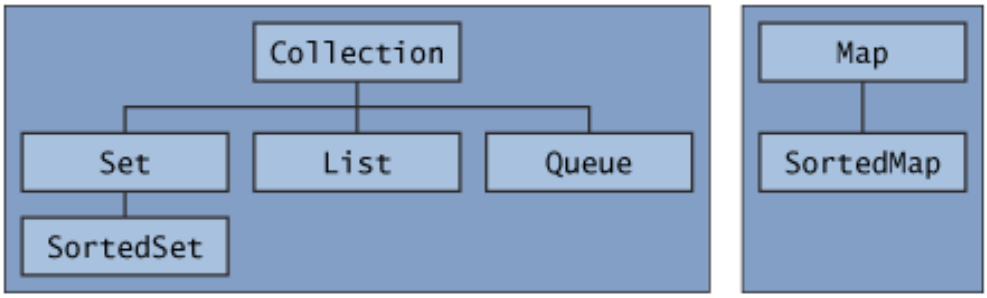
5 Les collections

La bibliothèque standard Java inclut, dans son paquetage `java.util`, un ensemble de classes destinées à gérer le stockage d'objets dans des "collections" quelconques et d'opérer des traitements sur les objets contenus dans ces collections. En fonction des besoins (éléments ordonnés ou non, triés ou non, collections de taille fixe ou changeant fréquemment, etc.), on peut trouver une collection adaptée au problème courant. Les collections peuvent s'utiliser sans paramétrage (déconseillé, les éléments sont alors des `Object`, comme dans l'exemple initial de la pile - *cf.* 3.1.5 p. 11) ou paramétrées par le type de leurs éléments. Dans la suite on nommera "E" le type des éléments.

Les sections qui suivent décrivent les principaux concepts communs à toutes les collections, puis détaillent quelques-unes des fonctionnalités importantes de ces collections.

5.1 La hiérarchie des collections

Les interfaces représentant les collections sont représentées dans la figure suivante :



Le cas particulier des "maps" sera traité à part (voir section 5.6 page 23). Toutes les autres collections dérivent d'une interface commune, `Collection`, qui définit les fonctionnalités communes à l'ensemble des collections disponibles dans la bibliothèque.

5.2 Les fonctionnalités communes

L'interface `java.util.Collection` déclare les fonctionnalités communes aux collections Java. Parmi les principales (liste non exhaustive) :

- L'ajout d'un élément :
  `boolean add(E element) ;`
- le retrait d'un élément :
  `boolean remove(E element) ;`
- la taille de la collection :
  `int size() ;`
- un prédicat pour savoir si la collection est vide :
  `boolean isEmpty() ;`
- un prédicat pour tester l'appartenance d'un élément à la collection :
  `boolean contains(E element) ;`

Cette interface est étendue par les interfaces décrites dans les sections suivantes. En ce qui concerne les classes implémentant ces interfaces, on les a regroupées dans le tableau suivant :

Interface	Classe	caractéristique
`Set`	`HashSet`	+ efficace
	`TreeSet`	éléments triés
`List`	`ArrayList`	accès direct performant
	`LinkedList`	ajout / retraits performants[2]
`Map`	`HashMap`	+ efficace
	`TreeMap`	éléments triés (par clés)

5.3 Les ensembles

Dans ces collections, la duplication d'éléments est interdite : ajouter un élément à un ensemble qui contient déjà cet élément revient à ne rien faire. Le test de contenance est réalisé à l'aide de la méthode `equals()`, que l'on peut donc redéfinir sur les éléments.

À noter : les `HashSet` fonctionnent à l'aide d'une table de hachage, il peut donc être judicieux de (re-)définir sur les éléments la fonction `hashCode()` afin d'assurer une répartition homogène des clés dans les classes de hachage.

On utilisera de préférence un `TreeSet` (implémenté, comme son nom l'indique, à l'aide d'un arbre binaire de recherche équilibré et non à l'aide d'une table de hachage) lorsqu'on souhaitera extraire les éléments dans un ordre donné. Les éléments d'un `TreeSet` doivent appartenir à une classe qui implémente l'interface `Comparator`. Un exemple d'utilisation des ensembles est donné en 5.5.3, p. 22.

Quelques autres fonctionalités des `Collection` peuvent être intéressantes dans le cas des `Set` :

- Test d'inclusion : s1 est un sous-ensemble de s2 si l'expression suivante est vérifiée :
  `s2.containsAll(s1) ;`
- Union : on peut ajouter à s1 les éléments de s2 en écrivant
  `s1.addAll(s2) ;`
- Intersection : on peut retirer à s1 les éléments qui ne figurent pas dans s2 en écrivant
  `s1.retainAll(s2) ;`
- Différence symétrique : on peut retirer à s1 les éléments qui figurent dans s2 en écrivant
  `s1.removeAll(s2) ;`

5.4 Les listes

Les listes se distinguent des ensembles par le fait qu'à chaque élément est associée une *position*, un *ordre* dans la liste, et que les doublons sont autorisés.

On a les fonctionnalités supplémentaires suivantes :

- manipulation des éléments (accès, ajout, retrait,...) par leur position dans la liste :
  ```
  E get(int pos) ;
  E set(int pos, E elt) ;
  boolean add(E elt) ;
  void add(int pos, E elt) ;
  E remove(int pos) ;
  ```
- recherche de la position d'un élément donné :
  `int indexOf(E elt) ;`
- extraction de sous-listes par la position des bornes :
  `List<E> subList(int from, int to) ;`[3]

On trouve deux implémentations des listes dans la bibliothèque standard :

1. Les listes chaînées, ou `LinkedList`, qui permettent d'ajouter ou retirer facilement des éléments (surtout aux extrémités) mais qui sont peu performantes an terme de temps d'accès aux éléments internes ;
2. les listes contigües, ou `ArrayList`, qui ont les propriétés inverses.

Les listes chaînées offrent quelques méthodes supplémentaires :

[3] NB : l'élément en position `from` est inclus, mais pas celui en position `to`

– ajout d'éléments aux extrémités :
```
void addLast(E elt) ;
void addFirst(E elt) ;
```
– accès aux extrémités :
```
E getFirst() ;
E getLast() ;
```
– retraits d'éléments aux extrémités :
```
E removeFirst() ;
E removeLast() ;
```
On remarque que l'on peut donc très facilement implémener, grâce aux `LinkedList`, des piles et des files simples. C'est la raison pour laquelle nous ne nous attarderons pas ici sur ces objets.

5.5 Traversée de collections

Pour comprendre l'intérêt d'un mécanisme homogène et performant de traversée des collections, prenons un exemple simple. On suppose qu'on utilise une liste chaînée de `Truc`, et l'on veut effectuer un certain `traitement` sur les objets de cette liste. On écrira donc quelque chose comme :

```
List<Truc> l = new LinkedList<Truc>() ;
...
for (int i = 0 ; i < l.size() ; i++)
  l.get(i).traitement() ;
```

Ce code compile et s'exécute correctement, mais on peut lui faire deux reproches majeurs :
– il est très peu performant : la méthode `get()` parcourt les $(i-1)$ premiers éléments de la liste avant d'accéder au i^{eme}, le temps d'exécution de la boucle sera donc proportionnel à

$$1 + 2 + ... + n = \frac{n(n+1)}{2} = O(n^2)$$

– il est propre aux listes : pour parcourir un ensemble, il faudra écrire un autre algorithme.
Pour palier ces deux objections, la bibliothèque Java offre deux solutions alternatives : les *itérateurs* d'une part, une boucle `for` propre aux collections d'autre part.

5.5.1 Les itérateurs

Un itérateur est un objet particulier adapté à chaque type de collection, qui permet entre autres[4] à un programme d'accéder en séquence à chacun des éléments de la collection sans avoir de connaissance sur l'implémentation en mémoire de ces éléments.

Toute collection peut créer un itérateur grâce à une méthode non encore mentionnée jusqu'ici, `iterator()`. Cette méthode crée et renvoie un itérateur sur la collection, "branché" sur le premier élément de la collection. Un tel itérateur est essentiellement capable de deux choses :

1. faire un accès à l'élément courant de la collection (cet accès fait passer l'itérateur à l'élément suivant) :
   ```
   E next() ;
   ```
2. dire si tous les éléments de la collection ont été visités, ou s'il en reste encore :
   ```
   boolean hasNext() ;
   ```

Ainsi, l'exemple précédent de la liste de `Truc` peut s'écrire beaucoup plus proprement en utilisant un itérateur :

```
List<Truc> l = new LinkedList<Truc>() ;
...
for (Iterator<Truc> i = l.iterator() ; i.hasNext() ; )
  i.next().traitement() ;
```

Cette écriture a deux avantages sur la précédente :

1. Le coût de la méthode `next()` est constant, on a donc un parcours en $O(n)$ (comme pour une boucle classique en C par exemple) et non plus en $O(n^2)$ comme dans l'exemple précédent.

[4] Nous nous tiendrons à cette fonctionnalité dans ce document

2. La syntaxe de la boucle est indépendante du type de collection : si on remplace
   `List<Truc> l = new LinkedList<Truc>() ;`
   par
   `Set<Truc> l = new THashSet<Truc>() ;`
   la boucle fonctionne exactement de la même manière.

5.5.2 La boucle for "spéciale collections"

La syntaxe en est très simple, et on retrouve les avantages mentionnés à la section précédente. Ainsi, la même boucle s'écrira :

```
List<Truc> l = new LinkedList<Truc>() ;
...
for (Truc t : l)
  t.traitement() ;
```

5.5.3 Exemple d'utilisation et comparaison

On donne ci-dessous un exemple de code utilisant les itérateurs puis la boucle for spécifique aux collections. Cet exemple illustre en outre la notion de critère de tri des éléments dans un ensemble :

```
// $Id: TestSet.java 1575 2008-11-06 10:13:45Z phil $


import java.util.*;

/** tool class to order Strings in a decreasing lexicographical manner */
class Decrease implements Comparator<String> {
  public int compare(String s1, String s2) {
    return s2.compareTo(s1);
  }
}

/**
 * Simple test of TreeSet and HashSet.
 *
 * This program illustrates two aspects of java Sets: <ul>
 * <li> the difference between {@link HashSet} (based on hash table) and
 *      {@link TreeSet} (based on red/black trees)
 * <li> the two ways of traversing a set, using either the "collection" version
 *   of the "for" loop, or a traditional iterator
 * </ul>
 * @author laroque@u-cergy.fr
 * @version $Rev: 1575 $
 */
public class TestSet {

  /**
   * test program.
   * Use three sets (first is hashed, second is default-sorted, third is
   * sorted using a {@link Decrease} object), add same elements in all sets
   * then display the set contents.
   */
  public static void main(String args[]) {

    Set<String> s1 = new HashSet<String>();
    Set<String> s2 = new TreeSet<String>();
    Set<String> s3 = new TreeSet<String>(new Decrease());

    String verse[] = { "In", "Xanadu", "did", "Kubla", "Kahn", "A",
```

```
                    "Stately", "pleasure", "dome", "decree" };

    // arrays use the "C" version of the for loop
    for (int i = 0; i < verse.length; i++) {
      s1.add(verse[i]);
      s2.add(verse[i]);
      s3.add(verse[i]);
    }

    // here we use the "collection" version of the for loop
    // first set is implemented using a hash table: "random" display order
    System.out.print("Hash     : ");
    for (String s : s1) System.out.print (s + " ");
    System.out.println();

    // second set is implemented using a balanced tree
    // (default order for Strings: dictionary)
    System.out.print("Increase : ");
    for (String s : s2) System.out.print (s + " ");
    System.out.println();

    // here we use the traditional iterator mechanism
    // third set uses a balanced tree oredered based on a Decrease object
    // (reverse dictionary)
    System.out.print("Decrease : ");
    for (Iterator<String> i = s3.iterator();i.hasNext();)
      System.out.print (i.next() + " ");
    System.out.println();
  }
}
```

5.6 Le cas spécial des "maps"

Les maps ne sont pas des collections au sens Java du terme (on remarque sur le graphique représentant la hiérarchie des collections, p. 19, qu'elles ne dérivent pas de l'interface `Collection`), mais leur rôle est néanmoins de regrouper non pas des objets mais des couples *clé - valeur*. Dans la suite on parlera donc de `Map<K,V>` où `K` représente le type des clés et `V` celui des valeurs associées. Techniquement le type des éléments d'une map est `Entry<K,V>` :

```
// Interface for entrySet elements
    public interface Entry<K,V> {
        K getKey() ;
        V getValue() ;
        V setValue(V value) ;
    }
```

Une map n'autorise pas la duplication des clés : une clé ne peut faire référence qu'au plus à une valeur dans la map. En ce sens on peut dire qu'elle modélise la notion de fonction mathématique $x \longmapsto y = f(x)$ où x est la clé et y la valeur associée. Lorsqu'on veut pouvoir associer plusieurs valeurs à une même clé, on choisira une `MultiMap`.

Les principales fonctionnalités (en plus des fonctions classiques des collections, `size()`, `isEmpty()` etc.) proposées par ces collections sont les suivantes :

- ajout d'un couple clé-valeur :
  `V put(K key, V val) ;`
- accès à une valeur depuis une clé :
  `V get(K key) ;`
- suppression de la valeur correspondant à une clé :
  `V remove(Key k) ;`
- test d'appartenance d'une clé ou d'une valeur (beaucoup plus lent) :
  `boolean containsKey(K key) ;`

```
boolean containsValue(V val) ;
```
– Accès à l'ensemble des clés ou à celui des valeurs :
```
public Set<K> keySet() ;
public Collection<V> values() ;
```
Comme indiqué dans le tableau de la p. 20 représentant les implémentations possibles des collections, on a deux possibilités pour créer une map, soit utiliser une `HashMap`, soit une `TreeMap`. Les considérations sont identiques au cas des `Set` déjà traité.

5.7 Algorithmes sur les collections

Terminons ce rapide tour d'horizon en mentionnant que les principaux algorithmes opérant sur les collections sont implémentés en Java et regroupés dans la classe `Collections`. Il s'agit principalement de fonctions statiques[5] permettant entre autres :
– de trier les éléments d'une collection (si une relation d'ordre naturelle existe ou si on en explicite une) :
```
Collections.sort(maCollection) ;
```
– de renverser `reverse()`, copier `copy()` une collection ;
– de faire des recherches efficaces dans une collection triée :
```
int pos = Collections.binarySearch(maCollection, maCle) ;
```
– de compter le nombre d'occurrences d'un élément (`frequency`) ;
– de trouver les extrema d'une collection (`min` et `max`)

6 Les exceptions

6.1 Le principe

Dans la programmation traditionnelle, on est souvent confronté à un problème de communication entre les différents acteurs de la réalisation d'un logiciel (du moins dès lors que sa taille interdit la réalisation par une personne seule).

Ce problème est encore exacerbé en approche par objet, dans la mesure où cette approche favorise la réutilisation du logiciel, ne serait-ce que par sa modularité naturelle.

Il peut se résumer en ces termes : comment gérer une situation exceptionnelle dans un module réutilisable ?

Supposons l'existence de deux acteurs, le concepteur du module de code (CM) et l'utilisateur de ce module (UM), qui est aussi le concepteur d'une application intégrant le module. *A priori*, on suppose que CM a conçu le module avant d'avoir connaissance de l'existence de UM (par exemple, le module est un composant logiciel en vente sur catalogue).

Lorsqu'une situation exceptionnelle survient à l'intérieur du module, le contexte est le suivant :
– CM peut détecter cette situation exceptionnelle (il a accès au code, il peut ajouter un test, etc.), MAIS il ne peut pas faire le choix d'une stratégie de traitement, car celle-ci dépend de l'application dans laquelle le module est plongé (songeons à un arrêt brusque du programme dans un pilote automatique de navette spatiale...) ;
– UM saurait comment traiter le problème (il maîtrise l'application), MAIS il ne peut bien souvent pas le détecter, le composant lui apparaissant en général comme une "boîte noire".

Il est donc indispensable d'établir un protocole de communication efficace entre CM et UM. C'est pour répondre à ce besoin qu'a été mise au point la technique des exceptions. Cette technique n'est pas propre à JAVA, on la trouve déjà en 83 dans ADA, on la retrouve en 86 avec C++.

Elle consiste à permettre à CM, lorsqu'il a détecté une telle situation anormale, d'envoyer une mise en garde à UM (en JAVA, sous la forme d'un Objet) renseignant ce dernier sur la nature du problème. L'avantage de cette approche est de donner à chacun la responsabilité qu'il est capable d'assumer : CM prévient, mais ne traite pas ; UM ne peut détecter, mais sait traiter lorsqu'il est prévenu.

6.2 Déclaration d'une exception

En JAVA, les exceptions sont des objets. Toute exception doit dériver (directement ou indirectement) de java.lang.Throwable :

[5] Il faudra donc toujours utiliser `Collections.nomMethode` pour invoquer ces méthodes

```
public abstract class Throwable {
  public String getMessage() ;
  public void printStackTrace() ;
  ...
}
```

La classe `Exception` dérive de `Throwable`. Elle sert en général de classe de base aux exceptions définies par l'utilisateur.

Exemple simple :

```
public class PileVide extends Exception {
  public PileVide () { super("empty stack !") ; }
}
```

6.3 Envoi d'une exception

CM doit, lorsqu'il prévoit d'envoyer une exception à UM, le déclarer dans le prototype de la méthode concernée. Il utilise pour cela un mot réservé, `throws`.

Exemple :

```
public class Pile {
  public Object depiler() throws PileVide { ... }
  ...
}
```

Lorsque, dans le code, la situation exceptionnelle survient, il lance alors effectivement l'exception par une clause `throw` :

```
public Object depiler() throws PileVide {
  if (top == null) throw new PileVide() ;
  ... // traitement normal
}
```

6.4 Gestion d'une exception

UM a alors le choix entre

- Ignorer l'exception. Elle est alors propagée au niveau du code appelant. Dans ce cas, tout se passe comme si c'était UM lui-même qui avait lancé l'exception, il doit donc l'avoir déclarée dans une clause `throws` dans le prototype de la méthode qui appelle le module.
- Traiter l'exception. Pour cela, il doit faire deux choses :
  - encapsuler le code "à risque" dans une clause `try`, ce qui permet d' "armer" la détection des exceptions (sinon, il y aura systématiquement transmission de l'exception au code appelant) ;
  - faire suivre la clause `try` d'une ou plusieurs clauses `catch`, chacune d'elle explicitant le traitement d'une exception particulière.

Exemple :

```
...          // exceptions non detectables
try {        // exceptions detectables
  Object o = maPile.depiler() ;
  doSomethingWith(o) ;
}
catch (PileVide pv) {
  ...        // traitement de l'exception en cas de pile vide
}
```

Lorsqu'une `PileVide` est lancée par CM (dans le code de la méthode `depiler()`), le contrôle est dérouté dans le code de UM : on sort immédiatement du `try` et le code correspondant au `catch` de `PileVide` est exécuté.

Lorsqu'un bloc `try` est susceptible de déclencher plus d'une exception - et qu'il est donc suivi de plusieurs `catch` -, l'ordre des exceptions traitées dans les `catch` est significatif. En effet, c'est le bloc correspondant à la *première* exception compatible avec l'exception lancée qui sera exécuté. Il est donc nécessaire, en cas de liens d'héritage entre les exceptions, de placer les exceptions *les plus précises d'abord* (classes dérivées).

À noter : une clause particulière, `finally`, peut être placée à la suite de l'ensemble des `catch`. Elle sera systématiquement exécutée, et permet donc une mise en facteur de code commun à l'ensemble des clauses `catch`.

7 Les threads

Un *thread* est un flôt de contrôle interne à un processus (*lightweight process*). Il se distingue des processus ordinaires par le fait que deux threads partagent leur espace mémoire (*i.e.* accéder aux mêmes données physiques), qui est celui du processus qui les contient, alors que les espaces sont dupliqués dans deux processus classiques.

7.1 Création d'un thread

En JAVA, les threads sont des objets. On peut créer un thread de deux façons différentes :
- en implémentant l'interface `java.lang.Runnable` dans une classe quelconque, et en instanciant la classe `java.lang.Thread` avec une instance de cette classe en paramètre de construction[6] ;
- en dérivant de la classe `java.lang.Thread`.

```
public interface Runnable {
  public void run() ;
}
public class Thread implements Runnable {
  public Thread() {...}
  public Thread(Runnable r) { ... }
  ...
}
```

On choisira la première solution quand la classe à définir doit hériter de quelque chose d'autre (par exemple de `java.applet.Applet`) : l'héritage étant simple en Java, on ne pourrait pas hériter aussi de `Thread`. Il faut alors créer une instance de `Thread` en l'initialisant à l'aide de notre objet comme paramètre du constructeur :

```
public class Exemple implements Runnable {
  ...
  public void run() { ... }
  public void declenche() {
    Thread t = new Thread(this) ;
    t.start() ;
  }
}
```

7.2 Etats d'un thread

7.2.1 Activation

Une fois créé, le thread doit être activé. Cette activation se fait par la méthode `start()`, qui déclenche automatiquement la méthode `run()`. Chaque thread est doté d'une priorité (de 1 à 10), qui peut être fixée à la création ou plus tard (*via* les méthodes `setPriority()` et `getPriority()`).

7.2.2 Suspension

Un thread actif peut être suspendu de plusieurs façons différentes :
- pendant une durée donnée, par la méthode `sleep()` ;
- pour permettre l'exécution d'un thread concurrent, par la méthode `yield()` ;
- par une interruption explicite, méthodes `interrupt()` et `isInterrupted()` (la méthode `stop()` est *deprecated* et ne doit pas être utilisée) ;
- en attente d'une opération d'E/S.

7.2.3 Indépendance / système d'exploitation

Dans la mesure où le mécanisme des threads se veut aussi indépendant des couches inférieures (en particulier l'O.S.) que possible, il doit pouvoir fonctionner de la même façon avec un OS préemptif (comme UNIX) et avec un OS coopératif (comme Windows 98). Dans ce but, un thread peut signaler qu'il est disposé à "rendre provisoirement la main" à un autre thread, via la méthode (statique) `yield()`.

[6] C'est la méthode à choisir la plupart du temps.

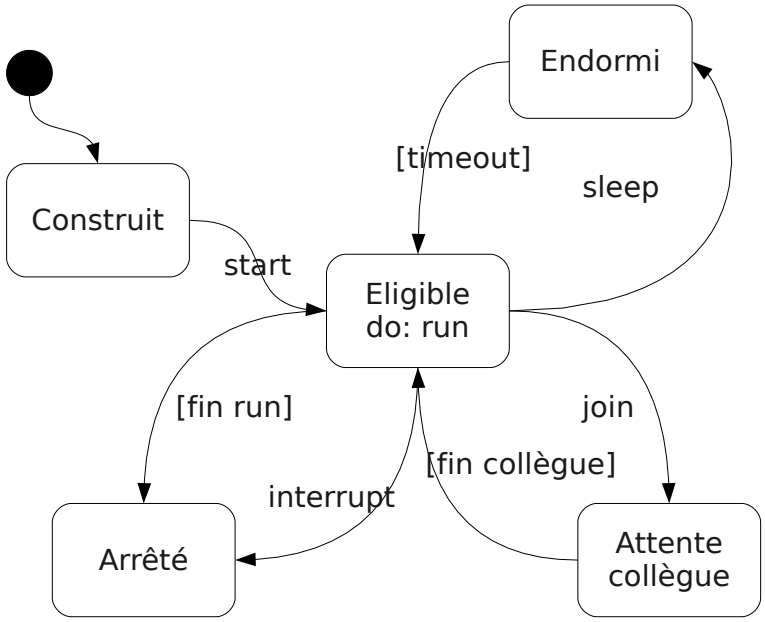


FIG. 2 – Les états possibles d'un thread

Le thread remplaçant sera choisi parmi les threads de même priorité.

Dans le même ordre d'idée, la méthode `join()` sert à attendre la fin d'un thread avant de reprendre le déroulement séquentiel des opérations du thread courant

7.2.4 Résumé

La plupart des méthodes décrites ci-dessus sont susceptibles de lancer l'exception `InterruptedException`, tout code utilisant des threads devra donc en tenir compte.

La figure 2 résume les différents états possibles pour un thread.

7.3 Terminaison d'un thread

La fin normale d'un thread est la fin de l'exécution de sa méthode `run()`. Néanmoins, un thread peut être interrompu par un autre thread. Lorsqu'on veut interrompre un thread on lui envoie un message `interrupt()`. Le thread destinataire peut tester s'il est interrompu au moyen de la méthode `isInterrupted()`.

Lorsque le thread est bloqué (par exemple pendant un `sleep()` ou un `wait()`) il reçoit une `InterruptedException`.

L'exemple simple suivant montre comment on peut tester dans un thread une demande d'interruption :

```
// $Id: TestInterrupt.java 1390 2007-12-18 10:40:09Z phil $

package pl.threads;

import java.io.Console;

/**
 * Thread termination using interrupt() / isInterrupted().
 * This simple program illustrates the use of interrupt() and isInterrupted()
 * to control thread termination. The user hit the RET key to start a
 * counting thread. The threads runs until another RET is hit. <p>
 * Use{@link java.io.Console} to ease keyboard usage.
 * @author laroque@u-cergy.fr
 * @version $Rev: 1390 $
 */
public class TestInterrupt extends Thread {
  /** Keep counting until interrupted (no sleep) */
```

```
  public void run() {
    int i = 0;
    while (!isInterrupted()) {
      i++;
      if (i%10000000 == 0) System.out.println(i);
    }
  }

  /** creates and start a thread, then wait for input to interrupt it */
  public static void main(String ... args) {
    TestInterrupt ti = new TestInterrupt();
    Console cons = System.console();
    cons.printf("Hit RETURN to start thread : ");
    String s = cons.readLine();
    ti.start();
    cons.printf("Hit RETURN again to interrupt thread when bored%n");
    s = cons.readLine();
    ti.interrupt();
    cons.printf("bye bye%n", s);
  }
}
```

7.4 Notion de concurrence

L'exemple qui suit montre - dans un contexte très simple - qu'une fois lancé, un thread s'exécute en concurrence avec les autres threads existants.

Dans cet exemple, on définit un objet contenant un `Compteur`, dont la valeur initiale est donnée sur la ligne de commande. Cet objet lance deux threads, l'un chargé d'incrémenter le compteur, l'autre de le décrémenter. Le programme s'arrête lorsque le `Compteur` est redescendu à 0 (le thread de décrémentation a "gagné"), ou lorsqu'il a doublé sa valeur initiale (le thread d'incrémentation a gagné) :

```
// $Id: SimpleThreads.java 1590 2008-11-19 09:28:58Z phil $

package pl.threads;

import pl.base.Compteur;

/** Illustration de la notion de programme multi-threadé.
 * Ce programme crée deux threads qui travaillent sur un Compteur commun.
 * L'un des threads incrémente aléatoirement le compteur, l'autre le décrémente.
 * La valeur initiale du compteur est donnée par la ligne de commande
 * (par défaut, voir {@link #DEFVAL}). Lorsque le compteur
 * atteint 0 ou le double de sa valeur initiale, le programme stoppe.
 * <p>
 * On trace les deux threads pour mettre en évidence leur imbrication
 * <p>
 * @author laroque@u-cergy.fr
 * @version $Revision: 1590 $
 */
public class SimpleThreads {

  /** la valeur par défaut du compteur (si aucun paramètre
   * sur la ligne de commande)
   */
  public static final int DEFVAL = 5;

  /** les deux threads concurrents */
  protected Thread decr, incr;
```

```
  /** valeur initiale du compteur */
  protected int initValue;

  /** le compteur */
  protected Compteur c;

  /** construction : création des threads et initialisation du compteur
   * @param iv la valeur initiale du compteur
   */
  public SimpleThreads(int iv) {
    initValue = iv;
    c = new Compteur();
    for (int i = 0; i < iv; i++) c.incr();
    decr = new Decr(c);
    incr = new Incr(c);
  }

  /** le main() admet un paramètre sur la ligne de commande: la valeur initiale
   * du compteur
   */
  public static void main(String args[]) {
    int value = DEFVAL;
    try {
      value = Integer.parseInt(args[0]);
    } catch (Exception e){}

    SimpleThreads st = new SimpleThreads(value);

    System.out.println("Initial value = " + value);

    // démarrage des deux threads
    st.decr.start();
    st.incr.start();

    // on attend qu'une des limites soit atteinte
    while (st.c.valeur() > 0 && st.c.valeur() < 2 * st.initValue);

    // interruption des deux threads
    st.decr.interrupt();
    st.incr.interrupt();
    System.out.println("Fin du programme sur value = " + st.c.valeur());
  }
}

/** Récupère un Compteur et le décrémente aléatoirement
 * jusqu'à ce qu'on l'interrompe
 */
class Decr extends Thread {
  protected Compteur c;
  public Decr(Compteur c) { this.c = c; }
  public void run() {
    while (!isInterrupted()) {
      // attente d'1/2s max.
      try { sleep((int)(500 * Math.random())); }
      catch (InterruptedException e) { return; }
      c.decr();
      System.out.println("Decr: value == " + c.valeur());
    }
```

```
  }
}

/** Récupère un Compteur et l'incrémente aléatoirement
 * jusqu'à ce qu'on l'interrompe
 */
class Incr extends Thread {
  protected Compteur c;
  public Incr(Compteur c) { this.c = c; }
  public void run() {
    while (!isInterrupted()) {
      // attente d'1/2s max.
      try { sleep((int)(500 * Math.random())); }
      catch (InterruptedException e) {return; }
      c.incr();
      System.out.println("Incr: value == " + c.valeur());
    }
  }
}
```

7.5 Notions de synchronisation

N.B. : cette section aborde des notions qui dépassent le cadre de la seconde année de licence, et peut être ignorée lors d'une première lecture.

Le multi-threading entraîne des risques de dysfonctionnements accrus du code, principalement en raison des accès concurrents aux ressources, et de l'ordre en général imprévisibles dans lequel se font ces accès.

7.5.1 Sections critiques

JAVA permet de définir des sections de code *critiques* ou *atomiques*, c'est-à-dire des sections de code dans lesquelles un seul thread à la fois peut se trouver, et qui ne sont pas interruptibles.

Une section atomique peut être un objet (c'est-à-dire l'ensemble des méthodes de sa classe), une méthode, ou même un ensemble d'instructions à l'intérieur d'une méthode. Dans tous les cas, on doit utiliser la clause `synchronized`. Dès qu'une portion de code est ainsi synchronisée, la JVM crée et maintient un *moniteur* dessus.

Exemples :

```
synchronized(o) {
  doSomethingWith(o);
}

public synchronized void maMethode() { ... }
```

Tous les threads tentant d'accéder à la portion de code verrouillée sont mis en attente automatiquement.

Il est évident que si le recours à cette possibilité permet de sécuriser le code, il le rend aussi beaucoup moins efficace ; il est donc nécessaire de réfléchir à ce qui doit effectivement être synchronisé et de ne pas synchroniser n'importe quelle partie du code...

7.5.2 Synchronisation par `wait/notify`

Il peut arriver que la section critique ne soit pas un mécanisme suffisant pour assurer l'ordre dans lequel les opérations s'effectuent : elle n'assure en fait que la séquentialité des accès, par leur ordre. Pour aller plus loin, il faut disposer d'un mécanisme permttant à un thread d'attendre un certain état du système avant de continuer son exécution. C'est précisément le rôle des deux méthodes `wait()` et `notify()`. Ces méthodes ne sont pas définies dans la classe `Thread` mais directement dans `Object`, ce qui permet à un thread d'attendre sur une condition portant sur n'importe quel objet.

Quand un thread est suspendu par un `wait()`, il devient non éligible. Tant que l'objet receveur du `wait` n'a pas reçu le message `notify()`, le thread est suspendu. Il devient à nouveau éligible, à la réception du `notify`, s'il est le premier à avoir été suspendu. En effet, il est possible que plusieurs threads

attendent le même événement ; dans ce cas, ils sont placés dans une file d'où ils ressortiront à leur tour, après le nombre adéquat de `notify`.

7.5.3 Un exemple de synchronisation

Le code ci-dessous décrit un mécanisme simple de lecteur – écrivain : le lecteur ne peut lire que s'il y a quelque chose de nouveau écrit par l'écrivain, et ce dernier ne peut rien écrire de nouveau tant que le lecteur n'a pas lu la dernière donnée écrite.

La synchronisation est assurée à bas niveau par un objet "bidule" (`Gizmo`), qui contient la donnée écrite et à lire lorsqu'il y en a une : il sert donc de relai entre l'écrivain et le lecteur. On notera que, de ce fait, ces derniers n'ont aucun effort de synchronisation à faire et se contentent, l'un d'écrire ce qu'il a à écrire, l'autre de lire sans se préoccuper explicitement de savoir si quelque chose est disponible.

```
// $Id: TestThreads.java 821 2006-11-14 10:04:18Z phil $

package pl.threads;

/** example of thread synchronization in Java.
 * BASICS: <ul>
 * <li> objects can be monitored. Defining synchronized methods in a class
 * automatically locks the object: only one thread can enter the method at a
 * time.
 * <li> the stop() method is unsafe, it is necessary to make sure the run()
 * cannot loop indefinitely
 * <li> the wait()/notifyAll() pair acts as a P/V pair for semaphores
 *</ul>
 * inspired from SUN's java tutorial<p>
 * A reader/writer pair (R/W) communicate via a Gizmo (G). W sometimes writes
 * integer values in G, for R to read them. W can only write when R has read
 * the last entered value, and R cannot read twice the same value without W
 * re-writing it. <p>
 * Synchronization is entirely handled by G.
 * @author laroque@u-cergy.fr
 * @version $Rev: 821 $
 */
public class TestThreads {
  Reader l;
  Writer e;
  Gizmo b;
  public TestThreads() {
    b = new Gizmo();
    l = new Reader(b);
    e = new Writer(b);
    l.start();
    e.start();
    try {l.join();} catch (InterruptedException ie) {}
    System.out.println("finished");
  }
  public static void main(String args[]) {
    new TestThreads();
  }
}

/** Handles low-level synchronisation between Reader and Writer.
 * read() and write() methods are synchronized to make sure only
 * one thread accesses them at a time. <p>
 * We use the wait/notifyAll pair to avoid active wait in both
 * methods.
 * @author Philippe.Laroque@dept-info.u-cergy.fr
```

```
 * @version $Revision: 821 $
 */
class Gizmo {
  private int value;  // current value in the gizmo
  private boolean available; // this value can be read?
  Gizmo () { available = false; }
  /** wait until the value is available. The call to wait() makes current
   * thread uneligible, so it does not compete for the monitoir's lock. <p>
   * Woken up by a call to notifyAll (in write, @see Gizmo#write).
   * @return the value, as soon as it's available
   */
  synchronized int read() {
    if (!available)
      try { wait(); } catch (InterruptedException ie) {}
    available = false; // not to be read again
    notifyAll();  // to wake writer up
    return value;
  }
  /** wait until the value as been read. The call to xait() makes current
   * thread uneligible, so it does not compete for the monitoir's lock. <p>
   * Woken up by a call to notifyAll (in read, @see Gizmo#read).
   * @param val the new value in the gizmo
   */
  synchronized void write(int val) {
    if (available)
      try { System.err.print("w");wait(); } catch (InterruptedException ie) {}
    value = val;  // replaces old value
    available = true; // can be read now
    notifyAll();  // to wake reader up
  }
}

/** Read a value form a Gizmo when it is available.
 * Synchronization work is entirely handled by the gizmo (@see Gizmo),
 * so this code is straightforward.
 * @author Philippe.Laroque@dept-info.u-cergy.fr
 * @version $Revision: 821 $
 */
class Reader extends Thread {
  Gizmo b;
  Reader(Gizmo b) { this.b = b; }
  public void run() {
    for (int i = 1; i <= 10; i++) {
      System.out.println("Just read : " + b.read());
    }
  }
}

/**
 * Synchronization work is entirely handled by the gizmo (@see Gizmo),
 * so this code is straightforward.
 * @author Philippe.Laroque@dept-info.u-cergy.fr
 * @version $Revision: 821 $
 */
class Writer extends Thread {
  Gizmo b;
  Writer(Gizmo b) { this.b = b; }
  public void run() {
```

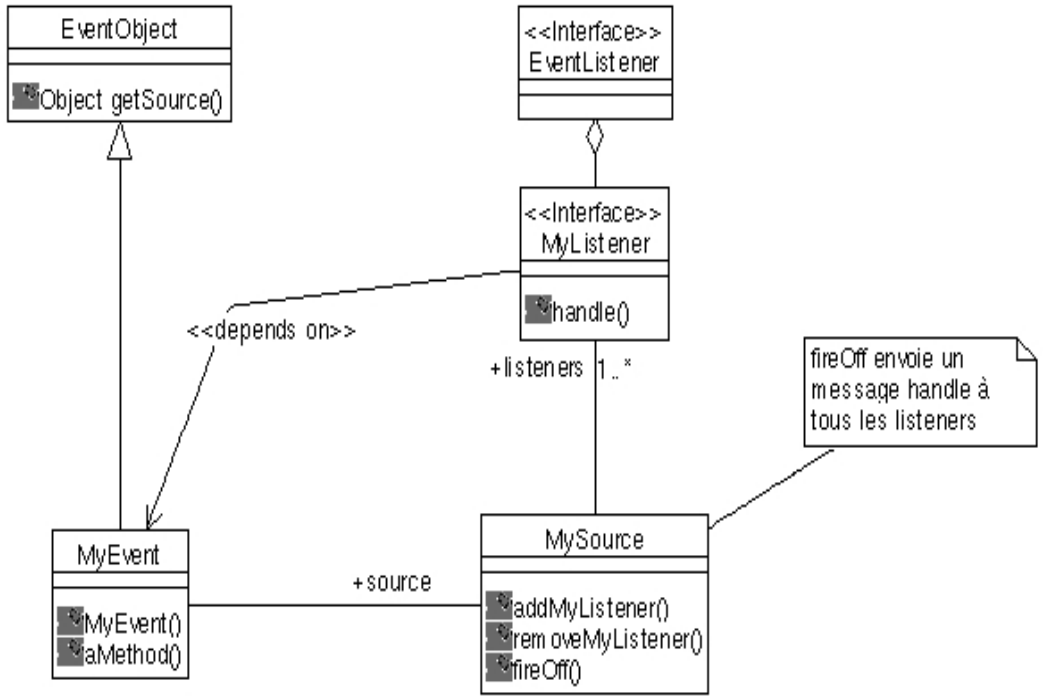


FIG. 3 – Le mécanisme d'écoute des événements

```
    for (int i = 1; i <= 10; i++) {
      b.write(2*i);
      try { sleep( (int)(1000 * Math.random())); }
      catch (InterruptedException ie) {}
    }
  }
}
```

8 Les interfaces graphiques

Ecrire un mécanisme portable de gestion d'objets graphiques n'est pas une mince affaire. JAVA y parvient dans des limites raisonnables, même si çà et là on perçoit quelques différences d'une plate-forme à l'autre.

Le principe de fonctionnement de cette gestion est la programmation événementielle, qui est décrite à la section suivante. Une liste des principaux composants graphiques et des événements auxquels ils sont sensibles est ensuite présentée. La section se termine avec un survol du principe de placement automatique des composants via les *layout managers*.

8.1 Programmation événementielle

8.1.1 Introduction

Le principe de la programmation événementielle dépasse le cadre du traitement des objets graphiques en JAVA, mais il est très employé dans le domaine de la réalisation d'IHM, et nous resterons dans ce cadre ici.

L'idée est de renoncer à un algorithme linéaire classique (avec contrôle centralisé par le programme principal), au profit d'une distribution du contrôle vers les différents composants du programme. Par exemple, un bouton est inactif tant que l'utilisateur ne le presse pas ; en revanche, dès que c'est le cas l'algorithme doit réagir.

Pour permettre cette réactivité, un système de *publication* et d'*abonnements* à des *événements* est mis en place :

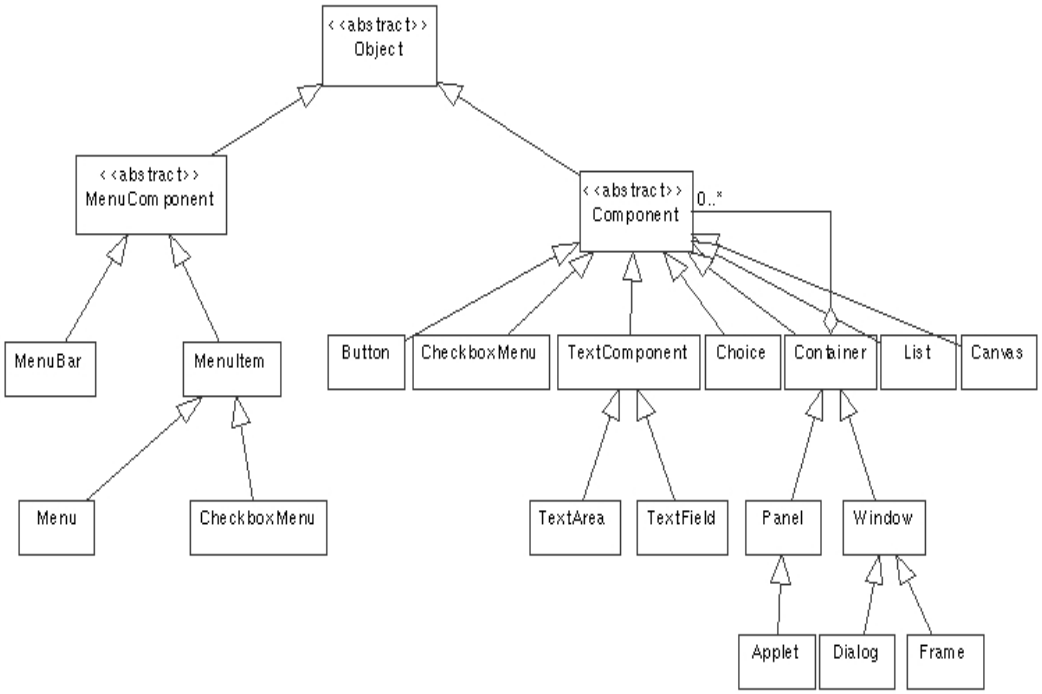


FIG. 4 – Les principaux composants de l'AWT

- Quand un objet dépend d'actions sur un composant graphique, il se met en attente d'évenements en provenance de cet objet (on dira qu'il l'*écoute*).
- Lorsqu'un objet graphique est sollicité (par exemple un clic sur un bouton, une modification dans un champ texte...), il émet un événement avertissant de cette sollicitation.
- Tous les écouteurs de l'objet sont alors avertis : ils reçoivent l'événement émis. En JAVA, un événement est un objet, aussi il peut véhiculer une quantité arbitraire d'information permettant aux écouteurs de prendre les décisions correctes en fonction de la nature de l'évenement.

La bibliothèque standard du JDK renferme donc la définition de trois hiérarchies :

- une hiérarchie d'objets graphiques ;
- une hiérarchie d'événements émis par ces objets ;
- une hiérarchie d'écouteurs attentifs à ces événements.

Un résumé de ce mécanisme de publication et d'écoute est reproduit en figure 3.

8.2 Composants et événements liés aux interfaces graphiques

8.2.1 Les principaux composants

Les composants graphiques de base font partie de l'AWT (Abstract Windowing Toolkit). On les retrouve dans le paquetage `java.awt` et dans quelques sous-paquetages. Une hiérarchie des principaux composants est résumée à la figure 4.

8.2.2 Les principaux événements et écouteurs

Chaque composant est susceptible d'émettre un ou plusieurs événements. A chaque type d'événement (dont les principaux sont reproduits à la figure 5) correspond un écouteur (listés en relation avec les sources des événements qu'ils écoutent, figure 6).

Si les événements sont des classes, chaque écouteur est implémenté en JAVA sous forme d'une interface. Ceci oblige donc à définir la totalité des méthodes déclarées dans l'interface lorsqu'on désire écouter un événement donné. Pour certains écouteurs, comme `ActionListener`, ce n'est pas un problème (une seule méthode), mais pour d'autres plus riches, cela peut réduire l'intérêt porté à l'interface.

Pour résoudre ce problème, le JDK fournit pour les interfaces riches des classes appelées *adapters*, qui implémentent ces interfaces avec un comportement de base (en général, les méthodes ne font rien).

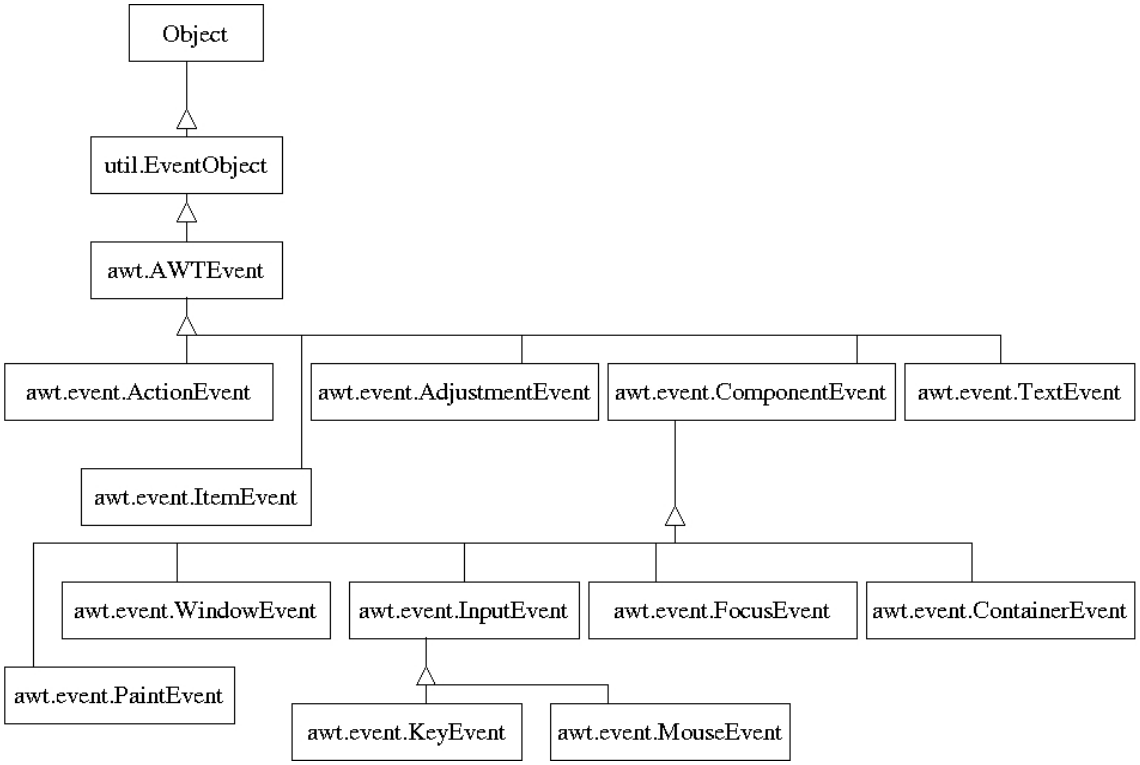


FIG. 5 – Les principaux événements de l'AWT

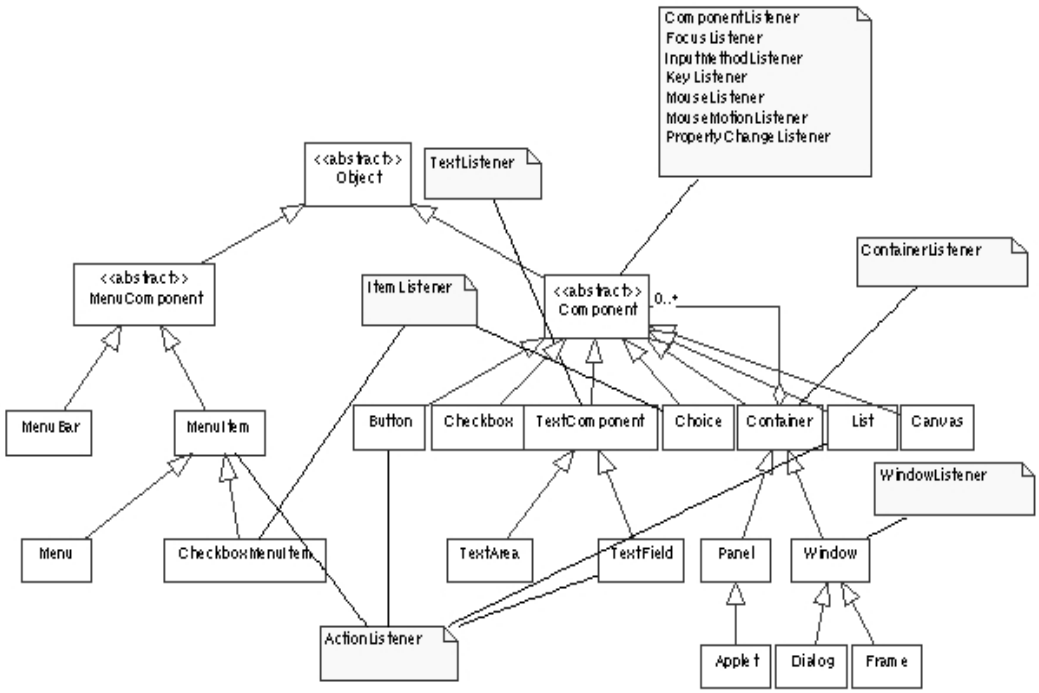


FIG. 6 – Les écouteurs de l'AWT

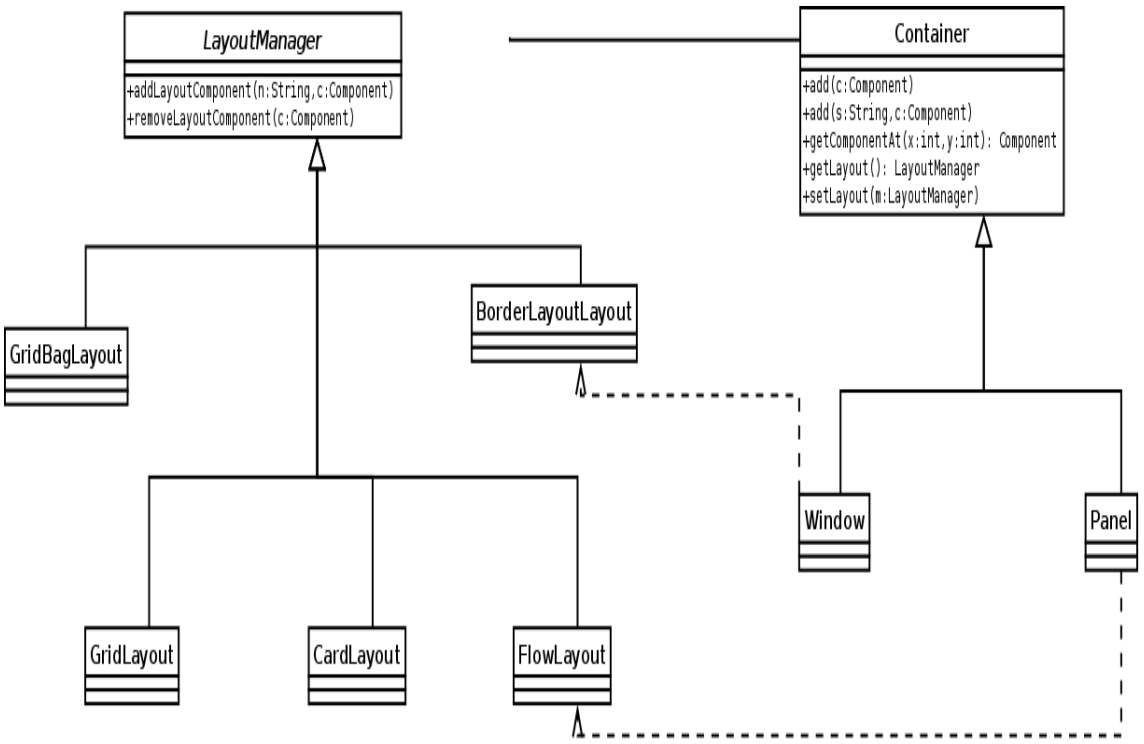


FIG. 7 – Les types de layout managers

Lorsqu'une classe désire écouter un événement donné, elle peut hériter de l'*adapter* adéquat. Elle n'a alors qu'à spécifier la ou les quelques méthodes correspondant à son comportement spécifique, ce qui facilite grandement le travail et réduit la taille du code à produire.

8.3 Les layouts

Le placement des composants dans une IHM n'est pas chose aisée, dès lors qu'on souhaite favoriser la portabilité. Les concepteurs de JAVA ont défini un mécanisme permettant d'abstraire cette tâche de placement à l'intérieur d'objets appelés des *layout managers*. Un tel objet se caractérise en général par une ou plusieurs méthodes permettant d'ajouter un composant, la stratégie de placement étant encapsulée dans la méthode d'ajout. La figure 7 indique le layout manager associé par défaut aux divers conteneurs de l'AWT. Bien entendu, le programmeur est libre de choisir un *layout manager* différent (méthode `setLayout()`).

9 Les applets

Les applets sont des applications Java particulières à plusieurs titres :

- Elles n'ont pas de méthode `main()`, et ne peuvent donc pas être lancées via une invocation directe de la machine virtuelle, comme n'importe quelle autre application Java.
- Elles sont destinées à tourner dans un environnement de type "browser web" : lorsque la page web qui contient l'appel à l'applet est chargée sur le browser, le code de l'applet est téléchargé sur le client depuis le serveur web qui détient la page. Ce code est ensuite exécuté par la JVM contenue dans le browser du client.
- Pour des raisons faciles à comprendre, il est nécessaire de brider les possibilités d'action de ces codes extérieurs. Ainsi, une applet possède un champ restreint d'opérations. Par exemple, elle ne peut ouvrir un fichier sur le disque local du client (sauf mise en place d'une politique de sécurité particulière, dont l'exposé dépasse le cadre de ce cours d'introduction. Pour plus de renseignements, le lecteur intéressé peut avec profit consulter
  `http://java.sun.com/docs/books/tutorial/applet/overview/index.html`

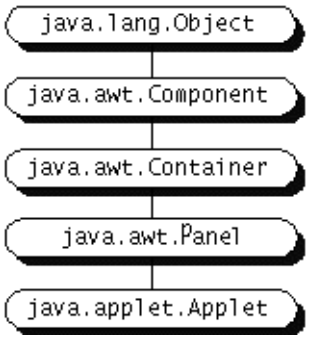


FIG. 8 – La classe Applet et sa hiérarchie

9.1 Création d'une applet

La création d'une applet se fait en trois étapes :

1. Edition du fichier source. La classe créée doit dériver de `java.applet.Applet` (voir figure 8).
2. Compilation du fichier source.
3. Edition de la page HTML avec insertion d'un ordre de lancement de l'applet. Cet ordre est donné via une balise particulière, `APPLET`.

Voici un exemple de page HTML simple lançant une applet :

```
<HTML>
<HEAD>
<TITLE> A Simple Program </TITLE>
</HEAD>
<BODY>

Voici une execution d'applet :
<APPLET CODE="HelloWorld"
  CODEBASE="/path/to/classes"
  WIDTH=150 HEIGHT=25
  ALT="Si vous voyez ca, c'est que l'applet n'a pu se lancer">
<param name="NOM" value="phil">
Ce navigateur ignore le tag "APPLET" </APPLET>
</BODY>
</HTML>
```

9.2 Cycle de vie d'une applet

N'ayant pas de `main`, l'applet doit fournir à la JVM qui l'accueille un "point d'entrée". C'est la méthode `init()` qui remplit ce rôle. Cette méthode est appelée juste après l'instanciation, donc juste après l'invocation du constructeur s'il existe. La différence entre `init()` et le constructeur vient de ce que certains browsers permettent de recharger une applet. Dans ce cas, l'applet invoque à nouveau la méthode `init()`, mais évidemment pas le constructeur...

Lorsque la page contenant l'applet est remplacée dans le browser par une autre (ou lorsque le browser est iconifié), l'applet reçoit le message `stop()`. Réciproquement, lorsque la page réapparaît, l'applet reçoit le message `start()`.

Enfin, lorsque le browser est fermé, l'applet est détruite, via la méthode `destroy()`.

Ces étapes sont résumées dans la figure 9.

9.3 Applets archivées

Les applets très simples sont autonomes, en ce sens qu'elles se contentent de faire appel à une classe pour fonctionner. Dans les cas plus réalistes, il faut faire collaborer plusieurs classes pour faire fonctionner l'applet. Il est alors impossible, dans la balise `APPLET` du fichier html, de se contenter des paramètres `CODE` et `CODEBASE`.

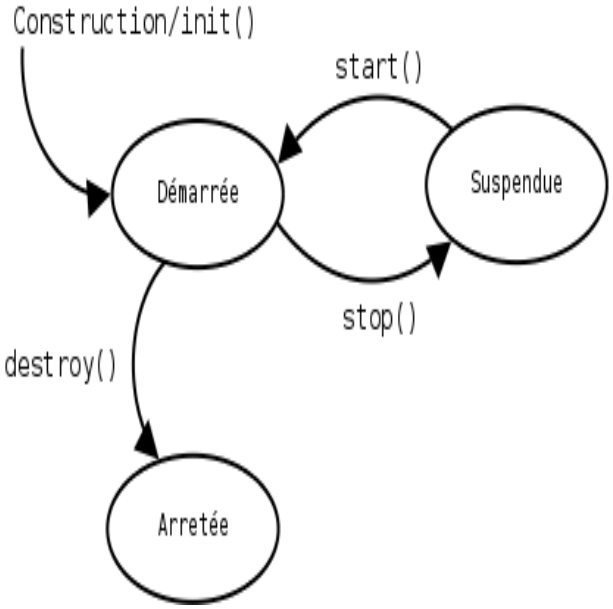


FIG. 9 – Le cycle de vie d'une applet

La solution dans un tel cas consiste à grouper l'ensemble des classes nécessaires dans une archive et à mentionner cette archive dans la balise `APPLET`, à l'aide du paramètre `ARCHIVE` :

```
<applet
    code="monPackage.MonApplet"
    archive="monArchive.jar"
    alt="Si vous voyez ca, c'est que l'applet n'a pu se lancer"
    height="180"
    width="450">
</applet>
```

9.3.1 Manipulation d'archives

Les archives se manipulent à l'aide de la commande `jar` (pour *Java ARchive*). Cette commande sert
- à créer une archive (extension `.jar`) à partir de fichiers et de répertoires (stockage récursif dans ce dernier cas) : option "c" (create) ;
- à lister le contenu d'une archive : option "t" ;
- à extraire de l'archive tout ou partie de son contenu : option "x".

Par exemple, supposons que l'on ait une applet dont le nom est `MonApplet`, qui appartient au paquetage `monPaquetage` et a besoin de classes du paquetage `autrePaquetage`. Pour créer l'archive qui va contenir le code nécessaire, il faut se rendre dans le répertoire où sont stockés les fichiers `''.class''` et entrer la commande suivante :

```
jar cvf monArchive.jar monPaquetage autrePaquetage
```

Ceci va créer le fichier `monArchive.jar` qui va contenir le code nécessaire à l'exécution de l'applet. Les options "v" et "f" sont respectivement là pour afficher des messages au fur et à mesure de la construction de l'archive (mode *verbose*) et pour indiquer le *nom* du fichier archive, `monArchive.jar`. On peut vérifier le contenu de cette archive par la commande

```
jar tvf monArchive.jar
```

Enfin, on peut reconstituer les répertoires d'origine (les répertoires `monPaquetage` et `autrePaquetage`) en les désarchivant, par la commande

```
jar xvf monArchive.jar
```

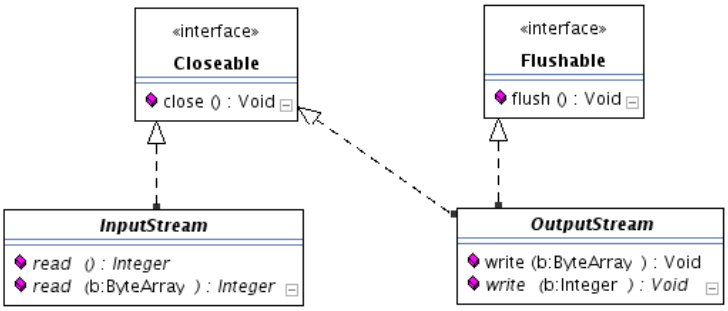


FIG. 10 – Les classes d'E/S sur des octets

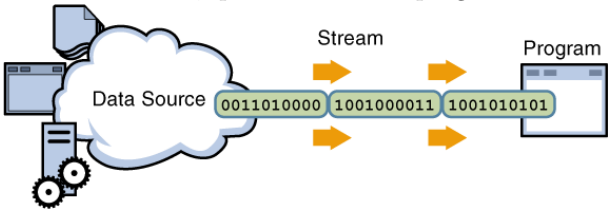
10 Les entrées - sorties

10.1 Notion de flux - types de flux

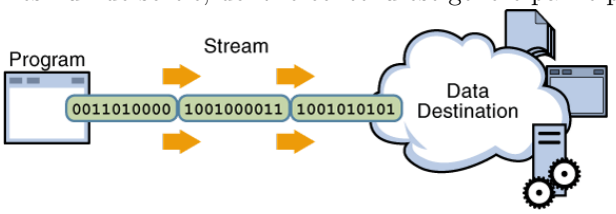
En JAVA (comme sous UNIX par exemple) les E/S se font sur des flux (*streams*), que l'on peut voir comme une généralisation de la notion de fichier : un flux peut représenter un fichier, un périphérique, une zone mémoire etc.

On distingue deux types de flux :

1. Les flux d'entrée, qui alimentent le programme en données



2. Les flux de sortie, dont le contenu est généré par le programme.

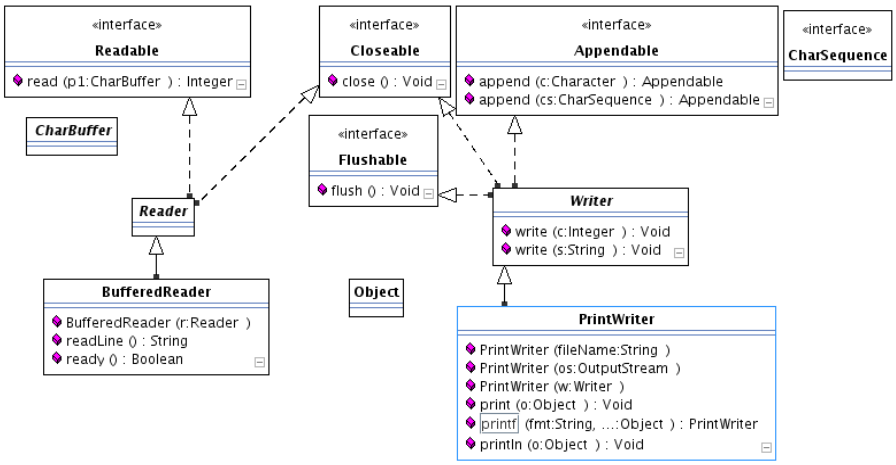


10.2 E/S de bas niveau : les byte streams

On les utilise quand les données ne sont pas formatées : les E/S sont alors effectuées un octet à la fois. Les classes de base, `InputStream` et `OutputStream`, sont représentées figure 10. Elles sont abstraites, font partie du paquetage `java.io` et toutes les méthodes de la figure lancent l'exception `java.io.IOException`.

Le programme suivant, tiré du site de SUN, illustre le fonctionnement de deux des classes dérivées de ces classes abstraites, FileInputStream et FileOutputStream. On notera la raison pour laquelle les fonctions de lecture renvoient un entier et non un simple octet, la valeur -1 indiquant la fin du flux :

```
import java.io.*;
public class CopyBytes {
  public static void main(String[] args) throws IOException {
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
      in = new FileInputStream("in.txt");
      out = new FileOutputStream("out.txt");
      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
```



30

FIG. 11 – Les flux de caractères

```
      }
    } finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
        out.close();
      }
    }
  }
}
```

10.3 Cas général : les flux typés

En général, lorsqu'on a une idée précise du contenu du flux que l'on doit lire / écrire, il est préférable d'utiliser des flux typés, car leurs primitives sont de plus haut niveau. Les flux de caractères par exemple tiennent automatiquement compte de la localisation. Ainsi, on pourrait reprendre le programme précédent de manière identique en remplaçant juste les types `FileInputStream` et `FileOutputStream` respectivement par `FileReader` et `FileWriter` (qui dérivent des classes `Reader` et `Writer`, voir figure 11). Ceux-ci permettent néanmoins des E/S plus sophistiquées, notamment par la prise en compte de la notion de ligne (saisie) et de format (affichage). On peut alors transformer ainsi le programme précédent :

```
import java.io.*;
public class CopyLines {
  public static void main(String[] args) throws IOException {
    BufferedReader in = null;
    PrintWriter out = null;
    try {
      in = new BufferedReader(new FileReader("in.txt"));
      out = new PrintWriter(new FileWriter("out.txt"));
      String l;
      while ((l = in.readLine()) != null) {
        out.println(l);
      }
    } finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
        out.close();
      }
```

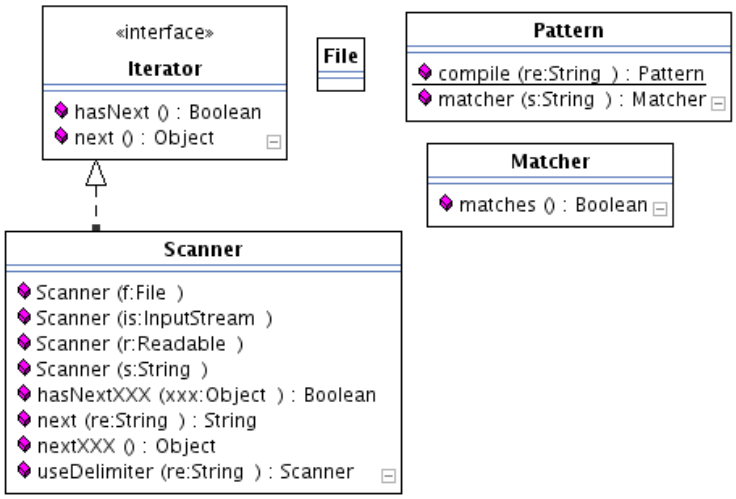


FIG. 12 – La classe Scanner pour l'analyse lexicale simple

```
    }
  }
}
```

10.4 Formatage des E/S

Il est possible de sophistiquer le traitement des E/S, notamment en utilisant la classe `Scanner` en entrée (voir figure 12) pour une analyse lexicale simple et les classes `PrintWriter` (caractères) et `PrintStream` (octets, similaire à PrintWriter pour les fonctionnalités) en sortie. À noter, dans ce dernier cas, l'utilisation de "%n" au lieu de "\n" comme c'est le cas en C par exemple. En effet, "\n" provoque systématiquement un *linefeed*, donc est dépendant du système d'exploitation.

Le programme qui suit illustre l'utilisation d'un tel objet pour ne comptabiliser, lors de la lecture d'un flux de type texte, que les nombres, afin d'en faire la somme[7] :

```
import java.io.*;
import java.util.*;
public class ScanSum {
  public static void main(String[] args) throws IOException {
    Scanner s = null;
    double sum = 0;
    try {
      s = new Scanner(new BufferedReader(new FileReader("numbers.txt")));
      s.useLocale(Locale.US);
      while (s.hasNext()) {
        if (s.hasNextDouble()) {
          sum += s.nextDouble();
        } else {
          s.next();
        }
      }
    } finally {
      s.close();
    }
    System.out.printf("Sum = %10.3f%n",sum);
  }
}
```

[7] On notera l'indication explicite de la Locale, indispensable puisque le format des nombres est très variable...

10.5 Gestion du clavier

La classe `java.io.Console` permet de gérer les E/S plus facilement qu'en utilisant des flux classiques. Elle possède entre autres des méthodes de saisie de ligne, avec possibilité de masquer le texte entré (saisie de mot de passe). Voici un exemple simple d'utilisation de cet outil :

```
import java.io.*;
import java.util.*;
public class Password {
  public static void main (String args[]) throws IOException {
    Console c = System.console();
    if (c == null) {
      System.err.println("No console.");
      System.exit(1);
    }
    String login = c.readLine("Enter your login : ");
    char [] oldPassword = c.readPassword("Enter your old password : ");
    if (verify(login, oldPassword)) {
      boolean noMatch;
      do {
        char [] newPassword1 = c.readPassword("Enter your new password : ");
        char [] newPassword2 = c.readPassword("Enter new password again : ");
        noMatch =! Arrays.equals(newPassword1, newPassword2);
        if (noMatch) {
          c.format("Passwords don't match. Try again.%n");
        } else {
          change(login, newPassword1);
          c.format("Password for %s changed.%n", login);
        }
        Arrays.fill(newPassword1, ' ');
        Arrays.fill(newPassword2, ' ');
      } while (noMatch);
    }
    Arrays.fill(oldPassword, ' ');
  }
  //Dummy verify method.
  static boolean verify(String login, char[] password) {
    return true;
  }
  //Dummy change method.
  static void change(String login, char[] password) {}
}
```

10.6 E/S binaires

Les E/S binaires (sur les types simples et les `String`) sont réalisées à partir de classes qui implémentent les interfaces `DataInput` et `DataOutput` (voir figure 13), notamment `DataInputStream` et `DataOutputStream`.

L'exemple suivant montre comment utiliser ces flux dans le cas des sorties. Le as des entrées s'en déduit immédiatement, chaque méthode de type `writeXXX` ayant son équivalent en `readXXX` :

```
static final String dataFile = "invoicedata";
static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
static final int[] units = { 12, 8, 13, 29, 50 };
static final String[] descs = {
  "Java T-shirt",
  "Java Mug",
  "Duke Juggling Dolls",
  "Java Pin",
  "Java Key Chain"
};
```

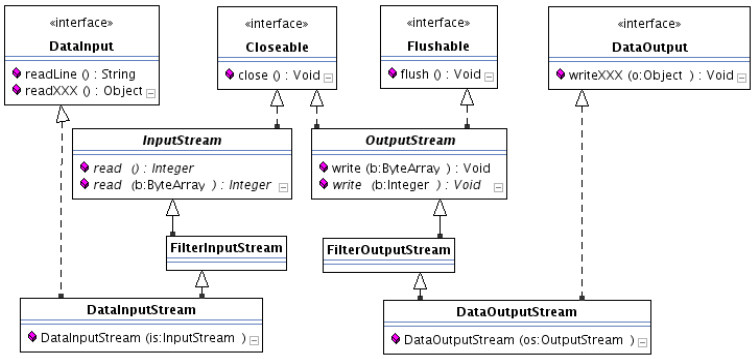


FIG. 13 – Les flux d'E/S binaires

```
out = new DataOutputStream (new BufferedOutputStream
        (new FileOutputStream (dataFile))) ;
for (int i = 0 ; i < prices.length ; i ++) {
    out.writeDouble(prices[i]) ;
    out.writeInt(units[i]) ;
    out.writeUTF(descs[i]) ;
}
```

10.7 Cas particulier important : la sérialisation

Le cas de la gestion des objets dans les fichier est original et mérite qu'on s'y attarde quelque peu. Le problème de la sauvegarde d'un objet dans un fichier est bien entendu la conservation de ses liens avec d'autres objets du système dynamique auquel il appartient.

10.7.1 Le principe

La solution mise en œuvre en Java consiste à sérialiser l'objet (c'est-à-dire à le transformer en une chaîne d'octets que l'on peut placer dans un fichier) ainsi que tous les objets référencés par les attributs de l'objet sauvegardé, et ainsi de suite jusquà obtenir la fermeture transitive de la relation, c'est-à-dire l'ensemble des objets liés, directement ou indirectement, à celui que l'on désire sauvegarder.

Lors de la restauration, l'objet est ainsi recréé avec l'intégralité de ses liens. On peut considérer que la séralisation s'apparente à une "photographie" de l'objet et de tous les objets auxquel il est lié. Les classes qui gèrent cette sérialisation sont `ObjectInputStream` et `ObjectOutputStream`. Pour qu'un objet soit sérialisable, il faut et il suffit qu'il implémente l'interface `java.io.Serializable` (une interface vide, implémentée par la plupart des classes de l'API Java).

10.7.2 Exemple de fonctionnement

Sérialisation de deux objets :

```
FileOutputStream out = new FileOutputStream("theTime") ;
ObjectOutputStream s = new ObjectOutputStream(out) ;
s.writeObject("Today") ;
s.writeObject(new Date()) ;
s.flush() ;
```

Récupération de deux objets sérialisés (dans l'ordre où ils ont été écrits !) :

```
FileInputStream in = new FileInputStream("theTime") ;
ObjectInputStream s = new ObjectInputStream(in) ;
String today = (String)s.readObject() ;
Date date = (Date)s.readObject() ;
```

Table des matières