Introduction à la programmation orientée objets en JAVA

Olivier Sigaud

Edition 2005-2006

Table des matières

1 Introduction **5**
- 1.1 Vocation de ce document 5
- 1.2 De la programmation structurée à la programmation orientée objets . . 6

2 Des objets et des classes **8**
- 2.1 Introduction 8
- 2.2 Les classes 8
  - 2.2.1 Nature des classes 8
  - 2.2.2 Éléments de syntaxe 10
  - 2.2.3 Conventions de notation 11
  - 2.2.4 Un exemple commenté 11
- 2.3 Les attributs 12
- 2.4 Les méthodes 14
  - 2.4.1 Signature et surcharge 14
  - 2.4.2 Des fonctions aux méthodes 15
- 2.5 Les membres statiques 16
  - 2.5.1 Les attributs statiques 16
  - 2.5.2 Les méthodes statiques 17

3 Cycle de vie des objets **18**
- 3.1 Introduction 18
- 3.2 Allocation mémoire et construction des objets 18
  - 3.2.1 Constructeur par défaut 19
  - 3.2.2 `this` 20
  - 3.2.3 Constructeurs de copie 20

MCours.com

3.2.4 Les clones . . . 23
3.2.5 Passage de paramètres . . . 25
3.3 Initialisation . . . 26
3.4 Recyclage de la mémoire . . . 26

4 L'encapsulation 27
4.1 Introduction . . . 27
4.2 Le contrôle d'accès . . . 28
4.2.1 Contrôle d'accès aux classes au sein d'un package . . . 28
4.2.2 Contrôle d'accès aux attributs et méthodes . . . 28
4.3 Les accesseurs . . . 29
4.4 Les packages . . . 29
4.4.1 Intérêt des packages . . . 29
4.4.2 Usage des packages . . . 30
4.4.3 `classpath` . . . 31

5 L'héritage 32
5.1 Introduction . . . 32
5.1.1 L'instruction `super()` . . . 33
5.1.2 Le mot-clé `super` . . . 34
5.2 L'abstraction . . . 34
5.2.1 Les méthodes abstraites . . . 34
5.2.2 La redéfinition . . . 35
5.2.3 Les classes abstraites . . . 36
5.3 La classe `Object` . . . 36
5.3.1 Les méthodes de la classe `Object` . . . 36
5.4 Le polymorphisme . . . 37
5.4.1 Conversion de type de base . . . 37
5.4.2 Conversion de type entre classes . . . 37
5.5 Les aléas du polymorphisme . . . 40
5.5.1 Attention aux signatures . . . 40
5.5.2 Attention au type de retour . . . 41
5.6 Le mot clé `final` . . . 43
5.7 L'héritage multiple . . . 43
5.8 Les interfaces . . . 44
5.9 Héritage et composition . . . 46

6 Les exceptions **47**
6.1 Introduction 47
6.2 Les classes d'exception 48
6.3 Usage des exceptions 48

7 Les interfaces graphiques **50**
7.1 Le modèle MVC 50
7.2 La gestion des événements 51
7.3 Démarrer avec Swing 52

8 Les Threads **53**
8.1 La classe `Thread` 54
8.1.1 Méthodes de la classe `Thread` 54
8.2 Création et Exécution 55
8.3 Relations entre `threads` 55
8.4 Synchronisation 56
8.4.1 Relation d'exclusion mutuelle : moniteurs 56
8.4.2 Communication à l'intérieur d'un bloc de synchronisation . . 57

9 JAVADOC 58

10 La réflexivité **60**

11 Quelques exemples de code utile **61**
11.1 Écriture dans un fichier 61
11.2 Sérialisation 63
11.3 Lecture du contenu d'une URL sur `internet` 64

A L'infra-structure de JAVA (Th. Bommart) **66**
A.1 La machine virtuelle JAVA 66
A.1.1 Définition et rôle 66
A.1.2 Les bytecodes JAVA 67
A.1.3 Les pièces virtuelles 67
A.1.4 Les peu, mais preux et valeureux registres 68
A.1.5 La zone des méthodes et le compteur de programme 69
A.1.6 La pile JAVA et les registres associés 69
A.1.7 La zone du « tas », « garbage collectée » 70
A.2 Le « style de vie » fichier .class 70
A.2.1 Né pour voyager 70
A.2.2 Qu'y a-t-il dans un fichier .class ? 70

A.2.3 Nombre magique et numéros de versions 71
A.2.4 La zone des constantes 72
A.2.5 Drapeaux d'accès 73
A.2.6 La partie « `this_class` » 73
A.2.7 La partie « `super_class` » 73
A.2.8 Interfaces 73
A.2.9 Les attributs 74
A.2.10 Les méthodes 74
A.2.11 Les propriétés 74
A.3 La zone du tas recyclable (garbage-collected heap) de la machine virtuelle JAVA 75
A.3.1 Introduction 75
A.3.2 Pourquoi recycler la mémoire ? 76
A.4 Algorithmes de recyclage 77
A.4.1 Recycleurs basés sur le comptage des références 77
A.4.2 Recycleurs basés sur le traçage des objets 79
A.4.3 Recycleurs compacteurs 80
A.4.4 Recycleurs copieurs 81

B Liens **82**

1 Introduction

1.1 Vocation de ce document

L'objectif principal de ce cours est la présentation des concepts essentiels de la programmation orientée objets. Dans ce cadre, le langage JAVA ne constitue qu'un support illustratif spécifique, que nous avons choisi en fonction de notre objectif pédagogique plutôt qu'en fonction de considérations industrielles, qui nous auraient plutôt poussées vers C++. En conséquence, nous avons mis l'accent sur la programmation orientée objets plutôt que sur le langage JAVA lui-même.

Par ailleurs, nous sommes partis du principe que vous connaissez déjà le langage C. Nous ne présenterons pas les aspects de JAVA qui sont hérités du C : les types de base, la syntaxe fonctionnelle, les instructions générales (branchements, boucles, ...), la notion de bloc, etc.

Pour présenter un langage informatique et, en particulier, un langage orienté objets, la principale difficulté tient au fait que de nombreux aspects sont interdépendants, si bien qu'il est difficile de présenter certaines notions sans faire appel à d'autre notions qui n'ont pas encore été présentées. Le néophyte se trouve alors contraint d'accepter que soient différées certaines explications qui seraient pourtant nécessaires à la compréhension de ce qui lui est présenté.

Face à cet état de fait, nous avons choisi dans ce document de faire un rapide tour d'horizon des notions essentielles de la programmation orientée objets telle qu'elles se présentent en JAVA, plutôt que de vous présenter les innombrables bibliothèques de composants qui sont à votre disposition.

Il existe une bibliographie très abondante et très bien faite sur tous les composants que fournit JAVA, que ce document n'a pas vocation à remplacer. Il vous est donc conseillé de lire rapidement ce qui suit, puis de vous reporter à la documentation technique référencée dans la bibliographie lorsque vous vous mettrez à utiliser concrètement JAVA.

Notre espoir est que, suite à la lecture de ce document, vous ayez la maîtrise des aspects fondamentaux qui constitueront le bon point de départ pour bien programmer en JAVA. Il ne vous restera alors qu'à élargir le spectre de vos connaissances par des lectures plus approfondies et, surtout, à pratiquer la programmation pour acquérir une expérience que la lecture seule ne peut vous donner.

1.2 De la programmation structurée à la programmation orientée objets

D'une façon très générale, programmer consiste à écrire une séquence d'instructions de telle façon qu'un ordinateur produise une sortie appropriée pour une certaine entrée.

Même si, en pratique, le programme qu'exécute un ordinateur se ramène toujours à une longue séquence d'instructions, cette longue séquence n'est pas la représentation la plus appropriée pour concevoir, écrire et comprendre un programme dès lors qu'il devient complexe.

Pour remédier à cet état de fait, dés les balbutiements de l'informatique, les informaticiens ont intégré dans leurs langages des variables, des boucles et des tests qui permettent de structurer les programmes qu'ils écrivent.

Mais c'est surtout la décomposition fonctionnelle qui s'est avérée fondamentale pour la structuration des programmes. Selon le principe de la décomposition fonctionnelle, un programme peut être vu comme une fonction de très haut niveau (la fonction principale) qui se décompose en un ensemble de fonctions plus simples, qui elles-mêmes se décomposent à leur tour et ainsi de suite jusqu'à un niveau vraiment élémentaire. On arrive alors à une vision hiérarchique du programme qui est beaucoup plus facile à appréhender qu'une séquence à plat.

Une bonne décomposition fonctionnelle se reconnaît aux faits que toutes les fonctions sont d'une longueur et d'une complexité raisonnable, que des fonctions de bas niveau sont réutilisées en plusieurs points dans l'arborescence et que la plupart des fonctions ont un sens autonome, indépendant du contexte particulier dans lequel on les emploie.

Au fur et à mesure que la complexité des programmes s'est accrue, il est apparu de plus en plus manifeste que la clef d'une bonne conception était la possibilité de réutiliser des éléments du programme. De ce point de vue, l'approche fonctionnelle a fini par atteindre une limite. La décomposition des fonctions ne s'avère pas suffisamment structurante pour appréhender les énormes programmes de plusieurs millions de lignes de code qui se développent de plus en plus fréquemment à travers le monde.

La programmation orientée objets peut être vue comme un pas supplémentaire effectué en direction d'une plus grande structuration. Les informaticiens confrontés à des codes de plus en plus importants ont fini par se rendre compte que les données manipulées par un programme jouent un rôle structurant au moins aussi important que les traitements effectués sur ces données.

En particulier, il s'est avéré pratique de regrouper certaines données dans des *structures* permettant de représenter des relations parties-tout entre les éléments manipulés

par le programme. Ces structures sont les ancêtres des objets.

La programmation orientée objets reprend tels quels tous les ingrédients de la programmation structurée que nous venons de mentionner. Mais, par rapport à l'usage des structures dans des langages tels C, elle fait un pas de plus en associant les traitements à ces structures.

La conception d'un programme s'en trouve sensiblement modifiée car elle ne se contente plus de décomposer les traitements en se laissant guider par la seule perspective fonctionnelle. Pour concevoir dans la perspective orientée objets, on commence par identifier les structures pertinentes pour la réalisation du programme. Puis, idéalement, on identifie l'ensemble des traitements qui s'appliquent naturellement à ces structures, indépendamment de toute visée applicative. C'est seulement à ce stade que l'on construit l'ensemble des traitements constituant le programme sur la base de ces traitements « naturels ». Un programme est alors vu comme un ensemble de traitements qui s'appliquent sur certains objets et déclenchent d'autres traitements sur d'autres objets.

Cette modification de perspective apparemment bénigne est en fait le point de départ d'un véritable bouleversement.

Tout d'abord, l'ensemble constitué d'une structure et de ses traitements, c'est-à-dire dans le nouveau langage d'une *classe* et ses *méthodes*, dispose d'une autonomie fonctionnelle qui favorise son développement indépendamment du reste du programme et sa réutilisation dans d'autres programmes. Par ailleurs, une classe peut représenter un type d'objet complexe constitué d'autres objets plus simples. Dans ce cas, l'ensemble des classes intervenant dans la définition de la classe principale constitue un tout autonome qui peut être isolé et réutilisé. Mieux, il devient possible de ne donner à l'utilisateur de l'objet complexe qu'une vue d'ensemble qui lui masque les détails de la réalisation faisant intervenir des objets plus simples. Cette autre caractéristique de la programmation orientée objets s'appelle l'*encapsulation*.

En outre, il apparaît que des objets appartenant à une même famille réalisent un certain nombre de traitements similaires, si bien qu'on peut factoriser les traitements communs au niveau d'une classe plus générique qui les représente tous. Cette propriété, qui est à la base de l'*héritage*, constitue une avancée décisive en matière de structuration des programmes, permettant de grandes économies de codage et favorisant encore la réutilisation.

Notre objectif dans ce document est de vous convaincre qu'un langage de programmation orientée objets est beaucoup plus riche qu'un simple langage fonctionnel. Vous pouvez utiliser le C++ ou JAVA comme un C amélioré, mais ce faisant vous vous privez de tout ce qui fait la puissance des langages orientés objets. Ce sont tous ces points que nous allons aborder dans la suite.

2 Des objets et des classes

2.1 Introduction

Lorsqu'on écrit un programme informatique, c'est pour résoudre un certain problème. Le programme s'exprime sous la forme d'un certain nombre de procédures à appliquer à des données, tandis que le problème s'exprime sous la forme d'un ensemble de traitements à appliquer à des objets.

Selon cette perspective, concevoir un programme dans un langage procédural classique, c'est effectuer la conception dans la langage de la solution, tandis que la programmation orientée objets permet de l'effectuer dans le langage du problème.

Tous les informaticiens expérimentés s'accordent à dire que la programmation dans un langage procédural est plus simple et s'apprend plus vite que la programmation orientée objets. Mais, parce qu'elle permet de travailler directement sur la représentation du problème, la programmation orientée objets permet de réaliser une conception qui colle beaucoup plus exactement au problème auquel on s'attaque. Le résultat est un programme plus facile à comprendre, même pour un non-informaticien, plus facile à maintenir et plus facile à réutiliser sur des problèmes proches.

La notion centrale de la programmation orientée objets est évidemment la notion d'*objet*. Dans un programme orienté objets, toutes les données manipulées sont des objets. En JAVA, même le programme exécutable est un objet sur lequel le programme lui-même peut agir.

Pour représenter concrètement des objets en programmation orientée objets (on dit pour implémenter ces objets), on dispose de *classes*. Pour agir sur les objets, on dispose de *méthodes*. C'est ce que nous allons voir dans la suite.

2.2 Les classes

2.2.1 Nature des classes

Les programmes écrits dans un langage orienté objets manipulent explicitement des *objets*. Chaque objet manipulé est le représentant d'une *classe*. Une classe désigne donc un *type générique* dont les objets sont des *instances*.

Le fait que des objets soient des instances d'une même classe signifie donc qu'ils ont quelque chose en commun. Ils partagent en général une certaine structure et des traitements qu'on peut leur appliquer, ce qui n'empêche pas qu'ils soient tous différents les uns des autres.

Par exemple, toutes les voitures sont des objets qui ont quatre roues, un volant, un moteur et une carrosserie. Toutes les voitures peuvent être mises en marche, stoppées, conduites, lavées, repeintes, etc.

Mais certaines voitures ont trois portes, d'autres en ont cinq, toutes n'ont pas la même longueur ou des moteurs de même puissance, toutes ne peuvent pas contenir le même nombre de passagers, ou encore des modèles exactement identiques peuvent différer par la couleur.

Ces différences vont en général être représentées en programmation orientée objets par le fait que tous les objets de type `Voiture` (toutes les instances de la classe `Voiture`) ont les mêmes *attributs*, mais ces attributs prennent des *valeurs* différentes pour représenter des voitures différentes. Mais surtout, même si la voiture de mon voisin est exactement identique à la mienne, ce ne sont pourtant pas les mêmes voitures, il s'agit bien de deux objets différents.

La classe voiture représente donc la structure générale que partagent toutes les voitures. Chaque objet de la classe voiture sera construit conformément à cette structure en associant une valeur à chacune des variables.

En JAVA un programme est structuré intégralement sous la forme d'un ensemble de classes. Il n'y a aucune variable globale qui soit externe à une classe. Le code source de chaque classe se trouve stocké dans un fichier qui a pour nom le nom de la classe suivi de « .java ». En principe, ce fichier ne doit contenir que cette classe. En pratique, il est toléré de regrouper plusieurs classes dans le même fichier, mais ce n'est pas conseillé.

En particulier, mettre une classe par fichier permet de retrouver le code d'une classe plus rapidement au sein d'un répertoire. Le compilateur émet des messages d'alerte si cette convention n'est pas respectée.

Il y a ici une simplification importante vis-à-vis du langage C++, dans lequel on distingue un fichier *header* avec une extension « .hh » ou « .H » qui contient les déclarations et un fichier de code avec une extension « .cc » ou « .C » qui contient le code des méthodes. En outre, contrairement à C++, JAVA gère de façon transparente pour l'utilisateur les cas où plusieurs classes ont à se connaître mutuellement, ce qui complique pourtant la compilation.

Une classe contient généralement un certain nombre d'éléments qui sont ses *attributs* ou ses *champs*. Elle contient aussi un ensemble de traitements qui sont ses méthodes. Par ailleurs, on peut trouver au sein du corps de chaque classe une fonction `main()` qui sert de point d'entrée pour l'exécution d'un programme. Cette fonction `main()` permet ainsi d'exécuter le code de la classe indépendamment de tout autre fichier. Cette dernière fonctionnalité est extrêmement pratique pour tester le comportement de la classe en cours de développement [1].

[1]Nous reviendrons dans la section 2.5 page 16 sur l'usage de la fonction `main()`

2.2.2 Éléments de syntaxe

La syntaxe de JAVA est héritée de celle du C++, qui repose elle-même sur celle du C. Donc l'ensemble des notations qui ne sont pas spécifiques des classes, attributs et méthodes sont les mêmes qu'en C.

En ce qui concerne les classes, elles sont repérées par le mot-clé `class` suivi d'un nom de classe, puis d'un bloc entouré d'une accolade ouvrante et d'une accolade fermante. Au contraire du C++, ce bloc n'est pas suivi d'un point-virgule final. À titre d'exemple, voici le squelette de la classe `Voiture` que nous avons décrit plus haut.

```
class Voiture
{
protected static final byte      nbPortes = 4;
protected double                 longueur;
protected byte                   nbPassagers;
protected Volant                 monVolant;
protected Moteur                 monMoteur;
protected Carrosserie            maCarrosserie;

public  Voiture()
  {
    // ici le code du contructeur
  }

public void demarrer()
  {
    // ici le code de la méthode demarrer()
  }

public void conduire()
  {
    // ici le code de la méthode conduire()
  }

public void stopper()
  {
    // ici le code de la méthode stopper()
  }
public static void main(String args[])
  {
    Voiture maVoiture = new Voiture();
    maVoiture.demarrer();
    maVoiture.conduire();
    maVoiture.stopper();
  }
}
```

Les attributs se notent comme des variables internes à la classe, et les méthodes comme des fonctions. Un exemple est donné dans la suite.

Tous les cas particuliers seront présentés au fil du texte et au travers d'exemples.

2.2.3 Conventions de notation

Comme en programmation procédurale classique, il est important en programmation orientée objets d'employer des noms significatifs. Cela est vrai aussi bien pour les classes, les attributs et les méthodes que pour les variables.

La programmation orientée objets faisant souvent appel à des classes et des méthodes qui ont un sens concret, le nom est généralement suffisant pour comprendre ce que fait une méthode, sans aller voir le détail du code.

Les noms de classes commencent généralement par une majuscule, ceux des attributs, des variables et des méthodes par une minuscule.

La volonté d'employer des noms significatifs produit souvent des noms de méthodes constitués de plusieurs mots accolés les uns aux autres. La convention en JAVA est que, lorsqu'un nom est constitué de plusieurs mots accolés, chacun des noms successifs commence par une majuscule. En général, le premier mot d'un nom de méthode est un verbe.

Par exemple, la classe `CompteEnBanque` aura une méthode `majSolde()` ou, mieux, `mettreAJourSolde()`.

Le code JAVA étant souvent mis à disposition sur `internet`, il est fréquent que tous les noms soient en anglais. En particulier, les noms des accesseurs [2] commencent généralement par `get` pour la lecture et `set` pour la modification puis sont suivis du nom de l'attribut commençant par une majuscule.

2.2.4 Un exemple commenté

Le code suivant représente un exemple de classe `CompteEnBanque`. Les mots-clés `package, import, public` et `protected` ainsi que l'usage des commentaires seront présentés plus loin.

La classe `CompteEnBanque` est dotée d'un constructeur [3] qui met le solde à 0 et des accesseurs pour lire et modifier le solde. On voit dans le constructeur un exemple d'appel de méthode au sein d'une autre méthode.

```
// ====================================================== //
//    Fichier:                       compteEnBanque.java   //
//     Auteur:                     osd (Olivier Sigaud)   //
// ====================================================== //

package Banque;

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

[2] Sur les accesseurs, voir la section 4.3 page 29.

[3] Sur les constructeurs, voir la section 3.2 page 18.

```
  /**
   * Classe représentant un compte en banque
   */

public class CompteEnBanque
{
protected        int solde; // solde du compte

  /**
   * constructeur
   */
  public CompteEnBanque()
  {
    setSolde(0);
  }
  /**
   * positionne le solde
   */
public void setSolde(int montant)
  {
    solde        =        montant;
  }
  /**
   * renvoie le solde
   */
public int getSolde()
  {
    return solde;
  }

public static void main(String[] args)
  {
    CompteEnBanque cpt = new CompteEnBanque();
    cpt.setSolde(50);
    System.out.println("solde courant :" + cpt.getSolde());
  }

  // fin de CompteEnBanque
}
```

2.3 Les attributs

Les attributs servent à représenter diverses relations.

• Ils peuvent représenter des caractéristiques ou qualités d'un objet. Dans le cas d'un objet concret comme une voiture, sa couleur, sa masse, sa cylindrée sont représentées par des attributs.

• Ils peuvent aussi représenter des composants d'un objet complexe. Le moteur, les roues, la boîte de vitesse d'une voiture sont aussi représentés par des attributs.

• Ils peuvent enfin représenter d'autres objets qui sont en relation externe avec

l'objet, mais qui sont nécessaires à la réalisation de certains traitements de l'objet. Le réseau routier, le conducteur, les autres véhicules sont aussi des attributs potentiels d'une classe `Voiture`.

Tous les attributs sont dotés d'un *type* qui est défini lors de la déclaration de la classe. Il peut s'agir soit d'un type de base, soit d'une autre classe. Mais chaque classe représentant un attribut contient à son tour des attributs plus simples si bien qu'en fin de compte toute l'arborescence des classes repose sur des types de base.

La syntaxe générale est la suivante :

```
visibilité class NomDeClasse
{
visibilité mode type nomDAttribut_1;
...
visibilité mode type nomDAttribut_n;

visibilité typeDeRetour nomDeMethode(params)
  {
    // corps de la méthode;
  }

}
```

Par exemple :

```
public class Voiture
{
protected String marqueConstructeur;
...
private static int cptr;

public String getMaxSpeed(int rapportDeBoîte)
  {
    // corps de la méthode;
  }

}
```

On place donc en général les attributs au début de la définition de la classe. La visibilité, argument optionnel qui vaut `public`, `private` ou `protected`, sera expliquée dans la section 4.2 page 28 sur le contrôle d'accès. Le mode, lui aussi optionnel, peut contenir les mots clés `static` et `final` [4] Le type est soit un type de base du langage, (`boolean, byte, short, int, long, float, double, char, void`), soit le nom d'une autre classe du programme. Le nom de l'attribut, enfin, est une variable connue partout à l'intérieur de tout objet de cette classe.

Pour ceux qui ont l'habitude de la programmation procédurale classique, les attributs peuvent être vus comme des variables globales internes à un objet. Ainsi, l'attribut

[4] Sur `static`, voir la section 2.5 page 16, sur `final` voir la section 5.6 page 43.

d'un objet garde sa valeur tout au long de la vie de l'objet, il peut être lu ou modifié dans toutes les méthodes de l'objet.

Cela implique qu'il n'est pas nécessaire de rappeler son type lorsqu'on l'utilise dans une méthode. Cela implique aussi qu'il faut éviter d'employer dans une méthode une variable locale qui a le même nom qu'un attribut, pour éviter toute confusion.

En JAVA, contrairement à ce qui se passe en C++, il n'est pas nécessaire de faire attention à la précédence des déclarations entre différentes classes lorsqu'une classe est l'attribut d'une autre. Le programme ne manipulant que des références sur les objets et non les objets eux-mêmes, il ne peut y avoir de problème d'imbrication mutuelle entre classes.

2.4 Les méthodes

2.4.1 Signature et surcharge

Les traitements associés à une classe sont appelées ses *méthodes*. Dans la mesure du possible, il faut s'efforcer d'associer à une classe des méthodes qui correspondent à des traitements qui sont naturels pour les objets de cette classe. Cela permet de réutiliser la classe dans un autre contexte que le programme pour lequel elle a été développée.

Il existe un certain nombre de méthodes particulières, que nous présenterons dans la suite. Ce sont les accesseurs [5] et les constructeurs [6].

Comme les fonctions de la programmation procédurale, les méthodes ont des paramètres et des valeurs de retour. On appelle *surcharge* d'une méthode le fait qu'une même classe ait plusieurs méthodes qui portent le même nom. En effet, deux méthodes des objets d'une même classe peuvent porter le même nom et renvoyer le même type d'objet à condition de ne pas avoir les mêmes paramètres. Cela est vrai aussi bien pour les méthodes classiques que pour les constructeurs.

On appelle la *signature* d'une méthode l'ensemble constitué du type de la valeur de retour, du nom de la méthode, du type de l'objet sur lequel elle s'exerce et du type des paramètres qui lui sont passés. Ainsi, deux méthodes d'objets de la même classe qui portent le même nom mais n'ont pas les mêmes paramètres n'ont pas la même signature et JAVA peut les distinguer.

À noter toutefois qu'il est interdit qu'une classe ait deux méthodes de mêmes noms et de même type de paramètres qui renvoient deux objets de types différents. Ce point induit certains à considérer que la signature est composée uniquement du nom de la méthode et du type des paramètres.

[5]Sur les accesseurs, voir la section 4.3 page 29.
[6]Sur les constructeurs, voir la section 3.2 page 18.

La possibilité de surcharger une méthode constitue un atout indéniable de la programmation orientée objets vis-à-vis de la programmation procédurale classique. En effet, des traitements similaires qui s'exercent sur des objets différents peuvent porter le même nom, ce qui n'est pas le cas dans des langages classiques où toutes les fonctions doivent avoir un nom différent.

2.4.2 Des fonctions aux méthodes

Alors qu'en surface on pourrait confondre les fonctions et les méthodes, le mode d'usage d'une méthode est sensiblement différent de celui d'une fonction.

Une fonction est un traitement rattaché à l'arborescence du programme auquel on passe en paramètres les structures auquel il s'applique. Une méthode, au contraire, est attachée à une classe et s'applique à un objet de cette classe sans qu'il soit nécessaire de passer celui-ci en paramètres. L'objet auquel une méthode s'applique est une sorte de contexte implicite qui n'est signalé que par le fait que l'on agit directement sur les attributs de cet objet particulier.

Par exemple, alors qu'en C la fonction `demarrer()` aura l'allure suivante :

```
struct Voiture
{
Embrayage      embrayage;
Moteur         moteur;
BoiteDeVitesse boite;
};


void demarrer(Voiture v)
{
  enfoncer(v.embrayage);
  faireTourner(v.moteur);
  passerVitesse(v.boite,1);
  relacher(v.embrayage);
}
```

en JAVA elle s'écrira plutôt :

```
class Voiture
{
Embrayage      embrayage;
Moteur         moteur;
BoiteDeVitesse boite;

public void demarrer()
{
  embrayage.enfoncer();
  moteur.faireTourner();
  boite.passerVitesse(1);
  embrayage.relacher();
```

```
}

}
```

2.5 Les membres statiques

2.5.1 Les attributs statiques

Il existe une copie de chaque attribut dans chaque instance d'une classe donnée, c'est-à-dire chaque objet engendré à partir de cette classe. Ainsi, l'attribut peut avoir une valeur différente pour chacun des objets de la classe.

Dans le cas des attributs statiques, au contraire, il n'existe qu'une seule copie de l'attribut pour tous les objets de la classe : l'attribut est rattaché à la classe elle-même et non à ses instances. Par conséquent, tous les objets partagent la même valeur pour l'attribut et si un seul objet modifie cette valeur, elle sera modifiée pour tous.

Un usage classique est donné par les constantes. La valeur des constantes étant la même pour tous les objets, il est plus économique d'en avoir une seule copie dans la classe [7].

Un autre exemple classique d'usage d'attribut statique est donné par l'exemple suivant :

```
class MaClasse
{
public long id;
private static long next =0;

  MaClasse()
  {
    id        =        next++;
  }

public static void printNext()
  {
    System.out.println(next);
  }
}
```

Chaque appel du constructeur `MaClasse()` incrémente le générateur de numéro d'identification `next`, si bien que chaque objet aura son propre numéro, rangé dans l'attribut `id`.

[7] Sur les constantes, voir la section 5.6 page 43.

2.5.2 Les méthodes statiques

Les méthodes statiques vérifient le même principe que les attributs statiques. Il n'en existe qu'une copie par classe, c'est une méthode de la classe plutôt que de ses instances. Par construction, une méthode statique ne peut avoir accès qu'aux membres (attributs et méthodes) statiques d'une classe.

Pour appeler une méthode statique d'une classe, on utilise en général le nom de la classe plutôt qu'une de ses instances. Cela évite d'avoir à appeler le constructeur de la classe et de construire une instance. L'intérêt des méthodes statiques est en effet de pouvoir être appelées alors qu'on ne dispose pas d'un objet de la classe dans laquelle elle est définie.

Ainsi, dans l'exemple ci-dessus, pour afficher la valeur du compteur, on utilisera

```
MaClasse.printNext();
```

plutôt que

```
MaClasse instance = new MaClasse();
instance.printNext();
```

ce qui aurait l'effet fâcheux de décaler la valeur du compteur puisqu'un objet nouveau serait créé.

La méthode `main()` est *statique*, c'est donc une méthode de la classe qui la contient plutôt que des objets de cette classe, en ceci qu'il n'y a pas un objet de la classe qui lui serve de contexte. Pour pouvoir appeler les méthodes d'un objet au sein de la méthode `main()` que contient la classe correspondante, il est nécessaire de construire une instance de cette classe au sein de la méthode avec l'instruction `new`. Cela sera expliqué dans la section 3.2 page 18.

Le paramètre de la fonction `main()` est un tableau de chaînes de caractères qui représente les chaînes de caractères passées en argument dans la ligne de commande, comme cela se fait en C et en C++.

MCours.com

3 Cycle de vie des objets

3.1 Introduction

On appelle cycle de vie d'un objet l'ensemble des états par lesquels il passe au cours de l'exécution d'un programme. En règle générale, les étapes de la vie d'un objet sont sa construction, son initialisation, parfois sa modification ou sa copie, puis sa destruction et son élimination de la mémoire.

3.2 Allocation mémoire et construction des objets

Dans tout langage informatique, la déclaration d'une variable se traduit par la réservation d'un emplacement mémoire qui sert à stocker la valeur de cette variable.

Par exemple, l'instruction `int a = 1;` réserve un emplacement mémoire de taille suffisante pour stocker un entier, puis dépose la valeur 1 dans cet emplacement mémoire. Le mécanisme est le même pour tous les types de base. Qu'en est-il pour les objets ?

L'emplacement mémoire pour stocker un objet est en général plus important que pour un type de base. En C++, la déclaration d'un objet, par exemple l'instruction `Voiture mavoiture;`, fait la même chose que pour un type de base. Il alloue à la variable `mavoiture` l'emplacement nécessaire au stockage d'un objet de type `Voiture`.

En JAVA, c'est un peu différent. L'instruction `Voiture mavoiture;` réserve un emplacement mémoire pour une *référence* sur un objet de type `Voiture`, c'est-à-dire une variable dont la valeur est l'adresse d'un objet de type `Voiture`. Mais, à ce stade, l'emplacement mémoire pour l'objet de type `Voiture` lui-même n'est pas encore alloué. La valeur de la variable `mavoiture` est `null` et toute tentative d'accéder à un attribut ou à une méthode non statiques de la classe `Voiture` à partir de cette variable lève l'exception `NullPointerException`[8].

Pour que le pointeur `mavoiture` référence effectivement un objet, il faut appeler un *constructeur*. Les constructeurs sont les méthodes employées pour créer un objet d'une classe donnée et éventuellement l'initialiser. Le constructeur se reconnaît au fait qu'il porte le même nom que la classe et ne mentionne pas de type de retour.

On invoque un constructeur à l'aide de l'instruction `new`. L'instruction complète pour créer un objet de type `Voiture` est :

`Voiture mavoiture = new Voiture();`.

C'est l'instruction `new Voiture();` qui alloue l'emplacement mémoire pour l'objet de type `Voiture`, en appelant un constructeur de la classe `Voiture`. L'ins-

[8] Sur les exceptions, voir la section 6 page 47.

truction `new`, qui effectue la réservation de la mémoire, renvoie l'adresse de la zone allouée. En conséquence, la variable `mavoiture` est affectée par l'opérateur « = » avec cette adresse en guise de valeur.

Pour résumer, l'instruction `Voiture mavoiture = new Voiture();` :

• crée une nouvelle variable, la variable `mavoiture`, à laquelle est affectée un emplacement ;

• alloue un emplacement pour une instance de la classe `Voiture` ;

• affecte la valeur de la variable `mavoiture` avec l'adresse de cet emplacement.

Graphiquement, la situation est représentée sur la figure 1.

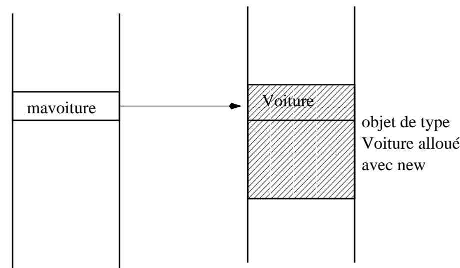


FIG. 1 – Implantation mémoire de `mavoiture`

On dit que la variable `mavoiture` est une *référence* sur un objet de la classe `Voiture`. On a l'impression en JAVA de manipuler des objets, mais on ne manipule en pratique que des références sur ces objets. Ce fait a des conséquences importantes, que nous détaillerons dans les sections 3.2.3 à 3.2.5 pages 20 à 25.

3.2.1 Constructeur par défaut

On peut surcharger les constructeurs aussi bien que tout autre type de méthode. Cela signifie qu'une même classe peut disposer d'autant de constructeurs que l'on veut.

Si l'on n'écrit aucun constructeur pour une classe donnée, il existe toujours un *constructeur par défaut* qui se contente d'allouer un emplacement mémoire. Pour une classe `MaClasse`, ce constructeur correspond au code suivant :

```
public class MaClasse
{
public  MaClasse()
   {
   }
}
```

Ce constructeur par défaut existe à condition que le programmeur ne fournit pas un constructeur explicite.

3.2.2 `this`

Pour qu'un objet puisse connaître sa référence sur lui-même, il existe le mot-clé `this`. Cela permet à l'objet de transmettre sa référence à un autre objet, par exemple pour s'abonner à un service qu'il fournit.

Voici un exemple extrêmement schématique :

```
class MailDistrib extends Vector
{
  public static void register(Object client)
  {
    addElement(client);
  }

  public static void distribute()
  {
    for (int i=0;i<size();++i)
      {
        distributeMailTo(elementAt(i));
      }
  }
}

public class OneClient
{
  public OneClient()
  {
    // ...
    MailDistrib.register(this);
  }
}
```

3.2.3 Constructeurs de copie

Examinons le code suivant :

```
Voiture mavoiture = new Voiture();
Voiture autre = mavoiture;
```

La seconde instruction consiste à créer une nouvelle référence, nommée `autre`, à laquelle est affectée la valeur de la référence `mavoiture`. Une fois l'instruction exécutée, la situation en mémoire est représentée sur la figure 2.

Il apparaît que `mavoiture` et `autre` sont des références sur le même objet de type `Voiture`. Par conséquent, si l'on applique à `autre` des méthodes qui modifient les valeurs de certains de ses attributs alors, sans que ce soit explicite, **on modifie aussi les valeurs des attributs de** `mavoiture` **puisque, en fait, c'est le même objet**.

Si, au lieu de disposer de deux références sur le même objet, on souhaite disposer de deux objets identiques, mais stockés dans deux emplacements mémoire distincts, l'instruction `Voiture autre = mavoiture;` ne suffit pas. Il faut appeler ce qu'on

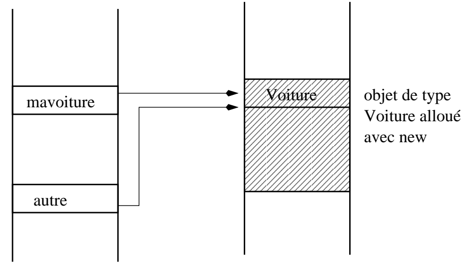


FIG. 2 – Implantation mémoire de `mavoiture` et `autre`

appelle un *constructeur de copie*.

Supposons une classe `Voiture` ultra-simplifiée :

```
class Voiture
{
  byte        nbPortes;
  long        cylindree;

  Voiture(Voiture v)
  {
    nbPortes    = v.nbPortes;
    cylindree   = v.cylindree;
  }
}
```

La méthode proposée est un constructeur de copie. On l'appelle avec l'instruction `Voiture autre = new Voiture(mavoiture);` Cette fois, le comportement en mémoire est représenté sur la figure 3 :

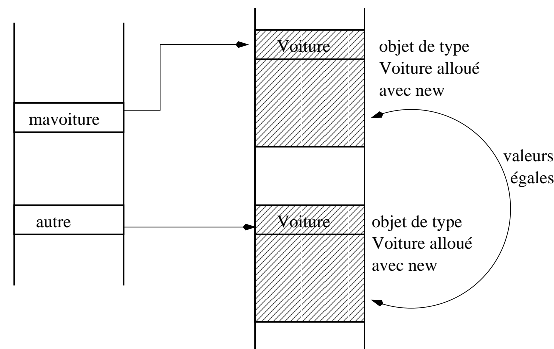


FIG. 3 – Copie mémoire de `mavoiture` dans `autre`

On peut alors modifier les valeurs des attributs de `autre` sans affecter ceux de `mavoiture`.

Mais il faut bien avoir à l'esprit le fait que, lorsque les attributs sont eux-mêmes des objets, le problème se reproduit. Par exemple, ajoutons à notre classe `Voiture` un attribut de type `Carrosserie`. Si l'on se contente d'affecter les attributs de la copie comme égaux à ceux de l'original avec le code suivant :

```
public class Voiture
{
  byte        nbPortes;
  long        cylindree;
  Carrosserie maCarros;

  /**
   * Constructeur de copie de Voiture
   */

  Voiture(Voiture v)
  {
    maCarros    = v.maCarros;
    nbPortes    = v.nbPortes;
    cylindree   = v.cylindree;
  }
}
```

alors on se retrouve dans la situation de la figure 4.

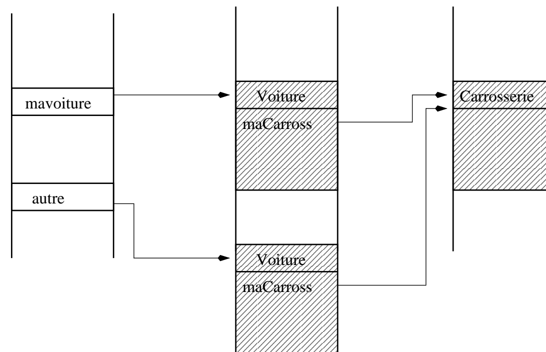


FIG. 4 – Copie mémoire inadaptée de `mavoiture` dans `autre`

Pour éviter ce problème, il faut écrire le constructeur de copie de la façon suivante :

```
public class Voiture
{
  byte        nbPortes;
  long        cylindree;
  Carrosserie maCarros;

  /**
   * Constructeur de copie de Voiture
   */
```

```
  Voiture(Voiture v)
  {
    maCarros    = new Carrosserie(v.maCarros);
    nbPortes    = v.nbPortes;
    cylindree   = v.cylindree;
  }
}
```

La situation est alors représentée sur la figure 5.

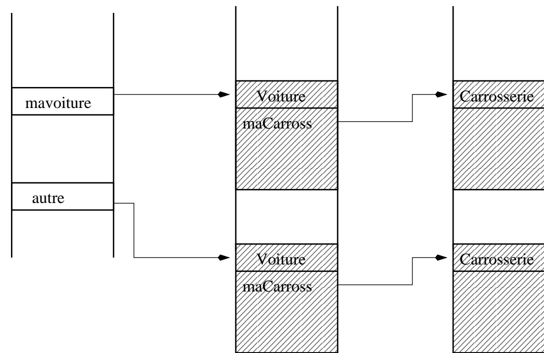


FIG. 5 – Copie mémoire correcte de `mavoiture` dans `autre`

On voit que, pour chacun des attributs, il est indispensable d'appeler le constructeur de copie de l'attribut, faute de quoi les objets `mavoiture` et `autre` partageront le même attribut `maCarros`, avec les conséquences indiquées précédemment.

Le non-respect de cette consigne est l'une des principales sources de bugs en JAVA. Ces bugs sont souvent très difficiles à dépister, ils se traduisent par le fait que les valeurs des attributs de certains objets se modifient « mystérieusement » dans des parties de code où l'on n'y fait pas référence.

Une règle d'or, lorsqu'on programme en JAVA, est de créer des constructeurs de copie pour toutes les classes que l'on manipule et de s'assurer que le constructeur de copie de chaque classe englobante appelle bien les constructeurs de copie de tous ses attributs.

3.2.4 Les clones

Plutôt que d'utiliser un constructeur de copie comme nous venons de le présenter, un usage répandu en JAVA consiste à utiliser une méthode `clone()` qui se présente de la manière suivante :

```
public class Voiture
{
```

```
protected  byte        nbPortes;
protected  long        cylindree;
protected  Carrosserie maCarros;

  /**
   * Constructeur de Voiture
   */

public  Voiture(byte nbp, long cyl, Carrosserie car)
  {
    maCarros = car;
    nbPortes    = nbp;
    cylindree   = cyl;
  }

  /**
   * méthode de clonage
   * @return Un clone de la voiture
   */

public  Voiture clone()
  {
    Voiture retour = new Voiture(nbPortes,
                                 cylindree,
                                 new Carrosserie(maCarros));
    return retour;
  }
}
```

On l'appelle avec l'instruction :

`Voiture copie = mavoiture.clone();`

Cet usage est encouragé en JAVA par le fait que la classe `Object` fournit une méthode `clone()` par défaut [9]. Tout objet qui veut utiliser cette méthode se doit d'implémenter l'interface `Cloneable` [10], faute de quoi l'appel de la méthode `clone()` lève une exception `CloneNotSupportedException` [11].

Il convient d'être aussi prudent dans l'usage que l'on fait de la méthode `clone()` que dans celui des constructeurs de copie. En effet, la méthode `clone()` fournie par défaut effectue une copie de toutes les valeurs des attributs. Si l'un des attributs est un objet, on se retrouve dans la situation décrite par la figure 4.

Par conséquent, dès qu'une classe possède au moins un attribut qui est un objet, il faut redéfinir [12] la méthode `clone()` en s'assurant qu'elle appelle bien la méthode `clone()` de chacun de ses attributs.

[9] Sur la classe `Object`, voir la section 5.3 page 36
[10] Sur les interfaces, voir la section 5.8 page 44
[11] Sur les exceptions, voir la section 6 page 47
[12] Sur la redéfinition, voir la section 5.2.2 page 35

3.2.5 Passage de paramètres

En JAVA, le passage de paramètres s'effectue par valeur. Cela signifie que, lors de l'appel d'une méthode, JAVA effectue une copie de la variable provenant de la méthode appelante et passée en paramètre. C'est sur la copie que travaille la méthode appelée. La portée de cette copie est limitée à la méthode dans laquelle elle est passée en paramètres.

Par conséquent, si la valeur de la variable est modifiée dans la méthode appelée, cette modification n'est pas répercutée dans la méthode appelante, à moins que la variable ne soit renvoyée en valeur de retour et affectée explicitement à la variable de la méthode appelante.

Ainsi, le code suivant :

```
class MaClasse
{

public void maMethode()
  {
    int a = 2;
    modifie(a);
    System.out.println(a);
  }

public void modifie(int param)
  {
    param = param +3;
    System.out.print(param + " puis ");
  }

public static void main(String[] args)
  {
    MaClasse c;
    c.maMethode();
  }
}
```

affiche en sortie « 5 puis 2 ». La valeur de `a`, dans la fonction appelante, n'est pas affectée par la modification de `param` dans la fonction appelée.

À présent, si le paramètre est un objet, la situation est différente. Nous avons dit plus haut que JAVA manipule des références sur des objets plutôt que les objets eux-mêmes. Ainsi, lorsqu'on passe un objet en paramètres, c'est en fait une référence sur l'objet qui est passée et copiée, et non l'objet lui-même. Comme nous l'avons expliqué, la copie de la référence construite par la méthode appelée pointe sur le même objet que la référence construite par la méthode appelante. Par conséquent, toute modification de l'objet à l'intérieur de la méthode appelée est une modification de l'objet tel qu'il est vu dans la méthode appelante. On est dans le cas décrit par la figure 2.

Si l'on souhaite modifier un objet dans une méthode sans toucher à l'original transmis par une méthode appelante, il est absolument nécessaire de commencer par réaliser explicitement une copie de l'objet à l'aide du constructeur de copie ou de la méthode `clone()`, pour travailler sur la copie. Cette copie peut être faite par la méthode appelante ou bien dans la méthode appelée. Le second cas est généralement préférable, car la portée de la copie sera locale à la méthode dans laquelle elle est construite.

Là aussi, une extrême vigilance s'impose pour éviter les bugs.

3.3 Initialisation

Lorsqu'un objet est créé, tous ses champs sont initialisés à une valeur nulle (`false` pour un booléen, `null` pour une référence et 0 pour une valeur). Le constructeur est alors invoqué. Son travail se fait en trois phases.

1. Invocation éventuelle du constructeur de la classe mère [13] ;
2. Initialisation des champs en utilisant la valeur d'initialisation spécifiée lors de leur déclaration ;
3. Exécution du corps du constructeur.

Il est vivement recommandé d'initialiser explicitement toutes les variables *dès leur déclaration*, plutôt que de les déclarer d'abord et de ne les initialiser qu'ensuite. Si vous cherchez à les utilisez entre le moment où elles sont déclarées et le moment où elles sont utilisées, vous risquez des mauvaises surprises lors de l'exécution. Heureusement, le compilateur JAVA signale la plupart de ces tentatives d'utilisation illicites de variables non initialisées.

3.4 Recyclage de la mémoire

Nous avons montré comment un programme JAVA manipule des références sur des emplacements mémoires dans lesquels sont stockés des objets. Tous ces objets sont construits par une allocation dynamique en invoquant l'instruction `new`. Se pose alors le problème de la libération de la mémoire ainsi allouée.

Ce problème est bien connu pour être l'un des pires casse-têtes de la programmation en C++. En effet, si l'on ne libère jamais la mémoire qui ne sert plus, on sature tôt ou tard tout l'espace disponible et on finit par obtenir le message « *Out Of Memory Error* ». En revanche, si l'on libère un emplacement mémoire trop tôt, toute tentative pour lire la donnée qui n'a pas été conservée risque d'engendrer des erreurs, qui peuvent être pernicieuses si une autre donnée est venue s'installer à la place.

[13] Sur l'héritage, voir la section 5.1.1 page 33.

Il faut donc trouver le juste moment pour libérer chaque emplacement mémoire alloué, ce qui se fait en C++ avec l'instruction `delete`. De nombreuses études ont montré que les programmeurs en C++ passaient un temps considérable à chercher où placer des instructions `delete`, ou alors ne s'en souciaient pas et se retrouvaient avec des programmes truffés de bugs. Ces considérations ont poussé les concepteurs du langage JAVA à automatiser la libération de la mémoire.

L'outil qui s'occupe de la libération de la mémoire s'appelle le « *garbage collector* », que l'on traduit en français par « ramasse-miettes ».

Les principes de fonctionnement des divers *garbage collectors* seront exposés en annexe dans la section A.3, page 75.

4 L'encapsulation

4.1 Introduction

L'une des grandes forces de la programmation orientée objets est qu'elle permet de donner à l'utilisateur d'un code une vue de haut niveau lui masquant les détails de la réalisation des traitements qu'il invoque. La technique qui consiste à masquer une partie des objets et méthodes impliqués dans un traitement pour n'en laisser voir que la couche de surface s'appelle *encapsulation*.

L'encapsulation est essentielle pour structurer la conception d'un programme. Elle permet de découper une application en modules aussi indépendants les uns des autres que possible en minimisant les interactions entre les modules.

En conséquence, chaque module devient plus aisément réutilisable dans un autre programme et il devient plus facile de remplacer un module par un autre dans un programme existant.

L'encapsulation opère à plusieurs niveaux.

• Au sein d'une classe, seules les méthodes publiques sont visibles de l'extérieur. Tous les traitements internes à la classe, réalisés pour rendre les services externes, peuvent (et doivent en principe) être masqués.

• Une classe peut faire appel à d'autres classes pour réaliser des traitements internes. Dans ce cas, seule la classe externe est visible, les autres sont masquées. Le démarrage d'une voiture fournit un bon exemple d'une telle situation. Si vous n'êtes pas mécanicien, vous ne vous préoccupez pas de savoir ce qui se passe dans votre voiture lorsque vous démarrez. Seule la classe `Voiture` et sa méthode `demarrer()` sont visibles. Les classes `Demarreur`, `Injection`, `Bougie`, etc. et tous les traitements qu'elles effectuent ne vous intéressent pas, vous n'avez pas à les connaître.

• Cette situation se généralise au niveau d'un *package* ou *paquetage* [14]. Un package est vu comme une librairie qui doit rendre un certain nombre de services. Seules quelques classes et méthodes du package constituent son interface avec les programmes qui l'utilisent et doivent être visibles. Le reste est une infrastructure qui doit rester masquée.

4.2 Le contrôle d'accès

Pour réaliser l'encapsulation, on dispose d'un ensemble de mots-clés de contrôle d'accès aux classes, méthodes et données.

4.2.1 Contrôle d'accès aux classes au sein d'un package

Pour les classes au sein d'un package, il n'y a que deux niveaux de visibilité.

• Si la classe est déclarée *public*, elle est visible dans la totalité du programme, donc y compris à l'extérieur du package.

• Sinon, elle n'est visible que par les classes du package dans lequel elle se trouve.

La syntaxe pour déclarer une classe publique consiste à écrire

```
public class MaClass {
```

au lieu de

```
class MaClass {
```

4.2.2 Contrôle d'accès aux attributs et méthodes

Pour les méthodes et attributs au sein des classes, le programmeur en JAVA dispose de quatre niveaux de contrôle d'accès, qu'il fixe à l'aide de trois modificateurs de visibilité.

• *public* : les éléments publics sont accessibles sans aucune restriction. En particulier, les membres publics d'une classe sont hérités par ses sous-classes.

• *protected* : les éléments protégés ne sont accessibles que depuis la classe et les sous-classes qui en héritent.

• *private* : les éléments privés ne sont accessibles que depuis la classe elle-même. Les membres privés d'une classe ne sont pas hérités par ses sous-classes.

[14] « Paquetage » est la traduction française de l'anglais « *package* ». Dans la pratique, c'est plutôt l'anglais « *package* » qui est utilisé. Nous nous conformerons ici à cet usage. Sur les packages, voir la section 4.4 page 29

• La visibilité par défaut, quand rien n'est spécifié, donne l'accessibilité d'un élément à tout autre objet du même package. La visibilité par défaut est donc équivalente à *public*.

4.3 Les accesseurs

En général, les attributs d'une classe sont déclarés privés ou protégés, ce qui signifie que seuls des objets de la classe ou de ses sous-classes peuvent les lire et modifier. Pour accéder à ces attributs de l'extérieur, on ajoute à la classe des méthodes spécialement conçues à cet effet, que l'on appelle « *accesseurs* » – et parfois « *modifieurs* » quand il s'agit de modifier la valeur.

L'usage systématique des accesseurs est recommandé pour des considérations de réutilisation. En particulier si, suite à une évolution de la conception, on décide de supprimer un attribut dans lequel on déposait le résultat d'un calcul, alors au lieu de renvoyer la valeur de l'attribut, l'accesseur renverra la valeur de retour du calcul. Il suffit de modifier le code de l'accesseur lui-même, c'est-à-dire une seule ligne du programme, alors qu'il aurait fallu modifier toutes les lignes où l'attribut était utilisé si l'on n'avait pas employé d'accesseur.

Comme on l'a dit dans la section 2.2.3 page 11, les noms des accesseurs commencent généralement par `get` pour la lecture et `set` pour la modification puis sont suivis du nom de l'attribut commençant par une majuscule. Un exemple de tels accesseurs est donné à la page 11 dans la section 2.2.3.

4.4 Les packages

4.4.1 Intérêt des packages

Les packages constituent un moyen commode pour découper une application volumineuse en modules bien distincts tout en gérant efficacement les conflits de nom entre ces modules.

Le découpage en modules est une étape délicate de la conception orientée objets. Toute la difficulté consiste à identifier des modules suffisamment indépendants les uns des autres pour minimiser les interactions et permettre des développements séparés des différents modules. Ces points sont traités plus en détail dans [Sig05].

Pour ce qui est des conventions de nommage, il faut avoir en tête le fait que les conflits de noms constituent un problème important lorsque l'on développe du code réutilisable.

Tout d'abord, il faut éviter qu'une classe porte le même nom que le package qui la contient. Bien que donner le même nom à une classe et à son package soit en théorie

parfaitement licite en JAVA, il s'avère en pratique que certaines machines virtuelles ne s'y retrouvent pas dans une telle situation.

Par ailleurs, le choix des noms des packages pose un problème plus vaste. Quel que soit le soin apporté par un concepteur au choix des noms de classes et de méthodes, il est probable que, tôt ou tard, quelqu'un d'autre voudra assembler les packages en question avec d'autres packages qui utilise ces mêmes noms avec une signification différente.

Pour éviter ces problèmes de conflits de noms, la solution standard adoptée par beaucoup de langages consiste généralement à placer des « préfixes de package » au début de chaque classe ou méthode. De telles conventions créent des contextes de nommage garantissant qu'un nom utilisé dans un package n'entrera pas en conflit avec un nom utilisé dans un autre package.

Éviter les conflits de nom est d'autant plus crucial en JAVA que le langage se caractérise par la possibilité de réutiliser des fragments de code écrits dans le monde entier et mis à disposition sur `internet`.

Si deux programmeurs donnent le même nom à un ou plusieurs packages qui contiennent des classes de même nom, alors il devient impossible de faire cohabiter ces packages dans un même programme. Pour remédier à ce problème, une convention veut que l'arborescence des packages installés sur `internet` commence toujours par le nom de domaine du site sur lequel ce package est développé, les éléments du nom apparaissant dans un ordre inversé.

Ainsi, un package `emission` développé sur le site de `france-info.com` sera désigné par `com.france-info.emission`.

4.4.2 Usage des packages

Tout fichier décrivant une classe appartenant à un package doit commencer par déclarer son appartenance à ce package, à l'aide de l'instruction :

```
package nomDuPackage;
```

En JAVA, pour désigner une classe qui est définie dans un autre package, on a le choix entre *importer* la classe de façon à ce qu'elle soit connue localement, ou bien faire précéder chaque occurrence du nom de la classe par le nom du package dans lequel elle est définie.

Ainsi, pour désigner au sein du package `loisir` la classe `Voiture` qui est dans le package `vehicule`, on a le choix entre le code suivant :

```
package loisir;

import vehicule.Voiture;
```

```
class Voyage
{
  public void voyager()
  {
    Voiture mavoiture = new Voiture();
    ...
  }
  ...
}
```

ou bien le code suivant :

```
package loisir;

class Voyage
{
  public void voyager()
  {
    vehicule.Voiture mavoiture = new vehicule.Voiture();
    ...
  }
  ...
}
```

Les packages sont construits selon une structure arborescente qui doit correspondre à l'arborescence des répertoires dans lesquels ils sont rangés. Les noms successifs le long d'un chemin de l'arborescence apparaissent séparés par des points lorsqu'ils sont invoqués par la commande `import`.

Par exemple, si l'on a l'arborescence

`biensDeConsommation/vehicule/deuxRoues`

`biensDeConsommation/vehicule/autres`

`biensDeConsommation/aliments`

`biensDeConsommation/meubles`

Alors l'instruction `import biensDeConsommation.*;` importe la totalité de l'arborescence, l'instruction `import biensDeConsommation.meuble.*;` importe la totalité du package `meubles` et l'instruction

`import biensDeConsommation.vehicule.deuxRoues.Moto;`

importe uniquement la classe `Moto`.

4.4.3 `classpath`

Le `classpath` est une variable d'environnement qui permet de spécifier l'ensemble des chemins dans lesquels JAVA doit aller chercher les classes susceptibles d'être nécessaires à l'exécution d'un code.

Sous UNIX et tous les systèmes d'exploitation apparentés, en csh et tcsh, on positionne le `classpath` à l'aide de la commande

setenv CLASSPATH chemin1 :chemin2 :... :cheminN.
En bash, on écrit *export CLASSPATH=chemin1 :ch2 :... :cheminN.*

5 L'héritage

5.1 Introduction

Il arrive que la structure de deux objets proches soit suffisamment différente pour qu'on ne puisse se contenter de les distinguer par des différences de valeurs sur des attributs. Dans ce cas, on est amené à construire des classes différentes. Pour reprendre l'exemple qui nous a servi d'introduction à la notion de classe, après avoir créé une classe `Voiture` qui permet de représenter toutes sortes de voitures différentes, plutôt que de représenter les camions comme des voitures particulièrement grosses, dotées de plus de roues et parfois de remorques, on créera une seconde classe `Camion` pour représenter des camions.

Mais les camions et les voitures ont tout de même beaucoup d'attributs et de méthodes en commun. Il serait dommage de décrire à nouveau dans la classe `Camion` tout ce qui l'a été dans la classe `Voiture`. La solution que propose la programmation orientée objets face à une telle situation consiste à créer une classe `Vehicule` qui contient tout ce qu'il y a de commun aux voitures et aux camions, puis à utiliser l'*héritage* pour dire que les voitures et les camions sont des types plus spécifiques de véhicules et décrire leurs particularités.

L'objectif général de l'héritage est de ne décrire un traitement qu'une seule fois quand il s'applique à plusieurs classes. En effet, décrire le même traitement plusieurs fois pour plusieurs classes est coûteux. Lorsqu'on modifie le traitement dans l'une des classes, il faut penser à le modifier dans toutes les classes. Cela fait perdre du temps et peut conduire à des pertes de cohérence.

La solution, lorsqu'un même traitement s'applique à plusieurs classes, consiste à définir une nouvelle classe plus générique que toutes ces classes et à définir le traitement au niveau de cette classe.

On définit alors toutes les *sous-classes* comme des classes *plus spécifiques* que la classe en question. On dit qu'elles *héritent* de cette classe. La classe plus générique est alors appelée leur *classe mère* ou *super classe*. Grâce à l'héritage, les traitements applicables à la classe mère peuvent être appliqués à toutes les sous-classes sans qu'il soit nécessaire de les réécrire [15].

Une sous classe hérite à la fois des méthodes et des attributs de sa classe mère. Il n'est donc nécessaire de répéter ni les uns ni les autres au niveau de la sous-classe.

[15] sous réserve qu'ils ne soient pas déclarés `private`, voir la section 4.2 page 28 sur le contrôle d'accès.

5.1.1 L'instruction `super()`

Le constructeur d'une sous-classe doit toujours commencer par appeler le constructeur de sa classe mère. Cela se fait avec l'instruction `super()`. On passe à `super()` les paramètres nécessaires à l'appel du constructeur de la classe mère. En général, ces paramètres constituent un sous-ensemble des paramètres du constructeur de la sous-classe. Considérons l'exemple suivant :

```
public class VehiculeARoues
{
protected int nbRoues;

public VehiculeARoues(int nbr)
  {
    nbRoues = nbr;
  }
}

public class Velo extends VehiculeARoues
{
protected int nbVitesses;

public Velo(int nbv)
  {
    super(2); // un vélo a toujours 2 roues
    nbVitesses = nbv;
  }
}

public class Camion extends VehiculeARoues
{
protected double cylindree;

public Camion(int nbr, double cyl)
  {
    super(nbr);
    cylindree = cyl;
  }
}
```

Si le constructeur de la classe mère d'une classe n'est pas appelé explicitement, c'est le constructeur sans aucun paramètre qui est appelé. Un tel constructeur n'existe que s'il a été défini ou si aucun constructeur n'a été défini, auquel cas il y a un constructeur par défaut [16].

[16]Sur les constructeurs par défaut, voir la section 3.2.1 page 19.

5.1.2 Le mot-clé `super`

Le mot-clé `super` permet de désigner l'objet père de l'objet courant, pour appeler une méthode d'un objet de la classe mère lorsqu'elle n'a pas la même définition au niveau de la sous-classe. On peut par exemple réaliser l'appel suivant :

```
public class B extends A
{
public String decritMere()
  {
    return super.toString();
  }
}
```

5.2 L'abstraction

5.2.1 Les méthodes abstraites

Il peut être difficile de spécifier un comportement particulier pour une classe générique. Par exemple, bien que toutes les formes fermées en deux dimensions aient une surface, on peut ne pas désirer calculer cette surface pour une forme quelconque. Par contre, on sait calculer la surface d'un rectangle, un triangle ou un cercle.

Par ailleurs, si l'on sait calculer la surface, on veut pouvoir écrire une méthode `plusEtenduQue()` qui détermine si une autre forme possède une surface supérieure ou inférieure.

Pour ce faire, on est conduit à définir au niveau de la classe générique `Forme` une méthode `calculerSurface()` dont on ne précise pas le comportement, mais qui sera appelée dans la méthode `plusEtenduQue()` s'appliquant sur différentes formes. Cette méthode fait appel à `calculerSurface()` et elle est générique.

Dans ce cas, on dit que la méthode `calculerSurface()` est abstraite pour la classe `Forme`. Cela implique que toutes les classes qui héritent de cette classe devront spécifier concrètement la méthode `calculerSurface()`, faute de quoi elles seront considérées aussi comme abstraites.

```
// ======================================================= //
//    Fichier:                               Forme.java    //
//     Auteur:                     osd (Olivier Sigaud)    //
// ======================================================= //

package Geometrie;

import java.io.*;
import java.util.*;
import java.lang.*;
import java.math.*;
  /**
   * Classe représentant une forme quelconque
```

```
   */

public class Forme
{
  /**
   * calcule la surface
   */
  abstract public long calculerSurface();

 public boolean plusEtenduQue(Forme autre)
  {
    long        s1        =        calculerSurface();
    long        s2        =        autre.calculerSurface();
    return (s1>s2);
  }
  // fin de Forme
}

// ======================================================= //
//    Fichier:                            Rectangle.java   //
//     Auteur:                      osd (Olivier Sigaud)   //
// ======================================================= //

package Geometrie;

import java.io.*;
import java.util.*;
import java.lang.*;
import java.math.*;
  /**
   * Classe représentant un rectangle
   */

public class Rectangle
{
  int longueur;
  int largeur;
  /**
   * calcule la surface
   */
  public long calculerSurface()
  {
    return (long)longueur*largeur;
  }
  // fin de Rectangle
}
```

5.2.2 La redéfinition

On appelle *redéfinition* le fait de donner à une classe spécifique une méthode qui porte le même nom et possède la même signature qu'une méthode d'un objet de la classe générique dont elle hérite, mais dont le code est différent de cette méthode plus générique. Dans le cas où la méthode plus générique est abstraite, on ne parle pas de

redéfinition mais d'*instanciation*.

5.2.3 Les classes abstraites

Une classe est déclarée *abstraite* lorsque l'on veut interdire que le programme manipule des objets de cette classe.

En pratique, toute classe qui possède au moins une méthode abstraite est considérée comme abstraite, même si elle n'est pas déclarée comme telle. Le compilateur refuse tout appel du constructeur sur une telle classe.

Le fait qu'une classe soit abstraite implique donc nécessairement qu'il existe des classes qui en dérivent et qui instancient ses méthodes abstraites.

Dans l'exemple précédent, aucun objet de type `Forme` n'est créé, le programme ne peut manipuler en pratique que des rectangles, des triangles ou des cercles. On peut déclarer la classe `Forme` abstraite. Cela se fait en remplaçant `public class Forme` par `public abstract class Forme`.

5.3 La classe `Object`

Certains traitements sont si génériques qu'ils peuvent s'appliquer à tous les objets envisageables dans n'importe quel programme. Par exemple, un ensemble, une liste, ou de façon plus générale une collection quelconque peut contenir n'importe quel type d'objet et effectuer des opérations ensemblistes sur ces objets indépendamment de leur type : les compter, les trier, etc.

La classe `Object` sert à définir l'objet le plus générique qui soit. Toute classe peut être conçue comme héritant de la classe `Object`.

5.3.1 Les méthodes de la classe `Object`

- `public boolean equals(Object obj)`
  Teste l'égalité des valeurs de l'objet receveur et de l'objet passé en paramètre.
- `public int hashCode()`
  Retourne le code de hachage de cet objet, pour le ranger dans une `Hashtable`.
- `public Object clone() throws CloneNotSupportedException`
  Retourne un clone de cet objet, voir la section 3.2.4 page 23.
- `public final Class getClass()`
  Retourne un objet de type `Class` qui représente la classe de cet objet et contient notamment le nom de la classe, accessible par `getName()`.

– `public void finalize() throws Throwable`
finalise l'objet lors de l'appel du *garbage collector* [17]

5.4 Le polymorphisme

On appelle polymorphisme le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe. En JAVA, le moyen de réaliser le polymorphisme consiste à « caster » [18] une classe en une autre. On parle aussi de conversion de type.

On ne peut caster que d'un type de base dans un autre, ou alors d'une classe vers une autre classe plus générique ou plus spécifique. Toute autre tentative de « cast » est acceptée à la compilation mais lève l'exception `ClassCastException` lors de l'exécution [19].

5.4.1 Conversion de type de base

Le cast entre types de base sert à faire des conversions de type utiles pour les opérations mathématiques. Par exemple, on veut disposer de l'équivalence entre les entiers valant 3, qu'ils soient représentés par un `byte`, un `short`, un `int` ou un `long` et les réels valant aussi 3 (ou 3.0), qu'ils soient représentés par un `float` ou un `double`.

Quelques exemples de tels casts :

```
double d        =        (double)3.96;
short  s        =        (short)d;

        => s = 3

byte  s        =        (byte)3;
float f        =        (float)s;

        => f = 3.0
```

Il s'agit d'un cas très particulier de polymorphisme, puisque les types de base n'héritent pas les uns des autres. À noter que JAVA fournit des classes encapsulant les types de base pour que l'on puisse manipuler ces types comme des objets.

5.4.2 Conversion de type entre classes

Le polymorphisme permet de décrire un comportement comme s'il s'appliquait à un objet générique alors que dans la pratique il s'applique à un objet spécifique. Si le

[17] Sur le recyclage de la mémoire, voir en annexe la section A.3 page 75.
[18] « caster » est un infâme anglicisme, mais c'est ce que tout le monde utilise en pratique. Si vous souhaitez vous démarquer en employant un mot bien de chez nous, utilisez « trans-typer » pour « caster » et « transtypage » pour « cast ».
[19] Sur les exceptions, voir la section 6 page 47

comportement appelle sur l'objet manipulé des méthodes qui existent aussi bien au niveau générique qu'au niveau spécifique, il n'est pas nécessaire d'écrire une conversion de type explicite. Le programme JAVA sait lors de l'exécution que l'objet manipulé est bien d'un type spécifique et va chercher la méthode définie au niveau de la classe spécifique.

Considérons l'exemple suivant :

```
class Voiture
{
  demarrer()
  {
    contact.set(true);
    injecteur.demarrer();
  }
}

class VoitureDiesel extends Voiture
{
  demarrer()
  {
    contact.set(true);
    resistance.chauffer();
    injecteur.demarrer();
  }
}

class Traction_1910 extends Voiture
{
  demarrer()
  {
    manivelle.tourner();
  }
}

class Automobiliste
{
  public void voyagerAvec(Voiture v)
  {
    v.demarrer();
    // ...
  }
}
```

Dans cet exemple, la méthode `voyagerAvec(Voiture v)` appelle la méthode `demarrer()` correspondant au type de voiture qui est passé en paramètres.

Mieux, si l'on dispose d'une collection de voitures que l'on veut toutes démarrer successivement, on écrit la méthode suivante :

```
class Garagiste
{
  public void demarrerToutes(Hashtable collec)
  {
```

```
    Enumeration itr = collec.elements();
    while (itr.hasMoreElements())
      {
        // il faut un cast car nextElement() renvoie un Object
        ((Voiture)itr.nextElement()).demarrer();
      }
  }
}
```

Malgré le cast explicite vers le type `Voiture`, JAVA est tout de même capable d'appeler la méthode correspondant au type effectif de l'objet, *sous réserve que les méthodes aient bien la même signature*.

Mais on peut aussi utiliser la conversion de type explicite lorsque l'on écrit une méthode agissant sur un objet décrit à un niveau générique, mais qu'il faut appeler à un moment donné une méthode que seul l'objet spécifique possède. C'est ce que nous venons de faire pour appeler la méthode `demarrer()` alors que l'objet retourné était de type `Object`.

Typiquement, lorsqu'on veut introduire un objet d'un certain type dans un conteneur, l'objet sera casté vers la classe `Object` par la méthode d'insertion dans le conteneur et il faut le caster à nouveau dans son type d'origine lorsqu'on le récupère à la sortie du conteneur.

On caste plus souvent un objet d'une classe donnée vers une classe plus spécifique que vers une classe plus générique. Cela permet de décrire un objet à un niveau générique, puis de lui appliquer des traitements spécifiques en fonction de sa vraie nature.

Un exemple de ce genre est donné dans le code suivant, dans lequel les diverses façons de voyager doivent être précisées en fonction du type spécifique de véhicule.

```
class Voyageur
{
public void voyagerAvec(Vehicule v)
  {
    v.demarrer()
    if (v.getClass().getName().equals("Bateau"))
        {
          // ici le cast est indispensable car la classe Vehicule
          // n'a pas de méthode naviguer()
          ((Bateau)v).naviguer();
        }
    else if (v.getClass().getName().equals("Avion"))
        {
          // ici le cast est indispensable car la classe Vehicule
          // n'a pas de méthode voler()
          ((Avion)v).voler();
        }
    else v.rouler();
  }
}
```

Si l'on omet le cast, c'est lors de la compilation que le code sera rejeté, car JAVA ignore que l'objet passé sera d'un type plus spécifique. Par contre, à l'exécution, si l'on essaie de passer un objet qui n'est pas du type `Vehicule` ou de l'un de ses sous-types, l'exception `ClassCastException` est levée [20].

On notera que, dans un tel exemple, la solution la plus propre aurait consisté à définir une méthode générique `avancer()` puis à réaliser le comportement de cette méthode dans les classes représentant chacun des types de véhicule. Il faut donc s'efforcer de décrire le plus souvent possible les choses à un niveau générique et ne spécialiser les comportements que là où c'est indispensable.

5.5 Les aléas du polymorphisme

Les exemples précédents pourraient vous faire penser que, JAVA connaissant en permanence le type des objets manipulés, le polymorphisme ne pose jamais aucun problème. Il y a cependant des restrictions de taille qu'il est important de connaître.

5.5.1 Attention aux signatures

La première restriction à l'usage « transparent » du polymorphisme vient de ce que la conversion implicite de type fonctionne parfaitement sous réserve que la méthode appelée pour l'objet générique et celle appelée pour l'objet spécifique aient exactement la même signature. Or deux signatures ne sont pas identiques si l'un des paramètres de la méthode est simplement plus spécifique que le même paramètre pour l'autre méthode.

Imaginons par exemple une classe `Voiture` et une classe `VoitureDiesel` qui ont toutes les deux une méthode `comparer()` qui permet de comparer avec une voiture de même type. Si l'on écrit :

```
class Voiture
{
  public boolean comparer(Voiture v)
  {
    // ...
  }
}

class VoitureDiesel extends Voiture
{
  public boolean comparer(VoitureDiesel v)
  {
    // ...
  }
}
```

[20]Sur les exceptions, voir la section 6 page 47

```
class Garagiste
{
  public void comparerToutes(Hashtable collec, Voiture v2)
  {
    Enumeration itr = collec.elements();
    while (itr.hasMoreElements())
      {
        // il faut un cast car nextElement() renvoie un Object
        ((Voiture)itr.nextElement()).comparer(v2);
      }
  }
}
```

Alors quel que soit le type de l'objet retrouvé dans la collection et le type de l'objet passé en paramètre de `comparerToutes()`, c'est la méthode `comparer()` de la classe `Voiture` qui sera utilisée car la classe `VoitureDiesel` ne dispose pas d'une méthode `comparer(Voiture v)` pour comparer avec un objet de type `Voiture`.

Pour que la comparaison se passe bien, il faut écrire :

```
class VoitureDiesel extends Voiture
{
public boolean comparer(Voiture v)
  {
    // vérifier que l'argument passé est bien du type VoitureDiesel
    if (v.getClass().getName().equals("VoitureDiesel"))
      {
        super.comparer(v);
        // ici les comparaisons spécifiques
      }
    else return false;
    // ...
  }
}
```

De cette façon, c'est la bonne méthode de comparaison qui sera appelée quel que soit le type de l'objet passé en paramètre de `comparerToutes()`.

5.5.2 Attention au type de retour

L'autre élément que JAVA ne convertit pas alors qu'on aimerait qu'il le fasse est le type de retour des méthodes. Considérons l'exemple suivant :

```
class MonVecteur
{
  // renvoie un extrait du vecteur entre le début et la fin
public MonVecteur getSubSet(int debut, int fin)
  {
    MonVecteur retour = new MonVecteur();
    // ...
  }
```

```
}

class MaSequence extends MonVecteur
{
}
```

Alors il n'est pas possible à un objet de type `MaSequence` d'appeler la méthode `getSubSet(int debut, int fin)` et de recevoir en retour un objet de type `MaSequence` qui serait un extrait de cette séquence. L'objet créé en retour de `getSubSet(int debut, int fin)` est de type `MonVecteur` et aucun cast n'y changera rien.

Il n'y a que deux solutions :

– soit on réécrit intégralement `getSubSet(int debut, int fin)` au sein de `MaSequence` afin qu'elle renvoie un objet de type `MaSequence`. Cela a le mérite d'être simple, mais ce n'est pas très élégant en terme d'héritage.

– soit on fait en sorte que la méthode `getSubSet(int debut, int fin)` appelle une autre méthode `create()` qui renvoie un objet appartenant à la classe `MonVecteur` lorsqu'elle est appelée par un objet de type `MonVecteur` et un objet de type `MaSequence` si c'est par un objet de type `MaSequence`. C'est un peu plus élégant, mais tout de même pas très satisfaisant.

Voici le code qui réalise la seconde solution :

```
class MonVecteur
{
  public MonVecteur create()
  {
    return new MonVecteur();
  }
  // renvoie un extrait du vecteur entre le début et la fin
public MonVecteur getSubSet(int debut, int fin)
  {
    // reçoit un objet de type MaSequence ou MonVecteur selon
    // l'origine de l'appel
    MonVecteur retour = create();
    // ...
  }
}

class MaSequence extends MonVecteur
{
  public MonVecteur create()
  {
    return new MaSequence();
  }
}
```

5.6 Le mot clé `final`

Le mot clé `final` permet d'indiquer qu'un élément ne sera pas modifié. Dans le cas d'une variable, `final` indique que la variable est une constante. En règle générale, les constantes sont aussi déclarées statiques, pour gagner de la place en mémoire. Par exemple, on trouvera souvent la syntaxe suivante :

```
protected static final double gravitation = 9.81;
```

Dans le cas d'une méthode ou d'une classe, `final` indique que cette méthode ou cette classe ne peut être *dérivée*. S'il s'agit d'une classe, cela signifie qu'aucune autre classe ne peut hériter de cette classe. S'il s'agit d'une méthode, cela signifie que les classes qui héritent de la classe dans laquelle cette méthode est définie ne peuvent pas *redéfinir* cette méthode. Les sous-classes doivent utiliser cette méthode telle quelle.

Outre le fait que cela permet au compilateur de réaliser certaines optimisations, cet emploi de `final` permet au programmeur de se prémunir contre le contournement de certaines de ses méthodes dont il veut être certain qu'elles seront utilisées telles quelles. Ainsi, un utilisateur indélicat pourrait redéfinir la méthode `checkPassword()` pour ne pas avoir à entrer un mot de passe sur un code qu'il a piraté.

5.7 L'héritage multiple

Il arrive que, dans un même programme, on soit conduit à adopter différents points de vue sur le même objet. Par exemple, une voiture peut être vue à la fois comme un moyen de transport et comme un bien de consommation.

Dans un tel cas, on aimerait pouvoir associer à une classe donnée les traitements génériques qui peuvent lui être appliqués selon les différents points de vue. Cela revient à dire que l'on aimerait qu'une classe puisse hériter de plusieurs classes à la fois. Dans notre exemple, on aimerait que la classe `Voiture` hérite à la fois de la classe `MoyenDeTransport` et de la classe `BienDeConsommation` pour pouvoir lui appliquer les méthodes `voyager()` de `MoyenDeTransport` et `revendre` de `BienDeConsommation`, par exemple.

Cette possibilité, que l'on nomme *héritage multiple*, est offerte en C++ mais elle ne l'est pas en JAVA. La raison de ce choix est que, dans certains cas pathologiques, l'héritage multiple peut conduire à des incohérences qu'il est nécessaire d'éviter.

C'est le cas à chaque fois que les deux classes dont on veut hériter ont un même attribut qui prend deux valeurs différentes selon la classe, ou lorsqu'elles ont deux méthodes de même signature qui effectuent des traitements différents. Lorsque la sous-classe appelle la méthode en question, on ne sait pas à quelle méthode parmi celles de ses deux classes mères il est fait référence, ce qui est très gênant si les comportements sont différents.

5.8 Les interfaces

Il existe une différence sensible entre la notion d'interface telle qu'elle apparaît en C++ et ce que recouvre cette notion en JAVA. En C++, l'interface d'une classe est l'ensemble des méthodes et attributs publics de cette classe, c'est-à-dire tout ce qui est accessible de l'extérieur de la classe [21]. Par contre, en JAVA, une interface est une entité spécifique du langage qui ressemble à une classe abstraite. Il y a deux façons de considérer une interface au sens JAVA.

• On peut la voir comme une classe complètement abstraite, dont toutes les méthodes sont abstraites. On dit qu'une classe *hérite* d'une autre classe, mais qu'elle *implémente* une interface.

• On peut aussi la voir comme une spécification de toutes les méthodes que toute classe qui veut implémenter cette interface doit contenir.

C'est la seconde vue qui domine en JAVA. L'interface est d'abord un outil de conception. Une interface se présente comme une classe allégée qui ne contient que des signatures de méthodes et éventuellement des constantes.

Soit par exemple l'interface `Mobile` :

```
public Interface Mobile
{
  protected boolean allerAuPoint(Point A);
  protected Point   getPosition();
  protected void    setInitialPosition();
}
```

Une interface ne contient aucun traitement. En conséquence, une interface ne fait que *spécifier* un ensemble de méthodes d'interaction. Toute classe qui implémente cette interface devra nécessairement disposer de la totalité des méthodes dont la signature est décrite dans l'interface. L'interface est donc un moyen dont dispose le concepteur pour imposer un certain mode d'interaction.

Des traitements génériques peuvent porter sur des objets décrits comme des interfaces. Il suffit pour cela qu'ils appellent exclusivement les méthodes présentes dans la déclaration de l'interface.

Ainsi, si un concepteur développe un ensemble de traitements génériques portant sur des interfaces, un autre programmeur qui veut que ses objets puissent bénéficier de ces traitements génériques n'a qu'à faire en sorte qu'ils implémentent ces interfaces.

Par conséquent, le concepteur des traitements génériques n'a pas besoin de savoir comment les objets spécifiques réaliseront les traitements qu'il invoque.

Soit par exemple la méthode générique `faireParcourir()`, attachée à une classe quelconque, qui s'applique à l'interface `Mobile` précédemment décrite :

[21] Sur le contrôle d'accès, voir la section 4 page 27

```
public void faireParcourir(Mobile mobile,
                           int nbPoints,
                           Point[] itineraire)
{
  mobile.setPositionInitiale(itineraire[0]);
  for (int i=1;i<nbPoints;++i)
    {
      boolean franchi = mobile.allerAuPoint(itineraire[i]);
      if (franchi==false)
        {
          System.out.println("Point " + i + " : "
                             + itineraire[i]
                             + " non rejoint");
        }
    }
}
```

La méthode `faireParcourir()` place le mobile au premier point et essaye de lui faire rejoindre tous les points de l'itinéraire les uns après les autres. Si l'un des points n'est pas rejoint, la méthode affiche un message d'erreur et continue le traitement à partir de la position courante et vers le point suivant sur l'itinéraire.

Il n'est pas nécessaire de savoir dans le détail quel est le fonctionnement de la méthode `allerAuPoint(Point A)` des divers mobiles possibles pour écrire la méthode `faireParcourir()`. Le comportement sera pourtant très différent selon que le mobile est une pièce d'échec sur un échiquier ou bien une voiture sur un réseau routier.

Pour qu'on puisse lui appliquer la méthode `faireParcourir()`, une classe `Avion` doit donc implémenter l'interface `Mobile`. Cela se réalise de la façon suivante :

```
class Avion implements Mobile
{
protected  Point position;

public  void setInitialPosition(Point p)
  {
    position = p;
  }

public  Point getPosition()
  {
    return position;
  }

public  boolean        allerAuPoint(Point p)
  {
    double altitude  = 10000;
    double distance  = calculerDistance(getPosition(),p);
    double direction = calculerCap(getPosition(),p);
    return peutVoler(distance,direction,altitude);
```

```
  }
}
```

On voit alors en quoi l'usage des interfaces ressemble à celui de l'héritage. À première vue, la notion d'interface semble redondante avec celle de classe abstraite. Une interface serait juste une classe abstraite dans laquelle il ne serait pas possible de coder des traitements génériques. Ainsi, notre interface `Mobile` aurait pu être déclarée comme une classe abstraite et incorporer la méthode `faireParcourir()`.

La principale différence vient de ce que, alors que l'héritage multiple est impossible, une classe peut implémenter autant d'interfaces que l'on veut et les interfaces peuvent hériter les unes des autres. En effet, les interfaces ne sont pas sujettes aux dangers qui ont conduit les concepteurs de JAVA à rejeter l'héritage multiple, puisqu'elles n'incorporent aucun traitement ni aucun attribut.

5.9 Héritage et composition

Il est fréquent qu'une classe incorpore une autre classe qui réalise la majeure partie des traitements proposés par la classe externe. Les méthodes de la classe externe consistent alors simplement en une redirection, un appel direct à une méthode (souvent du même nom dans la classe interne). Dans ce cas, il est légitime de se demander s'il faut représenter la relation entre les deux classes par un héritage ou par une composition, c'est-à-dire en faisant de la seconde classe un attribut de la première.

Par exemple, on peut se demander s'il vaut mieux dire qu'une banque virtuelle, représentée par la classe `VirtualBank` *est* une liste de compte bancaires, représentée par la classe `BankAccountList`, à laquelle s'ajoute un nom de banque et des services, ou bien si l'on dit plutôt que la banque virtuelle *contient* une liste de compte bancaires et les autres éléments.

L'intérêt d'utiliser l'héritage tient au fait qu'il n'est pas nécessaire de réécrire toutes les méthodes de `BankAccountList` au niveau de `VirtualBank`. Cette dernière en hérite directement. Si l'on choisit d'utiliser la composition, au contraire, il faudra employer systématiquement des *redirections* du type :

```
public class VirtualBank
{
  protected BankAccountList maListe;

// exemple de redirection : la méthode ne fait que transmettre
// son appel à une autre classe.

public  void maMethode()
    {
      maListe.maMethode();
    }
}
```

L'héritage semble donc à privilégier lorsque la plupart des méthodes du composant peuvent être réemployées telles quelles au niveau de l'objet principal.

Par contre, si l'objet principal contient plusieurs composants également importants, un choix s'impose compte tenu de l'impossibilité d'utiliser l'héritage multiple en JAVA.

Dans tous les cas, la sémantique du domaine doit aider à guider les choix de conception. Afin de décider si l'on peut faire hériter la classe X de la classe Y, demandez-vous si vous pouvez affirmer sans sourciller qu'un X *est un* Y.

D'une façon générale, il ne faut pas abuser de l'héritage. Plus vous utilisez une hiérarchie complexe de classes, plus votre programme devient contraint, donc difficile à faire évoluer. Une remise en cause à un niveau donné peut alors avoir des conséquences en de nombreux autres points de la hiérarchie.

De plus, l'interdiction d'employer l'héritage multiple peut vous amener à revoir complètement votre conception le jour où un point de vue nouveau vient modifier celui que vous aviez adopté.

Enfin, s'il y a beaucoup de niveaux dans vos hiérarchies, vous finirez par ne plus savoir où se trouve le code effectivement utilisé par une des classes de plus bas niveau.

Pour toutes ces raisons, le choix entre l'héritage et la composition est toujours délicat et requiert avant tout de l'expérience en conception orientée objets.

6 Les exceptions

6.1 Introduction

Pour certaines méthodes spécifiques, certaines causes d'erreur peuvent être prévues à l'avance. C'est le cas lorsque la méthode doit être utilisée dans un contexte bien particulier qui risque de ne pas être respecté par l'utilisateur.

Par exemple, une méthode qui ouvre un fichier en lecture ne fonctionnera pas si le fichier n'existe pas. Cette cause d'erreur peut être prévue par le programmeur de la librairie de gestion des fichiers.

Dans un tel cas, le programmeur peut tester un certain nombre de causes d'erreur potentielles et associer à la méthode des comportements particuliers si ces erreurs sont détectées.

Le mécanisme approprié en JAVA pour traiter ces erreurs est fourni par les *exceptions*.

6.2 Les classes d'exception

Les exceptions sont des objets particuliers dont les classes dérivent toutes de la classe `Exception`, qui elle-même dérive de la classe `Throwable` [22]. Cette classe contient au moins une chaîne de caractères pour décrire le type d'exception.

Il existe un certain nombre d'exceptions génériques définies dans le langage et correspondant à des situations fréquentes ou des erreurs de programmation classiques. Un certain nombre d'entre elles ont déjà été présentées au fil de ce document.

Mais le programmeur peut définir ses propres classes d'exception en les faisant hériter de la classe `Exception`.

6.3 Usage des exceptions

Les exceptions sont levées à l'aide de l'instruction `throw`, qui prend une instance d'exception en paramètre. Une méthode qui est susceptible de lever une exception doit être signalée à l'interpréteur ou au compilateur en ajoutant la clause `throws` suivi du type de l'exception lors de la déclaration de la méthode.

Un exemple est donné dans la suite :

```
// ======================================================== //
//    Fichier:                         MonException.java    //
//     Auteur:                      osd (Olivier Sigaud)    //
// ======================================================== //

public class MonException extends Exception
{

  /**
   * constructeur
   */
public MonException()
  {
    // appel du constructeur de la classe Exception
    super("Ceci est une exception perso");
  }
  // fin de MonException
}

// ======================================================= //
//    Fichier:                           MaClasse.java     //
//     Auteur:                      osd (Olivier Sigaud)   //
// ======================================================= //

/**
   * Classe quelconque
```

[22]Le nom `Throwable` laisse à penser que cette classe a d'abord été conçue comme une interface, puis le modèle a évolué.

```
   */

public class MaClasse
{

  /**
   * constructeur
   */
public MaClasse()
  {
  }
  /**
   * méthode qui renvoie une exception
   */
public void maMethodeGenerantException() throws MonException
  {
    // appel du constructeur de la classe MonException
    if (erreur) throw new MonException("Message d'erreur");
  }
  // fin de MaClasse
}
```

Lorsqu'une méthode est susceptible de lever une exception, toutes les méthodes qui l'appellent doivent obligatoirement *capturer* cette exception.

Cette capture s'effectue à l'aide de la séquence d'instructions suivante :

```
try
{
  maMethodeGenerantException();
}
catch(MonException e)
{
  // traitement du cas où l'exception est levée
}
```

Un traitement typique du cas où l'exception est levée est le suivant :

```
...
catch(MonException e)
{
  e.printStackTrace();
  System.exit(-1);
}
```

La première instruction, `e.printStackTrace();`, affiche l'arbre d'appel de la méthode qui a engendré l'exception. On sait ainsi immédiatement dans quel contexte l'erreur s'est produite.

La seconde instruction, `System.exit(-1);`, arrête le programme avec une valeur de retour signalant une erreur.

7 Les interfaces graphiques

La facilité de mise au point d'interfaces graphiques en JAVA est l'un des éléments qui contribuent le plus massivement au succès de ce langage de programmation.

Les premières versions de JAVA étaient livrées avec un package nommé AWT [23] chargé de faire le lien entre le langage et les serveurs graphiques des différents systèmes d'exploitation dans lesquels la machine virtuelle de JAVA est implantée.

Cette situation présentait un inconvénient majeur. Les concepteurs de machines virtuelles JAVA étaient contraints de fournir un package AWT différent pour chaque système d'exploitation. Outre le sur-coût que cela implique, cela entraînait des disparités de fonctionnement d'un système d'exploitation à l'autre, ce qui est contraire à la volonté de portabilité maximale affichée par les concepteurs de JAVA.

C'est pour mettre fin à cette situation que le package SWING a été mis au point à partir de la version 1.1 du JDK. De l'extérieur, le programmeur habitué à l'AWT a l'impression de retrouver ce dont il a l'habitude quand il passe à SWING. La plupart des classes de l'AWT ont leur équivalent dans SWING, à une légère altération du nom près, pour permettre la cohabitation entre les deux familles de classes : `Frame` devient `JFrame`, `Panel` devient `JPanel`, etc.

Mais les choses ont profondément changé quand on y regarde de plus près. Tout d'abord, SWING ne repose plus sur les serveurs graphiques des différents systèmes d'exploitation. Toutes les fonctionnalités de haut niveau offertes par ces serveurs – gestion de fenêtres, d'événements, de composants graphiques divers – ont été recodées directement au cœur du package SWING. Seul subsiste l'appel qui permet d'allumer un pixel donné dans une couleur donnée, ce qui garantit une portabilité totale d'un système d'exploitation à l'autre.

Ensuite, le modèle de programmation auquel incite l'utilisation de SWING est plus conforme à ce qui est largement préconisé en matière de méthodologie de développement d'interfaces graphiques. Il est en effet naturel quand on utilise SWING de développer une interface graphique conformément au modèle MVC, qui s'impose comme un standard de modélisation d'interfaces.

7.1 Le modèle MVC

Le modèle MVC est un « *design pattern* » issu de Smalltalk-80. MVC signifie *Model View Controller*. Il s'agit d'une façon d'organiser les différents composants d'un logiciel de façon à faciliter la réutilisation et éviter des problèmes de conception et d'implémentation. Ce modèle est très générique et ne s'applique pas qu'aux interfaces

[23] Pour *Abstract Window Toolkit*

graphiques, même si c'est pour la conception d'interfaces graphiques qu'il est le plus utile.

Comme son nom l'indique, ce modèle contient trois composants :

1. **Model** : la partie « modèle » correspond au cœur de l'application. Elle contient les données que le programme manipule et gère les changements d'états induits par les opérations sur ces données.
2. **View** : la partie « vue » correspond à l'interface en sortie présentée à l'utilisateur pour lui donner connaissance de l'état de la partie « modèle ».
3. **Controller** : la partie « contrôleur » correspond à l'interface en entrée présentée à l'utilisateur pour interagir avec la partie « modèle ».

À une même partie « modèle » peuvent être associées plusieurs vues, destinées à plusieurs utilisateurs ou à un seul. À chaque fois que l'état de la partie « modèle » change, il faut informer du changement chacune des vues concernées par la modification, afin que celles-ci mettent à jour l'information présentée aux utilisateurs. De même, chaque action de l'utilisateur doit être transmise par un contrôleur à la partie « modèle » pour donner lieu à une action sur les données.

La façon la plus propre de gérer ces interactions entre vues, modèle et contrôleur est de notifier tout changement qui concerne l'un ou l'autre des éléments sous la forme d'un événement. Les différents composants s'abonnent à des sources d'événements et se voient notifier ceux-ci lorsqu'ils se produisent.

Nous exposons la façon de gérer les événements en JAVA dans la section suivante.

7.2 La gestion des événements

En JAVA, les événements sont des objets de la classe `Event` ou des classes qui en dérivent, telles que `ActionEvent`, `ChangeEvent`, ...

Pour qu'un objet soit apte à réagir à des événements de type `ActionEvent`, sa classe doit implémenter l'interface `actionListener`. De même, elle implémentera l'interface `changeListener` pour un événement de type `ChangeEvent`. Pour passer avec la source le contrat qui l'abonne à la réception d'un événement, l'objet doit implémenter une méthode `actionPerformed()` qui prend en argument l'événement reçu.

L'exemple ci-dessous représente le traitement effectué lorsqu'un utilisateur réalise une opération pour quitter l'application, qu'il ait appuyé sur un bouton "Quit", sélectionné "Quit" dans un menu ou réalisé toute autre opération du même type.

```
class Quit implements ActionListener
{
public void actionPerformed(ActionEvent e)
```


MCours.com

```
    {
        System.out.println("action Quit sur Objet " + this);
        System.exit(0);
    }
}
```

Il suffit donc qu'une classe implémente l'interface `actionListener` et qu'elle dispose d'une méthode `actionPerformed()` pour qu'elle soit capable de gérer un événement auquel elle doit être sensible.

Par ailleurs, pour indiquer à quelle source d'événement une classe est sensible, celle-ci doit s'abonner à cet événement auprès d'un objet de la classe source en lui envoyant un message d'abonnement (c'est-à-dire en appelant une méthode particulière de cette classe).

Par exemple, tout composant graphique à partir duquel l'utilisateur est susceptible d'émettre un événement doit déclarer à quel `Listener` il est associé. Pour cela, le composant graphique doit appeler la méthode `addXXXListener()` où `XXX` désigne un type d'événement tel que `Action, Mouse, Change...` en passant en paramètre l'objet destinataire qui se chargera de traiter ses événements. On peut ainsi associer plusieurs gestionnaires d'événements à une même source d'événement de type `Action` en lui appliquant plusieurs fois la méthode `addActionListener()`.

7.3 Démarrer avec Swing

Il n'est pas question dans le cadre restreint de ce polycopié de présenter les divers composants graphiques proposés dans SWING. Le choix est immense, les composants sont souvent de très haut niveau et répondent à la plupart des besoins communs à d'innombrables applications.

Avant de vous lancer dans le développement d'une interface graphique, commencez systématiquement par vous demander s'il n'existe pas un composant tout fait qui correspond exactement à ce dont vous avez besoin. Pour procéder à un choix de composant, vous n'avez qu'à vous rendre sur le site suivant :
http ://java.sun.com/docs/books/tutorial/index.html puis cliquer sur le lien « *Creating a GUI with JFC/Swing* », puis « *Using Swing components* », et enfin « *A visual Index to the Swing Components* », ce qui vous donne accès au catalogue.

Nous ne donnons donc ici que les quelques instructions de base qui permettent de démarrer une interface graphique avec SWING.

Les packages à importer sont les suivants :

```
import javax.swing.*;
import javax.swing.event.*;
```

La première instruction consiste à créer une `JFrame`, fenêtre de haut niveau.

```
JFrame jf = new JFrame();
```

Puis on crée un conteneur d'éléments graphiques de type `JPanel`.

```
JPanel panel = new JPanel();
```

On peut fixer les valeurs d'un certain nombre de propriétés de ce conteneur, comme par exemple sa taille ou le type de représentation de la bordure de la fenêtre.

```
panel.setBorder(
    BorderFactory.createEmptyBorder(50,50,50,50));
```

L'ajout d'un bouton dans l'interface graphique s'effectue avec la séquence d'instructions suivantes :

```
JButton button1=new JButton("OK");
button1.addActionListener(this);
jp.add(button1);
```

Il en est de même avec tout autre composant graphique. On commence par créer le composant, on lui associe un gestionnaire d'événement qui est souvent la classe qui contient le composant, puis on ajoute le composant dans l'interface. Il faudra bien sûr s'assurer de la réception correcte des événements, comme cela a été expliqué à la section 7.2, page 51.

Une fois le conteneur rempli avec un certain nombre de composants, il reste à associer le conteneur à la fenêtre principale.

```
jf.setContentPane(panel);
```

Puis on donne l'ordre d'affichage de la fenêtre...

```
jf.pack();
jf.setVisible(true);
```

...et le tour est joué.

8 Les Threads

La partie de ce polycopié consacrée aux `threads` *s'appuie largement sur un autre polycopié écrit par Emmanuel Chailloux, que vous trouverez à l'adresse suivante :* *http ://www.pps.jussieu.fr/~emmanuel/Public/enseignement/ POD_2000_WWW/Poly/poly2.html*

Les threads sont des processus légers qui implémentent un modèle de parallélisme à mémoire partagée. Ces processus ne dépendent pas du système d'exploitation, mais de l'implantation de la JVM, ce qui assure leur portabilité.

Plusieurs threads peuvent partager leur contexte d'exécution sur une même machine virtuelle auquel cas ils se partagent la même zone mémoire. Lancer plusieurs threads sur la même machine virtuelle ne permet pas d'aller plus vite, cela sert uniquement à exprimer proprement des algorithmess concurrents. Ceci dit, une fois que des algorithmes sont exprimés sous forme de threads, rien n'empêche de les distribuer vraiment avec RMI ou CORBA.

8.1 La classe Thread

Les threads possèdent un nom et une priorité.

Une fois créé, un thread peut être dans les états suivants :

- exécutable
- en cours d'exécution
- bloqué
- mort

8.1.1 Méthodes de la classe Thread

La classe Thread est dotée des méthodes suivantes :

- **start()** : rend un thread « exécutable », et lance la méthode run().
- **run()** : exécution proprement dite ;
- **stop()** : arrête un thread (dépréciée).
- **suspend()** : met un thread dans l'état bloqué (dépréciée).
- **resume()** : fait passer un thread de l'état « bloqué » à l'état « exécutable » (dépréciée).
- **wait()** : met un thread dans l'état bloqué.
- **notify()** : fait passer un thread de l'état « bloqué » à l'état « exécutable ».
- **sleep(long temps)** : met un thread dans l'état bloqué pendant une durée exprimée en millisecondes.
- **yield()** : fait passer de l'état « en cours d'exécution » à l'état « exécutable ».
- **join()** : attend la fin d'un autre thread.
- **setPriority(int niveau)** : modifie la priorité d'un thread (cette priorité est une valeur entre Thread.MIN_PRIORITY et Thread.MAX_PRIORITY, qui sont des constantes statiques)).

8.2 Création et Exécution

Il y a deux façons de créer un `thread`. On peut étendre la classe `Thread` et redéfinir la méthode `public void run()`. C'est ce qu'on fait le plus souvent. On peut aussi créer une classe qui implémente l'interface `Runnable` et définit une méthode `public void run()` :

Un `thread` s'exécute jusqu'au moment où :

- un `thread` de plus grande priorité devient « exécutable » ;
- une méthode `yield()` ou `sleep()` est lancée ;
- `start()` se termine ;
- son quota de temps a expiré pour les systèmes implantant un temps partagé.

8.3 Relations entre `threads`

Un programme exécutant plusieurs `threads` attend que tous soient finis avant de se terminer. La première façon d'utiliser plusieurs `threads` au sein d'un même programme est de ne pas se soucier de leurs relations temporelles. Il faut tout de même être conscient du fait qu'ils peuvent partager des données en mémoire, ce qui peut poser des problèmes. On fait alors appel à des mécanismes de synchronisation, pour garantir l'exclusion mutuelle des accès à la mémoire.

On commence par donner ci-dessous un exemple de lancement de deux `threads` identiques sans relation de synchronisation :

```
class TestThreads
{
    public static void main (String []args)
    {
        MyThread mt1 = new MyThread(1);
        MyThread mt2 = new MyThread(2);
        mt1.setPriority(10);
        mt2.setPriority(10);
        mt1.start();
        mt2.start();
    }
}
class MyThread extends Thread
{
    int num;
    int st = 1;

    MyThread(int n)
    {
        num = n;
    }

    public void run()
    {
```

```
        int i=0;
        while (st < 5000001)
         {
            if ((st % 1000000) == 0)
              {
                System.out.println("MT"+num+" trace = "+st);
              }
            st++;
            yield();
         }
    }
}
```

Il y a une différence de comportement selon l'implantation de la JVM sur les systèmes gérant le temps partagé et les autres. Dans le premier cas, le multi-threading est préemptif alors que, dans le second cas, il est coopératif. Pour écrire un code portable sur différents systèmes, il est préférable de ne pas faire d'hypothèse sur l'implantation des `threads`.

On peut affecter aux `threads` différents niveaux de priorité. Il n'y a pas de mécanisme de préemption à priorité égale. Pour s'assurer du comportement d'un programme mettant en jeu plusieurs `threads`, il est plus sûr d'avoir recours à la synchronisation.

8.4 Synchronisation

8.4.1 Relation d'exclusion mutuelle : moniteurs

Les `threads` JAVA implantent un mécanisme de « moniteurs » protégeant l'accès aux objets « synchronisés ». L'exclusion mutuelle s'effectue sur l'appel des méthodes déclarées `synchronized` dans la définition de la classe de l'objet que l'on veut protéger. On peut déclarer `synchronized` un objet :

```
synchronized Object o = new Object(...);
```

ou une méthode :

```
synchronized type_de_retour nom_methode( ..) { ...}
```

Une seule exécution d'une méthode déclarée `synchronized` peut être effectuée en même temps, les autres appels sont alors bloqués. Il existe deux méthodes primitives principales `wait()` et `notify()` qui permettent une communication entre `threads` ayant accès au même moniteur. Un seul `thread` peut prendre le verrou (les autres attendent). Le verrou est rendu à la sortie du bloc de synchronisation.

Un moniteur est associé à une donnée et aux méthodes qui verrouillent l'accès à cette donnée. Il y a création d'un moniteur pour chaque objet qui possède au moins

une méthode `synchronized`. Quand un `thread` a pris le moniteur sur un objet (c'est-à-dire est entré dans l'exécution d'une méthode « synchronisée »), les autres `threads` voulant exécuter une méthode « synchronisée » sur cet objet sont bloqués. Quand le premier `thread` a terminé l'exécution du code de la méthode « synchronisée », il libère le moniteur qui peut alors être pris par un autre `thread`. Les moniteurs permettent aussi la communication entre `threads` par le mécanisme d'attente/notification (`wait()` et `notify()`).

8.4.2 Communication à l'intérieur d'un bloc de synchronisation

- o.wait() : relache le verrou et attend une notification
- o.notify() : relance une tâche en attente (une au hasard). Elle doit d'abord réacquérir le verrou.
- o.notifyAll() : relance toutes les tâches.

Exclusion mutuelle avec communication Producteur/Consommateur :

On donne ci-dessous un exemple de code réalisant une relation entre producteur et consommateur :

```
public class Sync {
    public static void main (String [] args) {
        Producteur p = new Producteur();
        p. start();
        Consommateur c = new Consommateur(p);
        c.start();
    }
}

class Consommateur extends Thread {
    Producteur mien;
    Consommateur(Producteur un){mien=un;}

    public void run() {
        while(true) {
            String r = mien.consomme();
            System.out.println("Consommation : "+r);
            try {sleep((int)(1000*Math.random()));}
            catch (Exception e0) {}
        }
    }
}

class Producteur extends Thread
{
    private String [] buffer = new String [8];
    private int ip = 0;
    private  int ic = 0;

    public void run() {
```

```
        while(true) {
            produce();
        }
    }

    synchronized void produce()
    {
        while (ip-ic+1 > buffer.length) {
            try{
                wait();
            }
            catch(Exception e) {}
        }
        buffer[ip % 8] = "Machine "+ip;
        System.out.println("produit : ["+(ip%8)+"] "+buffer[ip%8]);
        ip++;
        notify();
    }

    synchronized String consomme() {
        while (ip == ic) {try{wait();} catch(Exception e) {}}
        notify();
        return buffer[ic++%8];
    }
}
```

9 JAVADOC

JAVADOC est un utilitaire associé à JAVA qui génère automatiquement une documentation de code JAVA au format HTML.

La documentation fait apparaître l'ensemble des classes et de leurs relations hiérarchiques avec différents points d'entrée, par index, par package ou par arborescence d'héritage. Pour chaque classe apparaissent tous ses attributs et méthodes, avec la signature complète de chaque méthode.

Un usage intensif des liens hypertextes permet de naviguer efficacement parmi les définitions des diverses classes, notamment lorsqu'une classe est utilisée en paramètre d'une méthode d'une autre classe.

JAVADOC se distingue surtout par l'usage remarquable qu'il fait des commentaires insérés dans le code. Il existe un format standard pour les commentaires, que tout programmeur se doit de respecter s'il veut que ses commentaires apparaissent dans la documentation générée par JAVADOC.

Tout commentaire récupéré par JAVADOC doit être de la forme

```
/**
* Ici, le texte du commentaire
*
```

```
* ...
*/
```

On peut insérer dans le commentaire les références de l'auteur de l'entité commentée, son *e-mail*, voire tout lien hypertexte que l'on souhaite faire apparaître dans la documentation, de la manière suivante :

```
/**
* ...
*@author O. Sigaud (olivier.sigaud@lip6.fr)
* < A HREF= "Adapt/Action.html" > Lien < /A >
*/
```

Les autres commentaires, notamment ceux qui commencent par « // », sont ignorés, ainsi que les commentaires qui figurent dans le corps des méthodes.

On peut mettre des commentaires en en-tête d'une classe, pour expliquer ce que représente la classe, quel rôle elle joue dans le programme et/ou comment on s'en sert. C'est en quelque sorte une vue externe sur la classe. Il est fréquent d'insérer dans ces commentaires des liens HTML vers une autre classe à l'aide du mot clé `see` :

```
/**
* ...
*@see AutreClasse
*/
```

On peut aussi mettre des commentaires au-dessus de la définition de chaque attribut, pour expliquer ce que représente l'attribut.

On peut enfin mettre des commentaires en en-tête de chaque méthode. Là encore, le commentaire doit expliquer à quoi sert la méthode, ce qu'elle fait, et éventuellement comment elle le fait.

Le format des commentaires de méthodes est plus riche que celui des autres commentaires, car il permet de décrire tous les paramètres passés à la méthode, ainsi que sa valeur de retour. La syntaxe est la suivante :

```
/**
* ...
*@param pour chaque paramètre passé
*@return pour la valeur de retour
*/
```

On peut aussi ajouter le mot clé `deprecated` pour indiquer que la méthode commentée n'est plus utilisée, en général parce qu'une autre méthode rend le même service. Cela se fait de la façon suivante :

```
/**
* ...
```

```
*@deprecated
*/
```

Le compilateur émet un message d'alerte à chaque fois qu'il rencontre une méthode qui appelle une méthode notée `deprecated`.

La fonctionnalité d'insertion automatique de commentaires dans la documentation HTML d'un code constitue assurément pour les programmeurs une incitation très positive à commenter le plus clairement possible leurs programmes.

Cette fonctionnalité est d'autant plus précieuse qu'elle va de pair avec la « culture JAVA » qui consiste à mettre son code sur `Internet` à disposition d'autres programmeurs qui l'utiliseront sans consulter l'auteur et doivent donc comprendre le code de façon autonome.

10 La réflexivité

On appelle « réflexivité » le fait qu'un objet puisse consulter la structure de la classe à laquelle il est rattaché.

La réflexivité se traduit en JAVA par un ensemble de mécanismes qui augmentent considérablement la puissance du langage. Un objet peut par exemple s'auto-interroger pour savoir quel est son type, il peut se demander s'il hérite d'un type particulier, il peut récupérer le nom de sa classe dans une chaîne de caractères ou encore sauvegarder son état complet dans une autre chaîne de caractères. On peut aussi créer un objet à la volée à partir d'une chaîne de caractères décrivant le nom de la classe et les paramètres du constructeur qui sera appelé pour la création.

L'ensemble de ces mécanismes est assez disparate, nous ne les décrirons pas en détail ici. Pour en savoir plus sur la liste qui vient d'être énoncée, consultez la documentation disponible sur les points suivants.

– **instanceof** : opérateur qui vérifie si un objet est d'un type donné ou de l'une des sous-classes de ce type ;
– **getClass()** : méthode de la classe `Object` qui renvoie le type de l'objet (sous la forme d'un objet, instance de la classe `Class`, dans laquelle toute classe est décrite réflexivement). La classe `Class` possède notamment une méthode `getName()` qui renvoie le nom de la classe et s'avère bien utile ;
– **Serializable** : interface qui permet la persistance [24] ;
– **forname()** : méthode de la classe `Object` qui permet de créer un objet d'une classe à la volée à partir d'une chaîne de caractères contenant le nom de la classe et les paramètres du constructeur.

[24] voir la section 11.2 à la page 63.

Pour vous donner une idée de ce que peut faire l'introspection, essayez le bout de code suivant :

```
import java.lang.reflect.*;

public class Lecture {
  public static void main(String args[]) {
    Class c = null;
    Field[] champs = null;
    Method[] methodes = null;

    try {
      c = Class.forName(args[0]);
      champs = c.getDeclaredFields();
      methodes = c.getMethods();
    }
    catch (ClassNotFoundException e) { // ...;
                                       System.exit(0);}
    catch (SecurityException e) { // PB d'autorisation
                                        System.exit(0);}

    for (int i=0; i< champs.length;i++) {
      Field uc = champs[i];
      System.out.println("champs "+i+" : "+uc);
    }

    for (int i = 0; i < methodes.length; i++) {
      Method um = methodes[i];
      System.out.println("methodes "+i+ " : " + um);
    }
  }

}
```

Appliquez-le par exemple à des classes présentes dans les packages fournis avec JAVA.

11 Quelques exemples de code utile

11.1 Écriture dans un fichier

La classe `FilesWriter` ci-dessous permet d'écrire dans un fichier dont le nom a été passé au constructeur dans une `String`. Elle se charge de l'ouverture et de la fermeture du fichier.

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

public class FilesWriter implements Cloneable
{
```

```
protected FileOutputStream reportFileStream= null;
protected PrintWriter      reportWriter    = null;
protected String           outFileName;

public FilesWriter(String ofn)
  {
    outFileName                =        ofn;
  }
  /**
   *  Open the files that the reports will be written to
   *  during this test run. Called automatically by the
   *  constructor. Any re-call will re-open the files for
   *  writing, causing the previous file to be overwritten.
   **/
public void open ()
  {
    // Try to close the files if they are already open
    close();

    // Now try to open the files
    try
      {
        reportFileStream = new FileOutputStream (outFileName);
        reportWriter = new PrintWriter (reportFileStream, true);
      }
    catch (Exception e)
      {
        // Cannot create the file, so set it to null
        reportFileStream = null;
        e.printStackTrace();
      }
  }

  /**
   *  Close the files that the reports have been written
   *  to during the test run. Call once per run.
   *  Automatically called when the application closes.
   **/

public void close()
  {
    try
      {
        if (reportFileStream != null)
          reportFileStream.close();
      }
    catch (IOException e)
      {
        e.printStackTrace();
 // Do nothing - won't occur if the files have opened successfully
      }
  }
public PrintWriter out()
  {
    return reportWriter;
```

```
  }

  // usage du FilesWriter
public static void main(String args[])
  {
    String fileName = "nomFichier";
    FilesWriter        writer = new FilesWriter(fileName);
    System.out.println("Ouverture du fichier :" + fileName);
    writer.open();
    (writer.out()).println("donnée à inclure");
    writer.close();
  }
  // fin de FilesWriter
}
```

11.2 Sérialisation

On appelle « sérialisation » le processus consistant à écrire le contenu d'un objet dans un fichier pour pouvoir le recharger lors d'une exécution ultérieure dans l'état où il était au moment de sa sauvegarde.

Pour qu'on puisse appliquer la sérialisation à ses objets, une classe doit implémenter l'interface `Serializable` [25]. Ses attributs doivent faire de même, ainsi que les attributs de ces attributs, etc.

On peut alors sauvegarder et recharger le contenu de l'objet à l'aide des deux méthodes suivantes :

```
class MaClasse implements Serializable
{
  /**
   * Charge tout un objet de type MaClasse.
   * <table align=abscenter cellspacing=0 cellpadding=2 >
   * @param <tr><td>in </td><td>la streamer de lecture</td></tr>
   * </table>
   */

  private void readObject(ObjectInputStream in) throws IOException,
                                                       ClassNotFoundException
    {
        try
            {
                in.readObject();
            }
        catch(Exception e)
            {
                System.out.println("Erreur de chargement");
                e.printStackTrace();
            }
    }
```

[25] Sur les interfaces, voir la section 5.8 page 44

```
/**
 *Sauve tout l'objet de type MaClasse.
 *<table align=abscenter cellspacing=0 cellpadding=2 >
 * @param <tr><td>nomfic </td><td>la structure de fichier</td></tr>
 * </table>
 */

 private void writeObject(ObjectOutputStream out) throws IOException
    {
        try
            {
                out.writeObject(this);
            }
        catch(IOException e)
            {
                System.out.println("Erreur de sauvegarde");
                e.printStackTrace();
            }

    }
public static void main(String args[])
  {
      MaClasse        titi        =        new        MaClasse();
      System.out.println(titi);
      try{
      FileOutputStream fic_out= new FileOutputStream("test");
      ObjectOutputStream out= new ObjectOutputStream(fic_out);
      titi.writeObject(out);

      FileInputStream fic_in        = new FileInputStream("test");
      ObjectInputStream in         = new ObjectInputStream(fic_in);
      titi.readObject(in);
      System.out.println(titi);
      }
      catch(Exception e)
          {
              System.out.println("Erreur de sérialisation");
              e.printStackTrace();
          }
  }
  // fin de  MaClasse
}
```

11.3 Lecture du contenu d'une URL sur `internet`

Le langage JAVA a été orienté dès son origine vers l'échange d'informations sur `internet`, notamment au travers de la définition des applets qui assurent la sécurité des données présentes sur un site donné. À ce titre, de nombreuses classes ont été développées pour la manipulation de données présentes sur `internet`.

Le code qui suit permet de se connecter à une page web et d'en récupérer le contenu textuel. Il est extrêmement simple de le modifier pour en faire un automate de parcours

d'`internet`. Il suffit de repérer les liens dans le texte et d'appliquer récursivement les mêmes méthodes.

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

class LireURL
{
    public static void main(String args[])
    {
        URL url=null;
        String nomURL;
        if (args.length <1)
            nomURL="http://www.google.com";
        else
            nomURL=args[0];
        try
            {
                url=new URL(nomURL);
            }
        catch (MalformedURLException e)
            {
                System.out.println("URL incorrecte : "+nomURL);
                System.exit(1);
            }
        try {
            URLConnection connexion = url.openConnection();
            LineNumberReader lecture =
                new LineNumberReader(new InputStreamReader(
                                   connexion.getInputStream()));
            int         length = connexion.getContentLength();
            System.out.println("Longueur contenu "+ length);
            System.out.println("Type contenu "+ connexion.getContentType());
            System.out.println("Contenu :");
            String ligne;
            for (int i=0;i<length;i++)
                {
                    ligne = lecture.readLine();
                    if (ligne==null) System.exit(0);
                    System.out.println(ligne);
                }

        }
        catch (IOException e)
            {
                System.out.println("URL non autorisée: "+nomURL+"\n");
                System.exit(2);
            }
    }
}
```

A L'infra-structure de JAVA (Th. Bommart)

Ce qui suit est une libre retranscription (traduction et légers remaniements) de la série d'articles « *under the hood* » [26] de Bill Venners parue sur le site `internet`

http ://www.javaworld.com

Ces documents sont soumis à un copyright IDG.net :

• Pour la première partie : June 1996. Translated and reprinted with permission. http ://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.html

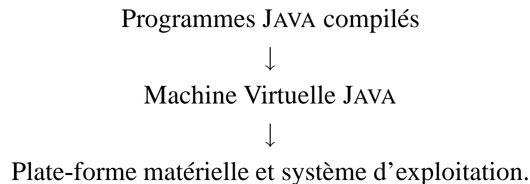
• Pour la seconde partie : July 1996. Translated and reprinted with permission. http ://www.javaworld.com/javaworld/jw-07-1996/jw-07-classfile.html

• Pour la troisième partie : August 1996. Translated and reprinted with permission. http ://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html

A.1 La machine virtuelle JAVA

A.1.1 Définition et rôle

La machine virtuelle JAVA ou JVM (pour Java Virtual Machine) est un ordinateur virtuel capable d'exécuter des programmes java compilés. La machine est virtuelle parce qu'elle est en général implémentée sous forme d'un programme au sommet d'un système d'exploitation sur une vraie machine. Tous les programmes JAVA sont compilés pour être exécutés par la JVM. La machine virtuelle JAVA doit être d'abord installée sur un ordinateur, avant de pouvoir faire tourner des programmes.



La machine virtuelle JAVA joue un rôle central dans la portabilité du langage JAVA. Elle forme une couche d'abstraction entre les programmes compilés et le matériel et le système d'exploitation sous-jacent. La portabilité est assurée parce que les programmes JAVA tournent sur la machine virtuelle, indépendamment de l'implémentation particulière de celle-ci.

Un des objectifs critiques du concept de machine virtuelle est que celle-ci soit la plus petite possible. Ce faisant, on peut la placer dans des télévisions, des téléphones cellulaires ou des ordinateurs personnels, entre autres.

[26] Sous le capot

A.1.2 Les bytecodes JAVA

Les programmes JAVA sont compilés sous une forme appelée « bytecodes JAVA ». La machine virtuelle JAVA exécute des bytecodes JAVA, donc on peut considérer les bytecodes comme le « langage machine » de la machine virtuelle JAVA. Le compilateur JAVA lit les programmes sources JAVA (les fichiers « *.java », traduit ces fichiers en bytecodes JAVA et place ces bytecodes dans des fichiers « *.class ». Le compilateur engendre un fichier `.class` pour chaque classe contenue dans les fichiers sources.

Source JAVA → **Compilateur java** → bytecodes java

Pour la machine virtuelle JAVA, une suite de bytecodes est une séquence d'instructions. Chaque instruction consiste en un octet de code-opération suivi de zéro, un ou plusieurs octets opérandes.

Pour chaque type d'octet du code-opération, on définit un code mnémonique. Les codes mnémoniques peuvent être vus comme le langage d'assemblage de la machine virtuelle. Par exemple, il existe une instruction qui provoque la mise d'un zéro au sommet de la pile. Le code mnémonique de cette instruction est *iconst_0*, et la valeur de son code-opération est 60 hexadécimal. Cette instruction n'a pas d'opérandes. Une autre instruction provoque un saut inconditionnel de l'exécution du programme en avant ou en arrière de la mémoire. Cette instruction nécessite un opérande, une valeur signée sur 16 bits qui spécifie l'ampleur du saut par rapport à la situation actuelle du compteur de programme. Le code mnémonique de cette instruction est *goto*, et son code-opératoire est a7 en hexadécimal.

code opération en hexadécimal	mnémonique du code opération et remarques
60	*iconst_0* (code opération)
–	(pas d'opérande)
a7	*goto* (code opération)
ff	1er octet de l'opérande offset de saut (sur deux octets)
f9	2ème octet de l'opérande offset de saut

FIG. 6 – Exemple de code machine de la JVM (extrait)

A.1.3 Les pièces virtuelles

La partie matérielle virtuelle de la machine virtuelle JAVA peut être décomposée en quatre parties principales : les registres, la pile, la zone de « tas » [27] qui est « garbage collectée » et la zone des méthodes. Ces parties sont virtuelles, tout comme la machine

[27] *heap*

qui les héberge, mais elles doivent exister dans chaque implémentation particulière (chaque programme) de machine virtuelle.

Une adresse dans la machine virtuelle est codée sur 32 bits. La machine virtuelle peut donc adresser jusqu'à 4 gigabytes (2 à la puissance 32) de mémoire, chaque emplacement mémoire étant constitué d'un octet. Chaque registre de la machine virtuelle est capable de stocker une adresse de 32 bits. La pile, la zone de tas et la zone des méthodes résident quelque part dans ces 4 gigabytes de mémoire adressable. Le choix de l'emplacement exact de ces zones de mémoire est un des choix du réalisateur d'une implémentation particulière de machine virtuelle.

Un « mot » dans la machine virtuelle JAVA est codé sur 32 bits. La machine virtuelle JAVA possède un petit nombre de types primitifs : *byte* (octet, 8 bits), *short* (entier court, 16 bits), *int* (entier, 32 bits), *long* (entier long, 64 bits), *float* (flottant court, 32 bits), *double* (flottant long, 64 bits) et *char* (caractère, 16 bits). A l'exception du type *char*, qui code des caractères `Unicode`, tous les types numériques sont signés. Ces types internes sont directement accessibles par le programmeur JAVA. Un autre type primitif est le type « pointeur sur objet » [28] qui est une adresse de 32 bits capable de référencer un objet dans la zone du tas.

Parce qu'elle contient des bytecodes, la zone des méthodes est alignée sur des frontières d'octets. La pile et la zone de tas sont alignées sur des frontières de mots de 32 bits.

A.1.4 Les peu, mais preux et valeureux registres

La machine virtuelle possède un compteur de programme et trois registres qui gèrent la pile. Elle ne possède que quelque registres parce que ses code-opérations opèrent en priorité sur la pile. Cette conception très orientée « machine à base de pile » permet de conserver un petit jeu d'instructions et une implémentation efficace.

La machine virtuelle utilise le compteur de programme (*pc-register*), pour conserver la position en mémoire où elle doit exécuter les instructions. Trois autres registres : *optop register* ou sommet de la zone opérandes, *frame register* ou registre de contexte d'exécution, *vars register* ou registre de zone des variables, pointent vers différents endroits du « *stack frame* » de la méthode en cours d'exécution. Le *stack frame* d'une méthode en cours d'exécution contient l'état (variables locales, résultats intermédiaires de calculs, etc..) pour une invocation particulière de cette méthode.

[28] *object handle*

A.1.5 La zone des méthodes et le compteur de programme

La zone des méthodes est l'endroit où sont stockés les bytecodes. Le compteur de programme pointe toujours vers un certain octet dans la zone des méthodes, c'est-à-dire qu'il contient l'adresse de cette zone. Le compteur de programme est utilisé pour conserver une trace dans le fil d'exécution des instructions. Après qu'une instruction de bytecode a été exécutée, le compteur de programme contient l'adresse de la prochaine instruction à exécuter. Celle-ci est soit l'adresse de la précédente instruction incrémentée de la quantité d'octets de celle-ci (code-opération plus opérande), soit une autre adresse dans le cas d'un saut.

A.1.6 La pile JAVA et les registres associés

La pile JAVA est utilisée pour stocker les paramètres et les résultats de l'exécution des bytecodes, pour passer des paramètres et récupérer des résultats d'appels de méthodes et pour conserver l'état (paramètres, variables..) d'une méthode appelante pendant l'appel d'une autre méthode. L'état d'une méthode en cours d'exécution est appelé « *stack frame* » (contexte d'exécution) et représente une certaine zone de la pile. Les registres *vars*, *frame* et *optop* pointent vers différents endroits de ce *stack frame*.

Il y a trois sections dans le *stack frame* : les variables locales, l'environnement d'exécution et la pile des opérandes. La zone des variables locales contient toutes les variables locales en cours utilisées pendant l'exécution de la méthode. Elle est pointée par le registre « *vars* ». La section de l'environnement d'exécution est utilisée pour conserver des informations sur la manipulation de la pile elle-même. Cette zone est pointée par le *frame register*. La zone opérande de la pile est utilisée comme zone de travail pour les instructions. C'est là que sont placés les paramètres des instructions et les résultats de leurs exécution. Le sommet de la zone opérande (la première adresse libre) est pointée par le registre *optop* [29].

La zone de l'environnement d'exécution est en général placée en sandwich entre la zone des variables locales et la zone de pile des opérandes. Cette zone des opérandes de la méthode en cours d'exécution est toujours la section la plus haute de la pile et le registre *optop* (sommet de la zone opérande) pointe en conséquence toujours au sommet de la pile JAVA globale.

[29] c'est donc un « pointeur de pile » classique

A.1.7 La zone du « tas », « garbage collectée »

La zone du tas est l'endroit ou « vivent » les objets d'un programme JAVA. À chaque création d'objet par l'opérateur *new*, la mémoire consommée par cet objet est allouée dans cette zone du tas. Le langage JAVA ne permet pas de libérer explicitement ou de dés-allouer directement cette mémoire. Par contre, l'environnement d'exécution [30] conserve les références vers chaque objet sur le tas et libère automatiquement la mémoire occupée par les objets qui ne sont référencés. C'est ce processus que l'on appelle « ramassage des déchets » [31].

A.2 Le « style de vie » fichier .class

Un des composants clefs de JAVA est le format `.class`, dans lequel sont compilés tous les programmes en JAVA. Ce format peut être chargé par toutes les implémentations de machines virtuelles et représente un moyen standardisé de transmission de classes compilées sur un réseau.

A.2.1 Né pour voyager

Le format « .class » est un format de fichier spécialement défini pour héberger le résultat de la compilation d'un code écrit en JAVA. Le code source en JAVA est compilé sous cette forme et ces fichiers dans ce format peuvent être téléchargés via un réseau avant d'être chargés et exécutés par la machine virtuelle. Ce format est indépendant de la plate-forme d'exécution finale. Les fichiers dans ce format contiennent des bytecodes (octets de code-opérations).

A.2.2 Qu'y a-t-il dans un fichier .class ?

Un fichier `.class` contient tout ce dont la machine virtuelle a besoin pour utiliser une classe ou une interface. Dans l'ordre d'apparition, on trouve : le nombre magique, la zone des constantes [32], les drapeaux d'accès [33], la classe elle-même, la super-classe dont elle hérite, les interfaces qu'elle implémente, ses méthodes et attributs.

Ces informations sont stockées sur une longueur variable, c'est-à-dire que l'on ne peut prévoir à l'avance la taille de ces fichiers avant de les charger. Par exemple, le nombre de méthodes listées est différent suivant les classes et dépend directement du nombre de méthodes déclarées dans le code source. Ce type d'information nécessite

[30] *runtime environment*
[31] *garbage collection*
[32] *constant pool*
[33] *access flags*

Nombre magique
0xCAFEBABE

Zone des constantes

Drapeaux d'accès

Classe

Super–classe

Interfaces
implémentées

méthodes

attributs

FIG. 7 – Exemple de contenu d'un .class

donc un en-tête précisant sa taille ou sa longueur, qui permet à la machine virtuelle de la charger correctement.

Les informations sont généralement écrites dans un fichier `.class`, sans recouvrement entre elles. Toutes les zones sont alignées sur des limites d'octet [34] ce qui permet au format de rester concis et facile à transporter.

L'ordre des composants dans le format `.class` est strictement défini. Par exemple, les huit premiers octets contiennent nécessairement le nombre magique et les numéros de version, la zone des constantes démarre au neuvième octet et la zone des drapeaux d'accès vient après. Mais comme la zone des drapeaux d'accès est de longueur variable, la machine virtuelle ne sait véritablement où commence la zone des drapeaux qu'une fois qu'elle a fini de lire la zone des constantes. Les deux octets suivants cette zone des constantes constituent donc les drapeaux ou bits de contrôle d'accès.

A.2.3 Nombre magique et numéros de versions

Les quatre premiers octets de n'importe quel fichier `.class` sont toujours "0xCAFEBABE" (soit la suite de petits entiers codés sur un octet : 12, 10, 15, 14, 11, 10, 11,

[34] la plus petite limite possible sur la quasi-totalité des systèmes

14, tout ceci en décimal). Ce nombre magique rend les fichiers `.class` facilement reconnaissables parce qu'il y a peu de chances de trouver accidentellement d'autre formats de fichiers commençant par la même suite d'octets. C'est aussi un nombre magique parce qu'il semble vraiment sorti d'un chapeau, son origine remontant à une période antérieure au choix du nom de « JAVA » [35] pour le langage, quand il fallait choisir un code facile à retenir et amusant. C'est par pure coïncidence que la référence aux jolies serveuses du Peet's Coffee a semblé préfigurer le nom « JAVA ».

Les quatre octets suivants contiennent les numéros de versions majeurs et mineurs. Ces numéros identifient la version du format que respecte ce fichier particulier et permet à la machine virtuelle de vérifier la validité du chargement : chaque machine virtuelle connaît une limite supérieure aux numéros de versions qu'elle peut charger, ce qui lui permet de rejeter des fichiers `.class` de versions plus récentes que cette limite.

A.2.4 La zone des constantes

Les constantes stockées dans cette zone peuvent être, par exemple, les chaînes de caractères littérales, les valeurs des variables finales, les noms de classe ou d'interface, les noms des variables ou des types, les noms des méthodes et leur signature [36].

La zone des constantes est organisée sous forme d'un tableau de longueur variable comportant plusieurs éléments. Chaque constante occupe un élément dans ce tableau. Partout dans le fichier `.class`, les constantes sont référencées par un index entier qui indique leur position dans ce même tableau. La première constante démarre avec un index valant 1, la deuxième a un index valant 2, etc. La zone des constantes est précédée de sa taille, ainsi la machine virtuelle sait combien de constantes elle doit charger, pour pouvoir lire la suite.

Chaque élément de la zone des constantes démarre avec une étiquette d'un octet qui spécifie le type de constante à cette position dans le tableau. Quand la machine virtuelle lit cette étiquette, elle est capable d'interpréter ce qui suit. Par exemple, si l'étiquette indique une constante de type `String`, la machine virtuelle sait que les deux octets suivants indiquent la longueur de cette chaîne. Ensuite, la JVM s'attend à trouver autant d'octets contenant les caractères de la chaîne qu'indiqué par cette longueur.

Dans la suite de cet article, nous nous référerons quelquefois au $n^{i\grave{e}me}$ élément de la zone des constantes sous la forme *constant_pool[n]*. Ceci a un sens dans la mesure ou la zone des constantes est organisée comme un tableau, mais il ne faut pas perdre de vue que ces éléments peuvent avoir des types et des tailles différents et que l'index du premier élément vaut 1 (et non pas 0 comme dans les tableaux classiques).

---

[35] café en argot californien

[36] La signature d'une méthode est définie comme la donnée de son nom, du nombre et du type de ses arguments d'entrée et de son type de retour, voir section 2.4.1, page 14

A.2.5 Drapeaux d'accès

Les deux premiers octets après la zone des constantes, la zone des drapeaux d'accès, indiquent si ce fichier `.class` définit une classe ou une interface et si cette classe ou interface est publique ou abstraite. Si c'est une classe, ils indiquent en outre si cette classe est finale.

A.2.6 La partie « `this_class` »

Les deux octets suivants, la partie *this_class* du fichier `.class`, est un index dans la zone des constantes. La constante référencée *constant_pool[this_class]*, possède deux parties : une étiquette de un octet et un index de nom [37] sur deux octets. L'étiquette aura pour valeur *CONSTANT_Class*, une valeur qui indique que cet élément contient des informations sur une classe ou une interface. *Constant_pool[name_index]* est une chaîne constante contenant le nom de la classe ou de l'interface.

Cette partie « *this_class* » donne un éclairage sur la manière dont la zone des constantes est utilisée. « *this_class* » n'est rien d'autre qu'un index vers la zone des constantes.

Quand la machine virtuelle recherche la valeur de *constant_pool[this_class]*, elle trouve un élément qui s'identifie lui-même comme une constante de classe à cause de son étiquette. La machine virtuelle sait que les éléments *CONSTANT_Class* possèdent toujours un index sur deux octets appelé *name_index*, immédiatement après l'étiquette. Elle regarde donc à la valeur *Constant_pool[name_index]* pour trouver le nom de la classe ou de l'interface.

A.2.7 La partie « `super_class` »

Après la zone « *this_class* », on trouve la zone *super_class*, un autre index sur deux octets vers la zone des constantes. *constant_pool[super_class]* est un élément de type *CONSTANT_Class* dont la partie index pointe vers le nom de la super-classe dont hérite la classe courante.

A.2.8 Interfaces

La partie « interfaces » commence par un décompte sur deux octets du nombre d'interfaces implémentées par la classe (ou l'interface) courante définie par le fichier. Immédiatement après vient un tableau d'index dans la zone des constantes, un par interface implémentée. Chaque interface implémentée est représentée par un élément

[37] *name_index*

CONSTANT_Class (étiquette + index sur deux octets) qui pointe vers le nom de cette interface.

A.2.9 Les attributs

La partie « attributs » commence par un décompte sur deux octets du nombre d'attributs de la classe ou de l'interface considérée. Après ce décompte vient un tableau d'éléments de longueur variable, un pour chaque attribut. Chaque structure détaille les informations telles que le nom de l'attribut, son type et sa valeur constante si l'attribut est une variable finale. Certaines informations sont directement contenues dans la structure elle-même et certaines résident dans la zone des constantes et sont pointées par la structure.

Les seuls attributs qui apparaissent dans cette partie sont ceux qui ont été déclarés dans le fichier source de la classe ou de l'interface considérée. Aucun attribut de superclasse ou de super-interface n'y apparaît.

A.2.10 Les méthodes

La partie des méthodes commence par un décompte sur deux octets du nombre de méthodes de la classe ou de l'interface considérée. Ce décompte ne vaut que pour les méthodes explicitement définies pour la classe ou l'interface considérée, aucune méthode héritée n'est comptée. Ensuite, on trouve les méthodes elles-mêmes. Pour chaque méthode, une structure regroupe des informations telles que la signature de la méthode, le nombre de mots nécessaires à réserver dans la pile pour les variables locales, le nombre maximum de mots de pile à réserver pour les opérandes, une table des exceptions capturées par la méthode, la séquence des bytecodes correspondant au corps de la méthode et une table de numéros de lignes.

A.2.11 Les propriétés

En fin de fichier `.class`, on trouve la zone des propriétés qui donne des informations générales sur la classe ou l'interface considérée. Cette zone commence par un décompte de ces propriétés sur deux octets, puis viennent les propriétés elles-mêmes. Un exemple de propriété est la propriété « fichier source » qui recèle le nom du fichier source à partir duquel a été compilé le code de la classe considérée. Les machines virtuelles écartent généralement sans le dire les propriétés qu'elles ne reconnaissent pas.

A.3 La zone du tas recyclable (garbage-collected heap) de la machine virtuelle JAVA

Résumé : Un des éléments clefs de l'environnement JAVA est sa « zone de tas recyclable » qui permet d'allouer des objets et de libérer ceux qui ne sont plus référencés. Parce que cette zone du tas est recyclable automatiquement, les programmeurs n'ont plus à explicitement libérer la mémoire. Voici une introduction à cet élément clef.

A.3.1 Introduction

La zone du tas de la machine virtuelle stocke tous les objets créés par l'exécution d'un programme JAVA. Ces objets sont créés à l'aide de l'opérateur JAVA « new ». La mémoire nécessaire à ces objets est dès lors immédiatement allouée dans la zone du tas. Le recyclage [38] est un processus qui permet de libérer automatiquement les objets qui ne sont plus référencés par le programme. Ceci libère le programmeur de la tâche du suivi des moments où il doit libérer la mémoire et évite ainsi beaucoup de problèmes difficiles à traiter et de bugs potentiels.

Le terme « garbage collection » implique que les objets qui ne sont plus nécessaires au programme en cours d'exécution sont des déchets [39] et qu'ils peuvent être détruits. Une métaphore plus moderne et plus précise pourrait être celle du « recyclage de la mémoire ». Quand un objet n'est plus nécessaire, l'espace qu'il occupe dans la zone du tas doit être recyclé pour être disponible pour les éventuels futurs nouveaux objets. Le recycleur [40] doit donc déterminer quels objets ne sont plus référencés, et libérer l'espace qu'ils occupent. Pendant ce processus de libération de la mémoire, le *garbage collector* doit activer toutes les méthodes de finalisation de ces objets en cours de recyclage.

En plus de récupérer l'espace occupé par les objets inutilisés, le *garbage collector* permet de combattre la fragmentation de la mémoire. Cette fragmentation est naturelle et normale pendant l'exécution d'un programme. De nouveaux objets sont créés et alloués, d'autres sont détruits et l'espace qu'ils occupaient est libéré, ce qui crée des trous entre les objets encore utiles. Une requête d'allocation de nouveaux objets ne pourrait être honorée que par extension de la zone du tas, même si la somme des espaces inoccupés est suffisante *a priori* pour cette requête. Ceci arrive s'il n'y a pas assez d'espace contigu dans lequel le nouvel objet puisse être casé. Dans un système à mémoire virtuelle, le nombre et la répétition des demandes de pages supplémentaires nécessaires et l'extension continue de la zone du tas peut notablement dégrader les performances

[38] *garbage collection*
[39] *garbage*
[40] *Garbage Collector (ou GC)*

d'un programme.

Cette section ne décrit pas le fonctionnement d'un *garbage collector* « officiel » puisqu'aucun programme de ce genre n'existe en fait. Les spécifications de la machine virtuelle précisent seulement que la zone du tas doit être recyclée, mais ne disent rien sur la façon dont un recycleur de mémoire doit fonctionner. Les concepteurs de chaque machine virtuelle doit décider comment implémenter ce recyclage. Nous décrivons dans la suite plusieurs techniques de recyclage.

A.3.2 Pourquoi recycler la mémoire ?

Le recyclage de la mémoire libère le programmeur de la charge de libération de la mémoire allouée. Savoir quand exactement libérer la mémoire peut être extrêmement difficile. Laisser ce travail à la machine virtuelle représente plusieurs avantages. Tout d'abord, cela rend plus productif le programmeur. Dans les environnements et langages sans recyclage automatique de la mémoire, le programmeur peut passer de longues heures (voire des jours et des semaines) à traquer un problème évanescent d'erreur en mémoire. Dans l'environnement JAVA, le programmeur peut utiliser ce temps pour prendre de l'avance sur son planning ou rentrer chez lui plus tôt.

Un deuxième avantage du recyclage est qu'il aide à assurer l'intégrité du programme. Le recyclage est une part importante de la stratégie de sécurité de l'environnement JAVA. Les programmeurs JAVA sont incapables de « planter » accidentellement (ou volontairement) la machine virtuelle en ne libérant pas la mémoire.

Un désavantage potentiel du recyclage de la mémoire de la zone du tas est qu'il rajoute une charge de calcul qui peut dégrader les performances du programme. La machine virtuelle doit garder trace de toutes les références vers les objets, appeler les méthodes de finalisation et libérer l'espace récupérable, tout en assurant l'activité principale du programme de l'utilisateur. Cette charge globale est assurément plus grande que si l'utilisateur assure lui-même la gestion de la libération de la mémoire. Enfin, les programmeurs d'un environnement avec recyclage ont peu ou pas de contrôle sur les cycles CPU dévolus à la récupération des ressources mobilisées par les objets inutiles.

Heureusement, de très bons algorithmes de recyclage ont été développés et des performances correctes peuvent être atteintes pour presque toutes les applications, même les plus exigeantes. Parce que le *garbage collector* fonctionne avec son propre `thread`, il fonctionnera la plupart du temps d'une manière complètement transparente vis-à-vis du programme principal. De plus, si un programmeur veut explicitement demander un recyclage de la mémoire à un moment donné, les méthodes *System.gc()* ou *Runtime.gc()* peuvent être appelées pour une action immédiate de celui-ci.

Le programmeur JAVA doit garder à l'esprit que c'est le *garbage collector* qui appelle les méthodes de finalisation sur les objets. Parce qu'il n'est généralement pas

possible de prédire exactement quand les objets seront récupérés, il n'est en général pas possible de prévoir exactement quand les méthodes de finalisation seront appelées. En conséquence, les programmeurs ne doivent pas écrire du code dont le comportement correct serait basé sur la date d'appel de ces méthodes. Par exemple, si une méthode de finalisation, appelée sur un objet récupérable, libère une ressource apte à être de nouveau réclamée plus tard par le même programme, cette ressource ne sera pas disponible tant que le *garbage collector* n'a pas appelé la méthode de finalisation. Si, de plus, la ressource est réclamée avant cette échéance, un problème grave peut survenir.

A.4 Algorithmes de recyclage

Une grande quantité d'études ont été menées dans le domaine des algorithmes de recyclage des objets. Beaucoup de techniques développées à cette occasion peuvent être employées dans une machine virtuelle. La gestion de la zone du tas et son algorithme de recyclage est un des domaines où les concepteurs de machines virtuelles peuvent espérer se distinguer dans la compétition sur les performances.

Tout algorithme de recyclage de la mémoire se doit de faire au moins deux choses. Premièrement, il doit détecter les objets « déchets » récupérables. Deuxièmement, il doit réclamer l'espace du tas utilisés par ces objets et le rendre disponible pour le reste du programme.

Il existe deux approches de base pour distinguer les objets vivants des objets déchets : le comptage des références et le traçage. Les *garbage collectors* basés sur le comptage de références opèrent en actualisant un compteur pour chaque objet dans la zone du tas. Ce compteur indique le nombre de références vers cet objet. Les *garbage collectors* basés sur le traçage, eux, cartographient le graphe des références qui partent d'un ensemble de *points d'ancrages*. Les objets rencontrés lors de ce processus sont marqués d'une certaine manière. Quand la trace est terminée, les objets non marqués sont considérés comme inaccessibles et peuvent donc être récupérés.

A.4.1 Recycleurs basés sur le comptage des références

Cette technique est une des premières dans l'histoire des stratégies de recyclage des objets. Un compteur de références est maintenu pour chaque objet. Quand un objet est créé, son compteur est positionné à un. Quand tout autre objet ou point d'ancrage est assigné avec une référence à cet objet, le compteur est incrémenté. Quand une référence vers cet objet sort du contexte (sortie d'une zone d'une variable locale par exemple) ou si la référence prend une autre valeur, le compteur est décrémenté de un. Tout objet dont le compteur arrive à zéro est déclaré récupérable. Quand un objet est récupéré, tous les objets qu'il référence ont leurs compteurs décrémentés. De cette manière, le

recyclage d'un objet peut conduire à plusieurs recyclages d'autres objets.

L'avantage de cette technique est qu'elle peut se dérouler dans de petits laps de temps entremêlés avec l'exécution du programme utilisateur. Cette caractéristique la destine tout particulièrement au domaine du temps réel où un programme utilisateur ne peut pas être interrompu très longtemps.

Le désavantage du comptage de références est qu'il ne détecte pas les références cycliques. Des objets reliés par une référence cycliques sont deux ou plusieurs objets qui se référencent les uns les autres, par exemple, un objet père qui pointe sur un objet fils qui pointe lui même sur le père. Ces objets n'auront jamais leurs compteurs de références à zéro, même s'ils ne sont plus accessibles à partir des points d'ancrages du programme principal. En effet, supposons la situation dans laquelle un objet A possède une référence sur un objet B et, réciproquement, B possède une référence sur A. Par ailleurs, un objet C possède une référence sur A qui est le seul point d'entrée dans l'anneau (A,B). Alors, comme il apparaît sur la figure 8, il existe deux références sur A (celle de B et celle de C) et une référence sur B (celle de A).

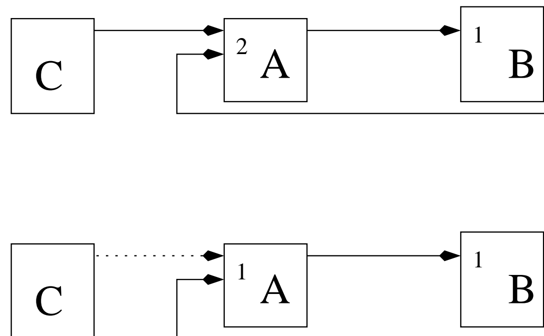


FIG. 8 – Perte de la référence sur un anneau

Lorsqu'on libère l'emplacement mémoire correspondant à l'objet C, celui-ci libère sa référence sur A. Le compteur de références sur A tombe à 1, donc l'emplacement mémoire correspondant à l'objet A n'est pas libéré. Pourtant, C était le seul point d'entrée dans le programme vers l'anneau (A, B). A et B se connaissent mutuellement, mais ils ne seront jamais libérés puisqu'aucun autre élément du programme n'est susceptible de décrémenter leur compteur de référence (ils ne sont plus accessibles à partir d'aucun point d'ancrage).

Cette situation se reproduit à chaque fois que des références forment un anneau, ce qui est très fréquent.

Un autre désavantage est le surcroît de temps CPU dédié à l'incrémentation ou la décrémentation des compteurs. À cause de ces désavantages, le comptage de références

n'est plus très en vogue à l'heure actuelle. Il est fort probable que les machine virtuelles que vous rencontrerez dans le monde réel soient pourvues d'algorithmes de traçage pour la gestion de leur zone de tas.

A.4.2 Recycleurs basés sur le traçage des objets

La phase de détection des objets récupérables est habituellement réalisée en définissant un ensemble d'ancrages et en calculant une accessibilité à partir de ces ancrages. Un objet est accessible s'il existe un chemin à partir d'un de ces points d'ancrage par lequel le programme utilisateur peut y accéder. Les points d'ancrages sont toujours basiquement accessibles. Tout objet accessible dans ce sens est déclaré « vivant », ceux qui ne le sont pas sont considérés comme « déchets », parce qu'il ne peuvent plus influer sur le cours restant de l'exécution du programme.

Les recycleurs basés sur le traçage des objets tracent le graphe des références aux objets qui démarrent des points d'ancrages. Les objets rencontrés pendant le traçage sont marqués d'une certaine manière. Ce marquage est généralement fait soit en basculant un indicateur dans l'objet lui-même ou un indicateur dans une zone de bitmap séparée. À la fin du traçage, les objets non marqués, réputés donc inaccessibles, sont aptes à la récupération.

L'algorithme de traçage de base est appelé « *mark and sweep* » (marque et balaye). Ce nom se réfère aux deux phases du processus. Dans la phase de marquage, le recycleur traverse l'arbre des références et marque chaque objet qu'il rencontre. Dans la phase de *sweep* (balayage), les objets non marqués sont récupérés et la mémoire récupérée est rendue disponible au programme utilisateur. Dans une machine virtuelle, la phase de balayage doit inclure l'appel aux méthodes de finalisation sur les objets en cours de récupération.

Certains objets possèdent des méthodes de finalisation, d'autres pas. Les objets non marqués sans méthode de finalisation peuvent être récupérés immédiatement, sauf s'ils sont référencés par un objet finalisable non marqué. Tous les objets référencés par un objet finalisable doivent rester dans la zone du tas tant que cet objet n'a pas été finalisé.

Dans une machine virtuelle, la façon d'organiser les points d'ancrage est dépendant de l'implémentation, mais elle inclut toujours les références atteintes à partir des variables locales, les références atteintes par la zone opérande de tout *stack frame* de tout `thread` de la JVM et toutes les références atteintes à partir des variables de classe. Dans une machine virtuelle, tout objet réside dans la zone de tas. Les variables locales résident dans la pile JAVA et chaque `thread` d'exécution possède sa propre pile d'exécution. Chaque variable locale est donc soit un pointeur sur un objet, soit un type primitif comme un `int`, un `char` ou un `float`. Donc les points d'ancrages d'une machine virtuelle comprendront toutes les références (tous les pointeurs) aux objets de

toutes les piles de tous les `threads`. Une autre source de points d'ancrages est donnée par l'ensemble des références à des objets, tels que les `Strings`, dans la zone des constantes des classes chargées. Cette zone des constantes peut référencer des chaînes de caractères stockées dans la zone du tas, comme le nom de la classe, le nom de la super-classe ou de la super-interface, les noms de champs et leurs signatures, les noms des méthodes et leurs signatures.

Tout objet référençable à partir d'un point d'ancrage est donc un objet vivant. De plus, tout objet référencé à partir d'un objet vivant est lui aussi accessible. Comme le programme utilisateur peut accéder à tous ces objets, ils doivent rester dans la zone du tas. Tout autre objet non accessible peut donc être récupéré.

La machine virtuelle peut être implémentée de manière à faire la différence entre une référence à un objet original ou une référence par le biais d'un type primitif (par exemple, un `int` qui fait 32 bits de long) qui pourrait apparaître comme une référence valide à un objet. Cependant, certains algorithmes peuvent choisir de ne pas faire cette distinction entre références à de vrais objets originaux et « fausses références ». De tels algorithmes sont appelés « conservatifs » parce qu'ils ne libèrent pas toujours tous les objets non référencés. Quelquefois, un objet déchet sera faussement considéré comme vivant par un algorithme conservatif, parce qu'une « fausse référence » référence pointe sur lui. Ces algorithmes conservatifs présentent généralement une vitesse accrue de recyclage en compensation du fait qu'ils ne récupèrent pas toujours tous les objets.

A.4.3 Recycleurs compacteurs

Les recycleurs des machines virtuelles ont souvent une stratégie pour combattre la fragmentation de la zone du tas. Deux stratégies souvent utilisées par les recycleurs du type « *mark and sweep* » sont le compactage et le copiage. Ces deux techniques déménagent carrément les objets sur le champ pour réduire la fragmentation. Les algorithmes compacteurs « tassent » les objets vivants dans une extrémité libre de la zone du tas. À la fin de ce processus, l'autre extrémité de la zone du tas devient une grande zone d'espace libre et contigu. Toutes les références vers les objets déménagés sont modifiées pour qu'elles pointent vers leurs nouveaux emplacements.

Cette modification des adresses des objets « déménageables » est quelquefois réalisée simplement en rajoutant un degré supplémentaire d'indirection entre les références et les vrais objets. Au lieu de pointer directement vers les objets dans la zone du tas, les références vers des objets pointent vers une table de pointeurs qui pointent eux-mêmes vers les vrais objets dans la zone du tas. Quand un objet est déménagé (son adresse change) seul le pointeur dans la table est changé avec cette nouvelle adresse. Toutes les références à cet objet dans le programme utilisateur continueront à pointer vers la table au même endroit sans rien changer. Bien qu'elle simplifie la défragmentation de

la zone du tas, cette approche grève les performances à chaque accès aux objets.

A.4.4 Recycleurs copieurs

Les recycleurs copieurs déménagent tous les objets vivants vers une nouvelle zone. Au fur et à mesure de leur déménagement, les objets sont collés les uns aux autres dans leur nouvel emplacement, supprimant ainsi tout espace qui existait entre eux à leur ancien emplacement. Cet ancien emplacement devient une zone libre. L'avantage de cette méthode est que les objets peuvent être déplacés au fur et à mesure de leur découverte pendant la traversée de l'arbre depuis les points d'ancrages. Il n'y a plus dès lors de séparation entre les phases de marquage et de balayage. Au moment où l'on déménage un objet, le point d'ancrage amenant à cet objet est évidemment changé avec le nouvel emplacement de l'objet copié. Mais qu'advient-il si plus tard un autre objet référence un « vieil » objet déjà déménagé ? On utilise pour cela un pointeur supplémentaire [41] entre l'ancien objet et le nouveau, qui permettra au *garbage collector* de placer l'adresse correcte dans cette référence découverte tardivement.

Un type classique de recycleur copieur est appelé recycleur « *stop and copy* ». Dans ce type de recycleur, la zone du tas est séparée en deux parties, dont une seule sert à un moment donné. Les objets sont alloués à partir d'une de ces régions jusqu'à ce qu'elle soit saturée. À ce moment, le programme en exécution est stoppé et la zone du tas est explorée (traversée). Les objets vivants sont copiés dans l'autre région au fur et à mesure de leur découverte pendant la traversée. Quand la traversée et les copies sont terminées, le programme utilisateur reprend. La mémoire est alors allouée à partir de la nouvelle zone de tas, jusqu'à ce qu'elle soit saturée elle aussi. À ce moment là, le programme utilisateur est encore stoppé. La zone est re-traversée et les objets sont de nouveau copiés dans la première zone. Le coût majeur associé à cette technique est qu'il faut deux fois plus de mémoire qu'une quantité donnée allouée à la zone de tas, puisqu'à tout moment, seule une moitié de cette mémoire est utilisée.

[41] *forwarding pointer*

B Liens

Le site de Javasoft : *http ://www.javasoft.com/* (incontournable, mais terriblement mal agencé)

L'API de JAVA 1.4.2 : *http ://java.sun.com/j2se/1.4.2/docs/api/index.html*

Les tutorials de JAVA : *http ://java.sun.com/docs/books/tutorial/index.html*

Le site du cours : *http ://animatlab.lip6.fr/Sigaud/teach/in204*

Un cours sur la programmation orientée objets distribuée :
http ://www.pps.jussieu.fr/˜emmanuel/Public/enseignement/POD_2000_WWW/Poly/poly2.html

Références

[AG96] K. Arnold and J. Gosling. *Le langage Java*. Traduction S. Chaumette et A. Minuissi, Thomson Publishing, 1996.

[Eck98] B. Eckel. *Thinking in Java*. Prentice Hall, 1998.

[Flaon] D. Flanagan. *Java in a nutshell*. O'Reilly, 1997, second edition.

[KR85] D. Kernigham and A. Richie. *Le langage C*. Hermes, France, 1985.

[Mir99] A. Mirecourt. *Le développeur Java2*. Osman Eyrolles Multimedia, France, 1999.

[Sig05] O. Sigaud. *Introduction à la modélisation orientée objets avec UML*. support du cours in204 « Génie logiciel et programmation orientée objets » de l'ENSTA, 2005.

MCours.com