Gestion de la mémoire

Introduction

Il y a plusieurs façons d'utiliser la mémoire dans un langage de programmation.

Les principales sont:

- La gestion statique de la mémoire
- La gestion automatique de la mémoire
- La gestion dynamique de la mémoire

La gestion statique de la mémoire.

Cette façon de gérer la mémoire est relativement simple à comprendre.
La mémoire est subdivisée en millions de cellules élémentaires adressables appelées octets ou "bytes".
Une région de mémoire (ou plusieurs) est réservée pour y placer les variables et les constantes qui ne changeront pas de place en mémoire pour tout le temps de l'exécution du programme. Le programme accède ces variables à l'aide de leurs adresses.

Ces régions sont communément appelées "régions statiques". C'est là qu'on trouve entre autres certaines constantes littérales (ex: 3.2, 12433 etc.) et les variables dites static

La gestion automatique de la mémoire (usage de la pile).

Lorsqu'un programme est compilé (ou interprété selon le cas), le compilateur assigne les variables déclarées, au début des méthodes et fonctions, dans le haut d'un espace mémoire appelé "pile".

Cette région de mémoire s'appelle pile par analogie avec une pile d'assiettes:

> Pile d'assiettes
> Si nous voulons une assiette, nous la prenons du dessus de la pile et non quelque part dans le milieu de celle-ci. De la même façon, lorsque nous replaçons une assiette, nous la replaçons sur le dessus de la pile et non en son milieu.

Exemple de programmes et d'états correspondants de la pile.

```
Public static void main(String a[])
{
      int a, b;
      float x;
      int c, d;

// point 1

      a = 5;
      c = 9;
      d = 20;
      x = 22.1;

// point 1a

      b = fnb();

// point 1b

      x = fnx(a);

// point 1c

      a = fnb();

// point 1d
}


public static double fnx(int xa)
{
      int g, h;

// point 4

      g = 7;
      h = 22;

//point 4a

      return g + h + xa;
}
```

```
public static int fnb()
{
      int u, v;
      float y;
      int w;

// point 2

      u = 51;
      v = 53;
      y = 41.2;

//point 2a

      w = fnw();

// point 2b

      return u + v + (int)y + w;
}


public static int fnw()
{
      int q, r;
      float z;
      int p;

// point 3

      q = 2;
      p = 3;
      r = 19;
      z = 41.2;

// point 3a

      return r + q + p + (int)z;
}
```

Gestion de la mémoire

Différents états de la pile lors de l'exécution du main situé à la page précédente.

Chaque colonne représente l'état de la pile au point indiqué dans le commentaire du programme.

1	1a	2	2a	3	3a	2b	1b	4	4a	1c	2	2b	1d	
				haut	haut									
				p: 0	3									
				z0.0	41.2									
				r: 0	19									
		haut	haut	q: 0	2	haut					haut	haut		
		w: 0	0	0	0	65		haut	haut		w: 0	65		
		y0.0	41.2	41.2	41.2	41.2		h: 0	22		y0.0	41.2		
		v: 0	53	53	53	53		g: 0	7		v: 0	53		
haut	haut	u: 0	51	51	51	51	haut	xa 5	5	haut	u: 0	51	haut	
d: 0	20	20	20	20	20	20	20	20	20	20	20	20	20	
c: 0	9	9	9	9	9	9	9	9	9	9	9	9	9	
x 0.0	22.1	22.1	22.1	22.1	22.1	22.1	22.1	22.1	22.1	34.0	34.0	34.0	34.0	
b: 0	0	0	0	0	0	0	210	210	210	210	210	210	210	
a: 0	5	5	5	5	5	5	5	5	5	5	5	5	210	haut

Nous voyons bien que la pile monte et descend lors de l'appel des diverses fonctions.

Remarquez entre autres qu'au point 4 les variables placées dans le haut de la pile ont remplacé celles du point 2.

Une conséquence de ceci est l'économie de mémoire: en effet, comme les variables placées au haut de la pile (ou empilées) remplacent celles qui ont été enlevées du haut de la pile (ou dépilées), seule la mémoire requise par les fonctions qui sont actives (en train de s'exécuter) est utilisée par le programme. Ce qui donne dans notre exemple: les variables du point 2 n'étant plus requises au point 1b, elles ont été enlevées du haut de la pile (ou dépilées) ce qui libère de la mémoire. Ensuite lorsque fnx(a) s'exécute au point 1b à 1c, la fonction fnx() requiert la mémoire dont elle a besoin de la pile: ses variables sont placées dans le haut de la pile (ou empilées) et remplacent donc celles de la fonction fnb() dont l'exécution s'est terminée.

Règles générales: Lors de l'exécution d'une fonction, les variables et paramètres déclarés dans la fonction sont empilés.
Lorsque la fonction se termine, ses variables et paramètres sont dépilés.

Une autre conséquence, plus importante pour le programmeur, de cette façon de gérer la mémoire est que lorsqu'une fonction se termine, les valeurs actuelles de ses

variables locales sont détruites. Donc ces valeurs ne sont plus accessibles lors d'un appel subséquent. (ex: 2b suivi de 1d)
La plupart des compilateurs modernes placent les variables locales des programmes dans une telle région. Par exemple: C, C++, Java, Pascal, Simula, VB7 etc. etc.

> Note historique
> Le premier compilateur à utiliser cette notion fut Algol 60. Publié en 1960 ce langage révolutionna la théorie des langages et méthodes de compilation utilisées jusqu'alors; il devint un standard utilisé par la communauté scientifique internationale.
> Quelques années plus tard, vers 1965, la compagnie IBM publia un langage nommé PL/1, utilisé jusqu'à nos jours, qui était basé sur cette notion de pile.

La gestion dynamique de la mémoire (usage du "heap")

Note: le mot "heap" signifie tas en français.

Cette forme de gestion de mémoire est principalement utilisée pour y placer les objets générés en cours d'exécution. À chaque fois que le verbe **new** est exécuté, il y a un segment de mémoire du "heap" qui est réservé pour y placer des données (un objet). La longueur du segment réservé correspond évidemment à la taille de l'objet requis.

Afin de localiser le segment de mémoire, l'adresse mémoire la plus petite de l'objet généré est utilisée. Cette adresse est aussi nommée "référence" (Il y a d'autres façons de représenter des références cependant ces méthodes terminent toujours par une adresse mémoire).

Dans le programme. la référence est souvent gardée dans une variable locale ou static c'est-à-dire dans la pile ou dans une région statique de mémoire. Dans le cas de la composition, soit d'objets composés d'autres objets, la référence à un objet B contenu dans un objet A est conservée dans le segment du "heap" correspondant à l'objet A.

Gestion de la mémoire

Exemple simple (sans composition)

```
public class Affaire{
      int mA;
      int mB;
      double mX;
      int mC;
}
```

```
public static void main(String a[])
{
      int u;

      u = 10;

// point 1

      fna();

// point 5

      . . .
}
```

```
public static void fna()
{
      int a, b;

// références vers objets Affaire
      Affaire affA, affB;

// point 2

      a = 5;
      b = 19;
      affA = new Affaire();

// point 3

      affB = affA;

// point 4

      . . .

}
```

Pile

1	2	3	4	5
	haut	haut	haut	
	affB: null	0	200,000	
	affA: null	200,000	200,000	
	b: 0	19	19	
haut	a: 0	5	5	haut
u: 10	10	10	10	10

"Heap"

	←Adresse: 200,000				
	mA: 0	mB: 0	mX: 0.0	mC: 0	

Initialement, au point 2, la fonction fna() voit ses variables initialisées à leurs valeurs par défaut soit 0 pour int et null pour les références.

Ensuite, au point 3, nous remarquons que la référence à l'objet Affaire est gardée dans la variable affA située dans la pile tandis que l'objet créé est gardé dans le "heap" à l'adresse 200,000. Les propriétés de l'objet n'étant pas initialisées par un constructeur, chacune a reçu la valeur par défaut correspondant à son type: soit 0 pour int et 0.0 pour double.

À la fin de fna(), au point 4, nous voyons que la référence affB a reçu une copie de la référence affA: soit une copie de l'adresse de l'objet contenue dans affA. Remarquez que l'objet lui-même n'est pas copié dans ce cas; il existe en une seule copie sauf que deux variables différentes s'y réfèrent.

À la fin du main, au point 5, l'objet existe encore dans le "heap" cependant aucune variable du programme ne s'y réfère (ou contient son adresse). Dans un langage comme C ou C++ une telle situation est considérée comme une erreur (appelée coulage de mémoire [de l'anglais "memory leak"]) Dans le langage Java (ou VB) ceci n'est pas une erreur car la récupération de ces espaces perdus se fait automatiquement au besoin (voir plus loin).

Exemple plus complexe (avec composition)

```
public class BonneAffaire{
      int mU;
      int mV;
      Affaire mAff;
      int mW;

      public BonneAffaire()
      {}

      public BonneAffaire(int x)
      {
            mU = x;
            mAff = new Affaire();
      }
}

public static void main(String a[])
{
      int u = 10, k;
// point 1
      fnb()
// point 7

// Cette boucle fait cent millions
// d'itérations
      for(k=0; k < 100000000; k++)
            fnb();
// point 8
}
```

```
public static void fnb()
{
      int a, b;
      BonneAffaire bonA, bonB, bonC;

// point 2

      a = 5;
      b = 19;

// point 3

      bonA = new BonneAffaire();

// point 4

      bonC = new BonneAffaire(b);

// point 5

      bonB = bonA;

// point 6
}
```

Pile

point 1	2	3	4	5	6	7	8
	haut	haut	haut	haut	haut		
	bonC: null	null	null	400,000	400,000		
	bonB: null	null	null	null	300,000		
	bonA: null	null	300,000	300,000	300,000		
	b: 0	19	19	19	19		
haut	a: 0	5	5	5	5	haut	haut
u: 10	10	10	10	10	10	10	10
k: 0	0	0	0	0	0	0	100,000,000

"Heap"

	←Adresse 300,000				
	mU: 0	mV:0	mAff: null	mW: 0	
	←Adresse 400,000				
	mU: 19	mV: 0	mAff: 450,000	mW: 0	
←Adresse: 450,000					
mA: 0	mB: 0	mX: 0.0		mC: 0	

Les points 1 à 4 et 6 de cet exemple sont semblables aux points 1 à 4 de l'exemple précédent, sauf qu'une référence à un objet (mAff) doit être initialisée par défaut (à null) dans l'objet BonneAffaire créé.

Nous remarquons dans cet exemple que l'objet Affaire, référé par mAff, contenu dans la classe BonneAffaire doit être généré. Dans le premier constructeur soit BonneAffaire(), il ne l'est pas donc mAff sera initialisée à null lors de l'utilisation de ce constructeur. Ceci correspond aux points 3 et 4: pour l'objet BonneAffaire placé à l'adresse 300,000 de l'exemple.
L'adresse de l'objet BonneAffaire généré dans ce cas-ci est gardée dans la variable bonA de la fonction fnb().

Dans le deuxième constructeur [soit BonneAffaire(int x)], un objet Affaire est généré; donc une référence vers l'objet Affaire (placé à l'adresse 450,000 dans le "heap" pour les besoins d'illustration de cet exemple) sera placée dans la propriété mAff de l'objet. Ceci correspond au point 4 et 5 pour l'objet BonneAffaire placé à l'adresse 400,000 de l'exemple.
L'adresse de l'objet BonneAffaire généré dans ce cas-ci est gardée dans la variable bonC de la fonction fnb().

Le point 7 de cet exemple est semblable au point 5 de l'exemple précédent. Nous pouvons y appliquer les mêmes commentaires.

Que dire maintenant du point 8? Lors de l'exécution de la boucle, la fonction fnb() est appelée cent millions de fois ce qui produira la réservation de mémoire pour deux objets BonneAffaire (environ 16 octets chacun) et un objet Affaire (environ 20 octets) à chaque tour de boucle. Il est à remarquer que la fonction fnb() ne détruit pas la mémoire dont elle n'a plus besoin dans le "heap" avant de terminer car il lui est impossible de le faire en Java (ou VB). La mémoire requise pour exécuter la partie de ce programme placée entre les points 7 et 8 sera donc d'environ 52 octets (16 * 2 + 20) par itération pour un total de 52 * 100,000,000 soit 5.2 milliards d'octets! Si l'ordinateur sur lequel ce programme sera exécuté a une mémoire centrale de moins de 5.2 giga octets, le "heap" sera rempli à pleine capacité plusieurs fois.

Au point 8, remarquez que tout l'espace utilisé par la boucle est considéré comme des détritus par notre programme (car il ne s'en sert pas) et par Java (ou VB) car aucune référence du programme (soit de la pile ou des régions statiques) ne contient l'adresse d'un quelconque de ces segments du "heap".
Lors d'une itération de la boucle la fonction fnb() n'utilise que 52 octets du "heap"; aussitôt qu'elle se termine ces 52 octets, n'étant plus référés par une variable de la pile, deviennent des "vidanges".

Dans la plupart des cas nos ordinateurs ne contiennent pas autant de mémoire centrale ce qui fait que le ramasseur d'ordures de Java (ou VB) appelé System.gc()

(pour "garbage collector") sera automatiquement appelé plus d'une fois dans la boucle afin de libérer assez d'espace pour que le programme fonctionne.

Dans le cas de notre boucle que se passera-t-il dans la pile? Elle sera tout simplement empilée et dépilée cent millions de fois. Son aspect ne sera jamais autre qu'un des états 1 à 8 imagés dans le tableau ci-dessus.

La pile ne requiert pas plus de mémoire pour cent millions d'appels de fnb() que pour un seul appel de fnb() .

Java et la mémoire inaccessible

Comme nous l'avons déjà vu, durant l'exécution d'un programme Java, vous êtes emmené à créer de objet (la plupart du temps en utilisant l'opérateur new). Ces objets, crées dans une mémoires appelée tas (heap), sont utilisés, puis au bout d'un certain temps, tombent aux oubliettes.

En effet, en Java, un objet est obligatoirement référencé par un pointeur (même si l'on ne manipule pas directement cette entité). Ce pointeur est connut soit par l'intermédiaire d'un identificateur (une variable, un paramètre d'appel de méthode, ou un attribut de classe), soit il fait parti d'une collection (un tableau ou, pourquoi pas, un objet de la classe *java.util.Vector*, qui de toutes façon encapsule un tableau élémentaire).

Mais que ce passe t'il dés lors que l'identificateur utilisé pour référencer l'objet (ou celui du conteneur) devient hors de portée ? L'exemple suivant illustre le problème.

```
public void fctQuelconque() {
    MyClass myObject =  new MyClass();
    myObject.appelMethode1();
    myObject.appelMethode2();
}
```

Jusqu'à maintenant, nous avons dit que cette zone de mémoire (l'espace occupé par les attributs de l'objet inutilisé) était simplement désallouée. C'est vrai, mais le mécanisme est en fait un petit peu plus complexe que cela. Un système relativement complexe, le garbage collector, est chargé de repérer les objets qui ne sont plus référencés, est de les supprimer de la mémoire.

Un petit problème se pose malgré tout : on risque de morceler la mémoire de telle sorte qu'elle ressemble au final à un gros gruyère. Pour palier ce problème, le garbage collector est aussi chargé de défragmenter les zones de mémoire pour laisser de vastes plages de mémoire plutôt de d'infinis petit trous. Mais d'autres circonstances peuvent aussi amener à devoir libérer l'espace mémoire utilisé par un objet. Voici quelques exemples.

```
public void fctQuelconque() {
    MyClass myObject =  new MyClass();
    myObject.appelMethode1();
    myObject =  new MyClass();
    myObject.appelMethode2();
}
```

```
public void fctQuelconque() {
    MyClass myObject =  new MyClass();
    myObject.appelMethode1();
    myObject =  null;  // L'objet n'est plus
utilisable !
    // . . .
}
```

D'autres exemples de code pourraient être mis en évidence, notamment au niveau de la manipulation des attributs de classes. Je vous laisse pensez à ces problèmes.

Le ramasse miettes

Le ramasse miettes (ou Garbage Collector en anglais) est un un système qui est chargé de libérer les espaces mémoire qui sont devenus inutilisables.

Il s'avère que le ramasse miettes ne fait pas exactement partie des spécifications du langage Java. Or toutes les implémentations de machines virtuelles actuelles en possède un. Cependant, leur fonctionnement peut différer d'une implémentation à une autre. Dans la suite de ce chapitre, nous allons étudier le fonctionnement du ramasse miettes de la machine virtuelle du JDK.

La JVM du JDK

Le problème principal du Garbage Collector est de déterminer quels sont les objets susceptible d'être supprimés de la mémoire.

Une première approche consisterait à dire qu'il suffit de compter le nombre de référence sur un objet. A chaque duplication d'un pointeur sur un objet, il suffirait d'incrémenter ce nombre. A chaque perte de référence, il faudrait alors le décrémenter. Si cette valeur tombe à zéro, l'objet peut être supprimé. Or cette approche présente un problème : si deux objets mettent en place des références circulaires, et si ces objets ne sont plus utilisables, ils ne seraient pas pour autant susceptible d'être supprimés de la mémoire.

Une approche plus sérieuse consiste à parcourir tous les objets référençable à partir des objets racines et de les marquer comme accessible. Une fois cette étapes réalisée, tous objets non accessible est donc supprimables, et ce quelques soit le nombre de références qui le pointe. Ce mécanisme est qualifié (en anglais) de "mark-sweep garbage collector" (marquer-balayer).

Mode de fonctionnement du ramasse miettes

Le ramasse miette du JDK peut de plus fonctionner selon deux modes : un mode synchrone et un mode asynchrone.

Le ramasse miettes fonctionnent en mode asynchrone dès lors que votre programme est en attente. Ainsi le ramasse miettes peut travailler sans réellement ralentir votre application, en profitant par exemple, du fait qu'il est en attente d'un évènement utilisateur sur son l'interface graphique. Dans ce mode, seul le marquage des objets supprimables est réalisé.

Mais le ramasse miettes peut aussi fonctionner selon un autre mode : le mode synchrone. Dans ce cas, il répond à une demande explicite de nettoyage de la mémoire. Cette demande peut intervenir dans deux cas particuliers. Le premier cas est déclenché dès lors que la machine virtuelle cherche à instancier un nouvel objet et que l'espace libre passe en dessous d'une certaine valeur (c'est la JVM qui invoque le GC). Le seconde cas correspond à une demande explicite, de la part du programme.

Pour invoquer explicitement le GC, il suffit d'exécuter l'ordre suivant : *System.gc()*;

Il s'avère que le mode asynchrone de fonctionnement du GC peut être inhibé. Dans ce cas, lorsque le GC est invoqué (de manière synchrone) il doit aussi préalablement marquer les objets supprimables.

Pour désactiver le mode asynchrone du GC, il faut lancer la JVM (Java Virtual Machine) dans un mode particulier. Pour ce faire, utilisez l'option -noasyncgc sur la ligne de commande.

Les finaliseurs

Les finaliseurs sont des méthodes qui sont invoquée dès lors que le GC souhaite détruire l'objet. Ainsi, si votre objet à une dernière volonté, il pourra l'exaucer. En fait, il n'y a qu'un seul finaliseur par classe. Son prototype est des plus simple : il ne prend aucun paramètres et renvoie un résultat vide (void). Son nom est ... et oui, finalize.

Ces méthodes (parfois appelée destructeurs) sont fort utiles dans les langages de programmation orientés objets : elles permettent le plus souvent, lors de la destruction de l'objet, de libérer les ressources allouées durant la construction d'un objet. Par les terme de ressources, c'est de la mémoire qui est le plus souvent sous-entendu. Or, en Java, la mémoire est automatiquement désallouée. Ces méthodes sont elles donc utiles en Java ? La réponse est simple : oui, mais pas souvent.

Ces méthodes sont nécessaire en Java. Lors de la destruction de l'objet, ce dernier peut avoir à désallouer des ressources autres que de la mémoire. Sans elles, et dans certains cas, la programmation de vos programme serait bien plus lourde. Mais il est bien clair, que les finaliseurs (ou destructeurs) sont bien moins utiles qu'en C++, par exemple.

L'exemple suivant est fort intéressant : il permet de bien mettre en évidence le fonctionnement du ramasse miettes. En effet il créer un grand nombre d'objets, rendu de suite inutiles. En conséquence, le seul limite de mémoire disponible est fréquemment atteint, ce qui a pour but de lancer le "garbage collector".

```
public class Garbage {
    String id;
```

```
    public Garbage(int i) {
        id = "" + i;
        System.out.println("Nouvel objet " + id + "
- ");
    }

    public void finalize() {
        System.out.println("Destruction de objet " +
id + " - ");
    }

    public static void main(String args[]) {
        for(int i=0;i<100000;i++) new Garbage(i);
    }
}
```

Notez un détail important : l'affichage sur la console est un mécanisme coûteux en temps d'exécution.

La gestion de la mémoire

Java s'occupe de la **gestion automatique de l'allocation dynamique de la mémoire**. Une zone mémoire est **dynamiquement allouée** à un objet lors de sa création.

```
Classe Variable_objet = new Classe();
//Variable_objet représente une référence
//de l'instance de classe new Classe()
```

*Lorsque cette zone mémoire n'est plus référencée par un objet du programme, elle est **automatiquement libérée** par le GC.*

```
Variable_objet = null;
//Variable_objet ne référence //plus d'instance de classe
```

*Plusieurs objets distincts d'un programme occupent un espace mémoire plus ou moins important sur un système informatique, en sachant que **les ressources mémoires sont certainement limitées**. Un usage incontrôlé de la mémoire risque à terme de **provoquer des conflits de ressources** tels que :*

- *une **libération intempestive d'une zone allouée** (donc utilisée par un objet),*
- *une **indisponibilité d'une zone désallouée** (donc libérée par un objet).*

Le système GC interdit l'utilisation d'une ressource déjà utilisée et évite dans la mesure du possible, les pertes de mémoires.

En outre, il existe deux types de perte de mémoire :

- ***les hard leaks (fuites mémoires graves)** sont provoquées lorsqu'il n'existe pas de système de nettoyage, et partant, aucun référencement des zones mémoires inutilisées qui n'ont pu être libérées,*

- ***les soft leaks (fuites mémoires légères)*** *surviennent lorsqu'un programme possède des zones mémoires devenues inutilisables car elles n'ont pu être libérées correctement et deviennent, ainsi, irrécupérables pour le programme.*

En ce qui concerne ***les fuites de mémoires légères*** *(soft leaks), il suffit d'****affecter la valeur null aux objets qui ne seront plus utilisés*** *par le programme. Ensuite, le système GC détruira les références et libérera les ressources pour une utilisation ultérieure.*

```
import java.io.*;
public class Execution {
  public static void main(String argv[]) {
  new Execution("fichier.bat");
  }

  public Execution(String commande) {
  try {
      String ligne;
      Process processus = Runtime.getRuntime().exec(commande);
      InputStreamReader flux_entree =
                new InputStreamReader(processus.getInputStream())
      BufferedReader entree = new BufferedReader();
      while ((ligne = entree.readLine()) != null) {
        System.out.println(ligne);
      }
    ligne = null;
      processus = null;
      flux_entree = null;
      entree = null;
  }
  catch (Exception e) {
      e.printStackTrace();
  }
  }
}
```

```
fichier.bat
echo "Bonjour à tous et à toutes !"
```

Le système de Garbage Collector agît essentiellement sur les fuites mémoires graves (hard leaks) en s'occupant de :

1. *la surveillance de l'utilisation des objets,*
2. *la détermination des objets devenus inutiles,*
3. *l'information des objets sur leur inutilisation,*
4. *la destruction des objets inutiles,*
5. *la récupération des ressources libérées.*

Le fonctionnement du *Garbage Collector*

La Machine Virtuelle Java (JVM) comptabilise en permanence le nombre de références désignant chaque objet d'un programme. Ainsi, lorsque le nombre de références est égal à zéro, cela signifie que l'objet n'est plus utilisé et par conséquent, que sa zone mémoire doit être libérée.

Le système de **Garbage Collector** accomplit ses opérations en tâche de fond à partir d'un thread indépendant de faible priorité et en exécution asynchrone.

Le GC conserve la trace de chacun des objets créés et utilisés dans un programme. Que les objets soient accessibles ou inaccessibles, le GC les prend tous en compte.

Lorsqu'**une instance de classe en mémoire ne possède plus de lien avec le programme**, c'est-à-dire, que plus aucune variable ne référence cet objet, alors ce dernier est jugé inutile. Devenu inaccessible, l'objet peut alors être soumis au traitement du GC, en l'occurrence,il appele, si elle existe, la méthode *finalize()* de l'objet en question pour accomplir un traitement de finalisation, puis s'il y a lieu opére sa destruction et récupère sa zone mémoire.

Cependant, **le processus de *Garbage Collector* est indéterminable**.
Il n'est absolument **pas garanti sur tous les objets déréférencés**, car le programme peut se terminer avant que le traitement n'ait même commencé.
En effet, étant donné la nature du mode d'exécution du système GC (thread indépendant et exécution asynchrone), il est **impossible de prévoir quand le GC s'exécutera**, et encore moins, quels objets inaccessibles il traitera.

```
public class GarbageCollector {
  ProcessusGarbage pG;
  int max;

  public static void main(String[] args) {
  int nb = 275;
  if(args.length > 0)
     nb = (new Integer(args[0])).intValue();
  GarbageCollector oGC = new GarbageCollector(n);
  oGC.executer();
  }

  public GarbageCollector(int nb) {
  max = nb;
  }

  void executer() {
  for(int i = 1; i < max; i++)
     pG = new ProcessusGarbage(i);
  }
}

class ProcessusGarbage {
  double[] tableau;
  int num_objet;

  public ProcessusGarbage(int nb) {
  num_objet = nb;
```

```
  tableau = new double[n];
  tableau[0] = 0;
  for(int i = 1; i < n; i++)
   tableau[i] = Math.pow(tableau[i - 1], 1.7E+308);
  }

  protected void finalize() {
  System.out.println("L'objet num" + num_objet
                  + " a ete traite par le Garbage Collector.");
  }
}
```

Dans cet exemple, on pourra remarquer que tous les objets créés devenus inaccessibles ne sont pas traités par le GC.

La méthode *finalize()* est automatiquement appelée lorsque le système GC a déterminé qu'un objet est devenu inaccessible. C'est pourquoi, dans l'exemple précédent, une information à propos du traitement de l'objet a pu être affichée.

La méthode *finalize()*

La méthode *finalize()* définie dans la classe *Java.lang.Object*, permet d'effectuer des opérations de finalisation en général une opération de nettoyage d'un objet avant que le système de *Garbage Collector* ne s'en charge ou que ce dernier ne peut accomplir, comme dans le cas d'un objet de système de fichier, ou encore de connexion réseau.

```
protected void finalize() throws Throwable
```

La méthode *finalize()* est **automatiquement appelée si elle existe** dans la classe de l'objet, avant que le GC n'entreprenne le recyclage de ce dernier.

Un objet déréférencé peut par l'intermédiaire de la méthode de finalisation redevenir atteignable, c'est-à-dire retrouver une référence dans le programme.

Subséquemment, **le GC vérifie à nouveau l'accessibilité de l'objet par le programme** et si celui-ci possède effectivement, une nouvelle référence, alors il **abandonne le processus de récupération des ressources** pour l'objet en cours.

La méthode *finalize()* n'est jamais appelée plus d'une fois pour un objet donné.

La méthode *System.runFinalization()* lance la méthode *finalize()* de n'importe quel objet inatteignable avant que le processus de Garbage Collector ne commence.

```
public static void runFinalization()
```

```
System.runFinalization();

Runtime.getRuntime().runFinalization();
//Equivalent à System.runFinalization();
```

L'appel de cette méthode **incite la Machine Virtuelle Java (JVM) à effectuer des tentatives d'exécution des méthodes *finalize()*** des objets qui ont été trouvés pour leur inaccessibilité, mais dont les méthodes de finalisation n'ont pas encore été lancées.

Les méthodes *gc()*

Les méthodes *Runtime.gc()* et *System.gc()* définies dans la classe *java.lang*, permettent de suggérer l'exécution du nettoyage de la mémoire par le système de Garbage Collector.

public static void **gc()**

Runtime.gc();

System.gc();

Runtime.getRuntime().gc();
// Equivalent à System.gc();

L'appel de l'une de ces méthodes **incite la Machine Virtuelle Java (JVM) à effectuer des tentatives de recyclage d'objets inutilisés** afin de récupérer leurs zones mémoires de sorte à les rendre disponible pour une réutilisation ultérieure et rapide.

La JVM exécute le processus de recyclage automatiquement si nécessaire, dans un thread séparé, **même si la méthode *gc()* n'a pas été invoquée explicitement**.

Les méthodes *Runtime.gc()* et *System.gc()* ne garantissent pas l'exécution du recyclage de la mémoire, mais seulement qu'il sera probable.

Les méthodes *totalMemory* et *freeMemory*

Deux méthodes *totalMemory* et *freeMemory* permettent de connaître la quantité de mémoire disponible sur le système informatique, où est installé la Machine Virtuelle Java (JVM).

La méthode *Runtime.totalMemory()* retourne la taille totale de la mémoire dont dispose la Machine Virtuelle Java.

public long totalMemory()

long taille = Runtime.**totalMemory()**;

La valeur de cette méthode peut varier en fonction de l'environnement hôte.

La méthode *Runtime.freeMemory()* retourne la quantité de mémoire libre du système.

public long freeMemory()

long taille = Runtime.**freeMemory()**;

L'appel de la méthode *gc()* pour le nettoyage de la mémoire, **peut entrainer une augmentation de la valeur retournée** par *freeMemory()*.

Le nombre retourné par *totalMemory()* ou *freeMemory()*, mesurant la quantité de mémoire disponible, est **exprimé en octets** (bytes). Son type est un entier long.

Quantité de mémoire disponible

Deux méthodes *totalMemory* et *freeMemory* permettent de connaître la quantité de mémoire disponible sur le système informatique, où est installé la Machine Virtuelle Java (JVM).

La méthode *Runtime.totalMemory()* retourne la taille totale de la mémoire dont dispose la Machine Virtuelle Java.

public long totalMemory()

long taille = Runtime.**totalMemory()**;

La valeur de cette méthode peut varier en fonction de l'environnement hôte.

La méthode *Runtime.freeMemory()* retourne la quantité de mémoire libre du système.

public long freeMemory()

long taille = Runtime.**freeMemory()**;

L'appel de la méthode *gc()* pour le nettoyage de la mémoire, **peut entrainer une augmentation de la valeur retournée** par *freeMemory()*.

Le nombre retourné par *totalMemory()* ou *freeMemory()*, mesurant la quantité de mémoire disponible, est **exprimé en octets** (bytes). Son type est un entier long.

FIN DES NOTES