POLYCOPIE DE JAVA

1. Introduction

Ce document doit, à terme, être un polycopié Java pour les étudiants de MIAIF2, DESS, MASS, MST2.

Plan du cours

2. Présentation générale
3. Le minimum sur les entrées sorties en Java
4. Spécificités de Java par rapport à C++
5. Classes et objets
6. Construction destruction initialisation d'objets
7. Généralisation et héritage
8. Deux classes usuelles : la classe `Object` et la classe `Vector`.
9. Interfaces et paquetages
10. Exceptions
11. Multi-tâches
12. Fichiers
13. Les classes et les types prédéfinis
14. L'environnement de développement

Références

Gosling & Arnold : «le langage Java», traduction Miniussi & Chaumette, International Thomson Publishing.
Livre clair et très adapté pour les débutants, écrit par le créateur du langage.

Niemeyer & Peck : « JAVA par la pratique », Editions O'Reilly.
Livre qui couvre l'ensemble des aspects de Java par des exemples.

Irène Charon : « le langage Java, concepts et pratique », Hermès.
(http://www.infres.enst.fr/~charon/coursJava/)
Cours en ligne avec exercices corrigés. Très bien.

Jérôme Bourgeault : « Java : la maîtrise », Les Guides de formation Tsoft, Eyrolles.
Livre exhaustif avec AWT, JDBC, JavaBeans, JFC, Swing, etc.

« Java Corner » : Un service d'information sur le langage JAVA réalisé par des membres du Centre Universitaire d'Informatique (CUI) de l'Université de Genève.
(http://cuiwww.unige.ch/java/)

enfin, quelques liens vers des pages java :
(http://www.math-info.univ-paris5.fr/~bouzy/java.html)

2. Présentation générale

Java a été conçu par James Gosling en 1994 chez Sun. L'idée était d'avoir un langage de développement simple, portable, orienté objet, interprété. Java reprend la syntaxe de C++ en le simplifiant. Java offre aussi un ensemble de classes pour développer des applications de types très variés (réseau, interface graphique, multi-tâches, etc.)

Le but de ces notes de cours est de donner un premier aperçu sur Java en ne présentant que ce qu'il est nécessaire de connaître pour *bien programmer objet avec Java*. Java comprend bien d'autres aspects (programmation graphique, applets, programmation réseau, multi-tâches) que ce document ne prend pas en compte. Ce document suppose connu le langage C, mais pas C++. Enfin, mieux vaut avoir des rudiments sur l'orienté objet sans que cela soit vraiment bloquant pour le lire.

3. Le minimum vital sur les entrées sorties

Il n'est pas trivial d'écrire le programme « BONJOUR LE MONDE » en Java. Le but de ce paragraphe est de montrer comment écrire ce programme et de présenter une classe Entree qui encapsule les fonctions d'entrée en Java. Ceci afin d'être capable d'amorcer l'apprentissage de la programmation en Java.

Ecrire sur la sortie standard

Le premier programme Java à écrire est le suivant :

```
class Machin {
    public static void main(String argv[])
    {
          System.out.println (« Bonjour le monde ! ») ;
    }
}
```

Ainsi, la sortie de ce premier programme Java est :

```
Bonjour le monde !
```

Pour les personnes patientes et pas curieuse, passez au paragraphe suivant, vous n'avez pas besoin d'en savoir plus et le reste sera expliqué plus tard.

Pour les curieux impatients :
« class Machin » est la déclaration d'une classe (obligatoire en Java).
« public » signifie que main est visible de l'extérieure de la classe machin.
« static » signifie que main n'est pas associée a une instance mais à une classe…
« main » est le nom de la méthode…
« argv » est le tableau de chaînes de caractères habituels du C.
« String » est le nom de la classe chaîne de caractères
« System » est le nom de la classe système
« out » est le nom de l'instance pour effectuer des sorties
« println » est le nom de la méthode qui imprime une ligne avec un retour chariot.

Lire sur l'entrée standard

On peut également lire des informations sur l'entrée standard du programme. Pour cela, les primitives Java sont dans la classe System.in. Mais elles sont d'un niveau trop bas et un peu compliqué. Nous présentons donc la classe Keyboard (ne faisant donc pas partie de Java) qui permettra de faire des entrées avec des fonctions plus simples.

La classe Keyboard.

```
// keyboard.java
import java.io.*;
public class Keyboard
{
    public static char getChar() ;
    public static String getString() ;
    public static int getInt() ;
    public static void pause() ;
}
```

La méthode getString() retourne la chaîne de caractères lue sur l'entrée standard. Elle retourne tous les caractères tapés jusqu'au retour chariot. La méthode getInt() est analogue à getString, mais convertit la chaîne lue sur l'entrée standard en un entier qu'elle retourne. La méthode pause() attend un retour chariot sur l'entrée standard. Elle est utile pour mettre un point d'arrêt dans le programme. On peut ainsi éviter que la fenêtre d'exécution du programme s'en aille trop vite en plaçant un pause() à la fin du programme. NB : ces méthodes sont minimales et ne sont réutilisables que pour débuter la programmation des entrées-sorties en Java. Elles devront être améliorées ensuite selon les besoins du programmeur.

4. Spécificités de Java par rapport à C++

Ce paragraphe rassemble les spécificités de Java par rapport à C++. Il est donc à sauter si vous ne connaissez pas C++.

- Pas de pointeurs, que des références…
- Interprété et dynamique
- Portable
- Multi-tache

Pas de pointeurs

Java fait une partie de sa publicité auprès des programmeurs (débutants) en clamant qu'en Java il n'y a pas de pointeurs mais que des références et en sous-entendant donc que Java simplifie la programmation. Nous pensons qu'il est nécessaire de faire un paragraphe expliquant que ce n'est pas vrai. Pour cela, ce paragraphe rappelle d'abord ce que sont des pointeurs et des références en C++, puis il explique pourquoi programmer avec des références Java est aussi difficile que de programmer avec des pointeurs en C++.

Rappel sur les pointeurs et références

Ce paragraphe rappelle le strict minimum sur ce que sont un pointeur ou une référence. En C, lorsque l'on déclare une variable avec un nom et un type, un emplacement mémoire du type de la variable est créé à une certaine adresse avec son nom pour y accéder. L'emplacement mémoire recevra la valeur de la variable lors d'une affectation.

```
int x ;          // une déclaration en C
x = 3 ;          // une affectation en C
```

Le C permet de manipuler dans le programme, les adresses des emplacements mémoire des variables. &x désigne l'adresse de la variable x. On peut déclarer des pointeurs d'un type qui sont des variables contenant des adresses de variables de ce type avec le symbole *.

```
int * p ;        // un pointeur sur un entier
p = &x ;         // p vaut l'adresse de x
```

L'opérateur * s'appelle l'opérateur de déréférencement et *p désigne la valeur contenue dans l'emplacement mémoire dont l'adresse et p.

```
int y = *p ;     // y vaut 3
```

On peut aussi déclarer une référence à une variable existante. Une référence se déclare avec l'opérateur &. Une référence est un synonyme - un alias - pour désigner l'emplacement mémoire d'une variable.

```
int & z = x ;   // z est une référence a x, z vaut 3
```

Si on change la valeur de x, la valeur de z est aussi changée et inversement. Idem avec *p.

```
x = 4;     // x et z valent 4, *p aussi
z = 5;     // x et z valent 5, *p aussi
*p = 6;    // *p, x et z valent 6
```

Noter que les opérateurs & et * sont inverses l'un de l'autre. On a toujours :

```
*(&x) = x        et    &(*p) = p
```

Attention, on ne peut pas déréférencer un pointeur qui ne contient pas une adresse valide.

```
int * q ;
*q = 7 ;  // plantage
```

Il faut initialiser le pointeur avant :

```
int * q = &x;
*q = 7 ;  // ok
```

La manipulation des pointeurs en C ou en C++ est réputée difficile et source d'erreurs fatales pour l'exécution du programme.

Références Java ou pointeurs C++, relier des objets restera toujours une tâche délicate !

Un pointeur C++ est une sorte de référence au sens large. La manipulation des références Java est analogue à celle des pointeurs C++ et elle est tout aussi difficile. Dans les deux cas, on a un synonyme d'une variable et on cherche la variable ; dans les deux cas, pointeur C++ ou référence Java, le problème du programmeur est d'avoir un synonyme valide pour chercher la variable ; si le synonyme est invalide (un pointeur C++ qui contient une adresse d'un emplacement mémoire ne correspondant pas à la variable recherchée ou une référence Java ne correspondant pas à la variable recherchée) l'exécution du programme sera incorrecte, que l'on soit en C++ ou en Java.

Donc, attention ! Le fait qu'en Java, les pointeurs aient disparu, ne simplifie en rien la tâche du programmeur souhaitant manipuler des liens inter-objet. Il faudra toujours mettre à jour ces liens et cela sera d'une difficulté égale de le faire en Java avec des références ou avec des pointeurs en C++ !

Interprété et dynamique

Exemple:

Alice écrit un « programme » qui ajoute 4 à une donnée.
Alice écrit dans le langage L.
Programme écrit en langage L : A = B + 4
Elle sait que l'entrée sera représentée par B et que le résultat sera représenté par A.

Compilation et exécution

Une compilation consiste à transformer un programme écrit dans un langage L en un programme écrit dans un langage M, exécutable après coup par une machine qui sait interpréter ou exécuter le langage M.

Langage M
registre R = (B)
registre S = 4
registre R = registre R + registre S
(A) = registre R

La compilation consiste à transformer le premier programme en le second. Pour chaque instruction écrite en langage L, le compilateur engendre une ou plusieurs instructions écrites en langage M. Le compilateur prépare le travail. L'exécution consiste à faire lire les instructions du second programme écrit en langage M.

Interprétation

Une interprétation consiste à exécuter directement un code écrit dans un langage L sans le transformer avant de l'exécuter. Sur la machine, il existe un programme nommé interpréteur I de programmes écrits en langage L.

A l'exécution, l'interpréteur lit le programme de Alice, par exemple de gauche à droite.
Il lit A, il associe A au registre R.
Il lit =, il sait qu'il va affecter un valeur à R.
Il lit B, il associe B au registre S.
Il lit +, il sait qu'il va additionner quelquechose à B.
Il lit 4, il ajoute 4 au registre S.
Il affecte la valeur de S à R.
Il écrit le résultat de A qui est contenu dans R.

L'interpréteur fait donc 2 choses en même temps : il transforme les instructions écrites en langage L en instruction écrites en langage M et il les exécute.

Compilation versus interprétation

Si on dispose un programme écrit en langage L et d'une machine qui exécute du langage M, il faut soit :

- avoir un compilateur qui transforme le programme du langage L vers le langage M. Par exemple, si on a un programme C++, et 3 machines : un PC, un Mac et une Sun il faudra 3 compilateurs.
- avoir un interpréteur qui comprend le programme en langage L. Par exemple, si on a un programme C++, et 3 machines : un PC, un Mac et une Sun il faudra 3 interpréteurs différents.

L'interprétation est une opération longue car elle fait les 2 choses simultanément :

- transformer les instructions
- et les exécuter.

Un programme compilé tourne plus vite qu'un programme interprété

La compilation est une opération longue car elle transforme les instructions. Mais un programme, une fois compilé, tourne plus vite que le même programme interprété car la transformation est déjà faite par la compilation. Seule l'exécution des instructions intervient lorsque le programme compilé tourne.

Un programme compilé est plus long à mettre au point qu'un programme interprété

Par contre, il est plus rapide de programmer avec un langage interprété qu'avec un langage compilé. En effet, étant donné un programme à mettre au point, le cycle (éditer, engendrer l'application, se mettre dans les conditions de test, exécuter) est plus rapide si l'application est interprétée. En effet, avec un interpréteur, votre application donnant un erreur dans une certaine condition, vous voulez faire disparaître l'erreur : vous devez modifier votre programme, recharger la modification, remonter en arrière dans votre exécution pour vous mettre dans les conditions du test et exécuter. Avec un langage

interprété, vous n'avez pas besoin de relancer votre programme depuis le début. Par contre, avec un compilateur, si vous voulez faire disparaître une erreur : vous devez modifier votre programme, recompiler votre application, redémarrer votre programme au début, exécuter pour vous mettre dans les conditions du test et exécuter.

Java est un langage semi-interprété et semi-compilé

Lors de sa mise au point, un programme écrit en langage java est d'abord compilé en un « bytecode » java. Lors de son exécution sur une machine, un programme « bytecode » java est interprété par l'interpréteur java de la machine. Dans ce sens, on dit que java est semi-interprété (l'interpréteur java n'interprète pas du langage java mais du bytecode). On peut dire aussi qu'il est semi-compilé au sens où le compilateur java transforme le source java en un programme écrit en bytecode et non pas en code machine.

Java, langage interprété, offre l'avantage d'être plus facile à programmer que s'il était totalement compilé.

Par contre, Java est lent comparé à C++ (~30 fois plus lent) car il est interprété. Pour résoudre ce problème, on peut utiliser un compilateur « natif », c'est à dire un compilateur qui transforme un source java en un exécutable machine. Mais là, on perd les avantages qui font la publicité de Java.

Portable

On dit que Java est « portable ». Ce paragraphe explique dans quelle mesure cela est vrai. Quand on écrit un programme on peut définir sa portabilité de deux manière : portabilité « de langage » ou portabilité de « code compilé ».

Portabilité de source

Alice écrit un programme en langage L. Les fichiers contenant le programme en langage L s'appelle le source du programme. Alice peut par exemple compiler le source pour avoir un exécutable. Puis elle exécute le programme sur sa machine. Ensuite imaginons qu'elle donne le source à Bob qui dispose d'une machine de type différent. Si Bob possède un compilateur de programmes écrits en langage L, il peut compiler le source et produire un exécutable pour sa machine. Si Bob n'a pas besoin de modifier le source du programme, on dit que le source du programme est portable. Il s'agit de la portabilité du source.

Portabilité d'exécutable.

Alice à écrit un programme en langage L, compilé en langage M sur une machine. Elle donne l'exécutable à Bob. Si Bob possède une machine sachant interpréter le langage M, l'exécutable est portable. Il s'agit de la portabilité d'exécutable. Cette technique est intéressante si Alice ne veut pas donner les sources de son programmes à Bob et seulement un exécutable.

Lorsque Alice commercialise son logiciel, la portabilité d'exécutable n'est bien sûr pas obligatoire mais intéressante. Si Alice veut commercialiser un logiciel écrit dans un langage sans portabilité d'exécutable, elle doit engendrer autant d'exécutables différents qu'il y a de types de machine différents avec des compilateurs différents. C'est le cas actuel de l'industrie informatique avec des langages comme C ou C++.

Lorsque Alice se lance dans la programmation distribuée sur plusieurs machines de types différents sur un réseau, un programme P qui s'exécute sur une machine X peut avoir besoin à « run-time » de récupérer un programme Q situé sur une autre machine Y et de le lancer sur X. Si il fallait récupérer le source de Q et recompiler le programme Q avant de le lancer, le programme P serait trop lent. Il devient alors nécessaire de récupérer Q sous forme exécutable. Dans le cadre de la programmation distribuée, la portabilité d'exécutable devient donc très utile, voire obligatoire.

Dans ce cadre, Java permet à priori la portabilité de langage et de code compilé.

Multi-tâches

Avec la classe Thread, Java offre au programmeur la possibilité de programmer en multi-taches. Ce qui est un gros avantage, surtout pour développer des applications réelles avec de multiples interfaces (homme-machine, réseau, base de données, etc.) avec le monde extérieur. (Cf paragraphe plus loin dans ce document.)

5. Classes et objets

Ce paragraphe aborde les caractéristiques de Java vis-à-vis de la programmation orientée objet. Il rappelle d'abord le vocabulaire objet, notamment ce que signifie encapsulation, puis il montre comment déclarer une classe Java, comment définir le corps des méthodes, la distinction entre privé et publique, les constructeurs et enfin les attributs statiques d'une classe.

Vocabulaire objet

Ce paragraphe rappelle le vocabulaire employé en orienté objet. Et il donne la correspondance entre le vocabulaire objet et le vocabulaire spécifique de Java.

Objet

Un objet est une entité, qui possède un état et un comportement. Pour définir ce qu'est un objet, les livres donnent souvent une équation du style de :

objet = état + comportement

Classe

Vue de la programmation objet, une classe est un type structuré de données. Nous verrons qu'une classe Java est le prolongement des structures C (mot-clé 'struct').Vue de la modélisation objet, une classe correspond à un concept du domaine modélisé. Une classe regroupe des objets qui ont des propriétés et des comportements communs. Une classe est donc un ensemble d'objets. En Java, pour une classe, on dit aussi une classe.

Instance

Pour désigner un objet de la classe, on dit aussi une « instance ». « instance » est un anglicisme qui possède une signification proche de celle de « exemple » en français. On dit souvent qu'une instance « instancie » une classe. Cela signifie que l'instance est un exemple de la classe. En Java, pour désigner une instance on dit indifféremment un objet ou une instance.

Attribut

L'état d'un objet est l'ensemble des valeurs de ses attributs. Un attribut est une propriété de l'objet. En Java, on dit aussi attribut.

Méthode

Vue de la modélisation objet, une méthode est une opération que l'on peut effectuer sur un objet. Vue de la programmation objet, une méthode est une fonction qui s'applique sur une instance de la classe. En Java on dit aussi méthode.

Message

A l'origine de la programmation orientée objet, l'objectif était que les objets d'un programme aient des comportements concurrents (comme des tâches dans une environnement multi-tâche ou des process en Unix). On voulait qu'ils « communiquent

» en « envoyant » et « recevant » des « messages ». L'objectif fixé était louable mais malheureusement trop ambitieux pour l'époque.

Actuellement, tous les langages objet implémentent les objets de manière fonctionnelle (ou procédurale). C'est-à-dire qu'au lieu d'envoyer un message à un objet considéré comme une tâche ou un processus, on appelle une méthode, c'est-à-dire une fonction tout simplement... Donc, il faut être bien conscient de la correspondance suivante :

envoyer un message à un objet	=	appeler une méthode associée à l'objet
recevoir un message	=	entrer dans le corps de la méthode appelée

Encapsulation des données

La POO pure suit le principe d' « encapsulation » au sens strict : tout attribut d'une classe est caché dans la classe ; il est invisible de l'extérieur de la classe, il n'est accédé depuis l'extérieur de la classe que par une méthode de lecture et une méthode d'écriture situées dans l'interface de la classe, c'est-à-dire visibles de l'extérieur. Le gros avantage de cette approche est d'assurer qu'un attribut n'est modifié que par la méthode d'écriture. Si par exemple, un attribut est une année qui doit être compris entre 0 et 99, la méthode d'écriture contient le test de vérification avant la modification de l'année. Le test est fait à un seul endroit du programme. De plus, avec cette technique, si un jour les spécifications de l'accès à l'attribut changent (imaginons par exemple que le nombre doit être supérieur à 0), le programmeur n'aura à modifier que la méthode d'accès à l'attribut (en y enlevant le test d'infériorité à 99). Sans cela, il aurait été obligé de changer le programme à tous les endroits où celui-ci modifie cet attribut. D'un point de vue de génie logiciel, cette technique est donc intelligente. Un autre avantage de cette approche existe dans le contexte de la programmation concurrente : afin d'assurer qu'un attribut ne soit pas modifié par deux processus concurrents en même temps ou bien afin d'assurer qu'un processus ne lise pas un attribut alors que celui-ci est en cours de modification par un autre processus, on protège l'accès aux attributs par ces méthodes d'accès.

Pour répondre au principe d'encapsulation, Java, comme C++, prévoit les mot-clés 'private' et 'public' pour dire si un attribut est visible ou non d'un programme utilisateur de la classe. En ce sens, il permet soit de suivre le principe au sens strict, soit de n'encapsuler que certains attributs ou méthodes d'une classe ; il permet de ne pas suivre le principe d'encapsulation au sens strict. C'est une bonne chose car de nombreuses applications ne nécessitent pas d'encapsuler toutes les données. D'autres langages objet, au contraire de Java, obligent le programmeur à suivre ce principe (Smalltalk par exemple).

En ce qui concerne la pédagogie de la POO, il existe deux attitudes : obliger les débutants à suivre ce principe à la lettre, (n'importe quel livre de POO commence par ça malheureusement) ou bien ne pas le faire. Si on oblige le suivi de ce principe, dés qu'un débutant déclare un attribut x de type T dans une classe X, il le met en privé et il écrit obligatoirement deux méthodes d'accès publiques : la méthode de lecture `T getX(){ return x ;}` et la méthode d'écriture `setX(T x0) { x = x0 ; }`. C'est dramatique d'un point de vue pédagogique. Primo, il faut bien être d'accord avec les débutants qui constatent immédiatement que ces méthodes « ne font rien ». Secondo, alors que le but de la programmation orientée objet est d'écrire des programmes structurés et concis, le débutant se trouve entravé par l'écriture de 2 méthodes à chaque fois qu'il veut déclarer

1 attribut. Sa tâche de programmation est multipliée par 3 et encombrée de méthodes inutiles. Pédagogiquement, d'autres concepts de la conception OO sont fondamentaux et plus importants que l'encapsulation des données : par exemple, le placement adéquat des attributs et méthodes, un, dans les bonnes classes (1er obstacle important, pourtant aucun livre de POO n'en parle) et, deux, dans le bon type « de classe » ou « d'instance », statique ou non statique (cf. plus loin) (2ème obstacle important). Bref, si vous êtes débutants, si vous ne faites pas de programmation concurrente, si vous ne développez pas de gros programmes qui évolueront, ne suivez pas forcément le principe d'encapsulation.

Déclaration et définition de classe

En Java, la déclaration et la définition de la classe sont faites en une seule fois (contrairement à C++ où l'on déclare la classe dans un .h et où l'on définit la classe dans un .cpp). Pour déclarer une classe Bidule en Java, on utilise le mot-clé 'class' :

```
class Bidule { ...}
```

Le mot-clé `class` de Java est identique a celui de C++ et la syntaxe est quasi-identique : la seule différence est qu'il n'y a pas de ; après }.

Exemple :

```
class Point {
   private int x ;                    // un attribut
   private int y ;                    // un autre attribut
   public Point(int a, int b)         // constructeur
   {
         x = a;
         y = b;
   }
   public  void deplace(int dx, int dy)     // une méthode
   {
         x += dx;
   y += dy;
   }
   public  void affiche()             // encore une méthode
         System.out.println (« x = «  + x );
         System.out.println (« y = «  + y );
   }
}
```

x et y sont des attributs de la classe Point. Point est le « constructeur ». deplace() et affiche() sont des méthodes de la classe Point. Noter que le corps de la méthode est placé juste après la déclaration de la méthode. Voici un exemple d'utilisation de la classe Point par un programme :

```
public static void main(String argv[]) {
   Point a = new Point(5,2) ;
   a.affiche() ;
   a.deplace(8, 4) ;
   a.affiche() ;
   Point b = new Point(-1, 1) ;
   b.affiche() ;
}
```

Pour appeler la méthode déplace sur l'objet référencé par 'a', noter qu'il faut écrire le nom de l'objet (a), un point (.) et le nom de la méthode (deplace).

Attributs privés ou publiques

Pour l'exemple précédent, une instruction du style de a.x = 3 ; dans le main() serait rejetée à la compilation car x est un attribut privé de la classe Point.

NB : Si x était un attribut publique de la classe Point on pourrait le faire. Il faut noter que la syntaxe est la même que pour les méthodes : on écrit le nom de l'objet (a), un point (.) et le nom de l'attribut (x).

La déclaration d'une classe est toujours du type :

```
class Toto {
private ...          // des attributs ou méthodes prives
public ...           // des attributs ou méthodes publiques
   ...               // etc.
}
```

On doit mettre un mot-clé « public » ou « private » pour chaque attribut ou méthode.
Le principe d'encapsulation met les attributs en privés avec, pour chaque attribut, une méthode de lecture et une méthode d'écriture.

Construction et destruction d'objets

Une caractéristique de la POO est de maîtriser la création d'objet, leur initialisation et leur destruction. Java répond en partie à ce besoin avec la notion de constructeur d'objet pour la création et l'initialisation. Mais, il n'y a pas destructeur en Java. Ce paragraphe traite donc les constructeurs et les destructeurs en général en POO. Enfin une discussion suit sur l'absence de destructeur obligatoire en Java.

Constructeur

Un constructeur d'objet est une méthode particulière d'une classe qui est appelée lors de la création d'un objet, quel qu'il soit. Son but est :

(a) d'allouer un emplacement mémoire pour l'objet,
(b) d'initialiser les attributs de l'objet avec de bonnes valeurs de départ et
(c) de retourner l'adresse de l'emplacement mémoire choisi.

A partir du moment où un constructeur existe, il n'est pas possible de créer un objet sans passer par lui et donc on doit fournir les arguments requis. On est certain que l'objet est créé avec les bons attributs de départ.

Destructeur

Il n'y a pas de destructeur en Java. Cependant, un destructeur d'objet est une méthode très commune en POO qui :

(d) remet l'objet dans un état terminal et
(e) libère l'emplacement mémoire associé à l'objet.

L'opération (d) est fondamentale selon nous pour une raison de justesse et de correction du programme. Lorsque l'utilisateur d'un programme « détruit » une instance du programme, il ne doit pas voir cette instance ensuite. Normal !

Prenons l'exemple du responsable du « garage » qui veut détruire un « mécanicien » qui vient de démissionner du garage. Il effectue l'opération « mécanicien détruire » au travers de l'interface homme-machine puis il affiche la liste des mécaniciens pour vérifier que celui-ci a effectivement disparu de son logiciel.

Pour que le programme soit juste et correct, l'opération « mécanicien détruire » doit se traduire par la suppression de la référence de l'instance correspondant au mécanicien de la liste des mécaniciens du garage. Si le mécanicien était en relation avec d'autres objets, le programme doit supprimer les références vers le mécanicien avant de le détruire. Toutes les lignes de codes correspondantes à la mise à jour de références se trouvent nécessairement à un endroit dans le programme. Nous conseillons de les mettre dans une méthode de la classe Mécanicien (que nous pouvons appeler d(), demission(), depart() ou pourquoi pas détruire() ?) .

Lorsqu'un langage prévoit un destructeur d'objet, on a coutume de mettre la méthode d() dans le destructeur. Dans ce cas, le langage libère l'emplacement mémoire de l'objet (e) juste après la remise de l'objet et de son voisinage dans l'état initial.

« Ramasse-miettes » ou « garbage collector »

Java n'a pas de destructeur d'objet. Java possède un « rammasse-miettes » qui libère les emplacements mémoire occupés par les objets non référencés. Le rammasse-miettes fait donc l'opération (e). Le programmeur doit faire (d) afin que l'objet soit non référencé. Nous préférons programmer avec (d) explicitement dans une méthode que nous appelons « destructeur » et nous faisons (e) juste après. Nous mettons (d) et (e) dans un destructeur et dans ce cas, il n'y a pas besoin de rammasse-miettes. Java fournit tout de même la méthode 'finalize()' pour faire (e) de manière contrôlée. Si le programme n'a pas de problème d'utilisation de la mémoire, il est vrai (e) n'a pas besoin d'être effectuée par le programme et l'utilisation du rammasse-miettes est pleinement justifiée.

Les fichiers .java

En Java, on ne sépare pas obligatoirement la déclaration et la définition des classes comme en C++ (*.h et *.cpp). On met le tout dans un fichier *.java. Pour séparer déclarations et définitions, on peut utiliser les interfaces. Un fichier *.java contenant des classes X, Y et Z compilé produit les fichiers X.class, Y.class et Z.class qui sont les « bytecodes » des classes X, Y et Z.

Attributs ou méthodes statiques

L'orienté objet définit deux types de propriétés dans une classe : les propriétés associées à une instance de la classe (les propriétés dites « d'instance ») et les propriétés associées à une classe (les méthodes dites « de classe »).

Avec le qualificatif '**static'** avant un attribut ou méthode d'une classe le programmeur peut spécifier que celui-ci est « de classe ». On dit aussi statique. Une propriété sans le mot-clé static est une propriété « d'instance ».

```
class Exemple {
   private static int a = 3;
```

```
    private float b ;
    public void affiche() { ... }
    public static void affiche_tout() { ... }
    public Exemple(float x) { b = x ; }
    ...
    public static void main (String argv[]) {
          Exemple e = new Exemple(1) ;
          Exemple f = new Exemple(2) ;
    }
 }
```

Dans la classe Exemple, a est un attribut statique, b un attribut non statique.
e et f sont des objets avec e.b = 1 et f.b = 2
e et f ont l'attribut 'a' en commun e.a = f.a = 3.

La méthode affiche_tout de la classe Exemple est « de classe » ou statique et la méthode affiche est « d'instance ».

Bien qu'il soit possible de le faire au début du programme principal, il est possible et plus cohérent d'initialiser les attributs statiques d'une classes avec un bloc statique dans la déclaration de la classe. Cela peut être pratique pour initialiser un tableau statique.

```
class ExempleT {
    public static int a[2];
    static {
          for (int i=0 ; i<2 ; i++) a[i]=3 ;
    }
}
```

Autoréférence: le mot-clé *this*

Quand le programme est dans une méthode d'instance, le programmeur peut manipuler La référence de l'instance courante avec le mot-clef '**this'**.

```
 class Point { …
    public void affiche() { System.out.println(“ adresse = ” + this); }
 }
```

Dans un constructeur, `this` est utile pour mettre la référence de l'objet construit dans un tableau ou dans une liste d'instances ou dans un attribut d'un autre objet.

```
class MaClasse {
    public static Vector mesInstances = new Vector();
    ...
    public MaClasse(. . .) {
          . . .
          MesInstances.addElement(this) ;
    }
    . . .
 }
```

Pour tenir à jour une référence inter-objet, `this` est utile. Par exemple :

```
class Dessous { public Dessus monDessus; ... }
class Dessus {
    public Dessous monDessous;
    ...
    public void poserInstanceSurDessous(Dessous d) {
          monDessous = d ;
          d.monDessus = this ;
```

```
    }
}
```

`this` est un paramètre implicite des méthodes d'instance.

`this` n'a évidemment pas de sens dans une méthode de classe.

Appel des méthodes

L'appel d'une méthode peut se faire de plusieurs manières selon que la méthode est une méthode d'instance ou bien de classe. Les règles sont les suivantes.

Pour une méthode de classe, le critère important est le *lieu de l'appel*:
- Appel dans la classe ou dans une classe descendante : une méthode statique s'appelle avec son nom seul (cas A).
- sinon, une méthode statique s'appelle avec son nom préfixé du nom de la classe. (cas B).

Pour une méthode d'instance, le critère important est *l'instance* sur laquelle on appelle la méthode:
- Avec l'instance courante, une méthode d'instance sur l'instance courante s'appelle avec son nom seul (cas C) [ou bien, mais c'est redondant, préfixé de this. (cas D)].
- Avec une autre instance que l'instance courante une méthode d'instance s'appelle avec son nom au sens strict préfixé du nom de l'instance et d'un point (.) (cas E).

```
class Truc { …
    public void a()
    {
          d() ;                    //  cas A
          b() ;                    //  cas C
          this.b() ;               //  cas D
          Truc t ;
          …
          t.b() ;                  //  cas E
    }
    public void b() {  ... }
    public static void c()
    {
          d() ;                    // cas A
          Truc t ;
          …
          t.b() ;                  // cas E
    }
    public static void d() { ... }
}
class Test { …
    public static void main(String argv[]) {
          Truc.d() ;               // cas B
          Truc t ; t.b() ;         // cas E
    }
}
```

Bien sur, on peut être redondant :

```
void a() {
    Truc.d() ;                     //  cas A
    Truc.b() ;                     //  cas C
    Truc t ; t.Truc.b() ;          //  cas E
}
```

6. Construction, destruction et initialisation d'objets

Ce paragraphe montre les différents comportements des objets Java lors de leur durée de vie. Il précise les qualificatifs des objets suivant leur type de déclaration, comment ils sont initialisés et surtout le cas du constructeur par recopie.

Les différents qualificatifs des objets suivant leur type de déclaration

Les références

Les références sont des variables contenant un nom d'objet.

```
void f() {
   Truc t ;          // t est une référence sur un Truc,
   …                 // t n'est pas un objet
   {
   Bidule b ;        // b est une référence sur un Bidule, pas un objet
   …
   }                 // b n'est plus visible après cette ligne
}                    // t n'est plus visible après cette ligne
```

t est visible dans la fonction f et b est dans le bloc.

null est la référence nulle ou 0.

les objets statiques

Un objet statique est un objet déclaré avec le mot-clé static dans une déclaration de classe ou dans une fonction ou bien à l'extérieur de toute fonction.

Exemple :

```
static Point a = new Point(1, 7);
```

les objets dynamiques

Ce sont eux qui font tout l'intérêt de la programmation orienté objet.
Un objet dynamique est un objet créé avec « new » dans le corps d'une méthode.

```
class Test {
      public static void main (String args []) {
            System.out.println("&& debut main");
            Point adr = new Point(3, 7);
            fct(adr);
            System.out.println("&& fin main");
      }
      public static void fct (Point adp) {
            System.out.println("&& debut fct");
            adp.detruire();
            System.out.println("&& fin fct");
      }
}
class Point {
      public Point (int x, int y) {
            System.out.println("++ appel constructeur");
      }
      public detruire() { System.out.println("-- appel destructeur"); }
}
```

Initialisation d'un objet lors de sa déclaration

Avec l'utilisation systématique des constructeurs Java ne permet pas de créer un objet sans l'initialiser en même temps. Pour créer un objet on doit appeler un constructeur déclaré. Une classe peut contenir plusieurs constructeurs, c'est même conseillé. Avec :

```
class Point {
    public int x ;
    public int y ;
    public Point(int abs) { x = abs ; y = 0 ; }
    public Point(Point p) { x = p.x ; y = p.y ; }
          ...
 }
```

le programmeur peut écrire :

```
Point a = new Point(3);
Point b = new Point(a) ;
```

mais pas :

```
Point c = new Point(3, b) ;
Point d = new Point(3, 4) ;
```

Constructeur par recopie, référence vers un objet, valeur d'un objet

Ce paragraphe traite du cas où le constructeur d'une classe possède un unique paramètre qui est une référence à un objet de la classe. On appelle alors ce constructeur, le *constructeur par recopie*. Ce cas est très utilisé car bien souvent, dans un programme, on veut dupliquer un objet (afin, par exemple, de travailler sur sa copie et de garder une sauvegarde).

Le constructeur par recopie peut être déclaré de la manière suivante :

```
public Point (Point);
```

Pour dupliquer l'objet a dans un objet b, le programmeur écrira simplement :

```
Point b = new Point(a);
```

Le programme suivant montre la différence qui existe entre une référence et une valeur d'objet :

```
Point x;                        // x est une référence non initialisée
Point y = new Point(1, 2);      // y est une référence vers un objet
Point z = y;                    // z est une référence
                                // vers l'objet référencé par y
Point w = new Point(y);         // w référence un objet dont les
                                // valeurs sont identiques aux
                                // valeurs de l'objet référencé par y
if (z==y) …       // true car z et y référencent le même objet
if (w==y) …       // false car w et y référencent des objets différents
if (z.equals(y)) …// true car les valeurs de l'objet
                  // référencé par z égalent les valeurs de l'objet
                  // référencé par y (c'est le même objet)
if (w.equals(y)) …// true car les valeurs de l'objet référencé
                  // par w égalent les valeurs de l'objet
                  // référencé par y.
```

Plutot que d'utiliser le constructeur par recopie, on peut préférer utiliser et redéfinir (cf § sur l'héritage) la méthode `clone` de la class Object :

```
Point b = a.clone();
```

7. Généralisation et héritage

Les concepts de généralisation et d'héritage sont fondamentaux en Java (et en POO en général). Ils permettent d'écrire des programmes simplement et ils permettent la réutilisation de programmes.

1. 1er aperçu de la généralisation et de l'héritage en Java

Si on la classe `Point` définie par :

```
class Point {
   public int x;
   public int y;
   public Point(int a, int b) { x = a; y = b; }
   public void deplace(int dx, int dy) { x += dx; y += dy; }
   public void affiche() { System.out.println( "x = " + x + "y = " + y) ; }
}
```

Il est possible définir une sous-classe `PointCol` de la classe `Point` grâce au mot-clé 'extends' :

```
class PointCol extends Point {
   public int couleur;
   public PointCol(int a, int b, int c) { super(a, b) ; colore(c); }
   public void colore(int c) { couleur = c }
}
```

Dans cet exemple, on a défini une « sous-classe » (`PointCol`) à partir d'une « super-classe » (`Point`). Cela signifie que la classe `PointCol` est une extension de la classe `Point`. La terminologie des super-classes et sous-classes est identique à celle de UML.

On dit que la classe `PointCol` « hérite » de la classe `Point`. Cela signifie que les attributs et méthodes de la classe `Point` sont implicitement des attributs ou méthodes de la classe `PointCol`.

```
PointCol pc = new PointCol(1, 2, 3) ;
pc.deplace(4, 5) ;                  // correct car 'deplace' est héritée
pc.colore(6) ;                      // correct comme d'hab
Point p = new PointCol(7, 8) ;
p.colore(9) ;     // ERREUR ! l'héritage ne marche pas dans ce sens !
```

2. Redéfinition de méthodes et de attributs

Une méthode de la sous-classe peut avoir la même signature qu'une fonction de la super-classe. On dit alors que la méthode de la super-classe est *REdéfinie* dans la sous-classe. Quand on est dans la sous-classe, on appelle la méthode de la sous-classe de façon habituelle. Si on veut appeler la méthode de la super-classe, il faut préciser le nom de la super-classe avec le mot-clé 'super' en préfixe.

Exemple:

```
class PointCol extends Point { ...
   public void affiche() {
         super.affiche();
         System.out.println("couleur " + couleur);
   }
}
```

```
...
 public static void main(String argv[]) {
    PointCol p = new PointCol(10, 20, 5);
    p.affiche();
    p.super.affiche();
    p.deplace(2,4);
    p.affiche();
    p.colore(2);
    p.affiche();
 }
```

Pour les attributs, c'est analogue.

```
    class A { ... int a; char b; ... };
    class B extends A {float a; ... };
```

Si b est de type B, b.a fait référence à l'attribut 'a' de type float de la classe B. On accède à l'attribut 'a' de type int par b.super.a. L'attribut 'a' de la classe B s'ajoute à celui de la classe A. Il ne le remplace pas.

3. Appel des constructeurs

Les constructeurs n'obéissent pas à des règles précises comme en C++. Cependant, il est d'usage en POO que le constructeur de la sous-classe appelle le constructeur de la super-classe au début de son corps. Ainsi, on transmet les paramètres du constructeur de la sous-classe au constructeur de la super-classe:

```
 class PointCol extends Point {
 public int couleur;
    public PointCol(int x, int y, int c) {
          super (x, y) ; // appel du constructeur de la super-classe
          couleur = c ;
    }
    ...
 }
```

Si l'on utilise un destructeur, lors de l'appel du destructeur de la sous-classe on pourra suivre le même esprit que pour le constructeur:

```
class Point { ...
    public void detruire () {
          System.out.println( "-- destruc Point " + x + " " + y);
    }
    ...
}
class PointCol extends Point { ...
    public void detruire () {
        System.out.println(   "--   destruc   PointCol   "   +   couleur)   ;
        super.detruire() ;
    }
    ...
}
```

4. Contrôle des accès

« private » et « public »

Une sous-classe peut accéder aux attributs publiques de la super-classe et appeler les méthodes publiques de la super-classe. Une sous-classe n'a pas accès aux attributs privés de la super-classe.

Exemple :

```
class Point {
    public int x;
    private int y;
    public Point(int a, int b) { x = a; y = b; }
    public  void deplace(int dx, int dy) { x += dx; y += dy; }
    public  void affiche() { System.out.println("x = " + x + "y = " + y) ; }
}
class PointCol extends Point {
    public  int couleur;
    public  PointCol(int, int, int) {}
    public  void colore(int c) { couleur = c ; }
}
```

La classe `PointCol` peut accéder à `x` et ne peut pas accéder à `y`. Les classes utilisatrices U de la sous-classe suivent les mêmes règles d'accès à la super-classe que la sous-classe.

« protected »

Avec le mot-clé « protected » on faire en sorte que:

- La sous-classe a accès aux attributs protégés de la super-classe.
- Les classes utilisatrices de la super-classe (ou des sous-classes) n'y ont pas accès.

Exemple :

```
class Point {
    public int x;
    public int y;
    public Point(int a, int b) { x = a; y = b; }
    protected void deplace(int dx, int dy) { x += dx; y += dy; }
    protected void affiche() { System.out.println("x = " + x + "y = " + y);}
}
```

5. L'héritage en général

Ce paragraphe relie le vocabulaire employé en Java avec celui de l'orienté objet en général.

Si B hérite de A on dit aussi :

• B est dérivée de A	C++	A est la classe de base de B
• B est un A	objet	
• B spécialise A		A généralise B
• B est une classe fille de A		A est la classe mère de B
• B est une sous-classe de A	UML, java	A est une super-classe de B
• B is a A	(anglais ☺)	

On parle de la relation 'est-une-sorte-de' ou de la relation 'is-a'.

Dans le cas ou plusieurs classes hérite les unes des autres, on parle de taxonomie de classes ou d'arbre d'héritage.

C est un B D est un B E est un B G est un F B est un A F est un A

On dit que C est une descendante de A. A est une ascendante de C

L'héritage multiple n'existe pas en Java.

6. Compatibilité entre objets d'une super-classe et objets d'une sous-classe

Puisque la relation d'héritage est une relation forte on peut se demander si l'on peut affecter un objet de la super-classe dans un objet de la sous-classe ou pas et inversement.

Tout objet de la classe dérivée est un objet de la classe de base et que Java fait des conversions implicites de type. On peut mettre un objet de type dérivé vers un type de base mais pas l'inverse.

Si A est la super-classe et B la sous-classe, on a :

```
A a;
B b;
```

alors a = b est légal alors que b = a est illégal.

Par contre, très souvent, dans une sous-classe, une méthode de la super-classe retourne (une référence sur) un objet de la classe de base, mais puisque l'appel a été fait dans la sous-classe le programmeur sait que l'objet retourné est du type de la sous-classe donc il peut violer la règle avec l'opérateur de cast en écrivant :

```
b = (B) a ;
```

Evidemment, cette opération peut amener des catastrophes si l'objet n'est pas de la classe dérivée B mais d'une autre classe dérivée C, complètement différente de la classe B.

7. Méthodes et classes abstraites

Java propose un mécanisme pour traiter des classes ou méthodes abstraites.

classes abstraites

Une classe est abstraite si elle n'est pas instanciable, c'est-à-dire si elle ne possède pas de constructeur.

méthodes abstraites

Une méthode est abstraite si elle n'est pas définie.

notation

Pour spécifier qu'une classe ou une méthode est abstraite on utilise le mot-clé 'abstract'.

8. Deux classes de base: la classe `Object` et la classe `Vector`

Deux classes servent beaucoup et sont indispensables à connaître en Java : la classe `Object` et la classe `Vector`.

La classe `Object`

C'est la super-classe de toutes les classes java. Les signatures de ses principales méthodes sont :

```
protected Object clone() ;
public String toString() ;
public boolean equals(Object) ;
```

La méthode `clone` est l'équivalent du constructeur par recopie de C++.

La méthode `toString` renvoie une chaine de caractères.

La méthode `equals` renvoie `true` si les deux objets sont « égaux ».

Le principe de ces méthodes est la redéfinition des méthodes dans les sous-classes de la classe `Object`. Ces méthodes sont donc des méthodes par défaut.

Toute la littérature à propos des constructeurs par recopie en C++ s'applique donc tout à fait à la méthode `clone` de Java. Par défaut, la méthode `clone` recopie un à un les attributs des l'objet instance dans les attributs de l'objet créé.

Il est aussi agréable de redéfinir une méthode `toString` dans chaque classe car elle sera implicitement appelée par un `System.out.print(this) ;` Par défaut, la méthode `toString` renvoie une String du style NomDeLaClasse@adresseMemoire.

Il est enfin agréable d'avoir une redéfinition de la méthode `equals` de façon à distinguer l'égalité de deux références d'objet et l'égalité des contenus des objets référés par deux références. Par défaut, la méthode `equals` teste l'égalité des références.

La classe `Vector`

C'est une classe du package `java.util` permettant de gérer des listes d'objets, donc très pratique.

(Elle est analogue à la classe Liste décrite plus loin dans ce document. La classe Liste ne fait pas partie du langage Java ; La présence de la classe Liste dans ce document est justifiée par un souci d'homogénéité avec C++).

Les principales méthodes de la classe `Vector` sont :

```
    public Object firstElement();
    public void addElement(Object);
    public removeElement(Object);
    public Object elementAt(int);
    public boolean contains(Object);
    public int size();
```

Un exemple simple d'utilisation :

```
public class Bibli{

    public Vector mesLivres;

    public Bibli(){
          Vector mesLivres = new Vector();
    }
    …
    {
          …
          Livre L = new Livre(…) ;
          mesLivres.addElement(L);  // ajouter le livre dans 'mesLivres'
          …
          // traitement sur tous les elements du vecteur 'liste'
          for(int i=0;i<mesLivres.size();i++) {
                Livre liv=(Livre)mesLivres.elementAt(i);
                … // traitement sur le livre 'liv'
          }
          …
    }
}
```

9. Interfaces et paquetages

Ce paragraphe est un lien entre d'une part la notion classique de « module » en Génie Logiciel et, d'autre part, les interfaces et paquetages Java. Un module de GL est constitué d'une « interface », visible des utilisateurs du module, et d'un « corps », visible du seul programmeur responsable de l'implémentation du module. En Java, les notions d'interface et de paquetage ressemble à cette notion.

Interfaces

La structure syntaxique des classes Java ne permet pas une coupure naturelle entre les informations publiques et privées d'une classe : la définition des méthodes est accolée à leur déclaration (contrairement à C++ qui oblige le programmeur à placer la déclaration dans les *.h et les définitions dans les *.cpp). Cependant, Java prévoit des *interfaces* qui ne disposent que de méthodes abstraites (des déclarations de méthodes en fait) et de données. On utilise le mot-clé ‘`interface`’. Une interface Java correspond à la notion classique d'interface en génie logiciel. Une classe pourra alors implémenter une interface. On utilisera pour cela le mot-clé ‘`implements`’.

Exemple :

```
interface I {
    int a = 0 ;
    void tourner(int) ;
}
class C implements I {
    void tourner(int x) {
          …
    }
}
```

Sous cet angle, une classe est donc vue comme un morceau du corps d'un module de génie logiciel. Une classe peut implémenter plusieurs interfaces ; une interface peut étendre (mot-clé ‘`extends`’) plusieurs interfaces.

Paquetages

La notion de paquetage existe en Java, elle correspond plus ou moins à la notion de module au sens du Génie Logiciel. Un paquetage Java est surtout un ensemble de classes et d'interfaces que l'on peut regrouper dans un fichier que l'on place ensuite dans une arborescence de répertoires. Un paquetage Java permet donc de structurer l'ensemble des fichiers *.java et *.class d'une application dans des répertoires, afin de les retrouver facilement à l'utilisation. Pour définir un paquetage, on utilise le mot-clé ‘`package`’ à la première ligne d'un fichier *.java. Chaque élément du paquetage peut être privé (mot-clé ‘`private`’) ou public (mot-clé ‘`public`’).

Exemple : imaginons que le fichier contenant le paquetage ‘`monPaquetage`’ s'appelle `truc.java` et que ce fichier contient les informations suivantes :

```
package monPaquetage ;        // définition du package
public class maClasse {
      ...
      void tourner(int x) { … }
}
public class monAutreClasse {
      ...
      void virer(int x) { … }
```

```
    }
```

Pour utiliser une classe ou une interface dans un paquetage, on utilise le mot-clé ‘import’ suivi du chemin indiquant comment retouver la classe ou l’interface.

```
// toto.java
import monPaquetage.maClasse ;    //utilis. ‘maClasse’ de ‘monPaquetage’
...
    maClasse c; // correct
    ...
    c.tourner(4) ;
```

Ici, on suppose que toto.java est placé dans un répertoire qui contient le sous-répertoire ‘monPaquetage’ qui contient le fichier truc.class. Ainsi ‘import monPaquetage.maClasse ;’ permet de voir la classe ‘maClasse’ située dans le paquetage ‘monPaquetage’ depuis le fichier toto.java.

Si toto.java veut utiliser toutes les classes du paquetage ‘monPaquetage’, on peut écrire :

```
import monPaquetage.* ;     //utilis. des classes de ‘monPaquetage’
```

La structuration des paquetages dans les répertoires peut être faite à plusieurs niveaux. Par exemple, si le répertoire ‘monPaquetage’ contient un sous-répertoire ‘sonPaquet’ contenant un fichier bidule.class correspondant au source suivant :

```
    package sonPaquet ;
    public class saClasse {
          ...
          void allerToutDroit() { … }
    }
```

Alors, depuis le fichier toto.java, on pourra utiliser la classe ‘saClasse’ en écrivant par exemple :

```
// toto.java
...
import monPaquetage.sonPaquet.saClasse ;
    ...
    saClasse s;
    ...
    s.allerToutDroit() ;
```

10. Exceptions

La gestion d'erreurs faites avec des paramètres de retour dans des fonctions est une technique relativement lourde à gérer pour le programmeur. La gestion d'exceptions est un mécanisme qui est plus léger car indépendant des paramètres de retour d'une fonction. Java permet de gérer des exceptions.

Créer une exception

Une traitement devenant incorrect, comparé à son objectif, peut lancer une exception avec le mot-clé `throw` :

```
{
      // ici un traitement est incorrect
      throw monException;
}
```

Evidemment `monException` doit être un type connu. Pour cela, on doit indiquer au minimum que `monException` est une extension de la classe pré-définie `Exception` :

```
class monException extends Exception {}
```

Spécifier une exception éventuelle au niveau supérieur

La méthode contenant un lancement d'exception potentiel doit spécifier cette exception éventuelle à son appelant, en utilisant le mot-clé `throws` dans sa signature. Ci-dessous, la signature de la méthode `maMethodeAvecExceptionPossible` précise à son appelant qu'elle peut signaler l'exception `monException`.

```
void maMethodeAvecExceptionPossible() throws monException {
      …
}
```

Récupérer ou ignorer une exception ?

L'appelant d'une méthode lançant des exceptions a le choix entre :

- ignorer l'exception et la transmettre au niveau appelant supérieur en utilisant à nouveau le mot-clé `throws`,
- récupérer l'exception avec des blocs de code qui utilisent les mot-clés `try` et `catch`.

Ignorer une exception

Dans le premier cas, l'exemple donne :

```
void maMethodeAppelanteSansRecupération() throws MonException {
      …
      maMethodeAvecExceptionPossible() ;
      …
} // transmet au niveau supérieur
```

Récupérer une exception

Dans le second cas, il donne :

```
void maMethodeAppelanteAvecRecupération() {
      …
      try { // essai d'appeler la méthode
            maMethodeAvecExceptionPossible() ;
      }
      catch (MonException) { // récupère l'exception
            System.out.println(« Exception levee ») ;
            exit(0) ;
      }
}
```

Dans cet exemple, `maMethodeAppelanteAvecRecupération` choisit de ne pas transmettre l'exception au niveau supérieur, elle récupère l'exception en affichant le message « `Exception levee` » et en arrêtant le programme. D'autres récupérations moins brutales sont évidemment possibles. Noter que la signature de `maMethodeAppelanteAvecRecupération` ne contient plus le `throws`.

Mécanisme transitif

Dans cet exemple, nous avons vu comment sont gérées les exceptions entre un niveau appelant et un niveau appelé. Ce mécanisme est général à tous les niveaux d'appel. Les exceptions peuvent se transmettre de méthode en méthode au travers de l'arbre des appels jusqu'à trouver une méthode contenant un gestionnaire de cette exception (bloc `try-catch`). Si aucun gestionnaire d'exceptions n'est trouvé, le programme `main` se termine en générant l'exception.

Attribut, instance d'exception

Dans l'exemple précédent nous avons utilisé une classe exception, mais on peut aussi utiliser une *instance* d'une classe exception, définir des *attributs*, et même des méthodes dans la classe exception étendue :

```
class monException extends Exception {
      public int monAttribut ;
}
```

Ainsi on peut lancer une instance d'exception :

```
{
      monException except = new monException() ;
      except.monAttribut = -1 ;
      throw except;
      …
}
```

On peut alors traiter l'exception en utilisant la valeur de `monAttribut` de l'exception :

```
void maMethodeAppelanteAvecRecupération() {
      …
      try { // essai d'appeler la méthode
            maMethodeAvecExceptionPossible() ;
      }
      catch (MonException e) { // récupère l'exception
            System.out.println(« Exception levee ») ;
            if (e.monAttribut==-1) exit(0) ;
      }
}
```

11. Multi-tâches

Java permet de définir des processus légers ou « thread » au sein d'un programme. Cette technique est très utile dés qu'un programme doit gérer de façon asynchrone plusieurs sources d'informations (sur le réseau, sur le clavier, sur un écran graphique, etc.).

Tâche, processus, « thread » et processus « léger »

Rappels : une tâche ou processus est une exécution d'un programme de manière asynchrone au sein d'un système d'exploitation. Exemple : un processus Unix. Un processus « léger » est une tâche à l'intérieur d'un langage gère le multi-tâches. Un processus léger n'est pas vu du système d'exploitation mais seulement du langage multi-tâches. Un « thread » est un processus léger géré par Java. Si un programme Java A lance 3 threads et que un programme Java B lance 4 threads et que l'on lance un processus Unix sur A et un processus Unix sur B, on aura 2 processus Unix avec en tout 7 threads.

Etat d'un processus (léger ou pas)

Un processus peut être :

- ? inactif (il n'est pas lancé)
- ? actif (il est lancé)
- ? en attente (il est lancé mais n'a pas toutes ses ressources, il les attend)
- ? prêt (il est lancé, il a toutes ses ressources sauf le processeur)
- ? en cours (il est lancé, il a toutes ses ressources même le processeur)

Définir et lancer un thread

On peut définir un thread de deux manières : avec la classe `Thread` ou bien avec l'interface `Runnable` de java. Dans les deux cas, on doit avoir une méthode `run()` contenant le corps de l'exécution du thread. Le thread est lancé avec l'appel de la méthode `start()`.

Avec la classe Thread :

```
// definition par extension de la classe Thread
class Process extends Thread {
public void run() {
      …
}
}
// utilisation
Process p = new Process() ;
p.start() ;
```

Cette première technique est adaptée si on veut qu'un objet de la classe `Process` utilise les méthodes de la classe `Thread`. Par contre, elle est inutilisable si la classe `Process` est par ailleurs une sous-classe d'une autre classe car il n'y a pas d'héritage multiple en Java.

Avec l'interface Runnable :

```
// definition par implementation de l'interface Runnable
class Proc implements Runnable {
public void run() {
      …
}
}
// utilisation
Proc p = new Proc() ;
Thread t = new Thread(p, « mon thread »)
t.run() ;
```

Cette dernière technique est adaptée en Java si on veut que `Proc` soit par ailleurs une sous-classe d'une autre. L'inconvénient de cette technique est l'impossibilité d'appeler les méthodes de la classe `Thread`.

Synchronisation et événements

Le mot-clé `synchronized` permet de modéliser les section critiques et les méthodes `wait()`, `notify()` et `notifyAll()` permettent de modéliser la transmission d'événements ou signaux entre threads.

Section critique avec `synchronized`

Un seul processus à la fois exécute le code contenu dans une section critique associée à un objet. Pour spécifier qu'une section de code est critique associée à un objet, on utilise le mot-clé 'synchronized' et, à la création du thread, on lui passe cet objet en paramètre dans le constructeur. On peut explicitement spécifier la section critique sur l'objet :

```
synchronized (objet) {
      // section critique
}
```

ou implicitement en mettant le mot-clé `synchronized` devant une méthode d'instance :

```
synchronized void maMethodeSynchronisee() {
   // section critique
}
```

Exemple concret :

```
class Ballon extends Object {
    public String nom;
    public Ballon(String n) { nom = n; }
    public synchronized void prendre_donner(Joueur j) {
          j.sleep(1000);
          System.out.println(j + " LACHE le ballon ");
    }
    public static void main(String args[]) {
          Ballon a = new Ballon("adidas", 0);
          Joueur gauche = new Joueur(a, "zidane");
          Joueur droite = new Joueur(a, "ronaldo");
          gauche.start();
          droite.start();
    }
}
```

```
class Joueur extends Thread {
    public Ballon monBallon;
    public String nom;
    public Joueur(Ballon b, String n) { monBallon = b; nom = n; }
    public void run() { for(;;) monBallon.prendre_donner(this); }
}
```

Ici on a défini la classe `Joueur` comme extension de la classe `Thread`. On a passé le ballon « adidas » au constructeur des deux threads. Il sert donc pour synchroniser les deux threads. La méthode `prendre_donner` appliquée au ballon « adidas » ne peut être exécutée que par un seul `Thread` à la fois car elle est déclarée avec le mot-clé `synchronized`. L'exécution sera donc une alternance d'exécutions du thread « ronaldo » et du thread « zidane ».

Il faut retenir qu'en cours d'exécution, un thread rentrant dans une section critique demande l'accès à l'objet associé. Si la section critique est libre, le thread rentre effectivement dans la section critique et l'occupe. Sinon, le thread attend et dés que la section critique est libérée, il y rentre effectivement. Avec le mot-clé `synchronized`, on a donc la notion de sémaphore à une place.

Les méthodes d'envoi et réception de signaux `wait()`, `notify()` et `notifyAll()`

On peut améliorer le mécanisme précédent avec les méthodes `wait()`, `notify()` et `notifyAll()`.

Un thread qui occupe une section critique associée à un objet peut la libérer en appelant la méthode '`wait()`' sur l'objet. En faisant cela, le thread se bloque et attend qu'un autre thread fasse un '`notify()`' sur le même objet.

Pour chaque objet, Java gère une file d'attente des threads en attente d'un `notify()` sur cet objet. Dés qu'un thread utilisant cet objet fait un '`notify()`', Java met le premier thread de la file d'attente dans l'état prêt. Ainsi ce thread pourra à nouveau avoir la main lorsque la section critique sera libre.

Si un thread fait un `notifyAll()` sur un objet associé à une section critique, tous les threads qui avaient effectué un `wait()` sur cet objet sont mis dans l'état prêt.

Bien noter qu'un thread qui fait un `wait()` dans une section critique n'est plus occupant de la section critique. Ce qui permet à un autre thread de rentrer dans la section critique (et donc de faire un `notify()` ou un `notifyAll()` qui débloquera les threads qui attendent).

Bien noter aussi qu'une section critique associée à un objet peut être fragmentée et non connexe.

Avec ce mécanisme, on peut alors implémenter une classe `Semaphore` au sens habituel. Un sémaphore possède des places qui peuvent être réservées et libérées par des `Processus`, extensions des threads.

```
class Semaphore {
    public int places;      // nombre de places max
    public int libres;      // nombre de places libres
    public Semaphore(int p) { places = p; libres = p; }
    public synchronized void reserver(Processus p) {
          if (libres>0) { p.etat = 2; libres--; }
          else { p.etat = 1; wait(); p.etat = 2; libres--;}
    }
    public synchronized void liberer(Processus p) {
          if (libres < places) { libres++; notify(); p.etat = 0;}
    }
    public static void main(String args[]) {  // programme de test
          Semaphore s = new Semaphore(2);
          Processus p1 = new Processus(s, "a", 10);
          Processus p2 = new Processus(s, "b", 15);
          Processus p3 = new Processus(s, "c", 25);
          p1.start(); Thread.sleep(1000);
          p2.start(); Thread.sleep(1000);
          p3.start();
    }
}
class Processus extends Thread {
    public static Vector mesProcessus = new Vector();
    public Semaphore monSemaphore;
    public String nom;
    public int periode;
    public int etat;  // 0 = libre, 1 = attente , 2 = propri
    public Processus(Semaphore s, String n, int p) {
          monSemaphore = s; nom = n; periode = p;
          etat = 0; nombre = 0;
          mesProcessus.addElement(this);
    }
    public void run() {
          System.out.println(mesProcessus);
          for(;;) {
                monSemaphore.reserver(this);
                System.out.println(mesProcessus);
                sleep(periode*1000);
                monSemaphore.liberer(this);
                System.out.println(mesProcessus);
                sleep(periode*1000);
          }
    }
}
```

(les blocs `try catch` autour des `sleep` et `wait` ont été enlevés pour la lisibilité du programme).

Gérer explicitement l'activité des threads avec des méthodes

Même si il n'est pas très propre, de gérer l'activité des threads en suspendant, arrêtant, redémarrant, endormant un thread explicitement plutôt qu'en utilisant des sémaphores, Java permet de le faire avec les primitives :

```
sleep(int)  // endormir pour un temps donné
suspend()   // suspendre
stop()      // arrêter définitivement
yield()     // rendre le processeur, passer de 'prêt' à 'en attente'
join()      // attendre la fin d'un processus
resume()    // redémarrer après suspension
```

On peut donner des priorités aux threads avec les méthodes `getPriority()` et `setPriority(int)` allant de 1 à 10, 5 étant la priorité normale. Un processus de priorité forte sera exécuté en premier.

12. Traitement des fichiers

A partir d'un exemple simple, ce paragraphe explique comment il est possible de sauvegarder et charger des données dans un fichier. Ce genre de manipulation est très utile pour stocker les informations engendrées par une exécution du logiciel avec un utilisateur.

Dans l'exemple, une bibliothèque (classe `Bibli`) contient des livres (class `Livre`). La bibliothèque contient une liste des livres de type `Vector`.

```
public class Livre{
    public int _noReference;
    public String _titre;
    public String _auteur;
    public int _nbExemplaires;
    …
}
public class Bibli{
    public Vector _listeLivres ;
    public Bibli(){
          Vector _listeLivres = new Vector();
    }
    …
}
```

Pour ouvrir un fichier (et pouvoir y écrire des informations de manière souple ensuite), on appelle successivement des constructeurs des classe `File`, `FileOutputStream` et `DataOutputStream`.

```
    try{
          File livres = new File("livres");
          FileOutputStream fosLivres = new FileOutputStream(livres);
          DataOutputStream dosLivres = new DataOutputStream(fosLivres);
          _BMuni.sauverLivres(dosLivres);
    }
    catch(IOException e) {
          System.out.println("Problemes de Fichiers!");
    }
```

Pour écrire effectivement dans le fichier, on utilise les méthodes `writeInt()`, `writeUTF()` de la classe `DataOutputStream`.

```
    public void sauverLivres(DataOutputStream dos) throws IOException{
          dos.writeInt(this._listeLivres.size());
          for(int i=0;i<this._listeLivres.size();i++){
                Livre liv =(Livre)_listeLivres.elementAt(i);
                dos.writeInt(liv._noReference);
                dos.writeUTF(liv._titre);
                dos.writeUTF(liv._auteur);
                dos.writeInt(liv._nbExemplaires);
          }
    }
```

Pour ouvrir un fichier (et pouvoir y lire des informations de manière souple ensuite), on appelle successivement des constructeurs des classe `File`, `FileInputStream` et `DataInputStream`..

```
    Bibli _BMuni = new Bibli();
    File sauvegardeLivres = new File("livres");
    FileInputStream fisLiv = new FileInputStream(sauvegardeLivres);
    DataInputStream disLiv = new DataInputStream(fisLiv);
```

```
_BMuni.recupLivres(disLiv);
```

Pour lire des informations sauvegardées de la manière décrite ci-dessus dans un fichier, on utilise les méthodes `readInt()`, `readUTF()` de la classe `DataInputStream`..

```
public void recupLivres(DataInputStream dis) throws IOException{
      int tailleFichier=dis.readInt();
      for(int i=0;i<tailleFichier;i++){
            int noRef=dis.readInt();
            String titre=dis.readUTF();
            String auteur=dis.readUTF();
            int nbExemp=dis.readInt();
            Livre l = new Livre(noRef,titre, auteur, nbExemp);
      }
}
```

13. L'environnement de développement

En général, on fait du java avec un environnement de développement, soit en tant que tel (visual J++, visual Age, visual Café etc.), soit avec l'environnement de développement minimal que l'on peut récupérer gratuitement sur le web (Java Developer Kit : JDK), soit avec un browser (navigateur) web. L'utilisation de tels environnements, si elle facilite la tâche des programmeurs avancés, est souvent le premier obstacle qui arrête net l'élan du programmeur débutant. Ce paragraphe rassemble quelques informations indispensables pour démarrer en utilisant l'un ou l'autre des environnements.

Visual xxx

Avant de commencer quoi que ce soit en visual XXX (Café, Age, J++, etc), il est essentiel créer un nouveau «projet» et/ou un «workspace» puis de savoir les ré-ouvrir lors de sessions ultérieures.

créer un « projet » ou un « workspace »

Un « projet » (ou un « workspace ») est une enveloppe autour de votre logiciel qu'il est nécessaire de créer. Pour créer un projet, cliquer sur le bouton « file » de la barre du haut, puis sur « new ». Une fenêtre apparaît. Dans la case « location » écrire le nom du répertoire où vous voulez placer le répertoire de votre projet. Attention, ce répertoire va contenir vos sources. Il est important de ne pas oublier d'écrire son nom sinon votre projet va être mis dans un répertoire qui visual XXX choisit à votre place et que vous ne saurez pas nécessairement retrouver. Dans la case «project name» écrire le nom du projet créé (par exemple « garage »). Sélectionner Java Project puis cliquer sur « OK ».

Revenir dans windows. Un répertoire a été créé et il porte le nom du projet. Le répertoire « garage » ne contient que quelques fichiers pour la gestion du projet (.dsp, .dsw, etc...).

Si vous voulez utiliser des sources java, que vous possédez par ailleurs, faire 2 choses : en windows, ramener ces fichiers dans le répertoire du projet et, en visual XXX, les ajouter au projet. Si vous n'avez pas de fichiers java à utiliser (c'est le cas lorsque vous débutez), vous pouvez sauter la lecture de ce paragraphe. Pour les placer dans le répertoire du projet, faire un « copier-coller » depuis le répertoire où se trouvent vos fichiers .java vers le répertoire de votre projet. On voit alors que le répertoire du projet contient les fichiers . java et les fichiers de gestion du projet (.dsp, .dsw, etc.). Rien de difficile. Pour les ajouter au projet, revenir dans visual XXX. Faire « Project » « add to project » « files ». Sélectionner les fichiers qui vous intéresse. Cliquer sur « OK ». Maintenant, en cliquant sur le + de « garage files », vous devez voir vos fihciers dans le projet et en cliquant sur le + de « garage classes », vous devez voir les classes java décrites par vos fichiers .java.

Il faut savoir que visual XXX crée toujours un « workspace » en plus du projet. Un « workspace » est une enveloppe autour de plusieurs projets. Cette notion est inutile pour l'instant car vous n'avez besoin que d'un seul projet.

réouvrir un projet ou un workspace existant

Pour ouvrir un projet ou workspace déjà créé la dernière fois, faire « file » « open wokspace » ou bien « file » « open » et sélectionner le fichier « garage.dsp » ou « garage.dsw » selon que vous voulez ouvrir le projet ou le workspace. Votre projet est chargé. Ces opérations de création et de réouverture ne sont pas évidentes au début mais elles doivent absolument devenir des automatismes car elles sont effectuées souvent.

Construire et démarrer l'application

Faire « build » pour construire l'application. Dans la fenêtre en bas, si votre programme ne contient pas d'erreur, apparaît le message : « 0 error(s), 0 warning(s) ». Sinon, corriger les erreurs et refaire « build » tant qu'il y a des erreurs.

Faire « execute » pour lancer l'application. La première fois que l'on effectue le démarrage une fenêtre apparaît avec deux possibilités : « applet » ou « stand-alone ». Cliquer sur « stand-alone » pour avoir une application console classique et sur « applet » pour avoir une applet (mini-application s'insérant dans une page web).

A la question « class file name », répondre par le nom de la classe contenant le programme « main » de votre application. Attention aux majuscules-minuscules, sinon java ne trouvera pas le programme principal et votre application ne démarrera pas.

Java Developer Kit (JDK)

Le Java Developer Kit (JDK) est un environnement de développement gratuit récupérable sur le site Web de Sun. Son utilisation est très simple avec des commandes lignes. En Linux, on l'utilise avec essentiellement deux commandes :

'javac' (compilateur),
'java' (interpréteur).

Sous Linux, **pour compiler** un fichier Ballon.java il suffit de faire :

```
javac Ballon.java
```

Et un fichier Ballon.class est engendré ; il s'agit du « bytecode » java.

Pour exécuter (interpréter) un fichier Ballon.class il suffit de faire :

```
java Ballon
```

Et le programme principal 'main()' de la classe Ballon est exécuté.

Attention, pour installer le JDK, des variables d'environnement peuvent être à positionner correctement :

JDK_PATH : le répertoire où est installé le JDK
PATH : le répertoire ou se trouve les binaires 'java' et 'javac' ;
On peut avoir en général PATH = JDK_PATH /bin
CLASSPATH : le répertoire où javac met les classes compilées.
JAVA_HOME/bin : le répertoire contenant les commandes java et javac

En bash, faire éventuellement un `export CLASSPATH=.` pour que les fichiers .class soient placés et recherchés dans le répertoire courant.