# Introduction à la programmation Java

F. Barthélemy

28 février 2005

## 1 But du cours

Le but de ce cours est d'entrer dans le langage JAVA à un niveau de description moyen : plus détaillé que dans le premier cours, sans pour autant entrer dans tous les détails qui devront être cherchés ailleurs. On désire donner une connaissance suffisante pour commencer à programmer (ED et TP).

## 2 Bases du langage

### 2.1 Les valeurs en JAVA

Les valeurs manipulées sont des soit des objets, soit des tableaux, soit des objet d'un type primitif.

Ce qui n'est pas objet :
- les valeurs de types prédéfinis, dits types primitifs
- les tableaux.

Ce qui est objet :
- Les instances de classes prédéfinies (ex : les chaînes, String)
- Les instances de classes définies par l'utilisateur.

Les types primitifs :
- types entiers :
  - byte (8 bits)
  - short (16 bits)
  - int (32 bits)
  - long (64 bits) ex : 0L, 123l
- types flottants :
  - float (32 bits) ex : 0.f 123.45f
  - double (64 bits) ex : 0.8 56.89
- type char ex : 'a' 'e'
- type boolean , avec deux valeurs true flase

Il n'y a pas d'autres types primitifs.

Les tableaux ont un statut spécial :
- Ce ne sont pas des objets :
  - il n'y a pas de méthodes
  - il n'est pas possible d'hériter des tableaux
- mais ils ont en commun avec les objets d'avoir un état qui change au fil du temps.

Les tableaux :
- sont toujours à une dimension (mais on peut faire des tableaux de tableaux).
- sont indicés par des valeurs de types int à partir de 0.
- la longueur d'un tableau est une propriété dynamique du tableau : elle ne fait pas partie de son type. Une variable peut contenir successivement des tableaux de différentes tailles.

Déclaration de tableau :

```
int n; // ceci est un entier, pas un tableau
int[] t; // ceci est un tableau d'entiers
Compte[] tc; // ceci est un tableau de comptes
int[][] tt; // ceci est un tableau de tableaux d'entiers
int[] n = {1,2,3}; // tableau éinitialis
```

Création d'un nouveau tableau :

```
t = new int[5];
tc = new Compte[23];
tt = new int[3][4];
```

Accès aux éléments d'un tableau :

```
t[0] = 1;
tc = new Compte();
tt[2][2] = t[0] + 3;
```

Voici un exemple complet avec des tableaux :

```
class Tableau{
    int[] tab = {2, 4, 7};
    void affiche(){
        for (int i=0; i<tab.length; i++){
            System.out.print(" " + tab[i]);
        }
        System.out.println();
    }
    void foisDeux(){
        for (int i=0; i<tab.length; i++){
            tab[i] = tab[i] * 2;
        }
    }
    void ajoute(int x){
        int[] nt = tab;
        tab = new int[nt.length+1];
        for (int i=0; i<nt.length; i++){
            tab[i] =  nt[i];
        }
        tab[tab.length-1] = x;
    }
    public static void main(String[] args){
        Tableau t = new Tableau();
        t.foisDeux();
        t.ajoute(5);
        t.affiche();
    }
}
```

Les chaînes de caractères sont des objets de la classe prédéfinie String. Ce ne sont pas des tableaux de caractères. On ne peut pas modifier la valeur d'une chaîne (par exemple, on ne peut pas changer la valeur d'un caractère comme dans un tableau). Il y a de nombreuses méthodes prédéfinies.

## 2.2 Variables et types

Une variable est toujours déclarée avec un type.

Une variable a toujours une valeur par défaut qui dépent de son type :

- 0 pour les int, 0.f pour les float, false pour les boolean, etc
- null pour les tableaux et les objets

Certaines conversions de types sont automatiques :

- on peut affecter une valeur int dans une variable long
- on peut mettre un objet d'une sous-classe du type déclaré d'une variable (ex : CompteSecurise dans une variable de type Compte)

Il y a pour cela un ensemble de règles décrivant les conversions possibles et celles qui ne le sont pas. Les règles sont assez compliquées. On ne les détaille pas ici.

On peut aussi utiliser une conversion explicite (cast). Là encore, tout n'est pas possible.

Possible : conversion d'un byte en char. Conversion d'une sous-classe vers une super-classe (ex : Compte -¿ Object). Impossible : conversion d'un int en tableau de Compte.

```
char c;
c = (char) 38;
Object o;
o = (Object) new Compte();
```

## 2.3 Les instructions de contrôle

Elles sont classiques.

```
// bloc
{
   int n;
   n = 1+2;
};
// conditionnelle
if (true)
  n = 0;  // une seule instruction (ou un bloc)
else
  n = 1;  // une seule instruction (ou un bloc)
// for
int i;
for (i=0;i < 4;i++)
  System.out.println(i);  // une seule instruction (ou un bloc)
// while
while (true)
  System.out.println("Ca boucle");  // une seule instruction (ou un bloc)
// do while
do
  System.out.println("Ca boucle");  // une seule instruction (ou un bloc)
while (true);
```

JAVA est langage purement objet : il n'y a pas de variables globales, pas de fonctions, pas de procédures : les variables et les méthodes apparaissent seulement dans des classes.

# 3 Les exceptions

Comme dans beaucoup de langages, il existe un moyen de gérer les erreurs intervenant à l'exécution : les exceptions. Ce moyen peut également être utilisé à d'autre fins (ex : sortie de boucle), dans un style de programmation particulier mais correct.

Une exception doit être déclarée, ensuite elle peut être levée et éventuellement récupérée.

Une exception est un objet, instance de la classe prédéfinie Exception (ou d'une sous-classe d'Exception).

```
Exception monException = new Exception();
```

Pour lever une exception, on utilise l'instruction throw

```
throw monException
```

Cela a pour effet de suspendre tout calcul, sauf si l'exception est récupérée par un bloc appelant du code où l'exception est levée.

Prenons l'exemple des comptes sécurisés. On peut utiliser une exception pour empêcher le retrait d'une somme quand le compte n'est pas approvisionné.

```
class CompteSecurise extends Compte{
  Exception provisionInsuffisante = new Exception();
  void retrait(int n) throws Exception {
    if (solde < n)
      throw provisionInsuffisante;
    solde = solde - n;
  }
}

  ...

  CompteSecurise durand = new CompteSecurise();
  durand.depot(200);
  durand.retrait(300);
  ...
```

Dans le cas ou on essaie de retirer plus d'argent qu'il n'y en a, l'exception est levée et l'exécution du programme s'arrete, avec une message du type :

```
java.lang.Exception
        at ess.<clinit>(ess.java:13)
```

Notez que chaque méthode doit déclarer les exceptions qui peuvetn être levées au cours de l'exécution de leur corps (cf exemple ci-dessus).

L'utilisateur de la méthode, sachant que l'exception provisionInsuffisante peut être levée, peut prévoir une facon moins brutale de gérer ce cas que l'arrêt pur et simple de l'exécution.

```
  CompteSecurise durand = new CompteSecurise();
  durand.depot(200);
  try{
      durand.retrait(300);
      System.out.println("Retrait effectue");
      }
  catch (Exception pi)
      {
      System.out.println("Provision insuffisante");
      };
```

L'exemple du compte securisé illustre la gestion d'erreurs. On peut aussi utiliser une exception pour sortir d'une boucle, sans qu'il y ait d'erreur :

```
boolean trouve = false;
int[] tab;
```

```
Exception sortie = new Exception("Sortiee");  // "Sortiee" est un
                              // attribut de l'exception que l'on
                              // utilise pour l'identifier èaprs.
boolean rechercher(int n){
  int i;
  methodeQuiInitialise(tab);
  try
   {
   for(i=0; i < tab.length; i++){
      if (tab[i] = n){
         trouve = vrai;
         throw sortie;
         };
      }
  }
  catch (Exception s){
      if ("Sortiee".equals(s.getMessage()))
         ;
      else
         throw s;  // l'exception écapture n'est pas la bonne
   };
  return trouve;
  }
```

L'exception est utilisée pour sortir de la boucle for dès que l'entier a été trouvé dans le tableau.
Voici un exemple complet qui illustre le comportement des exceptions :

```
class Exo12_2{
    static void meth1(int p) throws Exc1, Exc2{
        Terminal.ecrireStringln("Debut de meth1");
        if (p == 0) throw new Exc1 ();
        Terminal.ecrireStringln("Milieu de meth1");
        if(p ==1) throw new Exc2 ();
        Terminal.ecrireStringln("Fin de meth1");
    }
    static void meth2(int p) throws Exc1, Exc2, Exc3{
        Terminal.ecrireStringln("Debut de meth2");
        if (p == 2) throw new Exc3();
        Terminal.ecrireStringln("Milieu de meth2");
        meth1(p);
        Terminal.ecrireStringln("Fin de meth2");
    }
    static void meth3(int p) throws Exc1, Exc3{
        Terminal.ecrireStringln("Debut de meth3");
        try{
            meth2(p);
        }catch(Exc2 ex){
            Terminal.ecrireStringln("Debut de meth3");
        }
        Terminal.ecrireStringln("Fin de meth3");
    }
    public static void main(String[] args)
                                    throws Exc1, Exc3{
        Terminal.ecrireString("Entrez un nombre: ");
```

```
        meth3(Terminal.lireInt());
    }
}
```

# 4 Retour sur les classes

Nous avons déjà vu au chapitre précédant ce qu'est une classe et nous avons vu certains des composants pouvant apparaître : ils s'agit des variables et méthodes d'instance. Il en existe d'autres.

Composants d'une classe :

- 
- constructeurs
- variables d'instance
- variables de classe
- méthodes d'instance
- méthodes de classe
- initialiseurs

## 4.1 This et super

A l'intérieur d'une méthode d'instance, il peut être utile de faire référence à l'objet sur lequel cette méthode est appliquée. Or cet objet n'est connu qu'à l'appel de la méthode et pas au moment de la définition de cette méthode. De plus, elle peut s'appliquer sur beaucoup d'objets différents.

Quel que soit l'objet en question, il est désigné par le nom this.

Le nom super désigne le même objet, mais avec un changement de type. Par exemple, dans une méthode de CompteRemunere , super est équivalent à (Compte) this. Ceci permet d'accéder aux méthode de la super-classe qui sont masquées. En pratique, on n'en a vraiment besoin que pour l'appel du constructeur de la super-classe dans la définition d'un constructeur (cf ci-dessous).

Quand une méthode est invoquée sans qu'un objet soit précisé, implicitement, c'est sur this qu'elle s'applique.

Exemple 1 : on peut réécrire la classe des comptes en mentionnant toujours explicitement l'objet sur lequel on invoque une méthode.

```
class Compte{
  int solde = 0;
  void depot (int n){
    this.solde = this.solde + n;
    }
  void retrait (int n){
    this.solde = this.solde - n;
    }
  void print (){
    System.out.println(this.solde);
    }
}
```

Mais il y a aussi des cas où on est obligé de recourir explicitement à this : par exemple quand une méthode doit passer en paramètre l'objet sur lequel elle est appelée en paramètre à une autre méthode.

Prenons l'exemple des formes géométriques. On peut vouloir les placer dans un plan. On peut avoir plusieurs plans, et déplacer les formes de l'un dans l'autre.

```
class Plan{
   Forme[] contenu = new Forme[12];
```

```
    int premierLibre = 0;
    void ajouter(Forme f){
        contenu[premierLibre]=f;
        premierLibre++;
        f.setPlan(this);
    }
    ...
}

class Forme{
    Plan plan;
    int origine;
    void setPlan(Plan p){
        plan = p;
    }
    void placerDans(Plan p){
        p.ajouter(this);
    }
}
```

L'usage de super est beaucoup plus limité. Il peut servir à utiliser une méthode de la super-classe masquée dans la classe courante. Il peut servir aussi à invoquer un constructeur de la super-classe comme nous allons le voir tout de suite.

## 4.2 Les constructeurs

Les constructeurs servent à créer des objets, grâce à la construction new.

On peut déclarer et définir des constructeurs d'une classe. Cela ressemble à la définition d'une méthode mais :

- un constructeur ne renvoie pas de valeur (il crée un objet). Il n'y a pas de type de valeur de retour.
- un constructeur doit porter le même nom que la classe des objets qu'il crée.

Pour le reste, un constructeur est comme une méthode :

- il a une liste de paramètres
- il a un corps
- il peut être déclaré public, private ou protected
- il a un corps

Exemple : on peut définir la classe des points d'un plan comme suit.

```
class point{
  int abcisse;
  int ordonnee;
  point(int x, int y){
    abcisse=x;
    ordonnee=y;
    }
  ...
}
```

On utilise ici le constructeur pour initialiser les variables d'instance par des valeurs qui ne sont pas connues à la compilation de la classe mais seulement au moment de la création de l'objet. C'est une utilisation classique d'un constructeurs.

On appelle le constructeur de la facon suivante :

```
  ...
```

```
  point p;
  p = new point(1,2);
  ...
```

Si aucun constructeurs n'est défini dans une classe, un constructeur par défaut qui ne prend pas de paramètre est créé. C'est lui que nous avons utilisé jusqu'ici.

Plusieurs constructeurs peuvent être donnés pour une seule et même classe pourvu que la liste des paramètres les différencie (soit par le nombre de paramètres, soit par leurs types).

Par exemple :

```
class point{
  int abcisse;
  int ordonnee;
  point(int x, int y){
    abcisse=x;
    ordonnee=y;
    }
  point(){
    abcisse=0;
    ordonnee=0;
    }
  point(int x){
    abcisse=x;
    ordonnee=0;
    }
  ...
}
```

On pourra alors choisir entre les trois constructeurs pour créer un point :

```
...
point p;
p = new point();
p = new point(2);
p = new point(1,2);
...
```

La forme des arguments permet de déterminer sans ambiguité le constructeur appelé.

Il s'agit de surcharge du constructeur. Le même phénomène apparaît pour les méthodes. Par exemple, pour les affichages à l'écran, l'objet System.out possède plusieurs fonctions println permettant d'imprimer différents types de valeurs.

```
    public void println();
    public void println(boolean b);
    public void println(char c);
    public void println(char s[]);
    public void println(double d);
    public void println(float f);
    public void println(int i);
    public void println(long l);
    public void println(Object obj);
    public void println(String s);
```

Une dernière chose à propos des constructeurs : la première ligne du corps d'un constructeur peut être une invocation d'un autre constructeur de la classe ou de la super-classe, sans qu'un objet différent soit créé. Pour cela, on a une construction différente, sans new .

Exemple :

```
class point{
  int abcisse;
  int ordonnee;
  point(int x, int y){
    super();
    abcisse=x;
    ordonnee=y;
    }
  point(){
    this(0,0);
    }
  point(int x){
    this(x,0);
    }
  ...
}
```

### 4.3 Variables et méthodes de classe

Les variables de classe : il s'agit de variables qui sont partagées par tous les objets d'une classe. Ces variables ne sont donc pas allouées à la création d'un objet (comme les variables d'instance), mais à la création de la classe.

Les variables de classes sont déclarées avec le mot clé static.

Un exemple d'utilisation d'une variable de classe : un compteur qui compte le nombre d'instances d'une classe créées au cours d'une exécution :

```
class Point{
    int x, y;
    static int compteur = 0;
    Point(int x1, int y1){
        x = x1;
        y = y1;
        compteur++;
    }
    void translation(int dx, int dy){
        x = x + dx;
        y = y + dy;
    }
    protected void finalize() throws Throwable{
        compteur--;
        super.finalize();
    }
    public static void main(String[] argv){
        Point P1, P2;
        System.out.println(Point.compteur);
        P1 = new Point(0,0);
        System.out.println(Point.compteur);
        System.out.println(P1.compteur);
        P2 = new Point(5,6);
        P1 = new Point(7,8);
        new Point(4,2);
        System.out.println(P2.compteur);
        System.gc();
```

```
        System.out.println(P2.compteur);
    }
}
```

Les variables de classe sont connues des objets, mais on peut également les dénommer via le nom de la classe.

```
...
point p = new Point(10,15);
p.compteur;
Point.compteur;
...
```

Les méthodes de classes sont des méthodes qui n'utilisent et modifient que des variables de classe. Cela signifie que ces méthodes peuvent s'exécuter indépendamment de l'état d'un objet. On peut donc elles aussi les appeler soit sur la classe soit sur n'importe quelle instance de la classe avec le même effet.

Exemple :

```
class Point{
  static int compteur = 0;
  int abcisse;
  int ordonnee;
  Point(int x, int y){
    abcisse=x;
    ordonnee=y;
    compteur++;
    }
  static void afficheCompteur(){
    System.out.println(compteur)
    }
  ...
};
...
point p = new Point(10,15);
p.afficheCompteur;
Point.afficheCompteur();
...
```

Les méthodes et variables de classe sont le patrimoine commun de tous les objets de la classe. Elles existent même si la classe n'a aucune instance (aucun objet de ce type créé.

Comme les autres composants, les méthodes et variables de classe peuvent être déclarées public, private ou protected.

En cas de besoin, un bloc d'initialisation statique peut être donné pour initialiser les variables de classe.

## 4.4 Règles de visibilité

Il existe 4 modes qui caractérisent la visibilité des composants d'une classe :

- private : le composant n'est visible que par les méthodes de la classe
- public : le composant est visible de partout
- protected : le composant est visible à l'intérieur de la classe, des sous-classes et des autres classes du même package (fichier)
- par défaut, sans déclaration spéciale, le composant est visible à l'intérieur du package seulement

Ces définitions font référence à la notion de package. Nous n'insisterons pas ici sur la notion de package. Un package est un ensemble de classes. Son nom est également un chemin d'accès (répertoire) donné à partir d'une racine contenue dans la variable d'environnement CLASSPATH.

Par exemple, la classe prédéfinie DataInputStream appartient au package java.io. Son nom complet est java.io.DataInputStream. Elle est contenue dans le répertoire java/io/ et le fichier compilé qui la contient s'appelle DataInputStream.class. Le répertoire java/io est lui-même contenu dans un répertoire qui varie d'une machine à l'autre. Par exemple, sur ma machine, ce répertoire est : /usr/lib/java/classes/ . Le fichier source DataInputStream.java est déclaré comme appartenant au package à l'aide de la clause :

```
package java.io;
```

On peut utiliser les objets d'un package en les dénotant par leur nom complet. L'exemple suivant montre la déclaration d'une variable de type DataInputStream .

```
java.io.DataInputStream DIS;
DIS = new java.io.DataInputStream(...);
```

Une clause d'importation de package permet d'utiliser directement les objets du package, sans utiliser le nom complet (gestion de noms).

Reprenons notre exemple :

```
import java.io.DataInputStream;
DataInputStream DIS;
DIS = new DataInputStream (...);
```

Par défaut, sans déclaration spéciale, les classes que l'on crée sont ajoutées dans un package sans nom. C'est ce que nous allons utiliser au début pour les TP.

# 5 Abstraction

## 5.1 Classes abstraites

Notion de classe abstraite : ce sont des classes qui déclarent des méthodes sans définir le code correspondant. Il y a déclaration de méthodes ne pouvant pas être exécutées. On ne peut pas instancier une classe abstraite (créer des objets de cette classe). Elles ne servent qu'à créer un noeud intermédiaire dans la hiérarchie des classes. Les sous-classes concrètes devront implémenter toutes les méthodes abstraites de la super-classe.

```
abstract class Arbre {
    abstract boolean estVide();
}
class AVide extends Arbre {
    AVide(){}
    boolean estVide() {
        return true;
    }
}
class ACons extends Arbre {
    int label;
    Arbre gauche, droite;
    ACons(int label, Arbre gauche, Arbre droite) {
        this.label = label;
        this.gauche = gauche;
        this.droite = droite;
    }
```

```
    boolean estVide() {
        return false;
    }
}
class Test{
    public static void main(String[] args){
        Arbre a =
            new ACons(1,new ACons(2,new AVide(),new AVide()),
                      new ACons(3,new AVide(),new AVide()));
        System.out.println(a.estVide());
    }
}
```

Autre exemple :

```
abstract class truc{
  abstract void m();
}

class sousTruc extends truc{
  void m(){
    System.out.println("je m'execute");
  }
}

class abst{
  public static void main(String[] arg){
    truc t;
    t = new sousTruc();
    t.m();
  }
}
```

On voit dans cet exemple que le fait d'être abstrait n'empêche pas de déclarer une variable de type truc. Si on n'avait pas déclaré la méthode abstraite, on n'aurait pas pu utiliser la méthode m sur t.

```
class truc{
   int bidon = 46;
}

class sousTruc extends truc{
  void m(){
    System.out.println("je m'execute");
  }
}

class abstBis{
  public static void main(String[] arg){
    truc t;
    t = new sousTruc();
    t.m();
  }
}
```

```
$ javac abstBis.java
abstBis.java:15: Method m() not found in class truc.
    t.m();
      ^
1 error
```

Les classes abstraites sont un support facilitant la liaison retardée.

## 5.2 Interfaces

Une interface est un type qui regroupe des objets de différentes classes non nécessairement reliés par une relation d'héritage. C'est un myen de créer des types indépendants de la hiérarchie des classes.

Une interface consiste en une déclaration de constantes et de méthodes. Déclaration, c'est à dire le nom, le type des paramètres le type du résultat, mais pas la définition. Exemple :

```
interface affichage{
   void print();
}
```

Les méthodes d'une interface sont implicitement public et les variables d'une interface sont implicitement public static final

Une fois qu'une interface existe, pour que cette construction soit intéressante, il faut définir plusieurs classes qui *implémentent* cette interface, c'est à dire qui fournissent des valeurs aux constantes et des corps aux méthodes déclarées dans l'interface et qui de plus sont déclarées explicitement comme implémentant l'interface.

```
class Compte implements affichage{
  int solde = 0;
  void depot (int n){
    solde = solde + n;
    }
  void retrait (int n){
    solde = solde - n;
    }
  void print (){
    System.out.println(solde);
    }
}

class Liste implements affichage{
   int[] tab = new int(8);
   ...
   void print(){
      int i;
      System.out.print("{");
      for (i=0; i < tab.length; i++){
        System.out.print(tab[i]);
        if (i != tab.length - 1)
            System.out.print(",");
      };
      System.out.print("}");
   }
   ...
}
```

Bien que les différentes classes ne soient pas liées par des relations d'héritage, elles ont un sous-ensemble de fonctionnalités communes qui les font se ressembler (celles de l'interface). On peut utiliser l'interface comme un type. On peut déclarer une variable ayant pour type l'interface, et affecter à cette variable des instances de n'importe quelle classe implémentant l'interface.

```
affichage a;
if ((char) System.in.read()='o';)
   a = new Compte();
else
   a = new Liste();
a.print();
```

Autre exemple : supposons qu'on ait écrit différentes classes décrivant différentes structures de données (listes, piles, arbres, anneaux) contenant des entiers. Bien que toutes différentes, et a priori sans lien d'héritage, ces différentes structures peuvent partager une interface du genre :

```
interface dataStruct{
   boolean estVide();  // teste si la srtucture vide
   boolean recherche(int I); // recherche d'uné élment dans la structure
   int somme(); // renvoie la somme desé élments
```

Dans ce dernier cas, on peut hésiter entre définir une interface ou créer une classe abstraite.

- A certains égards les interfaces sont plus générales que les classes abstraites :
  - elles ne supposent pas de relation d'héritage entre les classes qui les implémentent
  - une classe ne peut hériter que d'une super-classe, mais peut implémenter plusieurs interfaces
- A d'autres égards, les classes abstraites sont plus puissantes :
  - elles peuvent contenir des variables
  - elles peuvent contenir des variables et méthodes non public
  - elles peuvent mélanger des composants abstraits et concrets

Comme les classes abstraites, les interfaces fournissent un support à la liaison retardée. Une variable peut contenir des objets de différentes classes et dans certains cas, le choix d'une méthode envoyée sur cette objet ne peut être effectué qu'à l'exécution.

Et voici un exemple complet d'utilisation d'interface tiré du support de cours de Philippe Durif de l'université de Lille (`http://www.lifl.fr/`bogaert/clfc/polyDurif/poly.html ).

```
public interface Collection {
   boolean empty ()          ;
   void    put   (Object o) ;
   Object  get   ()          ;
}
public class Pile implements Collection {
   private Object t [] = new Object [20] ;
   private int n = 0 ;

   public boolean empty()        { return n == 0 ; }
   public void    put  (Object o){ t [n++] = o ; }
   public Object  get  ()        { return t [--n] ; }

   public Object  top  ()        { return t [n-1] ; }
     // top() ne fait pas partie de l'interface
}
public class File implements Collection {
   private Object t [] ;
   private int n = 0 ;
   public File (int taille) { t = new Object [taille] ; }
```

```
   public boolean empty ()          { return n == 0 ; }
   public void     put    (Object o) { t [n++] = o ; }
   public Object   get    () {
      Object o = first () ;
      n = n-1 ;
      for (int i = 0 ; i < n ; i++) {
         t [i] = t [i+1] ;
      }
      return o ;
   }
   public Object first  ()       { return t [0] ; }
     // first() ne fait pas partie de l'interface
}
class ViderCollection {
   public static void vider (Collection coll) {
      while (! coll.empty ()) coll.get () ;
   }
}

class EssaiCollection {
   public static void main (String argv []) {
     Pile p = new Pile () ;
     File f = new File (15) ;
     Collection c = new File (20) ;
     // ...
     ViderCollection.vider (p) ;
     ViderCollection.vider (f) ;
     ViderCollection.vider (c) ;
   }
}
```

### 5.3 API

Les classes prédéfinies de JAVA sont souvent décrite par la donnée des déclarations des variables et méthodes non privées de la classe. Par déclaration, il faut entendre, le nom, le type des arguments et de la valeur de retour.

Parfois, la seule déclaration est assez explicite. Par exemple, dans l'interface d'une classe de fichiers :

```
  public void close();
```

est assez clair. En revanche :

```
  public int read(byte  b[], int  off, int  len);
```

est beaucoup moins explicite.

## 6 Compiler et exécuter un programme

Nous donnons ici les informations pour compiler et exécuter un programme en utilisant le JDK (JAVA Development Kit) de SUN.

Nous avons vu comment écrire des classes. Nous nous trouvons dans la même situation qu'un programmeur qui a écrit des fonctions et procédures, dans un langage classique. Il reste, pour exécuter quelque chose, à écrire le corps du programme. En C, on écrirait la fonction main . En

JAVA, on écrit une méthode main . Cette méthode doit être déclarée public static . Elle prend en paramètre un tableau de String et retourne void.

Exemple :

```
class Appel{
  public static main(String[] a){
    Compte martin;
    martin = new Compte();
    martin.depot(500);
    martin.retrait(100);
    martin.print();
  }
}
```

Par convention, la méthode de classe (static) main doit apparaître dans une classe ayant le même nom que le fichier qui la contient, avec l'extension .java , soit dans l'exemple ci-dessus Appel.java.

La compilation du fichier se fait avec la commande javac .

```
% javac Appel.java
```

Si tout se passe bien (aucune erreur n'est détectée, un fichier Appel.class contenant le code compilé est créé. On peut l'exécuter avec la commande java.

```
% java Appel
400
```