Chapitre 3

Programmation avancée en Java

Nous avons vu, dans le chapitre précédent, la mise en place de la programmation fondamentale (c'est-à-dire de la programmation structurée) en Java. Nous continuons dans ce chapitre avec la programmation avancée, c'est-à-dire essentiellement la programmation modulaire (les sous-programme étant des *méthodes* en programmation orientée objet), la manipulation des tableaux, des chaînes de caractères, des fichiers, la récursivité, les structures de données dynamiques et le passage d'arguments en ligne.

3.1 La programmation modulaire en Java

Nous avons déjà vu la notion et l'importance de la programmation modulaire lors de notre initiation à la programmation. Le concept de *sous-programme* est implémenté, dans les langages C et C++, sous la forme de *fonction*. Puisque Java est un langage orienté objet (presque) pur, il n'y a pas de sous-programme (ou fonction) à proprement parler, mais uniquement des **méthodes**, c'est-à-dire des fonctions au sens du langage C permettant une action sur les données d'un objet.

Une méthode pourra être publique (utilisable par l'utilisateur) ou privée. Ce dernier cas correspond bien à la notion de sous-programme proprement dit, non visible par l'utilisateur.

3.1.1 Définition d'une méthode

Introduction.- En langage C existent les notions de déclaration et de définition des fonctions. La notion de déclaration n'existe pas en Java ; seule existe la notion de définition. Les méthodes se définissent et s'utilisent essentiellement comme les fonctions du langage C.

Définition.- La syntaxe de la définition est la suivante :

```
modificateur type nom (type1 var1, ..., typen varn)
{
// corps de la méthode
}
```

comme en C++, avec quelques modificateurs de méthodes supplémentaires, éventuellement cumulables.

La définition peut se placer n'importe où à l'intérieur de la classe, avant ou après qu'elle soit appelée, cela n'a pas d'importance.

Les modificateurs de méthode.- Les **modificateurs de méthode** fixent la visibilité, les caractéristiques d'héritage, les conditions de redéfinition et les propriétés lors du multi-tâche. Les modificateurs sont :

- **public** : la méthode est accessible aux méthodes de la classe, bien sûr, mais également aux méthodes des autres classes ;
- **private** : l'usage de la méthode est réservée aux méthodes de la même classe ;
- **protected** : la méthode est héritée par les sous-classes, dans lesquelles elle est également 'protected' ; si on fait fi de l'héritage, 'protected' est la même chose que 'private' ;
- **final** : la méthode ne peut plus être modifiée (surcharge interdite) ;
- **static** : la méthode est associée à une classe et non à un objet ; nous avons déjà rencontré l'utilisation de telles méthodes, par exemple `Math.sin` (on indique le nom de la classe et non celui d'un objet) ;
- **synchronized** : nous y reviendrons lors de la mise en place du multi-tâche.

Mode de passage des arguments.- Nous avons vu qu'en langage C tous les arguments sont passés par valeur, ce qui est gênant lorsqu'on veut changer la valeur d'un argument ; il faut alors passer l'adresse de l'argument désiré et non l'argument lui-même. En langage C++ on a à la fois le passage par valeur et le passage par référence.

En Java les arguments sont passés par valeur pour les types élémentaires et par référence pour les objets. Cela signifie que l'on peut changer les données d'un objet dans une méthode.

3.1.2 Premier exemple

Écrivons un programme permettant d'entrer trois entiers puis d'afficher le maximum de ces entiers.

3.1.2.1 Cas d'une application

Nous pouvons utiliser l'application suivante.

```
import java.io.*;

class Max1
   {
   public static int maximum(int x, int y, int z)
      {
      return Math.max(x, Math.max(y,z));
      }
   public static void main(String args[]) throws IOException
      {
      InputStreamReader fich;
      BufferedReader ligne;
      String mot;
      int n1, n2, n3, m;
      fich = new InputStreamReader(System.in);
      ligne = new BufferedReader(fich);
      System.out.print("Premier entier : ");
      mot = ligne.readLine();
      n1 = Integer.parseInt(mot);
      System.out.print("Deuxi\u00E8me entier : ");
      mot = ligne.readLine();
      n2 = Integer.parseInt(mot);
      System.out.print("Troisi\u00E8me entier : ");
      mot = ligne.readLine();
      n3 = Integer.parseInt(mot);
      m = maximum(n1, n2, n3);
      System.out.println("Le plus grand est : " + m);
     }
   }
```

3.1.2.2 Cas d'une applet

Nous allons, par raison d'homogénéité, placer également le résultat dans un champ de texte en utilisant la (nouvelle) méthode `setText()`.

```
import java.applet.Applet;
import java.awt.*;

public class Max2 extends Applet
   {
   Label prompt1, prompt2, prompt3, prompt4;
   TextField entier1, entier2, entier3, result;
```

```
int a, b, c, d;

public void init()
   {
   prompt1 = new Label("Entrez le premier entier : ");
   add(prompt1);
   entier1 = new TextField("0",10);
   add(entier1);
   prompt2 = new Label("Entrez le second entier : ");
   add(prompt2);
   entier2 = new TextField("0",10);
   add(entier2);
   prompt3 = new Label("Entrez le troisi\u00E8me entier : ");
   add(prompt3);
   entier3 = new TextField("0",10);
   add(entier3);
   prompt4 = new Label("La valeur maximum est : ");
   add(prompt4);
   result = new TextField("0",10);
   add(result);
   }

public int maximum(int x, int y, int z)
   {
   return Math.max(x, Math.max(y,z));
   }

public boolean action(Event e, Object o)
   {
   a = Integer.parseInt(entier1.getText());
   b = Integer.parseInt(entier2.getText());
   c = Integer.parseInt(entier3.getText());
   d = maximum(a,b,c);
   result.setText(Integer.toString(d));
   return true;
   }
}
```

Remarques.- 1^o) Puisqu'il y a plusieurs étiquettes et plusieurs champs de texte, nous sommes obligés, pour faire référence à l'entité qui nous intéresse, d'utiliser l'opérateur point.

2^o) Nous n'avons pas encore abordé la mise en page, aussi la disposition des prompteurs et des champs de texte ne sera-t-elle pas nécessairement harmonieuse. Nous y reviendrons plus tard.

3^o) La méthode `toString()` de la classe `Integer` permet de convertir une expression de type `int` en une chaîne de caractères.

4^o) La méthode `setText()` de la classe `TextField` permet d'afficher un texte dans un champ de texte.

3.1.3 Exemple de passage par référence

Le passage par valeur suffisait pour l'exemple précédent. Considérons le cas où l'on a besoin d'un passage par référence.

Problème.- Écrire un programme Java demandant deux entiers, échangeant ces deux entiers (par l'intermédiaire d'une méthode) et affichant le résultat pour vérifier que l'échange a bien été réalisé.

Solution.- Puisque les types élémentaires sont passés par valeur, écrire une méthode :

```
void echange(int a, int b)
```

ne conviendrait pas puisque, quel que soit le corps de cette méthode, les valeurs de `a` et de `b` ne seraient pas changées dans la méthode principale.

On va donc **encapsuler** le type élémentaire dans une classe. On peut penser à utiliser la classe élémentaire `Integer` avec une nouvelle méthode :

```
void echange(Integer A, Integer B)
```

mais, malheureusement, s'il existe bien une méthode pour récupérer la donnée (du type élémentaire `int`), il n'en existe pas pour la changer.

Il existe seulement des constructeurs. Ceux-ci modifient la référence à l'objet, donc celui-ci ne sera pas modifié dans la méthode principale.

Nous allons donc créer une classe `Entier` encapsulant une donnée de type `int` et comprenant deux méthodes, permettant de modifier et de récupérer cette donnée.

Le programme.- Ceci nous conduit au programme suivant :

```
class Entier
   {
   int val;
   public void set(int a)
      {
      val = a;
      }
   public int get()
      {
      return val;
      }
   }

public class Echange
   {
   public static void swap(Entier A, Entier B)
      {
      int x;
      x = A.get();
      A.set(B.get());
      B.set(x);
      }

   public static void main(String args[])
```

```
    {
    int a, b;
    Entier A, B;
    a = 3;
    b = 5;
    System.out.println("Avant a = " + a);
    System.out.println("      b = " + b);

    A = new Entier();
    B = new Entier();
    A.set(a);
    B.set(b);
    swap(A, B);

    System.out.println("Apr\u00E8s a = " + A.get());
    System.out.println("      b = " + B.get());
    }
}
```

Remarque.- Une seule classe doit être publique, celle qui a le même nom que le nom du fichier.

3.1.4 Les méthodes prédéfinies des applets

Comme nous l'avons déjà vu, les applets sont des classes dérivées de la classe `Applet`, contenue dans le paquetage `java.applet`. Des méthodes sont déclarées pour cette classe et sont appelées par la méthode principale de la classe `Applet`, donc automatiquement lors de l'exécution d'une applet du point de vue du programmeur des applets. Cependant le corps de ces méthodes est vide ; il appartient au concepteur de l'applet de les surcharger éventuellement (ce qui n'est pas obligatoire, comme nous nous en sommes aperçus).

Méthode `void init()`.- Après chargement, la méthode `void init()` est appelée. On écrit donc dans le corps de celle-ci, en cas de besoin, les tâches d'initialisation qui doivent être menées à bien avant l'exécution de l'applet.

Méthode `void start()`.- Cette méthode est appelée automatiquement après chargement et initialisation de l'applet.

Méthode `void stop()`.- Lorsqu'on quitte une page HTML dans laquelle s'exécute une applet, le système d'exécution appelle automatiquement la méthode `void stop()`.

Méthode `void destroy()`.- Elle intervient après qu'une applet ait été arrêtée. Elle est prévue pour prendre soin de libérer les ressources accaparées par l'applet.

Méthode `void paint(Graphics g)`.- Elle est appelée à la première apparition de l'applet puis chaque fois qu'elle est ramenée à l'écran (par changement de fenêtre active). Sa fonction consiste à afficher l'applet à l'écran.

On ne peut pas faire appel directement à cette méthode. On peut cependant utiliser la méthode `repaint();` pour faire appel à elle, comme nous l'avons déjà vu.

3.2 Les tableaux

Les tableaux sont implémentés essentiellement comme en langage C, à la différence près que ce sont des objets. Ceci implique qu'on doit absolument les instantier.

3.2.1 Mise en place des tableaux à une dimension

Nom d'un tableau.- Tout tableau a un nom, qui est un identificateur non utilisé pour autre chose.

Déclaration.- La déclaration se fait de l'une des deux façons suivantes :

```
type[] nom;
```

ou :

```
type nom[];
```

les crochets précédant ou suivant le nom du tableau au choix de l'utilisateur.

On remarquera qu'aucune dimension n'est donnée à ce moment. La première façon est évidemment inspirée du langage C, la seconde est plus logique : le type du tableau est `type []`.

Instantiation.- Le tableau doit être instantié. C'est à ce moment l'on réserve de l'emplacement mémoire en indiquant sa dimension maximum :

```
nom = new type[entier];
```

où `entier` est une expression entière (à valeur positive), qui n'est pas nécessairement une constante (contrairement à ce qui se passe en langage C ou C++).

Remarque.- Il n'existe pas d'instruction `delete` correspondant à l'instruction `new`. Il n'y a pas de libération explicite en Java. La libération est faite automatiquement à la fin du bloc par le **récupérateur de mémoire** (ou **ramasse-miettes**, *garbage collector* en anglais, ou **GC** ; on utilise aussi l'expression *glaneur de cellules* en français pour pouvoir disposer du même acronyme).

Accès à un élément.- Comme en langage C, les éléments sont repérés par un indice variant de 0 à longueur moins un. Pour désigner l'élément d'indice i, on utilise l'expression :

```
nom[i]
```

Il y a cependant une différence essentielle entre Java et les langages C/C++. En Java, le système vérifie que l'indice se trouve bien dans l'intervalle voulu ; s'il n'en est pas ainsi, une erreur est déclenchée, plus exactement une exception est levée.

Longueur d'un tableau.- Le tableau est un objet, comme nous l'avons déjà dit. Un de ses membres données permet, à tout instant, d'en connaître sa longueur. Il suffit d'utiliser l'expression :

`nom.length`.

Initialisation lors de la déclaration.- Comme en langage C, on peut initialiser un tableau lors de sa déclaration :

```
int tab[] = {23, 45, 67, 12, 89 25};
```

Dans ce cas, il ne faut évidemment pas l'instantier, sinon on détruit l'initialisation. L'initialisation lui attribue automatiquement une longueur.

3.2.2 Exemples

3.2.2.1 Exemple d'applet

Initialisons un tableau dont nous faisons afficher les valeurs.

```
import java.applet.Applet;
import java.awt.*;

public class tableau extends Applet
   {
   int tab[] = {23, 45, 67, 83, 25, 24};

   public void paint(Graphics g)
      {
      int i, y;
      y = 25; // numero de ligne en pixels
      g.drawString("Indice", 25, y);
      g.drawstring("Valeur", 100, y);
      for (i = 0; i < tab.length; i++)
         {
         y = y + 15;
         g.drawString("" + i, 25, y);
         g.drawString("" + tab[i], 100, y);
         }
      }
   }
```

3.2.2.2 Exemple d'application

Écrivons une application Java permettant de demander le nombre d'étudiants d'une promotion donnée, d'entrer les notes des étudiants à un partiel et d'afficher la moyenne de la promotion et le nombre de notes supérieures à la moyenne de la promotion.

```
import java.io.*;

public class note
   {
   public static void main(String args []) throws IOException
      {
      InputStreamReader fich;
      BufferedReader ligne;
      String mot;
      Integer N;
      int i, n, sup;
      Double X;
      double m;
      double[] note;
      // Saisie du nombre d'etudiants
      System.out.print("Nombre d'etudiants : ");
      fich = new InputStreamReader(System.in);
      ligne = new BufferedReader(fich);
```

```
      mot = ligne.readLine();
      N = Integer.valueOf(mot);
      n = N.intValue();
      // Creation du tableau
      note = new double[n];
      // Saisie des notes
      m = 0;
      for (i=0 ; i < n ; i++)
         {
         System.out.print("note numero " + (i+1) + " : ");
         mot = ligne.readLine();
         X = Double.valueOf(mot);
         note[i] = X.doubleValue();
         m = m + note[i];
         }
      // Calcul de la moyenne
      m = m/n;
      System.out.println("La moyenne est de : " + m);
      // Calcul du nombre de notes superieures a la moyenne
      sup = 0;
      for (i = 0 ; i < n ; i++)
         if (note[i] >= m) sup++;
      System.out.println("Le nombre de notes superieures a la");
      System.out.println("moyenne de la classe est de : " + sup);
      }
   }
```

3.2.3 Mise en place des tableaux à plusieurs dimensions

Principe général.- Comme en C/C++, ce sont des tableaux de tableaux. On a donc, par exemple :

```
int tab[][];
tab = new int [3][5];
```

Initialisation lors de la déclaration.- Elle existe, comme en C/C++ :

```
int tab[][] = {{ 1, 2 }, {3, 4 }};
```

mais elle est à éviter, pour des raisons de lisibilité.

Tableaux à nombre de colonnes variables.- L'instantiation nous permet d'affecter un nombre de colonnes dépendant de la ligne que l'on considère :

```
int b[][];
b = new int [2][];
b[0] = new int [5];
b[1] = new int [3];
```

ou encore :

```
int b[][] = {{ 1, 2}, { 3, 4, 5 }};
```

3.2.4 Passage des tableaux en paramètre

3.2.4.1 Passage par référence

Les tableaux étant des objets, ils sont toujours passés par référence. Vérifions-le avec le programme de tri suivant. On a, par exemple, dans le cas d'une applet :

```
import java.applet.Applet;
import java.awt.*;

public class tableau extends Applet
   {
   int tab[] = {23, 45, 67, 83, 25, 24, 11, 53};

   public void tri(int a[])
      {
      int i, j, tmp;
      for (i = a.length - 1; i > 0; i--)
      for (j = 0; j < i; j++)
         if (a[j] > a[j+1])
            {
            tmp = a[j];
            a[j] = a[j+1];
            a[j+1] = tmp;
            }
      }

   public void paint(Graphics g)
      {
      int i, y;
      y = 25; // numero de ligne en pixels
      g.drawString("Indice", 25, y);
      g.drawstring("Tableau", 100, y);
      g.drawString("Tableau trie", 175, y);
      for (i = 0; i < tab.length; i++)
         {
         y = y + 15;
         g.drawString("" + i, 25, y);
         g.drawString("" + tab[i], 100, y);
         }
      tri(tab);
      y = 25;
      for (i = 0; i < tab.length; i++)
         {
         y = y + 15;
         g.drawString("" + tab[i], 175, y);
         }
      }
   }
```

Remarques.- 1^o) Il est inutile de spécifier la dimension effective du tableau dans l'en-tête de la

méthode de tri puisqu'on peut retrouver celle-ci grâce à un attribut de l'objet.

2^o) Faites attention ! : en redimensionnant la fenêtre de l'applet, la méthode `paint()` est appelée à nouveau et donc le premier tableau apparaîtra comme déjà trié.

3.2.4.2 Application au passage par référence des entités

On peut utiliser un tableau pour passer des entités (qui ne sont pas des objets mais d'un type élémentaire) par référence. Il suffit de les encapsuler dans un tableau.

3.3 Les chaînes de caractères

Contrairement à ce qui se passe en langage C ou en C++, les chaînes de caractères ne sont pas des tableaux en Java mais des objets de la classe `String`, faisant partie du paquetage de base `java.lang`.

3.3.1 Méthodes principales de la classe String

Introduction. Nous n'avons pas à savoir comment est implémentée cette classe, mais seulement quelles sont les méthodes existant pour manipuler les objets de cette classe.

Déclaration.- Nous avons déjà vu comment déclarer un objet d'une classe, et donc de la classe `String`. Il n'y a rien de particulier :

```
String Prenom;
```

Instantiation.- Un objet de la classe `String` est instantié par défaut (au mot vide), contrairement à ce qui se passe pour toutes les autres classes. On peut aussi l'instantier explicitement :

```
String Prenom;
Prenom = new String(); // sans intérêt
```

ou :

```
Prenom = new String(″Paul″);
```

Initialisation lors de la déclaration.- On peut initialiser une chaîne de caractères lors de sa déclaration :

```
String Prenom = ″Paul″;
```

Affectation.- Contrairement à ce qui se passe en langage C ou en C++, il existe une affectation des chaînes de caractères en Java, utilisant le symbole habituel '=' :

```
Prenom = ″Paul″;
Prenom = Prenom1;
```

Concaténation.- Comme nous l'avons déjà vu, la concaténation est implémentée, en utilisant le symbole '+' :

```
Prenom = ″Bonjour ″ + Prenom;
```

Comme nous l'avons déjà vu également, si on concatène une chaîne de caractères avec un objet qui n'est pas une chaîne de caractères (comme, par exemple, un nombre), cet objet est converti en une chaîne de caractères. Pour toute classe (tout au moins celles implémentées par Sun), il existe une méthode permettant de convertir un objet de cette classe en une chaîne de caractères.

Saisie et affichage.- Nous avons déjà vu comment afficher une chaîne de caractères, que ce soit dans une application ou dans une applet. C'est relativement simple. Nous avons également vu comment saisir une chaîne de caractères au clavier : c'est moins simple.

Longueur.- Comme pour un tableau, l'attribut `length` donne la longueur d'une chaîne de caractères :

```
String Prenom = ″Paul″;
int n;
n = Prenom.length;
```

Comparaison.- La méthode :

```
boolean equals(String)
```

de la classe `String` permet de tester si une chaîne de caractères est égale à une autre :

```
if (Prenom.equals(″Paul″)) - - - ;
```

La méthode :

```
int compareTo(String);
```

de la classe `String` est l'analogue de la fonction `strcmp()` du langage C : elle renvoie un entier négatif si la chaîne de caractères vient (strictement) avant l'argument dans l'ordre lexicographique, l'entier nul si elles sont égales et un entier strictement positif si la chaîne de caractères vient (strictement) après l'argument dans l'ordre lexicographique.

Conversion.- Nous avons déjà vu comment convertir une chaîne de caractères en un objet numérique, et vice versa.

3.3.2 Un exemple

Écrire une application Java qui demande un prénom et répond par 'Bien le bonjour Paul' s'il s'agit de Paul, et sinon par 'Bonjour' suivi du prénom.

```
import java.io.*;

public class paul
   {
   public static void main(String args[]) throws IOException
      {
      InputStreamReader fich;
      BufferedReader ligne;
      String Prenom;
      fich = new InputStreamReader(System.in);
      ligne = new BufferedReader(fich);
      System.out.print("Quel est votre prenom : ");
      Prenom = ligne.readLine();
      if (Prenom.equals("Paul"))
         System.out.println("Bien le bonjour Paul");
      else
         System.out.println("Bonjour " + Prenom);
         }
   }
```

3.4 Les fichiers

Pour des raisons de sécurité, on ne peut pas lire ou écrire dans un fichier local via une applet. Nous allons donc nous intéresser uniquement aux applications, et pour un kit postérieur au JDK 1.1.

3.4.1 Lecture dans un fichier

3.4.1.1 Manipulations

Introduction.- Java ne fait pas la différence entre fichier texte et fichier binaire : il n'y a que des fichiers de caractères. On ne peut accéder aux fichiers que de façon séquentielle.

Nom physique.- La syntaxe du nom physique d'un fichier dépend du système d'exploitation. Il sera récupéré comme chaîne de caractères.

Nom logique.- La classe des fichiers en lecture est `FileInputStream`, faisant partie du paquetage `java.io` (qu'il faut donc importer).

Assignation.- Le lien entre le fichier physique et le fichier logique se fait au moment de l'instantiation du fichier logique, de façon naturelle :

```
nomlogique = new FileInputStream(nomphysique);
```

Flot d'entrée et tampon.- Le nom logique correspond au nom '`System.in`' du clavier. Comme nous l'avons vu dans le cas de la saisie au clavier, nous avons besoin d'un flot d'entrée, de la classe `InputStreamReader`, et d'un tampon, de la classe `BufferedReader`. Les assignations correspondantes se font comme dans le cas d'une saisie au clavier.

Lecture proprement dite.- Dans le cas d'une lecture ligne à ligne, on peut utiliser la méthode :

```
String readLine()
```

de la classe `BufferedReader`, déjà rencontrée.

Détection de fin de fichier.- Lorsqu'on est arrivé à la fin du fichier, la méthode `readLine()` renvoie la valeur `null`.

Fermeture du fichier.- On ferme le fichier en utilisant la méthode `close()` de la classe `FileInputStream`.

3.4.1.2 Exemple

Écrivons une application Java demandant le nom (physique) d'un fichier (texte) et affichant le contenu de ce fichier à l'écran.

```
import java.io.*;

public class affiche
   {
   public static void main(String args[]) throws IOException
      {
      InputStreamReader fich;
```

```
      BufferedReader clavier;
      String nomphysique;
      fich = new InputStreamReader(System.in);
      clavier = new BufferedReader(fich);
      System.out.print("Nom du fichier texte a afficher : ");
      nomphysique = clavier.readLine();

      FileInputStream nomlogique;
      InputStreamReader lecturefichier;
      BufferedReader fichier;
      String ligne;
      nomlogique = new FileInputStream(nomphysique);
      lecturefichier = new InputStreamReader(nomlogique);
      fichier = new BufferedReader(lecturefichier);
      ligne = fichier.readLine();
      while (ligne != null)
         {
         System.out.println(ligne);
         ligne = fichier.readLine();
         }
      nomlogique.close();
      }
   }
```

Exercice.- *Réécrire cette application de façon à avoir une pause toutes les 23 lignes.*

3.4.2 Écriture dans un fichier

3.4.2.1 Manipulations

Introduction.- Les manipulations pour l'écriture sont pratiquement les mêmes que pour la lecture.

Nom logique.- La classe des fichiers en écriture est `FileOutputStream`, faisant également partie du paquetage `java.io`.

Flot de sortie.- Nous avons besoin d'un flot de sortie, de la classe `PrintStream`. Par contre, nous n'avons pas besoin de tampon.

Écriture proprement dite.- L'écriture se fait grâce aux méthodes de la classe `PrintStream` déjà vues, `print()` et `println()`.

Fermeture.- On ferme le fichier grâce à la méthode `close()`, appliquée à l'objet de la classe `FileOutputStream`.

3.4.2.2 Exemple

Écrivons une application Java qui demande le nom (physique) d'un fichier (texte existant), le nom d'un fichier (texte à créer) et qui copie le contenu du premier fichier dans le second.

Nous allons recopier ligne par ligne. N'oublions pas que le passage à la ligne n'est pas pris en compte lors de la lecture ; il faut donc le générer.

```
import java.io.*;

public class copie
   {
   public static void main(String args[]) throws IOException
      {
      InputStreamReader fich;
      BufferedReader clavier;
      String nomphysique;
      fich = new InputStreamReader(System.in);
      clavier = new BufferedReader(fich);

      System.out.print("Nom du fichier texte a copier : ");
      nomphysique = clavier.readLine();
      FileInputStream nomlogique1;
      InputStreamReader lecturefichier;
      BufferedReader fichier;
      String ligne;
      nomlogique1 = new FileInputStream(nomphysique);
      lecturefichier = new InputStreamReader(nomlogique1);
      fichier = new BufferedReader(lecturefichier);

      System.out.print("Nom du fichier texte a creer : ");
      nomphysique = clavier.readLine();
      FileOutputStream nomlogique2;
      PrintStream ecriturefichier;
      nomlogique2 = new FileOutputStream(nomphysique);
      ecriturefichier = new PrintStream(nomlogique2);

      ligne = fichier.readLine();
      while (ligne != null)
         {
         ecriturefichier.println(ligne);
         ligne = fichier.readLine();
         }
      nomlogique1.close();
      nomlogique2.close();
      }
   }
```

3.5 Récursivité

Introduction.- Nous avons vu la notion de récursivité lors de l'initiation (avancée) à la programmation. La mise en place de la récursivité en Java se fait comme dans les langages C et C++, c'est-à-dire en définissant une méthode récursivement, dans le cas de récursivité directe.

Exemple.- Écrivons une application Java demandant un entier n et affichant la valeur F_n du n-ième terme de la suite de Fibonacci, définie, rappelons-le, de la façon suivante :

$$F_0 = F_1 = 1\,;$$
$$F_{n+2} = F_{n+1} + F_n.$$

Nous avons, par exemple, l'application suivante :

```
import java.io.*;

class fibonacci
   {
   public static int fibo(int n)
      {
      if ((n == 0) || (n == 1)) return 1;
      else return fibo(n-1) + fibo(n-2);
      }

   public static void main(String args[]) throws IOException
      {
      InputStreamReader fich;
      BufferedReader ligne;
      String mot;
      Integer N;
      int n;
      fich = new InputStreamReader(System.in);
      ligne = new BufferedReader(fich);
      System.out.print("n = ");
      mot = ligne.readLine();
      N = Integer.valueOf(mot);
      n = N.intValue();
      System.out.println("F(n) = " + fibo(n));
      }
   }
```

On remarquera que l'exécution est très lente.

3.6 Structures de données dynamiques

Introduction.- La mise en place des structures de données dynamiques se fait, comme dans le langage C++, grâce à des **classes auto-référentes**. Il y a cependant une différence : il n'y a pas d'utilisation explicite des pointeurs en Java ; mais, puisque toute référence à un objet se fait par référence, c'est comme si nous avions un pointeur sur cet objet.

Exemple 1.- Reprenons l'exemple classique d'une liste chaînée. Nous allons construire une liste chaînée d'entiers et l'afficher dans l'ordre inverse. Nous avons intérêt à définir une classe élément de la liste chaînée, disons `Sommet`, avant de définir la classe `List` proprement dite.

```
class Sommet
   {
   int n;
   Sommet suivant;
```

```
   public Sommet(int nn)
      {
      n = nn;
      suivant = null;
      }
   public Sommet(int nn, sommet next)
      {
      n = nn;
      suivant = next;
      }
   public int valeur()
      {
      return n;
      }
   public Sommet suiv()
      {
      return suivant;
      }
   }
class List
   {
   Sommet premier;

   public list()
      {
      premier = null;
      }
   public boolean vide()
      {
      if (premier == null) return true;
      else return false;
      }
   public void insert(int nn)
      {
      if (vide()) premier = new Sommet(nn);
      else premier = new Sommet(nn, premier);
      }
   public void affiche()
      {
      System.out.print("{");
      Sommet index = premier;
      while (index != null)
         {
         System.out.print(index.valeur() + " ,");
         index = index.suiv();
         }
      System.out.println("}");
      }
   }
```

```
class testlist
   {
   public static void main(String args[])
      {
      List liste;
      liste = new List();
      liste.insert(3);
      liste.insert(9);
      liste.insert(5);
      liste.affiche();
      }
   }
```

Exemple 2.- (**Arbre binaire de tri**)

Un **arbre binaire** *est une structure de donnée dynamique définie récursivement : on part d'un nœud appelé* **racine**, *chaque nœud comprend une donnée, un* **fils gauche** *et un* **fils droit**, *chacun étant vide ou étant lui-même un nœud. Un nœud sans fils est appelé une feuille.*

- 1^o) Définir les classes `noeud` des éléments d'un arbre binaire d'entiers naturels et `arbre` en Java.

La classe `noeud` comprendra trois données (un entier, qui est sa valeur, et deux nœuds : le fils gauche et le fils droit) et un constructeur ayant un argument entier permettant de créer un nœud feuille dont la valeur est cet entier.

La classe `arbre` comprendra une donnée privée, la racine, qui est un nœud, et le constructeur par défaut qui construit un arbre vide (sans nœud).

[On ne peut pas faire en faire grand chose pour l'instant, mais on va améliorer ces classes.]

Dans un **arbre binaire de tri**, *on insère les éléments récursivement de la façon suivante : on a une liste d'entiers (non triée) sans doublon ; on va placer un nœud à l'arbre par entier ; le premier entier est l'attribut valeur de la racine ; pour chaque entier suivant, si l'entier est strictement inférieur à la valeur de la racine, un nœud est ajouté à gauche à la bonne place, s'il est supérieur à la racine, il est ajouté à droite. Par exemple si on a l'arbre :*

```
            5
     3              19

1        4       10      23
```

et que l'on veut ajouter 12, on obtient l'arbre :

```
            5
     3              19

1        4       10      23

                 12
```

- 2^o) a) Ajouter la méthode `inserer` (un entier) à la classe `noeud` ayant un argument entier d : si d est strictement inférieur à la valeur du nœud et si le fils gauche est nul, on remplace ce

fils gauche par une feuille de valeur d, si le fils gauche est non nul, on insère (récursivement) la valeur d à ce fils gauche ; si d est strictement supérieur à la valeur du nœud, on a un traitement analogue pour le fils droit.

b) Ajouter la méthode `inserer` (un nœud) à la classe `Arbre` dont l'argument est un entier : si la racine est nulle, la racine devient une feuille dont la valeur est cet entier ; sinon on insère cette valeur entière à la racine.

La **traversée en ordre** *d'un arbre binaire consiste à afficher les valeurs de ses nœuds récursivement de la façon suivante : si non vide traversée en ordre du fils gauche, racine, traversée en ordre du fils droit.*

On voit que, pour un arbre binaire de tri, on affiche ainsi les éléments dans l'ordre.

- 3^o) Ajouter une méthode `traversee` à la classe `Arbre`, qui affiche les éléments sur une ligne. Cette méthode fait appel à une méthode auxiliaire privée d'argument un nœud, qu'on appellera `assistant`.

- 4^o) Écrire une classe de test qui insère les valeurs 34, 23, 67, 12, 56, 3 et 5 à un arbre de tri et qui traverse cet arbre, c'est-à-dire qui trie la liste d'entiers ci-dessus.

[On a :

```
/* TestArbre.java */

class noeud
{
    noeud gauche;
    int val;
    noeud droit;

    public noeud(int d)
    {
    gauche = null;
    val = d;
    droit = null;
    }

    public void inserer(int d)
    {
    if (d < val)
        {
        if (gauche == null) gauche = new noeud(d);
        else gauche.inserer(d);
        }
    if (d > val)
        {
        if (droit == null) droit = new noeud(d);
        else droit.inserer(d);
        }
    }
}
```

```
class arbre
{
    private noeud racine;

    public arbre()
    {
    racine = null;
    }

    public void inserer(int d)
    {
    if (racine == null) racine = new noeud(d);
    else racine.inserer(d);
    }

    public void traversee()
    {
    assistant(racine);
    System.out.println();
    }

    private void assistant(noeud n)
    {
    if (n != null)
        {
        assistant(n.gauche);
        System.out.print(n.val + " ");
        assistant(n.droit);
       }
    }
}

public class TestArbre
{
    public static void main(String args [])
    {
    arbre t;
    t = new arbre();
    t.inserer(34);
    t.inserer(23);
    t.inserer(67);
    t.inserer(12);
    t.inserer(56);
    t.inserer(3);
    t.inserer(5);
    t.traversee();
    }
}
```

3.7 Passage d'arguments en ligne

Introduction.- On peut passer des arguments lorsqu'on lance une application Java. Ces arguments sont récupérés dans le tableau de String `args`, qui ne nous a pas encore servi jusqu'à maintenant. Le premier argument est `args[0]` et ainsi de suite.

Exemple.- Écrivons une application Java `More` effectuant la même chose que l'utilitaire `more` de UNIX, c'est-à-dire qu'il prend en argument le nom d'un fichier (texte) et qu'il l'affiche à l'écran en faisant une pause toutes les 23 lignes.

```
import java.io.*;

public class More
   {
   public static void main(String args[]) throws IOException
      {
      FileInputStream nomlogique;
      InputStreamReader lecturefichier;
      BufferedReader clavier, fichier;
      clavier = new BufferedReader(
                new InputStreamReader(
                System.in));
      String ligne;
      int i = 0;
      nomlogique = new FileInputStream(args[0]);
      lecturefichier = new InputStreamReader(nomlogique);
      fichier = new BufferedReader(lecturefichier);
      ligne = fichier.readLine();
      while (ligne != null)
         {
         i++;
         System.out.println(ligne);
         if (i % 23 == 0)
            {
            System.out.println("--- More ---");
            clavier.readLine();
            }
         ligne = fichier.readLine();
         }
      nomlogique.close();
      }
   }
```

Puisque Java est interprété, et non compilé, pour faire afficher, par exemple, le source de notre programme, on écrira :

```
java More More.java
```