Chapitre 3

Java, langage de développement objet

3.1 Présentation de Java

3.1.1 Objectifs

Le langage Java a été développé afin de pouvoir générer des applications qui soient indépendantes des machines et de leur système d'exploitation. Une autre caractéristique du langage est de pouvoir écrire des applications structurellement distribuées sur des réseaux. Il appartient, par ailleurs, à la famille des langages objets purs où rien ne peut exister en dehors des classes.

3.1.2 Historique

Au début des années 1990, une équipe de développeurs de la société SUN Microsystems travaille sur l'implémentation du langage OAK pour l'intégrer en domotique, notamment pour le développement de la télévision interactive. Il fallait que les codes des applications dans ce langage soit peu volumineux, efficaces et indépendants de l'architecture. A cette époque, la télévision interactive n'a pas connue l'essor escompté alors que le développement d'Internet et du Web faisait apparaître des besoins urgents de même nature. En 1995, OAK devient Java et se trouve popularisé rapidement pour ses possibilités de développement liés au Web. Les années qui suivent font connaître à Java une popularité en tant que langage généraliste qui dépasse les prévisions de SUN. Les grands industriels du développement l'adopte rapidement en l'espace de quelques années. Pour les années à venir, les industriels projettent la création de puces électroniques dédiées à Java ... un juste retour vers les préoccupations initiales des concepteurs originaux de OAK/Java.

Certes Java est un langage propriétaire, mais de licence ouverte, ce qui lui a permis de se faire adopter par tous les professionnels du développement avec un engouement sans précédent par rapport aux autres langages plus anciens.

3.1.3 Une machine virtuelle

Le langage Java est un langage compilé et interprété. Cette définition contradictoire s'explique par le fait que le code source est transformé dans un byte-code universel exécutable par une machine virtuelle. Cette machine virtuelle peut être installée à partir d'une distribution du

langage comme le Java Development Kit (JDK) de SUN. Elle peut être également intégrée sous une forme compactée dans un navigateur afin qu'il puisse exécuter des applets Java.

Ces caractéristiques confèrent au langage des propriétés de portabilité sans précédents, mais ont aussi un coût correspondant au fonctionnement de la machine virtuelle qui réduit les performances d'exécution du programme. Afin de résoudre ce problème, un effort important est consenti pour développer des compilateurs à la volée en code machine qui fonctionnent lors de l'exécution du programme (compilateurs identifiés sous la dénomination de JIT - Just In Time).

3.1.4 Caractéristiques

Le langage Java possède une syntaxe inspirée du C++. Il se veut plus propre en terme de développement objet, ne permettant pas de construction en dehors des classes. Il se veut aussi plus simple en affranchissant le programmeur de toute la gestion dynamique des objets construits, grâce au fonctionnement d'un ramasse-miettes (Garbage Collector) dont la fonction est d'identifier et d'éliminer tous les objets qui ne sont plus référencés.

Java propose des exécutions parallèles de programmes grâce à une utilisation qui se veut plus facile des *threads* (ou processus légers) par rapport au langage C. Il possède également des aspects liés à la distribution grâce à ses possibilités d'intégration dans des documents Web distribués par les applets, mais également avec la bibliothèque RMI (Remote Methods Invocation) qui propose de la programmation d'objets répartis. L'évolution de cette bibliothèque la fait converger progressivement vers CORBA, le standard dans le domaine des objets répartis. Les bibliothèques CORBA sont aujourd'hui intégrées dans les distributions du JDK 1.2 ou supérieures de SUN.

3.2 Types primitifs et structures de contrôle

3.2.1 Types primitifs

Comme tout langage de programmation évolué, Java possède un certain nombre de types de données de base :
– le type `boolean` qui prend une des 2 valeurs `true` ou `false` sur 1 octet ;
– le type `char` qui correspond à un caractère sur 2 octets ;
– les types `byte, short, int` et `long` qui sont 4 types d'entiers stockés respectivement sur 1, 2, 4 et 8 octets ;
– les types `float` et `double` qui sont 2 types de flottants stockés respectivement sur 4 et 8 octets.
Les déclarations de telles variables peuvent se faire n'importe où dans le code mais avant leur utilisation, comme en C++.

On utilise les opérateurs usuels sur les variables de ces types avec la syntaxe du C.

3.2.2 Structures de contrôle

Ce sont essentiellement les mêmes constructions qu'en C, à savoir

– L'affectation qui se fait par l'opérateur = et qui consiste à recopier la valeur de la variable primaire à droite du symbole dans la variable située à gauche.
– Les structures conditionnelles :
  – `if (cond) instruction1; [else instruction2;]`
  – `switch (selecteur) {`
    `case c1 : instructions1;`
    `...`
    `case cn : instructionsN;`
    `default : instructionsNP;`
    }
– Les structures répétitives ou boucles :
  – `for (initialisation; condition; instructionDeSuite)`,
  – `while (condition) instruction;`
  – `do instruction; while (condition)`.

Attention : Les types primitifs sont passés par valeur lors des appels de méthode (ce qui n'est pas le cas pour les objets, comme on le verra plus tard).

3.3 Classes et objets en Java

3.3.1 Définition d'une classe

Une classe permet de définir un type d'objets associant des données et des opérations sur celles-ci appelées *méthodes*. Les données et les *méthodes* sont appelés *composants* de la classe.

Exemple : Robot se déplaçant sur une grille 2D
– données :
  – X, Y, orientation
– méthodes :
  – Initialisations
  – Avancer
  – TournerADroite

```
class Robot {
        // quelques constantes
        public static final int Nord = 1;
        public static final int Est = 2;
        public static final int Sud = 3;
        public static final int Ouest = 4;

        // données
        public int X;
        public int Y;
        public int orientation ;

        // constructeurs
        public Robot (int x, int y, int o)
        { X=x; Y=y; orientation=o; }
```

```
// autre constructeur
public Robot ()
{ X=0; Y=0; orientation=Nord; }

// methodes
public void avancer ()
{ switch (orientation)
  { case Nord : Y=Y+1; break;
         case Est : X=X+1; break;
    case Sud : Y=Y-1; break;
         case Ouest : X=X-1; break;
  };
}
public void tournerADroite ()
{ switch (orientation)
  { case Nord : orientation=Est; break;
    case Est : orientation=Sud; break;
    case Sud : orientation=Ouest; break;
    case Ouest : orientation=Nord; break;
  };
}
```

Un autre exemple qui sera développer plus loin permettra de définir une classe `Vecteur` représentant des vecteurs au sens mathématique. A chaque vecteur, seront rattachés :
– une structure de données - un tableau - pour le stockage de ses coefficients ;
– des traitements comme les suivants,
  – son addition avec un autre vecteur,
  – son produit scalaire avec un autre vecteur,
  – ...
  – et aussi son procédé de création.

3.3.2 Déclaration, création et destruction d'objets

Pour manipuler un objet (par exemple, un vecteur particulier de la classe `Robot`), on doit tout d'abord déclarer une *référence* sur la classe correspondant au type d'objet. Cette opération permet de réserver en mémoire une adresse qui référencera l'objet. Par exemple, pour déclarer un objet de type `Robot` on écrira :

```
Robot totor;
```

Pour que cet objet soit réellement construit, c'est à dire que la référence désigne un emplacement à partir duquel on pourra accéder aux caractéristiques de l'objet, on devra appeler l'opération `new` qui alloue cet emplacement en mémoire et le renvoie à la référence de l'objet. On dispose alors d'une nouvelle *instance* de l'objet. Par exemple, pour la construction effective de notre robot totor, on écrira :

```
totor = new Robot(5, 12, Sud);
```

Cette instruction fait appel à un procédé de construction d'objets, appelé *constructeur*, qui est défini par défaut, mais qui peut aussi être redéfini comme opération dans la classe `Robot`. Si cette instruction de construction n'est pas effectuée, la référence associée à `x` est initialisée par défaut à `NULL`, qui est une adresse fictive ne pointant vers aucun emplacement mémoire réel.

Par ailleurs, si l'on a besoin de référencer l'objet courant dans la définition de la classe, cette référence se fait par le mot réservé `this` qui vaut donc l'adresse de l'objet courant dans une *instance* de la classe.

La destruction des objets est pris en charge par le *Garbage Collector*, appelé encore *ramasse-miettes* dans sa dénomination francisée. Cet outil logiciel fonctionne en même temps que le programme, il recherche, identifie et supprime de la mémoire les objets qui ne sont plus référençables.

On peut toutefois ajouter à chaque classe un service `finalize()`, qui sera appelé au moment de la destruction de l'objet, s'il est utile d'effectuer des opérations spécifiques à cet instant. Par exemple, dans la classe Robot, on pourrait trouver :

```
class Robot { ...
   void finalize() {Systeme.out.println(‘‘fin’’);}
}
```

Comme nous l'avons indiqué précédemment, le nom d'un objet permet de définir une *référence*, c'est à dire une adresse. Il est alors possible de faire une affectation entre deux objets de même nature. Une telle opération va donc recopier l'adresse de l'objet affecté. Ainsi une affectation est notablement différente lorsqu'elle se fait entre des variables de type simple (int, char, double, ...) ou entre des objets.

Le passage de paramètres dans les fonctions se fait par valeur, comme en C. Ainsi, soit le paramètre est de type simple alors on recopie sa valeur dans celui de la fonction appelée, soit c'est un objet, on recopie alors l'adresse référencée par l'objet (qui est bien sa valeur) dans celui de la fonction appelée.

3.3.3 Utilisation des objets, implémentation des méthodes, références

On accède aux différents composants par la notation pointée :
– `totor.X;`
– `totor.avancer();`
En Java, les implémentations des méthodes sont rédigées à l'intérieur de la définition de la classe.
– boolean superieur (float x, float y) return x¿y ;
Par ailleurs, l'affectation correspond à recopier la référence.

```
UneClasse objet1 = new UneClasse();
UneClasse objet2 = objet1;
-> objet1 et objet2 correspondent à la même référence
```

Pour créer une copie, on utilise la méthode clone, prédéfinie dans chaque objet par défaut :

```
UneClasse objet3 = objet1.clone()
```

Illustration des références et des comparaisons :

```
totor = new Robot(); rotot = totor; d2r2 = new Robot();
if (totor == d2r2) // faux
if (rotot == totor) // vrai
```

3.3.4 Tableaux

Un tableau va permettre de stocker un certain nombre d'éléments de même type dans une structure de données qui dispose d'un index pour y accéder. Un tableau en Java est un objet à part entière.
Par exemple, un tableau monodimensionnel de flottants de type `double` sera déclaré de la manière suivante :

```
double monTableau[];
```

ou encore

```
double[] monTableau;
```

Comme nous l'avons expliqué précédemment, cette déclaration a permis d'affecter une référence au nom du tableau une référence (c'est à dire une adresse, initialisée par défaut à `NULL`). Pour construire effectivement le tableau, c'est à dire disposer de plusieurs emplacements mémoire, par exemple 10, qui lui sont propres, on invoque l'opération de construction d'objets suivante :

```
monTableau = new double[10];
```

Ces deux instructions de constructions peuvent s'écrire en une seule fois :

```
double[] monTableau = new double[10];
```

Il est également possible d'initialiser la construction en affectant un tableau constant de la manière suivante :

```
double[] montableau = {0.0, 1.1, 3.5};
```

Les tableaux en tant qu'objets proposent un certain nombre d'opérations, notamment ils possèdent une donnée propre `length` qui renvoie la taille du tableau.
Par exemple, les instructions suivantes permettent d'afficher le contenu du tableau précédent, en utilisant la méthode[1] d'affichage standard sur l'écran `System.out.print`[2] :

```
for (int i=0; i<montableau.length; i++)
    System.out.print(montableau[i]+" ");
```

L'exemple précédent nous apprend par ailleurs que, comme en C, le premier indice d'un tableau est `0`.
On dispose aussi de mécanismes de vérification de dépassement de bornes des tableaux que l'on peut interroger par l'intermédiaire des techniques d'*exceptions*[3] que l'on décrira après. L'exception concernée se nomme `ArrayIndexOutOfBoundsException`.

[1] voir 3.3.1
[2] voir 3.8 pour une description générale des méthodes d'entrées/sorties
[3] voir 3.7

3.3.5 Construction de la classe vecteur

Nous allons contruire maintenant une classe `Vecteur` permettant de manipuler des vecteurs au sens mathématique. C'est une classe élémentaire allégée de certains concepts de programmation objet qui seront présentés et introduits progressivement dans la suite de cet ouvrage.
Dans cette classe, on utilise une structure de données, appelée `composant`, correspondant à un tableau où seront stockés les coefficients du vecteur.
On définit deux *constructeurs* qui sont des méthodes portant le nom de la classe et qui ne renvoie pas de résultat :
– Le premier constructeur construit un vecteur dont le nombre de composants est donné en paramètre.
– Le second constructeur construit un vecteur en recopiant un tableau passé en paramètre.
Par l'intermédiaire de ces deux *constructeurs*, il est ainsi possible de définir des méthodes de même nom, à condition qu'elles diffèrent au niveau du type ou du nombre de paramètres ou de leur résultat renvoyé. On parle alors de *surcharge* de méthodes.
On définit différentes méthodes :
– `elt` renvoyant la valeur de sa composante dont l'indice est donné en paramètre ;
– `toElt` permettant de modifier la composante dont l'indice et la nouvelle valeur sont passés en paramètres ;
– `dim` renvoyant la taille du vecteur ;
– `afficher` affichant les valeurs de ses composantes ;
– `add` renvoyant un vecteur qui est la somme du vecteur courant avec le vecteur passé en paramètre ;
– `prodScalaire` renvoyant le produit scalaire du vecteur courant avec le vecteur passé en paramètre.
Cet exemple permet d'illustrer la façon dont on accède aux composants de l'objet courant, en invoquant simplement leur nom, mais aussi la façon dont on accède aux composants d'un objet extérieur, en invoquant le nom de la composante precédée d'un point et du nom de l'objet en question.
L'écriture de la classe `Vecteur` est la suivante,

```
class Vecteur {
    double[] composant;

    // constructeurs
    Vecteur(int dim) { composant = new double[dim]; }

    Vecteur(double tableau[]) { composant = tableau; }

    // acces a la composante i
    double elt(int i) { return composant[i]; }

    // modification de la composante i
    void toElt(int i, double x) { composant[i] = x; }

    // renvoie sa tailla
    int dim() { return composant.length; }

    // affiche ses composantes
```

```
    void afficher() {
      for (int i=0; i<dim(); i++)
          System.out.print(elt(i)+" ");
      System.out.println("");
    }

    // renvoie sa somme avec le vecteur en parametre
    Vecteur add(Vecteur x) {
      Vecteur w = new Vecteur(dim());
      for (int i=0; i<dim(); i++)
          w.toElt(i, elt(i) + x.elt(i));
      return w;
    }

    // renvoie son produit scalaire avec le vecteur en parametre
    double prodScalaire(Vecteur x) {
      double p = 0;
      for (int i=0; i<dim(); i++)
          p += elt(i)*x.elt(i);
      return p;
    }
}
```

Nous donnons dans le listing suivant un exemple de classe qui va contenir un programme principal, c'est-à-dire une méthode de type `public static void main(String args[])` . Le paramètre `args` correspond à d'éventuels arguments d'appel.

```
class TestVecteur {
    public static void main(String args[]) {
        double []t1 = {1.0, 2.0, 3.0};
        double []t2 = {5.5, 7.5, 9.5};

        Vecteur x1 = new Vecteur(3);
        for (int i=0; i<x1.dim(); i++)
            x1.toElt(i, t1[i]);
        System.out.println("premier vecteur :");
        x1.afficher();

        Vecteur x2 = new Vecteur(t2);
        System.out.println("deuxieme vecteur :");
        x2.afficher();

        Vecteur x3 = x1.add(x2);
        System.out.println("leur somme vaut :");
        x3.afficher();

        double produit=x1.prodScalaire(x2);
        System.out.println("leur produit scalaire vaut : "+ produit);
    }
}
```

Les commandes à utiliser pour compiler et exécuter le programme seront décrites au paragraphe 3.5.2. Le programme génère alors l'affichage suivant à l'exécution :

```
premier vecteur :
1.0 2.0 3.0
deuxieme vecteur :
5.5 7.5 9.5
leur somme vaut :
6.5 9.5 12.5
leur produit scalaire vaut : 49.0
```

3.3.6 Composants de type `static`

Il est possible de définir des *composants*[4], données ou méthodes, qui ne sont pas rattachés de manière propre à chaque objet instancié, c'est-à-dire à chaque instance de la classe, mais qui sont communs à toutes. Pour cela il suffit de déclarer le composant avec le qualificatif `static`. Par exemple, on peut ajouter à la classe vecteur une donnée entière qui va compter le nombre d'objets de la classe qui ont été instanciés. On peut également remplacer la méthode `add` par une méthode qui est `static` et qui prend 2 objets `Vecteur` en paramètres : on redonne à l'écriture de cette fonction une apparence de symétrie sur ses arguments correspondant à la propriété de communtativité de l'addition. Ci-dessous nous avons partiellement réécrit la classe `Vecteur` en prenant en compte ces modifications.

```
class Vecteur {
    double[] composant;
    static int nb =0;

    // constructeurs
    Vecteur(int dim) {
        composant = new double[dim];
        nb++; System.out.println("creation de l'objet "+nb);
    }

    Vecteur(double tableau[]) {
        composant = tableau;
        nb++; System.out.println("creation de l'objet "+nb);
    }

    ...

    // renvoie sa somme avec le vecteur en parametre
    static Vecteur add(Vecteur x, Vecteur y) {
      vecteur w = new vecteur(x.dim());
      for (int i=0; i<x.dim(); i++)
          w.toElt(i, x.elt(i) + y.elt(i));
      return w;
    }
```

[4] voir 3.3.1

```
    ...

}
```

Pour accéder à la nouvelle méthode `add`, on procédera de la manière suivante :

```
double[] t1 = {1.0, 3.2, 5.3};
double[] t2 = {3.0, 4.1, 6.3};
Vecteur x1 = new Vecteur(t1);
Vecteur x2 = new Vecteur(t2);
Vecteur x3 = Vecteur.add(x1, x2);
```

3.3.7 Composants de type `public` et de type `private`

Une notion fondamentale en programmation objet consiste à séparer, dans la description ou l'implémentation des objets, les parties visibles de l'extérieur et que l'on appelle *interface* de celles qui n'ont pas besoin d'être connues à l'extérieur de l'objet.
Les composants de la première partie porteront alors le qualificatif `public` et ceux de la seconde le qualificatif `private`. Dans notre exemple de classe vecteur, nous avons défini les opérations d'accès en lecture (fonction `elt`) et en écriture (fonction `toElt`) dans un objet, si bien qu'il n'est jamais utile d'accéder au tableau `composant` interne à la classe. Ainsi nous déclarerons `public` les deux fonctions `elt` et `toElt` mais `private` le tableau `composant`.
L'intérêt de séparer ainsi les parties publiques des parties privées est de garantir une évolutivité possible des classes, sans avoir à modifier les programmes qui les utilisent, à partir du moment où l'on conserve leur interface publique. Ainsi la classe `vecteur` pourra utiliser des types de structures de données autres que des tableaux pour stocker ses composantes. On pourra par exemple, utiliser un stockage directe sur fichier (pour des vecteurs de dimension importante) ou encore un stockage spécifique pour des structures creuses (en ne stockant que les coefficients non nuls et leur position). Il suffira alors de redéfinir correctement les deux fonctions d'accès en lecture (`elt`) et en écriture (`toElt`) en respectant leur mode d'appel. Tous les programmes utilisant des classes vecteurs n'auront alors pas lieu de subir la moindre modification pour pouvoir utiliser ces nouveaux types de vecteurs.
La classe vecteur pourra alors être partiellement réécrite avec des parties publiques et privées, comme ci-dessous :

```
class Vecteur {
    private double[] composant;
    static int nb =0;

    // acces a la composante i
    public double elt(int i) { return composant[i]; }

    // modification de la composante i
    public void toElt(int i, double x) { composant[i] = x; }

    ...

}
```

L'utilisation de ces fonctions n'est pas affectée par ces déclarations supplémentaires : on introduit simplement des limitations aux composants comme décrit précédemment.
Il est à noter qu'en n'indiquant ni `public` ni `private`, les composants sont considérés comme étant définis `public` par défaut, sauf si l'on se trouve dans un autre *package* que celui de la classe considérée - nous reviendrons sur cette nuance dans le paragraphe 3.5.3.

3.3.8 Chaînes de caractères

Les chaînes de caractères en Java sont des objets, *instances* de la classe prédéfinie `String`, et elles référencent des chaînes constantes. On pourra les déclarer comme dans l'exemple suivant :

```
String ch1 = new String("bonjour");
```

ou encore, sous une forme condensée qui est spécifique au type `String` :

```
String ch1 = "bonjour";
```

La chaîne `"bonjour"` est ici constante mais `ch1` peut être réaffectée pour référencer une autre chaîne constante, comme dans l'exemple suivant :

```
String ch2 = "au revoir";
ch1 = ch2;
```

L'ensemble des méthodes de la classe String peut être obtenu en consultant la documentation de l'API (Application Programming Interface) associée au JDK utilisé. Cette documentation, au format HTML, se révèle insdispensable dans la pratique, pour pouvoir accéder aux descriptions des interfaces des nombreuses classes proposées dans Java. Toutefois nous allons examiner quelques unes des méthodes les plus utiles relatives à la classe String :

- La méthode `static String valueOf(int i)` renvoie une chaîne contenant la valeur de `i`. Cette fonction existe aussi pour des paramètres de tous les types primaires. Notons que c'est une méthode statique qui s'appelle, par exemple, de la manière suivante : `String.valueOf(12)` et qui retourne ici la chaîne "12".
- La méthode `boolean equals(String s)` compare le contenu de la chaîne courante avec la chaîne s.
- La méthode `String concat(String s)` renvoie la concaténation de la chaîne courante (celle qui va préfixée la fonction `concat`) et de `s`. Il faut noter toutefois que les concaténations de chaînes peuvent se faire simplement avec l'opérateur +, comme on peut le remarquer dans les appels de la fonction d'affichage dans certains des exemples qui précèdent.
- La méthode `int length()` renvoie la longueur de la chaîne courante.
- La méthode `int indexOf(int c)` renvoie la position de la première occurence du charactère de code ASCII `c`. Elle renvoie `-1` si ce caractère n'apparaît pas.
- La méthode `char charAt(int i)` renvoie le caractère à la position `i`.

Nous avons vu que les objets `String` référencent des chaînes constantes. On peut, en fait, travailler avec des chaînes modifiables, en utilisant des objets de la classe prédéfinie `StringBuffer` dont on va décrire, sommairement, les méthodes essentielles. Là encore, pour plus d'informations, on consultera la documentation en ligne de l'API.
Les différents constructeurs de la classe `StringBuffer` sont :

– `StringBuffer()` permettant de créer une chaîne vide ;
– `StringBuffer(int dim)` permettant de créer une chaîne de longueur `dim` ;
– `StringBuffer(String s)` permettant de créer une chaîne contenant `s`.
Les principales méthodes de la classe `StringBuffer` sont :
– `int length()` renvoyant la longueur de la chaîne ;
– `StringBuffer append (String s)` ajoutant `s` à la fin de la chaîne courante ;
– `String toString()` renvoyant dans une chaîne constante la chaîne modifiable courante.

3.4 Classes internes

Une classe interne est une classe définit à l'intérieur d'une autre classe. Il s'agit typiquement d'une classe locale qui peut être invisible à l'extérieur de la classe englobante avec le qualificatif private.

Exemple : classe pile avec des maillons chaînés

```
class PileEnt {
  private class Maillon {
    public int info;
    public Maillon suivant;
    public Maillon(int e, Maillon s) {info=e; suivant=s;}
  }
  private Maillon sommet;
  public PileEnt() {sommet=null;}
  public void empiler(int e)
    {sommet=new Maillon(e, sommet);}
  public void depiler() {sommet=sommet.suivant;}
  public int lire() {return sommet.info;}
  public boolean vide() {return (sommet==null); }
}
```

Pour accéder à une classe interne non privée, en dehors de la classe englobante, on devra utiliser la construction suivante :
`classeEnglobante.classeInterne`

Il y a pratiquement une classe interne spécifique par objet instancié de la classe englobante. La classe interne peut alors accéder aux composants de la classe englobante.

Exemple : On crée dans la classe `PileEnt` une classe interne `Parcours` pour pouvoir construire différents parcours utilisés simultanément.

```
class PileEnt {
  private int sommet;
  private int T[] = new int[100];
  public PileEnt() {sommet=0;}
  public void empiler(int e) {T[sommet++] = e;}
  public void depiler() { sommet-- ;}
  public int lire() { return T[sommet-1]; }
```

```
  public boolean vide() {return (sommet==0); }

  public class Parcours {
    private int courant;
    public Parcours() {courant=sommet;}
    public int element() {return T[courant-1];}
    public void suivant() {courant--;}
    public boolean estEnFin() {return (courant==0);}
  }
}
```

Exemple d'utilisation :

```
class Testpile {
  public static void main(String args[]) {
    PileEnt p= new PileEnt(); ?
    PileEnt.Parcours pa1= p.new Parcours();
    PileEnt.Parcours pa2= p.new Parcours();
    System.out.println(pa1.element()); pa1.suivant();
    System.out.println(pa2.element()); pa2.suivant();
    ...
  }
}
```

3.5 Organisation des fichiers sources d'un programme Java

3.5.1 Structure des fichiers sources

Un programme Java correspond à une collection de classes dans un ou plusieurs fichiers sources dont l'extension est `"java"`. L'un de ces fichiers doit contenir une classe qui implémente la méthode `public static void main(String args[])`, comme cela est fait dans l'exemple précédent de construction de la classe vecteur et de son programme de test.

3.5.2 Commandes de compilation et de lancement d'un programme

Pour compiler et exécuter les programmes java on utilise les outils fournis avec le JDK. On commence par lancer la compilation des fichiers avec la commande `javac`. Par exemple, pour les deux fichiers relatifs à notre classe vecteur et à son programme de test, on écrira :

```
javac Vecteur.java
javac TestVecteur.java
```

Deux fichiers : `Vecteur.class` et `TestVecteur.class` ont été générés et correspondent aux noms de toutes les classes définies dans les fichiers sources. Ce sont des fichiers en byte-code portables sur toute machine devant être traités par la machine virtuelle java lancée par la commande `java`. On exécute donc le programme principal, qui est dans la classe `TestVecteur`, en tapant la commande :

```
java TestVecteur
```

3.5.3 Packages

Un package permet de regrouper un ensemble de classes. L'instruction `package nomPack` en début de fichier indique que les classes qui y sont définies appartiennent au package nommé `nomPack`. Pour accéder aux classes de ce package, lorsque l'on est dans une classe qui n'y appartient pas, on utilise la dénomination `nomPack.className`, où `className` désigne le nom de la classe.
Les désignations de package suivent un schéma de construction arborescent du type : `name.subname.subsubname`. Il existe un lien entre les noms de package et les répertoires où se trouvent les classes y appartenant . Par exemple, une classe `watch` appartenant au package `time.clock` doit se trouver dans le fichier `time/clock/watch.class`.
Les répertoires où Java effectue sa recherche de packages sont définis dans la variable d'environnement : `CLASSPATH`.
L'instruction `import packageName` permet d'utiliser des classes du package défini, sans avoir besoin de les préfixer par leur nom de package. On peut importer toutes les classes d'un package en utilisant un import du type `import packageName.*;`, mais, **ATTENTION**, l'import d'un niveau de package ne permet pas d'importer les packages qui sont en-dessous dans l'arborescence des répertoires.
Voici un exemple d'illustration qui montre une organisation de classes java, dans différents répertoires et leur utilisation.
La variable d'environnement `CLASSPATH` doit être dans le fichier `.profile` ou dans `.bashrc`, par exemple, sous Unix, de la manière suivante :

```
CLASSPATH = ~/myJavaClass
```

Voici maintenant des extraits de différents fichiers rangés dans les répertoires indiqués :
– le fichier /myJavaClass/bibMat/calVecteur/Vecteur.java correspond à

```
package bibMat.calVecteur;
public class Vecteur { ... }
```

– le fichier /myJavaClass/bibMat/calMatrice/Matrice.java correspond à

```
package bibMat.calMatrice;
public class Matrice { ... }
```

– le fichier /myJavaClass/calUtil/TestBibMat.java correspond à

```
package calUtil;
import bibMat.calMatrice.*;
public class TestBibMat {
  public static void main (String args[]) {
    bibMat.calVecteur.Vecteur x =
        new bibMat.calVecteur.Vecteur(3);
    Matrice M = new Matrice(3, 3);
    ...
  }
}
```

3.5.4 Utilisation des packages

Un package regroupe des classes qui portent sur un même domaine Au début de chaque fichier, on met package nomPackage La hiérarchie des package se retrouve au niveau de l'arborescence des fichiers et répertoires.
– Import nomPackage : pour utiliser une classe, on précisera le nom du package
– Import nomPackage.UneClasse : seule UneClasse est importée
– Import nomPackage.* : importe toutes les classes du package (mais pas les classes des sous-package !)

3.5.5 Visibilité des composants dans les packages

On a vu précédemment que l'accessibilité des composants d'une classe se fait grâce aux qualificatifs `private` ou `public`. En fait, elle est également liée aux localisations dans les packages. Ainsi, lorsqu'un composant ne précise pas sa nature (`private` ou `public`) alors il est, par défaut, `public` dans les classes du package (ou du répertoire) auquel il appartient et `private` en dehors.
Nous illustrons ces propos avec l'exemple suivant :
– Package P1 :

```
class C1 {
  public int xa;
  int xc;
  private int xd;
}
class C2 { ... }
```

– Package P2 :

```
class C3 { ... }
```

Dans cet exemple, la classe `C2` peut accéder à `xa` et `xc`. Par contre `C3` ne peut accéder qu'à `xa` uniquement.

3.5.6 packages prédéfinis en Java

Le langage Java possède un grand nombre de packages prédéfinis regroupés par thèmes. Ces packages et les classes qu'ils contiennent sont décrits de manière exhaustive dans une documentation fournie par Sun et qui reprend l'ensemble des interfaces. Cette documentation est appelée API (Application Programming Interface) et est distribuée sous un format HTML, permettant ainsi une navigation hypertexte particulièrement adaptée aux recherches d'informations nécessaires pour le développement de programmes. On y trouve principalement :
– `java.lang` qui correspond aux classes de base (chaînes, math, ...),
– `java.util` qui correspond aux structures de données (vector, piles, files, ...),
– `java.io` qui correspond aux entrées/sorties,
– `java.awt` qui correspond au graphisme et fenêtrage,
– `java.net` qui correspond aux communications Internet,
– `java.applet` qui correspond aux insertions dans des documents HTML.

3.6 Héritage

3.6.1 Construire une classe dérivée

La notion d'héritage est importante en programmation objet. Elle permet de définir une classe dérivée à partir d'une autre classe dont on dit qu'elle *hérite*. La classe dérivée possède alors, par défaut, l'ensemble des composants de la classe dont elle hérite - on l'appelle classe mère - sauf si des restrictions ont été posés sur ces composants. L'héritage est donc un concept essentiel renforçant les propriétes de réutilisabilité des programmes objets.
L'utilisation répétée de l'héritage sur des classes successives conduit à la construction d'une hiérarchie entre elles, que l'on peut schématiser par un arbre d'héritage. La figure 3.1 présente un exemple d'arbre d'héritage construit sur des classes permettant de représenter des figures géométriques.

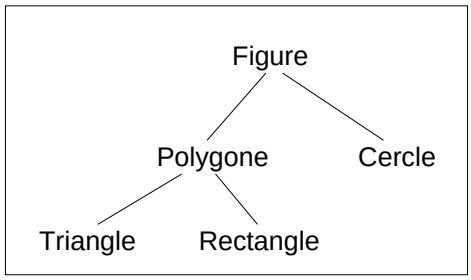

FIG. 3.1: Arbre d'héritage de classes d'objets géométriques

La construction effective des différentes classes de la figure 3.1 est faite dans l'exemple complet du paragraphe 3.6.5.
Pour définir une classe ClB qui dérive de la classe ClA, on fera une déclaration du type :

```
class ClB extends ClA { ... }
```

Un objet de la classe ClB est alors aussi un objet de la classe ClA, il peut être utilisé partout où un objet de la classe ClA est attendu.
On peut tester l'appartenance d'un objet à une classe grâce à l'opérateur `instanceof`, comme dans l'exemple qui suit :

```
ClB x;
if ( x instanceof ClA)
    System.out.println(''voici un objet ClA !'');
```

L'exécution des lignes précédentes provoque l'affichage de "`voici un objet ClA !`".

3.6.2 Constructeur d'une classe dérivée

Si l'on définit un constructeur d'une classe dérivée, celui-ci doit explicitement faire appel à un constructeur de la classe mère, auquel on accède avec la méthode prédéfinie `super( ... )`. Si cet appel explicite n'est pas fait, l'exécution du programme provoquera un appel implicite du constructeur par défaut, c'est à dire sans paramètre, de la classe mère ; il faut donc impérativement que celui-ci existe sinon une erreur de compilation sera diagnostiquée. Ainsi,

si l'on a défini un ou des constructeurs dans la classe mère, une de ses versions devra être sans paramètre.
L'appel explicite du constructeur de la classe mère se fait par la méthode prédéfinie `super( ... )`, cet appel est obligatoirement la première instruction du constructeur. On ne peut donc transmettre que des valeurs de paramètres du constructeur courant lors de l'appel de `super( ... )`.
On retrouve de telles constructions dans l'exemple du paragraphe 3.6.5 où la classe `Figure` possède un composant de type `Point` correspondant à son origine et pouvant être initialisé par le paramètre `x` du constructeur `Figure(Point p)`. On définit la classe `Cercle` dérivant de `Figure`, dont le constructeur est défini par :

```
Cercle (Point centre, double r)
{ super(centre); ... }
```

3.6.3 Accessibilité : public, protected et private

On a vu précédemment que les composants d'une classe pouvaient être éventuellement qualifiés des termes `public` ou `private`. Il existe, en fait, un troisième qualificatif `protected` qui indique que ce composant est accessible uniquement dans les classes dérivées et les classes du même package.
Par exemple, soit la classe ClA définie de la manière suivante :

```
package aa;
public class ClA {
    protected int JJ;
    ...
}
```

et la classe ClB définie ainsi :

```
package bb;
import aa.*;
class ClB extends ClA {
    void PP() {
        JJ++;  // autorisé
        ClB b;
        b.JJ++ // autorisé
        ClA a;
        a.JJ++ // interdit
    }
}
```

3.6.4 Méthodes virtuelles et classes abstraites

Une classe peut annoncer une méthode sans la définir, on dit alors que la classe est abstraite. Elle doit être introduite avec le mot clé `abstract`.
Par exemple, on peut définir la classe abstraite `ClA` suivante et une classe `ClB` qui en dérive.

```
abstract class ClA {
    abstract void fctP() ;
    void fctQ() { ... };
    ...
}
class ClB extends ClA {
    void fctP() { ... };
    ...
}
```

En raison de sa définition incomplète, il est impossible d'*instancier* une classe abstraite qui ne sert qu'à la construction de classes dérivées. Ces dernières devront redéfinir toutes les méthodes abstraites, pour ne pas l'être elles-mêmes et pouvoir ainsi être instanciées.

3.6.5 Un exemple : quelques objets géométriques

Nous donnons un exemple d'implémentation des différentes classes décrites sur la figure 3.1. Nous commençons par définir une classe `Point` qui servira dans les autres classes :

```
class Point {
  double abscisse;
  double ordonnee;
  Point(double x, double y)
    {abscisse=x; ordonnee=y;}
  Point(Point p)
    {abscisse=p.abscisse; ordonnee=p.ordonnee;}
  static double distance(Point p, Point q) {
    double dx=p.abscisse-q.abscisse;
    double dy=p.ordonnee-q.ordonnee;
    return Math.sqrt(dx*dx+dy*dy);
  }
}
```

Nous définissons ensuite une classe abstraite pour définir le type `Figure` constitué de deux méthodes abstraites d'affichage et de calcul de périmètre :

```
abstract class Figure {
  private static final Point zero=new Point(0,0);
        Point origine;
  Figure(){origine=zero;}
  Figure(Point p){origine=new Point(p);}
  abstract double perimetre();
  abstract void affiche();
}
```

La classe `Cercle` qui suit dérive de la classe `Figure` en implémentant ses deux méthodes abstraites `perimetre` et `affiche` :

```
class Cercle extends Figure {
  private static final double pi=3.141592;
```

```
  double rayon;
  Cercle(Point centre, double r)
    {super(centre); rayon=r;}
  double perimetre()
    {return 2*pi*rayon;}
  void affiche() {
    System.out.println("Cercle");
    System.out.println("rayon : " + rayon +" et centre : " +
                       "(" + origine.abscisse +
                       "," + origine.ordonnee + ")  ");
    }
}
```

La classe `Polygone` dérive de la classe `Figure`. Elle se caractérise par un tableau de `Point` :

```
class Polygone extends Figure {
  Point sommet[]= new Point[100];
  int nbs;
  Polygone(){nbs=0;}
  Polygone(Point[] m, int n) {
    super(m[0]);
    nbs=n;
    for (int i=0; i<n; i++)
      sommet[i]=m[i];
  }
  double lcote(int i) {
    if (i<nbs)
      return Point.distance(sommet[i-1], sommet[i]);
    else
      return Point.distance(sommet[i-1], sommet[0]);
  }
  double perimetre() {
    double somme=0;
    for (int i=1; i<=nbs; i++)
      somme += lcote(i);
    return somme;
  }
  void affiche() {
    System.out.println("Polygone");
    for (int i=0; i<nbs; i++)
      System.out.print("(" + sommet[i].abscisse +
                       "," + sommet[i].ordonnee + ")  ");
    System.out.println();
  }
}
```

La classe `Triangle` est une classe élémentaire dérivée de la classe `polygone` :

```
class Triangle extends Polygone {
  Triangle(Point[] m) { super(m,3); }
}
```

La classe `Rectangle` dérive de la classe `Polygone`. Elle est définie avec ses caractéristiques mathématiques classiques, c'est à dire la longueur de ses côtés et un de ses sommets, le sommet inférieur gauche. On a alors une bonne illustration de l'utilisation des constructeurs successifs des classes dérivées. Pour appeler le constructeur de `Polygone` qui possède le tableau de ses sommets en paramètres, il faudrait tout d'abord faire la construction de ce tableau à partir des caractéristiques de `Rectangle`, ce qui n'est pas possible, car l'appel de `super` doit être la première opération. Il faut donc utiliser le constructeur par défaut dans la classe `Polygone` qui sera appelé au début de l'exécution du constructeur de `Rectangle`. La composante `sommet` de la classe sera alors construite plus loin :

```
class Rectangle extends Polygone {
  double largeur;
  double longueur;
  Rectangle(Point m, double lo, double la) {
     // appel implicite du constructeur de
     // Polygone sans parametre
     // l'appel du constructeur de Polygone avec parametres
     // ne peut se faire car il faut d'abord construire
     // le tableau a lui transmettre qui ne peut se faire
     // qu'apres l'appel explicite ou implicite de super
     Point P1= new Point(m.abscisse+ lo, m.ordonnee);
     Point P2= new Point(m.abscisse, m.ordonnee+ la);
     Point P3= new Point(m.abscisse+ lo, m.ordonnee+ la);
     Point mr[]={m, P1, P3, P2};
     sommet=mr;
     nbs=4;
     largeur= la;
     longueur= lo;
  }
}
```

Voici un programme principal de test :

```
class Geometrie {
  public static void main(String args[]) {
    Point P1= new Point(3,4);
    Point P2= new Point(4,4);
    Point P3= new Point(0,0);
    Point P4= new Point(1,0);
    Point P5= new Point(0,1);
    Point [] TabP={P3, P4, P5};
    Cercle c= new Cercle(P1,2);
    Rectangle r= new Rectangle(P2,5,2);
    Triangle t= new Triangle(TabP);
    Figure f; //autorise, mais pas new Figure !
    System.out.println("perimetre cercle");

    f=c; f.affiche(); // appel de affiche de Figure
                      // puis de sa forme derivee de cercle
    System.out.println("perimetre : " + f.perimetre());
```

```
        f=r; f.affiche();
        System.out.println("perimetre : " + f.perimetre());

        f=t; f.affiche();
        System.out.println("perimetre : " + f.perimetre());
      }
    }
```

L'exécution du programme affiche :

```
java Geometrie
perimetre cercle
Cercle
rayon : 2.0 et centre : (3.0,4.0)
perimetre : 12.566368
Polygone
(4.0,4.0)  (9.0,4.0)  (9.0,6.0)  (4.0,6.0)
perimetre : 14.0
Polygone
(0.0,0.0)  (1.0,0.0)  (0.0,1.0)
perimetre : 3.414213562373095
```

3.6.6 Interfaces

Une *interface*, en Java, permet de décrire un modèle de construction de classe dans lequel on n'indique uniquement que les en-têtes des méthodes. Cela équivaut, d'une certaine manière, à une classe où toutes les méthodes sont abstraites.
On dira qu'une classe *implémente* une *interface*, si elle redéfinit toutes les méthodes décrites dans cette *interface*.
Par exemple, on définit un modèle de problème par une interface qui comporte deux méthodes `poserProbleme` et `resoudreProbleme` :

```
interface AResoudre {
    void poserProbleme();
    void resoudreProbleme();
}
```

On peut alors construire une classe `equationPremierDegre` qui implémente l'interface `aResoudre` :

```
class EquationPremierDegre implements AResoudre {
    double coefx, coef1, solution;
    void poserProbleme()
    {  // on lira coefx et coef1 }
    void resoudreProbleme()
    {  // on affecte une valeur a solution }
    ...
}
```

Un autre intérêt des interfaces provient de la limitation de Java en terme d'héritage multiple. En effet, Java ne permet pas qu'une classe dérivée puisse avoir plusieurs classes mères et donc de bénéficier des composants définis dans ces différentes classes. Pour pallier cette limitation, on devra utiliser conjointement l'héritage et l'implémentation d'interfaces. Il est, par ailleurs, possible d'implémenter plusieurs interfaces en Java, contrairement à l'héritage.

3.6.7 Passage d'une fonction en paramètre d'une méthode

Nous nous intéressons au problème du passage d'une fonction en tant que paramètre dans une méthode. Par exemple, on souhaite décrire dans une classe un procédé qui approche le calcul d'une dérivée d'une fonction réelle d'une variable réelle par un taux d'accroissement :

$$f'(x) \simeq \frac{f(x+h/2) - f(x-h/2)}{h}$$

Nous allons montrer comme ce procédé peut être décrit d'une manière générique, c'est à dire en utilisant une fonction abstraite f, paramètre du procédé. On applique alors ce procédé de dérivation numérique à une fonction particulière en la transmettant par l'intermédiaire de ce paramètre.

Dans beaucoup de langages, comme le C ou le C++, le passage d'une fonction en paramètre s'effectue en gérant un pointeur qui contient l'adresse effective du code de la fonction. Le langage Java ne proposant pas de gestion explicite de pointeur, on doit alors créer une *enveloppe* de type objet contenant une méthode correspondante à l'évaluation de la fonction. C'est cette classe *enveloppe* qui correspond au paramètre à gérer.
Il faut donc commencer par définir une classe abstraite, ou une interface, qui décrit les fonctionnalités minimales de la classe correspondant au paramètre fonctionnel.
Nous donnons un exemple qui s'appuie sur l'utilisation d'une interface minimale caractérisant une fonction réelle d'une variable réelle. Les nombres réels sont implémentés par le type `double` :

```
interface FoncD2D { public double calcul(double x); }
```

Nous pouvons alors utiliser cette interface pour décrire un procédé générique de calcul de dérivation numérique, comme décrit ci-dessus :

```
class DiffFinies {
    public double derivOrdre1 (FoncD2D f, double x, double h) {
        return (f.calcul(x+h/2) - f.calcul(x+h/2))/h ;
    }
}
```

Pour utiliser ce procédé, il suffit maintenant de définir une fonction particulière dans une classe qui implémente l'interface `FoncD2D` :

```
class FoncCarre implements FoncD2D {
    public double calcul (double x) { return x*x ; }
}
```

Le programme principal suivant va alors construire un objet de la classe `FoncCarre` qui permet d'utiliser la fonction particulière qui y est définie. Il construit aussi un objet de la classe `DiffFinies` qui permet d'utiliser le procédé de dérivation numérique qui est invoqué sur l'objet de type `FoncCarre`, reconnu comme une implémentation de `FoncD2D` :

```
class TestDiffFinies {
    public static void main (String args[]) {
        FoncCarre f = new FoncCarre();
        DiffFinies df = new DiffFinies();
        System.out.println ("Différences finies d'ordre un "+
                            "en 1 de pas 0.01 : "+
                            df.derivOrdre1(f, 1, 0.01));
    }
}
```

Le résultat de l'exécution est :

```
java TestDiffFinies
Différences finies d'ordre un en 1
de pas 0.01 : 1.9999999999999685
```

3.7 Exceptions

3.7.1 Notions générales

L'introduction de la notion d'*exception* dans un langage de programmation a pour but de simplifier le traitement de certaines situations. Ces dernières sont considérées comme exceptionnelles, au sens où leur détection nécessite de les gérer, en les sortant du contexte dans lequelle elles sont détectées.
Dans les langages ne gérant pas spécifiquement ces situations d'exception, il est nécessaire d'utiliser de nombreuses instructions conditionnelles successives, afin de réorienter le déroulement du programme de manière adéquate. De tels processus peuvent conduire à une complexité du programme dont la lecture finit par s'alourdir.
Certains langages proposent de manière facultative l'utilisation des exceptions (c'est le cas du C++). Avec Java, on a l'obligation de les gérer lorsque certains appels de méthodes sont succeptibles, de par leur conception, de déclancher des traitements d'exception.
Une gestion d'exception est caractérisée par une séquence

`try - Catch - finally`

qui correspond typiquement au déroulement suivant :

```
    try {
        //séquence succeptible de déclancher une exception
        ...
    }
    catch (classException e1) {
        //traitement à effectuer si e1 a été déclanchée
        ...
```

```
}
catch ( ....) { ... } //autre déclanchement éventuel
finally {
    //traitement effectué avec ou sans déclanchement d'exception
    ...
}
```

Nous donnons ci-dessous un exemple de programme qui récupère les arguments fournis au lancement du programme. Il calcule et affiche la moyenne de ces arguments lorsque ce sont des entiers. Il déclanche un traitement d'exception si l'un des arguments ne correspond pas à un entier.

```
class exceptionCatch {
    static int moyenne (String[] liste) {
        int somme=0, entier, nbNotes=0, i;
        for (i=0; i<liste.length; i++)
            try {
                entier=Integer.parseInt(liste[i]);
                // conversion chaîne en valeur entière
                somme += entier; nbNotes++;
            }
            catch (NumberFormatException e) {
                System.out.println("note: "+(i+1)+" invalide");
            }
        return somme/nbNotes;
    }

    public static void main (String[]argv) {
        System.out.println("moyenne "+moyenne(argv));
    }
}
```

Une exécution possible du programme est la suivante :

```
java exceptionCatch ha 15 12 13.5
note: 1 invalide
note: 4 invalide
moyenne 13
```

3.7.2 Définir sa propre exception

Pour définir sa propre exception, il faut définir une classe qui hérite de la classe prédéfinie `Exception`. On pourra surcharger, en particulier, la méthode prédéfinie `toString()` dont la chaîne renvoyée correspond au message affiché, lorsque l'on demande d'afficher l'exception elle-même.
Une méthode succeptible de déclancher une exception devra avoir une clause `throws`, suivie de l'exception déclanchable dans son en-tête. Dans le corps de la méthode, on précisera dans quelle condition l'exception est déclanchée, en invoquant l'opérateur `throw`.
On donne ci-après un complément du programme précédent déclanchant une exception, lors du calcul de la moyenne d'un tableau, s'il ne contient pas d'éléments.

```
class ExceptionRien extends Exception {
    public String toString() {
        return ("aucune note !");
    }
}
class ExceptionThrow {
    static int moyenne (String[] liste) throws ExceptionRien {
        int somme=0, entier, nbNotes=0, i;
        for (i=0; i<liste.length; i++)
            try {
                entier=Integer.parseInt(liste[i]);
                // conversion chaîne en valeur entière
                somme += entier; nbNotes++;
            }
            catch (NumberFormatException e) {
                System.out.println("note: "+(i+1)+" invalide");
            }
        if (nbNotes == 0) throw new ExceptionRien();
        return somme/nbNotes;
    }
    public static void main (String[]argv) {
        try {
            System.out.println("moyenne "+moyenne(argv));
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Voici deux exécutions successives du programme précédent :

```
java exceptionThrow q d 1 3 5.2
note: 1 invalide
note: 2 invalide
note: 5 invalide
moyenne 2

java exceptionThrow q d 3.1 a 5.2
note: 1 invalide
note: 2 invalide
note: 3 invalide
note: 4 invalide
note: 5 invalide
aucune note !
```

3.8 Entrées/Sorties

3.8.1 Classes de gestion de flux

Dans le langage Java, deux types d'entrées/sorties sont utilisables :
– Les entrées/sorties traditionnelles, c'est-à-dire utilisant les flux de communications "par défaut", à savoir le clavier ou l'écran, ou encore les fichiers ;
– Les entrées/sorties basées sur des interactions avec un système de fenêtrage. Celles-ci seront développées dans le chapitre sur le graphisme.

Nous présentons, dans la suite, des notions basées sur l'API 1.1 et qui ont été enrichies significativement par rapport aux versions précédentes. Les raisons de ces enrichissements sont principalement dues à des questions d'efficacité et à la possibilité d'utiliser des codes internationaux (UNICODE) qui enrichissent le code ASCII avec des caractères accentués, entre autres. Les principales classes de gestion de flux sont organisées suivant la hiérarchie d'héritage décrite dans la figure 3.2.

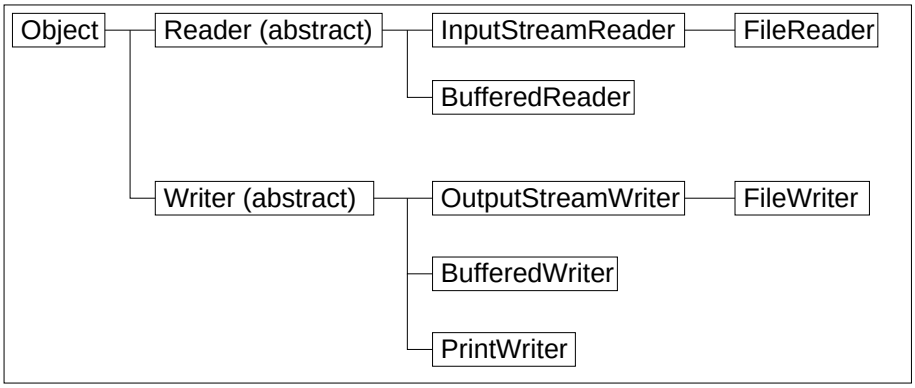

FIG. 3.2: Hiérarchie des classes de gestion de flux

Dans cette hiérarchie, les classes suivantes apparaissent :
– La classe `Object` qui est la classe de base en Java, dont toutes les autres héritenté ;
– Les classes abstraites `Reader` et `Writer` qui concernent respectivement les flux de caractères pour les lectures et les écritures ;
– Les classes `InputStreamReader` et `OutputStreamWriter` qui permettent de faire la traduction des données brutes en caractères UNICODE et inversement ;
– Les classes `BufferedReader` et `BufferedWriter` qui permettent l'utilisation d'une mémoire tampon pour les entrées-sorties. Cette mémoire est indispensable pour l'utilisation des périphériques standards (écran et clavier) ;
– Les classes `FileReader` et `FileWriter` qui permettent l'utilisation de fichiers ;
– La classe `PrintWriter` qui permet l'écriture de données formattées semblables aux affichages à l'écran.

Cette liste n'est pas exhaustive mais correspond aux principales classes que nous serons amenés à utiliser dans la suite.

3.8.2 Saisies au clavier

Pour effectuer une saisie au clavier, on construit successivement :

– Un flux `InputStreamReader` avec le flux de l'entrée standard, à savoir `System.in` ;
– Un flux de lecture bufferisé de type `BufferedReader` à partir du flux précédent.

On présente, ci-après, un exemple typique de lecture au clavier. Dans cet exemple, on lit dans le flux bufferisé avec la méthode `readLine()` permettant la lecture d'un chaîne de caractères jusqu'à ce que l'on rencontre un saut de ligne. La chaîne de caractère est alors convertie en entier, avec la méthode `() parseInt`, ou en flottant, avec un construction plus complexe qui utilise la méthode `floatValue()`, sur l'objet de la classe `Float` obtenu en appelant `valueof()`. Cette dernière méthode est statique et a pour paramètre la chaîne lue. Il faut noter qu'à partir du JDK 1.2, on pourra plus simplement appeler la méthode `parseFloat()`, similaire à `parseInt()`.

On remarquera que, dans cet exemple, il est nécessaire de gérer le déclanchement éventuel d'exceptions peuvant se produire,

– soit par un problème de lecture de flux qui provoque le déclanchement de l'exception `IOException` ;
– soit par un problème de conversion de chaîne de caractères en valeur numérique qui provoque l'exception `NumberFormatException`.

```
import java.io.*;
class Testio {
  public static void main(String[] args) {
    InputStreamReader fluxlu = new InputStreamReader(System.in);
    BufferedReader lecbuf = new BufferedReader(fluxlu);

    try {
      System.out.print("taper 1 ligne de caracteres : ");
      String line = lecbuf.readLine();
      System.out.println("ligne lue : " + line);

      System.out.print("taper 1 nombre entier : ");
      line = lecbuf.readLine();
      int i = Integer.parseInt(line);
      System.out.println("entier lu : " + i);

      System.out.print("taper 1 nombre reel : ");
      line = lecbuf.readLine();
      float f = Float.valueOf(line).floatValue();
      System.out.println("réel lu : " + f);

      System.out.println("somme des deux nombres : " + (i+f) );
    }
    catch(IOException e) {
      System.out.println("erreur de lecture"); }

    catch(NumberFormatException e) {
      System.out.println("erreur conversion chaine-entier"); }
  }
}
```

L'affichage obtenu à la suite de l'exécution du programme est le suivant :

```
java Testio
taper 1 ligne de caracteres : java sait lire
ligne lue : java sait lire
taper 1 nombre entier : 3
entier lu : 3
taper 1 nombre reel : 12.5
réel lu : 12.5
somme des deux nombres : 15.5
```

3.8.3 Lecture d'un fichier

Une lecture dans un fichier est effectué dans le programme suivant. Il est similaire au précédent avec, toutefois, quelques différences :

– `FileReader` est la classe instanciée pour construire le flux d'entrée à partir du nom du fichier. Cette instanciation pourra déclancher l'exception `FileNotFoundException`, si le fichier n'est pas trouvé.
– La méthode `close()` ferme le fichier.
– On utilise une boucle qui détecte la fin de fichier, suite au résultat d'une lecture qui renvoie la constante `null`.

```
import java.io.*;
class Testfile {
  public static void main(String[] args) {
    FileReader fichier=null;
    BufferedReader lecbuf;

    String line;
    float f, somme=0;
    int nbnombre=0;

    try {
      fichier = new FileReader("donnee.dat");
      lecbuf = new BufferedReader(fichier);

      while ( (line = lecbuf.readLine()) != null ) {
        f = Float.valueOf(line).floatValue();
        somme += f; nbnombre++;
        System.out.println("nombre lu : " + f);
      }
      if ( nbnombre > 0 )
        System.out.println("Moyenne : " + (somme/nbnombre));
    }
    catch(FileNotFoundException e) {
      System.out.println("fichier donnee.dat inexistant !"); }

    catch(IOException e) {
      System.out.println("erreur de lecture"); }

    catch(NumberFormatException e) {
      System.out.println("erreur conversion chaine-entier"); }
```

```
    finally {
      if (fichier!=null)
        try { fichier.close(); }
        catch(IOException e) {}
    }
  }
}
```

On exécute le programme précédent avec le fichier `"donnee.dat"` suivant :

```
2.3
1.2
3.4
2.1
5.2
```

L'affichage, produit par le programme, est alors le suivant :

```
java Testfile
nombre lu : 2.3
nombre lu : 1.2
nombre lu : 3.4
nombre lu : 2.1
nombre lu : 5.2
Moyenne : 2.84
```

3.8.4 Ecriture dans un fichier

Le programme suivant va utiliser un fichier dans lequel on écrit. On construit un flux de type `FileWriter`, utilisé dans un tampon d'écriture (de type `BufferedWriter`). Un flux, de type `PrinterWriter`, permet alors d'effectuer des écritures formattées avec la méthode `println`, comme on le fait couramment sur la sortie standard, c'est-à-dire l'écran.

```
import java.io.*;
class Testfileout {
  public static void main(String[] args) throws IOException {
    FileWriter fichier = new FileWriter("out.txt");
    BufferedWriter ecrbuf = new BufferedWriter(fichier);
    PrintWriter out = new PrintWriter(ecrbuf);
    out.println("coucou");
    out.println(5.6);
    System.out.println("fin d'ecriture dans le fichier out.txt");
    out.close();
  }
}
```

3.8.5 Compléments

Il est possible de lire et d'écrire des données brutes (donc non formattées) avec les classes `DataInputStream` et `DataOutputStream`. Les fichiers, ainsi construits, ne sont pas lisibles directement sous un éditeur de texte, par exemple, mais leur taille est plus réduite.
La classe `StringTokenizer` est une classe qui permet d'instancier un petit analyseur de texte qui peut, par exemple, découper des lignes en sous-chaînes de caractères, en reconnaissant un certain nombre de séparateurs (blancs, virgules, ...).
La classe `java.io.File` permet de faire des manipulations de fichiers, similaires aux commandes d'un système d'exploitation. Elle permet par exemple, de lister un répertoire, de renommer ou supprimer un fichier, etc.

3.9 Les threads

3.9.1 Introduction

– Un programme est caractérisé par
  – son code
  – son espace mémoire (données)

– Sur un OS évolué, exécution concurrente possible des processus avec "temps partagé" géré par un scheduler

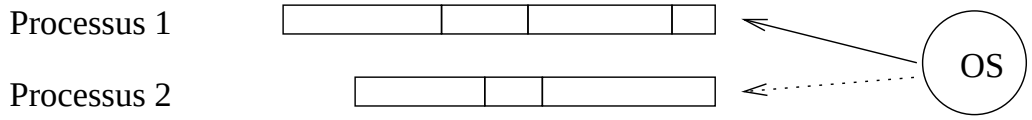

FIG. 3.3: Exécutions concurrentes

Si processus indépendants :
– contexte différent à chaque changement de processus ;
– possibilité de partager une mémoire (lourd à gérer) ;
– communications parfois nécessaires.

Une solution plus simple : *les threads* ou processus légers qui s'exécutent en parallèle mais partagent les données.

Exemple : concurrence de traitement lors de chargements d'images avec les navigateurs Web ou environnement graphique Java.

3.9.2 Création de threads avec Java

Threads gérés dans différents langages. Gestion lourde avec C et simplifiée avec Java qui propose deux solutions :
– Objet héritant de la classe `java.lang.Thread`. Deux objets de cette classe peuvent s'exécuter directement en concurrence.

– Objet implémentant l'interface `java.lang.Runnable`
  – On doit définir une méthode `run()` ;
  – On range ces objets dans des objets Threads (enveloppes) qui s'exécutent concurremment.

Exemple 1 : `TestThread1.java`

```
class TPrint extends Thread {

    String txt;
    int attente;

    public TPrint(String t, int p) {
        txt = t;
        attente = p;
    }

    public void run() {
        for (int i=0; i<8; i++) {
            System.out.print(txt+i+" ");
            try {
                sleep(attente);
            }
            catch(InterruptedException e) {};
        }
    }
}

public class TestThread1 {
    static public void main(String args[]) {
        TPrint a = new TPrint("A", 100);
        TPrint b = new TPrint("B", 200);
        a.start();
        b.start();
    }
}

// résultat de l'exécution :
// A0 B0 A1 B1 A2 A3 B2 A4 A5 B3 A6 A7 B4 B5 B6 B7
```

Exemple 2 : `TestThread2.java`

```
class TPrint implements Runnable {

    String txt;
    int attente;

    public TPrint(String t, int p) {
        txt = t;
        attente = p;
    }
```

```
    public void run() {
        for (int i=0; i<8; i++) {
            System.out.print(txt+i+" ");
            try {
                Thread.currentThread().sleep(attente);
            }
            catch(InterruptedException e) {};
        }
    }
}

public class TestThread2 {
    static public void main(String args[]) {
        TPrint a = new TPrint("A", 100);
        TPrint b = new TPrint("B", 200);
        new Thread(a).start();
        new Thread(b).start();
    }
}

// résultat de l'exécution :
// A0 B0 A1 B1 A2 A3 B2 A4 A5 B3 A6 A7 B4 B5 B6 B7
```

3.9.3 Les méthodes de gestion des threads

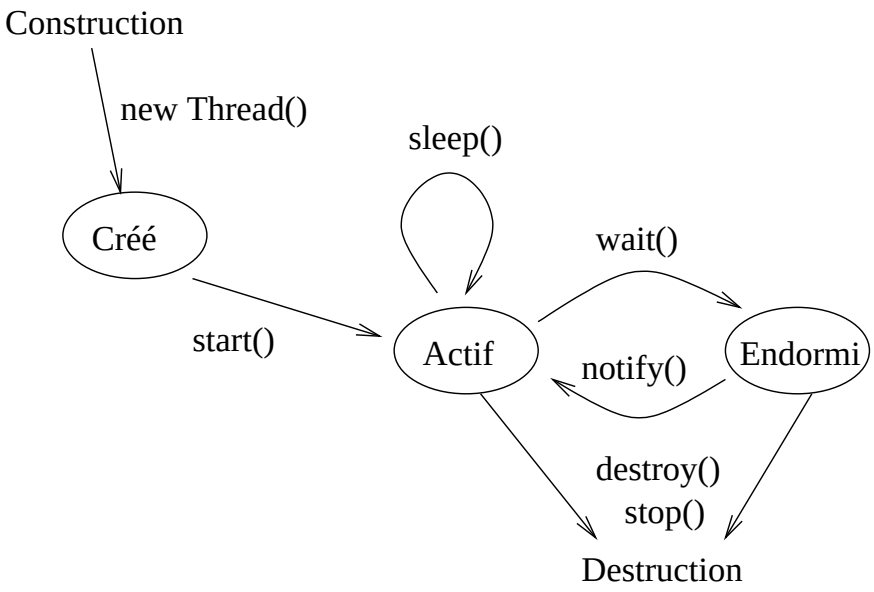

FIG. 3.4: Cycle de vie d'un thread

3.9.4 Synchronisation et Exclusion

Lorsque plusieurs threads travaillent sur des mêmes ressources, il est parfois utile ou nécessaire de limiter le parallélisme en assurant qu'un certain objet ne subisse pas en même temps plusieurs séquences d'actions concurrentes.

Exemple : Dans `TestThread3.java`, chaque thread affiche des mots caractère par caractère. Les mots sont mélangés. On souhaite conserver le parallélisme du traitement en garantissant que chaque mot entier ne soit pas coupé.

```
class TPrint extends Thread {

    String txt;

    public TPrint(String t) {
        txt = t;
    }

    public void run() {
        for (int j=0; j<3; j++) {
            for (int i=0; i<txt.length(); i++) {
                System.out.print(txt.charAt(i));
                try { sleep(100); }
                catch (InterruptedException e) {};
            }
        }
    }
}

public class TestThread3 {
    static public void main(String args[]) {
        TPrint a = new TPrint("bonjour ");
        TPrint b = new TPrint("au revoir ");
        a.start();
        b.start();
    }
}

// resultat de l'exécution :
// baoun jroeuvro ibro najuo urre vbooinrj oauur  revoir
```

Construction d'une classe moniteur

On définit une classe "moniteur" contenant une méthode "synchronized" qui va permettre l'écriture d'un mot dans son entier : lorsque cette méthode est appelée dans un thread, elle bloque le déroulement des autres threads jusqu'à ce qu'elle soit arrivée à la fin de son traitement.

Attention : pour que ce mécanisme fonctionne, il faut que les threads partagent le même moniteur. Ici, cela se traduit par la définition, dans la classe des threads, d'un champ "static" pour contenir le moniteur sur lequel on invoquera les méthodes synchronisées.
(cf. `TestThread4.java`)

```
class MoniteurImpression {
    synchronized public void imprime(String t) {
        for (int i=0; i<t.length(); i++) {
            System.out.print(t.charAt(i));
            try { Thread.currentThread().sleep(100); }
            catch (InterruptedException e) {};
        }
    }
}

class TPrint extends Thread {

    String txt;
    static MoniteurImpression mImp = new MoniteurImpression();

    public TPrint(String t) {
        txt = t;
    }

    public void run() {
        for (int j=0; j<3; j++)
            mImp.imprime(txt);
    }
}

public class TestThread4 {
    static public void main(String args[]) {
        TPrint a = new TPrint("bonjour ");
        TPrint b = new TPrint("au revoir ");
        a.start();
        b.start();
    }
}

// résultat de l'exécution :
// bonjour au revoir bonjour au revoir bonjour au revoir
```

3.9.5 Attente explicite : wait() - notify()

Les appels aux méthodes synchronisées sont appelés *sections critiques*. Il faut les utiliser avec prudence :
– elles suppriment la caractère concurrent du traitement d'où une dégradation des performances ;
– elles peuvent conduire à des situations d'interblocage.

Scénario Producteur/Consommateur
– Une mémoire "tampon" est gérée par un producteur d'objets et un consommateur. Cette mémoire ne peut contenir qu'un objet.
– On synchronise l'accès à cette mémoire (production et consommation non simultanées).

– Si le producteur doit déposé un objet alors qu'il en existe déjà un dans la mémoire tampon, il attends ... blocage !
– Même type de blocage pour un consommateur qui attends le dépot d'un objet.

Relâchement d'exclusion

La solution du problème précédent passe par la mise en œuvre d'une procédure de relâche d'exclusion :

– L'attente du Producteur ou du Consommation se fait au moyen de la méthode `wait()` (susceptible de déclencher une exception `InterruptedException`) qui relâche l'exclusion sur l'objet.
– Un processus sort de l'attente lorsqu'un autre processus exécute la méthode `notify()` (qui relance *un* processus en attente) ou la méthode `notifyAll()` (qui relance *tous* les processus en attente)

```
class MoniteurProdCons {
    String tampon;
    boolean estVide = true;

    synchronized void prod(String m) {
        if (!estVide) {
            System.out.println("Producteur attend");
            try { wait(); }
            catch (InterruptedException e) {}
        }
        System.out.println("Produit : "+ m);
        tampon = m; estVide=false; notify();
    }

    synchronized void cons() {
        if (estVide) {
            System.out.println("Consommateur attend");
            try { wait(); }
            catch (InterruptedException e) {}
        }
        System.out.println("Consomme : "+ tampon);
        estVide=true; notify();
    }
}

class Producteur extends Thread {
    MoniteurProdCons tampon;

    public Producteur (MoniteurProdCons t) {
        tampon = t;
    }

    public void run() {
        tampon.prod("message1"); tampon.prod("message2");
        try { sleep(100); }
        catch(InterruptedException e) {}
```

```
        tampon.prod("message3");
    }
}

class Consommateur extends Thread {
    MoniteurProdCons tampon;

    public Consommateur(MoniteurProdCons t) {
        tampon = t;
    }

    public void run() {
        tampon.cons(); tampon.cons(); tampon.cons();
    }
}

public class ProdConsTest {
    public static void main(String argv[]) {
        MoniteurProdCons tampon = new MoniteurProdCons();
        new Producteur(tampon).start();
        new Consommateur(tampon).start();
    }
}
```