2002-2003

UFR Mathématique de la Décision

Deuxième Année de DEUG Science mention MASS

Programmation Objet I

Fabrice Rossi et Fabien Napolitano

Chapitre 1

Rappel sur les objets

Dans ce chapitre nous rappelons quelques notions indispensables pour la suite du cours.

1 Programmation d'une classe : l'exemple des nombres complexes

Par défaut Java ne propose pas de classe permettant de représenter les nombres complexes. Dans cette partie nous allons voir comment écrire une telle classe en rappelant des éléments de programmation vus en première année. Une classe comprend en général les éléments suivants :

- des variables d'instances
- un ou plusieurs constructeurs
- des méthodes d'instance

Les variables d'instances sont propre à chaque objet de la classe : la modification d'une variable d'instance d'un objet ne modifie pas la valeur de la variable correspondante d'un autre objet. Le ou les constructeurs permettent d'initialiser les variables d'instances et en particulier de "construire" les variables d'instance de type objet. Enfin les méthodes d'instance permettent d'utiliser les objets créés. Par ailleurs dans certains cas il est utile de disposer des autres éléments suivants :

- des variables de classe
- des méthodes de classe

Les variables de classe sont partagées par tous les objets de la même classe (en particulier il en existe une seule instance). Les méthode de classe sont en général des méthodes génériques qui n'ont pas besoin d'être appelées par une instance de la classe.

1.1 Variables d'instance

Dans le cas des nombres complexes nous devrons utiliser deux variables d'instance de type `double` : `x` et `y` pour représenter respectivement la partie réelle et imaginaire du nombre complexe. Par ailleurs nous voulons que ces variables soient accessibles directement depuis une autre classe. Nous les déclarerons donc avec le modificateur `public` :

```
public double x ;
public double y ;
```

(pour interdire l'accès depuis une autre classe il suffit de remplacer `public` par `private`).

Remarque 1. En théorie il est recommandé d'avoir le moins possible de variables de type `public` afin de pouvoir modifier par la suite la programmation d'une classe sans modifier l'interface avec les autres classes. Néammoins dans certains cas il est beaucoup plus naturel de permettre l'accés direct et la pratique recommande alors de passer par des variables `public` afin d'éviter la programmation de nombreuses méthodes inutiles (du type `getX()`, `setX(double x)`, etc).

1.2 Constructeurs

Pour construire un nombre complexe nous voulons donner trois possibilités différentes à l'utilisateur : construction du nombre complexe nul avec un constructeur sans paramètre, construction d'un nombre réel pure en utilisant un constructeur avec un paramètre, construction d'un nombre complexe avec un constructeurs à deux paramètres.

```
public Complexe() {
        this.x=0 ;
        this.y=0 ;
}
public Complexe(double x) {
        this.x=x ;
        this.y=0 ;
}
public Complexe(double x,double y) {
        this.x=x ;
        this.y=y ;
}
```

Ici, l'utilisation de `this` permet de spécifier au compilateur que la variable utilisée est la variable d'instance de l'objet appelant. Cela permet d'avoir les meme noms pour les paramètres du constructeur que pour les variables d'instance, ce qui en général est plus clair. **Attention dans cet exemple si vous n'utilisez pas `this` les variables d'instances ne seront pas initialisées !**.

Remarque 2. Si vous ne programmez pas de constructeurs dans la classe `Complexe` (et seulement dans ce cas), Java créera automatiquement un constructeur par défaut sans paramètre. Le constructeur par défaut se contente d'initialiser les variables d'instance à zero (`null` pour les objets le cas échéant).

1.3 Méthodes d'instance

Pour manipulez les nombres complexes il faut programmer des méthodes d'instances. Ces méthodes permettent soit d'agir par effet de bord sur le nombre complexe appelant soit de renvoyer le résultat d'un calcul effectué à partir du nombre complexe appelant et des paramètres. Voici trois exemples de méthodes d'instance permettant respectivement de calculer la somme de deux complexes, de conjuguer un complexe et de calculer le carré de la norme d'un complexe :

```
public Complexe plus(Complexe z) {
        return new Complexe(x+z.x,y+z.y) ;
}
public void conjugue() {
        y=-y ;
}
public double norme2() {
        return x*x+y*y ;
}
```

Vous remarquerez que dans ces méthodes nous n'avons pas utilisé le mot clef `this` pour accéder aux variables d'instance. En effet c'est inutile car il n'y a pas d'ambiguité possible.

1.4 La méthode toString

Parmi les méthodes d'instance il en existe un certain nombre qui sont "universelles". La plupart d'entre elles vous sont inconnus néammoins vous connaissez déja la méthode `toString`. Cette méthode doit **toujours** avoir la signature suivante : `public String toString()`. Son but est de renvoyer une chaine de caractères représentant l'objet appelant. Cette chaine est affichée automatiquement (sans appel explicite à `toString`) lorsque vous utilisez `System.out.print(objet)`. Voici un exemple de programmation de `toString` pour les nombres complexes :

```
public String toString() {
        if (y>0)
                return x+"+ i" +y ;
        if (y<0)
                return x+"- i"+(-y) ;
        return x+"" ;
}
```

1.5 Variable de classe

Parmi les nombres complexes il en est un qui est plus particulièrement utilisé : l'imaginaire pure i. Afin de pouvoir l'utiliser sans avoir à le recréer nous créons une variables de classes contenant i :

```
public static Complexe i=new Complexe(0,1);
```

Cette variable est déclarée de préférence au début de la classe (à coté des variables d'instance). Le mot clef `static` indique que c'est une variable de classe : il en existe une seule instance commune à tous les objets de type complexe. Pour accéder à la variable de classe `i` il faut écrire `Complexe.i`. En général pour accéder à une variable de classe il faut écrire : `NomDeLaClasse.nomDeLaVariable`.

1.6 Méthode de classe

Dans le cas des nombres complexes nous n'avons pas réellement besoin de méthode générique. Néammoins il peut être utile de disposer d'une méthode renvoyant un nombre complexe à partir des coordonnées polaires. Etant donné qu'il est impossible d'écrire un nouveau constructeur prenant en paramètre deux nombres réels nous allons utiliser une méthode de classe :

```
public static Complexe polaire(double r,double theta) {
        return new Complexe(r*Math.cos(theta),r*Math.sin(theta)) ;
}
```

Pour utiliser une méthode de classe il faut écrire `NomDeLaClasse.nomDeLaMethode(...)`.

1.7 Exemple d'utilisation

Au final la classe complexe que nous avons programmée se présente comme suit (vous pouvez bien sur rajouter de nombreuses autres méthodes) :

Complexe

```
1 public class Complexe {
2
3         public double x ;
4         public double y ;
5
6         public final static Complexe i=new Complexe(0,1);
```

```
7
8         public Complexe() {
9                 this.x=0 ;
10                this.y=0 ;
11        }
12        public Complexe(double x) {
13                this.x=x ;
14                this.y=0 ;
15        }
16        public Complexe(double x,double y) {
17                this.x=x ;
18                this.y=y ;
19        }
20        public static Complexe polaire(double r,double theta) {
21                return new Complexe(r*Math.cos(theta),r*Math.sin(theta)) ;
22        }
23        public Complexe plus(Complexe z) {
24                return new Complexe(x+z.x,y+z.y) ;
25        }
26        public void conjugue() {
27                y=-y ;
28        }
29        public double norme2() {
30                return x*x+y*y ;
31        }
32        public String toString() {
33                if (y>0)
34                        return x+" + " +y+" i" ;
35                if (y<0)
36                        return x+" - "+(-y)+" i" ;
37                return x+"" ;
38        }
39}
```

Voici un exemple d'utilisation de la classe `Complexe` (nous avons volontairement programmé la méthode `main` dans une autre classe afin de montrer l'appel correct des variables et méthodes de classe) :

MainComplexe

```
1 public class MainComplexe  {
2
3         public static void main(String[] arg) {
4                 /* construction de a=0, b=1.0, c=1.0+i */
5                 Complexe a=new Complexe() ;
6                 Complexe b=new Complexe(1.) ;
7                 Complexe c=new Complexe(1.,1.) ;
8
9                 /* affichages de a, b, c */
10                System.out.println(a) ;
11                System.out.println(b) ;
```

```
12                  System.out.println(c) ;
13
14                  /* modification des variables d'instances de a */
15                  a.x=-1. ;
16                  a.y=-2. ;
17                  System.out.println(a) ;
18
19                  /* conjugaison de c (effet de bord) et nouvel affichage de c */
20                  c.conjugue() ;
21                  System.out.println(c) ;
22
23                  /*  calcul de c+i et affichage du résultat */
24                  System.out.println(c.plus(Complexe.i)) ;
25
26                  /* stockage de c+i dans a et affichagede a */
27                  a=c.plus(Complexe.i) ;
28                  System.out.println(a) ;
29
30                  /* creation et affichage du nombre complexe 2*e^(i pi/2) */
31                  Complexe d=Complexe.polaire(2,Math.PI/2) ;
32                  System.out.println(d) ;
33
34                  /* affichage de la norme de d au carré */
35                  System.out.println(d.norme2()) ;
36          }
37  }
```

Lorsque l'on exécute ce programme on obtient l'affichage suivant (remarquez l'imprécision dans le calcul de $cos(\pi/4)$) :

```
0.0
1.0
1.0 + 1.0 i
-1.0 - 2.0 i
1.0 - 1.0 i
1.0
1.0
1.2246063538223773E-16 + 2.0 i
4.0
```

2 Rappels sur la manipulation des variables

En Java il existe huit types fondamentaux : `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char` (cf. poly de première année). A partir de ces types fondamentaux, on peut créer par combinaison une infinité de nouveaux types objets. Néammoins la principale différence entre les objets et les types fondamentaux n'est pas que les objets sont composés et les types fondamentaux simples. La principale différence entre les objets et les types fondamentaux est la façon dont ils sont manipulés par la machine virtuelle Java. En particulier nous allons voir que **les variables de type fondamental sont manipulées par valeur** et **les variables de type objet sont manipulées par référence**.

2.1 Gestion de la mémoire

Tout d'abord il est important de comprendre comment la mémoire est gérée en Java. Pour simplifier on peut dire que la mémoire de l'ordinateur est divisée en deux parties : la pile et le tas. Tout ce qui se trouve dans la pile est accessible directement. Au contraire tout ce qui se trouve dans le tas n'est accessible que indirectement en utilisant des références (adresse mémoire) contenues dans la pile. Lors de la création d'une variable, deux cas se présentent :

- si la variable est de type fondamental alors sa valeur est stockée directement dans la pile
- si la variable est de type objet alors son contenu est stocké dans le tas et seule une référence vers le contenu est stockée dans la pile.

Par exemple après le programme suivant :

```
int n=7 ;
Complexe z=new Complexe(1,2) ;
int[] tab=new int[3] ;
```

La mémoire sera organisée comme indiqué sur la figure 1 (rappelez vous que en Java les tableaux sont des objets).

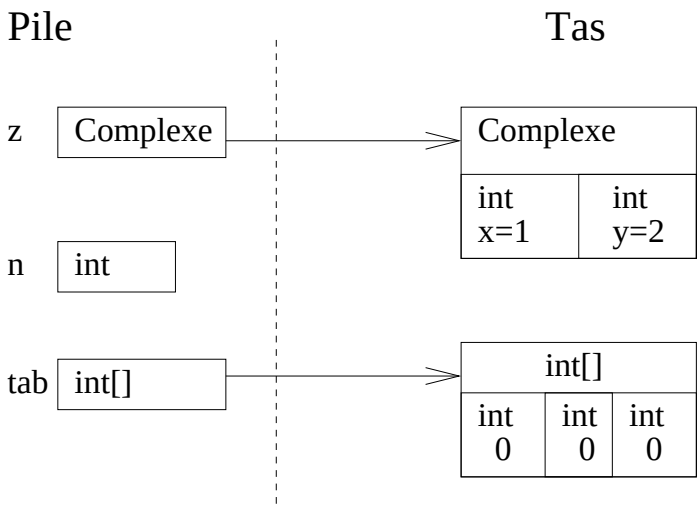


FIG. 1 – Stockage des types fondamentaux et des objets

2.2 Passage de paramètres à une méthode

De la même façon lors du passage d'une variable en paramètre à une méthode, deux cas se présentent :

- si la variable est de type fondamental alors une copie de sa valeur est passée en paramètre à la méthode
- si la variable est de type objet alors une copie de la référence vers cet objet est passée en paramètre à la méthode.

En particulier dans le second cas le contenu de l'objet n'est pas recopié avant le passage à la méthode. Cela explique pourquoi on peut avoir des effets de bord avec les objets et pas avec les variables de type fondamental.

2.3 Exemple : les objets autoréférents

Le principe de manipulation des objets permet d'obtenir des comportements paradoxaux comme les objets autoréférents. Considérons par exemple le programme suivant :

Autoreferent

```
1  public class Autoreferent {
2          public Autoreferent x ;
3
4          public static void main(String[] arg) {
5                  Autoreferent  a=new Autoreferent()  ;
6                  Autoreferent b=new Autoreferent() ;
7                  Autoreferent c=new Autoreferent() ;
8                  a.x=b ;
9                  c.x=c ;
10         }
11 }
```

La classe `Autoreferent` contient une seule variable d'instance également de type `Autoreferent`. Chaque objet de type `Autoreferent` fait donc référence à un autre objet du même type mais à priori distinct. La classe ne comporte pas de constructeur explicite. Donc un constructeur par défaut, sans paramètre, est créé automatiquement. Ce constructeur se contente d'initialiser les variables à `null`. Dans la classe `main` on crée trois objets de type `Autoreferent`. L'objet `a` fait référence à `b`, l'objet `b` ne fait référence à rien et enfin l'objet `c` fait référence à lui même ! On a donc la représentation mémoire donnée figure 2.

Cet exemple peut paraître caricatural. En fait il est très fréquent d'avoir des boucles de références comme ici (par exemple dans des structures de graphes) et il est très important de comprendre ce qui se passe dans un tel cas. Par exemple imaginons que la classe `Autoreferent` contiennent une méthode `toString` faisant elle même appel à la méthode `toString` de la variable d'instance `x`. Dans ce cas l'appel de `toString` par l'objet `c` se traduirait par une boucle infini.

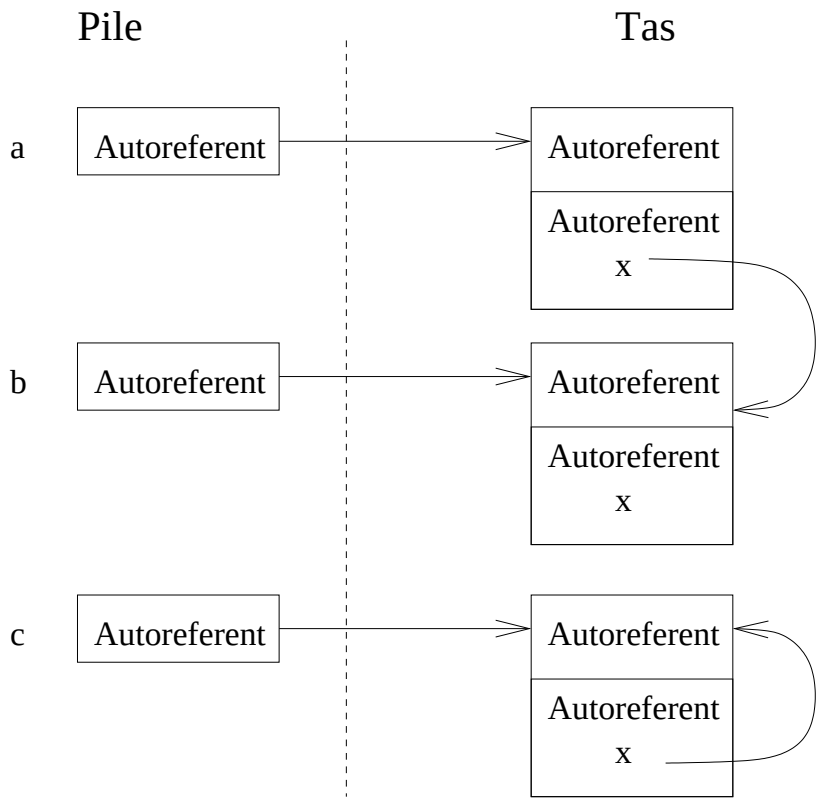


FIG. 2 – Stockage d'objets autoreferents

PROJET I

Dessin de l'ensemble de Mandelbrot

Les ensembles de Mandelbrot et de Julia sont des sous-ensembles fractals du plan complexe. Ils sont définis très simplement à partir de la divergence d'une suite de nombre complexes dépendant d'un paramètre. Dans cet exercice vous allez apprendre à construire ces deux ensembles. Vous pourrez ensuite utiliser les notions de programmations graphiques contenues dans le polycopié de première année afin de les représenter à l'écran.

1 Nombres complexes

Les ensembles de Mandelbrot et Julia étant défini à partir de suite de nombres complexes, il est très pratique de programmer une classe `Complexe` permettant de représenter ces nombres. Chaque nombre complexe est défini par deux variables d'instances de type `double` représentant respectivement la partie réelle et la partie imaginaire. Les fonctions utiles pour la suite devront permettre de calculer la somme, le produit et le carré de la norme d'un nombre complexe. Ces opérations sont bien sur définies d'après les formules suivantes :

$$\begin{array}{rl} \text{somme :} & (x_1 + iy_1) + (x_2 + iy_2) = (x_1 + x_2) + i(y_1 + y_2) \\ \text{produit :} & (x_1 + iy_1)(x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1) \\ \text{carré de la norme :} & |x + iy|^2 = x^2 + y^2 \end{array}$$

Programmez une classe `Complexe` comportant trois méthodes d'instances correspondant aux operations citées ainsi que un constructeur permettant de fixer la partie réelle et la partie imaginaire du nombre complexe.

2 Ensemble de Mandelbrot : définition et algorithme de calcul

Soit z un nombre complexe. On considère la suite u_n, $n \in \mathbb{N}$, de nombres complexes, définie par la récurrence :

$$\begin{array}{l} u_0 = z \\ u_{n+1} = u_n^2 + z \end{array} \tag{1}$$

Par définition z appartient à l'ensemble de Mandelbrot si et seulement si la suite u_n reste bornée (elle ne diverge pas). En général, il est extrèmement difficile de savoir si un nombre appartient à l'ensemble de Mandelbrot (si c'était facile alors l'ensemble de Mandelbrot ne serait pas fractal !). Néammoins il existe un critère simple permettant de savoir en un temps fini si la suite diverge :

Théorème 1. *la suite u_n diverge si et seulement si un des u_n a une norme supérieure à deux.*

En pratique il se peut que la divergence de la suite u_n n'apparaisse pas dans les premiers termes de la suite. Pour savoir si la suite u_n diverge nous utiliserons donc informatiquement l'algorithme décrit dans la table 1. Le paramètre *iteration* correspond approximativement au temps alloués pour le calcul (si *iteration* est multiplié par deux le temps de calcul sera multiplié par deux environ). Il faut donc choisir une valeur de *iteration* assez faible. Néammoins plus *iteration* est faible moins les détails de l'ensemble de Mandelbrot apparaissent. Un bon compromis pour *iteration* est en général de 128 mais tout dépend de la partie du plan complexe que vous voulez representer. Une partie de l'ensemble de Mandelbrot calculée avec ce nombre d'itérations est représentée figure 3.

Données :
– un nombre complexe z dont on veut savoir si il appartient à l'ensemble de Mandelbrot
– un entier *iteration* : le nombre maximum de pas de calcul

Résultat : le nombre n à partir duquel $|u_n| > 2$ ou 0 si $|u_n| < 2$ pour $n \in [0, iteration]$.

Algorithme :
– $n = 0$, $u = z$
– tant que $|u|^2 < 4$ et $n < iteration$
  – ajouter 1 à n.
  – remplacer u par $u^2 + z$
– si $n < iteration$ le résultat est n sinon le résultat est 0.

TAB. 1 – Algorithme de calcul de l'ensemble de Mandelbrot

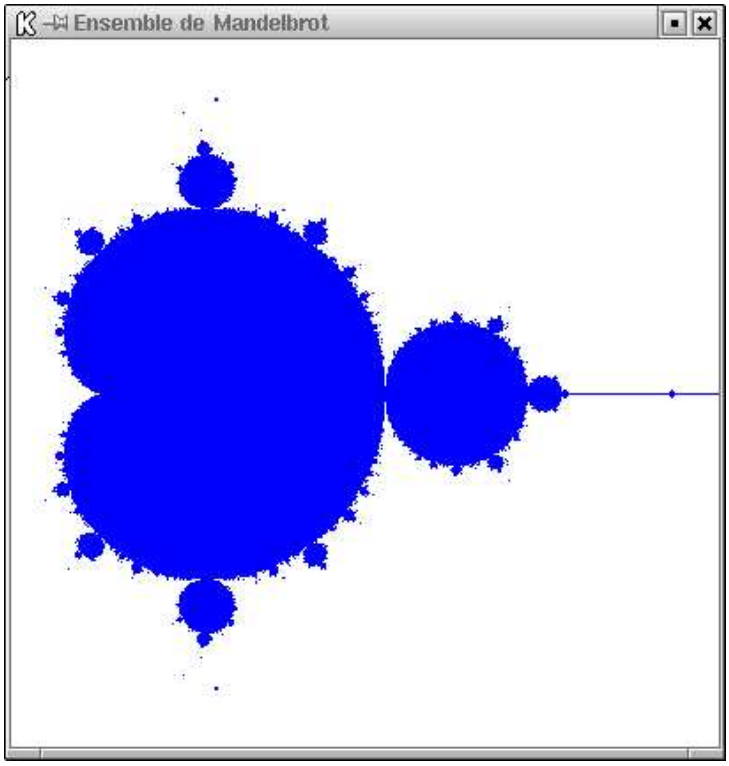


FIG. 3 – Exemple d'affichage donné par le programme Mandelbrot.

3 Programmation

Pour calculer informatiquement l'ensemble de Mandelbrot nous allons creer une nouvelle classe : la classe `Mandelbrot`. Un objet de type `Mandelbrot` sera caractérisé par les paramètres suivant :

– le nombre *iterations* de pas maximal dans la calcul de la suite de la suite u_n pour chaque z
– les coordonnés x_{min},x_{max},y_{min},y_{max} définissant la partie du plan complexe que l'on veut représenter :

$$\begin{aligned} x_{min} \leq \mathrm{Re}\, z \leq x_{max} \\ y_{min} \leq \mathrm{Im}\, z \leq y_{max} \end{aligned} \tag{2}$$

Par ailleurs chaque objet Mandelbrot devra contenir un tableau d'entiers à deux dimensions : `divergence`. Dans un premier temps on suppose que les deux dimensions de `divergence` coincident : `divergence` est un tableau de taille $d \times d$. Les valeurs contenus dans les cases de divergence décrivent le dessin devant apparaître dans la fenêtre graphique : le point de coordonnees k, l de la fenêtre graphique sera représenté en bleu si `divergence[k][l]` est égal à zero et en noir sinon (en particulier les dimensions de la fenêtre graphique représentant l'ensemble de Mandelbrot devront coincider avec les dimensions de `divergence`). Le calcul du tableau `divergence` se fait en utilisant l'algorithme de la table 2. D'après cet algorithme la case k, l de `divergence` contient zero si et seulement si $z_{k,l}$ appartient à l'ensemble de Mandelbrot. En particulier le dessin obtenu dans la fenêtre graphique après calcul de `divergence` est la partie de l'ensemble de Mandelbrot contenu dans le rectangle $x_{min} \leq \mathrm{Re}\, z \leq x_{max}$, $y_{min} \leq \mathrm{Im}\, z \leq y_{max}$ du plan complexe.

Données :
- un entier d (taille du tableau `divergence`)
- les coordonnees x_{min},x_{max},y_{min},y_{max} de la partie du plan complexe que l'on veut représenter
- l'entier *iteration*

Algorithme : Pour chaque entiers k, l, entre 0 et $d-1$ au sens large, on calcul le contenu de la case (k, l) de `divergence` comme suit :
- à (k, l) on associe le nombre complexe $z_{k,l}$ donné par

$$z_{k,l} = x_{max} + \frac{k}{d}(x_{min} - x_{max})) + i(y_{min} + \frac{l}{d}(y_{max} - y_{min}) \tag{3}$$

- dans la case (k, l) de `divergence` on met la valeur obtenu en appliquant à $z_{k,l}$ l'algorithme de la table 1.

TAB. 2 – Calcul des valeurs des cases du tableau divergence

Programmez une classe `Mandelbrot` permettant de représenter l'ensemble de Mandelbrot graphiquement. Vous pourrez ajouter au classes graphiques un écouteur de clavier permettant par exemple de zoomer en avant et en arrière et de se déplacer dans le plan complexe.

4 Ensemble de Julia

La définition et la programmation informatique de l'ensemble de Julia sont très similaire à celles de l'ensemble de Mandelbrot. La différence principale est qu'il n'existe pas qu'un seul ensemble de Julia mais toute une famille dépendant d'un paramètre complexe p.

Soit z un nombre complexe. Pour savoir si z appartient à l'ensemble de Julia on considère la suite u_n, $n \in \mathbb{N}$, définie par récurrence :

$$\begin{aligned} u_0 &= z \\ u_{n+1} &= u_n^2 + p \end{aligned} \tag{4}$$

Comme dans le cas de l'ensemble de Mandelbrot, le point z appartient à l'ensemble de Julia ssi la suite ne diverge pas. La condition pour que cette suite diverge est que l'un des nombres u_n soit de norme supérieure à deux.

Programmez une classe `Julia` permettant de représenter l'ensemble de Julia (pour commencer vous utiliserez comme paramètre $p = -0.7795 + i0.134$). Experimentez l'effet des changements du paramètre. Un exemple de dessin obtenu avec le programme `Julia` est donné sur la figure 4.

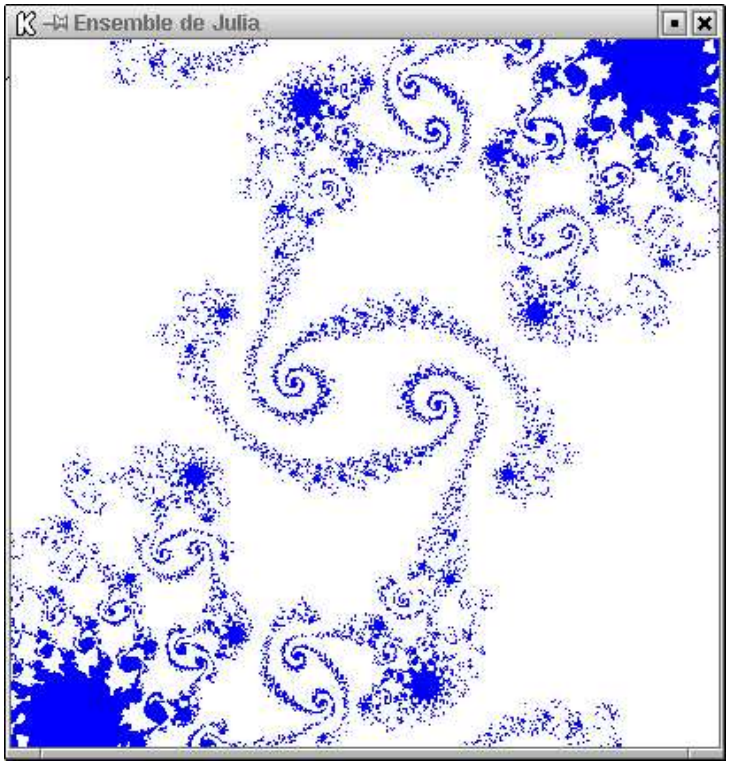


FIG. 4 – Exemple d'affichage donné par le programme Julia.

PROJET II

Reconaissance de caractères par réseau de neurones

Aujourd'hui de nombreux programmes permettent de déchiffrer un texte manuscrit ou de comprendre la parole. Le but de ce problème est de vous donner une idée (très simple) des techniques utilisées en construisant un réseau de neurones capables de reconnaitre des caractères graphiques. Dans la première partie vous programmerez une classe `Caractere` représentant des caractères graphiques. Dans la deuxième partie vous découvrirez le fonctionnement d'un réseau de neurones puis programmerez un réseau capable de reconnaitre les caractères.

1 Caractères graphiques

Un caractère graphique est representé par un rectangle de largeur et longueur données dont chaque point peut être blanc ou noir.

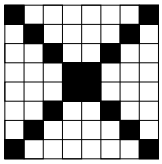


Fig. 5 – Représentation graphique du caractère X

Informatiquement un caractère sera représenté par un tableau d'entiers `pixels` à deux dimensions. Chaque case du tableau vaudra soit 1 si le pixel correspondant est noir, soit -1 si le pixel correspondant est blanc. La classe `Caractere` possèdera donc une variable d'instance `pixel` et quatre variables de classes (*i.e.* commune à tous les objets de la classe) : `largeur=8`, `longueur=8`, `blanc=-1`, `noir=1`.

1. Programmez un constructeur sans paramètre pour la classe `Caractere`. Ce constructeur construit le caractère espace (tous les pixels sont blancs).
2. Ajoutez des méthodes d'instances `void setPixel(int i,int j,intcouleur)` et `int getPixel(int i,int j)` permettant respectivement de modifier la couleur du pixel (i, j) et d'obtenir la couleur du pixel (i, j).
3. Programmez une méthode `String toString()` renvoyant une chaine de caractères représentant le caractère appelant..
4. Dans la méthode `main`, créez le `Caractere` X et affichez le pour vérifier que ça marche.

2 Réseau de neurones de type Hopfield

Les neurones constituant le cerveau sont des cellules nerveuses communiquant les unes avec les autres par l'intermédiaire de synapses. Lors de la communication d'une information par l'intermédiaire des synapses, l'information est amplifiée ou atténuée.

Les réseaux de neurones informatiques on été conçus en prenant modèle sur le fonctionnement du cerveau. L'ensemble des neurones est représenté par un tableau `neurones` de nombres entier, chaque nombre entier représentant l'information contenue dans un neurone. Pour simplifier on peut considérer que chaque neurone contient soit 1 soit -1. Les synapses sont représentés par une matrice carrée `synapse` de nombre entiers. La matrice `synapse` indique la valeur des liaisons entre les neurones. Par exemple si `synapse[i][j]` est un nombre positif les neurones i et j tendront à avoir le même état. Au contraire si `synapse[i][j]` est un nombre négatif les neurones i et j tendront à avoir des état opposés. La classe `Hopfield` représentant un réseau de neurones aura donc deux variables d'instances : `int[] neurones` et `int[][] synapses`. Si le nombre de neurones est n alors la taille de la matrice `synapse` est $n \times n$.

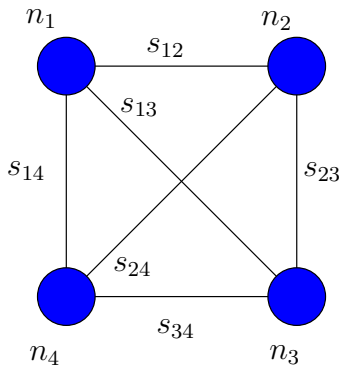


FIG. 6 – Un réseau de quatre neurones

Pour permettre au réseau de reconnaitre un `Caractere` il faut d'abord stocker ce `Caractere` dans l'ensemble des neurones. Chaque pixel composant le `Caractere` est stocké dans un neurone. En particulier il faut autant de neurones que de pixels dans un `Caractere`. La valeur du pixel de coordonnées (i, j) est stocké dans la case $i + j \times largeur$ où $largeur$ est la largeur d'un `Caractere` en pixels.

1. Ecrire un constructeur sans paramètre pour la classe `Hopfield`. Pour trouver le nombre des neurones nécessaires vous utiliserez les variables `largeur` et `longueur` définies dans la classe `Caractere`.
2. Ecrivez des méthodes `void setNeurone(int i,int valeur)` et `int getNeurone()` permettant d'accéder aux informations contenues dans les neurones.
3. Programmez une méthode d'instance `void setCaractere(Caractere c)` permettant de stocker un caractère dans le réseau de neurones. Programmez une méthode `Caractere getCaractere()` permettant d'obtenir le `Caractere` correspondant à l'information stockée dans les neurones du réseau.

La reconnaissance des caractères par le réseau de neurones se déroule en deux phases. Dans la première phase, l'utilisateur apprend au réseau la liste des caractères à reconnaitres. Cela permet de déterminer la liste des coefficients de la matrices `synapse`. Dans la seconde phase l'utilisateur demande au réseau de reconnaitre un caractère mal écrit. Le réseau de neurone utilise alors la matrice `synapse` pour transformer l'information correspondant à ce caractère mal écrit en information correspondant à un caractère bien écrit.

Pour apprendre un `Caractere` on utilise l'algorithme suivant :

Données :

– un `Caractere`

Résultat : apprentissage du `Caractere` par le réseau de neurones.

Algorithme :
- stocker le `Caractere` dans le réseau.
- Ajouter à chaque case (i, j) de la matrice `synapse` la valeur du produit du neurone i par le neurone j.

La deuxième étape de cet algorithme fonctionne de la façon suivante. Si les neurones i et j ont la même valeur (1 ou -1) alors le produit de leurs valeurs est 1 et on renforce la liaison entre les deux neurones. Le réseau cherchera à corréler positivement les neurones i et j lors de l'étape de reconnaissance (il tendront à avoir la même valeur). Si les neurones i et j ont un signe opposé (l'un vaut 1 l'autre -1) alors leur produit est -1 et le réseau cherchera à décorréler les neurones i et j lors de l'étape de reconnaissance.

3. Programmez une méthode d'instance `apprend()` appliquant la deuxième partie de l'algorithme d'apprentissage (modification des coefficients de `synapse`).

La reconnaissance d'un `Caractere` est un peu plus complexe. Chaque neurone va recevoir tour à tour l'information envoyée dans le réseau par les autres neurones et se modifier en fonction de l'information reçue. Pour cela il faut d'abord choisir au hazard l'ordre dans lequel les neurones vont recevoir l'information. Le plus simple est de générer au hazard une permutation des entiers de 1 a n où n est le nombre de neurones.

Données :
- un entier n

Résultat : Un tableau d'entier de taille n contenant les entiers de 1 à n dans le désordre.

Algorithme :
- créer un tableau d'entier `tab` de taille n.
- remplir la case i de `tab` avec la valeur i
- tant que $n - 1 \geq 1$
  - choisir un nombre réel r au hazard entre 0 et 1 (1 non compris).
  - échanger les cases $n - 1$ et $(n - 1)r$ (n'oubliez pas de transformer $(n - 1)r$ en `int`).
  - retirer 1 à n.
- renvoyer le tableau `tab`.

4. Programmez une méthode de classe `int[] shuffle(int n)` appliquant l'algorithme précédent.

L'algorithme de transformation du réseau de neurones due à la transmission de l'information dans le réseau est le suivant :

Données :
- le vecteur `neurone` de taille n et la matrice `synapse`.

Résultat : le vecteur `neurone` modifié par la transformation de l'information.

Algorithme :
- génerer un tableau `ordre` contenant une permutation des entiers de 1 à n.
- pour i allant de 0 à $n - 1$
  - calculer $somme = \sum_{j=0}^{j=n-1} synapses[ordre[i]][j] \times neurones[j]$.
  - si $somme > 0$ remplacer $neurones[ordre[i]]$ par 1.
  - si $somme < 0$ remplacer $neurones[ordre[i]]$ par -1.

5. Programmez une méthode d'instance `void transforme()` appliquant l'algorithme de transformation du réseau.

En général il suffit de transformer une fois le réseau pour reconnaitre un `Caractere` : le nouvel état du réseau correspondant au `Caractere` bien écrit le plus proche du `Caractere` mal écrit qui était stocké dans le réseau. Néammoins il faut parfois plusieurs itérations avant de parvenir à reconnaitre un `Caractere`. Pour savoir quand s'arrèter on utilise l'énergie du réseau donnée par la formule :

$$E = -\sum synapses[i][j] \times neurones[i] \times neurones[j]$$

(pour ceux qui ont fait de la physique c'est pratiquement la même formule que pour l'énergie cinétique)

A chaque tranformation du réseau l'énergie décroit. Quand l'énergie à atteint un minimum alors le réseau n'évolura plus et on peut s'arrêter.

6. Ecrire une méthode d'instance `void reconnaissance()` transformant le réseau tant que l'énergie continue à décroitre.
7. Ecrire une méthode `main` dans laquel vous apprendrez au réseau plusieurs `Caractere`s (par exemple x, +,=,0) que vous essairerez ensuite de lui faire reconnaitre en envoyant au réseau des `Caractere`s mal écrit (par exemple + avec une seule barre).

Chapitre 2

Le type Object

Pour le moment vous avez vu comment réunir plusieurs variables de types distincts dans une même structure en créant de nouvelles classes. Bien que cela soit très pratique, par exemple pour obtenir des programmes plus structurés et plus clairs, cette approche reste très limitée. Par exemple imaginons que vous ayez créé une classe `PaireString` permettant de regrouper deux chaines de caractères dans une même structure (cette classe peut être utile par exemple pour représenter des couples nom/prénom). Par définition votre classe `PaireString` ne pourra pas être utilisée pour regrouper des objets dont le type n'est pas `String`, ne serait ce que des `StringBuffer` par exemple. Si nous voulons utiliser une classe regroupant des paires de `StringBuffer`, il vous faudra donc tout reprogrammer bien que les deux classes soient à l'évidence très proche. Dans ce cas les problèmes suivant risquent d'apparaitre :

- erreurs de recopie (il ne suffit pas de faire un copier coller)
- place sur le disque des diffférentes versions de la classe `Paire`
- toute modification de la classe `PaireString` (par exemple une correction d'erreur) doit être répercuté sur les autres classes `Paire`.

Nous allons voir dans ce chapitre une méthode beaucoup plus élégante pour résoudre ce problème : la classe `Object`.

1 Programmation abstraite de la classe Paire

Imaginons que vous ayez programmée une classe `PaireString` comme indiquée dans l'introduction :

PaireString

```
1  public class PaireString {
2          private String gauche ;
3          private String droite ;
4
5          public PaireString(String gauche,String droite) {
6                  this.gauche=gauche ;
7                  this.droite=droite ;
8          }
9
10         public String getGauche() {
11                 return gauche ;
12         }
13         public String getDroite() {
14                 return droite ;
```

```
15          }
16  }
```

Il est clair que cette classe n'est pas adaptée pour réunir des paires d'objets dont le type n'est pas `String`, ne serait-ce que parceque les variables d'instances sont des `String`. A la place de cette classe nous voudrions programmer une classe capable de contenir n'importe quel objet abstrait. Plus précisemment, nous aimerions écrire une classe `Paire` de la forme suivante (où `Object` représente n'importe quel type objet) :

Paire

```
1   public class Paire {
2           private Object gauche ;
3           private Object droite ;
4
5           public Paire(Object gauche,Object droite) {
6                   this.gauche=gauche ;
7                   this.droite=droite ;
8           }
9
10          public Object getGauche() {
11                  return gauche ;
12          }
13          public Object getDroite() {
14                  return droite ;
15          }
16  }
```

Il se trouve que ce programme est tout à fait valable en Java (vous pouvez verifier qu'il compile sans erreur). **Il existe en Java un type objet plus général que tous les autres types : le type** `Object`. Tous les objets ont le type `Object` en plus de leur type propre : en particulier une variable de type `Object` peut contenir n'importe quel type objet.

Nous pouvons donc utiliser notre classe `Paire` comme suit :

```
String s=new String("Ciceron") ;
String t=new String("Epictète") ;
Paire p=new Paire(s,t) ;
System.out.println(p.getGauche()) ;
```

L'affichage obtenu à l'exécution sera :

```
Ciceron
```

Contrairement à la classe `PaireString` nous pouvons aussi regrouper des `StringBuffer` comme dans le programme suivant :

```
StringBuffer s=new StringBuffer("Ciceron") ;
StringBuffer t=new StringBuffer("Epictète") ;
Paire p=new Paire(s,t) ;
System.out.println(p.getGauche()) ;
```

Enfin nous pouvons mélanger les deux comportements et créer des paires d'objets de types différents comme dans l'exemple suivant :

```
StringBuffer s=new StringBuffer("Ciceron") ;
String t=new String("Epictète") ;
Paire p=new Paire(s,t) ;
System.out.println(p.getGauche()) ;
```

2 Règles d'affectation : transtypage implicite

Les règles d'affectation lors de l'utilisation des `Object`s sont les suivantes :

1. **il est possible de placer dans une variable de type `Object` une référence vers un objet de n'importe quel type** ;
2. **on ne peut placer une référence vers un objet de type `Object` que dans une variable de type `Object`.**

Par exemple le programme suivant est correct :

```
String s="Caton";
Object o=s;
```

Au contraire le programme suivant est incorrect à cause de la dernière ligne :

```
String s="Marc-Aurele";
Object o=s;
s=o; // impossible
```

Remarque 1. A titre de rappel le programme suivant est également incorrect :

```
String s="Seneque";
StringBuffer sb=s;
```

Nous sommes donc en ce qui concerne les objets dans une situation assez proche de celle des types fondamentaux. Dans le cas des types fondamentaux il existe des compatibilités permettant de réaliser des transtypages implicites. Par exemple le morceau de code suivant est correct :

```
int i=2 ;
double d=2.5 ;
```

De la même façon il existe des incompatibilités. Par exemple le morceau de code suivant est incorrect :

```
int i=25.5 ;
```

Il ne faut pas cependant pousser l'analogie avec les types fondamentaux trop loin. En effet, comme pour les types fondamentaux, on recopie le contenu des variables dans la pile, c'est-à-dire, dans le cas des object, les références et non pas les contenus dans le tas. L'exemple suivant illustre graphiquement les différences entre les types fondamentaux et les types objets :

Exemple 1 :

On considère le programme suivant :

```
String s="Diogène";
Object o=s;
int i=12;
double x=i;
```

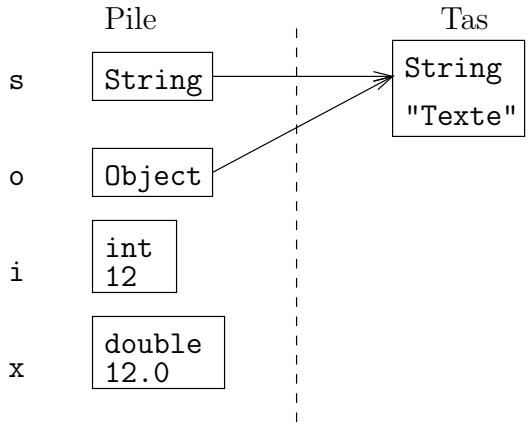


FIG. 1 – Exemple d'utilisation de `Object`

La figure 1 donne l'état de la mémoire après l'exécution des quatre lignes du programme. J'ai volontairement écrit `12` pour la version entière et `12.0` pour le `double` afin de bien montrer que la recopie de `i` dans `x` se traduit par un changement de nature de la valeur. De même, il y a bien deux flèches, c'est-à-dire deux références, l'une dans `s` et l'autre dans `o`. Par contre, ces deux références désignent le *même* objet.

Remarque 2. Il faut bien comprendre qu'une variable de type `Object` ne peut contenir que des références vers des objets, pas des valeurs de type fondamental. Il est par exemple **impossible** d'écrire :

```
Object x=12;
```

3 Règles d'affectation : transtypage explicite

3.1 La demande de conversion

Comme vous l'avez vu dans le polycopié de première année, il existe pour les types fondamentaux, une technique permettant de passer outre les interdictions du compilateur. En effet, on peut par exemple transformer un `double` en un `int`. Cette opération est un **transtypage explicite**[1] et se réalise par exemple de la façon suivante :

```
double x=12.3;
int i=(int)x;
```

En l'absence de `(int)`, l'ordinateur n'accepterait pas la deuxième ligne qui tente de placer une valeur de type `double` dans une variable de type `int`. La présence de `(int)` indique à l'ordinateur de transformer la valeur en une valeur de type `int`.

Remarque 3. Dans les deux cas (de `int` vers de `double` ou dans l'autre sens), il y a conversion. Dans un sens (`int` vers `double`), on ne peut pas perdre d'information, ce qui explique pourquoi l'ordinateur accepte toujours la conversion. Dans l'autre cas, on peut perdre de l'information (c'est le cas de l'exemple précédent, où on perd la partie non entière de `12.3`) et l'ordinateur demande donc qu'on indique explicitement qu'on veut faire la conversion.

Dans le cas des objets, il existe une construction similaire :

[1] ou une *conversion de type*, ou encore un *cast* (en anglais).

```
String s="Lucrece";
Object o=s;
String t=(String)o;
System.out.println(t);
```

Le plus intéressant dans cette conversion, est qu'elle ne perd aucune information : en effet, le programme affiche `Texte`. Au contraire, la conversion numérique perd parfois de l'information, comme le montre l'exemple suivant :

```
double x=12.3;
int i=(int)x;
System.out.println(i);
```

Ce programme affiche bien sûr `12`.

3.2 L'aspect dynamique

Il existe une autre différence très importante entre le transtypage des types fondamentaux et celui des types objets. Considérons en effet le programme suivant :

Bug4

```
1 public class Bug4 {
2   public static void main(String[] arg) {
3     int[] t=new int[] {1,2,3};
4     Object o=t;
5     String s=(String)o;
6   }
7 }
```

Nous savons déjà que la quatrième ligne est correcte, car la règle donnée dans la section précédente indique qu'il est toujours possible de placer une référence vers un objet de n'importe quel type dans une variable de type `Object`. Plus étonnant, la ligne 5 est considérée comme correcte et le programme compile normalement. L'exécution du programme donne l'affichage suivant[2] :

```
java.lang.ClassCastException : java.lang.Object
        at Bug4.main(Bug4.java :5)
```

Pour simplifier, on peut dire que l'ordinateur abstrait détecte la tentative de conversion impossible d'une référence vers un tableau d'entiers en une référence de type `String` et provoque une erreur d'exécution (une *exception*). A l'execution l'odinateur regarde le véritable type de l'objet dans le tas en suivant la référence contenue dans la pile.

Règle simplifiée pour le transtypage explicite : la conversion d'une variable de type `Object` vers un autre type objet est toujours possible syntactiquement et est toujours acceptée à la compilation. Par contre la conversion n'est sémantiquement possible que vers le type réel de l'objet donnée dans le tas. Un essai de conversion vers un autre type que le type réel se traduit par une erreur à l'exécution.

Fort heureusement, il existe une technique permettant de tester si une conversion est possible avant de la faire, ceci afin d'éviter toute erreur : il s'agit de l'opérateur `instanceof`. Cet opérateur permet de tester si le type de l'objet désigné par une référence est compatible avec un type précisé dans le test. Considérons l'exemple suivant :

[2] en utilisant la machine virtuelle de Sun.

```
String s="Parmenide";
Object o=s;
System.out.println(o instanceof Object);
System.out.println(o instanceof String);
System.out.println(o instanceof int[]);
```

Ce programme affiche :

```
true
true
false
```

En effet, l'objet auquel `o` fait référence est de type `String`, donc on peut placer la référence contenue dans `o` dans une variable de type `Object` (comme pour toutes les références) mais aussi dans une variable de type `String` (ce qui demande un transtypage). Par contre, il est impossible de placer une telle référence dans une variable de type `int[]`.

Remarque 4. Pour expliquer le premier affichage il faut se souvenir que tout objet a le type `Object` en plus de son type de base.

Remarque 5. On peut se demander pourquoi l'ordinateur demande d'indiquer explicitement une opération de conversion si de toute manière il va faire une vérification à l'exécution du programme.

Il faut d'abord comprendre que la vérification à **l'exécution** est indispensable. Les exemples proposés ici sont simples, mais il est facile d'écrire un programme pour lequel l'objet désigné par une référence contenue dans une variable dépend d'une décision de l'utilisateur (voir l'exemple 2 : dans ce cas, il est impossible de savoir à la **compilation** du programme si la conversion sera possible ou non.

D'autre part, il est normal que l'ordinateur demande d'indiquer explicitement les conversions car celles-ci peuvent rendre le programme incorrect : un programme avec conversion peut très bien ne pas s'exécuter correctement. L'ordinateur vérifie donc simplement que vous n'avez pas fait une erreur de programmation et que vous êtes bien conscient des erreurs possibles dans votre programme.

Exemple 2 :

Voici un exemple simple de programme dans lequel le type de l'objet désigné par une référence dépend d'une décision de l'utilisateur :

TypeUtilisateur

```
1  import dauphine.util.*;
2  public class TypeUtilisateur {
3    public static void main(String[] arg) {
4      System.out.println("Donner un entier");
5      int i=StandardInput.readInt();
6      Object x;
7      if(i>0)
8        x="Toto";
9      else
10       x=new StringBuffer("Toto");
11     String s=(String)x;
12   }
13 }
```

Suivant la valeur entrée par l'utilisateur, ce programme s'exécute correctement ou non.

Il est important de noter que pour les types fondamentaux comme pour les types objets, certaines tentatives de conversion sont rejetées à la **compilation**. Ce sont les tentatives pour lesquelles le compilateur peut assurer qu'elles ne pourront jamais réussir, quel que soit le contenu des variables intervenant dans la conversion. Pour les types fondamentaux, seules les conversions numériques sont possibles, ce qui interdit les lignes suivantes :

```
boolean b=true;
double x=(double)b;
```

Il faut cependant garder à l'esprit que `char` est un type numérique un peu particulier et que la conversion suivante est correcte :

```
int i=2323;
char c=(char)i;
```

Pour les types objets, le problème est plus simple : la référence à convertir possède un certain type (déterminé à la compilation). Une variable de ce type peut éventuellement contenir une référence vers un objet d'un autre type (si la référence est de type `Object`, une variable de ce type peut désigner n'importe quel objet, par exemple). On ne peut tenter une conversion que si type demandé fait partie des types possibles. On sait par exemple qu'une variable de type `StringBuffer` ne peut contenir qu'une référence vers un objet de type `StringBuffer`. De ce fait, le compilateur rejette la deuxième ligne du programme suivant :

```
StringBuffer sb=new StringBuffer("");
int[] t=(int[])sb;
```

En fait, pour l'instant, nous ne pouvons que transtyper des références de type `Object`. Nous verrons par la suite de nombreux autres cas.

4 Méthodes de la classe `Object`

Dans de nombreuses circonstances, nous allons manipuler des références de type `Object`. Il est donc important de connaître les méthodes principales définies par la classe `Object`. Notons que s'il est possible de créer des instances de cette classe (grace au constructeur par défaut), cela ne présente en général rigoureusement aucun intérêt. En pratique l'intérêt des méthodes de la classe `Object` est de pouvoir être appelées automatiquement par héritage comme nous allons le voir succintement. Voici donc les méthodes principales de cette classe :

`String toString()`
: Transforme l'objet appelant en une chaîne de caractères. Cette chaîne est obtenue en indiquant le nom de la classe de l'objet, suivi du signe `@`, suivi du code de hachage de l'objet (voir méthode suivante), exprimé en base 16.

`int hashCode()`
: Renvoie un code de hachage représentant l'objet. Ce code est "le plus unique possible", c'est-à-dire qu'on espère que deux objets distincts auront deux codes différents. Grâce à ce code, on peut stocker l'objet dans une table de hachage, technique de stockage évoluée dont la présentation dépasse le cadre de ce cours.

`boolean equals(Object obj)`
: Compare l'objet appelant avec l'objet paramètre et renvoie `true` si et seulement si les deux objets sont *strictement identiques* c'est-à-dire que la référence appelante est égale à la référence paramètre.

`Class getClass()`

Renvoie un objet de type `Class` qui décrit la classe de l'objet appelant.

Considérons l'exemple suivant :

Exemple 3 :

MethodesObject

```
1  public class MethodesObject {
2    public static void main(String[] arg) {
3      int[] t={1,2,3};
4      Object o=t;
5      System.out.println(o.toString());
6      System.out.println(o.hashCode());
7      System.out.println(o.equals(t));
8      System.out.println(o.equals("toto"));
9    }
10 }
```

L'affichage produit est le suivant[3] :

```
[I@80d8219
135103001
true
false
```

La notation `[I` désigne le type objet `int[]` en `Java`.

L'intérêt des méthodes de la classe `Object` vient du fait que en java tous les objets héritent de la classe `Object`. En particulier tout objet Java possède par défaut les méthodes de la classe `Object`. Par ailleurs si un objet reprogramme une méthode de la classe `Object` ,*a*vec exactement la meme signature, alors lors de l'appel de méthode dynamique (à l'exécution), la méthode appelée sera la nouvelle méthode.

Exemple 4 :

Tous les objets possèdent par défaut une méthode `toString()`. Admettons que l'ordinateur abstrait soit capable d'afficher une chaîne de caractères donnée sous la forme d'un objet de type `String`. Nous savons que la méthode `println` peut être utilisée avec n'importe quel paramètre de type objet. Ceci n'est possible que si cette méthode demande en fait un paramètre de type `Object`. Voici donc comment la méthode `println` pourrait être programmée (il s'agit bien sûr d'une simplification de la réalité) :

```
public void internalPrintString(String s) {
        // méthode complexe qui affiche une chaîne de caractères
}
public void println(Object obj) \{
        if (obj==null)
                internalPrintString("null");
        else
                internalPrintString(obj.toString());
}
```

Il ne reste plus rien de magique dans cette écriture. Il s'agit simplement d'une utilisation de `Object`.

[3] le code de hachage n'est pas nécessairement toujours le même et peut varier *pour un même programme* selon l'ordinateur qui exécute le programme, et selon d'autres paramètres.

Exemple 5 :

Tous les objets possèdent par défaut une méthode `equals` qui par défaut compare les références. Par exemple le morceau de programme suivant affiche `false` bien que les nombres complexes `z1` et `z2` soient égaux.

```
Complexe z1=new Complexe(1,1) ;
Complexe z2=new Complexe(1,1) ;
System.out.println(z1.equals(z2)) ;
```

Néammoins il est très facile de modifier ce comportement : il suffit de reprogrammer la méthode `equals` dans la classe `Complexe` (attention à bien respecter la signature) :

```
public boolean equals(Object obj) {
        if (obj instanceof Complexe) {
                Complexe parametre=(Complexe)obj ;
                return (x==parametre.x && y==parametre.y) ;
        }
        return false ;
}
```

Une fois cette méthode inclue dans la classe `Complexe` le programme précédent affichera `true` car cette fois la nouvelle méthode `equals` sera utilisée.

A l'exécution, lors d'un appel de méthode d'instance c'est toujours le "vrai" type de l'objet appelant qui est utilisé. Plus précisémment lors d'un appel de méthode l'ordinateur regarde dans le tas le type de l'objet appelant (ce type peut être différent du type de la variable faisant référence à cet objet dans le tas). Puis l'ordinateur regarde si une méthode d'instance avec la signature appropriée a été définie dans la classe correspondant à ce type. Si c'est le cas cette méthode est appelée. Sinon c'est la méthode correspondante de la classe `Object` qui est appelée par défaut.

Exemple 6 :

Une fois programmée la méthode `equals` dans la classe `Complexe` le programme suivant affiche `true` :

```
Object z1=new Complexe(1,1) ;
Object z2=new Complexe(1,1) ;
System.out.println(z1.equals(z2)) ;
```

PROJET III

Tableau dynamique d'Object

En Java la taille d'un tableau ne peut pas changer en cours de programme. Ce comportement peut être très génant par exemple quand on ne connaît pas à l'avance le nombre d'éléments que l'on devra stocker dans le tableau. Les tableaux dynamiques permettent d'apporter une réponse à ce problème car leur taille peut changer au cours du programme. Java propose en standard une classe `ArrayList` programmant les tableaux dynamiques. Le but de ce problème est de programmer une classe `TableauDynamique` similaire à la classe `ArrayList`.

1 Programmation

Un tableau dynamique comporte deux variables d'instances : une variable entière `taille` indiquant le nombre d'objet stockés et une variable `memoire` de type tableau d'`Objects` contenant l'ensemble des éléments du tableau. Lorsque l'utilisateur veut aggrandir la taille du tableau il suffit de remplacer le tableau `memoire` par un autre tableau plus grand dans lequel on a recopié les éléments de `memoire`. En général le tableau `memoire` comporte plus de cases que le nombre d'objets stockés effectivement. Les cases libres servent à stocker de nouveaux objets. Cela permet de ne pas avoir à recopier le tableau `memoire` trop souvent. Pour l'utilisateur de la classe `TableauDynamique` tout doit être "transparent" : c'est à dire que l'utilisateur n'a jamais accès directement aux variables d'instance, au contraire toutes les manipulations se font automatiquement par l'intermédiaire des méthodes d'instances. La représentation mémoire d'un tableau dynamique est donnée figure 2.

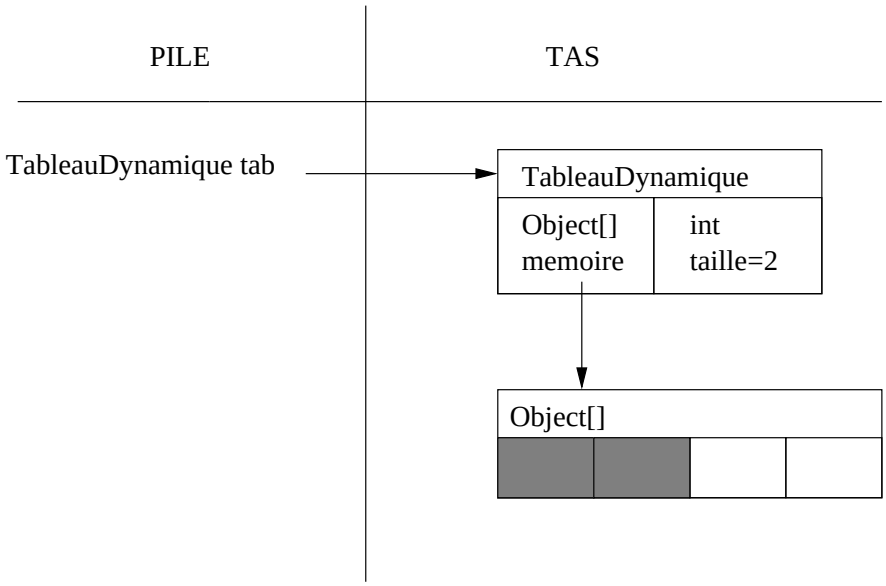


Fig. 2 – Représentation mémoire d'un tableau dynamique

1. Programmez un constructeur sans paramètre pour la classe `TableauDynamique`. Ce constructeur initialise les variables `taille` et `memoire`. Au début le tableau ne contient aucun objet

et donc $taille = 0$. Par contre il est judicieux de réserver dés le début plusieurs cases mémoires pour stocker les éléments ajoutés par l'utilisateur. Pour cela vous ajouterez à la classe `TableauDynamique` une constante de classe entière `tailleMemoireInitiale=10` que vous utiliserez pour creer le tableau `memoire`.

2. Ajoutez des méthodes `Object get(int i)` et `Object set(int i,Object valeur)` permettant respectivement d'obtenir le i-ème élément du tableau où de le modifier (la méthode `set` renvoie l'élément qui se trouvait à la i-ème position avant la modification. Pour chacune de ces méthodes vous vérifierez que $0 <= i < taille$. Si ce n'est pas le cas la méthode ne fait rien et renvoie `null`.
3. Programmez une méthode `int size()` renvoyant le nombre d'objets contenus dans le tableau dynamique.
4. Programmez une méthode `boolean isEmpty()` renvoyant `true` si et seulement si le tableau dynamique ne contient aucun élément.
5. Programmez une méthode `void ensureCapacity(int nouvelleTailleMemoire)` permettant de modifier la taille mémoire disponible pour stocker de nouveaux éléments.
6. Programmez une méthode `void add(Object obj)` ajoutant un nouvel objet à la fin du tableau dynamique. Avant d'ajouter un nouvel objet dans le tableau dynamique vous vous assurerez que la taille du tableau `memoire` est assez grande ; si ça n'est pas le cas
7. En s'inspirant de la question précédente, écrire une méthode `void add(int i,Object obj)` ajoutant l'objet `obj` à la i-ème position du tableau dynamique (en particulier tous les éléments après la i-ème position sont décalés d'un cran).
8. Programmez une méthode `Object remove(int position)` supprimant l'objet situé dans la case `position` du tableau dynamique (pour supprimer la i-ème case il suffit de recopier les éléments de positions $i+1$ à $taille-1$ dans les cases i à $taille-2$, puis de retirer 1 à `taille`). La méthode `remove` doit renvoyer l'objet supprimé.
9. Programmez une méthode `String toString()` renvoyant une chaine de caractères représentant le tableau appelant
10. Programmez une méthode `equals` permettant de comparer deux tableaux dynamiques (les éléments des deux tableaux seront également comparés avec la méthode `equals`).

2 Correction

Voici un exemple de programmation pour la classe `TableauDynamique` :

TableauDynamique

```
1  public class TableauDynamique {
2          private int taille ;
3          private Object[] memoire ;
4          private static int tailleMemoireInitiale=10 ;
5
6          public TableauDynamique() {
7                  memoire=new Object[tailleMemoireInitiale] ;
8          }
9
10         public Object get(int i) {
11                 if (i<0 || i>taille-1) return null ;
12                 return memoire[i] ;
```

```
13      }
14
15      public Object set(int i,Object valeur) {
16              if (i<0 || i>taille-1) return null ;
17              Object retour=get(i) ;
18              memoire[i]=valeur ;
19              return retour ;
20      }
21
22      public int size() {
23              return taille ;
24      }
25
26      public boolean isEmpty() {
27              return taille==0 ;
28      }
29
30      public void ensureCapacity(int nouvelleTailleMemoire) {
31              if (nouvelleTailleMemoire<=memoire.length) return ;
32              Object[] nouveau=new Object[nouvelleTailleMemoire] ;
33              for (int i=0;i<taille;i++)
34                      nouveau[i]=memoire[i] ;
35              memoire=nouveau ;
36      }
37
38      public void add(Object obj) {
39              if (taille+1>memoire.length)
40                      ensureCapacity(2*(taille+1)) ;
41              memoire[taille]=obj ;
42              taille++ ;
43      }
44
45      public void add(int i,Object obj) {
46              if (taille+1>memoire.length)
47                      ensureCapacity(2*(taille+1)) ;
48              for (int j=taille-1;j>=i;j--)
49                      memoire[j+1]=memoire[j] ;
50              memoire[i]=obj ;
51              taille++ ;
52      }
53
54      public Object remove(int position) {
55              Object retour=memoire[position] ;
56              for (int i=position;i<taille-1;i++)
57                      memoire[i]=memoire[i+1] ;
58              taille-- ;
59              return retour ;
60      }
61
```

```
62          public String toString() {
63                  if (taille==0) return "{}" ;
64                  StringBuffer sb=new StringBuffer("{") ;
65                  for (int i=0;i<taille-1;i++)  {
66                          sb.append(memoire[i]) ;
67                          sb.append(",") ;
68                  }
69                  sb.append(memoire[taille-1]) ;
70                  sb.append("}") ;
71                  return sb.toString() ;
72          }
73
74          public boolean equals(Object obj) {
75                  if (obj instanceof TableauDynamique) {
76                          TableauDynamique parametre=(TableauDynamique)obj ;
77                          if (parametre.size()!=size())
78                                  return false ;
79                          for (int i=0;i<size();i++)
80                                  if (!(parametre.get(i).equals(get(i))))
81                                          return false ;
82                          return true ;
83                  }
84                  return false ;
85          }
86  }
```

3 Exemple d'application

Voici un exemple d'utilisation (essayez de deviner ce que va afficher ce programme puis exécutez le pour vérifier) :

TestTableauDynamique

```
1   public class TestTableauDynamique {
2           public static void main(String[] arg) {
3                   TableauDynamique tab=new TableauDynamique() ;
4                   tab.add("Aristophane") ;
5                   tab.add(new StringBuffer("Sophocle")) ;
6                   System.out.println(tab.size()) ;
7                   tab.add(new int[] {1,2,3}) ;
8                   tab.add("Euripide") ;
9                   tab.add("Eschyle") ;
10                  System.out.println(tab.size()) ;
11                  System.out.println(tab) ;
12                  for (int i=0;i<5;i++)
13                          tab.add(tab.get(i)) ;
14                  System.out.println(tab) ;
15                  for (int i=0;i<5;i++)
16                          tab.remove(0) ;
17                  System.out.println(tab) ;
```

```
18          }
19  }
```

PROJET IV

Liste chainée simplifiée

Le principal avantage des tableaux dynamiques, à savoir la possibilité de changer la taille du tableau en cours de programme, s'accompagne d'un désavantage important : l'ajout ou le retrait d'un élément du tableau prend en moyenne un temps proportionnel au nombre d'éléments stockés. En effet ces deux opérations entrainent la recopie d'une grande partie des éléments du tableau (soit dans un nouveau tableau pour augmenter la mémoire de stockage, soit dans le même tableau pour décaler les éléments suivants l'élément supprimé). Les listes chainées sont une autre structure de donnée permettant de manipuler des tableaux de taille variables, dont les avantages et désavantages sont symétriques de ceux des tableaux dynamiques : l'ajout ou le retrait d'un élément au début ou à la fin d'une liste chainée prend un temps constant tandis que l'accès à un élément du tableau prend un temps proportionel au numéro de cet élément (dans le cas des tableaux dynamiques le temps d'accès à un élément est constant : il suffit de calculer l'adresse mémoire de cet élément). Java propose en standard une classe `LinkedList` implentant la structure de liste chainée. Le but de ce problème est de programmer une classe `ListeChainee` ayant le même comportement.

1 Principe des listes chainées

Dans un tableau dynamique toutes les cases sont regroupées à la suite les unes des autres dans la mémoire de l'ordinateur ce qui implique de recopier l'ensemble des cases lorsque l'on veut agrandir le tableau. Dans une liste chainee au contraire chaque case peut être placée n'importe où dans la mémoire indépendamment des autres cases. Ainsi pour ajouter ou retirer une case il n'est pas besoin de modifier les autres cases. Pour s'y retrouver dans cet ensemble de cases, chaque case possède deux liens : un lien vers la case précédente et un liens vers la case suivante. Ainsi l'ensemble des cases forme une chaine comme dessiné sur la figure 3 (en fait il serait plus orthodoxe de parler dans ce cas de liste doublement chainées).

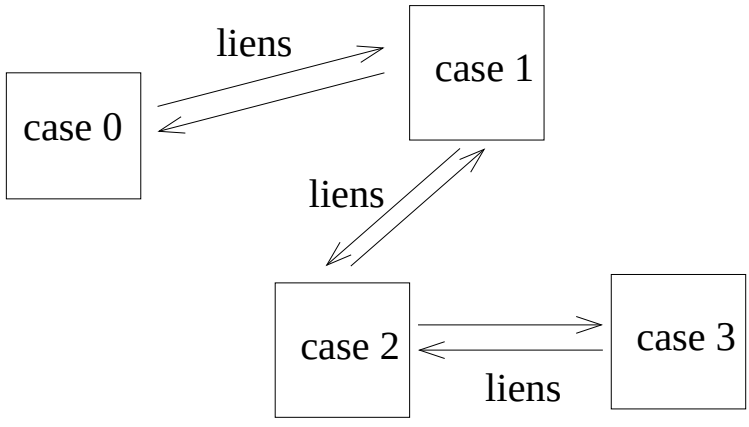


Fig. 3 – Représentation mémoire d'une liste chainée

Pour programmer les listes chainées nous aurons donc besoin de deux classes : une classe `CaseChainee` programmant les cases avec leurs liens, une classe `ListeChainee` permettant de manipuler l'ensemble de la chaine.

2 Programmation de la classe CaseChainee

La classe `CaseChainee` possède trois variables d'instance : une variable `contenu` de type `Object` permettant de stocker un éléments dans la case, deux variables `precedente` et `suivante` de type `CaseChainee` faisant référence aux cases précédentes et suivantes de la liste. Pour faciliter la programmation de la classe `ListeChainee`, on pourra supposer que ces trois variables sont de type `public` ; c'est à dire que l'on peut les manipuler depuis n'importe qu'elle classe Java. A la création les variables d'instances ne sont pas initialisées. En particulier vous n'écrirez pas de constructeur mais vous utiliserez le constructeur par défaut.

1. Ecrivez le squelette de la classe `CaseChainee`.
2. Programmez une méthode d'instance `void chaineSuivante(Object obj)` permettant de chainer une nouvelle case contenant l'objet `obj` à la suite de la case appelante. L'algorithme de cette méthode est le suivant (voir figure 4) :

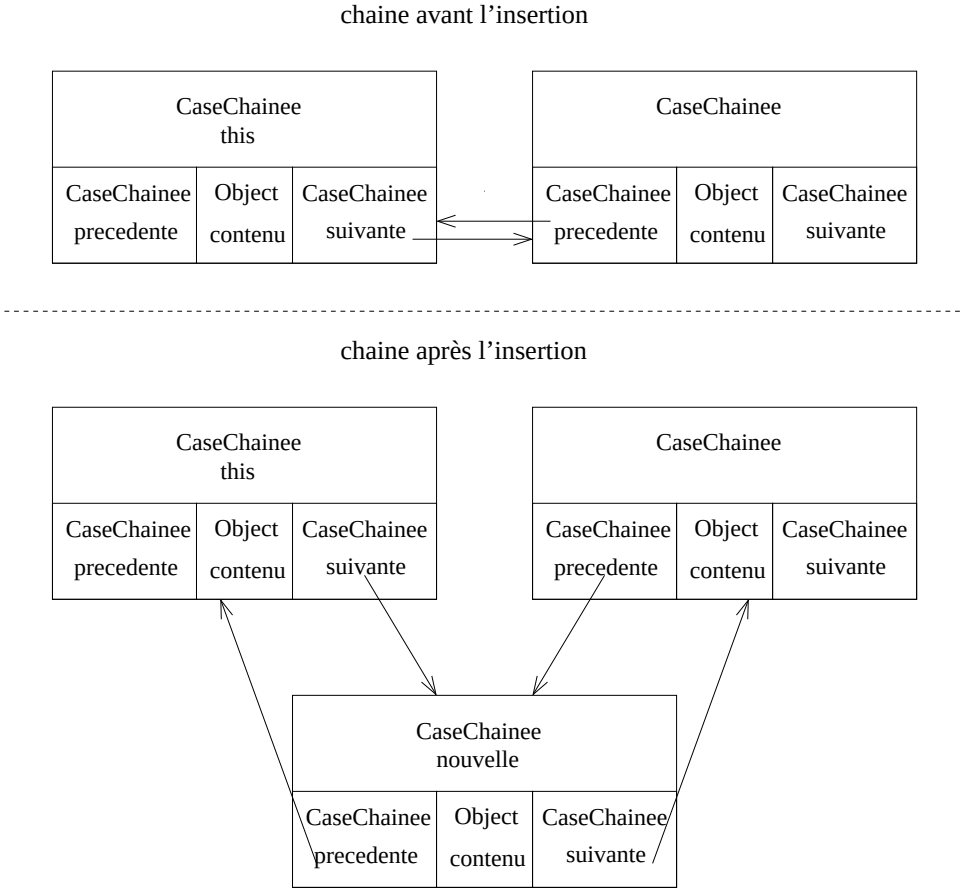


FIG. 4 – Insertion d'une nouvelle cellule à la suite de this

Données :
– une `CaseChainee this` (objet ayant appelé la méthode)
– un `Object obj`.

Résultat : Une nouvelle case, contenant l'objet `obj`, est chainée à la case appelante.

Algorithme :
– créer une `CaseChainee nouvelle`.

- stocker `obj` dans le contenu de la case `nouvelle`
- faire pointer la référence `suivante` de la case `nouvelle` au même endroit que la référence `suivante` de la case `this`.
- faire pointer la référence `precedente` de la case `nouvelle` vers `this`.
- si la référence `suivante` de `this` n'est pas égale à `null`, faire pointer la référence `precedente` de `suivante` vers `nouvelle`.
- faire pointer la référence `suivante` de `this` vers `nouvelle`.

3. En s'inspirant de l'algorithme précédent, ecrivez une méthode `void chainePrecedente(Object obj)` permettant de chainer une nouvelle case contenant l'objet `obj` avant la case appelante.
4. Programmez une méthode d'instance `Object supprime()` permettant de supprimer la case appelante de la chaine dans laquelle elle se trouve. L'algorithme de cette méthode est le suivant :

   Données :
   - une `CaseChainee` `this` (objet ayant appelé la méthode)

   Résultat : La case `this` est supprimée de la chaine dont elle faisait partie. La chaine est reformée avec les cases restantes : la case précédente de `this` est chainée à la case suivante de `this`. La méthode renvoie l'objet contenu dans la case supprimée.

   Algorithme :
   - stocker le contenu de la case appelante dans une nouvelle variable de type `Object` `resultat`.
   - remplacer le contenu de `this` par la référence `null`.
   - si la case précédente de `this` ne désigne pas la référence `null` alors la chainer avec la case suivante de `this` (la case suivante de la case précédente de `this` devient la case suivante de `this` comme indiqué sur la figure 5).
   - procéder de même pour chainer la case suivante de `this` avec sa case précédente.
   - remplacer les références `suivante` et `precedente` de `this` par `null`.
   - renvoyer la variable `resultat`.

3 Programmation de la classe ListeChainee

Le principe de la classe `ListeChainee` est le suivant : chaque `ListeChainee` possède deux variables d'instances `debut` et `fin` de type `CaseChainee`. Ces deux variables représentent des cases vides dont le seul intérêt est de permettre d'accéder aux autres cases de la liste (respectivement en partant du début ou de la fin de la liste de cases). En particulier ces cases ne seront jamais visibles pour l'utilisateur et elles ne seront jamais modifiées après la création de la liste. Par ailleurs chaque `ListeChainee` possède une variable d'instance `taille` indiquant le nombre d'objets stockés dans la liste. Cette variable sera mise a jour chaque fois que l'on ajoutera ou supprimera un élément de la liste.

1. Ecrire un constructeur sans paramètre pour `ListeChainee`. Ce constructeur doit construire les deux `CaseChainees` `debut` et `fin` puis les chainees afin que la case suivante de `debut` soit `fin` et que la case précédente de `fin` soit `début` (voir figure 6).
2. Ecrire une méthode `int size()` renvoyant le nombre d'objet stockés dans la liste.
3. Ecrire une méthode `boolean isEmpty()` renvoyant `true` si et seulement si la liste est vide.
4. Programmez une méthode `void addFirst(Object obj)` permettant d'ajouter un élément au début de la liste. Pour cela il suffit de chainer à la suite de la case `debut` une nouvelle case contenant l'objet `obj` puis d'ajouter un à `taille` (vous utiliserez bien sur la méthode `chaineSuivante` de la classe `CaseChainee`).

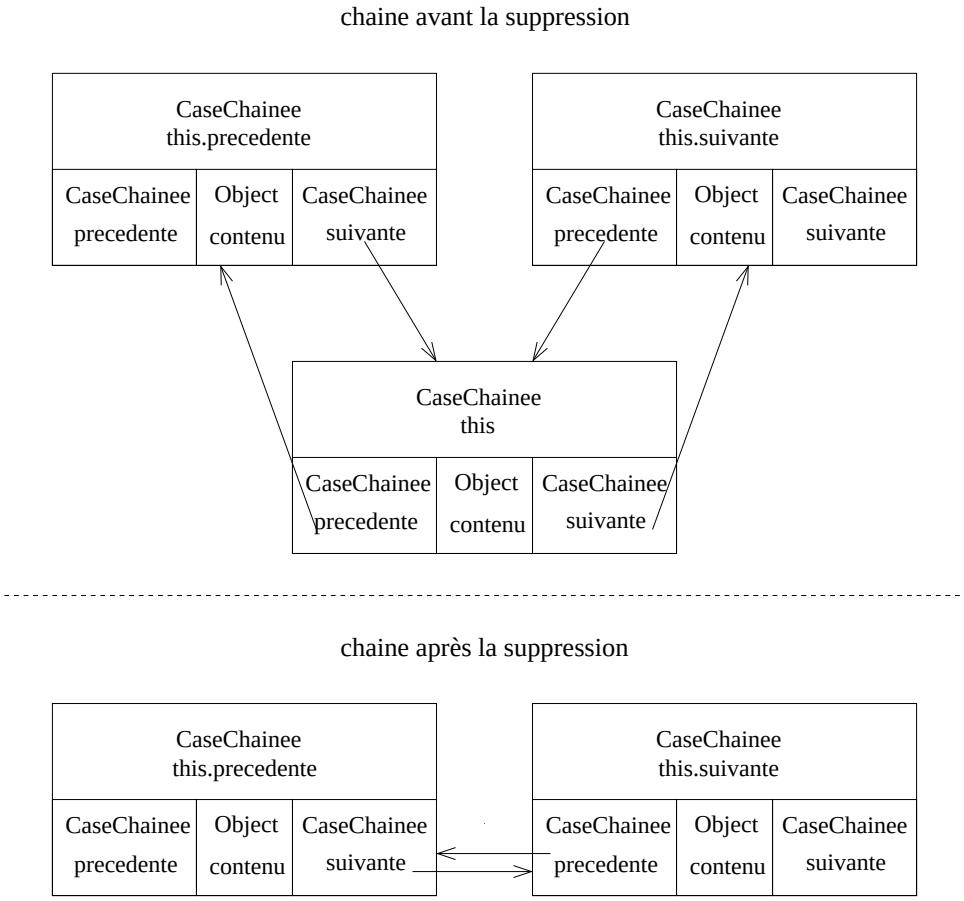


FIG. 5 – Supppression de la cellule this

5. Programmez une méthode `void addLast(Object obj)` permettant d'ajouter un objet à la fin de la liste.
6. Programmez une méthode `Object getFirst()` renvoyant le premier objet de la liste ; c'est à dire le contenu de la case suivante de `debut`.
7. Programmez une méthode `Object getLast()` renvoyant le dernier élément de la liste.
8. Programmez une méthode `Object removeFirst()` supprimant le premier élément de la liste (si la liste n'est pas vide) et renvoyant cet élément (ou `null` si la liste est vide). Pour cela vous utiliserez la méthode `supprime()` de la classe `CaseChainee`. N'oubliez pas de modifier la valeur de `taille` afin de tenir compte de la nouvelle taille de la liste.
9. Programmez une méthode `Object removeLast()` supprimant le dernier élément de la liste (si la liste n'est pas vide) et renvoyant l'élément supprimé.
10. Programmez une méthode `void add(int position,Object obj)` permettant d'ajouter l'objet `obj` à la position donnée dans la liste. L'algorithme de cette méthode est le suivant :

    Données :
    – une `ListeChainee this` (la `ListeChainee` appelante)
    – un entier `position`
    – un objet `obj`

    Résultat : L'objet `obj` est ajoutée à la position donnée dans la liste `this`.

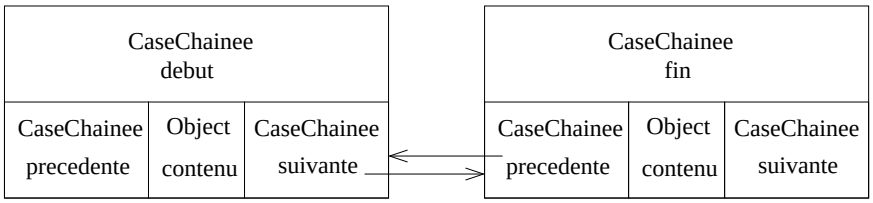


FIG. 6 – Représentation mémoire d'une ListeChainee vide

Algorithme :
- si *position* < 0 ou *position* $>$ *taille* il n'y a rien à faire.
- Creer une nouvelle variable `temporaire` de type `CaseChainee` faisant référence à la case `debut` de `this`.
- Répeter `position` fois :
  - remplacer `temporaire` par une référence vers la case suivante de `temporaire`.

  La variable `temporaire` pointe maintenant vers la case *position* $- 1$ de la liste chainée.
- chainer à la suite de temporaire une nouvelle `CaseChainee` contenant l'objet `obj`.
- ajouter un à la taille de `this`.

11. En s'inspirant de l'algorithme donné dans la question précédente programmez une méthode `Object get(int i)` renvoyant le contenu de la i-ème case de la liste appelante. De la même façon programmez une méthode `Object set(int i,Object obj)` permettant de modifier le contenu de la i-ème case de la liste appelante. Enfin programmez une méthode `Object remove(int i)` supprimant la i-ème case de la liste appelante. Les valeurs de retours de ces méthodes seront déteminées comme dans le cas des méthodes `getFirst`, `setFirst`, `removeFirst`.
12. Programmez une méthode `String toString()` permettant d'afficher une liste chainée (cette méthode renvoie la représentation de la liste sous forme de chaîne de caractères).
13. Programmez une méthode `equals` permettant de comparer deux listes chainées.

Chapitre 3

Listes génériques

Comme nous l'avons vu l'utilisation des propriétés de `Object` permet de créer une classe tableau dynamique dont les instances peuvent "contenir[1]" des objets de n'importe quel type. Ceci est extrêmement pratique pour ne pas avoir à reprogrammer une telle classe pour chaque nouvelle application demandant l'utilisation de tableaux dynamiques.

Dans la pratique, la situation est encore plus favorable. En effet, `Java` définit (à partir de la version 1.2 du kit de développement proposé par `Sun`[2]) un ensemble de classes **listes génériques**. Une classe *liste* est une classe dont les instances sont capables de contenir un nombre quelconque de références vers d'autres objets. Une telle classe est générique si les références peuvent être de n'importe quel type. En d'autres termes, `Java` définit des équivalent des tableaux dynamiques et des listes chainées. Le but de cette section est de découvrir ces équivalents.

1 La classe `ArrayList`

La classe `ArrayList` est la version incluse dans `Java` des tableaux dynamiques étudiés précédemment. Cette classe est bien entendu plus sophistiquée que la classe `TableauDynamique` mais son principe reste le même : on peut dire grossièrement qu'une instance de `ArrayList` contient un `Object[]` qui grossit quand on ajoute des éléments.

Pour utiliser les `ArrayList`s, il faut tenir compte du fait qu'ils sont définis dans le *package* `java.util`. Il faut donc commencer un programme qui les utilise par la ligne :

```
import java.util.ArrayList;
```

ou encore :

```
import java.util.*;
```

Voici une description des principales méthodes de cette classe (comme pour les tableaux classiques de `Java`, les positions des éléments commencent à `0`) :

`ArrayList()`
: Construit un tableau vide.

`ArrayList(int initialCapacity)`
: Construit un tableau **vide**, mais prévu pour ne pas avoir à grossir pour stocker au moins `initialCapacity` éléments. Il est important de noter que ce n'est pas du tout le même comportement que celui des `TableauDynamique`s.

[1] Bien entendu, un objet n'est jamais contenu dans un autre, mais pour simplifier cette section, nous écrirons "contenu dans" pour signifier que le contenant contient une référence vers le contenu.

[2] voir `http ://java.sun.com`

`void add(int index, Object element)`

Insère la référence `element` à la position `index`. Décale toutes les références suivantes d'une case. Ajoute un à la taille du tableau. On ne peut pas ajouter un élément en position `p>n` si le tableau contient `n` cases.

`boolean add(Object o)`

Ajoute la référence `o` à la fin du tableau. Cette méthode renvoie toujours `true` (nous verrons plus loin le sens de la valeur de vérité renvoyée). Ajoute un à la taille du tableau.

`void clear()`

Supprime tous les éléments du tableau.

`boolean contains(Object o)`

Renvoie `true` si et seulement si l'objet désigné par `o` est référencé dans le tableau. Pour comparer `o` au contenu du tableau, cette méthode utilise la méthode `equals` de l'objet paramètre.

`void ensureCapacity(int minCapacity)`

Fait en sorte que le tableau appelant n'ait plus à grossir (après l'appel) pour contenir au moins `minCapacity` éléments.

`boolean equals(Object o)`

Renvoie `true` si et seulement si le paramètre `o` fait référence à une liste (un tableau dynamique) dont le contenu est strictement le même que le tableau appelant (éléments identiques au sens de `equals` et dans le même ordre dans le tableau).

`Object get(int index)`

Renvoie l'objet de position `index` contenu dans le tableau.

`int indexOf(Object o)`

Renvoie la position de la première occurrence de l'objet `o` dans le tableau appelant (et `-1` si l'objet n'est pas trouvé). Se base sur la méthode `equals` (comme `contains`).

`boolean isEmpty()`

Renvoie `true` si et seulement si le tableau appelant est vide.

`int lastIndexOf(Object o)`

Renvoie la position de la dernière occurrence de l'objet `o` dans le tableau appelant (et `-1` si l'objet n'est pas trouvé). Se base sur la méthode `equals`.

`boolean remove(Object o)`

Supprime, si elle existe, la première occurrence dans le tableau appelant de l'objet `o` (égalité au sens de la méthode `equals`). Renvoie `true` si et seulement si une suppression a bien eu lieu. Diminue de un la taille du tableau.

`Object remove(int index)`

Renvoie l'objet de position `index` du tableau, après l'avoir supprimé du tableau. Décale les éléments suivants la position `index` si nécessaire. Diminue de un la taille du tableau.

`Object set(int index, Object element)`

Remplace l'élément de position `index` du tableau appelant par `element`. Renvoie l'ancienne valeur de la case en question. Cette méthode ne peut pas être utilisée pour ajouter des cases au tableau.

`int size()`

Renvoie le nombre de cases du tableau appelant.

`Object[] toArray()`
: Renvoie un tableau de type `Object[]` contenant dans l'ordre tous les éléments du tableau appelant. Le résultat est totalement indépendant du tableau appelant : on peut modifier l'un sans conséquence pour l'autre. Cependant les références contenues dans les deux tableaux sont les mêmes, ce qui signifie qu'une modification d'un objet référencé par l'un des deux tableaux sera visible dans l'autre. En fait, tout se passe comme quand on utilise la méthode `clone` d'un tableau.

`String toString()`
: Renvoie la représentation du tableau appelant sous forme d'une chaîne de caractères. Cette représentation est obtenue en donnant la liste des conversions des éléments sous forme de chaînes de caractères (obtenues par `toString`), séparées par des virgules et entourées d'une paire de crochets.

`void trimToSize()`
: Fait en sorte que le tableau appelant soit exactement de la taille requise pour stocker son contenu actuel.

Comme nous allons le voir dans la suite la plupart des méthodes de la classe `ArrayList` se retrouvent dans la classe `LinkedList` représentant les listes chainées.

2 La classe `LinkedList`

La classe `LinkedList` est la version inclue dans `Java` des listes chainées vues précédemment. Comme vous pouvez le voir dans la table 2, la plupart des méthodes de `LinkedList` sont déja présentes dans `ArrayList`. Nous allons donc expliquer uniquement les méthodes propres à `LinkedList`.

Pour utiliser les `LinkedList`s, il faut tenir compte du fait qu'ils sont définis dans le *package* `java.util`. Il faut donc commencer un programme qui les utilise par la ligne :

```
import java.util.LinkedList;
```

ou encore :

```
import java.util.*;
```

Voici la liste des méthodes propres à `LinkedList`

`LinkedList()`
: construit une liste chainée vide

`void addFirst(Object obj)`
: ajoute `obj` au début de la liste (`obj` devient le premier élément)

`void addLast(Object obj)`
: ajoute `obj` à la fin de la liste (`obj` devient le dernier élément)

`Object getFirst()`
: renvoie le premier élément de la liste

`Object getLast()`
: renvoie le dernier élément de la liste

`Object removeFirst()`
: supprime le premier élément de la liste et le renvoie

`Object removeLast()`
: supprime le dernier élément de la liste et le renvoie

ArrayList

```
1  public class ArrayList {
2          /*
3           *  constructeurs
4           */
5          public ArrayList() ;
6          public ArrayList(int initialCapacity) ;
7          /*
8           *  methodes communes a ArrayList et LinkedList
9           */
10         public void add(int index,Object obj) ;
11         public boolean add(Object obj) ;
12         public void clear() ;
13         public boolean contains(Object obj) ;
14         public boolean equals(Object obj) ;
15         public Object get(int index) ;
16         public int indexOf(Object obj) ;
17         public boolean isEmpty() ;
18         public int lastIndexOf(Object obj) ;
19         public boolean remove(Object obj) ;
20         public Object remove(int index) ;
21         public Object set(int index,Object element) ;
22         public int size() ;
23         public Object[] toArray() ;
24         public String toString() ;
25         /*
26          *  methodes propres a ArrrayList
27          */
28         public void trimToSize() ;
29         public void ensureCapacity(int minCapacity) ;
30 }
31
32
33
```

TAB. 1 – Squelette de la classe ArrayList

LinkedList

```
1  public class LinkedList {
2          /*
3           *  constructeur
4           */
5           public LinkedList() ;
6           /*
7            *  methodes communes a ArrayList et LinkedList
8            */
9          public void add(int index,Object obj) ;
10         public boolean add(Object obj) ;
11         public void clear() ;
12         public boolean contains(Object obj) ;
13         public boolean equals(Object obj) ;
14         public Object get(int index) ;
15         public int indexOf(Object obj) ;
16         public boolean isEmpty() ;
17         public int lastIndexOf(Object obj) ;
18         public boolean remove(Object obj) ;
19         public Object remove(int index) ;
20         public Object set(int index,Object element) ;
21         public int size() ;
22         public Object[] toArray() ;
23         public String toString() ;
24         /*
25          *  methodes propres a LinkedList
26          */
27         public void addFirst(Object obj) ;
28         public void addLast(Object obj) ;
29         public Object getFirst() ;
30         public Object getLast() ;
31         public Object removeFirst() ;
32         public Object removeLast() ;
33 }
```

TAB. 2 – Squelette de la classe LinkedList

PROJET V

LSystem[3]

Une fractale est une figure géométrique telle que chacune de ses parties ressemble à la figure toute entière. De nombreuses méthodes permettent de générer des fractales en utilisant par exemple des suites de nombres complexes (comme les fractales de Mandelbrot et Julia que nous avons vu plus haut). En 1968 Aristid Lindenmayer, un biologiste, imagina un formalisme permettant de représenter la croissance des plantes mathématiquement. Ce formalisme connu sous le nom de *LSystem* permet non seulement de génerer des plantes de type fractal mais aussi de nombreux autres objets fractals comme le flocon de neige de Von Koch (figure 1). Le but de ce problème est de vous permettre de programmer ce formalisme en Java.

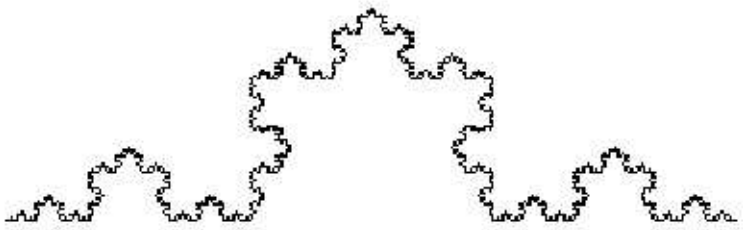


Fig. 1 – Flocon de neige de Von Koch

1 Principe

Un LSystem se compose de deux parties : la première partie est un système de génération formel de chaîne de caractères fractales, la seconde partie est un interpréteur permettant de transformer une chaine de caractère fractale en dessin. Pour générer une chaine de caractère fractale, il suffit de partir d'une chaine initiale appelée *axiome* puis de la transformer en utilisant des règles de remplacement. Par exemple on peut génerer une chaine fractale en utilisant les éléments suivants :

$$\begin{array}{ll} \text{axiome :} & G \\ \text{règles :} & G \mapsto F[-G]+G \\ & F \mapsto FF \end{array}$$

Dans ce cas les états successifs du LSystem seront :

itération	état
0	G
1	$F[-G]+G$
2	$FF[-F[-G]+G]+F[-G]+G$
3	$FFFF[-FF[-F[-G]+G]+F[-G]+G]+FF[-F[-G]+G]+F[-G]+G$
...	...

[3]d'après G.W. Flake, *The Computational Beauty of Nature*, MIT press1998

En laissant ce processus de transformations se poursuivre infiniment nous obtiendrons une chaine de caractères fractale c'est à dire se répétant à toutes les échelles quasi identiquement. La première partie du programme que vous allez écrire va permettre de générer de telles chaines.

Une fois obtenue une chaine de caractère fractale, cette chaîne sera transformée en dessin. Pour cela on suppose qu'une tortue se déplace sur l'écran et exécute tour à tour les instructions contenues dans la chaine de caractère (c'est le principe du langage `Logo`). Chaque caractère est associé à une instruction grâce au tableau suivant :

caractère	action
`F`	la tortue avance en ligne droite d'une longueur fixée en tracant une ligne.
`G`	la tortue avance en ligne droite d'une longueur fixée sans tracer de ligne.
`+`	la tortue tourne à droite d'un angle fixé.
`-`	la tortue tourne à gauche d'un angle fixé.
`[`	la tortue enregistre sa position et sa direction actuelle.
`]`	la tortue revient à la dernière position et direction enregistrées.

2 Programmation de la classe LSystem

La classe `LSystem`, dont le squelette est donné plus bas, permet de représenter informatiquement un LSystème.

```
public class LSystem {
        /* variable permettant de generer une chaine de caractères fractale */
        private String axiome ;
        private String regleF,regleG ;
        private StringBuffer etat ;
        /* variable utile pour le dessin */
        private double dangle ;

        public LSystem(String axiome,String regleF,String regleG,double dangle) ;

        public void transforme() ;
        public String etat() ;
        public double angle() ;
        public String toString() ;
}
```

La méthode `void transforme()` permet de remplacer l'état actuel du `LSystem` par l'état suivant. Pour effectuer cette transformation, il suffit de remplacer chaque caractère `F` (respectivement `G`) de la chaine `etat` par la chaine `regleF` (respectivement `regleG`). Les méthodes `String etat()` et `double angle()` permettent respectivement d'obtenir l'état actuel du LSystem sous forme de `String` et l'angle élémentaire de rotation du LSystem (contenu dans la variable `dangle`). Programmez la classe `LSystem`.

3 Programmation d'une pile de donnée

Afin de pouvoir enregistrer les positions de la tortue lors du dessin d'un LSystem, on veut disposer d'une classe représentant une pile de donnée. Une pile de donnée est une structure dynamique, comme les listes chainées et les tableaux dynamiques, dont le fonctionnement est inspiré de la notion usuelle de pile (par exemple une pile d'assiette) : ainsi vous pouvez ajouter un élément au sommet de la pile ou retirer l'élément qui se trouve au sommet.

signature de la méthode	action associée
`Pile()`	construit une nouvelle pile vide
`Object push(Object obj)`	empile l'objet `obj` au sommet de la pile
`Object pop()`	renvoie l'objet au sommet de la pile et le supprime

1. En utilisant au choix les tableaux dynamiques ou les listes chainées, programmer une classe `Pile` implantant ce comportement.
2. Est il plus judicieux d'utiliser les tableaux dynamiques ou les listes chainées pour programmer la classe pile ? Expliquez pourquoi.

4 La classe Tortue

Pour représenter la position de la tortue sur l'écran vous utiliserez la classe `Position` dont le squelette est le suivant :

```
public class Position {
        /* coordonnees */
        public double x,y ;
        /* direction de déplacement */
        public double direction ;
}
```

Par ailleurs afin de représenter le LSystem dans une fenêtre graphique vous programmerez une classe tortue dont le squelette est le suivant :

```
public class Tortue {
        private Pile memoire ;
        private Position actuelle ;

        public Tortue(Position initiale) ;
        public void dessine(Graphics g, String aDessiner, double dl,double dangle) ;
}
```

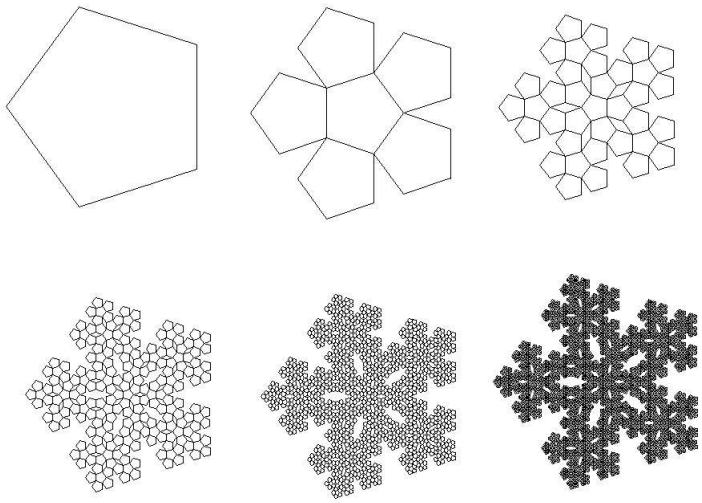
La méthode `dessine` permet de dessiner sur l'objet graphique `g`, la suite des instructions donnée par la chaine de caractère `aDessiner`. Le paramètre `dl` indique la longueur des déplacements de la tortue entre deux positions (et donc également la longueur des segments composant le dessin final). Enfin le paramètre `dangle` indique l'angle de rotation élémentaire à utiliser lorsque la tortue tourne à droite ou à gauche. L'ensemble des instructions nécessaires pour le dessin est récapitulé dans le tableau suivant :

caractère	instructions correspondantes
`F`	- stocker `position.x` et `position.y` dans deux variables temporaires `x` et `y` - ajouter $dl \times sin(position.direction)$ à `position.x` et $-dl \times cos(position.direction)$ à `position.y`. - tracer une ligne entre `(x,y)` et `(position.x,position.y)`.
`G`	- ajouter $dl \times sin(position.direction)$ à `position.x` et $-dl \times cos(position.direction)$ à `position.y`.
`+`	- ajouter *dangle* à `position.direction`.
`-`	- retrancher *dangle* à `position.direction`.
`[`	- créer une nouvelle variable de type `Position` contenant une copie de la position actuelle. - empiler la copie au sommet de la pile `memoire`.
`]`	- remplacer `position` par le sommet de la pile `memoire` et supprimer ce sommet de la pile.

5 Programmation de l'interface graphique

Pour programmer l'interface graphique vous créerez une classe `PanelLSystem` héritant de `JPanel`. La méthode `paint` de la classe `PanelLSystem` appelera la méthode `dessine` de la classe `Tortue`. Pour cela vous devrez donc choisir non seulement la position initiale de la tortue mais aussi le paramètre `dl`. Le paramètre `dangle` est lié au LSystem que vous voulez dessiner (dans les exemples qui suivent il est affiché à coté du dessin). Par contre `dl` ainsi que la position initiale doivent être modifiables par l'utilisateur afin par exemple de pouvoir zoomer sur le dessin. Pour cela vous programmerez un écouteur de clavier. Par ailleurs, les LSystem étant des systèmes itératifs (obtenus par transformation successives) il est judicieux de prévoir une touche permettant à l'utilisateur de passer à l'état suivant du LSystem (en général les dessins intéressants sont obtenus à partir de 3 ou 4 transformations comme vous pouvez le voir sur la figure 2).

FIG. 2 – Six états du Flocon de neige de Penrose

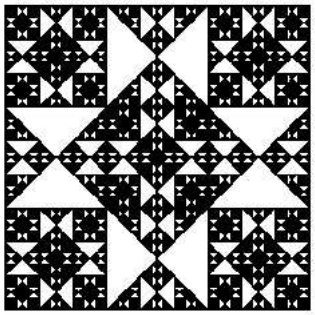


Dans les pages suivantes vous trouverez quelques exemples de LSystems. A coté de chaque dessin est indiqué le nom du fractal correspondant, l'axiome, les règles de générations, et l'angle de rotation élémentaire. Bien sur vous pouvez créer vos propres courbes fractales en vous inspirant de celles ci.

Bush
axiome F
F=FF+[+F-F-F]-[-F+F+F]
dangle 25

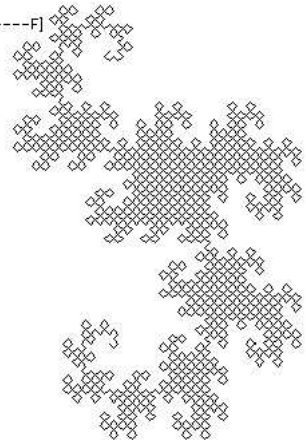
Buisson

Carpet
axiome F-F-F-F
F=F[F]-F+F[--F]+F-F
dangle 90



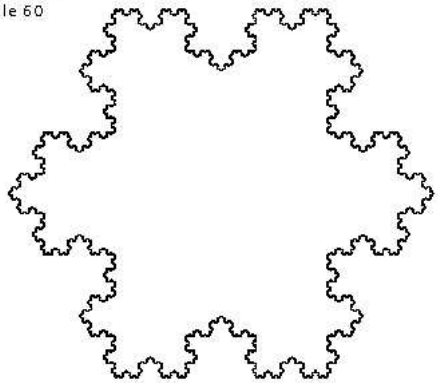
Tapis

Dragon-Curve
axiome F
F=[+F][+G--G----F]
G=-G++G-
dangle 45

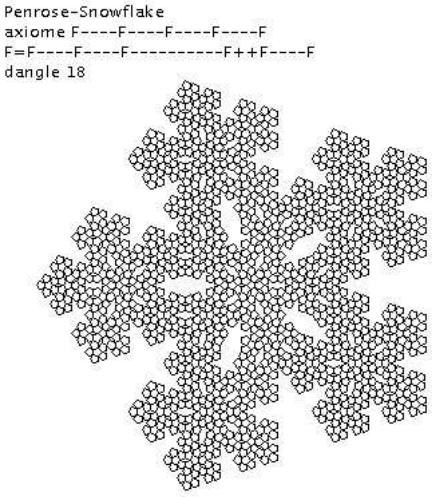


Courbe du dragon

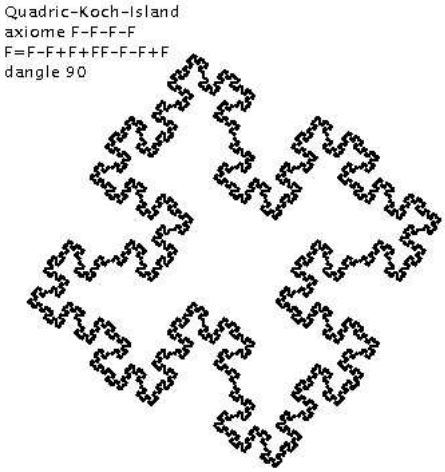
Koch-Island
axiome F++F++F
F=F-F++F-F
dangle 60



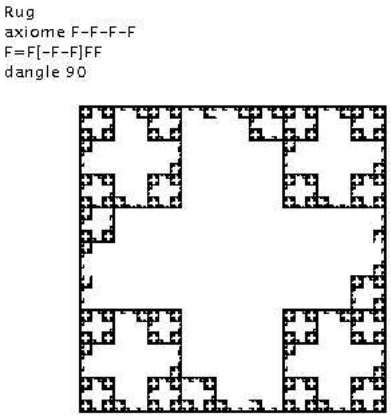
Ile de Koch



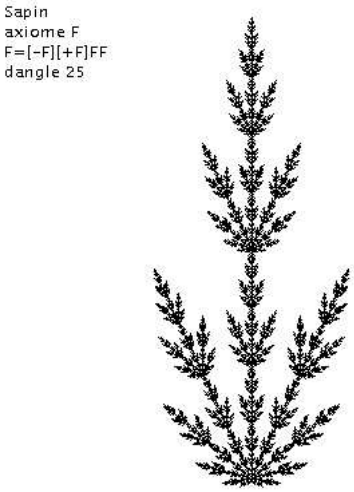
Flocon de neige de Penrose



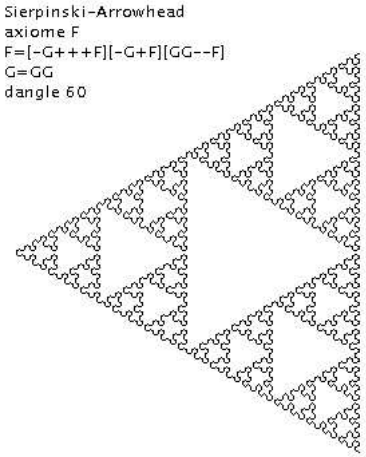
Ile de Koch quadratique



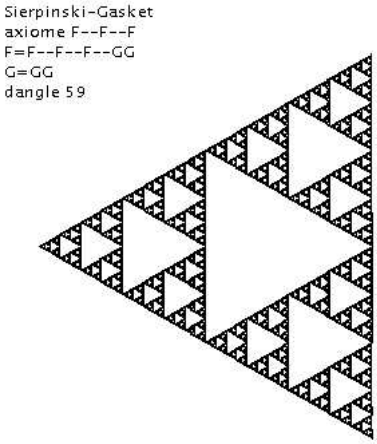
Carpette



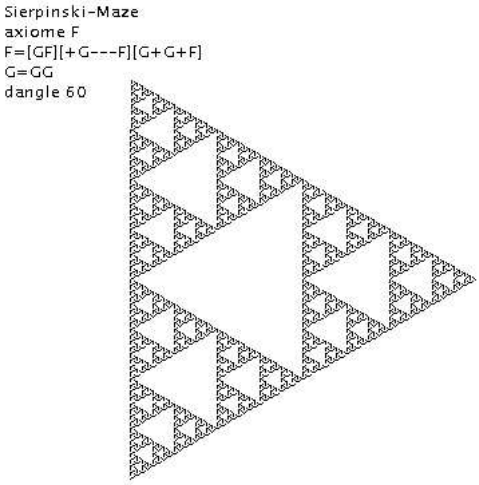
Sapin



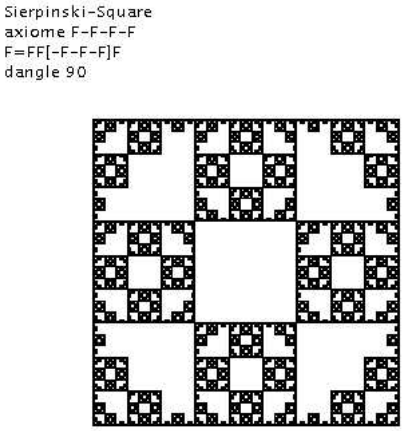
Tête de flèche de Sierpinski



garniture de Sierpinski



Labyrinthe de Sierpinski



Carre de Sierpinski

Chapitre 4

Traitement des types fondamentaux

L'utilisation de `Object` pose deux problèmes. Tout d'abord on ne peut utiliser que les méthodes de `Object`, ce qui est très limité. De plus, comme les types fondamentaux ne sont pas des types objets, il est par exemple **impossible** de placer un `int` dans une variable de type `Object`. Donc, quand on crée par exemple un tableau dynamique d'`Object`s, on ne peut pas l'utiliser pour stocker des valeurs d'un type fondamental. Nous allons voir dans cette section comment résoudre ce problème, la solution retenue nous permettant d'aborder aussi le premier problème.

Le traitement des types fondamentaux est basé sur l'existence pour chaque type fondamental d'une classe *enveloppe*[1] spécialement prévue pour contenir une valeur du type en question.

1 Les classes enveloppes

A chaque type fondamental correspond une classe portant (presque) le même nom et permettant le stockage sous forme objet d'une valeur du type fondamental considéré. Le tableau qui suit donne la correspondance entre type fondamental et classe enveloppe :

Type	classe
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

Les enveloppes correspondant à des types fondamentaux numériques (entiers et réels) sont appelées des **enveloppes numériques**.

Chaque classe possède un constructeur prenant comme paramètre une valeur du type fondamental qu'elle représente. Elle possède aussi une méthode *`type`*`Value` renvoyant la valeur représentée (*`type`* est remplacé par le nom du type fondamental représenté, ce qui donne par exemple `floatValue` pour la classe `Float`). Les classes enveloppes ne sont pas modifiables. Elles redéfinissent la méthode `toString` afin d'afficher la valeur qu'elles représentent. Voici un exemple d'utilisation des classes enveloppes :

Exemple 1 :

[1] un *wrapper* en anglais, c'est-à-dire un emballage.

WrapperDemo

```
1  public class WrapperDemo {
2    public static void main(String[] arg) {
3      Double d=new Double(2.5);
4      Boolean b=new Boolean(true);
5      System.out.println(d);
6      System.out.println(b);
7      System.out.println(d.doubleValue());
8      double x=d.doubleValue();
9      d=new Double(2*x);
10     System.out.println(d);
11     Object o=new Integer(2);
12     System.out.println(o);
13     Object[] to={new Integer(2),
14                  new Double(2.5),
15                  "ABC"};
16     for(int i=0;i<to.length;i++)
17       System.out.println(to[i].getClass());
18   }
19 }
```

Ce programme affiche les lignes suivantes :

```
2.5
true
2.5
5.0
2
class java.lang.Integer
class java.lang.Double
class java.lang.String
```

La dernière partie est particulièrement intéressante, car elle montre qu'on peut mélanger dans un même tableau des objets de natures très différentes, en utilisant en particulier des types fondamentaux, représentés grâce aux classes enveloppes.

Les classes enveloppes définissent toutes une variable de classe non modifiable, appelée `TYPE`. Cette variable contient une référence vers un objet de type `Class` décrivant le type fondamental associé. Voici un exemple de programme utilisant cette variable :

Exemple 2 :

WrapperClass

```
1  public class WrapperClass {
2    public static void main(String[] arg) {
3      Class c=Integer.TYPE;
4      System.out.println(c);
5      System.out.println(c.isPrimitive());
6      Integer i=new Integer(2);
7      Class d=i.getClass();
8      System.out.println(d);
9      System.out.println(d.isPrimitive());
10   }
```

```
11 }
```

L'affichage produit est le suivant :

```
int
true
class java.lang.Integer
false
```

On constate que `Integer.TYPE` représente bien le type fondamental `int` alors que, si `i` fait référence à un objet de type `Integer`, `i.getClass()` représente le type objet `Integer`.

Terminons cette première approche par un "bêtisier" :

Les types fondamentaux ne sont pas des types objets

Nous avons déjà évoqué ce point. Il faut donc bien garder à l'esprit que des instructions comme `Object o='a';` ou `Object o=2.5;` sont incorrectes et rejetées par le compilateur.

Les classes enveloppes utilisent des constructeurs

On ne peut donc pas écrire des instructions de la forme `Double d=2.5;` ou `Boolean t=true;` qui sont rejetées par le compilateur.

Les classes enveloppes ne sont pas des types fondamentaux

On ne peut donc pas passer directement d'un objet enveloppe à un type fondamental, et le programme suivant est rejeté par le compilateur (à cause de la deuxième ligne) :

```
Double d=new Double(2.5);
double x=d;
```

Les classes enveloppes ne sont pas "ordonnées"

Nous savons que les types fondamentaux sont ordonnés et qu'on peut par exemple placer une valeur de type `int` dans une variable de type `double`, comme par exemple `double x=2;`. Cette propriété n'est pas vérifiée par les classes enveloppes numériques et le programme suivant est donc rejeté à la compilation (à cause de la deuxième ligne) :

```
Integer g=new Integer(2);
Double x=g;
```

Le transtypage numérique ne s'applique pas aux classes enveloppes

Nous savons qu'il est possible de convertir explicitement des valeurs d'un type numérique dans un autre, par exemple `int i=(int)2.3;`. Ceci n'est pas possible avec les classes enveloppes, ce qui rend le programme suivant incorrect (le compilateur n'accepte pas la seconde ligne) :

```
Double d=new Double(1.7);
Integer g=(Integer)d;
```

Les règles qui viennent d'être présentées sont parfaitement logiques, en accord total avec tout ce qui a été dit dans ce chapitre. Il m'a semblé important de les énoncer pour bien illustrer les différences entre les types fondamentaux et les enveloppes associées.

2 Les enveloppes numériques

2.1 Le type `Number`

Comme nous venons de le voir, les enveloppes numériques sont très différentes des types fondamentaux qu'elles représentent, en particulier au niveau des conversions. Comparons en effet des transtypages faisant intervenir `int` et `double`, puis `Integer` et `Double` :

WrapperCast

```
1  public class WrapperCast {
2    public static void main(String[] arg) {
3      double x=1.8;
4      int y=(int)x;
5      x=y; // transtypage implicite ici
6      Double d=new Double(1.8);
7      Integer i=new Integer((int)d.doubleValue());
8      d=new Double(i.intValue());
9    }
10 }
```

La manipulation des classes enveloppes est donc assez lourde, et implique surtout la création régulière de nouveaux objets (à cause de l'immuabilité des objets considérés), ce qui réduit l'efficacité des programmes les utilisant.

Pour remédier à cet inconvénient, `Java` propose un type objet spécial[2], le type `Number`. Ce type va jouer pour les enveloppes numériques le rôle que joue `Object` pour tous les types objets. Tout d'abord, une variable de type `Number` peut recevoir une référence vers un objet de type enveloppe numérique quelconque, comme l'illustre le programme suivant :

WrapperNumber

```
1  public class WrapperNumber {
2    public static void main(String[] arg) {
3      Double d=new Double(-1.5);
4      Integer i=new Integer(2);
5      Number n=d;
6      Number p=i;
7    }
8  }
```

Une référence de type `Number`, comme pour n'importe quel type objet, peut être placée dans une variable de type `Object`. De plus, comme la classe `Object` pour les types objets, la classe `Number` propose des méthodes communes à toutes les enveloppes numériques :

`int intValue()`
: Renvoie la valeur numérique représentée par l'objet appelant convertie en `int`.

`long longValue()`
: Renvoie la valeur numérique représentée par l'objet appelant convertie en `long`.

`float floatValue()`
: Renvoie la valeur numérique représentée par l'objet appelant convertie en `float`.

`double doubleValue()`
: Renvoie la valeur numérique représentée par l'objet appelant convertie en `double`.

`byte byteValue()`
: Renvoie la valeur numérique représentée par l'objet appelant convertie en `byte`.

`short shortValue()`
: Renvoie la valeur numérique représentée par l'objet appelant convertie en `short`.

[2] il s'agit d'une classe abstraite, concept que nous étudierons dans le chapitre consacré à l'héritage.

Remarque 1. Contrairement à `Object`, la classe `Number` ne possède pas de constructeur et il est donc impossible de créer un objet de type `Number`. On peut seulement utiliser des variables de type `Number`.

Voici maintenant un exemple d'utilisation des méthodes de `Number` :

WrapperNumberUse

```
1  public class WrapperNumberUse {
2    public static void main(String[] arg) {
3      Number[] nbs={new Double(-1.6),
4                    new Integer(3),
5                    new Float(1.7f),
6                    new Long(2544)};
7      for(int i=0;i<nbs.length;i++)
8        System.out.println(i+"->"+nbs[i].intValue());
9      System.out.println("--------");
10     for(int i=0;i<nbs.length;i++)
11       System.out.println(i+"->"+nbs[i].floatValue());
12     System.out.println("--------");
13     for(int i=0;i<nbs.length;i++)
14       System.out.println(i+"->"+nbs[i].byteValue());
15   }
16 }
```

Ce programme affiche les lignes suivantes :

```
0->-1
1->3
2->1
3->2544
-----
0->-1.6
1->3.0
2->1.7
3->2544.0
-----
0->-1
1->3
2->1
3->-16
```

La figure 1 donne l'état de la mémoire avant l'exécution des boucles dans le programme étudié.

2.2 Les méthodes

Toutes les méthodes de la classe `Number` sont utilisables avec un objet d'une classe enveloppe numérique quelconque (ce que nous venons d'utiliser dans l'exemple précédent). Le programme suivant est donc correct :

WrapperNumberTest

```
1  public class WrapperNumberTest {
2    public static void main(String[] arg) {
3      Double d=new Double(-1.6);
4      System.out.println(d.byteValue());
```

```
5       System.out.println(d.shortValue());
6       System.out.println(d.intValue());
7       System.out.println(d.longValue());
8       System.out.println(d.floatValue());
9       System.out.println(d.doubleValue());
10    }
11  }
```

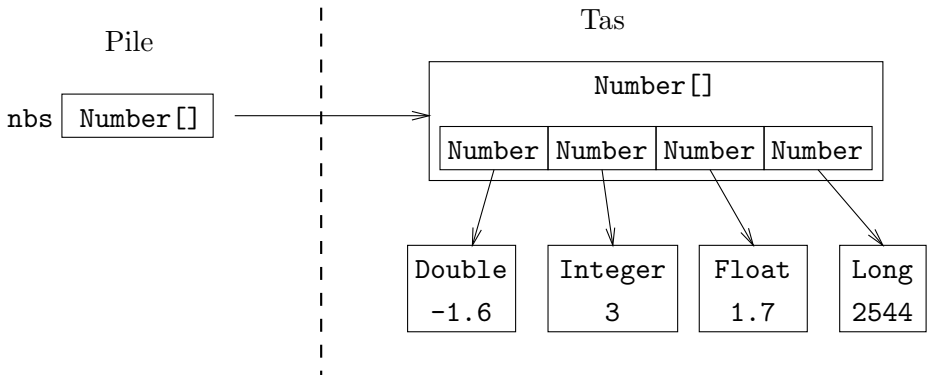


FIG. 1 – Un tableau de `Number`

Il affiche :

```
-1
-1
-1
-1
-1.6
-1.6
```

La situation est cependant un peu différente de celle de `Object`. En effet, `Object` définit des méthodes, qui peuvent éventuellement être reprogrammées dans les classes. Au contraire, `Number` se contente d'établir un *contrat*, c'est-à-dire une liste de méthodes qui doivent être programmée. Ce sont donc les enveloppes numériques elles-mêmes qui implantent (programment) toutes les méthodes d'accès de la forme `typeValue`.

Remarque 2. On obtient ici un élément de réponse au problème posé par l'utilisation de `Object` : une variable de ce type peut faire référence à un objet de n'importe quel type, mais on ne peut alors utiliser que les méthodes de la classe `Object`. Grâce à `Number`, nous avons aussi une technique permettant de traiter uniformément plusieurs classes, sans pour autant être limité aux méthodes de `Object`.

2.3 Représentation graphique

On peut représenter les classes enveloppes sous forme d'un arbre, en utilisant le fait que les enveloppes numériques peuvent être "transformées" en `Number`. On obtient ainsi la représentation de la figure 2.

Remarque 3. Il faut bien retenir que le transtypage automatique fonctionne du bas vers le haut de l'arbre (en suivant les flèches) et par étape. On peut donc placer une référence vers un `Integer`

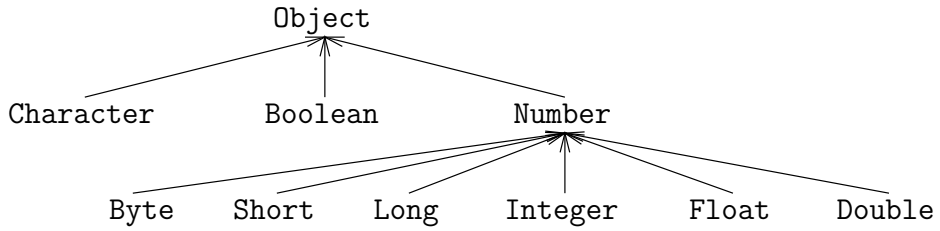


FIG. 2 – Les classes enveloppes

dans une variable de type `Number`. On peut aussi placer une référence de type `Number` dans une variable de type `Object`.

De la même façon, le cheminement inverse peut se faire par étape. Si une référence désigne un `Double` et est placée dans une variable de type `Object`, on peut transtyper la référence en un `Number` ou directement en un `Double`.

Pour finir, le transtypage direct entre branche n'est pas possible. Si on dispose d'une référence de type `Number`, on ne peut pas la convertir en une référence de type `Boolean` (tout simplement car une référence de type `Number` ne peut en aucun cas désigner un objet de type `Boolean`).

2.4 Autres fonctionnalités des enveloppes numériques

Pour clore cette section consacrée aux enveloppes numériques, on peut évoquer quelques possibilités pratiques intéressantes, mais relativement éloignées du propos de ce chapitre :

`MIN_VALUE` et `MAX_VALUE`

Chaque enveloppe numérique définit deux constantes de classe `MIN_VALUE` et `MAX_VALUE`. Elles correspondent respectivement aux valeurs minimale et maximale utilisable pour le type correspondant. Par exemple `Integer.MIN_VALUE` contient donc la valeur -2^{31}, alors que `Byte.MAX_VALUE` correspond à $2^7 - 1$.

Attention, la classe `Number` ne possède pas de telles constantes (ce qui est logique car le type d'une constante dépend de la classe de l'objet).

`NEGATIVE_INFINITY`, `POSITIVE_INFINITY` et `NaN`

Les classes `Float` et `Double` définissent trois constantes de classes (de type `float` pour la première et `double` pour la seconde). Ces trois constantes sont différentes de toutes les valeurs classiques prises par un `float` ou un `double`. Par exemple, `Double.POSITIVE_INFINITY` est différent de `Double.MAX_VALUE`. La constante `NEGATIVE_INFINITY` est une représentation informatique de $-\infty$ et en possède les propriétés : elle est plus petite que tout autre nombre ($\forall x \in \mathbb{R},\ -\infty < x$), si on lui ajoute un réel classique, on obtient elle-même ($\forall x \in \mathbb{R},\ -\infty + x = -\infty$), etc. De la même façon, `POSITIVE_INFINITY` représente $+\infty$ et vérifie ses propriétés. Un problème se pose quand on cherche à faire des opérations au résultat mathématiquement indéfini, comme par exemple $-\infty + \infty$. Dans ce cas, on obtient `NaN` (abréviation pour *Not a Number*).

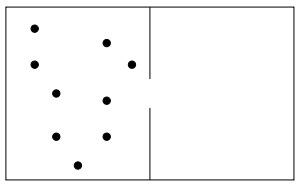
PROJET VI

Calculs d'entropie

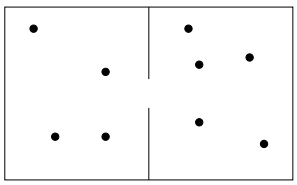
L'entropie est une notion issue de la thermodynamique permettant de quantifier la quantité de désordre (où de façon équivalente d'information) dans un système physique. Dans ce projet nous allons utiliser une variation de cette notion afin de quantifier la quantité d'information dans un texte (et même plus généralement dans une liste d'objets quelconques).

1 Qu'est ce que l'entropie ?

La notion d'entropie a été inventée au XIX ème siècle pour décrire qualitativement un système physique comportant un très grand nombre de particules. En effet pour un tel système, comme par exemple un gaz, il est impossible de décrire explicitement les positions et vitesses de chaque particules en utilisant les équations classiques de la mécaniques (l'ensemble d'équations obtenus seraient impossible à résoudre). On doit donc se contenter de variables globales décrivant qualitativement le système (par exemple pour un gaz la pression, le volume et la température). En général on peut toujours définir les deux variables énergie et entropie. L'énergie du système est approximativement égale à la somme des énergies des particules le composant. L'entropie du système quantifie le désordre dans le système. D'après le premier principe de la thermodynamique, l'énergie se conserve au cours du temps. D'après le deuxième principe de la thermodynamique l'entropie augmente toujours au cours du temps. Considérons par exemple une boite séparée en deux compartiments, contenant des particules de gaz pouvant passer d'un compartiment à l'autre grace à un trou dans la paroi centrale (figure 3). Il est évident que le désordre du système est plus grand quand les particules ne sont pas toutes "rangées" dans le même compartiment. En particulier l'entropie est plus grande quand les particules se répartissent dans les deux compartiments. D'après le deuxième principe si l'on part de la configuration dans laquelle les particules sont rangées dans le premier compartiment, au bout d'un temps assez long les particules vont se répartir dans les deux compartiments : le gaz tend à occuper tout l'espace disponible.



Toutes les particules sont dans le premier compartiment : l'entropie est faible.

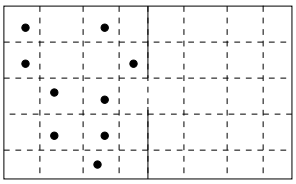


Les particules se répartissent dans les deux compartiments : l'entropie est grande.

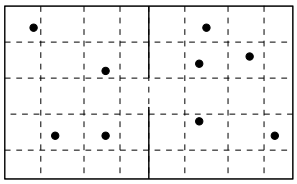
Fig. 3 – Particules dans une boite comportant deux compartiments

Supposons maintenant que chacun des compartiment comporte un quadrillage (figure 4). Dans ce cas pour coder la position des particules dans le premier cas (toutes les particules sont dans le même compartiment), il nous faut une matrice de taille 4×5 (chaque case de la matrice contient 1 si la case correspondante de la boite contient une particule, et 0 sinon). Dans le deuxième cas une matrice 4×5 ne suffit plus car certaines particules sont dans l'autre compartiment : il faut une matrice 8×5. En particulier la quantité d'information nécessaire pour décrire le deuxième état

est deux fois plus grande. Ainsi, comme on le voit avec cette démonstration très peu rigoureuse, l'entropie correspond à la quantité d'information permettant de décrire le système. Ce type d'idée a permis d'introduire la notion d'entropie dans des domaines très éloigné du monde physique pour laquelle elle avait été introduite au départ. En particulier l'entropie est une notion clef de la théorie de l'information.



Le système se décrit par une matrice 4×5
La quantité d'information nécessaire est 20.



Le système se décrit par une matrice 8×5
La quantité d'information nécessaire est 40.

FIG. 4 – Codage de la position des particules

2 Entropie d'une liste d'éléments

Dans le cas d'une liste d'éléments on peut introduire une notion d'entropie correspondant à la quantité d'information dans la liste :

Definition 1. Etant donnée une liste `a` d'éléments dont les fréquences sont données par la liste `f`, l'entropie de `a` est la somme : $\sum -f_i \log f_i$, où f_i désigne le i-ème élément de la liste `f`.

Remarque 1. La fréquence d'un élément est le nombre de fois ou cet élément apparait divisé par le nombre total d'éléments.

Plus l'entropie est grande plus la liste contient d'information c'est à dire plus il est difficile de la compresser (comme nous le verrons dans un prochain projet consacré au codage de Huffman).

Exemple 1 :

Soit la liste d'éléments `a={x,y,y,z,x,x}` alors la liste des fréquences des éléments de `a` est `f={3,2,1}` (`x` apparait trois fois, `y` deux fois et `z` une seule fois). L'entropie de la liste est donc : $-\frac{3}{6}\log\frac{3}{6} - \frac{2}{6}\log\frac{2}{6} - \frac{1}{6}\log\frac{1}{6}$.

Exemple 2 :

Si tous les éléments de la liste `a` sont identiques alors l'entropie vaut zero car $\log 1 = 0$. Dans ce cas la liste est maximalement compressible : il suffit pour la décrire d'indiquer son unique élément et le nombre de fois où il apparait.

1. Ecrire une méthode de classe permettant de trier les éléments d'une `ArrayList a`. On supposera que tous les éléments de la liste `a` sont des `String`s. En particulier, après transtypages en `String` deux éléments de `a` pourront être comparés en utilisant la méthode `int compareTo(Object obj)`.
2. Ecrire une méthode de classe prenant en paramètre une `ArrayList a` et renvoyant une nouvelle `ArrayList` correspondant à la liste des fréquences des éléments de `a`. Pour cela vous commencerez par trier les éléments de `a`. Par ailleurs vous représenterez les fréquences des éléments de `a` comme des `Double`s.

3. Ecrire une méthode de classe prenant en paramètre une `ArrayList f` dont les éléments sont des `Double` et renvoyant l'entropie correspondante (on suppose que tous les éléments de la liste `f` représentent des réels strictement positifs).
4. Ecrire une méthode de classe prenant en paramètre une chaine de caractère et renvoyant la liste des mots contenus dans cette chaine.
5. Utilisez les questions précédentes pour écrire une méthode calculant l'entropie d'un texte quelconque. Vous pouvez utiliser la classe `File` donnée en appendice afin de lire un texte depuis un fichier.

File

```
import java.io.*;
public class File {
// Read file  (lit le fichier fl et renvoie le contenu)
public static String readFile(String fl) {
        try {
                FileInputStream fs = new FileInputStream(fl);
                BufferedReader ds= new BufferedReader(new InputStreamReader(fs));
                StringBuffer text = new StringBuffer(fs.available());
                String str = new String();
                while (str != null) {
                        str = ds.readLine();
                        if (str != null) {
                                text.append(str) ;
                                text.append('\n') ;
                        }
                }
                return text.toString() ;
        } catch (Exception err) {
                System.out.println("Cannot open file.");
                return new String() ;
        }
}
// Write file (écrit le texte txt dans le fichier fl)
public static void writeFile(String fl, String txt) {
        try {
                FileOutputStream fs = new FileOutputStream(fl);
                DataOutputStream ds = new DataOutputStream(fs);
                String ls = System.getProperty("line.separator");
                for (int i = 0; i < txt.length(); i++) {
                  char ch = txt.charAt(i);
                switch (ch) {
                        case '\n':
                                        ds.writeBytes(ls);
                                        break;
                        default: ds.write(ch);
                 }
                }
        } catch (Exception err) {
                System.out.println("Cannot save file.");
        }
```

```
41 }
42 }
```