Travaux dirigés et pratiques
Java

Licence 2 Mathématiques et Informatique
UFR Sciences et Techniques
Université du Havre

Damien Olivier

8 février 2011

Avertissement

Certains exercices ont été empruntés à d'autres enseignants, je les remercie ici et je m'excuse de ne pas les avoir cités dans ce document. Les étudiants sont ce qu'ils sont et choisissent parfois la facilité en allant directement chercher les solutions j'ai donc choisi de ne pas leur faciliter la tache.

Chapitre 1

Outils

1.1 Eclipse

Eclipse est un environnement de développement libre permettant de mettre en œuvre des projets de développements dans de nombreux langages comme C++, Python, Ruby et Java ... Nous nous contenterons ici de Java.

1.1.1 Introduction

La première chose consiste à intaller Eclipse que vous pouver télécharcher à l'url suivante : http ://www.eclipse.org/downloads/.

1.1.2 Utilisation d'Eclipse

Chapitre 2

Tour de chauffe

Cette partie à pour objectif de revoir les bases qui vous ont déjà été enseignées au premier semestre.

2.1 Tableaux

Exercice 1

I L'objectif de cet exercice est de revoir les structures du langage et les tableaux. Ainsi dans cet exercice on représentera les polynômes à coefficients réels à l'aide de tableaux de réels `double`. Par exemple $P(x) = \sum_{i=0}^{n} a_i * x^i$ sera représenté par un tableau `t` tel que `t[i]` = a_i pour tout i entre 0 et n. Il y a plusieurs représentations possibles pour un même polynôme : on peut toujours élargir une représentation sous forme de tableau par un autre tableau plus long et complété par des zéros. On va commencer par écrire des méthodes utilitaires.

- I.1 Pour les nombres "flottants" (`double`, `float`), il n'y a pas de test d'égalité fiable. Écrire une méthode `egalite()` prenant deux arguments de type `double` et qui retourne `true` lorsqu'ils sont voisins à une constante ϵ près (avec par exemple $\epsilon = 0.000001$).
- I.2 Écrire une méthode `litPolynome()` qui demande à l'utilisateur de donner les coefficients et crée le polynome correspondant, c'est à dire le tableau et retourne celui-ci.
- I.3 Écrire une méthode `degrePolynome()` qui retourne le degré d'un polynôme. Par convention, la méthode retournera -1 dans le cas du polynôme nul.
- I.4 Écrire une méthode `creeMonome()` qui fabrique la représentation d'un monôme $a * x_i$.
- I.5 Écrire une méthode `coefficient()` qui prend en paramètres un polynôme et un entier positif (correspondant à un exposant), et qui retourne le coefficient correspondant. Pour un entier excédant la taille du tableau représentant le polynôme, la méthode doit retourner zéro.
- I.6 Écrire une méthode d'affichage `affichePolynome()`.

II Fonctions arithmétiques simples

II.1 Écrire les méthodes de somme et de soustraction de polynômes.

II.2 Écrire la méthode de multiplication de deux polynômes.

III Dérivée d'un polynôme

III.1 Écrire la méthode dérivant un polynôme.

IV Division euclidienne

IV.1 Les polynômes à coefficients réels sont munis d'une opération de division euclidienne analogue à celle des entiers. On a la propriété suivante : si A et B sont deux polynômes (avec $B \neq 0$), il existe un unique couple de polynômes Q (le quotient) et R (le reste) tels que :
– le degré de R est strictement inférieur à celui de B ;
– $A = BQ + R$

V Algorithme d'Euclide appliqué aux polynômes.
On s'intéresse au PGCD (Plus Grand Commun Diviseur) d'un couple de polynômes $(P_1, P_2) = (0, 0)$. Attention, dans le cas des polynômes, il n'y a pas unicité du PGCD. Il y a une infinité de PGCD possibles pour un couple de polynômes, qui diffèrent par un coefficient de proportionnalité réel non nul. On peut donc déduire l'ensemble des PGCD de deux polynômes à partir d'un PGCD quelconque. On va utiliser l'algorithme d'Euclide et l'adapter aux polynômes.
L'algorithme d'Euclide est basé sur la propriété suivante : si a et $b \neq 0$ sont deux entiers positifs, et si $r1$ est le reste de la division euclidienne de a par b, alors $pgcd(a, b) = pgcd(b, r1)$. On recommence ensuite la même opération sur le couple $(b, r1)$, ce qui donne un nouveau reste $r2$, on poursuit avec le couple $(r1, r2)$ et ainsi de suite, jusqu'à tomber sur un reste rn nul. On a alors $pgcd(a, b) = pgcd(b, r1) = pgcd(r1, r2) = ... = pgcd(rn, 0)$; or on voit immédiatement que $pgcd(rn, 0) = rn$.

V.1 Quelle est la condition d'arrêt dans le cas des polynômes ?

V.2 Écrire la méthode `pgcd()` basée sur cet algorithme.

V.3 Écrire un programme complet qui calcule un PGCD de deux polynômes. On choisit d'afficher le PGCD dont le coefficient dominant est égal à 1.

Exercice 2

I Les matrices
Les matrices sont des tableaux à deux dimensions de nombres réels, on va donc les représenter par des variables de type `double[][]`. La syntaxe des manipulations de base sur ces tableaux est la suivante :

```
1 double[][] a;              // déclaration d'une matrice
2 a = new double[m][n];      // création d'une matrice à m lignes
3                            // et n colonnes
4 a[i][j]                    // accès à l'élément j de la ligne i
5 a.length                   // nombre de lignes d'une matrice
6 a[0].length                // nombre de colonnes d'une matrice
```

Soit une matrice de taille $m \times n$, par convention on décide que le premier indice représente le numéro de ligne de 0 à $m - 1$ et le deuxième indice le numéro de colonne de 0 à $n - 1$.

I.1 Écrire une méthode `litMatrice()` qui demande à l'utilisateur les dimensions de sa matrice et ses coefficients et renvoie le tableau correspondant.

I.2 Écrire une fonction `afficheMatrice()` qui affiche une matrice ligne par ligne.

I.3 Soit A et B deux matrices $m \times n$, dont les éléments sont respectivement a_{ij} et b_{ij} , la matrice $A+B$ est la matrice de même taille dont les éléments sont les $a_{ij} + b_{ij}$. Écrire une méthode `addition()` qui prend en arguments deux matrices (qu'on supposera de même taille) et renvoie leur somme sous la forme d'une nouvelle matrice.

I.4 Soit $A = (a_{ij})$ une matrice $m \times n$ et x est un nombre réel, la matrice xA est la matrice $m \times n$ dont les éléments sont les xa_{ij}. Écrire une méthode `prodScal()` qui prend en arguments un réel x et une matrice A et qui renvoie la matrice produit de A par x.

I.5 Soit $A = (a_{ij})$ une matrice $m \times n$, sa transposée ${}^tA$ est la matrice à n lignes et m colonnes dont les éléments sont les a_{ji}. Écrire une méthode qui calcule la transposée d'une matrice.

I.6 Soit $A = (a_{ij})$ une matrice $m \times n$ et $B = (b_{ij})$ une matrice $n \times m$ le produit AB est la matrice à m lignes et p colonnes définie par

$$AB = \begin{pmatrix} c_{0,0} & \cdots & c_{0,p-1} \\ \vdots & \ddots & \vdots \\ c_{m-1,0} & \cdots & c_{m-1,p-1} \end{pmatrix} \quad \text{avec } c_{ik} = \sum_{j=0}^{n-1} a_{ij} b_{jk} \qquad (2.1)$$

Écrire une méthode qui prend en arguments deux matrices A et B de tailles supposées compatibles et renvoie leur produit.

2.2 Listes

Exercice 3

I Reprendre l'exercice sur les polynômes en utilisant des listes linéaires chainées pour représenter les polynomes.

II Quelles sont les parties du code qui diffèrent ?

III Existe t-il une solution pour rendre le code plus indépendant de la représentation ? (On apportera la réponse par la suite).

Chapitre 3

Héritage et Interface

3.1 Héritage

En Java l'héritage est un héritage simple (une classe fille a une unique classe mère). Lorsqu'une classe hérite d'une autre classe, elle l'étend. Une classe fille qui a hérité d'une classe mère possède tous les attributs et toutes les méthodes de la classe mère, elle ne peut cependant pas utiliser les membres de la classe mère qui sont privés.

```
1 class Fille extends Mere
2 {
3   // ...
4 }
```

Exercice 4

I On désire réaliser un programme Java permettant d'écrire facilement des histoires de Western. Dans nos histoires, nous aurons des brigands, des cowboys, des shérifs, des barmen et des dames en détresses.

I.1 Les intervenants de nos histoires sont tous des humains. Un humain est caractérisé par son nom et sa boisson favorite. La boisson favorite d'un humain est, par défaut, de l'eau. Un humain pourra parler. On aura donc une méthode `parle(texte)` qui affiche : *(nom de l'humain) - texte*. Un humain pourra également se présenter *(il dit bonjour, son nom, et indique sa boisson favorite)*, et boire *(il dira "Ah ! un bon verre de (sa boisson favorite) ! GLOUPS !")*. Toutes les variables de la classe `Humain` seront privées. Pour connaître le nom d'un humain, on aura besoin d'une méthode `quelEstTonNom()` qui renvoie la chaîne contenant le nom de cet humain. De même, on aura besoin d'une méthode pour connaître sa boisson favorite.
Réalisez la classe `Humain`, dont le constructeur reçoit le nom de l'humain créé en paramètre. Réalisez une classe `Histoire` contenant votre méthode `main()`. Dans cette histoire, vous créerez un humain qui se présente et qui boit.

I.2 Les dames, les cowboys et les brigands sont tous des humains. Par la même, ils ont tous un nom et peuvent tous se présenter. Par contre, il y a certaines différences entre ces trois classes d'individus. Les brigands peuvent kidnapper les dames, les dames peuvent se faire enlever et se faire libérer et

les cowboys peuvent les libérer.
Une dame est caractérisée par la couleur de sa robe (une chaîne de caractère), et par son état (`libre` ou `captive`). Elle peut se faire kidnapper (auquel cas elle hurle), se faire libérer par un cowboy (elle remercie alors le héros qui l'a libéré). Elle peut également changer de robe (tout en s'écriant "*Regardez ma nouvelle robe (couleur de la robe) !*)". Un brigand est un humain qui est caractérisé par un look ("`méchant`" par défaut), un nombre de dames enlevées, la récompense offerte lorsqu'il est capturé (100 $ par défaut), un `boolean` indiquant s'il est en prison ou pas. Il peut kidnapper une dame (auquel cas, il s'exclame *"Ah ah ! (nom de la dame), tu es mienne désormais !"*. Il peut également se faire emprisonner par un cowboy (il s'écrit alors *"Damned, je suis fait ! (nom du cowboy), tu m'as eu !"*). On dispose également d'une méthode pour connaître la récompense obtenue en cas de capture. Un cowboy est un humain qui est caractérisé par sa popularité (0 pour commencer, augmente de 1 à chaque dame délivrée) et un adjectif le caractérisant ("`vaillant`" par défaut). Un cowboy peut tirer sur un brigand (un commentaire indique alors (*"Le (adjectif) (nom) tire sur (nom du méchant). PAN !"*) et le cowboy s'exclame (*"Prend ça, crapule !"*). Il peut également libérer une dame (en la flattant).
Réalisez les trois classes `Brigand`, `Cowboy` et `Dame`. Modifiez votre histoire pour tester ces classes.

I.3 Au sein d'une classe, il est également possible de changer les attributs et méthodes de la classe mère. Pour cela, il suffit simplement de les redéfinir dans la classe fille. On parle alors de surcharge. Quand on demande son nom à une dame, elle répond *Miss (son nom)* et un brigand dira *(son nom) le (son look)* (par exemple *Bob le méchant*). Un cowboy dira simplement son nom (ce sont des gens simples). Surchargez la méthode `quelEstTonNom()` dans les classes `Brigand` et `Dame`. Testez. Les méthodes que vous avez écrite "remplaceront" (on parle de surcharge) la méthode `quelEstTonNom()` de la classe `Humain`.

I.4 On désire aussi changer le mode de présentation des brigands, dames et cowboys. Un brigand parlera aussi de son look et du nombre de dames qu'il a enlevé et de la récompense offerte pour sa capture. Une dame ne pourra s'empêcher de parler de la couleur de sa robe, alors qu'un cowboy dira ce que les autres disent de lui (son adjectif) et parlera de sa popularité. Si on récrit la méthode qui permet a une personne de se présenter, la méthode de la classe `Humain` sera remplacée. On désire quand même appeler cette méthode. Le brigand Bob se présentera par exemple de la manière suivante :
(Bob) - Bonjour, je suis Bob le méchant et j'aime le Tord-Boyaux.
(la méthode de présentation de la classe Humain)
(Bob) - J'ai l'air méchant et j'ai déjà kidnappé 5 dames !
(Bob) - Ma tête est mise a prix 100 $!
On veut donner une boisson par défaut à chaque sous-classe d'humain (lait pour la dame, tord-boyaux pour le brigand et whisky pour le cowboy). Modifiez vos classes en ce sens. Testez.

I.5 Un barman est un humain dont la boisson favorite est le Vin. Il peut servir n'importe quel humain, en lui donnant un verre de sa boisson favorite. Il est caractérisé par le nom de son bar. Par défaut, il s'agira du bar *Chez (nom du*

barman) . La classe `Barman` aura deux constructeurs. Soit on crée un barman en indiquant uniquement son nom, soit on indique également le nom de son bar. Quand un barman se présente, il n'oublie pas de mentionner le nom de son bar. Quand un barman parle, il termine toutes ses phrases par "*Coco.*". Réalisez la classe `Barman`. Testez.

I.6 On ajoute à notre histoire des shérifs. Un shérif est un cowboy qui peut coffrer des brigands (en criant "*Au nom de la loi, je vous arrête !*". Il peut également rechercher un brigand. Un commentaire indique alors qu'il placarde une affiche dans toute la ville, et il dit, par exemple, "*OYEZ OYEZ BRAVE GENS ! ! 200 $ a qui arrêtera Bob le brigand mort ou vif ! !*". Tout le monde s'accorde pour dire que les shérifs sont honnêtes. Il est caractérisé par le nombre de brigands qu'il a coffré, information qu'il ne manquera pas de préciser lorsqu'il se présente. Il refuse de se faire appeler autrement que par *Shérif son nom* . Réalisez la classe `Sherif`. Testez.

I.7 Comme un shérif est un cowboy, on peut créer un cowboy en faisant :

```
1                Cowboy Clint = new Sherif("Clint");
```

Testez. Quelles sont les méthodes qui sont appelées (quand le cowboy se présente par exemple). Peut-on demander à ce cowboy de coffrer un brigand ?

II On désire faire des histoires de western plus intéressantes, en faisant intervenir des `Ripoux`. Un ripoux est un shérif qui est brigand. Comment feriez vous ? Mais là c'est une autre histoire . . .

Exercice 5

I On cherche à écrire des classes représentant des figures géométriques et des volumes. Pour chaque classe on définira une méthode `toString()`.

I.1 Définir une classe `Figure` sachant qu'une figure est représentée par la position de son origine dans le plan ;

I.2 Définir une classe `Cercle` dont on connaît le centre, le rayon et dont on calcule la surface lors de la création des instances. Cette classe doit posséder une méthode permettant de modifier le rayon. Quel problème cela peut-il engendrer ?

I.3 Ajoutez une méthode `c1.plusGrand(Cercle c2)` renvoyant `true` si le cercle `c1` est plus grand que `c2`.
Comment écrire une méthode ayant un comportement similaire, mais prenant les deux cercles en argument ?

I.4 Définir une classe `TestGeo` vous permettant de tester les classes précédentes.

Exercice 6

I On va définir des classes qui permettent de résoudre des équations du second degré. Les choix effectués sont mathématiquement contestables mais permettent d'illustrer notre propos. Soit :

$$y = ax^2 + bx + c \text{ avec } a \in \mathbb{R}, b \in \mathbb{R}, c \in \mathbb{R}$$

I.1 On recherche dans un premier temps uniquement les racines réelles. Définir une classe `EquationDu2DegreeSolReel` qui permette de représenter une telle équation et calculer ses solutions si elles existent. Vous protégerez au maximum vos attributs et vous définirez également, en particulier, une méthode `toString()`, une méthode `getX1()` et une méthode `getX2()` permettant d'obtenir les valeurs des racines si elles existent, si ce n'est pas le cas vous devrez lancer une exception `IllegalAccessError`.

I.2 Définir une classe `TestEquationDu2DegreeSolReel` qui permet de calculer les solutions d'une équation dont vous aurez au préalable fourni les coefficients constants.

I.3 On cherche maintenant à étendre notre précédente classe en cherchant à calculer les solutions qu'elles soient réelles ou complexes. Quels sont les problèmes qui se posent ?

I.4 Définir en utilisant le mécanisme d'héritage une classe `EquationDu2Degree` comportant un attribut permettant de savoir si les solutions sont complexes et une méthode permettant la consultation de cette valeur. Définir les méthodes nécessaires pour résoudre l'équation.

I.5 Définir une classe testant la classe précédente.

II En java il existe un moyen de regrouper de grouper des classes voisines dans un package et même des sous-packages ... On va définir un package `libperso.numerique` l'utiliser pour résoudre les équations du second degré.

II.1 Définir une classe `Complexe` implantant les différentes opérations habituelles sur les nombres complexes (addition, module, ...). Cette classe doit appartenir au package `libperso.numerique`.

II.2 En utilisant la classe `Complexe` définir une classe qui permette de résoudre des équations du second degré quelque soit la nature des racines.

3.2 Classes abstraites

Dans la hiérarchie des classes on peut être amené à définir une classe abstraite, c'est à dire une classe générale qui sert de moule (de modèle) aux classes dérivées. Une telle classe n'est pas instantiable et peut contenir des méthodes qui sont également abstraites. Ces méthodes ne sont pas définies dans la classe mais devront obligatoirement être définies dans les classes dérivées non abstraites.

```
1  abstract class UneClasseAbstraite
2  {
3    // ....
4    public abstract void uneMethode();
5  }
6
7  class DeriveeDeClasseAbstraite extends UneClasseAbstraite
8  {
9    // ....
10   public void uneMethode()
11   {
12     System.out.println("uneMethode() est maintenant définie");
13   }
14 }
```

Exercice 7

I On va définir une classe abstraite `SuiteRecurrente` qui doit permettre de stocker le premier terme de la suite, de calculer le terme suivant connaissant le précédent, de calculer le terme de rang n et la somme de $n+1$ premier termes.

I.1 Définir une classe `SuiteRecurrente` possédant :

1. Un attribut qui représente le premier terme ;
2. Une méthode `suivant()` qui calcule le terme suivant connaissant le précédent ;
3. Une méthode `terme()` qui calcule le terme de rang n ;
4. Une méthode `somme()` qui calcule la somme des termes de 0 à n.

I.2 Définir une classe `SuiteGeometrique` qui représente les suites géométriques (terme précédent multiplié par une constante).

I.3 Écrire une classe qui calcule un terme d'un rang donné fournit par un utilisateur ainsi que la somme des termes jusqu'à ce rang.

II Modifier si besoin les classes et écrire une classe qui calcule une suite de Fibonacci. dont la relation de récurrence est la suivante :

$$\begin{cases} \forall n > 0, T_{n+2} = T_n + T_{n+1} \\ \\ \text{Conditions initiales : } T_0 = T_1 = 1 \end{cases} \tag{3.1}$$

Exercice 8

I On cherche à écrire un programme qui permette de gérer une bibliothèque. On considère que la bibliothèque comporte des ouvrages : livres,périodiques et CD. Les livres ont comme propriétés : auteur, titre et éditeur ; les périodiques : nom, numéro et périodicité ; enfin les CDs : titre et auteur. De plus tous les ouvrages ont une date d'emprunt (potentiellement nulle), une cote (le numéro par ordre de création). Chaque objet doit posséder une méthode `toString()` renvoyant toutes les informations sur l'ouvrage sous forme d'une chaîne de caractères. On devra prévoir d'implémenter la liste des ouvrages soit dans un tableau de taille variable ou dans une liste linéaire chaînée.

I.1 Représentez les différentes classes sous la forme d'un diagramme UML. On souhaite que tous les attributs soient déclarés privés et que l'on puisse y accéder de l'extérieur que par des méthodes.

I.2 Implémenter vos différentes classes et en particulier la classe `BibVector` qui représente le contenu de la bibliothèque sachant que la liste des ouvrages est stockée dans un `Vector`.

I.3 Tester vos classes en ajoutant, supprimant ... des ouvrages et en consultant le contenu de la bibliothèque.

II Reprendre l'exercice en utilisant une liste linéaire chaînée pour représenter la bibliothèque. Vous utiliserez la classe `LinkedList` de l'API Java.

3.3 Interfaces

Une interface est constituée d'un ensemble d'opérations autorisées qui spécifient les services offerts par une classe. Une interface est une classe abstraite dont toutes les méthodes sont abstraites. Une interface doit être implémentée par d'autres classes qui de ce fait offrent les services décrits par l'interface.

Une interface permet de dire qu'une classe "*sait faire quelque chose*" alors que l'héritage signifie "*est-un*".

```
interface CapableDeFaireQQChose {
  // Des méthodes abstraites
  public void faitQQChose();  // Ex
}

class implements CapableDeFaire {
  // ....
  public void faitQQChose() // Implémentation de la méthode
                            // abstraite de l'interface
  {
    // Corps de la méthode
  }
}
```

Exercice 9

I Nous voila revenu au Far-West et comme nous l'avions annoncé la corruption y règne, les ripoux y sont présents. On va donc créer une interface `HorsLaLoi`. Cette interface indiquera qu'une classe qui désigne un hors la loi doit être munie d'une méthode pour kidnapper les dames, d'une pour se faire emprisonner par un cowboy, d'une pour donner son nom, et d'une autre pour donner la valeur de la récompense. On mettra par exemple :

```
public interface HorsLaLoi {
  public void emprisonne(Cowboy c);
  public void kidnappe(Dame dame);
  public int getMiseAPrix();
  public String quelEstTonNom();
}
```

La classe `Brigand` devient :

```
public class brigand extends Humain implements HorsLaLoi { }
```

Dans la classe `Brigand`, il faut être bien sur que toutes les méthodes décrites dans l'interface `HorsLaLoi` existent !

I.1 Écrivez l'interface `HorsLaLoi` et dites qu'un brigand est un hors-la-loi.

I.2 On peut également utiliser l'interface `HorsLaLoi` dans les méthodes. Ainsi, un cowboy n'arrête pas seulement les brigands, mais tous les hors-la-loi. De même une dame peut se faire kidnapper par n'importe quelle sorte de hors-la-loi. Modifiez les méthodes concernées pour tenir compte de ceci (vous devriez normalement uniquement remplacer `Brigand` par `HorsLaLoi`). Testez.

I.3 Créez maintenant la classe `Ripoux` en utilisant l'interface `HorsLaLoi`. Créez un ripou dans votre histoire et testez.

I.4 Une femme brigand est une dame qui a décidé de passer de passer du côté des hors-la-loi. Créez la classe `FemmeBrigand` et testez.

I.5 *Ugh !* Un indien est un humain qui est caractérisé par son nombre de plumes (qu'il mentionne quand il se présente) et son totem (*Coyote* par défaut). Il termine toutes ses phrases par *Ugh !*. Sa boisson favorite est le jus de racine. Un indien peut scalper un visage pale (il peut alors s'ajouter une plume). Par conséquent, un visage pale est un humain qu'on pourra scalper. Un visage pale disposera donc d'une méthode `scalp()` (il s'écriera alors, par exemple, `Aïe ma tête !`. Réalisez la classe `Indien` et l'interface `VisagePale`. En particulier, les brigands, les cowboys et les dames sont des visages pales. Testez.

I.6 Générique de fin

```
import java.io.*;
import javax.sound.sampled.*;


public class sound {

    private AudioFormat format;
    private byte[] samples;

    public sound(String filename){
        try{
            AudioInputStream stream =
               AudioSystem.getAudioInputStream(new File(filename));
            format = stream.getFormat();
            samples = getSamples(stream);
        }
        catch (UnsupportedAudioFileException e){
            e.printStackTrace();
    }
    catch (IOException e){
            e.printStackTrace();
        }
    }

    public byte[] getSamples(){
        return samples;
    }

    public byte[] getSamples(AudioInputStream stream){
        int length = (int)(stream.getFrameLength() * format.getFrameSize());
        byte[] samples = new byte[length];
        DataInputStream in = new DataInputStream(stream);
        try{
            in.readFully(samples);
        }
        catch (IOException e){
            e.printStackTrace();
        }
        return samples;
    }
```

```
41
42
43      public void play(InputStream source){
44          // 100 ms buffer for real time change to the sound stream
45          int bufferSize = format.getFrameSize()
46                           * Math.round(format.getSampleRate() / 10);
47          byte[] buffer = new byte[bufferSize];
48          SourceDataLine line;
49          try{
50              DataLine.Info info =
51                 new DataLine.Info(SourceDataLine.class, format);
52              line = (SourceDataLine)AudioSystem.getLine(info);
53              line.open(format, bufferSize);
54          }
55          catch (LineUnavailableException e){
56              e.printStackTrace();
57              return;
58          }
59          line.start();
60          try{
61              int numBytesRead = 0;
62              while (numBytesRead != -1){
63                  numBytesRead = source.read(buffer, 0, buffer.length);
64                  if (numBytesRead != -1)
65                      line.write(buffer, 0, numBytesRead);
66              }
67          }
68          catch (IOException e){
69              e.printStackTrace();
70          }
71          line.drain();
72          line.close();
73      }
74
75      public static void main(String[] args){
76          sound player = new sound("beep.wav"); // <- votre son
77          InputStream stream = new ByteArrayInputStream(player.getSamples());
78          player.play(stream);
79          System.exit(0);
80      }
81  }
```

Exercice 10

I Nous nous intéressons au tri d'un tableau d'objets quelconques.

I.1 Les objets que nous devons trier doivent être comparables pour pouvoir être triés. C'est pourquoi nous allons d'abord définir une classe abstraite, `EstComparable`.

I.2 Tester vos objets comparables en programmant de nouvelles classes dérivées de la classe `EstComparable`.

I.3 Supposons que l'on désire trier des chaînes de caractères. Pour cela, implémentez une classe dérivée de `EstComparable`. Quels sont les problèmes ?

I.4 Reprendre le problème en définissant non plus une classe abstraite mais une interface `EstComparable`.

Exercice 11

I Programmer la classe `CollectionTriee`. Il s'agit de l'implémentation d'une `Collection` qui représente un ensemble d'objets triés. Pour pouvoir être triés, les objets doivent implémenter une interface commune. Cette interface existe au sein de l'API standard de Java : `Comparable`

I.1 La collection peut conserver les éléments qui la composent dans une structure quelconque, du moment que les propriétés suivantes sont vérifiées :
- A chaque fois qu'un élément est ajouté à la collection, une exception est levée si cet élément ne peut être comparé aux autres éléments déjà présents dans la collection. A vous de déterminer quelle est le type d'exception le plus pertinent à envoyer parmi celles qui sont fournies dans l'API standard du JDK.
- L'objet de type `Iterator` renvoyé par la méthode `iterator()` doit parcourir les éléments dans l'ordre croissant.
- Un objet peut être ajouté plusieurs fois dans votre collection. Si c'est le cas, il figurera plusieurs fois dans la liste. Cela vous interdit donc d'utiliser directement la classe `TreeSet`.
- D'après l'interface `Collection` de l'API standard, la méthode `toString()` renvoie une chaîne de caractères représentant la liste des éléments composant une collection. Pour `CollectionTriee`, la liste doit être triée dans l'ordre croissant.

Vous découvrirez un certain nombre d'astuces en lisant la documentation de la classe `Comparable`.

I.2 Testez votre classe avec le programme `TestCollectionTriee.java`

```
1  import java.util.Collection;
2  import java.util.Iterator;
3
4
5  public class TestCollectionTriee
6  {
7    public static void main(String[] args)
8    {
9      Collection collection=new CollectionTriee();
10
11     System.out.println("␣->␣J'ajoute␣les␣éléments␣suivants:");
12     System.out.println("machin");
13     collection.add("machin");
14     System.out.println("truc");
15     collection.add("truc");
16     System.out.println("bidule");
17     collection.add("bidule");
18     System.out.println("chose");
19     collection.add("chose");
20
21     System.out.println("␣->␣Ma␣collection␣contient:␣"+collection);
22
23     System.out.println("␣->␣J'essaye␣d'ajouter␣un␣objet␣non-comparable");
24     try
```

```
25        {
26          collection.add(new Object());
27        }
28        catch (Exception e)
29        {
30          e.printStackTrace();
31        }
32
33        System.out.println("␣->␣J'essaye␣d'ajouter␣un␣nombre");
34        try
35        {
36          collection.add(new Integer(123));
37        }
38        catch (Exception e)
39        {
40          e.printStackTrace();
41        }
42
43        System.out.println("␣->␣J'ajoute␣une␣nouvelle␣fois␣'machin'");
44        collection.add("machin");
45        System.out.println("␣->␣Ma␣collection␣contient:␣"+collection);
46
47        System.out.println("␣->␣Je␣parcours␣ma␣collection");
48        for (Iterator i=collection.iterator(); i.hasNext(); )
49        {
50          System.out.println(i.next());
51        }
52      }
53  }
```

Exercice 12

I On se propose de réaliser les différentes classes nécessaires pour intégrer une fonction d'une variable définie dans $\mathbb{R}$. Les méthodes d'intégration numérique sont utilisées en général lorsque trouver la primitive d'une fonction f est compliqué ou qu'elle est inconnue (la primitive). Un autre cas peut également se présenter lorsque f n'est connue que par points. Pour notre problème, on considère uniquement les intégrales du type :

$$I = \int_a^b f(x)\,dx$$

où $[a, b]$ est un intervalle fini de $\mathbb{R}$ et f est continue sur $[a, b]$.

I.1 Définir une interface `FonctionDUneVariable` qui va permettre de passer en argument de méthodes une fonction mathématique. Cette interface doit comporter deux méthodes `estDefinie()` qui retournera vrai si la fonction est définie au point demandé et une méthode `calculFonctionEn()` qui retourne la valeur de la fonction en un point x.

I.2 Définir une classe `IntegrationTrapeze` qui permette de calculer l'intégrale d'une fonction réelle d'une variable réelle sur un intervalle $[a, b]$ en

utilisant la méthode des trapèzes. On subdivise l'intervalle $[a, b]$ en n sous-intervalles égaux et on approche l'intégrale de la fonction sur la subdivision $[x_i, x_{i+1}]$ par le trapèze passant par $(x_i, 0), (x_{i+1}, 0), (x_{i+1}, f(x_{i+1}),$ et $(x_i, f(x_i))$.

I.3 Tester la classe `IntegrationTrapeze` pour $\int_0^{\Pi/2} sin(x)\,dx$ et pour $\int_1^9 \frac{1}{x}\,dx$.

I.4 L'erreur d'approximation commise dans la méthode précédente dépend de la largeur h de chaque subdivision. On pourrait penser que plus h est petit, plus l'approximation est bonne. Malheureusement, on augmente le nombre d'opérations de manière importante et on obtient un cumul d'erreur d'arrondis non négligeable. Le problème est donc de savoir quel est le bon choix pour h ? On se base sur le principe suivant :

- h doit être petit sur les subdivisions où la fonction à intégrer varie beaucoup ;
- h peut-être plus grand pour les subdivisions où la fonction à intégrer varie peu.

On applique le procédé suivant : pour un intervalle donné $[a, b]$, on effectue deux calculs de l'intégrale le premier avec 10 subdivisions le second avec 5. Si la différence obtenue entre ces deux calculs est faible, il n'est pas utile d'aller plus loin et on garde comme résultat le premier calcul. Si la différence entre ces deux calculs est importante, on subdivise l'intervalle $[a, b]$ en deux sous-intervalles et on réapplique le processus de calcul initial. Écrire la classe correspondante.

I.5 Écrire une classe pour tester cette méthode adaptative. Vous pourrez, par exemple, tester cette méthode pour $f(x) = \frac{sin(x)}{x}$, $\int_0^{10*\Pi} f(x)\,dx$. Vous prendrez en compte la singularité en 0, en fixant $f(0) = 1$. Comparez vos résultats avec la méthode des trapèzes.

Chapitre 4

Entrées/Sorties

Un programme a souvent besoin d'échanger des informations pour recevoir des données d'une source ou pour envoyer des données vers un destinataire. La source et la destination de ces échanges peuvent être de nature multiple :

- un fichier ;
- une socket réseau ;
- un autre programme ;
- etc ...

De la même façon, la nature des données échangées peut être diverse : du texte, des images, du son, etc ... Un flux est en quelque sorte un canal dans lequel de l'information transite. L'ordre dans lequel l'information y est transmise est respecté, le traitement est séquentiel. Un flux peut être : soit une source d'octets à partir de laquelle il est possible de lire de l'information ; Soit une destination d'octets dans laquelle il est possible d'écrire de l'information. En java, les flux peuvent être divisés en plusieurs catégories : les flux d'entrée (`input stream`) et les flux de sortie (`output stream`), les flux de traitement de caractères et les flux de traitement d'octets· Java définit des flux pour lire ou écrire des données mais aussi des classes qui permettent de faire des traitements sur les données du flux. Ces classes doivent être associées à un flux de lecture ou d'écriture et sont considérées comme des filtres. Par exemple, il existe des filtres qui permettent de mettre les données traitées dans un tampon (buffer) pour les traiter par lots. Toutes ces classes sont regroupées dans le package `java.io`.

4.1 Gestion de fichiers

Exercice 13

I Manipulation de fichiers et de répertoires
L'objectif est de prendre en main les classes standards `File`, `FileReader`, et `FileWriter`. Tout d'abord, lisez la documentation fournie. Ensuite vous pouvez commencer à tester ces classes. Ecrivez des petits programmes pour :

I.1 Tester l'existance d'un fichier/répertoire ;

I.2 Lister un repertoire ;

I.3 Lire le contenu d'un fichier et l'afficher sur la sortie standard ;

I.4 Lire un texte au clavier, le nom d'un fichier et sauvegarder le texte dans le fichier.

I.5 Copier un fichier ;

I.6 Supprimer un fichier ;

I.7 Déplacer un fichier.

4.2 Utilisation d'un fichier

Exercice 14

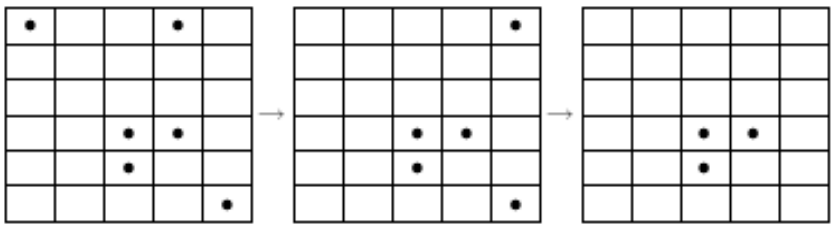
I **Le jeu de la vie**.

Cet exercice nous servira par la suite, on complétera cet exercice en réalisant une interface graphique et en mettant également en œuvre les threads. Dans un premier temps on se contentera d'un affichage texte et de la lecture d'une configuration initiale à partir d'un fichier texte.

Le jeu de la vie a été proposé par J.H. Conway qui s'est intéressé aux travaux de J. Von Neumann concernant les automates autoréplicateurs et les automates cellulaires. Le jeu de la vie qui n'est pas un jeu au sens où on l'entend habituellement se développe sur une grille que l'on représente génèralement sous la forme d'un tore. Il n'y a donc pas de frontière et le coté gauche est connecté au coté droit et le haut au bas. Chaque case de la grille peut contenir éventuellement une cellule. L'évolution du jeu est basée sur trois règles élémentaires :

– *R1* : il y a naissance d'une cellule dans une case vide entourée de trois cellules vivantes exactement ;
– *R2* : une cellule vivante entourée de deux ou trois cellules vivantes survit ;
– *R3* : dans tous les autres cas la cellule meurt.

Voici un exemple sur deux cycles, la configuration est ensuite stable.



Pour programmer ce problème, nous allons modéliser les cellules par des objets d'une classe `Cellule`, la grille de cellules sera représentée par un objet de la classe `Environment` et enfin une classe `JeuDeLaVie` qui gérera l'évolution de l'environnement et son affichage. Ces classes appartiendront un paquetage `jeudelavie`. Vous documenterez toutes vos classes, ...et méthodes.

I.1 On va commencer par définir des interfaces qui permettent de gérer à la fois la grille et son affichage. Cet affichage évoluera d'une version "texte" à une version graphique dans l'avenir. Pour cela commencez par définir une interface `Grille`. Une `Grille` est composée de `nbLignes` * `nbColonnes` cases. Chaque case à une couleur spécifique et un caractère peut être affiché à l'intérieur. L'interface est composée des accesseurs.

```
1 public interface Grille {
2     public int getNbColonnes();
3     public int getNbLignes();
4     public java.awt.Color getCouleurEn(Position p);
```

```
5       public char getCaractereEn(Position p);
6   }
```

Euh là j'ai du tout vous dire, cela ne compte pas pour une question. ...

I.2 Définir la classe `Position` qui définit donc une position dans la grille. N'oubliez pas constructeur, accesseurs, `toString()` et `equals()`.

I.3 Définir une interface `AfficheGrille` comportant en particulier une méthode `affiche(...)`

I.4 Définir une implémentation de votre interface, l'affichage doit se faire sous forme texte. On modifiera par la suite cette implémentation en utilisant l'AWT ou Swing.

I.5 Maintenant que l'on a défini le terrain de jeu, on peut passer à la simulation.

- Une cellule est définie par son état actif ou inactif. Afin de permettre l'affichage d'une cellule, en mode texte ou graphique, on décide qu'une cellule active sera caractérisée par le caractère * et la couleur `java.awt.Color.blue` et une cellule inactive par le caractère espace (`' '`) et la couleur blanche (utilisez des attributs de classe tels que `ACTIVE_COLOR`, `INACTIVE_COLOR`, `ACTIVE_CHAR`, `INACTIVE_CHAR`).
- L'environnement est une grille torique, afin de pouvoir facilement utiliser aussi bien les classes d'affichage en mode texte ou graphique, un objet qui permet de representer l'environnement sera du type `Grille`.
- Il faut gérer l'évolution simultanée de toutes les cellules de l'environnement, et répéter cette évolution pendant un nombre donné de cycles. On veut un affichage du nouvel état de l'environnement après chaque étape de l'évolution (en fonction du jeu de la vie créé cet affichage pourra être graphique ou texte). Vous pouvez choisir de partir d'un état de l environnement aléatoire (éventuellement avec un pourcentage de cellules actives fixé) ou d'une configuration prédéfinie. La classe `JeuDeLaVie` pourra ressembler à cela :

```
1   public class JeuDeLaVie {
2     // l'environnement torique
3     private Environnement environnement;
4     // pour l'affichage
5     private AfficheGrille visualisation;
6
7     public JeuDeLaVie (int width, int height) {
8       environnement = new Environnement(width, height);
9     }
10
11    /** initialise l environnement avec des cellules dont
12        l'état initial est tiré aléatoirement */
13    public void randomInit() { ... }
14
15    /** fixe l'affichage */
16    public void setAfficheGrille(AfficheGrille visualisation)
17    { ... }
18
19    /** exécute le jeu de la vie pendant nbSteps cycles
20        et affiche l environnement après chaque cycle */
21    public void execute(int nbSteps) { ... }
22
```

```
23    /** execute un seul cycle du jeu de la vie.
24        Toutes les cellules évoluent simultanément. */
25    private void executeOneCycle() { ... }
26 }
```

On veut pouvoir, pour une initialisation de l'environnement aléatoire par exemple, exécuter le jeu de la vie par une ligne de commande de la forme : `java JeuDeLaVie -lg 20 -col 10 -nbcycle 100` ou encore `java JeuDelaVie -f config.vie` `config.vie` étant un fichier contenant les informations suivantes :

- Taille de la grille ;
- Nombre de générations maximum ;
- Position des cellules initiales.

Chapitre 5

Interfaces Graphiques

Une caractéristique remarquable du langage Java est de permettre l'écriture de programmes portables avec des interfaces-utilisateur graphiques, alors que partout ailleurs de telles interfaces sont réalisées au moyen de bibliothèques séparées du langage et très dépendantes du systèmes d'exploitation sur lequel l'application doit tourner, ou du moins de la couche graphique employée. La première bibliothèque pour réaliser des interfaces-utilisateur graphiques en Java a été `AWT`, acronyme de `Abstract Windowing Toolkit`. Fondamentalement, l'ensemble des composants de AWT est défini comme la partie commune des ensembles des composants graphiques existant sur les différents systèmes sur lesquels la machine Java tourne. Ainsi, chaque composant d'`AWT` est implémenté par un composant homologue (peer) appartenant au système hôte, qui en assure le fonctionnement. `AWT` est une bibliothèque graphique originale et pratique, qui a beaucoup contribué au succès de Java, mais elle a deux inconvénients : puisque ses composants sont censés exister sur tous les environnements visées, leur nombre est forcément réduit ; d'autre part, même réputés identiques, des composants appartenant à des systèmes différents présentent malgrè tout des différences de fonctionnement qui finissent par limiter la portabilité des programmes qui les utilisent. C'est la raison pour laquelle AWT a été complétée par une bibliothèque plus puissante, `Swing`]. Apparue comme une extension (c.a.d. une bibliothèque optionnelle) dans d'anciennes versions de Java, elle appartient à la bibliothèque standard depuis l'apparition de Java 2. En `Swing`, un petit nombre de composants de haut niveau (les cadres et les boîtes de dialogue) sont appariés à des composants homologues de l'environnement graphique sous-jacent, mais tous les autres composants qui, par définition, n'existent qu'inclus dans un des précédents sont écrits en "pur Java" et donc indépendants du système hôte. Au final, `Swing` va plus loin que `AWT` mais ne la remplace pas ; au contraire, les principes de base et un grand nombre d'éléments de `Swing` sont ceux de `AWT`. Ensemble, `AWT` et `Swing` constituent la bibliothèque officiellement nommée `JFC` (pour Java Foundation Classes).

Les classes de l'`AWT` constituent le paquet `java.awt` et un ensemble de paquets dont les noms commencent par `java.awt` : `java.awt.event`, `java.awt.font`, `java.awt.image`, ...Les classes de `Swing` forment le paquet `javax.swing` et un ensemble de paquets dont les noms commencent par `javax.swing` : `javax.swing.border`, `javax.swing.filechooser`, `javax.swing.tree`, ...

Pour créer une interface graphique utilisateur, on utilise des gestionnaires de placement pour disposer des composants à l'intérieur de conteneurs et contrôler leur taille et leur position. Chaque composant est susceptible d'émettre des événements. À chaque

événement correspond une ou plusieurs méthodes susceptibles d'être déclenchées par le gestionnaire d'événements auprès d'objets appelés `listeners` (ou observateurs/écouteurs). Les listeners doivent donc être des instances de classes qui implémentent ces méthodes. Pour être listener il faut être instance d'une classe qui implémentera l'interface listener prévue pour l'événement considéré. Il peut y avoir autant de listeners qu'on peut le souhaiter abonnés à une source d'événements (un composant), tous les listeners sont prévenus lors de l'émission dun événement. La plupart des événements sont définis dans le paquetage `java.awt.event`.

5.1 L'AWT

Exercice 15

I **Le jeu de Sokoban**

Nous allons réaliser un jeu de Sokoban. Dans ce jeu un petit personnage doit placer des caisses sur des emplacement prédéfinis en les poussant uniquement. Le jeu se joue sur des cartes qui spécifient l'emplacement des murs et les positions finales des caisses.

I.1 Définir une classe `Carte` qui possède les méthodes suivantes :

1. Un constructeur qui prend en entrée un fichier décrivant la carte ;
2. `placeCaisse(c,x,y)` place la caisse `c` à la position `(x,y)` ;
3. `retireCaisse(x,y)` retire la caisse placée à la position `(x,y)` ;
4. `Mur(x,y)` indique si oui ou non il y a un mur à la position `(x,y)` ;
5. `position(x,y)` fait de même pour les positions finales des caisses ;
6. `caisse(x,y)` nous renvoie l'objet caisse qui se trouve à la position `(x,y)` (et `null` si il n'y en a pas).
7. `paint(g)` qui dessine la carte sur le graphique (un carré gris pour les murs et un rond bleu pour les positions finales).

I.2 Les lutins

On réalise maintenant les classes correspondantes aux caisses et au personnage (classe `Caisse` et classe `Joueur`). Ces deux classes ont des éléments en communs. Une caisse ou un joueur peuvent être dessinés et déplacés. Créer une classe `Lutin` qui contiendra les éléments communs. Un lutin (élément graphique mobile, sprite en anglais) est caractérisé par sa position `(x,y)`, la carte sur lequel il évolue et peut être manipulé par les méthodes suivantes :

1. `paint(Graphics g)` : il s'affiche à sa position sur le graphique `g` (on travaillera sur quadrillage 20 x 20, les coordonnées sont donc à multiplier par 20). Pour l'instant, un lutin n'affiche rien de particulier.
2. Les méthodes pour le déplacer `haut()`, `bas()`, `gauche()` et `droite()`. Ces fonctions renvoient `true` si le déplacement a pu s'effectuer, `false` sinon.

I.3 Les caisses

Une caisse est un lutin particulier. Quand on la dessine, elle affiche une caisse. La couleur de cette caisse varie selon qu'elle soit sur une position finale de caisse ou non. Quand on demande à une caisse de se déplacer, elle

ne le peut pas toujours : si elle a en face d'elle un mur ou une autre caisse, le déplacement est refusé. Attention à conserver la cohérence de l'état de la carte.

I.4 Le joueur
Un joueur est également un lutin particulier. Quand il veut se déplacer, il ne le peut pas forcément. S'il y a un mur devant lui, il ne peut pas se déplacer. S'il y a une caisse devant lui, il ne peut se déplacer que si la caisse peut également se déplacer. S'il n'y a rien devant lui, tout va bien. Vous choisirez également l'apparence du personnage.

I.5 La fenêtre
Le jeu va se dérouler dans une fenêtre (`Frame`). Vous penserez à réécrire la fonction `paint` (qui demandera à la carte de s'afficher, puis à tous les lutins) Pour les besoins du jeu, la carte sera créée à l'aide du fichier "carte.map", le joueur sera placé aux coordonnées (2,5) et 4 caisses seront créées aux coordonnées (2,4), (4,5), (6,6) et (7,5) (vous pourrez améliorer cela par la suite).

II Commentez toutes vos classes pour pouvoir utiliser `javadoc`. Vous le ferez bien sur durant le développement. Utilisez javadoc.

III Testez votre classe `Sokoban` en codant les déplacements en "dur" dans votre code.

IV Gérez les événements sur votre fenêtre principale.

V Gérez les déplacements du joueur à partir des événements issus du clavier.

VI Modifiez votre programme pour que dans la fenêtre s'affiche le nombre de caisses bien placées.

VII Faîtes une `Applet` de votre programme.

5.2 Swing

Exercice 16

I **Réalisation d'une calculette élémentaire**
Le programme qu'on vous demande de réaliser ici simule une calculette du modèle le plus simple.
- Onze touches (dix chiffres et le point décimal) permettent de composer les nombres.
- Quatre touches correspondent aux quatre opérations que la calculette sait faire. Il n'y a pas de touche =, chaque pression d'une touche d'opération effectue l'opération en attente.
- Enfin, une touche *C* permet d'effacer le nombre en cours de saisie ou, si aucun nombre n'est en cours de saisie, le dernier nombre précédemment saisi ou calculé.
- Un interrupteur à trois positions (représenté ici par trois "boutons-radio") permet de choisir le nombre de décimales parmi : aucune, deux ou six.
- Bien entendu, on peut saisir les nombres au clavier (de l'ordinateur) : il suffit de cliquer auparavant dans la fenêtre d'affichage de la calculette.

Indications. Il conviendra de maintenir un indicateur qui renseigne sur le statut du nombre affiché par la calculette. Ce peut être un nombre en cours de saisie

(et alors chaque touche numérique frappée lui ajoute des chiffres) ou bien ce peut être un nombre achevé, comme le résultat d'une opération (la frappe d'une touche doit alors le faire disparaître et le ranger comme premier opérande d'une opération à venir).
La frappe d'une touche d'opération effectue l'éventuelle opération en attente et met en attente l'opération correspondant à la touche pressée. La frappe de deux touches d'opération consécutives fait oublier la première, et retenir la deuxième.
Par exemple, si l'opérateur frappe successivement les touches 1 2 3 $\times$ 2 $+$ 2 4 $\times$ / 9 et $+$, on aura le comportement suivant :

touche pressée	affichage
1	1
2	12
3	123
$\times$	123,00
2	2
$+$	246,00
2	2
4	24
$\times$	270,00
/	270,00
9	9
$+$	30,00

Exercice 17

I **Le jeu de la vie (suite)**. Vous avez utilisé precedemment une sortie texte pour montrer l'évolution du jeu de la vie, en réutilisant le code déjà écrit et en utilisant Swing, vous devez réaliser une sortie graphique.

I.1 En utilisant l'interface *AfficheGrille* définir une interface graphique dans une `JFrame` qui affiche à chaque pas de temps le contenu de votre grille.

I.2 Ajouter un menu permettant de choisir un fichier initial et de sauver l'état de votre grille. Pour choisir un fichier initial de configuration vous utiliserez un `JFileChooser`.

I.3 Modifier et compléter votre interface avec une barre d'outils comportants des boutons : *run*, *stop* et *step* ;

I.4 L'utilisateur doit pouvoir maintenant cliquer sur votre grille, pour ajouter ou supprimer des cellules.

Exercice 18

I **La ronde des fourmis**
On considère une grille sur laquelle on va placer des fourmis. Chaque cellule de la grille contient une flèche qui indique la direction (nord, sud, est, ouest) que doit emprunter la fourmi qui arrive dessus. Les déplacements s'effectuent de la manière suivante :
– Les fourmis sont déposées les unes après les autres au centre de la grille ;
– Lorsque la fourmi arrive sur une cellule non occupée, elle y reste définitivement ;

– Lorsque la fourmi arrive sur une cellule occupée elle fait tourner la flèche de la cellule ; d'un quart de tour dans le sens des aiguilles d'une montre et se déplace vers la cellule pointée ;

Chapitre 6

Les Threads

Un thread est une unité d'éxecution faisant partie d'un programme. Cette unité fonctionne de façon autonome et parallèlement à d'autres threads. En fait, sur une machine mono processeur, chaque unité se voit attribuer des intervalles de temps au cours desquels elles ont le droit d'utiliser le processeur pour accomplir leurs traitements. La gestion de ces unités de temps par le système d'exploitation est appellée scheduling. Il existe deux grands types de scheduler :

- Le découpage de temps : le système d'exploitation (Windows par exemple) attribue un intervalle de temps prédéfini quelque soit le thread et la priorité qu'il peut avoir. ·
- La préemption : utilisée par les systèmes de type Unix. Le système attribue les intervalles de temps en tenant compte de la priorité d'exécution de chaque thread. Les threads possédant une priorité plus élevée s'exécutent avant ceux possédant une priorité plus faible.

· Le principal avantage des threads est de pouvoir répartir différents traitements d'un même programme en plusieurs unités distinctes pour permettre leur exécution "simultanée". La classe `java.lang.Thread` et l'interface `java.lang.Runnable` sont les bases pour le développement des threads en java. Par exemple, pour exécuter des applets dans un thread, il faut que celles ci implémentent l'interface `Runnable`. Le cycle de vie d'un thread est toujours le même qu'il hérite de la classe `Thread` ou qu'il implémente l'interface `Runnable`. L'objet correspondant au thread doit être créé, puis la méthode `start()` est appelée qui à son tour invoque la méthode `run()`.

Exercice 19

I *Nombre de Fibonacci*

Écrire un programme qui calcule le n^{ième} nombre de Fibonnaci F_n en utilisant la relation de récurrence suivante :

$$F_n = F_{n-1} + F_{n-2} \text{ pour } n \geq 2$$

et

$$F_0 = 1, F_1 = 1 \text{ pour } 0 \leq n < 2$$

Pour calculer F_n, la méthode run doit créer deux threads qui calcule F_{n-1} et F_{n-2} récursivement. Le thread principal attend les deux threads qu'il a créé avec la méthode `join` pour terminer le calcul.

Exercice 20

I En vous inspirant de l'exercice précédent reprendre le problème d'intégration adaptative en utilisant des threads.

Exercice 21

I **Tris parallèles**.

I.1 Voici un algorithme de tri en ordre croissant d'une tranche de tableau comprise entre les éléments d'indices `debut` et `fin` :

```
1  trier(debut, fin) {
2    si (fin - debut < 2) { // on a fini, pas d'appel récursif
3      si (t[debut] > t[fin])
4        echanger(t[debut], t[fin])
5    }
6    sinon {
7      milieu = (debut + fin) / 2
8      trier(debut, milieu)
9      trier(milieu + 1, fin)
10     //fusion des 2 moitiés debut à milieu et milieu + 1 à fin
11     fusion(debut, fin)
12   }
13 }
```

Écrire une version mono-tâche de cet algorithme.

I.2 On remarque que les 2 tris qui sont effectués avant la fusion sont indépendants l'un de l'autre et il est donc facile des les faire exécuter en parallèle par 2 threads. En utilisant `wait()` et `notify()`, codez une version multi-tâches de cet algorithme. Si vous lancez des threads depuis la méthode `main()`, n'oubliez pas d'attendre la fin de leur exécution avant d'afficher le résultat final. Vous pouvez pour cela utiliser `join()`.

I.3 Codez une nouvelle version en utilisant cette fois-ci `join()` au lieu de `wait()` et `notify()`. C'est plus facile mais souvent `join()` n'est pas assez souple pour remplacer `wait()` et `notify()`,

Exercice 22

I Reprendre le jeu de la vie et utiliser des threads.