Université de Nice - Sophia Antipolis
Faculté des Sciences

DEUG MIAS MP1

Programmation 2001-02

3. ETENDRE LA CLASSE TURTLE…

A. Des tortues ivres

Mais oui, vous vous en souvenez : il s'agissait du dernier exercice du partiel n°2 du 1er semestre ! Vous deviez écrire une classe d'objets `TortueIvre` dont les instances étaient des « tortues ivres » ne réagissant qu'au seul message *forward* et l'interprétant en faisant un *forward* usuel dans une direction aléatoire :

```
import unsa.Turtle;

class TortueIvre {                      // solution par composition
  private Turtle t;                     // une tortue en attribut privé

  TortueIvre() {                                // le constructeur
    t = new Turtle();
  }

  void forward(int distance) {                  // le forward spécial…
    double angle = Math.random() * 360;
    t.left(angle);
    t.forward(distance);
  }

  public static void main(String[] args) {      // pour tester…
    TortueIvre lea = new TortueIvre();
    for(int i = 0; i < 20; i++)
      lea.forward(30);
  }
}
```

Cette manière de définir des objets en utilisant d'autres types d'objets se nomme la **composition** dans le monde Java qui a son petit jargon technique… Notez qu'avec cette solution, une « tortue ivre » n'est plus une tortue, elle « contient » une tortue qui d'ailleurs ne sait pas faire autre chose que le *forward* spécial !

L'autre manière de faire consiste à jouer sur l'**héritage**, c'est-à-dire à étendre la classe `Turtle`. Dans cette optique, un peu différente, une « tortue ivre » reste une tortue, dotée d'un `forward` différent des tortues « normales ». Quant aux autres méthodes, on peut les laisser inchangées [sauf `back` quand même] ou les modifier.

Exercice 3.1 a) Ecrire la classe `TortueIvre` comme sous-classe de `Turtle`. Cette classe n'aura qu'un seul constructeur `TortueIvre(Color c)` créant une tortue ivre dont la couleur du crayon est l'objet `c` de la classe `Color` [rappel : `Color.red` etc, et n'oubliez pas d'importer `java.awt.Color`]. Vous redéfinirez les méthodes `forward(…)` et `back(…)`. Les autres méthodes tortue resteront inchangées. Vous rajouterez une méthode d'instance `carreDistOrigine()` retournant le carré de la distance de la tortue à l'origine O du repère.

b) Compilez votre classe, puis – au toplevel – faites avancer une tortue ivre 20 fois d'une longueur de 30.

c) Un physicien des particules s'intéresse à la variable aléatoire représentée par le carré d^2 de la distance d de la tortue à l'origine après qu'elle ait fait n pas unitaires [de longueur 1]. En tant que programmeur, pouvez-vous l'aider à faire une conjecture sur l'espérance $E(d^2)$ de cette variable aléatoire ? *Indication : prendre par exemple n = 1000 pas, et faire une moyenne sur 100 expériences…*

➽ Que se passe-t-il si l'on redéfinit en la désactivant la méthode `setPosition(…)` dans la classe `TortueIvre` :

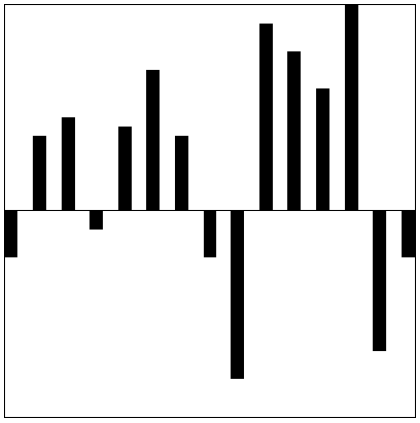
```
public void setPosition(double x, double y) {
  System.out.println("La méthode setPosition est désactivée pour les tortues ivres !");
}
```

B. La classe TortueDiagramme

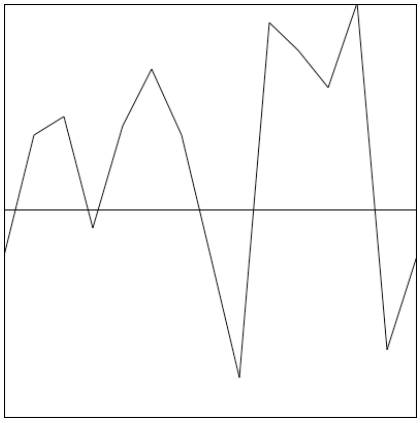
Les instances de cette sous-classe de `Turtle` seront des tortues disposant en variable privée d'un tableau de nombres approchés [double], et sachant afficher un tel tableau de manière graphique, sous la forme de bâtons ou de points reliés par des segments. Par exemple, supposons que la « tortue diagramme » de nom `lea` gère un tableau de températures :

```
TabData = {-5,8,10,-2,9,15,8,-5,-18,20,17,13,22,-15,-5}
```

Nous souhaitons demander à `lea` d'avoir l'amabilité de dessiner ce tableau sous l'une des deux formes ci-dessous :



`lea.histogramme();`



`lea.ligneBrisee();`

Dans les deux cas, la figure prend toute la fenêtre, aussi bien en largeur qu'en hauteur. La largeur des bâtons de gauche est calculée en fonction du nombre n de bâtons sachant qu'une fenêtre tortue est un carré de 400 pixels de côté, et que l'espacement entre deux bâtons est égal à la largeur d'un bâton. Dans la fenêtre de droite, on se contente de relier les points par un segment et l'espacement entre deux abscisses sur l'axe Ox est aussi calculée en fonction de n. Vous noterez que l'on s'arrange pour que le bâton le plus long soit de hauteur 200 [nous « normaliserons » le tableau] !

Exercice 3.2 **a)** Si le nombre de bâtons est égal à n [longueur du tableau `tabData`], quelle sera la largeur d'un bâton ? Quel sera l'espacement entre deux bâtons ?
b) Dans la fenêtre de droite, quel sera l'espacement entre deux abscisses sur l'axe Ox en fonction de n ?

Variable privée et constructeur

Exercice 3.3 Commencez à rédiger la classe **`TortueDiagramme`** qui étend la classe `Turtle`.

a) Elle comporte une seule variable privée **`tabData`** qui sera un tableau de nombres approchés. L'unique constructeur prendra un seul argument : un tableau de nombres approchés qui servira à initialiser la variable `tabData`.

b) Ecrivez la méthode **`toString()`**. Elle rendra comme résultat une chaîne de la forme :

```
TortueDiagramme[15,{-5.0,8.0,10.0,-2.0,9.0,15.0,8.0,-5.0,-18.0,20.0,17.0,13.0,22.0,-15.0,-5.0}]
```

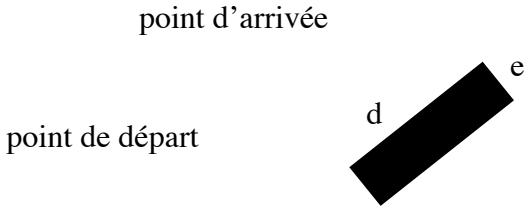
Utilitaire de normalisation [méthode de classe]

Exercice 3.4 Le contenu du tableau `tabData` étant quelconque, on peut y trouver des nombres trop grands pour la fenêtre, ou inversement trop petits à visualiser. Vous allez donc écrire une méthode statique **`normalise(…)`** de signature : **`double[ ] → double[]`** qui va prendre un tableau `t` de nombres approchés et retourne un nouveau tableau obtenu à partir de `t` en forçant l'élément de plus grande valeur absolue à avoir une valeur absolue égale à 200 [le bâton le plus long touchera donc juste le bord de la fenêtre]. La méthode va commencer à calculer `m = Max{|t[i]|}` puis construira un tableau de même longueur que `t` en appliquera un coefficient multiplicatif convenable.

Par exemple, la normalisation du tableau `{1, -3, 2}` donnera le tableau `{66.66..,-200.0,133.33…}`.

Méthodes d'instance

Exercice 3.5 Une petite méthode d'instance utilitaire susceptible d'être ré-utilisée plus tard. La méthode **`gForward(int d, int e)`** qui fait avancer la tortue de d pixels en direction de son cap courant mais en laissant une trace épaisse [«*gras forward*»]. Elle trace en réalité un rectangle plein de longueur `d` et de largeur `e`, la trajectoire de la tortue formant le côté gauche de ce rectangle.



Exercice 3.6 Vous allez commencer par écrire la méthode d'instance **`histogramme()`** sans argument et sans résultat. Elle commence par tracer l'axe Ox qui partage la fenêtre en deux au milieu. Ensuite, elle décide de travailler sur une copie «normalisée» [exercice 3.4] du tableau `tabData`. Après avoir calculé la largeur d'un bâton [égale à l'espacement entre deux bâtons, exercice 3.2], elle parcourt le tableau normalisé contenant la hauteur des bâtons et dessine ces derniers [avec `gForward(…)` de l'exercice 3.5].

Exercice 3.7 Ensuite, la méthode **`ligneBrisee()`**. Elle est un peu plus simple que la précédente. Elle travaille elle aussi sur une copie normalisée de `tabData`. Les lignes sont des segments tracés par des `forward` ordinaires. On relie les sommets des bâtons, pas de problème particulier [utiliser l'exercice 3.2 pour l'espacement entre les points]…

N.B. i) Chacune des deux méthodes `histogramme()` et `ligneBrisee()` calcule un tableau «normalisé», en réalité le même. Vous pouvez réfléchir à la manière de programme pour que chacune profite du calcul effectué par l'autre [on ne sait pas laquelle sera invoquée en premier]. Bien entendu, on laisse `tabData` intact car il se peut que l'on ultérieurement à faire par exemple des calculs statistiques sur les données originales.
ii) Pour tester les deux méthodes avec la même tortue, donc dans la même fenêtre, il est bon de disposer d'un outil de temporisation, une méthode `delay(int ms)` à qui on passe un entier représentant une durée en millisecondes [par exemple 4000] et dont le but consiste à ne rien faire pendant ce laps de temps. C'est l'exercice qui suit.

Exercice 3.8 Ecrire la méthode `delay(…)` de signature `int` → `void`. Elle utilisera la méthode primitive `System.currentTimeMillis()` qui retourne un entier `long` correspondant à l'heure actuelle, exprimée en millisecondes, depuis une origine des temps arbitraire. Il suffit alors de boucler sans rien faire [enfin si, en gardant l'œil fixé sur le chrono] jusqu'à ce que la durée écoulée dépasse l'argument…