Université de Nice - Sophia Antipolis
Faculté des Sciences

DEUG MIAS MP1

Programmation 2001-02

4. LA TORTUE DU MATHEUX

A. LES CLASSES ANONYMES.

Lorsque l'on instancie un objet, il est créé selon le 'moule' donné par sa classe. Si pour un objet on souhaite modifier le comportement d'une méthode, avec les mécanismes que l'on a introduits jusqu'à maintenant, on est obligé de créer une nouvelle classe rien que pour lui. Pour éviter de devoir déclarer des classes pour une instanciation on utilise une classe anonyme.

Définition : Une **classe anonyme** est une classe déclarée à l'instanciation d'un objet et pour cet objet uniquement. Elle est dite anonyme car elle n'a pas de nom et par conséquent pas de constructeur. Les constructeurs utilisés seront ceux de la classe mère. Une classe anonyme peut avoir des initialisations de variables, des méthodes, et elle peut surcharger les méthodes qu'elle hérite.

Les classes anonymes évitent en particulier la multiplication des classes mais elles obscurcissent le code en introduisant une déclaration de classe au milieu d'une méthode. On évite donc de faire des classes anonymes trop grandes, et surtout on fait bien attention à l'indentation du code pour ne pas se perdre.

Les classes anonymes ont accès aux variables d'instance et de classe de la classe à l'intérieur de laquelle elles sont crées. Elles ont aussi accès aux paramètres et aux variables locales de la méthode où elles sont créées à condition qu'elles soient déclarées **final**.

La syntaxe est la suivante : après l'instruction `new LaClasseMere ()` et avant le point virgule, on ajoute un bloc de déclaration permettant d'étendre et de modifier la déclaration de `LaClasseMere` comme pour une sous-classe classique. L'objet issu du new est un objet de la classe anonyme qui étend la classe `LaClasseMere`. Comme sa classe est anonyme on ne peut le récupérer que dans une variable du type `LaClasseMere` ou d'un type ancêtre de `LaClasseMere`. Considérez l'exemple des deux classes suivantes :

```
class FonctionParametree {
  public double x(double t) { return 0; }
  public double y(double t) { return 0; }
}
```

Ces méthodes renvoient toujours 0

```
class Test {
  public static void main(String[] arg){
    FonctionParametree ellipse = new FonctionParametree(){
      public double x(double t){ return 3*Math.cos(t);}
      public double y(double t){ return 4*Math.sin(t);}
    };
    System.out.println("x(2)="+ellipse.x(2)+" y(2)="+ellipse.y(2));
  }
}
```

Classe anonyme sous classe de FonctionParametree

Redéfinition des méthodes pour la classe anonyme

Le point virgule du new !!!!

```
Welcome to DrJava.
> java Test
x(2)=-1.2484405096414273 y(2)=3.637189707302727
>
```

La surcharge est effective : cet objet est bien une instance de la classe anonyme puisqu'il ne renvoie pas 0.

Ici, l'utilisation des classes anonymes permet de ne pas avoir à définir une classe pour chaque fonction paramétrée, et de ce contenter d'étendre la classe et de surcharger les méthodes lors de la création d'une fonction particulière.

Exercice 4.1 Implantation de l'exemple : Implantez ces deux classes dans deux fichiers séparés `FonctionParametree.java` et `Test.java` et testez les tel que cela est fait dans l'exemple.

B. MODELISATION DES FONCTION PARAMETREES ET DES SUITES RECURRENTES.

L'exemple précédent donnait la classe des fonctions paramétrées. Les fonctions cartésiennes sont des fonctions paramétrées pour lesquelles on connaît une fonction f telle que $y=f(x)$ et les fonctions polaires sont des fonctions paramétrées pour lesquelles on connaît $r=f(q)$ où r est la distance à l'origine et q l'angle (Ox,r).

Exercice 4.2 Implantation des fonctions

a) Dans un fichier `FonctionCartesienne.java` implantez la classe `FonctionCartesienne` qui étend la classe `FonctionParametree` et ajoute une méthode `public double getValeur(double x)` correspondant à la fonction *f*. Redéfinissez aussi les méthodes `public double y(double t)` et `public double x(double t)` pour les adapter aux fonctions cartésiennes.

b) Dans un fichier FonctionPolaire.java implantez la classe `FonctionPolaire` qui étend la classe `FonctionParametree` et ajoute une méthode `public double r(double teta)`. Redéfinissez aussi les méthodes `public double y(double t)` et `public double x(double t)` pour les adapter aux fonctions polaires.

c) Dans le `main()` de la classe `Test` créez une spirale ($r=q/4$) dont vous afficherez les coordonnées pour $q=10$ et une fonction sinus dont vous afficherez la valeur pour $x=1$

Une suite récurrente u est définie par: $u_{n+1}=f(u_n)$ et u_0 où f est une fonction cartésienne telle que définie précédémment et u_0 est une constante d'initialisation de la suite. Les suites arithmétiques sont des suites récurrentes dotées d'une raison r telle que $f(x)=x+r$. Les suites géométriques sont des suites récurrentes dotées d'une raison r telle que $f(x)=x*r$.

Exercice 4.3 Implantation des suites

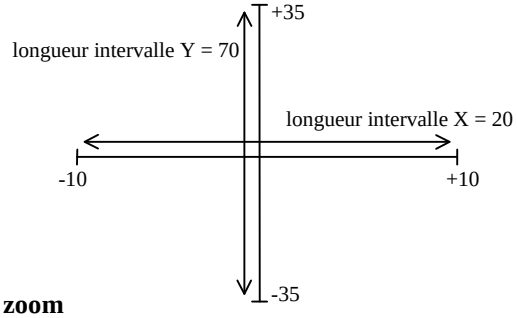
a) Dans un fichier `SuiteRecurrente.java` implantez la classe `SuiteRecurrente` ayant : (1) un constructeur qui attend une valeur initiale et une fonction cartésienne (2) une méthode `initialise()` qui remet la suite au rang $n=0$ et (3) méthode `termeSuivant()` qui donne un terme de la suite et passe au suivant.

b) Dans deux fichiers séparés implantez les classes `SuiteArithmetique` et `SuiteGeometrique` n'ayant qu'un seul constructeur qui attend la valeur initiale et la raison. Ces classes étendent et utilisent `SuiteRecurrente`.

c) Dans le `main()` de la classe classe `Test` créez une suite arithmétique, une suite géométrique, une suite récurrente pour lesquelles vous afficherez les cinq premiers termes.

C. LA TORTUE MATHEUSE.

Nous allons maintenant utiliser les tortues pour tracer nos fonctions et nos suites. Une tortue dispose d'une fenêtre de 400×400 points. A ces dimensions nous allons faire correspondre une longueur d'intervalle X et une longueur d'intervalle Y centrés sur l'origine du repère (le repère ne sera donc pas forcément orthonormé). Par exemple, une longueur d'intervalle X égale à 20 unités et une longueur d'intervalle Y égale à 70 unités donnera un repère gradué de $x=-10$ à $x=+10$ et de $y=-35$ à $y=+35$:



Exercice 4.4 Repère et rapports de zoom

a) Dans un fichier TortueTraceuse.java implantez la classe `TortueTraceuse` qui étend la classe `Turtle`.

b) La classe `TortueTraceuse` a six variables d'instances:

```
double maLongueurIntervalleX;       // Longueur réelle de l'intervalle des X
double maLongueurIntervalleY;       // Longueur réelle de l'intervalle des Y
double monPasEnX;   // Variation élémentaire en X correspondant à un point à l'écran
double monPasEnY;   // Variation élémentaire en Y correspondant à un point à l'écran
double monFacteurEnX;         // Facteur de zoom en X
double monFacteurEnY;         // Facteur de zoom en Y
```

c) Le constructeur `public TortueTraceuse(double longueurIntervalleX, double longueurIntervalleY)` initialise ces constantes et trace le repère avec une graduation unitaire pour *Ox* et *Oy*

d) Testez votre œuvre d'art en affichant un repère sur un intervalle de longueur 20 en X et de longueur 10 en Y.

Les valeurs calculées pour `monPasEnX`, `monPasEnY`, `monFacteurEnX`, `monFacteurEnY`, seront utilisées pour tracer les courbes à l'échelle : les pas donnent la correspondance entre la taille d'un point et la longueur d'un segment correspondant dans l'intervalle en X ou en Y, et les facteurs donnent l'inverse.

Exercice 4.5 **Tracé de fonctions et de suites.**

a) Ajoutez la méthode `public void trace(FonctionParametree uneFonctionParametree, Color uneCouleur, double tMin, double tMax, double tInc)` permettant de tracer des fonctions paramétrées pour un paramètre t variant de `tMin` à `tMax` par pas de `tInc`.

b) Utilisez cette méthode pour tracer l'ellipse et la spirale définies précédemment.

c) Ajoutez une méthode plus pratique pour tracer les fonctions cartésiennes :
`public void trace(FonctionCartesienne uneFonctionCartesienne, Color uneCouleur)`
vous réutiliserez la méthode `trace` précédente avec un intervalle et un incrément connus (si si, je vous l'assure).

d) Utilisez cette méthode pour tracer la fonction sinus définie précédemment.

e) Ajoutez la méthode `public public void trace(SuiteRecurrente uneSuite, Color uneCouleur){` permettant de tracer des suites récurrentes.

f) Utilisez cette méthode pour tracer les trois suites définies précédemment.

Pour une fonction cartésienne f on peut obtenir une approximation de sa dérivée *f'* en *x* par $\frac{f(x+h)-f(x)}{h}$ lorsque *h* est très petit. En informatique on n'aime pas les infinis, mais comme toutes les valeurs réelles sont approximatives on n'en a pas besoin : on prendra un *h* tellement petit que la précision sera suffisante.

Exercice 4.6 **Tracé d'une dérivée de fonction cartésienne.**

a) Ajoutez la méthode `public public void derive(FonctionCartesienne uneFonctionCartesienne, Color uneCouleur)` permettant de tracer la dérivée d'une fonction en utilisant la formule $\frac{f(x+h)-f(x)}{h}$ ou *x* est l'abscisse du point courant et *h* est le pas de l'intervalle des X.

b) Utilisez cette méthode pour tracer la dérivée de la fonction sinus définie précédemment.

Pour une fonction cartésienne *f* on peut obtenir une approximation de la valeur primitive (à une constante près) par la méthode des rectangles : on fait la somme de l'aire géométrique (l'aire est signée : elle peut être négative) de rectangles dont la base est très petite (nous utiliserons le pas de notre intervalle en X) et dont la hauteur est égale à la valeur de la fonction :

$$\sum_{i=0..n-1} f\left(\frac{x\cdot i}{n}\right)\cdot\frac{x}{n}\xrightarrow[n\to\infty]{}\int_0^x f(t)dt$$

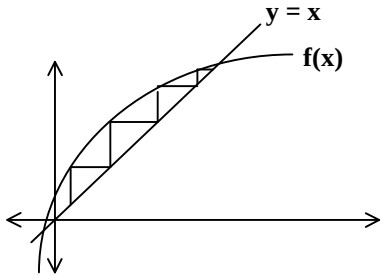
On obtient ainsi une approximation de la primitive de *f* qui s'annule en 0. Pour obtenir une valeur de cette même primitive avant 0 il suffit de faire des soustractions en partant de 0.

Exercice 4.7 **Tracé d'une primitive de fonction cartésienne.**

a) Ajoutez la méthode `public public void integre(FonctionCartesienne uneFonctionCartesienne, Color uneCouleur, double constante)` permettant de tracer la primitive qui prend la valeur `constante` en 0.

b) Utilisez cette méthode pour tracer la primitive de la fonction sinus (définie précédemment) qui vaut -1 en 0.

Enfin notre modélisation mathématique des fonctions et des suites nous donne tous les outils pour programmer la méthode du **point fixe** pour résoudre *x=f(x)*. On peut montrer que $u_{n+1}=f(u_n)$ converge vers la valeur *s* solution de *x=f(x)* si *f* est contractante *i.e.* si $|f(x)-f(y)|<k\times|x-y|$ avec $0<k<1$ et en particulier si $|f'(s)|\leq k<1$. (cf. http://chronomath.irem.univ-mrs.fr/chronomath/PtFixe.html)



Exercice 4.8 **Méthode du point fixe.**

a) Ajoutez la méthode `public void pointFixe(FonctionCartesienne uneFonctionCartesienne, double xDeDepart, double epsilon)` permettant de trouver le point fixe avec une précision de `epsilon`. Vous utiliserez une suite définie grâce à la fonction `uneFonctionCartesienne` et à la valeur de départ `xDeDepart`. Comme montré sur le dessin, la méthode tracera (1) la fonction cartésienne (2) la droite *y = x*, et (3) les itérations de recherche du point fixe. Enfin, la méthode affichera l'approximation obtenue.

b) Testez votre méthode sur $f(x)=2\cdot\log(x^2+2)$ puis sur $f(x)=3\cdot\log(8-x)$