Université de Nice - Sophia Antipolis
Faculté des Sciences

DEUG MIAS MP1

Programmation 2000-01

6. COMPLEMENTS SUR LES CLASSES. INTERFACES

Les « classes anonymes » sont des classes sans nom, qui ne servent qu'une seule fois. Les « interfaces » sont des sortes de classes incomplètes, des « contrats » qu'il faut implémenter concrètement dans une classe. Ces notions vont nous servir dans les feuilles suivantes, pour construire des programmes avec des composants graphiques…

A. Les « classes anonymes »

En principe on nomme une classe avec le mot-clé `class` suivi d'un nom de classe, suivi éventuellement du mot-clé `extends` si l'on crée une sous-classe :

```
class Compte                class CompteEpargne extends Compte
{...}                       {...}
```

Il se trouve qu'il peut être intéressant d'utiliser « à la volée » une classe sans nom [on dit *anonyme*] car elle ne servira qu'une seule fois, de manière temporaire. En réalité on devrait parler de *sous-classe anonyme* car on procède par extension d'une classe existante avec redéfinition de certaines méthodes. Et l'instantiation se fait en même temps [en effet une classe anonyme ne peut avoir de constructeur puisqu'elle n'a pas de nom !] :

```
class TestClasseAnonyme
{  public static void main(String[] args)
  {  Compte c = new Compte(2000);                          // usuel
     System.out.println(c);
     c = new Compte(500)                                   // avec redéfinition de méthode
               {  public String toString()
                  {  return "<bilan=" + this.solde + ">";     // solde est "protected"
                  }
               } ;
     System.out.println(c);
}
```

Pour obtenir une instance d'une classe anonyme dérivée de la classe A, la syntaxe la plus courante est en effet :

```
new A (<paramètres>) { <redéfinition de méthodes de A>}
```

Application : passer une fonction en paramètre à une méthode !

Nous verrons d'autres applications avec les interfaces graphiques plus tard. Mais pour les matheux, le fait de pouvoir travailler avec une fonction quelconque est important, par exemple pour calculer une racine approchée de `f(x)=0` lorsque `f : [a,b]` → **R** est continue et vérifie `f(a) < 0 < f(b)`, avec une précision `epsilon` [cf Exercice 2.7] :

Nous aimerions bien qu'il existe un type `Function` et que l'on puisse écrire :

```
static double racineDicho(Function f, double a, double b, double epsilon)
```

En Java, `Function` doit donc être une classe. Une fonction, instance de cette classe, sera un objet capable de dire ce qu'il vaut en un point `x`, autrement dit doté d'une méthode d'instance que nous décidons d'appeler `valueAt(…)` :

```
class Function
{   double valueAt(double x)   // destinée a être redéfinie dans des classes anonymes !!!
    {  return(x);              // par défaut, c'est donc l'identité x → x
    }
}
```

Exercice 6.1 a) Compiler la classe `Function` ci-dessus une fois pour toutes dans un fichier séparé. Testez-la dans le `main(…)` de `Numerik`.

b) Programmer ensuite dans votre classe `Numerik` la méthode `racineDicho(…)` avec la signature ci-dessus. Dans le `main(…)` de `Numerik`, calculer des solutions approchées de $x^3-5x^2+1=0$ et de $\ln(x^2+\sin(x))=3$ dans `[1,10]`, et faites afficher la plus grande. Vous utiliserez bien entendu deux instanciations de classes anonymes.

B. Les « classes enveloppantes » des types primitifs

Vous savez que `int`, `double`, `boolean`, etc. ne sont pas des classes mais des types primitifs. L'entier 12 n'est pas un objet. Dans certaines situations cependant, il est bon de considérer que 12 est un objet. Supposons que l'on veuille écrire un algorithme de tri de tableau. Va-t-on l'écrire pour les tableaux d'`int`, les tableaux de `double`, les tableaux de `String`, les tableaux de `Point`, etc ? Non bien sûr, ce serait du gâchis intellectuel, on va traiter une fois pour toutes des tableaux d'objets [`Object`] soumis à une relation de comparaison. Nous traiterons du problème de la comparaison « abstraite » dans le paragraphe suivant D. Concentrons-nous sur une classe enveloppante [*wrapper class*] comme `Integer` par exemple, qui enveloppe le type primitif `int`.

Exercice 6.2 a) Cherchez dans l'API la classe `Integer`. Dans quel package réside-t-elle ? Est-il besoin de l'importer ? Quel est son constructeur ? Comment construisez-vous l'objet `x` de type `Integer` représentant l'entier 12 ? Inversement, comment récupérez-vous la valeur de `x` en tant que `int` ? Si `x` et `y` sont deux objets de type `Integer`, comment feriez-vous afficher le plus petit des deux [faites la comparaison directement sur les objets `x` et `y`, pas sur les `int` associés]. Ecrivez une classe avec un `main(…)` qui teste tout cela.
b) Dans cette même classe, construisez un tableau de 10 objets de type `Integer`, aléatoires dans [100,200]. Programmez l'algorithme qui fasse afficher le minimum de ce tableau.
N.B. Vous comprenez maintenant l'écriture `Integer.MAX_VALUE` que vous avez vue en TP au 1er semestre pour le plus grand entier dans le type `int` ?

C. Les « interfaces » en Java

Un dernier concept, celui d'interface. Une *interface*, c'est un peu comme une classe mais dont les méthodes restent « abstraites », l'interface ne fournit que leurs en-têtes. Considérons par exemple l'interface suivante, qui est fournie avec l'API de Java 2 mais qu'il serait trivial de programmer puisqu'elle tient en deux lignes :

```
public interface Comparable
{  public int compareTo (Object x);          // on ne donne que l'en-tête…
}
```

Une interface est un type, comme une classe. Mais un objet ne pourra être de type `Comparable` que s'il appartient à une classe qui implémente cette interface.

Définition : **une classe « implémente » une interface si elle définit toutes les méthodes de l'interface.** Par exemple la classe `Integer`, telle qu'on la voit dans l'API de Java 2, est [presque] déclarée ainsi :

```
public class Integer implements Comparable
```

Puisque `Integer` implémente l'interface `Comparable`, elle remplit son *contrat* : définir concrètement la méthode d'instance `compareTo(Object x)`. Bien entendu, elle vérifiera si `x` est bien une instance de la classe `Integer` [voire de `Double`] avant de faire la comparaison.

Exercice 6.3 Supposons que `Integer` n'existe pas. Ecrivez-en une version réduite `MyInteger` implémentant l'interface `Comparable`, avec un constructeur `MyInteger(int n)` et munie des méthodes d'instance `intValue()` et `toString()`.

Exercice 6.4 Vérifiez que la classe `String` implémente l'interface `Comparable`. Comment comparer deux chaînes ?

Application : Recherche du minimum d'un tableau d'objets comparables

Exercice 6.5 Dans `Numerik`, programmez une méthode générale de recherche de minimum:

```
static Comparable minTableau (Comparable[] tab)
```

prenant un tableau d'objets *comparables* [i.e. appartenant à une classe implémentant l'interface `Comparable`], et retournant le plus petit élément de ce tableau [au sens de la relation `compareTo(…)`]. Testez votre méthode sur un tableau d'objets de type `Integer`, puis de type `MyInteger`, puis de type `String`.