Mécanismes d'abstraction

Méthode et classe abstraite

Mot-clé **abstract**

- Une méthode déclarée **abstract** ne possède pas de corps:

```
abstract void methodeAbstraite();
```

- Une classe comportant une méthode abstraite doit être déclarée comme abstraite:

```
abstract class ClasseAbstraite { ... }
```

- Une classe abstraite ne peut pas être instanciée.

```
ClasseAbstraite unObjet; //compile
unObjet = new ClasseAbstraite(); //ne compile pas
```

A quoi ça sert ?

L'abstraction sert à regrouper sous une même étiquette une catégorie d'objets, éventuellement de classes différentes, qui ont un point commun. Si seul ce point commun compte dans un programme, il est raisonnable que tous les objets de cette catégorie puissent être utilisés quelle que soit leur classe.

Par exemple, on peut raisonnablement dessiner une moto, une voiture, un train, un avion quand on nous le demande. En revanche, si on nous demande de dessiner un véhicule, qui est une catégorie plus abstraite, on voudra plus de précisions. Cependant, on sait que l'action de se déplacer est commune à tous les véhicules.

Ex.1. Integration/Point de variation (5 min)

- Téléchargez le fichier **`Integration.java`** qui permet d'intégrer numériquement la fonction $g : \mathbb{R} \to \mathbb{R}$ telle que $g(x) = x^2$.

```
    System.out.println("sur ["+a+","+b+"] par pas de "+delta);
    double sum = 0;
    for (double x = a; x <= b; x += delta)
      sum += x*x * delta;
    return sum;
```

- Encapsulez l'évaluation de la fonction dans la méthode `evaluer` d'une classe `FonctionCarre` que vous instanciez au début du calcul.

```
    sum += g.evaluer(x) * delta;
```

Ex.2. Integration/Parametre (10 min)

- Dans le code client, on veut maintenant approcher l'intégration de n'importe quel trinôme $t : \mathbb{R} \to \mathbb{R}$ tel que $t(x) = ax^2 + bx + c$.
- Ecrivez une classe `Trinome` qui modélise la fonction `t`. Ajoutez un paramètre formel de type `Trinome` à la méthode de calcul et modifiez l'appel de la méthode en conséquence.

Ex.3. Integration/Abstraction (10 min)

- Dans le code client, on veut maintenant approcher l'intégration de n'importe quelle fonction de $\mathbb{R}$ dans $\mathbb{R}$.
- Créez une classe abstraite `FonctionDeRDansR` possédant une méthode abstraite `evaluer` qui donne $f(x)$ à partir de x.
- Modifiez le type du paramètre formel de `Trinome` à `FonctionDeRDansR` dans la méthode `calculer`.
- Créez la classe `Sinusoide` qui modélise la fonction $s : \mathbb{R} \to \mathbb{R}$ telle que $s(x) = Acos(x) + Bsin(x)$. Testez avec une instance de la classe `Sinusoide`.

Interfaces

Interface

Une interface est un ensemble de requêtes. Toutes les instances des classes **implémentant** une même interface répondent (à leur manière) à toutes ces requêtes et sont donc de ce point de vue interchangeables.

Il existe des appareils très différents fournissant un signal audio/vidéo (lecteur Blu-ray, ordinateur, console de jeu). Vous pouvez pourtant tous les relier à votre téléviseur par un connecteur approprié (HDMI) pourvu qu'ils respectent tous la même interface (norme et prise).

Syntaxe

Une interface `I` liste toutes les requêtes qu'on peut adresser aux instances des classes l'implémentant:

```
interface I {
```

```
  void unePremiereRequete();
  ...
}
```

Une classe implémentant `I` est déclarée ainsi:

```
class A implements I { ... }
```

NB. Une classe peut dériver d'une autre et implémenter plusieurs interfaces:

```
class B extends A implements J, K { ... }
```

Polymorphisme

Interfaces et classes (abstraites) partagent le mécanisme de polymorphisme; des objets de classes différentes sont interchangeables à partir du moment où leurs classes héritent d'une même classe parente ou implémentent la même interface.

```
abstract class A { ... }
class B extends A { ... }
interface I { ... }
class C implements I { ... }
```

```
A objetA = new B(); //transtypage ascendant implicite
I objetI = new C(); //idem
```

Classe abstraite vs interface

- Une classe *purement* abstraite, sans champs et dont toutes les méthodes sont abstraites, ressemble à une interface.
- La différence est subtile:
    - on préferera une interface pour exiger d'une classe, qu'elle possède des capacités, pouvant être transversales à de nombreuses classes différentes. La notion de *capacité* se retrouve dans le fait qu'une classe peut implémenter plusieurs interfaces.
    - on préferera une classe abstraite pour modéliser le dénominateur commun à plusieurs classes de même *nature*. Une classe ne peut hériter que d'une seule autre classe.

Exemple d'application

On veut coder des algorithmes opérant sur des graphes. Mais les graphes peuvent être représentés par différentes structures de données (matrice d'incidence, d'adjacence, collections

de noeuds et d'arêtes intereliés par des références).

On va séparer `Graph` (la classe offrant l'accès aux algorithmes) et `GraphStruct` (l'interface implémentée par les différentes structures de données de graphe); une instance de `Graph` manipulera une instance de `GraphStruct`.

On a déjà implémenté un calcul du nombre de composantes connexes comme preuve de concept dans **`Graph.java`**.

Ex.4. Interface (5 min)

- Créez une classe appelée `Node`. Il est raisonnable de choisir qu'un noeud possède au moins un numéro qui l'identifie dans le graphe.
- Créez une interface appelée `GraphStruct` possédant deux méthodes:
  - `Node[] getNodes()` (renvoie l'ensemble des noeuds),
  - `Node[] getNeighbors(Node aNode)` (renvoie les voisins d'un noeud donné).
- Compilez les classes `Graph`, `GraphStruct`, `Node` pour s'assurer que les noms concordent.

Ex.5. Structure de données (15 min)

- Ecrivez une classe appelée `GraphStructByAdjMat`, qui étend **`SquareMatrix`** et qui implémente l'interface `GraphStruct`.

Ex.6. Test (5 min)

- Ecrivez du code client pour tester le calcul du nombre de composantes connexes:
  - sur trois noeuds isolés (3 composantes)
  - sur trois noeuds dont deux sont reliés par une arête (2 composantes),
  - etc.

Ce qu'il faut retenir

- Une interface définit ce que sait faire les classes qui l'implémentent. Une classe peut implémenter plusieurs interfaces.
- Une classe mère définit un dénominateur commun qu'enrichissent ses classes filles. Une classe ne peut dériver que d'une seule classe. Quand la classe mère est abstraite, elle n'est pas instanciable.
- Dans les deux cas, il y a polymorphisme: les objets de même type sont interchangeables.
- Pour écrire du code générique et réutilisable, mieux vaut programmer pour une interface, plutôt que pour des objets particuliers.

Héritage de classe ?

Finalement, pour savoir si l'héritage entre deux classes est approprié, l'important est de se demander si on veut exploiter

1. la propriété d'extension de code.
2. la propriété de polymorphisme,

Si on veut exploiter ces deux propriétés en même temps, l'héritage convient (mais c'est plutôt rare).

Dans le cas 1., une relation de composition pourrait être préférée.

Dans le cas 2., mieux vaut considérer l'utilisation d'une classe abstraite ou d'une interface.