Programmation et Conception Objet
Semestre 3
TP 1

Samuel Delepoulle – Franck Vandewiele
Jérémy Dagbert – Simon Clerbout

année 2021 – 2022

Héritage

Rappel

Une classe est caractérisée par
— un ensemble *d'attributs* qui décrivent dans quel état se trouve un objet ;
— un ensemble de *méthodes* qui permettent d'agir sur l'état d'un objet.

Lorsque l'on produit une application, il n'est pas rare que certaines classes partagent des fonctionnalités communes. Le mécanisme d'*héritage* permet alors de centraliser du code partagé dans une *classe mère* afin de le partager avec les classes qui en héritent : ses *classes filles* ou *classes dérivées* ou encore *sous-classes*. Les classes filles disposent de tous les attributs et méthodes de leur classe mère.

Le mécanisme d'héritage permet également aux classes filles d'être dotées de caractéristiques que ne possédait pas leur classe mère via :
— de nouveaux attributs ou de nouvelles méthodes
— une redéfinition de certaines des méthodes de la classe mère afin de leur donner un comportement différent.

Abstraction

L'héritage permet de modéliser des concepts précis (classes filles, spécialisées) en les dérivant de concepts plus larges (classes mères, plus générales). En utilisant l'héritage, on peut construire des architectures logicielles en commençant par en dessiner les grandes lignes (définition de classes mères générales, utilisées comme jalons) avant de s'intéresser aux détails (classes filles plus concrètes.)

En poussant cette logique à l'extrême, on peut ressentir le besoin d'avoir des classes très générales, au point qu'il devient difficile de spécifier le code de certaines de leurs

méthodes, parce que le code à exécuter dépendra toujours de la classe fille : on sait que ces méthodes seront utilisées par chacune des sous-classes, mais chacune aura besoin d'une implémentation spécifique.

Par exemple, un véhicule peut disposer d'une méthode `reparer()`, mais la façon dont un véhicule sera réparé dépend du type de véhicule (voiture électrique, camion, scooter, tricycle, ...).

Une solution simple consiste à mettre un corps vide à la méthode de la classe générale ; malheureusement, si l'utilisateur d'une des sous-classes oublie de redéfinir le comportement de cette méthode, des erreurs très difficiles à localiser peuvent survenir... Cela nécessite donc un effort de vigilance important du développeur des sous-classes, qui devra toujours s'assurer que les méthodes héritées dont les méthodes étaient vides reçoivent une implémentation adaptée.

Pour résoudre ce problème, Java offre la possibilité d'écrire des classes **abstraites**, dans lesquelles l'utilisateur définit uniquement la signature des méthodes, la définition de leur corps devant impérativement être faite dans chaque sous-classe. Ces méthodes sont alors elles-mêmes *abstraites.* La définition d'attributs et de méthodes ayant un corps (« méthodes concrètes ») est cependant toujours possible. La syntaxe de la définition d'une classe abstraite est la suivante :

```
abstract class NomClasseAbstraite {
  /* atttributs */

  /* methodes abstraites */
  abstract ClasseDeRetour nomMethodeAbstraite1(...);
  ...
  abstract ClasseDeRetour nomMethodeAbstraiteN(...);

  /* methodes explicites */
  ClasseDeRetour nomMethode1(...){...}
  ...
  ClasseDeRetour nomMethodeN(...){...}
}
```

Les interfaces

Il est également possible de déclarer des interfaces. Celles-ci ne comportent que des signatures de méthodes et des constantes.

La syntaxe pour déclarer une interface est la suivante :

```
interface NomInterface {
  /* attributs constants
     toujours public static final, mention qui peut être omise */
  public static final int CONSTANTE = 42;
```

```
  /* methodes
     implicitement abstract
     toujours public, mention qui peut être omise */
  public ClasseDeRetour nomMethodeAbstraite1(...);
  ...
  public ClasseDeRetour nomMethodeAbstraiteN(...);

}
```

Les interfaces sont utiles pour définir des comportements généraux que doivent avoir tous les objets dans une hiérarchie de classe.

Quelques remarques :

- une interface peut hériter d'une autre interface (mot clé `extends`) et même de plusieurs autres (héritage multiple).
- une classe peut **implémenter** une ou plusieurs interfaces, on utilise le mot clé `implements` qui suit la déclaration de la classe.
- Le principe du polymorphisme s'applique aux interfaces :
  - toute instance d'une classe est également une instance de sa ou ses super-classes ;
  - toute instance d'une classe est également une instance de sa ou ses interfaces ;
  - toute instance d'une classe est également une instance des interfaces implémentées par ses super-classes.

Exercice 1

Dans un fichier appelé `VolumeCalculable.java`, écrire une interface `VolumeCalculable` définie comme suit :

```
public interface VolumeCalculable {

        /** Calcul du volume */
        public double volume();

        /** Calcul du poids */
        public double poids();

        /** Calcul de la surface */
        public double surface();
}
```

Sauvegarder le fichier et le compiler. Vérifiez qu'un code binaire a bien été produit (`VolumeCalculable.class`)

Écrire ensuite une classe abstraite `Volume`, qui regroupera les caractéristiques générales d'un volume de matière cette classe implémentera l'interface `VolumeCalculable`. Parmi ces caractéristiques, vous définirez :

- les attributs privés `matiere` et `densite`, qui correspondront respectivement au nom de la matière dont est constitué le volume et à la densité de cette matière ;
- un constructeur prenant en paramètre une valeur pour chacun des deux attributs précédents ;
- un constructeur par recopie ;
- les méthodes abstraites `volume()` et `surface()` n'ont pas besoin d'être redéfinies pour l'instant (elle le seront des les sous-classes) ;
- la méthode concrète `poids()`, qui calcule et retourne le poids du volume ;
- une méthode `toString()` qui permet de retourner une chaîne de caractères contenant la phrase `"en ..."`, dans laquelle les `...` seront remplacés par le nom de la matière dont est constitué le volume.

Exercice 2

Écrire une classe `Sphere` qui hérite de `Volume` et dont les caractéristiques propres seront :

- un attribut privé `rayon`, représentant le rayon de la sphère ;
- un constructeur prenant comme paramètres une valeur de rayon, un nom de matière et une valeur de densité ;
- un constructeur par recopie ;
- une implémentation propre aux sphères des méthodes `volume()` et `surface()`.

On rappelle que le volume d'une sphère est donné par la formule $\frac{4}{3}\pi R^3$, dans laquelle R représente le rayon de la sphère et que la surface d'une sphère est donnée par la formule $4\pi R^2$.

Vous testerez les fonctionnalités de cette classe au travers d'une classe nommée `EssaiVolume`.

Exercice 3

Ajouter une méthode `toString()` à la classe `Sphere` ; cette méthode retournera une chaîne de caractères contenant la phrase `"sphere de rayon X en ..."`, dans laquelle `X` sera remplacé par la valeur du rayon et les `...` par la matière dont est constituée la sphère.

Testez cette nouvelle méthode dans la classe `EssaiVolume`.

Exercice 4

Écrire une classe `Parallelepipede` qui hérite de `Volume` et dont les caractéristiques propres seront :

— des attributs privés `longueur`, `largeur` et `hauteur` représentant respectivement la longueur, la largeur et la hauteur du parallélépipède ;
— un constructeur prenant comme paramètres une valeur de longueur, de largeur et de hauteur, un nom de matière et une valeur de densité ;
— un constructeur par recopie ;
— une implémentation propre aux parallélépipèdes des méthodes `volume()` et `surface()` ;
— une méthode `toString()` retournant une chaîne de caractères contenant la phrase `"parallelepipede de dimensions Lxlxh en ..."`, dans laquelle `L`, `l` et `h` seront remplacés respectivement par la longueur, la largeur et la hauteur du parallélépipède et les `...` par la matière dont est constituée le parallélépipède.

Tester les fonctionnalités de cette nouvelle classe au travers de la classe `EssaiVolume`.

Exercice 5

Modifier la classe `EssaiVolume` de manière à ce qu'elle crée une liste (`ArrayList`) de 10 volumes et remplisse chacune des cases **aléatoirement** avec une sphère ou un parallélépipède.

Lorsque la liste sera remplie, vous effectuerez l'affichage de son contenu (dans une boucle à part).

Exercice 6

Écrire une classe `Cylindre` qui hérite de `Volume` et dont le caractéristiques propres seront :

— des attributs privés `hauteur` et `diametre` représentant respectivement la hauteur et le diamètre du cylindre ;
— un constructeur prenant comme paramètres une valeur de hauteur et de diamètre, un nom de matière et une valeur de densité ;
— un constructeur par recopie ;
— une implémentation propre aux cylindres des méthodes `volume()` et `surface()` ;
— une méthode `toString()` retournant une chaîne de caractères contenant la phrase `"cylindre de dimensions hxd en ..."`, dans laquelle `h` et `d` seront remplacés respectivement par la hauteur et le diamètre du cylindre et les `...` par la matière dont est constituée le cylindre.

On rappelle que :

— circonférence d'un cercle $= 2\pi R$, où R est le rayon du cercle ;
— surface d'un cercle $= \pi R^2$;
— volume d'un cylindre = surface du cercle $\times$ hauteur du cylindre.

Quelles sont le modifications à apporter à la classe `EssaiVolume` pour y incorporer les cylindres ?

Exercice 7

Écrire une classe `CylindreCreux` qui hérite de `Cylindre` et dont les caractéristiques propres seront :

- un attribut privé `diametreInterne` représentant le diamètre du creux interne au cylindre ;
- un constructeur prenant comme paramètres une valeur de hauteur, de diamètre et de diamètre interne, un nom de matière et une valeur de densité ;
- un constructeur par recopie ;
- une implémentation propre aux cylindres creux des méthodes `volume()` et `surface()` ;
- une méthode `toString()` retournant une chaîne de caractères contenant la phrase `"cylindre de dimensions hxd en ... avec un creux central de diametre D"`, dans laquelle `h` et `d` seront remplacés respectivement par la hauteur et le diamètre du cylindre, les `...` par la matière dont est constituée le cylindre et `D` par le diamètre interne.

Y a-t-il des modifications à apporter à la classe `Cylindre` pour y incorporer les cylindres creux ?

Quelles sont le modifications à apporter à la classe `EssaiVolume` pour y intégrer les cylindres creux ?

Exercice 8

Tester le fonctionnement de la méthode `equals` sur deux instances différentes de sphères.

```
Sphere s1 = new Sphere("verre", 2.0 , 5.0);
Sphere s2 = new Sphere("verre", 2.0 , 5.0);

System.out.println("les sphères sont égales : " + s1.equals(s2));
```

Que pensez-vous du résultat ? Corrigez le problème pour la classe `Sphere` puis pour toutes les autres classes.