A3P/IPO 2022/2023

Cours 5

© Denis BUREAU, ESIEE Paris

Sommaire

1. Le polymorphisme
2. **Classe abstraite** / constructeurs
3. **Méthode abstraite** / appels
4. **Interface** *(et apparté spécial Java 8)*
5. **Interfaces du JDK**
6. *Création et héritage d'interface*
7. La classe **`Double`**
8. **`instanceof`**
9. Paquetages
10. Attributs statiques / constantes

1. Le polymorphisme

- Vocabulaire : C'est le fait qu'un même appel de méthode (par exemple, `vF.perimetre()`) n'exécute pas toujours les mêmes instructions, en fonction du **type constaté** de `vF`

- Il permet notamment de se passer de multiples tests sur le véritable type de `vF`.

- Le polymorphisme implique donc la **redéfinition** (donc l'**héritage**), et est très souvent utilisé avec les classes/méthodes **abstraites**.

2.1. Classe abstraite

- non instanciable
- ajouter `abstract` juste après `public`
- classe incomplète
  car contient **en général** au moins une méthode non implémentée (abstraite)

2.2. Constructeurs ?

- Classe abstraite `A`
  => non instanciable
  => constructeurs inutiles ?
- Non, car il faut bien initialiser les attributs (qui doivent rester privés).

- Chaque constructeur de sous-classe appellera un constructeur de `A` par `super(…);` *(en première instruction !)*

2.3. Constructeurs

- ```
  public abstract class A {
     private int attribut1;
     public A( ♥ int p0 ) {
        this.attribut1 = p0;   }
  } // A
  ```
- ```
  public class SC extends A {
     private int attribut2;
     public SC( ♥ int p1, ♥ int p2 ) {
        super( p1 );
        this.attribut2 = p2;
     }
  } // SC
  ```

♥ ≡ final

3.1. Méthode abstraite

- méthode abstraite = non implémentée = signature sans corps => **non exécutable**
- ajouter `abstract` juste après `public` et remplacer le corps `{...}` par un simple `;`
- est destinée à être redéfinie dans les sous-classes :
  si une sous-classe ne redéfinit pas toutes les méthodes abstraites de ses super-classes *(pourquoi ce pluriel ?)*, cette sous-classe devra elle-même être abstraite.

3.2. Appels de méthodes

- Dans une classe abstraite `A`, une méthode `mi` peut très bien appeler une méthode abstraite `ma` :
  `public void mi() { … this.ma(); … }`
- `mi` sera appelée sur un objet d'une sous-classe, et appellera sur cet objet courant la version de `ma` définie dans cette sous-classe
  (toujours le type constaté à l'exécution)
- La classe `A` ne pourrait pas se compiler si on ne déclarait pas la méthode `ma`
  (déclarer ne veut pas dire implémenter).

4.1. Interface

- sorte de « classe » totalement abstraite,
  sans attribut ni constructeur,
  et uniquement des méthodes abstraites.
- aucun attribut **d'instance**,
  aucune méthode implémentée
  => il n'y a rien à « hériter » => pas `extends`
- on peut par contre s'engager à « respecter »
  une interface *(une sorte de contrat)*
  => on s'engage à implémenter toutes ses méthodes
  (sinon la classe devra être abstraite)
  => on ajoute `implements UneInterface`
  à la fin de la signature de la classe

4.2. *Interface en java 8*

- Les nouveautés java 8 ne sont pas au programme de cette unité.
- Une interface java 8 peut comporter des `static` methods.
- Une interface java 8 peut comporter des `default` methods.
- Pour cette unité, les interfaces ne peuvent comporter que des `abstract` methods.

4.3. fromage ET desserts

- Implémenter une interface n'est pas incompatible avec hériter d'une classe
- On peut s'engager à implémenter plusieurs interfaces *(séparées par des virgules)*
- Exemple :

```
public class C
           extends S
           implements I1, I2, I3
 { … }
```

5. Interfaces du JDK

- interface `Comparable`
  pour comparer 2 objets => relation d'ordre

- interface `ActionListener`
  pour indiquer qu'un objet peut réagir au clic d'un bouton

- interfaces (et classes abstraites) pour gérer les collections d'objets
  (voir prochains TP / TD / cours)

6.1. Créer une interface

- Il est possible de créer ses propres interfaces *(pour structurer son code)* :
- Déclaration par `public interface` au lieu de `public abstract class`
- Toutes les méthodes sont automatiquement `public abstract` => 2 mots inutiles
- Tous les « attributs » *(constantes)* sont automatiquement `public static final` => 3 mots inutiles

6.2. Héritage d'interfaces

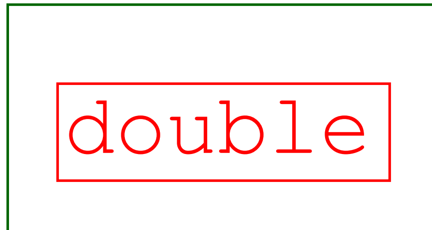
- Une interface `J` peut hériter d'une interface `I`
==> s'engager à respecter `J` oblige à implémenter
toutes les méthodes de `J` et
toutes les méthodes *(héritées)* de `I`
- Plus fort : une interface `K` peut hériter de plusieurs interfaces ==> s'engager à respecter `K` oblige à implémenter toutes les méthodes de `K` et toutes les méthodes de toutes les interfaces dont elle hérite :

```
public interface K extends I1, I2, I3
{ ... }
```

- **L'héritage multiple** existe donc bel et bien en Java *(mais pas pour les classes ...)* !

7. La classe **Double**

- Existe dans le JDK
- N'a qu'un seul attribut de type `double`
- A un constructeur naturel, et des méthodes




```
Double vD = new Double(1.2);
if (vD.compareTo(autreD)) ...
```

8. `instanceof`

- Nouveau mot java !
- *unObjet* `instanceof` *UnType* vaut `true` **aussi souvent que possible**, par exemple :
- ```
  class A extends B implements C
  class B extends D implements E,F
  interface C extends G,H
  Object a = new A();
  ```
  `a instanceof X` vaudra `true` si X vaut `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`, **ou** `Object`
- Par contre, si `Object b = new B();` `a.getClass() != b.getClass()`
  ```

9. Paquetages *(packages)*

- = répertoires contenant des classes.
- **Paquetage anonyme** : répertoire courant.
  C'est pourquoi pas besoin d' `import` pour les 5 classes du projet.
- **Paquetage `java.lang`** *(language)*.
  Contient `System`, `String`, `Math`, ...
  Donc pas besoin d' `import` non plus.
- Mais nécessaire **dans les autres cas** :
  `java.util.`**`Scanner`** `ou .`**`HashMap`**
- Mauvaise prog^on^ : `import paquetage.*;`

10.1. Attributs statiques

- = attributs de classe = variables de classe (partagés => en un seul exemplaire)
```
private static int attributStatique;
```
- une méthode statique (ou non) peut accéder aux attributs statiques
- *Pourquoi pas initialisés dans le constructeur ?*
  => bloc statique :
```
static { sAttributStatique = 12; }
```

10.2. Constantes

`static final`

- attributs statiques, publics ou privés

```
public static final
                    int NOM_EN_MAJ = 1000;
```

- ou constantes locales
  (sans `public`*/*`private` *ni* `static`*)*

```
final int NOM_EN_MAJ = 1000;
```

- une fois définies, il faut les utiliser !
  (*plus de nombres qui traînent ailleurs dans le programme*)

Dans le JDK ?

Révision

Notions de cours

TD

Vos questions ?

TP

Projet