A3P/IPO 2023/2024

Cours 4

Sommaire

1. Héritage et `super();`
2. Type déclaré / constaté
3. Redéfinition de méthode et `super.`
4. Object et 3 méthodes
5. Méthodes & attributs statiques, constantes
6. A ne pas confondre ! *(révision)*
7. **Projet :** Concepts objet

L'héritage

- relation entre classes C1 et C2
- seulement si C2 « ***est une sorte de*** » de C1
- une seule super-classe *(mère)* directe,
  mais des super-classes par transitivité,
  une ou des sous-classes *(filles)*.
- (qui ?) hérite de tous les attributs (même privés)
  et de toutes les méthodes (non privées !)
- Et les constructeurs ?

constructeurs

- Constructeur naturel dans une sous-classe : combien de paramètres ?
  => Combien d'**attributs d'instance** possède chaque objet de cette sous-classe ?
- Mais à quels attributs peut-on accéder dans une méthode de la sous-classe (pensez au `private`) ?
- D'où la nécessité de `super(...);` en première instruction *(comme pour le `this(...);` dans une même classe)*

Syntaxe

- Signature de la sous-classe suivie de
  `… extends ClasseMère`

- Aucune modification dans la classe mère.

- ***Jamais*** `extends Classe1, Classe2`
  ni `extends Classe1 extends Classe2`

Type déclaré / constaté

- ***type déclaré***
  lors de la déclaration,
  (donc uniquement utile à la compilation)
- ***type constaté***
  à l'exécution,
  (donc uniquement utile à l'exécution),
  c'est ce qui est retourné par `getClass()`

- `TypeD vRef = new TypeC();`
  ssi `class TypeC extends TypeD`

Exemple

unOiseau 5124
unAnimal 5124
vO2 5124

- *Pourquoi* `Animal vA1 = new Oiseau();`
  et pas `Oiseau vO1 = new Animal();` *?*

- ```
  Oiseau unOiseau = new Oiseau();  →@5124
  Animal unAnimal = unOiseau;
  Oiseau vO2 = unAnimal; // pb de compilation
  ```

- => `Oiseau vO2 = (Oiseau)unAnimal;`
  peut provoquer une `ClassCastException`
  sauf si `if (unAnimal.getClass()==...)`
  ou s'il existe une autre « certitude »
  ```

Redéfinition de méthode

- dans une sous-classe,
  si **exactement** la même signature
- `@Override` juste avant la signature

  pour que le compilateur le vérifie,
  dans votre intérêt !

- *Contrairement à la surcharge de méthode*
  qui a lieu dans la même classe :
  2 méthodes de même nom, mais de signatures différentes !

- `super.méthode()` pour appeler la version de `méthode` dans la super-classe

  Si pas redéfinie, `this.méthode()` suffit !

La classe `Object`

- *mère de toutes les classes* => `extends Object` automatique, pourtant un seul `extends` !?

- contient méthodes `toString`, `equals`, `getClass`, et d'autres ..., mais :
  - **`toString`** à redéfinir car sinon, retourne une adresse telle que `"@af02cc96"` *(le JDK l'utilise)*
  - **`equals`** à redéfinir car sinon, retourne la même chose que `==` *(le JDK l'utilise)*
  - **`getClass`** ne peut pas être redéfinie (*mais fonctionne très bien comme ça*), retourne le type constaté

Méthodes de classe

- Jusqu'ici, toutes les méthodes étaient des **méthodes d'instance**, car on devait les appeler sur une instance : `instance.méthode()`
- Il existe des **méthodes de classe**, qu'on peut appeler sur la classe : `Classe.méthode()`
  Exemple : `Math.sqrt(1.44)`
- Pour déclarer une telle méthode dans ses propres classes, il suffit d'ajouter `static` juste après `public` dans la signature.
- Expliquez pourquoi `this.` est interdit dans une méthode statique.

Attributs de classe

- = attributs de classe = variables de classe (partagés => en un seul exemplaire)

```
private static int sAttributStatique;
```

- une méthode statique (ou non) peut accéder aux attributs statiques

- *Pourquoi pas initialisés dans le constructeur ?*
  => bloc statique :

```
static { sAttributStatique = 12; }
```

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*)

Concepts objet *(projet)*

- **Duplication de code** : affichage des sorties
  `printWelcome`, `goRoom`, `printLocationInfo`
- **Couplage** :
  l'implémentation doit pouvoir changer
  sans forcer d'autres classes à changer
  `aNorthExit`, `aSouthExit`, …
- **Encapsulation** : pas d'accès direct
  attribut private => accesseur `getExit()`
- **Cohésion** : responsabilité précise
  par exemple, produire l'info, ne pas l'afficher
- **Extensibilité** : *par exemple,* plus de directions !

A ne pas confondre :

- Classe / objet
- Variables / méthodes
- Attributs / paramètres / variables locales
- Fonctions / procédures / constructeurs
- Définition… / appel… …de méthode
- Types primitifs / objets
- Expression / instruction
- Retour de valeur / affichage
- Affectation / comparaison

Forum !