A3P/IPO 2024/2025

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. **Projet :** Concepts objet
7. `Scanner`
8. `HashMap`
9. `Set`, `keySet()`, for each
10. A ne pas confondre ! *(révision)*

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 ?
- … `class Avion extends VTC { ~~public VTC(…) {~~`

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`*

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)*

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; // perd ses ailes ?
  Oiseau vO2 = unAnimal; // pb de compilation
  ```

- => `Oiseau vO2 = (Oiseau)unAnimal;`
  `// retrouve ses ailes ?`
  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

```
public ??? int getCompteur() {…
```

- *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*)

Scanner

- Classe du JDK => `import java.util.Scanner;`
- 1ère utilisation : lire au clavier =>

```
vSc1 = new Scanner( System.in );
String vLigne = vSc1.nextLine();
```

- 2ème utilisation : découper une chaîne =>

```
vSc2 = new Scanner( vLigne );
String vMot = vSc2.next();
```

- 3ème utilisation : extraire des nombres =>

```
int vI = vSc1ou2.nextInt();
double vD = vSc1ou2.nextDouble();
```

- 4ème utilisation : lire dans des fichiers de texte

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 !

HashMap

- HashMap ≈ « tableau associatif » qui contient un ensemble d'associations clé → valeur
- Classe du JDK => `import java.util.HashMap;`
- Plus général qu'un tableau classique qui impose le type `int` pour les indices
  (clé, valeur) => 2 types objets
- Donc pas `typeValeurs[]` mais

```
HashMap<typeClés, typeValeurs> vMaHM;
```

- `vMaHM.put( maClé, maValeur );`
- `quelleValeur = vMaHM.get( maClé );`

`HashMap` : exemple

- Plutôt que de stocker les sorties d'une pièce dans un tableau de 4 Rooms :
  tab[0] serait le nord ? tab[2] l'est ou l'ouest ?
- Associons chaque sortie Room à la direction "North" ou "South" …
- Donc dans la classe Room : un attribut
  `HashMap<String, Room> aSorties;`
- `aSorties.put( "North", vCuisine );`
- `quelleRoom = aSorties.get( "North" );`
- `null` si la clé n'est pas présente

8. `Set, keySet,` *for each*

- `keySet` est une fonction de HashMap qui retourne l'ensemble des clés :

```
??? = vMaHM.keySet();
```

- Donc pour `aSorties`, un ensemble de String, classe du JDK :

```
Set<String> vMesClés;
```

- On peut facilement parcourir tous ses éléments :

```
for ( String vS : vMesClés ) {
    S.o.p( vS );
} // appelée boucle for each
```

- Utilisable aussi sur les tableaux !

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

Question ?