Polymorphisme

Module : POO
Enseignante: BOUZAROURA Ahlam
Année universitaire: 2019-2020

Introduction

Le troisième grand principe de la POO est le concept du polymorphisme.

Considérons une lignée de classes :

C0 →C1 → Cn-1 → … →Cn

où ***Ci → Cj*** indique que la classe ***Cj*** est dérivée de la classe ***Ci.*** Cela entraîne que la classe ***Cj*** a toutes les caractéristiques de la classe ***Ci*** plus d'autres. Soient des objets ***Oi*** de type ***Ci***. Il est légal d'écrire : ***Oi=Oj*** avec ***j>i***

Donc un objet ***Oj*** de type ***Cj*** contient en lui un objet de type ***Ci.***

Polymorphisme

Le fait qu'une variable ***Oi*** de classe ***Ci*** puisse en fait référencer non seulement un objet de la classe ***Ci*** mais en fait tout objet dérivé de la classe ***Ci*** est appelé ***polymorphisme*** :

la faculté pour une variable de référencer différents types d'objets.

Le ***polymorphisme*** peut être vu comme la capacité de choisir dynamiquement la méthode qui correspond au type réel de l’objet.

Upcasting

Le « ***cast*** » (en français ***transtypage***) consiste à modifier le type d'une variable ou d'une expression. Il existe deux types de transtypages: le transtypage implicite et le transtypage explicite.

- Le transtypage implicite(Upcasting): est traité automatiquement par le compilateur lors d'une affectation ou du passage d'un paramètre effectif. Un transtypage peut être implicite si le type cible a un plus grand domaine que le type d'origine (gain de précision), il est possible d'affecter directement un int à un double, le compilateur effectue automatiquement la conversion.

DownCasting

En revanche lorsqu'on veut convertir une variable ou une expression dans un type qui lui fait perdre de la précision (domaine de valeurs plus restreint), il faut réaliser un transtypage explicite (Downcasting)(sinon erreur du compilateur).

Par exemple, convertir un float en int fait perdre les chiffres après la virgule, convertir un long en short donne des résultats aberrants pour les grands entiers (qui ne peuvent pas être représentés avec un short): dans ces cas, il faut donc réaliser un ***cast explicite ou DownCasting***.

Exemple de DownCasting

Pour réaliser un cast explicite, on met entre parenthèses le nom du type dans lequel on veut convertir suivi du nom de la variable (ou de l'expression entre parenthèses) qu'on veut transtyper.

❑ **Exmple :**

```
double d; int i;
d=i;//transtypage implicite,
i = (int) d; //transtypage d'un double
en int pour affectation
```

Casting en POO

Le ***casting*** en **POO** consiste au fait de forcer le compilateur à considérer un objet comme étant d'un type qui n'est pas le type déclaré ou réel de l'objet. En autre terme, convertir un objet d'une classe en un objet d'une autre classe si les classes ont un lien d'héritage.

En Java, les seuls ***casts*** autorisés entre classes sont les **casts** entre classe mère et classe fille.

- **Upcasting**: Le transtypage d'un objet dans le sens

fille → mère est implicite.

- **Downcasting:** le transtypage dans le sens ***mère → fille*** doit être explicite et n'est pas toujours possible.

UpCasting en POO

Il est toujours possible d'utiliser une référence de la classe mère pour désigner un objet d'une classe dérivée (fille, petite-fille et toute la descendance). Cela est logique si on se dit qu'un objet dérivé EST un objet de base.

•Pour ***caster*** un objet **o** en classe **C** : on écrit **(C) o**

- **Exemple : vehicule est la mére de vélo;**
- Velo v = new Velo();
- **Vehicule v2 = (Vehicule) v;// cast implicite**
- **Vehicule v2 = v// je peut écrire directement**

- **Exemple:** On a une classe de base ***Etudiant*** ayant comme une sous-classe la classe ***EtudSportif*** . Si on écrit:

```
Etudiant e;
EtudSportif es = new EtudSportif();
e = es; //transtypage implicite
d'un étudiant en etudiantsportif
```

e = es implique un transtypage implicite. La référence ***e*** du type ***Etudiant*** est alors utilisée pour désigner un objet de type ***EtudSportif***. L'objet désigné par ***es***, qui est du type ***EtudSportif*** est donc transtypé en ***Etudiant***.

DownCasting en POO

Le transtypage ***explicite*** des références est utilisé pour convertir le type d'une référence dans un type dérivé. C'est logique d'etre ***explicite*** si l'on se dit qu'un objet d'une classe mère n'est pas un objet de ses classes filles (un Etudiant n'est pas un EtudSportif).

si l'on voulait utiliser la référence ***Etudiant*** (qui est du type ***Etudiant*** mais qui désigne un ***EtudSportif***) pour invoquer une méthode spécifique de la classe ***EtudiantSportif(exemple méthode affiche())***, il faudrait réaliser un transtypage de ***Etudiant*** en ***EtudSportif*** (sinon, le compilateur refuserait,

Pourquoi le upcasting est automatique, mais le downcasting doit être manuel?

La réponse est simple le upcasting ne peut jamais échouer.

Mais si vous avez un groupe de différents d'Etudiants et que vous voulez les faire descendre tous à un ***EtudSportif***, alors il y a une chance, que certains de ces Etudiants sont en fait des ***informaticiens*** , et le processus échoue, en lançant ***ClassCastException***.

Définition du polymorphisme

Le terme polymorphisme décrit la caractéristique d'un élément qui peut prendre plusieurs formes, comme l'eau qui se trouve à l'état solide, liquide ou gazeux.

En programmation orienté objets, on appelle polymorphisme

- le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe.
- le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.

Polymorphisme & Héritage

« Le polymorphisme constitue la troisième caractéristique essentielle d'un langage orienté objet après l'abstraction des données (encapsulation) et l'héritage »

Bruce Eckel « Thinkingin JAVA »

Le polymorphisme, conséquence directe de l'héritage, permet à un même message, dont l'existence est prévue dans une superclasse, de s'exécuter différemment, selon que l'objet qui le reçoit est d'une sous-classe ou d'une autre.

Liaison dynamique -1-

Lorsque le type de l'objet est déterminé à l'heure compilée (par le compilateur), il est connu comme la **liaison statique**. Lorsque le type de l'objet est déterminé au moment de l'exécution, il est connu comme **liaison dynamique.**

Exemple: Soit la classe ***Animal*** mère de la classe ***Chat***:

```
Animal a =new Chat();
```

le type d'objet ***a*** ne peut pas être déterminé par le compilateur, car l'instance de Chat est également une instance de Animal. Donc, compilateur ne connaît pas son type, seulement son type de base.

```
public class Animal{
public void mange(){
System.out.println("animal mange");}
}
class Chat extends Animal{
public void mange(){
System.out.println("chat mange...");}
public static void main(String args[]){
Animal a =new Chat();
a.mange();
}
}
```

```
public static void main(String args[]){
Animal a =new Chat();
a.mange();
}
```

A l'invocation d'une méthode, le choix de l'implémentation à exécuter ne se fait pas en fonction du type déclaré de la référence à l'objet, mais en fonction du type réel de l'objet.

a désigne un objet de type **Chat**, même si c'est une référence de type **Animal** donc c'est la méthode **implémentée dans Chat qui est exécutée.**

L'affichage sera le suivant:

```
chat mange...
```

Liaison dynamique -2-

Ce mécanisme montre un aspect important du polymorphisme :

le choix du code à exécuter (pour une méthode polymorphe) ne se fait pas statiquement à la compilation mais dynamiquement à l'exécution.

Polymorphisme & héritage multi-niveau

Soit le trois classes A, B et C où A est la superclasse de B et B est la superclasse de C, on veut savoir comment le mécanisme de polymorphisme fonctionne par cet exemple.

```
• Public Class A {
  Public void faire() {
  System.out.println("niveau a"); } }
• Public class B extends A {
  Public void faire() {
  System.out.println("niveau b"); } }
• Public class C extends B {}
• class QuelleMéthode {
  public static void main(String[] argv){
A a;
a = new A(); a.faire(); // instruction 1
a = new B(); a.faire(); // instruction 2
a = new C(); a.faire(); // instruction 3
} }
```

On considère l'ensemble de classes ci-dessus. L'objectif est de savoir quelle est la sortie de la méthode ***main*** de la classe ***QuelleMéthode***.

Dans la méthode ***main*** :

- La variable a est déclarée de type A.
- Lors de l'instruction 1, la variable **a** référence un objet de type A ; de toute évidence, la méthode faire utilisée à l'exécution est la méthode **faire de la classe A.**

pour l'instruction 2. À l'exécution, on obtient niveau B.

- Si le compilateur devait décider de quelle méthode faire il s'agit, puisque l'instruction est a.faire();, il regarderait le type de la variable a, il s'agit de A, et déciderait donc qu'il s'agit de la méthode faire de la classe A ; on obtiendrai niveau A. En effet, le compilateur ne peut pas savoir quel objet sera référencé par la variable a à l'exécution (même si on le voit clairement dans notre petit programme qui est particulièrement simple) ; le compilateur ne remonte jamais dans l'historique des instructions, et serait d'ailleurs incapable de le faire dans la plupart des situations. Le compilateur se limite à vérifier que la classe A possède une méthode faire ayant des paramètres correspondant aux paramètres de l'appel ; si ce n'était pas le cas, la compilation des instructions a.faire n'aboutirait pas.

- La méthode faire à exécuter est décider à l'exécution. La machine virtuelle (par qui passe l'exécution du programme) "regarde" quel est l'objet référencé par la variable a ; comme il s'agit d'un objet de type B, c'est la méthode faire de la classe B qui est utilisée. Le fait que la méthode soit déterminée à l'exécution s'exprime par le fait qu'il s'agit d'une liaison dynamique vers la méthode faire .
- Au moment de **l'instruction 3**, l'objet référencé par a est de type C ; la classe C ne redéfinit pas la méthode faire ; la méthode utilisée est alors celle héritée de B.

Remarque Importante

Contrairement aux méthodes d'instance, les appels aux méthodes de classe sont résolus **statiquement** à la compilation.

Si une instance est utilisée pour l'invocation, la classe utilisée sera celle du type déclaré et non du type réel de l'objet.

Il n'y a pas de polymorphisme sur les méthodes de classe(les méthodes statiques)