

INFO-H-200

Programmation orientée objet

Séance d'exercices 4
Héritage, classes abstraites, interfaces et polymorphisme

Université libre de Bruxelles
École polytechnique de Bruxelles

Professeur : Hugues Bersini

2015-2016

Héritage

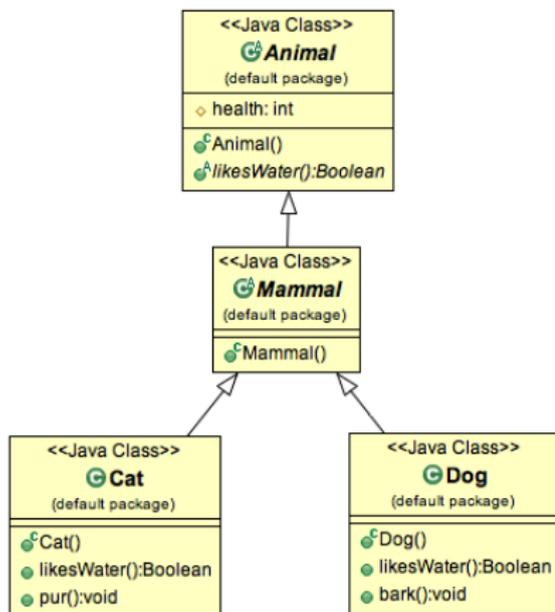
L'héritage permet la réutilisation de code. Ce mécanisme permet de développer de nouvelles classes à partir de classes existantes en étendant celles-ci.

On appelle la classe existante "mère" alors que la nouvelle, étendant la classe mère est appelée la classe "fille".

- Une classe ne peut pas hériter de plusieurs classes.
- Les méthodes que l'on veut redéfinir dans la classe fille doivent être réécrites ainsi que les méthodes et attributs complémentaires.
- **super()** : Fonctionne selon le même principe que **this** mais fait référence à la classe mère.
- Si vous redéfinissez une méthode de la classe mère dans la classe fille, celle-ci ne peut voir sa visibilité restreinte.
- Ce principe d'héritage n'est pas limité à un seul niveau, mère - fille, mais peut s'étendre à plusieurs niveaux.
- Il est possible d'empêcher l'héritage d'une classe, dans ce cas, elle doit être déclarée "final".
- On peut empêcher la redéfinition d'une méthode dans une sous classe en la spécifiant "final".
- Les méthodes statiques ne sont pas héritées et ne peuvent ainsi pas être redéfinies.

Une classe fille possède donc l'ensemble des propriétés de la classe mère ainsi que ces méthodes.

Héritage



- + : public
- - : private
- # : protected

Polymorphisme

Le polymorphisme est un concept OO dont l'idée est d'autoriser le même code à être utilisé avec différents types, ce qui permet des implémentations plus abstraites et générales.

Considérons un arbre d'héritage : une méthode dite polymorphe est une fonction qui a plusieurs formes. Sa forme dépend donc de la classe dans laquelle elle se situe !

Transtypage

En Java, un objet peut être *casté* en un autre objet (typecasting ou transtypage) :

- soit en un objet d'une sous classe (downcasting)
- soit en un objet d'une super-classe (upcasting)

Une erreur de compilation aura lieu dans le cas d'un transtypage impossible (entre objets non liés par héritage)

Une exception sera déclenchée lors de l'exécution si l'objet *casté* n'est pas compatible avec le type dans lequel il a été *casté*.

L'upcasting est automatique. Le downcasting nécessite un opérateur :

```
Animal a = new Cat(...);  
Cat c = (Cat) a;
```

Attention ! "Caster" n'implique pas de changer l'objet ! On l'identifie juste différemment. Un objet Chat upcasté en Animal restera un Chat. Néanmoins, votre chat sera désormais traité comme n'importe quel animal. Les attributs et méthodes spécifiques au Chat seront donc cachées juste qu'à ce que l'on fasse un downcasting.

Classes abstraites

Une **classe abstraite** ne peut être instanciée mais peut être sous-classée. Elle peut contenir ou non des méthodes abstraites.

Une **méthode abstraite** est une méthode déclarée mais non implémentée.

Si une classe inclut des méthodes abstraites, elle doit être abstraite.

Une classe qui hérite d'une classe abstraite doit implémenter toutes les méthodes abstraites de la classe parente. Sinon, elle doit être déclarée abstraite.

AnAbstractClass
- aField
+ doSomething()
+ doAnotherThing()

```
public abstract class Mammal extends Animal {
    public int getHealth(){
        return health;
    }
    public abstract Boolean likeswater();
}
```

Utilité ? Créer un modèle de classe ! Par exemple, la classe Animal n'a pas à être instanciée. C'est un type d'objet générique. Ce que vous désirez instancier, ce sont des chats, des chiens, etc... des animaux bien spécifiques.

Les interfaces

Une interface est un cas particulier d'une classe abstraite qui ne peut contenir que des constantes et signatures de méthodes. Il s'agit d'une définition fonctionnelle d'un type de données ou l'on ne fait apparaître que les signatures que l'on veut voir définies (i.e : Visibilité, valeur de retour, paramètres)

Il n'y a pas de niveau de protection à préciser :

- Les constantes sont implicitement 'public static' et 'final'
- Les méthodes sont implicitement 'public'

Elle ne peut être qu'implémentée par des classes ou étendue par d'autres interfaces. Elle ne peut pas être instanciée.

En Java, une classe ne peut hériter que d'une autre classe mais peut implémenter plusieurs interfaces. De fait, un objet peut avoir plusieurs types : le type de sa classe et les types des interfaces que cette classe implémente.

[Documentation](#)

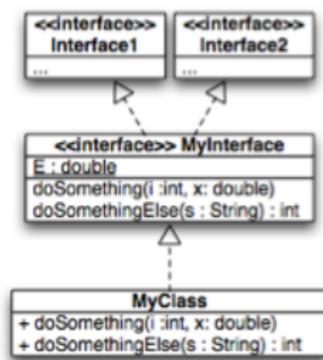
Les interfaces (suite)

```
public interface MyInterface extends Interface1, Interface2{
    double E = 2.7182;
    void doSomething(int i, double x);
    int doSomething(String s);
}

public class MyClass implements MyInterface{
    public void doSomething(int i, double x){
        ...
    }

    public int doSomething(String s){
        ...
    }
}
```

```
MyInterface m = new MyClass(); // Interface utilisée comme type
m.doSomething(4,2.1);
```



Récapitulatif

On peut empêcher un héritage à l'aide de l'instruction **final class**.

On peut empêcher l'instanciation d'une classe à l'aide de l'instruction **abstract class**

Les interfaces sont un cas particulier d'une classe abstraite qui ne peut contenir que des constantes et signatures de méthodes.

Exceptions et redéfinition

Une méthode qui redéfinit une méthode d'une super-classe peut uniquement déclencher les exceptions prévues dans la méthode de la super-classe.

Elle ne peut pas déclencher d'exception d'un type non compatible.

Construction d'un objet



```
public class A {  
    public A(){  
        ...  
    }  
}  
public class B extends A {  
    public B(){  
        ...  
    }  
}  
public class C extends B {  
    public C(){  
        ...  
    }  
}
```

Lors de la construction d'un objet de la classe C :

- le constructeur de C commence par appeler le constructeur de B
- le constructeur de B commence par appeler le constructeur de A
- le constructeur de A appelle le constructeur d'Object
- le constructeur d'Object s'exécute
- le constructeur de A se termine
- le constructeur de B se termine
- le constructeur de C se termine

Garbage collector

En Java, quand un objet n'est plus référencé, il est détruit par un ramasse-miettes ou garbage collector.

Avant la destruction d'un objet, sa méthode 'void finalize();' est appelée.

Une bonne pratique est d'utiliser *try-catch-finally* dans les *finalize* pour être certain de ne pas oublier de fermer les ressources réservées par les super-classes.

```
protected void finalize() throws Throwable {
    try{
        close(); // Close open files
    } finally{
        super.finalize();
    }
}
```

Si une exception est déclenchée dans un *finalize*, la finalisation est arrêtée mais l'exception est ignorée.

La méthode *finalize* n'est jamais appelée qu'une seule fois sur un objet.

Association, composition, agrégation

- Une **association** est une relation sémantique entre des classes qui définit un ensemble de liens.
 - Un homme est associé à son épouse
- Une **agrégation** est une association dans laquelle il y a un lien d'appartenance entre les deux objets associés (contenant/contenu, possession,...)
 - Un homme possède un chat
- Une **composition** (ou agrégation forte) est une agrégation dans laquelle la disparition du composite entraîne la disparition des composants.
 - Si un arbre meurt, ses feuilles ne servent plus à rien (on ne peut pas les mettre sur un autre arbre, au contraire de roues sur une voiture)

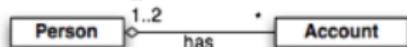
Il s'agit surtout d'une différence sémantique qui impliquera des changements dans votre implémentation au niveau du cycle de vie des objets.

Association, composition, agrégation (suite)

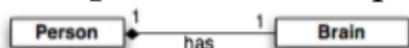
- **Association** : Une personne travaille pour une et une seule compagnie.



- **Agrégation** : Une personne possède entre 0 et n comptes en banque



- **Composition** : Une personne a un et un seul cerveau



Cardinalités :

- $*$, n , $m..*$ où $n > 0$ et $m \geq 0$
- Par défaut, la cardinalité est 1
- Dans une composition, la cardinalité du côté de l'agrégat ne peut être que 1 ou 0..1

Le nom de l'association est facultatif

Association multiples

En Java, les associations multiples peuvent par exemple s'implémenter dans des collections comme *ArrayList*, *HashMap*,...

Par exemple :

```
import java.util.*;

...

ArrayList<Animal> animals = new ArrayList<Animal>();
animals.add(new Cat());
animals.add(new Dog());
...
foreach(Animal a: animals){
    a.doSomething();
}
```

Question !

Sachant comment fonctionne le garbage collector, quelle différence voyez vous entre l'implémentation d'une agrégation et celle d'une composition.