Cours 4

I. L'héritage *(caractéristique des LOO)*

I.1 Relation :
- héritage = "est une sorte de"
- différence avec "est composé de"
(exemple de mauvais héritage : `Vehicule` = sorte de `Rectangle` avec 2 `Cercles` en plus)
(doit avoir un sens ==> technique de conception, pas de programmation)

I.2 Exemple : *moyen_de*_Transport/Avion
a) diagramme des classes
b) objets en mémoire
c) code java (`extends`)

I.3 Vocabulaire (une classe fille / sous-classe / classe dérivée, hérite / dérive / étend, d'une classe mère / super-classe)

I.4 Héritage unique entre classes *(contrairement à C++, compensé par le mécanisme d'* `Interface`*)* mais transitivité

I.5 Ce qui est hérité *(règle: la sous-classe hérite de la super-classe les attributs et les méthodes)*

I.6 Exemples d'utilisation et conversion (cast) :

```
a) Avion     a1 = new Avion();        a1.vole();     a1.roule();
b) Transport t1 = new Transport();    t1.roule();    t1.vole();
c) Transport t2 = new Avion();        t2.roule();    t2.vole();
   Avion     a2 = (Avion)t2;          a2.vole();     a2.roule();
d) Avion     a3 = (Avion)t1;
e) Résumé : a1 = t1;   a1 = (Avion)t1;   t1 = a1;
```

Légende : erreur de compilation ou erreur à l'exécution ==> exception

I.7 Résumé
- `TypeMère réfMère = (TypeMère)réfFille` ≡ `TypeMère réfMère = réfFille` car `réfFille` **est_une_sorte_de** `TypeMère`
- `TypeDérivé réfFille = (TypeDérivé)réfMère;`
- compilation: type statique/déclaré, exécution: type dynamique/constaté

I.8 Redéfinition et polymorphisme
a) Redéfinir dans la sous-classe une méthode de la super-classe
La redéfinition (sous-classe donc classe différente, même nom, paramètres identiques, type retourné compatible ==> identique ou sous-classe)
est différente de (contraire à)
la surcharge (même classe, même nom, paramètres différents, peu importe le type retourné)
b) Exemple :
Dans `Transport` : `void freine() { S.o.p( "Appuyer sur le frein" ); }`
Dans `Avion` : `void freine() { S.o.p( "Inverser les reacteurs" ); }`

```
t1.freine();   a1.freine();
t1 = a1;       t1.freine();   (liaison dynamique)
```

c) `@Override` (uniquement si **non private** et mêmes nom et paramètres)
d) Si la méthode `freine()` de la classe `Avion` a besoin d'appeler la méthode `freine()` de la classe `Transport` pour éviter la duplication de code, il faut écrire `super.freine();` et non pas simplement `freine();` équivalent à `this.freine();` qui provoquerait une récursion infinie.

I.9 Les constructeurs, `super()`
a) Situation habituelle :

```
public Transport( int pRo1, boolean pCo1 ) { this.aNbRoues=pRo1;
this.aEstComplet=pCo1; }
```

- 1ère version : `public Avion( int pRo2, boolean pCo2, int pRe2 ) { this.aNbRoues=pRo2; this.aEstComplet=pCo2; this.aNbReacteurs=pRe2; }`
- 2ème version : `public Avion( int pRo2, boolean pCo2, int pRe2 ) { super( pRo2, pCo2 ); this.aNbReacteurs=pRe2; }`

`super(...);` doit forcément être la 1ère instruction du constructeur.

b) Situation par défaut *(à ne pas utiliser)* :

```
public Transport() { }
public Avion()     { super(); }
```

c) Hors héritage :
un constructeur d'une classe peut en appeler un autre de la même classe pour éviter de la duplication de code par : `this();` ou `this(`*paramètres*`);`

(forcément 1ère instruction, comme pour `super();`*)* ⇒

Règle générale : Tout appel de constructeur ne peut être écrit que comme première instruction d'un autre constructeur.

d) Exception à la règle I.4) :

La sous-classe hérite de la super-classe les attributs et les méthodes sauf les constructeurs.
Raison évidente : le constructeur doit porter le même nom que la classe !

e) Contrainte *(pour mémoire)* :

- si ∃ dans `Transport` un constructeur avec paramètre(s), alors constructeur obligatoire dans toutes les sous-classes

- si ∃ dans `Transport` un constructeur sans paramètre, alors constructeur non obligatoire dans les sous-classes

I.10 Protection

a) problème: hérité ≠ accessible. Résolu dans les constructeurs, mais ... Exemple, dans `Avion` :

```
void vole() { if (aEstComplet) S.o.p( "peut atterrir" );
     else S.o.p( "doit bruler du carburant avant d'atterrir" ); }
```

b) 3 solutions pour les attributs :

Solution 1 : `public boolean aEstComplet;`

Inconvénient : n'importe quelle classe peut accéder/modifier cet attribut. ⇒ NON

Solution 2 : `protected boolean aEstComplet;`

Avantage : la classe et ses sous-classes peuvent y accéder.

Inconvénient : toutes les classes du même paquetage peuvent aussi y accéder. ⇒ NON

Solution 3 :
```
private boolean aEstComplet;
public boolean isComplet() { return this.aEstComplet; }
```
Avantage : sécurité maximale, accès ≠ modification.

Inconvénient : on doit définir un accesseur, et éventuellement un modificateur. ⇒ OUI

c) 2 solutions pour les méthodes :

Solution 1 : `public boolean estComplet()`

Inconvénient : n'importe quelle classe peut appeler cette méthode, ce qui est souvent souhaité ⇒ OUI

Solution 2 : `protected boolean estComplet()`

Avantage : la classe et ses sous-classes peuvent appeler cette méthode, mais aucune classe hors du paquetage. ⇒ OUI

(exception à la règle du I.4: sauf les méthodes privées)

d) 4ème niveau de protection : *(package-private)* = aucun mot java

protection	mot-clé java	la classe courante	autre classe du paquetage	sous-classe dans un autre paquetage	reste du monde
privé	`private`	Oui	Non	Non	Non
paquetage		Oui	Oui	Non	Non
protégé	`protected`	Oui	Oui	Oui	Non
public	`public`	Oui	Oui	Oui	Oui

II. La classe `Object`

II.1 mère de toutes les classes

a) héritage direct

```
class Transport extends Object
```
(par défaut)

b) héritage indirect

```
class Avion extends Transport
```

II.2 `@Override public String toString()`

a) existe dans `Object`, retourne `"@abcdef"`

b) déjà redéfinie dans toutes les classes du JDK

c) **à redéfinir** dans toute nouvelle classe *(si on veut pouvoir l'afficher)*

d) appelée automatiquement à chaque conversion en `String` et notamment dans un `System.out.println( );`

II.3 `@Override public boolean equals( Object pO )`

a) existe dans `Object`, compare les références

b) déjà redéfinie dans toutes les classes du JDK

c) **à redéfinir** dans toute nouvelle classe *(si on veut pouvoir faire des comparaisons)*

d) signature OBLIGATOIRE (pas `equals( Avion pA )` sinon ce n'est pas une redéfinition)

e) propriétés :

1) x.equals(null) retourne faux

2) x.equals(x) retourne vrai *(réflexivité)*

3) x.equals(y)==y.equals(x) *(symétrie)* ==> x et y de même classe

f) cas particulier des `String` : **Attention** à == !

II.4 `@Override public void finalize()` *peut être redéfinie*

(appelée automatiquement par le GC avant la destruction de l'objet)

a) existe dans `Object`, ne fait rien `{ }`

b) déjà redéfinie dans les classes du JDK qui le nécessitent

c) à redéfinir si nécessaire dans une nouvelle classe

II.5 `final Class getClass()` *NE peut être redéfinie* (voir aussi l'opérateur `instanceof`, est_une_instance_de ou peut_etre_converti_en). Exemples :

`t2.getClass() == a2.getClass()` ≡ `t2.getClass() == Avion.class`

`t2 instanceof Avion && t2 instanceof Transport && t2 instanceof Object == true`

`null instanceof AnyClass` retourne toujours `false`

`anyNonNullReference instanceof Object` retourne toujours `true`

II.6 Il existe plusieurs autres méthodes *(non étudiées dans cette unité)*

par exemple `clone()` ou `hashCode()`

II.7 Exemple de hiériarchie de classes :

```
Object <|-- ...
        |-- ... <|-- JComponent <|-- JTextComponent
        |-- ...                  |--- JEditorPane (editeur de texte)
                                 |--- JTextArea (zone multilignes)
                                 |--- JTextField (zone monoligne)
```

Lire le poly :

tout jusqu'à la section 2.4.1, sections 2.5, 3.1, **3.2.2, 3.6**, 4, 5.2.0, 6, 7.1, 7.2, 8.1, 8.2.1.1, 8.2.2.1, **9.1 à 9.5**, 13.3, et annexes 6 & 7