

**Université de Nice - Sophia Antipolis**  
**Faculté des Sciences**

DEUG MIAS MP1

**Programmation 2000-01**

**4. SOUS-CLASSES ET HERITAGE**

**A. Le mot-clé « this »**

```
class Point
{
    private int x;
    private int y;

    Point (int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    Point ()
    {
        this(0,0);
    }

    int getX ()
    {
        return this.x;
    }

    void setX (int x)
    {
        this.x = x;
    }
}
```

*// this fait référence à l'instance courante*

*// this fait référence au constructeur binaire*

*// une méthode accesseur*  
*// <==> return x ; [sous-entendu : celui de this]*

*// une méthode modificateur*

Vous savez déjà qu'une classe d'objets peut avoir plusieurs *constructeurs*, par exemple l'embryon de classe `Point` ci-dessus. C'est l'occasion de voir deux usages du mot réservé `this` en Java :

- Invoqué en première ligne d'un constructeur, `this(...)` invoque un autre constructeur de la même classe, muni des bons paramètres. Il peut être suivi d'autres instructions. Cet usage est le moins fréquent.
- Invoqué ailleurs dans une classe, il représente un objet : l'instance courante, ici de type `Point`. A l'intérieur d'une classe, son usage est implicite, comme dans la méthode `getX()`. Certains le mettent systématiquement par souci de lisibilité, ou en cas de doute, notamment lorsque certains paramètres sont aussi des noms de champs de l'objet, ici `x` dans `setX(int x)`. C'est l'usage le plus fréquent.

**Exercice 4.1** a) Rajoutez à la classe `Point` les méthodes d'instance `getY(...)`, `setY(...)` et `toString(...)`.

b) Ecrire la méthode d'instance :

```
void translate(int x, int y)
```

en utilisant explicitement l'objet `this`.

**B. La notion de sous-classe et l'héritage**

Concept assez intuitif, mais attention à la réalisation technique. Vous savez que Java est organisé en *classes*, qui sont des sous-classes de la classe `Object`. On trouve par exemple dans l'API :

```
Object                // tous les objets
|
java.awt.Component    // les composants graphiques
|
java.awt.Button        // les boutons à cliquer
```

Dans cet exemple, la classe `Button` du package `java.awt` est une *sous-classe directe* de la classe `Component` du même package, mais puisque `Component` est une sous-classe de la classe `Object`, il en résulte par transitivité que `Button` est une sous-classe de `Object`. Cette dernière est donc la *racine* de la hiérarchie des classes. Elle est située dans le package de base `java.lang` qui est chargé automatiquement [inutile de l'importer contrairement à `java.awt`] et qui contient aussi les classes `Math`, `String` et `System` entre autres :

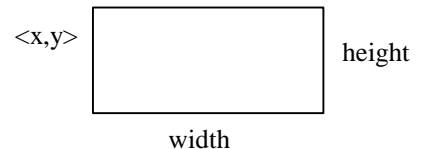
```
class A    <==>    class A extends Object
{...}          {...}
```

Plus généralement, pour exprimer en Java que **B est une sous-classe directe de la classe A**, ou que **B étend A** :

```
class B extends A
{ ... }
```

Très grossièrement, on étend une classe A lorsqu'on veut manipuler des instances de A ayant des caractéristiques particulières, en profitant donc [on dit **en héritant**] des caractéristiques générales de A. Un chat est un animal particulier, il a les attributs généraux d'un animal, il sait faire ce que savent faire les animaux en général, sauf qu'il sait en plus miauler et qu'il n'aime pas l'eau. La classe des chats étend donc la classe des animaux [ce qu'il ne faut pas transformer en : « l'ensemble des chats est plus étendu que l'ensemble des animaux », attention aux contre-sens possibles !].

*N.B. Travail inévitable en graphisme par exemple, déjà dans les applets, où vous étendiez la classe `java.applet.Applet`. Dans la classe `java.awt.Rectangle` de l'API, un rectangle est un objet défini par les coordonnées `x` et `y` de son coin supérieur gauche, par sa largeur `width` et sa hauteur `height`, donc par 4 champs [qui sont d'ailleurs publics !].*



**Exercice 4.2** Prenez 5 minutes pour butiner la classe `Rectangle` de l'API avec votre navigateur préféré.

**Exercice 4.3** a) Vous allez maintenant étendre la classe `Rectangle` en programmant une sous-classe `ColoredRect`. Une instance de `ColoredRect` sera un rectangle doté en plus d'un attribut privé `color` de type `Color`.

```
import java.awt.*;
class ColoredRect extends Rectangle
{ private Color color;
  ColoredRect(int x, int y, int width, int height, Color color)
  { super(x, y, width, height);          // super : appel au constructeur de la classe-mère
    this.color = color;
  }
  ColoredRect()
  { ... }                                // défaut : coin en <0,0>, largeur=hauteur=100, noir
  Color getColor()
  { ... }
  void setColor(Color color)
  { ... }
  public String toString()
  { return "<" + super.toString() + ", " + color + ">" ;
  }
}
```

b) Ecrivez une classe `TestColoredRect` qui définisse un rectangle vert à côtés aléatoires dans [100,200], fasse afficher sa largeur, puis change sa couleur en bleu, puis le fasse afficher [avec `System.out.println(...)`, pas graphiquement !]

*N.B. i) Le mot réservé **super** : vous aurez noté la première ligne du constructeur qui fait appel au constructeur `Rectangle(x, y, width, height)`. Si vous ne faites pas appel explicitement à un constructeur de la classe-mère, Java le fera pour vous en invoquant le constructeur sans paramètre `Rectangle()`. Attention, si vous le faites, faites-le en première ligne ! Il est conseillé de le faire explicitement.*

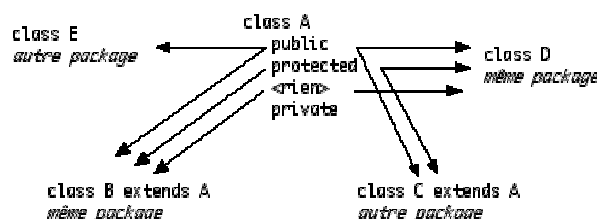
*ii) Comme **this**, le mot **super** a un second sens. Supposons que B étende A et que vous redéfinissiez dans B une méthode d'instance `foo(...)` de la classe A. Dans la classe B, `foo(...)` signifiera la méthode de B. Peut-on cependant forcer depuis l'intérieur d'une méthode d'instance de B l'invocation de la méthode `foo(...)` de A ? Oui, en demandant `super.foo(...)`. Dans ce contexte, **super** désigne la classe-mère A.*

• Pourquoi `toString()` est-elle déclarée `public` ? Parce qu'elle l'est dans `Object` et que :

*Lorsqu'on redéfinit une méthode dans une sous-classe, on ne peut pas la munir de **droits d'accès** plus restreints que la méthode définie dans la sur-classe.*

**Droits d'accès ?? Une sous-classe a-t-elle toujours accès aux champs et méthodes de sa classe-mère ?**

Question délicate. Il y a 4 niveaux de visibilité, du plus laxiste `public` au plus restreint `private`.



N.B. i) Rappelons que dans ce cours vous ne créez pas de package, donc vous pouvez lire « même répertoire » à la place de « même package »...

ii) Le schéma ci-dessus dit *en ce qui nous concerne* qu'une variable ou méthode qui n'est pas déclarée **private** dans une classe A sera visible dans toute classe du même répertoire, que ce soit une sous-classe ou pas.

## C. Un exemple complet : un compte bancaire

Où l'on résume un peu la situation, et où l'on découvre expérimentalement deux notions nouvelles...

**Exercice 4.4** a) Ecrire une classe **Compte**. Une instance modélisera un compte bancaire, muni d'une seule variable d'instance privée **solde** de type **double**. Vous prendrez deux constructeurs **Compte()** et **Compte(double solde)**. Vous aurez une méthode accesseur **getSolde()** pour consulter le solde, deux méthodes modificateurs **retrait(double montant)** permettant de retirer de l'argent sur le compte, et **déposer(double montant)** permettant de déposer de l'argent sur le compte. Et bien entendu une méthode **toString()** pour faire afficher proprement un compte bancaire. Testez votre classe dans une classe **TestCompte**, en ouvrant deux comptes distincts.

b) Rajouter une méthode d'instance **transférer(...)** permettant de transférer une somme d'un compte vers un autre.

c) [*première notion nouvelle*] Que faut-il rajouter à la classe précédente pour pouvoir dire à chaque ouverture de compte combien de comptes ont déjà été ouverts ?

*Indication : ce genre de renseignement appartient-il à un compte bancaire ou à la Banque ?*

d) Ecrire une sous-classe **CompteEpargne** de la classe **Compte**. Un compte épargne disposera en plus d'une variable **taux** de type **double** représentant le taux d'intérêt, et d'une méthode **ajouterIntérêts()** qui ajoute au solde courant la rémunération au taux d'intérêt prévu.

e) La variable **taux** est-elle propre à une instance ou la même pour toutes les instances ? Critiquez le cas échéant le choix que vous aviez fait en d).

f) Un **CompteEpargne** est en particulier un **Compte**. Vous pouvez vous en persuader avec l'opérateur **instanceof** de Java :

```
CompteEpargne c = new CompteEpargne() ;  
if (c instanceof Compte) System.out.println("c est un Compte !") ;
```

g) [*seconde notion nouvelle*] Un **CompteEpargne** étant ainsi donc un **Compte**, il est parfaitement légal d'écrire :

```
Compte c = new CompteEpargne() ;  
CompteEpargne e ;
```

Pouvez-vous en écrivant **e = c** affecter à la variable **e** la variable **c** ? Si vous avez des doutes, considérez la situation :

```
double c = 5 ;           // dans un double je peux mettre un int  
int e ;                  // mais puis-je écrire e = c ?
```

*N.B. Les deux **notions nouvelles** se nomment respectivement :*

- A. question c) : les **variables statiques** [ou variables de classe], opposées aux variables d'instance. Elles appartiennent à la classe et non à une instance de la classe. Comme les variables d'instance, elles peuvent ou non être privées. Exemple que vous connaissez bien : **Math.PI**, comment est-elle déclarée dans **Math** ?*
  - B. question g) : la **conversion de type**. J'ai un **Compte c** dont je suis sûr qu'il est aussi un **CompteEpargne**, je peux donc forcer son type, pour le voir sous un angle différent, celui d'un **CompteEpargne**. Mais il faut que ce soit raisonnable, je ne pourrais pas forcer son type en **Rectangle** par exemple !*
-