Fiche 6 : extension d'une classe

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
https://www.lri.fr/~blsk/IPO/

V1, automne 2022

Extension d'une classe Le mécanisme d'*extension* permet de définir une nouvelle classe en réutilisant le contenu d'une classe pré-existante. Ceci permet d'éviter des redondances dans le code, en nous dispensant de dupliquer le code de méthodes qui devraient sinon se trouver dans plusieurs classes.

Ainsi, si l'on doit manipuler deux classes `Point` et `PointCouleur` représentant respectivement un point dans le plan, et un point du plan doté d'une couleur, définies par les deux codes suivants,

```
class Point {
  private double x, y;
  public double norme() { return Math.sqrt(x*x + y*y); }
}

class PointCouleur {
  private double x, y;
  private String couleur;
  public double norme()      { return Math.sqrt(x*x + y*y); }
  public String getCouleur() { return this.couleur; }
}
```

on peut s'éviter la redondance de la déclaration des attributs `x` et `y` et du code de calcul de la norme en déclarant que la classe `PointCouleur` est une extension de la classe `Point`. On indique ceci à l'aide du mot-clé **`extends`** dans la ligne de déclaration de la classe. On n'écrit alors plus dans la définition de `PointCouleur` que les choses qui viennent *en plus* de ce qui est déjà présent dans `Point`.

```
class PointCouleur extends Point {
  private String couleur;
  public String getCouleur() { return this.couleur; }
}
```

Avec cette définition, on indique que toute instance de la classe `PointCouleur` possède, en plus de l'attribut `couleur` et de la méthode `getCouleur` explicitement déclarés, les champs définis par la classe `Point`. C'est-à-dire : les deux attributs `x` et `y` et la méthode `norme`. La classe `PointCouleur` est appelée une *sous-classe*, ou *classe fille*, et la classe `Point` une *super-classe*, ou *classe mère*.

On peut ajouter à la classe `Point` un constructeur de la manière habituelle, mais la situation est un peu différente pour le constructeur de `PointCouleur`.

```
public Point(double x, double y) {
  this.x = x;
  this.y = y;
}
```

```
public PointCouleur(double x, double y, String couleur) {
  this.x = x;     // incorrect
  this.y = y;     // incorrect
  this.couleur = couleur;
}
```

En effet, ces deux classes sont des classes *différentes*. Les attributs `x` et `y` de `Point`, qui sont privés, ne sont donc pas accessibles depuis la classe `PointCouleur`. À la place, on peut appeler le constructeur de la classe `Point` dans la définition du constructeur de `PointCouleur`, pour lui déléguer l'initialisation de ces deux champs. On le fait avec le mot-clé **`super`**, qui désigne la super-classe.

```
public PointCouleur(double x, double y, String couleur) {
  super(x, y);    // correct
  this.couleur = couleur;
}
```

Notez que l'on évite ainsi une redondance de plus dans le code !

Alternativement, on aurait pu donner aux attributs `x` et `y` une visibilité intermédiaire entre **`private`** et **`public`**, nommée **`protected`**, qui rend un champ accessible aux éventuelles sous-classes de la classe courante.

Sous-typage La classe `PointCouleur` possédant implicitement toutes les méthodes de la classe `Point`, on peut calculer la norme d'un objet `PointCouleur` :

```
PointCouleur pc = new PointCouleur(1., 2., "vert");
System.out.println(pc.norme());
System.out.println(pc.getCouleur());
```

En outre, toute instance de la classe `PointCouleur` peut également être vue comme une instance de la classe `Point`. Ainsi, on peut stocker un objet `PointCouleur` dans une variable variable destinée à contenir un `Point`.

```
Point p = new PointCouleur(1., 2., "vert");
System.out.println(p.norme());
```

De même, on peut donner un `PointCouleur` en argument à toute méthode qui attendrait un `Point`. Si la classe `Point` possède une méthode

```
public double distance(Point other) {
  Point v = new Point(this.x-other.x, this.y-other.y);
  return v.norme();
}
```

telle que `p1.distance(p2)` renvoie la distance entre le point `p1` et le point `p2`, on peut lui fournir en paramètre des `PointCouleur` aussi bien que des `Point`. On dit que le type `PointCouleur` est un *sous-type* du type `Point`, c'est-à-dire un *cas particulier* de `Point`.

À noter : cette notion de « cas particulier » ne fonctionne que dans un sens ! On ne peut pas prendre un `Point` quelconque et lui appliquer un traitement spécifique aux `PointCouleur`.

```
Point p = new Point(1., 2.);
String s = p.getCouleur();   // incorrect
```

Et pour cause, le point `p` de cet exemple appartient à la classe `Point`. Il ne possède donc pas d'attribut `couleur` dont on pourrait renvoyer le contenu. L'appel `p.getCouleur()` est rejeté par le compilateur, car le type `Point` déclaré pour `p` ne propose pas de méthode `getCouleur`.

Transtypage (*cast*) Les vérifications du compilateur sont faites sur la base des types déclarés pour les différents éléments. Ainsi, l'appel `p.getCouleur()` est rejeté même dans la situation suivante, où l'objet associé à `p` est effectivement un `PointCouleur`.

```
Point p = new PointCouleur(1., 2., "vert");
String s = p.getCouleur();   // rejeté à la vérification des types
```

Si l'on a de bonnes raisons de savoir qu'une variable `p` de type `Point` donnée contient plus précisément un `PointCouleur`, on peut cependant l'indiquer, en faisant figurer ce type entre parenthèses devant `p`. On parle de *transtypage*. Lors de la vérification des types, l'élément `(PointCouleur)p` sera donc considéré comme un `PointCouleur`, dont on peut bien appeler la méthode `getCouleur`.

```
String s = ((PointCouleur)p).getCouleur();   // accepté, mais peut échouer
```

Dans ce cas, Java vérifie lors de l'exécution que l'instance réelle associée à `p` est bien du type `PointCouleur`. Si ce n'était pas le cas, l'exécution du programme serait interrompue et indiquerait une `ClassCastException`.

À noter : appliqué à des classes, ce transtypage n'est qu'une *indication* donnée au compilateur, et ne peut être utilisé qu'entre des classes qui sont effectivement parentes. Il n'implique aucun autre effet à l'exécution que la vérification que l'on vient de décrire, et en particulier il ne modifie pas l'objet lui-même. Les choses sont différentes lorsque l'on applique ce même opérateur à des types de base, où on peut cette fois avoir réellement une *conversion* des données sous-jacentes au bon format.

```
int x = (int)1.41;           // changement de représentation concrète
Integer x = (Integer)1.41;   // incorrect, car Double et Integer incompatibles
```

Le mot-clé **`instanceof`** permet, à l'exécution, de tester l'appartenance d'un objet à une classe donnée, pour ne faire de transtypage que dans les cas légitimes.

```
String s;
if (p instanceof PointCouleur) {
  s = ((PointCouleur)p).getCouleur();   // accepté, et ne peut pas échouer
}
```

On a déjà utilisé ce mécanisme dans la définition de méthodes `equals(Object other)`, pour tester l'égalité entre un objet courant d'une classe que l'on est en train de définir, et un autre objet a priori quelconque.

Hiérarchie de classes Il est possible d'enchaîner les extensions, c'est-à-dire définir une classe étendant une autre classe qui aurait elle-même été obtenue par une extension.

```
class PointClignotant extends PointCouleur {
  private boolean jour;
  public void inverse() { this.jour = !this.jour; }
}
```

Cette nouvelle classe possède alors, en plus de ce qu'elle définit elle-même explicitement, tous les éléments définissant `PointCouleur`, dont certains sont eux-mêmes hérités de `Point`.

Au sommet de cette hiérarchie, on trouve en Java une classe prédéfinie `Object`, dont toutes les autres classes sont une extension (directe ou non). Ainsi, lorsque l'on écrit

```
class Point { ... }
```

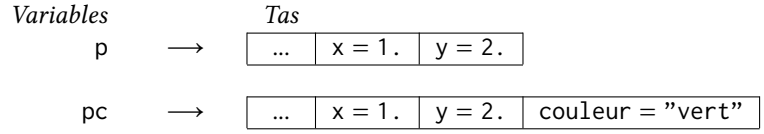
la classe `Point` est en réalité définie comme une extension de la classe `Object`.

```
class Point extends Object /* implicite */ { ... }
```

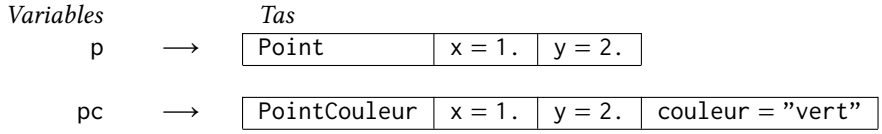
Et lorsque l'on définit `PointCouleur` ou `PointClignotant` comme des extensions (directes ou non) de `Point`, ces nouvelles classes deviennent égalements des extensions (indirectes) de `Object`.

C'est par ce mécanisme que le type `Object` désigne l'ensemble des objets que l'on peut construire en Java : toute instance d'une classe `C` quelconque, par sous-typage, est encore une instance des classes que `C` étend, et donc en particulier d'`Object`.

Modèle mémoire On a déjà évoqué que tout objet manipulé en Java est matérialisé par une structure de données en mémoire, contenant notamment les valeurs de chacun des attributs. Ces structures sont placées dans le *tas* mémoire, et on manipule ensuite concrètement chaque objet par l'intermédiaire d'un pointeur désignant l'adresse correspondante du tas.



En plus des valeurs des différents attributs, la structure de données matérialisant un objet contient une indication de la classe « réelle » de l'objet, c'est-à-dire de la classe du constructeur qui a été utilisé au moment de sa création.

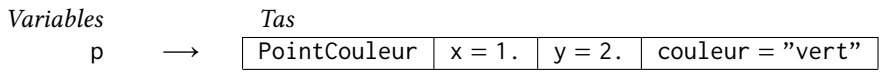


Durant la vie d'un objet, les seules choses modifiées sont les valeurs des attributs. La structure générale, et l'indication de la classe réelle, restent indéfiniment telles qu'elles ont été fixées à la création de l'objet.

Lorsque l'on utilise un objet `PointCouleur` comme un objet de type `Point`, par exemple après une déclaration

```
Point p = new PointCouleur(1., 2., "vert");
```

la représentation concrète de l'objet désigné par `p` est donc bien toujours la représentation d'un `PointCouleur`.



C'est ceci qui permet de réaliser des tests de type **`instanceof`**, ou encore de transtyper la variable `p` vers le type `PointCouleur`, lorsque la structure sous-jacente est effectivement une structure concrète de `PointCouleur`.

Redéfinition En guise de tremplin pour la suite : la classe `Object` contient elle-même des définitions de méthodes, que possèdent donc toutes les classes étendant `Object`, c'est-à-dire toutes les classes. Parmi ces méthodes, on a :

```
public String toString() { ... }
public boolean equals(Object other) { ... }
```

et on peut facilement observer l'effet de `toString` : cette méthode renvoie une chaîne de caractères indiquant la classe de l'objet ainsi que son adresse mémoire, et est utilisée par les fonctions comme `System.out.println`.

```
PointCouleur@5ca881b5
```

En revanche, lorsque l'on définit une méthode `toString` dans une nouvelle classe, cette nouvelle méthode vient prendre la place (pour cette classe seulement) de la méthode `toString` héritée de `Object`. On parle ici de *redéfinition* de méthode (*override*).