Fiche 2 : définition de classes

IPO – Introduction à la programmation objet

Florent Hivert @ Université Paris-Saclay, d'après Thibaut Balabonski
http://www.lri.fr/~hivert/COURS/IPO/

Structure générale Une *classe* définit un type d'objets qui pourront être manipulés dans un programme. La définition d'une classe comporte :

- une description des données qui constituent l'état interne d'un objet (les *attributs*),
- une ou plusieurs fonctions dédiées à la création d'un nouvel objet (les *constructeurs*),
- l'ensemble des opérations que peut réaliser un objet de cette classe (les *méthodes*).

Syntaxe : la définition d'une classe est introduite par les mots-clés **`public class`**, suivis du nom donné à la classe. La définition elle-même est ensuite placée entre accolades, et énumère les attributs, constructeurs et méthodes.

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

Attributs On déclare les *attributs* comme on déclarerait des variables, en fournissant leur type et leur nom. On place généralement avant cette déclaration le mot-clé **`private`**.

```
  // deux attributs : coordonnées du point
  private double x, y;
```

Chaque objet d'une classe possède ses propres valeurs pour chaque attribut de la classe. Dans notre exemple, toute instance de la classe `Point` possède donc une valeur pour `x` et une valeur pour `y`. On accède à l'attribut `x` de l'objet `o` avec la notation pointée `o.x`, *lorsque cet accès est effectivement autorisé.*

Méthodes Une *méthode* est une opération qui peut être réalisée sur les objets de la classe que l'on est en train de définir. La définition d'une méthode se présente comme une définition de fonction, avec un nom, un type de retour, d'éventuels paramètres, et un code entre accolades. On place généralement le mot-clé **`public`** avant cela.

```
  // calcul de la distance au point de coordonnées (0, 0)
  public double norme() {
    ...
  }
```

La méthode ci-dessus, appelée `norme`, prend zéro paramètres et renvoie un nombre de type **`double`** représentant la distance à l'origine du point auquel on l'applique. On invoque cette méthode `norme` pour un objet `o` avec la notation pointée `o.norme()`. Dans une telle invocation, l'objet `o` est appelé le *paramètre implicite*. Les éventuels autres paramètres passés entre parenthèses sont les paramètres explicites.

Dans le corps d'une méthode, on peut faire référence à ses éventuels paramètres mais également à l'objet lui-même auquel la méthode est appliquée. Cet objet est désigné par le mot-clé **`this`**. On fait référence à l'attribut `x` de l'objet exécutant la méthode avec la notation **`this`**`.x`. On peut donc compléter le corps de la méthode `norme` par la ligne

```
    return Math.sqrt(this.x * this.x + this.y * this.y);
```

Pendant l'exécution d'une méthode déclenchée par un appel `p.norme()`, l'objet **`this`** auquel on fait référence est précisément le paramètre implicite `p`.

Constructeurs Un *constructeur* est une fonction dont le nom est précisément le nom de la classe. Son code a pour objectif d'initialiser les attributs de l'objet construit, en utilisant plus ou moins directement des paramètres fournis. Le constructeur ne donne pas de type de retour (on sait qu'il sert précisément à initialiser un objet de la classe) et est généralement précédé du mot-clé **`public`**.

```
  // construction d'un point avec les coordonnées fournies
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
```

Notez, dans une instruction telle que **`this`**`.x = x;`, la différence entre le paramètre `x` du constructeur et l'attribut `x` de l'objet construit. Le deuxième est désigné par **`this`**`.x`.

Dans un fragment de code Java, on peut créer un objet à l'aide du mot-clé **new** et du constructeur de la classe choisie.

```
new Point(1.0, 2.0)
```

Dans les cas simples, une telle construction est directement associée à la déclaration d'une variable, dont le type est le nom de la classe.

```
Point p = new Point(1.0, 2.0);
```

L'objet ainsi construit est matérialisé physiquement par une petite structure en mémoire, contenant notamment les valeurs `1.0` et `2.0` associées à ses attributs `x` et `y`. La variable `p` contient alors un pointeur vers cette structure.

Affichage Tout objet `o` peut être affiché par l'instruction `System.out.println(o);`. Par défaut, l'information affichée est l'adresse de la structure correspondante en mémoire, ce qui n'est pas toujours le plus informatif :

```
Point@659e0bfd
```

Pour obtenir un affichage plus centré sur le contenu d'un objet, il faut définir dans la classe une méthode `toString` renvoyant la chaîne de caractères que l'on souhaite afficher pour un objet donné.

```
public String toString() {
  return "(" + this.x + ",␣" + this.y + ")";
}
```

Note : pour que cette méthode d'affichage soit prise en compte par `System.out.println`, il faut scrupuleusement respecter la signature de la méthode : mot-clé **public**, type de retour `String`, pas de paramètres.

Égalité L'opérateur == en Java permet de tester l'égalité de deux valeurs. Appliqué à deux objets `o1` et `o2`, cet opérateur vérifie que `o1` et `o2` sont *physiquement* identiques, c'est-à-dire que les structures sous-jacentes occupent les mêmes adresses en mémoire.

Pour comparer deux objets sur la base de leur contenu, on peut définir une méthode dédiée, qui compare l'objet courant avec un autre objet donné en paramètre.

```
public boolean egale(Point p) {
  return this.x == p.x && this.y == p.y;
}
```

On peut alors remplacer la comparaison *physique* `o1 == o2` par la comparaison *logique* `o1.egale(o2)`.

Par convention, la méthode que l'on définit pour cette comparaison a une signature légèrement différente, qui accepte en paramètre un objet de n'importe quel type (et porte le nom anglais `equals`).

```
public boolean equals(Object o) {
  if (o instanceof Point) {
    Point p = (Point)o;  // p pointe vers le même objet que o, mais a le bon type
    return this.egale(p);
  } else {
    return false;
  }
}
```

Cette version a une forme très codifiée : le paramètre est du type `Object` qui représente n'importe quel objet de n'importe quelle classe, puis un test avec le mot-clé **instanceof** sépare les cas où cet objet est effectivement de la classe voulue ou non. Si l'objet appartient à la bonne classe, on le *convertit* pour indiquer au vérificateur de types comment considérer l'objet (ici, comme un `Point`) puis on peut utiliser la méthode de comparaison précédente. Enfin, dans le cas où l'objet n'appartient pas à la bonne classe on peut de toute façon renvoyer **false**.

Encapsulation Selon les principes de la programmation objet, on interagit avec un objet en utilisant ses méthodes, et non en accédant directement à son contenu (c'est-à-dire à ses attributs). Il s'agit de séparer

1. la manière dont un objet fonctionne, et
2. la manière dont on l'utilise.

Un programme utilisant un objet donné n'a besoin de connaître que le *mode d'emploi* de cet objet. Le contenu exact et le fonctionnement interne sont des « détails d'implémentation » que, en tant qu'utilisateur, on n'a pas à connaître. Cette séparation a (au moins) trois avantages :

— cela fait moins de choses à connaître/comprendre pour l'utilisateur,

— cela permet d'apporter des modification au fonctionnement interne d'un objet (amélioration de l'efficacité, correction de bugs, adaptation à un nouvel environnement) sans que l'utilisateur ait besoin de s'adapter, tant que les modifications préservent le mode d'emploi,
— si le fonctionnement interne d'une classe repose sur une certaine cohérence entre les différents attributs d'un objet, les restrictions d'accès évitent qu'un utilisateur mal informé ne vienne invalider cette cohérence et perturber le fonctionnement de l'ensemble.

Ainsi, le mot-clé **`private`** utilisé dans les déclarations d'attributs indique que ces attributs font partie du fonctionnement interne de la classe et ne doivent pas être utilisés directement à l'extérieur de cette classe. À l'inverse, le mot-clé **`public`** du constructeur et des méthodes indique que tout le monde peut compter sur ces éléments, et les utiliser librement. Le compilateur Java vérifie que ces restrictions d'accès sont bien respectées.

En dehors de **`public`** et **`private`**, il existe deux autres niveaux d'accessibilité que nous verrons plus tard. Nous verrons également des situations dans lesquelles des attributs peuvent être publics, et des méthodes privées.

Invariants L'un des intérêts de l'encapsulation est de protéger les *invariants* d'un objet, c'est-à-dire les propriétés assurant la cohérence de son état interne. Pour une classe représentant des paires $\{x, y\}$ d'entiers, on peut par exemple proposer deux attributs pour les deux valeurs de la paire :

```
public class Paire {
  private int a, b;
}
```

Cependant, remarquez que les deux structures | Paire | a : 1 | b : 2 | et | Paire | a : 2 | b : 1 | représentent la même paire $\{1, 2\}$! On peut choisir une règle qui fixe une représentation unique pour ces deux versions, en décidant par exemple que le plus petit élément de $\{x, y\}$ sera stocké dans l'attribut `a`, et l'autre dans l'attribut `b`. On peut le forcer avec un constructeur écrit de la manière suivante.

```
  public Paire(int x, int y) {
    if (x < y) {
      this.a = x;
      this.b = y;
    } else {
      this.a = y;
      this.b = x;
    }
  }
```

Ensuite, les restrictions d'accès aux attributs `a` et `b` assure qu'aucune personne extérieure ne viendra directement manipuler ces attributs. Autrement dit, l'accès aux attributs est restreint à la personne qui développe cette classe elle-même, qui sait quelle contrainte d'ordre doit être respectée.

Un avantage d'un tel choix est que certaines méthodes vont devenir plus simples et plus efficaces : pour tester l'égalité entre deux paires, si on sait que $a < b$ pour tout objet, on peut se contenter de la méthode

```
public boolean egale(Paire p) {
  return this.a == p.a && this.b == p.b;
}
```

alors que sans cette hypothèse il aurait fallu tenir compte de plus de possibilités.

Méthodes statiques Une définition de classe Java peut également contenir des méthodes qui *n'ont pas accès* à un objet **`this`**. De telles méthodes sont qualifiées de « statiques » (on verra la raison de ce choix de vocabulaire plus tard) et correspondent au concept habituel de *fonction*.

Une telle méthode est déclarée avec le mot-clé **`static`**, et déclare explicitement tous ses paramètres.

```
public static double compareNorme(Point p1, Point p2) {
  return p1.norme() - p2.norme();
}
```

Cette méthode n'a pas accès à un objet **`this`**, et donc pas non plus accès aux attributs de la classe `Point`. Elle peut en revanche faire appel aux méthodes des objets passés en paramètres.

On appelle une méthode statique avec la notation pointée habituelle, en mettant à gauche du point le nom de la classe (plutôt qu'un objet comme dans une méthode habituelle).

```
double d = Point.compareNorme(p1, p2);
```

Note : le **`static`** apparaissant dans la déclaration de la fonction `main` est bien celui que l'on vient de décrire !