Programmation orientée objet et temps réelle avec Java

Visibilité des déclarations, encapsulation des données et portée des déclarations

Dominique Blouin
Ingénieur de recherche
Télécom Paris, Institut Polytechnique de Paris
dominique.blouin@telecom-paris.fr

Objectifs d'apprentissage

- **Visibilité des déclarations.**
- **Encapsulation des données.**
- **Portée des déclarations.**
- **Les classes `String` et `ArrayList`.**

Rappels

- Les **objets** sont des entités informatiques qui communiquent par envois de messages.
- Les objets contiennent des valeurs appelées **attributs**. Parmi les attributs, on peut trouver des **références** sur d'autres objets.
- Une référence sur un objet permet de lui envoyer un **message**.
- Pour chaque type de message que l'objet peut recevoir, l'objet connaît une **méthode** associée au type de message.
- Cette méthode est une procédure qui est **exécutée par l'objet** lorsqu'il reçoit le type de message associé.

Rappels

- Un type d'objet est décrit par une **classe**.
  - La classe décrit les attributs : nom et type de valeur.
  - La classe décrit les méthodes utilisées pour répondre aux messages.

- Le programmeur peut créer des objets à partir de la classe. C'est le processus d'**instanciation**.

- On dit que les objets sont des **instances** de la classe ou que les objets appartiennent à la classe.

Délégation et non-intrusion

- Une bonne pratique de l'OO est d'éviter d'agir depuis l'extérieur sur l'état d'un objet (**non-intrusion**).
- Ainsi, on évitera autant que possible les opérations de lecture et d'écriture des attributs.
- On va donc demander à l'objet de type **`Point`** d'afficher ses coordonnées (principe de **délégation**).
- La meilleure manière:

```
Point myPoint = new Point(10, 10);
myPoint.writeCoordinates();
```

Bonne pratique

- Exemple: afficher les coordonnées d'un point:

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

    void writeCoordinates() {
        System.out.print("x = ") ;
        System.out.println(x) ;
        System.out.print("y = ") ;
        System.out.println(y) ;
    }
}
```

- Pour accéder à un attribut ou à une méthode d'on objet, on utilise le caractère « . ».

```
Point myPoint = new Point(10, 10);
myPoint.writeCoordinates();
```

Mauvaise pratique

- Accès direct aux attributs :

```
Point myPoint = new Point(10, 10);
System.out.print("x = ");
System.out.println(myPoint.x);
System.out.print("y = ");
System.out.println(myPoint.y);
```

- Comment prévenir cette mauvaise pratique?

- Réponse: utiliser la **visibilité**.

Visibilité en Java

- Quatre types de visibilité :
  - **`public`** : Tout est accessible.
  - **`private`** : Accessible que depuis la classe.
  - **`package`** : Accessible que depuis le package.
  - **`protected`** : Accessible que depuis l'héritage (à voir plus tard…).

- Exemple:

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

- L'utilisation de la visibilité **`private`** empêche d'accéder aux attributs depuis l'extérieur de la classe.
  - Erreurs de compilation.

Point.java | Main.java

```
public class Main {

    public Main() {
    }

    public static void main(String[] args) {
        Point myPoint = new Point(10, 10);

        System.out.print("x = ");
        System.out.println(myPoint.x);
        System.out.print("y = ");
        System.out.println(myPoint.y);
    }
}
```

Organisation des classes: les packages

- Un **package** (paquetage en français) est une collection **logique** de classes.

- Dans un programme Java, les classes sont regroupées en packages.
  - Un package pour les classes réalisant des E/S (I/O).
  - Un package pour les classes réalisant des communications réseaux.
  - Un package pour les classes réalisant l'interface graphique.
  - Etc.

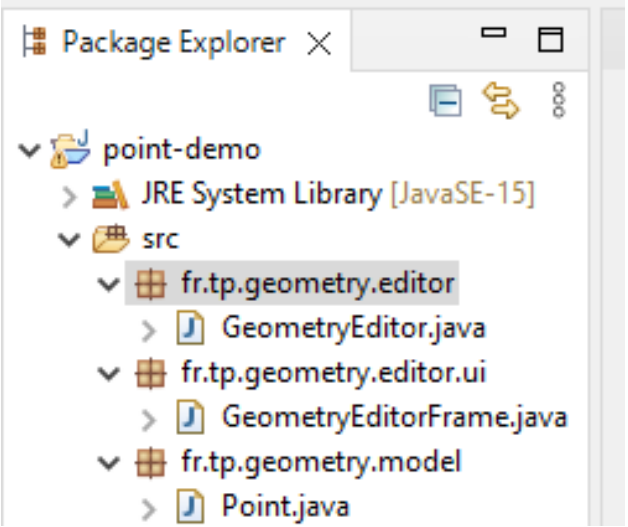
- Principe de cohésion forte et de couplage faible:
  - Classes devant être utilisées ensemble (couplage fort) regroupées dans un même package (cohésion forte).
  - Les classes ayant peu de dépendances sont regroupées dans des packages différents.

Déclaration de package

```
package fr.tp.geometry.model;

class Point {
    private int xCoord;
    private int yCoord;

    Point( int x,
           int y) {
        xCoord = x;
        yCoord = y;
    }
…
}
```



- Les packages sont **hiérarchiques**.
  - Un package peut contenir d'autres packages.
- Le fichier de la classe (*Point.java*) sera stocké dans un sous-répertoire ayant l'arborescence défini par le nom du package (les points sont remplacés par /).
  - ***../src/fr/tp/geometry/model/Point.java***
- Les noms de packages s'écrivent en **lettres minuscules**.

Espace de nom de package

- Par défaut, une classe ne pourra accéder à une autre classe déclarée dans un autre package qu'en spécifiant son nom **complet**.

```
package fr.tp.geometry.editor;

public class Main {

    public Main() {
    }

    public static void main(String[] args) {
        fr.tp.geometry.model.Point myPoint = new
fr.tp.geometry.model.Point(10, 10);

        myPoint.writeCoordinates();
    }
}
```

Import de package

- L'utilisation du nom complet rend le code difficile à lire.
- Pour solutionner ce problème, on peut utiliser le mot clé **import**:

```
package fr.tp.geometry.editor;

import fr.tp.geometry.model.Point;

public class Main {

    public Main() {
    }

    public static void main(String[] args) {
        Point myPoint = new Point(10, 10);

        myPoint.writeCoordinates();
    }
}
```

Import de toutes les classes d'un package

```
package fr.tp.geometry.editor;

import fr.tp.geometry.model.*; // « * » importe toutes
les classes du package

public class Main {

    public Main() {
    }

    public static void main(String[] args) {
        Point myPoint = new Point(10, 10);

        myPoint.writeCoordinates();
    }
}
```

Visibilité de package

- Si aucune visibilité n'est déclarée, la visibilité sera celle du **package**.

```
class Point {
    int xCoord;
    int yCoord;

    Point( int x,
      int y) {
        xCoord = x;
        yCoord = y;
    }
…
}
```

- Une classe ne pourra accéder à un élément de visibilité **package** d'une autre classe que si ces deux classes sont déclarées dans un même **package**.

- Si les classes sont dans deux packages différents, alors seulement les éléments déclarés avec une visibilité **`public`** seront accessibles.

Encapsulation des données

- Les attributs d'un objet représentent son état.

- Si les attributs d'un objet sont déclarés avec le mot-clé **`private`**, alors seul l'objet lui-même peut y accéder.

- On parle alors d'**encapsulation des données**.

```
class Point {
        private int xCoord;
        private int yCoord;
 }
```

Encapsulation des données

- L'encapsulation des données est **fondamentale** en OO.

- Si les données ne sont pas encapsulées, n'importe quel autre objet peut les modifier.
  - Cela peut conduire à des incohérences.

- L'encapsulation des données concerne:
  - La **consultation** des données.
  - La **modification** des données.

Consultation des données

- Si l'on désire connaître la valeur d'un attribut, il faut le **demander** à l'objet.

- Pour cela, on définit une méthode spécifique appelée un **getter** car le nom de la méthode commence habituellement par **get**.

- Un getter peut aussi être appelé **accesseur** ou **méthode d'accès**.

Exemples de déclaration d'accesseurs

```
class Point {

    private int xCoord;
    private int yCoord;

    public Point(int x,
                 int y) {
        xCoord = x;
        yCoord = y;
    }

    public int getxCoord() {
        return xCoord;
    }

    public int getyCoord() {
        return yCoord;
    }
    …
}
```

Exemples d'utilisation d'accesseurs

```
Point myPoint = new point(5, 7);
int xCoord = myPoint.getXCoord();
```

Modification des données

- Pour modifier un attribut d'un objet, il faut le lui demander.
- Pour cela, on définit une méthode spécifique appelée un **setter** car le nom de la méthode commence généralement par **set**.
- Un setter peut aussi être appelé **mutateur** ou **méthode de modification**.

Exemples de déclaration de mutateurs

```
class Point {

    private int xCoord;
    private int yCoord;

    public void setxCoord(int newX) {
        xCoord = newX;
    }

    public void setyCoord(int newY) {
        yCoord = newY;
    }
}
```

Pourquoi encapsuler ?

- L'encapsulation des données est un principe **très important**.
- Tous les attributs doivent être qualifiés avec le mot clé **`private`**, ce qui signifie que les attributs ne peuvent être consultés ou modifiés que depuis les **méthodes** de l'objet.
- Les raisons de ce principe sont multiples. Nous allons en voir quelques-unes.

Valider les valeurs

- Avoir des **setters** permet de **contrôler la validité** des valeurs que l'on veut attribuer aux attributs.

```
class Human {

    private int age;

    boolean setAge(int newAge) {
        if (newAge >= 0 && newAge < 150) {
            age = newAge;

            return true;
        }

        return false;
    }

    ...
}
```

Cohérence des données

- L'encapsulation permet de préserver la cohérence et l'intégrité des structures de données.

```
class Vector {

    private int dx;
    private int dy;

    private double length;

    Vector(int dx, int dy) {
        setDxDy(dx,dy);
    }

    void setDxDy(int newDx,int newDy) {
        dx = newDx;
        dy = newDy;
        length = Math.sqrt(dx*dx + dy*dy);
    }
}
```

Cohérence de l'application

- L'encapsulation de données aide à garantir la **cohérence de l'application**.

- Supposons que nos classes de figures géométriques sont en cours d'édition dans un éditeur:
  - Alors toute modification des caractéristiques d'une figure, c'est-à-dire de ses attributs, doit entraîner un rafraichissement de l'affichage dans la fenêtre d'édition.
  - Sinon, l'application sera incohérente : la vue (l'affichage) ne reflètera pas le modèle (les données).

- Ce sont les méthodes de modification des attributs qui seront en charge de réaliser un rafraîchissement de la fenêtre d'édition.

Transparence des mises en œuvre

- Une classe présente l'interface de ses objets avec le monde extérieur.
  - Les méthodes devront donc être visibles par tous les objets et seront donc qualifiées avec le mot clé **`public`**.

- Les attributs sont cachés et qualifiés avec le mot clé **`private`**.

- Si l'on encapsule correctement les données, il est possible de modifier la mise en œuvre des méthodes de manière transparente.

Exemple de transparence de mise en œuvre

- L'encapsulation permet de préserver la cohérence et l'intégrité des structures de données.

```
class Vector {

    private int dx;
    private int dy;

    private double length;

    public Vector(int dx, int dy) {
        setDxDy(dx,dy);
    }

    public void setDxDy(int newDx,int newDy) {
        dx = newDx;
        dy = newDy;
        length = Math.sqrt(dx*dx + dy*dy);
    }

    public double getLength() {
        return length;
    }
}
```

Exemple de transparence de mise en œuvre

- Supposons que l'on ait rarement besoin de connaître la longueur d'un vecteur:

```
class Vector {

    private int dx;
    private int dy;

    public double getLength() {
        return Math.sqrt(dx*dx + dy*dy);
    }

    public void setDxDy(int dx0, int dy0) {
        dx = dx0;
        dx = dy0;
    }
}
```

Encapsulation des données

- Parce que les données ont bien été **encapsulées** dans la classe **`Vector`**, on peut utiliser l'une ou l'autre implémentation de la classe de manière transparente.

- En effet, les deux classes implémentant la classe **`Vector`** ont la même **interface**, c'est-à-dire qu'elles présentent les mêmes **méthodes** à leurs utilisateurs.

- Si les données sont correctement encapsulées dans une classe, l'utilisateur de la classe n'a pas besoin de savoir **comment sont structurées les données** et **comment les méthodes sont programmées** à l'intérieur des objets de la classe.

Visibilité des classes

- On peut également définir une visibilité à une classe:

```
public class Vector {

    private int dx;
    private int dy;

    public double getLength() {
        return Math.sqrt(dx*dx + dy*dy);
    }

    public void setDxDy(int dx0, int dy0) {
        dx = dx0;
        dx = dy0;
    }
}
```

- Quelle est la signification de la visibilité d'une classe?

Portée des déclarations: Trois types de déclaration de variables

```
public class Test {
    private int myAttribute; // Declaration d'attribut

    public void myMethod(int myPar) { // Declaration de parametre
        ...
        int myLocalVar = 0; // Declaration de variable locale
        ...
    }
}
```

- Ces déclarations ont une **portée**.

- Portée: **partie** du programme où la déclaration est **visible:**
  - Les endroits dans le code où la variable peut être utilisée.

Portée des déclarations

- **Variable locale:**
  - La portée est **l'intérieur du bloc d'instructions** où la variable est déclarée.

```
{
    int myLocalVar = 0;
    …
    myLocalVar = myLocalVar + 8;
}
```

- **Attribut:**
  - La portée est la **classe entière**.
  - Un attribut est donc accessible dans **toutes les méthodes** de la classe.

- **Paramètre de méthode:**
  - La portée est la **méthode entière**.
  - La variable paramètre est **créée au début** de l'exécution de la méthode.
  - Elle **disparaît à la fin** de l'exécution de la méthode.

Retour sur le mot clé this

- Qu'arrive-t-il si un nom de paramètre est le même que celui d'un attribut?

```
public class Point {

    private int xCoord;

    private int yCoord;

    public Point(int xCoord,
                 int yCoord) {
        xCoord = xCoord;
        yCoord = yCoord;
    }
```

- Solution (pratique de codage recommandée):

```
public class Point {

    private int xCoord;
    private int yCoord;

    public Point(int xCoord, int yCoord) {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }
}
```

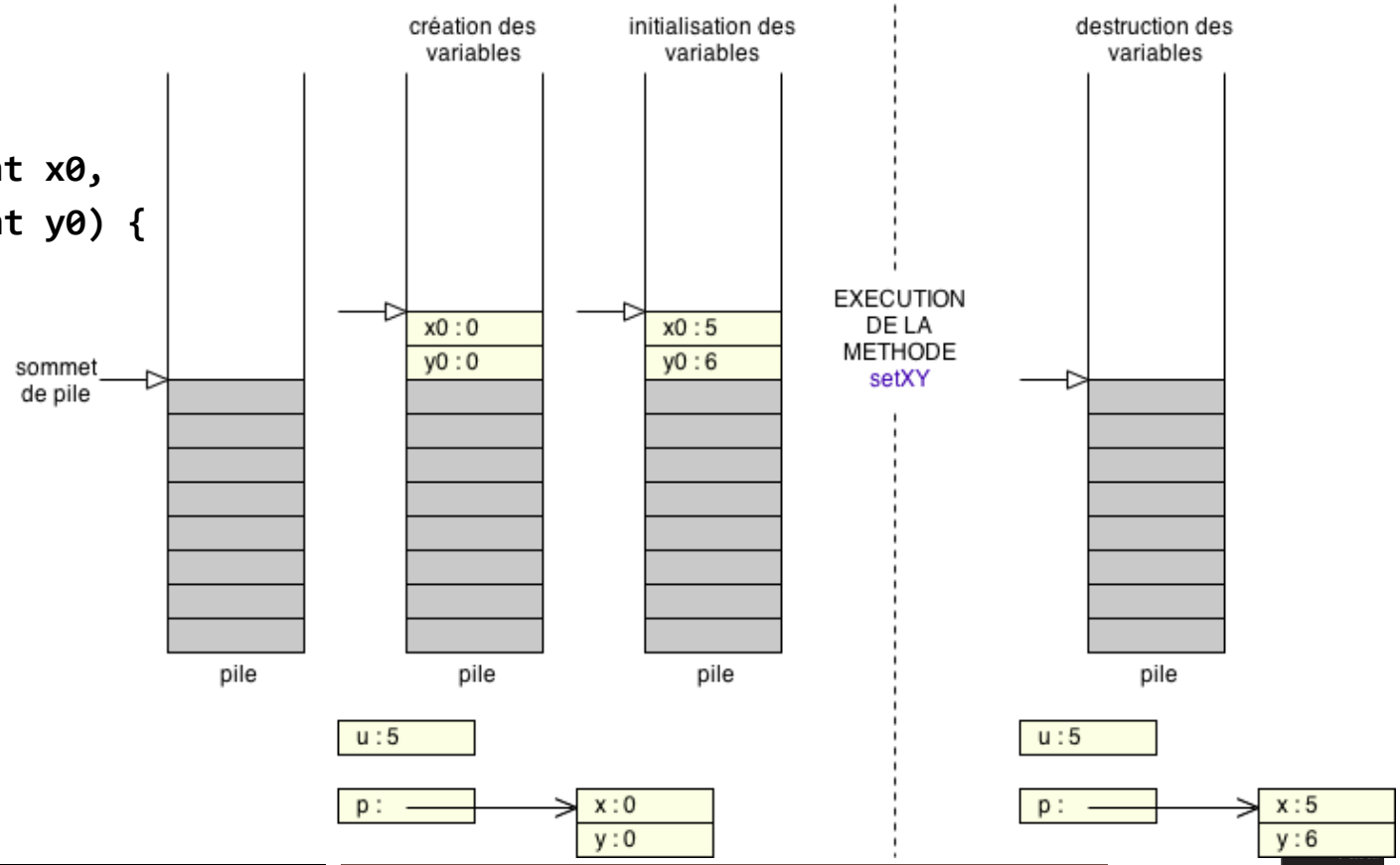
Passage de paramètres

- Java pratique le passage de paramètre **par valeur**.
  - Mis en place à l'aide d'une pile.

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

    public void setXY(int x0,
                      int y0) {
        x = x0;
        y = y0;
    }
}

Point p = new Point();
int u = 5;
p.setXY(u, u+1);
```



Les chaînes de caractères

- Chaîne de caractères: **suite de caractères**.
  - Par exemple le mot **`Java`**.

- En Java (et dans presque tous les langages), une chaîne de caractères s'écrit entre des guillemets (**"**).
  - **`"Java"`**
  - **`"Java is easy"`**

- Les chaînes de caractères sont des objets de la classe **`String`**.
  - Elle est définie dans le JDK.

- **`String s1 = null;   // Initialized with the null value`**
- **`String s2 = "Java"; // Initialized with a reference`**

- Sur le web, rechercher « **JAVA SE String** »

Concaténation de chaînes de caractères

- L'opérateur **+** peut prendre deux chaînes de caractères en paramètres.

```
String s1 = "Java";
String s2 = "coffee";
String s3 = s1 + " " + s2;
```

- L'opérateur **+** réalise la concaténation de ses arguments:

```
String s4 = s1 + s2
```

- Est équivalent à:

```
String s5 = s1.concat(s2)
```

Retour sur l'égalité

- Nous avons vu l'opérateur de test d'égalité noté **`x == y`**.

- Celui-ci ne fonctionne qu'avec les types scalaires **discret**:
  - **`char`**, **`byte`**, **`short`**, **`int`**, **`long`** et **`boolean`**.
  - Il ne fonctionne pas avec les types **`float`** et **`double`**.

- Qu'en est-il des types références?
  - L'opérateur `==` utilisé avec un type référence signifie que les deux opérandes sont des références sur **le même objet en mémoire**.

- Pour tester que deux objets sont égaux au sens large, les classes proposent une méthode nommée **`equals`**.
  - Ainsi pour tester l'égalité de deux chaînes de caractères **`s1`** et **`s2`** (elles sont composées des mêmes caractères dans le même ordre), on utilisera **`s1.equals(s2)`**.

La classe ArrayList

- Si **`C`** est une classe (par exemple **`Student)`**, un objet de la classe **`ArrayList<C>`** est une liste de références sur des objets de la classe **`C`**.

- Cette liste est vide lorsqu'elle est créée (longueur nulle), mais elle s'allonge automatiquement lorsqu'on lui ajoute des éléments.

- Les éléments de la liste sont indexés à partir de la valeur **`0`**.

La classe ArrayList

- La longueur de la liste est modifiée par les méthodes:

```
boolean add(C c)
boolean add(int index, C C)
boolean remove(int index)
C get(int index)
```

etc.

- Attention aux exceptions lors de l'utilisation de ces méthodes:
  - Utiliser la méthode **`size()`** pour vérifier qu'il existe bien un élément à l'index spécifié.

```
Student myStudent;

if (index > -1 && index < myList.size() {
    myStudent = myList.get(index);
}
else {
    myStudent = null;
}
```

Les génériques

- La classe **`ArrayList<C>`** est ce que l'on appelle une classe **générique**.

- Une classe générique est une classe qui est paramétrée par une autre classe.
  - La classe **`ArrayList<C>`** est paramétrée par la classe **`C`**.

`ArrayList<Student>`

- La programmation des classes génériques n'est pas au programme. Seule l'utilisation des classes génériques prédéfinies l'est.

La boucle for: syntaxe classique

```
ArrayList<Student> studentList = new ArayList();
studentList.add(…);
…

for (int index = 0; index < studentList.size(); index++) {
    Student student = studentList.get(index);
    System.out.println(student);
    …
}
```

- La variable **`index`** va prendre successivement pour valeur toutes les valeurs de 0 à la taille de la liste – 1.

- La variable **`student`** va prendre successivement pour valeur tous les éléments de la structure **`studentList`**.

La boucle for: syntaxe simplifiée

```
ArrayList<Student> studentList = new ArayList();
studentList.add(…);
…

for (Student student : studentList) {
    System.out.println(student);
    …
}
```

- La variable **`student`** va prendre successivement pour valeurs tous les éléments de la structure **`studentList`**.

- Cette méthode de parcours est utilisable avec toutes les structures de données proposées par le JDK de Java.
  - JDK 5 et plus.