ITI 1521. Introduction à l'informatique II

Héritage : réutilisabilité

by

Marcel Turcotte

Version du 29 janvier 2020

Préambule

Préambule

Aperçu

Aperçu

Héritage : réutilisabilité

Le concept d'héritage en Java favorise la réutilisation de code. L'héritage exprime une relation de type parent-enfant entre deux classes. La sous-classe possède les attributs et méthodes de la superclasse. Celle-ci peut aussi introduire de nouveaux attributs et de nouvelles méthodes. Finalement, la sous-classe peut redéfinir certaines méthodes de la superclasse.

Objectif général :

- Cette semaine, vous serez en mesure de structurer un ensemble de classe de façon hiérarchique à l'aide de l'héritage.

Une vidéo d'introduction :

- `https://www.youtube.com/watch?v=TuPV5om-mVQ`

Préambule

Objectifs d'apprentissage

Objectifs d'apprentissage

- **Décrire** le fonctionnement d'une application simple utilisant les concepts d'héritage.
- **Construire** une application simple à partir de sa spécification et de diagrammes UML.
- **Critiquer** l'usage du modificateur de visibilité «protected».

Lectures :

- Pages 7–31, 39–45 de E. Koffman et P. Wolfgang.

Préambule

Plan du module

Plan

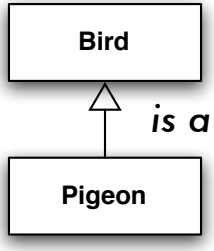
Généralisation/spécialisation

Introduction

- Les langages orienté objet offrent plusieurs mécanismes afin **structurer les programmes**.
- L'**héritage** est l'un de ces mécanismes et il favorise l'organisation des classes de façon **hiérarchique** (sous forme d'arborescence).
- Lorsqu'on dit que la programmation orientée objet favorise la **réutilisation de code** on fait alors référence à la notion d'héritage.

Définitions : superclasse et sous-classe

La classe située au-dessus, dans l'arbre d'héritage, s'appelle la **superclasse** (ou **parent**), alors que la classe située au-dessous s'appelle **sous-classe** (on dit aussi classe **dérivée**).

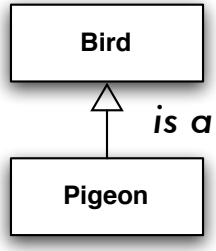


Dans cet exemple, **Bird** est la superclasse de **Pigeon**, c'est-à-dire que **Pigeon** est une sous-classe de **Bird**.

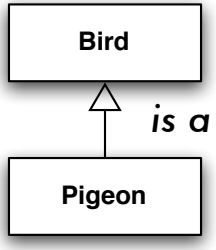
Java : extends

En Java, la relation «*is a*» (est un) est exprimée à l'aide du mot clé réservé **extends**.

```
public class Pigeon extends Bird {
    ...
}
```

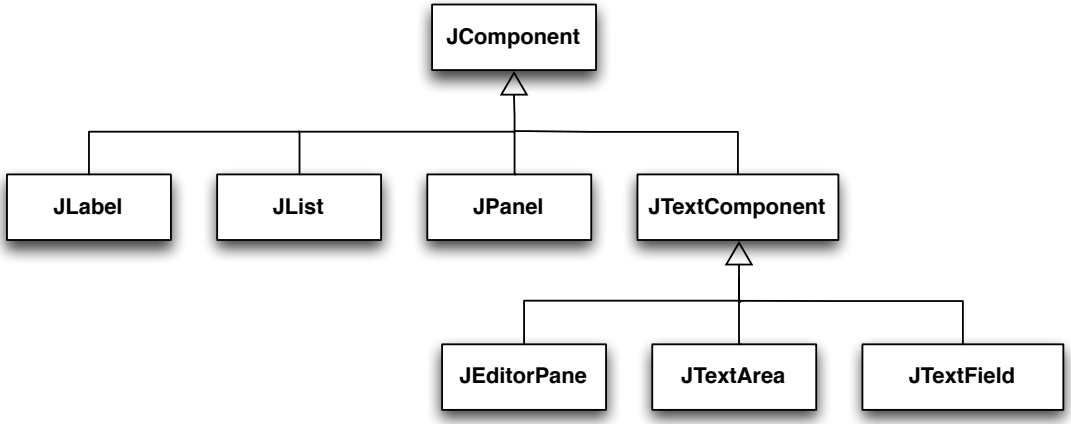


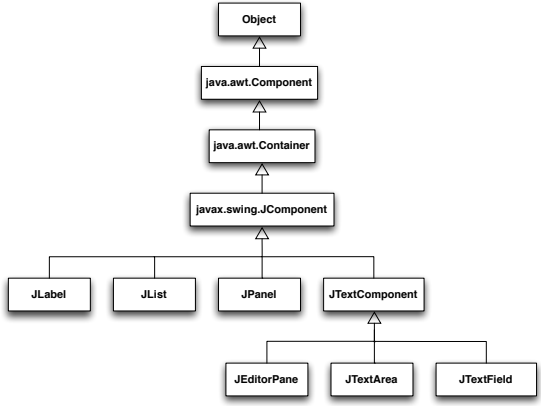
En **UML**, la relation «*is a*» est exprimée à l'aide d'une **ligne pleine** connectant l'enfant à son parent et telle qu'un **triangle ouvert** pointe dans la direction du parent.



Exemple : JComponent

- Un élément graphique s'appelle une composante graphique (**component**). Conséquemment, il existe une classe nommée **JComponent** qui définit les caractéristiques communes des composantes.
- Les sous-classes de **JComponent** incluent : JLabel, JList, JMenuBar, JPanel, JScrollBar, JTextComponent, et.





- **AWT** et **Swing** utilisent fortement l'héritage. La classe **Component** définit l'ensemble des méthodes communes aux objets graphiques, telles que **setBackground(Color c)** et **getX()**.
- La classe **Container** définie le comportement des objets graphiques pouvant contenir des objets graphiques, la classe définit les méthodes **add(Component component)** et **setLayout(LayoutManager mgr)**, entre autres.

Ça sert à quoi ?

Une classe hérite des **caractéristiques** (variables et méthodes) de sa superclasse.

1. Une sous-classe **hérite** des méthodes et variables de sa superclasse ;
2. Une sous-classe peut **introduire/ajouter** de nouvelles méthodes et variables ;
3. Une sous-classe peut **redéfinir** les méthodes de la superclasse.

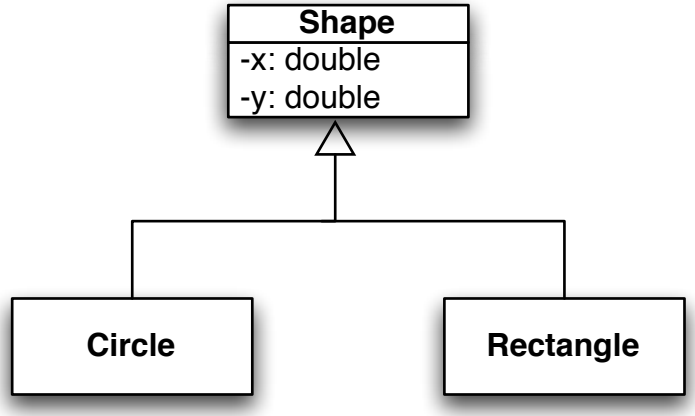
Puisqu'on ne peut qu'ajouter de nouveaux éléments, ou les redéfinir, une superclasse est plus **générale** que ses sous-classes, et inversement une sous-classe est plus **spécialisée** que sa superclasse.

Exemple

Exemple : Shape

- **Problème :** On doit construire un ensemble de classes afin de représenter des formes géométriques, telles que des cercles et des rectangles.
- **Tous les objets** doivent posséder deux variables d'instance, **x** et **y**, qui représentent la position de l'objet.

Exemple : Shape



```
public class Shape {

    private double x;
    private double y;

    public Shape() {
        x = 0;
        y = 0;
    }
}
```

```java
public class Shape {

    private double x;
    private double y;

    public Shape() {
        x = 0;
        y = 0;
    }
    public Shape(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

- Oui, oui ! Plusieurs méthodes (ou constructeurs) peuvent porter le même nom, à condition que les signatures des méthodes (constructeurs) diffèrent.
- Il s'agit de la **surcharge de nom** (**overloading**).

Méthodes d'accès

```
public class Shape {

    private double x;
    private double y;

    // ...

    public double getX() {
        return x;
    }
    public double getY() {
        return y;
    }

}
```

Méthodes toString

```
public class Shape {

    private double x;
    private double y;

    // ...

    public double getX() { return x; }
    public double getY() { return y; }

    public String toString() {
        return "Located at: (" + x + "," + y + ")";
    }

}
```

Exemple : Circle

```
public class Circle extends Shape {


}
```

- La déclaration ci-haut signifie que la classe **Circle** est une sous-classe de la classe **Shape**.
- Tous les objets de la classe **Circle** auront deux variables d'instance, **x** et **y**, ainsi que les méthodes suivantes, **getX()** et **getY()**.

Exemple : Circle

```
public class Circle extends Shape {

    // Variable d'instance
    private double radius;

}
```

Private versus protected

```
public class Circle extends Shape {

    private double radius;

    public Circle(double x, double y, double radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

}
```

La compilation du constructeur ci-dessus produira l'**erreur** suivante «*x has private access in Shape*» (pareillement pour y).

protected

On peut résoudre ce problème en déclarant les variables **protected** dans la superclasse **Shape**.

```
public class Shape {

    protected double x;
    protected double y;

    // ...
}
```

Private

- Je préfère garder la visibilité des variables **private**.
- Ce qui nous forcera à utiliser les **méthodes d'accès** de la superclasse.

super

```java
public class Circle extends Shape {

    private double radius;

    // Constructeurs

    public Circle() {
        super();
        radius = 0;
    }

    public Circle(double x, double y, double radius) {
        super(x, y);
        this.radius = radius;
    }

}
```

super

L'énoncé **super(. . .)** est un appel explicite au constructeur de la superclasse immédiate.

- Cette forme d'appel, super(. . .), n'apparaît **que dans un constructeur**
- Cet appel doit être le **premier énoncé** du constructeur
- Un appel de la forme **super()** est **automatiquement** inséré à moins que vous n'ajoutiez un appel **super(. . .)** vous-même ! ?

```java
public class Circle extends Shape {

    private double radius;

    public Circle() {
        super();
        radius = 0;
    }

    public Circle(double x, double y, double radius) {
        super(x, y);
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

}
```

Exemple : Rectangle

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    public Rectangle() {
        super();
        width = 0;
        height = 0;
    }

    public Rectangle(double x, double y, double width, double height) {
        super(x, y);
        this.width = width;
        this.height = height;
    }
}
```

Exemple : Rectangle

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    // ...

    public double getWidth() {
        return width;
    }

    public double getHeight() {
        return height;
    }
}
```

Utilisation

```
Circle c, d;

d = new Circle(100.0, 200.0, 10.0);
System.out.println(d.getRadius());

c = new Circle();
System.out.println(c.getX());

Rectangle r, s;

r = new Rectangle();
System.out.println(r.getWidth());

s = new Rectangle(50.0, 50.0, 10.0, 15.0);
System.out.println(s.getY());
```

Prologue

Résumé

- L'héritage nous permet d'organiser les classes de façon hiérarchique
- Le mot clé **extends** dans la signature d'une classe indique son parent

Prochain module

- Polymorphisme

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures : Abstraction and Design Using Java.
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (SIGE)

Université d'Ottawa