ITI 1121. Introduction to Computing II

Inheritance: reusability

by

Marcel Turcotte

Version January 30, 2020

Preamble

Preamble

Overview

Overview

Inheritance: reusability

The concept of inheritance in Java promotes code reuse. Inheritance expresses a parent-child relationship between two classes. The subclass has the attributes and methods of the superclass. The subclass can also introduce new attributes and methods. Finally, the subclass can redefine certain methods of the superclass.

General objective :

- This week you will be able to structure a collection of classes hierarchically using inheritance.

Preamble

Learning objectives

Learning objectives

- **Describe** the functioning of a simple application using inheritance concepts.
- **Build** a simple application from its specification and UML diagrams.
- **Criticize** the use of the visibility modifier "protected".

Lectures:

- Pages 7–31, 39–45 of E. Koffman and P. Wolfgang.

Preamble

Plan

Plan

1. Preamble
2. Generalization/specialization
3. Example
4. Prologue

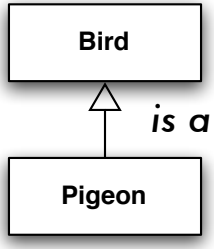
Generalization/specialization

Introduction

- Object-oriented languages offer several mechanisms to structure programs.
- **Inheritance** is one of these mechanisms and it favours the organization of classes in a **hierarchical** way (in the form of a tree structure).
- When we say that object-oriented programming favours **code reuse** we are referring to the notion of inheritance.

Definitions: superclass and subclass

The class above in the inheritance tree is called the **superclass** (or **parent**), while the class below is called **subclass** (also called **derived** class).

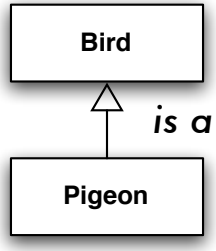


In this example, **Bird** is the superclass of **Pigeon**, that is, **Pigeon** is a subclass of **Bird**.

Java: extends

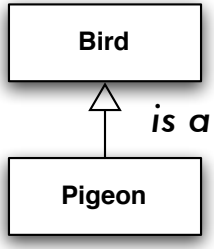
In Java, the relationship "*is a*" is expressed using the reserved keyword **extends**.

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



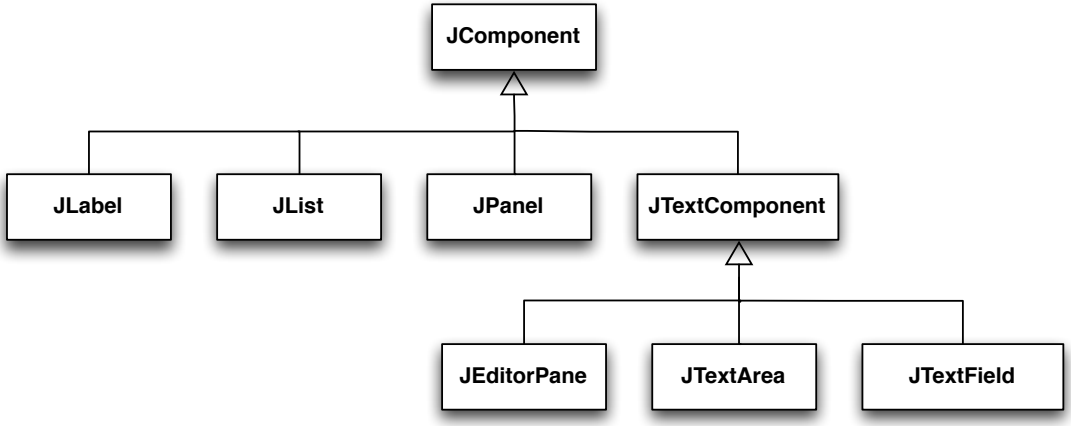
UML

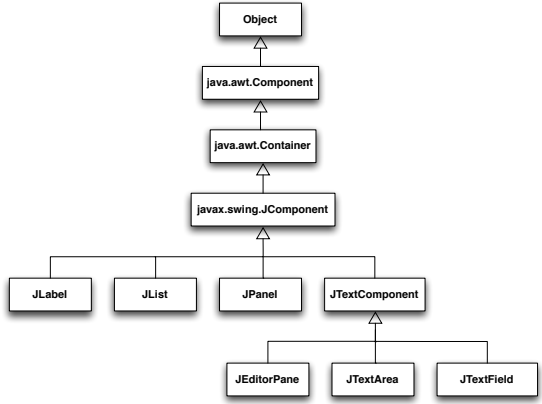
In **UML**, the relationship "*is a*" is expressed using a **full line** connecting the child to the parent and such that an **open triangle** points in the direction of the parent.



Example: JComponent

- A graphic element is called a graphic **component**. Consequently, there is a class named **JComponent**, which defines the common characteristics of the components.
- The subclasses of **JComponent** include: JLabel, JList, JMenuBar, JPanel, JScrollBar, JTextComponent, etc.





- **AWT** and **Swing** make heavy use of inheritance. The class **Component** defines the set of methods common to graphic objects, such as **setBackground(Color c)**and **getX()**.
- The **Container** class defines the behavior of graphical objects that can contain other graphical objects, the class defines the methods **add(Component component)** and **setLayout(LayoutManager mgr)**, among others.

What's it for?

A class inherits the **characteristics** (variables and methods) of its superclass.

1. A subclass **inherits** the methods and variables of its superclass;
2. A subclass can **introduce/add** new methods and variables;
3. A subclass can redefine the methods of the superclass.

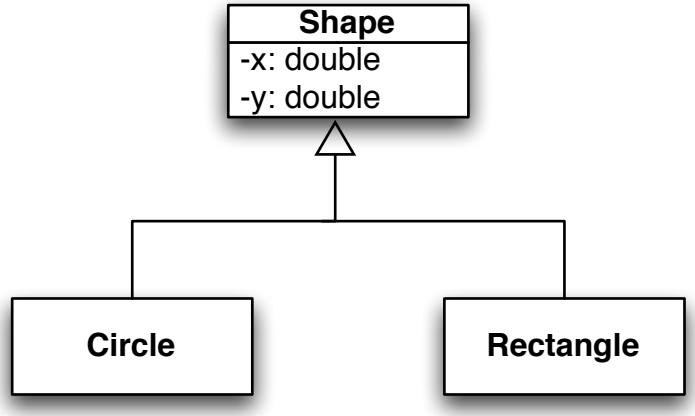
Since we can only add new elements, or redefine them, a superclass is more general than its subclasses, and conversely, and conversely, a subclass is more specialized than its superclass.

Example

Exemple: Shape

- **Problem:** We need to create a set of classes to represent geometric shapes, such as circles and rectangles.
- **All the objects** must have two instance variables, **x** and **y**, which represent the position of the object.

Example: Shape



```
public class Shape {

    private double x;
    private double y;

    public Shape() {
        x = 0.0;
        y = 0.0;
    }
}
```

```
public class Shape {

    private double x;
    private double y;

    public Shape() {
        x = 0.0;
        y = 0.0;
    }
    public Shape(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

- Yes, yes, yes, yes, yes! Several methods (or constructors) may have the same name, provided that the signatures of the methods (constructors) differ.
- It's called **overloading**.

Acces methods

```
public class Shape {

    private double x;
    private double y;

    // ...

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

}
```

Method 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 + ")";
    }

}
```

Example: Circle

```
public class Circle extends Shape {


}
```

- The above statement implies that the class **Circle** is a subclass of the class **Shape**.
- All the objects of the class **Circle** will have two instance variables, **x** and **y**, as well as the following methods, **getX()** and **getY()**.

Example: Circle

```
public class Circle extends Shape {

    // Instance variable
    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;
    }

}
```

Compiling the above will produce the following **error** "*x has private access in Shape*" (similarly for y).

Protected

The compile time error message can be eliminated by declaring the variables **protected** in the superclass **Shape**.

```
public class Shape {

    protected double x;
    protected double y;

    // ...
}
```

Private

- I prefer to keep visibility of the variables **private**.
- Forcing us to use the superclass's **access methods**.

super

```
public class Circle extends Shape {

    private double radius;

    // Constructors

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

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

}
```

Super

The statement **super(...)** is an explicit call to the constructor of the immediate superclass.

- This construct, **super(...)**, only appears in a constructor.
- That call must be the **first statement** of the constructor.
- A call of the form **super()** is automatically inserted unless you add a call **super(...)** yourself?

```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;
    }

}
```

Example: 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;
    }
}
```

Example: Rectangle

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    // ...

    public double getWidth() {
        return width;
    }

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

Usage

```
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

Summary

- Inheritance allows us to organize classes hierarchically.
- The keyword **extends** in the signature of a class indicates its parent.

Next module

- Polymorphism

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

School of Electrical Engineering and **Computer Science** (EECS)

University of Ottawa