ITI 1121. Introduction to Computing II *

Guy-Vincent Jourdan
(based on Marcel Turcotte's slides)
School of Electrical Engineering and Computer Science

Version of January 3, 2018

Abstract

- Inheritance
  - Introduction
  - Generalization/specialization
  - Polymorphism

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

Summary

$\Rightarrow$ In today's lecture, we look at other important features of object-oriented programming that help organizing and maintaining large software systems: *inheritance* and *polymorphism*.

Inheritance

OO languages offer tools to structure large systems. **Inheritance** is one of them.

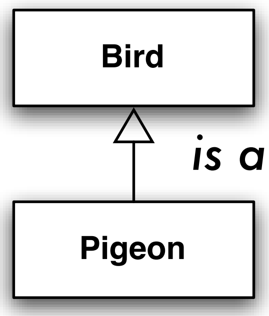
Inheritance allows to organize the classes hierarchically.

Inheritance favors **code reuse**!

Inheritance is one of the tools that help developing reusable components (classes).

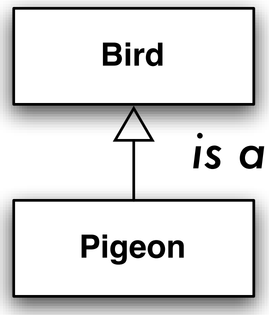
Inheritance

The class immediately above is called the **superclass** or **parent class** while the class immediately below is called the **subclass**, **child class** or **derived class**.



In this example, **Bird** is the superclass of **Pigeon**, i.e. **Pigeon** "is a" subclass of **Bird**.

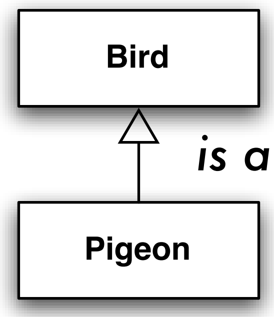
Inheritance



In Java, the "is a" relationship is expressed using the reserved keyword **extends**, as follows:

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

Inheritance



In UML, the "is a" relationship is expressed using a continuous line connecting the child to its parent, and an open triangle pointing towards the parent.

What does it mean?

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

1. a subclass inherits all the methods and variables of its superclass(es);

2. a subclass can introduce/add new methods and variables;

3. a subclass can override the methods of its superclass.

Because of 2 and 3, the subclass is a **specialization** of the superclass, i.e. the superclass is **more general** than its subclasses.

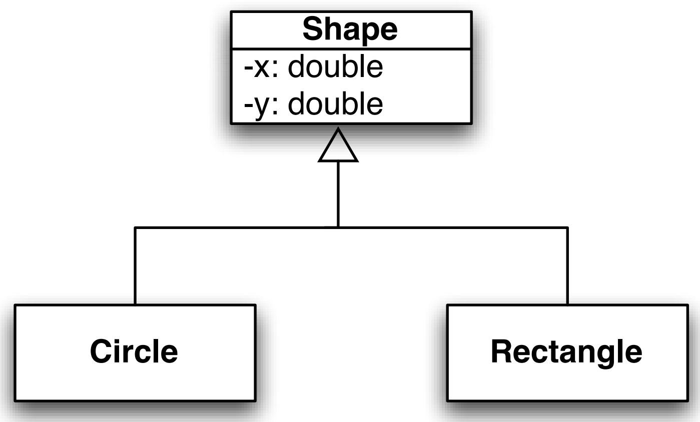
Shape

Variants of this example can be found in most textbooks about object-oriented programming.

Problem: A software system must be developed to represent various shapes, such as circles and rectangles.

All the shapes must have two instance variables, **x** and **y**, to represent the location of each object.

Shape



Shape

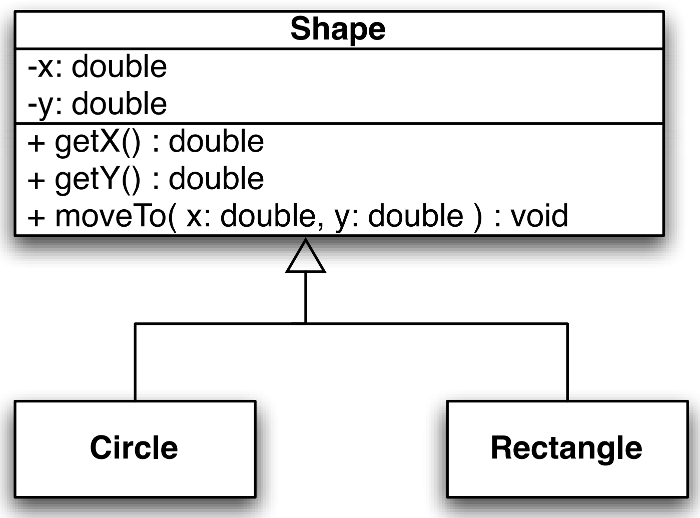
Furthermore, **all the shapes** should have the following methods:

```
double getX();                       // Returns the value of x

double getY();                       // Returns the value of y

void moveTo(double x, double y);     // Move the shape to a new location

double area();                       // Calculates the area of the shape

String toString();                   // Returns a String representation
```

Keep the specification in mind as we won't be able to implement it fully, at first.

Shape

The implementation of the first three methods would be the same for all kinds of shapes.



Shape

On the other hand, the calculation in the **area** method would depend on the kind of shape being dealt with.

Finally, the method **toString()** requires information from both levels, general and specific, all shapes should display their location and also their specific information, such as the radius in the case of a circle.

Shape

```
public class Shape {



}
```

Shape

```
public class Shape {

    private double x;
    private double y;

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

Shape

```
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;
    }
}
```

Shape

```
public class Shape {

    private double x;
    private double y;

    (...)

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

}
```

Adding the getters!

Shape

```
public class Shape {

    private double x;
    private double y;

    (...)

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

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

Shape

```
public class Shape {

    private double x;
    private double y;

    (...)

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

    public void moveTo( double x, double y ) {...}

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

Circle

```
public class Circle extends Shape {


}
```

The above declaration defines a class **Circle** that extends Shape, which means that an instance of the class **Circle** possesses two instance variables **x** and **y**, as well as the following methods: **getX()**, **getY()**, **moveTo(double x, double)** and **toString()**.

Circle

```
public class Circle extends Shape {

    // Instance variable
    private double radius;

}
```

The instance variables **x** and **y** and inherited (common to all **Shapes**). The variable **radius** is specific to a **Circle**.

Circle

```
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 particular construction can only appear in a constructor;
- Can only be the first statement of the constructor;
- The **super()** will be automatically inserted for you unless you insert a **super(. . .)** yourself!

⇒ If the first statement of a constructor is not an explicit call **super(. . .)**, Java inserts a call **super()**, which means that the superclass has to have a constructor of arity 0, or else a compile time error will occur. Remember, the default constructor, the one with arity 0, is no longer present if a constructor has been defined.

Circle

```
public class Circle extends Shape {

    private double radius;

    (...)

    // Getters
    public double getRadius() {
        return radius;
    }

}
```

Circle

Overridding **toString()**

```
public class Circle extends Shape {

    private double radius;

    (...)

    // Getters
    public double getRadius() {
        return radius;
    }

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

Private vs protected

With the current definition of the class **Shape**, compiling **Circle** will now cause the following error:

The compiler would complain saying "x has private access in Shape" (and similarly for **y**).

This is because an attribute declared private in the parent class cannot be accessed within the child class.

Private vs protected

To circumvent this and implement the constructor as above, the definition of **Shape** should be modified so that **x** and **y** would be declared **protected**:

```
public class Shape  {

    protected double x;
    protected double y;

    ...
}
```

When possible, it is preferable to maintain the visibility **private**.

The declaration of an instance variable **private** prevents the subclasses from accessing the variable.

“final” methods

What about the methods? We can prevent subclasses from overridding a method by making this method **final**.

```
public class Shape {
    // ...

    public final double getX() { return x; }
    public final double getY() { return y; }
    public final void moveTo( double x, double y ) {...}

    // ...
}
```

⇒ Now, no subclass of class **Shape** will be able to modify (override) the methods **getX()**, **getY()** and **moveTo(double, double)**.

Testing

```
public class Test  {
    public static void main(String[] args){
        Shape shape1 = new Shape();
        Shape shape2 = new Shape(3,5);
        Circle circle1 = new Circle();
        Circle circle2 = new Circle(5,6,7);

        System.out.println(shape1.toString());
        System.out.println(shape2);
        System.out.println(circle1+ " and " + circle2);

        shape1.moveTo(10,11);
        System.out.println(shape1);

        circle2.moveTo(20,21);
        System.out.println(circle2);
    }
}
```

Testing

```
> javac Test.java
> java Test
Located at: (0.0,0.0)
Located at: (3.0,5.0)
Located at: (0.0,0.0), with radius 0.0 and Located at: (5.0,6.0), with
Located at: (10.0,11.0)
Located at: (20.0,21.0), with radius 7.0
>
```

Similarly, we can define 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;
    }
```

```
    public double getWidth() {
        return width;
    }

    public double getHeight() {
        return height;
    }

    public void flip() {
        double tmp = width;
        width = height;
        height = tmp;
    }

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

}
```

Polymorphism

From the Greek words *polus* = many and *morphê* = forms, literally means has many forms.

1. *Ad hoc* polymorphism (overloading): a method name is associcated with different blocs of code

2. Inclusion (subtyping, data) polymorphism: an identifer (a reference variable) is associated with data of different types with the use of a subtype relation

In Java, a variable or a method is polymorphic if it refers to objects of more than one "class/type".

Method overloading

Method overloading means that two methods can have the same name but different signatures (the signature consists of the name and formal parameters of a method but not the return value).

Constructors are often overloaded, this occurs for the class `Shape`:

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

$\Rightarrow$ Method overloading is sometimes referred to as *ad hoc* polymorphism
(*ad hoc* = for a specific purpose).

Overloading (contd)

The class **PrintStream** has a specific **println** method for each primitive type (a good example of overloading):

```
println()
println( boolean )
println( char )
println( char[] )
println( double )
println( float )
println( int )
println( long )
```

“True” polymorphism: motivation 1

Problem: implement the method **isLeftOf** that returns **true** if **this Shape** is to the left of its argument.

isLeftOf

```
Circle c1 = new Circle( 10, 20, 5 );
Circle c2 = new Circle( 20, 10, 5 );
Rectangle r1 = new Rectangle( 0, 0, 1, 1 );
Rectangle r2 = new Rectangle( 100, 100, 200, 400 );

if ( c1.isLeftOf( c2 ) ) { ... }

if ( r1.isLeftOf( r2 ) ) { ... }

if ( r1.isLeftOf( c1 ) ) { ... }

if ( c1.isLeftOf( r1 ) ) { ... }
```

isLeftOf

```
public class Shape {



}
```

Absurd solution!

```
public boolean isLeftOf( Circle c ) {
    return getX() < c.getX();
}
public boolean isLeftOf( Rectangle r ) {
    return getX() < r.getX();
}
```

- As many implementations as kinds of shape!
- All the implementations are the same!
- Whenever a new kind of **Shape** is defined (say Triangle) then a method **isLeftOf** must be created!

Solution

Implement the method **isLeftOf** in the class **Shape** as follows.

```
public boolean isLeftOf( Shape s ) {
    return getX() < s.getX();
}
```

isLeftOf

```
Circle c = new Circle( 10, 20, 5 );
Rectangle r = new Rectangle( 0, 0, 1, 1 );

if ( c.isLeftOf( r ) ) { ... }
```

c designates an object of the class **Circle**, which inherits the method **isLeftOf**.

When the method **isLeftOf** is called, the value of the actual parameter, **r**, is copied into the formal parameter **s**.

This works because **Rectangle IS A Shape**.

Types

“A variable is a storage location and has an associated type, sometimes called its compile-time type, that is either a primitive type (§4.2) or a reference type (§4.3). A variable always contains a value that is assignment compatible (§5.2) with its type.”

“Assignment of a value of compile-time reference type S (source) to a variable of compile-time reference type T (target) is checked as follows:

- If S is a class type:
  - If T is a class type, then S must either be the same class as T, or S must be a subclass of T, or a compile-time error occurs.

”

$\Rightarrow$ Gosling et al. (2000) *The Java Language Specification.*

isLeftOf

Based on that definition, the following statement is valid.

```
Shape s;
Rectangle r;
r = new Rectangle( 0, 0, 1, 1 );
s = r;
```

but "**r = s**" is not!

Polymorphic variable

The variable **s** designates any object that is from a subclass of **Shape**.

```
Shape s;
s = new Circle( 0, 0, 1 );
s = new Rectangle( 10, 100, 10, 100 );

Circle c = new Circle( 0, 0, 1 );
s = c;

if ( s.getX() ) { ... }
```

When **s** is used to designate a **Circle**, the **Circle** is "seen as" a **Shape**, meaning that only the characteristics (methods and variables) of the class **Shape** can be used.

```
if ( s.getRadius() ) { ... }
```

The above statement **is not valid**. The method **getRadius()** is not defined in the class **Shape** (or its parents).

Polymorphism

Polymorphism is a powerful concept. The method **isLeftOf** can be used to compare not only **Circles** and **Rectangles** but also any future subclass of **Shape**.

```
public class Triangle extends Shape {
   // ...
}
```

“True” polymorphism: motivation 2

Problem: Ensure that every shape implements the method **area**.

Unlike method **isLeftOf**, we don’t have a possible implementation of method **area** right at the level of class **Shape** (because we wouldn’t know what to return).

Relying on every subclass of shape to “remember” to implement the method is dangerous (someone can forget). We must **force** this implementation.

It would also prevent code such as:

```
Shape s = new Circle(1,1,5);
Double d = s.area();
```

Solution: abstract

The solution is to declare the method **area()** abstract in the superclass **Shape**. An **abstract** method is declared using the keyword **abstract**, it has a signature but no body.

```
public class Shape {
  // ...

  public abstract double area();  // <----

  // ...
}
```

The above definition does not compile! Imagine creating an object of the class **Shape**, that object would have a method **area()** that has no statements attached to it!

Solution: abstract class

```
public abstract class Shape { // <---
  // ...

  public abstract double area();  // <----

  //...
}
```

A **class** that has an **abstract method** must be **abstract**. One cannot create an object of an abstract class! The statement “new Shape()” would cause a compile-time error.

Abstract classes

- A class that contains an **abstract method** (declared in that class or inherited) **must** be declared abstract;

- An abstract class cannot be used to create objects;

- A class that contains no abstract methods **can** also be declared abstract to prevent the creation of objects of this class. E.g. Employee, SalariedEmployee, HourlyEmployee.

Solution: abstract class

What have we achieved?

```
public class Triangle extends Shape {

}
```

```
Triangle.java:1: Triangle is not abstract and
does not override abstract method area() in Shape
public class Triangle extends Shape {
       ^
1 error
```

It is now **impossible** to create a concrete subclass of **Shape** that has no method **area()**!

Solution: abstract methods and classes

Finally, we can now use the method **area** on a reference variable of an instance of **Shape** ! (which will actually be an instance of a subclass)

```
public abstract class Shape {
  // ...

  public abstract double area();

  public int compareTo( Shape other ) {
    if ( area() < other.area() )
        return -1;
    else if ( area() == other.area() )
        return  0;
    else
        return  1;
  }
}
```