Programming in the Large II: Objects and Classes (Part 2)

188230 Advanced Computer Programming

Asst. Prof. Dr. Kanda Runapongsa Saikaew
(krunapon@kku.ac.th)
Department of Computer Engineering
Khon Kaen University

Agenda

- Inheritance
- Polymorphism
- Abstract Classes
- Special variables this and super
- Interfaces

Inheritance and Polymorphism

- OOP allows classes to express the similarities among objects that share some, but not all, of their structure and behavior
- Such similarities can be expressed using inheritance and polymorphism
- The term inheritance refers to the fact that one class can inherit part or all of its structure

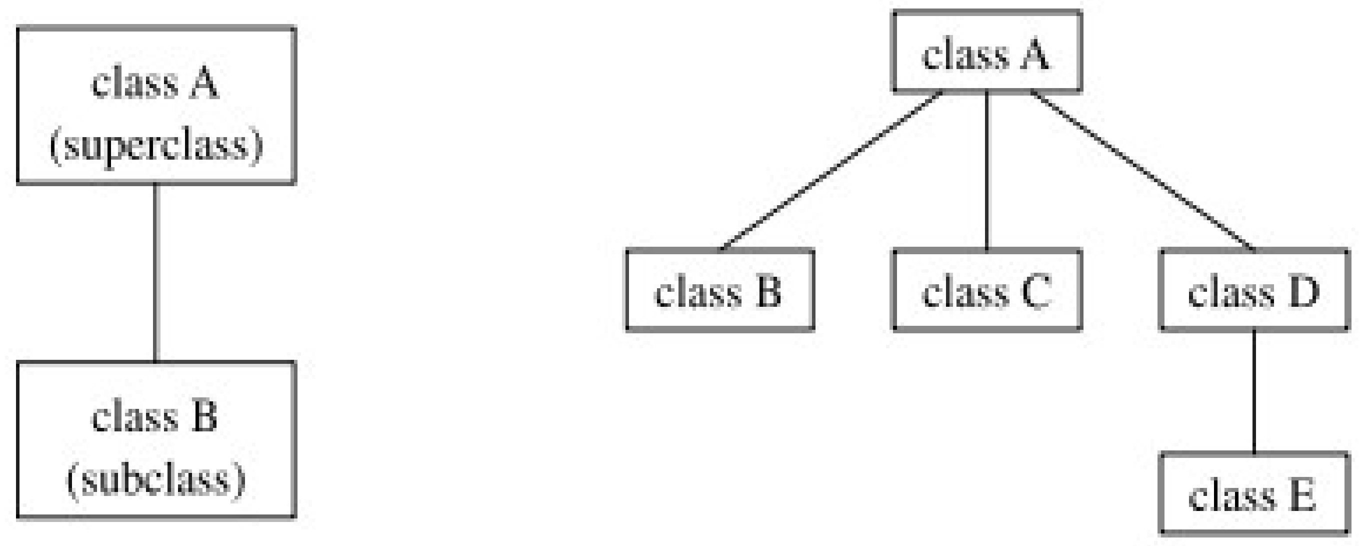
  and behavior from another class
- Polymorphism just means that different objects

  can respond to the same message in different ways

Inheritance

- The class that does the inheriting is said to be a subclass of the class from which it inherits
- If class B is a subclass of class A, we also say that class A is a superclass of class B
- A subclass can add to the structure and behavior that it inherits
- It can also replace or modify inherited behavior

Inheritance Diagram



- In the diagram shown on the right, above, classes B, C, and D are sibling classes
- Inheritance can also extend over several "generations" of classes
- This whole set of classes forms a small class hierarchy

Extending Existing Classes

- The existing class can be extended to make a subclass
- The syntax for this is

```
public class (subclass-name) extends (existing-class-name) {
        // Changes and additions
}
```

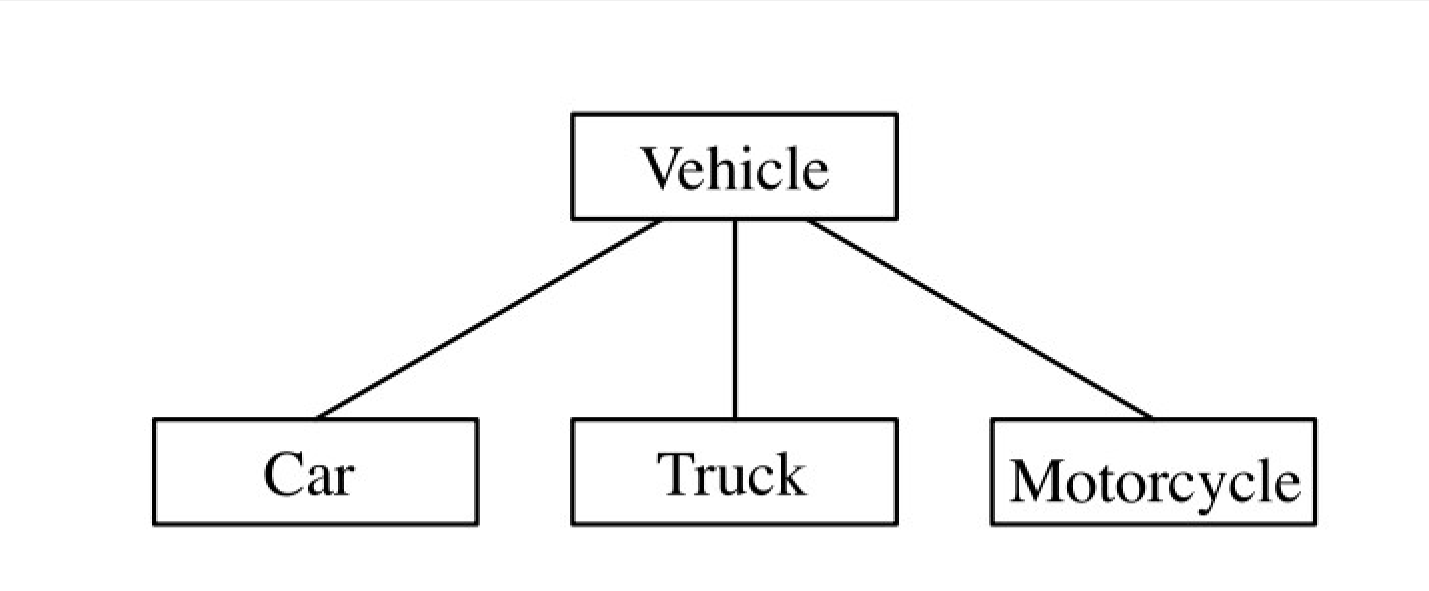
- Example

```
public class B extends A { … }
```

Examples: Vehicles

- Suppose that a program has to deal with motor vehicles, including cars, trucks, and motorcycles.
- The program could use a class named Vehicle to represent all types of vehicles
- Since cars, trucks, and motorcycles are types of vehicles, they would be represented by subclasses of the Vehicle class

Examples Diagram: Vehicle



Vehicle: Superclass

- The Vehicle class would include
  - Instance variables such as velocity
  - Instance methods such as getVelocity() and setVelocity()
- These are variables and methods common to all vehicles
- The three subclasses of Vehicle—Car, Truck, and Motorcycle—could then be used to hold variables and methods specific to particular types of vehicles

Subclasses of Vehicle

- The Car class might add an instance variable numberOfDoors
- The Truck class might have an instance variable numbeOfAxles
- The Motorcycle class could have a boolean variable hasBasket

Class Vehicle

```
public class Vehicle {
protected float velocity;
public float getVelocity() {
return velocity;
}
public void setVelocity(float newVelocity) {
velocity = newVelocity;
}
```

Class Car

```
class Car extends Vehicle {
private int numberOfDoors;
public int getNumberOfDoors() {
return numberOfDoors;
}
public void setNumberOfDoors(int newNumDoors) {
numberOfDoors = newNumDoors;
}
}
```

Testing Vehicle

```
public static void main(String[] args) {
Car myCar = new Car();
myCar.setVelocity(80);
myCar.setNumberOfDoors(4);
System.out.println("Velocity = " +
myCar.getVelocity() + " Number of doors = "
+ myCar.getNumberOfDoors());
}
```

Inherited Variables & Methods

- Suppose that myCar is a variable of type Car that has been declared and initialized with the statement

  ```
  Car myCar = new Car();
  ```

- Since class Car extends class Vehicle, a car also has all the structure and behavior of a vehicle
  - This means that a variable myCar.velocity exist
  - myCar.setVelocity() and myCar.getVelocity() also exist

A Variable and Inheritance

- Now, in the real world, cars, trucks, and motorcycles are in fact vehicles
- The same is true in a program. That is, an object of type Car or Truck or Motorcycle is automatically an object of type Vehicle too
- A variable that can hold a reference to an object of class A can also hold a reference to an object belonging to any subclass of A

Which Statements are Legal?

1. Vehicle myVehicle = myCar;
2. Vehicle myVehicle2 = new Car();
3. Car myCar2 = myVehicle;
4. Car myCar3 = new Vehicle();
5. Car myCar4 = (Car) myVehicle;

Checking Class of an Object

- The variable myVehicle holds a reference to a Vehicle object that happens to be an instance of the subclass, Car
- The object “remembers” that it is in fact a Car, and not just a Vehicle
- To test whether a given object belongs to a given class, using the instanceof operator. The test:

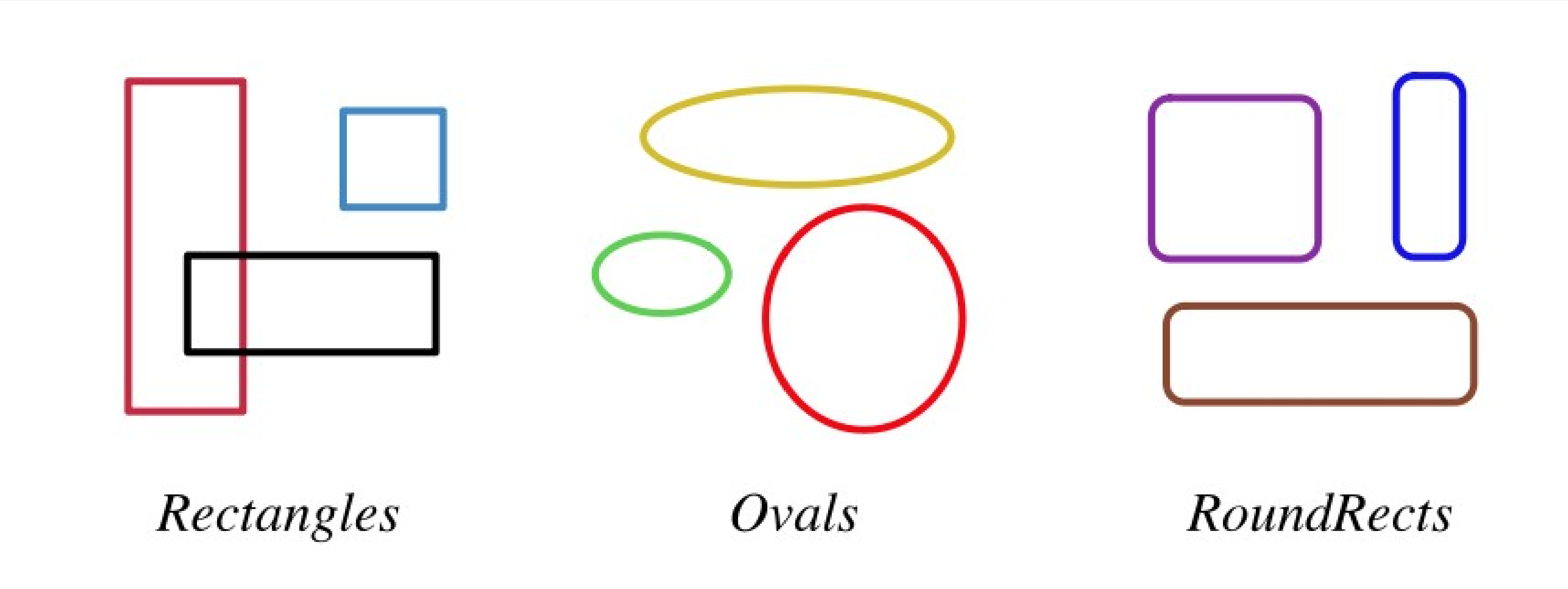
```
if (myVehicle instanceof Car) …
```

  - Determines whether the object referred to by myVehicle is in fact a car

Type Casting

- myCar = myVehicle; would be illegal because myVehicle could potentially refer to other types of vehicles that are not cars
- It's like we cannot assign an int value to a variable of type short, because not every int is a short
- Similarly, it will not allow you to assign a value of type Vehicle to a variable of type Car because not every vehicle is a car
- As in the case of ints and shorts, the solution here is to use type-casting.
  - myCar = (Car) myVehicle;

Examples: Shapes



- Three classes, Rectangle, Oval, and RoundRect, could be used to represent the three types of shapes
- These three classes would have a common superclass, Shape, to represent features that

  all three shapes have in common

Class Shape

```
public class Shape {
    protected String color;
    public void setColor(String newColor) {
     color = newColor;
    }
    public String getColor() {
     return color;
    }
    public void redraw() {}
}
```

Subclasses of Shape

-
  ```
  class RoundRect extends Shape {
   void redraw() {
   . . . // commands for drawing a rectangle
    } }
  ```

-
  ```
  class Oval extends Shape {
   void redraw() {
   . . . // commands for drawing a circle
    }}
  ```

Method redraw

- If oneShape is a variable of type Shape, it could refer to an object of any of the types,

  Rectangle, Oval, or RoundRect
- As a program executes, and the value of oneShape changes, it could even refer to objects of different types at different times
- Whenever the statement oneShape.redraw();

  is executed, the redraw method that is actually called is the one appropriate for the type of

  object to which oneShape actually refers.

Polymorphism

- It is possible that the very same statement “oneShape.redraw();” will call different methods and draw different shapes as it is executed over and over
- We say that the redraw() method is polymorphic
- A method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied
- Polymorphism is one of the major distinguishing features of object-oriented programming

Method redraw() in Shape

- Whenever a Rectangle, Oval, or RoundRect object has to draw itself, it is the redraw() method in the appropriate class that is executed
- This leaves open the question, What does the redraw() method in the Shape class do?
- How should it be defined?
  - We should leave it blank!
  - The fact is that the class Shape represents the abstract idea of a shape, and there is no way to draw such a thing

Abstract Classes

- You can have variables of type Shape, but the objects they refer to will always belong to one of the subclasses of Shape
- We say that Shape is an abstract class
- An abstract class is one that is not used to construct objects, but only as a basis for making subclasses

Abstract vs. Concrete Classes

- An abstract class exists only to express the common properties of all its subclasses
- A class that is not abstract is said to be concrete
- You can create objects belonging to a concrete class, but not to an abstract class.
- A variable whose type is given by an abstract class can only refer to objects that belong to concrete subclasses of the abstract class.

Abstract Method

- We say that the redraw() method in class Shape is an abstract method since

  it is never meant to be called
- The redraw() method in Shape has to be there only to tell the computer that all Shapes understand the redraw message
- As an abstract method, it exists merely to specify the common interface of all the actual, concrete versions of redraw() in the subclasses of Shape

Modifier “abstract”

- Shape and its redraw() method are semantically abstract
- You can also tell the computer, syntactically, that they are abstract by adding the modifier “abstract” to their definitions
- For an abstract method, the block of code that gives the implementation of an ordinary method

  is replaced by a semicolon
- An implementation must be provided for the abstract method in any concrete subclass of the abstract class

Abstract Class Shape

```
public abstract class Shape {
    protected String color;
    public void setColor(String newColor) {
    color = newColor;
    }
    public String getColor() {
    return color;
    }
    public abstract void redraw();
}
```

Abstract Class & Method

- Is this code legal? Why or why not?

```
class Triangle extends Shape {
    private float height, width;
}
```

- Is this code legal? Why or why not?

```
public static void main(String[] args) {
    Shape s = new Shape();
}
```

Special Variable “this”

- Java provides a special, predefined variable named “this” that you can use to refer to the object that contains the method
- This intent of the name, this, is to refer to “this object”
- If x is an instance variable in the same object, then this.x can be used as a full name for that variable
- If otherMethod() is an instance method

  in the same object, then this.otherMethod() could be used to call that method

Example of Using this

```
public class ThisDemo {
    private String name;
    public ThisDemo(String name) { this.name = name; }
    public void methodA() { System.out.println(“method A”);  }
    public String toString() { return "name:" + name;  }
     public void methodB() {
       this.methodA();
       System.out.println("method B");
       System.out.println(this);
   }
}
```

Example of Using this

```
public static void main(String[] args) {
  ThisDemo td = new ThisDemo("kku");
  td.methodB();
  }
```

- What is the output?

The Special Variable super

- Java also defines another special variable, named “super”, for use in the definitions of instance methods
- The variable super is for use in a subclass
- Like this, super refers to the object that contains the method but it remembers only that it belongs to the superclass of that class
- It can only be used to refer to methods and variables in the superclass

Using super with a Method

- Let’s say that the class that you are writing contains an instance method named

  doSomething()
- Consider the subroutine call statement super.doSomething()
  - It tries to execute a method named doSomething() from the superclass
  - If there is none—if the doSomething() method was an addition rather than a modification—you’ll get a syntax error

Using super with a Variable

- The reason super exists is so you can get access to things in the superclass that are hidden in the subclass
- For example, super.x always refers to an instance variable named x in the superclass
- The variable in the subclass does not replace

  the variable of the same name in the superclass; it merely hides it
- The variable from the superclass can still be accessed, using super.

Implementing a Method using super

- The major use of super is to override a method with a new method that extends the behavior of the inherited method
  - Instead of replacing that behavior entirely
- The new method can use super to call the method from the superclass
  - Then it can add additional code to provide additional behavior

Example: using super (1/3)

```
class Kid {
  String name;
  Kid() {
    name = "a kid";
  }
  public void play() {
    System.out.println(name + " likes to play with toys");
  }
}
```

Example: using super (2/3)

```
class SmallKid extends Kid {
String name;
SmallKid() { name = "a small kid"; }
SmallKid(String name) {this.name = name;}
public String toString() { return this.name + " " + super.name; }
public void play() {
super.play();
System.out.println(name + " likes to play with parents the most");
}
}
```

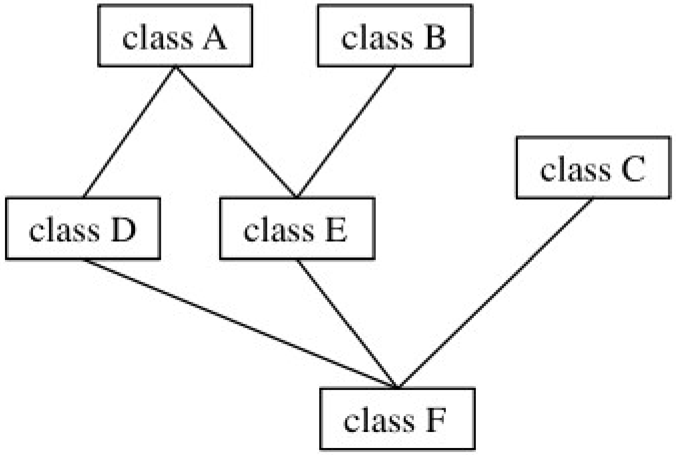
Example: using super (3/3)

```
public class SuperDemo {
public static void main(String[] args) {
SmallKid sk = new SmallKid("Ta");
System.out.println(sk);
sk.play();
}
}
```

- What is the output?

Multiple Inheritance

- Some object-oriented programming languages, such as C++, allow a class to extend two or more superclasses
- This is called multiple inheritance.



Multiple inheritance (NOT allowed in Java)

Interface

- Java does have a feature that can be used to accomplish many of the same goals as multiple inheritance: interfaces
- An “interface” in this sense consists of a set of instance method interfaces, without any associated implementations
- A class can implement an interface by providing an implementation for each of the methods specified by the interface

Interface Example

```
interface Drawable { public void draw(); }
interface Fillable { public void fill(); }
public class Line implements Drawable, Fillable {
public void draw() {
System.out.println("=== Drawing a line ===="); }
public void fill() {
System.out.println("=== Filling a line ===="); }
public static void main(String[] args) {
Line l = new Line();
l.draw(); l.fill();}}
```

References

- David J. Eck, ”Introduction to Programming Using Java”, Version 5.0, December 2006 http://math.hws.edu/javanotes/