


@NGUYEN Thi Thu Trang, trangntt@soict.hust.edu.vn

OBJECT-ORIENTED PROGRAMMING

## 7. ABSTRACT CLASS AND INTERFACE

Nguyen Thi Thu Trang  
trangntt@soict.hust.edu.vn



1050 01

1

2

## Outline

1. Redefine/Overriding
2. Abstract class
3. Single inheritance and multi-inheritance
4. Interface

1050 01

2

3

## Outline

1. Redefine/Overriding
2. Abstract class
3. Single inheritance and multi-inheritance
4. Interface

1050 01

3

4

## 1. Re-definition or Overriding

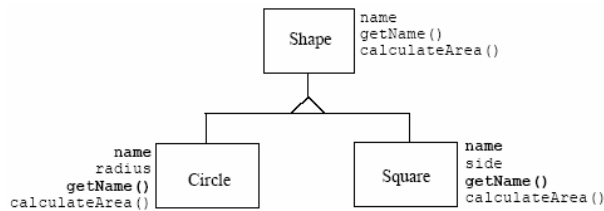
- A child class can define a method with the **same name** of a method in its parent class:
  - If the new method has the same name but different signature (number or data types of method's arguments)
    - Method Overloading
  - If the new method has the same name and signature
    - Re-definition or Overriding (Method Redefine/Override)

1050 01

4

## 1. Re-definition or Overriding (2)

- Overriding method will replace or add more details to the overridden method in the parent class
- Objects of child class will use the re-defined method



1050 04

7

## this and super

- **this** and **super** can use non-static methods/attributes and constructors
  - **this**: searching for methods/attributes in the current class
  - **super**: searching for methods/attributes in the direct parent class
- Keyword **super** allows re-using the source-code of a parent class in its child classes

1050 04

8

```

class Shape {
    protected String name;
    Shape(String n) { name = n; }
    public String getName() { return name; }
    public float calculateArea() { return 0.0f; }
}
class Circle extends Shape {
    private int radius;
    Circle(String n, int r){
        super(n);
        radius = r;
    }

    public float calculateArea() {
        float area = (float) (3.14 * radius * radius);
        return area;
    }
}
  
```

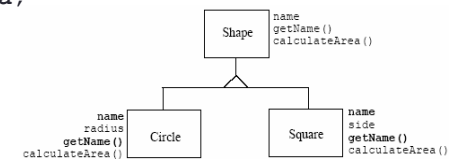
1050 04

10

```

class Square extends Shape {
    private int side;
    Square(String n, int s) {
        super(n);
        side = s;
    }

    public float calculateArea() {
        float area = (float) side * side;
        return area;
    }
}
  
```



1050 04

11

12

## Class Triangle

```
class Triangle extends Shape {
    private int base, height;
    Triangle(String n, int b, int h) {
        super(n);
        base = b; height = h;
    }
    public float calculateArea() {
        float area = 0.5f * base * height;
        return area;
    }
}
```

1050 11

12

13

```
package abc;
public class Person {
    private String name;
    private int age;
    public String getDetail() {
        String s = name + "," + age;
        return s;
    }
    private void pM(){}
}

import abc.Person;
public class Employee extends Person {
    double salary;
    public String getDetail() {
        String s = super.getDetail() + "," + salary;
        return s;
    }
}
```

1050 11

13

14

## Overriding Rules

- Overriding methods must have:
  - An argument list that is the same as the overridden method in the parent class => signature
  - The same return data types as the overridden method in the parent class
- Can not override:
  - Constant (final) methods in the parent class
  - Static methods in the parent class
  - Private methods in the parent class

1050 11

14

15

## Overriding Rules (2)

- Accessibility can not be more restricted in a child class (compared to in its parent class)
  - For example, if overriding a protected method, the new overriding method can only be protected or public, and can not be private.

1050 11

15

16

## Example

```
class Parent {
    public void doSomething() {}
    protected int doSomething2() {
        return 0;
    }
}
class Child extends Parent {
    protected void doSomething() {}
    protected void doSomething2() {}
}
```

cannot override: attempting to use incompatible return type

cannot override: attempting to assign weaker access privileges; was public

10/10/16

16

17

## Example: private

```
class Parent {
    public void doSomething() {}
    private int doSomething2() {
        return 0;
    }
}
class Child extends Parent {
    public void doSomething() {}
    private void doSomething2() {}
}
```

10/10/16

17

18

## Outline

1. Redefine/Overriding
2. Abstract class
3. Single inheritance and multi-inheritance
4. Interface

10/10/16

18

## Abstract Class

- An abstract class is a class that **we can not create its objects**. Abstract classes are often used to define "Generic concepts", playing the role of a basic class for others "detailed" classes.

- Using keyword abstract

```
public abstract class Product
{
    // contents
}
```

```
...Product aProduct = new Product(); //error
```

concrete class vs. abstract class

10/10/16

19

## 2. Abstract Class

- Can not create objects of an abstract class
- Is not complete, is often used as a parent class. Its children will complement the un-completed parts.

1050 06

20

## Abstract Class

- Abstract class can contain un-defined abstract methods
- Derived classes must re-define (overriding) these abstract methods
- Using abstract class plays an important role in software design. It defines common objects in inheritance tree, but these objects are too abstract to create their instances.

1050 06

21

## 2. Abstract Class (2)

- To be abstract, a class needs:
  - To be declared with **abstract** keyword
  - May contain abstract methods – that have only signatures without implementation
    - public abstract float calculateArea();
  - Child classes must implement the details of abstract methods of their parent class → Abstract classes can not be declared as final or static.
- If a class has one or more abstract methods, it must be an abstract class

1050 06

22

```

abstract class Shape {
    protected String name;
    Shape(String n) { name = n; }
    public String getName() { return name; }
    public abstract float calculateArea();
}

class Circle extends Shape {
    private int radius;
    Circle(String n, int r){
        super(n);
        radius = r;
    }
    public float calculateArea() {
        float area = (float) (3.14 * radius * radius);
        return area;
    }
}

```

```

classDiagram
    class Shape {
        <<abstract>>
        +String name
        +Shape(String n)
        +String getName()
        +float calculateArea()*
    }
    class Circle {
        +int radius
        +Circle(String n, int r)
        +float calculateArea()
    }
    class Square {
        +String name
        +int side
        +String getName()
        +float calculateArea()
    }
    Shape <|-- Circle
    Shape <|-- Square

```

Child class must override all the abstract methods of its parent class

1050 06

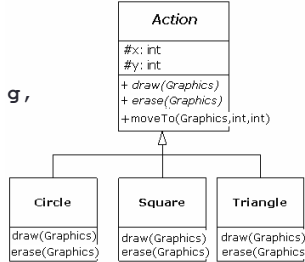
23

## Example of abstract class

```
import java.awt.Graphics;
abstract class Action {
    protected int x, y;
    public void moveTo(Graphics g,
        int x1, int y1) {
        erase(g);
        x = x1; y = y1;
        draw(g);
    }

    public abstract void erase(Graphics g);
    public abstract void draw(Graphics g);
}

..Circle c = new Circle();
c.moveTo(...);
```



24

## Example of abstract class (2)

```
class Circle extends Action {
    int radius;
    public Circle(int x, int y, int r) {
        super(x, y); radius = r;
    }
    public void draw(Graphics g) {
        System.out.println("Draw circle at ("
            + x + ", " + y + ")");
        g.drawOval(x-radius, y-radius,
            2*radius, 2*radius);
    }
    public void erase(Graphics g) {
        System.out.println("Erase circle at ("
            + x + ", " + y + ")");
        // paint the circle with background color...
    }
}
```

25

## Abstract Class

```
abstract class Point {
    private int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public void move(int dx, int dy) {
        x += dx; y += dy;
        plot();
    }
    public abstract void plot();
}
```

26

## Abstract Class

```
abstract class ColoredPoint extends Point {
    int color;
    public ColoredPoint(int x, int y, int color) {
        super(x, y); this.color = color;
    }
}

class SimpleColoredPoint extends ColoredPoint {
    public SimpleColoredPoint(int x, int y, int color) {
        super(x, y, color);
    }
    public void plot() {
        ...
        // code to plot a SimplePoint
    }
}
```

27

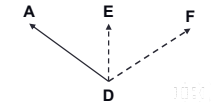
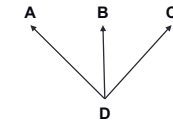
## Outline

1. Redefine/Overriding
2. Abstract class
- 3. Single inheritance and multi-inheritance
4. Interface

30

## Multiple and Single Inheritances

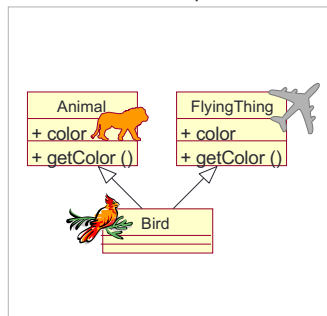
- Multiple Inheritance
  - A class can inherit several other classes
  - C++ supports multiple inheritance
- Single Inheritance
  - A class can inherit only one other class
  - Java supports only single inheritance
  - → Need to add the notion of Interface



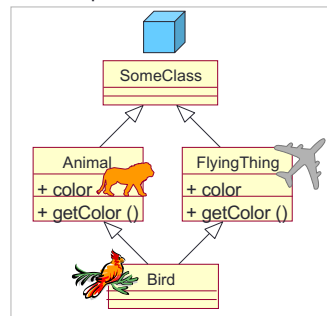
31

## Problems in Multiple Inheritance

Name clashes on  
attributes or operations



Repeated inheritance



Resolution of these problems is implementation-dependent

32

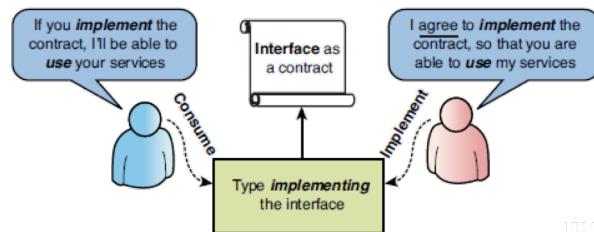
## Outline

1. Redefine/Overriding
2. Abstract class
3. Single inheritance and multi-inheritance
- 4. Interface

33

## What Is an Interface?

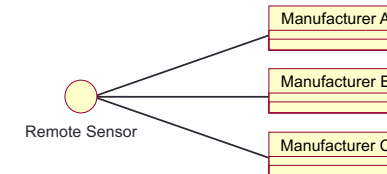
- A declaration of a coherent set of public features and obligations
- A contract between providers and consumers of services



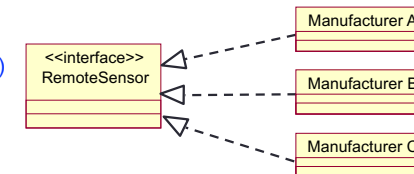
34

## Interface Representation in UML

Elided/Iconic Representation ("ball")

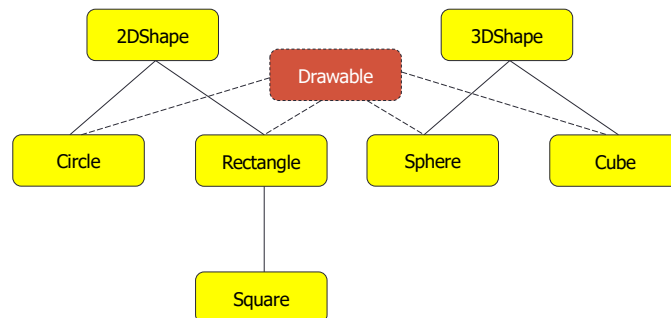


Canonical (Class/Stereotype) Representation



35

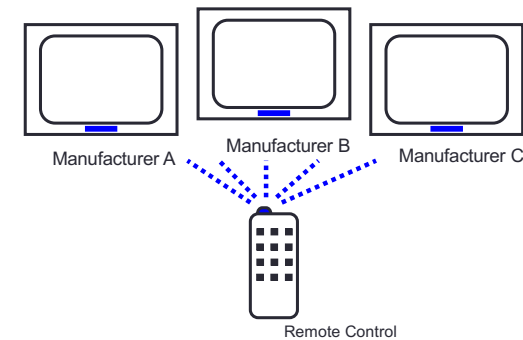
## Example: Interface



36

## Example: TV Interface

- A single interface for different concrete TVs



37



## Interface – Technique view (JAVA)

- An interface can be considered as a “class” that
  - Its methods and attributes are implicitly public
  - Its attributes are static and final (implicitly)
  - Its methods are abstract
- interface TVInterface {
 

```
public void turnOn();
public void turnOff();
public void changeChannel(int i);
```
- }
- class PanasonicTV implements TVInterface{
 

```
public void turnOn() { .... }
```
- }

38

## Interface in Java

- Allows a class to inherit (implement) multiple interfaces at the same time
- Can not directly instantiate
- To become an interface, we need
  - To use **interface** keyword to define
  - To write only:
    - method signature
    - static & final attributes
- Implementation class of interface
  - Abstract class
  - Concrete class: Must implement all the methods of the interface

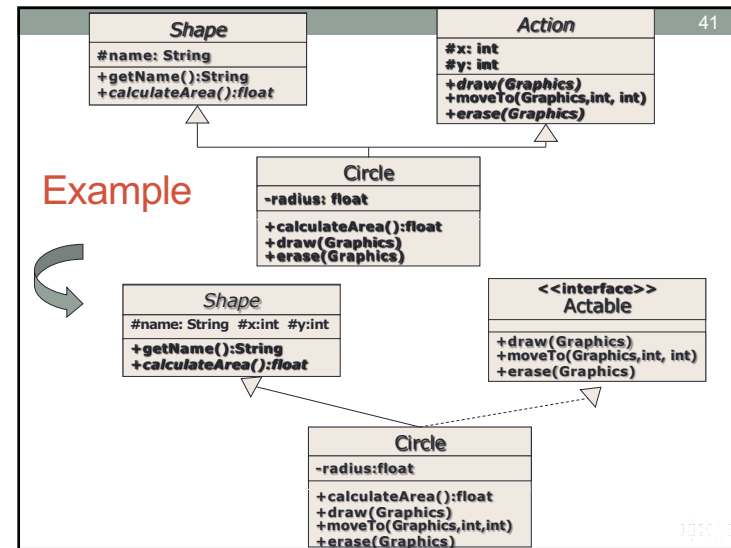
39

## Interface in Java

- Java syntax:
  - SubClass *extends* SuperClass *implements* ListOfInterfaces
  - SubInterface *extends* SuperInterface
- Example:
 

```
public interface Symmetrical {...}
public interface Movable {...}
public class Square extends Shape
    implements Symmetrical, Movable {
    ...
}
```

40



41

42

```
import java.awt.Graphics;
abstract class Shape {
    protected String name;
    protected int x, y;
    Shape(String n, int x, int y) {
        name = n; this.x = x; this.y = y;
    }
    public String getName() {
        return name;
    }
    public abstract float calculateArea();
}
interface Actable {
    public void draw(Graphics g);
    public void moveTo(Graphics g, int x1, int y1);
    public void erase(Graphics g);
}
```

1056 11

42

43

```
class Circle extends Shape implements Actable {
    private int radius;
    public Circle(String n, int x, int y, int r){
        super(n, x, y); radius = r;
    }
    public float calculateArea() {
        float area = (float) (3.14 * radius * radius);
        return area;
    }
    public void draw(Graphics g) {
        System.out.println("Draw circle at ("
            + x + ", " + y + ")");
        g.drawOval(x-radius, y-radius, 2*radius, 2*radius);
    }
    public void moveTo(Graphics g, int x1, int y1){
        erase(g); x = x1; y = y1; draw(g);
    }
    public void erase(Graphics g) {
        System.out.println("Erase circle at ("
            + x + ", " + y + ")");
        // paint the region with background color...
    }
}
```

1056 11

43

44

## Reading Assignment

- Compare Abstract Class and Interface
- Why can Interface resolve the problems of Multi-Inheritance?

1056 11

44