

OBJECT-ORIENTED LANGUAGE AND THEORY

## 7. ABSTRACT CLASS AND INTERFACE

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



1

2

### Outline

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

2

## Outline

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

3

## 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)

4

```
• class A {  
    a(){ .... }  
}  
  
• class B extends A {  
    a(String) {}  
}  
  
... B b = new B();  
    b.a();  
    b.a("test");
```

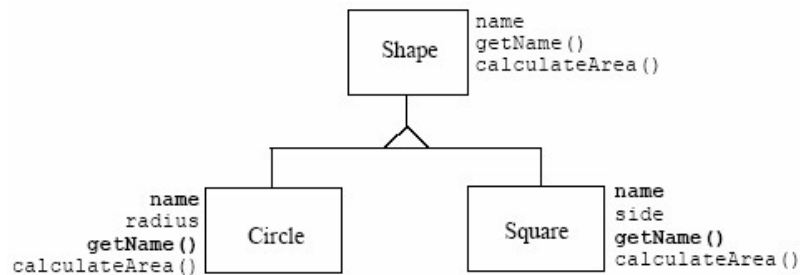
5

```
• ParentClass: aMethod() => overridden method  
  • ChildClass1: aMethod(), aMethod(String) => Overloading  
  • ChildClass2: aMethod() => Overriding/Redefinition method  
  
• ChildClass1 cc1 = new ChildClass1();  
• cc1.aMethod(); cc1.aMethod("a string");  
• ChildClass2 cc2 = new ChildClass2();  
• cc2.aMethod();
```

6

## 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



7

- `this()` and `this` => current object
- `super()` => Constructor of the parent class
- `super`: object of the parent class

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;
    }
}

```

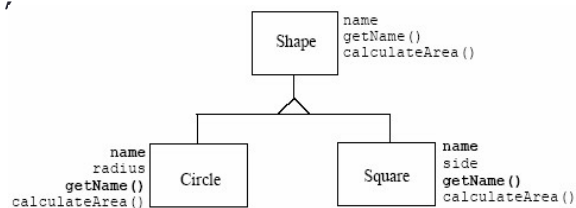
9

```

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;
    }
}

```



10

## 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;  
    }  
}
```

11

## 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

12

```

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;
    }
}

```

13

## 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

14

## 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.

15

## 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

16



## Example: private

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

17

## Outline

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

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

19

20

## 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.

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.

21

22

## 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

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;
    }
}

```

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

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

## Abstract Class

- Class ColoredPoint does not implement source code for the method plot(), hence it must be declared as abstract
- Can only create objects of the class SimpleColoredPoint.
- However, we can have:  
Point p = new SimpleColoredPoint(a, b, red); p.plot();

28

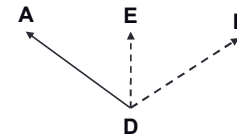
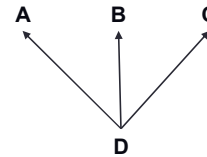
- abstract class A {  
    abstract void a();
- }
- class B extend A {
- }

## Outline

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

## Multiple and Single Inheritances

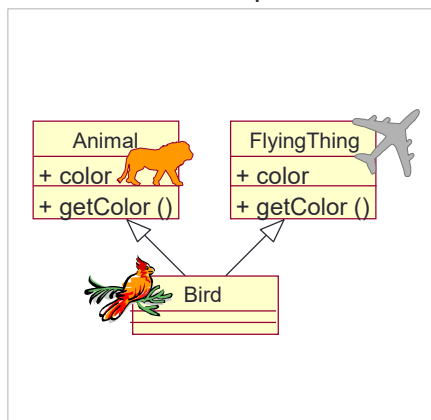
- Multiple Inheritance
  - A class can inherit several other classes
  - C++ supports multiple inheritance
- Single Inheritance
  - A class can inherit only one other class
  - Java supports only single inheritance
  - → Need to add the notion of Interface



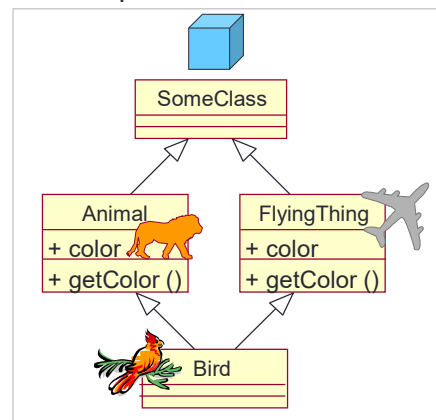
31

## 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

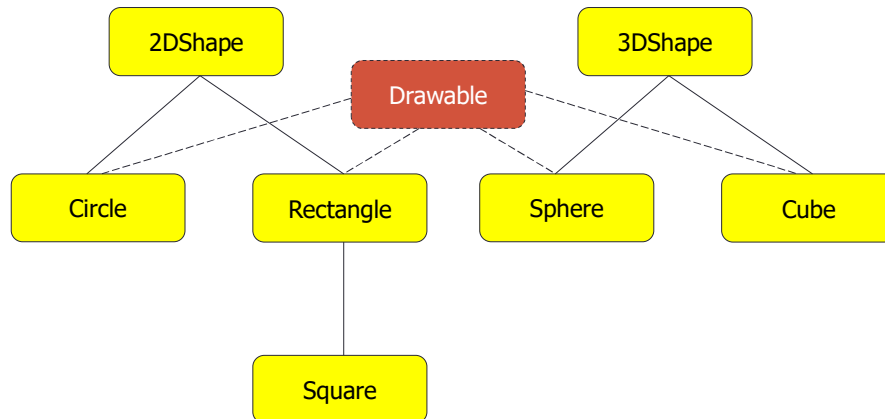
33

## Mix-in inheritance

- In this inheritance, a "class" will provide some functions in order to mix with other classes.
- A mixed class often re-uses some functions defined in the provider class but also inherits from another class.
- Is a mean that allows objects without relation in the hierarchy tree can communicate to each other.
- In Java the mix-in inheritance is done via Interface

34

## Interface



35

## Interface

- Interface: Corresponds to different implementations.
- Defines the border:
  - What How
  - Declaration and Implementation.

36

## Interface

- Interface does not implement any methods but defines the design structure in any class that uses it.
- An interface: 1 contract – in which software development teams agree on how their products communicate to each other, without knowing the details of product implementation of other teams.

37

## Example

- Class Bicycle – Class StoreKeeper:
  - StoreKeepers does not care about the characteristics what they keep, they care only the price and the id of products.
- Class AutonomousCar– GPS:
  - Car manufacturers produce cars with features: Start, Speed-up, Stop, Turn left, Turn right,..
  - GPS: Location information, Traffic status – Making decisions for controlling car
  - How does GPS control both car and space craft?

38

## Interface OperateCar

```
public interface OperateCar {  
  
    // Constant declaration– if any  
  
    // Method signature  
    int turn(Direction direction, // An enum with values RIGHT, LEFT  
            double radius, double startSpeed, double endSpeed);  
    int changeLanes(Direction direction, double startSpeed, double  
endSpeed);  
    int signalTurn(Direction direction, boolean signalOn);  
    int getRadarFront(double distanceToCar, double speedOfCar);  
    int getRadarRear(double distanceToCar, double speedOfCar);  
    .....  
    // Signatures of other methods  
}
```

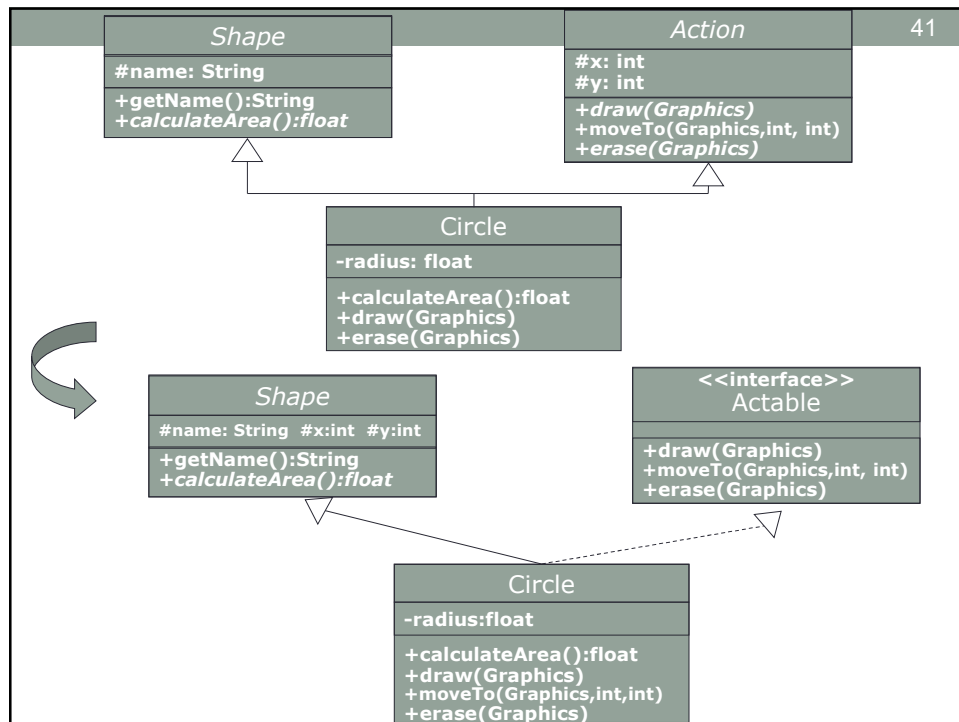
39

## Class OperateBMW760i

### // Car Manufacturer

```
public class OperateBMW760i implements OperateCar {  
  
    // cài đặt hợp đồng định nghĩa trong giao diện  
    int signalTurn(Direction direction, boolean signalOn) {  
        //code to turn BMW's LEFT turn indicator lights on  
        //code to turn BMW's LEFT turn indicator lights off  
        //code to turn BMW's RIGHT turn indicator lights on  
        //code to turn BMW's RIGHT turn indicator lights off  
    }  
  
    // Các phương thức khác, trong suốt với các clients của  
    interface  
}
```

40



41

42

## 4. Interface

- Allows a class to inherit (implement) multiple interfaces at the same time.
- Can not directly instantiate

42

## 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() { .... }  
}
```

43

44

## 4. Interface (2)

- 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

44

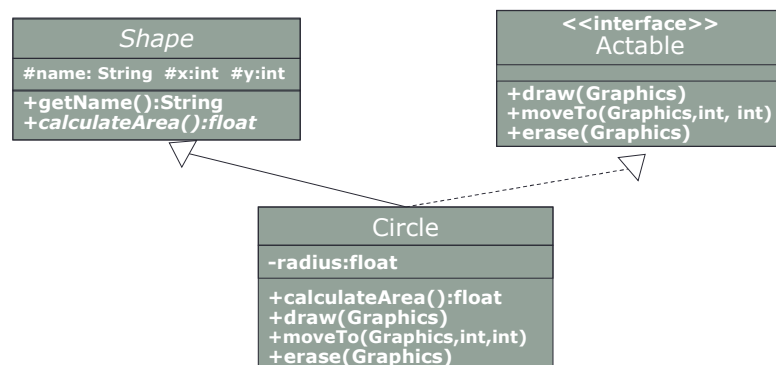
## 4. Interface (3)

- 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 {
    ...
}
```

45

## Example



46

```

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);
}

```

47

```

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...
    }
}

```

48



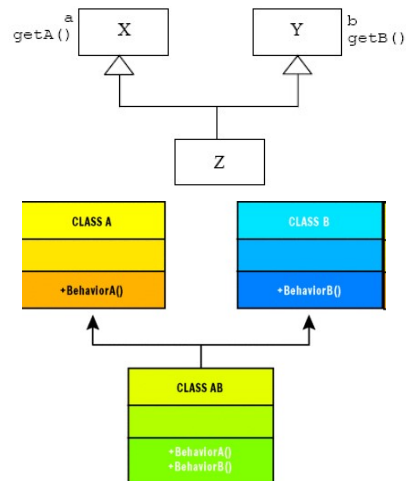
## Abstract class vs. Interface

- May or may not contain abstract methods, can contain instance methods
- Can contain protected and static methods
- Can contain final and non-final attributes
- A class can inherit only one abstract class
- Can contain only method signature
- Can contain only public functions without implementation
- Can contains only constant attributes
- A class can inherit multiple interfaces

49

## Disadvantages of Interface in solving Multiple Inheritance problems

- Does not provide a nature way for situations without inheritance conflicts
- Inheritance is to re-uses source code but Interface can not do this



50

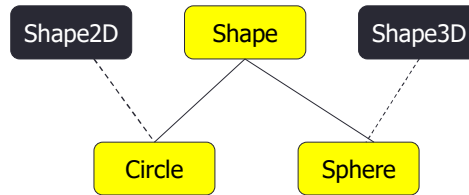
## Example

```
interface Shape2D {
    double getArea();
}
```

```
interface Shape3D {
    double getVolume();
}
```

```
class Point3D {
    double x, y, z;

    Point3D(double x, double y, double z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```



51

51/52

52

```
abstract class Shape {
    abstract void display();
}

class Circle extends Shape
implements Shape2D {
    Point3D center, p; // p is a point on
    circle

    Circle(Point3D center, Point3D p) {
        this.center = center;
        this.p = p;
    }

    public void display() {
        System.out.println("Circle");
    }

    public double getArea() {
        double dx = center.x - p.x;
        double dy = center.y - p.y;
        double d = dx * dx + dy * dy;
        double radius = Math.sqrt(d);
        return Math.PI * radius * radius;
    }
}
```

```
class Sphere extends Shape
implements Shape3D {
```

```
    Point3D center;
    double radius;
```

```
    Sphere(Point3D center, double radius) {
        this.center = center;
        this.radius = radius;
    }
```

```
    public void display() {
        System.out.println("Sphere");
    }
```

```
    public double getVolume() {
        return 4 * Math.PI * radius * radius * radius / 3;
    }
}
```

```
class Shapes {
```

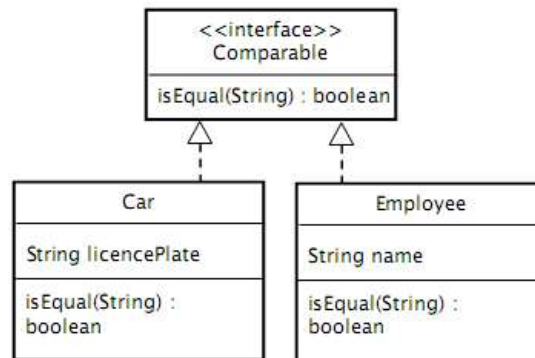
```
    public static void main(String args[]) {

        Circle c = new Circle(new Point3D(0, 0, 0), new
        Point3D(1, 0, 0));
        c.display();
        System.out.println(c.getArea());
        Sphere s = new Sphere(new Point3D(0, 0, 0), 1);
        s.display();
        System.out.println(s.getVolume());
    }
}
```

```
{ Result :
Circle
3.141592653589793
Sphere
4.1887902047863905
```

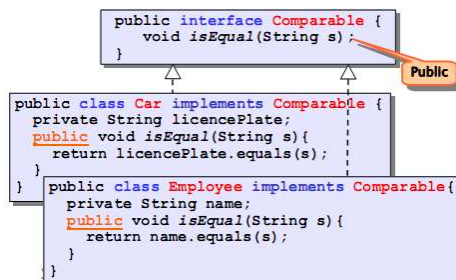
52

## interface Comparable /java.lang



53

## Application



54

## Application

```
public class Foo {  
    private Comparable objects[];  
    public Foo() {  
        objects = new Comparable[3];  
        objects[0] = new Employee();  
        objects[1] = new Car();  
        objects[2] = new Employee();  
    }  
    public Comparable find(String s) {  
        for(int i=0; i< objects.length; i++)  
            if(objects[i].isEqual(s)  
                return objects[i];  
    }  
}
```

55

56

56