OBJECT-ORIENTED LANGUAGE AND THEORY

7. ABSTRACT CLASS AND INTERFACE

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

Outline

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

Outline

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

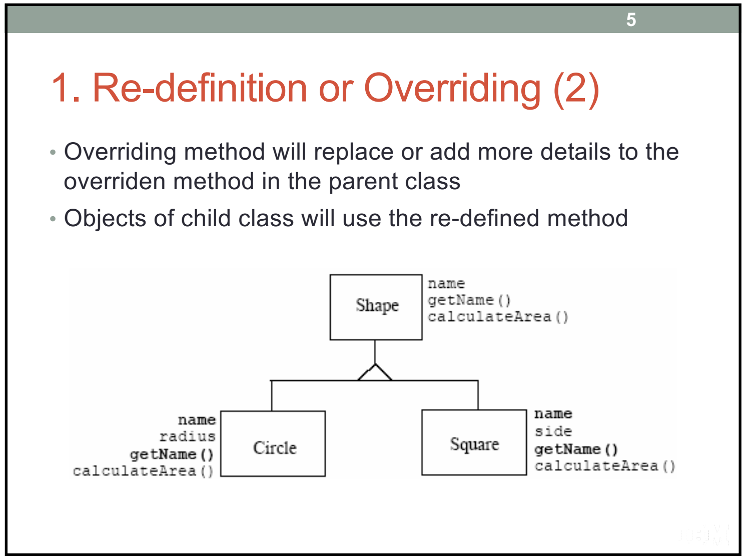
1. Re-definition or Overriding

- 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

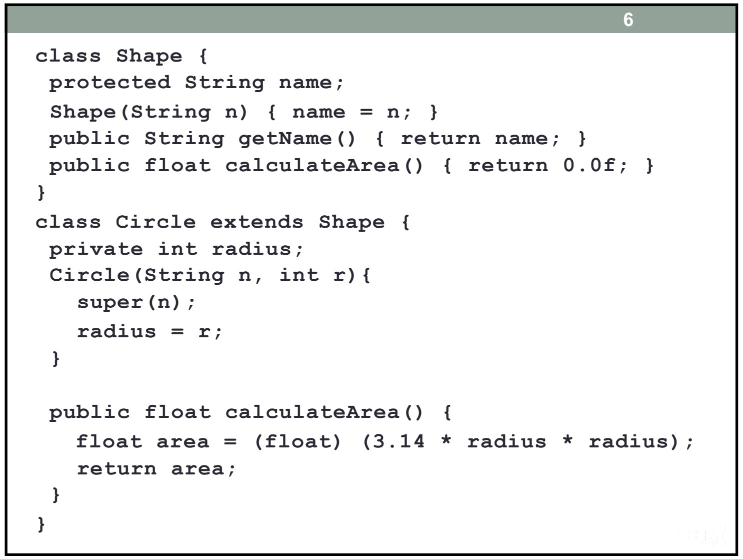
  (Method Redefine/Override)



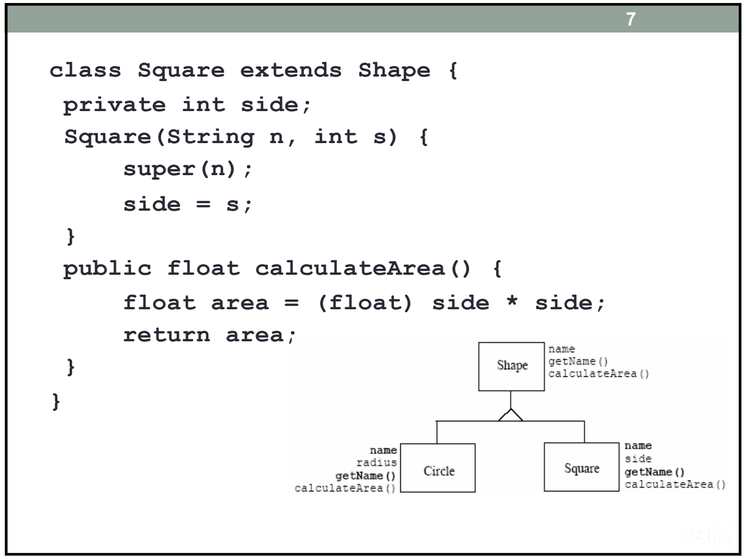
1. Re-definition or Overriding (2)

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

Shape
name
getName ()
calculateArea ()
name
radius
getName ()
calculateArea ()
Circle
Square
name
side
getName ()
calculateArea ()

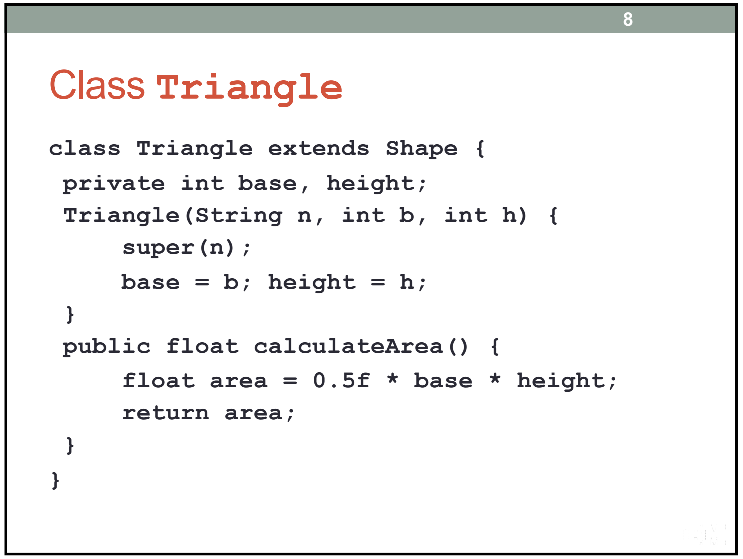


```
class Shape {
 protected String name;
 Shape(String n) { name = n; }
 public String getName() { return name; }
 public float calculateArea() { return 0.0f; }
}
class Circle extends Shape {
 private int radius;
 Circle(String n, int r){
   super(n);
   radius = r;
 }

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



```
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;
 }
}
```

Shape
name
getName ()
calculateArea ()
name
radius
getName ()
calculateArea ()
Circle
Square
name
side
getName ()
calculateArea ()



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;
 }
}
```

`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

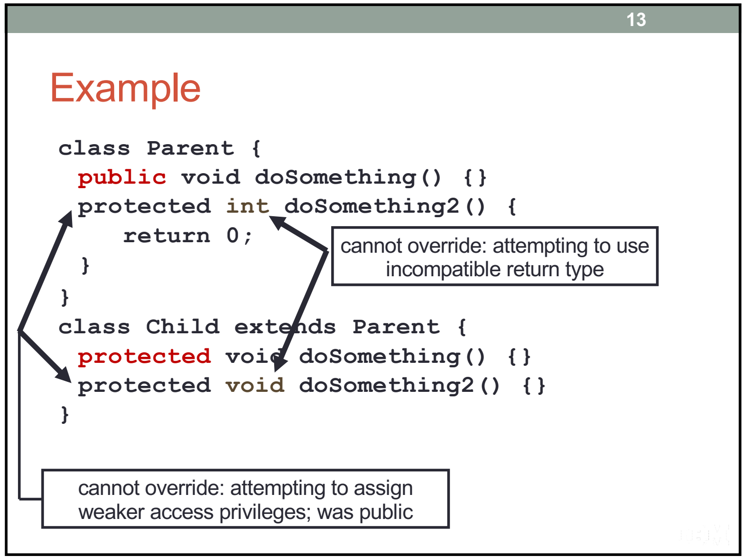
```
package abc;
public class Person {
 protected String name;
 protected int age;
 public String getDetail() {
      String s = name + "," + age;
      return s;
 }
}

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

Overriding Rules

- Overriding methods must have:
  - An argument list that is the same as the overriden method in the parent class
  - The same return data types as the overriden 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

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.

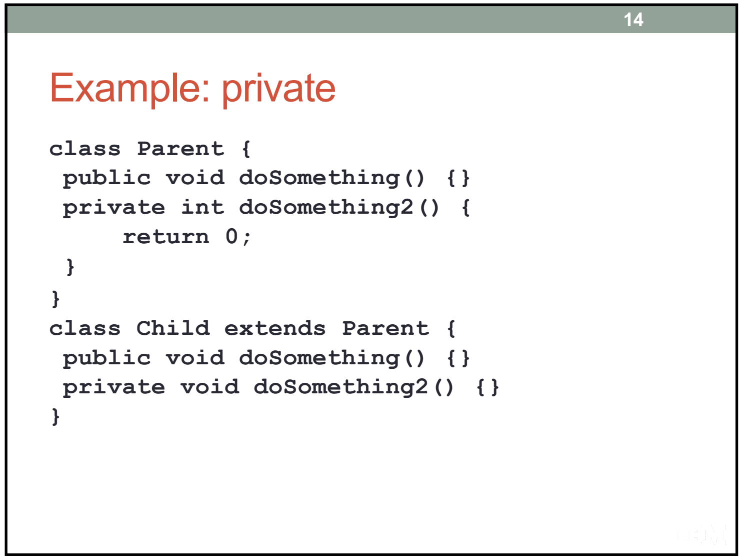


Example

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

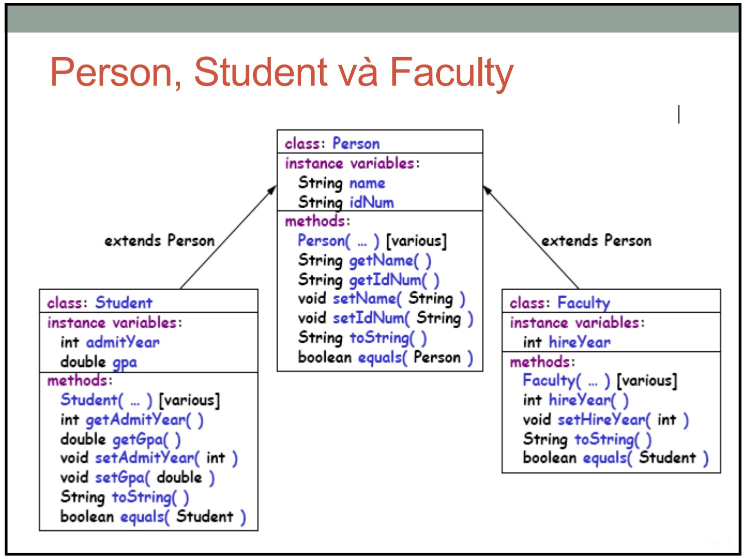
cannot override: attempting to use incompatible return type

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



Example: private

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

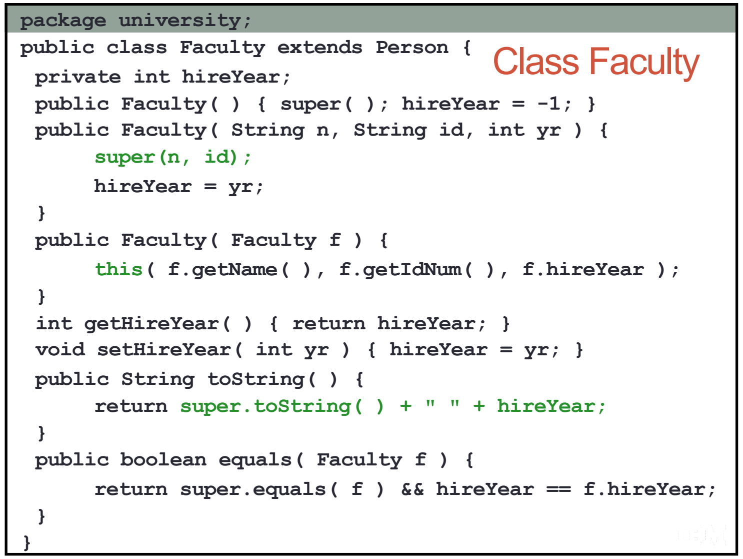


Person, Student và Faculty

class: Person
instance variables:
String name
String idNum
methods:
Person(...) [various]
String getName( )
String getIdNum( )
void setName(String)
void setIdNum(String)
String toString( )
boolean equals(Person)

class: Student
instance variables:
int admitYear
double gpa
methods:
Student(...) [various]
int getAdmitYear( )
double getGpa( )
void setAdmitYear(int)
void setGpa(double)
String toString( )
boolean equals(Student)

class: Faculty
instance variables:
int hireYear
methods:
Faculty(...) [various]
int hireYear( )
void setHireYear(int)
String toString( )
boolean equals(Student)



Class Faculty

```
package university;
public class Faculty extends Person {
 private int hireYear;
 public Faculty( ) { super( ); hireYear = -1; }
 public Faculty( String n, String id, int yr ) {
     super(n, id);
     hireYear = yr;
 }
 public Faculty( Faculty f ) {
     this( f.getName( ), f.getIdNum( ), f.hireYear );
 }
 int getHireYear( ) { return hireYear; }
 void setHireYear( int yr ) { hireYear = yr; }
 public String toString( ) {
     return super.toString( ) + " " + hireYear;
 }
 public boolean equals( Faculty f ) {
     return super.equals( f ) && hireYear == f.hireYear;
 }
}
```

Overriding

- When a derived class wants to change a function inherited from its parent class (super).

```
public class Person {
…
  public String toString( ) { … }
}
public class Student extends Person {
…
  public String toString( ) { … }
}
Student bob = new Student("Bob Goodstudent","123-45-
  6789",2004,4.0 );
System.out.println( "Bob's info: " + bob.toString( ) );
```

Re-define the method of the parent class

Calling to the method of the child class

Re-definition with final

- Sometimes we want to restrict the re-definition because of:
  - Correctness: The re-definition of a method in a derived class can lead the method to a wrong behavior
  - Efficiency: The dynamic linking mechanism is not efficient in time as the static linking mechanism. If a method should not be re-defined in derrived classes, we shold use the keyword final with the method

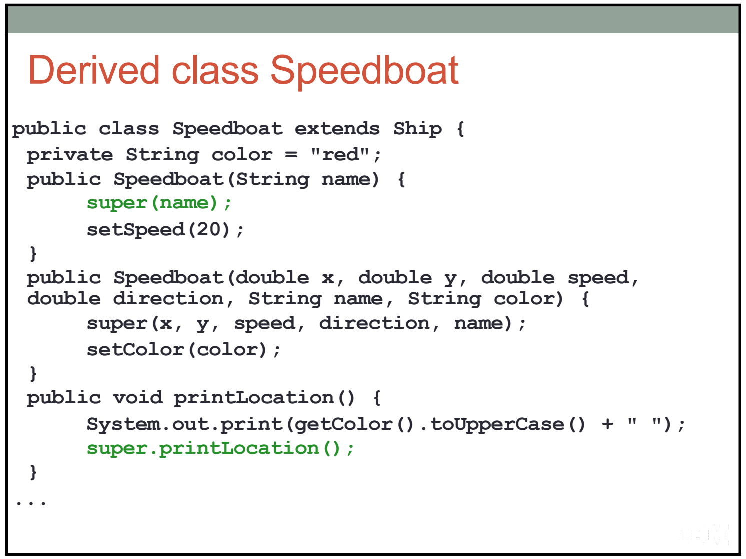
```
public final String baseName () {
return “Person”;}
}
```

Basic class ship

```
public class Ship {
  public double x=0.0, y=0.0, speed=1.0, direction=0.0;
  public String name;
  public Ship(double x, double y, double speed, double
  direction, String name) {
     this.x = x;
     this.y = y;
     this.speed = speed;
     this.direction = direction;
     this.name = name;
  }
  public Ship(String name) {
        this.name = name;
  }
  private double degreesToRadians(double degrees) {
        return(degrees * Math.PI / 180.0);
  }
  ...
```

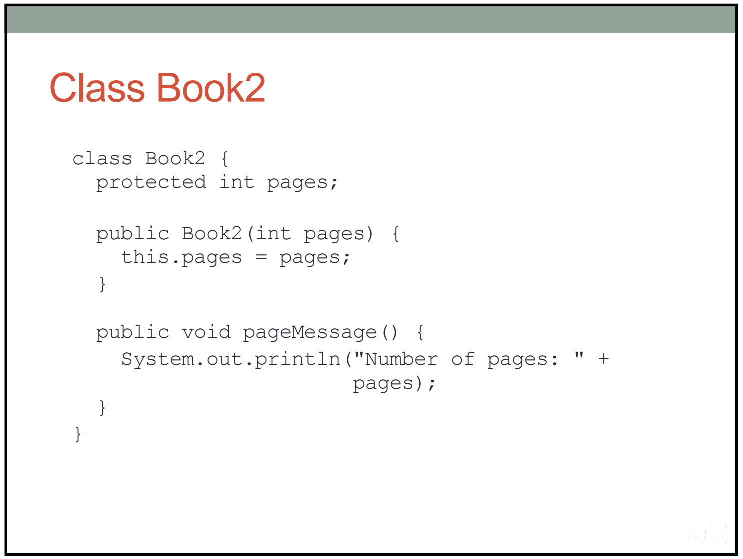
Basic class ship

```
 public void move() {
      move(1);
 }
 public void move(int steps) {
   double angle = degreesToRadians(direction);
   x = x + (double)steps * speed * Math.cos(angle);
   y = y + (double)steps * speed * Math.sin(angle);
 }
 public void printLocation() {
   System.out.println(name + " is at ("+ x + "," + y +
   ").");
 }
}
...
```



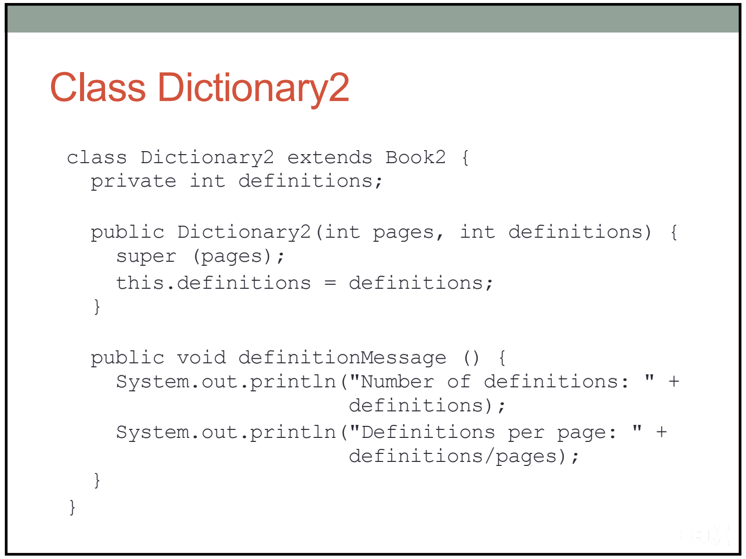
Derived class Speedboat

```
public class Speedboat extends Ship {
 private String color = "red";
 public Speedboat(String name) {
      super(name);
      setSpeed(20);
 }
 public Speedboat(double x, double y, double speed,
 double direction, String name, String color) {
      super(x, y, speed, direction, name);
      setColor(color);
 }
 public void printLocation() {
      System.out.print(getColor().toUpperCase() + " ");
      super.printLocation();
 }
...
```



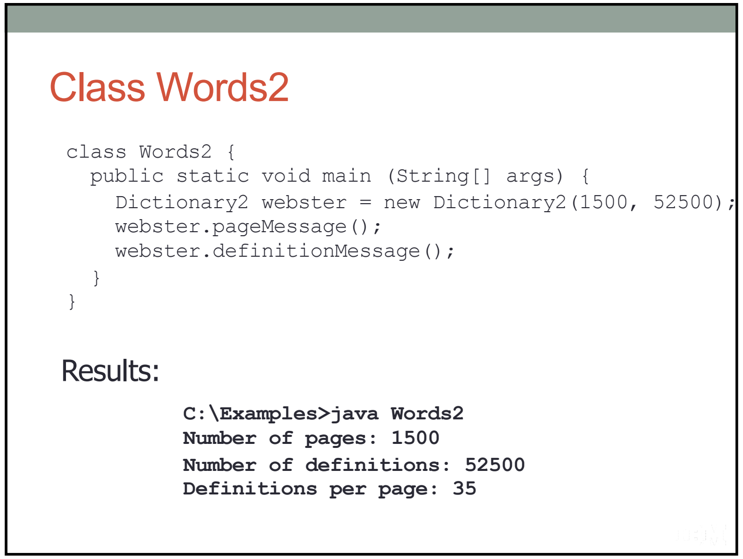
Class Book2

```
class Book2 {
  protected int pages;

  public Book2(int pages) {
    this.pages = pages;
  }

  public void pageMessage() {
    System.out.println("Number of pages: " +
                       pages);
  }
}
```



Class Dictionary2

```
class Dictionary2 extends Book2 {
  private int definitions;

  public Dictionary2(int pages, int definitions) {
    super (pages);
    this.definitions = definitions;
  }

  public void definitionMessage () {
    System.out.println("Number of definitions: " +
                       definitions);
    System.out.println("Definitions per page: " +
                       definitions/pages);
  }
}
```

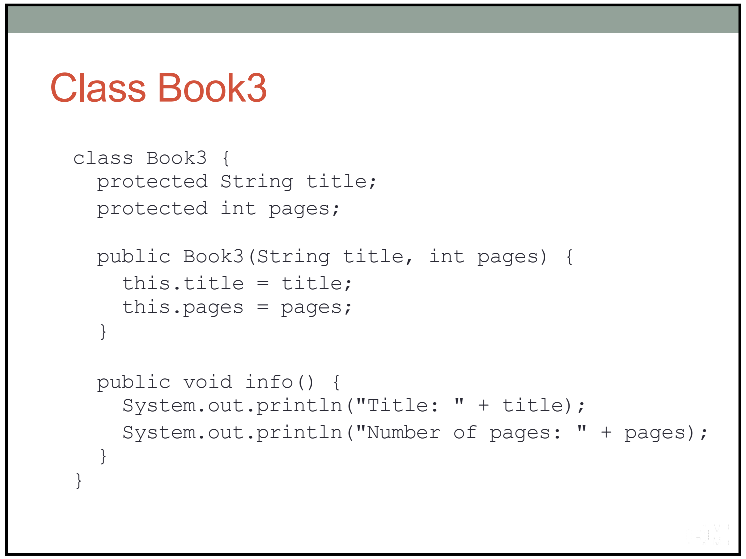


Class Words2

```
class Words2 {
  public static void main (String[] args) {
    Dictionary2 webster = new Dictionary2(1500, 52500);
    webster.pageMessage();
    webster.definitionMessage();
  }
}
```

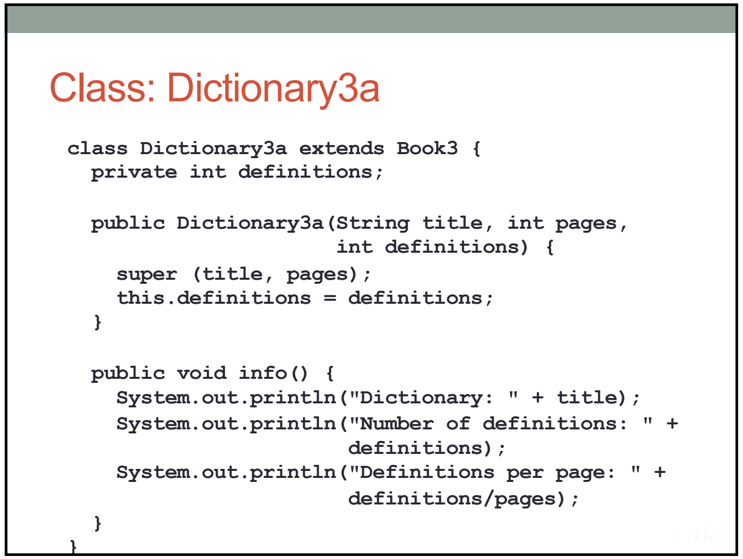
Results:

```
C:\Examples>java Words2
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35
```



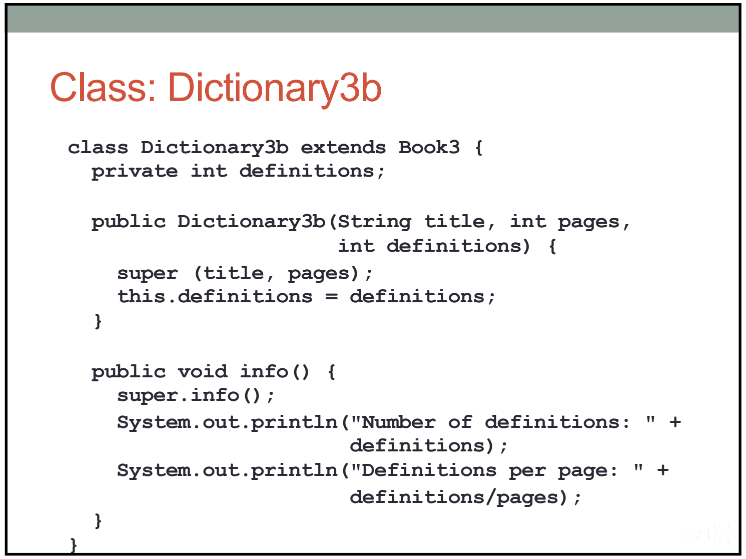
Class Book3

```
class Book3 {
  protected String title;
  protected int pages;

  public Book3(String title, int pages) {
    this.title = title;
    this.pages = pages;
  }

  public void info() {
    System.out.println("Title: " + title);
    System.out.println("Number of pages: " + pages);
  }
}
```



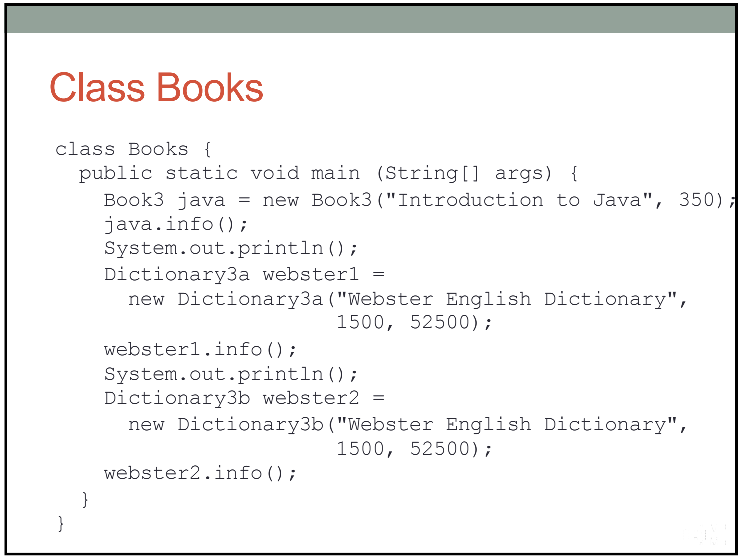
Class: Dictionary3a

```
class Dictionary3a extends Book3 {
  private int definitions;

  public Dictionary3a(String title, int pages,
                      int definitions) {
    super (title, pages);
    this.definitions = definitions;
  }

  public void info() {
    System.out.println("Dictionary: " + title);
    System.out.println("Number of definitions: " +
                     definitions);
    System.out.println("Definitions per page: " +
                     definitions/pages);
  }
}
```



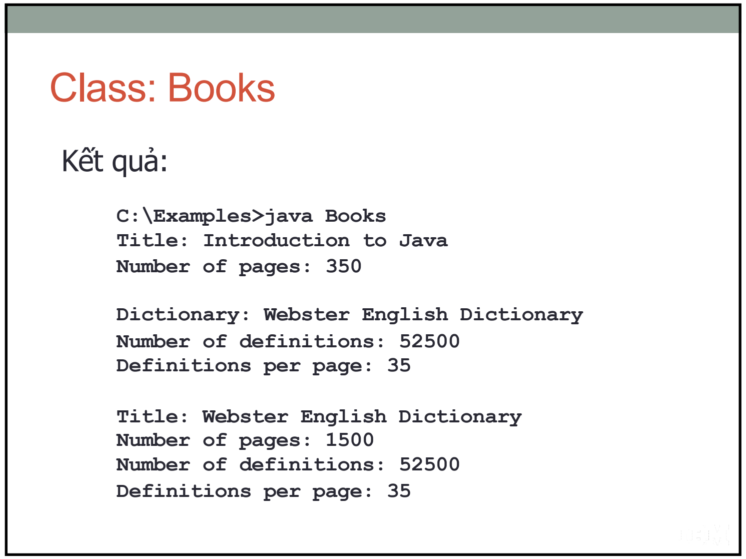
Class: Dictionary3b

```
class Dictionary3b extends Book3 {
  private int definitions;

  public Dictionary3b(String title, int pages,
                      int definitions) {
    super (title, pages);
    this.definitions = definitions;
  }

  public void info() {
    super.info();
    System.out.println("Number of definitions: " +
                     definitions);
    System.out.println("Definitions per page: " +
                     definitions/pages);
  }
}
```



Class Books

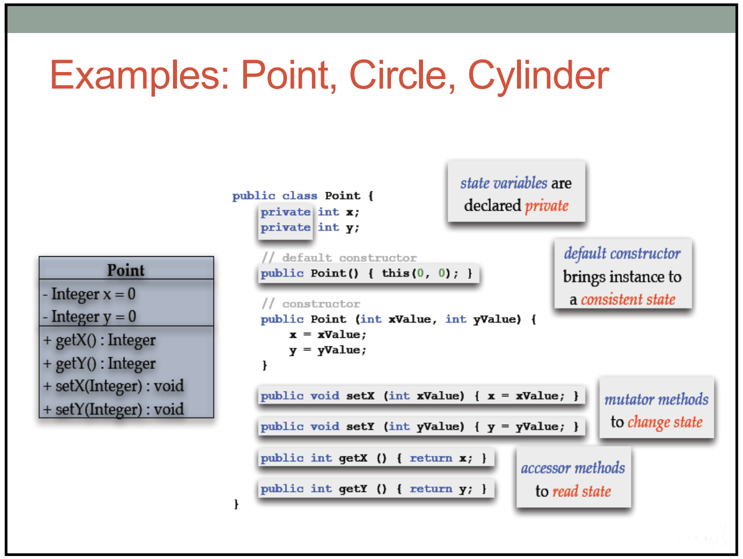
```
class Books {
  public static void main (String[] args) {
    Book3 java = new Book3("Introduction to Java", 350);
    java.info();
    System.out.println();
    Dictionary3a webster1 =
      new Dictionary3a("Webster English Dictionary",
                       1500, 52500);
    webster1.info();
    System.out.println();
    Dictionary3b webster2 =
      new Dictionary3b("Webster English Dictionary",
                       1500, 52500);
    webster2.info();
  }
}
```



Class: Books

Kết quả:

```
C:\Examples>java Books
Title: Introduction to Java
Number of pages: 350

Dictionary: Webster English Dictionary
Number of definitions: 52500
Definitions per page: 35

Title: Webster English Dictionary
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35
```



Examples: Point, Circle, Cylinder

Point
- Integer x = 0 - Integer y = 0
+ getX() : Integer + getY() : Integer + setX(Integer) : void + setY(Integer) : void

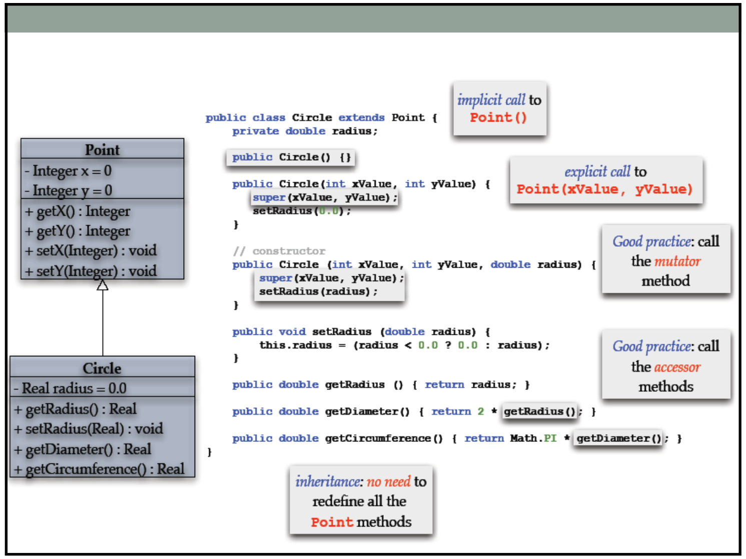
```
public class Point {
    private int x;
    private int y;

    // default constructor
    public Point() { this(0, 0); }

    // constructor
    public Point (int xValue, int yValue) {
        x = xValue;
        y = yValue;
    }

    public void setX (int xValue) { x = xValue; }

    public void setY (int yValue) { y = yValue; }

    public int getX () { return x; }

    public int getY () { return y; }
}
```

state variables are declared *private*

default constructor brings instance to a *consistent state*

mutator methods to *change state*

accessor methods to *read state*



Point
- Integer x = 0 - Integer y = 0
+ getX() : Integer + getY() : Integer + setX(Integer) : void + setY(Integer) : void

Circle
- Real radius = 0.0
+ getRadius() : Real + setRadius(Real) : void + getDiameter() : Real + getCircumference() : Real

```
public class Circle extends Point {
    private double radius;

    public Circle() {}

    public Circle(int xValue, int yValue) {
      super(xValue, yValue);
      setRadius(0.0);
    }

    // constructor
    public Circle (int xValue, int yValue, double radius) {
        super(xValue, yValue);
        setRadius(radius);
    }

    public void setRadius (double radius) {
        this.radius = (radius < 0.0 ? 0.0 : radius);
    }

    public double getRadius () { return radius; }

    public double getDiameter() { return 2 * getRadius(); }

    public double getCircumference() { return Math.PI * getDiameter(); }
}
```

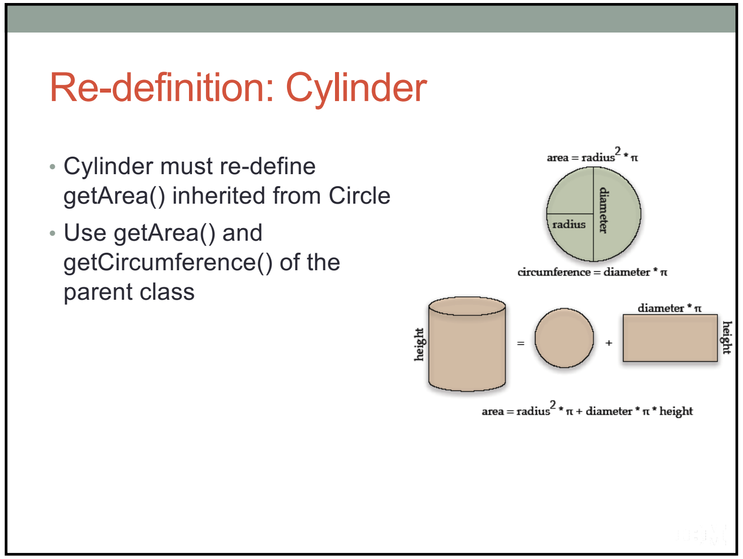
implicit call to `Point()`

explicit call to `Point(xValue, yValue)`

Good practice: call the *mutator* method

Good practice: call the *accessor* methods

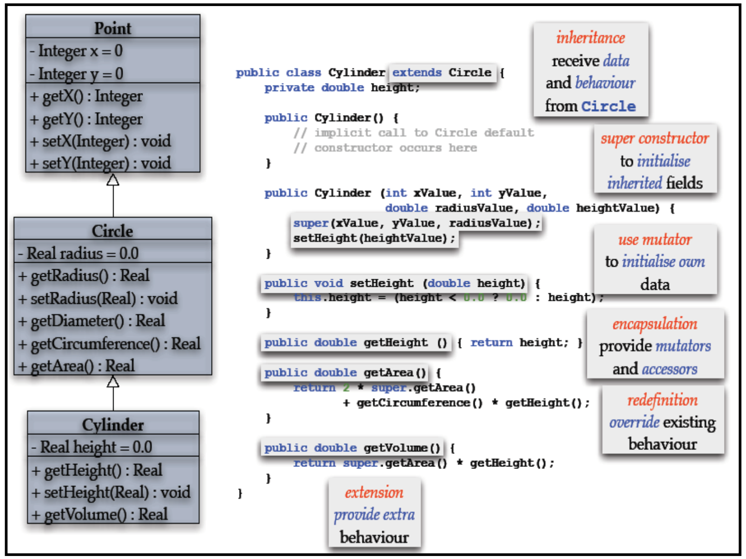
inheritance: *no need* to redefine all the `Point` methods



Re-definition: Cylinder

- Cylinder must re-define getArea() inherited from Circle
- Use getArea() and getCircumference() of the parent class

area = radius² * π
radius
diameter
circumference = diameter * π
height
=
+
diameter * π
height
area = radius² * π + diameter * π * height



Point
- Integer x = 0
- Integer y = 0
+ getX() : Integer
+ getY() : Integer
+ setX(Integer) : void
+ setY(Integer) : void

Circle
- Real radius = 0.0
+ getRadius() : Real
+ setRadius(Real) : void
+ getDiameter() : Real
+ getCircumference() : Real
+ getArea() : Real

Cylinder
- Real height = 0.0
+ getHeight() : Real
+ setHeight(Real) : void
+ getVolume() : Real

```
public class Cylinder extends Circle {
    private double height;

    public Cylinder() {
        // implicit call to Circle default
        // constructor occurs here
    }

    public Cylinder (int xValue, int yValue,
                     double radiusValue, double heightValue) {
        super(xValue, yValue, radiusValue);
        setHeight(heightValue);
    }

    public void setHeight (double height) {
        this.height = (height < 0.0 ? 0.0 : height);
    }

    public double getHeight () { return height; }

    public double getArea() {
        return 2 * super.getArea()
               + getCircumference() * getHeight();
    }

    public double getVolume() {
        return super.getArea() * getHeight();
    }
}
```

inheritance receive *data* and *behaviour* from Circle

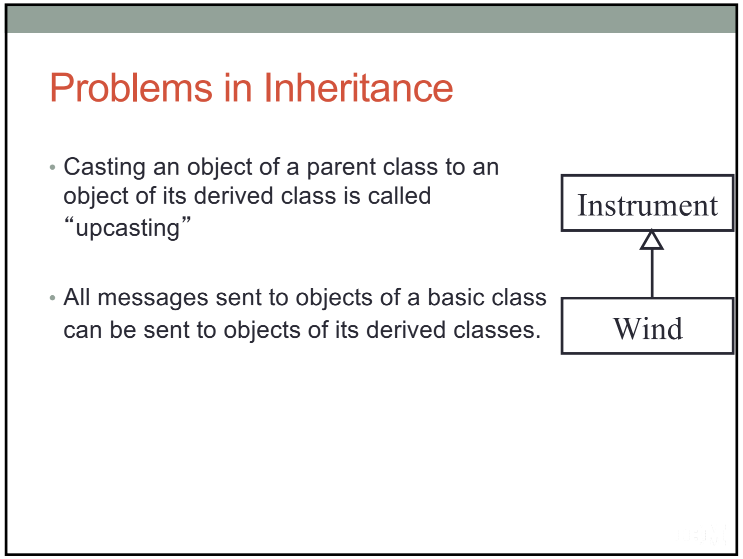
super constructor to *initialise inherited* fields

use mutator to *initialise own* data

encapsulation provide *mutators* and *accessors*

redefinition *override* existing behaviour

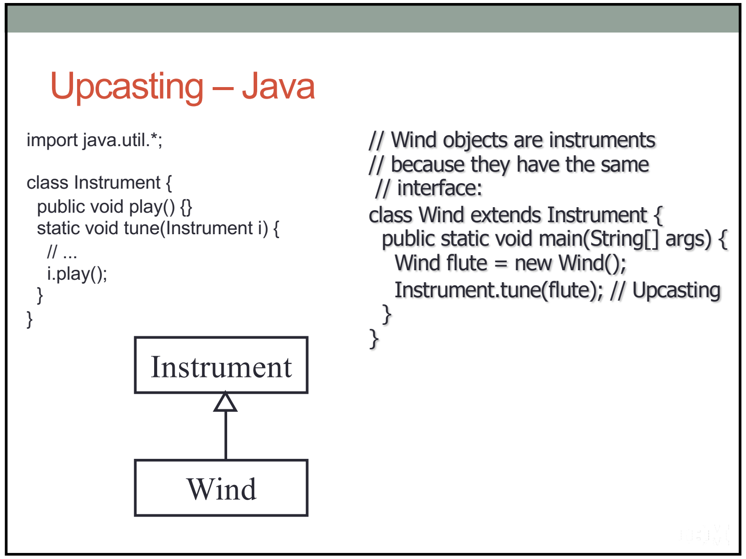
extension *provide extra* behaviour



Problems in Inheritance

- Casting an object of a parent class to an object of its derived class is called "upcasting"
- All messages sent to objects of a basic class can be sent to objects of its derived classes.

Instrument
Wind

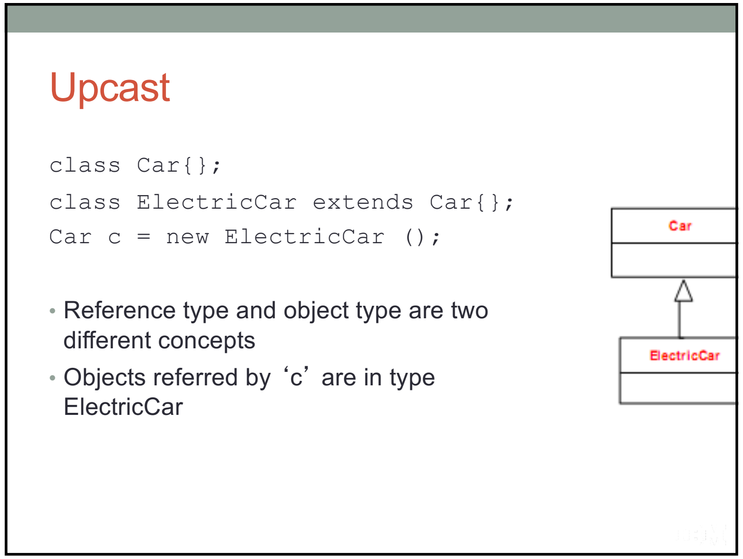


Upcasting – Java

```
import java.util.*;

class Instrument {
  public void play() {}
  static void tune(Instrument i) {
    // ...
    i.play();
  }
}
```

```
// Wind objects are instruments
// because they have the same
 // interface:
class Wind extends Instrument {
  public static void main(String[] args) {
    Wind flute = new Wind();
    Instrument.tune(flute); // Upcasting
  }
}
```

Instrument
Wind



Upcast

```
class Car{};
class ElectricCar extends Car{};
Car c = new ElectricCar ();
```

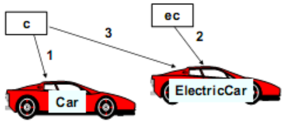
- Reference type and object type are two different concepts
- Objects referred by 'c' are in type ElectricCar

Car
ElectricCar

Upcast

- Car c = new Car();
- ElectricCar ec = new ElectricCar ();
- c = ec;

- Automatic upcast (implicit)
- Types of objects do not be changed



Down Cast

- When assigning an object of a more basic type to a derived type.
- Be careful in usage
- Do it explicitly

```
Car c = new ElectricCar(); // Up-casting not
 explicitly
c.recharge(); // Error
// Down-casting explicitly
ElectricCar ec = (ElectricCar)c;
ec.recharge(); // ok
```

Down Cast

Car c = new Car();
c.recharge(); // lỗi
// explicit downcast
ElectricCar ec = (ElectricCar)c;
ec.recharge(); // lỗi

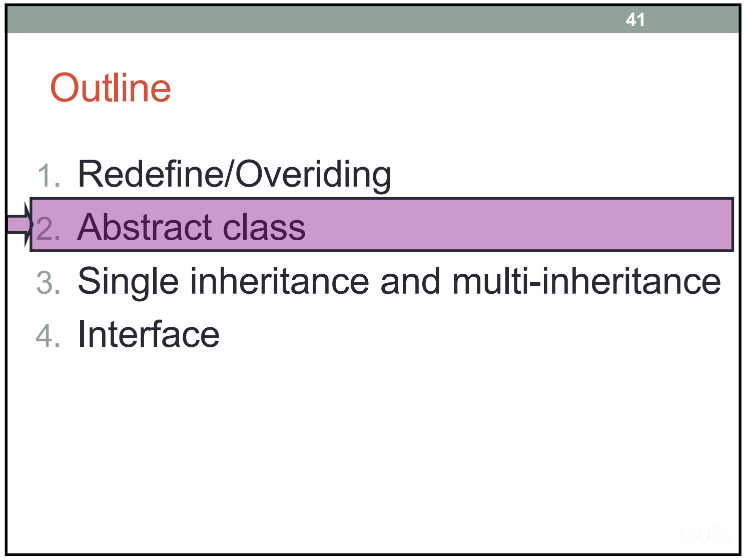
Runtime Error

Avoiding Down Cast Error

- Using instanceof operation

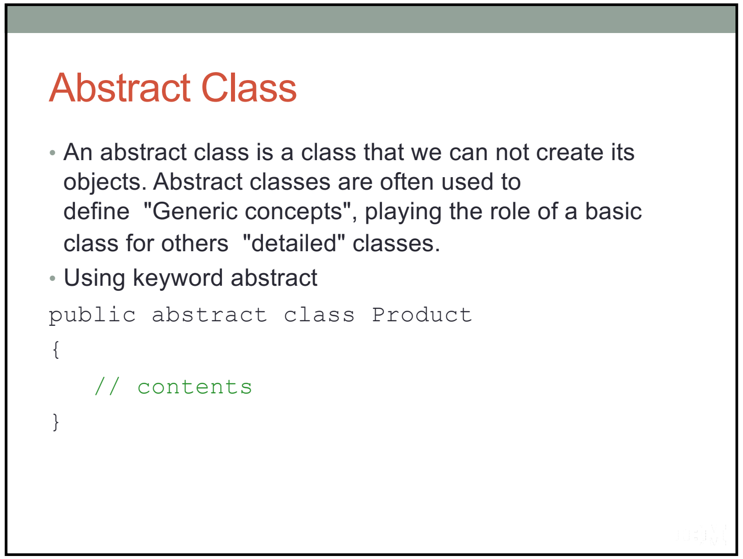
```
Car c = new Car();
ElectricCar ec;

if (c instanceof ElectricCar ){
 ec = (ElectricCar) c;
 ec.recharge();
}
```

((ElectricCar)c).recharge();



Outline

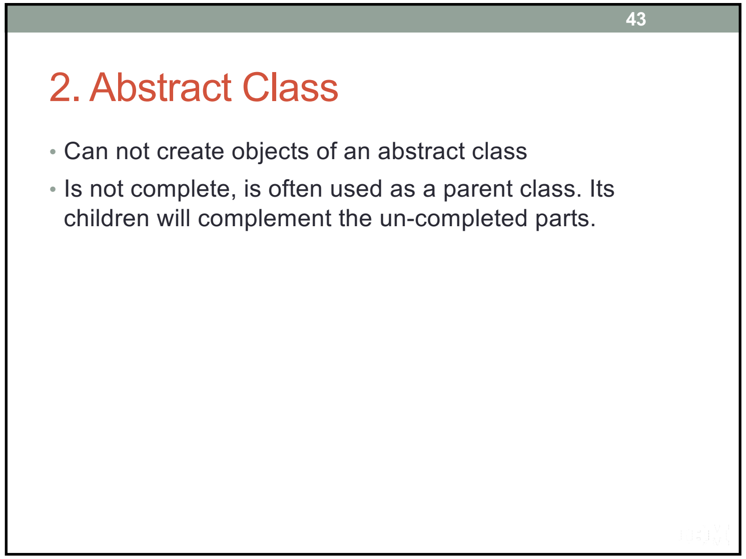
1. Redefine/Overiding
2. Abstract class
3. Single inheritance and multi-inheritance
4. Interface



Abstract Class

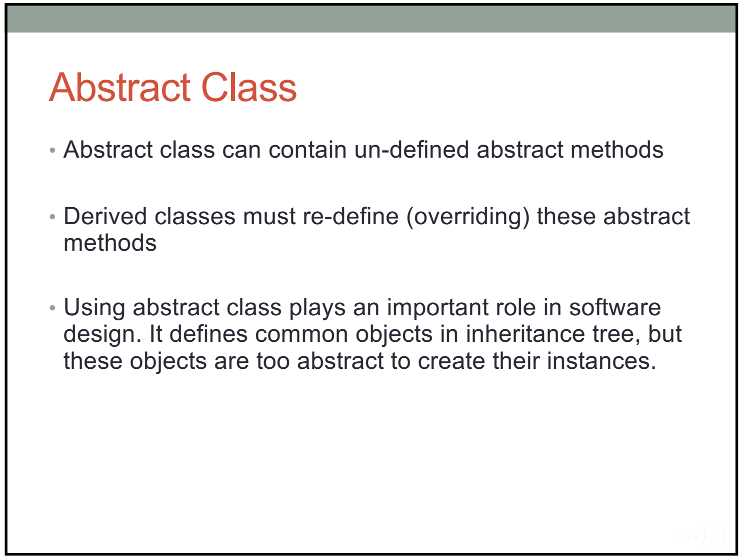
- An abstract class is a class that we can not create its objects. Abstract classes are often used to define "Generic concepts", playing the role of a basic class for others "detailed" classes.
- Using keyword abstract

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



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.



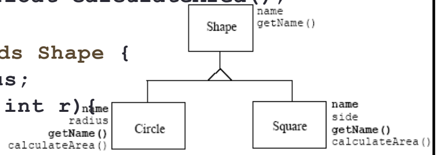
Abstract Class

- 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.

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

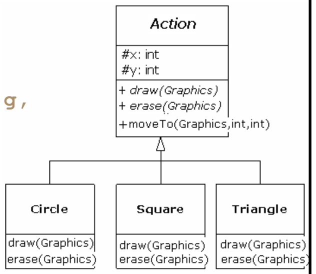
```
abstract class Shape {
  protected String name;
  Shape(String n) { name = n; }
  public String getName() { return name; }
  public abstract float calculateArea();
}
class Circle extends Shape {
  private int radius;
  Circle(String n, int r){
    super(n);
    radius = r;
  }

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



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

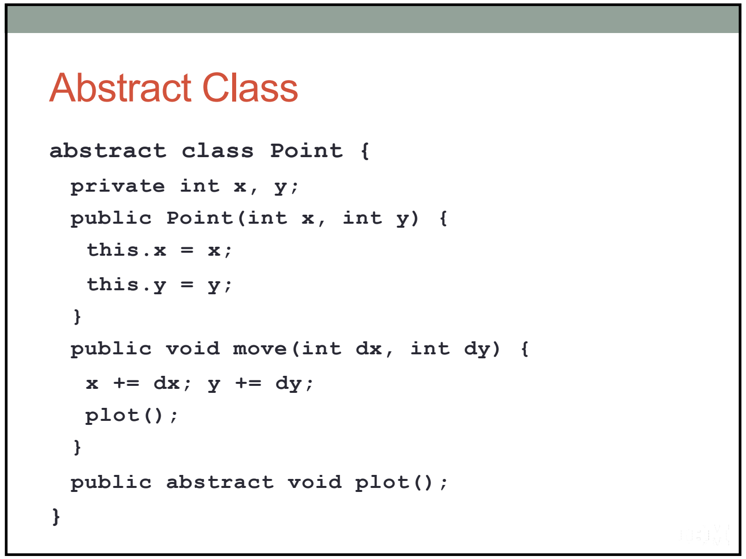
Example of abstract class

```
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);
  }

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



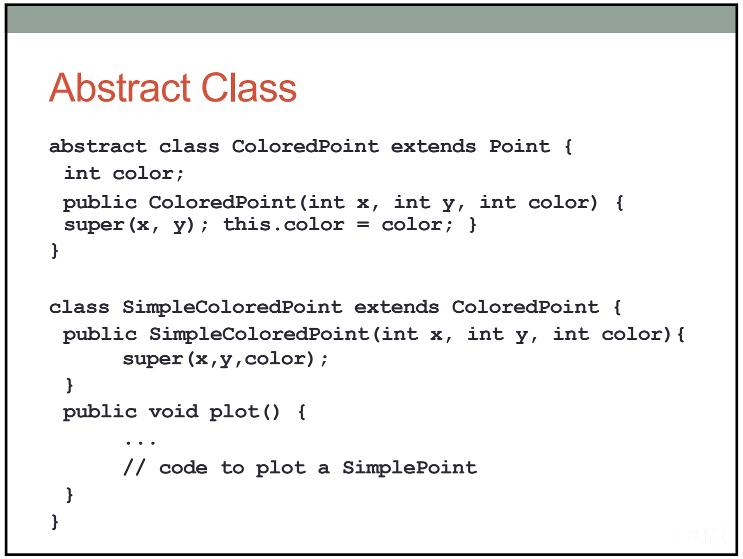
Example of abstract class (2)

```
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...
  }
}
```



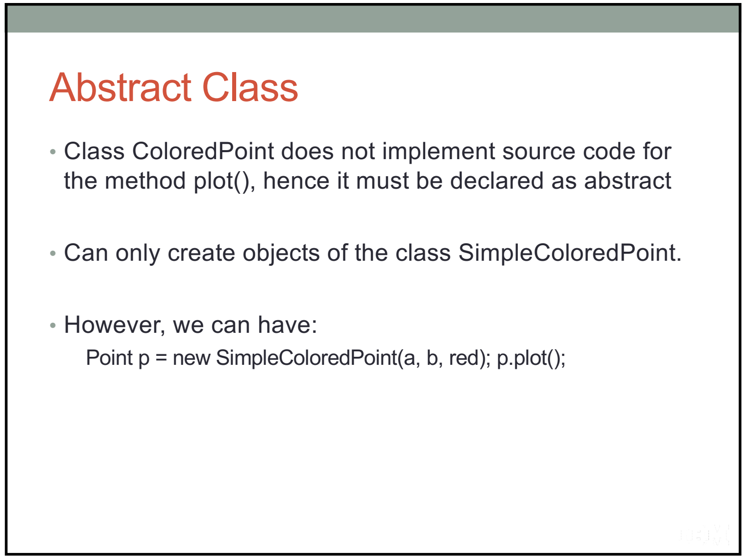
Abstract Class

```
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();
}
```



Abstract Class

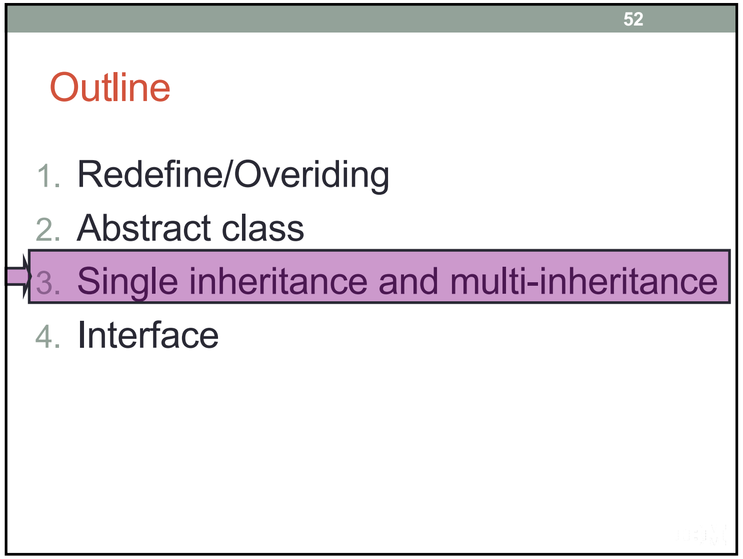
```
abstract class ColoredPoint extends Point {
 int color;
 public ColoredPoint(int x, int y, int color) {
 super(x, y); this.color = color; }
}

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



Abstract Class

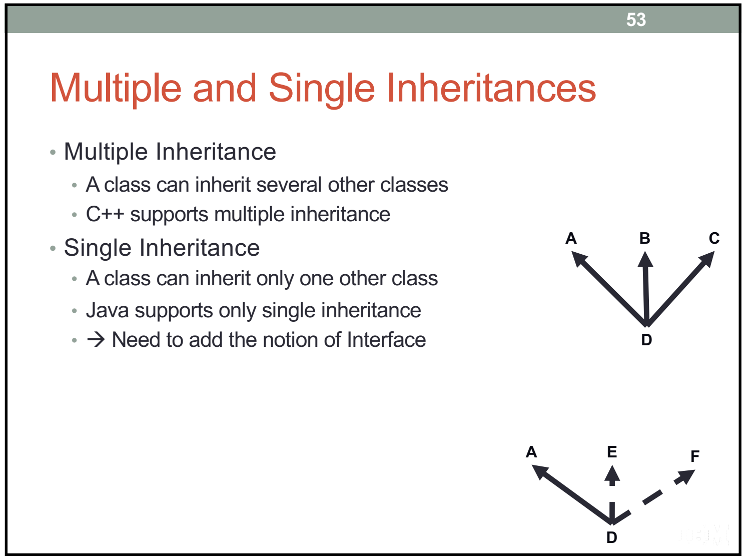
- Class ColoredPoint does not implement source code for the method plot(), hence it must be declared as abstract
- Can only create objects of the class SimpleColoredPoint.
- However, we can have:

  Point p = new SimpleColoredPoint(a, b, red); p.plot();



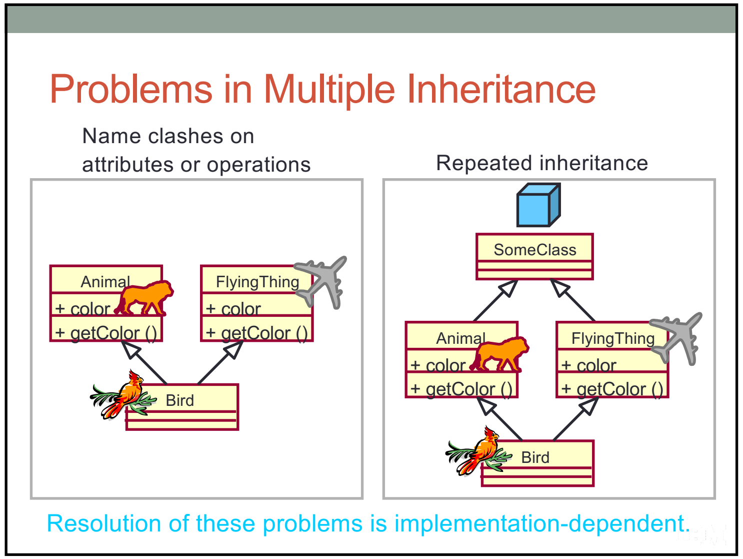
Outline

1. Redefine/Overiding
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

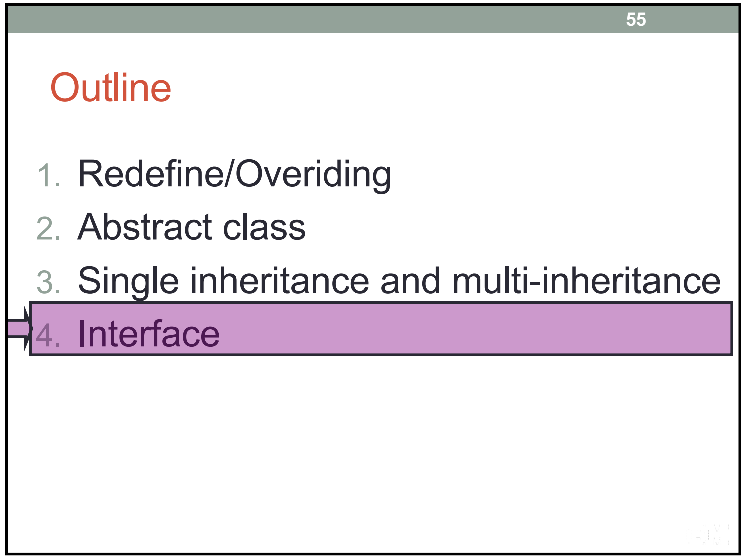


Problems in Multiple Inheritance

Name clashes on attributes or operations

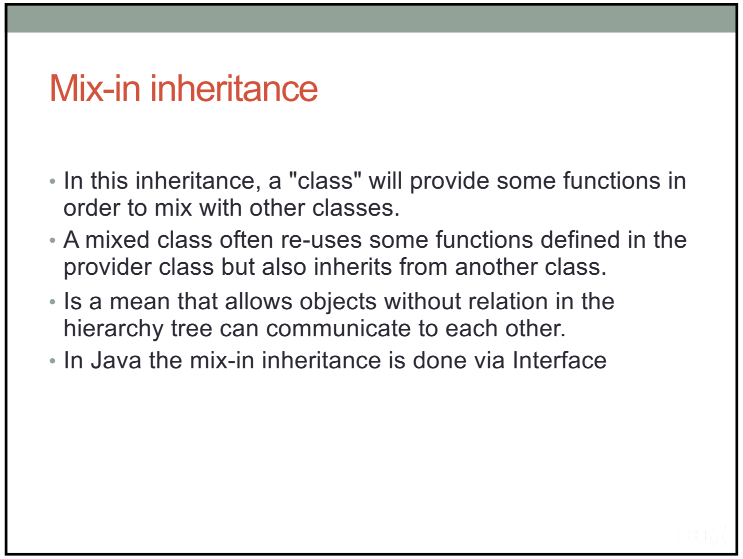
Repeated inheritance

Resolution of these problems is implementation-dependent.



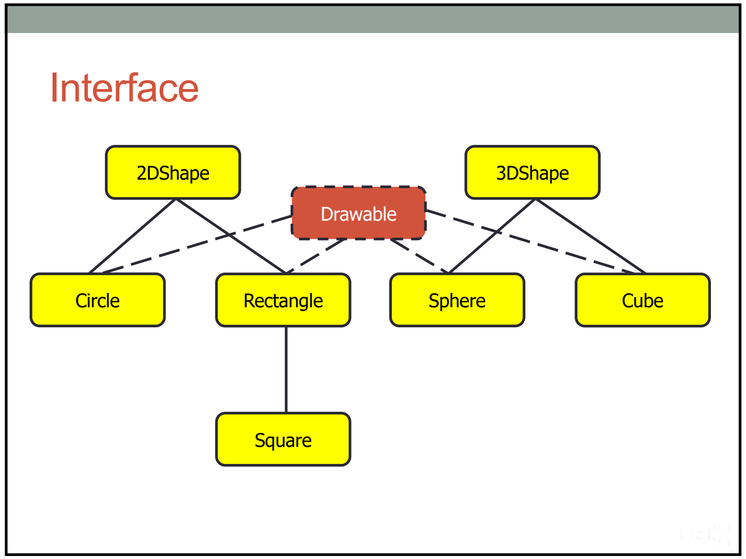
Outline

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



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



Interface

2DShape

3DShape

Drawable

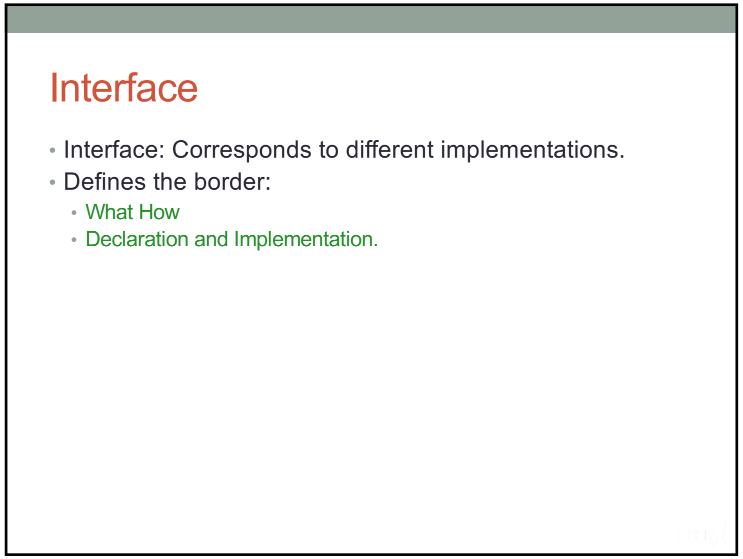
Circle

Rectangle

Sphere

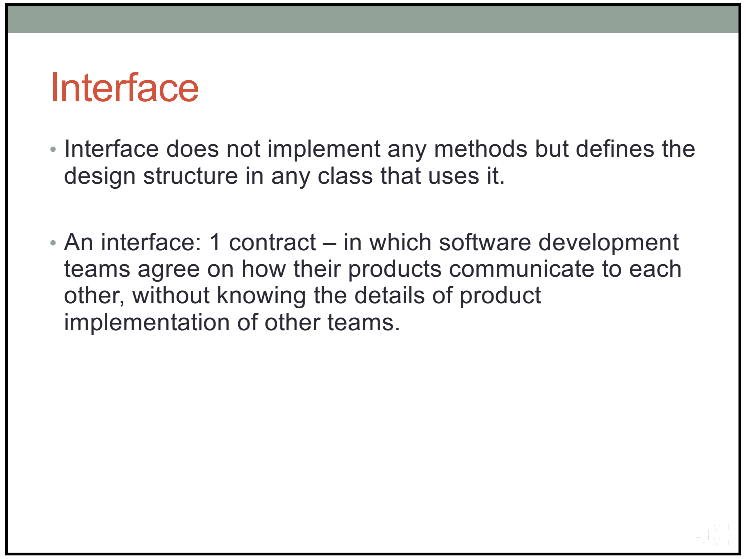
Cube

Square



Interface

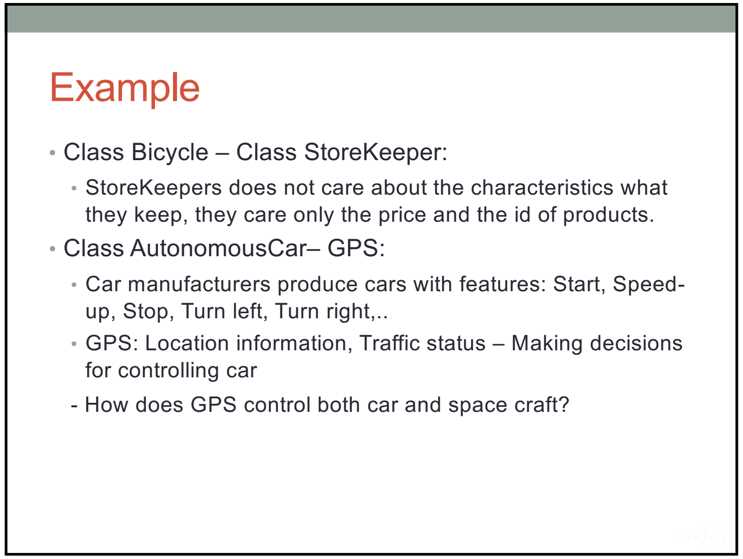
- Interface: Corresponds to different implementations.
- Defines the border:
  - What How
  - Declaration and Implementation.



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.



Example

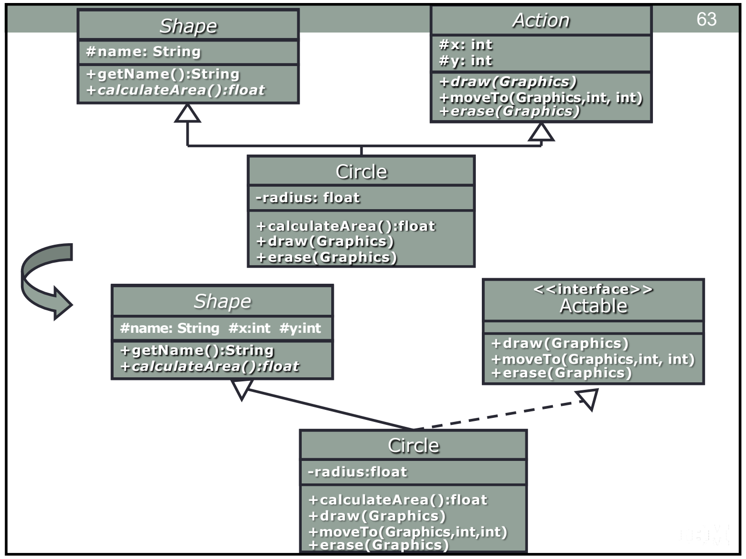
- Class Bicycle – Class StoreKeeper:
  - StoreKeepers does not care about the characteristics what they keep, they care only the price and the id of products.
- Class AutonomousCar– GPS:
  - Car manufacturers produce cars with features: Start, Speed-up, Stop, Turn left, Turn right,..
  - GPS: Location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

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
}
```

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
  }
```



Shape
#name: String
+getName():String
+calculateArea():float

Action
#x: int
#y: int
+draw(Graphics)
+moveTo(Graphics,int, int)
+erase(Graphics)

Circle
-radius: float
+calculateArea():float
+draw(Graphics)
+erase(Graphics)

Shape
#name: String #x:int #y:int
+getName():String
+calculateArea():float

<<interface>>
Actable
+draw(Graphics)
+moveTo(Graphics,int, int)
+erase(Graphics)

Circle
-radius:float
+calculateArea():float
+draw(Graphics)
+moveTo(Graphics,int,int)
+erase(Graphics)

4. Interface

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

Interface – Technique view (JAVA)

- An interface can be considered as a “class” that
  - Its methods and attributes are not explicitly public
  - Its attributes are static and final
  - Its methods are abstract

4. Interface (2)

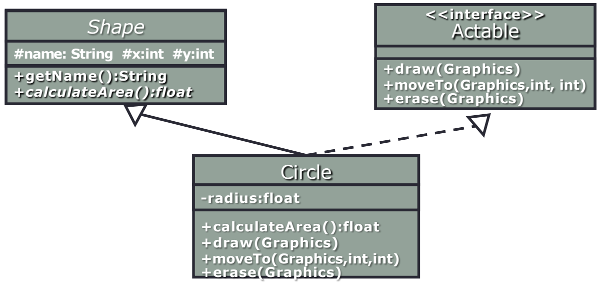
- To become an interface, we need
  - To use interface keyword to define
  - To write only:
    - method signature
    - static & final attributes
- Implementation class of interface
  - Either abstract class
  - Or must implement all the methods of the interface.

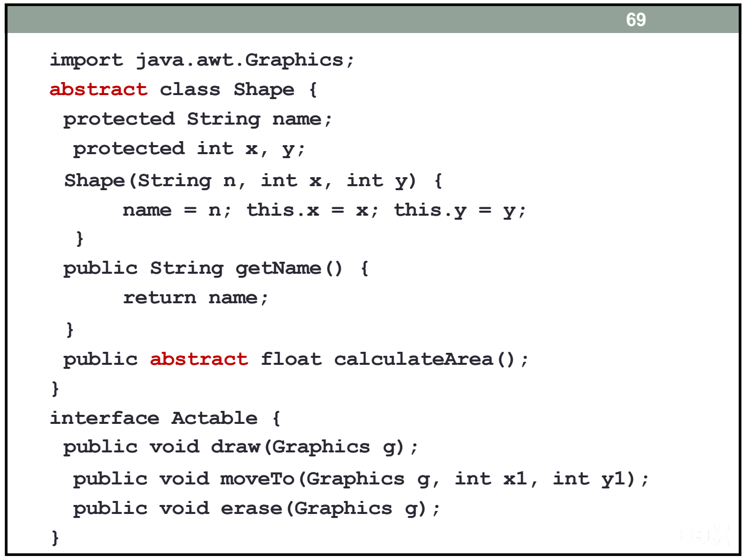
4. Interface (3)

- Java syntax:
  - **SubClas *extends* SuperClass *implements* ListOfIntefaces**
  - **SubInterface *extends* SuperInterface**
- Example:

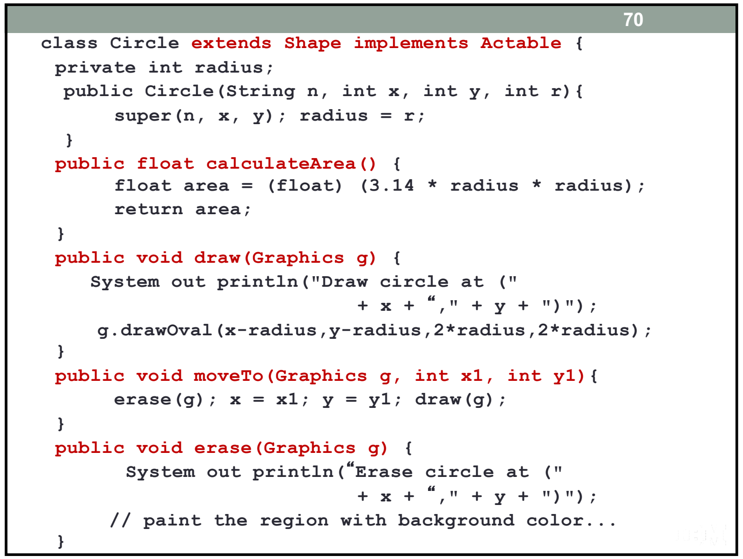
```
public interface Symmetrical {…}
public interface Movable {…}
public class Square extends Shape
             implements Symmetrical, Movable {
  ...
}
```

Example

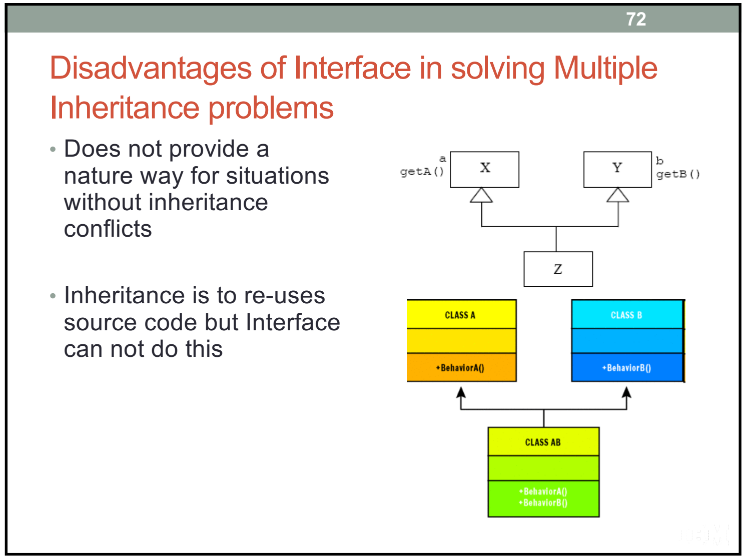




```
import java.awt.Graphics;
abstract class Shape {
 protected String name;
  protected int x, y;
 Shape(String n, int x, int y) {
      name = n; this.x = x; this.y = y;
  }
 public String getName() {
      return name;
 }
 public abstract float calculateArea();
}
interface Actable {
 public void draw(Graphics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



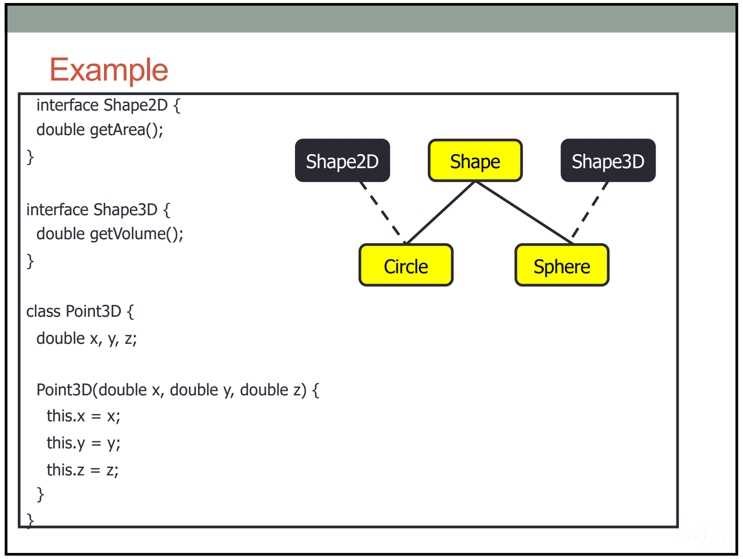
```
class Circle extends Shape implements Actable {
 private int radius;
  public Circle(String n, int x, int y, int r){
      super(n, x, y); radius = r;
  }
 public float calculateArea() {
      float area = (float) (3.14 * radius * radius);
      return area;
 }
 public void draw(Graphics g) {
    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...
 }
```



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

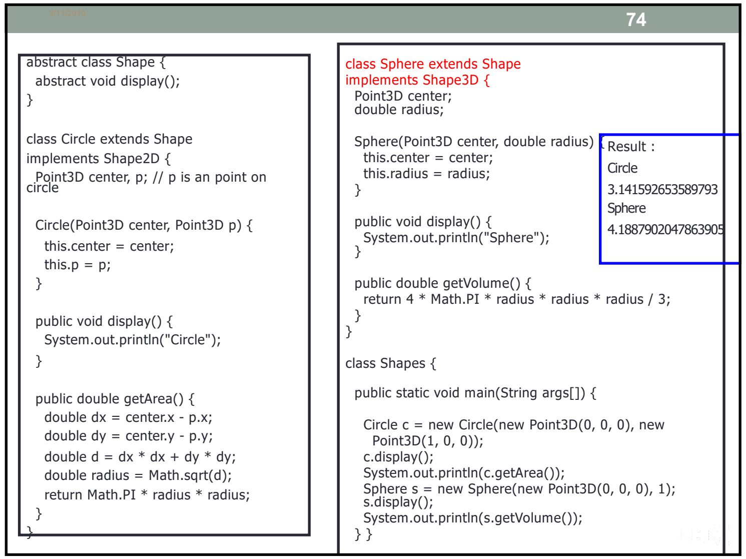
a
getA()
X
Y
b
getB()
Z
CLASS A
+BehaviorA()
CLASS B
+BehaviorB()
CLASS AB
+BehaviorA()
+BehaviorB()



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;
  }
}
```

Shape2D
Shape
Shape3D
Circle
Sphere

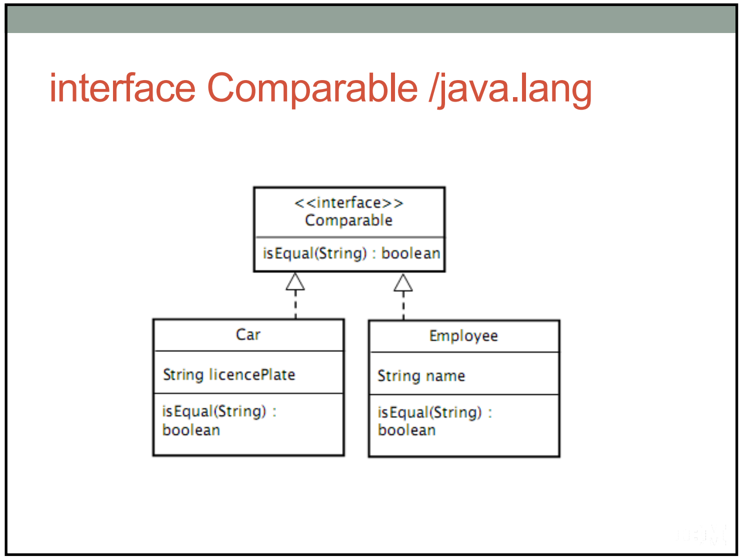


3/11/2010 74

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

class Circle extends Shape
implements Shape2D {
  Point3D center, p; // p is an 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;
  }
}
```

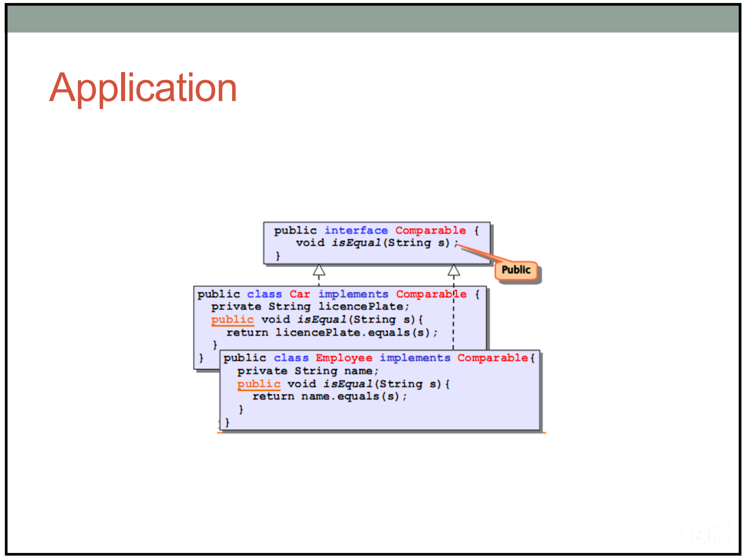
```
class Sphere extends Shape
implements Shape3D {
  Point3D center;
  double radius;

  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
```



interface Comparable /java.lang

<<interface>>
Comparable
isEqual(String) : boolean
Car
String licencePlate
isEqual(String) : boolean
Employee
String name
isEqual(String) : boolean



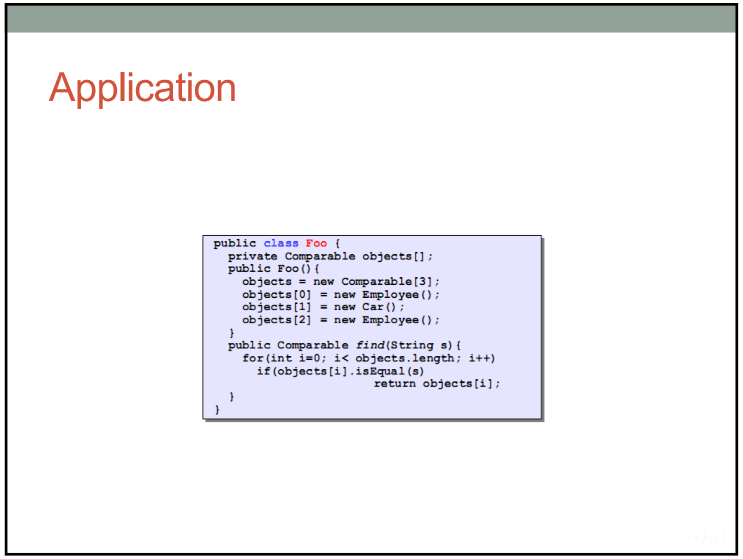
Application

```
public interface Comparable {
   void isEqual(String s);
}
```

Public

```
public class Car implements Comparable {
  private String licencePlate;
  public void isEqual(String s){
    return licencePlate.equals(s);
  }
}
```

```
public class Employee implements Comparable{
  private String name;
  public void isEqual(String s){
    return name.equals(s);
  }
}
```



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];
  }
}
```