Computer Science for Engineers

Exercise 7

Abstract classes. Interfaces

Prof. Dr. Dr.-Ing. J. Ovtcharova
Dipl. Wi.-Ing. Dan Gutu
16th of January 2009
public class A {
    protected int i = 1;
    public int f() {
        return i;
    }
}

public class B extends A {
    private int i = 2;
    public int f() {
        return -i;
    }
}

A a = new A();
System.out.println(a.i); // refers to A.i: 1
System.out.println(a.f()); // refers to A.f(): 1

B b = new B();
System.out.println(b.i); // refers to B.i: 2
System.out.println(b.f()); // refers to B.f(): -2

A aa = (A)b;
System.out.println(aa.i); // instance refers to A.i: 1
System.out.println(aa.f()); // Refers to B.f(): -2
Calling Hidden Methods (2)

class A {
    int i = 1;
    int f() { return i; }
}

class B extends A {
    int i; // this variable hides i in A
    int f() { // this method overwrites f() in A
        i = super.i + 1; // A.i is addressed like this
        return super.f() + i; // A.f() is called like this
    }
}

2b.i
B.f()
1a.i
A.f()
super.f()
a instance of A
b instance of B
public class ExpensiveCarpet {
    private String location = "Japan";
    public void productionInfo() {
        System.out.println("manufactured in " + location);
    }
}

public class Carpet extends ExpensiveCarpet {
    protected String location = "China";
    public void productionInfo() {
        System.out.println("produced in " + location);
    }
}

Carpet oneCarpet = new Carpet();
System.out.println(oneCarpet.location);
oneCarpet.productionInfo();

ExpendiveCarpet anExpensiveCarpet = new ExpensiveCarpet();
System.out.println(anExpensiveCarpet.location);
anExpensiveCarpet.productionInfo();

ExpensiveCarpet aCarpet = (ExpensiveCarpet) oneCarpet;
System.out.println(aCarpet.location);
aCarpet.productionInfo();
Method overloading

public class Circle {
    public double x, y, r;

    // Constructors and methods
    // ....

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

    public void setCenter(Point p) {
        c.x = p.getX();
        c.y = p.getY();
    }
}

Circle c1 = new Circle(0.0, 0.0, 1.0);
c1.setCenter(1.0, 3.0);

public class Point {
    public double x, y;

    // Constructors and methods
    // ....

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

Point p = new Point(1.0, 3.0);
c2.setCenter(p);
Outline of Exercise 7

Content of Exercise 7

1. Abstract classes
2. Interfaces
Abstraction (lat. *abstractus* – „pulled away“, perfect participle of *abs-trahere*: „to pull away, to take away, to separate“) means a thinking process thatformulates description terms (concept formation) for certain properties of substantial or non-substantial entities.

Non-substantial entities are e.g. „the cold“, „the readiness“, „the watchfulness“.

Substantial entities are e.g. „the bike“, „the house“, „the tree“ [wikipedia]
Abstract Classes - Motivation

- Aggregation of objects with the same properties.

```java
public abstract class Car {
    public abstract void start_driving();
}

public class Automatic extends Car {
    public void start_driving() {
        loosen_break(); accelerate();
    }
}

public class GearBox extends Car {
    public void start_driving() {
        clutch(point); loosen_break();
        gas(something); clutch(move); gas(full);
    }
}
```
Abstract Classes – Properties

• Classes that contain at least one abstract method, are abstract classes
• Abstract classes cannot be instantiated; however, abstract classes have at least one constructor; those can be called from a subclass
• An abstract method is a method, that has a method header, but no method body (no implementation):
  
  - public abstract int getVolumen();

  and:

• Classes are abstract if declared abstract:

  public abstract class Volume { .... }
Examples of Abstract Classes

```java
public abstract class Shape {
    public abstract double area();
    public abstract double circumference();
}
```

```java
public class Circle extends Shape {
    protected double r;
    protected static final double pi = 3.1415926535897932385;

    // Constructors
    public Circle() { r = 1.0; }
    public Circle(double r) { this.r = r; }

    // Methods
    public double area() { return pi * r * r; }
    public double circumference() { return 2 * pi * r; }
    public double getRadius() { return r; }
}
```

```java
public class Rectangle extends Shape {
    protected double w, h;

    // Constructors
    public Rectangle() { w = 0.0; h = 0.0; }
    public Rectangle(double w, double h) {
        this.w = w;
        this.h = h;
    }

    // Methods
    public double area() { return w * h; }
    public double circumference() { return 2 * (w + h); }
    public double getWidth() { return w; }
    public double getHeight() { return h; }
}
```
Use of Abstract Classes: Example

Example: Calculate the total area of many different shapes

// Create an array that includes the different shapes
Shape[] shapes = new Shape[3];

// Define the entries of this array
shapes[0] = new Circle(2.0);
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);

// Calculate the total area of the shapes
double totalArea = 0.0;
for (int i = 0; i < shapes.length; i++)
    totalArea += shapes[i].area();

Objects of subclasses of the abstract class Shape can form the elements of an array of type Shape[]. No explicit casting is necessary.

The methods area() and circumference() can be called for objects of the class Shape though the abstract class Shape doesn't implement them (they are abstract methods!).

Main characteristic of abstract classes!
Outline of Exercise 7

Content of Exercise 7

1. Abstract classes
2. Interfaces
• Interfaces are a **summary of abstract methods and constants**.
  • Abstract: method has no implementation

• Interfaces make certain principal functionalities, that need to be implemented in classes

• Interfaces are hence important for the design and concept, but less important for the implementation.

• The interface is **no class**

• The interface has no implicit constructor

• An interface cannot be instantiated

• Construction of an interface: syntax of a head

![](image)

• An interface describes a class with respect to its public methods.
Interfaces (2)

- Interfaces declare a type that only contains abstract methods as well as constants.
- The methods of an interface must not be labeled to be abstract.
- An interface thus is a tool of pure design whereas abstract classes represent a mixture of design and implementation.
- A class can implement an interface by implementing all the methods declared in an interface.
- Interface types have certain properties similar to class types i.e.:
  - Variables of interface types can be declared (handles).
  - Arguments of methods can be of interface types.
  - Even return values of methods are allowed to be of interface types.
  - But no instance of an interface can be created!

- A class can implement as many interfaces as desired.
In a class, **all** methods of the interface(s) must be implemented.

A class that has implemented an interface has the functionality of the interfaces.
- An object of the class also has its functionality.
- Can also be viewed as the object of the interface.

Interfaces can be complex data types for variables. Objects of all classes that implement that interface can be created.

(The class `ImplementedClass` implements the interface `InterfaceName`)

```java
InterfaceName variable = new ImplementedClass();
```
Example (1)

```java
public class ComplicatedClass { ... }

... 

ComplicatedClass[] f = new ComplicatedClass[50];

We want to sort the array:

public static void sort(ComplicatedClass[] f) {
    ...
}
public class AnotherComplicatedClass { ... }

... 

AnotherComplicatedClass[] f2 = new AnotherComplicatedClass[50];

This array we want to sort, too:

public static void sort
    (AnotherComplicatedClass[] f)
{
    ...
}
• Have a look into the Java-API:
  - Java.util.Arrays contains a method sort():
    public static void sort(Object[] a)
  - Interface Comparable:
    public interface Comparable {
        public int compareTo(Object o);
    }

  sort demands:

  Elements of the array have to implement the interface Comparable
  class ComplicatedClass implements Comparable {
      public int compareTo(Object o) { ... }
  }
Like a class, an interface can extend another interface. Such an interface is called a sub-interface.

```java
public interface DynamicallyScaleable extends Scaleable {
    void changeScale(int size);
}
```

A class implementing the interface `DynamicallyScaleable` must implement all methods of both interfaces, `DynamicallyScaleable` as well as `Scaleable`. 
Example of Interfaces (1)

**Shape**

- `area()`: double
- `circumference()`: double

**Circle**

- `getRadius()`: double

**Rectangle**

- `getWidth()`: double
- `getHeight()`: double

**DrawableCircle**

**DrawableRectangle**

- `setColor(Color)`
- `setPosition(double x, double y)`
- `draw(DrawWindow dw)`

**Interface**

- `Drawable`:
  - `setColor(Color)`
  - `setPosition(double x, double y)`
  - `draw(DrawWindow dw)`

**Class hierarchy of different shapes**

`Shape` is an abstract class

New classes extend the existing classes of different shapes and implement an interface for drawing

Interface for a drawable object
Example of Interfaces (2)

```java
public interface Drawable {
    public void setColor(Color c);
    public void setPosition(double x, double y);
    public void draw(DrawWindow dw);
}

public class DrawableRectangle extends Rectangle implements Drawable {
    // New instance variables
    private Color c;
    private double x, y;
    // A constructor
    public DrawableRectangle(double w, double h) {
        super(w, h);
    }
    // Here follows the implementation of the drawable methods of the interface.
    // All public methods of class Rectangle are inherited.
    public void setColor(Color c) {
        this.c = c;
    }
    public void setPosition(double x, double y) {
        this.x = x; this.y = y;
    }
    public void draw(DrawWindow dw) {
        dw.drawRect(x, y, w, h, c);
    }
}
```