TND002 Object-oriented programming

Lecture 7: Concrete superclasses 1

Mark Eric Dieckmann,
MIT division,
ITN, Linköping University.

January 31, 2023

Outline of the lecture

- Single inheritance.
- Instance methods and inheritance.
- Constructors and inheritance.

Why inheritance?

Variables and methods that are used by several classes are collected in one class.

We can extend the functionality of an existing and tested class and adapt it to our needs.

We use one class as a superclass and let one or more classes (subclasses) inherit its variables and methods.

The keyword for inheritance is *extends*. It is placed in the class declaration line of the subclass:

```
public class ClassName extends Superclass {
```

The class Rectangle

```
public class Rectangle {
private double xPos, yPos, xLen, yLen;
public double area;
public Rectangle(){ xPos = 1.0; yPos = 1.0;}
public void setSize(double a1, double a2){xLen=a1; yLen=a2;}
public double computeArea(){ area = xLen * yLen; return area;}
public String toString(){ return String.format("Position: (%3.1f ,
%3.1f) and area %3.1f", xPos, yPos, area);}}
```

Its attributes are its position in the x-y plane and its side lengths.

It has methods that set its two side lengths, compute its area, and a method that returns user-relevant information.

Single inheritance: the subclass

We want to extend our code framework to circles with radius r.

Create the empty class

```
public class Circle extends Rectangle {}
```

Now check in *main(..)* what **Circle** contains:

```
public static void main(String[] args) {
Circle theCircle = new Circle();
theCircle. // The full stop gives you the pop-up menu. }
```

The keyword *extends* lets **Circle** inherit all variables and methods from **Rectangle** that are not declared *private*.

The access modifier protected

public variables are visible to other classes in the project.

private variables are only visible inside the class.

protected makes variables visible to other classes in the same package (default visibility) and to subclasses in the project.

We replace

```
private double xPos, yPos, xLen, yLen;
public double area;
```

by the declaration

```
protected double xPos, yPos, xLen, yLen, area;
```

Adding a method to the subclass

Our current superclass **Rectangle** is

```
public class Rectangle {
protected double xPos, yPos, xLen, yLen, area;
public Rectangle(){ xPos = 1.0; yPos = 1.0;}
public void setSize(double a1, double a2){xLen=a1; yLen=a2;}
public double computeArea(){area=xLen*yLen; return area;}
public String toString(){ return String.format("Position: (%3.1f ,
%3.1f) and area %3.1f", xPos, yPos, area);}}
```

The size and area of a circle is set by its radius *r*.

```
public class Circle extends Rectangle {
private double r;
//The JVM places here implicitely a default constructor.
public void setSize(double a) {r=a;}
public double computeArea() {area = Math.PI*r*r; return area;}}
```

Content of Circle

Create the instance *Circle c = new Circle();* in main and look at its content in the popup menu *c.*

The circle inherited all instance variables of **Rectangle**.

It inherited *toString()* from **Rectangle** (replacing that of **Object**).

The circle inherited *setSize(double, double)* from **Rectangle** and has its own method *setSize(double)*.

Operator overloading also works for inherited methods.

The circle has only one method *computeArea()*.

The method in the subclass **Circle** overrides the method the circle inherited from the superclass.

Testing Circle and Rectangle

We run the method *main()*

```
public class Test {
public static void main(String[] args) {
Rectangle r = new Rectangle(); r.setSize(1.0,2.0);
r.computeArea(); System.out.println(r);
Circle c = new Circle(); c.setSize(1.0); c.computeArea();
System.out.println(c);}}
```

Console output:

```
Position: (1,0 , 1,0) and area 2,0
Position: (1,0 , 1,0) and area 3.1
```

toString(), *setSize(..)* and *computeArea()* work as they should.

Overloading and overriding of inherited methods

Overloading does not work for methods with the same name and argument list like *computeArea()*.

In an instance of **Circle**, the method *computeArea()* overrides the inherited method *computeArea()* of **Rectangle**.

$\Rightarrow$ We had one correct method *computeArea()* for the superclass and one correct *computeArea()* for the subclass.

setSize(..) is overloaded because we need two arguments for a rectangle while a circle requires only one.

setSize(double, double) is useless for circles and it can do harm if a user calls it.

Disabling setSize(double, double) in Circle

An instance of the superclass (here **Rectangle**) is an instance of the superclass.

An instance of the subclass (here **Circle**) is an instance of the superclass and subclass.

We can distinguish both by testing if the object has a subclass component.

We can test the reference variables *c* and *r* in *main()* or *this* in the object with *instanceof*:

```
if (!(this instanceof Circle)){ Code }
```

executes *Code* only if *this* points to an instance of **Rectangle**.

Cleaning up Rectangle

```
public class Rectangle {
protected double xPos, yPos, area; // inherited by Circle
private double xLen, yLen; // visible only in Rectangle
public Rectangle(){ xPos = 1.0; yPos = 1.0;}
public void setSize(double a1, double a2){
if (!(this instanceof Circle)){xLen = a1; yLen = a2; }}
public double computeArea(){ area = xLen * yLen; return area;}
public String toString(){ return String.format("Position: (%3.1f ,
%3.1f) and area %3.1f", xPos, yPos, area);}}
```

xLen and *yLen* are invisible in instances of **Circle**.

setSize(arg1, arg2) does nothing in instances of **Circle**.

The subclass Circle

```
public class Circle extends Rectangle {
private double r;
public void setSize(double a) {r=a;}
public double computeArea() {area = Math.PI*r*r; return area;}}
```

```
public class Test {
public static void main(String[] args) {
Rectangle r = new Rectangle(); r.setSize(1.0,2.0);
r.computeArea(); System.out.println(r);
Circle c = new Circle(); c.setSize(1.0); c.computeArea();
System.out.println(c);}}
```

Running *main(..)* gives the console output:

```
Position: (1,0 , 1,0) and area 2,0
Position: (1,0 , 1,0) and area 3.1 // why this position?
```

Constructors and inheritance

The values for the position are set only in *Rectangle()*

```
public class Rectangle {protected double xPos, yPos, area;
private double xLen, yLen;
public Rectangle(){ xPos = 1.0; yPos = 1.0;} // code continues
```

```
public class Circle extends Rectangle {
private double r;
public Circle(){ } /* Explicit constructor with no arguments and
code - like the implicit one the JVM does if we don't code one.*/
```

Circle() does an implicit call to *Rectangle()* that is not visible.

Implicit calls in *Circle()* or *Circle(..)* work if we only want to call the constructor *Rectangle()* with no arguments.

Explicit calls to the superclass constructor

```
public class Rectangle {
protected double xPos, yPos, area;
private double xLen, yLen;
public Rectangle(){ xPos = 1.0; yPos = 1.0;}
public Rectangle(double r1, double r2){xPos = r1; yPos = r2;}
// code continues
```

Explicit calls get to *Rectangle(..)* with arguments:

```
public class Circle extends Rectangle {
private double r;
public Circle(){super(); // More code can follow super().}
public Circle(double c1, double c2){super(c1, c2);}
// code continues
```

Summary

We have discussed in this lecture

- single inheritance.
- Instance methods and inheritance.
- Constructors and inheritance.