TND002 Object-oriented programming

Lecture 8: Concrete superclasses 2

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

February 1, 2023

Outline of the lecture

- We replace single inheritance by hierarchical inheritance.
- We look at how variables and methods are inherited.
- We use super to access variables and methods of the superclass.
- We discuss polymorphism.
- We show how we can cast classes (if possible).

Single inheritance

We have implemented the superclass **Rectangle** and the subclass **Circle**.

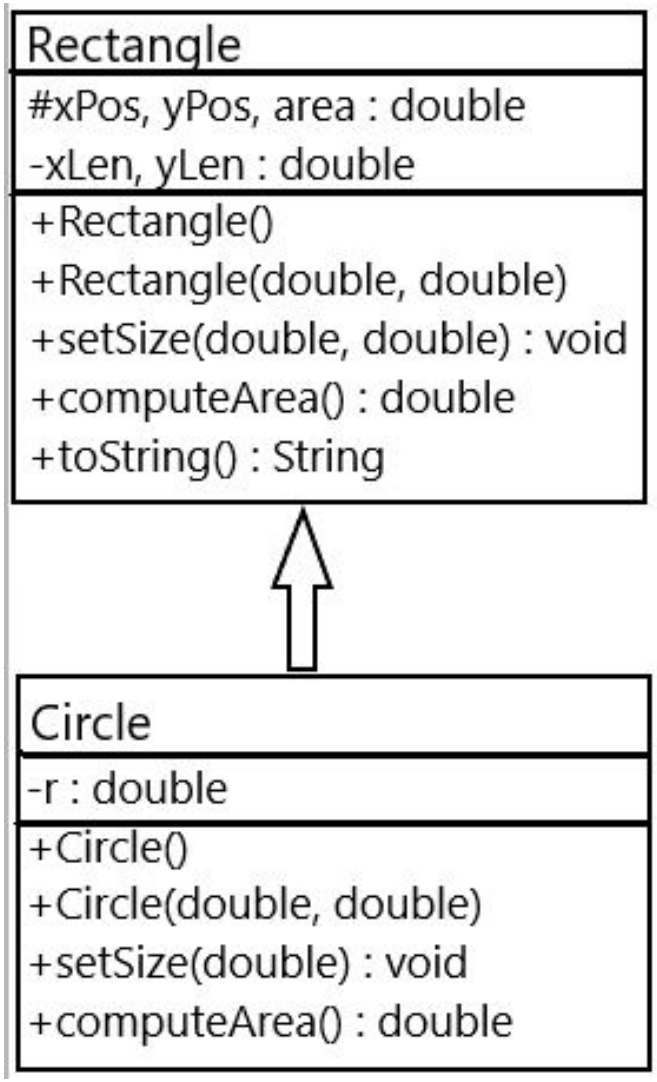
The arrow shows that **Circle** is a subclass of **Rectangle**

# denotes *protected*.

computeArea() in **Circle** overrides the inherited one.

In *overriding*, the JVM selects the correct method at runtime.

setSize(..) is overloaded in instances of **Circle**.



Hierarchical inheritance

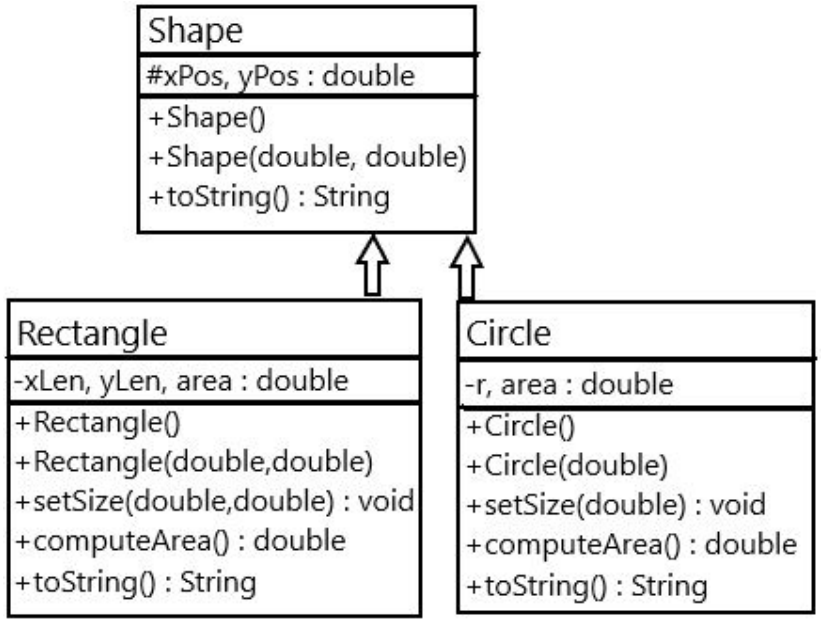
setSize(double, double) makes no sense for instances of **Circle** because we can only set the radius.

We cannot prevent **Circle** from inheriting *setSize(double, double)* (through overriding, access modifiers).

We place methods shared by **Rectangle** and **Circle** in the joint superclass **Shape**.

This arrangement is called hierarchical inheritance.

Each subclass can only have one superclass.



Implementing Shape

Instances of **Rectangle** and **Circle** share position coordinates.

We declare and initialize coordinates in **Shape**.

The *toString()* method returns information about them.

```
public class Shape {
protected double xPos, yPos;
public Shape() {xPos = 0.0; yPos=0.0;}
public Shape(double a1, double a2) {xPos = a1; yPos = a2;}
public String toString()
{return String.format("Position: (%3.1f , %3.1f)", xPos, yPos);}}
```

In our implementation, object size information is stored in the subclass and we cannot use it in *toString()* of **Shape**.

Implementing Rectangle

```
public class Rectangle extends Shape {
private double xLen, yLen, area;
public Rectangle(){super(); setSize(1.0,1.0); computeArea();}
public Rectangle(double a1, double a2)
{super(a1,a2); setSize(1.0,1.0); computeArea();}
public void setSize(double a1, double a2){xLen=a1;yLen=a2;}
public double computeArea() {area = xLen*yLen; return area;}
public String toString(){ return .. } }
```

toString() overrides the one inherited from **Shape**.

We can access overridden methods using *super*:

```
public String toString()
{return super.toString() + String.format(", area = %3.1f", area); }
```

Implementing Circle and completing our framework

We implement the subclass **Circle** in the same way.

```
public class Circle extends Shape {
private double r, area;
public Circle() {super(); setSize(1.0); computeArea();}
public Circle(double a1, double a2)
{super(a1,a2); setSize(1.0); computeArea();}
public void setSize(double a) {r=a;}
public double computeArea() {area = Math.PI*r*r; return area;}
public String toString()
{return super.toString() + String.format(", area = %3.1f", area);}}
```

Our hierarchy **Shape**, **Rectangle** and **Circle** works.

Shared variables and methods are located in **Shape**.

Overriding of instance methods and variables

Inheritance lets a subclass (also called derived class) inherit variables and methods from its superclass (or base class).

Instance variables and methods in the subclass override the inherited ones. We access overridden members using *super*.

```
public class Base {
protected String variable = "Base";
public String instanceMethod()
{return "Base class: " + variable;}}
```

```
public class Derived extends Base {
protected String variable = "Derived";
public String instanceMethod() // Overrides inherited method
{return super.instanceMethod() + " and " + super.variable;}}
```

Testing the sub- and superclass

```
public static void main(String[] args) {
Base b = new Base(); Derived d = new Derived();
System.out.println(b.variable);
System.out.println(b.instanceMethod());
System.out.println(d.variable);
System.out.println(d.instanceMethod());}
```

console output:

```
Base
Base class: Base
Derived
Base class: Base and Base
```

super. accesses the superclass part of the hybrid object.

Inheritance of class variables and methods

Class variables/methods are called with a class name and they are not part of the object. All objects have access to it.

They will not behave like instance variables/methods:
super. does not work because it requires an object.

```
public class Base {
protected static String variable = "Base";
protected static String variable2 = "Base2";
public static String classMethod()
{return "Base class: " + variable + " " + variable2;}}
```

```
public class Derived extends Base {
protected static String variable = "Derived";
public static String classMethod()
{return "Derived class: " + variable + " " + variable2;} }
```

Testing the sub-and superclass

```
public static void main(String[] args) {
System.out.println(Base.variable);
System.out.println(Base.variable2);
System.out.println(Base.classMethod());
System.out.println(Derived.variable);
System.out.println(Derived.variable2);
System.out.println(Derived.classMethod());}
```

```
Base
Base2
Base class: Base Base2
Derived
Base2
Derived class: Derived Base2
```

Class variables and methods hide the ones of the superclass with the same names. The subclass can access *variable2*.

Polymorphism

Definition: the condition of occurring in several different forms.

The JVM selects appropriate methods based on the argument list (overloading) or the type of the object (method overriding).

The JVM allows for a third useful feature: A subclass is an instance of both, the subclass and the superclass.

```
// We use classes from our previous example
Base b = new Derived(); // works
Derived d = new Derived(); // works
Derived d = new Base(); // does not work
```

The last line does not work, because a superclass does not have a subclass component (*extends* works only one way).

Subclass reference and subclass object

Consider the new classes **Base** and **Derived**

```
public class Base{
public String methodA(){return "Method A Base";}}
```

```
public class Derived extends Base {
public String methodA() {return "Method A Derived";}
public String methodB() {return "Method B Derived";}}
```

```
public static void main(String[] args) {
Derived b = new Derived();
System.out.println(b.methodA());
System.out.println(b.methodB());}
```

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

```
Method A Derived
Method B Derived
```

Superclass reference and subclass object

Let us attach a subclass object to a superclass reference

```
public static void main(String[] args){
Base b = new Derived();
System.out.println(b.methodA()); }
```

The console output is

```
Method A Derived
```

The JVM performs two steps.

It finds *methodA()* in **Base** at compile time (no error message).

At runtime, it overrides *methodA()* of **Base** with that of **Derived**.

Accessing methodB() of the subclass

The following code gives us an error message

```
public static void main(String[] args){
Base b = new Derived(); System.out.println(b.methodB());}
```

Base has no *methodB()* and we get an error at compile time.

If we want to access *methodB()* of **Derived** using the address in *b*, we need to do a cast.

We need to check if such a cast is possible.

```
public static void main(String[] args){
Base b = new Derived();
if (b instanceof Derived) {Derived d = (Derived) b;
System.out.println(d.methodB());}}
```

Summary

We have discussed how

- we replace single inheritance with hierarchical inheritance.
- variables and methods are inherited.
- we can use super to access instance variables and instance methods of the superclass.
- polymorphism works.
- we can cast classes (if possible).