TND002 Object-oriented programming

Lecture 10: Abstract classes and interfaces

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

February 15, 2023

Outline of the lecture

- We discuss abstract classes and methods.
- We discuss the interface **Comparable**$< T >$.
- We introduce the class method *sort(arg)* of **Collections**.

Motivation

We discussed hierarchical inheritance based on the superclass **Shape** and the subclasses **Rectangle**, and **Circle**.

Shape collected variables and methods used by both subclasses such as *setPosition(arg1, arg2)* and *toString()*.

Methods, which are specific for the actual shape, such as *computeArea()* were implemented in the subclasses.

Sometimes we want to enforce a standard for all subclasses: all should have a *computeArea()* method.

We also want to access *computeArea()* through a variable of type **Shape** (superclass reference for subclass object).

$\Rightarrow$ We need a superclass method *computeArea()*.

Solution we know: concrete method *computeArea()*

We can introduce a default method in the superclass:

```
public class Shape {private double xPos, yPos;
public Shape(double d1, double d2) {xPos = d1; yPos = d2;}
public void setPosition(double d1, double d2)
{xPos = d1; yPos = d2;}
public void computeArea(){ area = 0.0;}
public String toString()
{return String.format("Position: (%3.1f,%3.1f)",xPos,yPos);}}
```

All subclasses of **Shape** will inherit this method.

No matter how we implement *computeArea()*, the area cannot be set to the correct value in **Shape**.

An instance of **Shape** is useless on its own.

Better solution

Declare an abstract method *computeArea()* in the superclass:

```
public abstract void computeArea(); // no code body
```

Abstract methods are like constants, which have been declared but not initialized. We must initialize them before we use them.

A class, which contains an abstract method, must be abstract.

```
public abstract class Shape {private double xPos, yPos;
public Shape(double d1, double d2) {xPos = d1; yPos = d2;}
public void setPosition(double d1, double d2)
{xPos = d1; yPos = d2;}
public abstract void computeArea();
public String toString()
{return String.format("Position: (%3.1f,%3.1f)",xPos,yPos);}}
```

Abstract classes

An abstract class cannot be instantiated.

$\Rightarrow$ We cannot instantiate **Shape** on its own.

An abstract class can have subclasses.

If the subclass implements all inherited abstract methods (implements: adding code), it can be concrete.

If one or more abstract methods are left in the subclass, the subclass must be abstract too.

Implementing abstract classes

We can use code, which is defined in the abstract superclass,

```
public abstract class Shape {
// Other variables and methods
public abstract void computeArea();}
```

and implement the abstract method

```
public class Rectangle extends Shape {
private double xLen, yLen, area;
// Other variables and methods
public void computeArea(){area = xLen * yLen;}}
```

and use instances of **Rectangle** as before.

Using abstract classes

Although we can not create an instance of the abstract **Shape**, we can still use reference variables of this type:

```
public class MainClass {
public static void main(String[] args) {
// Shape myShape = new Shape(); // Error
// Rectangle myRectangle = new Shape(); // Error
Shape myShape = new Rectangle(); // Works
Rectangle myRectangle = new Rectangle(); // Works}}
```

We can thus define arrays for the abstract superclass **Shape** and add instances of all its subclasses to it.

Benefits of the enforced standard

We can only instantiate a subclass, which implemented all inherited abstract methods.

$\Rightarrow$ If an object of that subclass exists, it has implemented all abstract methods.

Other methods, which need the implemented abstract methods, can be applied to this object.

We used the existing *compareTo(arg)* method of **String** and implemented our own in **Vector** (lab 1) and **Dictionary** (lab 2).

If a class has an implemented method *compareTo(arg)*, we can use it to bubble-sort elements of a dynamic array.

Interfaces

Many standard methods in Java can be used on a class, which implemented an abstract method required by the other method.

Java allows only for single inheritance and we do not want to "waste" the abstract superclass for just this.

Interfaces are abstract classes with reduced functionality. Interfaces can contain abstract methods but no concrete ones.

A subclass *extends* a superclass and it *implements* an interface.

Any other method can test if an interface is implemented.

The interface Comparable<T>

Comparable<T> contains the abstract method *int compareTo(arg)*, which compares instances *arg* of **T**.

```
public class Rectangle implements Comparable<Rectangle>{
private double area; private int number;
public Rectangle(int arg1, double arg2)
{number=arg1; area = arg2;}
public int compareTo(Rectangle arg) {
if (area==arg.area) return 0;
else if (area < arg.area) return -1; else return 1;}
public String toString() {return String.format("Number:%2d and
area:%4.1f", number, area);}}
```

An instance of **Rectangle** is an instance of **Comparable**:

instanceof can check if *compareTo(arg)* is implemented.

Collections

Collections is a class, which consists of class methods that can be used for example on dynamic arrays.

The class method *Collections.sort(dynArr)* can sort the dynamic array *dynArr* based on the criterion in your *compareTo(arg)*.

```
public static void main(String[] args) {
ArrayList<Rectangle> tL = new ArrayList<Rectangle>();
tL.add(new Rectangle(1,1.0)); tL.add(new Rectangle(2,4.0));
tL.add(new Rectangle(3,0.5)); tL.add(new Rectangle(4,2.5));
Collections.sort(tL);
for (int i=0; i < tL.size(); i++) {System.out.println(tL.get(i));}}
```

You get the sequences 3, 1, 4, 2 sorted by increasing areas.

Comparing instances of different classes

We can also compare circles with each other.

```
public class Circle implements Comparable<Circle>{
private double area; private int number;
public Circle(int arg1, double arg2)
{number=arg1; area = arg2;}
public int compareTo(Circle arg) {
if (area==arg.area) return 0;
else if (area < arg.area) return -1; return 1;}
public String toString() {return String.format("Number:%2d and
area:%4.1f", number, area);}}
```

We compare instances of the class with instances of the class specified in $< T >$. They do not have to be the same.

The class in $< T >$ must match that of *arg* in *compareTo(arg)*.

Comparing rectangles and circles

Depending on how we define the class we can compare

```
public class Circle implements Comparable<Circle>{}
```

```
public class Circle implements Comparable<Rectangle>{}
```

```
public class Rectangle implements Comparable<Circle>{}
```

```
public class Rectangle implements Comparable<Rectangle>{}
```

We can, however, only pick one class that we can compare with **Rectangle** and one we can compare with **Circle**.

Comparing rectangles and circles

If we want all combinations of circles and rectangles, we introduce the superclass **Shape**.

```
public abstract class Shape implements Comparable<Shape>{
protected double area;
public int compareTo(Shape arg) {
if (area==arg.area) return 0;
else if (area < arg.area) return -1; else return 1;}}
```

This one does not have to be abstract (It can be anyway).

If we implement *toString(Shape arg)* in **Shape**, we must declare *area* in **Shape**.

We can also implement *toString(Shape arg)* in the subclasses and leave *area* there as long as **Shape** is abstract.

Comparing rectangles and circles

The two subclasses are:

```
public class Rectangle extends Shape {
private int number;
public Rectangle(int arg1, double arg2)
{number=arg1; area = arg2;}
public String toString() {return String.format("Rectangle:%2d
and area:%4.1f", number, area);} }
```

```
public class Circle extends Shape {
private int number;
public Circle(int arg1, double arg2)
{number=arg1; area = arg2;}
public String toString() {return String.format("Circle:%2d and
area:%4.1f", number, area);} }
```

Running main(..)

We can test our framework with:

```
public class MainClass {
public static void main(String[] args) {
ArrayList<Shape> tL = new ArrayList<Shape>();
tL.add(new Rectangle(1,0.5)); tL.add(new Rectangle(2,4.0));
tL.add(new Circle(1,3.0)); tL.add(new Circle(2,2.0));
Collections.sort(tL);
for (int i=0; i < tL.size(); i++) {
System.out.println(tL.get(i));}}
```

It gives us the sorted sequence Rectangle 1, Circle 2, Circle 1, and Rectangle 2.

Summary

We have discussed in this lecture

- abstract classes and methods.
- interfaces and in particular **Comparable**.
- the class method **sort(arg)** of **Collections**.