

# Objektorienterad Programmering

## Tentamen

2001-03-19  
kl. 09.00-13.00

1. I kursboken utvecklas ett program för att rita funktionsgrafer. Det finns en återanvändbar klass `MultiPlotter` som är lätt att använda för att få en fungerande plotter. Här ser Du ett exempel där vi ärver från `MultiPlotter` för att få en fungerande applet som ritat grafer för funktionerna sinus och cosinus.

```
import java.awt.Color;

public class PlotSineCosine extends MultiPlotter {
    public void initMultiPlotter() {
        Function sine = new Sine();
        Function cosine = new Cosine();
        addFunction(sine, Color.green);
        addFunction(cosine, Color.blue);
    }
}
```

Nyckeln i utvecklingen av detta program är interfacet

```
interface Function {
    double apply(double x);
}
```

som implementeras bland annat av klasserna `Sine` och `Cosine`. I denna uppgift skall vi fortsätta utveckling genom att erbjuda sätt att bygga nya funktioner från redan definierade funktioner. Vi vill kunna skriva program som t.e.

```
import java.awt.Color;

public class PlotCombinations extends MultiPlotter {
    public void initMultiPlotter() {
        Function sine = new Sine();
        Function cosine = new Cosine();
        addFunction(sine, Color.green);
        addFunction(cosine, Color.blue);
    }
}
```

```

        addFunction(new FunctionSum(sine,cosine),Color.orange);
        addFunction(new FunctionTranslate(sine,0.5),Color.gray);
    }
}

```

(a) 2 pts. Definiera en klass

```
public class FunctionSum implements Function
```

som kan användas för att bilda summan av två funktioner. I matematik definierar man summan av två funktioner  $(f + g)(x) = f(x) + g(x)$ . I konstrueraren skall man kunna ange de funktioner som skall summeras.

**lösningsförslag:**

```

class FunctionSum implements Function {

    private Function f,g;

    public FunctionSum(Function h1, Function h2){
        f=h1;
        g=h2;
    }

    public double apply(double x){
        return f.apply(x)+g.apply(x);
    }
}

```

(b) 2 pts. Definiera en klass

```
public class FunctionTranslate implements Function
```

som kan användas för att flytta en funktion sidledes. I matematik definierar man funktionens flytning:  $(translate\ f\ dx)(x) = f(x+dx)$ . I konstrueraren skall man kunna ange den funktion som skall flyttas samt den distans den skall flyttas.

**lösningsförslag:**

```

class FunctionTranslate implements Function {

    private Function f;
    private double dx;

    public FunctionTranslate(Function h, double d){
        f=h;
        dx = d;;
    }

    public double apply(double x){

```

```

        return f.apply(x+dx);
    }
}

```

2. *4 pts.* Definiera en klass `ArithmeticField` vars objekt kan användas som komponent i ett grafiskt användargränssnitt. Komponentens ska användas för att beräkna aritmetiska uttryck som man anger i text form. I din komponent skall, när användaren trycker på `return`, värdet av det inmatade uttrycket ersätta uttrycket i komponenten.

För att utföra beräkning kan Du använda Dig av den befintliga klassen `Evaluator` som erbjuder

```

public Evaluator(String expression)
public double eval()

```

*Tips:* det finns en component `JTextField` i swing paketet där man kan skriva in text. När man trycker på `return` tangenten genereras en `ActionEvent` som kan hanteras av en `ActionListener` kopplad till komponenten. Den har även metoder

```

public String getText()
public void setText(String s)

```

#### lösningsförslag:

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

```

```

// a text field that accepts an arithmetic expression and on return
// calculates its value.

```

```

public class ArithmeticField extends JTextField implements ActionListener{

    private Stack stack;
    private Evaluator e;

    public ArithmeticField(int col){
        super(col);
        stack = new Stack();
        addActionListener(this);
    }

    public void actionPerformed(ActionEvent e){
        e = new Evaluator(getText());
        setText(""+e.eval());
    }
}

```

```

public static void main(String [] a){
    JFrame f = new JFrame();
    f.setTitle("Calculator");
    f.getContentPane().add(new ArithmeticField(10));
    f.pack();
    f.setVisible(true);
}
}

```

3. I kursboken hittar vi följande återanvändbar klass som hjälpmedel för att definiera animerade applets

```

import java.awt.*;

public class AnimatedApplet extends java.applet.Applet
    implements Runnable{

    private Thread animationThread = null;
    private int delay;

    public void setDelay(int d){
        delay = d;
    }

    public int getDelay(){
        return delay;
    }

    public void start(){
        if(animationThread == null){
            animationThread = new Thread(this);
            animationThread.start();
        }
    }

    public void run(){
        while(Thread.currentThread() == animationThread){
            repaint();
            try{Thread.currentThread().sleep(delay);
            }catch(InterruptedException e){}
        }
    }

    public void stop(){
        animationThread = null;
    }
}

```

```
}
```

Det räcker att vi definierar metoderna `void paint(Graphics g)` och `void init()` för att få ett fungerande animerat applet. Det är på detta vi som t.ex. den digitala klockan definierades!

Många animerade applets kommer att bestå av ett antal animerade objekt. I denna uppgift skall Du definiera hjälpmedel för denna typ av program. Vi tänker oss att skriva program som följande:

```
public class MyApplet extends MultiObjectAnimatedApplet{
    public void initAnimation(){
        setDelay(100);
        addObject(new AnimatedString('Moving Around'));
    }
}
```

som bygger på

```
public abstract class MultiObjectAnimatedApplet extends AnimatedApplet{
    protected AnimatedCollection theObjects;
    public abstract void initAnimation();

    public void paint(java.awt.Graphics g){
        theObjects.move();
        theObjects.paint(g);
    }

    public void init(){
        theObjects = new AnimatedCollection();
        initAnimation();
    }

    public void addObject(AnimatedObject ao){
        theObjects.add(ao);
    }
}
```

Följande deluppgifter leder till definitionen av `AnimatedCollection`.

(a) *3 pts.* Komplettera definitionen av

```
import java.awt.*;
public abstract class AnimatedObject{
    // Du måste bestäma vilka attributer behövs!
    public AnimatedObject(int x, int y) { // din uppgift! }

    public void setAnimator(Animator a) { // din uppgift! }
```

```

    public void move(){// din uppgift!}

    public abstract void paint(Graphics g);
}

```

för objekt som kan röra sig. Metoden `move` skall använda sig av en `Animator` för att beräkna ändringen i objektets position. En lämplig definition av `Animator` är

```

public interface Animator{
    java.awt.Point deltaP();
}

```

som beskriver det beteende av objekt som beräknar en position ändring i form av en punkt bestående av ändring enligt  $x$  och ändring enligt  $y$ .

### **lösningsförslag:**

```

import java.awt.*;
class AnimatedObject{
    protected Point p;
    protected Animator animator;

    public AnimatedObject(int x, int y){
        p = new Point(x,y);
    }

    public void setAnimator(Animator a){
        animator = a;
    }

    public void move(){
        if (animator != null){
            Point deltaP = animator.deltaP();
            p.x += deltaP.x;
            p.y += deltaP.y;
        }
    }

    public abstract void paint(Graphics g);
}

```

(b) 3 pts. Definiera en klass

```

public class ConstantVelocity implements Animator

```

där metoden `deltaP` returnerar alltid samma `Point` (som står för hastigheten, hur mycket positionen varierar i en tidsenhet). I konstrueraren skall man kunna ange hastigheten i form av två heltal (hastighet enligt  $x$  och enligt  $y$ ).

**lösningsförslag:**

```
import java.awt.*;
public class ConstantVelocity implements Animator{
    protected Point velocity;

    public ConstantVelocity(int dx, int dy){
        velocity = new Point(dx,dy);
    }

    public Point deltaP(){return velocity;}
}
```

- (c) *3 pts.* Definiera en klass `public class AnimatedString extends AnimatedObject` för strängar som kan röra på sig. För ritning, finns det hos klassen `Graphics` metoden `void drawString(int x, int y, String s)`. I konstrueraren skall man kunna ange den sträng som skall animeras.

**lösningsförslag:**

```
import java.awt.*;
public class AnimatedString extends AnimatedObject{
    protected String theString;

    public AnimatedString(int x, int y, String s){
        super(x,y);
        theString = s;
        setAnimator(new ConstantVelocity(5,5));
    }

    public AnimatedString(String s){
        this(0,0,s);
    }

    public void paint(Graphics g){
        g.setColor(Color.red);
        g.drawString(theString,p.x,p.y);
    }
}
```

- (d) *3 pts.* Komplettera definitionen av

```

import java.util.*;
import java.awt.*;
public class AnimatedCollection{
    private Collection theObjects;

    public AnimatedCollection(){
        theObjects = new LinkedList();
    }

    public void add(AnimatedObject o){
        theObjects.add(o);
    }

    public void move(){// din uppgift!}

    public void paint(Graphics g){// din uppgift!}
}

```

**lösningsförslag:**

```

import java.util.*;
import java.awt.*;
public class AnimatedCollection{
    Collection theObjects;

    public AnimatedCollection(){
        theObjects = new LinkedList();
    }

    public void add(AnimatedObject o){
        theObjects.add(o);
    }

    public void move(){
        Iterator iter = theObjects.iterator();
        while(iter.hasNext()){
            ((AnimatedObject)iter.next()).move();
        }
    }

    public void paint(Graphics g){
        Iterator iter = theObjects.iterator();
        while(iter.hasNext()){
            ((AnimatedObject)iter.next()).paint(g);
        }
    }
}

```