

Läsanvisningar

Börja med att läsa kapitlen i kursboken, därefter resurserna på Internet och till sist själva lektionsmaterialet.

Kursboken (5-8:e upplagan): Kapitel 7

Internet: The Java Tutorial, Trail: 2D Graphics: - Overview of the Java 2D API Concepts
(<http://docs.oracle.com/javase/tutorial/2d/overview/index.html>)

Internet: The Java Tutorial, Trail: 2D Graphics: - Getting Started with Graphics
(<http://docs.oracle.com/javase/tutorial/2d/basic2d/index.html>)

Internet: The Java Tutorial, Trail: 2D Graphics: - Working with Geometry
(<http://docs.oracle.com/javase/tutorial/2d/geometry/index.html>)

Internet: The Java Tutorial, Trail: 2D Graphics: - Constructing Complex Shapes from Geometry Primitives
(<http://docs.oracle.com/javase/tutorial/2d/advanced/complexshapes.html>)

Internet: Oracle Developer Network: Samples: - Java 2D API Sample Programs
(<https://wikis.oracle.com/display/code/Multi-Media#Multi-Media-2DGraphics>)

Ej obligatoriskt. En sida med ett stort antal exempelprogram som täcker upp det mesta i Java 2D API.

Internet: Oracle Developer Network: Technical Articles and Tips: - Painting in AWT and Swing
(<http://www.oracle.com/technetwork/java/painting-140037.html>)

Ej obligatoriskt. Artikeln ovan förklarar i detalj hur ritprocessen fungerar. Ganska tekniskt och tar upp många saker vi inte nämnt, men kan vara bra att läsa för de som vill få en djupare förståelse. För de som inte orkar läsa hela artikeln bör åtminstone avsnittet Swing Painting Guidelines läsas.

2D-grafik och egna komponenter

Hela processen hur Swing ”ritar” de olika komponenterna, som till exempel en JButton, är ganska komplicerat att sätta sig in i och hur detta görs är inget vi tittar på i denna kurs. Denna lektion fokuserar i stället på hur vi i en befintlig komponent kan rita egen 2D-grafik (linjer, cirklar med mera). Som nämndes i lektion 4 är det metoden `paintComponent` och klassen `Graphics` som används för frihandsritning. Som ett exempel nämndes att vi i en JButton till exempel kunde rita ett rött kryss över knappen.

Graphics

Som ett första exempel på hur klassen `Graphics` används tittar vi på hur vi gör just detta.

```
// Skapa knappen
JButton jButton = new JButton("En knapp");

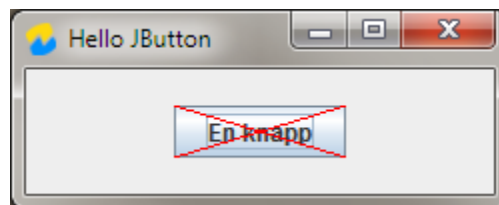
// Få referens till det Graphics-objekt som används för att rita i komponenten
Graphics g = jButton.getGraphics();
```

```
// Sätt färgen till röd
g.setColor(Color.RED);

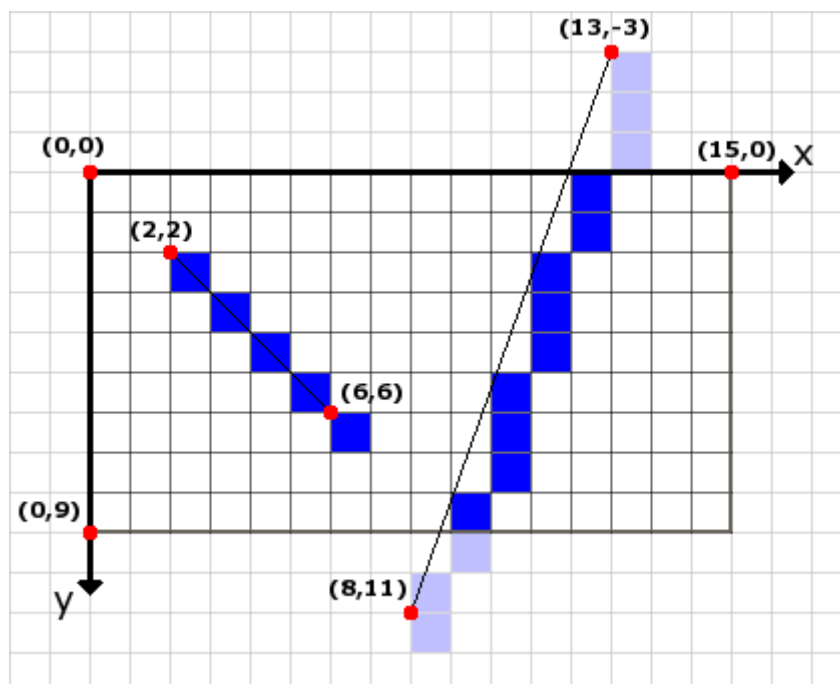
// Rita krysset
g.drawLine(0, 0, jButton.getWidth() - 1, jButton.getHeight() - 1);
g.drawLine(jButton.getWidth() - 1, 0, 0, jButton.getHeight() - 1);
```

I klassen `java.awt.Graphics` finns ett antal olika metoder som kan användas för att rita figurer, texter och bilder i en komponent. För att få tag på det `Graphics`-objekt som används för att rita i den aktuella komponenten anropar vi metoden `getGraphics` (finns i klassen `JComponent` och därmed alla komponenter i Swing). För att ange vilken färg som ska användas när vi ritar anropar vi `setColor` och skickar med ett `Color`-objekt. Denna färg (röd) används nu för alla ritoperationer till dess att vi på nytt anropar `setColor` med en annan färg.

För att tillsist rita själva krysset använder vi metoden `drawLine` som ritar en linje mellan två punkter. Vi tar reda på knappens aktuella bredd (`getWidth`) och höjd (`getHeight`) och använder dessa värden för att först rita en linje mellan knappens övre vänstra hörn $(0,0)$ och nedre högra hörn $(jButton.getWidth()-1, jButton.getHeight()-1)$ och dels mellan knappens övre högra hörn $(jButton.getWidth()-1, 0)$ och nedre vänstra hörn $(0, jButton.getHeight()-1)$.



Klassen `Graphics` använder ett koordinatsystem där övre vänstra hörnet anges av punkten $(0,0)$ och där det nedre högra punkten beror på aktuell komponents bredd och höjd. Det rektangulära området som utgörs av punkterna ovan anger det synliga området vi kan rita i. Den enhet som används i koordinatsystemet är en pixel och alla positioner (punkter) och dimensioner uttrycks i antal pixlar. Det är fullt möjligt att ange en punkt som ligger utanför det synliga området.



I figuren ovan utgörs det synliga området av punkterna (0,0) och (15,9) vilket innebär att den yta vi kan rita på har storleken 16x10 pixlar. Koordinaterna är oändligt tunna och ligger mellan pixlarna. När vi ritar något "fylls" pixeln nedanför och till höger om den koordinat som anges. I figuren ovan har vi ritat två linjer (drawLine). Den första linjen går mellan punkterna (2,2) och (6,6), vilket helt ligger inom det synliga området. Den andra linjen går mellan punkterna (13,-3) och (8,11), vilka startar och slutar utanför det synliga området. Det är endast pixlarna i det synliga området som syns.

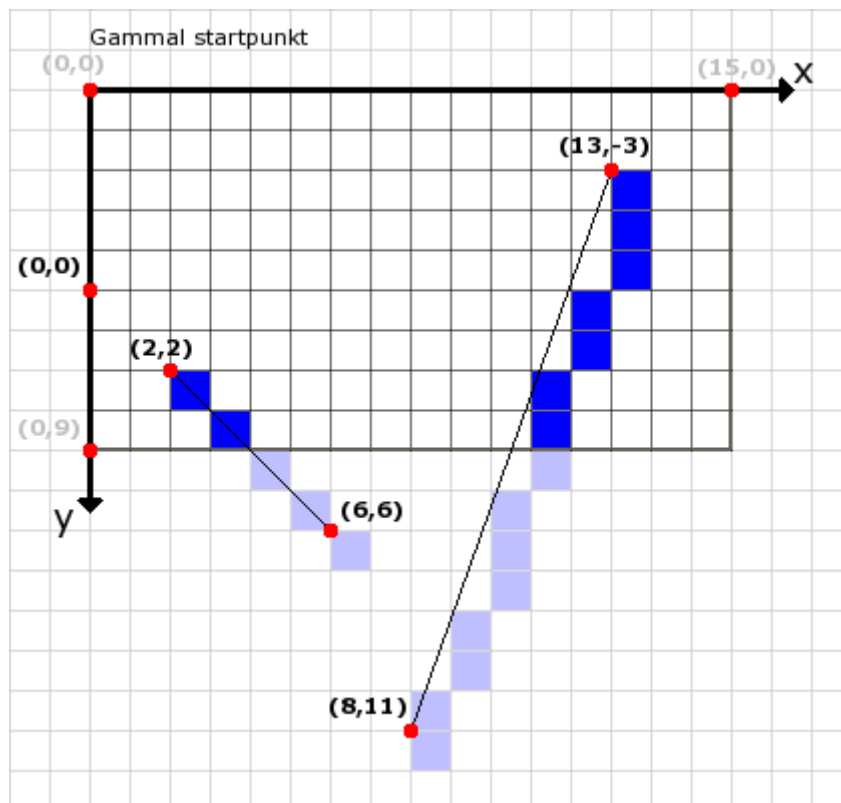
Att det är pixlarna nedan för och till höger som fylls är viktigt att komma ihåg när vi ritar nära högra och nedre kanten i en komponent. Detta är anledningen till varför vi subtraherat med 1 i exemplet med det röda krysset på knappen. Hade vi inte gjort det hade linjerna slutat precis nedanför och till höger om det synliga området.

Utöver metoder för att rita med finns det i klassen Graphics även metoder som kan användas för att ange vilken färg vi ska rita med (setColor), ange vilken font texter ska skrivas ut med (setFont), rensa en del av ritytan (clearRect), kopiera en del av ritytan till en annan position (copyArea) och ange det synliga området (setClip). Hur dessa metoder används kan du läsa om i Javas API för klassen Graphics.

En annan användbar metod är translate(int x, int y) som flyttar startpunkten (0,0) till punkten (x,y). Alla metदानrop där koordinater används kommer nu att utgå relativt den nya startpunkten. Det synliga området påverkas inte. Detta kan vara användbart om vi till exempel ska rita ett stapeldiagram och där vi är vana att använda negativa värden på y-axeln. Vi kan då flytta startpunkten så att y-axeln ligger på halva komponentens höjd:

```
g.translate(0, getWidth() / 2);
```

Utgår vi från figuren ovan kommer det nu att se ut så här i stället.



Den nya startpunkten ligger nu i punkt (0,5) enligt det ordinarie koordinatsystemet. Alla ritoperationer utgår nu från den nya startpunkten. Även varje nytt anrop till translate utgår från den nya startpunkten. Ett anrop till translate(0,0) återställer med andra ord inte koordinatsystemet till det ordinarie.

Ytterligare ett användningsområde för translate ser du längre ner när vi tittar på metoden drawPolygon (månhörningar).

I klassen Graphics finns ett 20-tal olika metoder vi kan använda för att rita linjer, cirklar, bilder, text med mera, antingen fyllda eller inte fyllda. Nedan ges några exempel på dessa metoder. Den komponent som används för att rita på har storleken 234x62 pixlar.

3D Rektangel

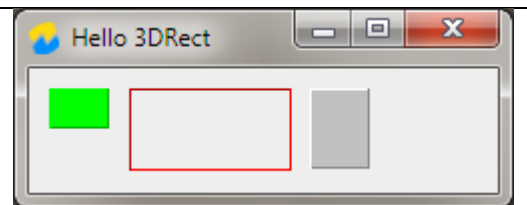
```
draw3DRect(int x, int y, int width, int height, boolean raised)
fill3DRect(int x, int y, int width, int height, boolean raised)
```

Metoderna ovan ritar en rektangel som ges en 3D-liknande effekt. Rektangeln börjar i punkt (x,y) och är width pixlar bred och height pixlar hög. Rektangeln ritas antingen nedtryckt eller upphöjd beroende på värdet i raised. Som du ser av bilden nedan syns 3D-effekten väldigt dåligt.

```
g.setColor(Color.GREEN);
g.fill3DRect(10, 10, 30, 20, true);

g.setColor(Color.RED);
g.draw3DRect(50, 10, 80, 40, false);

g.setColor(Color.LIGHT_GRAY);
g.fill3DRect(140, 10, 30, 40, true);
```



Båge

```
drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)
```

Metoderna ovan ritar en båge som ryms inuti den rektangel som börjar i punkt (x,y) och är width pixlar bred och height pixlar hög. Bågens startpunkt anges av startAngle och dess längd av arcAngle (båda i enheten grader).

```
g.drawArc(0, 10, 40, 40, 0, 90);
g.fillArc(40, 10, 40, 40, 0, 90);

g.drawArc(90, 10, 40, 40, 45, 90);
g.fillArc(130, 10, 40, 40, 45, 90);

g.setColor(Color.YELLOW);
g.fillArc(180, 10, 40, 40, 30, 300);
```



Linje

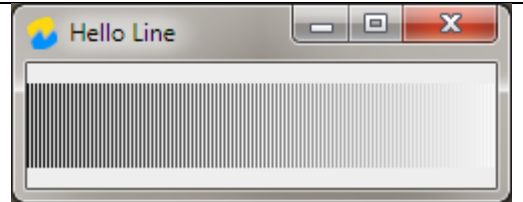
```
drawLine(int x1, int y1, int x2, int y2)
```

Metoden ovan ritar en linje mellan punkterna (x1,y1) och (x2,y2). Exemplet nedan ritar vertikala linjer som startar 10 pixlar nedanför komponentens övre kant och 10 pixlar ovanför komponentens nedre kant (notera att vi subtraherar med 11 för höjden, minns du varför?). Vi ritar linjer på

varannan (steps) x-koordinat med början i komponentens vänstra kant och slutar vid komponentens högra kant. För varje linje som ritas tonas färgen från svart till vit så att i början är färgen helt svart och i slutet är den helt vit.

```
int w = getWidth();
int h = getHeight();
int steps = 2;

for (int i = 0; i < w; i += steps) {
    int rgb = (int)((i / (double)w) * 255);
    g.setColor(new Color(rgb, rgb, rgb));
    g.drawLine(i, 10, i, h - 11);
}
```



Ellips

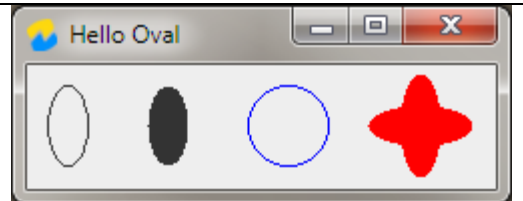
```
drawOval(int x, int y, int width, int height)
fillOval(int x, int y, int width, int height)
```

Metoderna ovan ritar en ellips som ryms inuti den rektangel som börjar i punkt (x,y) och är width pixlar bred och height pixlar hög.

```
g.drawOval(10, 10, 20, 40);
g.fillOval(60, 10, 20, 40);

g.setColor(Color.BLUE);
g.drawOval(110, 10, 40, 40);

g.setColor(Color.RED);
g.fillOval(170, 20, 52, 20);
g.fillOval(186, 4, 20, 52);
```



Månghörning

```
drawPolygon(int[] xPoints, int[] yPoints, int nPoints)
fillPolygon(int[] xPoints, int[] yPoints, int nPoints)
```

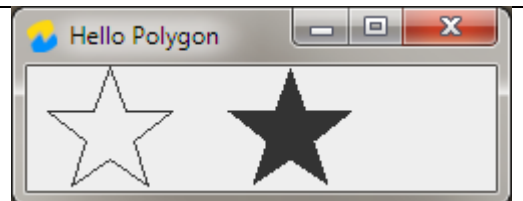
Metoderna ovan ritar en månghörning genom att rita en linje mellan punkterna som anges av arrayerna xPoints och yPoints. En linje ritas mellan första och sista punkten om dessa inte är samma punkt. Endast nPoints av punkterna i arrayerna används (normalt anges här hela längden av arrayen).

I exemplet nedan har vi i arrayerna xPoints och yPoints angett punkter för att rita en 5-uddig stjärna. Punkterna i arrayen utgår från (0,0). När vi sen ska rita ut dessa används translate för att placera första stjärnan 10 pixlar från vänstra kanten. När det sen är dags att rita den andra stjärnan gör vi ytterligare ett anrop till translate. Fördelen med att använda translate är att vi slipper en ny heltalsarray för andra stjärnan utan kan återanvända samma.

```
int[] xPoints =
    {0, 23, 31, 39, 62, 43, 50, 31, 12, 19};
int[] yPoints =
    {22, 22, 0, 22, 22, 37, 59, 46, 59, 37};
int nPoints = xPoints.length;

g.translate(10,0);
g.drawPolygon(xPoints, yPoints, nPoints);

g.translate(90,0);
```



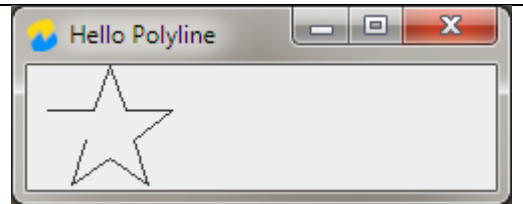
```
g.fillPolygon(xPoints, yPoints, nPoints);
```

Sammansatt linje

```
drawPolyline(int[] xPoints, int[] yPoints, int nPoints)
```

Metoden ovan påminner till stor del drawPolygon. Skillnaden är att ingen linje ritas mellan första och sista punkten om dessa inte är samma. Notera att det inte finns någon metod fillPolyline.

```
int[] xPoints =  
    {0, 23, 31, 39, 62, 43, 50, 31, 12, 19};  
int[] yPoints =  
    {22, 22, 0, 22, 22, 37, 59, 46, 59, 37};  
int nPoints = xPoints.length;  
  
g.translate(10,0);  
g.drawPolyline(xPoints, yPoints, nPoints);
```

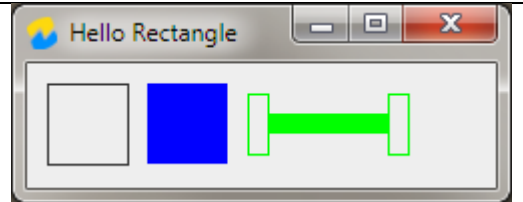


Rektangel

```
drawRect(int x, int y, int width, int height)  
fillRect(int x, int y, int width, int height)
```

Metoderna ovan ritar en rektangel vars övre vänstra hörn placeras i punkten (x,y). Rektangeln är width pixlar bred och height pixlar hög.

```
g.drawRect(10, 10, 40, 40);  
g.setColor(Color.BLUE);  
g.fillRect(60, 10, 40, 40);  
g.setColor(Color.GREEN);  
g.drawRect(110, 15, 10, 30);  
g.fillRect(120, 25, 60, 10);  
g.drawRect(180, 15, 10, 30);
```

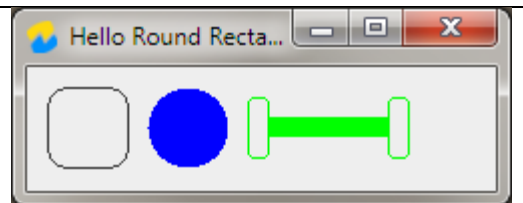


Rektangel med runda hörn

```
drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)  
fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)
```

Metoderna ovan ritar en rektangel vars övre vänstra hörn placeras i punkten (x,y). Rektangeln är width pixlar bred och height pixlar hög. Rundningen av de fyra hörnen bestäms av arcWidth och arcHeight.

```
g.drawRoundRect(10, 10, 40, 40, 20, 20);  
g.setColor(Color.BLUE);  
g.fillRoundRect(60, 10, 40, 40, 40, 40);  
g.setColor(Color.GREEN);  
g.drawRoundRect(110, 15, 10, 30, 5, 5);  
g.fillRoundRect(120, 25, 60, 10, 0, 0);  
g.drawRoundRect(180, 15, 10, 30, 5, 5);
```



Text

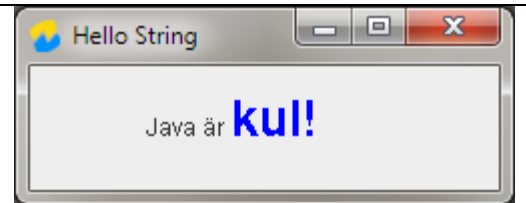
```
drawString(String str, int x, int y)
```

Metoden ovan ritar ut den text som anges av str. Texten placeras så att baslinjen (den linje på vilka de flesta bokstäver "står") för det första tecknet i strängen placeras på punkten (x,y). I exemplet nedan visas även hur vi kan ange vilket teckensnitt som ska användas. När vi skapar ett nytt Font-objekt måste vi ange namnet på teckensnittet, vilken stil den ska ha (PLAIN, BOLD eller ITALIC) och vilken teckenstorlek det ska vara. Vi skapar vårt Font-objekt genom att ta reda på namnet på nuvarande teckensnitt som används och därefter anger stilen till fet och storleken till 25.

```
g.drawString("Java är", 58, 35);

Font f = new Font(
    g.getFont().getName(), Font.BOLD, 25);

g.setFont(f);
g.setColor(Color.BLUE);
g.drawString("kul!", 100, 35);
```



När vi ritar text är det ofta nödvändigt att ta reda på hur stor yta texten upptar på ritytan med den font som för tillfället används. Detta för att kunna placera flera olika texter i förhållande till varandra. För att få hjälp med detta används klassen FontMetrics. Denna klass har ett stort antal metoder som kan anropas för att på olika sätt mäta den text som ska skrivas ut. Det är framför allt två av dessa som är av intresse.

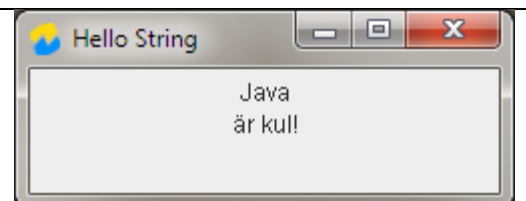
```
getHeight()
stringWidth(String str)
```

Den första metoden returnerar den totala höjden av texten och den andra metoden returnerar den totala bredden eller längden av texten. För att få ett FontMetrics-objekt anropar vi metoden getFontMetrics i klassen Graphics. Sätter vi ett nytt teckensnitt med metoden setFont måste vi återigen anropa getFontMetrics så att den baseras på det nya teckensnittet.

Vi kan till exempel använda FontMetrics för att centrera texten i en komponent samt skriva flera rader med text under varandra.

```
String line1 = "Java";
String line2 = "är kul!";
FontMetrics fm = g.getFontMetrics();
int textHeight = fm.getHeight();
int line1Width = fm.stringWidth(line1);
int line2Width = fm.stringWidth(line2);

g.drawString(line1, getWidth() / 2 -
    line1Width / 2, textHeight);
g.drawString(line2, getWidth() / 2 -
    line2Width / 2, textHeight * 2);
```



Fler metoder finner du i Javas API för klassen Graphics.

Skapa egna komponenter

Som du sett finns det ett stort antal färdiga komponenter i Swing som vi kan använda oss av när vi konstruerar vårt grafiska användargränssnitt. Utöver dessa komponenter är det även möjligt att själv skapa komponenter som vi kan lägga till i gränssnittet. Det finns i huvudsak två olika sätt att skapa

en komponent.

1. Skapa en komponent genom att ärva egenskaper från en befintlig komponent.
2. Skapa en komponent från "grunden" genom att använda en JPanel.

Givetvis finns det fler sätt att skapa komponenter på, men dessa två är ett enkelt och användbart sätt att göra det på.

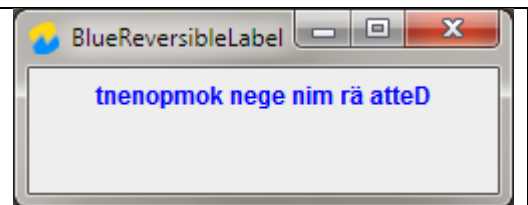
Ärva befintlig komponent

Med det första sättet kan vi ändra en befintlig komponents egenskaper genom att överskugga metoder från klassen som ärvs eller lägga till helt nya egenskaper genom att implementera nya metoder. Som ett exempel på detta kan vi skapa en JLabel vars text alltid är blå och där vi via ett metदानrop kan reversera texten.

```
public class BlueReversibleLabel extends JLabel {  
  
    public BlueReversibleLabel(String text) {  
        // Anropa superklassen så att JLabel initieras korrekt  
        super(text);  
        // Sätt förgrundsfärg till blått  
        setForeground(Color.BLUE);  
    }  
  
    public void reverse() {  
        // Använder StringBuffer för att enkelt reversera texten  
        StringBuffer text = new StringBuffer(getText());  
        text.reverse();  
        setText(text.toString());  
    }  
  
    @Override  
    public void setForeground(Color color) {  
        // Använd alltid blå färg  
        super.setForeground(Color.BLUE);  
    }  
}
```

Denna komponent kan vi nu använda i vårt GUI precis som vilken annan komponent som helst.

```
BlueReversibleLabel label;  
label = new BlueReversibleLabel("Detta är min  
egen komponent");  
label.reverse();  
label.setForeground(Color.RED);  
  
// Lägg till komponenten i fönstret  
add(label);
```



Skapa komponent från "grunden"

Det andra sättet att skapa en komponent på är egentligen en variant på det första. Vi skapar en klass som ärver JPanel och utgår därmed ju faktiskt från en befintlig komponent. Skillnaden är att denna komponent är tom. Fördelen med att använda en JPanel som utgångspunkt är dels att JPanel är en container och att vi därmed kan sätta layouthanterare och lägga till andra komponenter i den och dels att vi kan använda den som en rityta för att rita frihandsgrafik (eller en kombination av båda).

I lektion 4 pratade vi om hur vi bör skissa vårt GUI på ett papper och fundera över vilka ”delfönster” användargränssnittet består av. Varje delfönster använder sen en egen JPanel. Här är det vanligt att låta varje delfönster implementeras som en egen komponent som ärver JPanel. Fördelen är bland annat att vi delar upp koden i fler klasser och att vi kan låta varje delfönster själv hantera sin logik.

I lektion 4 skapade vi ett applikationsfönster som innehöll ett formulär för inmatning av kunddata till SmåStålar AB. Detta formulär kan vi implementera som en egen komponent där vi lägger till logik för att skapa ett Customer-objekt av inmatad data samt kontroll av obligatorisk data.

```
public class CustomerFormPanel extends JPanel {
    // Den text som ska stå i varje JLabel
    private String[] labels = {"Förnamn", "Efternamn", "Personnummer",
        "Adress", "Postnummer", "Postort", "E-post"};

    // De index som ska vara mnemonic i varje JLabel
    private int[] mnemonicIndex = {0, 0, 0, 0, 4, 4, 2};

    // De textfält som är obligatoriska att fylla i
    private int[] requiredFields = {0, 1, 2};

    // Array med alla JLabel
    private JLabel[] jLabels = new JLabel[labels.length];

    // Array med alla JTextField
    private JTextField[] jTextFields = new JTextField[labels.length];

    public CustomerFormPanel() {
        // Ange vilken layout som ska användas
        SpringLayout layout = new SpringLayout();
        setLayout(layout);

        // Initiera alla komponenter
        initComponents();
    }

    private void initComponents() {
        // Skapa alla komponenter
        for (int i = 0; i < labels.length; i++) {
            // Skapa JLabel och lägg till i panelen
            jLabels[i] = new JLabel(labels[i] + ":", JLabel.TRAILING);
            add(jLabels[i]);

            // Skapa JTextField och lägg till i panelen
            jTextFields[i] = new JTextField(10);
            jLabels[i].setLabelFor(jTextFields[i]);
            jLabels[i].setDisplayedMnemonic(labels[i].charAt(mnemonicIndex[i]));
            add(jTextFields[i]);
        }

        SpringUtilities.makeCompactGrid(
            this,
            labels.length, 2, // rows, cols
            6, 6, // initX, initY
            6, 6); // xPad, yPad
    }

    public boolean validateFormData() {
        boolean valid = true;

        // Kontrollera att det finns text i obligatoriska fält
    }
```

```
for (int i = 0; i < requiredFields.length; i++) {
    JTextField t = jTextFields[requiredFields[i]];

    // Om ingen text, sätt röd kant och begär fokus
    if (t.getText().length() == 0) {
        valid = false;
        t.setBorder(BorderFactory.createLineBorder(Color.RED, 2));
        t.requestFocus();
    }
    else {
        // Återställ kant till ursprunglig L&F
        t.setBorder(null);
        t.updateUI();
    }
}

// I stället för att returnera true/false bör vi kasta ett
// undantag om valideringen inte går igenom (ex. FormDataException)
return valid;
}

public Customer getCustomer() {
    Customer c = null;

    // Skapa ett Customer-objekt om valideringen är ok
    if (validateFormData()) {
        c = new Customer(jTextFields[0].getText(), jTextFields[1].getText(),
            jTextFields[2].getText(), jTextFields[3].getText(),
            jTextFields[4].getText(), jTextFields[5].getText());
    }

    // Returnerar null om valideringen inte är ok
    return c;
}
}
```

Det som är nytt här jämfört med exemplet i lektion 4 är först och främst att formuläret nu ligger i en egen klass som ärver JPanel. Vidare har vi lagt till en heltalsarray `requiredFields` i vilken vi anger index på de textfält som är obligatoriska att fylla i (förnamn, efternamn och personnummer). I konstruktorn anger vi vilken layout som ska användas och i `initComponents` skapar vi alla komponenter. I dessa skiljer inget från lektion 4.

I metoden `validateFormData` loopar vi `requiredFields` och för varje obligatoriskt textfält kontrollerar vi om längden på inmatad text är 0 (noll). Om så är fallet finns ingen text inmatad och vi sätter en röd kant (`Border`) runt textfältet för att tydligt indikera att textfältet måste fyllas i. Vidare ger vi textfältet fokus så att användaren direkt kan fylla i data utan att själv behöva klicka i textfältet med musen.

Förnamn:	<input type="text"/>
Efternamn:	<input type="text" value="Karlsson"/>
Personnummer:	<input type="text"/>
Adress:	<input type="text"/>
Postnummer:	<input type="text"/>
Postort:	<input type="text"/>
E-post:	<input type="text" value="kalle.karlsson@java"/>

Om data är inmatad (längden på texten är större än 0) återställer vi kanten runt textfältet till den kant som normalt används av nuvarande look-and-feel. Vi gör detta genom att sätta kanten till null och därefter anropa metoden `updateUI` som uppdaterar komponenten enligt den look-and-feel som används.

Metoden `validateFormData` låter vi returnera `true` om alla obligatoriska textfält är ifyllda och `false` om ett av dem saknar data. Här bör man i stället använda undantagshantering och kasta ett undantag om data saknas i ett textfält.

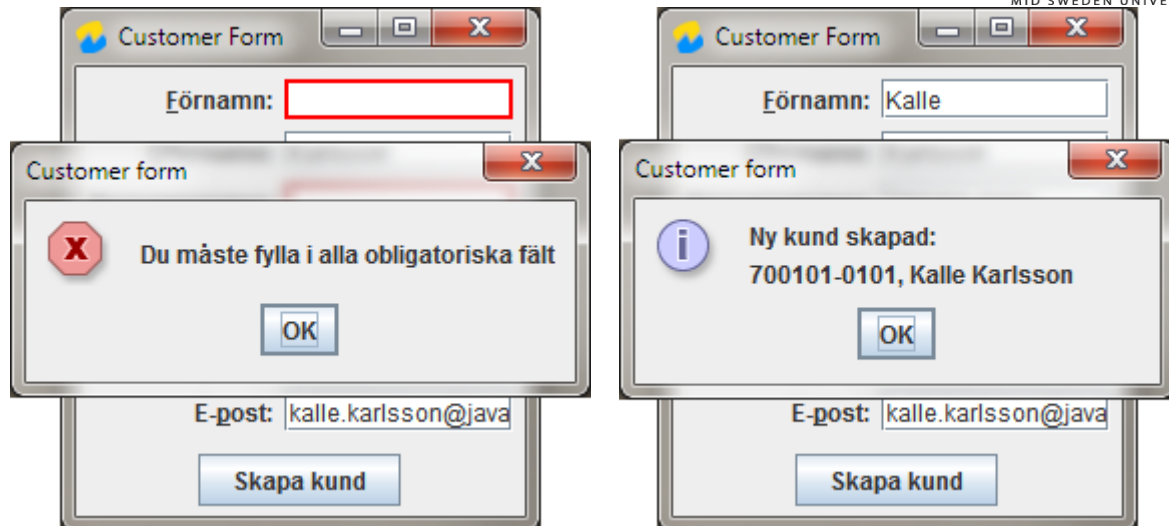
I metoden `getCustomer` validerar vi först formuläret genom ett anrop till `validateFormData`. Om detta gick bra (`true` returneras) skapar vi ett `Customer`-objekt, med data från alla textfälten, och returnerar objektet. Om valideringen misslyckas returnerar metoden `null`.

Det är relativt snabbt och enkelt gjort att utöka klassen så att vi kan använda formuläret även för att uppdatera data för en befintlig kund. Till exempel genom att lägga till en metod `setCustomer(Customer c)` i klassen. Använd sen `Customer`-objektet för att fylla textfälten med data från objektet.

För att nu använda denna komponent i vårt GUI skapar vi ett nytt `CustomerFormData`-objekt och lägga till den i till exempel en `JFrame`.

```
JFrame frame = new JFrame("Customer form");
CustomerFormPanel customerForm = new CustomerFormPanel();
frame.add(customerForm, BorderLayout.PAGE_START);
frame.setVisible(true);
...
Customer newCustomer = customerForm.getCustomer();
```

Gör detta förslagsvis tillsammans med en `JButton` som när den trycks på anropar metoden `getCustomer`. Undersök om det returnerade objektet refererar till `null` vilket innebär att alla obligatoriska textfält inte är ifyllda. Meddela i så fall användaren med hjälp av till exempel dialogrutor.



JPanel som rityta

Även om det som sagt är möjligt att rita frihandsgrafik i alla komponenter i Swing (som ärver JComponent) är det framför allt JPanel som används för detta. Vi skapar en ny klass som ärver JPanel och överskuggar sen metoden paintComponent.

```
public class MyPaintPanel extends JPanel {
    @Override
    public void paintComponent(Graphics g) {
        // Se till att ev. bakgrund och ram m.m. ritas om först
        super.paintComponent(g);
        // egen kod för att rita
    }
}
```

I metoden paintComponent använder vi Graphics-objektet för att rita det som ska synas i panelen. Det första vi alltid bör göra i metoden är ett anrop till superklassens paintComponent. Detta anrop ser till att eventuell bakgrund i panelen (om den inte är genomskinlig) ritas om samt att eventuell ram och alla andra komponenter som är inlagda i panelen ritas om på ett korrekt sätt.

Metoden paintComponent anropas automatiskt så snart den behöver ritas om av någon anledning (storleken kanske har ändrats eller den visas igen efter att ha varit dold av något annat). Vi ska själva aldrig anropa denna metod utan behöver vi manuellt rita om panelen anropar vi metoden repaint istället.

Som ett exempel skapar vi en JPanel i vilken vi ritar en pacman-figur som hela tiden upptar så stor yta som möjligt. Varje gång användaren klickar i panelen låter vi pacman-figuren växelvis stänga och öppna munnen.

Vi börjar med att skapa en klass som representerar själva pacman-figuren. Klassens instansvariabler lagrar information om figurens x- och y-koordinater, dess storlek och om munnen ska vara öppen eller stängd.

```
public class Pacman {
    private int x;    // x-koordinat
    private int y;    // y-koordinat
    private int size; // diameter i antal pixlar
    private boolean mouthOpen;
```

```
public Pacman() {
    x = y = 0;
    size = 40;
    mouthOpen = true;
}

public void setLocation(int x, int y) {
    this.x = x;
    this.y = y;
}

public void setSize(int size) {
    this.size = size;
}

public void toggleMouthOpen() {
    mouthOpen = !mouthOpen;
}

public void draw(Graphics g) {
    // Vinklar för öppen mun
    int startAngle = 30;
    int arcAngle = 300;

    if (!mouthOpen) {
        // Andra vinklar för stängd mun
        startAngle = 10;
        arcAngle = 340;
    }

    // Kroppen
    g.setColor(Color.YELLOW);
    g.fillArc(x - size / 2, y - size / 2, size, size, startAngle, arcAngle);

    // Svart kontur runt kroppen
    g.setColor(Color.BLACK);
    g.drawArc(x - size / 2, y - size / 2, size, size, startAngle, arcAngle);
    // Lite trigonometri för att rita munnens kontur
    g.drawLine(x, y,
        x + (int)(size / 2 * Math.cos(Math.toRadians(startAngle))),
        y + (int)(size / 2 * Math.sin(Math.toRadians(startAngle))));
    g.drawLine(x, y,
        x + (int)(size / 2 * Math.cos(Math.toRadians(arcAngle + startAngle))),
        y + (int)(size / 2 * Math.sin(Math.toRadians(arcAngle + startAngle))));

    // Ögat
    int eyeSize = size / 6;
    g.fillOval(x + size / 10, y - (int)(size / 2.5), eyeSize, eyeSize);
}
}
```

I konstruktorn sätter vi godtyckliga värden för figurens position, storlek och om munnen ska vara öppen eller stängd. Med metoden `setLocation` kan vi ange en ny position för figuren. Vill vi ge figuren en ny storlek anropar vi `setSize`. Metoden `toggleMouthOpen` ändrar om munnen är öppen eller stängd. Om munnen redan var öppen resulterar anropet i att munnen stängs och tvärtom.

Den intressanta metoden är `draw` (vi hade kunnat döpa metoden till vad som helst) som tar ett `Graphics`-objekt som argument. Med hjälp av detta `Graphics`-objekt ritas vi hur pacman-figuren ska se ut. Var på skärmen, dess storlek och om munnen ska vara öppen eller stängd avgör respektive

instansvariabel.

För att rita ut en Pacman i en panel skapar vi en ny klass som ärver JPanel.

```
public class PacmanPanel extends JPanel {
    private Pacman pacman;

    public PacmanPanel() {
        // Skapa en pacman
        pacman = new Pacman();

        // Lägg till en muslyssnare
        addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                // stäng/öppna mun och rita om
                pacman.toggleMouthOpen();
                repaint();
            }
        });
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        // Placera vår pacman mitt i panelen
        pacman.setLocation(getWidth() / 2, getHeight() / 2);

        // Sätt storlek på pacman till största möjliga
        pacman.setSize(Math.min(getWidth(), getHeight()) - 10);

        // Rita vår pacman
        pacman.draw(g);
    }
}
```

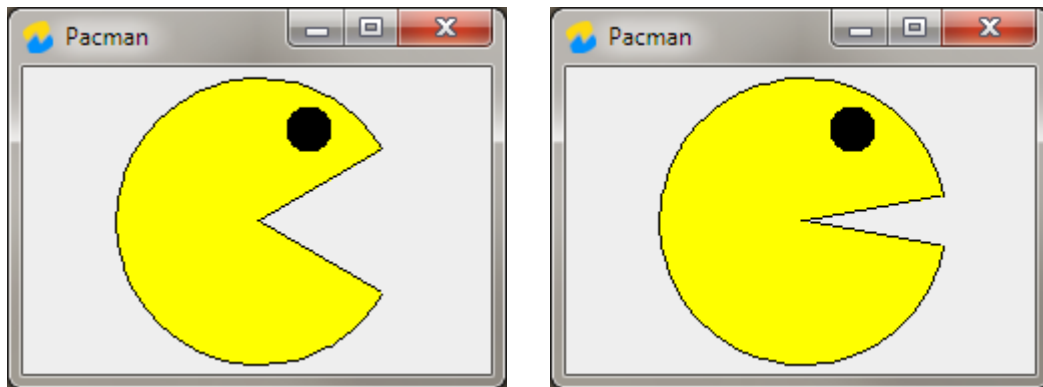
I panelens `paintComponent` placerar vi vår pacman mitt i panelen genom att tar reda på panelens bredd och höjd och dividera dessa värden med 2. Storleken sätts till det minsta värdet av panelens bredd och höjd (samt minskar ytterligare med 10 pixlar för att få en liten marginal till kanten). För att sen rita vår pacman anropar vi dess `draw`-metod och skickar med `Graphics`-objektet vi får från `paintComponent`. Figuren ritas då ut i panelen.

Vi har lagt till en `MouseListener` i panelen (`ActionListener` finns inte för `JPanel`) och implementerat den som en `MouseAdapter`. Anledningen är att vi endast är intresserad av när användaren klickar med musknappen (`mouseClicked`). Vi hade lika gärna kunnat låta klassen implementera gränssnittet `MouseListener`, men då hade vi varit tvungna att överskugga alla fem metoder i gränssnittet. När användaren klickar i panelen anropar vi `toggleMouthOpen` för att växla mellan öppen och stängd mun. För att panelen ska ritas om efter att munnen har ändrats anropar vi `repaint`. Detta anrop leder i sin tur att `paintComponent` anropas och figuren uppdateras.

Precis som i fallet med `CustomerFormPanel` skapar vi nu ett nytt `PacmanPanel`-objekt och lägger till den i en `JFrame`.

```
JFrame frame = new JFrame("Pacman");
PacmanPanel pacmanPanel = new PacmanPanel();
frame.add(pacmanPanel, BorderLayout.CENTER);
```

```
frame.setVisible(true);
```



Graphics2D

Som du har sett är det väldigt primitiva figurer vi kan rita med metoderna i klassen Graphics. Linjer är till exempel alltid en pixel bred och färger är alltid solida. Trots det klarar man sig faktiskt ganska långt med det Graphics erbjuder. För de som är intresserade av lite mer avancerade möjligheter att rita geometriska figurer finns klassen Graphics2D. Denna klass är en subclass till Graphics och förutom fler och mer avancerade figurer kan vi med klassen Graphics2D även rita linjer med olika tjocklek och utseende, vi kan ange färger som är delvis transparenta (genomskinliga) och fylla figurer med färger som skiftar i nyans eller mönster.

För att få en referens till ett Graphics2D-objekt kan vi göra en typomvandling på ett befintligt Graphics-objekt, till exempel det vi använder i metoden paintComponent:

```
public void paintComponent(Graphics g) {  
    super.paintComponent(g);  
    Graphics2D g2 = (Graphics2D)g;  
    // använd g2 istället för g för att rita  
}
```

Sättet att rita figurer med Graphics2D skiljer sig en del från att rita med Graphics. Med Graphics2D definieras geometriska figurer av gränssnittet Shape (i paketet java.awt). Vi kommer inte att gå in på djupet vad gäller hur vi ritar med Graphics2D utan nöjer oss med att visa det mest grundläggande. I korta drag skapar vi ett objekt av en klass som implementerar gränssnittet Shape. Vi ställer sen in de egenskaper objektet ska ha (position, storlek, färg, fyllning, kantlinje m.m.) och ritar sen objektet genom att anropa metoden draw eller fill i klassen Graphics2D och skickar objektet som argument till dessa metoder.

De klasser som implementerar gränssnittet Shape är bland annat: Arc2D, CubicCurve2D, Ellipse2D, Line2D, Path2D, QuadCurve2D, Rectangle2D, RoundRectangle2D samt Area och GeneralPath. Notera alla klasser vars namn slutar på 2D. Dessa är abstrakta klasser och vi kan därför inte skapa objekt direkt från dessa klasser. I stället har dessa klasser två subclasser som vi använder. Subklasserna har klassnamn som slutar på .Float och .Double och skillnaden mellan dessa klasser är vilken precision de använder internt för att lagra position och storlek med mera.

Subklasserna är definierade som publika och statiska klasser inuti respektive abstrakt basklass (så kallade nästlade klasser). Den abstrakta klassen Line2D har således subclasserna Float och Double vilka vi kommer åt genom att skriva Line2D.Float samt Line2D.Double. Samma gäller för övriga klasser.

För de flesta subklasser ovan lagras punkter som kontrollerar figurens position och storlek i publika instansvariabler. Den publika instansvariabeln `x1` och `y1` används till exempel för de flesta figurer för att lagra figurens start punkt (x,y). Deklarerar vi en geometrisk figur där typen är av superklassen måste vi använda get- och set-metoderna för att komma åt figurens värden, men om vi däremot deklarerar en figur där typen är någon av subklasserna (`.Float` eller `.Double`) kan vi direkt använda de publika instansvariablerna.

För att skapa en `Line2D` och rita ut den kan vi skriva:

```
Line2D line1 = new Line2D.Double(2, 2, 6, 6);
Line2D.Double line2 = new Line2D.Double(6, 6, 10, 12);
g2.draw(line1);
g2.draw(line2);
double line1X1 = line1.getX1();
double line2X1 = line2.x1;
```

Notera ovan att `line1` deklaras som en `Line2D` medan `line2` deklaras som en `Line2D.Double`. Detta gör att vi med `line2` direkt kommer åt instansvariabeln `x1`, medan vi för `line1` måste använda metoden `getX1` för att komma åt värdet på instansvariabeln `x1`.

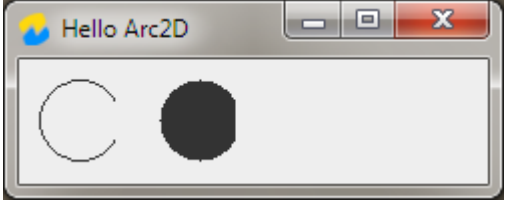
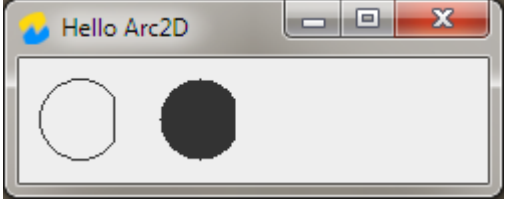
Vanligtvis använder vi subklassen `Double` för att skapa våra geometriska figurer. `Float` brukar vi normalt enbart använda när vi hanterar väldigt många geometriska figurer (eftersom vi då kan spara en hel del minne).

De geometriska figurerna `Ellipse2D`, `Line2D`, `Rectangle2D` och `RoundRectangle2D` fungerar i princip på samma sätt som motsvarande metoder i klassen `Graphics`. Dessa klasser tittar vi därför inte på i detalj här.

Båge

I `Graphics2D` representeras en båge av klassen `Arc2D`. Den enda skillnaden jämfört med hur vi ritar bågar med `Graphics` är att vi nu kan ange om, och i så fall hur, bågens start- och slutpunkter ska slutas. Vi anger detta som ett argument till konstruktorn när `Arc2D` skapas. Vi använder någon av följande konstanter (i klassen `Arc2D`):

- `OPEN` – Slut inte ändpunkterna.
- `CHORD` – Slut ändpunkterna genom att rita en linje från start- till slutpunkt.
- `PIE` – Slut ändpunkterna genom att rita en linje från bågens startpunkt till bågens centrum och en linje från bågens centrum till bågens slutpunkt.

<pre>Arc2D arc = new Arc2D.Double (10, 10, 40, 40, 30, 300, Arc2D.OPEN); g2.draw(arc); g2.translate(60, 0); g2.fill(arc);</pre>	
<pre>Arc2D arc = new Arc2D.Double (10, 10, 40, 40, 30, 300, Arc2D.CHORD); g2.draw(arc); g2.translate(60, 0); g2.fill(arc);</pre>	


```
Arc2D arc = new Arc2D.Double (
    10, 10, 40, 40, 30, 300, Arc2D.PIE);

g2.draw(arc);
g2.translate(60, 0);
g2.fill(arc);
```



Att rita konturen runt vår pacman-figur blir med andra ord betydligt enklare om vi använder Arc2D. Genom att använda Arc2D.PIE slipper vi använda trigonometri för att rita munnens linjer.

Kvadratisk kurva

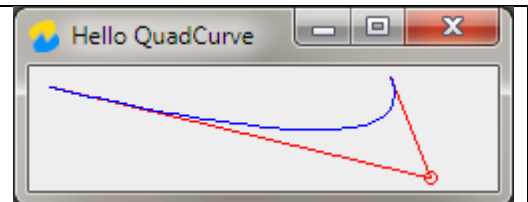
Med klassen QuadCurve2D kan vi skapa så kallade kvadratiske kurvor. En sådan kurva definieras genom att ange kurvans start- och slutpunkt samt en kontrollpunkt. Kontrollpunkten används för att ändra kurvans utseende. Koordinaterna för dessa punkter lagras i de publika instansvariablerna x1, y1, ctrlx, ctrly, x2 och y2. Kontrollpunkten ritas aldrig ut, men i exemplen nedan har vi valt att rita ut den tänkta kontrollpunkten samt linjer till kurvans start- och slutpunkt för att tydligare visa hur kontrollpunkten hänger ihop med kursvans utseende.

Kontrollpunkten och linjerna har skapats enligt följande:

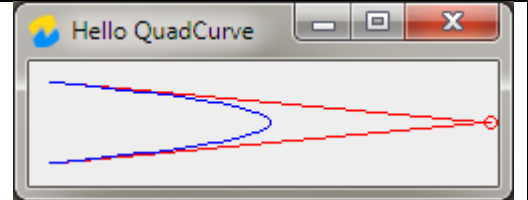
```
// Skapa kontrollpunkt och linjer
Line2D line1 = new Line2D.Double(curve.x1, curve.y1, curve.ctrlx, curve.ctrly);
Line2D line2 = new Line2D.Double(curve.x2, curve.y2, curve.ctrlx, curve.ctrly);
Ellipse2D ctrl = new Ellipse2D.Double (curve.ctrlx - 3, curve.ctrly - 3, 6, 6);

// rita kontrollpunkt och linjer
g2.setColor(Color.RED);
g2.draw(line1);
g2.draw(line2);
g2.draw(ctrl);
```

```
QuadCurve2D.Double curve = new
QuadCurve2D.Double (10, 10, 200, 55, 180, 5);
... // skapa kontrollpunkt och linjer
g2.setColor(Color.BLUE);
g2.draw(curve);
... // rita kontrollpunkt och linjer
```



```
QuadCurve2D.Double curve = new
QuadCurve2D.Double (10, 10, 230, 30, 10, 50);
... // skapa kontrollpunkt och linjer
g2.setColor(Color.BLUE);
g2.draw(curve);
... // rita kontrollpunkt och linjer
```



Kubisk kurva

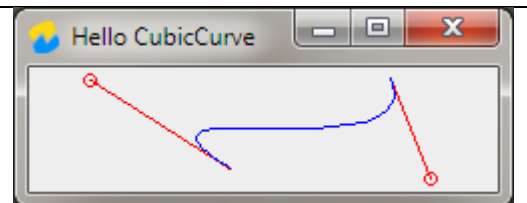
Med klassen CubicCurve2D kan vi skapa så kallade kubiska kurvor. En kubisk kurva påminner mycket om en kvadratisk kurva. Skillnaden är att vi här har två kontrollpunkter i stället för en. Den ena kontrollpunkten "hör ihop" med startpunkten och den andra kontrollpunkten "hör ihop" med slutpunkten. Koordinaterna för kurvans start-, slut- och kontrollpunkter lagras i de publika instansvariablerna x1, y1, x2, y2, ctrlx1, ctrly1, ctrlx2 och ctrly2. Som i föregående exempel har vi även här valt att rita ut kontrollpunkterna och linjer till kurvans start- och slutpunkt.

Kontrollpunkterna och linjerna har skapats enligt följande:

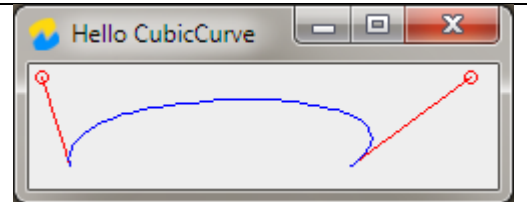
```
// Skapa kontrollpunkt och linjer
Line2D line1 = new Line2D.Double(curve.x1, curve.y1, curve.ctrlx1, curve.ctrly1);
Line2D line2 = new Line2D.Double(curve.x2, curve.y2, curve.ctrlx2, curve.ctrly2);
Ellipse2D ctrl1 = new Ellipse2D.Double(curve.ctrlx1 - 3, curve.ctrly1 - 3, 6, 6);
Ellipse2D ctrl2 = new Ellipse2D.Double(curve.ctrlx2 - 3, curve.ctrly2 - 3, 6, 6);

// rita kontrollpunkt och linjer
g2.setColor(Color.RED);
g2.draw(line1);
g2.draw(line2);
g2.draw(ctrl1);
g2.draw(ctrl2);
```

```
CubicCurve2D.Double curve = new CubicCurve2D.
Double (100, 50, 30, 6, 200, 55, 180, 5);
... // skapa kontrollpunkt och linjer
g2.setColor(Color.BLUE);
g2.draw(curve);
... // rita kontrollpunkt och linjer
```



```
CubicCurve2D.Double curve = new CubicCurve2D.
Double (20, 50, 6, 6, 220, 6, 160, 50);
... // skapa kontrollpunkt och linjer
g2.setColor(Color.BLUE);
g2.draw(curve);
... // rita kontrollpunkt och linjer
```



Bland exemplen som följer med lektionen finner du klassen CurveExample. Där ritas både en kvadratisk och kubisk kurva ut och du kan ta tag i start-, slut- och kontrollpunkterna och dra runt dessa för att se hur kurvorna ändras.

Sammanfattning

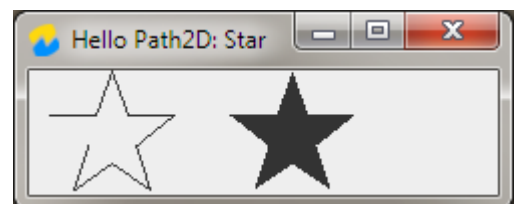
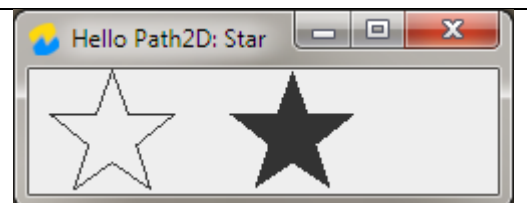
Path2D och GeneralPath (som är en subclass till Path2D.Float) kan användas för att efterlikna månghörningar (polygon) och sammansatta linjer (polyline) i Graphics. Exemplet nedan visar hur vi återskapar stjärnan från tidigare exempel med hjälp av Path2D.

```
double[] xPoints = {0, 23, 31, 39, 62, 43, 50,
31, 12, 19};
double[] yPoints = {22, 22, 0, 22, 22, 37, 59,
46, 59, 37};

Path2D star = new Path2D.Double();
star.moveTo(xPoints[0], yPoints[0]);

for (int i = 1; i < xPoints.length; i++) {
    star.lineTo(xPoints[i], yPoints[i]);
}

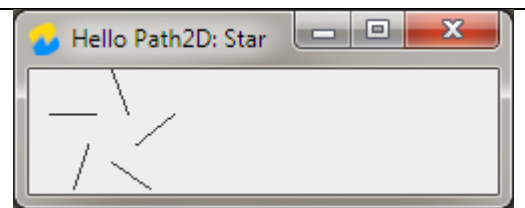
// Ta bort följande rad för ex. i bild 2
star.closePath();
g2.translate(10, 0);
g2.draw(star);
g2.translate(90, 0);
g2.fill(star);
```



För att ange startpunkten i en Path2D (och GeneralPath) anropar vi metoden `moveTo(x, y)`. Från denna punkt utgår nu den första linjen som ritas. För att rita en linje anropar vi metoden `lineTo(x, y)`. Som metodnamnet antyder innebär det att en linje ritas från nuvarande punkt till punkten som anges av `x` och `y`. Punkten (`x,y`) blir sen ny nuvarande punkt. När vi ritat alla linjer anropar vi metoden `closePath` som då drar en linje från nuvarande punkt till första punkten (som är den punkt som senaste anropet till `moveTo` angav). Tack vare anropet till `closePath` får vi en månghörning (polygon). Utesluter vi anropet till `closePath` får vi i stället en sammansatt linje (polyline).

Vi kan anropa metoden `moveTo` när som helst när vi skapar en Path2D, det behöver inte enbart vara för att ange figurens startpunkt. Det vill säga en sammansatt linje behöver inte vara sammanhängande utan kan bestå av flera separata delar. Om vi till exempel enbart vill rita ut varannan linje i stjärnan kan vi göra följande förändring i for-loopen. Att tänka på är att `closePath` som sagt ritas en linje till den punkt som senaste anropet till `moveTo` angav och inte till startpunkten.

```
for (int i = 1; i < xPoints.length; i++) {  
    if (i % 2 == 1) {  
        star.lineTo(xPoints[i], yPoints[i]);  
    }  
    else {  
        star.moveTo(xPoints[i], yPoints[i]);  
    }  
}
```

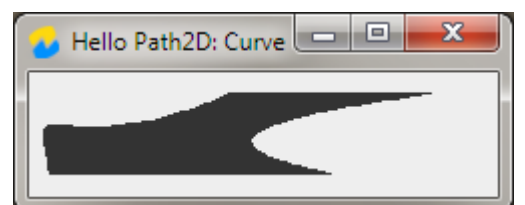
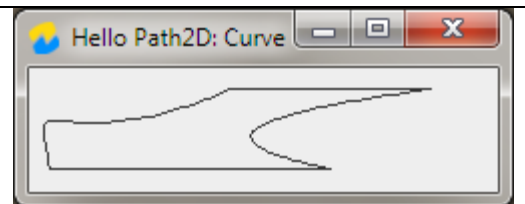


En stor skillnad med Path2D och GeneralPath jämfört med polygon och polyline i Graphics är att vi med Path2D kan använda både linjer, kurvor och andra geometriska figurer. I stället för `lineTo` anropar vi metoden `quadTo` för att lägga till en kvadratisk kurva och `curveTo` för att lägga till en kubisk kurva.

För att lägga till en valfri geometrisk figur anropar vi metoden `append`. Denna metod tar två argument där det första är en valfri geometrisk figur (Shape) och det andra är en boolean som anger om figuren ska sammanbindas med en linje (`lineTo`) till den punkt som senast angavs av `moveTo`. Är värdet `true` ritas en linje och är värdet `false` ritas ingen linje.

Följande exempel visar på användning av `lineTo`, `quadTo` och `curveTo`.

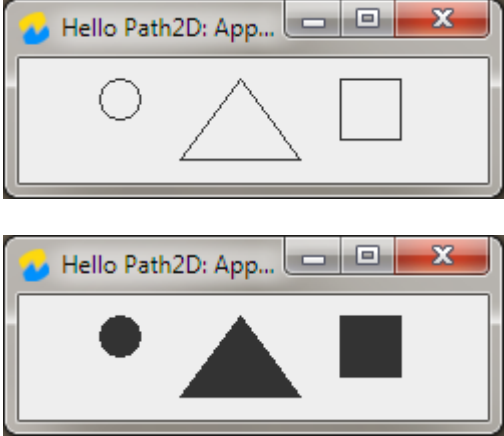
```
Path2D path = new Path2D.Double();  
  
path.moveTo(100, 10);  
path.lineTo(200, 10);  
path.quadTo(50, 30, 150, 50);  
path.lineTo(10, 50);  
path.curveTo(0, 0, 10, 50, 100, 10);  
  
// bild 1  
g2.draw(path);  
  
// bild 2  
g2.fill(path);
```



Först skapar vi ett `Path2D.Double`-objekt och flyttar startpunkten för nästa linje till (100, 10). Därefter ritas en linje från punkten (100, 10) till (200, 10). Från punkten (200, 10) ritas nu en kvadratisk kurva till punkten (150, 50) med en kontrollpunkt i (50, 30). Från slutpunkten (150, 50) på den kvadratiske kurvan ritas en linje till punkten (10, 50). Till sist ritas en kubisk kurva från

punkten (10, 50) till punkten (100, 10), som ju var vår utgångspunkt). För den kubiska kurvan används kontrollpunkterna (0, 0) och (10, 50).

Följande exempel visar på användning av append för att lägga till valfria figurer.

<pre>Rectangle2D square = new Rectangle2D.Double (160, 10, 30, 30); Ellipse2D circle = new Ellipse2D.Double(40, 10, 20, 20); Path2D path = new Path2D.Double(); path.moveTo(110, 10); path.lineTo(140, 50); path.lineTo(80, 50); path.closePath(); // triangel path.append(square, false); path.append(circle, false); g2.draw(path); // bild 1 g2.fill(path); // bild 2</pre>	
---	--

Vi börjar med att skapa en fyrkant (Rectangle2D) och en cirkel (Ellipse2D). Därefter använder vi en Path2D för att skapa en triangel. Cirkeln och kvadraten läggs tillsist in i Path2D genom anrop till append. Dessa figurer får den position och storlek som vi angav när figurerna skapades. Path2D kan användas för att gruppera samman olika figurer som hör ihop med varandra och som ska hanteras gemensamt (fyllas med samma färg, flyttas eller roteras m.m.).

Färger

I klassen Graphics anger vi vilken färg som ska användas när figurer ritas genom att anropa metoden setColor och som argument ange det Color-objekt som ska användas. Även om vi i Graphics2D kan använda samma metod för att ange en färg bör vi istället använda metoden setPaint. Denna metod tar som argument ett objekt av typen Paint (Paint är ett gränssnitt) och utöver att specificera en solid färg kan vi då även ange färger som skiftar i nyans eller mönster utifrån en bild.

Klassen Color implementerar gränssnittet Paint och vi kan därför använda vanliga Color-objekt i anropet till setPaint.

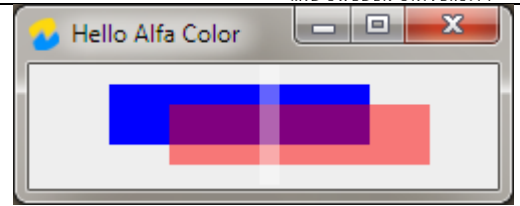
```
g2.setPaint(Color.BLUE);
g2.setPaint(new Color(200, 122, 32));
```

När vi skapar ett nytt Color-objekt anger vi färgens RGB-värden som ett heltal mellan 0 och 255. Dessa värden anger hur stor del av färgerna röd, grön och blå den nya färgen ska bestå av. Den färg som skapas är alltid helt solid. Det är möjligt att skapa färger som är helt eller delvis genomskinliga. Vi använder då en konstruktor i Color som tar ett fjärde argument. Detta argument anger färgens genomskinlighet och kallas för färgens alfavärde. Precis som med färgens RGB-värde är alfavärdet ett heltal mellan 0 och 255 där värdet 0 innebär en helt genomskinlig färg och värdet 255 innebär en helt ogenomskinlig färg (solid färg).

```
Rectangle2D r1 = new
    Rectangle2D.Double(40, 10, 130, 30);
Rectangle2D r2 = new
    Rectangle2D.Double(70, 20, 130, 30);
Rectangle2D r3 = new
    Rectangle2D.Double(115, 0, 10, 60);

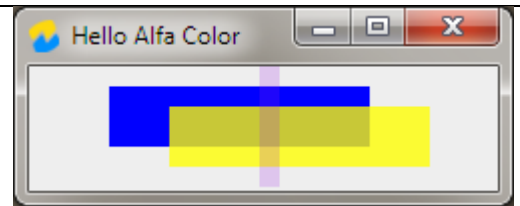
// Röd halvgenomskinlig färg
Color c1 = new Color(255, 0, 0, 128);
// Vit mer än halvgenomskinlig
Color c2 = new Color(255, 255, 255, 100);

g2.setPaint(Color.BLUE);
g2.fill(r1);
g2.setPaint(c1);
g2.fill(r2);
g2.setPaint(c2);
g2.fill(r3);
```



```
// Gul mindre än halv genomskinlig
Color c1 = new Color(255, 255, 0, 200);

// Lila väldigt genomskinlig
Color c2 = new Color(160, 32, 240, 50);
```



GradientPaint

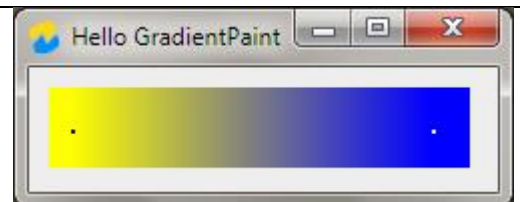
En annan klass som implementerar gränssnittet Paint är klassen GradientPaint. Med denna klass kan vi skapa en toning mellan en startfärg och slutfärg som sen kan användas för att fylla figurer. När vi skapar ett objekt av klassen anger vi en startpunkt och en färg för denna punkt samt en slutpunkt och en färg för slutpunkten. När en figur sen fylls med denna toning kommer färgen gradvis skifta från startfärgen till slutfärgen mellan start- och slutpunkten. Vardera sida om start- och slutpunkten kommer att fyllas med start- respektive slutfärgen. Om vi i stället vill att färgtoningen ska upprepas på vardera sida om start- och slutpunkten kan vi sist i konstruktorn lägga till true som argument.

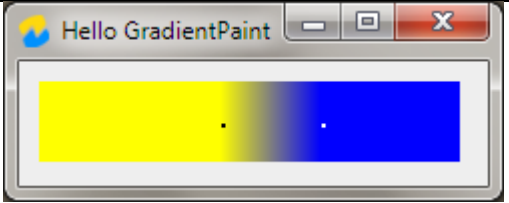
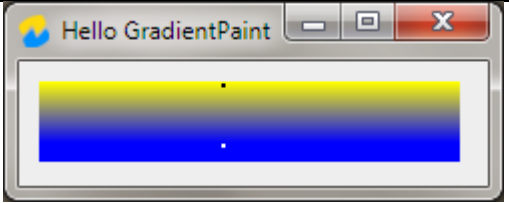
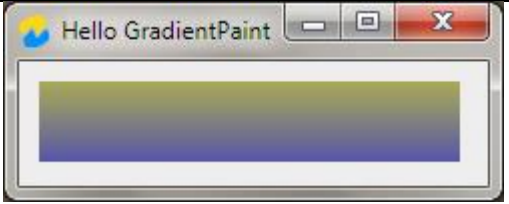
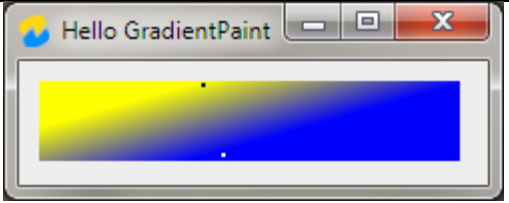
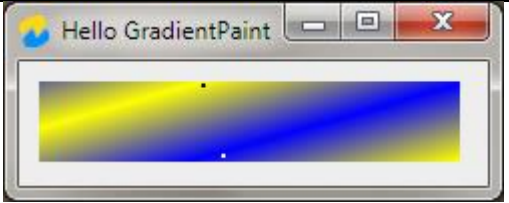
I exemplen nedan har vi ritat en svart punkt för att visa var startpunkten är och en vit punkt för att visa var slutpunkten är. Koden för att rita ut dessa punkter finns inte med nedan. Samma kod för att skapa Rectangle2D och GradientPaint samt för att rita och fylla rektangeln används i alla exempel. Det är endast värdena i x1, x2, y1 och y2 som skiljer de olika exemplen åt. Förutom i sista exemplet där en ny GradientPaint används.

```
int x1 = 20; int y1 = 30;
int x2 = 200; int y2 = 30;
Rectangle2D r1 = new Rectangle2D.Double(
    10, 10, 210, 40);

GradientPaint paint = new GradientPaint(
    x1, y1, Color.YELLOW, x2, y2, Color.BLUE);

g2.setPaint(paint);
g2.fill(r1);
```



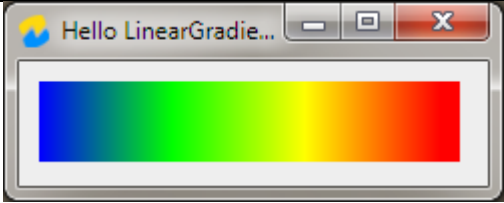
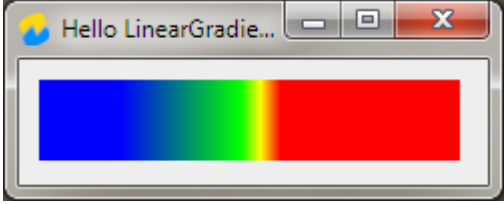
<pre>int x1 = 100; int y1 = 30; int x2 = 150; int y2 = 30;</pre>	
<pre>int x1 = 100; int y1 = 10; int x2 = 100; int y2 = 40;</pre>	
<pre>int x1 = 100; int y1 = -30; int x2 = 100; int y2 = 90;</pre>	
<pre>int x1 = 90; int y1 = 10; int x2 = 100; int y2 = 45;</pre>	
<pre>int x1 = 90; int y1 = 10; int x2 = 100; int y2 = 45; GradientPaint paint = new GradientPaint(x1, y1, Color.YELLOW, x2, y2, Color.BLUE, true);</pre>	

Som du ser i exempel 4 behöver inte start- och slutpunkten finnas inom det synliga området utan det går bra att ange dessa utanför. I det sista exemplet har vi i konstruktorn angett att toningen ska upprepas.

LinearGradientPaint

Från och med Java 6 (1.6) har det tillkommit ytterligare två klasser med vilka vi kan skapa färgtoningar. `LinearGradientPaint` fungerar i stort som `GradientPaint`, men med den skillnaden att vi kan skapa toningar mellan fler än två färger. För att använda klassen anger vi först start- och slutpunkt för toningen. Sen används en array av typen `Color` för att ange vilka färger vi ska tona mellan samt en array av typen `float` för att ange var, mellan start- och slutpunkten, dessa toningar ska börja. I float-arrayen anger vi värden mellan 0f och 1f inklusive. Värdet 0 innebär att toningen ska börja vid startpunkten och värdet 1 innebär att toningen ska börja vid slutpunkten. Värdet 0.5 innebär att toningen börjar precis mitt emellan start- och slutpunkten.

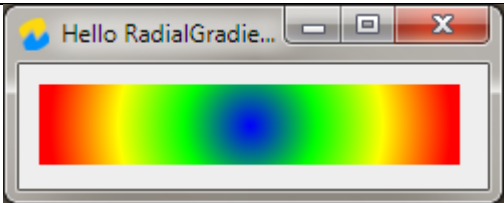
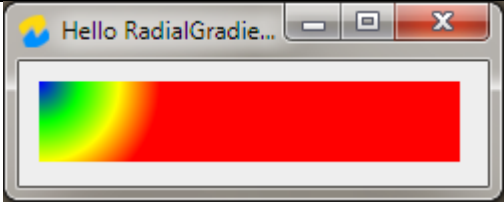
I exemplen nedan används samma kod för att skapa arrayen med färger, `Rectangle2D` och `LinearGradientPaint` samt för att rita och fylla rektangeln. Det är endast värdena i float-arrayen som skiljer sig åt.

<pre>Rectangle2D r1 = new Rectangle2D.Double(10, 10, 210, 40); Color[] colors = {Color.BLUE, Color.GREEN, Color.YELLOW, Color.RED}; float[] dist = {0f, 0.33f, 0.66f, 1f}; LinearGradientPaint paint = new LinearGradientPaint(10, 30, 210, 30, dist, colors); g2.setPaint(paint); g2.fill(r1);</pre>	
<pre>float[] dist = {0.2f, 0.50f, 0.55f, 0.6f};</pre>	

I första exemplet ovan skapas en LinearGradientPaint som tonar mellan blått och grönt de första 33% av längden mellan start- och slutpunkten. Den andra tredjedelen tonas mellan grönt och gult för att i den sista tredjedelen tonas mellan gult och rött. Precis som för GradientPaint finns det möjlighet att ange hur toningen eventuellt ska upprepas utan för start- och slutpunkten. Se Javas API för klassen LinearGradientPaint för mer information om detta.

RadialGradientPaint

I samband med LinearGradientPaint infördes även RadialGradientPaint. Vi skapar en sådan på i stort sett samma sätt som LinearGradientPaint genom att i två arrayer ange vilka färger som ska användas och var toningarna ska börja. Skillnaden är att vi endast anger en startpunkt (mittpunkt) och en radie i stället för en slutpunkt.

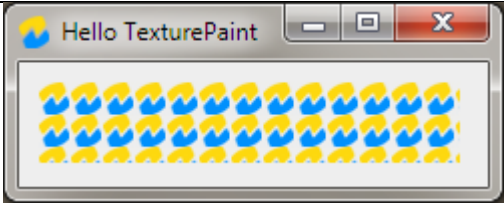
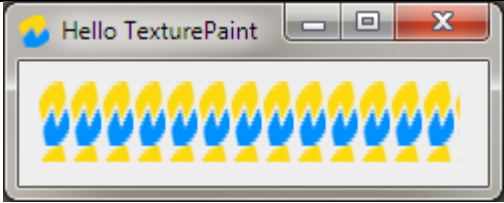
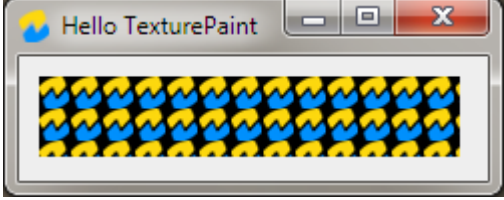
<pre>Rectangle2D r1 = new Rectangle2D.Double(10, 10, 210, 40); Color[] colors = {Color.BLUE, Color.GREEN, Color.YELLOW, Color.RED}; float[] dist = {0f, 0.33f, 0.66f, 1f}; RadialGradientPaint paint = new RadialGradientPaint(115, 30, 100, dist, colors); g2.setPaint(paint); g2.fill(r1);</pre>	
<pre>RadialGradientPaint paint = new RadialGradientPaint(10, 10, 60, dist, colors);</pre>	

I första exemplet ovan skapas en RadialGradientPaint som tonar mellan blått och grönt från

mittpunkten (115, 30) och de första 33% av radien (100) och så vidare. Precis som för LinearGradientPaint finns det möjlighet att ange hur toningen eventuellt ska upprepas utan för start- och slutpunkten. Det är även möjligt att ange var, i den cirkel som utgörs av mittpunkten och radien, toningen ska börja om den inte ska börja precis i centrum. Se Javas API för klassen RadialGradientPaint för mer information om detta.

TexturePaint

Den sista klassen som implementerar Paint är TexturePaint. Med denna klass kan vi använda en bild för att fylla en figur (Shape). Bilden kan vi antingen ladda från hårddisken, hämta från internet via en länk eller rita själv med Graphics2D. Bilden ska vara ett objekt av typen BufferedImage (paketet java.awt.image). När vi skapar en TexturePaint anger vi dels själva bilden och dels en rektangel (Rectangle2D). Med rektangeln definierar vi ett område i vilken bilden sen fylls ut i. Om detta område inte är lika stort som bilden skalas bilden om för att passa exakt i området. När vi sen använder TexturePaint för att fylla en figur kommer området med bilden att upprepas horisontellt och vertikalt inuti figuren.

<pre>BufferedImage image = null; try { image = ImageIO.read(new File("miun16x16.png")); } catch (IOException e) { } Rectangle2D anchor = new Rectangle2D.Double(10, 10, image.getWidth(), image.getHeight()); TexturePaint paint = new TexturePaint(image, anchor); Rectangle2D r1 = new Rectangle2D.Double(10, 10, 210, 40); g2.setPaint(paint); g2.fill(r1);</pre>	
<pre>Rectangle2D anchor = new Rectangle2D.Double(10, 10, image.getWidth(), image.getHeight() * 2);</pre>	
<pre>g2.setPaint(Color.BLACK); g2.fill(r1); g2.setPaint(paint); g2.fill(r1);</pre>	

För att ladda en bild från hårddisken kan vi använda klassen ImageIO (paketet javax.imageio). Jag går inte in på detaljer hur den klassen används utan behöver du ladda en bild från hårddisken använd helt enkelt koden från exemplet (klassen File nämns i lektion 8).

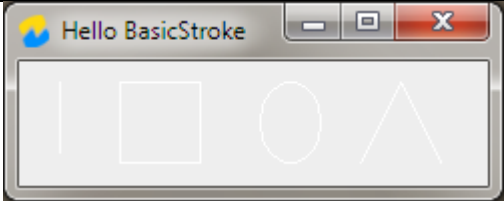
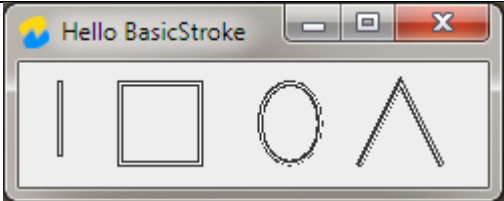
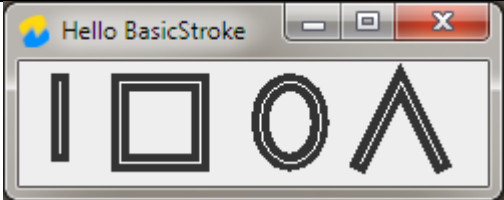
Objektet anchor (av typen Rectangle2D) använder vi för att definierar området bilden ska fyllas i. Vi sätter startpunkten, övre vänstra hörnet, till (10,10) och använder samma bredd och höjd som

bilden (som då inte skalas om). Anledningen till att startpunkten sätts till (10,10) är att bilden ska börja i hörnet på den rektangel som ska fyllas, som också har (10,10) som punkt för övre vänstra hörnet. I det andra exemplet definierar vi ett område som är dubbelt så hög som bilden. Som du ser skalas bilden då om automatiskt för att passa området. Bilder som innehåller transparenta områden behåller dessa när vi använder dem i en TexturePaint. I exempel 3 fyller vi först rektangeln med en svart färg och för att därefter fylla den med vår TexturePaint. Som du ser ”lyser” den svarta färgen igenom de områden i bilden som är transparenta.

Linjestil

För varje Shape (figur) är det möjligt att ange hur figurens yttre kontur ska ritas. Vi kan bestämma dess tjocklek och stil. Vi gör detta genom att skapa ett objekt av klassen BasicStroke (i paketet java.awt). Det enklaste sättet att skapa en BasicStroke är att i konstruktorn ange linjens tjocklek (som en float).

I exemplen nedan ritar vi en linje, rektangel, oval och en sammansatt linje med olika tjocklek på ”pennan” som ritar. Vi ritar först figuren med svart färg och olika tjocklek och därefter samma figurer igen, men med vit färg och tjocklek 1. Detta för att det mer tydligt ska framgå hur olika BasicStroke påverkar originalfiguren så att säga.

<pre>Line2D l = new Line2D.Double(20, 10, 20, 45); Rectangle2D r = new Rectangle2D.Double(50, 10, 40, 40); Ellipse2D e = new Ellipse2D.Double(120, 10, 30, 40); Path2D p = new Path2D.Double(); p.moveTo(170, 50); p.lineTo(190, 10); p.lineTo(210, 50); g2.setStroke(new BasicStroke(1f)); g2.draw(l); g2.draw(r); g2.draw(e); g2.draw(p); g2.setPaint(Color.WHITE); g2.setStroke(new BasicStroke(1f)); g2.draw(l); g2.draw(r); g2.draw(e); g2.draw(p); g2.setStroke(new BasicStroke(3f));</pre>	
	
<pre>g2.setStroke(new BasicStroke(9f));</pre>	

Med BasicStroke är det även möjligt att ange hur en linjes ändpunkter ska dekoreras. När vi skapar en BasicStroke kan någon av följande konstanter användas: CAP_BUTT (ingen dekoration), CAP_ROUND (en halvcirkel med en radie som är lika med halva tjockleken av pennan) och CAP_SQUARE (en halvkvadrat som är lika med halva tjockleken av pennan).

Vi kan även ange en dekoration för de punkter där två linjer i en figur möts. Här kan vi välja någon av följande konstanter: JOIN_MITER (en spetsig dekoration), JOIN_ROUND (en rund dekoration) och JOIN_BEVEL (en rak dekoration).

När vi skapar en BasicStroke där enbart tjocklek angetts används som default CAP_SQUARE som dekoration på ändpunkter och JOIN_MITER som dekoration på de punkter där två linjer möts. Detta ser vi exempel på i bilderna ovan. För Line2D ses effekten av att linjens kant sträcker sig både ovanför och nedanför den vita linjen. Att JOIN_MITER används ses ovan i Path2D genom att punkten (190,10) där de båda linjerna möts är spetsig.

<pre>BasicStroke s = new BasicStroke(11f, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER); g2.setStroke(b);</pre>	
<pre>BasicStroke s = new BasicStroke(11f, BasicStroke.CAP_ROUND, BasicStroke.JOIN_MITER); g2.setStroke(b);</pre>	
<pre>BasicStroke s = new BasicStroke(11f, BasicStroke.CAP_SQUARE, BasicStroke.JOIN_MITER); g2.setStroke(b);</pre>	
<pre>BasicStroke s = new BasicStroke(11f, BasicStroke.CAP_SQUARE, BasicStroke.JOIN_ROUND); g2.setStroke(b);</pre>	
<pre>BasicStroke s = new BasicStroke(11f, BasicStroke.CAP_SQUARE, BasicStroke.JOIN_BEVEL); g2.setStroke(b);</pre>	

För att skapa en streckad linje måste vi ange, förutom linjens tjocklek och dekorationer, även vilket streckmönster som ska användas, en gräns för när JOIN_BEVEL ska användas i stället för JOIN_MITER samt var den streckade linjen ska börja i förhållande till figurens startpunkt.

Vilket streckmönster som ska användas anges som en array av typen float. I arrayen anger vi hur långt ett streck (i den streckade linjen) ska vara och hur långt mellanrummet till näste streck ska vara. Vi kan ange fler än två värden i arrayen och på så sätt skapa väldigt avancerade streckade linjer.

```
float[] dash1 = {10f, 5f};
```

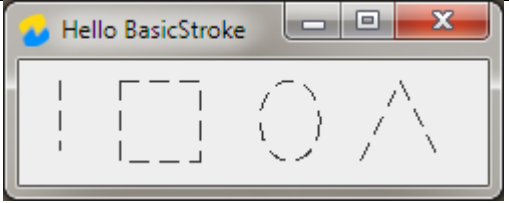
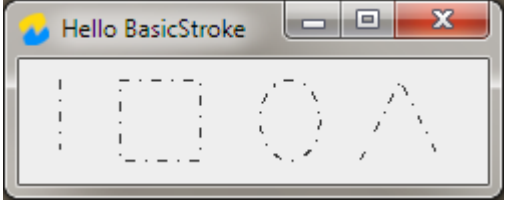
```
float[] dash2 = {5f, 5f, 1f, 5f};
```

Den första arrayen specificera ett streckmönster där varje linje (i den streckade linjen) är 10 enheter långt och varje mellanrum är 5 enheter långt. Den andra arrayen specificerar ett streckmönster som börjar med en linje som är 5 enheter långt, följt av ett mellanrum på 5 enheter, följt av en linje på en enhet som följs av ett mellanrum på 5 enheter. Därefter upprepas mönstret igen runt hela figuren.

Anledningen till att vi måste ange en gräns för när JOIN_BEVEL ska börja användas är att när vinkeln mellan två linjer i figuren blir allt för liten så är det inte lämpligt att använda en spetsig dekorationsstil. Sätt ett godtyckligt värde för detta.

Med hjälp av förskjutningen var den streckade linjen ska starta i förhållande till figurens startpunkt kan vi finjustera streckningens start. Vanligtvis sätter vi dock detta värde till 0.

I exemplen nedan används samma figurer som exemplen ovan. Dock ritas inte den vita originalfiguren ut.

<pre>float[] dash = {10f, 5f}; BasicStroke s = new BasicStroke(1f, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER, 10f, dash, 0f); g2.setStroke(s);</pre>	
<pre>float[] dash = {5f, 5f, 1f, 5f}; BasicStroke s = new BasicStroke(1f, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER, 10f, dash, 0f); g2.setStroke(s);</pre>	

Text

Enklare textsträngar skriver vi ut på samma sätt som med Graphics. Dvs vi kan specificera vilken font som ska användas genom att anropa setFont och för att skriva ut textsträngen anropar vi metoden drawString. En skillnad i Graphics2D är att vi kan omvandla en teckensträng till en figur (Shape) och sen hantera teckensträngen som alla andra figurer, som till exempel enbart rita konturen eller fylla strängen med en färgtoning. Omvandlingen involverar tre olika klasser som vi tidigare inte nämnt. Dessa klasser går vi inte in på i detalj vad de används till eller alla möjligheter klasserna erbjuder. I stället ges enbart exempel på hur vi använder dem på enklast sätt för att omvandla en textsträng till en figur.

Först behöver vi ett objekt av klassen FontRenderContext (i paketet java.awt.font). Detta objekt innehåller information som behövs för att på ett korrekt sätt kunna mäta texten (till exempel textsträngens bredd och höjd). Vi får detta objekt genom att anropa getFontRenderContext i klassen Graphics2D.

```
Graphics2D g2 = (Graphics2D)g;
FontRenderContext context = g2.getFontRenderContext();
```

För att utföra själva omvandlingen använder vi klassen TextLayout (också i paketet java.awt.font). Denna klass representerar en teckensträng som ett grafiskt objekt och hur detta objekt ser ut beror på vilken teckensträng som används, vilken font texten ska ritas ut i och vilken FontContextRender

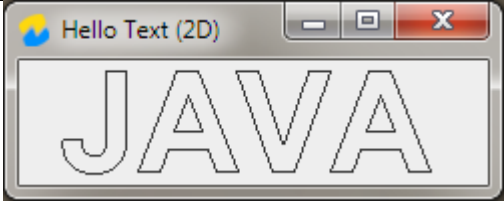
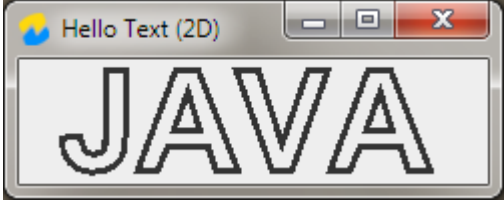
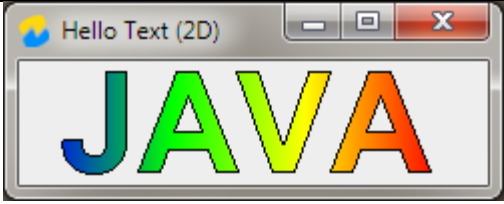
som används. Allt detta måste vi ange när vi skapar ett objekt av klassen `TextLayout`.

```
Font font = new Font(g2.getFont().getName(), Font.BOLD, 70);  
String text = "JAVA";  
TextLayout layout = new TextLayout(text, font, context);
```

Genom att nu anropa metoden `getOutline` i `TextLayout` får vi konturen på textsträngen som en figur (`Shape`). Metoden `getOutline` tar som argument ett objekt av klassen `AffineTransform`. Klassen `AffineTransform` (i paketet `java.awt.geom`) används för att manipulera koordinatsystemet som används (förenklat sett). Vi kan påverka koordinatsystemet genom att flytta utgångspunkten, motsvarande metoden `translate` vi använt tidigare, genom att rotera det och genom att skjuva (shear) det. När vi skapar en figur av en textsträng används `AffineTransform` för att ange vilken startpunkt (nedre vänstra hörnet på texten) den skapade figuren ska ha.

```
AffineTransform transform = AffineTransform.getTranslateInstance(20, 55);  
Shape s = layout.getOutline(transform);
```

Nu kan vi använda figuren och rita ut den som vanligt med metoderna `draw` och `fill`. Vi kan ange konturens tjocklek genom att anropa `setStroke` och vilken färg/fyllning figuren ska ha med metoden `setPaint`.

<pre>g2.draw(s);</pre>	
<pre>g2.setStroke(new BasicStroke(3f)); g2.draw(s);</pre>	
<pre>Color[] colors = {Color.BLUE, Color.GREEN, Color.YELLOW, Color.RED}; float[] dist = {0f, 0.33f, 0.66f, 1f}; LinearGradientPaint paint = new LinearGradientPaint(10, 30, 210, 30, dist, colors); g2.setPaint(paint); g2.fill(s); g2.setPaint(Color.BLACK); g2.draw(s);</pre>	

RenderingHints

Det sista vi tittar på i denna lektion är klassen `RenderingHints` (i paketet `java.awt`). Med denna klass kan vi påverka vilken kvalitet som en figur, text eller bild ska ritas i. Vi kan öka rithastigheten genom att sänka kvaliteten och tvärtom. Kvaliteten påverkas genom att ge vissa egenskaper nya värden. Dessa egenskaper och värden definieras som konstanter i `RenderingHints`.

För att ändra en eller flera egenskaper skapar vi ett objekt av `RenderingHints`, lägger till nyckel-värde-par (key-value pair) som specificerar egenskaper och dess värden, och anropar sen

setRenderingHints i Graphics2D. Vilka egenskaper som finns och vilka värden dessa kan ha kan du läsa om i Javas API för klassen RenderingHints.

Om vi endast är intresserad av att ändra en egenskap kan vi anropa metoden setRenderingHint i klassen Graphics 2D. Denna metod tar som argument den egenskap som ska ändras och dess värde.

Den enda egenskap vi tittar närmare på här är för att ange om antialiasing ska användas när figurer ritas. Antialiasing innebär i korta drag att kvaliteten förbättras genom att visuella tricks används för att få linjer att se rundare och mjukare ut än de egentligen är. Vi använder egenskapen RenderingHints.KEY_ANTIALIASING och ger den värdet RenderingHints.VALUE_ANTIALIAS_ON (kan även vara OFF eller DEFAULT).

Som du ser i exemplet nedan upplevs konturen på texten som betydligt mjukare och mindre kantig när antialiasing används.

```
Graphics2D g2 = (Graphics2D)g;  
g2.setRenderingHint(  
    RenderingHints.KEY_ANTIALIASING,  
    RenderingHints.VALUE_ANTIALIAS_ON);  
// samma kod som föregående exempel
```



Som du sett finns det väldigt mycket vi kan göra i Java när det gäller att rita egna figurer. Vissa delar i denna lektion har vi gått igenom väldigt grundligt, vissa delar mindre grundligt och en del delar inte alls. Förhoppningsvis har du i alla fall fått tillräckligt mycket på fötterna för att klara av det mesta du stöter på när det gäller att rita egna figurer.