Grafische Programmierung mit Java

Einführung

Bis jetzt bestanden unsere Anwendungen nur aus hässlichen DOS-Zeilen. Dieser Look ist jedoch schon seit einigen Jahren (man kann fast schon von einem Jahrzehnt sprechen) nicht mehr zeitgemäß. Da kann unsere Anwendung auch noch so innovativ sein, die meisten Leute werden sie wegen der veralteten Oberfläche nicht nutzen wollen.
Java wäre aber keine gute Programmiersprache wenn es für dieses Problem nicht auch eine Lösung gäbe. Das grundsätzliche Problem der grafischen Programmierung ist das Betriebssystem. Keiner wird mir widersprechen, wenn ich behaupte, dass alle Betriebssysteme unterschiedlich aussehen. Man könnte sich zwar auf eines beschränken, dann jedoch wäre der Gedanke von Java, nämlich gute Javaprogramme für alle Betriebssysteme, zerstört.

AWT

Java hat dies mit dem AWT, dem Abstract Window Toolkit gelöst; nur die Elemente die wirklich jede grafische Oberfläche implementiert (d.h. eingebaut) hat, werden eingesetzt. Die Virtual Machine (Laufzeitumgebung im Betriebssystem) von Java greift direkt auf Betriebssystemfunktionen zur Erzeugung von z.B. Buttons zu.
Nach ersten AWT-Versionen, die sich ganz strikt an diesen Grundsatz gehalten haben, wurde den Architekten von Java klar, dass das Leistungsangebot nicht für die Zukunft ausreicht. Darum ist man mit den Java Foundation Classes (JFC) einen anderen Weg gegangen. Man hat einfach die zusätzlichen Elemente komplett in Java eingebaut.

Fenster mit AWT

Wir wollen mal ganz einfach mit dem Fenster beginnen. Das Fenster (in Java der Frame) dient als Grundlage aller Dialoge mit dem Benutzer. Aus diesem Grund soll es auch die Grundlage unserer Arbeit mit dem AWT sein.
Wie schon erwähnt handelt es sich bei dem AWT um ein Packet von Java, aus diesem Grund muss es auch wie gewohnt eingebunden werden. Danach kann man ganz bequem die Klassen wie Frame (für Fenster) aus dem Packet nutzen. So sieht der Quelltext für ein kleines Fenster aus:

```
import java.awt.*;
public class Fenster1 extends Frame
{
  public Fenster1()
  {
    super("Unser erstes Fenster...");
    this.setSize(300,300);
    this.show();
  }
  public static void main(String args[])
  {
    Fenster1 f = new Fenster1();
  }
}
```

Wie an Quelltext zu sehen ist, werden alle Attribute und Methoden der Klasse Frame an unsere selbsterstellte Klasse Fenster1 vererbt. Neben dem Standardkonstruktor von Frame existiert noch ein weiterer, mit dem wir den Namen in der Titelleiste bestimmen können. Möchte man den Titel jedoch auch noch während der Laufzeit noch einmal ändern, so bietet sich die Methode *setTitle(String str)* an. Mit *.setSize()* wird die Größe des Fensters angegeben. Und erst mit der Anweisung *.show()* wird das Fenster sichtbar.

Wenn wir nun unser erstes Fenster einmal ausprobieren, so merken wir, dass das Fenster ohne unser Zutun schon recht viel kann. So kann man das Fenster vergrößern, verkleinern oder verschieben. Leider lässt sich das Programm aber noch nicht beenden. Wenn man auf das Kreuz (in Windows) drückt passiert nichts. Das ist verständlich, schließlich haben wir noch keine Ereignisbehandlungsroutine dafür geschrieben. Ohne diese Routine ist das Beenden nur durch drücken von [Str]+[C] im Konsolenfenster möglich.

Die paint()-Methode

Unser Fenster kann zwar so tolle Sachen wie größer oder kleiner werden, jedoch ist es sehr leer. Was soll man mit so einem Fenster denn anfangen?
Naja, es gibt aber Methoden die uns helfen das Fenster schöner und sinnvoller zu machen. Als nächster Schritt soll nämlich jetzt Text in unserem Fenster stehen. Dafür setzen wir die Methode *paint()* ein und übergeben dieser ein Objekt der Klasse *Graphics*. Die Klasse *Graphics* stellt eine große Anzahl von Methoden zum Erzeugen von Grafikobjekten (Text, Linien, Rechtecke) zur Verfügung, unter anderem auch die Methode *drawString*. Diese Methode ist für das Zeichnen von Text auf das Fenster zuständig.
So sieht der Quelltext für ein Fenster mit Text aus:

```
import java.awt.*;
public class Fenster2 extends Frame
{
  public Fenster2()
  {
     super("Unser zweites Fenster...");
     this.setSize(300,150);
     this.show();
  }
  public void paint (Graphics g)
  {
     g.drawString("Hallo Welt...",100,60);
     }
```

```
    public static void main(String args[])
    {
       Fenster2 f = new java4u();
    }
}
```

Man kann nur in der *paint()*-Methode auf das *Graphics*-Objekt zugreifen. Die *paint()*-Methode wiederum ist eigentlich zum Neuzeichnen des Fensters gedacht. Aber dadurch, dass das *Graphics*-Objekt an die *paint()*-Methode übergeben wird, wird auch bei jedem Neuzeichnen der Text mitaktualisiert. Das Neuzeichnen ist übrigens immer dann notwendig, wenn das Fenster verschoben, vergrößert oder verkleinert wurde. Da *drawString* mit Koordinaten arbeitet, wird durch das Neuzeichnen mittels *paint()* immer an der richtigen Stelle geschrieben.

Das *Graphics*-Objekt bietet einige Zusatzfunktionen, so speichert es z.B. die Komponente auf der zu zeichnen ist, die Koordinaten des Zeichenbereichs und die aktuelle Schriftart und Farbe.

Linien

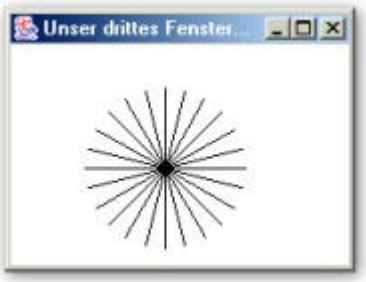
So nun wollen wir das *Graphics*-Objekt etwas anspruchsvoller nutzen. Wir haben ja schon erfahren, dass das *Graphics*-Objekt eine Vielzahl an Zeichenoperationen bietet. Neben dem Zeichnen von Text kann man auch Linien zeichnen lassen. Dies wollen wir nun tun. Wir wollen einen sogenannten Siemens-Stern zeichnen, d.h. es wird ein Stern der aus einigen Linien die im 360° Winkel angeordnet sind gezeichnet.
Das Fenster wird wie du sicher schon bemerkt hast als Koordinatensyste, behandelt. Der Ursprung (0,0) befindet sich in der oberen linken Ecke. Ein ideales Spielfeld für die Mathematik also!
Zum Zeichnen einer Linie wird die Methode *drawLine* verwendet, die folgendermaßen aufgebaut ist:
drawLine(int x1, int y1, int x2, int y2);

Ein Fenster in dem ein solcher Siemensstern gezeichnet wir sieht im Quelltext so aus:

```
import java.awt.*;
public class Fenster3 extends Frame
{
  public Fenster3()
  {
     super("Unser drittes Fenster...");
     this.setSize(500,200);
     this.show();
  }
```

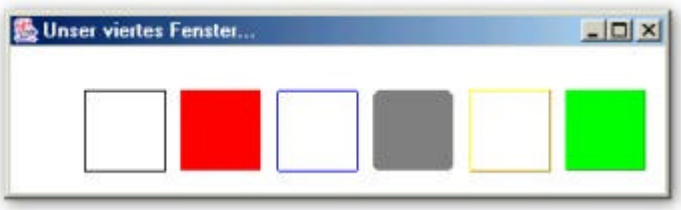
```
    public void paint (Graphics g)
    {
       int laenge=50;
       int x2=0;
       int y2=0;
       for (int i=0; i<360; i+=15)
       {
          y2= 100-(int)(Math.sin (Math.toRadians(i))*laenge);
          x2= 250-(int)(Math.cos (Math.toRadians(i))*laenge);
          g.drawLine(250,100, x2, y2);
       }
    }
    public static void main(String args[])
    {
       Fenster3 f = new Fenster3();
    }
}
```

Das Zeichnen der Linien ist nicht schwer, denn der Startpunkt ist immer gleich (250,100), nur die Endpunkte müssen berechnet werden.



Rechtecke

Bis jetzt waren alle unsere Objekte von Graphics einfarbig. Nun wollen wir den Inhalten unserer Fenster mal etwas Farbe einhauchen. Um die Farbe zu verändern, wird das Attribut color der Klasse Graphics mit der Methode setColor verändert.
Diesmal möchte ich auch nicht nur ein Rechteck ausprobieren, ich probiere mal alle möglichen Rechtecke aus, die Definition davon kann man wie von jedem Objekt aus der API-Dokumentation der JDK entnehmen. So soll unser Fenster mal aussehen:



Dafür brauchen wir folgenden Quelltext:

```
import java.awt.*;
```

```
public class Fenster4 extends Frame
{
  public Fenster4()
  {
    super("Unser viertes Fenster...");
    this.setSize(500,200);
    this.show();
  }
  public void paint(Graphics g)
  {
    g.setColor(Color.BLACK);
    g.drawRect(50,50,50,50);
    g.setColor(Color.RED);
    g.fillRect(110,50,50,50);
    g.setColor(Color.BLUE);
    g.drawRoundRect(170,50,50,50,5,5);
    g.setColor(Color.GRAY);
    g.fillRoundRect(230,50,50,50,10,10);
    g.setColor(Color.ORANGE);
    g.draw3DRect(290,50,50,50, true);
    g.setColor(Color.GREEN);
    g.fill3DRect(350,50,50,50,true);
  }
  public static void main(String args[])
  {
    Fenster4 f = new Fenster4();
  }
}
```

Ereignisverarbeitung

Bis jetzt hatten unsere Programme den Nachteil, dass sie sich nicht so wie man es gewohnt ist beenden lassen. Man musste immer den Umweg über die Eingabekonsole gehen. Diesen Nachteil wollen wir nun ausmerzen!

Wie ich schon damals erwähnt hatte, ist das Beenden von Programmen ein Ereignis. Und zwar ein Frame-Ereignis. Java arbeitet mit Listenern, die auf ein bestimmtes Ereignis warten, diese Listener müssen erst registriert werden. Grundsätzlich werden die Listener über eine Methode im Stil add*Window*Listener registriert (Window kann auch durch andere Ausdrücke ersetzt werden).

Die Methode ist folgendermaßen aufgebaut: *addWindowListener(WindowListener)*

Die Methode verlangt nach einem Listener, der in der Klasse WindowListener steht. Wenn man in der API dort nachsieht erkennt man, dass es sich nicht um eine Klasse, sondern um ein Interface handelt.

Ein Interface ist eine Klasse die nur den Methodenprototypen enthält, d.h. die Methoden haben nur einen Kopf, aber keine Anweisung (Quellcoderumpf).

Das Interface hat sieben Methoden, von der aber nur eine für uns interessant ist: *windowClosing(WindowEvent e)*.

Immer wenn ein Interface mehr als eine Methode hat, gibt es so genannte Adapterklassen, die das Interface beeinhalten (also alle Methoden mit einem leeren Quellcoderumpf). Wir greifen also über einen Adapter auf die *WindowListener-*

Funktionalität von Java zu und haben durch die Interface-Architektur die Möglichkeit den Listener an unsere Applikation anzupassen. Wir haben also freie Gestaltungsfreiheit ohne einengende Vorgaben durch Java.
Das hört sich jetzt kompliziert an, wenn man es aber erst im Quelltext siehst, wird man merken, dass alles ganz logisch ist. Nun also der Quelltext:

```
import java.awt.*;
public class Fenster5 extends Frame
{
    private MyWindowListener myWindowListener = new MyWindowListener();
    public Fenster5()
    {
        super("Unser fünftes Fenster...");
        this.setSize(300,150);
        this.show();
        this.addWindowListener(myWindowListener);
    }
    public void paint(Graphics g)
    {
        g.drawString("Hallo Welt, nun schaut doch mal!",100,60);
        g.drawString("Mich kann man beenden!", 130,60);
    }
    public static void main(String args[])
    {
        Fenster5 f = new Fenster5();
    }
}
```

MyWindowListener.java

```
import java.awt.event.*;
public class MyWindowListener extends WindowAdapter
{
  public void windowClosing(WindowEvent e)
  {
     System.exit(0);
  }
}
```

Im Konstruktor (Fenster6()) wird die Ereigniskette aufgebaut, in dem der Methode *addWindowListener* das Objekt *myWindowListener* übergeben wird. Das eigentliche Beenden findet wie schon angesprochen in der selbst mit Quelltext versehenen Methode *windowClosing* in der Klasse *MyWindowListener* statt. Das Beenden selbst lässt sich mit folgender Anweisung bewerkstelligen: *System.exit(0)*; aber man könnte noch mehr "TamTam" machen und noch Meldungen ausgeben, das Interface-System lässt uns da freie Hand.

Buttons

Bis jetzt konnte der Benutzer recht wenig machen, er konnte sich Schrift, Linien und Rechtecke ansehen. Am Ende konnte er das Fenster auch noch schließen, er konnte jedoch keine Aktionen durchführen. Damit der Benutzer auch mal was machen kann,

bekommt unser Programm nun einen Button.
Natürlich kommt der Button nicht von selbst, man muss ihn erst einmal auf dem Frame hinzufügen. Dafür gibt es die *add*-Methode des Frames. Jetzt hätten wir aber wieder das gleiche Problem wie beim Schließen-Button beim Frame. Es passiert nichts! Also machen wir es doch so wie beim Frame: Wird setzen einen Listener ein! Diesmal jedoch nicht den *WindowListener*, sondern den *ActionListener*. Dieser ist praktisch genauso aufgebaut wie der *WindowListener*: *addActionListener(ActionListener);*
Wie schon beim *WindowListener* braucht die Methode ein Objekt des Interfaces *ActionListener*. Dieses Interface bietet als einzige Methode *actionPerformed* an.
Wir schauen uns also wieder den Quelltext zum besseren Verständnis an:

```
import java.awt.*;
public class Fenster6 extends Frame
{
    private Button btnBeenden = new Button("Beenden");
    private MyWindowListener myWindowListener = new MyWindowListener();
    private MyActionListener myActionListener = new MyActionListener();
    public Fenster6()
    {
        super("Unser sechstes Fenster...");
        this.setSize(400,100);
        this.add(this.btnBeenden);
        this.show();
        this.addWindowListener(myWindowListener);
        this.btnBeenden.addActionListener(myActionListener);
    }
    public static void main(String args[])
    {
        Fenster6 f = new Fenster6();
    }
}
```

MyWindowListener.java

```
import java.awt.event.*;
public class MyWindowListener extends WindowAdapter
{
  public void windowClosing(WindowEvent e)
  {
    System.exit(0);
  }
}
```

MyActionListener.java

```
import java.awt.event.*;
public class MyActionListener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    System.exit(0);
  }
```

```
}
```

In der Klasse *MyActionListener* sollte einem ein ungewöhnliches Schlüsselwort auffallen: ***implements***. *implements* ist das Gegenstück der *Interfaces* zum *extends* der *Klassen* bei der Vererbung.
Wenn du das Programm ausführst, entdeckt man jedoch schon das nächste Problem:

Der Button füllt das ganze Fenster aus! Diesem Problem wollen wir uns nun widmen!

Layout-Manager
Wir haben gerade selbst gesehen was passiert wenn man keine absolute Positionierung (mit Pixelangaben) macht. Der Button füllt den ganzen Frame aus. Aber mal zugegeben, willst du bei jedem Button immer herum probieren wie groß er sein soll und die Pixelangaben dann ändern und jedes Mal auch noch langwierig kompilieren? Das ist schon ganz schön umständlich mit den Pixeln! Aber so ein ganzes Frame für nur einen Button ist reinste Verschwendung.
Aber wie immer gibt es in Java auch dafür eine komfortable Lösung: die Layout-Manager. Die Layout-Manager wenden Regeln zur Positionierung und Dimensionierung der Buttons an. Im Package awt gibt es zwar 5 Layout-Manager, ich möchte davon jedoch nur die drei wichtigsten behandeln. Und zwar *FlowLayout*, *BorderLayout* und *GridLayout*.

FlowLayout
Dieser Layout-Manager ordnet die Komponenten in Zeilen von links nach rechts an. Wenn eine Zeile voll ist, wird die nächste Komponente in der nächsten Zeile darunter angeordnet. *FlowLayout* setzt die Größe der Komponenten auf ein Minimum. Der FlowLayout-Manager hat folgenden Konstruktor: *FlowLayout(int ausrichtung, int horizAbstand, int vertAbstand)*
Wobei eigentlich sämtliche Parameter optional sind. Wenn nämlich nichts angegeben ist, werden die Komponenten zentriert angeordnet und die Abstände sowohl horizontal als auch vertikal auf 5 Pixel gesetzt. Abstände werden verständlicherweise in Pixel angegeben, bei der Ausrichtung gibt es folgende Wahlmöglichkeiten bestehend aus statischen Konstanten: *FlowLayout.LEFT*, *FlowLayout.RIGHT* und *FlowLayout.CENTER*
Ich habe mal folgenden Quelltext zur Verdeutlichung anzubieten:

```
import java.awt.*;
public class Fenster7 extends Frame
{
    private FlowLayout myLayout = new FlowLayout(FlowLayout.LEFT);
    private MyWindowListener myWindowListener = new MyWindowListener();
    private MyActionListener myActionListener = new MyActionListener();
```

```
    private Button btn1 = new Button("1");
    private Button btn2 = new Button("2");
    private Button btn3 = new Button("3");
    private Button btn4 = new Button("4");
    public Fenster7()
    {
       super("Unser siebentes Fenster...");
       this.setLayout(myLayout);
       this.setSize(200,100);
       this.add(this.btn1);
       this.add(this.btn2);
       this.add(this.btn3);
       this.add(this.btn4);
       this.show();
       this.addWindowListener(this.myWindowListener);
       this.btn1.addActionListener(this.myActionListener);
       this.btn2.addActionListener(this.myActionListener);
       this.btn3.addActionListener(this.myActionListener);
       this.btn4.addActionListener(this.myActionListener);
    }
    public static void main(String args[])
    {
       Fenster7 f = new Fenster7();
    }
}
```

MyWindowListener.java

```
import java.awt.event.*;
public class MyWindowListener extends WindowAdapter
{
  public void windowClosing(WindowEvent e)
  {
     System.exit(0);
  }
}
```

MyActionListener.java

```
import java.awt.event.*;
public class MyActionListener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    int wert = integer.parseInt(e.getActionCommand());
    switch(wert)
    {
      case 1: System.out.println("Button 1"); break;
      case 2: System.out.println("Button 2"); break;
      case 3: System.out.println("Button 3"); break;
      case 4: System.out.println("Button 4"); break;
    }
```

```
    }
}
```

In der *main*-Methode wird nur das Fenster erzeugt, das Aussehen wird im Konstruktor festgelegt. Der Aufruf des Layout-Managers funktioniert mit *setLayout*. Erst danach werden die einzelnen Komponenten dem Frame-Container hinzugefügt.
Da alle Buttons mit dem gleichen *ActionListener*-Objekt verbunden sind, muss man in der *MyActionListener*-Klasse eine Unterscheidung mittels einer *switch*-Auswahlabfrage durchführen. Die dafür nötigen Daten liefert das *ActionEvent*-Objekt aus der *actionPerformed*-Methode. Erst danach steht fest welcher Button der Auslöser war.
Jede grafische Komponente hat ein Attribut mit dem Namen *actionCommand*, wenn keine Änderung daran vorgenommen wurde stimmt dieses Attribut mit der Beschriftung überein. So können wir mit der Anweisung *getActionCommand*, dieses Attribut aus dem Auslöser auslesen. Nach einer Umwandlung des Strings in einen Integerwert kann man ihn als Selektor einsetzen.
Der Frame sieht so aus:

BorderLayout
Das *BorderLayout* orientiert sich an den Rändern, die durch die Himmelsrichtungen bezeichnet werden. Aber natürlich gibt es auch eine Mitte. Mit dem *BorderLayout* lassen sich maximal fünf Komponenten platzieren.
Das *BorderLayout* passt die Komponentengröße der Framegröße an. Die *BorderLayout*-Anweisung ist folgendermaßen aufgebaut: *BorderLayout(int horizAbstand, int vertAbstand)*
Die Abstände sind wieder optional, wird also nichts angegeben, wählt der Compiler automatisch den Abstand 0 Pixel. Es wird also kein Abstand gelassen, wer dies verhindern will, der muss die Abstände angeben. Da die Position der Komponente angegeben werden muss, ist es nötig die add-Methode anzupassen. Sie muss nun so aussehen:
add(Component komponentennamen, int pos)

In der Praxis müsste die Anweisung also so aussehen: *this.add(myButton, BorderLayout.CENTER)*. Hier sieht man schon eine der Positionsangaben im Einsatz; die anderen sehen so aus: *BorderLayout.EAST*, *WEST*, *NORTH*, *SOUTH*
Ich möchte diesmal nur die Quellcodedatei angeben, die sich gegenüber dem *FlowLayout* auch geändert hat. Wer das Beispiel nachvollziehen will, muss einfach noch die Dateien *MyWindowListener.java* und *MyActionListener.java* aus dem vorherigen Beispiel abschreiben (oder in den Workspace einbinden). Hier aber nun der Quelltext:

```
import java.awt.*;
```

```
public class Fenster8 extends Frame
{
    private BorderLayout borderLayout = new BorderLayout();
    private MyWindowListener myWindowListener = new MyWindowListener();
    private MyActionListener myActionListener = new MyActionListener();
    private Button btnNorth = new Button("1");
    private Button btnSouth = new Button("2");
    private Button btnEast = new Button("3");
    private Button btnWest = new Button("4");
    public Fenster8()
    {
        super("Unser achtes Fenster...");
        this.setLayout(borderLayout);
        this.setSize(300,300);
        this.add(btnNorth, BorderLayout.NORTH);
        this.add(btnSouth, BorderLayout.SOUTH);
        this.add(btnEast, BorderLayout.EAST);
        this.add(btnWest, BorderLayout.WEST);
        this.show();
        this.addWindowListener(this.myWindowListener);
        this.btnNorth.addActionListener(this.myActionListener);
        this.btnSouth.addActionListener(this.myActionListener);
        this.btnEast.addActionListener(this.myActionListener);
        this.btnWest.addActionListener(this.myActionListener);
    }
    public static void main(String args[])
    {
        Fenster8 f = new Fenster8();
    }
}
```

So sieht das *BorderLayout* aus:

GridLayout

Das *GridLayout* ordnet die Komponenten in Spalte von links nach rechts und die Zeilen von oben nach unten. Der Konstruktor ist folgendermaßen aufgebaut: *GridLay-*

out(int zeilen, int spalten, int hAbstand, int vAbstand)
Die Abstände sind wieder optional. Wenn man ENTWEDER für *zeilen* ODER für *spalten* als Wert '*0*' eingibt, so werden die Spalten oder Zeilen beliebig gefüllt. Es ist jedoch wichtig, dass immer nur einer der Parameter '*0*' als Wert hat.
Der Hauptquelltext sieht so aus:

```
import java.awt.*;
public class Fenster9 extends Frame
{
    private GridLayout myGridLayout = new GridLayout(2,0);
    private MyWindowListener myWindowListener = new MyWindowListener();
    private MyActionListener myActionListener = new MyActionListener();
    private Button btn1 = new Button("1");
    private Button btn2 = new Button("2");
    private Button btn3 = new Button("3");
    private Button btn4 = new Button("4");
    public Fenster9()
    {
        super("Unser neuntes Fenster...");
        this.setLayout(borderLayout);
        this.setSize(300,300);
        this.add(this.btn1);
        this.add(this.btn2);
        this.add(this.btn3);
        this.add(this.btn4);
        this.show();
        this.addWindowListener(this.myWindowListener);
        this.btn1.addActionListener(this.myActionListener);
        this.btn2.addActionListener(this.myActionListener);
        this.btn3.addActionListener(this.myActionListener);
        this.btn4.addActionListener(this.myActionListener);
    }
    public static void main(String args[])
    {
        Fenster9 f = new Fenster9();
    }
}
```

Unser *GridLayout*-Fenster sieht dann so aus:

Panels

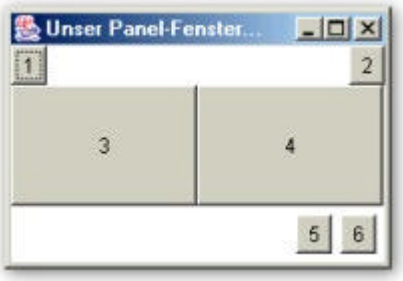
Was ist die logische Konsequenz aus allen drei vorherigen Layout-Managern: Am besten wäre es, wenn man sie kombinieren könnte! Ja, und genau das wollen wir nun tun. Mit *Panels* lassen sich die vorherigen Layout-Manager kombinieren um ein individuelles Layout zu entwerfen.
Ein *Panel* dient als Container-Objekt und ist selbst eine Komponente. So lassen sich mehrere *Panels* auf dem Frame wie Komponenten anordnen, bei dem jedoch jedes *Panel* seinen eigenen Layoutmanager hat. Der *Panel*-Konstruktor sieht so aus: *Panel(LayoutManager layout)*

Aber erst wenn man den Haupt-Quelltext gesehen hat, sieht man wie *Panels* wirklich arbeiten:

```
import java.awt.*;
public class Panel1 extends Frame
{
    private MyWindowListener myWindowListener = new MyWindowListener();
    private MyActionListener myActionListener = new MyActionListener();
    private Button btn1 = new Button("1");
    private Button btn2 = new Button("2");
    private Button btn3 = new Button("3");
    private Button btn4 = new Button("4");
    private Button btn5 = new Button("5");
    private Button btn6 = new Button("6");
    private Panel pnlNorth = new Panel(new BorderLayout());
    private Panel pnlCenter = new Panel(new GridLayout(1,2));
    private Panel pnlSouth = new Panel(new FlowLayout(FlowLayout.RIGHT));
    public Panel1()
    {
        super("Unser Panel-Fenster...");
        this.setSize(300,300);
        this.pnlNorth.add(btn1, BorderLayout.WEST);
        this.pnlNorth.add(btn2, BorderLayout.EAST);
        this.pnlCenter.add(btn3);
        this.pnlCenter.add(btn4);
        this.pnlSouth.add(btn5);
        this.pnlSouth.add(btn6);
        this.add(pnlNorth, BorderLayout.NORTH);
        this.add(pnlCenter, BorderLayout.CENTER);
        this.add(pnlSouth, BorderLayout.SOUTH);
        this.show();
        this.addWindowListener(this.myWindowListener);
        this.btn1.addActionListener(this.myActionListener);
        this.btn2.addActionListener(this.myActionListener);
        this.btn3.addActionListener(this.myActionListener);
        this.btn4.addActionListener(this.myActionListener);
    }
    public static void main(String args[])
    {
```

```
        Panel1 p = new Panel1();
    }
}
```

Ein Bild des Frames zeigt wie unsere *Panels* aussehen:



Wir erzeugen drei *Panels* die wir mittels *BorderLayout* übereinander anordnen. Im ersten Panel erzeugen wir ein *BorderLayout*, im zweiten ein *GridLayout* und im letzten ein *FlowLayout*. Schon ist unser *Panel*-Fenster fertig. Sieht doch garnicht so schlecht aus!

TextField/Label

Bis jetzt haben wir nur mit Buttons gearbeitet. Nun wollen wir zwei neue Komponenten einführen, das *TextField* und das *Label*. Ein *TextField* ist ein einzeiliges Textfeld zur Eingabe von Text. Das *Label* zeigt nur Text an. Die beiden wichtigsten Methoden sind *setText(String str)* und *getText()*, mit dem ersten wird die Beschriftung geändert, mit dem zweiten die Eingabe abgefragt.
Anhand eines Taschenrechners möchte ich nun die Verwendung der neuen Komponenten und der Layout-Manager zeigen:

```
import java.awt.*;
public class Rechner extends Frame
{
    private MyWindowListener myWindowListener = new MyWindowListener();
    private MyActionListener myActionListener = new MyActionListener(this);
    private Label lblUeberschrift = new Label("Taschenrechner");
    private Label lblZahl1 = new Label("1. Zahl");
    private Label lblZahl2 = new Label("2. Zahl");
    public Label lblErg = new Label("");
    public TextField tfZahl1 = new TextField(10);
    public TextField tfZahl2 = new TextField(10);
    public Button btnBeenden = new Button("Beenden");
    public Button btnPlus = new Button("+");
    public Button btnMinus = new Button("-");
    public Button btnMal = new Button("X");
    public Button btnDiv = new Button("/");
    private Panel pnlNorth = new Panel();
    private Panel pnlCenter = new Panel(new GridLayout(3,1));
    private Panel pnlSouth = new Panel(new GridLayout(1,0));
    private Panel pnlCenterZeile1 = new Panel(new FlowLay-
    out(FlowLayout.LEFT));
```

```
    private Panel pnlCenterZeile2 = new Panel(new FlowLay-
    out(FlowLayout.LEFT));

    public Rechner()
    {
        super("Taschenrechner.");
        this.lblUeberschrift.setFont(new Font("",Font.BOLD,16));
        this.pnlNorth.add(lblUeberschrift);
        this.pnlCenterZeile1.add(this.lblZahl1);
        this.pnlCenterZeile1.add(this.tfZahl1);
        this.pnlCenterZeile2.add(this.lblZahl2);
        this.pnlCenterZeile2.add(this.tfZahl2);
        this.pnlCenter.add(this.pnlCenterZeile1);
        this.pnlCenter.add(this.pnlCenterZeile2);
        this.pnlCenter.add(this.lblErg);
        this.pnlSouth.add(this.btnPlus);
        this.pnlSouth.add(this.btnMinus);
        this.pnlSouth.add(this.btnMal);
        this.pnlSouth.add(this.btnDiv);
        this.pnlSouth.add(this.btnBeenden);
        this.add(pnlNorth, BorderLayout.NORTH);
        this.add(pnlCenter, BorderLayout.CENTER);
        this.add(pnlSouth, BorderLayout.SOUTH);
        this.pack();
        this.show();
        this.addWindowListener(this.myWindowListener);
        this.btnPlus.addActionListener(myActionListener);
        this.btnMinus.addActionListener(this.myActionListener);
        this.btnMal.addActionListener(this.myActionListener);
        this.btnDiv.addActionListener(this.myActionListener);
        this.btnBeenden.addActionListener(this.myActionListener);
    }
    public static void main(String args[])
    {
        Rechner f = new Rechner();
    }
}
```

Wie im Quelltext zu sehen ist, wir das *Label* folgendermaßen erzeugt: *Label(String str)*, das *TextField* wird folgendermaßen erzeugt: *TextField(int length)*. Beim *Label* wird die Beschriftung als Parameter eingetragen und beim *TextField* die erlaubte Höchstmenge an Zeichen die eingegeben werden dürfen.
Wie beim vorherigen Projekt wird auch beim Taschenrechner das Frame in drei Panels unterteilt. Das mitterlere *Panel* wird nochmals in zwei Panels unterteilt - *CenterZeile1* und *CenterZeile2* - die untereinander angeordnet werden. So kann man das *BorderLayout* und das *FlowLayout* kombinieren, um die Beschriftung und das *TextField* perfekt zu positionieren. Für die Überschrift wird mit der *Font*-Anweisung eine besondere Schriftgestaltung gewählt: Fett und 16 pt groß.
Aufmerksamen wird aufgefallen sein, dass wir diesmal garnicht die Größe des Frame

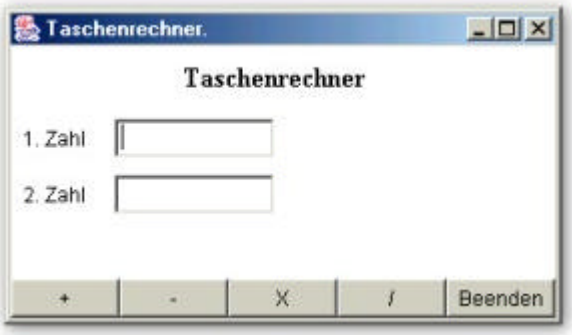
angegeben haben. Richtig, dafür gibt es aber das Schlüsselwort *.pack()* dass die Größe des Frame den Dimensionen der Komponenten anpasst. Der Frame wird also ohne ellenlanges Probieren automatisch perfekt dimensioniert.
Noch aber fehlt der *ActionListener*, hier nun der Quelltext:

MyActionListener.java

```
import java.awt.event.*;
import java.text.*;
public class MyActionListener implements ActionListener
{
  public Rechner f;
  public MyActionListener(Rechner f)
  {
      this.f=f;
  }
  public void actionPerformed(ActionEvent e)
  {
    Object obj = e.getSource();
    if(obj == f.btnBeenden)System.exit(0);
    Decimal Format df = new DecimalFormat("#,##0.0");
    double ergebnis=0;
    double zahl1=0;
    double zahl2=0;
    try
    {
      zahl1=Double.parseDouble(f.tfZahl1.getText().replace(',','.'));
      zahl2=Double.parseDouble(f.tfZahl2.getText().replace(',','.'));
      if(obj==f.btnPlus) ergebnis=zahl1+zahl2;
      else if(obj==f.btnMinus) ergebnis=zahl1-zahl2;
      else if(obj==f.btnMal) ergebnis=zahl1*zahl2;
      else if(obj==f.btnDiv) ergebnis=zahl1/zahl2;
      f.lblErg.setText("Ergebnis: "+df.format(ergebnis));
    }
    catch(NumberFormatException error)
    {
      f.lblErg.setText("Fehler bei Eingabefeld...");
    }
  }
}
```

Damit man auf die Komponenten der Klasse Rechner zugreifen kann, müssen wir ein Objekt der Klasse *Rechner* mit dem Namen f erstellen. Über dieses Objekt laufen alle Aktionen mit dem Frame.
Mit dem Objekt *obj* der Klasse *Object* kann man mittels *e.getSource* den Sender (also die Komponente die eine Action ausgelöst hat) feststellen.
Mit der Klasse *DecimalFormat* kann man Text sehr ansprechend formatieren. In unserem Beispiel sorgt *DecimalFormat* für Tausenderpunkte und zwei Nachkommastellen.
Der Taschenrechner merzt einen häufig gemachten Fehler aus: Da Java eine amerikanische Programmiersprache ist, unterliegt sie den dortigen Standards und in Ame-

rika verwendet man kein Komma sondern einen Punkt um die Nachkommastellen von der Zahl zu trennen. Mit *.replace* wird das Komma einfach durch einen Punkt ersetzt, ansonsten könnte man mit Kommazahlen (mit Komma als Trennzeichen) nicht in Java rechnen.
Und so sieht unser Machwerk aus:



TextArea
Soeben haben wir zwei neue Komponenten eingeführt. Damals hatte ich schon erwähnt, dass ein *TextField* ein einzeiliges Textfeld zur Eingabe von Text ist. Nun was wird nun also kommen? Genau! Ein mehrzeiliges Textfeld zur Eingabe von Text. Dieses mehrzeilige Textfeld nennt sich *TextArea*.
Für die *TextArea* gibt es folgenden Kontruktor: *TextArea(String text, int zeilen, int spalten, int scrollbars)*
Hier gilt, dass man keinen *text* angeben muss, auch kann man die *zeilen* und *spalten* weglassen, dann gibt man der *TextArea* jedoch gar keine Angabe wie sie aussehen soll. Als *scrollbars* kommen *TextArea.SCROLLBARS_BOTH*, *SCROLLBARS_HORIZONTAL_ONLY*, *SCROLLBARS_NONE* und *SCROLLBARS_VERTICAL_ONLY* in Frage.
Es gibt eine Vielzahl an Methoden, die wichtigsten sind jedoch *append* und *insert*. *append* fügt einen Text ans Ende des Textes ein und wird folgendermaßen definiert: *void append(String text)*
insert fügt einen String an einer eingegebenen Position ein und wird so definiert: *void insert(String text, int pos)*
Diesmal möchte ich einen kleinen Texteditor als Beispiel zur Verdeutlichung programmieren. So sieht dessen Quelltext aus:

```
import java.awt.*;
public class TEditor extends Frame
{
    private MyWindowListener myWindowListener = new MyWindowListener();
    private MyActionListener myActionListener = new MyActionListener(this);
    private Label lblOeffnen = new Label("Datei öffnen:");
    private Label lblSpeichern = new Label ("Speichern unter");
    public TextField tfOeffnen = new TextField(20);
    public TextField tfSpeichern = new TextField(20);
    public TextArea taText = new TextArea(15,50);
    public Button btnBeenden = new Button("Beenden");
    public Button btnOeffnen = new Button("Öffnen");
    public Button btnSpeichern = new Button("Speichern");
```

```
    private Panel pnlNorth = new Panel();

    public TEditor()
    {
        super("Text Editor");
        this.pnlNorth.add(this.lblOeffnen);
        this.pnlNorth.add(this.tfOeffnen);
        this.pnlNorth.add(this.btnOeffnen);
        this.pnlNorth.add(this.lblSpeichern);
        this.pnlNorth.add(this.tfSpeichern);
        this.pnlNorth.add(this.btnSpeichern);
        this.add(this.pnlNorth, BorderLayout.NORTH);
        this.add(this.taText, BorderLayout.CENTER);
        this.add(this.btnBeenden, BorderLayout.SOUTH);
        this.pack();
        this.show();
        this.addWindowListener(this.myWindowListener);
        this.btnOeffnen.addActionListener(myActionListener);
        this.btnSpeichern.addActionListener(this.myActionListener);
        this.btnBeenden.addActionListener(this.myActionListener);
    }
    public static void main(String args[])
    {
        TEditor f = new TEditor();
    }
}
```

MyActionListener.java

```
import java.awt.event.*;
import java.io.*;
public class MyActionListener implements ActionListener
{
   public TEditor f;
   public MyActionListener(TEditor f)
   {
         this.f=f;
   }
   public void actionPerformed(ActionEvent e)
   {
      Object obj = e.getSource();
      if(obj == f.btnOeffnen)
      {
         try
         {
            String zeile=null;
            BufferedReader in = new BufferedReader(new InputStreamRea-
            der(new FileInputStream(f.tfOeffnen.getText())));
            while((zeile=in.readLine())!=null)
```

```
                f.taText.append(zeile+"\n");
                in.close();
            }
            catch(FileNotFoundException err)
            {
                System.out.println("Datei nicht gefunden"+err);
            }
            catch(IOException err)
            {
                System.out.println("Lesefehler"+err);
            }
        }
        if(obj==f.btnSpeichern)
        {
            try
            {
                BufferedWriter out = new BufferedWriter(new OutputStream-
                Writer(new FileOutputStream(f.tfSpeichern.getText())));
                out.write(f.taText.getText());
                out.close();
            }
            catch(FileNotFoundException err)
            {
                System.out.println("Fehler beim Speichern"+err);
            }
            catch(IOException err)
            {
                System.out.println("Schreibfehler..."+err);
            }
        }
        if(obj==f.btnBeenden)System.exit(0);
    }
}
```

Die *MyActionListener*-Klasse ist verantwortlich für die Dateioperationen (Öffnen, Speichern) des Programms.
So sieht der Editor dann aus:

Der TextEditor arbeitet wirklich einwandfrei. Man kann ihn auch durch das Laden große Dateien (io.sys(108 KB)) nicht aus der Ruhe bringen, auch wenn es eine Ewigkeit dauert bis die Datei angezeigt wird.

FileDialog

Der Texteditor hat ja schon ganz gut funktioniert. Leider war das Öffnen und Speichern alles andere als komfortabel, da man den Pfad der Datei immer von Hand eingeben musste. Das wollen wir nun mit der Einführung einer neuen Komponente einfacher gestalten. Die Komponente heißt *FileDialog* und erzeugt die bekannten Öffnen- und Speichern-Dialoge die einem fast jedes Programm anzeigt.
Der *FileDialog* hat folgenden Konstruktor: *FileDialog(Frame parent, String title, int mode)*
Wie man sieht muss im Konstruktor ein Verweis auf den Frame gemacht werden, damit dessen Aktionen (größer, kleiner und schließen) von dem *FileDialog* mit gemacht werden können. Dieser Verweis wird mit *Frame parent* gemacht. Den Titel des Fensters könnte man noch mittels *title* ändern. Mit *mode* kann man den Modus Öffnen oder Speichern, den der Dialog abbilden soll, angeben. Öffnen lässt sich mittels *FileDialog.LOAD* einstellen und Speichern mittels *FileDialog.SAVE*
Die wichtigsten Methoden der *FileDialog*-Klasse sind *setDirectory*, *setFile*, *getDirectory*, *getFile*. Mit *setDirectory* wird ein vorgegebener Pfad gesetzt, mit *setFile* wird ein vorgegebener Dateinamen gesetzt, bzw. ein Filter (*.txt) gesetzt, mit *getDirectory* wird der ausgewählte Pfad (Verzeichnis) zurückgegeben und mit *getFile* der ausgewählte Dateiname, falls der Benutzer "Abbrechen" gedrückt hat, ist der Rückgabewert für Dateiname und Verzeichnis null. Die Methoden sind folgendermaßen aufgebaut:
public void setDirectory(String dir)
public void setFile(String file)
public String getDirectory()
public String getFile()

Hier nun nochmal das Beispiel TextEditor von eben, jedoch mit dem *FileDialog*:

```
import java.awt.*;
public class TEditor2 extends Frame
{
    private MyWindowListener myWindowListener = new MyWindowListener();
    private MyActionListener myActionListener = new MyActionListener(this);
    public TextArea taText = new TextArea(15,50);
    public Button btnBeenden = new Button("Beenden");
    public Button btnOeffnen = new Button("Öffnen");
    public Button btnSpeichern = new Button("Speichern");
    private Panel pnlNorth = new Panel();

    public TEditor2()
    {
        super("Text Editor Version 2");
        this.pnlNorth.add(this.btnOeffnen);
        this.pnlNorth.add(this.btnSpeichern);
        this.add(this.pnlNorth, BorderLayout.NORTH);
```

```
        this.add(this.taText, BorderLayout.CENTER);
        this.add(this.btnBeenden, BorderLayout.SOUTH);
        this.pack();
        this.show();
        this.addWindowListener(this.myWindowListener);
        this.btnOeffnen.addActionListener(myActionListener);
        this.btnSpeichern.addActionListener(this.myActionListener);
        this.btnBeenden.addActionListener(this.myActionListener);
    }
    public static void main(String args[])
    {
        TEditor2 f = new TEditor2();
    }
}
```

Wie Sie sehen, sehen Sie nichts! In der Hauptklasse taucht der *FileDialog* überhaupt nicht auf. Diese Komponente kommt erst jetzt in *MyActionListener* zum Einsatz:

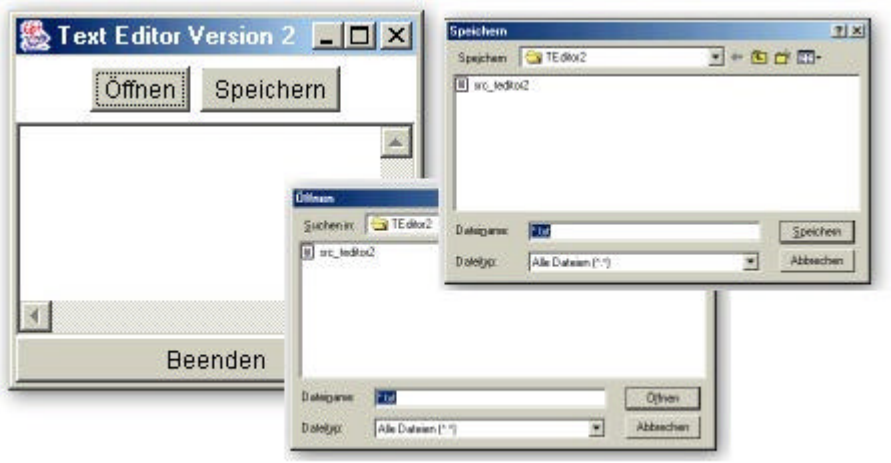
MyActionListener.java

```
import java.awt.event.*;
import java.awt.*;
import java.io.*;
public class MyActionListener implements ActionListener
{
  public TEditor2 f;
  public MyActionListener(TEditor2 f)
  {
       this.f=f;
  }
  public void actionPerformed(ActionEvent e)
  {
    Object obj = e.getSource();
    if(obj == f.btnOeffnen)
    {
       FileDialog fd = new FileDialog(f,"Öffnen",FileDialog.LOAD);
       fd.setFile("*.txt");
       fd.show();
       String file = fd.getDirectory()+fd.getFile();
       if(fd.getFile()!=null)
       {
         BufferedReader in =null;
         f.taText.selectAll();
         f.taText.replaceRange("",f.taText.getSelectionStart(),
         f.taText.getSelectionEnd());
         try
         {
           String zeile=null;
```

```
                in = new BufferedReader(new InputStreamReader(new ¬ FileInput-
                Stream(file)));
                while((zeile=in.readLine())!=null)
                f.taText.append(zeile+"\n");
                in.close();
            }
            catch(FileNotFoundException err)
            {
                System.out.println("Datei nicht gefunden"+err);
            }
            catch(IOException err)
            {
                System.out.println("Lesefehler"+err);
            }
        }
    }
    if(obj==f.btnSpeichern)
    {
        FileDialog fd = new FileDialog(f,"Speichern",FileDialog.SAVE);
        fd.setFile("*.txt");
        fd.show();
        String file = fd.getDirectory()+fd.getFile();
        if(fd.getFile()!=null)
        {
            BufferedWriter out =null;
            try
            {
                out = new BufferedWriter(new OutputStreamWriter(new ¬ FileOut-
                putStream(file)));
                out.write(f.taText.getText());
                out.close();
            }
            catch(FileNotFoundExcept ion err)
            {
                System.out.println("Fehler beim Speichern"+err);
            }
            catch(IOException err)
            {
                System.out.println("Schreibfehler..."+err);
            }
        }
    }
    if(obj==f.btnBeenden)System.exit(0);
  }
}
```

Hier nun sieht man den *FileDialog* mit den angesprochenen Methoden im Einsatz.

Außerdem wird vor jedem Laden die *TextArea* gesäubert (*replaceRange*). Ansonsten entspricht das Programm TEditor.
So sieht das Programm nun aus:



Checkbox
Eine *Checkbox* ist die einfachste Möglichkeit, eine Ja/Nein-Auswahl vom Benutzer zu erhalten. Dabei ist der Begriff '*Checkbox*' etwas irreführend, Kenner werden den Unterschied zwischen einer *Checkbox* und *Radio-Buttons* kennen, allen anderen sei gesagt, dass man beliebig viele *Checkboxen* ankreuzen kann, jedoch immer nur einen *Radio-Button*. Java scheint den Unterschied zumindestens im Namen nicht zu kennen, denn eine *CheckboxGroup* macht aus *Checkboxen Radio-Buttons*.
So ist der Konstruktor aufgebaut:
Checkbox(String label, CheckboxGroup group, boolean state)

Das Label wir immer rechts von der *Checkbox* angezeigt, die *CheckboxGroup* macht die *Checkboxen* zu *Radio-Buttons* und mit dem *state* kann man schon beim Entwurf festlegen, ob die *Checkbox* angekreuzt ist.
Unser Beispiel soll einen vom Benutzer eingegebenen Text formatieren, dabei soll der Benutzer ankreuzen können ob der Text fett oder/und kursiv ist und wie groß der Text werden soll. Und so sieht der Quelltext aus:

```
import java.awt.*;
import java.awt.event.*;
public class TFormat extends Frame
{
    public Checkbox cb1 = new Checkbox("fett");
    public Checkbox cb2 = new Checkbox("kursiv");
    public CheckboxGroup cbg = new CheckboxGroup();
    public Checkbox cb3 = new Checkbox("14 pt",cbg,false);
    public Checkbox cb4 = new Checkbox("16 pt",cbg,false);
    public TextField tf1 = new TextField(20);
    public Label lblDisplay = new Label("");
    public Button btnFormat = new Button("Formatieren");
    public Panel pnlNorth = new Panel(new GridLayout(1,2));
    public Panel pnlCenter = new Panel(new FlowLayout(FlowLayout.LEFT));
    public Panel pnlSouth = new Panel(new BorderLayout());
```

```
    public Panel pnlNorthL = new Panel(new FlowLayout(FlowLayout.LEFT));
    public Panel pnlNorthR = new Panel(new FlowLayout(FlowLayout.RIGHT));
    private MyWindowListener myWindowListener = new MyWindowListener();
    private MyActionListener myActionListener = new MyActionListener(this);
    public TFormat()
    {
        super("Text-Formatierung");
        this.pnlNorthL.add(this.cb1);
        this.pnlNorthL.add(this.cb2);
        this.pnlNorthR.add(this.cb3);
        this.pnlNorthR.add(this.cb4);
        this.pnlNorth.add(this.pnlNorthL);
        this.pnlNorth.add(this.pnlNorthR);
        this.pnlCenter.add(this.tf1);
        this.pnlCenter.add(this.btnFormat);
        this.pnlSouth.add(this.lblDisplay);
        this.add(this.pnlNorth,BorderLayout.NORTH);
        this.add(this.pnlCenter,BorderLayout.CENTER);
        this.add(this.pnlSouth,BorderLayout.SOUTH);
        this.pack();
        this.show();
        this.addWindowListener(this.myWindowListener);
        this.btnFormat.addActionListener(myActionListener);
    }
    public static void main(String args[])
    {
        TFormat f = new TFormat();
    }
}
```

MyActionListener.java

```
import java.awt.*;
import java.awt.event.*;
import java.text.*;
public class MyActionListener implements ActionListener
{
  public TFormat f;
  public MyActionListener(TFormat f)
  {
        this.f=f;
  }
  public void actionPerformed(ActionEvent e)
  {
    Object obj = e.getSource();
    int style;
    style = 0;
    int size;
```

```
            size = 12;
            if(obj == f.btnFormat)
            {
               if(f.cb1.getState()==true)
               {
                  style = 1;
               }
               if(f.cb2.getState()==true)
               {
                  style = 2;
               }
               if(f.cb3.getState()==true)
               {
                  size = 14;
               }
               if(f.cb4.getState()==true)
               {
                  size = 16;
               }
               String text;
               text = f.tf1.getText();
               f.lblDisplay.setFont(new Font("",style,size));
               f.lblDisplay.setText(text);
            }
         }
      }
```

Das einzig neue an diesem Quelltext ist die Verwendung der Methode *getState()*, damit wird überprüft, ob die Checkbox angekreuzt ist. So sieht unser Machwerk aus:

Man könnte das Problem aber auch anders angehen und die Formatierung des Textes direkt nach dem Ankreuzen durchführen (also ohne Button-Einsatz). Dafür gibt es den *ItemListener*. Dieser verarbeitet z.B. Checkboxereignisse und wird in der Hauptklasse so eingesetzt wie der *ActionListener*. Die Klasse *MyItemListener* implementiert die Schnittstelle ItemListener:
public class MyItemListener implements ItemListener

Diese schreibt nur eine Methode vor: *public void itemStateChanged(ItemEvent e)*
Also ob sich der Zustand der Checkbox geändert hat.

Choice

Die Komponente *Choice* stellt eine Pulldownliste dar, die man auch unter den Namen Optionsmenü oder Combobox kennt. Es gibt folgenden Konstruktor: *Choice()* Die einzelnen Elemente der Pulldownliste fügt man mit der Methode *add(String eintrag)* hinzu.

Wie bei der *CheckBox* werden Ereignisse der Zustandsänderung über die Schnittstelle *ItemListener* verarbeitet. Nun der Quelltext zu unserer Beispielanwendung, die die Hintergrundfarbe ändert, wenn eine andere Farbe in der Choice-Komponente ausgewählt wurde:

```
import java.awt.*;
public class Backgr extends Frame
{
    private MyItemListener myItemListener= new MyItemListener(this);
    public Label lblFarbe = new Label("Farbe:");
    public Choice chcFarbe = new Choice();
    public Backgr()
    {
        super("Background-Changer");
        this.setLayout(new FlowLayout());
        this.chcFarbe.add("rot");
        this.chcFarbe.add("grün");
        this.chcFarbe.add("blau");
        this.chcFarbe.add("gelb");
        this.add(this.lblFarbe);
        this.add(this.chcFarbe);
        this.setBackground(Color.red);
        this.setSize(200,100);
        this.show();
        this.addWindowListener(new MyWindowListener());
        this.chcFarbe.addItemListener(myItemListener);
    }
    public static void main(String args[])
    {
        Backgr f = new Backgr();
    }
}
```

MyItemListener.java

```
import java.awt.event.*;
import java.awt.*;
public class MyItemListener implements ItemListener
{
  public Backgr f;
  public MyItemListener(Backgr f)
  {
     this.f=f;
  }
  public void itemStateChanged(ItemEvent e)
  {
    switch(f.chcFarbe.getSelectedIndex())
```

```
        {
          case 0: f.lblFarbe.setBackground(Color.red);
                  f.setBackground(Color.red);break;
          case 1: f.lblFarbe.setBackground(Color.green);
                  f.setBackground(Color.green);break;
          case 2: f.lblFarbe.setBackground(Color.blue);
                  f.setBackground(Color.blue);break;
          case 3: f.lblFarbe.setBackground(Color.yellow);
                  f.setBackground(Color.yellow);break;
        }
        f.show();
      }
    }
```

Es gibt mehrere Möglichkeiten den selektierten Eintrag von *Choice* herauszufinden:

- *public String getSelcetedItem()* liefert den selektierten Eintrag als String zurück
- *public int getSelectedIndex()* liefert den selektierten Eintrag als int zurück

Bei einem Stringvergleich ist es allerdings unmöglich mit einer einfachen mehrfachen Auswahl den Vergleich durchzuführen. Bei dem Einsatz des int-Vergleichs muss man jedoch die genaue Position des Items in der Liste kennen.
So sieht das Programm dann aus:

List

Das *List*-Objekt ist der vorher behandelten *Choice*-Komponente nicht unähnlich. Der Unterschied besteht darin, dass bei *List* immer mehrere Einträge sichtbar sind. Sind mehr Einträge in der *List* als angezeigt werden können, so werden die restlichen nach dem Scrollen sichtbar.
Der Konstruktor sieht so aus: *List(int zeilen, boolean multipleMode)*
Wenn die Zahl der Zeilen nicht angegeben wird, ist die Größe allein durch den Layout-Manager bestimmt. Mit dem Parameter *multipleMode* kann festgelegt werden, ob mehrere Einträge gleichzeitig ausgewählt sein dürfen.
Hier nun der Quelltext unseres Beispielprogramms:

```
import java.awt.*;
public class Liste extends Frame
{
    public List lstNamen = new List(10,true);
    public Button btnHinzu = new Button(">>");
```

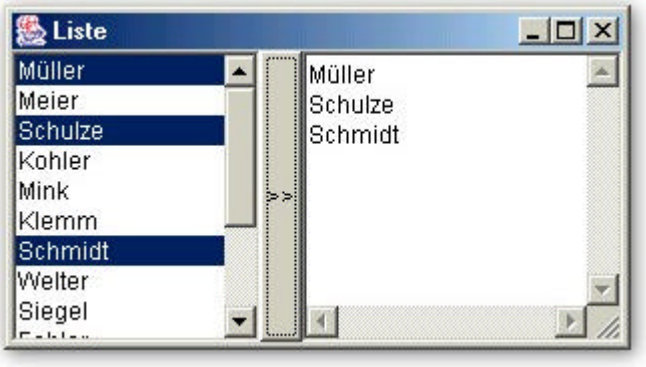
```
    public TextArea taNamen= new TextArea(10,20);
    private MyActionListener myActionListener = new MyActionListener(this);
    public Liste()
    {
        super("Liste");
        this.lstNamen.add("Müller");
        this.lstNamen.add("Meier");
        this.lstNamen.add("Schulze");
        this.lstNamen.add("Kohler");
        this.lstNamen.add("Mink");
        this.lstNamen.add("Klemm");
        this.lstNamen.add("Schmidt");
        this.lstNamen.add("Welter");
        this.lstNamen.add("Siegel");
        this.lstNamen.add("Fohler");
        this.lstNamen.add("Ryan");
        this.lstNamen.add("Gross");
        this.lstNamen.add("Rohr");
        this.lstNamen.add("Schott");
        this.add(this.lstNamen, BorderLayout.WEST);
        this.add(this.btnHinzu, BorderLayout.CENTER);
        this.add(this.taNamen, BorderLayout.EAST);
        this.pack();
        this.show();
        this.addWindowListener(new MyWindowListener());
        this.btnHinzu.addActionListener(myActionListener);
    }
    public static void main(String args[])
    {
        Liste f = new Liste();
    }
}
```

MyActionListener.java

```
import java.awt.event.*;
public class MyActionListener implements ActionListener
{
  public Liste f;
  public MyActionListener(Liste f)
  {
    this.f=f;
  }
  public void actionPerformed(ActionEvent e)
  {
    String feld[]=f.lstNamen.getSelectedItems();
    for(int i= 0; i<feld.length; i++)
    f.taNamen.append(feld[i]+"\n");
    f.show();
  }
```

```
}
```

Wird der Button gedrückt, werden alle selektierten Einträge der Liste über die Methode *getSelectedItem()* in ein String-Array geschrieben, das danach über eine Schleife in die *TextArea* hinzugefügt wird.
So sieht unser Machwerk dann aus:



Menüs
Natürlich gibt es auch die Möglichkeit ein Menü anzulegen. Mit folgendem Konstruktor wird in Java ein Menü angelegt:
Menu(String label, boolean beweglich)
Zusätzliche Menüpunkte können auch noch zur Laufzeit mit der Methode *setLabel()* hinzugefügt werden. Der Parameter *beweglich* ist nur unter UNIX einsetzbar, da Windows keine beweglichen Menüs, die an eine beliebige Stelle gezogen werden können, unterstützt.
Soll sich das Menü auch ausklappen lassen, dann muss man die *MenuBar* einsetzen. Folgender Konstruktor erzeugt eine leere *MenuBar*: *MenuBar menueZeile = new MenuBar();*
Als Menüpunkte kommen normale *MenuItems* infrage oder aber *CheckboxMenuItems*, die zusätzlich ein Häkchen vor dem Menüpunkt anzeigen können. Die Konstrutoren sehen so aus:
MenuItem(String label, MenuShortcut taste)
CheckboxMenuItem(String label, boolean status)

Beim *MenuItem* kann man mit dem Parameter *taste* eine Tastenkombination (z.B. [Alt]+[D] für Datei) angeben, wird diese gedrückt, wird der Menüpunkt ausgewählt. Diese Tastenkombination (auch *Acceleration-Key* genannt) wird nach folgendem Schema angegeben *KeyEvent.VK_D*
Also *VK* zuzüglich der Bezeichnung der entsprechenden Taste, *VK* repräsentiert also die [Alt]-Taste (beim Mac die Command-Taste).
Beim *CheckboxMenuItem* kann man mit dem Parameter *status* festlegen ob bereits das Häkchen vor dem Menupunkt gesetzt worden ist.
Ein Klick auf einen Menueintrag ruft (wie beim Button) ein *ActionEvent* hevor, darum ist das Menu auch so zu behandeln wie der Button. Ein *CheckboxMenuItem* erzeugt ein *ItemEvent*, wenn ein Häkchen gesetzt wurde.
So sieht der Quelltext unseres Beispielprogramms aus:

```
import java.awt.*;
import java.awt.event.*;
```

```
public class Editor3 extends Frame
{
    public TextArea taText = new TextArea(15,50);
    public MenuBar menuBar = new MenuBar();
    public Menu menuDatei = new Menu("Datei");
    public MenuItem miOpen = new MenuItem("Öffnen...", new MenuShort-
    cut(KeyEvent.VK_O));
    public MenuItem miSave = new MenuItem("Speichern...", new MenuShort-
    cut(KeyEvent.VK_S));
    public MenuItem miClose = new MenuItem("Schließen", new MenuShort-
    cut(KeyEvent.VK_C));
    public MenuItem miEnd = new MenuItem("Beenden", new MenuShort-
    cut(KeyEvent.VK_B));
    private MyActionListener myActionListener = new MyActionListener(this);
    public Editor3()
    {
        super("Editor3");
        this.menuDatei.add(miOpen);
        this.menuDatei.add(miSave);
        this.menuDatei.add(miClose);
        this.menuDatei.add(miEnd);
        this.menuBar.add(this.menuDatei);
        this.setMenuBar(this.menuBar);
        this.add(this.taText, BorderLayout.CENTER);
        this.pack();
        this.show();
        this.addWindowListener(new MyWindowListener());
        this.miOpen.addActionListener(myActionListener);
        this.miSave.addActionListener(myActionListener);
        this.miClose.addActionListener(myActionListener);
        this.miEnd.addActionListener(myActionListener);
    }
    public static void main(String args[])
    {
        Editor3 f = new Editor3();
    }
}
```

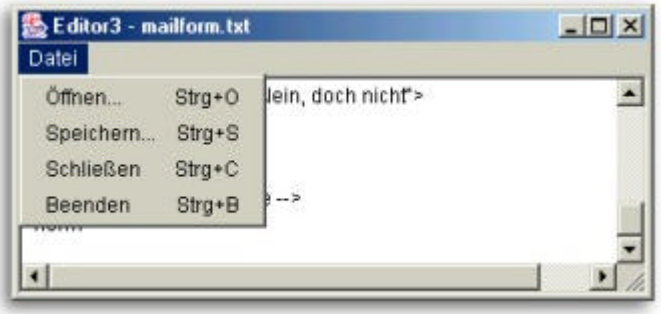
MyActionListener.java

```
import java.awt.event.*;
import java.awt.*;
import java.io.*;
public class MyActionListener implements ActionListener
{
  public Editor3 f;
  public MyActionListener(Editor3 f)
  {
     this.f=f;
  }
  public void actionPerformed(ActionEvent e)
```

```
{
    if(e.getSource()==f.miOpen)
    {
        FileDialog fd = new FileDialog(f, "Öffnen...", FileDialog.LOAD);
        fd.setFile("*.*");
        fd.show();
        String file = fd.getFile();
        if(file!=null)
        {
            this.fileClose();
            f.setTitle("Editor3 - "+file);
            file=fd.getDirectory()+file;
            BufferedReader in=null;
            try
            {
                String zeile=null;
                in = new BufferedReader(new InputStreamReader(new FileIn-
                putStream(file)));
                while((zeile=in.readLine())!=null)
                f.taText.append(zeile+"\n");
                in.close();
            }
            catch(FileNotFoundException err)
            {
                System.out.println("Datei-Fehler..."+err);
            }
            catch(IOException err)
            {
                System.out.println("Lesefehler..."+err);
            }
        }
    }
    if(e.getSource()==f.miSave)
    {
        FileDialog fd = new FileDialog(f, "Speichern...", FileDialog.SAVE);
        fd.setFile("");
        fd.show();
        String file = fd.getFile();
        if(file!=null)
        {
            f.setTitle("Editor3 - "+file);
            file=fd.getDirectory()+file;
            BufferedWriter out=null;
            try
            {
                out = new BufferedWriter(new OutputStreamWriter(new File-
                OutputStream(file)));
                out.write(f.taText.getText());
```

```
                out.close();
            }
            catch(FileNotFoundException err)
            {
                System.out.println("Datei-Fehler..."+err);
            }
            catch(IOException err)
            {
                System.out.println("Schreibfehler..."+err);
            }
        }
    }
    if(e.getSource()==f.miClose)
    {
        f.setTitle("Editor3");
        this.fileClose();
    }
    if(e.getSource()==f.miEnd) System.exit(0);
  }
  private void fileClose()
  {
    f.setTitle("Editor3");
    f.taText.selectAll();
    f.taText.replaceRange("",0,f.taText.getSelectionEnd());
  }
}
```

Wir haben mit der fileClose-Methode Anweisungen ausgegliedert. So kann man die Methode fileClose() an zwei Stellen (open und close) einsetzen und spart sich so unteranderem die Anweisungen zweimal schreiben zu müssen.
So sieht unser Frame mit dem Menü aus:



Swing
Wie ich eingangs erklärt habe, sind die *Swing*-Klassen die Weiterentwicklung des *AWT*. Sie sind ebenfalls zur grafischen Benutzeroberflächengestaltung entworfen, zeichnen jedoch jede Komponente durch Java-Code und greifen nicht wie das AWT auf Betriebssystemsfunktionen zurück. Somit sehen auch die durch Swing erstellten

Komponenten überall gleich aus.
Ich möchte nur kurz auf die Unterschiede von Swing zum AWT eingehen.

JFrame und JDialog

JFrame und *JDialog* sind neben *JWindow* und *JApplet* die einzigen Klassen von *Swing*, die auf die klassischen AWT-Oberklassen zugreifen. *JFrame* (*javax.swing.JFrame*) hat die AWT-Oberklasse *Frame* (*java.awt.Frame*) und *JDialog* (*javax.swing.JDialog*) hat *Dialog* (*java.awt.Dialog*) als Oberklasse.
Somit wird klar, wie das Swing-Konzept entworfen wurde: Die Darstellung des Fensterahmens (z.B. durch das JFrame/Frame-Objekt) bestimmt das Betriebssystem, das Innere des Fensters wird dagegen komplett von Java übernommen.

Aufbau der JRootPane

Soviel zu dem konzeptionellen Unterschied der beiden Techniken. Es gibt aber noch weitere Unterschiede und mit diesen wollen wir uns nun beschäftigen. Der innere Bereich des Fensters wird durch einen einzigen untergeordneten Container vom Typ *JRootPane* verwaltet, der wiederum aus zwei Komponenten besteht:

- Das *glassPane*-Objekt sitzt als oberste Komponente über dem kompletten sichtbaren Bereich des inneren Fensters und ist normalerweise nicht sichtbar. Der Sinn dieser Komponente besteht darin, einmal sichtbar gemacht, alle Ereignisse im Fensterbereich abzufangen, um auf diese Weise z.B. länger andauernde Aktionen zu schützen (Blockieren von Maus/Tastatur)
- Das *layeredPane*-Objekt verwaltet die untergeordneten Schichten des Fensters in mehreren Bereichen. Dabei werden die Komponenten einer höheren Schicht immer über die der darunter liegenden Schicht gezeichnet. Dadurch lassen sich u.a. Menüs konstruieren, deren Inhalt über den restlichen visuellen Dialogelementen gezeichnet werden muss.

Das *layeredPane*-Objekt enthält zwei untergeordnete Komponenten. Diese Komponenten sind die Menüleiste (vom Typ *JMenuBar*) sowie der darunter liegende Container *contentPane*, dem die visuellen Dialogkomponenten (z.B. Buttons) hinzugefügt werden.
Durch diesen Sachverhalt ergibt sich ein wesentlicher Unterschied beim Hinzufügen von Komponenten zu einem *Swing*-Fenster. Es reicht nicht mehr aus, die Komponenten einfach mittels *add*-Methode des *JFrame*-Objekts aufzurufen. Man muss die Komponenten dem *contentPane*-Objekt hinzufügen. Natürlich muss man dieses Objekt erst mit einer Methode (der getContentPane-Methode des Fensters) erzeugen.

JButton

Der *JButton* verhält sich genauso wie der aus dem AWT bekannte Button. Zusätzlich gibt es aber auch noch die Möglichkeit ein Icon auf dem Button zu platzieren. So sieht der Konstruktor aus: *JButton(String str, javax.swing.Icon icon)*
Man kann natürlich sowohl das Icon als auch den Text weglassen.
Beim folgenden Quelltext ist wieder die bekannte "Drei-Klassen-Gesellschaft" (Hauptklasse, MyActionListener, MyWindowListener) die wir bereits vom AWT kennen wieder zu finden. Hier ist die Hauptklasse:

```
import java.awt.*;
import javax.swing.*;
public class JFrame1 extends JFrame
{
```

```
    private MyWindowListener myWindowListener = new MyWindowListener();
    private MyActionListener myActionListener = new MyActionListener();
    private JButton btnBeenden = new JButton("Beenden", new ImageI-
    con("Kreuz.gif"));
    public JFrame1()
    {
       super("Unser erstes Swing-Fenster");
       Container cp = this.getContentPane();
       cp.setLayout(new FlowLayout());
       this.setSize(400,100);
       cp.add(this.btnBeenden);
       this.show();
       this.addWindowListener(myWindowListener);
       this.btnBeenden.addActionListener(myActionListener);
    }
    public static void main(String argv[])
    {
       JFrame1 fenster = new JFrame1();
    }
}
```

Hier sieht man die Unterschiede zum AWT-Konzept: Es wird ein *Container*-Objekt mit Verweis auf die *contentPane*-Ebene des Fensters angelegt. Das *contentPane*-Objekt ist direkt dafür verantwortlich, wie die Komponenten auf der Oberfläche platziert werden. Deswegen wird für das Container-Objekt der Layout-Manager definiert. Der *JButton* wird zur *ContentPane*-Ebene hinzugefügt.
Das Icon (in unserem Fall Kreuz.gif) muss im Verzeichnis liegen in dem sich auch die Quellcode-Dateien befinden.
Durch die Ähnlichkeit zum AWT-Design ändert sich am Aufbau der Listener nichts. Man kann die Listener im JCreator hinzufügen, indem man mittels Rechtsklick im *FileView* (links oben) auf den Projektnamen (z.B. JFrame1) das Kontextmenü öffnet und dort den Menüpunkt *Add* auswählt und dann zu *Add Existing Files* auswählt. Nun kann man die Dateien aus einem anderen Projekt hinzufügen (z.B. aus Fenster6). Es erscheint eine Abfrage ob man die Dateien adden will oder als externe Dateien im Projekt aufführen möchte. Wir wollen die Dateien adden. Schon sind die Dateien eingebunden.
Und so sieht das mit Swing designte Fenster aus: