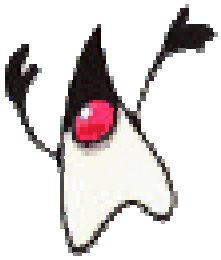
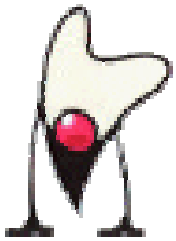
Einführung in die Grafikprogrammierung mit

Java Swing





In diesem Kapitel

- *An Hand eines einfachen Beispiels (Button) werden die wichtigsten Elemente der GUI Programmierung mit Swing erläutert*
  - *Ereignissteuerung*
  - *Adapterklassen*
  - *Layout Manager*

1. Swing - Einführendes Beispiel

1.1. Swing – Einführung

Zur Einführung in die Grafik-Programmierung schauen wir uns ein Beispiel an. Dabei steht die grafische Oberfläche weniger im Vordergrund; vielmehr geht es darum, an Hand eines Beispiels den Ablauf "„ereignisorientierte Programmierung“ kennen zu lernen.

1.2. Java Foundation Classes und Java Swing

Worum geht es überhaupt?
Als erstes wollen wir uns mit folgenden vier Fragen befassen:

- Was ist JFC und was ist Swing?
- Welches Release enthält welche Swing API’s
- Welches Swing Package sollte man einsetzen?
- Wie unterscheiden sich Swing Komponenten von AWT Komponenten?

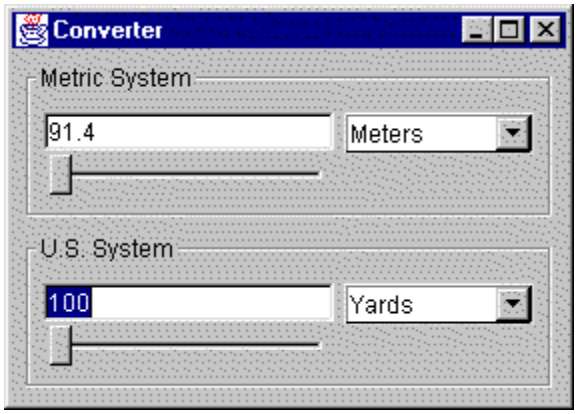
1.2.1. JFC und Swing

JFC steht für Java Foundation Classes, mit deren Hilfe grafische Oberflächen (GUI) gebaut werden können. JFC wurde 1997 angekündigt. Es enthält folgende Funktionsbereiche:

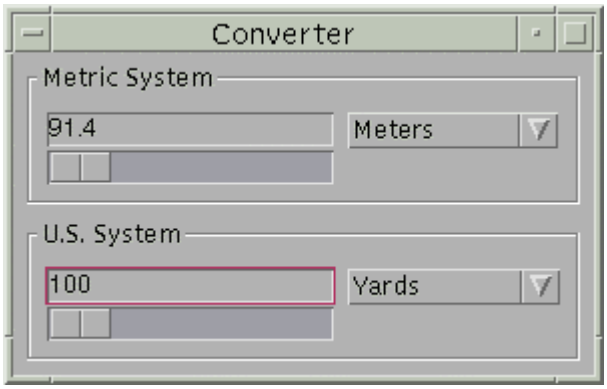
1. **Die Swing Komponenten**
   Diese enthalten zum Beispiel Knöpfe (Buttons), ... Tabellenfenster. Im Anhang finden Sie eine grafische Liste der wichtigsten Elemente.
2. **Anpassbare „Look and Feel“ Unterstützung**
   Damit können Programme unterschiedlichen Standards angepasst werden. Beispielsweise haben Sie die Möglichkeit eine Oberfläche im Stile einer Motif Oberfläche oder einer Wndows Oberfläche, ... oder einem selbst definierten Standard aussehen zu lassen.
3. **Accessibility API – Unterstützung**
   Alle Swing Komponenten stellen Accessibility API zur Verfügung.Mit ihr ist es möglich, auch ohne Maus Zugriff auf die Oberflächenelemente zu haben. Durch ihre hohe Flexibilität erlauben sie sogar Schnittstellen zur Spracherkennung oder beliebigen anderen Eingabegeräten.
4. **Java 2D**
   Die Java 2D API enthält viele neue Möglichkeiten für 2D-Grafik und Bildbearbeitung. Dabei ist es nun möglich, Bilder beliebig zu drehen oder Farbwerte zu manipulieren – um nur einige zu nennen.
5. **Drag & Drop**
   Das Drag & Drop erlaubt nun einen Datentransfer zwischen systemspezifischen Applikationen und Anwendungen, welche mit Java geschrieben wurden. Natürlich ist auch der Austausch unter verschiedenen Java-Aplikationen möglich.

1.2.1.1. Beispiel für unterschiedliche Look & Feel

Windows Version



Motif Version



1.2.2. Swing API Varianten

Es existieren m Moment noch zwei unterschiedliche Versionen von Swing:

- Swing als Bestandteil von JDK ab JDK 1.2
- Swing als JFC 1.1 für JDK 1.1

1.2.3. Welche Swing Variante sollte ich einsetzen?

Es hängt also von der Version von JDK ab, welche Variante Sie konkret einsetzen müssen oder können.

1.2.4. Inwiefern unterschieden sich Swing Komponenten von AWT ?

AWT wurde für die Version 1 von JDK entwickelt. Java 2 unterstützt zwar AWT noch immer; aber Sun empfiehlt AWT nicht mehr einzusetzen. Alle Swing Komponenten haben einen Namen, der mit J anfängt, zum Beispiel **JButton** im Package **javax.swing**. Das selbe Konstrukt unter AWT wäre also einfach **button** im Package **java.aw**t.

Da Swing ohne plattformabhängigen Programmcode implementiert wurde, stehen diese Funktionen auf allen Plattformen zur Verfügung. Auch die Funktionalität wurde im Vergleich zu AWT massiv erweitert.

Einige Beispiele, welche die Unterschiede zwischen AWT und Swing verdeutlichen sollten:

- Swing Buttons können mit Bildern hinterlegt werden.
- Swing Elemente können rund oder eckig sein (runde Knöpfe, eckige Knöpfe)
- Hilfstechnologien können leicht selbst den Text eines Knopfes bestimmen.
- Look & Feel kann plattformübergreifend definiert und verändert werden, selbst dynamisch, also im Laufe der Programmausführung.

Wir wollen im Folgenden das einfache Programm:

entwickeln.

Und hier die Vorgehensweise:

1. **Installation des neusten Releases von JDK**
   Das haben Sie bereits gemacht, falls Sie die aktuelle Version von JBuilder installiert haben.
2. **Kreieren eines Programms mit Swing Komponenten**

Das Beispielprogramm für die obige einfache Applikation:

```
import javax.swing.*;          //Package Name ab Java 2

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

public class SwingApplikation {
    private static String labelPrefix = "Anzahl Clicks: ";
    private int numClicks = 0;

    public Component createComponents() {
        final JLabel label = new JLabel(labelPrefix + "0    ");

        JButton button = new JButton("Das ist ein Swing Button!");
        button.setMnemonic(KeyEvent.VK_I);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                numClicks++;// Anzahl Mausklicks
                label.setText(labelPrefix + numClicks);
            }
        });
        label.setLabelFor(button);

        /*
         * Schaffen eines dünnen Rahmens :
         * JPanel +  "leerer" Rand.
         */
        JPanel pane = new JPanel();
        pane.setBorder(BorderFactory.createEmptyBorder(
                                        30, //top
                                        30, //left
                                        10, //bottom
                                        30) //right
                                        );
        pane.setLayout(new GridLayout(0, 1));
        pane.add(button);
        pane.add(label);

        return pane;
    }
```

```
    public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(
                UIManager.getCrossPlatformLookAndFeelClassName());
        } catch (Exception e) { }

//Definition des top-level Container und Hinzufügen weiterer
// Bestandteile.
        JFrame frame = new JFrame("Swing Beispiel");
        SwingApplikation app = new SwingApplikation();
        Component contents = app.createComponents();
        frame.getContentPane().add(contents, BorderLayout.CENTER);

        //Frame Aufbau abschliessen und anzeigen
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);//schliessen vom Fenster enablen
            }
        });
        frame.pack();            // optimiert den Layout
        frame.setVisible(true); // na ja
    }
}
```

3. **Übersetzen des Java Programms**
   Da wir JBuilder oder JDK 1.2+ verwenden, besteht kein Problem. Die Übersetzung sollte problemlos sein.
4. **Starten des Java Programms**
   Entweder lassen Sie das Programm aus der Entwicklungsumgebung heraus laufen, oder Sie starten es ab Kommandozeile
   java –cp . SwingApplikation
   wobei der Punkt nach –cp auf das aktuelle Verzeichnis zeigt.

Für Swing Applets geht man analog vor. Wir werden im Folgenden mehrere Applets entwickeln und testen.

1.3. Ein einfaches Swing Beispiel

Nachdem wir den Ablauf für die Erstellung und das Starten eines Swing Programms kennen, wollen wir in mehreren Schritten die obige Demo Applikation entwickeln. Sie dient als Orientierung für weitere Beispiele. Am Schluss werden wir ein einfaches Notepad entwickeln, mit Cut & Paste, Speichern und Lesen, und einfach den typischen Notepad Funktionen, inklusive Einfügen von Grafiken.

Fangen wir also einfach an: hier nochmals unsere Zielapplikation

Beim Klicken auf den Knopf wird der Zähler um eins erhöht und angezeigt. Nach 20 Klicken:

Im Einzelnen gehen wir wie folgt vor:

- importieren der Swing Pakete
- setzen des Look & Feel
- Definition eines Containers, in den die andern grafischen Komponenten eingefügt werden
- Definition des Knopfs und des Textes
- Hinzufügen der Komponenten zum Container bzw. Containers
- Hinzufügen von Umrandungen (Borders) um die Komponenten
- Programmierung der Ereignissteuerung
- Allfällige Behandlung von Threading Fragen
- Unterstützung von Assistenten-Technologien

1.3.1. Importieren der Swing Pakete

Diese Aufgabe ist natürlich einfach zu bewerkstelligen:

```
import javax.swing.*;          //Package Name ab Java 2
```

Unter Java 1.1 wurden die Pakete noch anders benannt: com.sun.java.swing....

Viele Swing Programme verwenden zudem einige der (alten) AWT Komponenten. Also importieren wir diese auch noch, wenigstens die wichtigsten:

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

1.3.2. Setzen des Look & Feel

Nehmen wir der Einfachheit halber an, dass sich der Benutzer nicht um das Look & Feel kümmern möchte und einfach das jeweilige Standard Look & Feel verwenden möchte, also Windows Look & Feel unter Windows, Motif Look & Feel unter Unix, ... (die Applikationen unter Java sind ja portabel). Dann können wir das Look & Feel im main Programm wie folgt angeben:

```
public static void main(String[] args) {
    try {
            UIManager.setLookAndFeel(
                UIManager.getCrossPlatformLookAndFeelClassName());
        } catch (Exception e) { }
```

1.3.3. Definition eines Containers, in den die andern grafischen Komponenten eingefügt werden

Typischerweise enthält ein GUI einen Toplevel Container, also ein Objekt, welches weitere Objekte enthalten kann. Ein solcher Container könnte sein, eine Instanz der Klassen:

- JFrame, im Falle einer Swing Applikation
- JDialog, im Falle eines Dialog Fensters
- JApplet , im Falle eines Applets

Sobald der Anwender den Container JFrame schliesst, endet die Swing Applikation.

```
public class SwingApplikation {
...
  public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(
                UIManager.getCrossPlatformLookAndFeelClassName());
        } catch (Exception e) { }

           JFrame frame = new JFrame("Swing Beispiel");
           // Definition der Komponenten
           ...
           // Hinzufügen der Komponenten zum Container
        frame.getContentPane().add(contents, BorderLayout.CENTER);

        //Frame Aufbau abschliessen und anzeigen
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);//schliessen vom Fenster enablen
            }
        });
        frame.pack();            // optimiert den Layout
        frame.setVisible(true); // na ja
    }
}
```

Mit den Details der Container Klassen werden wir uns noch beschäftigen.

1.3.4. Definition des Knopfs und des Textes

Die meisten grafischen Applikationen verwenden Knöpfe oder ähnliche Konstrukte. Schauen wir uns an, wie man den Knopf definiert und initialisiert:

```
  JButton button = new JButton("Das ist ein Swing Button!");
button.setMnemonic(KeyEvent.VK_I); // Alt-I
button.addActionListener(new ActionListener() {
    // Definition eines ActionListeners
    ...
});
```

Die erste Zeile definiert den Knopf und benennt ihn. Wir könnten auch ein Bild hinterlegen, indem wir den entsprechenden Konstruktor verwenden würden. Auch Mischformen, Text und Bild, sind möglich.

Als nächstes ermöglichen wir das Klicken des Knopfs ohne Mauseinsatz. Dafür definieren wir die Tastenkombination Alt-I als Mnemonik.

Der ActionListener wird dem Objekt **button**, der Instanz von **JButton** hinzugefügt. Im alten AWT System wurde ein völlig anderes Ereignismodell verwendet, welches allerdings zu sehr komplexem Programmcode führt. Wir werden uns dies, als Vergleich, noch ansehen.

Das Textlabel wird durch den folgenden Code definiert:

- **Initialisierung**

```
private static String labelPrefix = "Anzahl Clicks: ";
private int numClicks = 0;
```

- **GUI Code**

```
    final JLabel label = new JLabel(labelPrefix + "0    ");

                ...

            label.setText(labelPrefix + numClicks);
        }
});
label.setLabelFor(button);
```

Bisher war das Programm recht einfach, obschon wir den Event-Handler nicht gebaut haben.

1.3.5. Hinzufügen der Komponenten zum Container bzw. Containers

Swing Applikationen gruppieren die Labels und Knöpfe in einem Container (einem **JPanel**) bevor diese in einem Frame dargestellt werden.

Schauen wir uns den Programmcode für das Panel an:

```
  JPanel pane = new JPanel();
pane.setBorder(BorderFactory.createEmptyBorder(
                                    30, //top
                                    30, //left
                                    10, //bottom
                                    30) //right
                                    );
pane.setLayout(new GridLayout(0, 1));
```

```
pane.add(button);
pane.add(label);
```

Die erste Zeile kreiert das Panel.

Die zweite Zeile zeichnet noch einen Rahmen um das Panel. Dazu wird eine sogenannte Factory, ein spezielles *Entwurfsmuster*, verwendet.

Die dritte Zeile kreiert einen sogenannten Layout Manager, also ein Objekt, welches für die grafische Gestaltung der Oberfläche zuständig ist. Dieser Layout Manager ist besonders einfach: die grafischen Komponenten werden einfach linear aufgelistet, beziehungsweise angezeigt (GridLayout).

Die beiden weiteren Zeilen fügen dem Panel die Komponenten „button“ und „label“ hinzu. Damit wird das Panel für deren Layout verantwortlich. Der Layout Manager muss unter anderem die Dimensionen der grafischen Objekte bestimmen, um eine sinnvolle Anordnung der Komponenten vornehmen zu können.

1.3.6. Hinzufügen von Umrandungen (Borders) um die Komponenten

Den Programmcode für den Rahmen haben wir im vorigen Abschnitt bereits gesehen:

```
pane.setBorder(BorderFactory.createEmptyBorder(
                                30, //top
                                30, //left
                                10, //bottom
                                30) //right
                                );
```

Der Rand ist somit 30 Pixel oben, 30 Pixel auf der linken Seite, 10 Pixel unten und 30 Pixel auf der rechten Seite.

Die Eigenschaft „Border“ erbt **JPanel** von **JComponent**. Dieses Konzept werden wir im Rahmen der Besprechung der Layout Manager eingehender besprechen.

1.3.7. Programmierung der Ereignissteuerung

Unser einfaches Beispiel bietet nicht allzuviel Gelegenheiten Ereignisse zu definieren beziehungsweise abzufragen.

Unser Programm benötigt zwei Event Handler:

- Zum Feststellen der Mausklicks (oder der Alt-I Eingaben)
- Zum Schliessen des Fensters und Beenden des Programms

```
    ...
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
       numClicks++;// Anzahl Mausklicks
       label.setText(labelPrefix + numClicks);
   }
});
    ...

 //Frame Aufbau abschliessen und anzeigen
frame.addWindowListener(new WindowAdapter() {
```

```
            public void windowClosing(WindowEvent e) {
                System.exit(0);//schliessen vom Fenster enablen
            }
        });
```

Da die Ereignissteuerung das Kernstück dieser Art der Programmierung ist, werden wir dieses Thema als erstes im Detail besprechen.

1.3.8. Allfällige Behandlung von Threading Fragen

Unser Programm ist Thread Safe in folgendem Sinne:
die einzigen Veränderungen geschehen mit Hilfe des Event Handlers (die neuste Anzahl Klicks anzeigen). Der Event Handler läuft im selben Thread wie der main Thread. Es können also nicht zwei Threads gleichzeitig versuchen, die Grafik zu verändern. Konflikte sind somit in unserem einfachen Fall nicht möglich.

1.3.9. Unterstützung von Assistenten-Technologien

In unserem Beispiel verwenden wir keine zusätzlichen Technologien (Touch Screen, Spracheingabe, Sprachausgabe). Diese Technologien stellt Java aber im Standard zur Verfügung!

1.4. Vollständiges Beispielprogramm

```
import javax.swing.*;          //Package Name ab Java 2
import java.awt.*;
import java.awt.event.*;

public class SwingApplikation {
    private static String labelPrefix = "Anzahl Clicks: ";
    private int numClicks = 0;

    public Component createComponents() {
        final JLabel label = new JLabel(labelPrefix + "0    ");

        JButton button = new JButton("Das ist ein Swing Button!");
        button.setMnemonic(KeyEvent.VK_I);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                numClicks++;// Anzahl Mausklicks
                label.setText(labelPrefix + numClicks);
            }
        });
        label.setLabelFor(button);

        /*
         * Schaffen eines dünnen Rahmens :
         * JPanel +  "leerer" Rand.
         */
        JPanel pane = new JPanel();
        pane.setBorder(BorderFactory.createEmptyBorder(
                                        30, //top
                                        30, //left
                                        10, //bottom
                                        30) //right
                                        );
        pane.setLayout(new GridLayout(0, 1));
        pane.add(button);
        pane.add(label);

        return pane;
    }

    public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(
                UIManager.getCrossPlatformLookAndFeelClassName());
        } catch (Exception e) { }

//Definition des top-level Container; Hinzufügen von Bestandteilen.
        JFrame frame = new JFrame("Swing Beispiel");
        SwingApplikation app = new SwingApplikation();
        Component contents = app.createComponents();
        frame.getContentPane().add(contents, BorderLayout.CENTER);

        //Frame Aufbau abschliessen und anzeigen
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);//schliessen vom Fenster enablen
            }
        });
        frame.pack();            // optimiert den Layout
        frame.setVisible(true); // na ja
    }
}
```

1.5. Aufgaben

Im Skript haben wir eine einfache Anwendung von Grund auf von Hand entwickelt.
In der Regel wird man aber die Möglichkeiten der Entwicklungsumgebung ausnutzen wollen, in unserem Falle also den Design Modus von JBuilder.

Ihre Aufgabe:
Entwickeln Sie die selbe Applikation (Button, Text Label) im JBuilder unter Verwendung aller JBuilder Hilfsmittel:

- Design Modus
- wählen Sie nicht einfach "Datei neu Klasse" sondern "Datei neu ???"

Vergleichen Sie den von JBuilder generierten Code mit der Lösung im Skript.
Ihr Kommentar?

In diesem Kapitel
- *Ereignismodell*
- *Anwendung der Ereignissteuerung*
- *Standardereignisse des AWT*
- *Swing Ereignisse*
- *Zusammenfassung*

2. Ereignisse

2.1. Das Ereignismodell 1.1

Die Swing Klassen der JFC arbeiten ausschliesslich mit dem Ereignismodell des JDK1.1+. Das Funktionsprinzip des alten Modells im AWT 1.0 war sehr einfach. Alle Ereignisse wurden in der Klassenhierarchie entsprechend höher geleitet. Man zwang damit alle Objekte, jedes Ereignis durch Überladen der Methoden `handleEvent`() und `action`() abzufangen und zu verarbeiten. Das Programm musste durch aufwendige `if...then`-Anweisungen die Flut der Ereignisse auswerten und ermitteln, welche Objekte, welches Ereignis auslösten. In Anbetracht der ohnehin schon niedrigen Ausführungsgeschwindigkeit der grafischen Java Programme war diese Vorgehensweise für grössere Anwendungen untragbar.

Wesentlich mächtiger ist dagegen die Ereignissteuerung des JDKs 1.1+ auf der Basis von Callback Funktionen. Ereignisse werden nunmehr durch ihre Klassen und nicht durch ihre ID identifiziert. Ausserdem werden sie direkt von der Quelle zu einem bestimmten Empfänger (Listerer / Lauscher) weitergeleitet. Man nennt das Prinzip auch *Delegation*. Der Entwickler legt von vornherein fest, welche Objekte welche Ereignisse zu interessieren haben. Sie werden individuell verteilt. Objekte können nun ein Ereignis auslösen und es zu einem Listener oder einer Gruppe von Listeners delegieren. Will ein Objekt nur ein bestimmtes Ereignis weiterleiten, muss es mit einem Listener Objekt versehen werden. Die `set`/`add` Listener Methoden (zum Beispiel `addActionListener`()) übernehmen dies. Es ist jedoch nicht mehr möglich, Ereignisse vom Typ 1.0 zu empfangen bzw. weiterzuleiten. Obwohl JFC eine Erweiterung des AWT 1.1 darstellt (AWT 1.1 ist abwärtskompatibel zum AWT 1.0) haben die Swing Komponenten die Abwärtskompatibilität nicht geerbt.

Als Listener kann prinzipiell jede Klasse in Frage kommen. Sie muss lediglich die entsprechende Schnittstelle implementieren. Betrachten wir ein Beispiel mit dem `ActionListener`. Er ist wohl einer der häufigsten und am einfachsten anzuwendenden Listener und bietet nur eine Funktion : `actionPerformed`().Folgende Schritte zeigen seine Anwendung:

- Zuerst muss die Listener Klasse mit der `ActionListener` Schnittstelle ausgestattet werden. Falls jemand eine eigene Lauscher –Schnittstelle entwickelt, muss es sie mit `extends` erweitern.

```
public class MeinListener implements ActionListener {
...
}
```

- Es folgt die Implementation der Methoden des Listeners, das heistt was letztendlich geschehen soll, wenn das Ereignis eintritt. In diesem Beispiel ist es nur eine:

```
public class MeinListener implements ActionListener {
   ...
   public void actionPerformed(ActionEvent e)  {
         System.out.println(„Das Ereignis ist eingetreten“);
```

```
            // weitere Aktionen
            ...
      }
   ...
   }
```

- Nun wird die Oberfächenkomponente mit dem Lauscher verbunden. Derselbe Lauscher kann mehreren Komponenten gleichzeitig zugeordnet werden.

```
Button meinButton = new meinButton();
...
meinButton.addActionListener(new MeinListener() );
…
```

2.2. Anwendung der Ereignissteuerung

Das nächste Beispiel zeigt die Ereignissteuerung vom AWT 1.1 anhand eines einfachen Buttons. Bei seiner Bestätigung folgt ein Signalton. Es sei hier anzumerken, dass der Listener die Applet-Klasse selbst ist. Man übergibt bei `addActionListener()` einfach `this`.

Listing 2-1 Anwendung der Ereignissteuerung

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


public class Ereignissteuerung extends Applet implements ActionListener
{
  Button myButton;

  public void init()
  {
    setLayout(new BorderLayout());
    myButton = new Button("Hier klicken.");
    add("Center", myButton);
    myButton.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e)
  {
    Toolkit.getDefaultToolkit().beep();
    System.out.println("Ereignis eingetreten.");
  }
}
```

Und hier auch noch ein Snapshot diese Applets

Abbildung 2-1 Ereignissteuerung und Tonsignal

Das Beispiel verwendet nur einen Listener, der dem Button zugeordnet ist : `meinButton.addActionListener()`

Tatsache ist jedoch, dass eine Komponente mit beliebig vielen und unterschiedlichen Listeners registriert werden kann. Umgekehrt können unbegrenzt viele Komponenten sich einen Listener teilen. Das nächste Beispiel demonstriert dies anhand zweier Buttons und zweier Listener. Bei Button eins geht das Ergebnis nur zu einem Listener, Button zwei hingegen sendet das Ereignis an beide.

Listing 2-2 Anwendung mehrerer Listener

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


public class MehrereListener extends Applet implements ActionListener
{
  TextArea   singleArea;
  TextArea   multiArea;
  Button     singleButton, multiButton;

  public void init()
  {
    Label l = null;
    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints c = new GridBagConstraints();
    setLayout(gridbag);

    c.fill = GridBagConstraints.BOTH;
    c.gridwidth = GridBagConstraints.REMAINDER;
    l = new Label("Mehrfach-Lauscher:");
    gridbag.setConstraints(l, c);
    add(l);

    c.weighty = 1.0;
    singleArea = new TextArea(5, 20);
    singleArea.setEditable(false);
    gridbag.setConstraints(singleArea, c);
    add(singleArea);

    c.weightx = 0.0;
    c.weighty = 0.0;
    l = new Label("Einfach-Lauscher:");
    gridbag.setConstraints(l, c);
    add(l);

    c.weighty = 1.0;
    multiArea = new TextArea(5, 20);
    multiArea.setEditable(false);
    gridbag.setConstraints(multiArea, c);
    add(multiArea);

    c.weightx = 1.0;
    c.weighty = 0.0;
    c.gridwidth = 1;
    c.insets = new Insets(10, 10, 0, 10);
    singleButton = new Button("Einfaches Ereignis");
    gridbag.setConstraints(singleButton, c);
    add(singleButton);

    c.gridwidth = GridBagConstraints.REMAINDER;
    multiButton = new Button("Mehrfaches Ereignis");
    gridbag.setConstraints(multiButton, c);
    add(multiButton);

    singleButton.addActionListener(this);
    multiButton.addActionListener(this);

    multiButton.addActionListener(new SingleListener(multiArea));
  }

  public void actionPerformed(ActionEvent e)
  {
    singleArea.append(e.getActionCommand() + "\n");
  }
}
```

```
class SingleListener implements ActionListener
{
  TextArea textArea;

  public SingleListener(TextArea t)
  {
    textArea = t;
  }

  public void actionPerformed(ActionEvent e)
  {
    textArea.append(e.getActionCommand() + "\n");
  }
}
```

mit dem entsprechenden Snapshot :

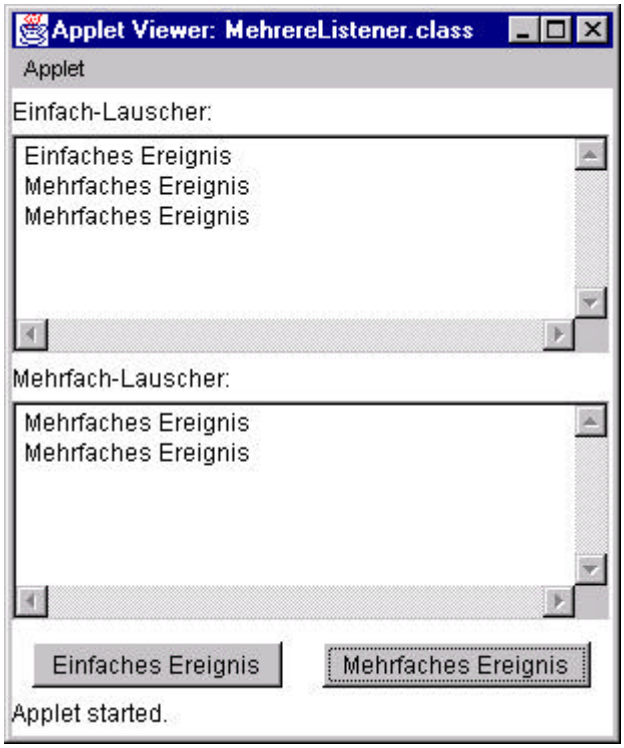


Abbildung 2-2 Anwendung mehrerer Listener

2.2.1. Weitere Ereignistypen

Bis jetzt ging es in den Beispielen nur um das ActionEvent. Doch auch die Verarbeitung etwas komplexerer Ereignisse soll demonstriert werden (wie dies bei Mausoperationen der Fall ist). Der MouseListener hat fünf Methoden zur Auswahl. Sie treten in Kraft, wenn z.B. die Maus bewegt, gedrückt oder losgelassen wird. Das folgende Beispiel beschreibt alle Mausereignisse,welche im freien Bereich eintreten:

Listing 2-3 Anwendung des MouseListener

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

```
public class MehrereListener extends Applet implements ActionListener
{
  TextArea   singleArea;
  TextArea   multiArea;
  Button     singleButton, multiButton;

  public void init()
  {
    Label l = null;
    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints c = new GridBagConstraints();
    setLayout(gridbag);

    c.fill = GridBagConstraints.BOTH;
    c.gridwidth = GridBagConstraints.REMAINDER;
    l = new Label("Einfach-Lauscher:");
    gridbag.setConstraints(l, c);
    add(l);

    c.weighty = 1.0;
    singleArea = new TextArea(5, 20);
    singleArea.setEditable(false);
    gridbag.setConstraints(singleArea, c);
    add(singleArea);

    c.weightx = 0.0;
    c.weighty = 0.0;
    l = new Label("Mehrfach-Lauscher:");
    gridbag.setConstraints(l, c);
    add(l);

    c.weighty = 1.0;
    multiArea = new TextArea(5, 20);
    multiArea.setEditable(false);
    gridbag.setConstraints(multiArea, c);
    add(multiArea);

    c.weightx = 1.0;
    c.weighty = 0.0;
    c.gridwidth = 1;
    c.insets = new Insets(10, 10, 0, 10);
    singleButton = new Button("Einfaches Ereignis");
    gridbag.setConstraints(singleButton, c);
    add(singleButton);

    c.gridwidth = GridBagConstraints.REMAINDER;
    multiButton = new Button("Mehrfaches Ereignis");
    gridbag.setConstraints(multiButton, c);
    add(multiButton);

    singleButton.addActionListener(this);
    multiButton.addActionListener(this);

    multiButton.addActionListener(new SingleListener(multiArea));
  }

  public void actionPerformed(ActionEvent e)
  {
    singleArea.append(e.getActionCommand() + "\n");
  }
}

class SingleListener implements ActionListener
{
  TextArea textArea;

  public SingleListener(TextArea t)
  {
```

```
    textArea = t;
  }

  public void actionPerformed(ActionEvent e)
  {
    textArea.append(e.getActionCommand() + "\n");
  }
}
```

Abbildung 2-3 Anwendung des MouseListener

2.3. AWT Ereignisse

Im AWT ist eine Reihe von Standardereignissen vorhanden, mit denen sich die Ereignisbehandlung der AWT- Komponenten gut abdecken lässt. Alle werden durch entsprechende Listener Schnittstellen implementiert, die im Paket `java.awt.event` zusammengefasst sind.

Die folgende Tabelle gibt eine Übersicht über alle Ereignisarten. Sie beschreibt in der ersten Spalte die Schnittstellen, in der zweiten die Adapter und schliesslich in der dritten Spalte die Methoden der AWT Ereignisse.

Schnittstelle	**Adapterklasse**	**Methoden**
`ActionListener`	Keine	`actionPerformed`
`AdjustableListener`	Keine	`adjustableValueChanged`

`ComponentListener`	`ComponentAdapter`	`componentHidden`
		`componentMoved`
		`componentResized`
		`componentShown`

`ContainerListener`	`ContainerAdapter`	`componentAdded` `componentRemoved`
`FocusListener`	`FocusAdapter`	`focusGained` `focusLost`
`ItemListener`	Keine	`itemStateChanged`

`KeyListener`	`KeyAdapter`	`keyPressed`
		`keyReleased`
		`keyTyped`

`MouseListener`	`MouseAdapter`	`mouseClicked`
		`mouseEntered`
		`mouseExited`
		`mousePressed`
		`mouseReleased`

`MouseMotionListener`	`MouseMotionAdapter`	`mouseDragged` `mouseMoved`
`TextListener`	Keine	`textValueChanged`

`WindowListener`	`WindowAdapter`	`windowActivated`
		`windowClosed`
		`windowClosing`
		`windowDeactivated`
		`windowDeiconified`
		`windowOpened`

Ereignisse werden in zwei Gruppen eingeteilt : Basisereignisse (*low-level events*) oder semantische Ereignisse (*semantic events*) :

- Basisereignisse geben Auskunft über Geschehnisse im Fenstersystem oder über Basiseingaben, wie Maus- und Tastenereignisse. Dazu gehören Komponenten-, Container, Fokus- und Fensterereignisse. Ereignisse von Komponenten zeigen deren Änderungen der Position, Grösse und Sichtbarkeit. Container-Ereignisse werden ausgelöst, wenn

Komponenten in einen Container eingefügt oder entfernt werden. Das Fenster informiert schliesslich über den momentanen Status eines Fensters, wie `Dialog` oder `Frame`.

- Zu semantischen Ereignissen gehören Aktions-, Einstellungs-, Eintrags- und Textereignisse. Sie sind das Ergebnis komponentenspezifischer Benutzeraktionen. Aktionsereignisse beispielsweise werden ausgelöst, wenn der Benutzer einen Button drückt oder ein Element aus einer Liste wählt. Betätigt er einen Rollbalken, kommt ein Einstellereignis zurück. Das Eintragsereignis kommt zum Zug, wenn der Benutzer eine Gruppe vonEinträgen aus einer Liste wählt. Das Textfeld löst ein Textereignis aus, wenn sich sein Text ändert.

Es folgt nun eine detaillierte Beschreibung der AWT Ereignisarten, der dazugehörigen Listener Schnittstellen und ihrer Methoden.

2.3.1. ActionEvent

Das `ActionEvent` gehört zu den häufigsten Ereignissen, die in Programmen auftreten. Es wird ausgelöst, wenn der Anwender einen Button drückt, einen Listeneintrag doppelklickt, einen Menübefehl wählt oder die Return Taste in einem Textfeld drückt,

2.3.1.1. Listener

`ActionListener`

2.3.1.2. Methoden des Listeners

- `public void` **`actionPerformed`**`(ActionEvent e)`
  sobald das `ActionEvent` auftritt wird diese Methode aufgerufen und das `ActionEvent` übergeben.

2.3.1.3. Methoden des Ereignisses

- `public String` **`getActionCommand()`**
  ermittelt das ereignisauslösende Kommando. Defaultmässig ist dies die Beschriftung der Komponente. Es kann jedoch explizit mit `setActionCommand`() verändert werden.
- `public int` **`getModifier()`**
  gibt Auskunft über die Tasten (Tastatur oder Maustasten), die während der Modifizierung der Komponente gedrückt wurden. Es handelt sich hierbei um eine Bitmaske, die mit den Konstanten der Klasse ActionEvent abgefragt werden können: `SHIFT_MASK`, `CTRL_MASK`, `META_MASK` und `ALT_MASK`. Zum Beispiel lässt sich durch eine logische UND Verknüpfung feststellen, ob die Alt-Taste gedrückt wurde:

```
actionEvent.getModifiers() & ActionEvent.ALT_MASK
```

  Wenn dieser Ausdruck nicht null ist, wurde die Alt Taste gedrückt.

2.3.2. AdjustmentEvent

Das Ereignis `AdjustmentEvent` tritt auf, wenn sich der Wert einer Komponente ändert, welche die `Adjustable` Schnittstelle implementiert. Im AWT ist dies nur der Rollbalken `ScrollBar`.

2.3.2.1. Listener

`AdjustmentListener`

2.3.2.2. Methoden des Listeners

- `public void` **`adjustmentValueChanged`**`(AdjustmentEvent e)`
  wird aufgerufen, sobald bei Wertänderungen das `AdjustmentEvent` eintritt

2.3.2.3. Methoden des Ereignisses

- `public Adjustable` **`getAdjustable`**`()`
  liefert das `Adjustable` Objekt zurück, welches das `AdjustableEvent` ausgelöst hat.
- `public int` **`getValue`**`()`
  gibt den aktuellen Wert zurück
- `public int` **`getAdjustmentType`**`()`
  gibt den Typen der Wertveränderung zurück. Sie sind in der Klasse AdjustmentEvent vordefiniert: `UNIT_INCREMENT, UNIT_DECREMENT, BLOCK_INCREMENT, BLOCK_DECREMENT` und `TRACK`.

2.3.3. ComponentEvent

Das `ComponentEvent` wird erzeugt, wenn sich die Eigenschaften der Komponente ändert. Es tritt auf, wenn es unsichtbar oder versteckt wird und wenn sich die Grösse oder Position ändert. Zum Beispiel kann ein Oberflächengenerator sich ihrer bedienen, um die Grösse einer Komponente anzuzeigen. Diese Information stecken im `ComponentEvent`.

2.3.3.1. Listener

`ComponentListener`

2.3.3.2. Methoden des Listeners

- `public void` **`componentHidden`**`(ComponentEvent e)`
  wird aufgerufen, wenn die Komponente durch die `setVisible`() Methode versteckt wird.
- `public void` **`componentMoved`**`(ComponentEvent e)`
  wird aufgerufen, wenn die Komponente durch den Benutzer bewegt wird. Das gilt aber nicht für den Container, der es enthält.
- `public void` **`componentResized`**`(ComponentEvent e)`
  wird aufgerufen, wenn sich die Grösse der Komponente ändert
- `public void` **`componentShown`**`(ComponentEvent e)`
  wird aufgerufen, wenn die Komponente durch die `setVisible`() Methode wieder sichtbar gemacht wird

2.3.3.3. Methoden des Ereignisses

- `public Component` **`getComponent`**`()`
  liefert die Komponente zurück, die das Ereignis ausgelöst hat.

2.3.4. ContainerEvent

Das AWT löst `ContainerEvent` aus, wenn dem Container eine Komponente hinzugefügt oder aus ihm entfernt wurde. Es dient allerdings nur zur Information. Das Ereignis wird ausgelöst, damit sich das AWT um das Layout kümmern kann. Für den normalen Programmablauf ist die Registrierung eines `ContainerListener` nicht zwingend erforderlich.

2.3.4.1. Listener

`ContainerListener`

2.3.4.2. Methoden des Listeners

- `public Container` **`getContainer`**`()`
  liefert den Container zurück, dem die Komponente hinzugefügt bzw. aus dem sie entfernt wurde
- `public Component` **`getChild`**`()`
  gibt die Komponente zurück, durch die das Ereignis ausgelöst wurde.

2.3.5. FocusEvent

Ein `FocusEvent` tritt auf, wenn eine Komponente den Tastaturfokus bekommt oder verliert. Die Komponente mit dem Fokus kann auf Tastatureingaben reagieren. Sie sind mit einem etwas dickeren Rahmen gekennzeichnet. Es kann immer nur eine Komponente den Fokus besitzen. Meist kann der Fokus mit der Tab Taste gewechselt werden, lässt sich aber auch programmtechnisch mit `requestFocus`() anfordern.

2.3.5.1. Listener

`ContainerListener`

2.3.5.2. Methoden des Listeners

- `public void` **`focusGained`**`(FocusEvent e)`
  wird aufgerufen, wenn die Komponente den Fokus erhält
- `public void` **`focusLost`**`(FocusEvent e)`
  wird aufgerufen, wenn die Komponente den Fokus verliert.

2.3.5.3. Methoden des Ereignisses

- `public boolean` **`isTemporary`**`()`
  gibt Auskunft darüber, ob das `FocusEvent` nur temporär oder permanent ist

2.3.6. InputEvent

Sie bildet die Basisklasse für alle Eingabeereignisse auf Komponentenebene. Bevor Eingabeereignisse von ihrer Ursprungsquelle verarbeitet werden können, werden sie zu den Listeners gesendet. Somit können die Listener und die Subklassen von Komponenten ein Ereignis konsumieren (`consume`) und die Quelle ihre Standardaktionen nicht mehr ausführen. Für das `InputEvent` selbst gibt es keine Listener Schnittstellen. Die Methoden werden jedoch an die Basisklassen weitergegeben, so dass mit Hilfe von deren Listener Schnittstellen ein Zugriff auf sie möglich ist.

2.3.6.1. Methoden des Ereignisses

- `public void` **`consume`**`()`
  konsumiert das Ereignis
- `public int` **`getModifiers`**`()`
  gibt die Steuerungstasten (SHIFT, CTRL, ...) und Maustasten zurück, die gedrückt wurden. Folgende Konstanten definieren die Tasten und können mit logisch ODER verknüpft werden : SHIFT_MASK, CTRL_MASK, META_MASK, ALT_MASK, ALT_GRAPH_MASK, BUTTON1_MASK
- `public long` **`getWhen`**`()`
  liefert den Zeitpunkt, an dem das Ereignis aufgetreten ist
- `public boolean` **`isConsumed`**`()`
  gibt an, ob das Ereignis aufgetreten ist.

- `public boolean` **`isAltDown`**`()`
  gibt an, ob die Alt Taste gedrückt wurde.
- `public boolean` **`isControlDown()`**
  gibt an, ob die Strg Taste gedrückt wurde.
- `public boolean` **`isMetaDown()`**
  gibt an, ob die Meta Taste gedrückt wurde
- `public boolean` **`isShiftDown()`**
  gibt an, ob die Shift Taste gedrückt wurde.

2.3.7. ItemEvent

Das Ereignis `ItemEvent` tritt nur bei Komponenten ein, die die `ItemSelectable` Schnittstelle implementieren. Meistens sind das Komponenten, die Ein/Aus Zustände aufweisen, Darunter fallen Checkboxen, Menu-Checkboxen, Kombinationsfelder und Listenelemente.

2.3.7.1. Listener

`ItemListener`

2.3.7.2. Methoden des Listeners

- `public void` **`itemStateChanged`**`(Itemevenet e)`
  wird aufgerufen, wenn sich der Zustand der Komponente ändert.

2.3.7.3. Methoden des Ereignisses

- `pulic Object` **`getItem`**`()`
  gibt das Element zurück, das vom Ereignis betroffen ist
- `public ItemSelectable` **`getItemSelectable`**`()`
  liefert den Auslöser des Ereignisses
- `public int` **`getStateChange`**`()`
  gibt den gewählten Zustand (Ein/Aus) zurück; im ActionEvent als SELECTED und DESELECTED definiert.

2.3.8. KeyEvent

Das KeyEvent wird ausgelöst, wenn eine Taste bzw. Tastenkombination über einer Komponente gedrückt, losgelassen oder kurz angeschlagen (drücken und gleich wieder loslassen) wurde.

2.3.8.1. Listener

`KeyListener`

2.3.8.2. Methoden des Listeners

- `public void` **`keyPressed`**`(KeyEvent e)`
  wird aufgerufen, sobald, eine Taste gedrückt wird
- `public void` **`keyReleased`**`(KeyEvent e)`
  wird aufgerufen, sobald eine Taste losgelassen wird
- `public void` **`keyTyped`**`(KeyEvent e)`
  wird aufgerufen, sobald eine Taste gedrückt wird, vorausgesetzt, man lässt sie auch wieder los.

2.3.8.3. Methoden des Ereignisses

- `public char` **`getKeyChar`**`()`
  gibt die Taste als `char`-Zeichen zurück, die das Ereignis ausgelöst hat
- `public int` **`getKeyCode`**`()`
  gibt den Tastaturcode der Taste zurück, die das Ereignis ausgelöst hat. Die Klasse

KeyEvent definiert alle Codes in Konstanten. Zum Beispiel entspricht VK_A der Taste „A“. Die vollständige Liste finden Sie in der API Beschreibung (JDK Doc).

- `public String` **`getKeyModifiers`**`(int modifiers)`
  gibt die Steuerungstasten (SHIFT, ALT, ...) als Zeichenkette zurück.
- `public void` **`isActionKey`**`()`
  gibt an, ob es sich bei der gedrückten Taste um eine gültige Action-Taste handelt.
- `public void` **`setKeyChar`**`(char keyChar)`
  setzt den `keyChar` Wert der gedrückten Taste
- `public void` **`setKeyCode`**`(int keyCode)`
  setzt den Tastaturcode der gedrückten Taste
- `public void` **`setModifiers`**`(int modifiers)`
  setzt die begleitenden Steuerungstasten für die gedrückte Taste

2.3.9. MouseEvent

Das Mausereignis gibt Auskunft, wie der Benutzer eine Komponente mit der Maus ansteuert. Das Mausereignis wird ausgelöst, sobald, der Mauszeiger den Bildbereich der Komponente betritt oder wieder verlässt, und wenn eine Maustaste gedrückt oder losgelassen wird. Man teilt dabei die Schnittstellen in einen `MouseListener` (für einfache Mausereignisse) und in einen `MouseMotionListener` (für Mausbewegungen), um eine übergrosse Flut von Ereignissen zu verhindern. Mausbewegungen sind ja für die Auswertung von einfachen Mausereignissen uninteressant.

2.3.9.1. Listener

`MouseListener`

2.3.9.2. Methoden des Listeners

- `public void` **`mouseClicked`**`(MouseEvent e)`
  wird aufgerufen, sobald die Maustaste über einer Komponente gedrückt und wieder losgelassen wurde
- `public void` **`mouseEntered`**`(MouseEvent e)`
  wird aufgerufen, sobald der Mauszeiger den Bereich der Komponente betritt
- `public void` **`mouseExited`**`(MouseEvent e)`
  wird aufgerufen, sobald der Mauszeiger den Bereich der Komponente verlässt
- `public void` **`mousePressed`**`(MouseEvent e)`
  wird aufgerufen, sobald die Maustaste über einer Komponente gedrückt wurde
- `public void` **`mouseReleased`**`(MouseEvenet e)`
  wird aufgerufen, sobald die Maustaste über einer Komponente losgelassen wurde

2.3.9.3. Methoden des Ereignisses

- `public int` **`getClickCount`**`()`
  gibt die Anzahl der erfolgten Mausklicks zurück
- `public Point` **`getPoint`**`()`
  gibt die Koordinaten des Mauszeigers zum Zeitpunkt des Ereigniseintritts zurück. Sie sind immer relativ zu den Koordinaten der Komponente, die das Ereignis ausgelöst hat.
- `public int` **`getX`**`()`
  gibt die X-Koordinate des Mauszeigers relativ zur Komponente zurück.
- `public int` **`getY`**`()`
  gibt die Y-Koordinate des Mauszeigers relativ zur Komponente zurück

- `public boolean` **`isPopupTrigger`**`()`
  gibt an, ob das Ereignis für das Aufklappen eines PopUp-Menüs gedacht ist
- `public void` **`translatePoint`**`(int x, int y)`
  verschiebt die Koordinaten des Mauszeigers um die Werte x und y.

2.3.10. PaintEvent

Hier handelt es sich um ein spezielles Ereignis. Es wird ausgelöst, wenn ein Element neu gezeichnet werden soll. Damit wird sichergestellt, dass das Neuzeichnen eines Elements mit den Ereignissen in der Ereignisschlange serialisiert wird. Es existiert hierfür keine Listener Schnittstelle. Es muss auch sichergestellt sein, dass die Methoden `paint`() und `update`() implementiert werden.

2.3.10.1. Methoden des Ereignisses

- `public Rectangle` **`getUpdateRect`**`()`
  gibt den Bereich als Rechteck zurück, der neu gezeichnet werden soll.
- `public Rectangle` **`setUpdateRect`**`(Rectangle updateRect)`
  setzt den Bereich als Rechteck, der neu gezeichnet werden soll.

2.3.11. TextEvent

Das Ereignis `TextEvent` wird bei Änderungen in einem Textfeld ausgelöst. Damit lassen sich Änderungen im Text feststellen. Das `TextEvent` enthält keine Methoden.

2.3.11.1. Listener

`TextListener`

2.3.11.2. Methoden des Listeners

- `public void` **`textValueChanged`**`(TextEvent e)`
  wird aufgerufen, sobald irgendwelche Änderungen im Textfeld stattfinden.

2.3.12. WindowEvent

Ein `WindowEvent` tritt ein, wenn ein Fenster geöffnet, geschlossen oder sonstige Fensteroperationen vom Benutzer durchgeführt werden.

2.3.12.1. Listener

`TextListener`

2.3.12.2. Methoden des Listeners

- `public void` **`windowActivated`**`(WindowEvent e)`
  wird aufgerufen, sobald das Fenster aktiviert wurde.
- `public void` **`windowClosed`**`(WindowEvent e)`
  wird aufgerufen, sobald das Fenster geschlossen wurde.
- `public void` **`windowClosing`**`(WindowEvent e)`
  wird aufgerufen, wenn sich das Fenster gerade schliessen will.
- `public void` **`windowDeactivated`**`(WindowEvent e)`
  wird aufgerufen, sobald das Fenster aktiviert wurde.
- `public void` **`windowDeiconified`**`(WindowEvent e)`
  wird aufgerufen, sobald das Fenster wieder hergestellt wurde.
- `public void` **`windowIconified`**`(WindowEvent e)`
  wird aufgerufen, sobald das Fenster minimiert wurde.
- `public void` **`windowOpened`**`(WindowEvent e)`
  wird aufgerufen, sobald das Fenster geöffnet wurde

2.3.12.3. Methoden des Ereignisses

- `public Window` **`getWindow`**`()`
  liefert das Fenster, welches für das Ereignis verantwortlich ist.

2.4. Swing Ereignisse

Neue Oberflächenkomponenten des Swing Pakets bringen auch neue Ereignisse mit sich. Sie befinden sich in javax.swing.event. Zum Teil sind sie sehr komponentenspezifisch. Sie werden in der folgenden Tabelle aufgelistet und danach detaillierter erklärt.

Schnittstelle	**Adapterklasse**	**Methoden**

`AncestorListener`	keine	`ancestorAdded`
		`ancestorMoved`
		`ancestorRemoved`

`CaretListener`	keine	`caretUpdate`
`ChangeListener`	keine	`stateChange`
`CellEditorListener`	keine	`editingCanceled` `editingStopped`

`DocumentListener`	keine	`changeUpdate`
		`insertUpdate`
		`removeUpdate`

`HyperlinkListener`	keine	`hyperlinkUpdate`

`InternalFrame-Listener`	`InternalFrameAdapter`	`internalFrameActivated`
		`internalFrameClosed`
		`internalFrameClosing`
		`internalFrameDeactivated`
		`internalFrameDeiconified`
		`internalFrameIconified`
		`internalFrameOpenend`

`ListDataListener`	keine	`contentsChanged`
		`intervalAdded`
		`intervalRemoved`

`ListSelectionListener`	keine	`valueChanged`

`MenuListener`	keine	`menuCanceled`
		`menuDeselected`
		`menuSelected`

`PopupMenuListener`	keine	`popupMenuCanceled`
		`popupMenuWillBecomeInvisible`
		`popupMenuWillBecomeVisible`

`TableColumnModel-Listener`	keine	`columnAdded`
		`columnMarginChanged`
		`columnMoved`
		`columnRemoved`
		`columnSelectionChanged`

`TableModelListener`	Keine	`tableChanged`
`TreeExpansion-Listener`	Keine	`treeCollapsed` `treeExpanded`

`TreeModelListener`	keine	`treeNodesChanged`
		`treeNodesInserted`
		`treeNodesRemoved`
		`treeStructureChanged`

`TreeSelection-Listener`	keine	`valueChanged`
`UndoableEditListener`	keine	`undoableEditHappened`

2.4.1. AncestorEvent

Das Ereignis `AncestorEvent` kommt vor, wenn Änderungen im Objekt vom Typ `JComponent` oder in einem seiner Vorfahren auftreten. Zu den Änderungen gehören Bewegungen der Komponente, wenn sie durch die `setVisible`() Methode oder durch Entfernen bzw. Hinzufügen der Komponente sichtbar bzw. unsichtbar wird.

2.4.1.1. Listener

`AncestorListener`

2.4.1.2. Methoden des Listeners

- `public void` **`ancestorAdded`**`(AncestorEvent e)`
  wird aufgerufen, sobald die Komponente sichtbar oder hinzugefügt wird
- `public void` **`ancestorRemoved`**`(AncestorEvent e)`
  wird aufgerufen, sobald die Komponente unsichtbar oder entfernt wird
- `public void` **`ancestorMoved`**`(AncestorEvent e)`
  wird aufgerufen, sobald die Komponente verschoben wird

2.4.1.3. Methoden des Ereignisses

- `public Container` **`getAncestor`**`()`
  liefert die geänderte Komponente
- `public Container` **`getAncestorParent`**`()`
  liefert die Oberklasse der geänderten Komponente
- `public Jcomponent` **`getComponent`**`()`
  liefert die Komponente, die mit dem Listener registriert wurde.

2.4.2. CaretEvent

Das Ereignis `CaretEvent` tritt auf, wenn sich der Cursor (=Caret) in einer Textkomponente bewegt oder die Markierung eines Textes sich ändert. Der `CaretListener` wird mit Komponenten der Instanz `JTextComponent` registriert.

2.4.2.1. Listener

`CaretListener`

2.4.2.2. Methoden des Listeners

- `public void` **`caretUpdate`**`(CaretEvent e)`
  wird aufgerufen, sobald sich die Position des Cursors ändert

2.4.2.3. Methoden des Ereignisses

- `public abstract int` **`getDot`**`()`
  liefert die Position des Cursors zurück
- `public abstract int` **`getMark`**`()`
  liefert die Endposition der Textmarkierung zurück. Falls keine Markierung vorliegt, ist der Wert mit dem von `getDot`() identisch.

2.4.3. ChangeEvent

Sobald sich der Zustand einer Komponente auf irgendeine Art ändert, wird das ChangeEvent ausgelöst. Es besitzt keine Methoden.

2.4.3.1. Listener

`ChangeListener`

2.4.3.2. Methoden des Listeners

- `public void stateChanged(ChangeEvent e)`
  wird aufgerufen, sobald sich der Zustand der Komponente ändert.

2.4.4. DocumentEvent

Das `DocumentEvent` tritt bei Änderungen jeglicher Art in Textdokumenten auf, wie sie die Textkomponenten einsetzen. Man registriert allerdings den `DocumentListener` mit der `Document` Klasse der Textkomponente anstatt mit ihr selbst. Zu beachten ist, dass es sich beim `DocumentEvent` ausnahmsweise um eine Schnittstelle und nicht um eine Klasse handelt.

2.4.4.1. Listener

`DocumentEvent`

2.4.4.2. Methoden des Listeners

- `public void changeUpdate(DocumentEvent e)`
  wird aufgerufen, sobald sich ein oder mehrere Textattribute im Dokument ändern
- `public void insertUpdate(DocumentEvent e)`
  wird aufgerufen, sobald Text in das Dokument eingefügt wird
- `public void removeUpdate(DocumentEvent e)`
  wird aufgerufen, sobald ein Text aus dem Dokument entfernt wird

2.4.4.3. Methoden des Ereignisses

- `public int getOffset()`
  liefert die Position, ab der die Änderung stattgefunden hat
- `public int getLength()`
  liefert die Länge der Änderung
- `public Document getDocument()`
  liefert das Dokument, in dem die Änderung stattgefunden hat.
- `public DocumentEvent.EventType getType()`
  liefert den Ereignistyp
- `public DocumentEvent.ElementChange getChange()`
  liefert Informationen über die Änderung. Sie enthalten die Elemente, welche eingefügt oder entfernt wurden, und deren Position.

2.4.5. HyperlinkEvent

Bei Änderungen oder Neuerungen eines Links zu einem HTML Dokument tritt das `HyperlinkEvent` ein.

2.4.5.1. Listener

`HyperlinkListener`

2.4.5.2. Methoden des Listeners

- `public void hyperlinkUpdate(HyperlinkEvent e)`
  wird aufgerufen, sobald der Link zu einem HTML Dokument geändert wurde.

2.4.5.3. Methoden des Ereignisses

- `public String getDescription()`
  liefert eine Beschreibung des Links
- `public HyperlinkEvent.EventType getEventType()`
  liefert eine Beschreibung des Links
- `public String getURL()`
  liefert die URL zu diesem Link

2.4.6. InternalFrameEvent

Das Verhältnis vom `InternalFrameEvent` zum `JInternalFrame` ist identisch mit dem von `WindowsEvent` zum `JFrame`. Der `InternalFrameListener` registriert die Fensterereignisse für interne Frames. Sie sind absolut identisch mit denen der normalen Frames. Das `InternalFrameEvent` selbst besitzt keine Methoden.

2.4.6.1. Listener

`InternalFrameListener`

2.4.6.2. Methoden des Listeners

- `public void` **`internalFrameOpened`**`(InternalFrameEvent e)`
  wird aufgerufen, sobald der interne Frame zum erstenmal angezeigt wird
- `public void` **`internalFrameClosing`**`(InternalFrameEvent e)`
  wird aufgerufen, sobald der interne Frame geschlossen wird. Dort kann neben der `setDefaultCloseOperation`() Methode das Schliessverhalten des Frames definiert werden.
- `public void` **`internalFrameClosed`**`(InternalFrameEvent e)`
  wird aufgerufen, sobald der interne Frame geschlossen und damit zerstört wurde.
- `public void` **`internalFrameIconified`**`(InternalFrameEvent e)`
  wird aufgerufen, sobald der interne Frame zum Icon minimiert wurde.
- `public void` **`internalFrameDeiconified`**`(InternalFrameEvent e)`
  wird aufgerufen, sobald der interne Frame wiederhergestellt wurde
- `public void` **`internalFrameActivated`**`(InternalFrameEvent e)`
  wird aufgerufen, sobald der interne Frame aktiviert wurde
- `public void` **`internalFrameDeactivated`**`(InternalFrameEvent e)`
  wird aufgerufen, sobald der interne Frame deaktiviert wurde

2.4.7. ListDataEvent

Ereignisse vom Typ `ListDataEvent` werden gesendet, wenn sich Daten in einer Liste ändern. Sei es durch das Hinzufügen oder Löschen von Listeneinträgen.

2.4.7.1. Listener

`ListDataListener`

2.4.7.2. Methoden des Listeners

- `public void` **`intervalAdded`**`(ListDataListener e)`
  wird aufgerufen, sobald ein neues Intervall in den Datenbereich der Liste eingefügt wurde.
- `public void` **`intervalRemoved`**`(ListDataListener e)`
  wird aufgerufen, sobald ein Intervall aus dem Datenbereich der Liste entfernt wurde.
- `public void` **`contentsChanged`**`(ListDataEvent e)`
  wird aufgerufen, sobald der Inhalt der Liste auf eine andere Art und Weise verändert wurde als in den letzten beiden Punkten

2.4.7.3. Methoden des Ereignisses

- `public int` **`getIndex0`**`()`
  liefert den unteren Index des Intervalls
- `public int` **`getIndex1`**`()`
  liefert den oberen Index des Intervalls
- `public int` **`getType`**`()`
  liefert den Ereignistyp : `CONTENTS_CHANGED, INTERVALL_ADDED` oder `INTERVALL_REMOVED`

2.4.8. ListSelectionEvent

Das `ListSelectionEvent` spezifiziert Änderungen in der Selektion von Listeneinträgen und wird gesendet, sobald diese Änderungen eintreten.

2.4.8.1. Listener

`ListSelectionListener`

2.4.8.2. Methoden des Listeners

- `public void` **`valueChanged`**`(ListSelectionEvent e)`
  wird aufgerufen, sobald sich der Wert der Auswahl ändert.

2.4.8.3. Methoden des Ereignisses

- `public int` **`getFirstIndex`**`()`
  liefert die oberste Zeile des Bereichs, dessen Auswahl sich geändert hat.
- `public int` **`getLastIndex`**`()`
  liefert die untere Zeile des Bereichs, dessen Auswahl sich geändert hat.
- `public boolean` **`getValueIsAdjusting`**`()`
  liefert true, wenn es sich hierbei um eins von mehreren Ereignissen handelt.
- `public String` **`toString`**`()`
  liefert das Ereignis in Form einer Zeichenkette zurück.

2.4.9. MenuEvent

Das `MenuEvent` tritt ein, wenn ein Menübefehl gewählt, aufgehoben oder abgebrochen wurde. Es besitzt keine Methoden.

2.4.9.1. Listener

`MenuListener`

2.4.9.2. Methoden des Listeners

- `public void` **`menuSelected`**`(MenuEvent e)`
  wird aufgerufen, sobald ein Menüfeld angewählt wurd
- `public void` **`menuDeselected`**`(MenuEvent e)`
  wird aufgerufen, sobald ein Menüfeld deselektiert wurde.
- `public void` **`menuCanceled`**`(MenuEvenet e)`
  wird aufgerufen, sobald die Wahl eines Menübefehls abgebrochen wurde.

2.4.10. PopupMenuEvent

Das `PopupMenuEvent` wird gesendet, wenn ein PopUp Menü aufgeklappt wird. Es enthält keine Methoden.

2.4.10.1. Listener

`PopupMenuListener`

2.4.10.2. Methoden des Listeners

- `public void` **`popupMenuWillBecomeVisible`**`(PopupMenuEvent e)`
  wird aufgerufen, bevor das PopUp Menü sichtbar wird.
- `public void` **`popupMenuWillBecomeInvisible`**`(PopupMenuEvent e)`
  wird aufgerufen, bevor das PopUp Menü unsichtbar wird
- `public void` **`popupMenuCanceled`**`(PopupMenuEvent e)`
  wird aufgerufen, sobald ein PopUp Menü abgebrochen wird.

2.4.11. TableColumnModelEvent

Das TableColumnModelEvent informiert darüber, ob sich der Datenbereich einer Tabellenspalte geändert hat, zum Beispiel durch Hinzufügen, Entfernen oder Bewegen einer Spalte.

2.4.11.1. Listener

`PopupMenuListener`

2.4.11.2. Methoden des Listeners

- `public void` **`columnAdded`**`(TableColumnModelEvent e)`
  wird aufgerufen, sobald eine Spalte dem Datenbereich hinzugefügt wurde.
- `public void` **`columnRemoved`**`(TableColumnModelEvent e)`
  wird aufgerufen, sobald eine Spalte im Datenbereicg entfernt wurde.
- `public void` **`columnMoved`**`(TableColumnModelEvent e)`
  wird aufgerufen, sobald sich die Position einer Spalte ändert.
- `public void` **`columnMarginChanged`**`(TableColumnModelEvent e)`
  wird aufgerufen, wenn eine Spalte durch Änderung des Rahmens bewegt wurde
- `public void` **`columnSelectionChanged`**`(TableColumnModelEvent e)`
  wird aufgerufen, sobald sich die Auswahl im Datenbereich geändert hat.

2.4.11.3. Methoden des Ereignisses

- `public int` **`getFromIndex`**`()`
  liefert den unteren Spaltenindex
- `public int` **`getToIndex`**`()`
  liefert den oberen Spaltenindex

22.1.12. TableModelEvent

Das Ereignis `TableModelEvent` informiert über Änderungen im Datenbereich einer Tabelle. Es beschreibt Änderungen und enthält die Spalten und Zeilen, die davon betroffen sind.

2.4.11.4. Listener

`TableModelListener`

2.4.11.5. Methoden des Listeners

- public void tableChanged(`TableModelEvent e)`
  wird aufgerufen, sobald sich der Datenbereich der Tabelle ändert und bringt im `TableModelEvent` Informationen über den geänderten Bereich mit.

2.4.11.6. Methoden des Ereignisses

- `public int` **`getFirstRow`**`()`
  liefert die erste Zeile der geänderten Daten. Dabei bezieht sich `HEADER_ROW` auf die Meta-Daten einer Spalte, wie zum Beispiel den Namen, den Typ oder die Sortierreihenfolge.
- `public int` **`getLastRow`**`()`
  liefert die erste Zeile der geänderten Daten
- `public int` **`getColumn`**`()`
  liefert die Spalte der geänderten Daten. Bei `ALL_COLUMNS` haben sich die Daten aller Spalten geändert.
- `public int` **`getType`**`()`
  liefert den Typus des Ereignisses oder auch die Art der Änderung: `INSERT, UPDATE` oder `DELETE`.

2.4.12. TreeExpansionEvent

Das `TreeExpansionEvent` identifiziert den Ast in einem Baum. Es wird ausgelöst, wenn dieser auf- oder zugeklappt wurde. Er kann gegebenenfalls mit `getSource`() angefordert werden.

2.4.12.1. Listener

`TreeExpansionEventListener`

2.4.12.2. Methoden des Listeners

- `public void` **`treeExpanded`**`(TreeExpansionEvent e)`
  wird aufgerufen, sobald der Baum aufgeklappt wird.
- public void **treeCollapsed**`(TreeExpansionEvent e)`
  wird aufgerufen, sobald der Baum zugeklappt wird.

2.4.12.3. Methoden des Ereignisses

- `public TreePath` **`getPath`**`()`
  liefert den Ast im Baum zurück, der auf- bzw. zugeklappt wurde.

2.4.13. TreeModelEvent

Das `TreeModelEvent` wird bei Änderungen im Datenbereich eines Baumes ausgelöst und enthält genauere Informationen über die Änderung.

2.4.13.1. Listener

`TreeModelListener`

2.4.13.2. Methoden des Listeners

- `public void` **`treeNodesChanged`**`(TreeModelEvent e)`
  wird aufgerufen, sobald sich ein oder mehrere Knoten auf irgendeine Art und Weise geändert haben.
- `public void` **`treeNodesInsertes`**`(TreeModelEvent e)`
  wird aufgerufen, sobald ein oder mehrere Knoten in einem Baum eingefügt wurden
- `public void` **`treeNodesRemoved`**`(TreeModelEvent e)`
  wird aufgerufen, sobald ein oder mehrere Knoten aus einem Baum entfernt wurden.
- `public void` **`treeStructuredChanged`**`(TreeModelEvent e)`
  wird aufgerufen, sobald sich die Struktur eines Astes verändert hat.

2.4.13.3. Methoden des Ereignisses

- `public TreePath` **`getTreePath`**`()`
  liefert den Ast im Baum zurück, der von der Änderung betroffen wurde.
- `public Object[ ]` **`getPath`**`()`
  liefert die Baumknoten als Objektliste zurück, die von der Änderung betroffen wurde.
- `public Object[ ]` **`getChildren`**
  liefert die Elemente des geänderten Knotens als Objektliste zurück
- `public int[ ]` **`getChildIndices`**`()`
  liefert die Indizes der gelöschten Elemente eines Knotens.
- `public String` **`toString`**`()`
  liefert die Beschreibung des Ereignisses als Zeichenkette zurück.

2.4.14. UndoableEditEvent

Ereignisse vom Typ `UndoableEditEvent` treten immer bei Operationen bei Komponenten ein, die auch wieder rückgängig gemacht werden können. Dies betrifft bis jetzt nur Textkomponenten. Solche Operationen umfassen das Einfügen und Löschen von Zeichen oder das Ändern der Textattribute.

2.4.14.1. Listener

`UndoableEditListener`

2.4.14.2. Methoden des Listeners

- `public void` **`undoableEditHappened`**`(UndoableEditEvent e)`
  wird aufgerufen, sobald eine der Operationen durchgeführt wurde, die auch rückgängig gemacht werden kann.

2.4.14.3. Methoden des Ereignisses

- `public UndoableEdit` **`getEdith`**`()`
  liefert die Operation zurück, die rückgängig gemacht werden kann.

2.5. Aufgaben

2.5.1. AWT Ereignisse

Schnittstelle	**Adapterklasse**	**Methoden**
ActionListener	Keine	actionPerformed
AdjustableListener	Keine	adjustableValueChanged
ComponentListener	ComponentAdapter	componentHidden componentMoved componentResized componentShown
ContainerListener	ContainerAdapter	componentAdded componentRemoved
FocusListener	FocusAdapter	focusGained focusLost
ItemListener	Keine	itemStateChanged
KeyListener	KeyAdapter	keyPressed keyReleased keyTyped
MouseListener	MouseAdapter	mouseClicked mouseEntered mouseExited mousePressed mouseReleased
MouseMotionListener	MouseMotionAdapter	mouseDragged mouseMoved
TextListener	Keine	textValueChanged
WindowListener	WindowAdapter	windowActivated windowClosed windowClosing windowDeactivated windowDeiconified windowOpened

Aufgabe:
Schreiben Sie ein Programm, welches die Position der Maus (in Pixel) als Text anzeigt.

Aufgabe:
Testen Sie den KeyListener, indem Sie ein Programm schreiben, welches in einem Fenster die Aktivierung der einzelnen Tasten anzeigt.

Aufgabe:
Schreiben Sie ein Programm, welches beim Schliessen des Fensters ein neues Dialog Fenster öffnet und einen Text, zum Beispiel“...das ist aber nicht nett!“ anzeigt.

Aufgabe:
Erläutern Sie am Beispiel des MouseEvents den Unterschied zwischen Methoden des Listeners und Methoden des Events.

In diesem Kapitel

- *LayoutManager*
- *LayoutManager2*
- *FlowLayout*
- *BorderLayout*
- *GridLayout*
- *GridBagLayout*
- *CardLayout*
- *BoxLayout*
- *ScrollPaneLayout*
- *AbsolutePositionierung*

3. *Layout Management*

Beim Layout Management geht es um die Anordnung verschiedener Komponenten wie zum Beispiel Buttons, Listboxen und so weiter – kurz : aller Swing Komponenten. Um das Layout dieser Komponenten so flexibel wie möglich zu gestalten, stellt Swing eine ganze Reihe verschiedener Layout Management Architekturen vor. Je nach Anordnung der Komponenten auf den diversen Containern werden verschiedene Layout Manager verwendet. Jede Klasse, die sich `java.awt.Container` abgeleitet ist, kann somit ein eigenes Layout besitzen. Bei der Anordnung spielen vor allem Grösse und Position eine wichtige Rolle. Durch diese Parameter sollte dem Benutzer eine Relation zwischen zusammengehörigen Komponenten vermittelt werden. Dank eines Layout Managements braucht sich der Programmierer nur noch teilweise um die Plazierung der Komponenten zu kümmern. Er muss somit keine absolute Positionen der Komponenten in einem Container berechnen und fest codieren, sondern fügt die Komponenten einfach dem Container hinzu. Das Management des Containers kümmert sich dann um die richtige Positionierung. Diese Art, ein Layout zu erstellen, ist wesentlich effektiver und leichter an neue Anforderungen anzupassen, als es mit der absoluten Positionierung je möglich wäre.

Jeder Container in Swing verfügt über sein eigenes Layout. Dies sind bei Applets und Panels das `FlowLayout` und bei Fenstern wie `JFrame` und `JDialog` das `BorderLayout`. Um den Default Layout Manager zu benutzen, braucht man nichts weiter zu utun, da dies durch den Aufruf des Konstruktors bereits erledigt wird. Um einen anderen Layout Manager zu verwenden, gibt es die Methode `setLayout`(). Diese Methode setzt dann ein gewünschtes Layout.

Die Methoden `add(), remove(), removeall(), layout(), preferredSize(), minimumSize`() und `maximumSize`() haben Einfluss auf den Layout Manager. Die Methoden `add(), remove(), removeAll`() dienen dem Hinzufügen und Entfernen von Komponenten des Containers. Die `layout`() Methode wird normalerweise durch die `paint`() Methode aufgerufen und dient zum Darstellen aller Komponenten des Containers. Die Methoden `preferredSize(), minimumSize`() und `maximumSize`() dienen dem Container als Angabe zur Darstellung der Komponenten. Sie könne allerdings nur als Vorschlag angesehen werden, da sie je nach Layout Manager nicht beachtet werden. So wird die Methode `preferredSize`() im `FlowLayout` beachtet, im `GridLayout` jedoch nicht.

Da einem Container wiederum eine Komponente zugewiesen werden kann, welche ein eigenes Layout besitzt, können beliebig komplexe Layouts erstellt werden.

Folgende Layout Manager sind in Java für die Erstellung von grafischen Benutzeroberflächen bereits implementiert:

- `BorderLayout`
- `CardLayout`
- `FlowLayout`
- `GridLayout`
- `GridBagLayout`
- `BoxLayout`
- `OverlayLayout`
- `ScrollPaneLayout`
- `ViewportLayout`

Einige Layout Manager können als Schnittstellen von `LayoutManager` oder `LayoutManager2` erstellt werden.

3.1. LayoutManager

Diese Schnittstelle wird für jene Layouts verwendet, in denen Position und Grösse vom Layout Manager und nicht durch `setPreferredSize`() bestimmt werden.

- `void` **`addLayoutComponent`**`(String name, Component comp)`
  fügt eine Komponente mit einem bestimmten Namen in einen Container ein
- `void` **`removeLayoutComponent`**`(Component comp)`
  löscht eine bestimmte Komponente aus einem Container
- `Dimension` **`preferredLayoutSize`**`(Container parent)`
  berechnet die bevorzugte Grösse für die angegebene Komponente
- `Dimension` **`minimumLayoutSize`**`(Container parent)`
  berechnet die minimale Grösse der angegebenen Komponente
- `void` **`layoutContainer`**`(Container parent)`
  erstellt das Layout eines Containers

`FlowLayout, GridLayout, ScrollPaneLayout` und `ViewportLayout` implementieren dieses Interface.

3.2. LayoutManager2

Diese Schnittstelle wird für solche Layouts verwendet, in denen die Position und Grösse eine Rolle spielen. Folgende Methoden müssen dabei implementiert werden:

- `void` **`addLayoutComponent`**`(Component c, Object constraints)`
  fügt unter Angabe eines Bedienungsobjekts einem Container eine Komponente hinzu.
- `public Dimension` **`maximumLayoutSize`**`(Container target)`
  gibt die maximale Grösse eines Containers hinsichtlich der eingefügten komponenten zurück
- `public float` **`getLayoutAlignmentX`**`(Container target)`
  gibt die vertikale Ausrichtung der Komponenten im Container an.
- `public float` **`getLayoutAlignmentY`**`(Container target)`
  gibt die horizontale Ausrichtung der Komponenten im Container an.

- `public void` **`invalidateLayout`**`(Container target)`
  gespeicherte Informationen über das Layout löschen.

BorderLayout, CardLayout, GridBagLayout, BoxLayout und **OverlayLayout** implementieren das Interface **LayoutManager2**.

3.3. FlowLayout

Bei diesem Layouttypus handelt es sich wohl um den einfachsten aller Layout Manager, die in Swing Verwendung finden. Die Ausrichtung der Komponenten erfolgt hierbei zeilenweise. Ist eine Zeile voll, wird automatisch eine neue Zeile begonnen. Die Komponenten werden mit ihrer bevorzugter Grösse mit `setPreferredSize`() eingetragen, die Komponenten von links nach rechts (default) zentriert (`FlowLayout.CENTER`) oder rechtsbündig (`FlowLayout.Right`) angeordnet. Der Abstand zwischen den Komponenten kann mit den Methoden `setHGap`() in horizontaler und `setVGap`() in vertikaler Richtung verändert werden.

Dieser Layouttypus eignet sich besonders für die Anordnung von Buttons oder aber Eingabefeldern, die alle linksbündig sein sollen.

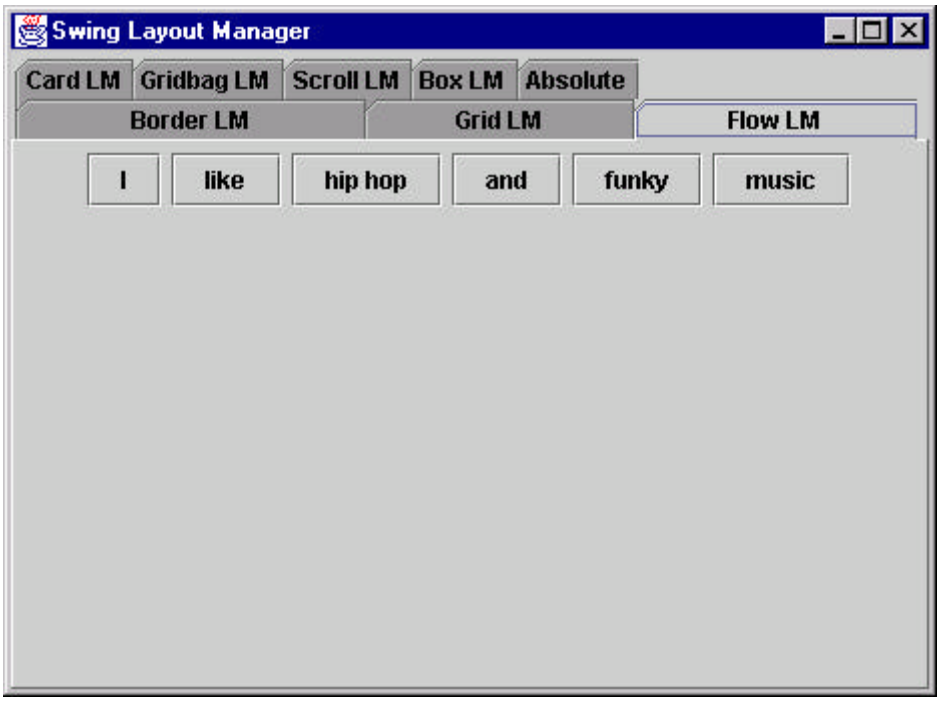


Abbildung 3-1

Und hier der Programmteil für den `FlowLayout`:

Listing 3-1

```
import java.awt.*;
import javax.swing.*;

public class Flow extends JPanel {
  public Flow() {
    setLayout(new FlowLayout());
    add (new JButton("I"));
    add (new JButton("like"));
    add (new JButton("hip hop"));
    add (new JButton("and"));
    add (new JButton("funky"));
    add (new JButton("music"));
  }
}
```

Das Rahmenprogramm, mit dem das obige Programm gestartet wird, sieht ebenfalls sehr einfach aus:

Listing 3-2

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

public class LayoutManagers extends JApplet
{
  public void init()
  {
    // getContentPane().add(new Flow());
    getContentPane().add(new TabbedPanel());
  }

  // Erstellt ein Panel mit Rahmen
  public static void main(String[] args)
  {
    Frame frame = new Frame("Swing Layout Manager");

    WindowListener l = new WindowAdapter() {
    public void windowClosing(WindowEvent e) {System.exit(0);}
    };
    frame.addWindowListener(l);

    LayoutManagers applet = new LayoutManagers();
    frame.add("Center", applet);
    applet.init();
    frame.pack();
    frame.show();
  }
}
```

Das Programm besteht im wesentlichen aus einem Frame, in das ein `TabbedPanel` einfügt wird.

3.4. BorderLayout

Der `BorderLayout` Manager ist der Standard Layout Manager für Frames und Dialoge. Dieser Layouttyp gliedert sich dabei in 5 Zonen: Zentrum, Norden, Süden, Osten und Westen. Dem Zentrum wird immer soviel Platz wie möglich eingeräumt. Die anderen vier Areale bekommen nur soviel Platz, wie sie benötigen. Um einen Abstand zwischen den einzelnen Komponenten zu gewährleisten, gibt es Methoden : `setHgap`() und `setVgap`().

Wird eine Komponente in das `BorderLayout` eingefügt, sollte die Methode `add`() benutzt werden, welche zwei Argumente braucht. Andernfalls ist eine korrekte Darstellung der hinzugefügten Komponenten nicht gewährleistet.

Im Beispiel sieht man die Funktionsweise des `BorderLayout` Managers.

Listing 3-3

```
import java.awt.*;
import javax.swing.*;

public class Border extends JPanel {
  public Border() {
    setLayout(new BorderLayout());
    add (new JButton("BorderLayout.CENTER"),BorderLayout.CENTER);
    add (new JButton("BorderLayout.EAST"),BorderLayout.EAST);
    add (new JButton("BorderLayout.SOUTH"),BorderLayout.SOUTH);
    add (new JButton("BorderLayout.NORTH"),BorderLayout.NORTH);
    add (new JButton("BorderLayout.WEST"),BorderLayout.WEST);
  }
}
```

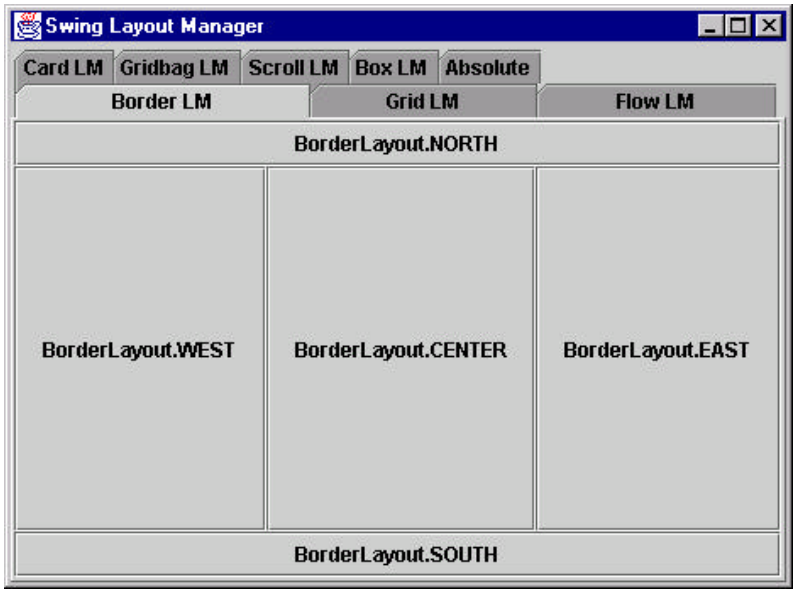


Abbildung 3-2

3.5. GidLayout

Das `GridLayout` benutzt zur Darstellung der hinzugefügten Komponenten eine Matrix. Die Anzahl der Elemente wird bei der Initialisierung vorgegeben, kann aber durch die Methoden `setColumns`() und `setRows`() noch verändert werden. In dieser Matrix erhalten alle Komponenten die gleiche Höhe und Breite. Die Grösse der Komponenten ermittelt sich dabei aus der Grösse des Panels durch die Anzahl Elemente in der Matrix. Wie beim `BorderLayout` kann auch hier mit `setHgap`() und `setVgap`() der horizontale und vertikale Abstand zwischen den Komponenten beliebig verändert werden.

`GridLayout` ermöglicht es also, mehrere Komponenten in einer Matrix darzustellen, wobei alle die gleiche Grösse besitzen.

Listing 3-4

```
import java.awt.*;
import javax.swing.*;

public class Grid extends JPanel {
  public Grid() {
    setLayout(new GridLayout(2,1));
    add (new JButton("GridLayout"));

    add (new JButton("is "));
    add (new JButton("nice"));
    add (new JButton("to"));
    add (new JButton("handle"));
  }
}
```

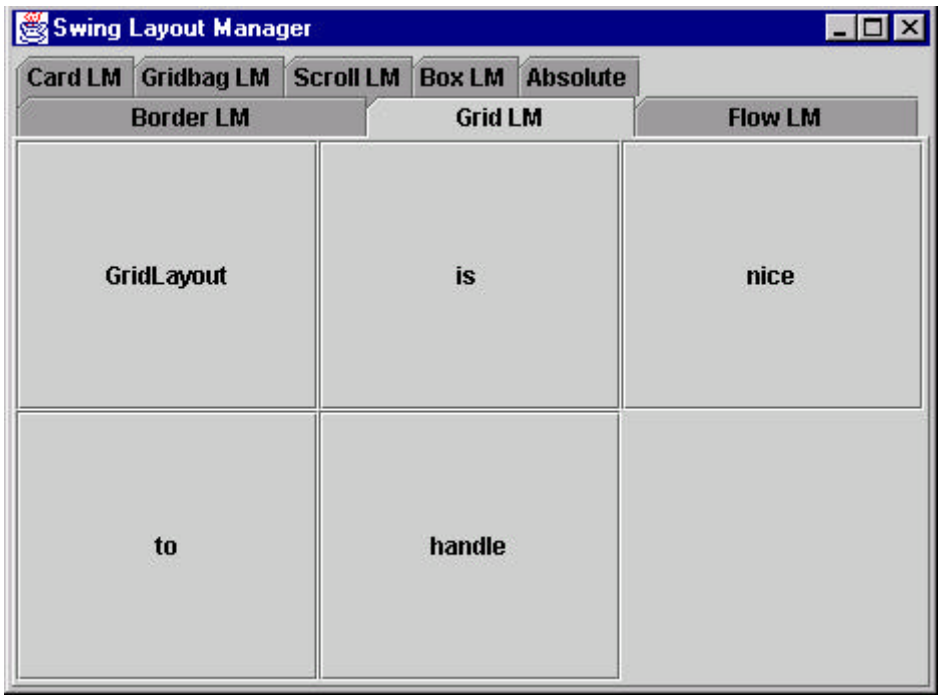


Abbildung 3-3

3.6. GridBagLayout

Beim `GridBagLayout` handelt es sich um den flexibelsten `LayoutManager`. Er ermöglicht es, die Komponenten beliebig auf einem Gitter auszurichten. Bei diesem Layouttyp gibt es nicht nur die Möglichkeit, Komponenten in Spalten und reihen einzutragen, sondern auch, deren Grösse zu bestimmen. So können Komponenten über mehrere Spalten und Reihen gehen und ihre Grösse beim Ändern des Hauptfensters anpassen. Um eine solch hohe Flexibilität zu erlangen, wird eine neue Klasse eingeführt. `GridBagConstraints` verwaltet dabei die jeweiligen Angaben zum `GridBagLayout` und setzt die Attribute mit `setConstraints`() in der `GridBagLayout` Klasse.

`GridBagConstraints` besitzt folgende Attribute:

- `gridx, gridy`
  zeigt die Position der Komponente in einer Komponente an. Die Spalte oben links hat die Adresse `gridx`=0, die oberste Reihe die Adresse `gridy`=0. Mit `GridBagConstraints.RELATIVE` (default) kann angegeben werden, ob die nächste Komponente gleich neben der vorherigen mit `gridx` oder darunter mit `gridy` hinzugefügt werden soll.
- `gridwidth, gridheight`
  gibt die Anzahl der Spalten (für `gridwidth`) und die Anzahl der Reihen (für `gridheight`) an, über die sich eine hinzugefügte Komponente erstreckt. Der Default Wert ist hier 1. Jede hinzugefügte Komponente füllt also genau eine Zeile aus. Um dem `LayoutManager` mitzuteilen, dass dies die letzte Komponente in einer Reihe (für `gridwidth`) oder Spalte (für `gridheight`) ist, wird die Konstante `GridBagConstraints.REMAINDER` verwendet.
- `fill`
  ist die hinzugefügte Komponente kleiner als der zur Verfügung stehende Platz, so kann mit den Konstanten `GridBagConstraints.NONE` eine Anpassung in vertikaler und horizontaler Richtung verhindert werden. `GridBagConstraints.HORIZONTAL` passt die Grösse der hinzugefügten Komponente nur in horizontaler Richtung an, `GridBagConstraints.VERTICAL` in vertikaler Richtung und `GridBagConstraints.BOTH` in beiden Richtungen.
- `ipadx, ipady`
  zeigt den Abstand der hinzugefügten Komponente zum Zellenrand an. Dieser Wert wird zur Minimalgrösse der Komponente addiert. Der Default-Wert ist hierbei 0. Bei der Angabe sollte nicht vergessen werden, dass der Abstand auf beiden Seiten zur minimalen Grösse addiert wird. Dies gilt für beide Richtungen, vertikal und horizontal.
- `Insets`
  gibt den äusseren Leerraum zwischen hinzugefügter Komponente und Zellenrand an. Der angegebene Wert ist dabei ein `Inset` Objekt. Per Default besitzt eine Komponente keinen äusseren Leerraum.
- `anchor`
  benötigt eine Komponente weniger Platz in einer Gitterzelle als notwendig, kann mit Hilfe der konstante `anchor` eine Ausrichtung festgelegt werden. Mögliche Werte sind hierbei:

  ```
  GridBagConstraints.CENTER (default)
  ```

```
GridBagConstraints.NORTH
GridBagConstraints.NORTHEAST
GridBagConstraints.EAST
GridBagConstraints.SOUTHEAST
GridBagConstraints.SOUTH
GridBagConstraints.SOUTHWEST
GridBagConstraints.WEST  und
GridBagConstraints.NORTHWEST
```

- `weightx, weighty`
  Angaben bezüglich `weightx` und `weighty` besitzen grossen Einfluss hinsichtlich des `resize` Verhaltens der Komponenten. Mit Hilfe dieser Gewichtungsmöglichkeiten kann der Platz zwischen den einzelnen Reihen und Spalten festgelegt werden. Die geringste Gewichtung ist dabei 0.0 (default); die höchste Gewichtung 1.0. Der `GridBagLayout` Manager fügt eine eigene Reihe zwischen den Komponenten hinzu. Um ein Gefühl für solche Gewichtungen zu bekommen, sollte man den unten angegebenen Sourcecode entsprechend abändern und eigene Gewichtungen einbauen.

Listing 3-5

```
import java.awt.*;
import javax.swing.*;

public class GridBag extends JPanel {
  private GridBagLayout   gridBagLayout;
  public GridBag() {
    gridBagLayout = new GridBagLayout();
    GridBagConstraints constraints = new GridBagConstraints();
    setLayout(gridBagLayout);
    constraints.insets = new Insets(4,4,4,4);
    constraints.ipadx =3;
    constraints.ipady =3;
    constraints.fill = GridBagConstraints.NONE;
    constraints.weightx = 0.5;
    constraints.weighty = 0.5;
    constraints.anchor = GridBagConstraints.CENTER;
    constraints.gridx = 0;
    constraints.gridy = 0;
    constraints.gridwidth = 1;
    constraints.gridheight = 1;
    buildButton("Button 1",gridBagLayout,constraints);
    constraints.gridx = 1;
    constraints.gridy = 0;
    constraints.gridwidth = 1;
    constraints.gridheight = 2;
    buildButton("Button 2",gridBagLayout,constraints);
    constraints.gridx = 2;
    constraints.gridy = 0;
    constraints.gridwidth = 2;
    constraints.gridheight = 1;
    buildButton("Button 3",gridBagLayout,constraints);
    constraints.gridx = 3;
    constraints.gridy = 1;
    constraints.gridwidth = 1;
    constraints.gridheight = 2;
    constraints.fill = GridBagConstraints.VERTICAL;
    buildButton("Button 4",gridBagLayout,constraints);
    constraints.gridx = 0;
    constraints.gridy = 1;
```

```
    constraints.gridwidth = 1;
    constraints.gridheight = 1;
    constraints.fill = GridBagConstraints.HORIZONTAL;
    buildButton("Button 5",gridBagLayout,constraints);
  }
  protected void buildButton(String name,
                             GridBagLayout gridbag,
                             GridBagConstraints c) {
    Button button = new Button(name);
    gridBagLayout.setConstraints(button, c);
    add(button);
 }
}
```

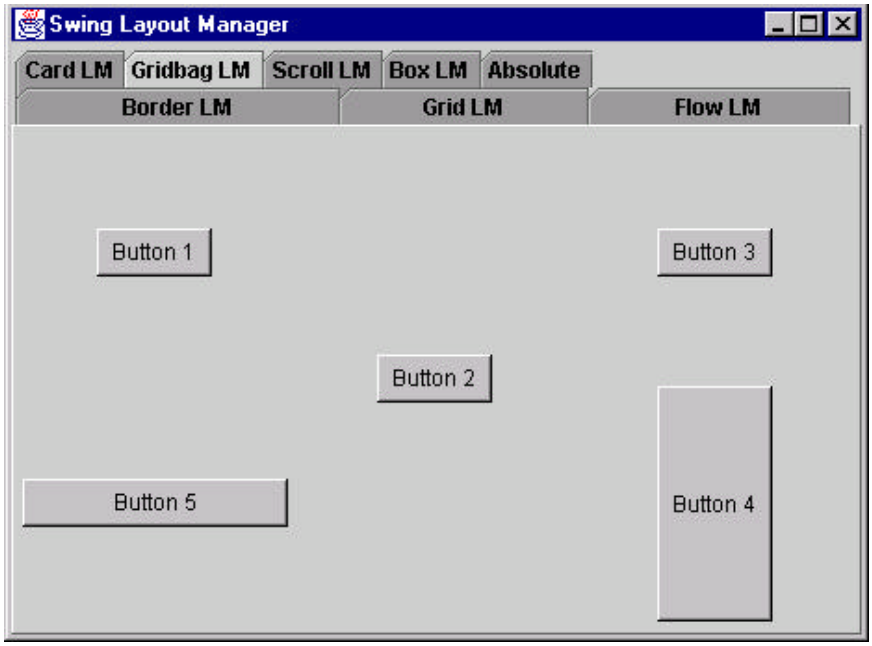


Abbildung 3-4

Durch Ändern einer Zeile in der Initialisierung des `GridBagLayoutConstraints` bekommt das Layout ein völlig anderes Aussehen.

Ersetzen wir im Listing oben die Constraints:
`constraints.fill = GridBagConstraints.NONE;`
durch
`constraints.fill = GridBagConstraints.BOTH;`

liefert folgendes verändertes Bild:

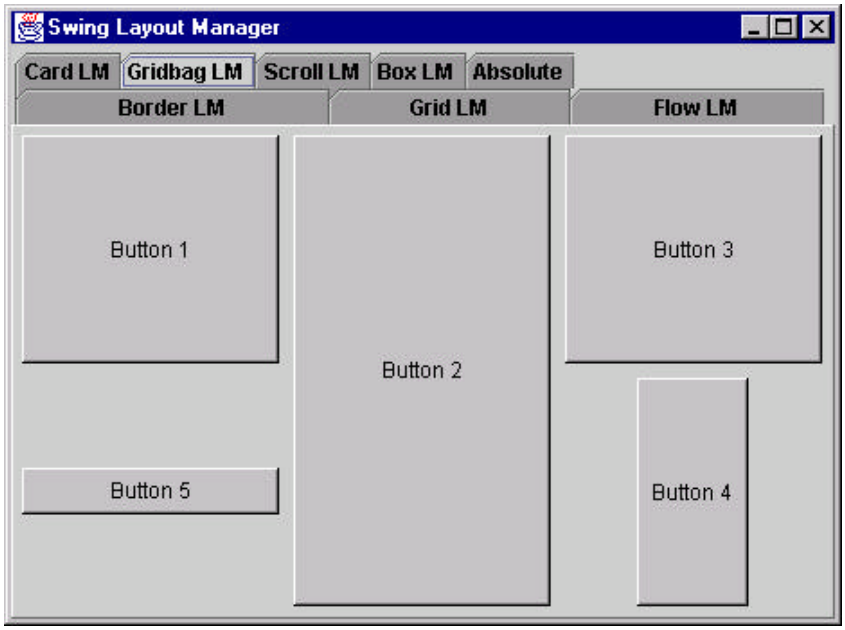


Abbildung 3-5

3.7. CardLayout

Dieser Layouttyp wird benutzt , wenn eine oder mehrere Komponenten den gleichen Platz in einem Layout eines `JPanels` verwenden. Wie in einem Kartenspiel ist jeweils nur die oberste Karte sichtbar. Es bestehen dabei folgende Möglichkeiten, die hinzugefügten Komponenten anzusehen:

- Mit den Methoden `first`() und `last`() kann zur ersten und letzten geschaltet werden. Entscheidend dabei ist die Reihenfolge, wann die Komponenten dem `JPanel` hinzugefügt worden sind.
- Mit den Methoden `next`() und `prev`() kann durch einen Stapel geschaltet werden
- Mit der Möglichkeit, einen Namen zu vergeben, kann `JPanel` durch einen Namen wieder gefunden werden. Dies macht vor allem bei mehreren Karten Sinn, um sie leichte zuordnen zu können.

Mit der Methode `show`() kann dann das gewünschte `JPanel` am Bildschirm angezeigt werden.

Das `CardLayout` ist bereits in einer Swing Komponente fix implementiert. Die `JTappedPane` besitzt bereits die Möglichkeit, mehrere Panels zu einem `JTappedPane` hinzuzufügen.

Die Schnittstelle zum Wechseln der Karten muss allerdings der Programmierer erst implementieren. Vergleichen Sie dazu das Beispiel im Programmcode:

Listing 3-6

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

public class Card extends JPanel {
```

```
    JPanel cards;
    final static String BUTTONPANEL = "Panel mit Buttons";
    final static String TEXTPANEL = "Panel mit TextField";
    public Card() {
      setLayout(new BorderLayout());
      setFont(new Font("Helvetica", Font.PLAIN, 14));

      //Put the Choice in a Panel to get a nicer look.
      JPanel cp = new JPanel();
      JComboBox cmb = new JComboBox();
      cmb.addItem(BUTTONPANEL);
      cmb.addItem(TEXTPANEL);
      cmb.addActionListener(new MyActionListener());
      cp.add(cmb);
      add("North", cp);
      cards = new JPanel();
      cards.setLayout(new CardLayout());

      JPanel p1 = new JPanel();
      p1.add(new JButton("I    "));
      p1.add(new JButton("like "));
      p1.add(new JButton("Swing"));

      JPanel p2 = new JPanel();
      p2.add(new JTextField("Swing text field", 20));
      cards.add(BUTTONPANEL, p1);
      cards.add(TEXTPANEL, p2);
      add("Center", cards);
    }

  class MyActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JComboBox cb = (JComboBox)e.getSource();
      String newSelection = (String)cb.getSelectedItem();
      ((CardLayout)cards.getLayout()).show(cards,(String)newSelection);
    }
  } // end inner class
}
```

Als erstes sehen Sie einige Buttons, dann folgt ein Textfeld Beispiel.

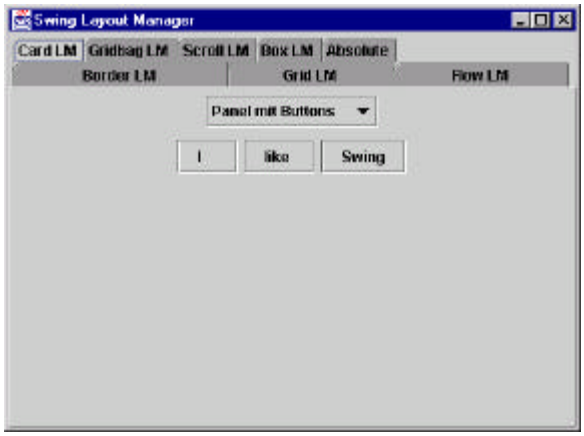


Abbildung 3-6

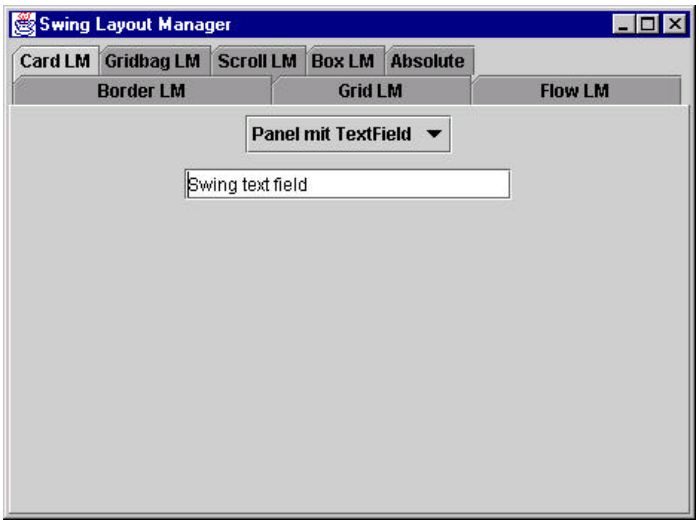


Abbildung 3-7

3.8. BoxLayout

Der `BoxLayout` Manager erlaubt es, mehrere Komponenten entweder vertikal oder horizontal aneinanderzureihen. Die Komponenten werden dabei nicht in die nächste Zeile oder Spalte umgebrochen, wenn die Grösse verändert wird (wie dies beim `FlowLayout` der Fall ist).

Werden mehrere Panels mit diesem Layouttyp nebeneinander geschaltet und dabei in ihrem Layout (X_AXIS oder Y_AXIS) variiert, so hat der Programmierer die gleichen Möglichkeiten, wie er sie vom `GridBagLayout` her kennt, ohne allerdings die Komplexität dieses Layouttyps zu verwenden.

Swing bietet bereits einen leichtgewichtigen Container, welcher das `BoxLayout` beinhaltet. Dieser Container stellt bereits Möglichkeiten zum leichten Umgang mit dem `BoxLayout` zur Verfügung. Es handelt sich dabei um die Klasse Box. Zu einer Box können weitere Boxen und andere Komponenten hinzugefügt werden. Der `BoxLayout` Manager ordnet dabei die hinzugefügten Komponenten von links nach rechts oder von oben nach unten an. Wird ein `BoxLayout` erzeugt, wird angegeben, ob es sich um ein X_AXIS (links nach rechts) oder um eine Y_AXIS (von oben nach unten) Ausrichtung handelt.

Das `BoxLayout` versucht, allen Komponenten, die hinzugefügt wurden, die gleiche Höhe und Breite zu geben. Referenzwert ist dabei die breiteste / höchste Komponente im Container. Falls es nicht möglich ist, die Komponenten so weit zu vergrössern, werden diese entsprechend den Einstellungen aufgrund von `getAlignmentX`() oder `getAlignmentY`() ausgerichtet. Der vorgegebene Wert dafür ist 0.5, was bedeutet, dass die vertikale Ausrichtung der Komponente die gleiche Y Koordinate wie die vertikal ausgerichteten Komponenten mit 0,5 Y Koordinate besitzen muss. Eine neue Box erhält man mit `createHorizontalBox`() oder `createVerticalBox`(). Die Klasse Box bietet dabei einige Methoden zur besseren Positionierung der Komponenten.

Die Methoden `createHorizontalGlue() / createVerticalGlue`() und `createGlue`() erstellen Zwischenräume unterschiedlicher Höhe oder Breite, um Komponenten zu trennen. Je nach Grösse der Box passen sich die darin befindlichen Komponenten dem Layout an.

Die Methoden `createHorizontalStrut`() und `createVerticalStrut`() erzeugen Zwischenräume fester Grösse. Auch wenn sich die Grösse der Box ändert, bleiben die Abstände zwischen den Komponenten gleich.

Diese erzeugten unsichtbare Räume können wie andere Komponenten auch der Box hinzugefügt werden.

Listing 3-7

```
import java.awt.*;
import javax.swing.*;

public class BoxPanel extends Box {

  static float ALLIGNMENT =0.0f;

  public BoxPanel() {
    super(BoxLayout.X_AXIS);
    Box b = new Box(BoxLayout.Y_AXIS);
    b.add(createVerticalGlue());
    b.add(createVerticalStrut(5));
    JLabel label = new JLabel("Vorname:");
    label.setAlignmentX(ALLIGNMENT);
    b.add( label);
    b.add(createVerticalStrut(5));
    label = new JLabel("Nachname:");
    label.setAlignmentX(ALLIGNMENT);
    b.add( label);
    b.add(createVerticalStrut(5));
    b.add(createVerticalGlue());
    Box b2 = new Box(BoxLayout.Y_AXIS);
    b2.add(createVerticalGlue());
    b2.add(createVerticalStrut(5));
    JTextField txtFirst = new JTextField("Bjarne");
    txtFirst.setAlignmentX(ALLIGNMENT);
    b2.add( txtFirst );
    b2.add(createVerticalStrut(5));
    JTextField txtLast = new JTextField("Stroustrup");
    txtLast.setAlignmentX(ALLIGNMENT);
    b2.add( txtLast );
    b2.add(createVerticalStrut(5));
    b2.add(createVerticalGlue());
    add( createHorizontalGlue() );
    add( b );
    add( b2 );
    add( createHorizontalGlue() );
  }
}
```

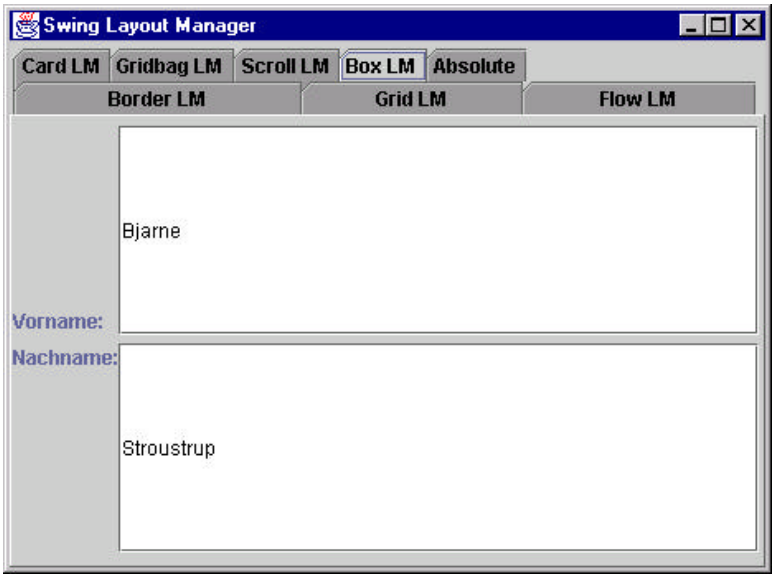


Abbildung 3-8

3.9. ScrollPaneLayout

Das ScrollPaneLayout selbst wird meist in Zusammenhang mit dem ScrollPane und deshalb kaum direkt verwendet, sondern im Zusammenhang mit Jviewport und JscrollPane.

Das nachfolgende Bild zeigt die Architektur des ScrollPaneLayouts.

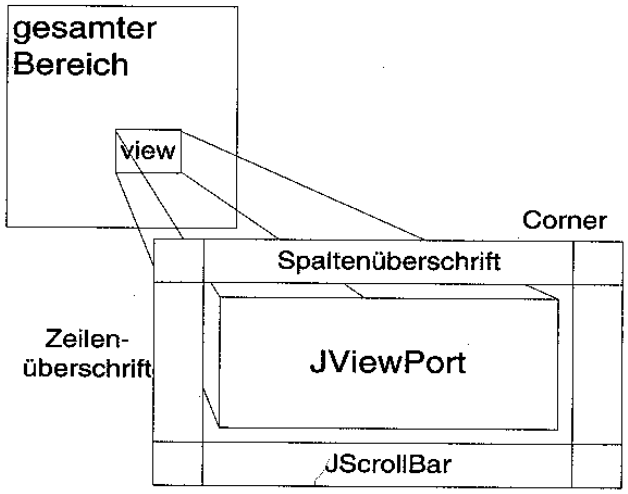


Abbildung 3-9 Funktionsweise des `JViewports`

Das `ScrollPaneLayout` definiert sich in neun verschiedenen Bereichen. Jeder Bereich wird von einem eigenen Objekt verwaltet und gezeichnet – das Zentrum (hier als `JViewPort` gekennzeichnet), die vier Ecken und die vier Seiten. Das Zentrum ist die einzige Komponente, die in allen `ScrollPanes` vorhanden ist. Alle vier Seiten sind optional. Werden zwei Seiten angezeigt, so erscheint automatisch die dazugehörige Ecke.

Das Zentrum zeigt die momentan sichtbare Fläche des gesamten Bereichs – die Sicht / View. Diese Sicht wird durch den `JViewPort` repräsentiert.

Die rechte und untere Seite beinhalten die `JScrollBar` Objekte. Diese Objekte sind optional und können durch entsprechende konstanten unterdrückt werden. Folgende Konstanten sind verfügbar:

- `ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED`
- `ScrollPaneConstants.VERTICAL_SCROLLBAR_NEVER`
- `ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS`
- `ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED`
- `ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER`
- `ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS`

Die linke und obere Seite enthalten eine Zeilen- beziehungsweise Spaltenüberschrift. Jede dieser Überschriften enthält ein `JViewPort` Objekt, damit es zusammen mit dem eigentlichen View bewegt werden kann. Das `JViewPort` Objekt für die Zeilenüberschrift bewegt sich dabei nur in vertikaler, das Objekt für die Spaltenübersicht nur in horizontaler Richtung.

Die Ecken zwischen den Überschriften und `JScrollBar` Objekten sind nur dann sichtbar, wenn diese benötigt werden. In diese Objekte kann jede beliebige Swing Komponente aufgenommen werden.

Listing 3-8

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class Scroll extends JPanel {

    private JLabel columnView;
    private JLabel rowView;

    private JScrollPane scrollPane;

    public Scroll() {
      // laden der Bilder

      ImageIcon lara        = new ImageIcon("lara.jpg");
      ImageIcon column      = new ImageIcon("column.gif");
      ImageIcon row           = new ImageIcon("row.gif");
      ImageIcon cornerImage   = new ImageIcon("corner.gif");


      JLabel picture = new JLabel(lara);

      scrollPane = new JScrollPane(picture,
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
        ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS
      );
      scrollPane.setPreferredSize(new Dimension(300, 250));
      scrollPane.setViewportBorder(

      BorderFactory.createLineBorder(Color.black));
```

```
        columnView = new JLabel(column);
        rowView = new JLabel(row);

        scrollPane.setColumnHeaderView(columnView);
        scrollPane.setRowHeaderView(rowView);

        JLabel cornerUL = new JLabel(cornerImage);
        JLabel cornerUR = new JLabel(cornerImage);
        JLabel cornerLL = new JLabel(cornerImage);

        scrollPane.setCorner(JScrollPane.UPPER_LEFT_CORNER, cornerUL);
        scrollPane.setCorner(JScrollPane.LOWER_LEFT_CORNER, cornerUR);
        scrollPane.setCorner(JScrollPane.UPPER_RIGHT_CORNER, cornerLL);

        setLayout(new BoxLayout(this, BoxLayout.X_AXIS));
        add(scrollPane);
        setBorder(BorderFactory.createEmptyBorder(10,10,10,10));
    }
}
```

Abbildung 3-10 `JScrollPane`

3.10. Absolute Positionierung

Es ist in Swing auch möglich, ohne Layout Manager zu arbeiten. Dies kann bei einfachen Layouts durchaus Sinn machen. Der Programmierer sollte dies allerdings tunlichst vermeiden oder auf ein Minimum beschränken, da er dadurch nur zusätzliche Probleme einhandeln kann. Er muss sich dann nämlich um Grössenänderungen, plattformabhängige Komponenten, verschiedene Schriftwarten und vieles mehr selber kümmern. Wenn die Änderung der Grösse, Wiederverwendbarkeit und Systemabhängigkeit keine Rolle spielen, macht diese Art der Positionierung Sinn.

Das folgende Beispiel zeigt die Möglichkeiten der absoluten Positionierung.

Listing 3-9

```
import java.awt.*;
import javax.swing.*;

public class Absolute extends JPanel {
  private JButton button1, button2, button3;

  public Absolute() {
    super();
    setLayout(null);

    setFont(new Font("Arial", Font.PLAIN, 12));

    button1 = new JButton("one");
    add(button1);

    button2 = new JButton("two");
    add(button2);

    button3 = new JButton("three");
    add(button3);
  }

  public void paint(Graphics g) {

    button1.reshape(20, 20 , 100, 30);
    button2.reshape(80, 100, 100, 30);
    button3.reshape(230, 50, 200, 60);

  }

}
```

Und hier der Snapshot:

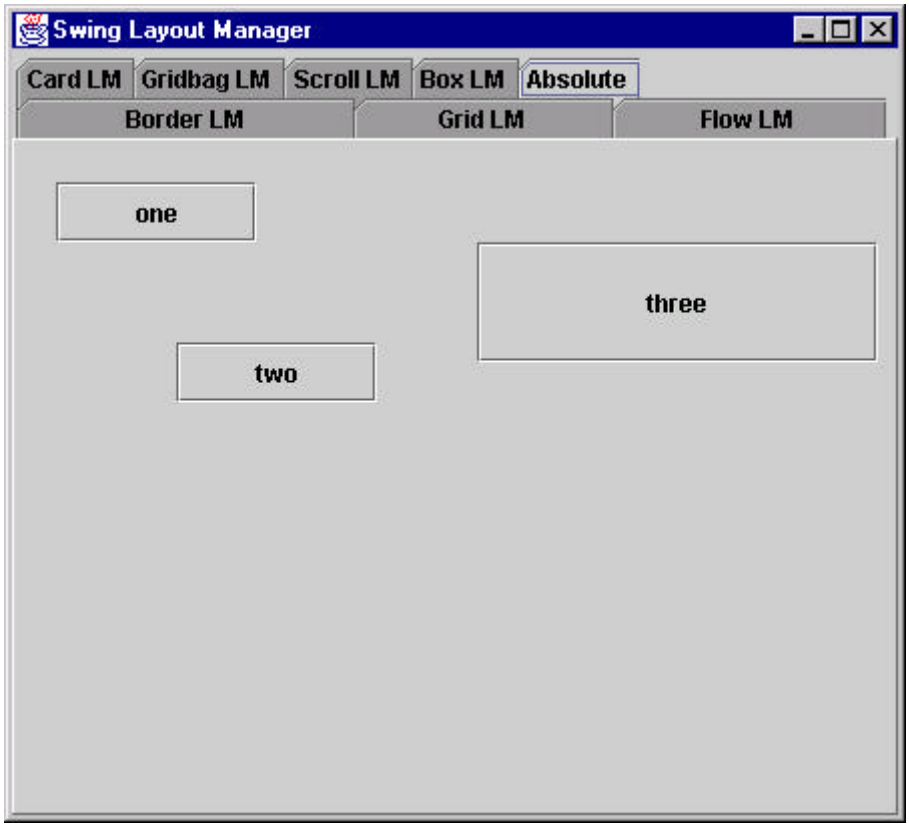


Abbildung 3-11 Absolute Positionierung

In diesem Kapitel
- *JComponent*
- *JPanel*
- *ImageIcon*
- *JLabel*
- *JToolTip*

4. Basiskomponenten

4.1. JComponent

Die Klasse `JComponent` ist die Basisklasse für alle sichtbaren Komponenten in Swing. Es gibt allerdings auch viele Swingklassen, die `JComponent` nicht als Basisklasse verwenden. Die Klasse `JComponent` enthält viele Methoden, die für den richtigen Umgang mit den Swing-Komponenten sehr wichtig sind. Aus der obigen Grafik ist ersichtlich, dass `JComponent` eine Unterklasse der AWT Komponenten ist und somit auch die Eigenschaften dieser AWT Klassen enthält.

Die Basisklasse für Swing-Komponenten enthält folgende Möglichkeiten:
- ein Pluggable Look & Feel, welches vom Entwickler oder zur Laufzeit vom Benutzer festgelegt werden kann.
- Komponenten, welche so konzipiert sind, dass sie als Kombination miteinander leicht zu benutzerspezifischen Komponenten werden können.
- Verarbeitung von Tastatureingaben, selbst mit verschachtelten Komponenten.
- Unterstützt die Verwendung von Abständen und Rändern, zur besseren Plazierung der Komponenten und somit zu einem besseren Design
- Festlegen von minimaler, bevorzugter und maximaler Grösse vom Komponenten. Diese Einstellungen werden dann von Layout Managern zur optimalen Darstellung verwendet.
- Anzeige eines Textes, wenn sich der Cursor über einer Swing Komponente befindet.
- Automatisches Scrollen innerhalb von Listen, Tabellen oder Trees, wenn der Benutzer sich mit der Maus im Drag-Modus befindet.

- einfache Darstellungsmöglichkeit von Dialogboxen mit Hilfe von statischen Methoden in der `JOptionPane` Klasse.
- Unterstützung bei der Fehlersuche im Grafikbereich durch die Möglichkeit, sich das Gezeichnete langsamer anzeigen zu lassen. Somit kann der Entwickler Fehler bei Grafikausgaben leichter finden.
- Unterstützung von mehreren Eingabemöglichkeiten neben Maus und Tastatur.
- Unterstützung der Internationalisierung und Lokalisierung (Nummerierung, Datumsformate, sowie Mehrsprachigkeit).

Die wohl am häufigsten benutzte Methode ist `setPreferredSize`(). Sie dient zum Setzen der Breite und Höhe einer Swing Komponente. Mit der Methode `getPreferredSize`() kann man sich die Dimension einer Komponente geben lassen und erhält somit die Höhe und Breite der Komponente. Bei dynamischen Layout wird eine Obergrenze für die maximal darzustellende Grösse einer Komponente mit `setMaximumSize`() angegeben. Mit `getMaximumSize`() kann dieser Wert ausgelesen werden. Das Gleiche gilt für die minimale Grösse einer Komponente. Diese wird mit `setMinimumSize`() gesetzt und mit `getMinimumSize`() ausgelesen.

Um eine visuelle Trennung zwischen den Komponenten zu ermöglichen, gibt es die Methode `setBorder`() und `getBorder`().

Um eine Komponente sichtbar zu machen, kann die Methode `setVisible`() verwendet werden. Je nach Übergabeparameter `true` oder `false` ist die Komponente sichtbar oder unsichtbar.

Mit der Methode setToolTipText() kann einer Komponente ein "Tooltip" zugewiesen werden. Dies ist eine Anzeige, die erscheint, wenn der Benutzer etwas länger bei einer Swing Komponente verharrt. Sie dient ihrer genaueren Beschreibung. Mit getToolTipText() kann der anzuzeigende Text ausgelesen werden.

Die nachfolgende Hierarchie zeigt alle von JComponent abgeleiteten Komponenten von Swing.

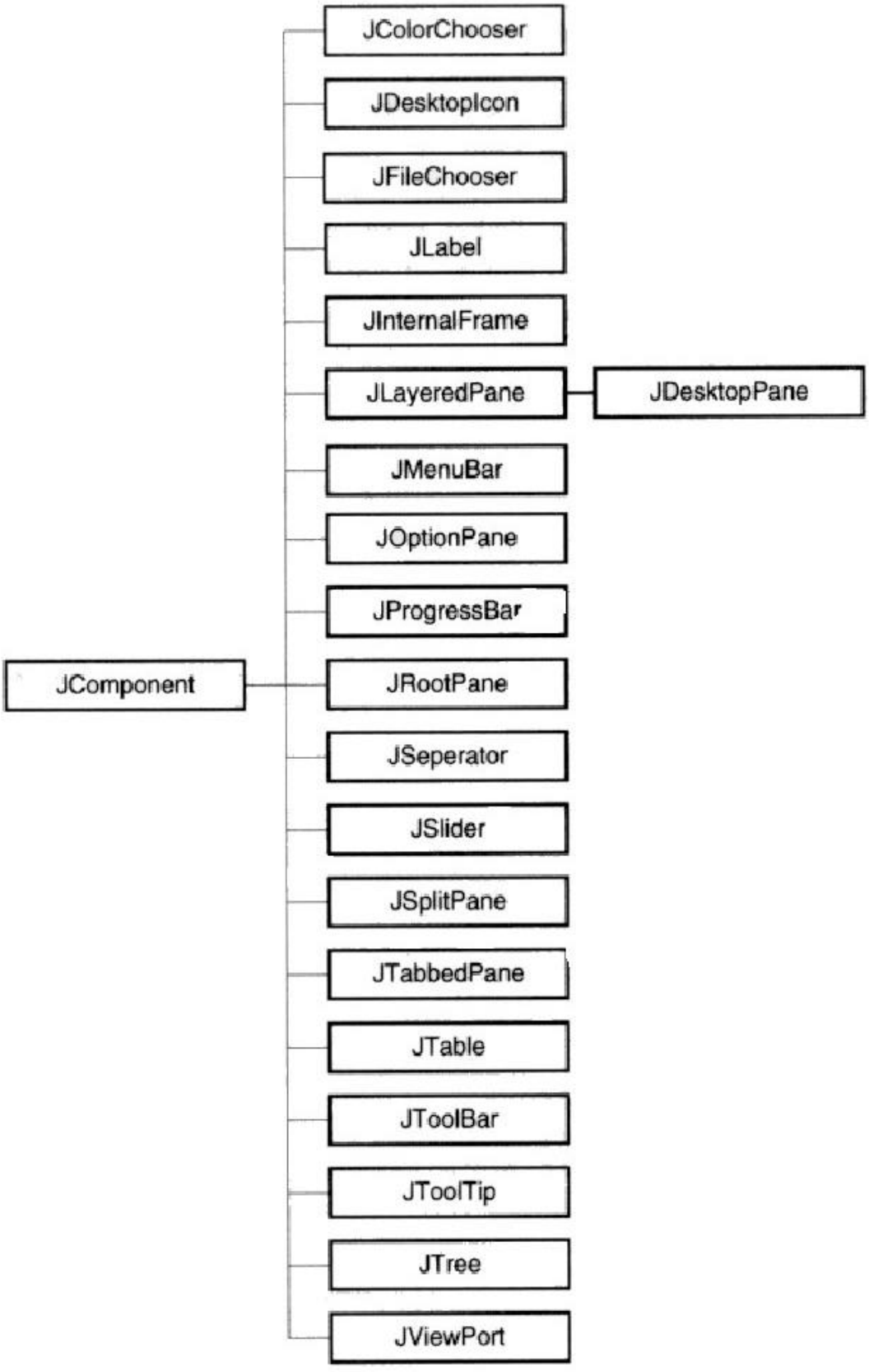


Abbildung 4-4-1 Klassen, die von JComponent abgeleitet sind

4.2. JPanel

`JPanel` ist wohl eine der einfachsten, doch zugleich wichtigsten Komponenten in Swing. Bei `JPanel` handelt es sich um ein Feld, in welches andere Komponenten eingefügt werden können. Die Komponente selbst ist dabei ein leichtgewichtiges Panel mit Unterstützung von Double Buffering. Dabei wird ein Bild im Hintergrund gezeichnet, während ein anderes im Vordergrund angezeigt wird. Dies verhindert ein Flimmern des Bildes für den Benutzer, falls das Zeichnen zu lange dauert. Das Double Buffering kann mit `setBuffered`() ein- und ausgeschaltet werden. Das `JPanel` ist selbst nicht sichtbar, ausser es wird mit `setBorder`() ein Rahmen hinzugefügt.
Ein weiteres Feature ist der bereits eingebaute Layout Manager in einem Panel. Der Layout Manager für `JPanel` ist der `FlowLayout`. Dabei werden alle Komponenten der Reihe nach angeordnet (siehe Layout Manager). Mit `setLayout`() kann allerdings ein anderes Layout aktiviert werden.

Da die Komponente selbst nicht sichtbar ist, können ihr mit der Methode `add`() neue Komponenten hinzugefügt werden. Das folgende Beispiel zeigt die Verwendung eines Panels.

Listing 4-4-1

```
package JPanelApplet;

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


public class JPanelDemo extends JApplet
{
  public void init()
  {
    getContentPane().add(new JPanelCode());
  }

  // Erstellt ein Panel mit Rahmen
  public static void main(String[] args)
  {
    Frame frame = new Frame("Panel Beispiel");

    WindowListener l = new WindowAdapter() {
        public void windowClosing(WindowEvent e) {System.exit(0);}
    };
    frame.addWindowListener(l);

    JPanelDemo applet = new JPanelDemo();
    frame.add("Center", applet);
    applet.init();
    frame.pack();
    frame.show();
  }

}

package JPanelApplet;

import java.awt.*;
```

```
import javax.swing.*;


public class JPanelCode extends JPanel {

	public JPanelCode() {
		super();
		// wird zum Panel hinzugefügt
		add (new JButton("Klick hier !"));
		add (new JTextField("Nichts passiert !!"));
		// Schachtelung ist möglich
		add (new JPanel());
		// Methode von JComponent
		setPreferredSize(new Dimension(300,50));

	}

}
```

Abbildung 4-4-2

Der `JButton` und das `JTextField` befinden sich innerhalb des `Panels`, das wiederum mit `getContentPanel`().`add`() dem `JFrame` Objekt hinzugefügt wurde. Dabei ist es möglich, eine beliebig tief verschachtelte Struktur mit Swing Komponenten herzustellen.

4.3. ImageIcon

Die Klasse `ImageIcon` implementiert das Interface `Icon`. Die Klasse `ImageIcon` lädt ein Bild mit Hilfe eines Dateinamens oder durch Angabe einer URL. Bei dem Bild können unterschiedliche Formate importiert werden. Sogar animierte GIF's können in der `ImageIcon` Klasse benutzt werden. Das Interface stellt dabei `getIconHeight`() zur Verfügung, um die Höhe des Icons zu erhalten. Dementsprechend kann mit der Methode `getIconWidth`() die Breite des Icons herausgefunden werden. Die Methode `paintIcon`() zeichnet das Icon dann auf einem bestimmten Platz einer Komponente. Die Methode `setDescription`() fügt einem Bild eine Erklärung hinzu. Mit `getDescription`() kann diese Information wieder abgerufen werden.

Listing 4-4-2

package ImageIconDemo;

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

public class ImageIconDemo extends JApplet
{
  public void init()
  {    getContentPane().add(new ImageLesenUndLaden(this));  }
  // Erstellt ein Panel mit Rahmen
  public static void main(String[] args)
  {
    Frame frame = new Frame("ImageIcon Swing Klasse");

    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {System.exit(0);}
    };
    frame.addWindowListener(l);
    ImageIconDemo applet = new ImageIconDemo();
    frame.add("Center", applet);
    applet.init();
    frame.pack();
    frame.show();
  }
}
```

```
package ImageIconDemo;

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

public class ImageLesenUndLaden extends JPanel {
    public ImageLesenUndLaden(ImageIconDemo applet) {
      Image img = Toolkit.getDefaultToolkit().getImage("move.gif");
      Image normalImage = Toolkit.getDefaultToolkit().getImage("blue.gif");
      ImageIcon icon = new ImageIcon(img);
      JButton btnImage = new JButton("Animated Gif");
      btnImage.setPreferredSize(new Dimension(150,40));
      btnImage.setIcon(icon);
      add(btnImage);
      JButton btnNormal = new JButton(new ImageIcon(normalImage));
            btnNormal.setPreferredSize(new Dimension(150,40));
      btnNormal.setText("Normal Button");
      add(btnNormal);
    }
}
```

Abbildung 4-4-3

4.4. JLabel

Die Klasse `JLabel` gehört zu den grundlegenden Elementen von Swing. Kaum eine Anwendung wird auf sie verzichten können. Swing Labels erlauben die Darstellung von Text, Bildern oder beides zugleich. Sie reagieren nicht auf Eingaben des Benutzers, sei es per Maus oder Taste. Sie erweisen sich als einfach zu handhabende Hilfsmittel für die Beschriftung von Oberflächen. Ihre Verwendung gleicht in der Grundfunktionalität dem Vorgänger Label von AWT, besitzt jedoch dank der Swing Erweiterungen einige Spezialfunktionen.

Die Festlegung des Textes, Bildes (Icons) sowie der Ausrichtung des Textes auf der horizontalen Ebene kann bereits zeitgleich mit der Erzeugung `des JLabel` einhergehen. Alle drei Angaben sind optional. Sie können aber auch im Laufe des Programms mit den Methoden `setText`(), `setIcon`() und `setHorizontalAlignment`() eingestellt werden. Zusätzlich entscheidet `setVerticalAlignment`(), ob der Inhalt des Labels oben, unten oder mittig angeordnet ist. Während sich die letzten Methoden auf den gesamten Inhalt des Labels beziehen, geben `setHorizontalTextPosition`() und `setVerticalTextPosition`() die Textposition relativ zum Icon vor. Deren Kombination ergibt eine Menge Darstellungsmöglichkeiten. Auch der Abstand zwischen Text und Bild ist mit `setIconTextGap`() justierbar. Und wenn der Programmierer einer bestimmten Schriftart den Vorzug geben will, kann er dies mit `setFont`() jederzeit tun.

Wie bereits erwähnt, gibt `JLabel` keine Reaktion auf Eingaben des Benutzers. Es ist jedoch möglich, sie einer Komponente (zum Beispiel einem Textfeld) mit `setLabelFor`() anzuhängen und einen Buchstaben des Labels mit `setDisplayedMnemonic`() als Tastaturkürzel zu bestimmen. Bei Betätigung des Kürzels bekommt das Textfeld den Fokus.

Listing 4-4-3 Generelles Rahmenprogramm für die folgenden Beispiele

package JLabelDemo;

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


public class SwingBeispiel extends JApplet
{
  public void init()
  {
    getContentPane().add(new Labels(this));
  }

  // Erstellt ein Panel mit Rahmen
  public static void main(String[] args)
  {
    Frame frame = new Frame("Swing Beispiel");

    WindowListener l = new WindowAdapter() {
        public void windowClosing(WindowEvent e) {System.exit(0);}
    };
    frame.addWindowListener(l);

    SwingBeispiel applet = new SwingBeispiel();
    frame.add("Center", applet);
```

```
    applet.init();
    frame.pack();
    frame.show();
  }

}
```

Listing 4-4

```
package JLabelDemo;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;


public class Labels extends JPanel
{
  ImageIcon      smileIcon;
  JLabel         label;
  EtchedBorder   etchedBorder;


  Labels(JApplet applet)
  {
    setLayout(new GridLayout(0, 2, 5, 5));
    etchedBorder = new EtchedBorder(EtchedBorder.RAISED);

    try
    {
      smileIcon = new ImageIcon("smile.gif");
    }
    catch(SecurityException se)      // Für Internet-Browser
    {
      smileIcon = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "smile.gif"));
    }

    add(label = new JLabel("Normales Label"));
    label.setBorder(etchedBorder);

    add(label = new JLabel(smileIcon));
    label.setBorder(etchedBorder);

    add(label = new JLabel("Links", smileIcon, JLabel.LEFT));
    label.setBorder(etchedBorder);

    add(label = new JLabel("Rechts", smileIcon, JLabel.RIGHT));
    label.setBorder(etchedBorder);

    add(label = new JLabel("Links unten", smileIcon,
      JLabel.CENTER));
    label.setHorizontalTextPosition(JLabel.LEFT);
    label.setVerticalTextPosition(JLabel.BOTTOM);
    label.setBorder(etchedBorder);

    add(label = new JLabel("Links oben", smileIcon, JLabel.CENTER));
    label.setHorizontalTextPosition(JLabel.LEFT);
    label.setVerticalTextPosition(JLabel.TOP);
    label.setBorder(etchedBorder);
```

```
    add(label = new JLabel("Rechts unten", smileIcon,
      JLabel.CENTER));
    label.setHorizontalTextPosition(JLabel.RIGHT);
    label.setVerticalTextPosition(JLabel.BOTTOM);
    label.setBorder(etchedBorder);

    add(label = new JLabel("Rechts oben", smileIcon,
      JLabel.CENTER));
    label.setHorizontalTextPosition(JLabel.RIGHT);
    label.setVerticalTextPosition(JLabel.TOP);
    label.setBorder(etchedBorder);

    add(label = new JLabel("Drunter", smileIcon, JLabel.CENTER));
    label.setHorizontalTextPosition(JLabel.CENTER);
    label.setVerticalTextPosition(JLabel.BOTTOM);
    label.setBorder(etchedBorder);

    add(label = new JLabel("Drüber", smileIcon, JLabel.CENTER));
    label.setHorizontalTextPosition(JLabel.CENTER);
    label.setVerticalTextPosition(JLabel.TOP);
    label.setBorder(etchedBorder);
  }
}
```

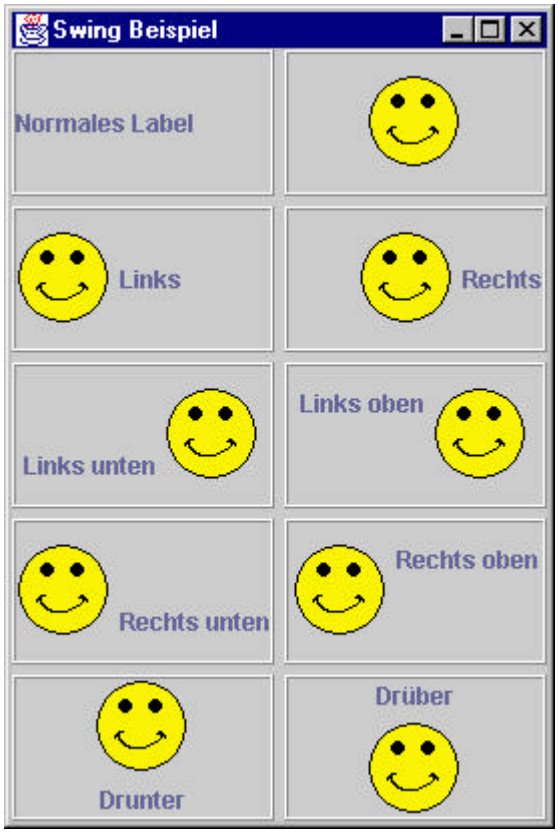


Abbildung 4-4

4.5. JToolTip

Tooltips vermitteln dem Anwender eine kurze Auskunft über die Funktion einer Komponente. Sie erscheinen, wenn der Mauszeiger über der Komponente innehält. Meistens handelt es sich nur um ein Schlagwort. Selten wird heute noch auf sie verzichtet. Nach der Erzeugung, die übrigens parameterlos erfolgt, wird mit `setTipText()` der Inhalt der Auskunft bestimmt. Mit `setComponent()` ordnet man dann den Tooltip einer Komponente zu.

Es ist aber überflüssig eine Komponente mit einem Tooltip zu versehen. Die Methode `setToolTipText()` von `JComponent` nimmt dem Anwender diese Arbeit ab.

In diesem Kapitel
- *AbstractBorder*
- *BevelBorder*
- *SoftBevelBorder*
- *CompoundBorder*
- *EmptyBorder*
- *EtchedBorder*
- *LineBorder*
- *MatteBorder*
- *TitleBorder*
- *BorderFactory*

5. Rahmen

Rahmen werden in vielerlei Hinsicht eingesetzt: in Fenstern, Buttons, Textfeldern oder in vielen andern Oberflächenkomponenten. Sie sind so unerlässlich wie Labels. Man findet sie auch in den verschiedensten Darstellungen. Einmal mit Schatten, dann mit Gravur und schliesslich mit gewöhnlichen Linien. Swing bietet hier ein Paket an Border-Klassen, die den Anforderungen moderner Oberflächen gerecht werden:
`javax.swing.border`
Für jeden Rahmentyp steht die entsprechende Klasse bereit, die im folgenden alle einzeln vorgestellt werden.

5.1. AbstractBorder

Diese Klasse entspricht einem Rahmen ohne Umfang. Sie selbst kann nicht verwendet werden, da es sich hier um eine abstrakte Klasse handelt. Sie implementiert die `Border` Schnittstelle und enthält die nötigen Basisfunktionen aller folgender Rahmenarten, die sie beerbt. Am interessantesten ist die `getBorderInsets`() Methode, die den Spielraum zwischen Komponenteninhalt und Rahmen ermitteln.

5.2. BevelBorder

Dabei handelt es sich um Rahmen mit 3D Effekt. Ob erhöht oder vertieft, muss schon im Konstruktor spezifiziert werden. Erlaubt sind BeverBorder.RAISED und BevelBorder.LOWERED. Die Farben des inneren und äusseren Schattens bzw. der belichteten Seiten liefert die Methoden

```
getHighlightInnerColor()
getShadowInnerColor()
getHighlightOuterColor()
getShadowOuterColor()
```

5.3. SoftBevelBorder

Eine Ableitung des `BevelBorder` unterscheidet sich lediglich in den leicht abgestampften Kanten. Ansonsten ist der Umgang derselbe wie beim `BevelBorder`.

5.4. CompoundBorder

Der `CompoundBorder` kombiniert zwei Rahmen zu einem. Er plaziert eine Rahmen in den Innenraum des andern. Es gibt also einen inneren und einen äusseren Rahmen. Das macht zum Beispiel Sinn, um zwischen den eigentlichen Rahmen und der zu rahmenden

Komponente einen Spielraum zu lassen. Man erzeugt einfach als inneren Rahmen einen `EmptyBorder` (siehe nächsten Abschnitt).

Beispiel:

```
BevelBorder rahmen = new BevelBorder(BevelBorder.LOWERED);
EmptyBorder spielraum = new EmptyBorder(0,0,4,4);
komponente.setBorder(new CompoundBorder(rahmen, spielraum) );
```

5.5. EmptyBorder

Dieser Rahmen bietet eigentlich nicht viel Aufregendes. Er ist, wie der Name schon aussagt, leer. Er zeichnet unsichtbare Linien, die lediglich den Platz einnehmen, den man im Konstruktor angibt. Er kann als Platzhalter zwischen den Oberflächenelementen verwendet werden.

5.6. EtchedBorder

Der `EtchedBorder` hat den Effekt einer Gravur. Er kann nach Wunsch erhöht oder vertieft sein. Die Farben für die Schatten und die belichteten Stellen sind variierbar. Alle Einstellungen sind im Konstruktor vorzunehmen.

5.7. LineBorder

Der `LineBorder` zeichnet einen einfarbigen Rahmen. Sowohl Farbe und Dicke der Linie können im Konstruktor angepasst werden.

5.8. MatteBorder

Diese Rahmenart ist abgeleitet von `EmptyBorder` und funktioniert ähnlich. Es wird allerdings der eingerückte Platz mit einer bestimmten Farbe gefüllt oder mit einem Icon gekachelt. Farbe oder Icon werden dem Konstruktor übergeben.

5.9. TitleBorder

Der `TitleBorder` verwendet einen bereits vorhandenen Rahmen und versieht ihn mit einer Überschrift oder Bezeichnung. Dabei kann man festlegen, welche Kante den Text bekommen soll und ob er über, auf oder unter der Kante erscheinen soll. Schriftart und Farbe des Textes kann der Programmierer ebenfalls frei wählen. Alle Angaben können schon im Konstruktor getätigt werden. Es existieren jedoch entsprechende Methoden, um die Einstellung während des Programmablaufs zu ändern: `setBorder(), setTitle(), setTitleJustification(), setTitlePosition(), setTitleFont()` und `setTitleColor()`.

5.10. BorderFactory

`BorderFactory` gehört zu `javax.swing` und nicht zum Paket `javax.swing.border`. Sie bietet dem Programmierer Dienstleitungen, um Rahmen auf einem anderen Weg zu erzeugen. Für jede der obengenannten Rahmenarten stellt `BorderFactory` statische `create` Funktionen zur Verfügung. Die Parameter die eben noch in den Konstruktor gehörten, müssen hier in den Funktionen stehen. Die Funktionen übernehmen das Instanzieren der Borderklassen und versuchen - soweit es geht - diese mehrmals zu verwenden. Wenn jemand zum Beispiel mit `createBevelBorder()` fünf identische Rahmen anlegt, wird in Wirklichkeit nur einer erzeugt.

Für jede Rahmenart gibt es eine `create` Funktion:

```
createBevelBorder()
createSoftBevelBorder()
createCompoundBorder()
createEtchedBorder()
createLineBorder()
createMatteBorder()
createTitleBorder()
```

Listing 5-1 Einfaches Beispiel verschiedener Rahmen (selber Rahmen wie oben)

```
package RahmenDemo;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;


public class Borders extends JPanel
{
  JLabel         borderLabel;
  ImageIcon      javaIcon;
  BevelBorder    bevelBorder;
  EtchedBorder   etchedBorder;

  Borders(JApplet applet)
  {
    setLayout(new GridLayout(0, 2, 5, 5));

    try
    {
      javaIcon = new ImageIcon("java.gif");
    }
    catch(SecurityException se)  // Für Internet-Browser
    {
      javaIcon = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "java.gif"));
    }

    // BevelBorder - Erhöht
    borderLabel = new JLabel("BevelBorder - RAISED");
    borderLabel.setHorizontalAlignment(SwingConstants.CENTER);
    borderLabel.setBorder(
      bevelBorder = new BevelBorder(BevelBorder.RAISED));
    add(borderLabel);

    // BevelBorder - Erniedrigt
    borderLabel = new JLabel("BevelBorder - LOWERED");
    borderLabel.setHorizontalAlignment(SwingConstants.CENTER);
    borderLabel.setBorder(new BevelBorder(BevelBorder.LOWERED));
    add(borderLabel);

    // SoftBevelBorder - Erhöht
    borderLabel = new JLabel("SoftBevelBorder - RAISED");
    borderLabel.setHorizontalAlignment(SwingConstants.CENTER);
    borderLabel.setBorder(new SoftBevelBorder(BevelBorder.RAISED));
    add(borderLabel);

    // SoftBevelBorder - Erniedrigt
    borderLabel = new JLabel("SoftBevelBorder - LOWERED");
    borderLabel.setHorizontalAlignment(SwingConstants.CENTER);
    borderLabel.setBorder(new SoftBevelBorder(BevelBorder.LOWERED));
```

```
    add(borderLabel);

    // EtchedBorder - Erhöht
    borderLabel = new JLabel("EtchedBorder - RAISED");
    borderLabel.setHorizontalAlignment(SwingConstants.CENTER);
    borderLabel.setBorder(
      etchedBorder = new EtchedBorder(EtchedBorder.RAISED));
    add(borderLabel);

    // EtchedBorder - Erniedrigt
    borderLabel = new JLabel("EtchedBorder - LOWERED");
    borderLabel.setHorizontalAlignment(SwingConstants.CENTER);
    borderLabel.setBorder(new EtchedBorder(EtchedBorder.LOWERED));
    add(borderLabel);

    // LineBorder - Schwarz - Dicke 1
    borderLabel = new JLabel("LineBorder - Schwarz - Dicke 1");
    borderLabel.setHorizontalAlignment(SwingConstants.CENTER);
    borderLabel.setBorder(new LineBorder(Color.black));
    add(borderLabel);

    // LineBorder - Blau - Dicke 4
    borderLabel = new JLabel("LineBorder - Blau - Dicke 4");
    borderLabel.setHorizontalAlignment(SwingConstants.CENTER);
    borderLabel.setBorder(new LineBorder(Color.blue, 4));
    add(borderLabel);

    // EmptyBorder
    borderLabel = new JLabel("EmptyBorder");
    borderLabel.setHorizontalAlignment(SwingConstants.CENTER);
    borderLabel.setBorder(new EmptyBorder(0, 0, 3, 3));
    add(borderLabel);

    // MatteBorder
    borderLabel = new JLabel("MatteBorder");
    borderLabel.setHorizontalAlignment(SwingConstants.CENTER);
    borderLabel.setBorder(
      new MatteBorder(16, 16, 16, 16, javaIcon));
    add(borderLabel);

    // TitledBorder 1 mit BevelBorder
    borderLabel = new JLabel("TitledBorder 1");
    borderLabel.setHorizontalAlignment(SwingConstants.CENTER);
    borderLabel.setBorder(
      new TitledBorder(bevelBorder, "Verwendet BevelBorder"));
    add(borderLabel);

    // TitledBorder 2 mit BevelBorder
    borderLabel = new JLabel("TitledBorder 2");
    borderLabel.setHorizontalAlignment(SwingConstants.CENTER);
    borderLabel.setBorder(
      new TitledBorder(etchedBorder, "Verwendet EtchedBorder",
      TitledBorder.RIGHT, 0,
      new Font("TimesRoman", Font.BOLD, 16)));
    add(borderLabel);
   }
}
```

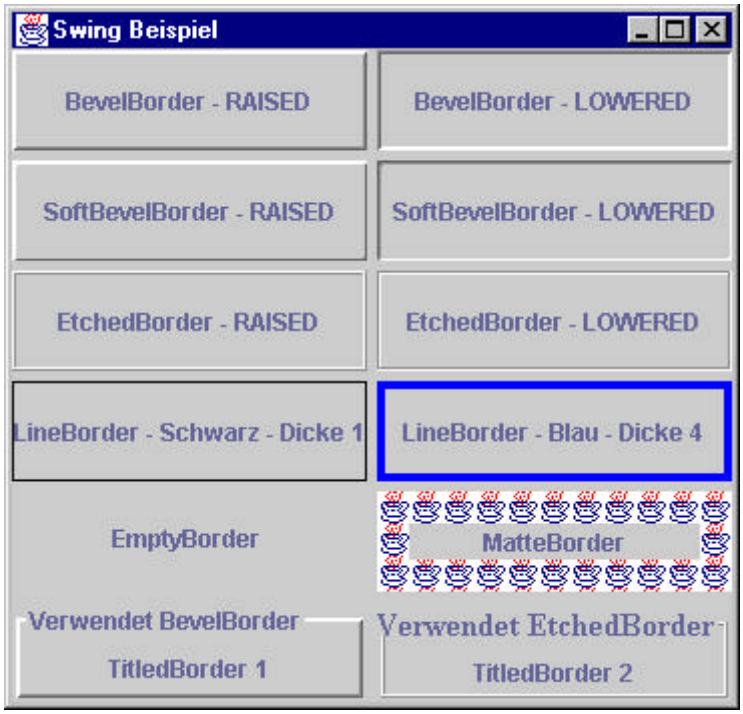
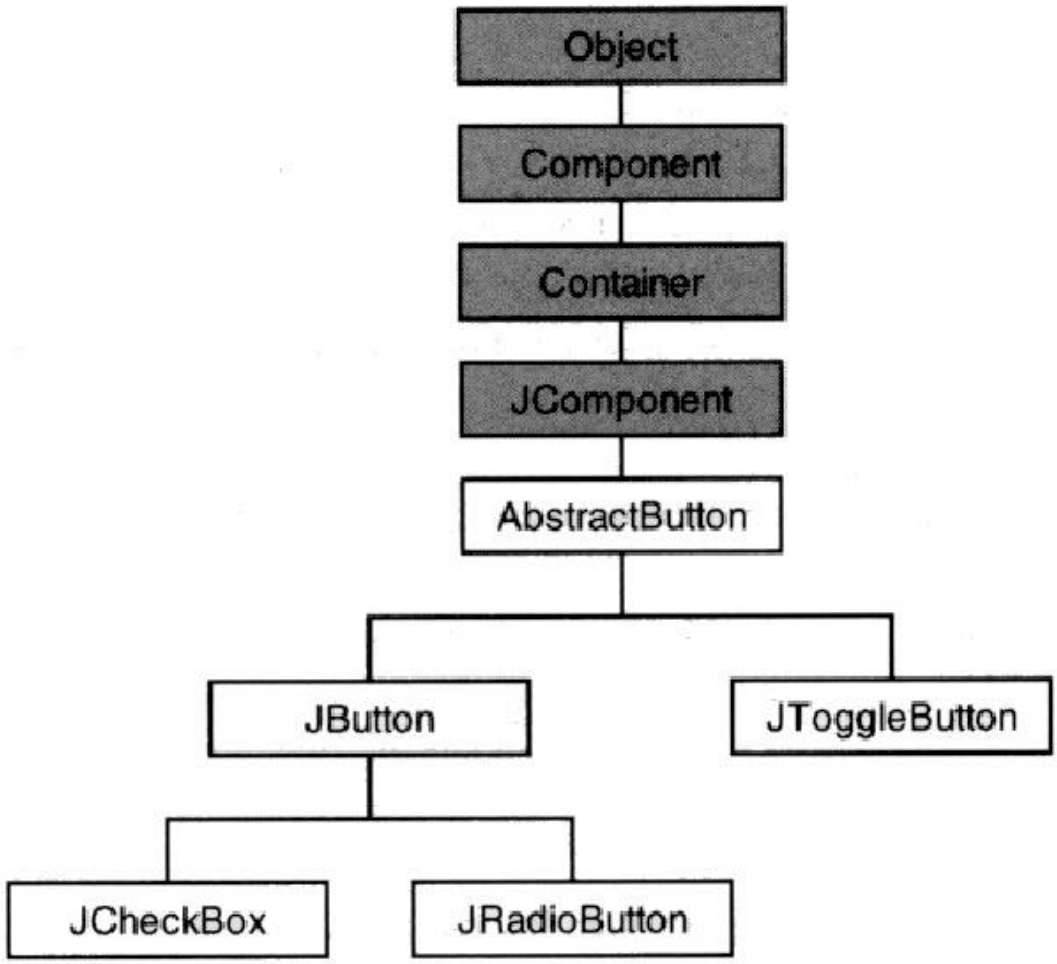


Abbildung 5-1 Bildschirm Darstellung der unterschiedlichen Rahmen

In diesem Kapitel
- *AbstractButton*
- *JButton*
- *JCheckBox*
- *JRadioButton & ButtonGroup*
- *JToggleButton*

6. *Buttons*



Bisher mussten sich die Anwender Javas mit dem Vorrat an Schaltflächen des AWTs zufriedengeben. Jener beschränkte sich auf die Klasse `Button` und `Checkbox`. Dabei gibt es, wie jeder erfahrene Programmierer bestätigen kann, ausreichend Situationen, in denen mehr Funktionen des Buttons nötig wären, sowie mehr Verwandte dieser Komponente. Die Komponentensammlung von Swing wurde sinnvoll ergänzt mit Klassen, die man für moderne Applikationen braucht: `JButton, JCheckBox, JRadioButton` und `JToggleButton`.

Es mag vielleicht aufgefallen sein, dass im obigen Diagramm einige Klassen fehlen, die ebenfalls von `AbstractButton` abgeleitet sind. Es handelt sich hierbei um Menükomponenten,. Ihnen widmet sich das Kapitel "Menüs".

6.1. AbstractButton

Die Speerspitze aller Button-Komponenten stellt die Klasse `AbstractButton` dar. Wie der Name schon sagt, ist sie eine abstrakte Klasse. Mehrere Komponenten wie zum Beispiel `JButton, JCheckBox` oder `JRadioButton` verwenden sie als Basis und erben ihre Methoden. Das ist sinnvoll, weil die Komponenten ähnliche Verhaltensweisen aufweisen. `AbstractButton` kann natürlich auch als Basis benutzerdefinierter Buttons dienen. Die wichtigsten Methoden vorab:

- `setMnemonic(char)`
  versieht den Button mit einem Tastenkürzel, welches unterstrichen im Text erscheint.
- `doClick()`
  Der Button wird vom Programm selbst abgeklickt.
- `setIcon(Icon)`
  besetzt die Komponente mit einem Icon
- `setDisabledIcon(Icon)`
  legt das Icon für einen deaktivierten Button fest.
- `setDiabledSelectedIcon(Icon)`
  bestimmt das Icon für einen selektierten und deaktivierten Button (zum Beispiel `JCheckBox, JToggleButton`)
- `setPressedIcon(Icon)`
  das Icon erscheint, wenn der Button gedrückt wird.
- `setRolloverIcon(Icon)`
  das Icon erscheint, wenn der Mauszeiger über dem Button liegt.
- `setRolloverSelectedIcon(Icon)`
  analog zu `setRolloverIcon`(), jedoch nur für selektierte Buttons (zum Beispiel `JCheckBox, JToggleButton`)
- `setSelectedIcon(Icon)`
  bestimmt das Icon für den selektierten Button (zum Beispiel `JCheckBox, JToggleButton`)
- `setVerticalTextPosition(int) / setHorizontalTextPosition(int)`
  Die Position des Textes wird relativ zum Icon ausgerichtet.
- `serVerticalAlignment(int) / setHorizontalAlignment(int)`
  positioniert den Inhalt (Icon und Text) im Button

6.2. JButton

Die erste Variante eines AbstractButton, die es zu untersuchen gibt, ist der Jbutton. Die Verwendung gleicht auf den ersten Blick der des Button im AWT. Das gilt auch für das Anlegen einer Schaltfläche.

Der Programmierer kann mit der Anwendung der Funktion des AbstractButton jene auf vielfältige Weise gestalten. Das folgende Programm zeigt einige Varianten der Anwendung in der Praxis. Ausserdem reagieren die Aktionen der Schaltflächen mit Hilfe des ActionListener. Klicken Sie bitte vom 2. bis zum 5. Button alle der Reihe nach an. Jeder von Ihnen verhält sich etwas anders.

Listing 6-1 Verwendung von JButton

```
package Buttons;

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


public class Buttons extends JPanel
{
  JButton button1, button2, button3, button4, button5;
  ImageIcon smileIcon1;
  ImageIcon smileIcon2;
  ImageIcon aniSmile;

  Buttons(JApplet applet)
  {
    setLayout(new GridLayout(0, 1));

    try
    {
      smileIcon1 = new ImageIcon("smile1.gif");
      smileIcon2 = new ImageIcon("smile2.gif");
      aniSmile = new ImageIcon("anismile.gif");
    }
    catch(SecurityException se)      // Für Internet-Browser
    {
      smileIcon1 = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "smile1.gif"));
      smileIcon2 = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "smile2.gif"));
      aniSmile = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "anismile.gif"));
    }

    // Button mit einem einfachen Icon
    button1 = new JButton(smileIcon1);
    add(button1);

    // Button mit Icon und Text
    // Die Beschriftung befindet sich rechts
```

```
    button2 = new JButton("Klick mich.", smileIcon1);
    button2.setActionCommand("2");
    button2.addActionListener(new buttonReaction());
    button2.setMnemonic('K');
    add(button2);

    // Ein Button mit einem animierten Image
    // Die Beschriftung befindet sich unten
    button3 = new JButton("Super! Jetzt mich.", smileIcon1);
    button3.setVerticalTextPosition(AbstractButton.BOTTOM);
    button3.setHorizontalTextPosition(AbstractButton.CENTER);
    button3.setEnabled(false);
    button3.setActionCommand("3");
    button3.addActionListener(new buttonReaction());
    // ImageObserver einstellen
    aniSmile.setImageObserver(button3);
    button3.setMnemonic('S');
    add(button3);

    // Der Button zeigt bei Auswahl ein zweites Icon
    button4 = new JButton("Dann mich.", smileIcon1);
    button4.setEnabled(false);
    button4.setHorizontalTextPosition(AbstractButton.LEFT);
    button4.setActionCommand("4");
    button4.addActionListener(new buttonReaction());
    // Icon, das beim Auswahl erscheinen soll
    button4.setPressedIcon(smileIcon2);
    button4.setMnemonic('D');
    add(button4);

    // Der Button reagiert wenn der Mauszeiger darüber liegt
    button5 = new JButton("Und noch mich.", smileIcon1);
    button5.setEnabled(false);
    button5.setVerticalTextPosition(AbstractButton.TOP);
    button3.setHorizontalTextPosition(AbstractButton.CENTER);
    button5.addActionListener(new buttonReaction());
    button5.setActionCommand("5");
    // Icon, das beim Mauszeiger erscheinen soll
    button5.setRolloverIcon(smileIcon2);
    // Reagiert, wenn der Mauszeiger darüber liegt
    button5.setRolloverEnabled(true);
    button5.setMnemonic('U');
    add(button5);
  }

  // Hier wird die Aktionen verarbeitet
  class buttonReaction implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      // den gedrückten Button deaktivieren
      ((JButton)(e.getSource())).setEnabled(false);
      switch(Integer.parseInt(e.getActionCommand()))
      {
        case 2:
          button3.setIcon(aniSmile);
          button3.setEnabled(true);
          break;
        case 3:
          button3.setIcon(smileIcon1);
          button4.setEnabled(true);
          break;
```

```
            case 4:
              button5.setEnabled(true);
              break;
            case 5:
              button2.setEnabled(true);
              break;
          }
        }
      }
    }
```

Abbildung 6-1 JButton

6.3. JCheckBox

Wie auch `JButton` setzt `JCheckBox` auf `AbstractButton` auf und entspricht `CheckBox` im AWT. Durch das zahlreiche Angebot der Methoden in `AbstractButton` lässt der Einsatz von Icons in Checkboxen nur wenig Wünsche offen. In manchen Fällen ist das Kästchen, das mit dem Häkchen als angekreuzt gilt, unzureichend. Der Programmierer wünscht sich vielleicht als Alternative diverse Symbole, die der Umgebung entsprechen und hat sich dafür ein paar Icons vorbereitet. Im AWT müsste man explizit eine eigene `paint`() Methode entwerfen, die das Zeichen des benutzerdefinierten Icons übernimmt. Einfacher lässt sich dies bei `JCheckBox` mit `setIcon`() und `setSelectedIcon`() realisieren.

Listing 6-2

```
package CheckBoxes;

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


public class CheckBoxes extends JPanel{
  JCheckBox      checkBox;
  ImageIcon      lightOff, lightOn;

  CheckBoxes(JApplet applet){
    setLayout(new GridLayout(2, 3));

    // CheckBoxes mit einfachen Text
    checkBox = new JCheckBox("Messer");
    checkBox.setMnemonic('M');  // Tastenbefehl festlegen
    add(checkBox);
    checkBox = new JCheckBox("Gabel");
    checkBox.setMnemonic('G');
    add(checkBox);
    checkBox = new JCheckBox("Löffel");
    checkBox.setMnemonic('L');
    add(checkBox);

    // CheckBoxes mit Images
    try
    {
      lightOff = new ImageIcon("lightoff.gif");
      lightOn  = new ImageIcon("lighton.gif");
    }
    catch(SecurityException se)
    {
      lightOff = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "lightoff.gif"));
      lightOn = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "lighton.gif"));
    }
    checkBox = new JCheckBox("Messer", lightOff);
    // Ersetzt das Häkchen durch ein Icon
    checkBox.setSelectedIcon(lightOn);
    add(checkBox);
    checkBox = new JCheckBox("Gabel",lightOff);
    checkBox.setSelectedIcon(lightOn);
```

```
    add(checkBox);
    checkBox = new JCheckBox("Löffel", lightOff);
    checkBox.setSelectedIcon(lightOn);
    add(checkBox);
  }
}
```

Das Rahmenprogramm ist wieder identisch mit dem bereits verwendeten.
Und hier noch ein Screenshot:

Abbildung 6-2

6.4. JRadioButton & ButtonGroup

Der `JRadioButton` ist ein weiterer Nachkomme von `AbstractButton`. Im AWT war ein `RadioButton` nichts anderes als eine `CheckBox`, die zu einer `CheckBoxGroup` gehört. Somit kommt konnte jeweils nur ein Button angewählt werden. Das Erzeugen der Instanz `JRadioButton` sorgt aber nur für das Aussehen eines `RadioButtons`. Er muss einer `ButtonGroup` (ebenfalls eine Klasse von Swing) hinzugefügt werden, um das Verhalten eines echten RadioButtons zu gewährleisten. Die `ButtonGroup` ist unsichtbar und hat keinen Einfluss auf die Anordnung der Buttons. Ansonsten unterscheidet sich die Funktionalität nur wenig von der `JCheckBox`. Ausserdem steht es dem Programmierer frei, an Stelle des Kreises eigene Icons in Szene zu setzen.

Listing 6-3 Verwendung von Jbutton & ButtonGroup

```
package RadioButtons;

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


public class RadioButtons extends JPanel
{
  JRadioButton   button;
  ButtonGroup    buttonGroup;
  ImageIcon      checkOff, checkOn;

  RadioButtons(JApplet applet)
  {
    setLayout(new GridLayout(2, 3));

    // Einfache Radiobuttons mit Text
    buttonGroup = new ButtonGroup();
    button = new JRadioButton("Gut", true);
    button.setMnemonic('G');
    buttonGroup.add(button);
    add(button);
    button = new JRadioButton("Besser");
    button.setMnemonic('B');
    buttonGroup.add(button);
    add(button);
    button = new JRadioButton("Java");
    button.setMnemonic('J');
    buttonGroup.add(button);
    add(button);

    // RadioButtons mit Icons und Text
    try
    {
      checkOff = new ImageIcon("radio_off.gif");
      checkOn = new ImageIcon("radio_on.gif");
    }
    catch(SecurityException se)
    {
      checkOff = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "radio_off.gif"));
      checkOn = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "radio_on.gif"));
```

```
    }
    buttonGroup = new ButtonGroup();
    button = new JRadioButton("Gut", checkOff);
    button.setSelectedIcon(checkOn);
    button.setSelected(true);
    buttonGroup.add(button);
    add(button);
    button = new JRadioButton("Besser", checkOff);
    button.setSelectedIcon(checkOn);
    buttonGroup.add(button);
    add(button);
    button = new JRadioButton("Java", checkOff);
    button.setSelectedIcon(checkOn);
    buttonGroup.add(button);
    add(button);
  }
}
```

Abbildung 6-3

6.5. JToggleButtons

Ein weiterer neuer Typ von Buttons in der Swing Bibliothek ist der `JToggleButton`. Im AWT gab es dafür kein Äquivalent. Ein `JToggleButton` ist vergleichbar mit einem `JRadioButton` oder einer `JCheckBox`, mit dem Aussehen eines `JButton`. Er bleibt beim Anwählen gedrückt und kann einer `ButtonGroup` hinzugefügt werden. Im Beispiel wird ein animiertes Gif für den selektierten `JToggleButton` eingesetzt.

Listing 6-4 Verwendung von JToggleButton

```
package ToggleButtons;

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

public class ToggleButtons extends JPanel{
  JPanel         panel;
  JToggleButton  button;
  ButtonGroup    buttonGroup;
  ImageIcon    iAniDino, iDino;

  public ToggleButtons(JApplet applet){
    super();

    // Erst das Hauptpannel definieren (für Hintergrund)
    setLayout(new BorderLayout());

    // Einfache TooggleButtons mit Text
    panel = new JPanel();
    buttonGroup = new ButtonGroup();
    button = new JToggleButton("Normal", true);
    button.setMnemonic('N');
    buttonGroup.add(button);
    panel.add(button);
    button = new JToggleButton("Fett");
    button.setFont(new Font("Dialog", Font.BOLD, 12));
    button.setMnemonic('F');
    buttonGroup.add(button);
    panel.add(button);
    button = new JToggleButton("Kursiv");
    button.setFont(new Font("Dialog", Font.ITALIC, 12));
    button.setMnemonic('K');
    buttonGroup.add(button);
    panel.add(button);
    button = new JToggleButton("Fett + Kursiv");
    button.setFont(
      new Font("Dialog", Font.BOLD | Font.ITALIC, 12));
    button.setMnemonic('+');
    buttonGroup.add(button);
    panel.add(button);

    add("North", panel);

    // ToggleButtons mit animierten Icon
    try
    {
      iDino = new ImageIcon("dino.gif");
      iAniDino = new ImageIcon("anidino.gif");
```

```
    }
    catch(SecurityException se){
      iDino = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "dino.gif"));
      iAniDino = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "anidino.gif"));
    }

    buttonGroup = new ButtonGroup();
    panel = new JPanel();

    button = new JToggleButton(iDino);
    // Button auf Bildgröße bringen
    button.setMargin(new Insets(0,0,0,0));
    iDino.setImageObserver(button);
    button.setSelectedIcon(iAniDino);
    button.setSelected(true);
    buttonGroup.add(button);
    panel.add(button);

    button = new JToggleButton(iDino);
    // Button auf Bildgröße bringen
    button.setMargin(new Insets(0,0,0,0));
    iDino.setImageObserver(button);
    button.setSelectedIcon(iAniDino);
    buttonGroup.add(button);
    panel.add(button);

    button = new JToggleButton(iDino);
    // Button auf Bildgröße bringen
    button.setMargin(new Insets(0,0,0,0));
    iDino.setImageObserver(button);
    button.setSelectedIcon(iAniDino);
    buttonGroup.add(button);
    panel.add(button);

    button = new JToggleButton(iDino);
    // Button auf Bildgröße bringen
    button.setMargin(new Insets(0,0,0,0));
    iDino.setImageObserver(button);
    button.setSelectedIcon(iAniDino);
    buttonGroup.add(button);
    panel.add(button);
    add("South", panel);
  }
}
```

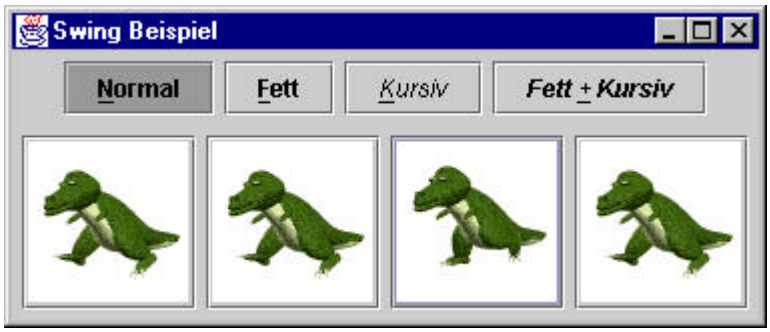


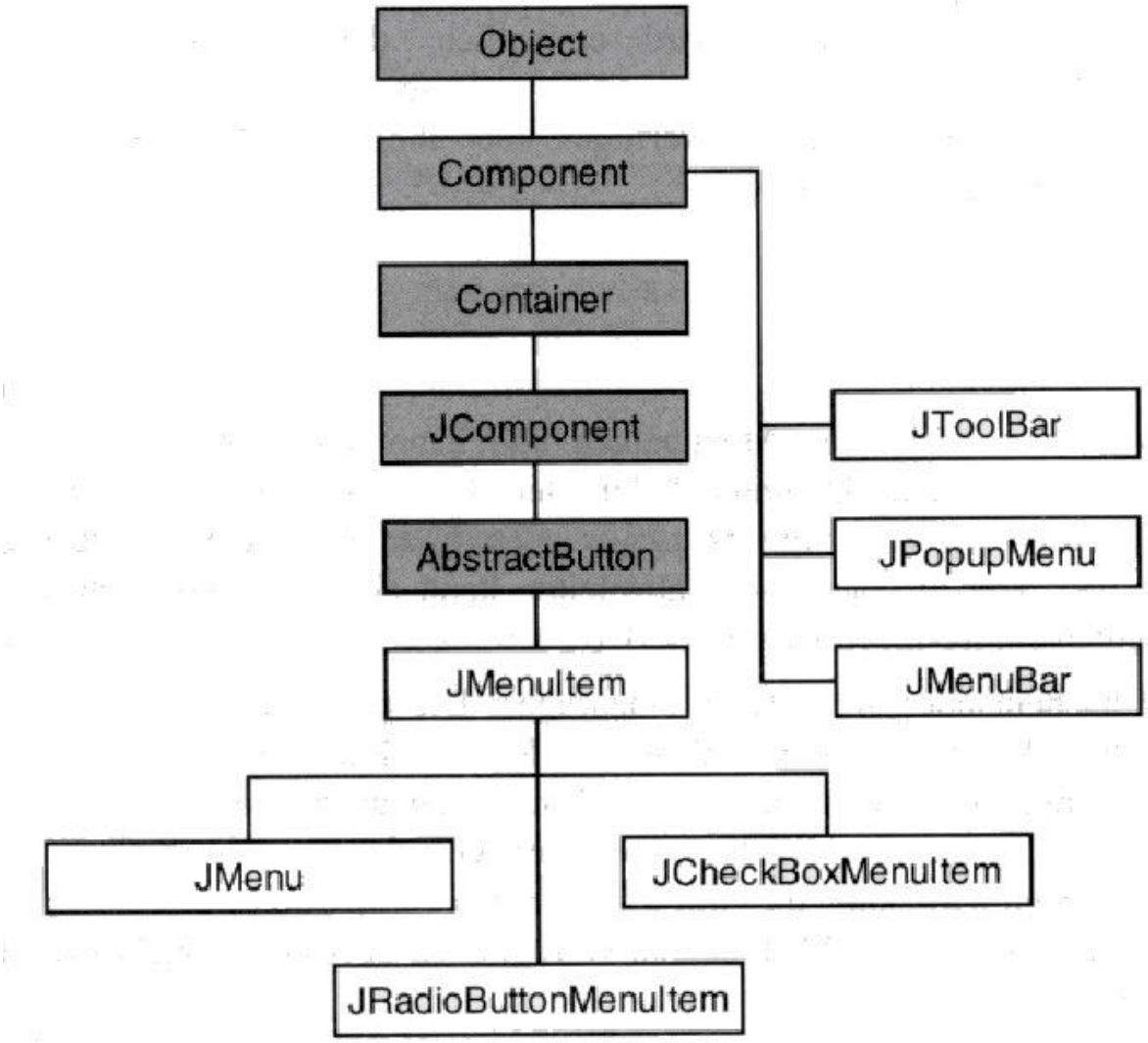
Abbildung 6-4

Dino ist ein animiertes Gif; das angewählte Bild, der angewählte Button erhält das animierte Gif. Die andern Buttons erhalten ein unanimiertes Gif.

In diesem Kapitel

- *PullDown Menu*
- *JMenuBar*
- *JMenu*
- *JMeniItem*
- *JCheckBoxMenuItem*
- *JRadioButtonMenuItem*
- *JSeparator*
- *JPopupMenu (Kontextmenü)*
- *JToolBar (Werkzeugleiste)*

7. Menüs



Das folgende Kapitel beschäftigt sich mit dem Menüsystem von Swing. Wenn man von Menüs spricht, denkt man meist an das PullDownMenü (hier sind diese Menüs nur ein Teil eines umfassenderen Menüsystems). Zusätzlich werden wir das Kontextmenü (PopUpMenü) und die Werkzeugkiste (Toolbars) kennen lernen.

7.1. PullDownMenü

Ein PullDownMenü besteht aus einem Menübalken (`JMenuBar`) am oberen Rand des Fensters mit einer Reihe von Menüs (JMenu). Diese beinhalten wiederum Menübefehle (`JMenuItem`). Anstelle eines Menübefehls kann auch ein Untermenü (ebenfalls `JMenu`) eingesetzt werden.

Das Menüsystem von Swing unterscheidet sich zunächst nicht wesentlich von dem des AWTs. Es bietet jedoch wesentliche Verbesserungen:

- die neuen Klassen `JRadioButtonMenuItem` und `JCheckBoxMenuItem` unterstützen Optionsschalter in einem Menü.
- wie man es bei Swing bereits gewohnt ist, können Bilder mit Menüeinträgen kombiniert werden.

Aus dem obigen Klassendiagramm wird ersichtlich, dass die Menükomponenten (`JMenuItem`, `JMenu, JCheckBoxMenuItem, JRadioButtonMenuItem`) von `AbstractButton` abgeleitet sind. Sie verfügen über ähnliche Fähigkeiten und Eigenschaften wie ihre Verwandten, die Buttons.

7.2. JMenuBar

Die Klasse `JMenuBar` bildet den Menübalken und kann nur in einem selbständigen Fenster untergebracht werden. Darunter fallen der `JFrame`, das `JApplet` und der `JDialog`. Die Einbindung erfolgt mit `setJMenuBar`() in einem der drei Fenstertypen. Zwar ist `JMenuBar` von `JComponent` abgeleitet, kann aber trotzdem nicht in andere Swing-Komponenten eingebunden werden (zum Beispiel `JPanel`).

Die Menüs sind nacheinander mit `add`() einzufügen. Die `getMenuCount`() Methode gibt Auskunft über die Anzahl der enthaltenen Menüs. Einzelne Menüs können mit `getMenu`() abgefragt werden, wobei dessen Position anzugeben ist. `isSelected`() überprüft, ob eine Komponente gewählt wurde. Wer will, dass der Menübalken nicht mit einem Rahmen versehen wird, kann dies mit `setBorderPainted`() erreichen, indem er als Parameter `false` eingibt. Der Menübalken verfügt auch über ein spezielles Hilfsmenü : das `setHelpMenu`().

7.3. JMenu

Das `JMenu` erzeugt das Menü und wird im Menübalken eingebettet. Ein `JMenu` nimmt Komponenten auf, die von `JMenuItem` abgeleitet sind. Es mag auf den ersten Blick merkwürdig erscheinen, dass `JMenu` eine Ableitung von `JMenuItem` ist und nicht umgekehrt (zuerst kommt doch das Menü, dann der Menübefehl). Dieses Prinzip bewirkt jedoch, dass ein `JMenu` mit `add`() einfach wieder in ein `JMenu` eingebunden werden kann und sich somit eine hierarchische Menüstruktur aufbauen lässt. Die `insert`() Methode fügt Menübefehle bzw. Menüs an bestimmten Stellen ein.

Die `addSeperator`() Methode legt Trennlinien zwischen die Menüeinträge. Der Programmierer wird von der Aufgabe entbunden, einen `JSeparator` zu instanzieren. `getItemCount`() ermittelt die Anzahl Menüeinträge (einschliesslich der Trennlinien).

7.4. JMenuItem

`JMenuItem` ist die Oberklasse aller Menüeinträge. Eine Instanz eines `JMenuItem` entspricht einem einfachen Menübefehl, der mit einem normalen Text versehen ist. Ihm obliegen aber die in `AbstractButton` angebotenen Dienstleitungen, und er kann entsprechend gestaltet werden.

Icon und Text übergibt man entweder im Konstruktor oder legt sie mit `setIcon`() und `setText`() fest. Die Ausrichtung des Textes bzw. des Icons erfolgen ebenfalls mit den Methoden von `AbstractButton`.

Tastenkürzel (Shortcuts) kann man mit `setAccelerator`() in einem Menüeintrag einbringen und erlauben den Benutzer, mit ihnen Menübefehle durch Drücken der entsprechenden Tastaturkombination, anzusteuern.

7.5. JCheckBoxItem

Wie man am Namen leicht erkennt, ist diese Klasse das Äquivalent zur `JCheckBox`, nur dass sie hier als Menüeintrag dient. Sie wird meist als Optionsschalter eingesetzt. Der Zustand kann bereits im Konstruktor oder mit `setState`() festgelegt werden. `getState`() liefert diesen wieder zurück. Was die Gestaltung angeht, sind dem Programmierer gegenüber `JCheckBox` keine Grenzen gesetzt. Wer mit dem gewöhnlichen Häkchen nicht zufrieden ist, greift auf die `setIcon` Methode zurück.

7.6. JRadioButtonMenuItem

Hier ist das Verhältnis zum `JRadioButton` dasselbe. Der Programmierer muss ebenfalls dafür sorgen, alle RadioButtons in einer `ButtonGroup` unterzubringen, die gegenseitig ausgeschlossen werden sollen.

7.7. JSeparator

Ein Separator ist eine Trennlinie zwischen den Menüeinträgen in Menüs und Werkzeugleisten (`JToolBar`). Er soll sich nach Möglichkeiten logisch trennen. Das Erzeugen einer Instanz von `JSeparator` ist in den meisten Fällen überflüssig. In Menüs und Werkzeugleisten nimmt `addSeparator`() diese Arbeit ab.

Listing 7-7-1 Verwendung von Menüs

```
package Menus;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;


public class Menus extends JPanel implements ActionListener
{
```

```
JMenuBar   menuBar;
JMenu      fileMenu;
JMenu      editMenu;
JMenu      optionMenu;
JMenu      toolBarMenu;
JMenu      fontMenu;
JMenu      fontSubMenu;
JMenu      helpMenu;
ButtonGroup fontSizeGroup;
JMenuItem miEditMenuItems[] = new JMenuItem[3];
JCheckBoxMenuItem miFontStyleItems[] = new JCheckBoxMenuItem[2];
JRadioButtonMenuItem  miFontSizeItems[] =
  new JRadioButtonMenuItem[5];
ImageIcon editIcons[] = new ImageIcon[3];

Menus(JApplet applet)
{
   setLayout(new BorderLayout());

  // Erzeugung einer Menueleiste
  menuBar = new JMenuBar();

  // Ein ganz gewöhliches File-Menue
  fileMenu = new JMenu("Datei");
  fileMenu.add(new JMenuItem("Neu"));
  fileMenu.add(new JMenuItem("Öffnen..."));
  fileMenu.add(new JMenuItem("Spiechern"));
  fileMenu.add(new JMenuItem("Speichern als..."));
  fileMenu.addSeparator();                  // Trennlinie
  JMenuItem item = new JMenuItem("Beenden");
  item.addActionListener(this);
  fileMenu.add(item);
  menuBar.add(fileMenu);


  // Ein Edit-Menue mit Images
  try
  {
    editIcons[0] = new ImageIcon("cut.gif");
    editIcons[1] = new ImageIcon("copy.gif");
    editIcons[2] = new ImageIcon("paste.gif");
  }
  catch(SecurityException se)      // Für Internet-Browser
  {
    editIcons[0] = new ImageIcon(
      applet.getImage(applet.getCodeBase(), "cut.gif"));
    editIcons[1] = new ImageIcon(
      applet.getImage(applet.getCodeBase(), "copy.gif"));
    editIcons[2] = new ImageIcon(
      applet.getImage(applet.getCodeBase(), "paste.gif"));
  }
  editMenu = new JMenu("Bearbeiten");
  editMenu.setMnemonic('B');
  miEditMenuItems[0] =
    new JMenuItem("Ausschneiden", editIcons[1]);
  miEditMenuItems[0].setMnemonic('A');
  miEditMenuItems[0].setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_X, ActionEvent.CTRL_MASK));
  miEditMenuItems[1] = new JMenuItem("Kopieren", editIcons[1]);
  miEditMenuItems[1].setMnemonic('K');
  miEditMenuItems[1].setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_C, ActionEvent.CTRL_MASK));
```

```
    miEditMenuItems[2] = new JMenuItem("Einfügen", editIcons[2]);
    miEditMenuItems[2].setMnemonic('E');
    miEditMenuItems[2].setAccelerator(
      KeyStroke.getKeyStroke(KeyEvent.VK_V, ActionEvent.CTRL_MASK));
    // Damit das Icon links erscheint
    miEditMenuItems[0].setHorizontalTextPosition(JButton.RIGHT);
    miEditMenuItems[1].setHorizontalTextPosition(JButton.RIGHT);
    miEditMenuItems[2].setHorizontalTextPosition(JButton.RIGHT);
    editMenu.add(miEditMenuItems[0]);
    editMenu.add(miEditMenuItems[1]);
    editMenu.add(miEditMenuItems[2]);
    menuBar.add(editMenu);


    // Ein Options-Menue mit Untermenues und Menueoptionen
    optionMenu = new JMenu("Optionen");

    // Toolbarmenu
    toolBarMenu = new JMenu("Werkezeugleisten");
    toolBarMenu.add(new JCheckBoxMenuItem("Standard", true));
    toolBarMenu.add(new JCheckBoxMenuItem("Statuszeile", true));
    toolBarMenu.add(new JCheckBoxMenuItem("Malleiste", true));
    toolBarMenu.add(new JCheckBoxMenuItem("Browserleiste", true));
    optionMenu.add(toolBarMenu);

    // Fontmenu
    fontMenu = new JMenu("Fonts");
    miFontStyleItems[0] = new JCheckBoxMenuItem("Fett", false);
    miFontStyleItems[1] = new JCheckBoxMenuItem("Kursiv", false);
    miFontStyleItems[0].setFont(
      new Font("Dialog", Font.BOLD, 12));
    miFontStyleItems[1].setFont(
      new Font("Dialog", Font.ITALIC, 12));
    fontMenu.add(miFontStyleItems[0]);
    fontMenu.add(miFontStyleItems[1]);
    fontMenu.addSeparator();                  // Trennlinie

    fontSizeGroup = new ButtonGroup();
    miFontSizeItems[0] = new JRadioButtonMenuItem("Sehr klein");
    miFontSizeItems[1] = new JRadioButtonMenuItem("Klein");
    miFontSizeItems[2] = new JRadioButtonMenuItem("Normal");
    miFontSizeItems[2].setSelected(true);
    miFontSizeItems[3] = new JRadioButtonMenuItem("Groß");
    miFontSizeItems[4] = new JRadioButtonMenuItem("Absolut groß");
    miFontSizeItems[0].setFont(new Font("Dialog", Font.PLAIN, 8));
    miFontSizeItems[1].setFont(new Font("Dialog", Font.PLAIN, 10));
    miFontSizeItems[2].setFont(new Font("Dialog", Font.PLAIN, 12));
    miFontSizeItems[3].setFont(new Font("Dialog", Font.PLAIN, 16));
    miFontSizeItems[4].setFont(new Font("Dialog", Font.PLAIN, 26));
    for (int i=0; i<5; i++)
    {
      fontMenu.add(miFontSizeItems[i]);
      fontSizeGroup.add(miFontSizeItems[i]);
    }

    optionMenu.add(fontMenu);
    menuBar.add(optionMenu);
    add("North", menuBar);
  }

  // Nür für den Menüpunkt Beenden
  public void actionPerformed(ActionEvent e)
```

```
  {
    System.exit(0);
  }
}
```

Abbildung 7-7-1 Menüs in Swing

7.8. Kontextmenü (JPopupMenu)

Mit dem `JPopupMenu` kann der Entwickler beliebige Objekte des Typs JComponent einem Kontextmenü zuordnen, welches mit der rechten Maustaste aufgerufen wird. Jede Swing Komponente kann mit individuellen Funktionen ausgestattet werden, auf die man mit Hilfe des Kontextmenüs Zugriff hat.

Die Programmierung ähnelt der von `JMenu`. Alle Komponenten, mit denen sich ein `JMenu` aufbauen lässt, funktioniert auch bei `JPopupMenu`. Die `addSeparator()` Methode wird ebenfalls unterstützt.

Listing 7-7-2 Verwendung von JPopupMenu

```
package PopupMenu;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;


public class PopupMenu extends JPanel
{
  JLabel         label1, label2;
  JPopupMenu     popupMenu1, popupMenu2;
  ImageIcon      icon;
  JMenuItem      menuItem;
  JCheckBoxMenuItem menuCheckBox;
  String   imageNames[] =
           {"duke.gif", "smile.gif", "new.gif", "open.gif",
            "save.gif", "delete.gif", "exit.gif", "toolbar.gif",
            "cut.gif", "copy.gif", "paste.gif", "properties.gif" };
  ImageIcon      images[] = new ImageIcon[12];
  MouseAdapter   ml;


  PopupMenu(JApplet applet)
  {
```

```
setLayout(new GridLayout(1, 2, 30, 20));

try
{
  for (int i=0; i<imageNames.length; i++)
    images[i] = new ImageIcon(imageNames[i]);
}
catch(SecurityException se)      // Für Internet-Browser
{
  for (int i=0; i<imageNames.length; i++)
    images[i] = new ImageIcon(
      applet.getImage(applet.getCodeBase(), imageNames[i]));
}

// MouseListener definieren
ml = new MouseAdapter(){
  public void mousePressed(MouseEvent e)
  {
    if (e.getModifiers() == InputEvent.BUTTON3_MASK)
      if (e.getSource() == label1)
        popupMenu1.show(label1, e.getX(), e.getY());
      else
        popupMenu2.show(label2, e.getX(), e.getY());
  }
};

// Label 1
add(label1 = new JLabel("Hier erscheint PopupMenu 1",
  images[0], JLabel.CENTER));
label1.setBorder(BorderFactory.createTitledBorder(
  new EtchedBorder(EtchedBorder.LOWERED)));
label1.setHorizontalAlignment(JLabel.CENTER);
label1.setVerticalTextPosition(JLabel.BOTTOM);
label1.setHorizontalTextPosition(JLabel.CENTER);
label1.addMouseListener(ml);

// Label 2
add(label2 = new JLabel("Hier erscheint PopupMenu 1",
  images[1], JLabel.CENTER));
label2.setBorder(BorderFactory.createTitledBorder(
  new EtchedBorder(EtchedBorder.LOWERED)));
label2.setHorizontalAlignment(JLabel.CENTER);
label2.setVerticalTextPosition(JLabel.BOTTOM);
label2.setHorizontalTextPosition(JLabel.CENTER);
label2.addMouseListener(ml);

// PopupMenu 1
popupMenu1 = new JPopupMenu("PopupMenu 1");
popupMenu1.setDefaultLightWeightPopupEnabled(true);
popupMenu1.setInvoker(label1);
popupMenu1.add(menuItem = new JMenuItem("Neu", images[2]));
menuItem.setHorizontalTextPosition(SwingConstants.RIGHT);
popupMenu1.add(menuItem =
  new JMenuItem("Öffnen...", images[3]));
menuItem.setHorizontalTextPosition(SwingConstants.RIGHT);
popupMenu1.addSeparator();
popupMenu1.add(menuItem =
  new JMenuItem("Speichern", images[4]));
menuItem.setHorizontalTextPosition(SwingConstants.RIGHT);
popupMenu1.add(menuItem =
  new JMenuItem("Speichern als...", images[4]));
menuItem.setHorizontalTextPosition(SwingConstants.RIGHT);
```

```
    popupMenu1.addSeparator();
    popupMenu1.add(menuItem = new JMenuItem("Delete", images[5]));
    menuItem.setHorizontalTextPosition(SwingConstants.RIGHT);
    popupMenu1.addSeparator();
    popupMenu1.add(menuItem = new JMenuItem("Beenden", images[6]));
    menuItem.setHorizontalTextPosition(SwingConstants.RIGHT);
    menuItem.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e)
      {
        System.exit(0);
      }
    });
    popupMenu1.addMouseListener(ml);

    // PopupMenu 2
    popupMenu2 = new JPopupMenu("PopupMenu 2");
    popupMenu2.setInvoker(label2);
    popupMenu2.add(menuCheckBox =
      new JCheckBoxMenuItem("Werkzeugleiste", images[7], true));
    menuCheckBox.setHorizontalTextPosition(SwingConstants.RIGHT);
    popupMenu2.addSeparator();
    popupMenu2.add(menuItem =
      new JMenuItem("Ausschneiden", images[8]));
    menuItem.setHorizontalTextPosition(SwingConstants.RIGHT);
    popupMenu2.add(menuItem =new JMenuItem("Kopieren", images[9]));
    menuItem.setHorizontalTextPosition(SwingConstants.RIGHT);
    popupMenu2.add(menuItem =
      new JMenuItem("Einfügen", images[10]));
    menuItem.setHorizontalTextPosition(SwingConstants.RIGHT);
    popupMenu2.addSeparator();
    popupMenu2.add(menuItem =
      new JMenuItem("Eigenschaften", images[11]));
    menuItem.setHorizontalTextPosition(SwingConstants.RIGHT);

    enableEvents(AWTEvent.COMPONENT_EVENT_MASK);
  }
}
```

Abbildung 7-7-2 JPopupMenu (rechte Maustaste aktiviert das Popup)

7.9. Werkzeugleisten (JToolbar)

Werkzeugleisten (die auch als Symbolleisten bekannt sind) enthalten bestimmte Befehle einer Anwendung. Meistens sind es Buttons, deren Funktion durch ein Icon symbolisiert werden. Beschriftete Buttons werden hier weniger eingesetzt, da sie zuviel Platz für sich beanspruchen. Aber auch andere Komponenten, wie Kombinationsfelder (z.B. für Suchbegriffe oder Fonts), finden sich öfters in einer Werkzeugleiste. Im Grunde handelt es sich um die am häufigsten verwendeten Funktionen. Der Anwender kann diese schneller erreichen als über das Menü. Es sollten wirklich nur die allerwichtigsten Befehle sein. Gelegentlich kommt es jedoch vor, dass bei funktionsreichen Anwendungen über das Ziel hinaus geschossen wird und die Symbolleisten einen grossen Teil des Bildschirms einnehmen.

Werkzeugleisten haben die Eigenschaft, überall plaziert werden zu können. Zieht man sie an eine Kante des Fensters, so richten sie sich entsprechend aus und docken sich an der Kante an. Werden sie dagegen mehr in die Fenstermitte gezogen, bekommen sie ihr eigenes Fenster. Ob sich eine Werkzeugleiste auch so flexibel handhaben lässt, hängt unter anderem vom eingestellten Layout Manager des Fensters ab. Nur das BorderLayout garantiert dafür, dass sie sich an jeder Kante plazieren lässt.

Entweder bei der Erzeugung oder mit `setOrientation`() kann man die Ausrichtung bestimmen. `JToolBar` versucht, sich entweder an der oberen oder linken Kante anzulegen. Alle Komponenten sind einfach mit `add`() einzufügen. Um sinnverwandte Buttons zu gruppieren, kann man sich dazwischen einen Abstand mit `addSeparator`() schaffen. Die Grösse des Abstandes ist frei definierbar. Einzelne Komponenten sind mit `getComponentAtIndex`() zu ermitteln. Manchmal ist eine starre Werkzeugleiste erwünscht, deren Position nicht verändert werden kann. Mit `setFloatable`(`false`) ist es nicht mehr möglich, sie an eine andere Stelle zu ziehen. Defaultmässig ist aber diese Eigenschaft auf `true` gesetzt.

`JToolBar` unterstützt selbst keine Listener. Man braucht jedoch nur darin enthaltenen Komponenten mit Listener auszustatten, um die Ereignisse verarbeiten zu können.

Listing 7-3 Verwendung von JToolBar

```
package Toolbar;
import java.awt.*;
import javax.swing.*;
public class Toolbar extends JPanel{
  JToolBar    toolBar1, toolBar2;
  JButton     button;
  String      toolBarNames1[] =
                {"Neu", "Öffnen", "Speichern", "Ausschneiden",
                 "Kopieren", "Einfügen"};
  String      toolBarNames2[] =
                {"Kompilieren", "Aufbauen", "Alles Aufbauen",
                 "Ausführen", "Debugger starten", "Breakpoint setzen"};
  String      iconNames1[] =
                {"new.gif", "open.gif", "save.gif", "cut.gif",
                 "copy.gif", "paste.gif"};
  String      iconNames2[] =
                {"compile.gif", "build.gif", "buildall.gif",
                 "execute.gif", "debug.gif", "breakpoint.gif"};
  ImageIcon   icons1[] = new ImageIcon[6];
```

```
  ImageIcon   icons2[] = new ImageIcon[6];
  Toolbar(JApplet applet){
    setLayout(new BorderLayout());
    try{
      for (int i=0; i<iconNames1.length; i++){
        icons1[i] = new ImageIcon(iconNames1[i]);
        icons2[i] = new ImageIcon(iconNames2[i]);
      }
    }// Für Internet-Browser
    catch(SecurityException se)     {
      for (int i=0; i<iconNames1.length; i++)      {
        icons1[i] = new ImageIcon(
          applet.getImage(applet.getCodeBase(), iconNames1[i]));
        icons2[i] = new ImageIcon(
          applet.getImage(applet.getCodeBase(), iconNames2[i]));
      }
    }
    toolBar1 = new JToolBar();
    toolBar1.setFloatable(true);
    for (int i=0; i<6; i++){
      button = new JButton(icons1[i]);      // Tooltips hinzufuegen
      button.setToolTipText(toolBarNames1[i]);
      button.setActionCommand(toolBarNames1[i]);
      toolBar1.add(button);
      if (i==2)
        toolBar1.addSeparator();
    }
    toolBar2 = new JToolBar();
    toolBar2.setFloatable(true);
    for (int i=0; i<6; i++){
      button = new JButton(icons2[i]);
      // Tooltips hinzufuegen
      button.setToolTipText(toolBarNames2[i]);
      button.setActionCommand(toolBarNames2[i]);
      toolBar2.add(button);
      if (i==2)
        toolBar2.addSeparator();
    }
    add("North",toolBar1); // Oben darstellen
    add("South", toolBar2); // Oben darstellen
  }
}
```

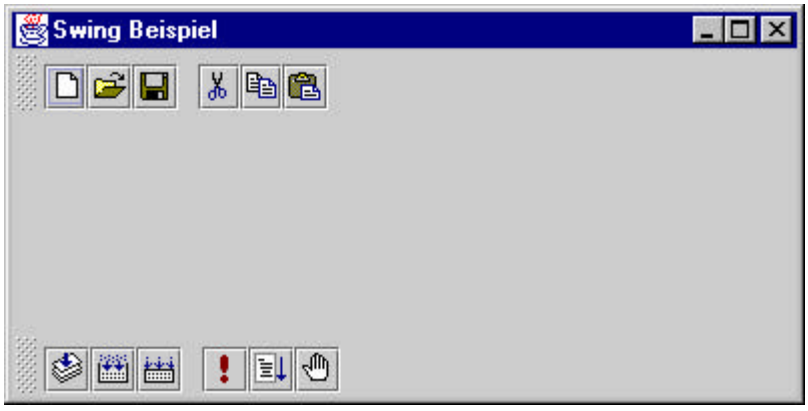
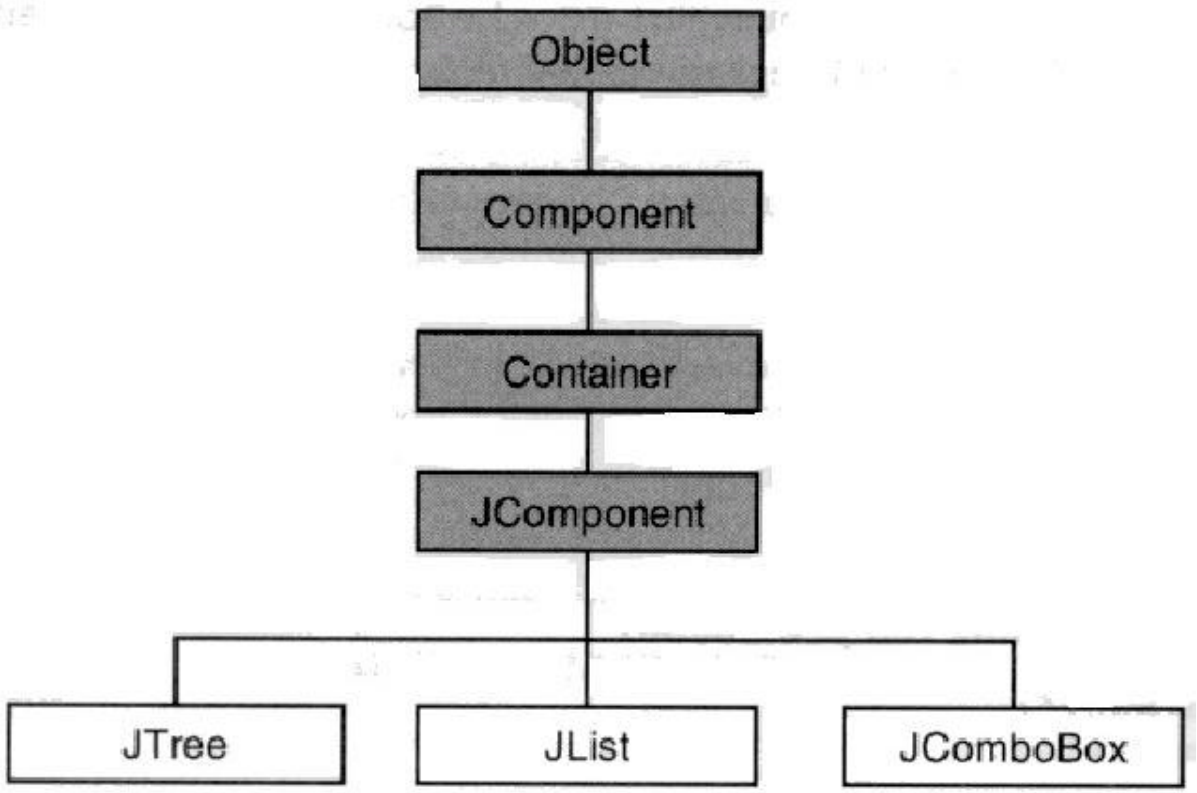


Abbildung 7-3 JToolBar

In diesem Kapitel
- *JComboBox*
- *JList*

8. Auswahlmöglichkeiten



Die obige Skizze, aus der Java Dokumentation, zeigt die Auswahlmöglichkeiten, welche von Swing unterstützt werden.

8.1. JComboBox

Das Kombinationsfeld `JComboBox` entspricht einem Eingabefeld, welches ein Menü von vordefinierten Werten im PopUp Prinzip anbietet. Geeignet ist es für die Auswahl eines einzigen Wertes. Soll es dem Benutzer erlaubt sein, beliebig viele Einträge zu wählen, ist die Liste (`JListe`) eine bessere Alternative. Das Element ist wegen seiner platzsparenden Eigenschaften besonders beliebt.

Das Kombinationsfeld von Swing funktioniert wie ihre Verwandte `Choice` aus den AWT Klassen. Der hauptsächliche Unterschied liegt in der optionalen Editierbarkeit und der Anwendung der MVC Architektur (Model-View-Controller Design Pattern). `JComboBox` reagiert auf Eingaben der Tastatur und versucht, die Eingaben des Benutzers mit einem passenden Listeneintrag zu ergänzen, ihre Editierbarkeit vorausgesetzt.

Sind die Listeneinträge von vornherein bekannt, legt man diese in einem Array ab und übergibt es dem Konstruktor. Neue Einträge lassen sich mit `addItem`() anfügen.

Den gewählten Eintrag gibt `getSelectedItem`() zurück. Interessiert man sich nur für seinen Index, so bekommt man ihn mit `getSelectedIndex`(). Umgekehrt bestimmt `setSelectedItem`() bzw. `setSelectedIndex`() ein Listenelement zum gewählten Eintrag. `setMaximumRowCount`() bestimmt die grösstmögliche Anzahl von Einträgen, die in der PopUp Liste sichtbar sein sollen. Unterliegt diese der Anzahl aller Einträge des Kombinationsfeldes, so verwendet es einen Rollbalken. Die `setEditable`() Methode legt fest, ob Tastatureingaben möglich sind oder nicht. Der Umfang der Einträge kann mit `getItemCount`() abgefragt werden. Wer sich ein bestimmtes Element aus dem Kombinationsfeld holen möchte, kann auf `getItemAt`() zurück greifen. Der Index des Eintrags ist zu übergeben.

Neben dem ActionListener unterstützt `JComboBox` den `ItemListener`. Er lässt sich mit `addItemListener`() registrieren. Wird das Ereignis `ItemEvent` gesendet, ruft der Listener itemStateChanged() auf.

Zwar ist für die Listeneinträge der Datentyp `Object` erlaubt - d.h. man könnte theoretisch jedes beliebige Kontrollelement (zum Beispiel Icons, Buttons, ...) aufnehmen - das Kombinationsfeld macht aber in seiner grundlegenden Form nur einen Aufruf der `toString`() Methode der einzelnen Listenelemente und stellt das Ergebnis in der PopUp Liste dar. Bei fast allen Komponenten ausser `JLabel` und String ist diese Vorgehensweise wenig sinnvoll. Die Verarbeitung anderer Komponenten bezieht sich auf das Thema der MVC Architektur und wird später noch genauer bearbeitet.

Listing 8-8-1 Einfaches Anwendungsbeispiel für JComboBox

```
package ComboBoxes;
import java.awt.*;
import javax.swing.*;

public class ComboBoxes extends JPanel{
  JComboBox  simpleComboBox, editComboBox;
  String    simpleNames[] =
    {"Architektur", "Bauingenieurwesen", "Elektrotechnik",
     "Informatik", "Mathematik", "Maschinenbau",
     "Mikrosystemtechnik", "Sozialwesen"};
  String    editNames[] = {"Zürich", "Bern", "Luzern"};

  ComboBoxes(JApplet applet) {
    setLayout(new GridLayout(3, 1, 0, 5));

    add(new JLabel("Studiengänge"));

    // Editierbare Combobox
    editComboBox = new JComboBox();
    // Editierbarkeit einschalten
    editComboBox.setEditable(true);
    // Namensliste aufbauen
    for (int i=0; i<editNames.length; i++)
      editComboBox.addItem(editNames[i]);
    add(editComboBox);

    // Beispiel für einfache Comboboxen
    simpleComboBox = new JComboBox();
    // Editierbarkeit ausschalten
    simpleComboBox.setEditable(false);
    // Namensliste aufbauen
    for (int i=0; i<simpleNames.length; i++)
```

Abbildung 8-8-1

```
      simpleComboBox.addItem(simpleNames[i]);
    add(simpleComboBox);
  }
}
```

8.2. JList

Das Listenfeld gehört wie das Kombinationsfeld ebenfalls zu den bekanntesten und verbreitetsten Auswahlmöglichkeiten. Im Gegensatz zum Kombinationsfeld werden die Listeneinträge sofort angezeigt, was natürlich mehr Platz einnimmt. Weiterhin steht es dem Benutzer frei, mehr als nur einen Eintrag gleichzeitig auszuwählen. Er ist jedoch auf das Angebot des Listenfeldes beschränkt. Neue Werte einzugeben, wie dies beim editierbaren Kombinationsfeld der Fall ist, wird dem Benutzer nicht erlaubt.

Der Inhalt der `JList` ist entweder im Konstruktor oder mit `setListData`() zu definieren.

Sie unterstützt drei verschiedene Arten der Selektionierung von Einträgen. Vielen Benutzern sind sie vom Windows Explorer bekannt. Der Programmierer kann mit Hilfe von `setSelectionMode`() dem Anwender drei Selektionsarten vorschreiben:

- `SINGLE_SELECTION` erlaubt nur einen einzigen Listeneintrag
- `SINGLE_INTERVAL_SELECTION` erweitert die Auswahl auf ein ganzes Intervall, unter Verwendung der Shift Taste.
- `MULTIPLE_INTERVAL_SELECTION` beschränkt sich nicht nur auf ein Intervall, sondern gibt dem Benutzer die Möglichkeit, jeden beliebigen Eintrag der Liste auszuwählen, solange er die Strg Taste gedrückt hält.

Will man feststellen, ob der Anwender überhaupt eine Wahl getroffen hat, prüft man einfach den Rückgabewert von `isSelectionEmpty`() auf `false`. Trifft dies zu, so kann das gewählte Element mit `getSelectedValue`() abgefragt werden. Handelt es sich um mehrere Elemente, sind diese mit `getSelectedValues`() zu ermitteln. Die Indizes der selektierten Elemente liefern `getSelectedIndex`() und `getSelectedIndices`().

Wurde die `JList` in ein Rollfeld mittels `JScrollPane` eingebettet, bestimmt `setVisibleRowCount`() die Anzahl der Listenelemente, die ohne Rollbalken sichtbar sein sollen. Es bewirkt somit die Grösse des rollbaren Viewports.

Um das Layout ein wenig zu verändern, kann man mit `setFixedCellWidth`() und `setFixedCellHeight`() die Breite bzw. Höhe eines Listenelementes vorgeben. Die Vordergrund- und Hintergrundfarbe gewählter Elemente ist mit `setSelectionForeground`() und `setSelectionBackground`() modifizierbar.

Listing 8-2 Verwendung von Jlist

```
package ListBox;

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


public class ListBox extends JPanel implements ActionListener
```

```
{
  JPanel       modePanel;
  JList        textList;
  JScrollPane  scrollPane;
  ButtonGroup  modeGroup;
  JRadioButton rbutton;
  String    streetNames[] =
  {"Ahornstr.", "Am Hopfengarten", "Am Klostergrund", "Bahnhofstr.",
   "Bergstr.", "Bruckdorfer Str.", "Buchenstr.", "Dahlienweg",
   "Donaustr.", "Enzianstr.", "Erikastr.", "Fährenweg", "Flurweg",
   "Heckenweg", "Hochweg", "Industriestr.", "Kirchweg", "Laberstr.",
   "Lilienstr.", "Margaretenstr.", "Mariaorter Str.",
   "Minoritenweg", "Nelkenstr.", "Rosenweg", "Sonnenstr.",
   "Tulpenstr.", "Vogelsanger Str."};

  ListBox(JApplet applet)
  {
    setLayout(new BorderLayout());

    // Einfache ListBox mit Text
    textList = new JList(streetNames);
    textList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    scrollPane = new JScrollPane(textList);
    add("Center", scrollPane);

    // Optionsschalter für SelectionMode
    modePanel = createTitledPanel("SelectionMode");
    modeGroup = new ButtonGroup();
    modePanel.add(rbutton =
      new JRadioButton("SINGLE_SELECTION"));
    modeGroup.add(rbutton);
    rbutton.addActionListener(this);
    rbutton.setSelected(true);
    modePanel.add(rbutton =
      new JRadioButton("SINGLE_INTERVAL_SELECTION"));
    modeGroup.add(rbutton);
    rbutton.addActionListener(this);
    modePanel.add(rbutton =
      new JRadioButton("MULTIPLE_INTERVAL_SELECTION"));
    modeGroup.add(rbutton);
    rbutton.addActionListener(this);
    add("North", modePanel);
  }

  // Erstellt ein Panel mit Rahmen
  public static JPanel createTitledPanel(String title)
  {
    JPanel p = new JPanel();
    p.setLayout(new GridLayout(0, 1, 2, 2));
    p.setBorder(BorderFactory.createTitledBorder(
      new EtchedBorder(), title, 0, 0,
      new Font("Dialog", Font.BOLD, 12)));
    return p;
  }

  public void actionPerformed(ActionEvent e)
  {
    if (e.getActionCommand() == "SINGLE_SELECTION")
      textList.setSelectionMode(
        ListSelectionModel.SINGLE_SELECTION);
    if (e.getActionCommand() == "SINGLE_INTERVAL_SELECTION")
      textList.setSelectionMode(
```

```
            ListSelectionModel.SINGLE_INTERVAL_SELECTION);
        if (e.getActionCommand() == "MULTIPLE_INTERVAL_SELECTION")
          textList.setSelectionMode(
            ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
    }
}
```

Das Rahmenprogramm ist auch in diesem Falle identisch mit jenem der früheren Beispiele. Es ist hier nicht abgebildet.

Dagegen wollen wir uns den Screen Snapshot ansehen (das Programm befindet sich auf dem Server) :

Abbildung 8-2 JList mit Multiple Intervall Selection

In diesem Kapitel

- *JScrollBar*
- *JProgressBar*
- *JSlider*

9. Analoge Komponenten

Der Begriff "Analoge Komponenten" ist eigentlich nicht ganz zutreffend, da in der Welt der Rechner alles auf digitaler Basis läuft. Dennoch wird der Eindruck erweckt, als können man mit ihnen Einstellungen stufenlos vornehmen. Tatsächlich kann ihr Raster fast beliebig fein gehalten werden, so dass ein praktisch analoges Verhalten vorliegt.

9.1. JScrollBar

Ein Scrollbar `JScrollBar` vertritt die Rollbalken unter den analogen Komponenten. Sie dienen weniger dazu, Zahlenwerte einzugeben, sondern vielmehr, den sichtbaren Bereich eines Dokuments, dessen Ausmasse die Bildschirmgrösse übertrifft, zu bestimmen. Im Idealfall entspricht das Grössenverhältnis vom Rollbalken zum Einstellknopf dem vom Dokument zum sichtbaren Bereich. Allerdings ist es in vielen Fällen nicht notwendig, ein `JScrollBar` explizit zu erzeugen. Rollfelder werden in der Regel mit `JScrollPane` angelegt, welches sich um die Rollbalken kümmert.

Ob es entweder ein horizontaler oder ein vertikaler Rollbalken werden soll,, bestimmt `setOrientation`(). Den augenblicklichen Wert setzt man mit `setValue`() und ermittelt ihn mit `getValue`(). Mit Hilfe von `setVisibleAmount`() definiert man die Grösse des Einstellknopfs. Schliesslich kann man die obere und untere Grenze vergeben:

`setMinimum`(), `setMaximum`().

Alle Vorgaben können auch von vornherein im Konstruktor angegeben werden. Ohne Angaben belegt der Konstruktor die Vorgaben mit
Defaultwerten. Auch kann man sie alle auf einmal mit `setValues`() in der beschriebenen Reihenfolge setzen.
Bei jeder Änderung des Wertes wird ein `ChangeEvent` erzeugt, das mit einem `ChangeListener` verarbeitet werden kann.
Die direkte Verwendung von `JScrollBar` kommt jedoch selten vor, so dass auf ein Beispiel verzichtet wird. In den meisten Fällen greift man auf `JScrollPane` zurück.

9.2. JSlider

Slider (auch als Schieberegler bekannt) sind den normalen Rollbalken sehr ähnlich. Sie geben dem Entwickler die Möglichkeit, Marken, Beschriftungen und Rahmen anzubringen. Ihr Aussehen erinnert an den Regler eines Eisenbahntrafos oder an einen Thermostat. Er ist daher besonders geeignet, Zahlen oder Wertebereiche einzugeben.
Dem Konstruktor können wahlweise Orientierungen (horizontal oder vertikal), der momentane Wert, sowie untere und obere Grenze übergeben werden. Keiner dieser Werte ist jedoch zwingend vorgeschrieben. Defaultmässig erzeugt der Konstruktor einen horizontal angelegten Slider mit einem Wertebereich zwischen 0 und 100 und zeigt auf den Zahlenwert 50. Die Einstellung der Werte kann auch entsprechend mit den Methoden `setOrientation`(), `setMinimum`(), `setMaximum`() und `setValue`() erfolgen.
Was einen Slider noch realistischer aussehen lässt, sind Markierungen, dargestellt als Striche. Dabei unterscheidet man zwischen Haupt- und Untermarkierungen. Die Hauptmarkierungen sind länger als die Untermarkierungen. Man aktiviert sie einfach mit `setPaintTicks`(). Verantwortlich für die Abstände zwischen ihnen sind die Methoden `setMajorTickSpacing`() und `setMinorSpacing`(). Soll die Bewegung gerastert werden, so dass sie genau auf den Markierungen verläuft, ruft man `setSnapToTicks`() auf.
Die Besonderheit liegt in der Beschriftung. Man aktiviert sie mit `setPaintLabel`(). Swing regelt die Beschriftung selbst - nicht immer ganz zufriedenstellend - möglicherweise sollen ja bestimmte Zahlen (zum Beispiel Grenzwerte) ständig angezeigt bleiben, eventuell mit einem speziellen Font, und ein Icon könnte die Bedeutung des Grenzwertes besonders unterstreichen. Eine benutzerdefinierte Beschriftung ist für den `JSlider` kein Problem. Sie wird mit `setLabelTable`() angelegt. Die Methode benötigt ein `Dictionnary`. Ein `Dictionnary` ist eine Art Hash-Tabelle, die alle Arten von Objekten aufnehmen kann, also auch Icons.
Um Bewegungen und Änderungen des Slider Wertes festzustellen, sendet `JSlider` jeweils ein `ChangeEvent` mit. Mit dem `ChangeListener` lässt es sich weiterverarbeiten.

Listing 9-1 Verwendung von JSlider

```
package Sliders;

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


public class Sliders extends JPanel implements
  ActionListener, ChangeListener
{
  JPanel      panel;
  JSlider     simpleSlider, tickSlider, snapSlider;
  JSlider     labelSlider, customSlider;
  Dictionary  labelTable;
  ImageIcon   icon[] = new ImageIcon[2];
  JLabel      label;
  JCheckBox   invertedCheck;
  JLabel      sliderLabel;
```

```
Sliders(JApplet applet)
{
  setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));

  // Einfacher Slider
  simpleSlider = new JSlider(JSlider.HORIZONTAL, 0, 100, 30);
  simpleSlider.setBorder(BorderFactory.createTitledBorder(
    new EtchedBorder(), "Einfacher Slider", 0, 0,
    new Font("Dialog", Font.BOLD, 12)));
  simpleSlider.addChangeListener(this);
  add(simpleSlider);

  // Mit Marken
  tickSlider = new JSlider(JSlider.HORIZONTAL, 0, 100, 70);
  tickSlider.setBorder(BorderFactory.createTitledBorder(
    new EtchedBorder(), "Mit Marken", 0, 0,
    new Font("Dialog", Font.BOLD, 12)));
  tickSlider.addChangeListener(this);
  tickSlider.setPaintTicks(true);
  tickSlider.setMajorTickSpacing(10);
  tickSlider.setMinorTickSpacing(5);
  add(tickSlider);

  // Mit Rasterung
  snapSlider = new JSlider(JSlider.HORIZONTAL, 0, 10, 5);
  snapSlider.setBorder(BorderFactory.createTitledBorder(
    new EtchedBorder(), "Mit Rasterung", 0, 0,
    new Font("Dialog", Font.BOLD, 12)));
  snapSlider.addChangeListener(this);
  snapSlider.setPaintTicks(true);
  snapSlider.setMajorTickSpacing(1);
  snapSlider.setSnapToTicks(true);
  add(snapSlider);

  // Mit Beschriftung
  labelSlider = new JSlider(JSlider.HORIZONTAL, 0, 100, 80);
  labelSlider.setBorder(BorderFactory.createTitledBorder(
    new EtchedBorder(), "Mit Beschriftung und Icons", 0, 0,
    new Font("Dialog", Font.BOLD, 12)));
  labelSlider.addChangeListener(this);
  labelSlider.setPaintTicks(true);
  labelSlider.setMajorTickSpacing(10);
  labelSlider.setMinorTickSpacing(1);
  labelSlider.setSnapToTicks(true);
  labelSlider.setPaintLabels(true);
  add(labelSlider);

  // Mit Spezialbeschriftung
  try
  {
    icon[0] = new ImageIcon("ok.gif");
    icon[1] = new ImageIcon("danger.gif");
  }
  catch(SecurityException se)      // Für Internet-Browser
  {
    icon[0] = new ImageIcon(
      applet.getImage(applet.getCodeBase(), "ok.gif"));
    icon[1] = new ImageIcon(
      applet.getImage(applet.getCodeBase(), "danger.gif"));
  }
  customSlider = new JSlider(JSlider.HORIZONTAL, 0, 10, 2);
```

```
    customSlider.setBorder(BorderFactory.createTitledBorder(
      new EtchedBorder(), "Mit Beschriftung", 0, 0,
      new Font("Dialog", Font.BOLD, 12)));
    customSlider.addChangeListener(this);
    customSlider.setPaintTicks(true);
    customSlider.setMajorTickSpacing(5);
    customSlider.setMinorTickSpacing(1);
    customSlider.setSnapToTicks(true);
    customSlider.setPaintLabels(true);
    labelTable = customSlider.getLabelTable();
    labelTable.put(new Integer(3), label =
      new JLabel("3 optimal", icon[0], JLabel.CENTER));
    labelTable.put(new Integer(8), label =
      new JLabel("8 Gefahr", icon[1], JLabel.CENTER));
    customSlider.setLabelTable(labelTable);
    add(customSlider);

    // Option: Invertiert
    add(panel = new JPanel());
    panel.setLayout(new GridLayout(0, 1, 10, 5));
    invertedCheck = new JCheckBox("Invertiert");
    invertedCheck.addActionListener(this);
    panel.add(invertedCheck);

    // Sliderwert
    sliderLabel = new JLabel("Sliderwert: " + 0);
    panel.add(sliderLabel);
  }


  // Erstellt ein Panel mit Rahmen
  public static JPanel createTitledPanel(String title)
  {
    JPanel p = new JPanel();
    p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
    p.setBorder(BorderFactory.createTitledBorder(
      new EtchedBorder(), title, 0, 0,
      new Font("Dialog", Font.BOLD, 12)));
    return p;
  }


  public void actionPerformed(ActionEvent e)
  {
    if (e.getActionCommand() == "Invertiert")
    {
      if (invertedCheck.isSelected())
      {
        simpleSlider.setInverted(true);
        tickSlider.setInverted(true);
        snapSlider.setInverted(true);
        labelSlider.setInverted(true);
        customSlider.setInverted(true);
      }
      else
      {
        simpleSlider.setInverted(false);
        tickSlider.setInverted(false);
        snapSlider.setInverted(false);
        labelSlider.setInverted(false);
        customSlider.setInverted(false);
      }
```

```
      repaint();
    }

  }

  public void stateChanged(ChangeEvent e)
  {
    JSlider s;
    s = (JSlider)e.getSource();
    sliderLabel.setText("Sliderwert: " + s.getValue());
  }
}

package Sliders;

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


public class SwingBeispiel extends JApplet
{
  public void init()
  {
    getContentPane().add(new Sliders(this));
  }

  // Erstellt ein Panel mit Rahmen
  public static void main(String[] args)
  {
    Frame frame = new Frame("SwingBeispiel");

    WindowListener l = new WindowAdapter() {
        public void windowClosing(WindowEvent e) {System.exit(0);}
    };
    frame.addWindowListener(l);

    SwingBeispiel applet = new SwingBeispiel();
    frame.add("Center", applet);
    applet.init();
    frame.setSize(520, 420);
    frame.show();
  }
}
```

Dabei verwenden wir den bereits bekannten Rahmen. Das obige Programm zeigt einfach einige der vielen möglichen Darstellungen eines Sliders.

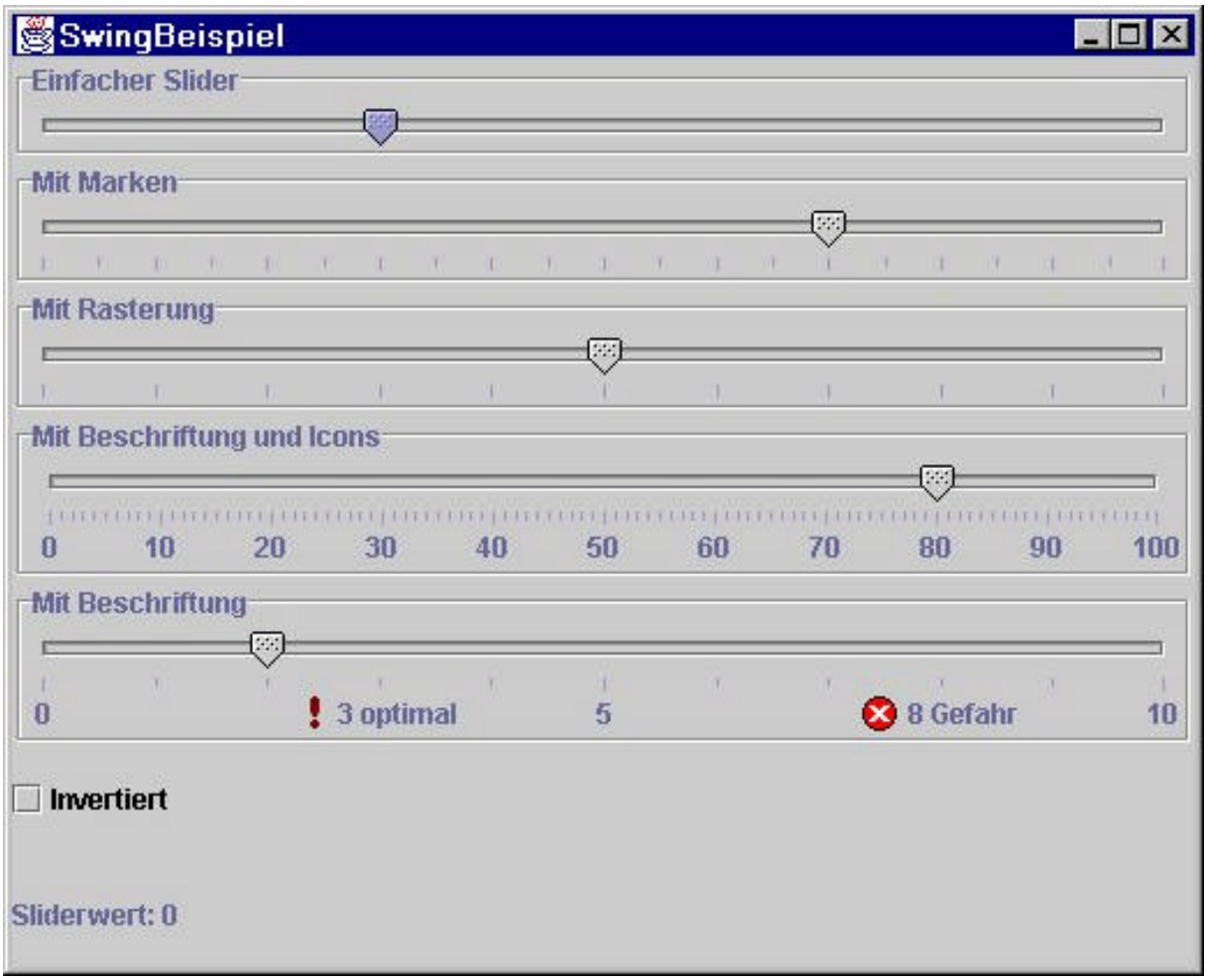


Abbildung 9-1 JSlider

9.3. JProgressBar

Fortschrittsanzeigen sind besonders bei längeren Arbeiten eines Programms nützlich. Sie informieren den Benutzer, wie viele Arbeitsschritte oder wie weit eine Operation bereits fortgeschritten ist. Auf diese Weise lässt sich die gesamte Abarbeitungszeit leichter einschätzen, und darüber hinaus hat der Benutzer die Gewissheit, dass seine Anwendung von einem Absturz verschont geblieben ist.

Das Anlegen eines `JProgressBar` gleicht fast dem des `JSlider`. Wieder steht es dem Programmierer frei, ob er die Orientierung und die beiden Extremwerte im Konstruktor oder mit `setOrientation`(), `setMinimum`() und `setMaximum`() definiert. Die Startposition kann im Konstruktor nicht angegeben werden. Sie ist defaultmässig 0. Die aktuelle Position ist aber jederzeit mit `setValue`() änderbar.

Mit `setStringPainted`() zeigt die Fortschrittsanzeige im Balken den aktuellen Wert als Prozentzahl an.

Zwar wird bei jeder Änderung ebenfalls ein `ChangeEvent` ausgelöst, doch die Verarbeitung

durch einen `ChangeListener` ist weniger interessant, da eine Fortschrittsanzeige nicht vom Anwender beeinflusst werden kann.

Listing 9-2 Verwendung von JProgressBar

```
package ProgressBar;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class ProgressBar extends JPanel implements ActionListener {
  JPanel        panel;
  JProgressBar  testProgressBar;
  JButton       lowerButton, higherButton;

  ProgressBar(JApplet applet) {
    setBorder(BorderFactory.createTitledBorder(
      new EtchedBorder(), "Einfache ProgressBar", 0, 0,
      new Font("Dialog", Font.BOLD, 12)));
    setLayout(new BorderLayout());
    testProgressBar = new JProgressBar();
    testProgressBar.setBounds(10, 10, 200, 30);
    testProgressBar.setMinimum(0);
    testProgressBar.setMaximum(10);
    testProgressBar.setValue(0);
    testProgressBar.setStringPainted(true);
    add("North", testProgressBar);

    panel = new JPanel();
    panel.setLayout(new FlowLayout(FlowLayout.CENTER));
    lowerButton = new JButton("Niedriger");
    lowerButton.setBounds(240, 10, 100, 30);
    lowerButton.addActionListener((ActionListener)this);
    higherButton = new JButton("Höher");
    higherButton.setBounds(360, 10, 100, 30);
    higherButton.addActionListener((ActionListener)this);
    panel.add(lowerButton);
    panel.add(higherButton);
    add("South", panel);
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand() == "Höher")
      testProgressBar.setValue(testProgressBar.getValue() + 1);
    else
      testProgressBar.setValue(testProgressBar.getValue() - 1);
  }
}


package ProgressBar;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SwingBeispiel extends JApplet {
  public void init()   {
    getContentPane().add(new ProgressBar(this));
  }
```

```
  // Erstellt ein Panel mit Rahmen
  public static void main(String[] args)  {
    Frame frame = new Frame("Swing Beispiel");

    WindowListener l = new WindowAdapter() {
        public void windowClosing(WindowEvent e) {System.exit(0);}
    };
    frame.addWindowListener(l);

    SwingBeispiel applet = new SwingBeispiel();
    frame.add("Center", applet);
    applet.init();
    frame.pack();
    frame.show();
  }

}
```

Soweit das Programm (mit dem normalen Swing Beispiel Rahmen gestartet).
Und so sieht dieses Beispiel grafisch aus:

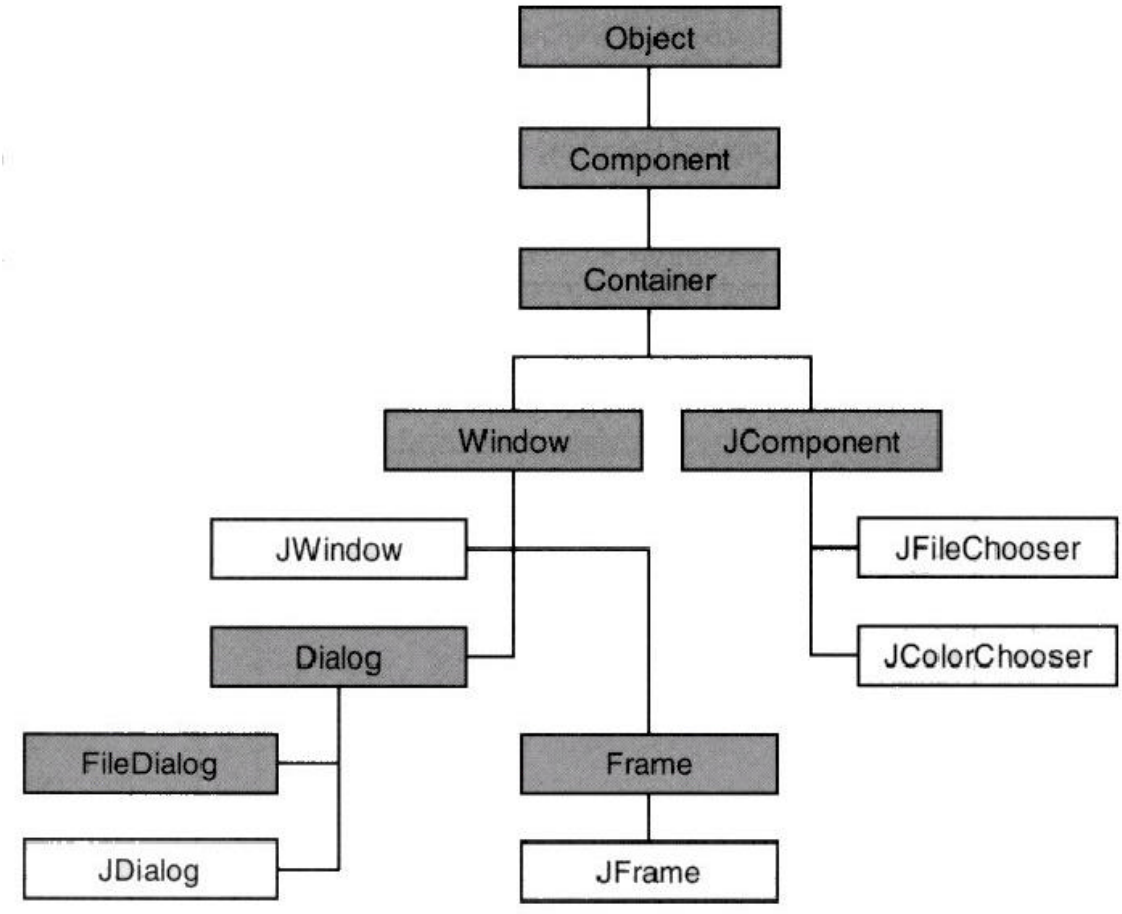
Abbildung 9-2 JProgressBar

Soweit analoge Komponenten, also Slider, Progressbar und deren Swing Implementation.

In diesem Kapitel

- *RootPane (JRootPane)*
  - *GlassPane*
  - *LayeredPane (JLayeredPane)*
  - *ContentPane*
  - *optionale Menüleiste (JMenuBar)*
- *JFrame*
- *Internal Frame (JinternalFrame & JDesktopPane)*
- *JDialog*
- *JOptionPane*
- *Spezialdialoge*
  - *JFileChooser*
  - *JColorChooser*

10. *Fenster und Dialoge*



Swing-Fenster gehören zur Gruppe der schwergewichtigen Komponenten und stellen somit eine Ausnahmeerscheinung von Swing dar. Wie im AWT weisen solche Komponenten nur *Peers* auf, die eine Schnittstelle zum Betriebsystem darstellen und keine eigenen Routinen besitzen. Deshalb nehmen Swing-Fenster das Aussehen des zugrundeliegenden Betriebsystems an. Auch das neue Pluggable Look & Feel hat keinen Einfluss darauf.

Die Klassen `JFrame, JApplet, JWindow` und `JDialog` gehören den Fenstern an. Sie unterscheiden sich zwar alle in vielen ihrer Eigenschaften, haben jedoch miteinander gemeinsam, dass sie alle nur eine `RootPane` (siehe unten) besitzen. Diese nimmt jedoch keine Komponenten auf. Die `add`() Methode funktioniert nicht so, wie man es beim AWT gewohnt war. Die `ContentPane` (siehe unten) ist normalerweise für die Aufnahme der Komponenten zuständig. Sie muss zuvor mit `getContentPane`() ermittelt werden, was sich auf die Programmierung folgendermassen auswirkt:

```
JFrame swingFrame = new JFrame();
JButton swingButton = new JButton("Test");
swingFrame.getContentPane().add(swingButton);
```

10.1. RootPane (JRootPane

Klassen vom Typ `JRootPane` werden fast nie selbst instanziert. Sie sind eher Bestandteil der schwergewichtigen Swing-Fenster `JFrame, JWindow, JDialog` und `JApplet`. Die `RootPane` besteht ihrerseits aus einem `GlassPane` (`JGlassPane`) und einem `LayeredPane (JLayeredPane`). Folgendes Diagramm verdeutlicht den Aufbau:

Abbildung 10-1 RootPane

Der `LayeredPane` kümmert sich um die Menüleiste (falls erforderlich) und um die `ContentPane`. Die vier Elemente haben folgende Funktionen:

10.1.1. GlassPane

Die `GlassPane` liegt über allen Elementen der `RootPane` und ist im Normalfall völlig transparent. Sie fängt Ereignisse ab, um sie der `RootPane` für die Weiterverarbeitung zu

übergeben. Nur weil sie von Grund auf unsichtbar ist, heisst das nicht, dass sie keine Komponenten darstellen könnte. Das PopUp Menu (`JPopupMenu`) und die Werkzeugkiste (`JToolBar`) bedienen sich ihrer, um sich über alle Komponenten einzublenden. Es kann auch nach Bedarf die `paint`() Methode für Zeichnungen in der `GlassPane` überlagert werden.

10.1.2. LayeredPane (JLayeredPane)

Die `LayeredPane` besteht aus der `ContentPane` und einer optionalen Menüliste. Sie bestimmt die Position, Grösse und Tiefe des Bereichs der beiden Elemente. Um Komponenten in ene `LayeredPane` einzubinden, ist stets die Tiefe mit anzugeben. Es können beliebig viele `LayeredPanes` an verschiedenen Positionen übereinandergelegt werden. Die Klasse `JDesktopPane` ist ein Spezialfall und wird für Internal Frames verwendet (folgt gleich).

Mit `add`() teilt man eine Komponente einem bestimmten Layer zu. Dabei gibt man die Komponente und einen Wert an, der die Tiefe des Layers enthält. Er ist vom Typ Integer.

```
layeredPane.add(new JButton(), new Integer(10) );
```

Eigentlich ist der Tiefenwert willkürlich. Die Klasse JLayeredPane bietet aber zur Vereinfachung fest vordefinierte Konstanten an, die jeweils auf einem Layer mit einer speziellen Bedeutung hinweisen. Bei ihrer Verwendung kann man sichergehen, dass die Komponenten richtig überlappen, und braucht sich nicht um die Wahl geeigneter Tiefen zu sorgen.

- `DEFAULT_LAYER`
  sozusagen der Standard-Layer. Er enthält gewöhnlich die meisten Komponenten und ist auf der untersten Ebene.
- `PALETTE_LAYER`
  liegt über dem Default-Layer. Er findet vor allem für Werkzeugkisten und Farbpaletten Verwendung, die man deshalb auch über andere Komponenten einblenden kann, ohne die darunterliegenden zu beeinflussen.
- `MODAL_LAYER`
  modale Dialoge (siehe `JOptionPane`) greifen auf den Modal-Layer zurück. Seine Komponenten liegen über allen Standard-Komponenten und Werkzeugleisten.
- `POPUP_LAYER`
  dieser Layer ist reserviert für Komponenten, die ein PopUp-Fenster für sich beansprucht. Zum Beispiel Kombinationsfelder oder Tooltips müssen an der obersten Stelle erscheinen und dürfen nicht durch andere Komponenten verdeckt werden.
- `DRAG_LAYER`
  während der Benutzer das Objekt mit dem Mausklick bewegt, verharrt es im Drag-Layer und darf unter keinen Umständen überlagert werden. Lässt der Benutzer das Objekt los, fällt es wieder zum normalen Layer zurück.

```
layeredPane.add(childComponent, JLayeredPane.DEFAULT_LAYER);
```

Dieses Prinzip kommt gerade bei den Internal Frames für MDI Anwendungen zur Geltung. Dort sollte man aber eher laufende Zahlen verwenden.

```
JInternalFrame intFrame1 = new JInternalFrame();
JInternalFrame intFrame1 = new JInternalFrame();
```

```
layeredPane.add(intFrame1, new Integer(11));
layeredPane.add(intFrame1, new Integer(11));
```

Die `setLayer`() Methode legt eine Komponente in einen bestimmten Layer. `getComponentLayer`() ermittelt die Zahl der Komponenten des angegebenen Layers.

10.1.3. ContentPane

Die `ContentPane` ist der eigentliche Träger aller sichtbaren Komponenten ausser der Menüleiste. Dort werden sie auch mit `add`() eingetragen. Einstellungen des `LayoutManagers` sind ebenfalls in der `ContentPane` zu tätigen. Ermittelt wird sie immer mit `getContentPane`().

10.1.4. Optionale Menüleiste (JMenuBar)

Die `ContentPane` teilt sich die `LayeredPane` mit der Menüleiste, falls sie angefordert wird. Sie ist ebenfalls ein Grundbestandteil der `RootPane`.

10.2. JFrame

Viele Java Anwendungen sind mit Frames entwickelt worden. Auch werden sie verwendet, um Applets nicht nur mit dem Internet Browser oder mit dem AppletViewer ablaufen zu lassen. Der JFrame ersetzt den `java.awt.Frame` und ist entsprechend erweitert worden. Einer der Hauptunterschiede liegt in der Tatsache, dass der `JFrame` eine `RootPane` besitzt. Er bedient sich der darin enthaltenen `ContentPane`, die das Oberhaupt aller Subkomponenten des Frames darstellt. Das hat natürlich auch Konsequenzen für die Programmierung. Beim gewöhnlichen AWT-Frame war man es gewohnt, Komponenten folgendermassen einzufügen:

```
Frame awtFrame;
awtFrame.add(button);
```

Beim `JFrame` muss zunächst die `ContentPane` ermittelt werden:

```
JFrame swingFrame;
swingFrame.getContentPane().add(button);
```

Das gleiche Verfahren gilt für das Setzen von Layout-Managern. Nach Kreieren eines Frames ist standardmässig das BorderLayout eingestellt.

Einige weitere Gründe sprechen für die Verwendung des `JFrame`:

- mit `set_DefaultCloseOperation`() lässt sich schnell eine Art Schliessoperation festlegen, die nach Betätigen des Schliesssymbols des Fensters ausgeführt werden soll. Swing unterstützt drei Stücke:
  `DO_NOTHING_ON_CLOSE` : es passiert nichts, es wird dagegen `windowClosing`() des `WindowListener` angestossen.
  `HIDE_ON_CLOSE` : macht das Frame unsichtbar, zerstört ihn aber nicht
  `DISPOSE_ON_CLOSE` : zerstört den Frame nach Aufruf aller Routinen des `WindowListener.`
- Swing-Menüs arbeiten sehr gut mit einem `JFrame` zusammen, wenn sie mit `setMenuBar()` eingebunden werden.

Ansonsten ist es empfehlenswert, Swing Komponenten oder Applets, welche Swing Komponenten verwenden, in `JFrames` statt in AWT-Frames einzusetzen.

Ausserdem sind einige Funktionen im Angebot, mit denen die Bestandteile des Frames ermittelt bzw. neu gesetzt werden:

- `getRootPane`() und `setRotPane`() : liefern bzw. setzen die `RootPane` des Frames. Die `RootPane` verwaltet die Innenarchitektur.
- `getGlassPane`() und `setGlassPane`() : liefern bzw. setzen die `GlassPane` des Frames.
- `getLayeredPane`() und `setLayeredPane`() : liefert bzw. setzt die `LayeredPane` des Frames.

10.3. Internal Frame (JInternalFrame & JDesktopPane)

Während normale Frames (`JFrame`) nur für sich stehen, lassen sich mit internen Frames MDI Anwendungen entwickeln. Der `DesktopPane` entspricht einer Art Rahmenanwendung auf dem Desktop. Das interne Frame ist ein Fenster innerhalb dieser Anwendung. Der Desktop ersetzt das jeweilige Betriebsystem und kann den internen Frames ein vom Entwickler gewünschtes Aussehen verleihen. Wie sich daraus schliessen lässt, sind sie leichtgewichtig.

Zunächst erzeugt man eine Instanz von `JDesktopPane`. Sie ist eine Ableitung von JLayeredPane, mit andern Worten: eine spezielle `LayeredPane`, die für die Verarbeitung von internen Frames ausgerichtet ist. Die Methode `getAllFrames`() liefert alle im `DesktopPane` befindlichen Frames in einem Array. `getAllFramesInLayer`() bezieht sich dagegen auf einen bestimmten Layer.

Nun können nacheinander Objekte vom Typ `JInternalFrame` erzeugt und in den `DesktopPane` eingefügt werden. Dies erfolgt einfach mit `add`(), weil der `DesktopPane` selbst die `ControlPane` ist. Interne Frames können genauso wie normale Frames mit einigen Eigenschaften versehen werden: `setMaximizable`() und `setMinimizable`() (maximieren und minimieren ermöglichen). Mit `setResizable`() lässt sich die Grösse des Frames beeinflussen und mit `setIconifiable`() erscheint nach der Minimierung ein Symbol. Den Titel bestimmt `setTitle`(). Alle eben beschriebenen Einstellungen lassen sich auch schon bei der Instanzierung vollziehen. Wie es sich für ein Swing Fenster gehört, kann zu guter Letzt ein Layout-Manager mit `setLayout`() spezifiziert werden.

Das interne Frame unterstützt einen `InternalFrameListener`. Er empfängt Ereignisse, die bei diversen Frame-Operationen entstehen, und kann sie mit entsprechenden Methoden weiterverarbeiten.

Listing 10-1 Verwendung von JInternalFrame

```
package InternalFrame;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.URL;
import java.lang.Math;
import javax.swing.*;


public class InternalFrame extends JPanel
  implements ActionListener
{
  JInternalFrame    creatorFrame;
  JInternalFrame    newFrame;
  JDesktopPane      desktopPane;
  JCheckBox         closableCheck, maxableCheck;
  JCheckBox         iconfiableCheck, resizeableCheck;
  JButton           button;
  JTextField        titleField, layerField;
  Container         creatorPane;
  ImageIcon         javaIcon;
  JPanel            panel;
```

```
int               frameCounter;


InternalFrame(JApplet applet){
  setLayout(new BorderLayout());
  desktopPane = new JDesktopPane();
  desktopPane.setOpaque(false);
  newFrame = null;
  frameCounter = 0;

  try{
    javaIcon = new ImageIcon("seashore.jpg");
  }
  catch(SecurityException se)      // Für Internet-Browser{
    javaIcon = new ImageIcon(applet.getImage(
      applet.getCodeBase(), "seashore.jpg"));
  }

  // Erstellt die Frames
  creatorFrame = new JInternalFrame("InternalFrame erstellen",
    false, false, false, true);
  creatorFrame.setBounds(50, 20, 280, 180);
  desktopPane.add(creatorFrame);
  creatorPane = creatorFrame.getContentPane();
  creatorPane.setLayout(new GridLayout(0 ,1, 10, 10));
  creatorPane.add(panel = new JPanel());
  panel.setLayout(new GridLayout(2, 2, 10, 5));
  closableCheck = new JCheckBox("schließbar");
  closableCheck.setSelected(true);
  panel.add(closableCheck);
  resizeableCheck = new JCheckBox("Größe änderbar");
  resizeableCheck.setSelected(true);
  panel.add(resizeableCheck);
  maxableCheck = new JCheckBox("miximierbar");
  maxableCheck.setSelected(true);
  panel.add(maxableCheck);
  iconfiableCheck = new JCheckBox("minimierbar");
  iconfiableCheck.setSelected(true);
  panel.add(iconfiableCheck);

  creatorPane.add(panel = new JPanel());
  panel.setLayout(new BorderLayout(10, 5));
  panel.add("West", new JLabel("Titel:", JLabel.RIGHT));
  panel.add("Center", titleField = new JTextField("Titel"));
  creatorPane.add(panel = new JPanel());
  panel.setLayout(new BorderLayout(10, 5));
  panel.add("West", new JLabel("Layer:", JLabel.RIGHT));
  panel.add("Center", layerField = new JTextField("1"));
  creatorPane.add(panel = new JPanel());
  panel.add(button = new JButton("Öffnen"));
  panel.setLayout(new GridLayout(1, 2, 10, 5));
  button.addActionListener(this);
  panel.add(button = new JButton("Alle schließen"));
  button.addActionListener(this);

  add("Center", desktopPane);
}

public void actionPerformed(ActionEvent e){

  if (e.getActionCommand() == "Öffnen"){
    int layer;
```

```
      newFrame = new JInternalFrame(titleField.getText(),
        resizeableCheck.isSelected(),
        closableCheck.isSelected(),
        maxableCheck.isSelected(),
        iconfiableCheck.isSelected());

      // Fensterposition per Zufallsgenerator
      newFrame.setBounds((int)(Math.random()*500)+30,
        (int)(Math.random()*300)+20, 300, 300);
      frameCounter++;
      switch (frameCounter % 3)
      {
        case 0:
          newFrame.setContentPane(new CustomScrollTextPane());
          break;
        case 1:
          newFrame.setContentPane(new ImageScrollPane(javaIcon));
          break;
        case 2:
          newFrame.setContentPane(new ButtonScrollPane());
          break;
      }
      layer = Integer.parseInt(layerField.getText());
      desktopPane.add(newFrame, new Integer(layer));
      newFrame.repaint();
      newFrame.toFront();
    }
    else
    {
      desktopPane.removeAll();
      desktopPane.add(creatorFrame);
      desktopPane.repaint();
      frameCounter=0;
    }
  }
}


class ImageScrollPane extends JScrollPane
{
  JLabel       label;

  public ImageScrollPane(ImageIcon i)
  {
    super();
    label = new JLabel(i);
    label.setOpaque(false);
    getViewport().add(label);
  }
}


class CustomScrollTextPane extends JScrollPane
{
  JTextArea    textArea;

  CustomScrollTextPane()
  {
    super();
    textArea = new JTextArea(loadTextFile("skat.txt"));
    getViewport().add(textArea);
```

```
  }

  public String loadTextFile(String filename)
  {
    String s = new String();
    File f;
    char[] buff = new char[50000];
    InputStream is;
    InputStreamReader reader;
    URL url;

    try
    {
      int nch;
      url = (new File(filename)).toURL();
      is = url.openStream();
      reader = new InputStreamReader(is);
      while ((nch = reader.read(buff, 0, buff.length)) != -1)
         s = s + new String(buff, 0, nch);
    }
    catch (java.io.IOException ex)
    {
      s = "Datei konnte nicht geladen werden: " + filename;
    }

    return s;
  }
}


class ButtonScrollPane extends JScrollPane
{
  JPanel  panel;

  ButtonScrollPane()
  {
    panel = new JPanel();
    panel.setLayout(new GridLayout(0, 1, 5, 5));
    for (int i=1; i<9; i++)
      panel.add(new JButton("Funktion " + i));
    getViewport().add(panel);
  }
}
```

Und so sieht das Ganze aus: mit Hilfe des Fensters links oben werden Fenster kreiert.


Swing Beispiel
InternalFrame erstellen
schließbar
Größe änderbar
miximierbar
minimierbar
Titel:
und noch eins
Layer:
5
Öffnen
Alle schließen
ein fenster
das zweite
Funktion 1
Funktion 2
Funktion 3
Funktion 4
Funktion 5
Funktion 6
Funktion 7
Funktion 8
und noch eins
1. Geschichtliches über Skat
Wie alte Zeugnisse belegen, ist Skat aus dem Spiel Scha
entstanden. Schafkopf wurde bereits 1810 mit 4 Personer
gespielt. Die Brommesche Tarockgesellschaft in Altenbur
bevorzugte jedoch Spiele, die zu dritt gespielt wurden, z.B.
das spanische L`hombre und das aus Italien stammende
Tarock. Es hatten sich so Dreier - Spielgemeinschaften
gefunden, die ungerne eine 4. Person aufnehmen wollten
Aus diesem Grunde versuchten manche Spieler nun das

10.4. JDialog

Dialoge in Swing basieren auf `JDialog`. Man kann die Klassen für benutzerdefinierte Dialoge einsetzen. Der `JDialog` hat als einzige Unterklasse ein `JRootPane`. Komponenten, die den `JDialog` mit aufgenommen werden, müssen auf die `ContentPane` des Dialoges zurück greifen. Man bekommt sie mittels `getContentPane`().

Beispiel:

```
dialog.getContentPane().setLayout(new BorderLayout() );
```

anstatt

```
dialog.setLayout(new BorderLayout() );
```

Als Grundlage benötigt der `JDialog` ein Frame (`JFrame`). Wird dieser nicht explizit im Konstruktor angegeben, *leiht* er sich sozusagen einen gemeinsame genutzten Frame aus. Ansonsten unterscheidet sich die Programmierung eines Dialoges nicht sehr von der eines Frames.

Listing 10-2 Verwendung von JDialog

```
package SwingDialog;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class SwingDialog extends JPanel implements ActionListener{
  JButton        button;
  CustomDialog   customDialog;
  ImageIcon      dukeIcon;

  SwingDialog(JApplet applet)  {
    add(button = new JButton("Öffne SwingDialog."));
    button.addActionListener(this);

    try    {
      dukeIcon = new ImageIcon("duke.gif");
    }
    catch(SecurityException se)  // Für Internet-Browser
    {
      dukeIcon = new ImageIcon(applet.getImage(
        applet.getCodeBase(), "duke.gif"));
    }

    customDialog = new CustomDialog(dukeIcon,
      "Hallo, das ist ein SwingDialog.");
  }

  public void actionPerformed(ActionEvent e)
  {
    customDialog.setVisible(true);
  }
}
```

```
class CustomDialog extends JDialog implements ActionListener
{
  JPanel   panel;
  JButton  button;
  ImageIcon icon;
  String    text;

  CustomDialog(ImageIcon i, String t)  {
    super();
    icon = i;
    text = t;
    setTitle("Ein SwingDialog");
    setModal(true);                  // Modaler Dialog
    setBounds(100, 100, 350, 150);
    getContentPane().setLayout(new BorderLayout());
    setFont(new Font("Dialog", Font.BOLD, 14));

    getContentPane().add("South", panel = new JPanel());
    panel.setLayout(new FlowLayout(FlowLayout.CENTER, 20, 0));
    panel.add(button = new JButton("OK"));
    button.addActionListener(this);
    panel.add(button = new JButton("Abbrechen"));
    button.addActionListener(this);

    getContentPane().add("Center",
      new JLabel(text, icon, JLabel.CENTER));
  }

  public void actionPerformed(ActionEvent e)
  {
    setVisible(false);
  }
}
```

10.5. JOptionPane

Während `JDialog` eher für benutzerdefinierte Dialoge von Bedeutung ist, bietet `JOptionPane` eine Reihe von Standarddialogen (zum Beispiel Warnmeldungen, Sicherheitsabfragen), ohne dass der Programmierer bei der Programmierung einen grossen Aufwand betreiben muss.

Insgesamt gibt es vier Grundtypen. Wirft man einen Blick auf die Referenz von `JOptionPane`, macht sie zunächst einen recht komplizierten Eindruck. Es stellt sich jedoch bald heraus, dass zur Erzeugung der Dialoge nur ein Aufruf einer statischen Methode genügt. Nicht einmal die Instanzierung von `JOptionPane` ist nötig. Für jede dieser Methoden existiert eine weitere Version, um Dialoge als interne Frames zu erzeugen.

- `showConfirmDialog() bzw. showInternalConfirmDialog()`
  zeigt einen Bestätigungsdialog.

  Hier muss der Benutzer in den meisten Fällen Entscheidungen treffen (zum Beispiel bei Sicherheitsabfragen). Das Icon hängt von der Meldungsart ab (Details folgen). Defaultmässig bietet der Dialog einen Button mit "OK" zur Bestätigung. Die Buttons können jedoch optional in vier verschiedenen Varianten eingestellt werden (siehe unten). Mit nur einem "OK" Button hat es die gleiche Funktion wie der Meldungsdialog.

- `showInputDialog() bzw. showInternalInputDialog()`

  In diesem Dialog können Eingaben in einem Textfeld getätigt werden. Nach Wunsch können auch fertige Alternativen in einem Kombinationsfeld angeboten werden. Im Gegensatz zum Meldungsdialog ist es dem Programmierer verwehrt, die Button zu variieren. Der Eingabedialog stellt den Benutzer immer vor die Wahl zwischen "OK" und "Cancel". Die Meldungsart bestimmt wiederum das Icon, hat aber sonst keinen Einfluss auf die Funktionalität.

- `showMessageDialog() bzw. showInternalMessageDialog()`

  Zeigt einen Meldungsdialog, der den Benutzer beispielsweise über Vorgänge oder vor bestimmten Aktionen warnen soll. Er besitzt immer nur einen "OK" Button, wobei das Icon sich wieder nach der Meldungsart richtet.

- `showOptionDialog() bzw. showInternalOptionDialog()`

  Dieser Dialog ist eine Art Bestätigungsdialog, doch sind für die Buttons nicht nur die vier Optionsarten (s.u.) vorgegeben. Die Anzahl und Beschriftung der Buttons ist frei definierbar. Für ihre Identifizierung werden sie von null an beginnend durchnummeriert.

Hinweis: alle Dialoge sind modal. Nichtmodale Dialoge sind nur mit `JDialog` zu erstellen.

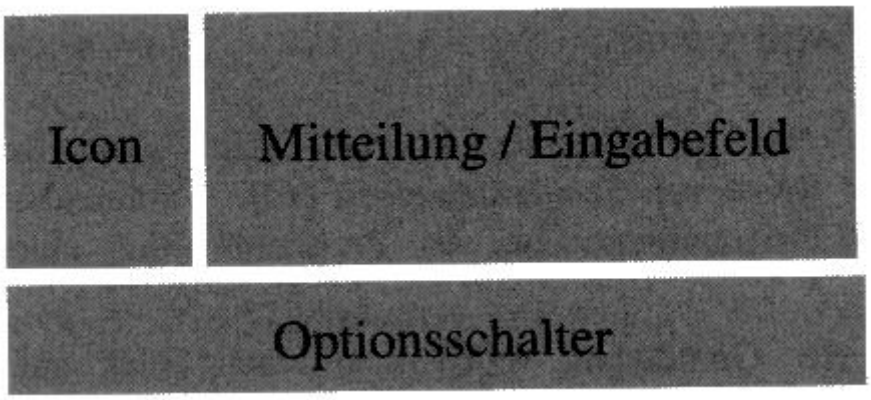


Abbildung 10-2 Dialogaufbau

Die Dialoge von `JOptionPane` besitzen dasselbe Layout. Sie bestehen aus einem Icon, einer Mitteilung und aus einem oder mehreren Optionsschaltern. Neben der Mitteilung kommt bei einem Eingabedialog eine Eingabezeile bzw. ein Kombinationsfeld hinzu.

10.5.1. Meldungsarten

Für die jeweilige Meldungsart zeigt der Dialog ein charakteristisches Icon. Diese Option wird jedoch bei einem benutzerdefinierten Icon ignoriert.

- `ERROR_MESSAGE`
  Fehlermeldung; als Icon wird ein Achteck angezeigt.
- `INFORMATION_MESSAGE`
  zeigt eine Informationsmeldung an; als Icon wird ein "i" in einem Kreis verwendet
- `WARNING_MESSAGE`
  Warnmeldung; als Icon wird ein "!" in einem Dreieck (wie im Strassenverkehr) verwendet
- `QUESTION_MESSAGE`
  Frage; als Icon wird ein Fragezeichen "?" in einem Viereck angezeigt
- `PLAIN_MESSAGE`
  allgemeine Meldung; hierfür wird kein Standardicon eingesetzt.

10.5.2. Optionswerte

Jeder der vier Optionswerte stellt eine Kombination von Optionsschaltern dar. Sie gelten nur für den Bestätigungsdialog und Optionsdialog. Werden bei einem Optionsdialog eigene Optionen übergeben, ist dieser Wert bedeutungslos.

- `DEFAULT_OPTION`
  nur der "OK" Button erscheint
- `YES_NO_OPTION`
  es erscheint ein "Yes" und ein "No" Button
- `YES_NO_CANCEL_OPTION`
  es erscheint ein "Yes", "No" und ein "Cancel" Button
- `OK_CANCEL_OPTION`
  es erscheinen ein "OK" und ein "Cancel" Button

Die Icons sehen Sie weiter unten im Beispielprogramm Screensnatshot.

10.5.3. Sonstige Übergabeparameter

- eine Komponente als Parent:
  von ihr bezieht der Dialog den Frame und bestimmt die Koordinaten

- die Mitteilung:
  sie ist vom Typ `Object`, so dass man so ziemlich alle Komponenten benutzen darf, die der Dialog verarbeiten kann. Im einfachsten Falle ist die Mitteilung ein gewöhnlicher `String`. Verwendet man eine Swing Komponente, so wird diese an Stelle der schriftlichen Mitteilung angezeigt. Ansonsten sind alle Objekte erlaubt, die die Methode `toString()` implementieren.

- die Optionen:
  sie gelten nur für den Eingabe- und Ausgabedialog. Hier können die gleichen Objekttypen wie bei der Mitteilung übergeben werden

- ein benutzerdefiniertes Icon:
  die Meldungsart wird dann ignoriert

- Titel des Dialoges

- Default-Eingabewerte für den Eingabedialog.

10.5.4. Rückgabeparameter

Sämtliche `showDialog` - Funktionen liefern bis auf den Meldungsdialog ein Ergebnis zurück. Seine Bedeutung entspricht der Art des Dialogs. Der Eingabedialog liefert nach der positiven Bestätigung den Eingabetext des Benutzers. Der Bestätigungs- und Optionsdialog geben die Nummer des gedrückten Optionsschalters zurück:

```
YES_OPTION, NO_OPTION, CANCEL_OPTION, OK_OPTION, CLOSED_OPTION
```

Listing 10-3 Verwendung von JOptionPane

```
package OptionPane;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class OptionPane extends JPanel implements ActionListener{
  JPanel       dialogButtonPanel; // Panel für die 4 Dialog-Buttons
  JPanel       optionPanel;       // für Optionen
  JPanel       messagePanel;      // Message-Typen
  JPanel       resultPanel;       // Dialogergebnisse
  JPanel       customPanel;
  JOptionPane  optionPane;
  JTextField   confirmField, inputField, optionField;
  ButtonGroup  optionGroup;
  ButtonGroup  messageGroup;
  JButton      button;
  JRadioButton radioButton;
  JCheckBox    customIconCheck, inputCheck;
  ImageIcon    dukeIcon;
  ImageIcon    customIcon;
```

```
String  inputValues[] =
        {"1. Möglichkeit: rechts ", "2. Möglichkeit: links",
         "3. Möglichkeit: vorne"};
String  messageNames[] =
        {"ERROR_MESSAGE", "INFORMATION_MESSAGE",
         "WARNING_MESSAGE", "QUESTION_MESSAGE", "PLAIN_MESSAGE"};
int  messageValues[] =
     {JOptionPane.ERROR_MESSAGE, JOptionPane.INFORMATION_MESSAGE,
      JOptionPane.WARNING_MESSAGE, JOptionPane.QUESTION_MESSAGE,
      JOptionPane.PLAIN_MESSAGE};
String  optionNames[] =
      {"DEFAULT_OPTION", "YES_NO_OPTION", "YES_NO_CANCEL_OPTION",
       "OK_CANCEL_OPTION"};
int optionValues[] =
    {JOptionPane.DEFAULT_OPTION, JOptionPane.YES_NO_OPTION,
  JOptionPane.YES_NO_CANCEL_OPTION, JOptionPane.OK_CANCEL_OPTION};
int     dialogOption = optionValues[0];
int     messageType  = messageValues[0];


OptionPane(JApplet applet)   {
  setLayout(null);
  try    {
    dukeIcon = new ImageIcon("duke.gif");
  }// Für Internet-Browser
  catch(SecurityException se)      {
    dukeIcon = new ImageIcon(
      applet.getImage(applet.getCodeBase(), "duke.gif"));
  }
  optionPane = new JOptionPane();
  optionPane.setInputValue("Test");

  // Buttons für die vier Dialogarten
  dialogButtonPanel = createTitledPanel("Standard-Dialoge",
    10, 10, 200, 230);
  dialogButtonPanel.add(button = new JButton("ConfirmDialog"));
  button.addActionListener(this);
  dialogButtonPanel.add(button = new JButton("InputDialog"));
  button.addActionListener(this);
  dialogButtonPanel.add(button = new JButton("MessageDialog"));
  button.addActionListener(this);
  dialogButtonPanel.add(button = new JButton("OptionDialog"));
  button.addActionListener(this);
  add(dialogButtonPanel);

  // Optionen für die Dialoge
  optionPanel = createTitledPanel("Dialogoptionen",
    240, 10, 210, 100);
  optionGroup = new ButtonGroup();
  for (int i=0; i<4; i++)
  {
    optionPanel.add(radioButton = new JRadioButton(optionNames[i]));
    radioButton.setSelected(i == 0);  // 1. Button selektiert
    radioButton.addActionListener(this);
    optionGroup.add(radioButton);
  }
  add(optionPanel);

  // Message-Typen für die Dialoge
  messagePanel = createTitledPanel("Message-Typen",
    240, 120, 210, 120);
  messageGroup = new ButtonGroup();
```

```
    for (int i=0; i<5; i++)
    {
      messagePanel.add(radioButton = new JRadioButton(messageNames[i]));
      radioButton.setSelected(i == 0);  // 1. Button selektiert
      radioButton.addActionListener(this);
      messageGroup.add(radioButton);
    }
    add(messagePanel);

    // Sonstige Optionen
    customIconCheck = new JCheckBox("Mit eigenem Icon");
    customIconCheck.setBounds(20, 240, 130, 20);
    add(customIconCheck);
    inputCheck = new JCheckBox(
      "Mit Auswahlmöglichkeit beim InputDialog und OptionDialog");
    inputCheck.setBounds(20, 260, 440, 20);
    add(inputCheck);

    // Ergebnisse der Dialoge
    resultPanel = createTitledPanel("Ergebnisse",
      10, 290, 440, 100);
    resultPanel.setLayout(new GridLayout(0, 2, 10, 5));
    resultPanel.add(new JLabel("ConfirmDialog"));
    resultPanel.add(confirmField = new JTextField());
    resultPanel.add(new JLabel("InputDialog"));
    resultPanel.add(inputField = new JTextField());
    resultPanel.add(new JLabel("OptionDialog"));
    resultPanel.add(optionField = new JTextField());
    add(resultPanel);
  }

  public void actionPerformed(ActionEvent e)
  {
    int       optionValue=0;    // gewählte Dialogoption
    String    inputText=null;   // Eingabetext für den InputDialog

    // Mit benutzerdefinierten Icon ?
    if (customIconCheck.isSelected())
      customIcon = dukeIcon;  // Ja
    else
      customIcon = null;      // Nein

    // Eine Dialogoption oder ein Message-Typ wurde ausgewählt
    if (e.getSource() instanceof JRadioButton)
    {
      for (int i=0; i<4; i++)   // Dialogoption
        if (e.getActionCommand() == optionNames[i])
          // setze neue Dialogoption
          dialogOption = optionValues[i];

      for (int i=0; i<5; i++)   // MessageTyp
        if (e.getActionCommand() == messageNames[i])
          // setze neuen MessageTyp
          messageType = messageValues[i];
    }

    // Ein Dialogtyp wurde gewählt
    if (e.getSource() instanceof JButton)
    {
      if (e.getActionCommand() == "ConfirmDialog")
      {
        optionValue = optionPane.showConfirmDialog(this,
```

```
          "Hallo! Das ist ein ConfirmDialog.", "ConfirmDialog",
          dialogOption, messageType, customIcon);
        // Ergebnis ins Ergebnisfeld schreiben
        confirmField.setText(Integer.toString(optionValue));
      }

      if (e.getActionCommand() == "InputDialog")
      {
        if (inputCheck.isSelected())
          // Mit ComboBox
          inputText = (String)optionPane.showInputDialog(this,
            "Hallo! Das ist ein InputDialog.", "InputDialog",
            messageType, customIcon, inputValues, inputValues[0]);
        else
          // Mit Textfeld
          inputText = (String)optionPane.showInputDialog(this,
            "Hallo! Das ist ein InputDialog.", "InputDialog",
            messageType, customIcon, null, null);
         // Ergebnis ins Ergebnisfeld schreiben
        inputField.setText(inputText);
      }

      // Einfacher MessageDialog
      if (e.getActionCommand() == "MessageDialog")
        optionPane.showMessageDialog(this,
          "Hallo! Das ist ein MessageDialog.",
          "MessageDialog", messageType, customIcon);

      // OptionDialog
      if (e.getActionCommand() == "OptionDialog")
      {
        if (inputCheck.isSelected())
          optionValue = optionPane.showOptionDialog(this,
            "Hallo! Das ist ein OptionDialog.", "OptionDialog",
            dialogOption, messageType, customIcon, inputValues,
            inputValues[0]); // Mit Optionen
        else
          optionValue = optionPane.showOptionDialog(this,
            "Hallo! Das ist ein OptionDialog.", "OptionDialog",
            dialogOption, messageType, customIcon, null, null);
        optionField.setText(Integer.toString(optionValue));
      }
    }
  }


  // Erstellt ein Panel mit Rahmen
  public static JPanel createTitledPanel(String title, int x, int y,
    int width, int height)
  {
    JPanel p = new JPanel();
    p.setLayout(new GridLayout(0, 1, 30, 5));
    p.setBorder(BorderFactory.createTitledBorder(
      new EtchedBorder(), title, 0, 0,
      new Font("Dialog", Font.BOLD, 12)));
    p.setBounds(x, y, width, height);
    return p;
  }
}
```

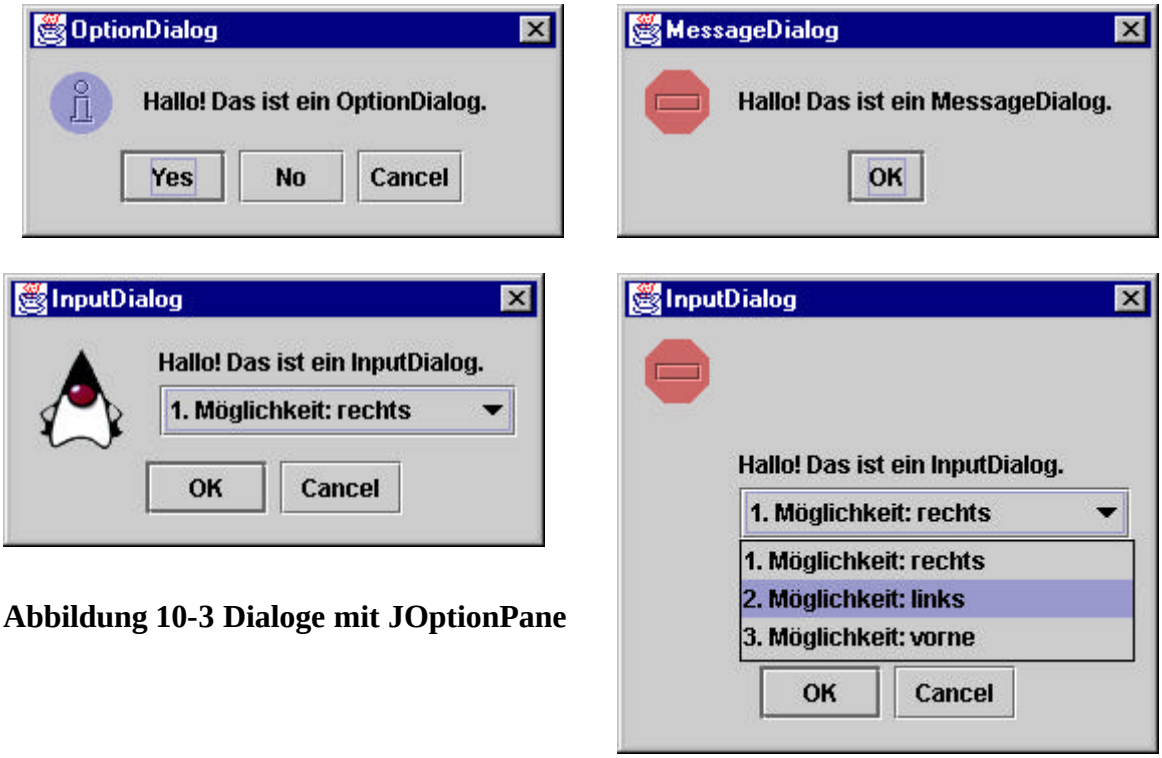


Abbildung 10-3 Dialoge mit JOptionPane

Abbildung 10-4 Steuerprogramm

10.6. Spezialdialoge

Swing bietet immerhin zwei äusserst nützliche Spezialdialoge. Sie werden in den meisten Anwendungen verschiedenster Bereiche angewendet. Um den Programmierer nicht jedesmal vor eine schwierige Aufgabe zu stellen, sind sie leicht verwendbar und mit umfangreichen Funktionen versehen. Nebenbei ist ein einheitlicher Layout in den Anwendungen sichergestellt. Die beiden folgenden Dialoge unterstützen den Anwender bei der Auswahl von Dateien und Farben.

10.6.1. JFileChooser

Die Klasse `JFileChooser` vertritt den File Dialog von Swing. Mit ihm kann der Benutzer schnell und komfortabel eine bzw. mehrere Dateien oder Verzeichnisse auswählen. Er erinnert stark an den Standard File Dialog von Windows. Im oberen Teil des Dialogs befindet sich ein Kombinationsfeld, welches alle vorhandenen Laufwerke und alle zuletzt besuchten Verzeichnisse auflistet. Es stellt eine Art History Funktion dar. Rechts davon sind einige Funktionsschalter. Der erste geht in das nächsthöherliegende Verzeichnis und der nächste in das Ursprungsverzeichnis. Weiterhin können neue Verzeichnisse erstellt werden. Die letzten beiden Funktionen bestimmen die Darstellung der Dateien und Verzeichnisse. Im unteren Teil stehen die Felder für den Dateinamen und den Dateityp.

Schon beim Anlegen von `JFileChooser` kann der Programmierer ein Ausgabeverzeichnis festlegen. Ansonsten wird das aktuelle Verzeichnis mit `setCurrentDirectory`() definiert und mit `getCurrentDirectory`() abgefragt. Den Titel des Dialogs gibt `setDialogTitle`() vor.

Nach Erzeugung der Instanz muss der Dialog mit einem Funktionsaufruf sichtbar gemacht werden. Grundsätzlich gibt es drei Arten von File-Dialogen, nämlich zum Öffnen, Speichern und für benutzerdefinierte Operationen. Gezeigt werden sie jeweils mit `showOpenDialog`(), `showSaveDialog`() und `showDialog`(). Letztere benötigt zusätzlich eine Bezeichnung des Buttons zur positiven Bestätigung der Auswahl (z.B. "OK"). Der Dialog ist modal. Erst nach dem Schliessen kann die Anwendung fortgeführt werden. Ob der Benutzer eine Auswahl getroffen oder die Operationen abgebrochen hat, ist im Rückgabewert zu finden: `APPROVE_OPTION` (für OK) oder `CANCEL_OPTION` (für Abbruch).

Mit `setSelectionMode`() kann die Wahl auf Dateien, Verzeichnisse oder beides festgelegt werden. `setMultipleSelection`() schaltet die Mehrfachwahl ein bzw. aus. Die gewählte Datei ist mit `getSelectedFile`() zu ermitteln. Bei einer Mehrfachauswahl hilft `getSelectedFiles`().

Die Funktion `getName`(), `getDescription`() und `getIcon`() liefern den Namen, eine Beschreibung und das Symbol einer Datei. Sie muss mit übergeben werden und von Typ `java.io.File` sein.

10.6.1.1. Dateifilter

`JFileChooser` zeigt von Grund auf alle Dateien an. Häufig ist der Benutzer einer bestimmten Anwendung nur an speziellen Dateitypen interessiert. Man möchte zum Beispiel mit einem Malprogramm nur JPEG-Bilder enladen. Die Liste aller Dateien im Dateidialog

würde nur verwirren und zur Selektion falscher Dateien mit ungültigem Format führen. In einem solchen Fall wäre ein wählbarer Dateifilter sinnvoll.

Ein Dateifilter wird von der abstrakten Klasse `FileFilter` im Paket `javax.swing.filechooser` implementiert. Für neue Dateifilter muss eine Klasse geschrieben werden, die von `FileFilter` erbt. Sie besitzt zwei Methoden, die es zu überschreiben gilt:

- `public abstract boolean accept(File f)`
  beim Einlesen der Dateiliste des Verzeichnisses werden alle Dateien mit `accept`() geprüft. Liefert sie `true` zurück, wird die Datei im Dialog angezeigt.
- `public abstract String getDescription()`
  liefert die Beschreibung des Dateifilters bzw. Dateitypen

Der Filter kann nun mit `setFileFilter`() gesetzt werden. Dann ist immer nur ein Filter gleichzeitig möglich.
Mit `addChoosableFileFilter`() baut man eine Liste von Dateifiltern auf, die in einem Kombinationsfeld im File-Dialog anwählbar sind.

Will man einen Filter nur einem Dateityp zuordnen, d.h. es sollen beispielsweise nur JPEG-Bilder sichtbar sein, so ist die `accept`() Methode folgendermassen zu implementieren:

```
public boolean accept(File file) {
      if (file.isDirectory() ) return true;

      String fileName = file.getName();
      int i = filename.lastIndexOf('.');

      if (i>o && i< fileName.length()-1) {
            String ext = fileName.subString(i+1).toLowerCase();
            if (ext.equals("jpg") ) return true;
            else return false;
      }
      return false;
}
```

Sollen sowohl JPEG Bilder als auch GIF Bilder in der Dateiliste sein, so ist

```
      if (ext.equals("jpg") ) return true;
```

durch

```
      if (ext.equals("jpg") || ext.equals("gif") ) return true;
```

zu ersetzen.

Das folgende Beispielprogramm verwendet einen Dateifilter für Textdateien.

Listing 10-4 Verwendung von JFileChooser

```
package FileChooser;
import java.awt.*;
import java.awt.event.*;
import java.io.File;
import javax.swing.*;
```

```
public class FileChooser extends JPanel implements ActionListener{
  JPanel        resultPanel;
  JButton       button;
  JTextField    fileField, pathField;
  JTextField    descriptionField, commandField;
  JFileChooser  fileChooser;
  JLabel        fileIcon;


  FileChooser(JApplet applet){
    setLayout(new BorderLayout(10, 5));

    fileChooser = new JFileChooser();
    fileChooser.setFileSelectionMode(
      JFileChooser.FILES_AND_DIRECTORIES);
    fileChooser.setMultiSelectionEnabled(false);
    fileChooser.setDialogTitle("Ein FileChooser-Beispiel");
    fileChooser.addChoosableFileFilter(new CustomFileFilter());

    add("North", button = new JButton("Öffne den FileChooser"));
    button.addActionListener(this);

    resultPanel = new JPanel();
    resultPanel.setLayout(new GridLayout(0, 2, 10, 5));
    resultPanel.add(new JLabel("Datei:"));
    resultPanel.add(fileField = new JTextField());
    resultPanel.add(new JLabel("Pfad:"));
    resultPanel.add(pathField = new JTextField());
    resultPanel.add(new JLabel("Beschreibung:"));
    resultPanel.add(descriptionField = new JTextField());
    resultPanel.add(new JLabel("OK oder Cancel:"));
    resultPanel.add(commandField = new JTextField());
    resultPanel.add(new JLabel("Dateisymbol:"));
    resultPanel.add(fileIcon = new JLabel());
    add("Center", resultPanel);
  }



  public void actionPerformed(ActionEvent e)  {
    if (fileChooser.showDialog(this,
          "OK")!=JFileChooser.CANCEL_OPTION){
      File  testFile = fileChooser.getSelectedFile();
      fileField.setText(testFile.toString());
      pathField.setText(fileChooser.getCurrentDirectory().toString());
      descriptionField.setText(fileChooser.getTypeDescription(testFile));
      fileIcon.setIcon(fileChooser.getIcon(testFile));
      commandField.setText("OK");
    }
    else
      commandField.setText("Cancel");
  }
}

class CustomFileFilter extends javax.swing.filechooser.FileFilter {
  String  extension, description;

  CustomFileFilter(){
    super();
    extension = "txt";
    description = "Textdateien";
  }
```

```
  public boolean accept(File file){
    if (file.isDirectory())
      return true;

    String fileName = file.getName();
    int i = fileName.lastIndexOf('.');

    if (i>0 &&  i<fileName.length() - 1){
      String ext = fileName.substring(i+1).toLowerCase();
      if (ext.equals(extension))
        return true;
      else
        return false;
    }

    return false;
  }

  public String getDescription(){
    return description;
  }
}
```

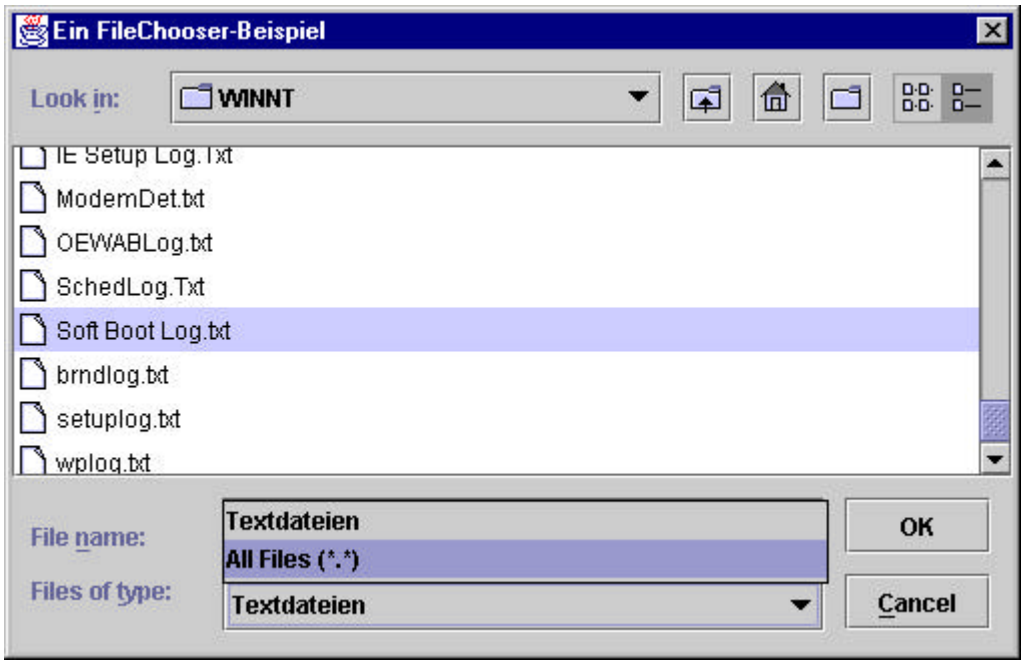


Abbildung 10-5 Einsatz eines Dateifilters

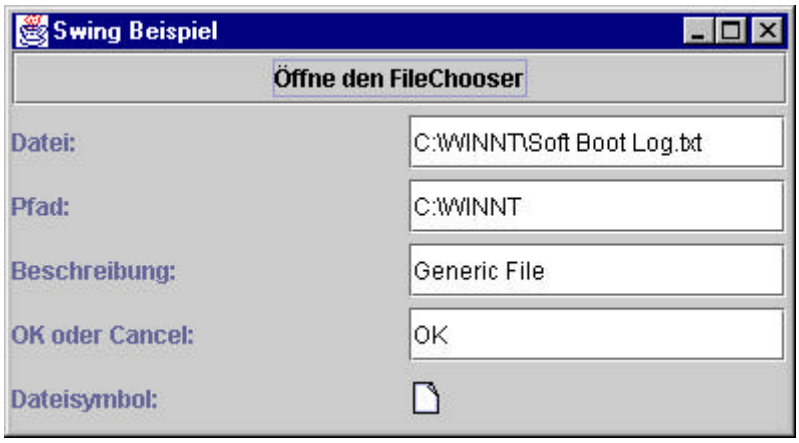


Abbildung 10-6 Ergebnis einer Dateiauswahl

10.6.1.2. JColorChooser

`JColorChooser` ist ein Standarddialog für Farbpaletten von Swing. Er ist besonders nützlich, um geeignete Farben für Mal- und Textverarbeitungsprogramme zu ermitteln. Der obere Teil des Dialoges zeigt eine Anzahl von vordefinierten Farbtönen, denen sich der Benutzer je nach Vorliebe bedienen kann. Trifft allerdings kein Farbton seinen Geschmack, kann er nach RGB umschalten, wo er die Rot-, Grün- und Blau-Anteile exakt eingeben kann. Im Mittelfeld des Dialoges ist eine kleine Vorschau. Sie soll einen Eindruck des Farbtons vermitteln.

Gleich beim Erzeugen von `JColorChooser` kann im Konstruktor eine Ausgangsfarbe belegt werden. Sie ist aber jederzeit mit `setColor`() änderbar. Erlaubt sind entweder die RGB- Anteile oder eine Instanz von `java.awt.Color`. Mit `showDialog`() wird der Farbdialog geöffnet. Hier sind auch Titel und Ausgangsfarbe mit anzugeben. Dieser Dialog ist ebenfalls modal. Der Rückgabewert gibt Auskunft über die gewählte Farbe oder ist `null` bei Abbruch.

Bei `JColorChooser` handelt es sich nicht um eine Instanz von `JDialog`, sondern vielmehr um eine Ableitung von `JComponet`. Beim Aufruf von `showDialog`() wird implizit der Dialog erzeugt und sichtbar gemacht. Will man jedoch den Farbdialog als Instanz von `JDialog`, wird dieser mit `createDialog`() erzeugt.

Listing 10-5 Verwendung von JColorChooser

```
package ColorChooser;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;


public class ColorChooser extends JPanel implements ActionListener
{
  JPanel   colorPanel, panel;
  JButton  chooserButton;
  JColorChooser colorChooser;
  JTextField  redField, greenField, blueField;  // Farbwerte

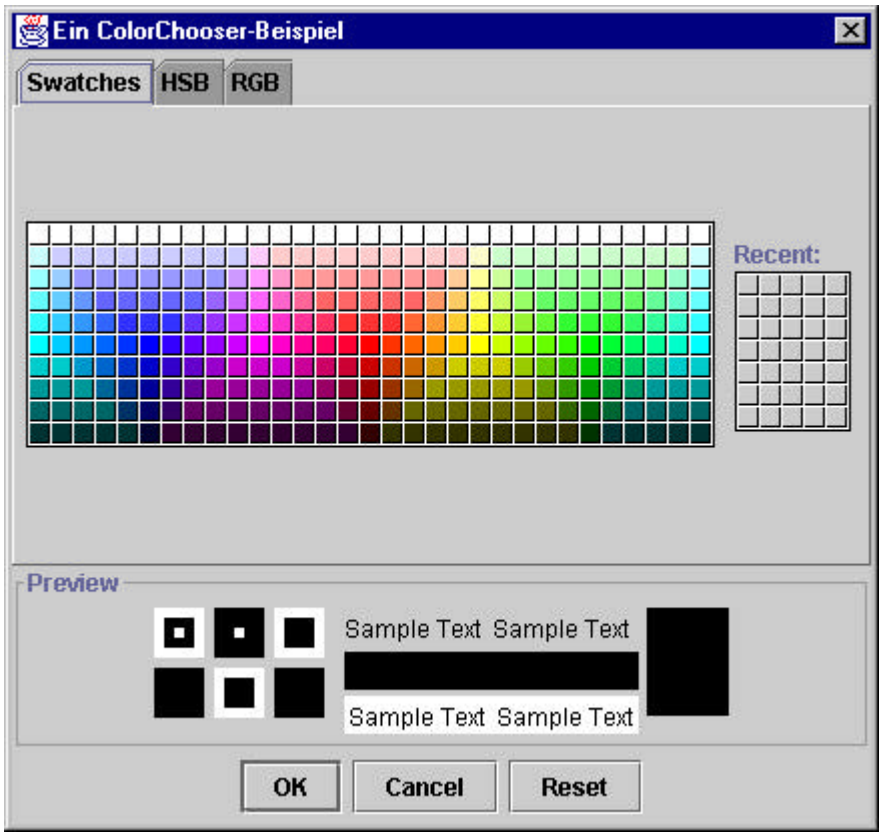
  ColorChooser(JApplet applet)
  {
    colorPanel = new JPanel();
    colorChooser = new JColorChooser();
    setLayout(new GridLayout(2, 1, 10, 10));
    colorPanel.setLayout(new BorderLayout(10, 5));

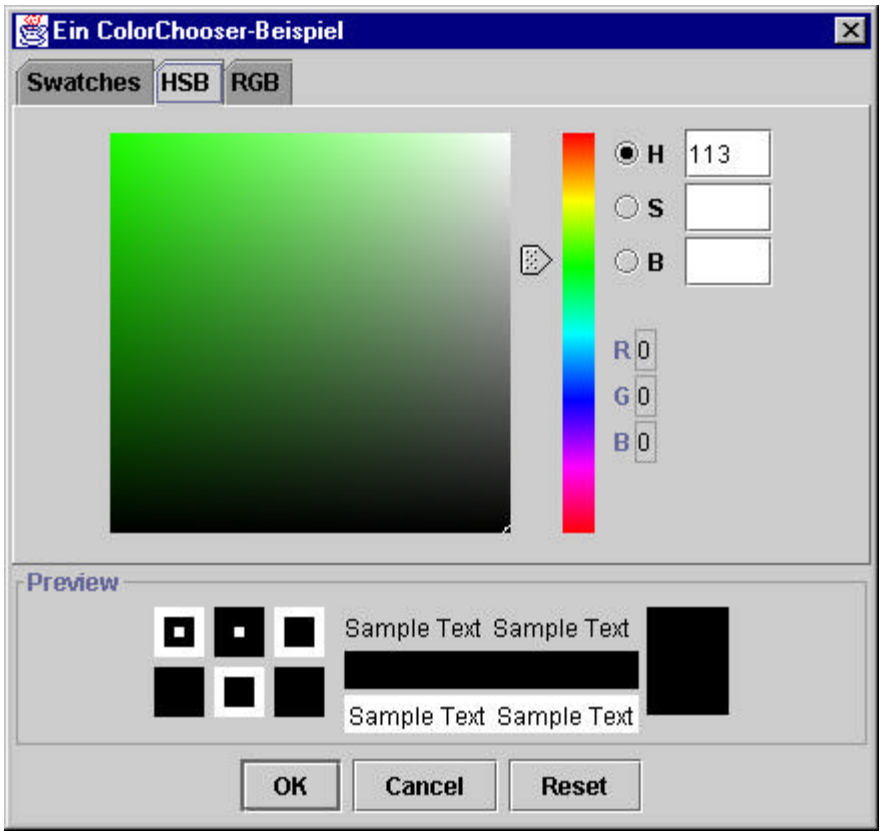
    add(chooserButton = new JButton("Öffne ColorChooser"));
    chooserButton.addActionListener(this);
    add(colorPanel);
    colorPanel.add("West", panel = new JPanel());
    panel.setLayout(new GridLayout(0, 1, 0, 5));
    panel.add(new JLabel("Rot:"));
    panel.add(new JLabel("Grün:"));
    panel.add(new JLabel("Blau:"));
    colorPanel.add("Center", panel = new JPanel());
    panel.setLayout(new GridLayout(0, 1, 0, 5));
    panel.add(redField = new JTextField("0"));
```

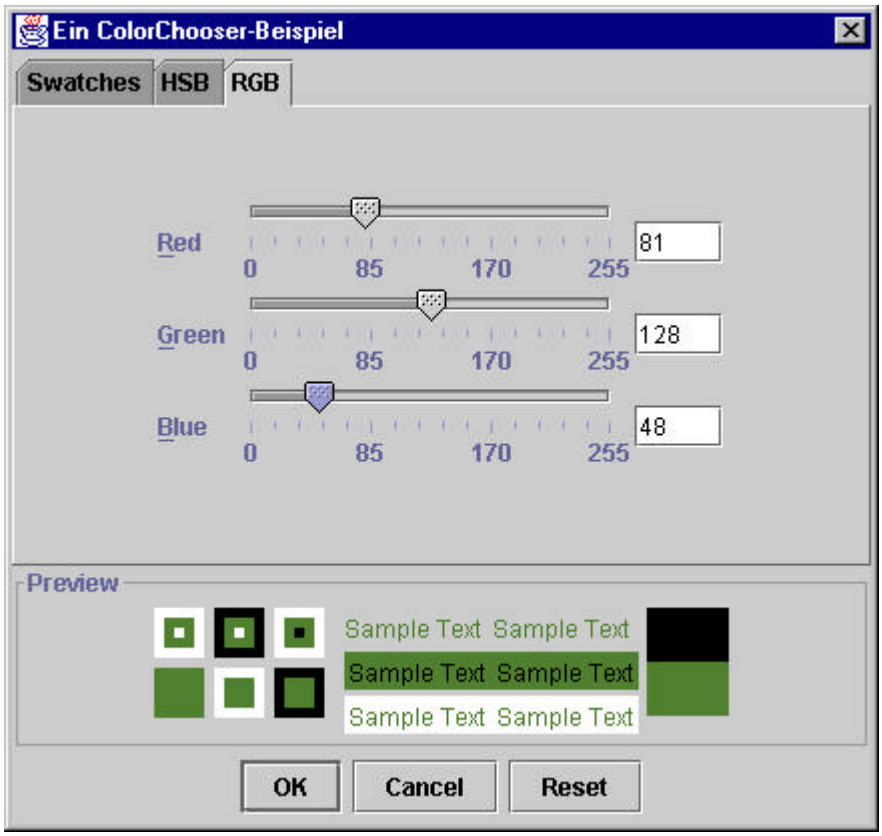
```
    panel.add(greenField = new JTextField("0"));
    panel.add(blueField = new JTextField("0"));
  }

  public void actionPerformed(ActionEvent e)
  {
    Color  c;
    int   r, g, b;

    r = Integer.parseInt(redField.getText());
    g = Integer.parseInt(greenField.getText());
    b = Integer.parseInt(blueField.getText());
    c = colorChooser.showDialog(this, "Ein ColorChooser-Beispiel",
      new Color(r, g, b));
    if (c!=null)
    {
      redField.setText(Integer.toString(c.getRed()));
      greenField.setText(Integer.toString(c.getGreen()));
      blueField.setText(Integer.toString(c.getBlue()));
    }
  }
}
```



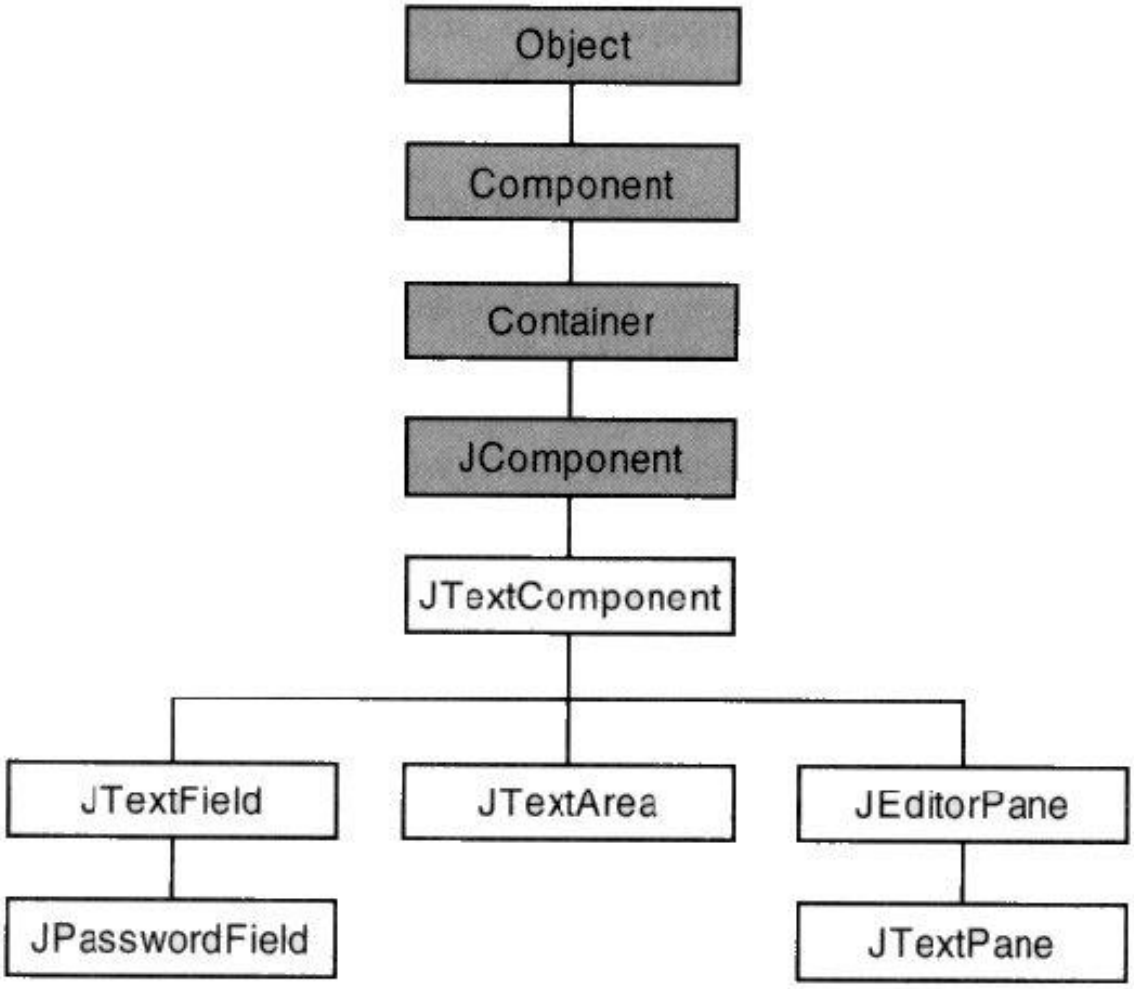

Ein ColorChooser-Beispiel
Swatches
HSB
RGB
H
113
S
B
R 0
G 0
B 0
Preview
Sample Text Sample Text
Sample Text Sample Text
OK
Cancel
Reset


Ein ColorChooser-Beispiel
Swatches
HSB
RGB
Red
81
Green
128
Blue
48
0
85
170
255
Preview
Sample Text Sample Text
Sample Text Sample Text
Sample Text Sample Text
OK
Cancel
Reset

In diesem Kapitel

- *JTextComponent*
- *JTextField*
- *JPpasswordField*
- *JTextArea*
- *JTextPane*

11. *Textkomponenten*



Ob man nun einen einfachen Texteditor wie Notepad oder schon ein kleines Textverarbeitungsprogramm, das über das Niveau des Wordpad-Programms hinausgeht, benutzt, Swing stellt sehr hilfreiche Werkzeuge zur Verfügung - ist es doch sehr mühselig, die Grundfunktionen selbst zu programmieren. Die Entwickler wären gezwungen, das Rad jedesmal neu zu erfinden oder auf kostenpflichtige Komponenten zurück zu greifen. Zum Glück bietet das Swing-Paket sehr mächtige Elemente, die sogar anspruchsvolleren Anwendungen genügen.

11.1. JTextComponet

In der Klasse `JTextComponent` (`javax.swing.text`) sind bereits alle wesentlichen Funktionen und Eigenschaften eines einfachen Texteditors enthalten. Sie ist jedoch eine abstrakte Klasse und wird deshalb selbst nicht instanziert. Wie aus dem Diagramm ersichtlich, ist sie die Grundlage aller Textkomponenten und stellt folgende Dienstleistungen bereit:

- die bekannten Editierfunktionen:

  `cut`() (ausschneiden), `copy`() (kopieren) und `paste`() (einfügen)
  Sie greifen tatsächlich auf die Zwischenablage des jeweiligen Betriebsystems zurück. Es lassen sich somit Texte mit anderen Anwendungen austauschen.

- grundlegende Selektierfunktionen:

  `setSelectionStart`() und `setSelectionEnd`() legen hingegen den markierten Bereich fest, wobei `selectAll`() den gesamten Text markiert. Die ersten beiden Funktionen können auch durch `select`() ersetzt werden. Man gibt einfach die Start- und Endposition an.

- Textbearbeitung:

  mit `setText`() bestimmt man den Inhalt der Textkomponente und bekommt ihn mit `getText`() wieder zurück. Werden bei `getText`() keine Parameter angegeben, so liefert die Funktion den kompletten Inhalt zurück. Es ist jedoch durch Angabe von Startposition und Länge möglich, nur einen Teil des Textinhaltes zu holen.
  `getSelectedText`() liefert den markierten Text bzw. er kann mit `replaceSelection`() durch neuen Text ersetzt werden.

- Funktionen für Eigenschaften:

  `setSelectedTextColor`() und `getSelectedTextColor`() betreffen die Farbe des markierten Textes. `setEditable`() entscheidet, ob Eingaben möglich sind oder nicht. Defaultmässig ist ein Textfeld editierbar.

Hinweis: man darf nicht vergessen, dass auch die Methoden von `JComponent` wie beispielswiese `setForeground`() zum Einstellen der Textfarbe zur Verfügung stehen.

Bei den nun folgenden Klassen handelt es sich um *echte* Textkomponenten, die man in den Anwendungen verwenden kann. Sie alle sind im Paket `javax.swing` enthalten.

11.2. JTextField

`JTextField` ist die einfachste Komponente von Swing: ein simples Textfeld für einzeilige Eingaben. Es wurde eingeführt, um die Abwärtskompatibilität zum AWT zu bewahren und entspricht im grossen und ganzen dem `java.awt.TextField`. Auch sind die meisten Methoden identisch.

Beim Erzeugen kann das Feld bereits mit einem Text vorbelegt und die Anzahl der sichtbaren Spalten definiert werden. Letzteres lässt sich auch mit `setColumns`() bewerkstelligen. Die Ausrichtung bestimmt `setHorizontalAlignment`(). Möglich sind `LEFT, RIGHT` und `CENTER`. Schriftarten lassen sich mit `setFont`() einstellen.

Das Bestätigen der Eingabe durch die Return-Taste löst ein `ActionEvent` aus, das sich von einem `ActionListener` verarbeiten lässt.

Listing 11-1 Verwendung von JTextField

```
package TextField;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;


public class TextField extends JPanel implements ActionListener
{
  JTextField     textField;

  TextField(JApplet applet)
  {
    setLayout(new GridLayout(0, 1, 10, 10));

    // Ein paar Textfelder
    add(new JTextField("Einfaches Textfeld"));
    add(textField = new JTextField("Textfeld mit rotem Text"));
    textField.addActionListener(this);
    textField.setForeground(Color.red);
    add(textField = new
      JTextField("Textfeld mit türkisfarbenen Hintergrund"));
    textField.setBackground(Color.cyan);
    add(textField = new
      JTextField("Textfeld mit gelben Text und blauen Hintergrund"));
    textField.setForeground(Color.yellow);
    textField.setBackground(Color.blue);
    add(textField = new JTextField("Zentriert"));
    textField.setHorizontalAlignment(SwingConstants.CENTER);
    add(textField = new JTextField("Rechtsbündig"));
    textField.setHorizontalAlignment(SwingConstants.RIGHT);
    add(textField = new JTextField("Courier - Fett und Kursiv"));
    textField.setFont(new Font("Courier", Font.BOLD | Font.ITALIC, 12));
  }
```

```
  public void actionPerformed(ActionEvent e)
  {
    System.out.println("ActionEvent");
  }
}
```

Abbildung 11-1 JTextField

11.3. JPasswordField

Das `JPasswordField` ist ein direkter Nachkomme von `JTextField` und eignet sich besonders zur Eingabe geheimer Passwörter. Es bietet gegenüber dem normalen Textfeld nicht viel Neues, ausser dass alle Buchstaben durch ein sogenanntes Echo-Zeichen ersetzt werden. Schliesslich muss der Inhalt eines Passwortes vor den Augen nichtautorisierter Personen verborgen bleiben. Die Echo-Zeichen dienen dem Benutzer lediglich als Kontrolle, ob die Anzahl der Buchstaben übereinstimmt.

Das Echo-Zeichen ist von Grund auf ein * und kann jederzeit mit `setEchoChar`() verändert werden. Natürlich ist das Kopieren des Feldinhaltes in die Zwischenablage des jeweiligen Betriebsystems verboten, doch dürfen Texte von dort eingefügt werden. Das Passwort ist im Programm mit `getPassword`() zu ermitteln.

Listing 11-2 Verwendung von JPasswordField

```
package PasswordField;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;


public class PasswordField extends JPanel implements ActionListener
{
  JPasswordField    passwordField;

  PasswordField(JApplet applet)
  {
    setLayout(new BorderLayout());
    passwordField = new JPasswordField();
    passwordField.addActionListener(this);
    passwordField.setEchoChar('0');

    add("Center", passwordField);
  }

  public void actionPerformed(ActionEvent e)
  {
    System.out.println("Test: " + passwordField.getPassword());
  }
}
```

Abbildung 11-2 JPasswordField Beispiel

11.4. JTextArea

Ein `JTextArea` ist ein mehrzeiliges Textfeld. Neben diesem Hauptunterschied gibt es ein paar Fähigkeiten, die es noch etwas komfortabler machen als das `JTextField`. Mit ihm lässt sich schon ein einfacher Texteditor ohne grossen Auswand entwickeln. Die Kompatibilität zu `java.awt.TextArea` ist auch hier unschwer zu erkennen.

Neben der Anzahl der sichtbaren Spalten mit `setColumns`() lässt sich auch die Zahl der sichtbaren Zeilen mit `setRows`() oder im Konstruktor einstellen. Nützlich erweist sich der automatische Zeilenumbruch, der mit `setLineWrap`() zu aktivieren ist. Es wird dann aber nur jedes Zeichen in die nächste Zeile geschrieben. Mit `setWarpStyleWord`() wird jedes Wort, das nicht mehr in die Zeile passt, in die nächste geschoben. Da `JTextArea` (oder genauer gesagt `JTextComponet`() die `Scrollable` Schnittstelle implementiert, unterstützt sie automatisches Rollen sowohl in vertikaler als auch in horizontaler Richtung. Aktiviert man den Zeilenumbruch, wird das Rollen in horizontaler Richtung natürlich ausgeschaltet. Um das Textfeld noch mit Rollbalken zu versehen, bettet man es einfach in ein `JScrollPane`.

Was noch zu jedem Texteditor gehört, ist eine flexible Tabulatorgrösse. Sie sind mit `setTabSize`() justierbar. `getLineCount`() gibt Auskunft über die Grösse des enthaltenen Textes in Zeilen.

Listing 11-3 Verwendung von JTextArea

```
package TextArea;
import java.awt.*;
import java.io.*;
import java.net.URL;
import javax.swing.*;


public class TextArea extends JPanel
{
  JTextArea      textArea;
  String         areaText;

  TextArea(JApplet applet)
  {
    setLayout(new BorderLayout());

    // TextArea-Komponente
    areaText = loadTextFile("text.txt");
    textArea = new JTextArea();
    textArea.setText(areaText);
    textArea.setLineWrap(true);
    textArea.setWrapStyleWord(true);
    add("Center", new JScrollPane(textArea));
  }

  // Lädt eine Textdatei ein
  public String loadTextFile(String filename)
  {
    String s = new String();
```

```
    File f;
    char[] buff = new char[50000];
    InputStream is;
    InputStreamReader reader;
    URL url;

    try
    {
      int nch;
      url = (new File(filename)).toURL();
      is = url.openStream();
      reader = new InputStreamReader(is);
      while ((nch = reader.read(buff, 0, buff.length)) != -1)
         s = s + new String(buff, 0, nch);
    }
    catch (java.io.IOException ex)
    {
      s = "Datei konnte nicht geladen werden: " + filename;
    }

    return s;
  }
}
```

Abbildung 11-3 JTextArea Anwendungsbeispiel

11.5. JTextPane

Bei dem `JTextPane` handelt es sich um eine wesentlich mächtigere Komponente als die vorhergehend besprochenen Verwandten. Es besitzt einige Funktionen und Eigenschaften, um einer anspruchsvollen Textverarbeitung gerecht zu werden. Das Hauptmerkmal liegt in der Verteilung von Textattributen an beliebig viele Textpassagen. Während man bei `JTextArea` nur Textattribute definieren konnte, die sich auf den gesamten Textinhalt auswirken, lassen sich nun an jeder Stelle im Text die Eigenschaften ändern (wie zum Beispiel Schriftart und -grösse und Farbe). Ausserdem unterstützt `JTextPane` Bilder und Icons.

Textattribute sind vom Typ `MutableAttributeSet` und können einzeln mit Hilfe der Methoden aus `StyleConstants` gesetzt werden. Die Zuordnung der Attribute erfolgt mit `setCharacterAttributes`() (für einzelne Zeichen) und `setParagraphAttributes`() (für Absätze) ab der aktuellen Cursor-Position. Den Cursor positioniert man mit `setCaretPosition`(). Bilder werden mit `insertIcon`() eingefügt. Es ist sogar möglich, mit `insertComponent`() Swing Komponenten in das Textfeld einzubetten. `replaceSelection`() ersetzt den markierten Bereich durch einen anderen Text.

Listing 11-4 Verwendung von JTextPane

```
package TextPane;
import java.awt.*;
import javax.swing.*;
import javax.swing.text.*;


public class TextPane extends JPanel
{
  JTextPane      textPane;
  MutableAttributeSet  attr1, attr2, attr3, attr4;
  ImageIcon      dukeIcon;
  String         textPaneStrings[] =
                  {"In einem JTextPane sind\n", "verschiedene\n",
                   "Fonts\n", "möglich.\n" };

  TextPane(JApplet applet)
  {
    setLayout(new BorderLayout());

    try
    {
      dukeIcon = new ImageIcon("duke.gif");
    }
    catch(SecurityException se)    // Für Internet-Browser
    {
      dukeIcon = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "duke.gif"));
    }

    textPane = new JTextPane();

    attr1 = new SimpleAttributeSet();
    attr2 = new SimpleAttributeSet();
```

```
    attr3 = new SimpleAttributeSet();
    attr4 = new SimpleAttributeSet();

    StyleConstants.setFontFamily(attr1, "Serif");
    StyleConstants.setFontSize(attr1, 14);
    StyleConstants.setFontFamily(attr2, "Serif");
    StyleConstants.setFontSize(attr2, 30);
    StyleConstants.setBold(attr2, true);
    StyleConstants.setUnderline(attr2, true);
    StyleConstants.setFontFamily(attr3, "Courier");
    StyleConstants.setFontSize(attr3, 40);
    StyleConstants.setBold(attr3, true);
    StyleConstants.setFontFamily(attr4, "Arial");
    StyleConstants.setFontSize(attr4, 40);
    StyleConstants.setBold(attr4, true);

    for (int i=0; i<textPaneStrings.length; i++)
      textPane.setText(textPane.getText() + textPaneStrings[i]);
    textPane.setParagraphAttributes(attr1, true);
    textPane.setCaretPosition(textPaneStrings[0].length());
    textPane.setParagraphAttributes(attr2, true);
    textPane.setCaretPosition(
      (textPaneStrings[0]+textPaneStrings[1]).length());
    textPane.setParagraphAttributes(attr3, true);
    textPane.setCaretPosition(
      (textPaneStrings[0]+textPaneStrings[1]+
      textPaneStrings[2]).length());
    textPane.setParagraphAttributes(attr4, true);
    textPane.setCaretPosition(
      (textPaneStrings[0]+textPaneStrings[1]+textPaneStrings[2]+
       textPaneStrings[3]).length());

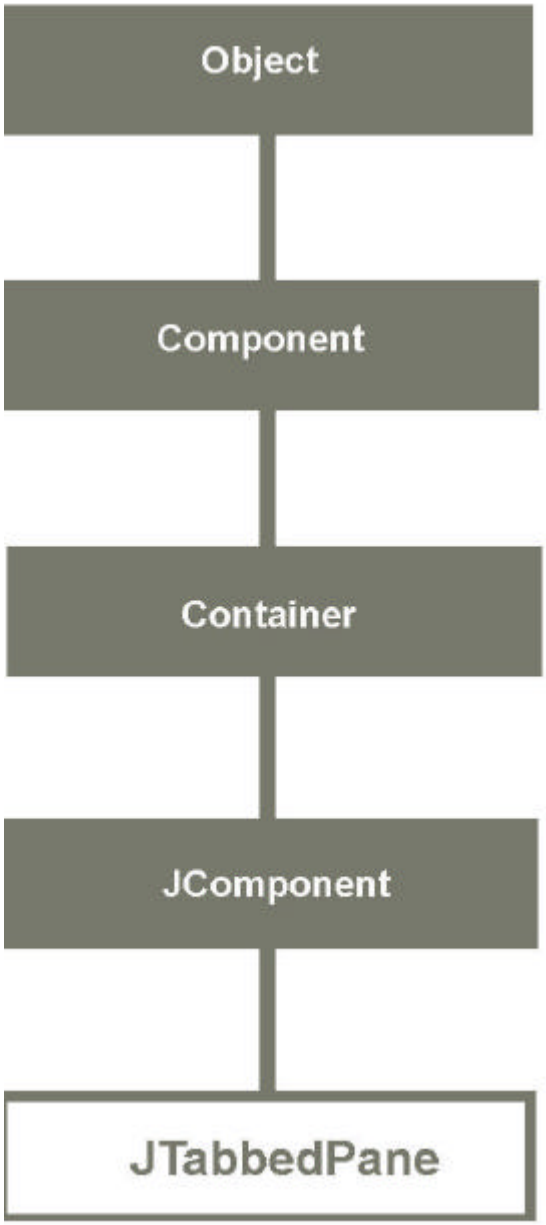
    textPane.insertIcon(dukeIcon);

    add("Center", new JScrollPane(textPane));
  }
}
```

Abbildung 11-4 JTextPane Anwendungsbeispiel

In diesem Kapitel
- *JTabbedPane*
- *JSplitPane*

12.
Registerkarten und Window Splitting



12.1. JTabbedPane

Die Klasse `JTabbedPane` bildet ein Feld von Registerkarten ab. Jede dieser Karten wird mit Etiketten (sogenannten *Tabs*) versehen, die entweder Text, ein Icon oder auch beides beinhalten. Mit den Etiketten kann dann der Benutzer die Karten in den Vordergrund bzw. Hintergrund schieben. Grundsätzlich erlaubt `JTabbedPane` jede beliebige Komponente von Typ `Component` als Registerkarte. Da aber eine Kante meist mehr als nur ein Element umfasst, ist es empfehlenswert, alle Elemente in ein `JPanel` unterzubringen und es in `JTabbedPane` aufzunehmen. Im Prinzip ähnelt es dem `CardLayout` des AWT, nur dass der Benutzer mit Hilfe der Etiketten selbst bestimmen kann, welches Feld oben liegt.

Im Konstruktor oder mit `setTabElement`() kann der Programmierer die Etiketten an eine der vier Kanten des Feldes plazieren. Standardmässig befinden sie sich oben. Neue Karten sind mit `addTab`() hinzuzufügen. Dies ist in mehreren Varianten möglich, so dass neben Panel und Titel auch Icon und ein Tooltip mit übergeben werden können. Will man eine neue Karte an einer bestimmten Seite einfügen, verwendet man `insertTab`() mit Angabe des Index. Mit `setComponentAt`() lassen sich auch Panels an vorgesehenen Stellen austauschen. `getTabCouch`() liefert die Anzahl der Registerkarten.

`JTabbedPane` erzeugt beim Umschalten zwischen den Kanten ein `ChangeEvent`, das mit Hilfe eines `ChangeListener` verarbeitet werden kann.

Listing 12-1 Verwendung von JTabbedPane

```
package TabbedPane;
import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;
public class TabbedPane extends JPanel{
  JPanel         optionPanel, treePanel, panel;
  JTabbedPane    tabbedPane;
  JLabel         pictureLabel;
  ImageIcon      picture, icon;
  DefaultMutableTreeNode treeNode, root;
  TabbedPane(JApplet applet){
    setLayout(new BorderLayout());
    tabbedPane = new JTabbedPane(JTabbedPane.BOTTOM);
    // eine Karte mit Bild
    try    {
      picture = new ImageIcon("seashore.jpg");
      icon = new ImageIcon("picicon.gif");
    }// Für Internet-Browser
    catch(SecurityException se)      {
      picture = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "seashore.jpg"));
      icon = new ImageIcon(
        applet.getImage(applet.getCodeBase(), "picicon.gif"));
    }
    pictureLabel = new JLabel(null, picture, JLabel.CENTER);
    tabbedPane.addTab("Bild", icon,
      new JScrollPane(pictureLabel), "Ein schönes Bild");
    optionPanel = new JPanel();
    optionPanel.setLayout(new GridLayout(1, 2, 10, 5));
    tabbedPane.addTab("Einstellungen", null,
      optionPanel, "Mehrere Einstellungen");
    optionPanel.add(panel = new JPanel());
    panel.setLayout(new FlowLayout(FlowLayout.CENTER));
```

```
    panel.add("Center", panel = new JPanel());
    panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
    for (int i=1; i<=7; i++)
      panel.add(new JCheckBox("Option " + i));
    optionPanel.add(panel = new JPanel());
    panel.setLayout(new GridLayout(0, 1, 10, 5));
    panel.add(new JButton("Allgemein..."));
    panel.add(new JButton("Ansicht..."));
    panel.add(new JButton("Drucker..."));
    panel.add(new JButton("Speichern..."));
    panel.add(new JButton("Format..."));
    // mit Baum
    treePanel = new JPanel();
    treePanel.setLayout(new BorderLayout());
    root = new DefaultMutableTreeNode("Baum");
    treePanel.add("Center", new JTree(root));
    tabbedPane.addTab("Baum", null, new JScrollPane(treePanel),
      "Ein Verzeichnisbaum");
    root.add(treeNode =
      new DefaultMutableTreeNode("Verzeichnis 1"));
    for (int i=1; i<=10; i++)
      treeNode.add(new DefaultMutableTreeNode("Datei " + i));
    root.add(treeNode =
      new DefaultMutableTreeNode("Verzeichnis 2"));
    for (int i=1; i<=10; i++)
      treeNode.add(new DefaultMutableTreeNode("Datei " + i));
    root.add(treeNode =
      new DefaultMutableTreeNode("Verzeichnis 3"));
    for (int i=1; i<=10; i++)
      treeNode.add(new DefaultMutableTreeNode("Datei " + i));

    add("Center", tabbedPane);
  }
}
```

Abbildung 12-1 JTabbedPane Beispiel, mit ToolTip (Ein Verzeichnisbaum)

12.2. JSplitPane

Mit `JSplitPane` können zwei Komponenten voneinander getrennt werden. Der Trennbalken kann vom Benutzer entweder horizontal oder vertikal verschoben werden, er bestimmt die Grössenverhältnisse der Komponenten (zum Beispiel einer Liste oder eines Verzeichnisbaumes) auf der linken Seite, während rechts die Komponente steht, die für das Anzeigen des Inhalts der Auswahl zuständig ist. Ein bekanntes Beispiel ist der Windows Explorer.

Man legt mit `setOrientation`() fest, ob die Komponenten horizontal (`JSplitPane.HORIZONTAL_SPLIT`) oder vertikal (`JSplitPane.VERTICAL_SPLIT`) zu trennen sind. Dabei bedeutet die horizontale Trennung, dass die beiden Komponenten links und rechts angeordnet sind und sich der Trennbalken horizontal verschieben lässt. Bei der vertikalen Trennung liegen sie dagegen übereinander. Je nachdem, wie man sich entschieden hat, sind nun die Komponenten mit den Methoden `setLeftComponent`(), `setRightComponent`() und `setBottomComponent`() zu definieren. Soll das Layout der Komponenten schon während des Verschiebens des Trennbalkens erneuert werden, kann der Programmierer dies mit `setContiunousLayout`() anordnen. Sind alle Parameter schon zum Programmstart bekannt, können alle vorhergehenden Aktionen mit einem Schlag im Konstruktor angegeben werden.

Die Methode `setDriverLocation`() setzt den Trennbalken seitens des Programms an die gewünschte Position. Dabei steht es dem Programmierer frei, eine prozentuale Angabe durch eine Gleitkommazahl zwischen null und eins oder mit der absoluten Anzahl der Pixel vorzunehmen. Bei der Ermittlung der Position mit `getDriverLocation`() erhält er dagegen den Wert nur in Pixel.

Als kleine Besonderheit lässt sich der Trenner mit zwei kleinen Pfeilen ausstatten. Wird er betätigt, springt er zur rechten oder linken bzw. oberen oder unteren Kante des Feldes. Die Methode heisst `setOneTouchExpandable`().

Listing 12-2 Verwendung von JSplitPane

```
package SplitPane;
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.tree.*;


public class SplitPane extends JPanel implements
  TreeSelectionListener
{
  JSplitPane     splitPane;
  JTree          simpleTree;
  DefaultMutableTreeNode  javaRoot;
  DefaultMutableTreeNode  classNode;
  JScrollPane    scrollPane;
  JTextArea     textArea;

  String    AWTNames[] =
  {"Button", "Canvas", "CardLayout", "Checkbox", "CheckboxGroup",
   "CheckboxMenuItem", "Choice", "Color", "Component",
```

```
    "Container", "Cursor", "Dialog", "Dimension", "Event",
    "EventQueue", "FileDialog", "FlowLayout", "Font",
    "FontMetrics", "Frame", "Graphics", "GridBagConstraints",
    "GridBagLayout", "GridLayout", "Image", "Insets", "Label",
    "etc."};
  String    JFCNames[] =
  {"JCheckBox", "JCheckBoxMenuItem", "JComboBox", "JComponent",
    "JDesktopPane", "JDialog", "JEditorPane", "JFrame",
    "JInternalFrame", "JLabel", "JLayeredPane", "JList","JMenu",
    "JMenuBar", "JMenuItem", "JOptionPane", "JPanel",
    "JPasswordField", "JPopupMenu", "JProgressBar","JRadioButton",
    "JRadioButtonMenuItem", "JRootPane", "JScrollBar",
    "JScrollPane", "JSeparator", "JSlider", "JSplitPane",
    "JTabbedPane", "JTable", "JTextArea", "JTextField", "etc."};


  SplitPane(JApplet applet)
  {
    setLayout(new BorderLayout());

    splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
      false);
    add("Center", splitPane);

    // Auf der linken Seite befindet sich der Klassenbaum
    javaRoot = new DefaultMutableTreeNode("Java Classes");
    javaRoot.add(classNode = new DefaultMutableTreeNode("AWT"));
    for (int i=0; i<28; i++)
      classNode.add(new DefaultMutableTreeNode(AWTNames[i]));
    javaRoot.add(classNode = new DefaultMutableTreeNode("JFC"));
    for (int i=0; i<33; i++)
      classNode.add(new DefaultMutableTreeNode(JFCNames[i]));
    javaRoot.add(new DefaultMutableTreeNode("IFC", false));
    simpleTree = new JTree(javaRoot);
    simpleTree.addTreeSelectionListener(this);
    scrollPane = new JScrollPane(simpleTree);
    splitPane.setLeftComponent(scrollPane);

    // Auf der rechten Seite befindet sich ein Textfeld
    textArea = new JTextArea("");
    textArea.setEditable(false);
    textArea.setFont(new Font("Courier", Font.BOLD, 16));
    splitPane.setRightComponent(textArea);
    splitPane.setOneTouchExpandable(true);

    splitPane.setDividerLocation(0.4);
  }

  public void valueChanged(TreeSelectionEvent e) {
    TreePath p = e.getPath();

    if (e.getPath().getPathCount() == 3)    {
      textArea.setText("Klasse:  " +
        p.getLastPathComponent().toString()
        + "\n\n" + "Hier kann die Dokumentation stehen.");
    }
    else
      textArea.setText("");
  }
}
```

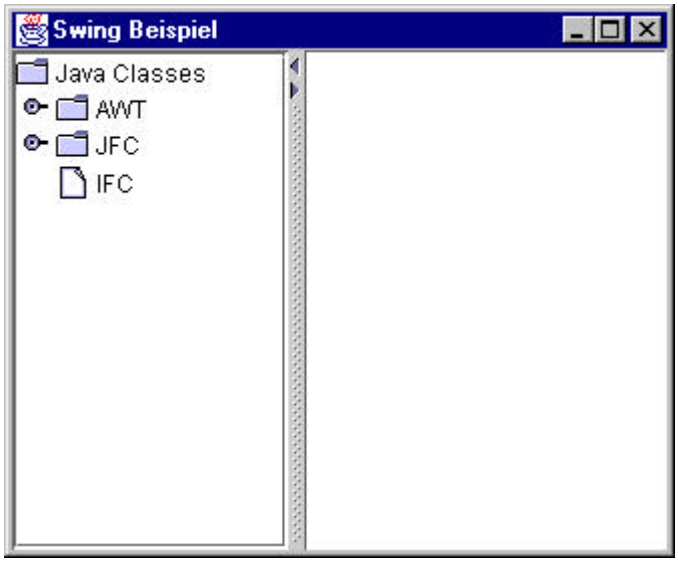


Abbildung 12-2 JSplitPane Anwendungsbeispiel

In diesem Kapitel
- *Swing und MVC*
- *JTree*
- *TreeNode*
- *MutableTreeNode*
- *TreeModel*
- *TreeSelectionModel*
- *TreeCellRender*

13. Model-View-Controller

Es gibt verschiedene Möglichkeiten, mit Swing grafische Benutzeroberflächen zu gestalten. Viele Swing Komponenten lassen sich dabei ebenso ansprechen, als wären sie AWT Komponenten. Wenn man dieses Verständnis als Grundlage für das MVC Konzept anwendet, ist es ein leichtes dieses Konzept zu erlernen.

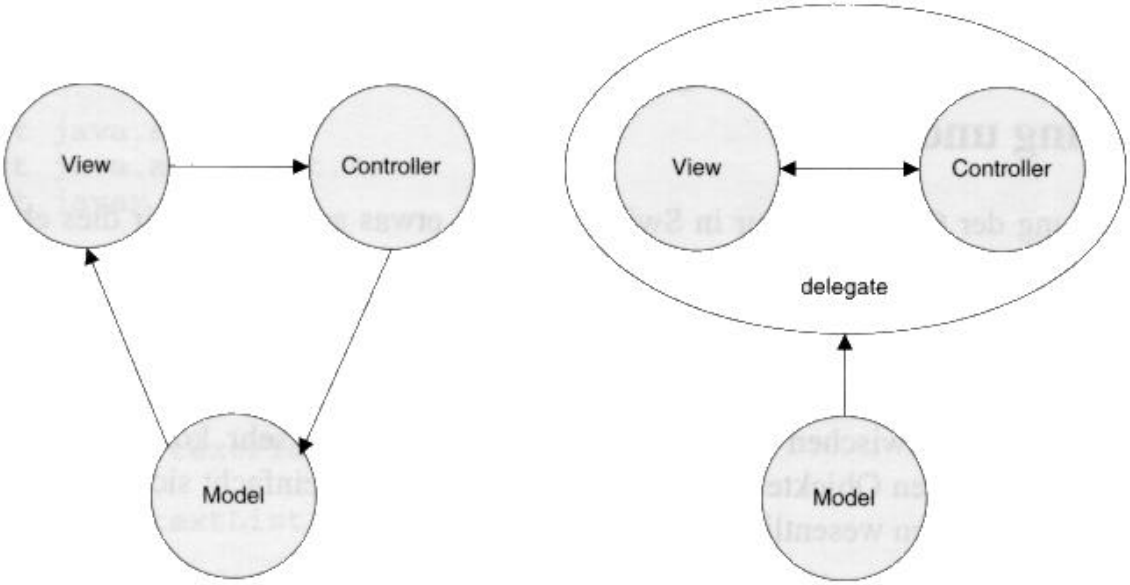


Abbildung 13-1

MVC in der Theorie und in Swing

Durch die Einführung einer Technologie, welche auch von der Programmiersprache Smalltalk her bekannt ist und besonders bei einer rein objektorientierten Sprache ihre Vorzüge ausspielen kann, gibt Swing dem Programmierer ein weitaus mächtigeres Werkzeug in die Hand, als es vom AWT bekannt ist: die MVC Architektur.

Dem Programmierer wurde dadurch die Möglichkeit gegeben, das Aussehen eines Widgets und dessen Reaktion auf Eingaben zu kontrollieren. Gerade bei komplexeren Widgets (z.B. Tabellen) ist diese Möglichkeit von grosser Bedeutung.

Im Folgenden werden wir einen Einblick gewinnen in die Funktionsweise des MVC (neben der grösseren Auswahl an Komponenten ist die Architektur des MVC ein weiterer Vorteil von Swing).

Wie schon erwähnt, enthalten auch andere Programmiersprachen dieselbe Architektur. Drei Kommunikationsobjekte stehen hier miteinander in Verbindung: das Modell, die Sicht / View und der Controller. Das Modell stellt die dahinterliegende logische Struktur dar. Die Sicht / View ist die visuelle Darstellung der Informationen und der Controller übernimmt die Verarbeitung der Benutzereingaben.

Durch diese Dreiteilung der einzelnen Objekte ist ein Höchstmass an Flexibilität gewährleistet. Da alle Objekte untereinander in Beziehung stehen, werden bei einer Änderung des Modells die mit ihm verbundenen Objekte benachrichtigt. Durch diese Teilung der Datenhaltung und deren Präsentation werden zwei wichtige Sachverhalte klar:

- ein Datenmodell kann in mehreren Views auf die verschiedenste Art und Weise dargestellt werden. Zum Beispiel ist es auf diese Weise möglich, eine Ansicht der Daten in Tabellenform und in einer Balkengrafik darzustellen.
- es ist möglich eine Ansicht zu ändern, ohne dabei die Datenhaltung zu beeinträchtigen. Eine View kann beliebig verändert werden, und das darunterliegende Datenmodell bleibt davon unbeeinträchtigt.

Ein View (Objekt) benutzt einen Controlle (ein Controller Objekt) , um auf Benutzereingaben zu reagieren, zum Beispiel bei einer Eingabe per Tastatur oder aufgrund der Bewegung eines anderen Eingabegerätes.

13.1. Swing und MVC

Die Darstellung der Objektstruktur in Swing ist dabei etwas anders, als wir dies eben kennen gelernt haben. Swing verwendet eine häufig benutzte Variante der MVC Architektur. Swing verbindet den Controller und die Sicht (View) in einem Objekt und nennt es Delegate. Ein Delegate ist somit beides (Controller und View); es stellt das Datenmodell dar, wie es eine Sicht darstellen würde, und verarbeitet die Benutzereingaben wie ein Controller. Die Kommunikation zwischen View und Controller ist dabei sehr komplex.

Durch die Kommunikation der beiden Objekte in einem Delegate-Objekt vereinfacht sich das Design der einzelnen Komponenten wesentlich.

Stellen Sie sich zum Beispiel ein Checkbox Widget vor: unabhängig von der visuellen Darstellung besitzt es einen Wert, der entweder `true` oder `false` sein kann. Dies korrespondiert mit dem Checkbox Modell. Wie diese beiden Stati angezeigt werden, hängt von Delegate-View ab.

Klickt der Benutzer mit der Maus auf die Checkbox, ist der Delegate Controller dafür zuständig, dass das Modell benachrichtigt wird. Normalerweise ist dem Delegate Objekt in diesem Fall auch eine Checkbox zugeordnet, und diese zeigt dann die Zustandänderung an.

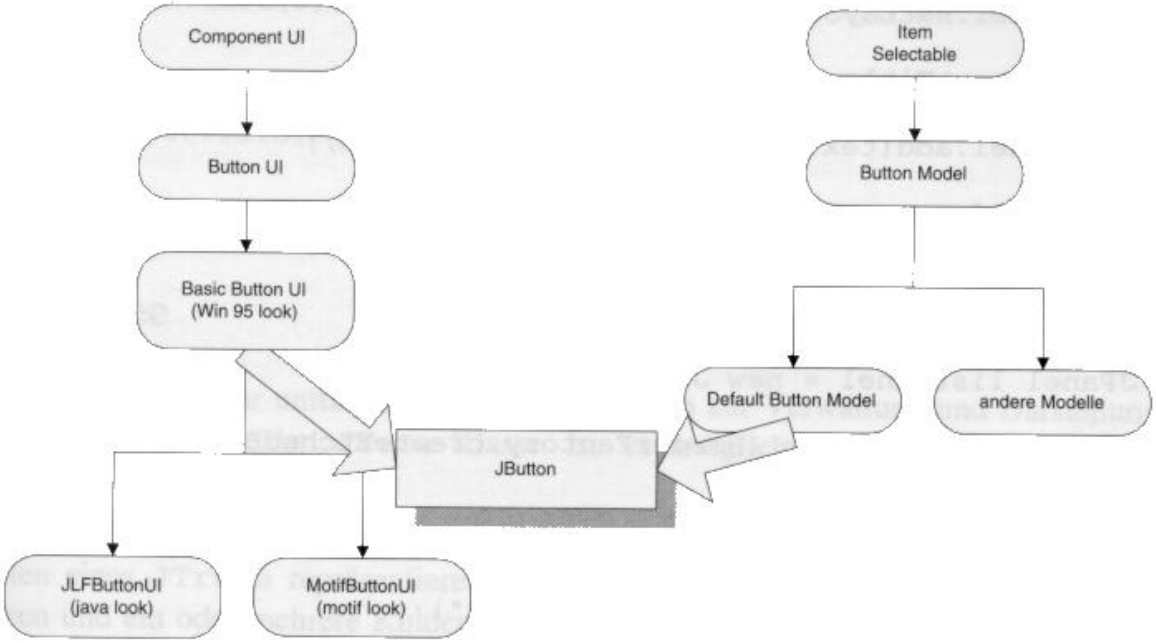


Abbildung 13-2 Zusammenhang zwischen Look & Feel und Modell am Beispiel von JButton

Listing 13-1 Beispiel einer MVC Anwendung

```
//Titel:        MVC Implementation
//Beschreibung:  MVC Implementation
package MVCImplementation;

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

public class SimpleEvent extends JFrame {
  static final int WIDTH = 150;
  static final int HEIGHT= 180;

  JTextField textField;
  JTextArea  textList;
  JScrollPane pane;
  // Konstruktor
  SimpleEvent(String lab) {
    super(lab);
    // alte Version
    // setLayout(new FlowLayout() );
    // neue Version
    getContentPane().setLayout(new FlowLayout() );
    setBackground(Color.lightGray);

    JPanel textPanel = new JPanel();
    textPanel.setBorder(BorderFactory.createEtchedBorder() );
    textPanel.setLayout(new BorderLayout() );
    JLabel textTitle = new JLabel("einfach Text eingeben und <RET>");
    textPanel.add(textTitle, BorderLayout.NORTH);
```

```
    textField = new JTextField();
    textPanel.add(textField, BorderLayout.SOUTH);
    textPanel.add(Box.createVerticalStrut(6) );
    JPanel listPanel = new JPanel();
    listPanel.setBorder(BorderFactory.createEtchedBorder() );
    listPanel.setLayout(new BoxLayout(listPanel, BoxLayout.Y_AXIS));
    JLabel title = new JLabel("Text Liste");
    listPanel.add(title);
    listPanel.add(Box.createVerticalStrut(10));
    textList = new JTextArea("", 6, 10);
    textList.setEnabled(false);
    pane = new JScrollPane(textList);
    listPanel.add(pane);
    listPanel.add(Box.createVerticalStrut(6));

    textField.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        textList.append(textField.getText() );
        textList.append("\n");
        pane.validate();
        textField.setText("");
     }});

    Container c = getContentPane();
    c.setLayout(new FlowLayout() );
    c.add(textPanel);
    c.add(Box.createHorizontalStrut(30));
    c.add(listPanel);
  }

  public static void main(String args[]) {
    SimpleEvent frame = new SimpleEvent("Einfaches Ereignis Beispiel");
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e){
        System.exit(0);
     }});
    frame.setSize(WIDTH, HEIGHT);
    frame.setVisible(true);
  }
}
```

Abbildung 13-3 Beispiel für eine einfache Ereignissteuerung :Eingabe in das Textfeld wird in die Textfläche übernommen

13.2. JTree

Swing besitzt ein sehr umfangreiches Set an Klassen zur Verwaltung und Darstellung von Bäumen. Mit Hilfe der `JTree`-Klasse ist es möglich, hierarchische Datenstrukturen darzustellen.

Ein `JTree` besteht dabei aus `TreeNode` Objekten, welche die einzelnen Äste oder Knoten eines `JTree` repräsentieren. Ein `TreeNode` besitzt keinen oder einen Elternknoten und ein oder mehrere Kinder Knoten. Die grosse Anzahl von Methoden der Klasse `DefaultMutableTreeNode` dient zur Manipulation von Knoten eines Baumes. Die wichtigsten Methoden lernen wir im Folgenden kennen.

13.2.1. TreeNode

Ein `TreeNode` Objekt legt die Eigenschaften eines Objekts innerhalb eines `JTree` fest. Die Methode `getParent`() liefert dabei den Eltern Knoten des Objekts zurück. Um alle Knoten in einer `Enumeration` Klasse durchgehen zu können, gibt es die Methode `children`(). Falls der Knoten keine weiteren Kinder besitzt, liefert die Methode `isLeaf`() den Wert true zurück. Die Methode `getChildrenAt`() liefert einen Knoten an einer bestimmten Position zurück. Es gibt jedoch noch weitere Methoden in diesem Interface `TreeNode`, die jedoch eher selten verwendet werden.

13.2.2. MutableTreeNode

Auch hier handelt es sich um ein Interface, welches von dem Interface `TreeNode` abgeleitet ist. `MutableTreeNode` ist für das Hinzufügen und Entfernen von `TreeNode` Objekten verantwortlich. Mit der Methode `insert`() wird ein neuer Knoten hinzugefügt. Als Parameter kann dabei noch der Index angegeben werden, wo genau der Knoten eingefügt werden soll.

Die Methode `remove`() gibt es mit Parameter Index und mit Angabe eines Knotenobjekts. Sie dient zum Entfernen von Knoten aus dem JTree. Eine Möglichkeit um einen gesamten Knoten mit Kindern in der Hierarchie zu verschieben, bietet die Methode `setParent`().Als Parameter wird der neue Knoten angegeben.

13.2.3. DefaultMutableTreeNode

Die Klasse `DefaultMutableTreeNode` benutzt das Interface `MutableTreeNode`. Die Methoden dieser Klasse lassen jede Art von Manipulation der `TreeNode` Objekte zu. Die Klasse selbst ist nicht thread-safe. Wird mit mehreren Threads gearbeitet, muss sich der Entwickler selbst um die Synchronisation kümmern. Die Methode `add`() fügt ein `MutableTreeNode` Objekt in einen `JTree` ein. Um die Wurzel des `JTree` zu bekommen, wird die Methode `getRoot`() verwendet. Sie liefert ein `TreeNode` Objekt zurück. Dieses Objekt kann mit der Methode `toString`() auch angezeigt werden. Die Methode `getLevel`() liefert die Hierarchiestufe des momentan verwendeten Knotens bis zur Wurzel.

Um das Modell und die Sicht eines `JTree` vollständig zu beschreiben, arbeiten drei verschiedene Schnittstellen zusammen - `TreeModel, TreeSelectionModel` und `TreeCellRenderer`.

13.2.4. TreeModel

`TreeModel` beschreibt die Datenstruktur des `JTree`. Die Klasse `JTree` stellt zwei Methoden zur Verfügung, mit denen eine Datenstruktur gesetzt werden kann. Die Klasse `TreeModel` legt dabei mit den folgenden Methoden fest, wie ein `JTree` eine Datenstruktur zu interpretieren hat.

- `getChild(Object parent, int index)`
  Die Methode liefert das entsprechende Objekt aus dem `Tree` Modell zurück

- `getChildCount(Object paranet)`
  liefert die Anzahl der Knoten zurück, die innerhalb des angegebenen Objekts existieren

- `getIndexOfChild(Object patent, Object child)`
  liefert den Index eines Knotens im `Object` Parent

- `getRoot()`
  liefert das Root Objekt des Modells zurück

- `isLeaf(Object node)`
  falls das angegebene Objekt ein leerer Knoten ist, d.h. keinen weiteren Knoten besitzt, wird hier `true` zurück gegeben.

Um den `JTree` wissen zu lassen, ob der Datenstruktur ein Element hinzugefügt worden ist, sind die Methoden `addTreeModelListener`() und `removeTreeModelListener`() zuständig.

Verfügt eine Klasse über die oben angegebenen Methoden, kann sie als `TreeModel` für `JTree` Komponenten verwendet werden. Die Klasse `DefaultTreeModel` ist eine einfache Implementierung dieser `TreeModel` Schnittstelle.

13.2.5. TreeSelectionModel

Die Schnittstelle `TreeSelectionModel` entspricht dem Controller im MVC Konzept. Sie ist dafür zuständig, wie die Kommunikation mit den angezeigten Daten vonstatten geht. Auch dieses Interface bietet einige interessante Methoden:

- `clearSelection`()
  hebt die Selektion eines oder mehrerer Knoten im `JTree` auf.

- `getSelectionMode`()
  liefert den Selektionsmodus zurück

- `setSelectionMode(int Mode)`
  dem Interface `TreeSelectionModel` stehen drei unterschiedliche Arten der Selektion zur Verfügung. Diese können entweder nur einen Pfad selektieren (`SINGLE_TREE_SELECTION`), oder es ist nur eine zusammenhängende Selektion möglich (`CONTIGOUS_TREE_SELECTION`), oder es können beliebige Pfade selektiert werden (`DISCONTIGOUS_TREE_SELECTION`).

- `isRowSelected(int row)`
  diese Methode gibt `true` zurück, falls ein Eintrag selektiert ist, der mit dem Übergabeparameter `row` übereinstimmt.

Die Klasse `SimpleSelectionModel` ist eine Implementierung der `TreeSelectionModel` Schnittstelle. Sie enthält eine vollständige Implementierung aller gängiger Varianten der Selektion von Komponenten.

13.2.6. TreeCellRender

Die Schnittstelle `TreeCellRenderer` wird von `JTree` benutzt, um Daten zu visualisieren. Dies entspricht in unserer Architektur der Sicht (View) eines Objekts. Der in jedem `JTree` standardmässige `TreeCellRenderer` wird durch die Klasse `DefaultTreeCellRenderer` implementiert. Um eine eigene Sicht der Dinge zu bekommen, muss die Klasse nur das Interface `TreeCellRenderer` implementieren. Diese Schnittstelle besitzt nur eine Methode, die es zu überschreiben gilt. Die Methode lautet:

```
getTreeCellRendererComponent( JTree tree,
                              Object value,
                              boolean selected,
                              boolean expanded,
                              boolean leaf,
                              int row,
                              boolean hasFocus);
```

Die übergebenen Parameter sind selbsterklärend, deshalb wird nicht weiter darauf eingegangen.

Das folgende Beispiel zeigt die Implementierung eines TreeCellRenderer.

Listing 13-2 Implementierung eines TreeCellRenderer

```
package TreeCellRender;
import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;

class SimpleTreeCellRenderer implements TreeCellRenderer {
      public Component getTreeCellRendererComponent(JTree tree,
            Object value,
            boolean selected,
            boolean expanded,
            boolean leaf,

            int row,

            boolean hasFocus) {
            ImageIcon icon = new ImageIcon("world.gif");
            JLabel label = new JLabel(value.toString(),icon,0);
            return label;
      }
}
```

Dieser Teil des Listings ist für die Initialisierung des `JTrees` und für das Erstellen des `SimpleTreeCellRender` verantwortlich. Mit der Methode `setCellRenderer`() kann jeder andere `Renderer` in einen `JTree` gesetzt werden.

Listing 13-3 Initialisieren des JTree

```
// Beispielprogramm für Treecontrols
package TreeCellRender;
import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;

public class Tree extends JPanel {
  JTree     simpleTree;
  DefaultMutableTreeNode      javaRoot;
  DefaultMutableTreeNode  classNode;
  JScrollPane scrollPane;

  String    JFCNames[] = {"JCheckBox", "JCheckBoxMenuItem", "JComboBox",
"JComponent", "JDesktopPane", "JDialog", "JEditorPane", "JFrame",
"JInternalFrame", "JLabel", "JLayeredPane", "JList","JMenu", "JMenuBar",
"JMenuItem", "JOptionPane", "JPanel", "JPasswordField", "JPopupMenu",
"JProgressBar","JRadioButton", "JRadioButtonMenuItem", "JRootPane",
"JScrollBar", "JScrollPane", "JSeparator", "JSlider", "JSplitPane",
"JTabbedPane", "JTable", "JTextArea", "JTextField", "etc."};

    Tree(JApplet applet)     {
    setLayout(new BorderLayout());
    add("North", new JLabel("Konkurierende Klassen"));
    javaRoot = new DefaultMutableTreeNode("Java Classes");
    javaRoot.add(classNode = new DefaultMutableTreeNode("JFC"));
    for (int i=0; i<33; i++)
      classNode.add(new DefaultMutableTreeNode(JFCNames[i]));
    simpleTree = new JTree(javaRoot);
    simpleTree.setCellRenderer(new SimpleTreeCellRenderer());
    scrollPane = new JScrollPane(simpleTree);
           add("Center", scrollPane);
      }
}
```

Und so sieht das Ganze aus (Kommentare zum Icon sind überflüssig):

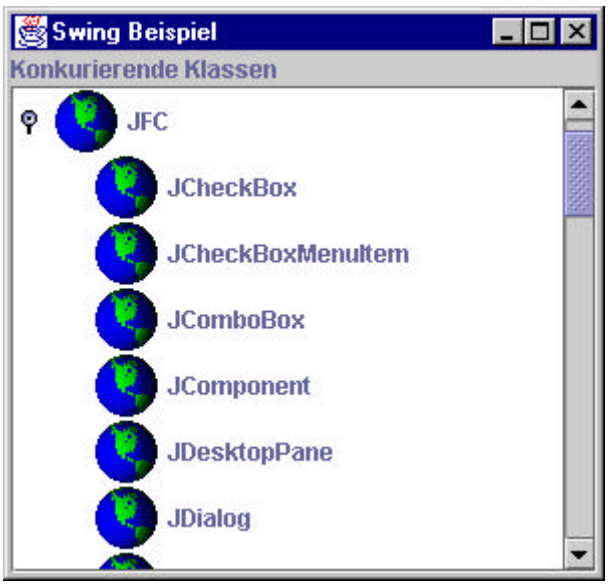


Abbildung 13-4 JTree mit eigenem TreeCellRendering

13.3. Drucken

Drucken war in Java lange Zeit eine schwierige Sache. Seit Java 2 wird dies anders. Das neue JDK bringt Flexibilität un Möglichkeiten, die man in älteren JDKs lange vermisst hatte oder die nur unzureichend vorhanden waren. Das neue Package `java.awt.print` bietet die Möglichkeit, alles zu drucken, was auf einen Grafik-Kontext zu zeichnen ist, einschliesslich Swing Komponenten und 2D Grafik.

Die neuen APIs sind sehr einfach in ihrer Anwendung. Dabei legt die Applikation fest, was zu drucken ist, und das Drucksystem entscheidet, wann jede einzelne Seite soweit ist. Sie funktioniert dabei nach einem Callback Prinzip. Dieses Prinzip erlaubt es dem Benutzer ausserdem, eine Grafik auf einem Drucker auszugeben, welcher selbst nicht über genügend Speicherplatz verfügt.
Da Swing Komponenten bereits durch ein Graphics Objekt gezeichnet werden, welches von AWT unterstützt wird, ist es einfach, diese Komponenten zu zeichnen. Da AWT Komponenten nicht mit Hilfe eines Grafik Contextes gezeichnet werden, muss diese Funktionalität erst implementiert werden. Dies geschieht über die Verwendung einer Schnittstelle namens `Printable`.

13.1.1. Drucken einer Swing Komponente

Um nun eine Swing Komponente in Java zu drucken, braucht man keine `paint`() Methode zu implementieren, wie dies beim Ausdrucken von AWT Komponenten der Fall wäre, sondern nur die Methode `print`(). Dies rührt daher, dass sich Swing Komponenten selbst zeichnen können.

Im folgenden Beispiel implementieren wir ein druckbares `JPanel`.

Listing 13-4 Druckbare JPanel

```
package MVCPrint;
import java.awt.*;
import java.awt.event.*;
import java.awt.print.*;
import javax.swing.*;

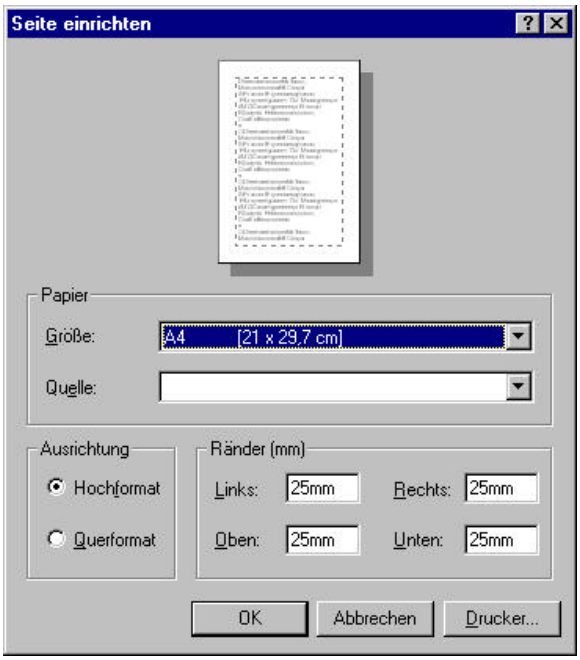
class PrintPanel extends JPanel implements Printable {

      JButton btnDialog;
      JPanel instance = this;
      public PrintPanel() {
            super();
            btnDialog = new JButton("Print");
            add(btnDialog);
            btnDialog.addActionListener(new MyListener());
      }
      private class MyListener implements ActionListener{
            public void actionPerformed(ActionEvent e) {
                  PrinterJob printJob = PrinterJob.getPrinterJob();
                  printJob.setPrintable((PrintPanel) instance);
                  PageFormat pf =
printJob.pageDialog(printJob.defaultPage());

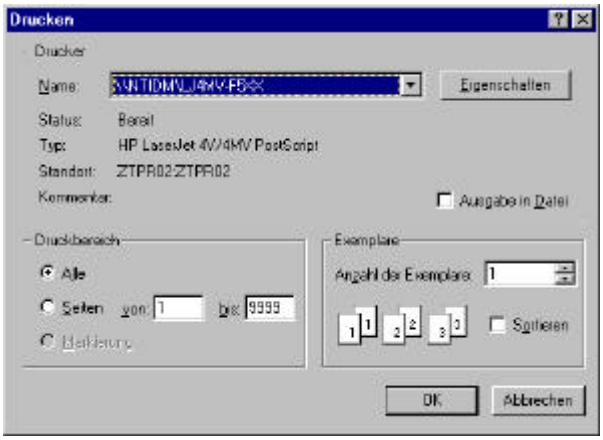
                  if(printJob.printDialog()) {
                        try {
                              printJob.print();
```

```
                        } catch(Exception PrinterException) {
                              System.out.println(PrinterException);
                        }
                  }
            }
      }
      public int print(Graphics g, PageFormat pf, int pi) {
            if (pi>=1) {
                  return Printable.NO_SUCH_PAGE;
            }
            Graphics2D graphics = (Graphics2D)g;
            graphics.translate(pf.getImageableX(),pf.getImageableY());
            paint(graphics);
            return Printable.PAGE_EXISTS;
      }
}
```

Und hier die Abfolge des Print Dialogs:



und schliesslich das Windows Print Fenster



13.3.1. Drucken mehrerer Seiten

Um auch mehrere Seiten auf komfortable Art und Weise drucken zu können, wurde in JDK die Klasse Book eingeführt. Die Seiten können in unterschiedlicher Reihenfolge angeordnet sein und unterschiedliche Formate aufweisen.

Die Klasse Book nimmt mit der Methode append() verschiedene Seiten auf. Für jede hinzugefügte Seite kann zuvor mit setOrientation() deren Ausrichtung bestimmt werden.

Beispiel für einen Codeausschnitt:

Listing 13-5 Druckjob Programmfragment

```
private class MeinBookListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
        PrinterJob printJob = PrinterJob.getPrinterJob();
        PageFormat landscape = printJob.defaultPage();
        PageFormat portrait = printJob.defaultPage();
        landscape.setOrientation(PageFormat.LANDSCAPE);
        portrait.setOrientation(PageFormat.PORTRAIT);

        Book book = new Book();
        book.append(Printable)Panel1.landscape);
        book.append(Printable)Page2.portrait,2);

        PrintJob.setPageable(ok);
        PageFormat pf = printJob.pageDialog(printJob.defaultPage() );

        if (printJob.printDialog() ) {
                try {
                        printJob.print();
                } catch (Exception PrinterException) {
                        System.err.println(PrinterException);
                }
        }}
}
```

Der Listener druckt dabei ein Buch, welches zwei Panels enthält. Panel 1 wird einmal ausgedruckt, Panel 2 zweimal, da dem append() ein weiterer Parameter zur Seitenanzahl übergeben wurde.

13.4. Look & Feel

AWT und Swing benutzen beide die gleiche Architektur und entkoppeln beide den zeichnenden Teil der Benutzeroberfläche von dem Teil, der die Benutzeroberfläche erstellt. Das Abstract Windows Toolkit (AWT) geht allerdings einen andern Weg; es benutzt sogenannte Peers, welche zwischen dem Betriebsystem und der Darstellung eines Objekts geschaltet sind. Dieser Ansatz wurde jedoch aufgrund vieler Einschränkungen nicht mehr weiter verfolgt, AWT hat hier einfach in eine Sackgasse geführt.

Swing bietet hingegen einen andern Ansatz. Swing Komponenten benutzen keine betriebsystemabhängigen Peers, sondern sind für das Zeichnen der jeweiligen Komponente (den Look) selbst zuständig. Dies ermöglicht es, verschiedene Looks auf einem Betriebsystem darzustellen und - was noch wichtiger ist - Komponenten unabhängig vom Betriebsystem zur Verfügung zu stellen. Da im MVC Modell von Swing View und Controller (Look & Feel) eine Einheit bilden, spricht man von Look & Feel.

Swing bietet die Möglichkeit, View und Controller (Look & Feel) während der Laufzeit eines Programms zu ändern und den jeweiligen Anforderungen des Benutzers somit anzupassen. Dies ist ein grosser Schritt, um vom Betriebsystem unabhängig zu werden. Der Benutzer kann das Aussehen seiner Applikation so gestalten, wie er es gewohnt ist. Ein Benutzer von Solaris kann somit auch unter dem MacIntosh Betriebsystem sein Motif Look & Feel beibehalten , die Frage ist warum er das möchte!

Folgende Look & Feels sind bereits in Java enthalten:

13.4.1.1. Java Look & Feel

Dieses Look & Feel wurde von Sun für Java entwickelt. Es stellt sozusagen das Standardlayout für Anwendungen in Swing dar. Dieser Style wird auch Metal-Look & Feel genannt und erlaubt dem Benutzer, durch die Auswahl verschiedener Themen, Schriften und Farbschemata für seine Applikation, das Lok & Feel zu wählen. Er bietet auch hinsichtlich Trees, ToolBars und Sliders Erweiterungen, die in andern Look & Feels nicht vorkommen.

13.4.1.2. Windows Look & Feel

Das Windows Look & Feel erlaubt die Darstellung von Swing Anwendungen im Windows Style. Windows Look & Feel darf aus rechtlichen Gründen allerdings nur auf Windows Plattformen Verwendung finden.

13.4.1.3. Motif Look & Feel

Motif ist vorallem unter Anwendern von Unix Betriebsystemen bekannt. Dabei handelt es sich um eine Benutzerschnittstelle, die als Aufsatz auf das normale X-Windows im Unix Bereich dient, um dem Benutzer eine einheitliche Schnittstelle zur Verfügung zu stellen. Swing stellt dieses Look & Feel allen Platformen zur Verfügung.

13.4.1.4. MacIntosh Look & Feel

Dieses Look & Feel stammt von Apple und ist für den MacIntosh bestimmt. Auch hier darf aus rechtlichen Gründen keine Verwendung auf anderen Betriebsystemen stattfinden.

Diese Look & Feels sind bereits bei der Auslieferung eines JDKs implementiert. Je nach Betriebsystem fehlt dabei das eine oder andere Look & Feel.

13.5. Ändern von Look & Feel

Das Ändern des Look & Feel ist in Java sehr einfach. Folgende Programmzeilen erledigen ein Umschalten des Look & Feel in Swing:

```
UIManager.setLookAndFeel(String str);
SwingUtilities.updateComponentTreeUI(this);
```

Die erste Programmzeile setzt das neue Look & Feel durch Angabe eines Strings. Die zweite Zeile führt einen Update des Look & Feels durch. Im folgenden Programm wird mit Hilfe eines PopUp Menüs das Look & Feel eines `JTree` neu gezeichnet.

Listing 13-6 Look & Feel dynamisch ändern

```
// Beispielprogramm für Look and Feel
package LookANDFeel;

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

public class PopupMenu extends JPanel implements MouseListener,
ActionListener
{
  static JLabel   label;
  JPopupMenu            popupMenu;
  JMenuItem       menuWindows,
                        menuMetal,
                        menuMotif,
                        menuMac;

      static String windows   = new
String("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
      static String metal           = new
String("javax.swing.plaf.metal.MetalLookAndFeel");
      static String motif           = new
String("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
      static String mac             = new
String("com.sun.java.swing.plaf.mac.MacLookAndFeel");

      PopupMenu(JApplet applet)
  {

            // Label
            add(label = new JLabel("Look & Feel ändern"));
            label.setBorder(BorderFactory.createTitledBorder(new
EtchedBorder(EtchedBorder.LOWERED)));
            label.setHorizontalAlignment(JLabel.CENTER);
            label.setVerticalTextPosition(JLabel.BOTTOM);
            label.setHorizontalTextPosition(JLabel.CENTER);
            label.addMouseListener(this);
            JTree tree = new JTree();
      JScrollPane sp = new JScrollPane(tree);
      sp.setPreferredSize(new Dimension(200,100));
      add(sp);
```

```
    // PopupMenu 1
    popupMenu = new JPopupMenu("PopupMenu");
    popupMenu.setDefaultLightWeightPopupEnabled(true);
    popupMenu.setInvoker(label);
    popupMenu.add(menuWindows = new JMenuItem("Windows"));
    popupMenu.add(menuMetal = new JMenuItem("Metal"));
    popupMenu.add(menuMotif = new JMenuItem("Motif"));
    popupMenu.add(menuMac = new JMenuItem("Mac"));

      popupMenu.addMouseListener(this);

            menuWindows.addActionListener(this);
            menuWindows.setActionCommand(windows);

            menuMetal.addActionListener(this);
            menuMetal.setActionCommand(metal);

            menuMotif.addActionListener(this);
      menuMotif.setActionCommand(motif);

            menuMac.addActionListener(this);
      menuMac.setActionCommand(mac);

      enableEvents(AWTEvent.COMPONENT_EVENT_MASK);

    // Text
    JLabel txtLabel = new JLabel("rechte Maustaste öffnet das PopUp");
    add(txtLabel);
    //label.setBorder(etchedBorder);
  }

  public void mousePressed(MouseEvent e)
  {
    if (e.getModifiers() == InputEvent.BUTTON3_MASK)
        popupMenu.show(label, e.getX(), e.getY());
  }

  public void mouseClicked(MouseEvent e) {}
  public void mouseReleased(MouseEvent e){}
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}

  public void actionPerformed(ActionEvent e)
  {
            String lnf = e.getActionCommand();
    try {
                  UIManager.setLookAndFeel(lnf);
                  SwingUtilities.updateComponentTreeUI(this);
            }
       catch (Exception exc) {
      System.err.println("Look & Feel wird nicht unterstützt.");
            }
  }
}
```

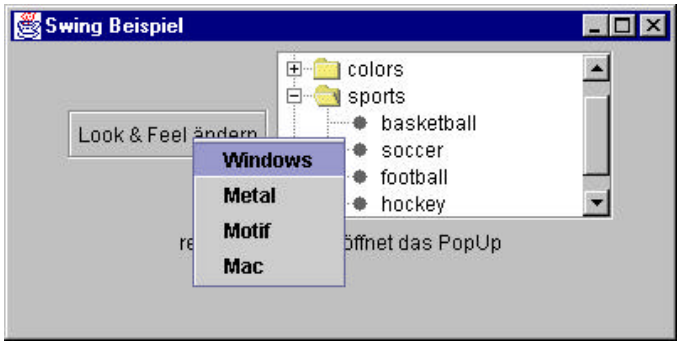


Abbildung 13-5 Look & Feel Beispiel : Windows UIManager

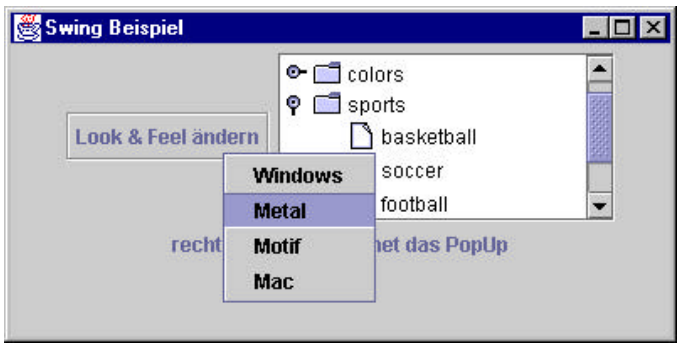


Abbildung 13-6 Look & Feel Beispiel : Metal UIManager (Java Standard)

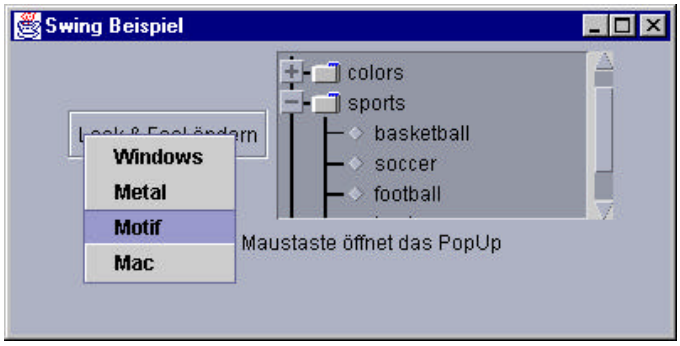


Abbildung 13-7 Look & Feel Beispiel : Motif UIManager (Unix Standard)

Bei Anwahl des MacIntosh Look & Feel wird eine Exception geworfen und eine entsprechende Fehlermeldung (abgefangen) ins Errorlog (System.err...) geschrieben.

13.5.1. Selbstdefinierte Themen unter Metal Look & Feel

Um ein selbstdefiniertes Thema unter dem Metal Look & Feel zu erstellen, zeigt nachfolgender Code die entsprechende Anweisungen. Dabei werden die Farbeinstellungen eines JInternalFrame verändert. Die verschiedenen Themen können dabei über das Menü verändert werden.

Der erste Teil des Programms erstellt die Umgebung: ein Fenster mit einem Menü und einer Auswahl über verschiedene Themen (Themes).

Listing 13-7 Rahmenprogramm

```
package Themen;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.plaf.metal.*;

public class SwingBeispiel extends JFrame {
      public SwingBeispiel() {
            getContentPane().add(new MetalWorks());
            buildMenus();
            pack();
            show();}
      public static void main(String[] args){
            SwingBeispiel frame = new SwingBeispiel();
    frame.setTitle("Definition eigener Themes");
    frame.setSize(300,200);}
      protected void buildMenus() {
                  JMenuBar menuBar = new JMenuBar();
                  menuBar.setOpaque(true);

                  // Themen
                  MetalTheme[] themes = { new DefaultMetalTheme(),
                        new GreenMetalTheme(),
                        new AquaMetalTheme()
                  };
                  // ab ins Menü
            JMenu themeMenu = new ThemeMenu("Themen", themes);
                  menuBar.add(buildFileMenu());
                  menuBar.add(themeMenu);
                  setJMenuBar(menuBar);}
      protected JMenu buildFileMenu() {
                  JMenu file = new JMenu("File");
                  JMenuItem quit = new JMenuItem("Quit");
                  quit.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent e){
                              System.exit(1);}
                  });
      file.add(quit);
                  return file;
    }
}
```

Der zweite Teil des Programms erstellt den `JInternalFrame` und weist ihm ein Layout zu:

Listing 13-8 Programmteil zum Erstellen des Internal Frame

```
package Themen;
import java.awt.*;
import javax.swing.*;
public class MetalWorks extends JPanel{

      private JDesktopPane desktopPane;
      private JInternalFrame creatorFrame;

    public MetalWorks() {
            setLayout(new BorderLayout());
            desktopPane = new JDesktopPane();
            desktopPane.setOpaque(false);
```

```
            creatorFrame = new JInternalFrame("InternalFrame erstellen",
false, false, false, true);
            creatorFrame.setBounds(50, 20, 250, 150);
            desktopPane.add(creatorFrame);
            add("Center", desktopPane);}
}
```

Abbildung 13-8 Internal Frame mit dem Standardthema

Dieser Teil des Programms ist für die Auswertung der Benutzereingaben zuständig. Je nach Auswahl im Menü wird danach ein anderes Thema zugewiesen, eventuell erst nach Anklicken des Intenal Frames. Durch das Verändern der Themen können sich dabei nicht nur die Farben, sondern auch andere Eigenschaften wie die Grösse der Fonts ändern lassen. Das gezeigte Bild oben liefert den gewohnten (Java) Metal Look.

Listing 13-9 Laden unterschiedlicher Themen

```
package Themen;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.*;
import javax.swing.plaf.metal.*;
public class ThemeMenu extends JMenu implements ActionListener{
  MetalTheme[] themes;
  public ThemeMenu(String name, MetalTheme[] themeA) {
    super(name);
    themes = themeA;
      ButtonGroup group = new ButtonGroup();
      for (int i = 0; i < themes.length; i++) {
            JRadioButtonMenuItem item = new JRadioButtonMenuItem(
themes[i].getName() );
            group.add(item);
            add( item );
            item.setActionCommand(i+"");
            item.addActionListener(this);
            if ( i == 0) item.setSelected(true);}
  }
  public void actionPerformed(ActionEvent e) {
    String numStr = e.getActionCommand();
```

```
    MetalTheme selectedTheme=themes[Integer.parseInt(numStr)];
    MetalLookAndFeel.setCurrentTheme(selectedTheme);
    try {
      UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
    } catch (Exception ex) {
      System.out.println("Metal Look & Feel konnte nicht geladen werden");
    }}}
```

Abbildung 13-9 Verschiedene Themen

Dieser Teil enthält den eigentlichen Farbwechsel der verschiedenen Themen. Dabei wird dem `JInternalFrame` eine neues Design (eine neue Farbe) der Oberfläche gegeben.

Listing 13-10 Themen - einfache Farbänderungen

```
package Themen;
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.*;
import javax.swing.plaf.metal.*;

public class GreenMetalTheme extends DefaultMetalTheme {
    public String getName() { return "Emerald"; }
    private final ColorUIResource primary1 = new ColorUIResource(49, 100,
49);
    private final ColorUIResource primary2 = new ColorUIResource(100, 148,
100);
    private final ColorUIResource primary3 = new ColorUIResource(145, 196,
145);
    protected ColorUIResource getPrimary1() { return primary1;}
    protected ColorUIResource getPrimary2() { return primary2;}
    protected ColorUIResource getPrimary3() { return primary3;} }
```

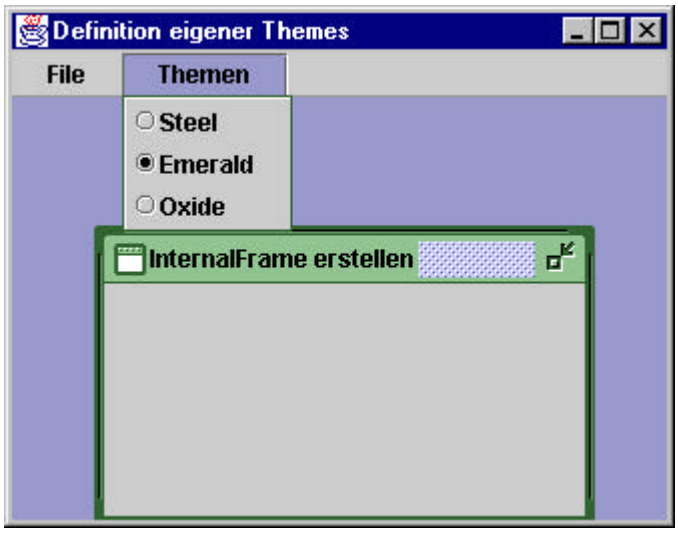


Abbildung 13-10 Beispiel eines Themas (Farbänderung)

Diese Art der Programmierung könnte gut bei Applikationen für verschiedene Benutzergruppen wie zum Beispiel Anwender mit einer Sehbehinderung eingesetzt werden, oder um mit einer Applikation dem Corporate Design, der Corporate Identity, gerecht zu werden. Die Möglichkeiten in Swing in diesem Falle bietet, sind aufgrund der Verbreitung von Java sehr vielfältig. Die Palette reicht von Notepads bis Touchscreens und vom Kühlschrank mit Java bis hin zum Mainframe eines Rechenzentrums.

Um auch das Look & Feel zu verändern, muss sich der Benutzer um einiges mehr kümmern als bei den Themen.

13.6. Das eigene Look & Feel

Ein eigenes Look & Feel für eine Oberfläche ist immer dann sinnvoll, wenn die Oberflächenelemente von Swing nicht mehr ausreichen. Für Entwickler von Oberflächen, welche die Kontrolle über das Aussehen einer Komponente übernehmen wollen, stellt Swing diese Möglichkeit zur Verfügung. Die Klasse `BasicLookAndFeel` ist dabei besonders gut geeignet, um das bestehende Look & Feel weiter zu verwenden, allerdings einzelne Komponenten im Aussehen zu verändern.

Beim Ableiten der Klasse `BasicLookAndFeel` müssen nur folgende Methoden implementiert werden, da sie in der Klasse `LookAndFeel` als `abstract` definiert wurden.

13.6.1. Ändern des Look & Feel

Die Methode `getName`() liefert den Namen des Look & Feel zurück. Der Name kann zum Beispiel in einem Menüeintrag verwendet werden. Die Methode `getDescription`() liefert eine Beschreibung für das Look & Feel. Der Rückgabewert wird meist in ToolTips angezeigt. Die Methode `isNativeLookAndFeel`() sollte dabei `true` zurück liefern, wenn es sich um eine `native` Implementierung eines Look & Feel handelt. Wird das Look & Feel auf Windows NT angewandt und das Windows Look & Feel verwendet, liefert diese Methode `true` zurück. Die Methode `isSupportedLookAndFeel`() liefert `true`, wenn keine systemspezifischen Resourcen verwendet werden, andernfalls `false`. Nun noch die letzte zu implementierende Methode `getID`() : sie dient Programmen und andern Services zur Identifizierung eines Look & Feel. Bekannte Namen sind hier "Motif", "Mac", "Metal" und ""Windows".

Wie dies funktioniert, zeigt das folgende Beispiel, in dem das Look & Feel eines `JButton` nach unseren Wünschen abgeändert wird.

Listing 13-11 Beispiel eines Look & Feel

```
package EigenesLook;
import java.awt.Color;
import javax.swing.UIDefaults;
import javax.swing.plaf.basic.BasicLookAndFeel;
public class MyLookAndFeel extends BasicLookAndFeel {
  public String getName() {
   return "MyLookAndFeel";
  }
  public String getDescription() {
   return "The My Look and Feel";}
  public boolean isNativeLookAndFeel() {
    return false;}
  public String getID() {
   return "MyLookAndFeel";  }
 public boolean isSupportedLookAndFeel() {
   return true;}
 protected void initClassDefaults (UIDefaults table) {
   super.initClassDefaults(table);
   table.put ("ButtonUI", "My.MyButtonUI");
 }
}
```

13.6.2. Die Klasse ButtonUI

In der Klasse `ButtonUI` werden für die zu implementierende Klasse das Aussehen und das Verhalten auf Benutzereingaben verarbeitet. Um das Zeichnen kümmert sich dabei die `ButtonBorder` Klasse und um die Verarbeitung von Benutzer-Events, die `ButtonListener` Klasse. Die Klasse ButtonBorder zeichnet dabei einen von uns festgelegen Rahmen um den Button, und die Klasse `ButtonListener` legt fest, wie auf Tastatur und Mauseingaben reagiert werden sollte.

In der `ButtonUI` Klasse werden neue Methoden verwendet: `installUI`() ist dafür zuständig, den `MouseListener` und die `Border` Klasse in der `ButtonUI` Klasse zu installieren. Diese Methode entfernt Border und Listener von der Klasse. Die Methode `paint`() ist dann für das Zeichnen des Buttons zuständig. In ihr wird das endgültige Aussehen des Buttons festgelegt.

Listing 13-12 Definition einer eigenen ButtonUI Klasse

```
package EigenesLook;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.ComponentUI;
import javax.swing.plaf.basic.*;
import java.io.Serializable;

import javax.swing.plaf.ButtonUI;

public class MyButtonUI extends ButtonUI
   implements Serializable {

 protected final static Insets defaultMargin =
     new Insets (2, 5, 2, 5);
 protected final static Font defaultFont =
     new Font ("Serif", Font.BOLD, 10);

 private final static Border defaultBorder =
   new CompoundBorder(
     MyButtonBorder.getButtonBorder(),
     new BasicBorders.MarginBorder());

 protected static final int textIconGap = 3;

 protected MyButtonListener listener;
 protected static ButtonUI buttonUI;

 public static ComponentUI createUI (JComponent c) {
   if (buttonUI == null) {
     buttonUI = new MyButtonUI();
   }
   return buttonUI;
 }

 public void installUI (JComponent c) {
   listener = new MyButtonListener(c);
   c.addMouseListener(listener);
```

```
    c.addMouseMotionListener(listener);
    c.setFont (defaultFont);
    if (c.getBorder() == null)
      c.setBorder (defaultBorder);
  }

  public void uninstallUI (JComponent c) {
    c.removeMouseListener (listener);
    c.removeMouseMotionListener (listener);
    if (c.getBorder() == defaultBorder)
      c.setBorder(null);
  }

  public void paint (Graphics g, JComponent c) {
    AbstractButton ab = (AbstractButton) c;
    ButtonModel bm = ab.getModel();

    Dimension size = ab.getSize();
    g.setFont (c.getFont());
    FontMetrics fm = g.getFontMetrics();

    int shiftOffset = 0;

    Rectangle viewRect = new Rectangle(size);
    Rectangle iconRect = new Rectangle();
    Rectangle textRect = new Rectangle();

    String text = SwingUtilities.layoutCompoundLabel (
      fm, ab.getText(), ab.getIcon(),
      ab.getVerticalAlignment(),
      ab.getHorizontalAlignment(),
      ab.getVerticalTextPosition(),
      ab.getHorizontalTextPosition(),
      viewRect, iconRect, textRect, textIconGap);

    if (bm.isArmed() && bm.isPressed()) {
      shiftOffset = 1;
    }

    // Paint background
    if (c.isOpaque()) {
      g.setColor (ab.getBackground());
      g.fillRect (0, 0, size.width, size.height);
    }


    // Draw Icon

    if (ab.getIcon() != null) {
      Icon icon = null;
      if (!bm.isEnabled()) {
        icon = ab.getDisabledIcon();
      } else if (bm.isPressed() && bm.isArmed()) {
        icon = ab.getPressedIcon();
      } else if (bm.isRollover()) {
        icon = ab.getRolloverIcon();
      }

      if (icon == null) {
        icon = ab.getIcon();
      }
```

```
    if (bm.isPressed() && bm.isArmed()) {
      icon.paintIcon (c, g, iconRect.x + shiftOffset,
        iconRect.y + shiftOffset);
    } else {
      icon.paintIcon (c, g, iconRect.x, iconRect.y);
    }
  }

  // Draw Text

  if ((text != null) && (text.length() != 0)) {
    if (bm.isEnabled()) {
      g.setColor (ab.getForeground());
      BasicGraphicsUtils.drawString (g, text,
       // bm.getKeyAccelerator(),
          0,
        textRect.x + shiftOffset,
        textRect.y + fm.getAscent() + shiftOffset);
    } else {
      g.setColor (ab.getBackground().brighter());
      BasicGraphicsUtils.drawString (g, text,
        // bm.getKeyAccelerator(),
        0,
        textRect.x, textRect.y + fm.getAscent());
      g.setColor (ab.getBackground().darker());
      BasicGraphicsUtils.drawString (g, text,
        // bm.getKeyAccelerator(),
          0,
        textRect.x - 1, textRect.y + fm.getAscent() - 1);
    }
  }
}

public Dimension getMinimumSize (JComponent c) {
  return getPreferredSize (c);
}

public Dimension getMaximumSize (JComponent c) {
  return getPreferredSize (c);
}

public Dimension getPreferredSize (JComponent c) {
  if ((c.getComponentCount() > 0) ||
      !(c instanceof AbstractButton)) {
    return null;
  }

  AbstractButton ab = (AbstractButton) c;
  Icon icon = ab.getIcon();
  String text = ab.getText();

  Font font = ab.getFont();
  FontMetrics fm = ab.getToolkit().getFontMetrics (font);

  Rectangle viewRect = new Rectangle (
    Short.MAX_VALUE, Short.MAX_VALUE);
  Rectangle iconRect = new Rectangle();
  Rectangle textRect = new Rectangle();

  SwingUtilities.layoutCompoundLabel (
    fm, text, icon,
    ab.getVerticalAlignment(),
```

```
      ab.getHorizontalAlignment(),
      ab.getVerticalTextPosition(),
      ab.getHorizontalTextPosition(),
      viewRect, iconRect, textRect, textIconGap);

    // find union of icon and text rectangles
    Rectangle rect = iconRect.union (textRect);

    Insets insets = getInsets (c);

    rect.width += insets.left + insets.right;
    rect.height += insets.top + insets.bottom;

    return rect.getSize();
  }

  public Insets getDefaultMargin (AbstractButton b) {
    return defaultMargin;
  }


  public Insets getInsets (JComponent c) {
    Border border = c.getBorder();
    Insets insets = ((border != null) ?
      border.getBorderInsets (c) :
      new Insets (0,0,0,0));
    return insets;
  }
}
```

13.6.3. Die eigene Listener Klasse

Der eigene Mouse Listener für das Look & Feel ist so zu konzipieren, das letzteres auch bei andern Komponenten implementiert werden kann. Ein Listener horcht so lange an einem Objekt, bis ein Event auftritt. Ist dies der Fall, wird eine Aktion ausgelöst. Da wir den Listener nun selber schreiben, könnte man festlegen, dass der Button nur gedrückt werden kann, wenn zuvor eine bestimmte Taste gedrückt wurde. Auch hier unterliegt das Verhalten des Buttons vollkommen dem Entwickler. Die hier vorgestellte Klasse `MyButtonListener` implementiert den `MouseListener`, wie man ihn von gewöhnlichen Buttons her kennt.

Listing 13-13

```
package EigenesLook;

import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

import javax.swing.*;

public class MyButtonListener implements MouseListener,
   MouseMotionListener, Serializable {
 AbstractButton ab;

 public MyButtonListener (JComponent c) {
   ab = (AbstractButton) c;}
 public void mouseMoved (MouseEvent e) {}
 public void mouseClicked (MouseEvent e) {}
```

```
 public void mouseDragged (MouseEvent e) {
   ButtonModel bm = ab.getModel();
   if (bm.isPressed()) {
     Graphics g = ab.getGraphics();
     if (g != null) {
       Rectangle r = g.getClipBounds();
       if (r.contains (e.getPoint()))
         bm.setArmed(true);
       else
         bm.setArmed(false);
     }
   }
 }

 public void mousePressed (MouseEvent e) {
   ButtonModel bm = ab.getModel();
   bm.setArmed(true);
   bm.setPressed(true);}

 public void mouseReleased (MouseEvent e) {
   ButtonModel bm = ab.getModel();
   bm.setPressed (false);
 }

 public void mouseEntered (MouseEvent e) {
   if (ab.getRolloverIcon() != null)
     ab.getModel().setRollover (true);
 }

 public void mouseExited (MouseEvent e) {
   if (ab.getRolloverIcon() != null)
     ab.getModel().setRollover (false);
 }
}
```

13.6.4. Die eigene Border Klasse

Die Border Klasse zeichnet nun den selbst entworfenen Button. Die Klasse zeichnet hier Dreiecke auf jede Seite des Buttons. Je nach Status des Buttons ändert dieser seine Farbe.

Listing 13-14 Definition einer eigenen Borderklasse

```
package EigenesLook;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.basic.BasicGraphicsUtils;

public class MyButtonBorder extends AbstractBorder {
 private static Border buttonBorder =
   new MyButtonBorder();
 public static Border getButtonBorder() {
   return buttonBorder;
 }
 public void paintBorder (Component c, Graphics g,
     int x, int y, int width, int height) {
   boolean pressed = false;
   boolean focused = false;
   if (c instanceof AbstractButton) {
```

```
      AbstractButton b = (AbstractButton)c;
      ButtonModel bm = b.getModel();

      pressed = bm.isPressed();
      focused = (pressed && bm.isArmed()) ||
                (b.isFocusPainted() && b.hasFocus());
    }
    Insets in = getBorderInsets(c);
    Polygon p1 = new Polygon ();
    p1.addPoint (x+in.left, y);
    p1.addPoint (x, y+(height/2));
    p1.addPoint (x+in.left, y+height);
    Polygon p2 = new Polygon ();
    p2.addPoint (x+width-in.right, y);
    p2.addPoint (x+width, y+(height/2));
    p2.addPoint (x+width-in.right, y+height);
    if (pressed) {
      g.setColor (c.getForeground());
    } else if (focused) {
      g.setColor (SystemColor.green);
    } else {
      g.setColor (SystemColor.red);
    }
    g.fillPolygon (p1);
    g.fillPolygon (p2);
  }

  public Insets getBorderInsets (Component c) {
    return new Insets (5, 10, 5, 10);
  }
}
```

13.6.5. Das Beispielprogramm

In der noch ausstehenden Klasse wird das nun definierte Look & Feel verwendet. Wie dies funktioniert, wurde bereits im Abschnitt "Look & Feel" bereits erklärt.

Listing 13-15 Beispiel LookAndFeel

```
package EigenesLook;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class BeispielLookAndFeel extends JPanel {
      public BeispielLookAndFeel () {
    // Create the buttons.
    JButton button = new JButton ("Ein neuer Button");
    ActionListener myListener = new ActionListener() {
      public void actionPerformed (ActionEvent e) {
        String lnfName = null;
        if (e.getActionCommand().equals ("Eigener Look")) {
          lnfName = "My.MyLookAndFeel";
        } else {
          lnfName =
            "javax.swing.plaf.metal.MetalLookAndFeel";
        }
        try {
          UIManager.setLookAndFeel(lnfName);
          SwingUtilities.updateComponentTreeUI (
            BeispielLookAndFeel.this);
          BeispielLookAndFeel.this.validate();
```

```
        } catch (Exception ex) {
          System.err.println (
           "Wechseln des LookAndFeel " + lnfName +" nicht möglich !!!");
        }
      }
    };
    ButtonGroup group = new ButtonGroup();
    JRadioButton basicButton =
      new JRadioButton ("Metal Look");
    basicButton.setSelected(true);
    basicButton.addActionListener (myListener);
    group.add (basicButton);
    JRadioButton myButton =
      new JRadioButton ("Eigener Look");
    myButton.addActionListener (myListener);
    group.add (myButton);
    add (button);
    add (basicButton);
    add (myButton);
  }
}
```

Obiges Beispiel zeigt die komplette Implementierung eines eigenen Look & Feel für einen `JButton`. Wie Sie sehen können, sollte man den notwendigen Arbeitsaufwand, um ein Look & Feel komplett neu aufzubauen, nicht zu unterschätzen. Dabei stellt ein Button noch ein einfaches Oberflächenelement dar. Die Implementierung eines `JTrees` ist um einiges aufwendiger und ist eine **Übungsaufgabe**.

13.7. JList & JComboBox

Die grundsätzliche Anwendung von `JList` und `JComboBox` wurde schon in den entsprechenden Kapiteln besprochen. Dort behandelten wir die Anzeige von reinen Textelementen. Komplexere Anzeigen (zum Beispiel mit Icons) waren noch nicht möglich. Um die beiden Komponenten mit diesen Fähigkeiten zu erweitern, muss die Datenstruktur und die Darstellung modifiziert werden. Wegen der engen Verwandtschaft von `JList` und `JComboBox` werden sie hier zusammengefasst. Sowohl `JList` als auch `JComboBox` setzen die gleiche Datenstruktur ein.

Die Datenstruktur wird von der Schnittstelle `ListModel` definiert. Natürlich wurde die Implementierung schon von der Klasse `AbstractListModel` vorbereitet. Eigene Datenstrukturen müssen von dieser Klasse abgeleitet werden. Bis jetzt kann `AbstractListModel` nur für `JList` eingesetzt werden. Der Einsatz für Kombinationsfelder (`JComboBox`) verlangt zusätzlich die Implementierung der Schnittstelle `ComboBoxModel`.

13.7.1. ListModel

- `public int getSize()`
  liefert die Anzahl der Elemente in der Datenstruktur

- `public Object getElementAt(int index)`
  liefert den Wert bzw. das Objekt des Eintrags, der durch `index` angegeben wird.

- `public void addListDataListener(ListDataListener list)`
  fügt einen `ListDataListener` in die Datenstruktur ein

- `public void removeListDataListener(ListDataListener list)`
  entfernt einen `ListDataListener` aus der Datenstruktur

Die Schnittstelle `ComboBoxModel` erweitert das Datenmodell um zwei weitere Methoden:

- `public Object getSelectedItem()`
  liefert das Objekt des selektierten Eintrags im Kombinationsfeld.

- `public void setSelectedItem(Object anItem)`
  setzt das Objekt des selektierten Eintrags im Kombinationsfeld

Das folgende Beispiel einer komplexeren Datenstruktur zeigt eine Liste von Weinsorten (Icon und Namen). Bei Selektion eines Eintrags wird dieser mit dem Preis der Weinsorte erweitert. Jeder Listeneintrag hat also drei Bestandteile :

1. ein Icon (`ImageIcon`)
2. den Namen der Weinsorte (`String`) und
3. den entsprechenden Preis (`String`)

Die Bestandteile werden in einer Hashtabelle verwahrt. Jeder Eintrag entspricht einer Hashtabelle (`Hashtable`). Dieser wird nach der Selektierung mit `getElementAt`()

ermittelt. Falls für den Eintrag noch keine Hash-Tabelle existiert, baut sie `getElementAt()` auf.

Listing 13-16 Definition des eigenen JList Modells

```
class MyListModel extends AbstractListModel implements ComboBoxModel
{
  JApplet     applet;         // Verbindung zum Applet
  Object      currentValue;
  String      fileNames[] =   // Namen der Bilddateien
                {"portugieser.gif", "riesling.gif", "rivaner.gif",
                 "rulaender.gif", "scheurebe.gif", "spaetlese.gif"};
  String      weinNames[] =   // Namen der Weinsorten
                {"Portugieser", "Riesling", "Rivaner",
                 "Ruländer", "Scheurebe", "Spätlese"};
  String      weinPreise[] =  // Preise der Weinsorten
                {"20.95 CHF", "18.50 CHF", "22.90 CHF",
                 "16.85 CHF", "22.95 CHF", "25.90 CHF"};
  ImageIcon   weinIcons[] = new ImageIcon[weinNames.length];
  // Listeneinträge
  Hashtable   modelCache[] = new Hashtable[getSize()];


  public MyListModel(JApplet a)
  {
    this.applet = a;

    try
    {
      for (int i=0; i<weinNames.length; i++)  // Icons einladen
        weinIcons[i] = new ImageIcon(fileNames[i]);
    }
    catch(SecurityException se)      // Für Internet-Browser
    {
      for (int i=0; i<weinNames.length; i++)  // Icons einladen
        weinIcons[i] = new ImageIcon(
          applet.getImage(applet.getCodeBase(), fileNames[i]));
    }
  }

  // Setzt den neuen Wert der Listenelements
  public void setSelectedItem(Object anObject)
  {
    currentValue = anObject;
    fireContentsChanged(this,-1,-1);
  }

  // Liefert den aktuellen wert zurück
  public Object getSelectedItem()
  {
    return currentValue;
  }

  // Liefert die Anzahl der Elemente
  public int getSize()
  {
    return weinNames.length;
  }

  public Object getElementAt(int index)
  {
```

```
    if(modelCache[index] != null)
      return modelCache[index];
    else
    {
      Hashtable result = new Hashtable();
      // Text des Elementes im unselektierten Zustand
      result.put("title",  weinNames[index]);
      // Text des Elementes im selektierten Zustand
      result.put("selectedTitle", weinPreise[index]);
      // Icon des Elementes
      result.put("image", weinIcons[index]);
      modelCache[index] = result;
      return result;
    }
  }
}
```

Die Datenstruktur ist vollständig definiert. Was fehlt ist die Anzeige (View) der Listenelemente. Die Standardanzeige kann mit der Hash-Tabelle nichts anfangen. Sie würde nur die `toString`() Methode aufrufen, deren Resultat unbrauchbar ist. Es ist also ein Renderer mit Hilfe der `ListCellRenderer` Schnittstelle zu implementieren. Sie enthält die Methode getListCellRendererComponent() und gibt die Komponente zurück, deren `paint`() Methode das Listenelement darstellt. Grundsätzlich kann jede Komponente als Renderer fungieren, solange sie die `ListCellRenderer` Schnittstelle implementiert. In unserem Beispiel ist es ein `JLabel`. Sie wird bei Aufruf von `getListCellRendererComponent`() mit Daten in der Hash Tabelle aufbereitet und als Ergebnis zurückgeliefert.

Listing 13-17 die Methode getListRendererComponent

```
public MyCellRenderer()
  {
    setOpaque(true);
  }
   public Component getListCellRendererComponent(JList listbox,
        Object value, int index, boolean isSelected, boolean
         cellHasFocus) {
     Hashtable h = (Hashtable) value;
     // Hashtabelle eines Listenelements
    setOpaque(true);

    if(value == null)    {
      setText("");
      setIcon(null);
    }
    else    {



      if(isSelected){  // selektiertes Listenelement
 setBackground(UIManager.getColor("ComboBox.selectedBackground"));
        setForeground(UIManager.getColor("ComboBox.selectedForeground"));
        setHorizontalTextPosition(JLabel.RIGHT);
        setIcon((ImageIcon)h.get("image"));
         // Namen der Weinsort + Preis
        setText((String)h.get("title") + ":  " +
          (String)h.get("selectedTitle"));
      }
      else { // unselektiertes Listenelement
```

```
        setIcon((ImageIcon)h.get("image"));
        setText((String)h.get("title"));
        setHorizontalTextPosition(JLabel.RIGHT);
        setBackground(UIManager.getColor("ComboBox.background"));
        setForeground(UIManager.getColor("ComboBox.foreground"));
      }
    }
    return this;
  }
```

13.7.2. ListSelectionModel

Auf welche Art und Weise der Benutzer einzelne oder ganze Bereiche von Objekten in einer Liste auswählen darf, geht auf das Konto des `ListSelectionModel`. `JList` bestimmt damit die Regeln für die Selektierung. In `JComboBox` gibt allerdings nur eine Art der Selektierung. Es ist jeweils nur ein Element zur gleichen Zeit anwählbar.

Die Klasse `DefaultListSelectionModel` ist eine simple Implementierung des `ListSelectionModel`. Dieses Modell beschreibt, ob `JList` sich im Single- oder Multi-Selection Modus befindet.

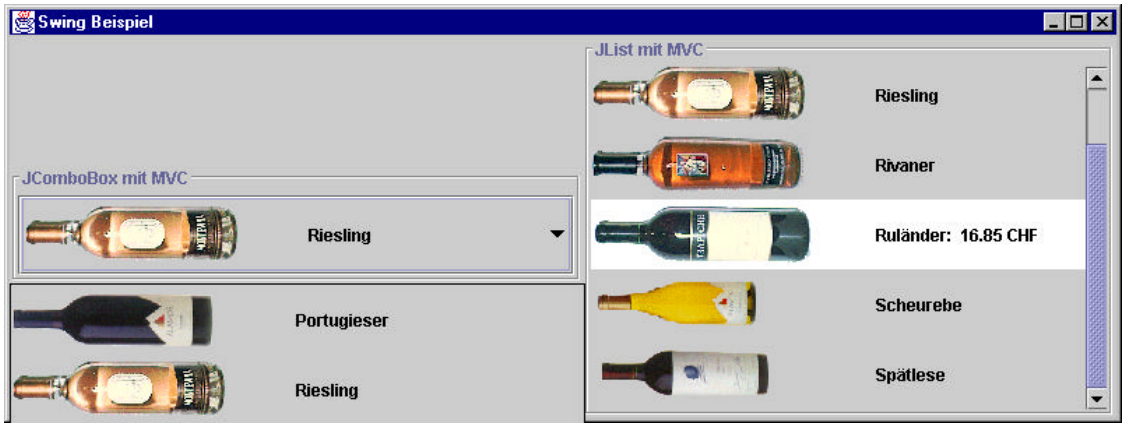


Abbildung 13-11 JComboBox und JList mit MVC

In diesem Kapitel
- *Erzeugung einer einfachen Tabelle*
- *Datenstruktur / Datenmodell einer Tabelle*
- *Ändern der Spaltenbreite*
- *Zelleneditoren und Renderer*
  - *Definition neuer Zellenrenderer*
  - *Definition neuer Zelleneditoren*

14. Tabellen

Für Darstellungen von Daten in tabellarischer Form eignet sich die Klasse `JTable` hervorragend. Auch wenn sie selbst noch keine Tabellenkalkulation ist, bildet sie das Grundgerüst dafür, wie in einem Beispiel gezeigt wird. Auch kann man sie bestens als Anzeige für Datenbanktabellen einsetzen. Der Funktionsumfang dieser Klasse ist äusserst mächtig, so dass nicht alle Punkte bis ins Detail abgedeckt werden können.

`JTable` gehört ebenfalls zu den Klassen, die voll und ganz die MVC Architektur von Swing unterstützen. Sie selbst bekommt die Rolle des Views und des Controllers. Die Daten werden im Datenmodell / in der Datenstruktur der Tabelle gespeichert (`TableModel`). Dazu werden wir später noch einiges kennen lernen.

14.1. Erzeugung einer einfachen Tabelle

Zuerst soll der einfachste Fall mit dem Umgang der Tabelle besprochen werden. Die Daten für den Tabelleninhalt und die Spaltenüberschriften können bei der Instanzierung von Jtable im Konstruktor übergeben werden. Nun kann der Programmierer die Daten in einer Objektmatrix (`Object[ ] [ ]`) und die Spaltenüberschriften in einem Objekt-Array (`Object[ ] `) ablegen. Er kann beides auch in Vektoren (`Vector`) unterbringen. Bei den folgenden Beispielen wird die erste Möglichkeit angewandt. Es zeigt eine fiktive Liste bestellter Weinflaschen einer Weinhandlung mit diversen Daten.

Listing 14-1 Grundkonstrukt einer Tabelle in Java Swing

```
JTable    weinTable;
  JScrollPane   scrollPane;
  Object    weinListe[][] = {
    {"Scheurebe", new Double(21.10)},
    {"Riesling", new Double(27.50)},
    {"Weißburgunder", new Double(26.60)},
    {"Spätburgunder", new Double(28.80)},
    {"Portugieser", new Double(22.40)}
  };
  Object    columnNames[] = {"Bestellnr.", "Rebsorte", "Abbildung",
"Sektpreis/Liter", "Flaschengr.", "Verkaufte Flaschen", "Gesamtpreis",
"Lieferung" };
  int       columnWidths[] = {70, 140, 70, 100, 90, 120, 130, 70};
  ImageIcon weinIcons[] = new ImageIcon[weinListe.length];
  {
    weinIcons[0] = new ImageIcon("scheurebe.gif");
    weinIcons[1] = new ImageIcon("riesling.gif");
```

```
    weinIcons[2] = new ImageIcon("weissburgunder.gif");
    weinIcons[3] = new ImageIcon("spaetburgunder.gif");
    weinIcons[4] = new ImageIcon("portugieser.gif");
  }

  Object  tableData[][] = {
    { new Integer(1), weinListe[0][0], weinIcons[0], weinListe[0][1], new
Double(0.2), new Integer(200), new Double(0), new Boolean(true)},
    { new Integer(2), weinListe[1][0], weinIcons[1], weinListe[1][1], new
Double(0.75), new Integer(500), new Double(0), new Boolean(false) },
    { new Integer(3), weinListe[2][0], weinIcons[2], weinListe[2][1], new
Double(1.0), new Integer(100), new Double(1), new Boolean(false) },
    { new Integer(4), weinListe[3][0], weinIcons[3], weinListe[3][1], new
Double(1.0), new Integer(800), new Double(0), new Boolean(true) },
    { new Integer(5), weinListe[4][0], weinIcons[4], weinListe[4][1], new
Double(0.2), new Integer(1000), new Double(0), new Boolean(false) },
    { new Integer(6), weinListe[2][0], weinIcons[2], weinListe[2][1], new
Double(0.75), new Integer(20), new Double(0), new Boolean(true) },
    { new Integer(7), weinListe[1][0], weinIcons[1], weinListe[1][1], new
Double(0.2), new Integer(700), new Double(0), new Boolean(true) },
    { new Integer(8), weinListe[0][0], weinIcons[0], weinListe[0][1], new
Double(1.0), new Integer(600), new Double(0), new Boolean(true) },
    { new Integer(9), weinListe[3][0], weinIcons[3], weinListe[3][1], new
Double(1.0), new Integer(600), new Double(0), new Boolean(false) },
  };
```

Hierbei ist anzumerken, dass alle Daten als Objekt vorliegen müssen d.h. auch Zahlen sind z.B. in Objekte von Typ Integer oder Double zu kapseln (wrappen).

Nun wird die Tabelle einfach erzeugt durch:

```
weinTable = new JTable(dataModel);
```

Der Vorteil dieser Methode liegt klar auf der Hand:
die Tabelle wird ohne grossen Aufwand auf den Bildschirm ausgegeben. Doch die Einbussen dieser Anwendung sind ebenso deutlich zu erkennen:

- jeder Zeile ist grundsätzlich editierbar
- die Ausgabe der Daten erfolgt ohne Berücksichtigung der Datentypen. Es wird von jedem Objekt das Resultat der `toString`() Methode angezeigt. Der Datentyp `boolean` wird als `true` oder `false` ausgegeben. Ausserdem fehlt die Formatierung der Zahlenfelder. Zum Beispiel sollte der Gesamtpreis im Währungsformat erscheinen. Weiterhin sind alle Zahlen linksbündig ausgerichtet anstatt rechtsbündig.
- die Daten müssen vorher in Arrays bzw. Vektoren untergebracht sein. Will man jedoch den Bestand einer Datenbank ausgeben, wäre es sinnvoller, das Ergebnis einer Abfrage direkt in die Tabelle zu übertragen, ohne vorher die Daten in Arrays oder Vektoren zu kopieren.

14.2. Datenmodell / Datenstruktur einer Tabelle

Das Datenmodell einer Tabelle wird durch die Schnittstelle `TableModel` spezifiziert. Die Klasse `AbstractTableModel` implementiert sie und verwaltet die Liste der Lauscher / Listener vom Typ `TableModelListener`. Sie sind für Tabellenereignisse zuständig (siehe Swing Ereignsse). Das erzeugte Datenmodell, die erzeugte Datenstruktur, wird im Konstruktor von `JTable` oder mit `setModel`() gesetzt. Es sind folgende Methoden vorhanden:

- `int getCiolumnCount()`
  liefert die Anzahl Spalten im Datenmodell, in der Tabelle

- `int getRowCount()`
  liefert die Anzahl der Zeilen im Datenmodell, in der Tabelle

- `Class getColumnClass(int column)`
  liefert die Klasse der Spalte

- `Object getColumnIdentifier(int column)`
  liefert den eindeutigen Indentifizierer der Spalte. Da vom Benutzer die Spaltenreihenfolge jederzeit geändert werden kann, ist er unentbehrlich.

- `String getColumnName(int column)`
  liefert den Namen der Spalte

- `Object getValueAt(int row, int column)`
  liefert den momentanen Zelleninhalt, der durch row und column identifiziert wird.

- `void setValueAt(Object aValue, int row, int column)`
  setzt den Wert einer Zelle, die durch `row` und `column` identifiziert wird. Es wird empfohlen, die Änderung mit `fireTableChanged`() abzuschliessen, um das Tabellenmodell über die Änderung zu informieren.

- `void addTableModelListener( TableModelListener li)`
  fügt einen neuen Listener in die Liste der `TableModelListener` ein

- `void removeTableListener( TableModelListener li)`
  entfernt einen Listener aus der Liste der `TableModelListener`.

Im vorherigen Beispiel erzeugte `JTable` sein eigenes Datenmodell anhand der Objekt-Arrays. Um ein eigenes Modell zu kreieren, müssen einige der eben genannten Funktionen überlagert werden:

Listing 14-2 Das Datenmodell der Tabelle

```
final TableModel dataModel = new AbstractTableModel(){
      public int getColumnCount(){
        return columnNames.length;
      }
```

```
      public int getRowCount(){
        return tableData.length;
      }

      public Object getValueAt(int row, int col){
        return tableData[row][col];
      }
      public String getColumnName(int column){
        return (String)columnNames[column];
      }
      public Class getColumnClass(int c){
        return getValueAt(0, c).getClass();
      }
      public boolean isCellEditable(int row, int col){
        if (col==1 || col==4 || col==5 || col==7)
          return true;
        else
          return false;
      }
      public void setValueAt(Object aValue, int row,
                                    int column){
        tableData[row][column] = aValue;
        if (column==3 || column==4 || column==5){
        double preis = (Double.valueOf(getValueAt(row,
3).toString())).doubleValue();
          double groesse = (Double.valueOf(getValueAt(row,
4).toString())).doubleValue();
          double flaschen = (Double.valueOf(getValueAt(row,
5).toString())).doubleValue();
          setValueAt(new Double(preis * groesse * flaschen), row, 6);
        }
        if (column==1){
          for (int i=0; i<weinListe.length; i++)
            if (weinListe[i][0].equals(aValue)){
              setValueAt(weinListe[i][1], row, 3);
              setValueAt(weinIcons[i], row, 2);
            }
        }
        // geänderten Daten werden angezeigt
        fireTableDataChanged();
      }
    };
```

Durch `getColumnClass`() kann der Datentyp der Spalte ermittelt werden. Die Darstellung des Tabelleninhaltes kann sich entsprechend anpassen. Zahlen werden nun rechtsbündig abgezeigt und der Typ `boolean` als Checkbox. Weiterhin kann man gezielt das Editieren einzelner Spalten erlauben.

Zum Beispiel:

```
public boolean isCellEditable(int row, int col){
        if (col==1 || col==4 || col==5 || col==7)
          return true;
        else
          return false;
       }
```

14.3. Ändern der Spaltenbreiten

Im ersten Beispiel besassen alle Spalten die gleiche Breite. Diese richtet sich wiederum nach der Breite der Tabelle. Ändert sich nun die Grösse einer Spalte,schrumpft bzw. wächst die Breite der rechts befindlichen Spalten.

Die Breite jeder einzelnen Spalte kann explizit mit setPreferredWidth() gesetzt werden. Diese Methode wirkt sich auch auf die relative Grösse aus. Sobald die Tabelle verkleinert wird, ändert sich die Spaltenbreiten so, dass deren Grössenverhältnisse die gleichen sind wie zuvor.

Zum Beispiel

```
int columnWidths[] = {70, 140, 70, 100, 90, 120, 130, 70};
…
// Spaltenbreiten
for (int i=0; i<dataModel.getColumnCount(); i++)
weinTable.getColumnModel().getColumn(i).
 setPreferredWidth(columnWidths[i]);
```

Das Anpassungsverhalten der Tabelle kann man mit `setAutoResizeMode`() bestimmen. Der Parameter hat folgende Auswirkungen:

`AUTO_RESIZE_SUBSEQUENT_COLUMNS`
Standard: sobald eine Spalte vergrössert oder verkleinert wird, ändert sich die Grösse der Tabelle nicht. Es werden die Breiten der rechts befindlichen Spalten angepasst.

`AUTO_RESIZE_NEXT_COLUMN`
passt nur die Breite der nächsten Spalte an

`AUTO_RESIZE_OFF`
sobald die Breite einer Spalte geändert wird, ändert sich die Grösse der Tabelle.

14.4. Zelleneditoren und Renderer

Das Konzept zur Darstellung der Daten in Tabellen verwendet nicht, wie man erwarten könnte, für jede Zelle eine separate Komponente. Aus Gründen der Performance benutzt eine Spalte eine einzige Komponente als Renderer zum Anzeigen ihrer Zellen. In vielen Fällen wird diese Komponente auch von mehreren oder sogar von allen Spalten geteilt. Sobald der Anwender eine Zelle editieren möchte, kümmert sich der Zelleneditor um das korrekte Eingabeverhalten.

Nehmen wir zum Beispiel die Spalte *Verkaufte Flaschen* der Weinliste. Es handelt sich hierbei um Zahlen vom Typ Integer. Um sie richtig darzustellen, wird standardmässig ein `JLabel` mit der Ausrichtung rechtsbündig verwendet. Beim Editieren des Zahlenwertes greift die Tabelle auf ein ebenfalls rechts ausgerichtetes `JTextField` zurück, um die Eingabe zu kontrollieren.

Damit die Tabelle für den jeweiligen Datentyp den geeigneten Renderer einsetzt, benötigt sie die Funktion `getColumnClass`(). Anhand der zurückgelieferten Klasse ermittelt sie den Datentyp der Spalte und weiss, welchen Renderer sie anwenden muss. Die Tabelle verfügt intern über eine Liste registrierter Zellenrenderer und vergleicht die Spaltenklasse mit denen der Liste. Bei Übereinstimmung wird der jeweilige Renderer verwendet. Kommt es zu keinem Treffer, wird der Textwert der Zelle durch Aufruf der `toString`() Methode angezeigt.

Momentan sind für folgende Datentypen entsprechende Renderer vorhanden:

- `Boolean`
  wird durch eine Checkbox dargestellt
- `Number` (und alle abgeleiteten Klassen)
  wird durch ein rechtsbündiges Label dargestellt
- `ImageIcon`
  Icons werden durch ein zentriertes Label dargestellt
- `Object`
  wird durch ein Label angezeigt, das den Textwert des Objekts enthält

Um diese Liste zu erweitern, müssen neue Zellenrenderer definiert werden.

14.4.1. Definition neuer Zellenrenderer

Zellenrenderer werden durch die Schnittstelle `TableCellRenderer` spezifiziert. Die Tabelle nimmt defaultmässig die Klasse `DefaultCellRenderer`, die diese Schnittstelle implementiert. Sie verfügt über folgende Methoden:

- `public void setForeground(Color c)`
- `public void setBackground(Color c)`
- ```
  public Component getTableCellRendererComponent(
                        JTable table, Object value,
                        boolean isSelected, boolean hasFocus,
                        int row, int column)
  ```
- `protected void setValue(Object value)`
  ```

In unserem Beispiel soll die Spalte Gesamtpreis im Währungsformat angezeigt werden. Zunächst wird die nötige Formatklasse `DecimalFormat` erzeugt:

```
DecimalFormat DMFormat = new DecimalFormat("#,##0.00 CHF");
```

Man instanziert einfach die Klasse `DefaultTableCellRenderer` und überschreibt die Methode `setValue`(). Da `DefaultTableCellRenderer` von `JLabel` abgeleitet ist, kann der Ausgabetext mit `setText`() gesetzt werden.

```
DefaultTableCellRenderer DMRenderer = new DefaultTableCellRenderer() {
      public void setValue(Object value){
        Double d = (Double)value;
        setText(DMFormat.format(d.doubleValue()));
      }
};
    DMRenderer.setHorizontalAlignment(JLabel.RIGHT);
```

Um Zellenrenderer festlegen zu können, ist erst die Spalte (`TableColumn`) mit `getColumn`() zu ermitteln. Übergabeparameter ist die Spaltenüberschrift. Danach kann der Renderer mit `setCellRenderer`() gesetzt werden.

```
weinTable.getColumn("Gesamtpreis").setCellRenderer(DMRenderer);
```

14.4.2. Definition neuer Zelleneditoren

Das Verfahren gleicht dem des Renderers. Ein Zelleneditor wird durch die Schnittstelle `TableCellEditor` beschrieben. Ebenfalls existiert die Klasse `DefaultCellEditor`, die diese Schnittstelle implementiert. `DefaultCellEditor` ist allerdings eine direkte Ableitung von `Object` und implementiert noch andere Schnittstellen, da sie auch für Bäume einsetzbar sind. Im Konstruktor gibt man dann die Komponente an, die als Zelleneditor fungieren soll. Dabei kann man sich entscheiden zwischen `JCheckBox`, `JComboBox` und `JTextField`.

Im Beispiel der Weinliste wurde für die Spalte *Weinsorte* ein Kombinationsfeld (`JComboBox`) benutzt, um komfortabel eine bestimmte Weinsorte aussuchen zu können. Dadurch wird ausgeschlossen, dass der Anwender sich vertippt oder versucht eine unbekannte Sorte einzugeben.

Zuerst wird das Kombinationsfeld ganz normal erzeugt und aufgebaut:

```
Object    weinListe[][] = {
    {"Scheurebe", new Double(21.10)},
    {"Riesling", new Double(27.50)},
    {"Weißburgunder", new Double(26.60)},
    {"Spätburgunder", new Double(28.80)},
    {"Portugieser", new Double(22.40)}
  };
...
// Weinsorten
    JComboBox  weinSorte = new JComboBox();
    for (int i=0; i<weinListe.length; i++)
      weinSorte.addItem(weinListe[i][0]);
```

Jetzt muss nur noch die Spalte mit `getColumn`() ermittelt und der Editor mit `setCellEditor`() gesetzt werden:

```
weinTable.getColumn("Rebsorte").setCellEditor(new
                    DefaultCellEditor(weinSorte));
```

Im zweiten Beispiel wurde in die Weinliste eine Spalte mit Abbildungen eingefügt. Neben den genannten Erweiterungen wurde die setValue() Methode so ergänzt, dass in der Tabelle automatisch ein Update erfolgt. Wählt der Benutzer eine andere Weinsorte aus, müssen die Abbildung und der Literpreis entsprechend angezeigt und der Gesamtpreis neu berechnet werden. Bei Änderungen der Flaschengrösse und der Anzahl verkaufter Flaschen muss der Gesamtpreis ebenfalls neu kalkuliert werden. Somit wird die Liste bereits zu einer Mini-Tabellenkalkulation.

```
public void setValueAt(Object aValue, int row, int column) {
   tableData[row][column] = aValue;
   if (column==3 || column==4 || column==5){
    double preis = (Double.valueOf(getValueAt(row, 3).
                                toString())).doubleValue();
    double groesse = (Double.valueOf(getValueAt(row, 4).
                                     toString())).doubleValue();
    double flaschen = (Double.valueOf(getValueAt(row, 5).
                                     toString())).doubleValue();
    setValueAt(new Double(preis * groesse * flaschen), row, 6);
  }
       if (column==1)        {
         for (int i=0; i<weinListe.length; i++)
           if (weinListe[i][0].equals(aValue)) {
             setValueAt(weinListe[i][1], row, 3);
             setValueAt(weinIcons[i], row, 2);
           }
       }
       // geänderten Daten werden angezeigt
       fireTableDataChanged();
     }
    };
```

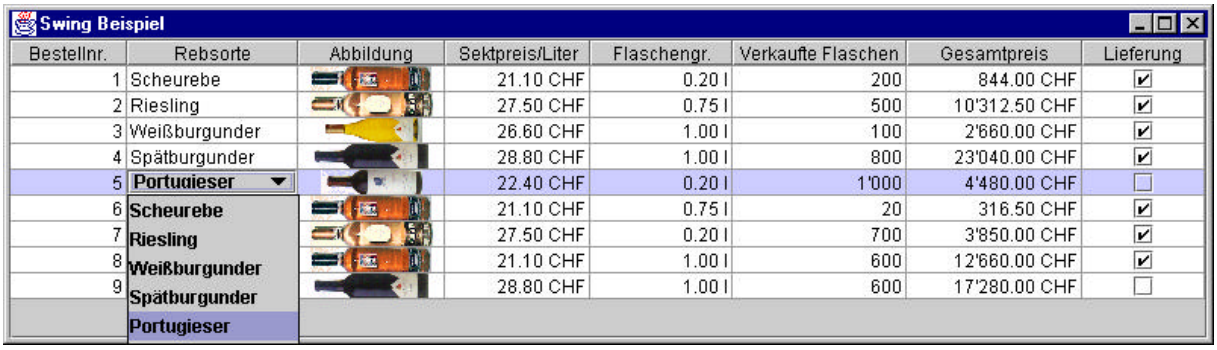


Abbildung 14-1 Weinverwaltung

In diesem Kapitel

- *Implementierung eines Dokuments*
  - *AbstractDocument*
  - *PlainDocument*
  - *DefaultStyleDocument*
- *Verwendung eines Dokuments*
- *Änderungen in Dokumenten (DocumentListener)*
- *Textaktionen*
- *Tastaturbefehle*
- *Beispiel Multipad*
  - *Menüs*
  - *Frames*
  - *File-Dialog Filter*
  - *Look & Feel*

15. *Das Dokumentensystem*

Als Abschluss des Themas "Swing" betrachten wir ein komplexeres Beispiel, mit Hilfe des Dokumentensystems von Swing. Das Ergebnis ist ein Multipad, also ein Notepad ähnliches Programm, welches neben Text auch Grafiken enthalten kann.

Das MVC Prinzip gilt auch für die meisten Textkomponenten von Swing. Es besteht eine strikte Trennung zwischen Textinhalt (`Document`) und der Ansicht bzw. der Anzeige (View). Der Inhalt wird meistens mittels der Schnittstelle `Document` implementiert. Für Text mit anspruchsvollerem Layout steht das `StyledDocument` zur Verfügung. Dargestellt wird es mit der View Klasse oder einer entsprechenden Ableitung.

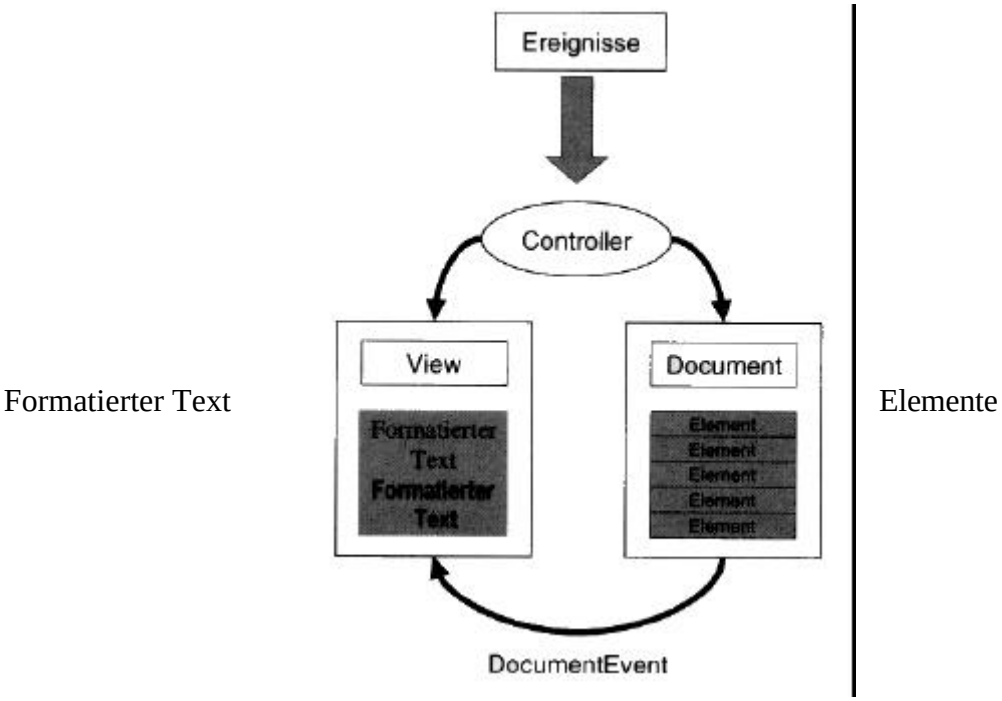


Abbildung 15-1 Ereignisse

Die Ereignisse empfängt meistens das Dokument `Document`. Bei Änderungen im `Document`, die auch für die View-Klasse interessant werden können, generiert das System ein `DocumentEvent` und sendet es an die View. Andere Ereignisse, wie das Markieren von Text mit der Maus, werden direkt vom `Document` zum View gesendet.

15.1. Implementierung eines Dokuments

Die `Document` Schnittstelle beschreibt eine unabhängige Struktur, um den Textinhalt zu modifizieren und Änderungen abzuspeichern. Die Struktur ist in grundlegende Elemente aufgeteilt. Das sind unter anderem Zeilen bzw. Absätze und die einzelnen Zeichen. Jedes einzelne Element ist mit beliebig vielen Attributen versehen. Sie enthalten Informationen über das Aussehen des Elements. Des weiteren erlaubt die Struktur die Aufzeichnung von Textänderungen für die UnDo Funktion.

Damit man nicht erst eigene Klassen entwickeln muss, die die `Document` Schnittstelle implementieren, sind bereits drei im Paket `javax.swing.text` enthalten.

15.1.1. AbstractDocument

Sie ist die Basisklasse für alle Dokumente. Eigene Klassen können von ihr abgeleitet werden. Sie selbst ist allerdings nicht zu instanzieren. Sie kümmert sich um die Zugriffsberechtigungen eines Dokuments. Es erlaubt nur einem Benutzer, schreibend auf ein Dokument zuzugreifen. Die Zahl der lesenden Zugriffe ist dagegen nicht beschränkt.

15.1.2. PlainDocument

`PlainDocument` ist eine direkte Ableitung von `AbstractDocument`. Sie unterstützt Texte nur mit Absatzattributen, nicht für einzelne Zeichen. Da die Zeichen nicht als Elemente zu ermitteln sind, können sie auch nicht mit Attributen ausgestattet werden (was aber nicht für Zeilen und Absätze gilt). Das `PlainDocument` eignet sich vor allem für einfachen Text ohne Schnörkel.

15.1.3. DefaultStyledDocument

Mit dem `DefaultStyledDocument`, das auch die Schnittstelle `StyledDocument` implementiert, können nun auch einzelne Zeichen beliebig formatiert werden. Das Anwendungsfeld des `DefaultStyledDocuments` liegt eher in der anspruchsvollen Textverarbeitung.

15.2. Verwendung eines Dokuments

Häufigen Einsatz findet beispielsweise das `DefaultStyledDocument` im `JTextPane`. Durch ihre mächtige Vielfalt an Stilen ist man nicht mehr an einfarbige Texte mit durchgehend gleichem Stil gebunden. Üblicherweise legt man zuerst das `DefaultStyledDocument` an und erzeugt anschliessend darauf die `JTextPane` mit Übergabe des Dokuments.

Listing 15-1 Programmfragment für ein JTextPanel mit Document kombiniert

```
DefaultStyledDocument meinDokument;
JTextPane textPane;
meinDokument = new DefaultStyledDocument();
textPane = new JTextPane(meinDokument);
```

Nun kommen wir zu den Textattributen; sie werden durch die Schnittstellen `AttributeSet` und `MutableAttributeSet` implementiert. Ersteres kann einen Satz von Attributen lesen, während `MutableAttribuetSet` auch Attribute einfügen und verändern kann. Swing stellt auch die Klasse `SimpleAttributeSet` bereit, welche diese Schnittstellen implementiert. Zusätzlich verwaltet sie die Attribute in einer Hash-Tabelle.

Listing 15-2 Programmfragment : Attributset

```
SimpleAttributeSet textAttr1, textAttr2;
textAttr1 = new SimpleAttributeSet();
textAttr2 = new SimpleAttributeSet();
```

Die Attribute selbst werden nun mit Hilfe der Methoden aus der Klasse `StyleConstant` gesetzt.

Listing 15-3 Setzen der Attribute

```
StyleConstants.setFontSize(textAttr1, 20);
StyleConstants.setFontFamily(textAttr1, "Dialog");
StyleConstants.setAllignment(textAttr1,
                             StyleConstants.ALIGN_CENTER);

StyleConstants.setItalic(textAttr2, true);
StyleConstants.setBold(textAttr2, true);
StyleConstants.setForeground(textAttr2, Color.red);
```

Danach fügt man einfach den Text in das Dokument der `JTextPane` mit den Attributen ein:

Listing 15-4 Einsetzen der Attribute

```
TextPane.insertString(meinDokument.getLength(),"Text 1", textAttr1);
TextPane.insertString(meinDokument.getLength(),"Text 2", textAttr2);
```

Wie im Beispiel der `JTextPane` schon besprochen wurde, können Textattribute auch mit `setCharacterAttributes`() und `setParagraphAttributes`() vergeben werden, ohne direkt auf das Dokument zuzugreifen.

15.3. Änderungen in Dokumenten (DocumentListener)

Es kann in einigen Fällen interessant sein, über Vorgänge in einem Textdokument Bescheid zu wissen. Um Änderungen im Dokument zu erfassen, wie z.B. das Einfügen oder Entfernen eines Textes oder andere Änderungen, verwendet man den `DocumentListener` im Paket `javax.swing.event`. Folgende Methoden geben Informationen über Eingaben des Benutzers:

- `insertUpdate(DocumentEvent e)`
  registriert alle Eingaben im Dokument

- `removeUpdate(DocumentEvent e)`
  wird aufgerufen, sobald ein Text oder andere Elemente aus dem Dokument entfernt werden.

- `changeUpdate(DocumentEvent e)`
  wird aufgerufen, sobald sich Attribute im Dokument ändern (zum Beispiel Schriftwart oder Ausrichtung).

Das `DocumentEvent` beschreibt das Element, das von den Änderungen betroffen ist. Unter anderem findet man dort auch Länge und Position im Dokument des Elements.

15.4. Textaktionen

Vielen Anwendern und Programmierern sind die gewöhnlichen Textaktionen bekannt (wie Ausschneiden, Kopieren, Löschen, Ausrichten, Text markieren usw.). Der übliche Weg, eine Aktion zu implementieren, wäre der `ActionListener`. Die Textkomponenten beinhalten jedoch schon alle Textaktionen, die sie unterstützen. Sie sind vom Typ `Action`. Es liegt nahe, diese zu verwenden und nicht ein zweites Mal zu definieren. Man kann sie mit `getAction`() ermitteln und in eine Hash-Tabelle (`HashTable`) laden, um sie mit ihrem Namen identifizieren zu können.

Listing 15-5 Programmfragment : Hash-Tabelle

```
private Hashtable createActionTable(JTextComponent textComponent) {
      private Hashtable actions = new Hashtable();
      Action[] actionsArray = textComponent.getActions();
      for (int i=0; i< actionsArray.length; i++) {
            Action a = actionsArray[i];
            actions.put(a.getValue(Action.NAME), a);
      }
      return actions;
}
```

Nun lässt sich bei Bedarf eine Textaktion durch Angabe des Namens besorgen, zum Beispiel mit `Action a = (Action)actions.get("Cut");`

Die Namen der Aktionen stehen allerdings auch im `DefaultEditorKit` und sind dort als Konstanten definiert. Ein weiterer Vorteil liegt darin, dass die Aktionen allen Komponenten zugewiesen werden können, die einen `ActionListener` unterstützen, wie beispielsweise Buttons und Menüpunkte.

Listing 15-6 Programmfragment : Editfunktionen

```
JMenu editMenu = new JMenu("Bearbeiten");
button = new JButton("Einfügen");

Action cutAction = actions.get(DefaultEditorKit.pasteAction);
button.addActionListener(cutAction);
editMenu.add(cutAction);
```

Natürlich ist dieses Vorgehen nur rentabel, wenn auch viele Textaktionen im Programm verwendet werden.

Bei Textstilen ist es etwas anders, da die Aktionen nicht im `DefaultEditorKit`, sondern im `StyleEditorKit` stehen. Angenommen, es soll in einer Textverarbeitung ein markierter Text mit Fettdruck formatiert werden:

Listing 15-7 Programmfragment : Formatieranweisung

```
JMenu editMenu = new JMenu("Edit");
Action a = new StyledEditorKit.BoldAction();
editMenu.edd(a);
```

15.5. Tastaturbefehle

Schliesslich lassen sich die Befehle auch mit Tastenkürzel (`KeyStroke`) verknüpfen. Man ermittelt die momentane Tastaturbelegung (`Keymap`) und fügt das Tastenkürzel mit `addActionForKeyStroke`() hinzu. Mit `removeKeyStrokeBinding`() werden sie entfernt.

Listing 15-8 Programmfragment : Tastaturbefehle

```
KeyMap keymap = textPane.getKeyMap();

// ausschneiden mit Ctrl-X
Action a = (Action)actions.get(DefaultEditorKit.cutAction);
KeyStroke key = KeyStroke.getKeyStroke(KeyEvent.VK_X,
                        Event.CTRL_MASK);
keymap.addActionForKeyStroke(key, a);
```

15.6. Beispiel : Multipad

Das Beispiel für Swing Dokumentationsysteme fällt diesmal etwas grösser aus, trägt aber zum besseren Verständnis bei. Es handelt sich hierbei um eine vollständige Textverarbeitung mit einigen Formatierungsmöglichkeiten. Viele der bisher besprochenen Themen zu Swing und zur Swing Programmierung werden nochmals in einer zusammenhängenden Anwendung behandelt.

Bei *Multipad* handelt es sich um eine MDI Textverarbeitung. Neben den Basisfunktionen unterstützt sie auch diverse Formatierungen wie Schriftart, Textgrösse, Ausrichtung und Textfarbe. Weiterhin ist das Einfügen von Bildern in GIF und JPEG Format erlaubt. Schliesslich kann das Look & Feel umgeschaltet werden. Das Hauptmerkmal ist jedoch die Unterstützung beliebig vieler Dokumente (Fenster).

Multipad funktioniert auf Basis der Klasse `JDesktopPane`. Jede neue Datei wird von einem `JInternalFrame` dargestellt. Jeder von ihnen enthält ein `JTextPane`, welches ein `StyledDocument` verarbeitet. Um auch die Frames mit diesen Komponenten und zusätzlichen Funktionen auszustatten, wurde eine Klasse `MultipadFrame` erstellt, die von `JInternalFrame` erbt. Der `MultipadFarme` bietet die Methoden, um das Dokument durch Serialisierung zu laden und zu speichern sowie Text- und Bilddateien einzubinden.

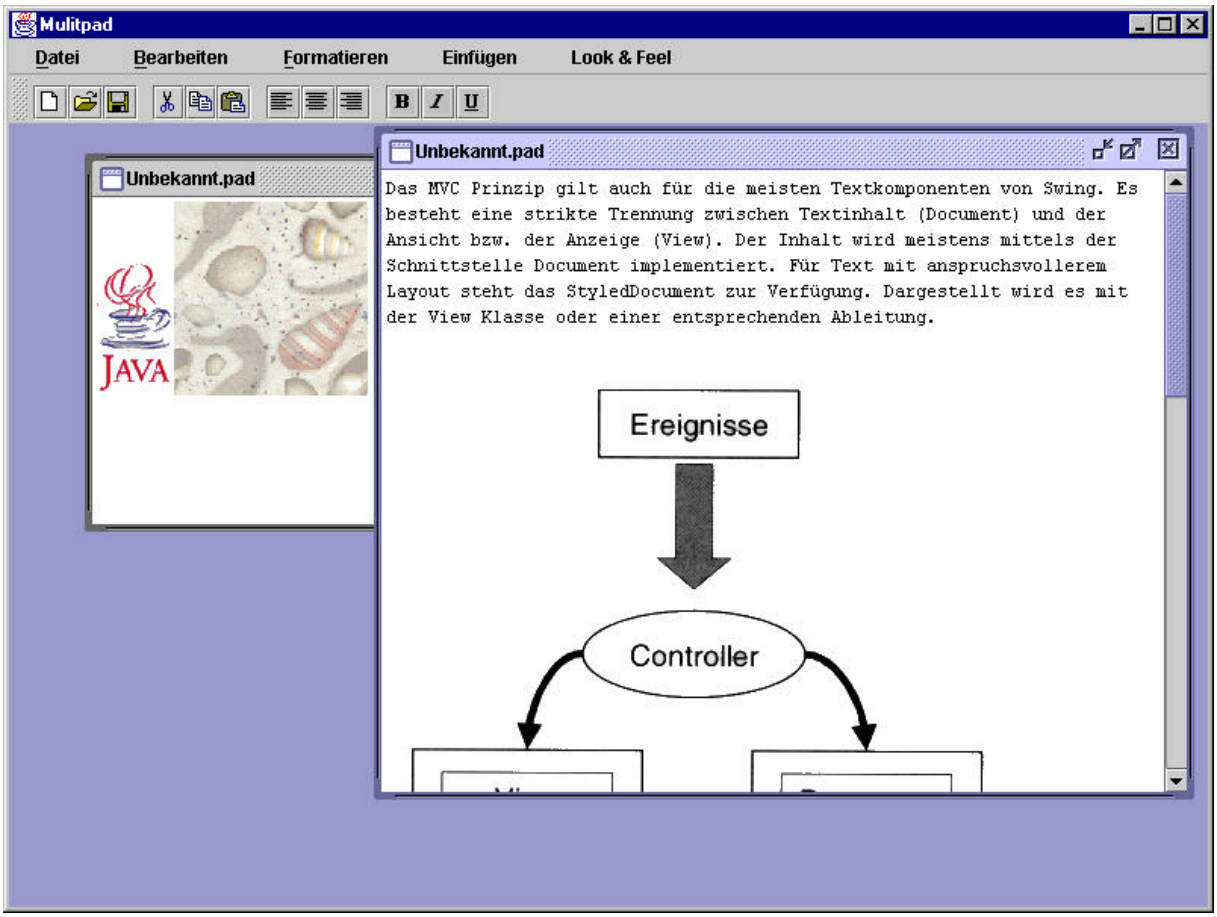


Abbildung 15-2 Beispieldokumente im Multipad

15.6.1. Menüs

Nach der Erzeugung des Panels und dem Laden der Menü-Icons werden nacheinander die Menüs aufgebaut. Ausser bei den Formatierungsfunktionen wurde für die Menüpunkte ein `ActionListener` als innere Klasse definiert. Einige Menüpunkte bieten eine Tastenkombination mit Ctrl-Taste. Bei Verwendung eines Icons muss die horizontale Textposition mit `setHorizontalTextPosition`() auf die rechte Seite gelegt werden. Folgendes Beispiel zeigt die Definition des Menüpunktes "Datei öffnen".

Listing 15-9 Datei öffnen Programmfragment

```
fileMenu.add(menuItem = new JMenuItem("Öffnen...", menuIcons[1]));
    menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O,
      ActionEvent.CTRL_MASK));
menuItem.setMnemonic('Ö');
menuItem.setHorizontalTextPosition(JButton.RIGHT);
toolBar.add(button = new JButton(menuIcons[1]));
al = new ActionListener(){

public void actionPerformed(ActionEvent e) { onFileOpen(); }

};

menuItem.addActionListener(al);
button.addActionListener(al);
```

Die Formatierungsfunktionen werden die Aktionen des `StyledEditorKit`. Das Beispiel für den Fettdruck benutzt `BoldAction`.

Listing 15-10 Programmfragment : Formatierunsgfunktion

```
formatMenu.add(subMenu = new JMenu("Zeichen"));
subMenu.add(menuItem = new JMenuItem("Fett"));
menuItem.setFont(new Font("Dialog", Font.BOLD, 12));
toolBar.add(button = new JButton(menuIcons[9]));

al = new StyledEditorKit.BoldAction();

menuItem.addActionListener(al);
button.addActionListener(al);
```

15.6.2. MultipadFrame : Serialisieren

Das Speichern des Dokuments führt die Methode `saveIt`() durch. Mit Hilfe von `FileOutputStream` und `ObjectOutput` wird zuerst ein Ausgabestrom für die gewünschte Datei generiert. Mit `writeObject`() werden danach das Dokument (`StyledDocument`) und die Masse (`Rectangle`) des Frames in die Datei geschrieben.

Listing 15-11 Programmfragment : saveIt()

```
public void saveIt(String fileName){
   if (!fileName.substring(fileName.length()-4).equals(".pad"))
    fileName = fileName + ".pad";
    File  f = new File(fileName);
    try {
      FileOutputStream fstrm = new FileOutputStream(f);
      ObjectOutput ostrm = new ObjectOutputStream(fstrm);
      ostrm.writeObject(frameDocument);
      ostrm.writeObject(getBounds());
      ostrm.flush();
      ostrm.close();
    }
    catch (IOException io) {
      System.err.println("Fehler beim Speichern: " +
                                                io.getMessage());
    }
    setTitle(fileName);
  }
```

Das Einlesen funktioniert analog mit `FileInputStream` und `ObjectInputStream` in der Methode `loadIt`().

Listing 15-12 Programmfragment : Laden eines Objekts

```
public void loadIt(String fileName)
  {
    File  f = new File(fileName);
    try {
      FileInputStream fstrm = new FileInputStream(f);
      ObjectInput istrm = new ObjectInputStream(fstrm);
      frameDocument = (StyledDocument) istrm.readObject();
      Rectangle bounds = (Rectangle) istrm.readObject();
      textPane.setStyledDocument(frameDocument);
      validate();
      setBounds(bounds);
      istrm.close();
    }
    catch (IOException io) {
      System.err.println("Loadefehler: " + io.getMessage());
    } catch (ClassNotFoundException cnf) {
      System.err.println("Formatfehler: " + cnf.getMessage());
    }
    setTitle(fileName);
  }
```

15.6.3. MultipadFrame : Einfügen von Bildern

Das Einfügen von Bildern ist sehr einfach. Die Klasse `JTextPane` kann mit `insertIcon`() Objekte vom Typ `ImageIcon` in das Dokument mit aufnehmen. Zuvor muss nur das `ImageIcon` anhand des Dateinamens eingeladen werden.

Listing 15-13 Programmfragment : Einfügen eines Bildes

```
public void insertImage(String fileName)
  {
    ImageIcon  image = new ImageIcon(fileName);
    if (image!=null)
      textPane.insertIcon(image);
  }
```

Einfach?!

15.6.4. Filter für den Datei Dialog

Dateifilter eines File-Dialogs werden mit `addChoosableFileFilter`() oder `setFileFilter`() gesetzt. Der Dateifilter wird von der Klasse `FileFilter` im Paket `javax.swing.filechooser` vertreten. Die Kernmethode von `FileFilter` ist `accept`(). Wenn der Dialog die Dateien eines Verzeichnisses auflistet, werden alle Dateien vorher mit `accept`() überprüft. Sie gibt `true` zurück, falls sie in der Dateiliste des Dialogs erscheinen soll, ansonsten `false`. Mit `getDescription`() wird die Beschreibung des Filters ermittelt. Für *Multipad* wurde eine eigene `FileFilter` Klasse (`CustomFileFilter`) entwickelt. Man gibt beim Anlegen im Konstruktor die Dateiendung und die Beschreibung an. Das folgende Beispiel zeigt die Anwendung für PAD Dateien (Endung für Dokumente von *Multipad*).

Listing 15-14 Programmfragment : FileChooser

```
public void onFileOpen()
  {
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.addChoosableFileFilter(new CustomFileFilter("pad",
"Multipad Dokumente"));
    fileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
    fileChooser.setMultiSelectionEnabled(false);
    fileChooser.setDialogTitle("Dokument laden...");
    if (fileChooser.showOpenDialog(this)==JFileChooser.APPROVE_OPTION)
    {
      setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
      onFileNew("");
      getActiveFrame().loadIt(fileChooser.getSelectedFile().toString());
      setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
    }
  }
```

Die `FileFilter` Klasse wurde folgendermassen implementiert:

Listing 15-15 Programmfragment : FileFilter

```
class CustomFileFilter extends javax.swing.filechooser.FileFilter
{
  String  extension, description;

  CustomFileFilter(String ex, String dsc)
  {
    super();
    extension = ex;
    description = dsc;
  }

  public boolean accept(File file)
  {
    if (file.isDirectory())
      return true;

    String fileName = file.getName();
    int i = fileName.lastIndexOf('.');

    if (i>0 &&  i<fileName.length() - 1)
    {
      String ext = fileName.substring(i+1).toLowerCase();
      if (ext.equals(extension))
        return true;
      else
        return false;
    }
    return false;
  }
  public String getDescription()
  {
    return description;
  }
}
```

15.6.5. Umschalten des Look & Feel

Das Umschalten in ein anderes Look & Feel ist schnell implementiert. Mit `setLookAndFeel()` der Klasse `UIManager` wird dieser Vorgang getätigt. Um sicherzugehen, dass wirklich alle Komponenten im gewählten Look & Feel neu gezeichnet werden, stösst man mit `updateComponentTreeUI()` einen Refresh an.

Listing 15-16 Programmfragment : Setzen und ändern des Look & Feel

```
public void setLookAndFeel(String lnf){
    String lnfName;

    lnfName = "javax.swing.plaf.metal.MetalLookAndFeel";
    if (lnf.equals("Motif"))
      lnfName = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
    if (lnf.equals("Windows"))
      lnfName = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";
    try
    {
      UIManager.setLookAndFeel(lnfName);
      SwingUtilities.updateComponentTreeUI(this);
```

```
    }
    catch (Exception exc) {
      System.err.println("Konnte das Look&Feel nicht laden: " + lnfName);
    }
  }
```

15.7. Anhang : Links und Quellen

Viele der Programmbeispiele stammen aus dem Web.

Hier einige der besseren Quellen / Web Adressen:

Magazine / News / Tips & Tricks	
Sun Java Web-Site	http://java.sun.com
Java Foundation Classes	http://java.sun.com/products/jfc
Java Developer Connection	http://java.sun.com/jdc
Java Online Documentation	http://java.sun.com/docs/books
Java World	http://www.javaworld.com
	enthält viele Beispiele mit Source Code
Developer	http://www.developer.com
Java Report Onlie	http://sigs.com/jro
CMP TechWeb Super Site	http://www.techweb.com
Web Developer Online	http://www.webdeveloper.com
Focus on Java	http://java.miningco.com/pjd
Pure Java Developers Journal	http://www.cobb.com/pjd
Wandering Man	http://www.wandering-man.com
JavaZine	http://www.JavaUniverse.com
Cafe au Lait	http://metalab.unc.edu/javafaq
	befindet sich auf dem www.switch.ch Mirror:
	http://sunsite.cnlab-switch.ch/javafaq
IBM Java	http://www.ibm.com/java
Microsoft	http://www.microsoft.com/java

Entwicklungsumgebungen	
Sun Java Workshop	http://www.sun.com/workshop/java
Inprise Borland JBuilder	http://www.inprise.com/jbuilder
Symantec Visual Cafe	http://cafe.symantec.com
IBM Visual Age for Java	http://www.software.ibm.com/ad/vajava
Microsoft Visual J++	http://www.microsoft.com/visualj
Tek-Tools Kawa	http://www.tek-tools.com/kawa
Tek-Tools jForge	http://www.tek-tools.com/jForge

GUI PROGRAMMIERUNG MIT SWING