GUI-KOMPONENTEN AUS JAVAX.SWING

Eine Auswahl an GUI-Komponenten aus javax.swing.

1 ALLGEMEINES:

Zur Verwendung der hier angegebenen Elemente werden hier einige allgemeine Punkte besprechen (die ersten 2 Punkte sollten reine Wiederholung sein).

1.1 Konstruktoren

Von allen Konstruktoren wird hier die Kopfzeile angegeben. Diese Kopfzeilen geben Aufschluss über die Verwendungsmöglichkeiten der Konstruktoren.

Generell werden die Konstruktoren nach Aufruf des Schlüsselwortes **`new`** verwendet. Die Konstruktoren werden hier an Beispiel eines Labels (siehe folgenden Abschnitt) besprochen. Für ein Label gibt es folgende Konstruktoren:

```
public JLabel()                                                    (a)
public JLabel(String text)                                         (b)
public JLabel(String text, int horizontalAlignment)                (c)
```

Konstruktor ohne Parameter (a)

Wenn ein Konstruktor ohne Parameter angegeben wird, dann kann auch ein Aufruf ohne Parameter erfolgen. Da es bei fast allen Klassen so einen Konstruktor gibt, kann diese Art des Aufrufes fast immer erfolgen. Der Aufruf sieht nun so aus:

```
new JLabel();
```

Da ein Konstruktor-Aufruf (das Erzeugen eines Objektes) nur sinnvoll ist, wenn das Ergebnis auch irgendwo festgehalten wird, sieht der vollständige Aufruf folgendermaßen aus (wenn man davon ausgeht, dass eine Referenzvariable mit dem Namen **`label1`** zuvor mit **`JLabel label1;`** deklariert wurde):

```
label1 = new JLabel();
```

Bei Konstruktoren ohne Parameter wird die entsprechende Komponente mit Standardwerten erzeugt.

Konstruktor mit einem Parameter (b)

Der zweite Konstruktor gibt an, dass hier ein String-Objekt zwischen die Klammern geschrieben gehört. Der Typ und die Anzahl der Parameter sind durch die Angabe der Variablen-Deklarationen festgelegt. In diesem Fall wird eine **`String`**-Referenzvariable mit dem Namen **`text`** deklariert, d.h. es muss beim Aufruf genau ein **`String`**-Objekt angegeben werden. Was dieses **`String`**-Objekt genau bewirkt, wird bei der Angabe der Konstruktoren anschließend im Text erklärt. So steht dort zum Beispiel:

*Wenn der Parameter **`text`** übergeben wird, verwendet das JLabel ihn als Beschriftung.*

Nun muss nur noch ein passender Text eingegeben werden, zum Beispiel "Hallo!".

```
label1 = new JLabel("Hallo!");
```

Konstruktor mit mehreren Parametern (c)

Wenn bei einem Konstruktor mehrere Parameter möglich sind, so müssen diese genau in der richtigen Reihenfolge angegeben werden. Auch hier sind Typ und Anzahl der Parameter durch die Deklaration der Variablen festgelegt. In unserem Fall braucht man ein **`String`**-Objekt und einen **`int`**-Wert. Die Bedeutung der einzelnen Punkte wird wieder im Text erklärt:

*Wenn der Parameter **`text`** übergeben wird, verwendet das JLabel ihn als Beschriftung. Der Parameter **`horizontalAlignment`** gibt an, wie die Beschriftung im JLabel entlang der horizontalen Achse plaziert werden soll, falls links und rechts mehr Platz als notwendig zur Verfügung steht. Hier kann eine der Konstanten **`LEFT`**, **`CENTER`** oder **`RIGHT`** aus dem Interface **`SwingConstants`** angegeben werden.*

Der Konstruktor braucht also ein **String**-Objekt als Beschriftung und einen **int**-Wert, der die Ausrichtung festlegt. Weiters wird beschrieben, dass als Ausrichtung nur eine von 3 Konstanten, nämlich **SwingConstants.LEFT**, **SwingConstants.RIGHT** und **SwingConstants.CENTER**, verwendet wird Die einzelnen Parameter werden durch Beistriche getrennt. Ein möglicher Konstruktor-Aufruf könnte folgendermaßen aussehen:

```
label1 = new JLabel("Hallo", SwingConstants.CENTER);
```

1.2 Methoden

Methoden werden auf bereits erzeugte Objekte angewendet. Irgendwo vorher muss bereits eine passende Referenzvariable deklariert und das Objekt erzeugt worden sein, bei einem Label z.B. mit den Zeilen:

```
JLabel label1;                      //Deklaration der Referenzvariablen
label1 = new JLabel("Hallo!");      //Erzeugen des Objektes und Zuweisung
```

Das Anwenden einer Methode auf ein Objekt erfolgt durch die Schreibweise:

Rückgabewert **=** *Referenzvariable***.***Methode***(***Parameter***);**

Ob eine Methode Rückgabewert oder Parameter hat, wird durch die Methoden-Auflistung angegeben. Sehen wir uns wieder einige Methoden von **JLabel** an:

```
public void setText(String text)
public String getText()
public void setHorizontalAlignment(int alignment)
public int getHorizontalAlignment()
```

Rückgabewert

Ob die Methode einen Rückgabewert hat, wird durch die Angabe des Rückgabetyps festgelegt. Der Rückgabetyp steht immer direkt vor dem Namen der Methode (gleich nach **public**). Es gibt 3 Möglichkeiten für einen Rückgabetyp:

1. **void**: Wenn void als Rückgabetyp steht, dann hat die Methode keinen Rückgabetyp. D.h. die Methode wird nur mit *Referenzvariable***.***Methode***(***Parameter***);** aufgerufen (ohne *Rückgabewert* **=**).
2. Variablentyp: Wenn ein Variablentyp als Rückgabetyp steht, dann muss als Rückgabewert einfach eine Variable vom gleichen Typ, wie der Rückgabetyp stehen, der Aufruf erfolgt dann mit *Variable* **=** *Referenzvariable***.***Methode***(***Parameter***);**.
3. Klasse: Wenn die Methode eine Klasse als Rückgabetyp hat, dann muss eine Referenzvariable vom Typ der Klasse vor der Methode stehen. Der Aufruf der Methode erfolgt dann mit *andereReferenzvariable* **=** *Referenzvariable***.***Methode***(***Parameter***);**

Für die angegebenen Methoden der Klasse **JLabel** sieht das dann folgendermaßen aus:

Methodenbeschreibung	Rückgabetyp	Beispiel
`public void setText(String text)`	**void** (Nichts)	`label1.setText("Goodbye!");`
`public String getText()`	Klasse **String**	`String ausgabe = label1.getText();`
`public void setHorizontalAlignment(int alignment)`	**void** (Nichts)	`label1.setHorizontalAlignment (SwingConstants.RIGHT);`
`public int getHorizontalAlignment()`	Variablentyp **int**	`int zahl = label1.getHorizontalAlignment();`

Parameter

Die Parameter folgen den gleichen Regeln wie bei den Konstruktoren. Bei den angegebenen Beispiel-Methoden der Klasse **JLabel** gibt es 2 Methoden ohne Parameter, eine Methode mit einem **String**-Objekt als Parameter und eine Methode mit einer **int**-Variablen als Parameter.

1.3 Ereignissteuerung:

Um ein Steuerelement auf Ereignisse reagieren zu lassen, muss zunächst einmal eine innere Klasse geschrieben werden (innerhalb von **public class**, aber nicht innerhalb einer Methode – wie z.B. dem Konstruktor). Diese hat die Form

```
private class Name implements XxxxxListener {
}
```

So eine Klasse kann für einen Button z.B. folgendermaßen aussehen:

```
private class ButtonHandler implements ActionListener {
}
```

Der Teil **implements** *Xxxxx***Listener** gibt an, dass hier ein Interface mit dem Namen *Xxxxx***Listener** verwendet wird. Interfaces zwingen uns dazu, ihre Methoden zu überschreiben. Diesen Methoden wird als Parameter das eigentliche Ereignis übergeben. Die Methode muss im Fall des **ActionListeners** den Namen **actionPerformed** haben, während das Ereignis ein **ActionEvent** ist. Dadurch sieht unsere **ActionListener**-Klasse folgendermaßen aus:

```
private class ButtonHandler implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    // Hier steht der Code, der angibt, wie auf das Ereignis reagiert
    // werden soll.
  }
}
```

Innerhalb dieser Klasse wird oft mit Hilfe der **getSource()**-Methode vom **ActionEvent**-Parameter bestimmt, welche Komponente das Ereignis ausgelöst hat, z.B. mit **if(e.getSource() == meinButton)** (siehe auch JButton-Beispiel). Dies ist einer der wenigen Fälle, wo ein direkter Vergleich der Objektreferenzen mit == sinnvoll ist, weil mich ja wirklich interessiert, ob das gleiche Button-Objekt im Speicher, das mit der Variable **meinButton** angesprochen wird, auch das Ereignis ausgelöst hat.

Aus dieser inneren Klasse muss nun innerhalb einer Methode (in unserem Fall innerhalb des Konstruktors) zuerst ein Objekt erzeugt werden. Ein Objekt unserer obigen Klasse müsste mit

```
ButtonHandler h1 = new ButtonHandler();
```

angelegt werden. Dieses **Listener**-Objekt muss nun mit einer **add***Xxxxx***Listener()**-Methode zum **Component** hinzugefügt werden. Bei einem Button mit dem Namen **meinButton** sieht dies so aus:

```
meinButton.addActionListener(h1);
```

Im Folgenden werden nur noch der Name der **Listener**-Klasse, der Name der Methode für das Ereignis und der Name des Ereignisses, das der Methode übergeben wird, angegeben.

2 Allgemeine Methoden

Es gibt etliche Methoden, die für viele Komponenten gemeinsam wichtig sind. Eine Auswahl davon stelle ich hier vor:

```
public void setText(String text)
public String getText()
```

Diese Methoden können bei allen Komponenten, die eine Beschriftung haben oder mit Text arbeiten (z.B. **JLabel**, **JButton**, **JTextField**,...), auf diese Beschriftung bzw. den Text zugreifen. Bei einigen Komponenten macht dies jedoch keinen Sinn, z.B bei **JSlider**. Sie haben daher diese Methoden nicht.

```
public void setOpaque(boolean isOpaque)
public void setBackground(Color bg)
```

Die erste Methode gibt an, ob die Komponente undurchsichtig ist (**isOpaque** hat den Wert **true**) oder darunterliegende Komponenten durchscheinen können. Das genaue Verhalten ist von Komponente unterschiedlich. Wenn die Komponente auf undurchsichtig gestellt ist, kann mit der Methode **setBackground** eine Hintergrundfarbe angegeben werden. Sie benötigt dazu ein Color-Objekt, das entweder mit einem Konstruktor selbst erzeugt wird oder mit Hilfe von vorgefertigten Konstanten übergeben wird (z.B. **label1.setBackground(Color.YELLOW);**).

```
public void setFont(Font font)
```

Mit dieser Methode kann man die Schriftart, -größe und den Schriftstil der GUI-Komponente angeben. Dazu muss erst ein Font-Objekt erzeugt werden. Dieses kann dann im Parameter angegeben werden, z.B.

```
Font schrift = new Font(Font.SANS_SERIF, Font.BOLD, 16);
label1.setFont(schrift);
```

Hier wird eine serifenlose Schrift gewählt, die fett und mit Schriftgröße 16 auf das **JLabel** angewendet wird.

public void setEnabled(boolean enabled)

Wenn bei **setEnabled** der Wert **false** übergeben wird, dann wird die Komponente deaktiviert. Das bedeutet, dass sie nicht mehr ausgewählt werden kann und auch ausgegraut dargestellt wird

3 JLabel

Ein **JLabel** dient zur Anzeige von kurzen Texten, die auch mit Icons dekoriert werden können. Ein **JLabel** hat keine Ereignissteuerung, es kann auch nicht den Fokus erhalten.

3.1 Konstruktoren

public JLabel()
public JLabel(String text)
public JLabel(String text, int horizontalAlignemnt)

Der Konstruktor ohne Parameter erzeugt ein **JLabel** ohne Text-Inhalt. Wenn der Parameter **text** übergeben wird, verwendet das **JLabel** ihn als Beschriftung. Der Parameter **horizontalAlignment** gibt an, wie die Beschriftung im **JLabel** entlang der horizontalen Achse plaziert werden soll, falls links und rechts mehr Platz als notwendig zur Verfügung steht. Hier kann eine der Konstanten **LEFT**, **CENTER** oder **RIGHT** aus dem Interface **SwingConstants** angegeben werden. Es gibt auch Konstruktoren, die ein Icon-Objekt als Parameter übernehmen können. Die Betrachtung von Icons sprengt aber hier den Rahmen.

3.2 Weitere Methoden

public void setHorizontalAlignment(int alignment)
public int getHorizontalAlignment()
public void setVerticalAlignment(int alignment)
public int getVerticalAlignment()

setHorizontalAlignment und **getHorizontalAlignment** ermöglichen den Zugriff auf die horizontale Ausrichtung des Texts. Der Parameter der set-Methode hat dieselbe Bedeutung wie im Konstruktor. Auf die Ausrichtung in vertikaler (senkrechter) Richtung kann mit den Methoden **setVerticalAlignment** und **getVerticalAlignment** zugegriffen werden. Als Parameter für die set-Methode können aus dem Interface **SwingConstants** die Konstanten **TOP**, **CENTER** und **BOTTOM** verwendet werden.

3.3 Beispiel:

```
import javax.swing.*;
public class LabelBeispiel extends JFrame {
  private LabelPanel lp;
  public LabelBeispiel() {
    super("LabelBeispiel");
    lp = new LabelPanel();
    this.add(lp);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setBounds(100, 200, 120, 150);
    this.setVisible(true);
  }
  public static void main(String[] args) {
    new LabelBeispiel();
  }
}
```

```java
import java.awt.*;    // Für die Layouts
import javax.swing.*; // Für die GUI-Komponenten

public class LabelPanel extends JPanel{
  private JLabel l1, l2, l3, l4;

  public LabelPanel() {
    this.setLayout(new GridLayout(4,1));
    l1 = new JLabel("Default");
    l2 = new JLabel("Links",SwingConstants.LEFT);
    l3 = new JLabel("Zentriert",SwingConstants.CENTER);
    l4 = new JLabel("Rechts",SwingConstants.RIGHT);
    // Eine Möglichkeit die Ausrichtung nachträglich zu ändern:
    // l4.setHorizontalAlignment(SwingConstants.CENTER);
    this.add(l1);
    this.add(l2);
    this.add(l3);
    this.add(l4);
  }
}
```

Die Frame-Klasse wird in den unteren Beispielen nicht mehr angegeben, um Platz zu sparen. Sie ist immer gleich aufgebaut und ändert nur die Werte für die Größe bei der Methode setBounds.

4 JButton

Ein **JButton** ist die Umsetzung eines "Druckknopfes" am Computer. Dazu hat er eine Beschriftung und kann auch Icons aufnehmen (Icons werden hier nicht besprochen).

4.1 Konstruktoren

public JButton()
public JButton(String text)

Der Konstruktor ohne Parameter erzeugt ein **JButton**-Objekt ohne Beschriftung. Beim Konstruktor mit String-Parameter kann die Beschriftung gleich mit angegeben werden.

Bei der Klasse **JButton** sind vor allem die allgemeinen Methoden für den alltäglichen Gebrauch wichtig.

4.2 Ereignissteuerung

Das Anklicken eines **JButtons** löst ein **ActionEvent** aus, das an die durch **addActionListener** registrierten Objekte gesendet wird.

public void addActionListener(ActionListener l)

Die Klassen dieser Objekte müssen dementsprechend das Interface **ActionListener** implementieren, damit die Methode **actionPerformed** vorhanden ist. Diese wird durch den Klick aufgerufen und sie bekommt über den Parameter das auslösende **ActionEvent** mitgeschickt.

public void actionPerformed(ActionEvent e)

4.3 Beispiel

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

public class ButtonPanel extends JPanel {
  private JButton setzen;
  private JButton loeschen;
  private JLabel ausgabe;
```

```
  public ButtonPanel() {
    this.setLayout(new GridLayout(3,1));
    ausgabe = new JLabel();
    setzen = new JButton("Setzen");
    loeschen = new JButton();
    //nachträgliches Setzen der Beschriftung
    loeschen.setText("Löschen");
    this.add(ausgabe);
    this.add(setzen);
    this.add(loeschen);
    ButtonHandler handler = new ButtonHandler();
    setzen.addActionListener(handler);
    loeschen.addActionListener(handler);
  }

  private class ButtonHandler implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent event) {
      if(event.getSource() == setzen) {
        ausgabe.setText("Beispieltext");
      }
      else if(event.getSource() == loeschen) {
        ausgabe.setText("");
      }
    }
  }
}
```

5 JCheckBox

Eine **JCheckBox** ist eine Komponente, die entweder ausgewählt oder nicht ausgewählt sein kann, was meistens durch ein Häkchen oder ein X symbolisiert wird. Zusätzlich hat eine Checkbox noch eine Beschriftung, die üblicher Weise rechts neben der eigentlichen Checkbox angezeigt wird.

5.1 Konstruktoren

```
public JCheckBox()
public JCheckBox(String text)
public JCheckBox(String text, boolean selected)
```

Der parameterlose Konstruktor erzeugt ein **JCheckBox** ohne Beschriftung mit dem Anfangszustand **false**. Der Text-Parameter bei den Konstruktoren setzt die Beschriftung auf den angegebenen Text. Über den Parameter **selected** kann man angeben, ob die Checkbox am Anfang ausgewählt ist (**selected** auf **true**) oder nicht (**selected** auf **false**).

5.2 Weitere Methoden

```
public boolean isSelected()
public void setSecelected(boolean selected)
```

Mit Hilfe dieser Methoden kann man den Zustand der Checkbox abfragen bzw. per Programmcode setzen.

5.3 Ereignissteuerung

Eine Checkbox kann mit der gleichen Ereignissteuerung arbeiten, wie ein Button, d.h. als Listener wird **ActionListener** mit der Methode **actionPerformed** verwendet. Der **ActionListener** reagiert auf das Klick-Ereignis der Checkbox.

Man kann aber auch den **`ItemListener`** verwenden. Dieser reagiert auf jegliche Änderung des Zustandes (von ausgewählt auf nicht ausgewählt und umgekehrt), auch wenn diese Änderung durch den Programmcode erfolgt. Die Methode zum Registrieren des **`ItemListeners`** lautet:

`public void addItemListener(ItemListener l)`

Die Klasse eines Objekts, das man damit registrieren möchte, muss damit das Interface ItemListener implementieren. Dadurch wird folgende Methode für diese Klasse vorgeschrieben:

`public void itemStateChanged(ItemEvent e)`

Diese Methode bekommt über den **`ItemEvent`**-Parameter die Informationen über die Checkbox, die das Event ausgelöst hat, mit geschickt.

5.4 Beispiel

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

public class CheckboxPanel extends JPanel{
  private JCheckBox cb1, cb2, cb3;
  private JLabel ausgabe;

  public CheckboxPanel (){
    this.setLayout(new GridLayout(4,1));
    cb1 = new JCheckBox("Checkbox 1");
    this.add(cb1);
    cb2 = new JCheckBox("Checkbox 2", true);
    this.add(cb2);
    cb3 = new JCheckBox("Checkbox 3", false);
    this.add(cb3);
    ausgabe = new JLabel("Ausgewählt: 2");
    this.add(ausgabe);
    CheckboxHandler cbh = new CheckboxHandler();
    cb1.addActionListener(cbh);
    cb2.addActionListener(cbh);
    cb3.addActionListener(cbh);
  }

  private class CheckboxHandler implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
      String ausgewaehlt = "";
      if(cb1.isSelected())
        ausgewaehlt = "1";
      if(cb2.isSelected()) {
        if(!ausgewaehlt.equals(""))
          ausgewaehlt = ausgewaehlt + ", ";
        ausgewaehlt = ausgewaehlt + "2";
      }
      if(cb3.isSelected()) {
        if(!ausgewaehlt.equals(""))
          ausgewaehlt = ausgewaehlt + ", ";
        ausgewaehlt = ausgewaehlt + "3";
      }
      ausgabe.setText("Ausgewählt: " + ausgewaehlt);
    }
  }
}
```

Die Variante mit dem ItemListener sieht sehr ähnlich aus. Die Funktionsweise ist in diesem Fall auch ident.

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

public class CheckboxPanel extends JPanel{
  private JCheckBox cb1;
  private JCheckBox cb2;
  private JCheckBox cb3;
  private JLabel ausgabe;

  public CheckboxPanel (){
    this.setLayout(new GridLayout(4,1));
    cb1 = new JCheckBox("Checkbox 1");
    this.add(cb1);
    cb2 = new JCheckBox("Checkbox 2", true);
    this.add(cb2);
    cb3 = new JCheckBox("Checkbox 3", false);
    this.add(cb3);
    ausgabe = new JLabel("Ausgewählt: 2");
    this.add(ausgabe);
    CheckboxHandler cbh = new CheckboxHandler();
    cb1.addItemListener(cbh);
    cb2.addItemListener(cbh);
    cb3.addItemListener(cbh);
  }

  private class CheckboxHandler implements ItemListener {
    @Override
    public void itemStateChanged(ItemEvent e) {
      System.out.println("ItemListener!");
      String ausgewaehlt = "";
      if(cb1.isSelected())
        ausgewaehlt = "1";
      if(cb2.isSelected()) {
        if(!ausgewaehlt.equals(""))
          ausgewaehlt = ausgewaehlt + ", ";
        ausgewaehlt = ausgewaehlt + "2";
      }
      if(cb3.isSelected()) {
        if(!ausgewaehlt.equals(""))
          ausgewaehlt = ausgewaehlt + ", ";
        ausgewaehlt = ausgewaehlt + "3";
      }
      ausgabe.setText("Ausgewählt: " + ausgewaehlt);
    }
  }
}
```

6 JRadioButton

Radiobuttons arbeiten ähnlich wie Checkboxen, nur dass von einer Gruppe immer nur ein Radiobutton ausgewählt werden kann. Die Konstruktoren bzw. Methoden sind daher analog zu den Möglichkeiten, die **`JCheckBox`** bietet. Genauso wie die Checkbox kann ein Radiobutton auch mit einem **`ActionListener`** und einem **`ItemListener`** arbeiten.

Da aber bestimmt werden muss, von welcher Gruppe von Radiobuttons immer nur einer ausgewählt werden soll, müssen sie zu einem **`ButtonGroup`**-Objekt hinzugefügt werden.

6.1 Beispiel

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

public class RadioButtonPanel extends JPanel{
  private JRadioButton rb1, rb2, rb3, rb4, rb5;
  private GrafikPanel gp;

  public RadioButtonPanel() {
    this.setLayout(new GridLayout(3,2));
    ButtonGroup group1 = new ButtonGroup();
    ButtonGroup group2 = new ButtonGroup();
    RadioButtonHandler rbh = new RadioButtonHandler();
    rb1 = new JRadioButton("rot",true);
    rb1.addActionListener(rbh);
    rb2 = new JRadioButton("blau",false);
    rb2.addActionListener(rbh);
    rb3 = new JRadioButton("gelb",false);
    rb3.addActionListener(rbh);
    rb4 = new JRadioButton("eckig",true);
    rb4.addActionListener(rbh);
    rb5 = new JRadioButton("rund",false);
    rb5.addActionListener(rbh);
    group1.add(rb1);
    group1.add(rb2);
    group1.add(rb3);
    group2.add(rb4);
    group2.add(rb5);
    this.add(rb1);
    this.add(rb4);
    this.add(rb2);
    this.add(rb5);
    this.add(rb3);
    gp = new GrafikPanel();
    this.add(gp);
  }

  private class RadioButtonHandler implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
      if(rb1.isSelected())
          gp.setColor(Color.RED);
      if(rb2.isSelected())
          gp.setColor(Color.BLUE);
      if(rb3.isSelected())
          gp.setColor(Color.YELLOW);
      if(rb4.isSelected())
          gp.setForm(GrafikPanel.RECHTECK);
      if(rb5.isSelected())
          gp.setForm(GrafikPanel.OVAL);
    }
  }
}
```

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

public class GrafikPanel extends JPanel{
```

```
  private Color c;
  private int form;
  public static final int OVAL = 1;
  public static final int RECHTECK = 2;

  public GrafikPanel() {
    this.c = Color.RED;
    this.form = 1;
  }

  @Override
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(c);
    switch(form) {
    case OVAL:
      g.fillOval(0,0, this.getWidth(), this.getHeight());
      break;
    case RECHTECK:
      g.fillRect(0,0, this.getWidth(), this.getHeight());
      break;
    }
  }

  public void setForm(int form) {
    this.form = form;
    repaint();
  }

  public void setColor(Color c) {
    this.c = c;
    repaint();
  }
}
```

7 JTextField

Ein **`JTextField`** ist eine Text-Komponente, die primär zum Eingeben von Text dient, die aber auch zur reinen Anzeige von Text verwendet werden kann. Beim Textfeld kann angegeben werden, wie viele Spalten (entspricht ungefähr der Anzahl der Zeichen) angezeigt werden sollen. Daraus berechnet sich das Textfeld seine bevorzugte Breite. Die bevorzugte Breite wird aber nur verwendet, wenn der Layout-Manager keine andere Breite vorgibt.

7.1 Konstruktoren

```
public JTextField()
public JTextField(int columns)
public JTextField(String text)
public JTextField(String text, int columns)
```

Der Konstruktor ohne Parameter erzeugt ein leeres Textfeld, dessen bevorzugte Breite minimal ist (so klein, dass man im Allgemeinen keinen Buchstaben sehen kann). Der zweite Konstruktor erzeugt ein leeres Textfeld, dessen bevorzugte Breite so groß ist, dass mindestens so viele Zeichen, wie bei **`columns`** angegeben, angezeigt werden können. Da meistens Proportionalschriften verwendet werden, wo die Buchstaben unterschiedlich breit sind, ist diese Angabe meist ein Richtwert und es sind im Allgemeinen auch mehr Buchstaben zu sehen. Der Konstruktor mit einem String-Parameter erzeugt ein Textfeld dessen Inhalt bereits auf den entsprechenden Text gesetzt ist. Und der letzte Konstruktor kombiniert die beiden vorherigen Konstruktoren.

7.2 Weitere Methoden

Das jede Textkomponente hat Methoden, die sich speziell auf die Eingabemöglichkeiten von Text beziehen. So gibt es z.B. Methoden, die verhindern, dass überhaupt Text eingegeben werden kann. Im Gegensatz zur Methode **setEnabled** (siehe Kapitel 2) wird hier nicht die ganze Komponente deaktiviert und damit ausgegraut dargestellt, sondern nur verhindert, dass Text eingegeben werden kann:

```
public void setEditable(boolean allowed)
public boolean isEditable()
```

Durch einen Aufruf von **setEditable** mit **false** als Parameter werden weitere Eingaben unterbunden. Der aktuelle Status kann mit **isEditable** abgefragt werden. Auf die aktuelle Cursorposition innerhalb des Textes kann mit den Methoden **getCaretPosition** und **setCaretPosition** zugegriffen werden:

```
public int getCaretPosition()
public void setCaretPosition(int position)
```

Weiters existieren Methoden, mit denen der eingegebene Text markiert werden kann.

```
public void selectAll()
public void select(int selectionStart, int selectionEnd)
```

selectAll markiert den kompletten Text und **select** den Bereich von **selectionStart** bis **selectionEnd** (wie immer beginnt in der Informatik die Zählung bei 0). Entsprechend liefern die beiden Methoden **getSelectionStart** und **getSelectionEnd** kann die Positionen der aktuellen Auswahl und die Methode **getSelectedText** liefert den ausgewählten Text:

```
public int getSelectionStart()
public int getSelectionEnd()
public String getSelectedText()
```

7.3 Ereignissteuerung

Textkomponenten bieten mehrere Möglichkeiten zur Ereignissteuerung. Die erste und eine von den einfacheren Möglichkeiten ist die Steuerung mittels **ActionListener**. Ein **ActionEvent** wird von einem Textfeld immer dann ausgelöst, wenn innerhalb des Textfeldes die ENTER-Taste gedrückt wird.

Des weiteren bietet ein Textfeld die Möglichkeit auf Änderungen der Cursorposition zu reagieren. Dieses Ereignis wird mit Hilfe eines **CaretListener**s überwacht, entsprechend heißt die Methode zum Registrieren einer entsprechenden Ereignissteuerung

```
public void addCaretListener(CaretListener l)
```

Die Klasse eines Objekts, das man damit registrieren möchte, muss damit das Interface CaretListener implementieren. Dadurch wird folgende Methode für diese Klasse vorgeschrieben:

```
public void caretUpdate(CaretEvent e)
```

Außerdem gibt es noch die Möglichkeit den Text des Textfeldes selbst zu überwachen. Da Swing-Komponenten und insbesondere Textkomponenten aber nach dem MVC-Prinzip gestaltet wurden, ist der Text nicht direkt in der GUI-Komponente gespeichert, sondern in einem eigenen Model. Dieses Model ist ein Objekt der Document-Klasse, das man sich zuerst von der Textkomponente holen muss, z.B. mit

```
Document doc = textfeld.getDocument();
```

erst auf diesem Document-Objekt kann nun eine entsprechende Ereignissteuerung mit der Methode

```
public void addDocumentListener(DocumentListener l)
```

registriert werden. Der DocumentListener überwacht 3 unterschiedliche Ereignisse: changedUpdate (wenn sich der Stil bei formatierten Textkomponenten ändert), insertUpdate (wenn ein Text eingefügt wird) und deleteUpdate (wenn ein Text gelöscht wird). Dementsprechend müssen auch 3 Methoden überschrieben werden, wenn man eine Klasse, die das DocumentListener-Interface implementiert, schreibt

```
public void changedUpdate(DocumentEvent e)

public void insertUpdate(DocumentEvent e)

public void deleteUpdate(DocumentEvent e)
```

7.4 Beispiel

```
import javax.swing.*;
import javax.swing.event.*; // Für DocumentListener, CaretListener
import javax.swing.text.*;  // Für die Klasse Document
import java.awt.*;
import java.awt.event.*;

public class TextFieldPanel extends JPanel {
  private JTextField tfEingabe;
  private JTextField tfLaenge;
  private JTextField tfCursor;
  private JLabel lText;
  private JLabel lLaenge;
  private JLabel lCursor;

  public TextFieldPanel() {
    this.setLayout(new BorderLayout());
    // Erzeugt ein Textfeld mit einer Breite von 15 Zeichen.
    // Die Breite ist aber beim oberen Bereich des
    // BorderLayouts nicht relevant und wird ignoriert
    tfEingabe = new JTextField(15);
    tfEingabe.setBackground(new Color(255,255,0));
    // Erzeugt ein Textfeld in dem der Wert 0 steht und das
    // 2 Zeichen breit ist
    tfLaenge = new JTextField(
        "" + tfEingabe.getText().length(), 2);
    // Komplettes Sperren des Textfeldes
    tfLaenge.setEnabled(false);
    lLaenge = new JLabel("Textlänge = ");
    tfCursor = new JTextField("0",2);
    // Nur Sperren der Eingabe vom Textfeld! Vergleiche beim Aus-
    // führen die Auswirkungen mit dem komplett gesperrten Textfeld
    tfCursor.setEditable(false);
    lCursor = new JLabel("Cursorposition = ");
    lText = new JLabel("zum Übernehmen des Textes Enter drücken");
    // Positionierung am Layout
    this.add(tfEingabe, BorderLayout.PAGE_START);
    JPanel pAus = new JPanel();
    pAus.add(lLaenge);
    pAus.add(tfLaenge);
    pAus.add(lCursor);
    pAus.add(tfCursor);
    this.add(pAus);
    this.add(lText, BorderLayout.PAGE_END);

    // Ereignissteuerung für die Enter-Taste
    ActionHandler ah = new ActionHandler();
    tfEingabe.addActionListener(ah);
    // Ereignissteuerung für den TextInhalt
    DocumentHandler th = new DocumentHandler();
    Document doc = tfEingabe.getDocument();
    doc.addDocumentListener(th);
    // Ereignissteuerung für die Cursor-Position
    CursorHandler ch = new CursorHandler();
    tfEingabe.addCaretListener(ch);
  }

  // Ereignissteuerung für die Enter-Taste
  private class ActionHandler implements ActionListener {
```

```
    @Override
    public void actionPerformed(ActionEvent e) {
      lText.setText(tfEingabe.getText());
    }
  }

  // Ereignissteuerung für den Inhalt des Textfeldes
  private class DocumentHandler implements DocumentListener {
    @Override
    public void changedUpdate(DocumentEvent ev) {
      // Nicht benötigte Methode, deswegen bleibt sie leer
    }
    @Override
    public void insertUpdate(DocumentEvent ev) {
      tfLaenge.setText(
          // Umwandlung von einer Zahl in einen String
          String.valueOf(tfEingabe.getText().length()));
    }
    @Override
    public void removeUpdate(DocumentEvent ev) {
      tfLaenge.setText(
          // Umwandlung von einer Zahl in einen String
          String.valueOf(tfEingabe.getText().length()));
    }
  }

  // Ereignissteuerung für die Cursor-Position
  private class CursorHandler implements CaretListener {
    @Override
    public void caretUpdate(CaretEvent ev) {
      if(ev.getSource() == tfEingabe) {
        tfCursor.setText(
          // Umwandlung von einer Zahl in einen String
          String.valueOf(tfEingabe.getCaretPosition()));
      }
    }
  }
}
```

8 JSlider

Ein **`JSlider`** stellt eine Komponente dar, die es erlaubt einen Knopf in einem vorgegebenen Wertebereich hin und her zu schieben. Er ist die einzige hier vorgestellte Komponente, die **kein `ActionEvent`** auslösen kann und damit auch **keinen `ActionListener`** zur Ereignissteuerung.

8.1 Konstruktoren

```
public JSlider()
public JSlider(int orientation)
public JSlider(int min, int max)
public JSlider(int min, int max, int value)
public JSlider(int orientation, int min, int max, int value)
```

Der parameterlose Konstruktor erzeugt einen horizontalen Schieberegler mit dem Wertebereich 0 bis 100 und einem Anfangswert von 50. Mit dem Parameter **`orientation`** kann die Lage festgelegt werden (horizontal oder vertikal). Hier kann eine der Konstanten **`SwingConstants.HORIZONTAL`** oder **`SwingConstants.VERTICAL`** angegeben werden. Die **`orientation SwingConstants.HORIZONTAL`** wird immer dann verwendet, wenn sie nicht explizit angegeben ist.

Weiters kann der Wertebereich zwischen dem der Slider eingestellt werden kann, mit Hilfe der Parameter **`min`** und **`max`** angegeben werden. Mit **`value`** kann der Anfangswert des Schiebers festgelegt werden.

8.2 Weitere Methoden

Mit den Methoden der Klasse **JSlider** können die Attribute, die beim Konstruktor angegeben werden können auch nachträglich gesetzt bzw. abgefragt werden.

```
public void setMinimum(int minimum)
public int getMinimum()
public void setMaximum(int minimum)
public int getMaximum()
public void setOrientation(int orientation)
public int getOrientation()

public void setValue(int value)
public int getValue()
```

Der aktuelle Wert, der mit **getValue**() abgefragt bzw. mit **setValue**() eingestellt wird, entspricht dabei jener Position, an der der Knopf des Schiebereglers aktuell steht. Weiters gibt es Methoden, mit denen eine Rasterung für den Regler angezeigt werden kann. Dabei müssen zuerst die betreffenden Rasterwerte mit **setMinorTickSpacing** für die kleinere Strichlierung und **setMajorTickSpacing** für die größere Strichlierung eingestellt werden. Das Ergebnis ist dann ähnlich wie bei einem Linieal, wo für jeden Milimeter eine kleinere Strichlierung angegeben ist und für jeden Zentimeter eine größere Strichlierung. Die Methoden dafür sind

```
public void setMinorTickSpacing(int n)
public void setMajorTickSpacing(int n)
```

Angezeigt wird diese Rasterung erst, nachdem bei der Methode **setPaintTicks true** als Parameter übergeben wird

```
public void setPaintTicks(boolean b)
```

Weiters können für die Major-Ticks auch die entsprechenden Zahlenwerte angezeigt werden.

```
public void setPaintLabels(boolean b)
```

Schließlich kann auch noch angegeben werden, dass nur die Tick-Werte eingestellt werden können. Der Knopf springt dann von Rastermarke zur Rastermarke, die dazwischenliegenden Werte werden ignoriert.

```
public void setSnapToTicks(boolean b)
```

8.3 Ereignissteuerung

Ein **JSlider** arbeitet mit **ChangeEvent**s, die immer dann gesendet werden, wenn sich der eingestellte Wert ändert, d.h. wenn der Knopf bewegt wird. Die Methode zum registrieren dieser Ereignissteuerung lautet entsprechend

```
public void addChangeListener(ChangeListener l)
```

Das Ereignis wird wie üblich der Methode, die im Listener vorgegeben ist als Parameter übergeben:

```
public void stateChanged(ChangeEvent e)
```

8.4 Beispiel

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

public class SliderPanel extends JPanel {
  private JSlider sRed;
  private JSlider sGreen;
  private JSlider sBlue;
  private JLabel lAusgabe;

  public SliderPanel() {
    JPanel bars = new JPanel();
    lAusgabe = new JLabel("Farbe", SwingConstants.CENTER);
```

```
    // Hintergrund des Labels auf undurchsichtig schalten
    lAusgabe.setOpaque(true);
    // Konstruktor mit Ausrichtung, Minimum, Maximum und Startwert
    sRed = new JSlider(SwingConstants.VERTICAL, 0, 255, 255);
    sBlue = new JSlider(SwingConstants.VERTICAL, 0, 255, 255);
    sGreen = new JSlider(SwingConstants.VERTICAL, 0, 255, 255);
    this.setLayout(new GridLayout(1,2));
    this.add(bars);
    this.add(lAusgabe);
    bars.setLayout(new GridLayout(1,3));
    bars.add(sRed);
    bars.add(sGreen);
    bars.add(sBlue);
    SliderHandler sh = new SliderHandler();
    sRed.addChangeListener(sh);
    sGreen.addChangeListener(sh);
    sBlue.addChangeListener(sh);
  }

  private class SliderHandler implements ChangeListener {
    @Override
    public void stateChanged(ChangeEvent arg0) {
      int r, g, b;
      r = sRed.getValue();
      g = sGreen.getValue();
      b = sBlue.getValue();
      lAusgabe.setBackground(new Color(r, g, b));
      lAusgabe.setText("Farbe: (" + r + ", " + g +", " + b + ")");
    }
  }
}
```

Wenn man eine Rasterung für die entsprechenden Slider möchte, dann kann man auch noch für jeden Slider mit Hilfe der beschriebenen Methoden im Konstruktor der Panel-Klasse eine Skala setzen, z.B:

```
  public SliderPanel() {
    JPanel bars = new JPanel();
    lAusgabe = new JLabel("Farbe", SwingConstants.CENTER);
    // Hintergrund des Labels auf undurchsichtig schalten
    lAusgabe.setOpaque(true);
    // Konstruktor mit Ausrichtung, Minimum, Maximum und Startwert
    sRed = new JSlider(SwingConstants.VERTICAL, 0, 255, 255);
    // Setzen des großen Rasterabstandes
    sRed.setMajorTickSpacing(50);
    // Setzen des kleinen Rasterabstandes
    sRed.setMinorTickSpacing(10);
    // Zeichnen der Rasterung
    sRed.setPaintTicks(true);
    // Rasterbeschriftung
    sRed.setPaintLabels(true);
    // Der Slider springt von Rasterpunkt zu Rasterpunkt
    sRed.setSnapToTicks(true);
...
```