

Vorlesung 1

Programmierung II

Java Swing & und ein wenig Threads (Nebenläufigkeit)

Dozent: Prof. Dr. Peter Kelb

Raumnummer: Z4030

e-mail: pkelb@hs-bremerhaven.de

Sprechzeiten: nach Vereinbarung

Sourcen: Elli

Ziel der Vorlesung:

- Programmierung graphischer Oberflächen in Java mit Swing
- Threads
- einfache Netzwerkprogrammierung

Parallel in Algorithmen:

- Sortierverfahren
- dynamische Datenstrukturen (Vektoren und Listen)
- Suchverfahren (Bäume und Hashing)

Voraussetzung:

- Vorlesung: Programmierung I (inhaltlich, nicht formal)

Bücher:

- weiter das Java Handbuch aus Programmieren I
- die Onlinedokumentation zur aktuellen JDK Version

Organisation:

- 2 SWS Vorlesung Swing für WInf
- 2 SWS Vorlesung Algorithmen für WInf und Inf
- 1 x 2 SWS Übung

Prüfung:

- Klausur zum Thema Swing und Algorithmen

Graphikprogrammierung: Motivation

Aufgabe: Text ausgeben auf der Konsole

```
System.out.println("Das war einfach.");
```

Aufgabe: Text ausgeben in einem eigenen Fenster

- Wie wird ein Fenster erzeugt?
- Wie wird der Text ausgegeben?
- Wo wird der Text ausgegeben?
- Welche Farbe hat er, welcher Font wird verwendet?
- Was passiert, wenn das Fenster geschlossen und wieder geöffnet wird?

Motivation (Fort.)

Aufgabe: einen roten Kreis in einem eigenen Fenster zeichnen

- wieder: Wie wird ein Fenster erzeugt?
- wieder: Was passiert, wenn das Fenster geschlossen und wieder geöffnet wird?
- Gibt es einen Befehl für Kreise, oder muss jeder Punkt berechnet werden?
- Wie wird die Farbe gesetzt?

Lösungsansätze in Java

Verschiedene Frameworks für Grafikprogrammierung

- Abstract Windowing Toolkit (AWT)
- Swing
- SWT
- Java FX

Fähigkeiten von Swing:

- graphische Primitivoperationen zum Zeichnen und Ausgeben von Text
- Steuerung des Programmablaufs mittels Maus-, Tasten- und Fensterevents
- einfache und komplexe Dialogelemente
- komplexe graphische Operationen für Bitmaps und Sound

Go!

<https://docs.oracle.com/javase/7/docs/api/javax/swing/JFrame.html>

Beispiel 1

```
import javax.swing.*;  
  
class Beispiel1 {  
  
    public static void main(String[] args) {  
        JFrame frame = new JFrame("Mein erstes Fenster");  
        frame.setSize(400,300);  
        frame.setVisible(true);  
    }  
}
```



Beispiel 1 (Fort.)

Erläuterung:

1. Importieren des Graphikpakets: `import javax.swing.*;`
2. Anlegen eines Frames: `new JFrame("Mein erstes Fenster");`
3. Setzen der Größe: `frame.setSize(400,300);`
4. Sichtbar machen: `frame.setVisible(true);`

Go!

Swing: Schließen von Fenstern

- Das Verhalten des Schließen-Buttons kann eingestellt werden
- Mittels des Befehls **setDefaultCloseOperation** kann zwischen den folgenden vier Verhalten unterschieden werden

DO NOTHING ON CLOSE

HIDE ON CLOSE

DISPOSE ON CLOSE

EXIT ON CLOSE

...

```
class Beispiel62_3 {  
    public static void main(String[] args) throws Exception {  
        final JFrame f = new JFrame("Juhu -- mein erstes Swing Fenster");  
        f.setBounds(100,100,400,300);  
        f.setDefaultCloseOperation(JFrame.DO NOTHING ON CLOSE);  
        f.setVisible(true);  
    }  
}
```

Beispiel 1 (Fort.)

JWindow

JFrame

JDialog

- Basisklasse JWindow
 - JFrame und JDialog abgeleitet
 - in der Regel werden nur JFrame und JDialog verwendet
 - JFrame als Hauptklasse für alle möglichen Fenster
 - JDialog wird für kurze Dialoge mit dem Benutzer verwendet
- Bsp:



Graphikkontext

Streams für Zeichen Ein- und Ausgabe

```
System.out.println("Das war einfach");
```

- out ist Objekt der Klasse PrintStream
- bietet Möglichkeiten zum Ausgeben von Zeichen: println
- Abstraktion von konkreten Ausgaberoutinen des BS

Analogon zu PrintStream ist Graphics

Graphikkontext (Fort.)

Graphics ist verwenden zur Graphikausgabe

```
Graphics g;
```

```
...
```

```
g.drawLine(0,0,200,100);
```

- zeichnet im Graphikkontext g eine Line von (0,0) nach (200,100)
- Abstraktion von konkreten Ausgaberoutinen des BS: wie die Line auf dem Bildschirm erscheint, ist Aufgabe von Java

Graphikkontext (Fort.)

```
Graphics g;  
...  
g.drawLine(0,0,200,100);
```

Hier entstehen sofort drei Fragen:

1. Welche Operationen gibt es für einen Graphikkontext?
2. Was sind das für Koordinaten (0,0) und (200,100)?
3. Wie hängt ein Graphikkontext mit einem Fenster zusammen?

Welche Operationen gibt es für einen Graphikkontext?

Einfache graphische Objekte können ausgegeben werden:

- Linien `drawLine`
- Rechtecke `drawRect`
- Rechtecke mit abgerundeten Ecken `drawRoundRect`
- Polygone `drawPolygon`
- Kreise `drawOval`
- Kreisbögen `drawArc`
- Text

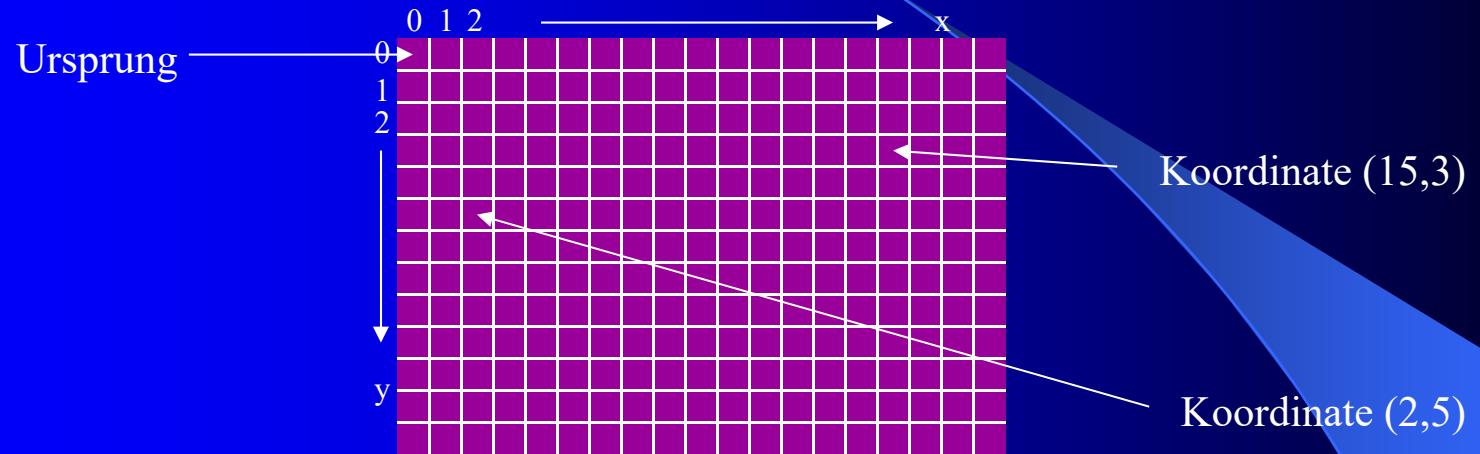
Zeichenbefehle der Graphics Klasse

Linien- oder Füllmodus? (Fort.)

für mehr Informationen
bitte die Dokumentation
der Java Klasse
Graphics anschauen

drawLine	---
drawRect	fillRect
drawRoundRect	fillRoundRect
drawPolygon	fillPolygon
drawPolyline	fillPolyline
drawOval	fillOval
drawArc	fillArc

Was sind das für Koordinaten $(0,0)$ und $(200,100)$?



Graphikkontext hat ein 2-dimensionales Koordinatensystem:

- Ursprungspunkt $(0,0)$ liegt links oben
- x-Werte erstrecken sich nach rechts
- y-Werte nach unten

Go!

Beispiel 2

ACHTUNG:
Negativbeispiel, nicht
zu Hause nachmachen

- Zu jedem Fenster gibt es einen Graphikkontext
- `getGraphics()` liefert diesen Graphikkontext

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

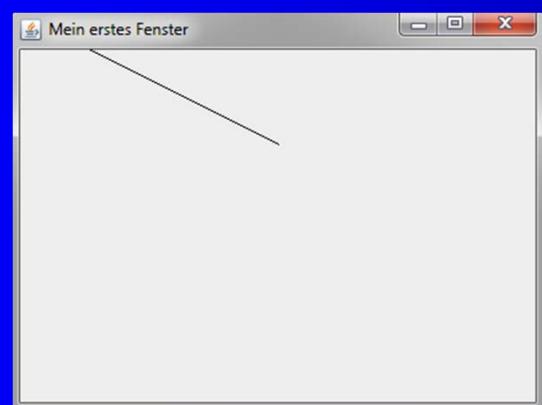
für die Klasse **Graphics**

```
class Beispiel2 {  
  
    public static void main(String[] args) {  
        JFrame frame = new JFrame("Mein erstes Fenster");  
        frame.setSize(400,300);  
        frame.setVisible(true);  
        Graphics g = frame.getGraphics();  
        g.drawLine(0,0,200,100);  
    }  
}
```

Beispiel 2 (Fort.)

Problem:

- Die Linie wird nicht gezeichnet
- wurde sie doch gezeichnet, geht sie wieder verloren



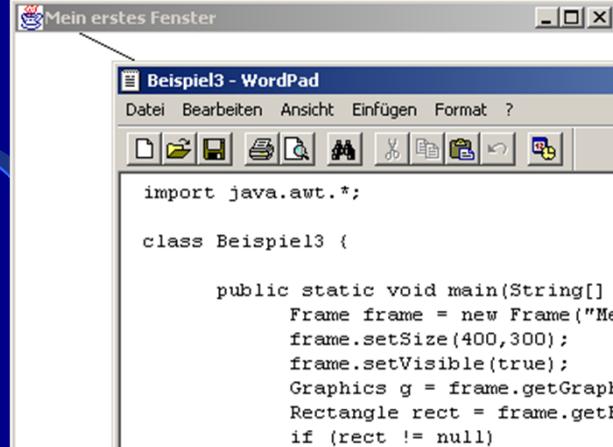
nach
Iconifizierung



Beispiel 2 (Fort.)

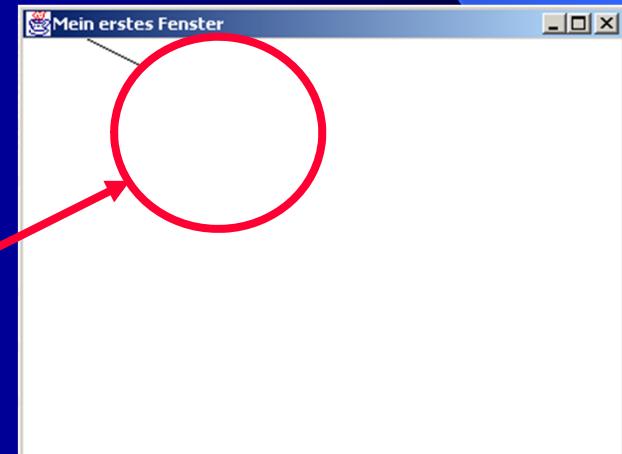
Erklärung:

- ein Graphikkontext merkt sich seinen Inhalt *nicht*
- beim Neuzeichnen des Fensters sind die Informationen verloren



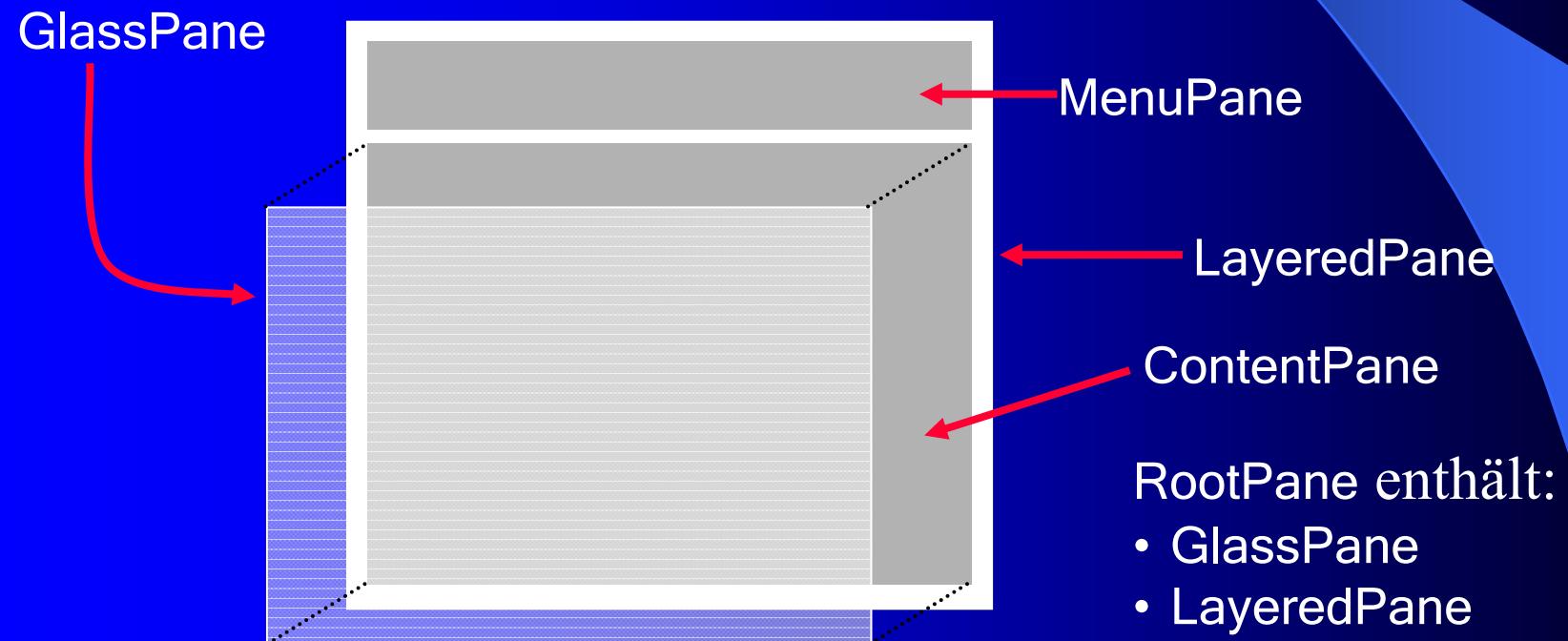
```
Mein erstes Fenster
Beispiel3 - WordPad
Datei Bearbeiten Ansicht Einfügen Format ?
import java.awt.*;
class Beispiel3 {
    public static void main(String[] args) {
        Frame frame = new Frame("Mein erstes Fenster");
        frame.setSize(400,300);
        frame.setVisible(true);
        Graphics g = frame.getGraphics();
        Rectangle rect = frame.getBounds();
        if (rect != null)
            g.drawRect(rect.x, rect.y, rect.width, rect.height);
    }
}
```

update ↓



Ein genauerer Blick auf JFrame

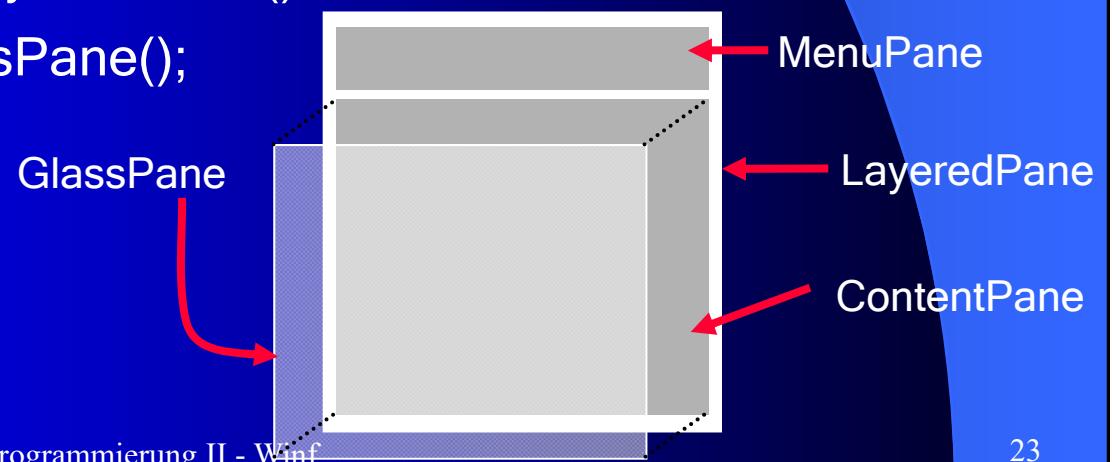
- ein Fenster in Swing besteht aus mehreren Teilen
- i.A. ist die ContentPane interessant, da man sie sieht



Zeichnen in einem JFrame

- MenuPane: enthält das Menu
- ContentPane: der freie Fensterbereich zum Zeichnen
- GlassPane: unsichtbare Schicht über dem Fenster; kann verwendet werden, um Low-Level Events abzufangen, um z.B. Mouseevents für **alle** Components zu bearbeiten

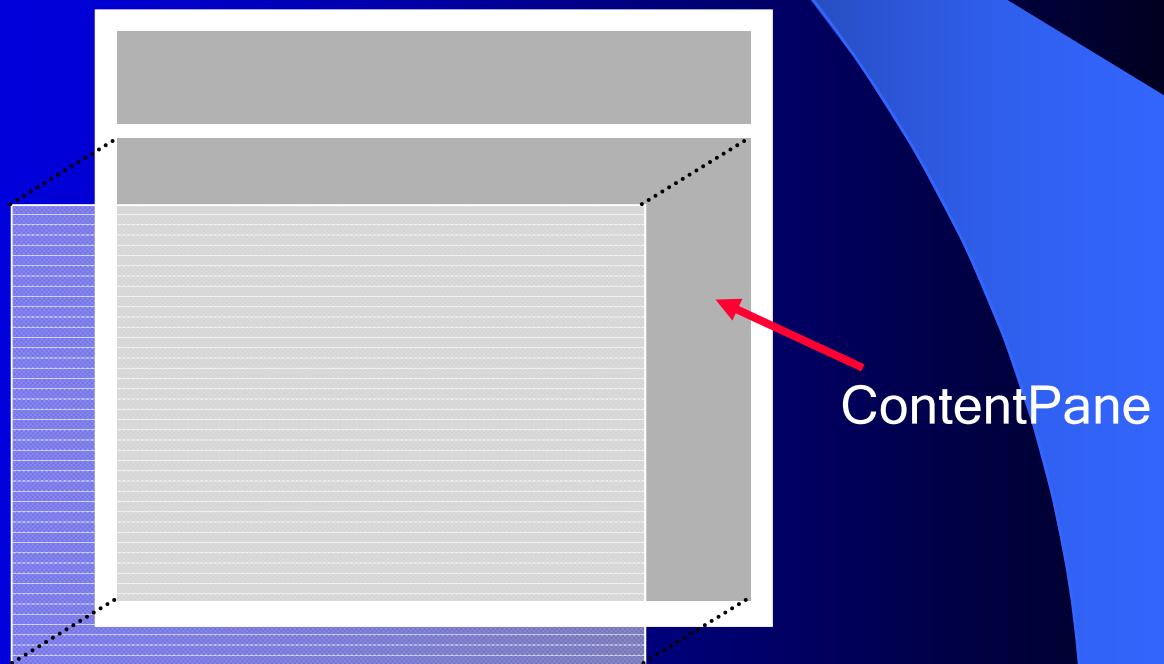
```
class JFrame ... {  
    public JRootPane getRootPane();  
    public Container getContentPane();  
    public JLayeredPane getLayeredPane();  
    public Component getGlassPane();  
    ...  
}
```



Zeichnen in einem JFrame (Fort.)

Allgemeines Vorgehen:

- eigene Klasse von JComponent ableiten
- in diese Klasse wird gezeichnet (paintComponent-Methode überlagern, da die paint-Methode selber von Swing verwendet wird)
- Objekt dieser Klasse zu dem ContentPane mittels der add-Methode hinzufügen



Die paintComponent(Graphics g) Methode

Wann muss ein Fenster (oder ein Teil) neu gezeichnet werden?

- beim ersten Mal
- immer, wenn ein Teil verdeckt war und wieder sichtbar wird
- unter Umständen häufiger (man kann nie wissen ...)

Was passiert, wenn ein Fenster (oder ein Teil) neu gezeichnet werden muss?

- die `paintComponent(Graphics g)` Methode der Fensterelemente wird aufgerufen

Go!

<https://docs.oracle.com/javase/7/docs/api/javax/swing/JComponent.html>

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

```
class MyComponent extends JComponent {
```

```
    @Override
```

```
    public void paintComponent(Graphics g) {  
        g.drawLine(0,0,200,100);
```

```
}
```

```
}
```

```
class Beispiel3 extends JFrame {
```

```
    public static void main(String[] args) {
```

```
        JFrame j = new JFrame("Mein erstes Fenster");
```

```
        j.add(new MyComponent());
```

```
        j.setSize(400,300);
```

```
        j.setVisible(true);
```

```
}
```

```
}
```

Beispiel 3

Der Graphikkontext
ist auch schon dabei

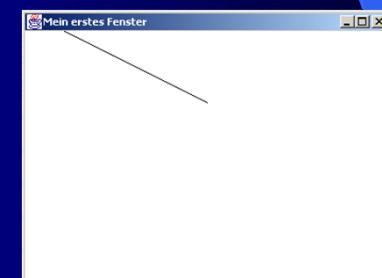
Beispiel 3 (Fort.)

Programmablauf:

```
JFrame j = new JFrame("Mein ...");  
j.add(new MyComponent());  
j.setSize(400,300);  
j.setVisible(true);
```

```
class MyComponent extends JComponent {  
    public void paintComponent(Graphics g) {  
        g.drawLine(0,0,200,100);  
    }  
}  
class Beispiel3 extends JFrame {  
    public static void main(String[] args) {  
        JFrame j = new JFrame("Mein ...");  
        j.add(new MyComponent());  
        j.setSize(400,300);  
        j.setVisible(true);  
    }  
}
```

paintComponent(Graphics g) wird aufgerufen
g.drawLine(0,0,200,100);



Beispiel 3 (Fort.)

Programmablauf (Fort.):

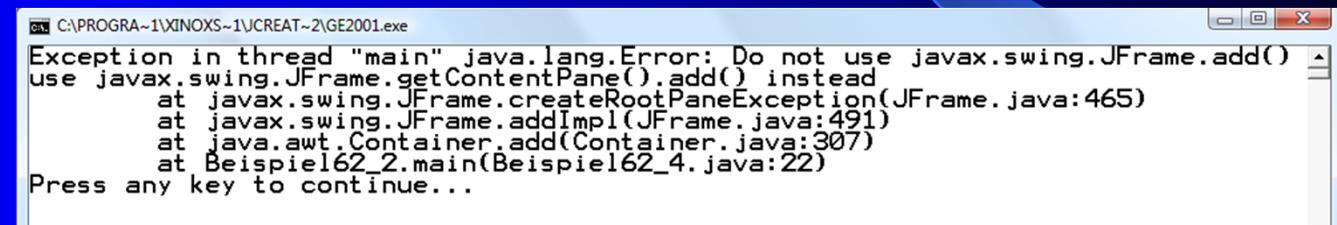
paintComponent(Graphics g) wird aufgerufen
g.drawLine(0,0,200,100);



Go!

getContentPane(): Früher war alles anders

- in alten Versionen von Java (< 1.5) musste man unter Swing der ContentPane die Elemente hinzufügen
- ein Hinzufügen zum Swing Fenster erzeugte ein Laufzeitfehler



...

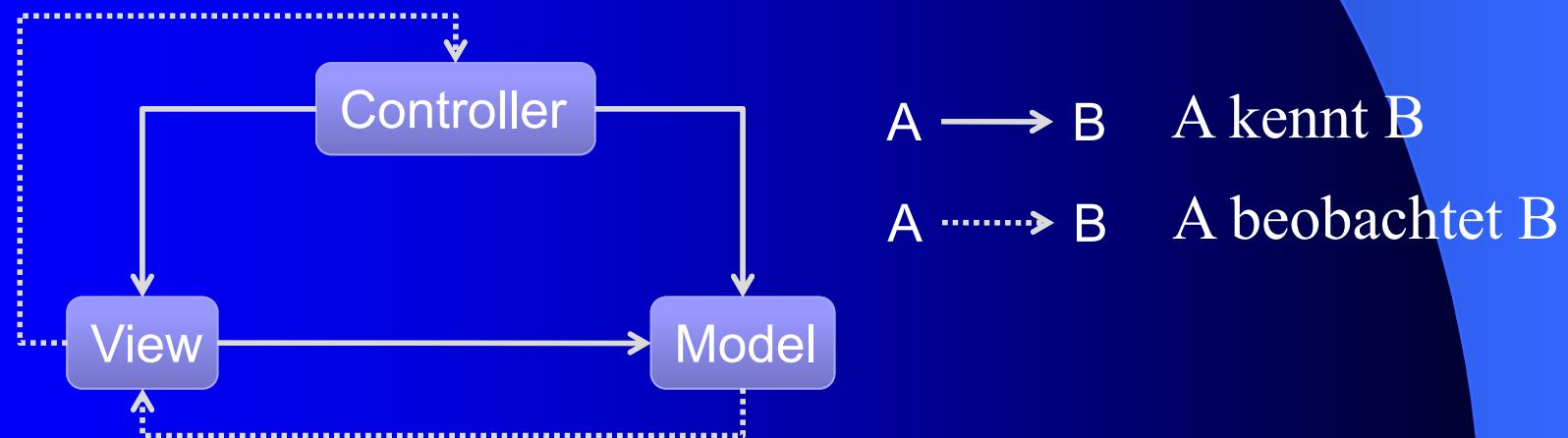
```
class Beispiel62_4 {
    public static void main(String[] args) throws Exception {
        final JFrame f = new JFrame("Juhu -- mein erstes Swing Fenster");
        f.setBounds(100,100,400,300);
        f.getContentPane().add(new MyContent());
        f.setVisible(true);
    }
}
```

WICHTIG: in alten Java Versionen die ContentPane verwenden

Vorlesung 2

Aufbau von Programmen mit Grafikanteil

- bisher: Programme waren recht kurz und übersichtlich
- zukünftig: Programme mit Grafikanteil werden deutlich umfangreicher und komplizierter
- daher: Programme brauchen eine Architektur, die die verschiedenen Anteile des Programms aufteilt, strukturiert und koordiniert
- Lösung: **Model – View – Controller (MVC) Konzept**



Aufbau von Programmen mit Grafikanteil (Forts.)

- Beispiel: ein Programm, dass
 - ein Fenster öffnet
 - immer wieder zufällig zwei Koordinaten produziert
 - im Fenster eine Line zwischen diesen beiden Koordinaten zieht

Beispiel 4

```
class Model {  
    int m_maxX;  
    int m_maxY;  
    int m_X1,m_X2,m_Y1,m_Y2;  
  
    Model(int maxX,int maxY) {  
        m_maxX = maxX;  
        m_maxY = maxY;  
        genNewKoor();  
    }  
  
    void genNewKoor() {  
        m_X1 = (int)(Math.random() * m_maxX);  
        m_X2 = (int)(Math.random() * m_maxX);  
        m_Y1 = (int)(Math.random() * m_maxY);  
        m_Y2 = (int)(Math.random() * m_maxY);  
    }  
}
```

Die Daten des Modells

Das Modell kann nur
neue Daten erzeugen

Beispiel 4 (Forts.)

```
class View extends JComponent {
```

```
    private Model m_Mod;
```

Die Ansicht kennt
das Modell

```
    View(Model mod) {  
        m_Mod = mod;  
    }
```

Zum Zeichnen wird
auf das Modell **nur
lesend** zugegriffen

```
    @Override  
    public void paintComponent(Graphics g) {  
        g.drawLine(m_Mod.m_X1,m_Mod.m_Y1,  
                  m_Mod.m_X2,m_Mod.m_Y2);  
    }
```

```
}
```

```
class Controller {
```

```
    private Model m_Mod;  
    private View m_View;
```

```
    Controller() {
```

```
        m_Mod = new Model(300,200);  
        m_View = new View(m_Mod);  
        JFrame j = new JFrame();  
        j.add(m_View);  
        j.setSize(m_Mod.m_maxX,m_Mod.m_maxY);  
        j.setVisible(true);
```

```
}
```

```
void simulate() {
```

```
    while (true) {
```

```
        try {Thread.sleep(500);}  
        catch (InterruptedException e) {}  
        m_Mod.genNewKoor();  
        m_View.repaint();
```

```
}
```

```
}
```

Prof. Dr. Peter Kelb

Beispiel 4 (Forts.)

Der Controller kennt
nicht nur das Modell
und die Ansicht ...

... sondern erzeugt
sie auch (muss aber
nicht sein)

Der Controller verändert
nicht die Daten, bewirkt
aber die Veränderungen

Go!

Beispiel 4 (Forts.)

```
class Beispiel4 {  
  
    public static void main(String[] args) throws Exception {  
        Controller c = new Controller();  
        c.simulate();  
    }  
}
```

Das Hauptprogramm
erzeugt nur noch den
Controller und startet
ihn

Aufbau von Programmen mit Grafikanteil (Forts.)

- bei kleineren Grafikprogrammen werden Controller und View auch zusammengefasst



- gleiches Beispiel
 - das Model ändert sich nicht
 - der Controlleranteil geht in die View ein

Go!

Beispiel 5 (Forts.)

```
class View extends JComponent {  
    Model m_Mod = new Model(300,200);  
    public void paintComponent(Graphics g) {  
        g.drawLine(m_Mod.m_X1,m_Mod.m_Y1,m_Mod.m_X2,m_Mod.m_Y2);  
    }  
    void simulate() {  
        while (true) {  
            try {  
                Thread.sleep(500);  
            } catch (InterruptedException e) {}  
            m_Mod.genNewKoor();  
            repaint();  
        }  
    }  
}
```

View/Controller
erzeugt das Model ...

... und enthält auch
den Kontrollanteil

```
class Beispiel5 {  
    public static void main(String[] args) throws Exception {  
        JFrame j = new JFrame();  
        View v = new View();  
        j.add(v);  
        j.setSize(v.m_Mod.m_maxX,v.m_Mod.m_maxY);  
        j.setVisible(true);  
        v.simulate();  
    }  
}
```

Text schreiben in Fenstern

Dazu gibt es u.a. die Methode:

```
class Graphics ...
```

```
...
```

```
public void drawString(String str,int x,int y);
```

schreibt str an die
Position (x,y)

- Schreibt den String str im Fenster „an der Position (x,y)“
- verwendet den Standardfont
- verwendet die Standardfarbe

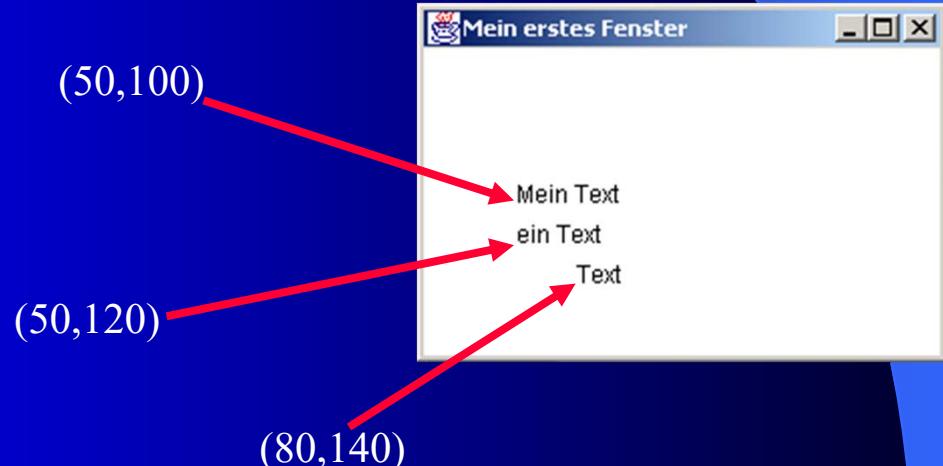
Go!

Beispiel 6

```
class Content extends JComponent {  
    @Override  
    public void paintComponent(Graphics g) {  
        String strText = new String("Mein\t\n Text");  
        g.drawString(strText,50,100);  
        g.drawChars(strText.toCharArray(),1,strText.length()-1,50,120);  
        g.drawBytes(strText.getBytes(),5,strText.length()-5,80,140);  
    }  
}
```

```
public class Beispiel6 {  
  
    public static void main(String[] args) {  
        JFrame j = new JFrame();  
        j.add(new Content());  
        j.setSize(400,300);  
        j.setVisible(true);  
        new Beispiel7();  
    }  
}
```

"	M	e	i	n		T	e	x	t	"
	0	1	2	3	4	5	6	7	8	



Fonts

- Der Standardfont wird gesetzt durch `setFont`

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

- Der Standardfont kann abgefragt werden durch `getFont`

```
public Font getFont();
```

Fonts (Fort.)

Neue Fonts werden mittels des Konstruktors

```
Font(String name,int style,int size);
```

erzeugt.

- `name` ist der Name des Fonts,
- `style` ist der Stil des Fonts (`Font.PLAIN`, `Font.BOLD`, `Font.ITALIC`),
- `size` gibt die Größe in Punkte an.

VORSICHT: nicht `style` und `size` verwechseln

Fonts (Fort.)

Welche Namen sind erlaubt?

- SansSerif (früher Helvetica)
- Serif (früher TimesRoman)
- Monospaced (früher Courier)
- Diese Fonts sollten auf allen JAVA Systemen vorhanden sein.
- Darüber hinaus gibt es noch weitere Fonts, die aber nicht auf jedem System verfügbar sind.
- Daher kann Ihr Programm u.U. auf einem anderen System die Schriften nicht richtig darstellen, wenn Sie andere Fonts verwenden.

Fonts (Fort.)

Welche Stile sind erlaubt?

- `Font.PLAIN` (Standardfont)
- `Font.BOLD` (Fett)
- `Font.ITALIC` (Kursiv)

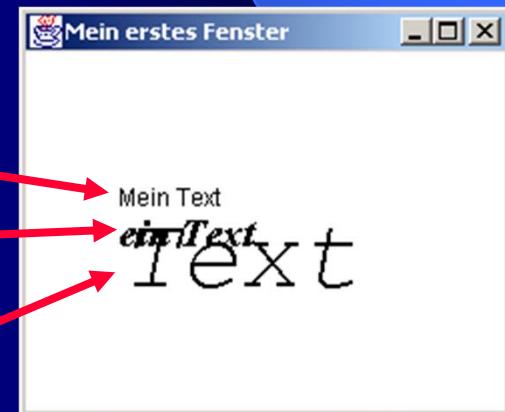
`Font.BOLD` | `Font.ITALIC` erzeugt kursiven Fettdruck

Go!

Beispiel 7

```
class Content extends JComponent {  
    public void paintComponent(Graphics g) {  
        String strText = new String("Mein Text");  
        g.drawString(strText,50,100);  
        g.setFont(new Font("Serif",Font.BOLD|Font.ITALIC,20));  
        g.drawChars(strText.toCharArray(),1,strText.length()-1,50,120);  
        g.setFont(new Font("Monospaced",Font.ITALIC,50));  
        g.drawBytes(strText.getBytes(),5,strText.length()-5,50,140);  
    }  
}  
  
public class Beispiel7 {  
  
    public static void main(String[] args) {  
        JFrame j = new JFrame();  
        j.add(new Content());  
        j.setSize(400,300);  
        j.setVisible(true);  
        new Beispiel7();  
    }  
}
```

Standardfont
"Serif" Font in kursiven
Fettdruck in 20 Punkten
"Monospaced" Font,
kursiv, in 50 Punkten



Verfügbare Fonts

Die Klasse `GraphicsEnvironment` hilft einem, alle verfügbaren Fonts zu ermitteln.

```
class GraphicsEnvironment
```

```
...
```

```
public static GraphicsEnvironment getLocalGraphicsEnvironment();  
public String [] getAvailableFontFamilyNames();
```

```
...
```

Go!

Beispiel 8

```
public void paintComponent(Graphics g) {  
    GraphicsEnvironment ge =  
        GraphicsEnvironment.getLocalGraphicsEnvironment();  
    String [] arFonts = ge.getAvailableFontFamilyNames();  
    for(int i = 0;i < arFonts.length;++i) {  
        g.setFont(new Font(arFonts[i],Font.PLAIN,12));  
        g.drawString(arFonts[i],  
                    10,  
                    getInsets().top + (i+1) * 12);  
    }  
}
```

nicht alle Fonts sind druckbar



Farben

Farben werden in JAVA in zwei Modellen dargestellt:

- RGB Farbmodell : additive Farbmischung von Rot, Grün, Blau
- HSB Farbmodell: drei Parameter: Farbton, Intensität, Helligkeit

JAVA unterstützt die Konvertierung zwischen diesen Modellen

im Folgenden: RGB Farbmodell

Farben (Fort.)

Farben werden in JAVA in der Klasse Color repräsentiert.

```
public Color(int red,int green,int blue);
```

Erzeugt eine Farbe mit den angegebenen Farbanteilen. red, green und blue müssen zwischen 0 und 255 liegen. 0 bedeutet kein Anteil, 255 bedeutet maximaler Anteil.

Color(255,255,255)	Weiß
Color(255,0,0)	Rot
Color(255,255,0)	Gelb

Farben (Fort.)

```
public Color(float red,float green,float blue);
```

Erzeugt eine Farbe mit den angegebenen Farbanteilen. red, green und blue müssen zwischen 0.0 und 1.0 liegen. 0.0 bedeutet kein Anteil, 1.0 bedeutet maximaler Anteil.

Color(1.0,1.0,1.0)	Weiß
Color(1.0,0.0,0.0)	Rot
Color(1.0,1.0,0.0)	Gelb

Farben (Fort.)

Vordefinierte Farben:

```
class Color
```

...

```
public static Color white;  
public static Color lightGray;  
public static Color gray;  
public static Color darkGray;  
public static Color black;  
public static Color red;  
public static Color blue;  
public static Color green;  
public static Color yellow;  
public static Color magenta;  
public static Color cyan;  
public static Color orange;  
public static Color pink;
```

...

Farben (Fort.)

Informationen über Farben:

```
class Color
```

```
...
public int getRed();
public int getGreen();
public int getBlue();
```

```
...
```

liefert den Rot-, Grün- bzw. Blauanteil einer Farbe.

Farben (Fort.)

Verwenden von Farben

```
class Graphics  
...  
public void setColor(Color c);  
public Color getColor();  
...
```

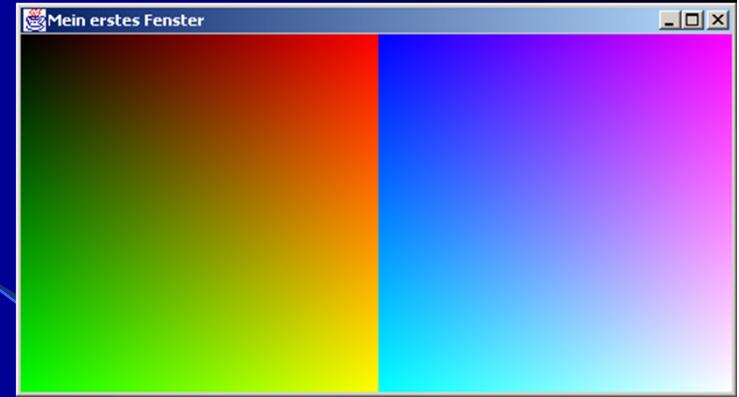
- `g.setColor(Color.red)` setzt im Graphikkontext `g` die aktuelle Zeichenfarbe auf Rot.
- `g.getColor()` liefert die aktuelle Zeichenfarbe des Graphikkontextes `g`

Go!

Beispiel 9

```
class Content extends JComponent {  
    public void paintComponent(Graphics g) {  
        for(int i = 0;i < 256;++i) {  
            for(int j = 0;j < 256;++j) {  
                g.setColor(new Color(i,j,0));  
                g.drawLine(i,j,i,j);  
                g.setColor(new Color(i,j,255));  
                g.drawLine(i+256,j,i+256,j);  
            }  
        }  
    }  
}
```

```
class Beispiel9 {  
  
    public static void main(String[] args) {  
        JFrame j = new JFrame();  
        j.add(new Content());  
        j.setSize(550,300);  
        j.setVisible(true);  
    }  
}
```



Farben (Fort.)

Systemfarben

```
class SystemColor
```

```
...  
public static Color desktop;  
public static Color control;  
...
```

- `desktop` ist die Hintergrundfarbe des Desktops.
- `control` ist die Hintergrundfarbe für Dialogelemente.

Diese Farben werden zur Laufzeit ermittelt.

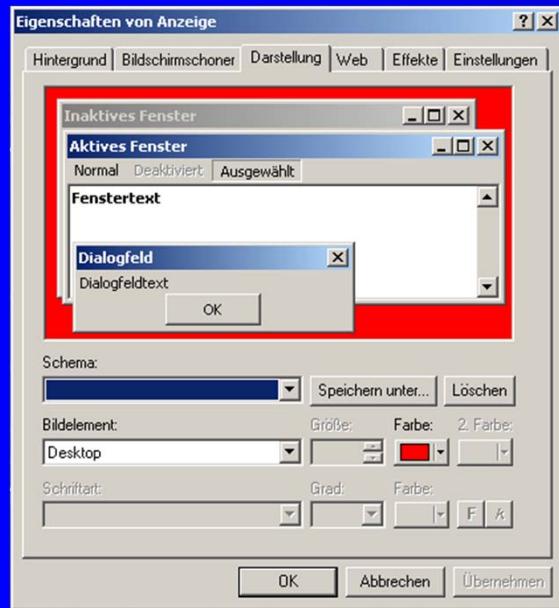
Go!

Beispiel 10

```
public void paintComponent(Graphics g) {  
    g.setColor(SystemColor.desktop);  
    g.fillRect(0,0,100,100);  
}
```



Beispiel 10 (Fort.)



```
public void paintComponent(Graphics g) {  
    g.setColor(SystemColor.desktop);  
    g.fillRect(0,0,100,100);  
}
```



Vorder- und Hintergrundfarbe

Zu jeder Komponente kann die Standardfarbe für der Vorder- und Hintergrund gesetzt werden.

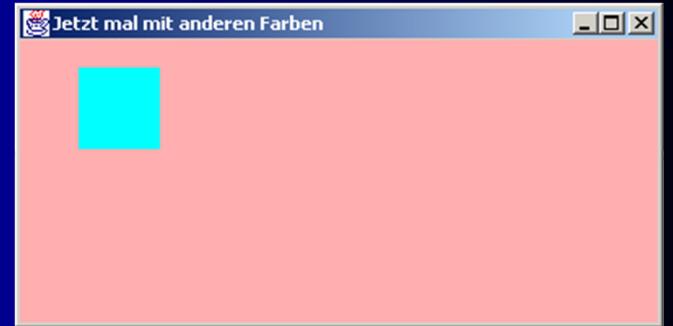
```
class JComponent {  
    ...  
    public void setBackground(Color c);  
    public void setForeground(Color c);  
    ...  
}
```

Go!

Beispiel 16

```
class Content extends JComponent {  
    @Override  
    public void paintComponent(Graphics g) {  
        g.fillRect(40,40,50,50);  
    }  
}
```

```
class Beispiel16 {  
  
    public static void main(String[] args) {  
        JFrame j = new JFrame("Jetzt mal mit anderen Farben");  
        j.add(new Content());  
        j.setBounds(200,300,400,200);  
        j.getContentPane().setForeground(Color.cyan);  
        j.getContentPane().setBackground(Color.pink);  
        j.setVisible(true);  
    }  
}
```



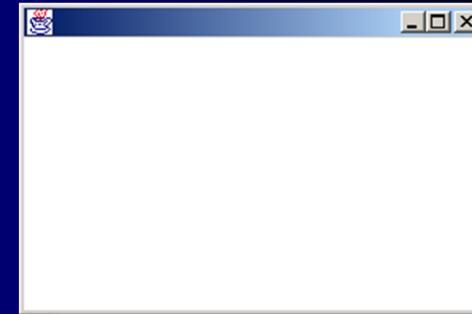
Wichtig: nicht das Fenster, sondern der Inhalt muss geändert werden

Vorlesung 3

Aufrufen von Fenstern

```
public JFrame();
```

erzeugt ein einfaches Fenster mit Rahmen und leerer Titelleiste



```
public JFrame(String strTitle);
```

erzeugt ein einfaches Fenster mit Rahmen und Titelleiste, die den String strTitle enthält



Go!

Aufrufen von Fenstern (Fort.)

```
public JWindow(JFrame parent);
```

erzeugt ein einfaches Fenster ohne Rahmen und ohne Titelleiste

```
public static void main(String[] args) {  
    JFrame top = new JFrame();  
    JFrame top2 = new JFrame("juhu");  
    top.setSize(800,600);  
    top.setVisible(true);  
    top2.setSize(800,600);  
    top2.setVisible(true);  
    JWindow w = new JWindow(top);  
    w.setSize(400,300);  
    w.setVisible(true);  
}
```



Aufrufen von Fenstern (Fort.)

```
...
JFrame top = new JFrame();
JWindow w = new JWindow(top);
...
...
```

Frage: Warum braucht ein einfaches Fenster (JWindow)
einen Vater-JFrame?

Aufrufen von Fenstern (Fort.)

Erzeugen eines Fensters, z.B.:

```
JFrame f = new JFrame();
```

sichtbar machen, d.h. auf dem Bildschirm anzeigen:

```
f.setVisible(true);
```

Schließen von Fenstern

Fenster werden durch den Befehl

```
setVisible(false);
```

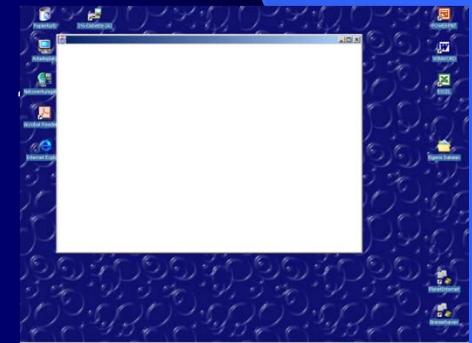
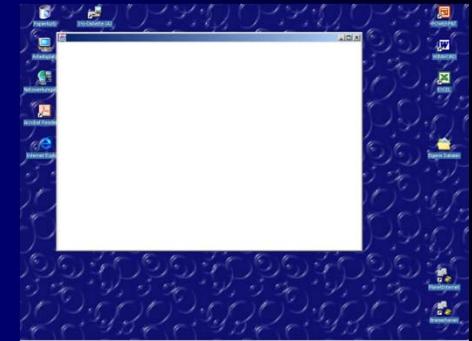
geschlossen, d.h. unsichtbar gemacht.

WICHTIG: Das Fenster existiert noch im Speicher und kann jederzeit wieder angezeigt werden.

Go!

Beispiel 12

```
import javax.swing.*;  
  
class Beispiel12 {  
    public static void main(String[] args) {  
        JFrame f = new JFrame();  
        f.setSize(700,500);  
        f.setVisible(true);  
        try {  
            Thread.sleep(700);  
        } catch (java.lang.InterruptedException i)  
        f.setVisible(false);  
        try {  
            Thread.sleep(700);  
        } catch (java.lang.InterruptedException i)  
        f.setVisible(true);  
    }  
}
```



Löschen von Fenstern

Fenster werden durch den Befehl

```
dispose();
```

gelöscht, d.h. unsichtbar gemacht und aus dem Speicher
gelöscht.

Beispiel 13

```
class Beispiel13 extends JComponent {  
    private int m_iCurrentNum = 0;  
    public void countDown(JFrame f) {  
        for(m_iCurrentNum = 10;m_iCurrentNum > 0;--m_iCurrentNum) {  
            try {Thread.sleep(1000);}  
            catch (InterruptedException e) {}  
            f.repaint(); ← Zeichnet das  
Fenster neu  
        }  
        f.dispose();  
    }  
    public static void main(String[] args) {  
        JFrame f = new JFrame();  
        Beispiel13 b = new Beispiel13();  
        f.add(b);  
        f.setSize(400,300);  
        f.setVisible(true);  
        b.countDown(f);  
        System.out.println("Schluss");  
    }  
}
```

Go!

Beispiel 13 (Fort.)

...

```
public void paintComponent(Graphics g) {  
    g.setFont(new Font("Serif",Font.BOLD + Font.ITALIC,80));  
    FontMetrics actualMetric = g.getFontMetrics();  
    String strText = Integer.toString(m_iCurrentNum);  
    final int fiTextWidth = actualMetric.stringWidth(strText);  
    final int fiTextHeight = actualMetric.getHeight();  
    final int fiTextBottom = actualMetric.getDescent() + actualMetric.getLeading();  
    final int fiVisibleWidth = getSize().width-getInsets().left-getInsets().right;  
    final int fiVisibleHeight = getSize().height-getInsets().top-getInsets().bottom;  
    g.clearRect(0,0,getBounds().width,getBounds().height);  
    g.drawString(strText,  
                getInsets().left + (fiVisibleWidth - fiTextWidth) / 2,  
                getInsets().top + (fiVisibleHeight + fiTextHeight) / 2 - fiTextBottom);  
    System.out.println(m_iCurrentNum);  
}
```

...

Drückt String strText
mittig im Fenster



Bildschirm- und Fenstergröße und -position

class Toolkit

...
public Dimension screenSize(); Bildschirmgröße

...

class JComponent

...
public void setSize(int width,int height); Fenstergröße
public void setSize(Dimension d);

public void setLocation(int x,int y);
public void setLocation(Point p); Fensterposition

public void setBounds(int x,int y,int width,int height);
public void setBounds(Rectangle r); Fenstergröße
und -position

...

Go!

Beispiel 14

```
class Content extends JComponent {  
    @Override  
    public void paintComponent(Graphics g) {  
        g.setColor(Color.red);  
        // g.setColor(Color.white);  
        g.fill3DRect(0,0,300,200,true);  
    }  
}  
  
class Beispiel14 extends JWindow {  
  
    public static void main(String[] args) {  
        JWindow w = new JWindow(new JFrame());  
        w.add(new Content());  
        Dimension dim = w.getToolkit().getScreenSize();  
        // w.setLocation(dim.width/2-150,dim.height/2-100);  
        // w.setSize(300,200);  
        w.setBounds(dim.width/2-150,dim.height/2-100,300,200);  
        w.setVisible(true);  
    }  
}
```



Anzeigezustand von Fenstern

- Fenster können "normal" oder "als Symbol" angezeigt werden.
- Setzen des Anzeigezustands:

```
public synchronized void setExtendedState(int state);
```

- mögliche Zustände
 - Frame.ICONIFIED
 - Frame.NORMAL

Die Titelleiste

Der Inhalt der Titelleiste kann im Konstruktor von Frame gesetzt werden.

- `public JFrame(String strTitle);`

Während des Programms kann der Titel umgesetzt werden,

- `public void setTitle(String strTitle);`

und abgefragt werden.

- `public String getTitle();`

Das Icon-Image

Werden Fenster "als Symbol" dargestellt, repräsentiert ein Icon das Fenster.

Standard in Java:



Das Icon wird gesetzt durch:

- `public void setIconImage(Image image);`

Images

Images können über das Toolkit geladen werden.

```
class Toolkit  
...  
public abstract Image getImage(String strFileName);  
public abstract Image getImage(URL url);  
...
```

Images werden im Graphikkontext gedruckt durch:

```
class Graphics  
...  
public void drawImage(Image image,...);  
...
```

Der Mauscursor

Die Form des Mauscursors kann für jede Component eigenständig definiert werden.

```
class JComponent  
...  
public void setCursor(Cursor cursor);  
...
```

Der Mauscursor (Fort.)

Die Klasse für die unterschiedlichen Mauscursor.

```
class Cursor  
...  
public Cursor(int type);  
public static Cursor getPredefinedCursor(int type);  
final public int CROSSHAIR_CURSOR;  
final public int DEFAULT_CURSOR;  
final public int E_RESIZE_CURSOR;  
...
```

WICHTIG: nicht neue Instanzen von Cursor erzeugen,
sondern mittels getPredefinedCursor(int type) einen
anderen Cursor wählen.

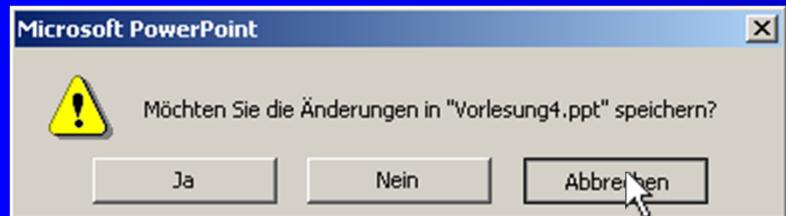
Go!

Beispiel 15

```
import javax.swing.*;  
import java.awt.*;  
  
class Beispiel15 {  
  
    public static void main(String[] args) throws Exception {  
        JFrame f = new JFrame();  
        f.setTitle("Es geht auch anders");  
        Dimension dim = f.getToolkit().getScreenSize();  
        f.setBounds(300,200,dim.width/2-150,dim.height/2-100);  
        Image img = f.getToolkit().getImage("start.gif");  
        f.setIconImage(img);  
        f.setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));  
        f.setVisible(true);  
        f.setExtendedState(Frame.ICONIFIED);  
    }  
}
```

Vorlesung 4

Event-Handling



Situation: Ein Button eines Dialogs wird gedrückt.

Frage: Was passiert in dieser Situation?

1. An den Button wird eine Nachricht geschickt, dass er gedrückt wird.
2. Der Button schaut nach, ob er auf das Drücken reagieren soll.
3. Wenn er reagieren soll, wird die Nachricht an den **"Sachbearbeiter"** weitergeschickt.

Event-Handling (Fort.)

Ereignis: Benutzer drückt Button



Window-Manager

Nachricht
"Du wurdest gedrückt"

Objekt der Klasse JButton

schaut nach, ob auf diese Nachricht reagiert werden soll

Ja

Objekt der Klasse "Reagierer" verarbeitet Event

Nein

Event-Handling (Fort.)

Beim Event-Handling sind genau die folgenden Objekte beteiligt:

1. das Objekt, dass das Event auslöst,
die **Ereignisquelle (Event Source)**
2. das Objekt, dass das Event
verarbeitet, der **Ereignisempfänger
(Event Listener)**

Objekt der Klasse
JButton

schaut nach, ob auf diese Nach-
richt reagiert werden soll

Ja

Objekt der Klasse
"Reagierer"
verarbeitet Event

Nein

Event-Handling (Fort.)

1. Frage: Wie kann der Button wissen, auf welche Ereignisse er reagieren soll?
2. Frage: Woher weiß der Button, welches Objekt das Ereignis verarbeitet?

Antwort:

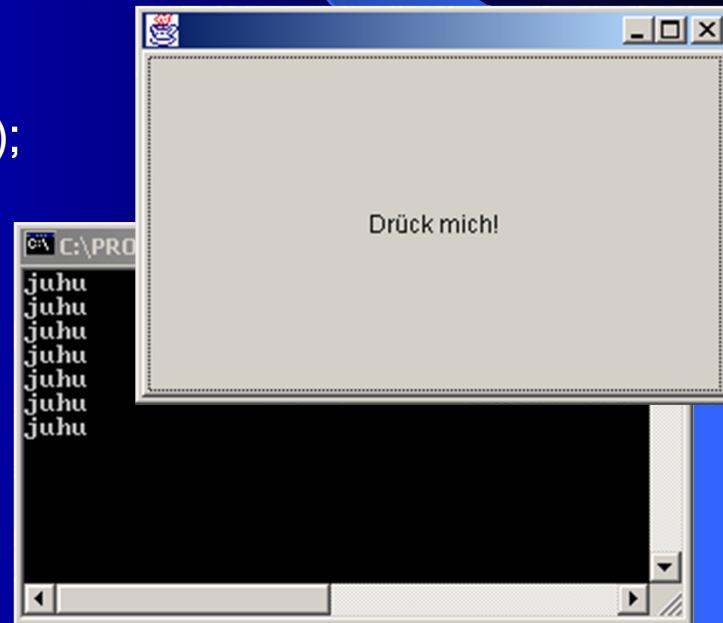
Das Objekt zum Verarbeiten des Events wird bei dem Button zu genau diesem Event registriert.

Go!

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class Reagierer implements ActionListener {
    int j = 0;
    public void actionPerformed(ActionEvent e) {
        System.out.println(++j + " juhu");
    }
}
class MyButton extends JButton {
    public MyButton() {
        super("Drück mich!");
        addActionListener(new Reagierer());
    }
}
class Beispiel22 {
    public static void main(String [] args) {
        JFrame f = new JFrame();
        f.setBounds(100,100,300,200);
        f.add(new MyButton());
        f.setVisible(true);
    }
}
```

[https://docs.oracle.com/javase/7/docs/api/java.awt.event/ActionListener.html](https://docs.oracle.com/javase/7/docs/api/java.awt/event/ActionListener.html)

Beispiel 22



```

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

class Reagierer implements ActionListener {
    int j = 0;
    public void actionPerformed(ActionEvent e) {
        System.out.println(++j + " juhu");
    }
}

class MyButton extends JButton {
    public MyButton() {
        super("Drück mich!");
        addActionListener(new Reagierer());
    }
}

class Beispiel22 {
    public static void main(String [] args) {
        JFrame f = new JFrame();
        f.setBounds(100,100,300,200);
        f.add(new MyButton());
        f.setVisible(true);
    }
}

```

Beispiel 22 (Fort.)

Reagierer implementiert das Interface ActionListener

MyButton erweitert JButton;
Buttons können auf Gedrücktwerden reagieren

registriert Reagierer bei MyButton;
Reagierer muss Interface ActionListener implementieren

Fügt dem Hauptfenster ein MyButton hinzu; der Rest ist wie früher gehabt

Beispiel 22 (Fort.)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class Reagierer implements ActionListener {
    int j = 0;
    public void actionPerformed(ActionEvent e) {
        System.out.println(++j + " juhu");
    }
}
class MyButton extends JButton {
    public MyButton() {
        super("Drück mich!");
        addActionListener(new Reagierer());
    }
}
class Beispiel22 {
    public static void main(String [] args) {
        JFrame f = new JFrame();
        f.setBounds(100,100,300,200);
        f.add(new MyButton());
        f.setVisible(true);
    }
}
```

Klasse Reagierer ist
eine klassische
Anwendung für eine
anonyme Klasse

Go!

```
class MyButton extends JButton {  
  
    public MyButton() {  
        super("Drück mich!");  
        addActionListener(new ActionListener() {  
            int j = 0;  
            public void actionPerformed(ActionEvent e) {  
                System.out.println(++j + " juhu");  
            }  
        });  
    }  
}
```

```
class Beispiel22_1 {  
  
    public static void main(String [] args) {  
        JFrame f = new JFrame();  
        f.setBounds(100,100,300,200);  
        f.add(new MyButton());  
        f.setVisible(true);  
    }  
}
```

Beispiel 22 (Fort.)

MyButton mit anonymer
Klasse, die auf das
Drücken reagiert

Go!

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

```
class Beispiel22_3 {
```

```
    public static void main(String [] args) {  
        JFrame j = new JFrame();  
        j.add(new JButton("Drück mich") {  
            { addActionListener(new ActionListener() {  
                public void actionPerformed(ActionEvent e) {  
                    System.out.println("juhu");  
                }  
            });  
        }  
    );  
    j.setBounds(100,100,300,200);  
    j.setVisible(true);  
}
```

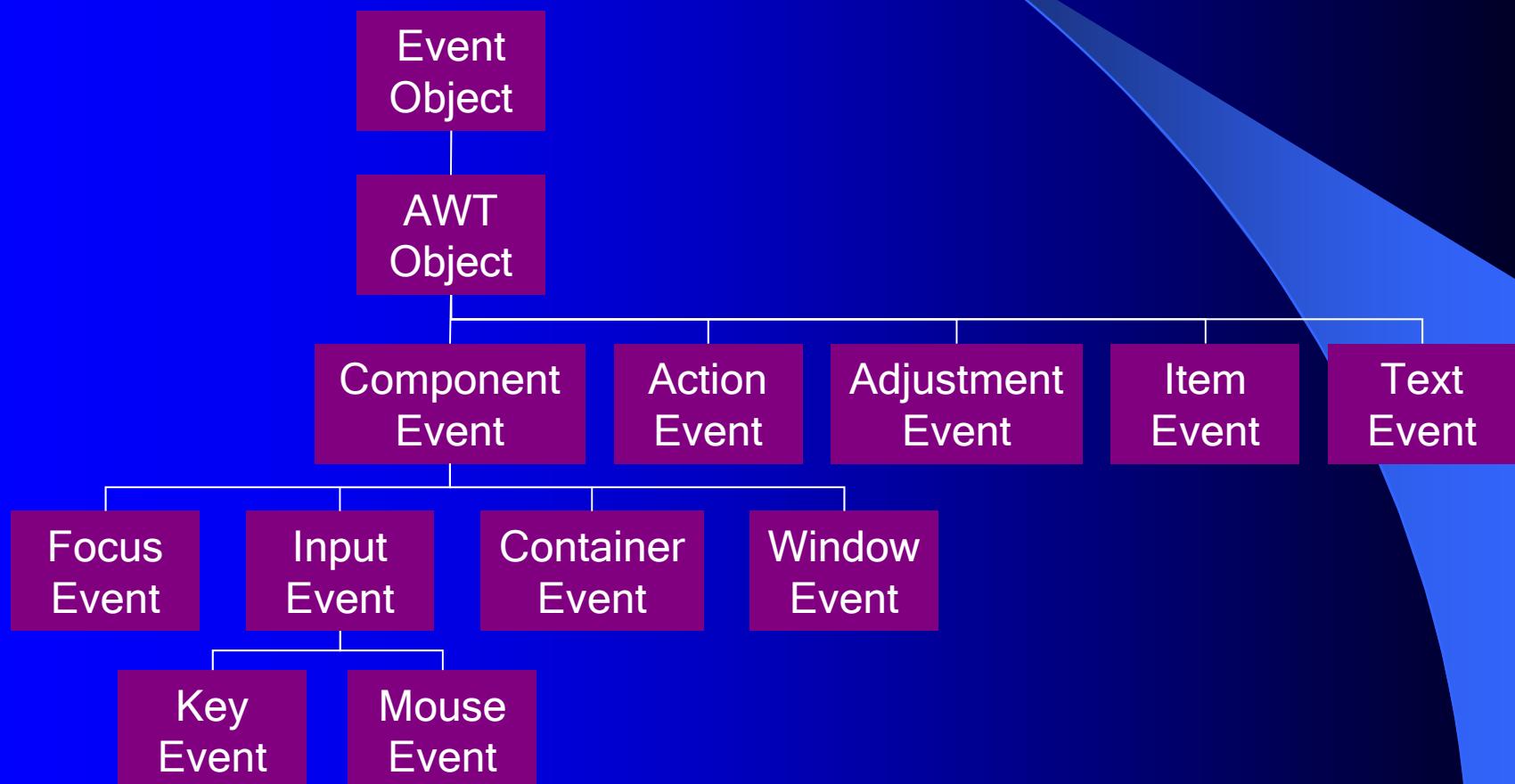
Beispiel 22 (Fort.)

anonyme Klasse als
Erweiterung der Button
Klasse

Konstruktor für
anonyme
JButton Klasse

Event-Typen

Welche Ereignisse (Events) gibt es, auf die man reagieren kann?



Event-Typen (Fort.)

Action Ereignisse

Listener-Interface	ActionListener
Registrierungsmethode	<code>void addActionListener(ActionListener l);</code>
Mögliche Ereignisquellen	JButton, JList, JMenuItem, JTextField
Ereignismethode	<code>public void actionPerformed(ActionEvent e);</code>
Bedeutung	eine Aktion wurde ausgelöst, z.B. der Button wurde gedrückt, ein Menu wurde ausgewählt, ein Text wurde in dem Textfeld eingegeben.

Event-Typen (Fort.)

Action Ereignisse: Beispiel

```
class MyButton extends JButton {  
    public MyButton() {  
        super("Drück mich!");  
        addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                System.out.println("Ups, gedrückt");  
            }  
        });  
    }  
}
```

Listener-Interface	ActionListener
Registrierungsmethode	public void addActionListener(ActionListener l);
Mögliche Ereignisquellen	JButton, JList, JMenuItem, JTextField
Ereignismethode	void actionPerformed(ActionEvent e);

Event-Typen (Fort.)

Window Ereignisse

Listener-Interface	WindowListener
Registrierungsmethode	<code>void addWindowListener(WindowListener l);</code>
Mögliche Ereignisquellen	JDialog, JFrame
Ereignismethode	<code>void windowActivated(WindowEvent e);</code> <code>void windowClosed(WindowEvent e);</code> <code>void windowClosing(WindowEvent e);</code> <code>void windowDeactivated(WindowEvent e);</code> <code>void windowDeiconified(WindowEvent e);</code> <code>void windowIconified(WindowEvent e);</code> <code>void windowOpened(WindowEvent e);</code>

Window Ereignisse: Beispiel

...

```
JFrame j = new JFrame();
j.setBounds(100,100,300,200);
j.addWindowListener(new WindowListener() {
    public void windowActivated(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
    public void windowDeactivated(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
});
j.setVisible(true);
```

Was ist das Problem?

Adapterklassen

Problem mit großen Listener Interfaces:

Es müssen immer alle Methoden implementiert werden, obwohl die meisten leer sind.

Lösung:

Zu allen Listener Interfaces gibt es die entsprechenden **Adapterklassen**, die alle Methoden mit leeren Methodenrumpfen implementiert. Verwenden Sie die entsprechenden Adapterklasse und überschreiben Sie nur die Methode, die Sie brauchen.

kanonische Namen: WindowListener \longleftrightarrow WindowAdapter

Adapterklassen (Fort.)

```
public interface WindowListener {  
    public void windowActivated(WindowEvent e);  
    public void windowClosed(WindowEvent e);  
    public void windowClosing(WindowEvent e);  
    public void windowDeactivated(WindowEvent e);  
    public void windowDeiconified(WindowEvent e);  
    public void windowIconified(WindowEvent e);  
    public void windowOpened(WindowEvent e);  
}
```

```
public class WindowAdapter implements WindowListener {  
    public void windowActivated(WindowEvent e) {}  
    public void windowClosed(WindowEvent e) {}  
    public void windowClosing(WindowEvent e) {}  
    public void windowDeactivated(WindowEvent e) {}  
    public void windowDeiconified(WindowEvent e) {}  
    public void windowIconified(WindowEvent e) {}  
    public void windowOpened(WindowEvent e) {}  
}
```

Go!

Beispiel 23

Window Ereignisse: Beispiel mit Adapterklasse

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

class Beispiel23_1 {
    public static void main(String [] args) {
        JFrame j = new JFrame();
        j.setBounds(100,100,300,200);
        j.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                j.dispose();
            }
            public void windowIconified(WindowEvent e) {
                j.setExtendedState(JFrame.NORMAL);
            }
        });
        j.setVisible(true);
    }
}
```

Deutlich übersichtlicher!

Go!

Gefahr bei Adapterklassen

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

class Beispiel23_2 {

    public static void main(String [] args) {
        JFrame j = new JFrame();
        j.setBounds(100,100,300,200);
        j.addWindowListener(new WindowAdapter() {
            public void windowclosing(WindowEvent e) {
                j.dispose();
            }
        });
        j.setVisible(true);
    }
}
```

Frage: Was macht dieses Programm?

Gefahr bei Adapterklassen (Forts.)

```
...  
public static void main(String [] args) {Fehler: ,c' statt ,C'  
    JFrame j = new JFrame();  
    j.setBounds(100,100,300,200);  
    j.addWindowListener(new WindowAdapter() {  
        public void windowclosing(WindowEvent e) {  
            j.dispose();  
        }  
    });  
    j.setVisible(true);  
}  
}
```

Problem:

- in **WindowAdapter** ist **windowClosing** bereits implementiert
- wir definieren **windowclosing**: eine neue Methode, die nichts mit **windowClosing** zu tun hat
- BS ruft **windowClosing**, nicht unsere **windowclosing** auf

Go!

Gefahr bei Adapterklassen (Forts.)

Lösung:

- Compiler mittels Pragmas sagen, dass eine Methode überlagert werden soll
- dann kann der Compiler Rechtschreibfehler aufdecken

```
class Beispiel23_3 {  
    public static void main(String [] args) {  
        JFrame j = new JFrame();  
        j.setBounds(100,100,300,200);  
        j.addWindowListener(new WindowAdapter() {  
            @Override  
            public void windowClosing(WindowEvent e) {  
                j.dispose();  
            }  
        });  
        j.setVisible(true);  
    }  
}
```

Sagt, dass nachfolgende Methode eine vorhandene Methode überlagern soll

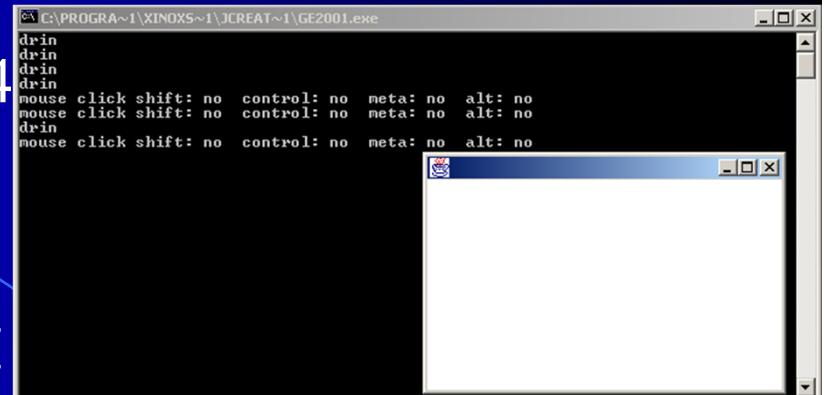
Event-Typen (Fort.)

Mouse Ereignisse

Listener-Interface	MouseListener
Registrierungsmethode	<code>void addMouseListener(MouseListener l);</code>
Mögliche Ereignisquellen	JComponent
Ereignismethode	<code>void mouseClicked(MouseEvent e);</code> <code>void mouseEntered(MouseEvent e);</code> <code>void mouseExited(MouseEvent e);</code> <code>void mousePressed(MouseEvent e);</code> <code>void mouseReleased(MouseEvent e);</code>

Go!

```
class Beispiel24 {  
    public static void main(String [] args) {  
        JFrame j = new JFrame();  
        j.setBounds(100,100,300,200);  
        j.setVisible(true);  
        j.addMouseListener(new MouseAdapter() {  
            @Override  
            public void mouseEntered(MouseEvent e) {  
                System.out.println("drin");  
            }  
            @Override  
            public void mouseClicked(MouseEvent e) {  
                System.out.print("mouse click");  
                System.out.print(" shift: " + (e.isShiftDown() ? "yes" : "no "));  
                System.out.print(" control: " + (e.isControlDown() ? "yes" : "no "));  
                System.out.print(" meta: " + (e.isMetaDown() ? "yes" : "no "));  
                System.out.print(" alt: " + (e.isAltDown() ? "yes" : "no "));  
                System.out.println(" Button: " + (e.getButton() == MouseEvent.BUTTON1  
                    ? "links"  
                    : e.getButton() == MouseEvent.BUTTON2  
                    ? "mitte"  
                    : "rechts"));  
            }  
        });  
    }  
}
```



Beispiel 24

Vorlesung 5

Event-Typen (Fort.)

MouseMotion Ereignisse

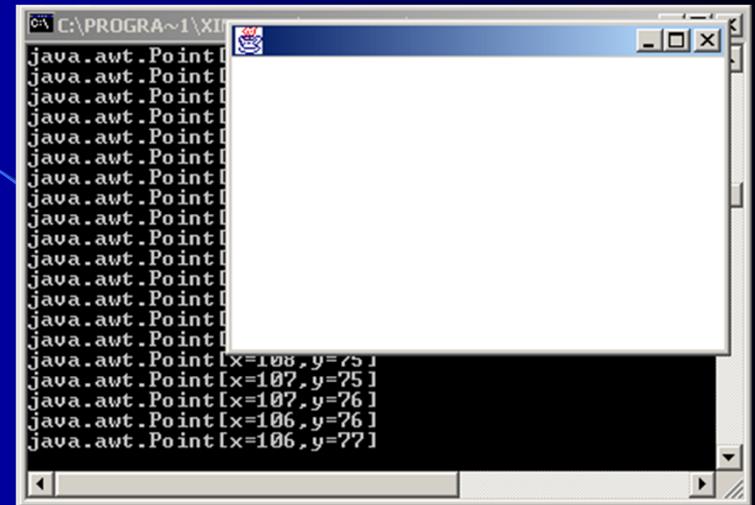
Listener-Interface	MouseListener
Registrierungsmethode	<code>void addMouseMotionListener(MouseMotionListener l);</code>
Mögliche Ereignisquellen	JComponent
Ereignismethode	<code>void mouseDragged(MouseEvent e);</code> <code>void mouseMoved(MouseEvent e);</code>
Bedeutung	Die Maus wurde bei gedrückter Maustaste bewegt (mouseDragged) bzw. bei nicht gedrückter Maustaste (mouseMoved).

Go!

Beispiel 24_1

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

```
class Beispiel24_1 {  
  
    public static void main(String [] args) {  
        JFrame j = new JFrame();  
        j.setBounds(100,100,300,200);  
        j.addMouseMotionListener(new MouseMotionAdapter()  
            @Override  
            public void mouseMoved(MouseEvent e) {  
                System.out.println(e.getPoint());  
            }  
        );  
        j.setVisible(true);  
    }  
}
```



Go!

Beispiel 24_2

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

class Beispiel24_2 extends JComponent {
    Point mPoint;
    Image mImg;
    public Beispiel24_2() {
        mImg = getToolkit().getImage("blauschatten.gif");
        addMouseMotionListener(new MouseMotionAdapter() {
            @Override
            public void mouseDragged(MouseEvent e) {
                mPoint = e.getPoint();
                repaint();
            }
        });
    }
    public void paintComponent(Graphics g) {
        if (mImg != null && mPoint != null)
            g.drawImage(mImg,mPoint.x,mPoint.y,this);
    }
    public static void main(String [] args) {
        JFrame j = new JFrame();
        j.add(new Beispiel24_2());
        j.setBounds(100,100,700,500);
        j.setVisible(true);
    }
}
```

Event-Typen (Fort.)

MouseWheel Ereignisse (ab JDK 1.4)

Listener-Interface	MouseListener
Registrierungsmethode	<code>void addMouseWheelListener(MouseWheelListener l);</code>
Mögliche Ereignisquellen	JComponent
Ereignismethode	<code>void mouseWheelMoved(MouseWheelEvent e);</code>
Bedeutung	Das Mausrad wurde bewegt.

Go!

Beispiel 24_3

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class Beispiel24_3 extends JComponent {
    int mY = 100;
    Image mlmg;
    public Beispiel24_3() {
        mlmg = Toolkit.getDefaultToolkit().getImage("blauschatten.gif");
        addMouseWheelListener(new MouseWheelListener() {
            public void mouseWheelMoved(MouseWheelEvent e) {
                // ...
                mY += e.getWheelRotation();
                mY += e.getUnitsToScroll();
                repaint();
            }
        });
    }
    public void paintComponent(Graphics g) {
        if (mlmg != null)
            g.drawImage(mlmg,getWidth()/2,mY,this);
    }
    public static void main(String [] args) {
        JFrame f = new JFrame();
        f.add(new Beispiel24_3());
        f.setBounds(100,100,700,500);
        f.setVisible(true);
    }
}
```

Liefert die Richtung zurück (immer 1 oder -1 pro Event)

Liefert die Richtung und die Einheiten zurück (zB. 3 oder -9)

Event-Typen (Fort.)

Key Ereignisse

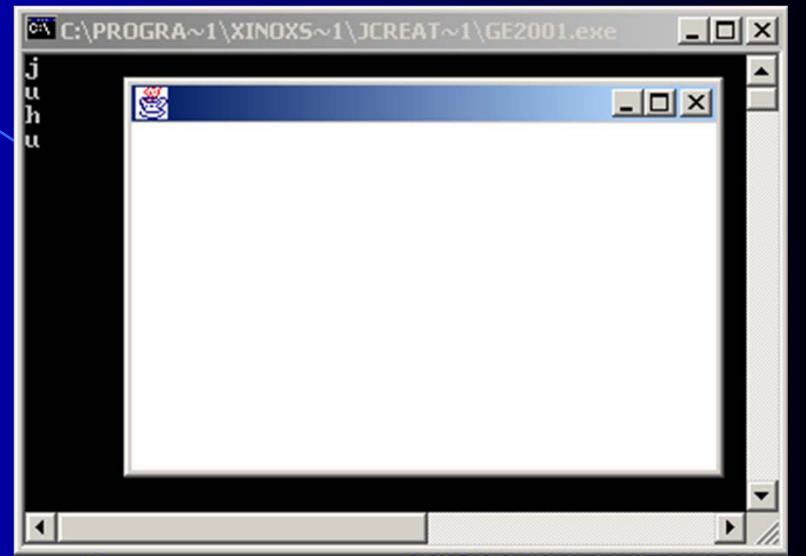
ListenerInterface	KeyListener
Registrierungsmethode	<code>void addKeyListener(KeyListener l);</code>
Mögliche Ereignisquellen	JComponent
Ereignismethode	<code>void keyPressed(KeyEvent e);</code> <code>void keyReleased(KeyEvent e);</code> <code>void keyTyped(KeyEvent e);</code>
Bedeutung	Eine Taste wurde gedrückt (<code>keyPressed</code>), oder losgelassen (<code>keyReleased</code>) oder gedrückt und losgelassen (<code>keyTyped</code>).

Go!

Event-Typen (Fort.)

Key Ereignisse: Beispiel

```
class Beispiel25 {  
    public static void main(String [] args) {  
        JFrame j = new JFrame();  
        j.addKeyListener(new KeyAdapter() {  
            // Unterschied: Shift und Control Taste  
            @Override  
            public void keyTyped(KeyEvent e) {  
                System.out.println(e.getKeyChar());  
            }  
        });  
        j.setBounds(100,100,300,200);  
        j.setVisible(true);  
    }  
}
```



Event-Typen (Fort.)

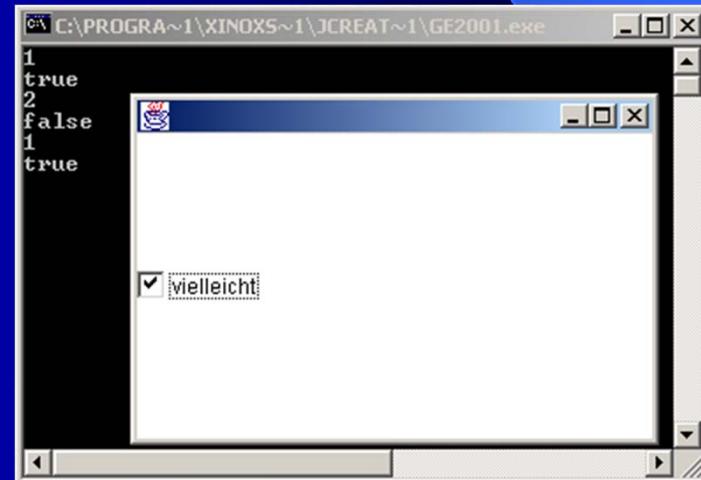
Item Ereignisse

ListenerInterface	ItemListener
Registrierungsmethode	<code>void addItemListener(ItemListener l);</code>
Mögliche Ereignisquellen	JCheckBox, JComboBox, JList, JCheckboxMenuItem
Ereignismethode	<code>void itemStateChanged(ItemEvent e);</code>
Bedeutung	Der Zustand hat sich verändert.

Go!

Beispiel 29

```
class Beispiel29 {  
    public static void main(String [] args) {  
        JFrame j = new JFrame();  
        j.add(new JCheckBox("vielleicht") {  
            {  
                addItemListener(new ItemListener() {  
                    public void itemStateChanged(ItemEvent e) {  
                        System.out.println(e.getStateChange() == ItemEvent.SELECTED);  
                        System.out.println(e.getStateChange());  
                        System.out.println(isSelected());  
                    }  
                });  
            }  
        });  
        j.setBounds(100,100,300,200);  
        j.setVisible(true);  
    }  
}
```



Event-Typen (Fort.)

Adjustment Ereignisse

ListenerInterface	AdjustmentListener
Registrierungs-methode	<code>void addAdjustmentListener(AdjustmentListener l);</code>
Mögliche Ereignisquellen	JScrollBar
Ereignismethode	<code>void adjustmentValueChanged(AdjustmentEvent e);</code>
Bedeutung	Der Wert wurde geändert.

Go!

Startwert

```
class Beispiel26 extends JFrame {  
    MyScrollbar m_Red = new MyScrollbar();  
    MyScrollbar m_Green = new MyScrollbar();  
    MyScrollbar m_Blue = new MyScrollbar();  
    class MyScrollbar extends JScrollBar {  
        int m_iValue;  
        MyScrollbar() {  
            super(VERTICAL,0,1,0,256);  
            addAdjustmentListener(new AdjustmentListener() {  
                public void adjustmentValueChanged(AdjustmentEvent e) {  
                    m_iValue = e.getValue();  
                    setColor();  
                }  
            });  
        }  
    }  
}
```

```
public Beispiel26() {  
    setBounds(100,100,300,200);setLayout(new FlowLayout());  
    add(m_Red);add(m_Green);add(m_Blue);  
    setColor();setVisible(true);  
}  
public void setColor() {
```

→ `getContentPane().setBackground(new Color(m_Red.m_iValue,m_Green.m_iValue,m_Blue.m_iValue));`

```
    }
```

```
    public static void main(String [] args) {  
        new Beispiel26();  
    }  
}
```

Beispiel 26

min. Wert

max. Wert

Frage: Wieso kann an dieser Stelle `setColor` aufgerufen werden?

Go!

Beispiel 26 (Fort.)

```
class Beispiel26_1 extends JFrame implements AdjustmentListener {  
    JScrollBar red = new JScrollBar(JScrollBar.VERTICAL,0,1,0,256);  
    JScrollBar green = new JScrollBar(JScrollBar.VERTICAL,0,1,0,256);  
    JScrollBar blue = new JScrollBar(JScrollBar.VERTICAL,0,1,0,256);
```

```
public Beispiel26_1() {  
    setBounds(100,100,300,200);  
    setLayout(new FlowLayout());  
    add(red);  
    add(green);  
    add(blue);  
    setVisible(true);  
    red.addAdjustmentListener(this);  
    green.addAdjustmentListener(this);  
    blue.addAdjustmentListener(this);  
}
```

```
public void adjustmentValueChanged(AdjustmentEvent e) {  
    getContentPane().setBackground(new Color(red.getValue(),green.getValue(),blue.getValue()));  
}  
  
public static void main(String [] args) {  
    new Beispiel26_1();  
}
```

Frage: Welches Programm ist
"schöner"?

Event-Typen (Fort.)

Focus Ereignisse

ListenerInterface	FocusListener
Registrierungsmethode	<code>void addFocusListener(FocusListener l);</code>
Mögliche Ereignisquellen	JComponent
Ereignismethode	<code>void focusGained(FocusEvent e);</code> <code>void focusLost(FocusEvent e);</code>
Bedeutung	Eine Komponente erhält den Focus (<code>focusGained</code>) bzw. verliert ihn (<code>focusLost</code>).

Event-Typen (Fort.)

Component Ereignisse

ListenerInterface	ComponentListener
Registrierungs-methode	<code>void addComponentListener(ComponentListener l);</code>
Mögliche Ereignisquellen	JComponent
Ereignismethode	<code>void componentHidden(ComponentEvent e);</code> <code>void componentMoved(ComponentEvent e);</code> <code>void componentResized(ComponentEvent e);</code> <code>void componentShown(ComponentEvent e);</code>
Bedeutung	Eine Komponente wird verdeckt (<code>setVisible(false)</code>), bewegt, in der Größe verändert oder sichtbar (<code>setVisible(true)</code>).

Go!

Beispiel 27

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

class Beispiel27 {

    public static void main(String [] args) {
        JFrame j = new JFrame();
        j.addComponentListener(new ComponentAdapter() {
            @Override
            public void componentResized(ComponentEvent e) {
                System.out.println("Resize: " + e);
            }
        });
        j.setBounds(100,100,300,200);
        j.setVisible(true);
    }
}
```

Go!

Beispiel 27_1

```
class Beispiel27_1 {  
  
    public static void main(String [] args) {  
        JFrame f = new JFrame();  
        f.setBounds(200,200,200,200);  
        f.setVisible(true);  
  
        JFrame j = new JFrame();  
        j.setBounds(100,100,300,200);  
        j.addComponentListener(new ComponentAdapter() {  
            @Override  
            public void componentResized(ComponentEvent e) {  
                java.awt.Dimension dim = j.getSize();  
                f.setSize(dim.width,dim.width);  
            }  
        });  
        j.setVisible(true);  
    }  
}
```

Event-Typen (Fort.)

Container Ereignisse

ListenerInterface	ContainerListener
Registrierungs-methode	<code>void addContainerListener(ContainerListener l);</code>
Mögliche Ereignisquellen	JContainer
Ereignismethode	<code>void componentAdded(ContainerEvent e);</code> <code>void componentRemoved(ContainerEvent e);</code>
Bedeutung	Eine Komponente wird dem Container hinzugefügt bzw. wird aus dem Container entfernt.

Lambda Ausdrücke

Problem mit anonymen Klassen zur Implementierung der ActionListener:

- immer wieder sehr viel Schreibarbeit
- die komplette Deklaration der anonymen Klasse könnte der Compiler selber schreiben
- daher ab Java 1.8: Lambda Ausdrücke (Begriff aus der funktionalen Programmierung)
- nur noch der Inhalt der Funktion muss implementiert werden
- nicht mehr implementiert werden muss:
 - Deklaration der anonymen Klasse
 - Deklaration der überlagerten Methode

Bisher!

Lambda Ausdrücke (Forts.)

Neu!

- Bisher:

...

```
addActionListener(
```

```
    new ActionListener() {
```

```
        public void actionPerformed(ActionEvent e) {
```

```
            System.out.println("juhu");
```

```
        }
```

```
    }
```

```
);
```

- mit Lambda Ausdrücken

...

```
addActionListener(
```

```
    e ->
```

```
        System.out.println("juhu")
```

```
);
```

Alles überflüssig, da nichts anderes geschrieben werden kann (Ausnahme: „e“ für den Namen des Arguments)



WICHTIG: kein
Semikolon hier !!!!



Lambda Ausdrücke (Forts.)

- Lambda Ausdrücke funktionieren nur bei Interfaces, die genau eine Methode enthalten
- sie funktionieren *nicht* bei
 - Interfaces mit mehreren Methoden
 - abstrakten Klassen
 - normale Klassen
- solche Interfaces heißen „funktionale Interfaces“ (sie spezifizieren im Wesentlichen eine Funktion)

Lambda Ausdrücke (Forts.)

- Lambda Ausdrücke: ein näherer Blick

```
interface Juhu {  
    public void doit();  
}  
  
public class Lambda1 {  
  
    public static void main(String[] args) {  
        Juhu j1 = new Juhu() {  
            public void doit() {  
                System.out.println("dies ist der alte Weg");  
            }  
        };  
        Juhu j2 = () -> System.out.println("mit Lambda Ausdruck");  
        j1.doit();  
        j2.doit();  
    }  
}
```

ohne Parameter müssen leere
Klammern gesetzt werden

Lambda Ausdrücke (Forts.)

- die Methoden können auch mehrere Parameter enthalten

```
interface Juhu {  
    public void doit(int i,float f);  
}  
  
public class Lambda2 {  
  
    public static void main(String[] args) {  
        Juhu j1 = new Juhu() {  
            public void doit(int i,float f) {  
                System.out.println("old school: i = " + i + " f = " + f);  
            }  
        };  
        Juhu j2 = (i,f) -> System.out.println("Lambda: i = " + i + " f = " + f);  
        j1.doit(12,34.6f);  
        j2.doit(5,7.8f);  
    }  
}
```

mehrere Parameter müssen
auch in Klammern gesetzt
werden

Go!

Lambda Ausdrücke (Forts.)

- mehrere Statements müssen in einem Block zusammengefasst werden

```
interface Juhu {  
    public void doit(int i,int j);  
}  
  
public class Lambda4 {  
  
    public static void main(String[] args) {  
        Juhu j = (i1,i2) -> {  
            System.out.println("jetzt kommt " + i1 + " mal die " + i2);  
            for(int i = 0;i < i1;++i)  
                System.out.println(i2);  
        };  
        j.doit(10,13);  
    }  
}
```



Lambda Ausdrücke (Forts.)

- die Methoden können auch einen Rückgabewert haben

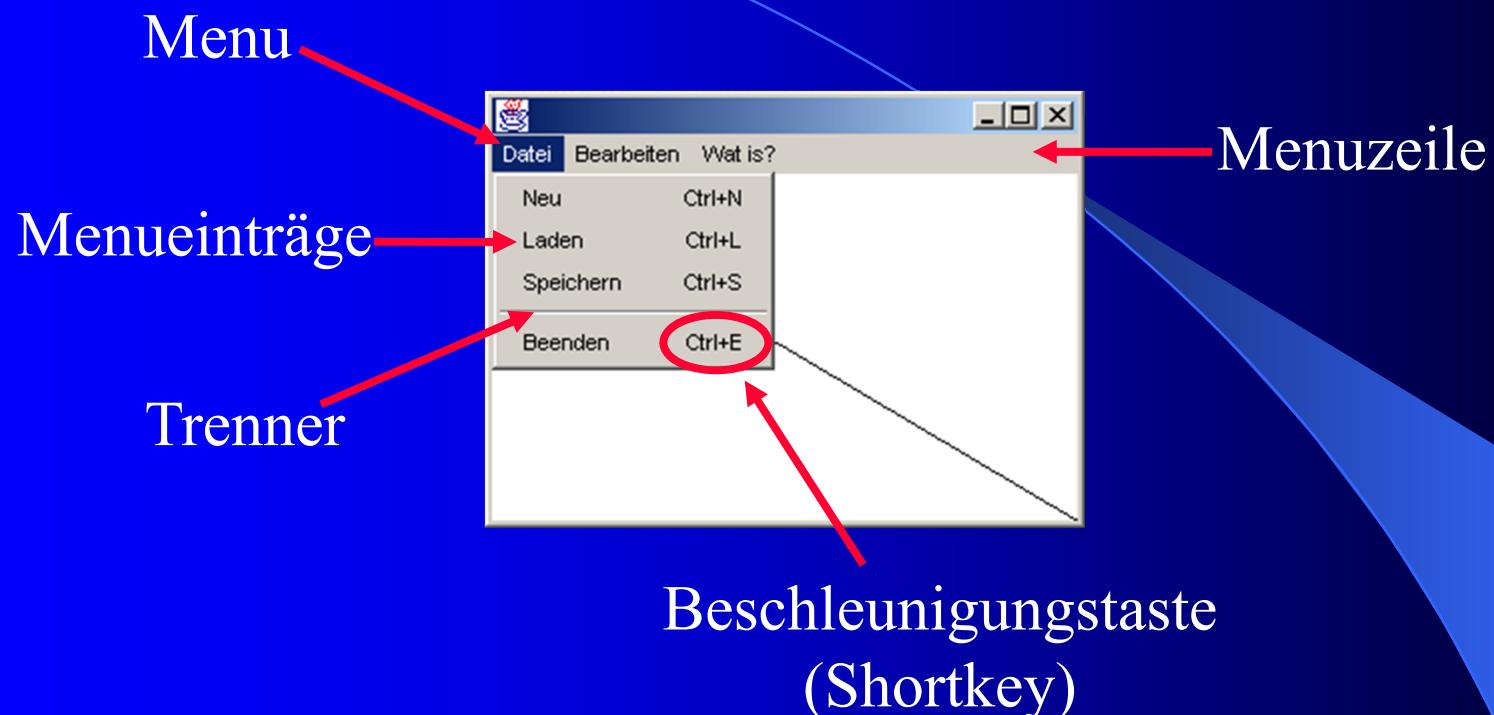
```
interface Juhu {  
    public int doit(int i,int j);  
}  
  
public class Lambda3 {  
  
    public static void main(String[] args) {  
        Juhu j1 = (i1,i2) -> {return i1*i2;};  
        Juhu j2 = (x,y) -> x / y;  
  
        int a = j1.doit(12,10);  
        int b = j2.doit(12,5);  
        System.out.println(a + " " + b);  
    }  
}
```

obwohl nur ein Statement (**return**)
muss es dennoch im Block stehen

Spezialfall „Lambda Ausdrücke“:
das **return** kann weggelassen
werden, dann auch ohne Block

Vorlesung 6

Menus



Sinn und Zweck von Menus:

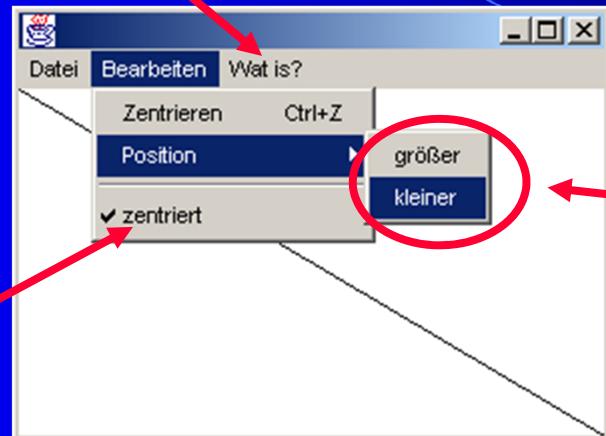
- Interaktion mit dem Benutzer
- Benutzer kann Aktionen auswählen und anstoßen
- Benutzer kann einfache Daten in dem Programm setzen

Menus (Fort.)

Menueintrag
als Checkbox

Hilfemenu

Untermenu



- ein Programm kann maximal **eine** Menuleiste haben
- eine Menuleiste kann **viele** Menus haben (und ein Hilfemenu)
- ein Menu kann **viele** Menueinträge haben
- ein Menueintrag kann selber wieder ein Menu sein
- ein Menueintrag kann **einen** Shortkey haben
- eine Checkbox kann ein Menueintrag sein

Menuleiste

- Eine Menuleiste wird durch die Klasse JMenuBar realisiert
- eine Menuleiste wird durch die **setJMenuBar**-Methode von Frame dem Fenster hinzugefügt
- die **getInsets**-Methode berücksichtigt dieses neue Element
- eine Menuleiste **ohne Menus** wird aber **nicht dargestellt**

```
import javax.swing.*;  
  
class Beispiel30_1 {  
  
    public static void main(String [] args) {  
        JFrame j = new JFrame();  
        j.setBounds(100,100,300,200);  
        JMenuBar menuBar = new JMenuBar();  
        j.setJMenuBar(menuBar);  
        j.setVisible(true);  
    }  
}
```

Menu

- Ein Menu wird durch die Klasse JMenu realisiert
- ein Menu wird der Menuleiste durch die add-Methode hinzugefügt

```
import javax.swing.*;  
  
class Beispiel30_2 {  
  
    public static void main(String [] args) {  
        JFrame j = new JFrame();  
        j.setBounds(100,100,300,200);  
        JMenuBar menuBar = new JMenuBar();  
        menuBar.add(new JMenu("Datei"));  
        j.setJMenuBar(menuBar);  
        j.getContentPane().setBackground(java.awt.Color.PINK);  
        j.setVisible(true);  
    }  
}
```

Menueinträge

- ein Menueintrag wird durch die Klasse JMenuItem realisiert
- ein Menueintrag wird einem Menu durch dessen add-Methode hinzugefügt
- ein Menueintrag kann auch direkt durch die void add(String) Methode von Menu erzeugt werden

```
import javax.swing.*;  
  
class Beispiel30_4 {  
    public static void main(String [] args) {  
        JFrame j = new JFrame();  
        j.setBounds(100,100,300,200);  
        JMenuBar menuBar = new JMenuBar();  
        JMenu menu = new JMenu("Datei");  
        menu.add(new JMenuItem("Neu"));  
        menu.add("Beenden");  
        menuBar.add(menu);  
        j.setJMenuBar(menuBar);  
        j.setVisible(true);  
    }  
}
```

Menueinträge (Fort.)

- ein Menueintrag kann selber wieder ein Menu sein
- ein solches Sub-Menu wird auch durch die **add**-Methode dem Menu hinzugefügt

```
import javax.swing.*;  
class Beispiel30_5 {  
    public static void main(String [] args) {  
        JFrame j = new JFrame();  
        j.setBounds(100,100,300,200);  
        JMenuBar menuBar = new JMenuBar();  
        JMenu menu = new JMenu("Datei");  
        JMenu sub = new JMenu("Speichern");  
        menu.add(new JMenuItem("Neu"));  
        menu.add(sub);  
        sub.add(new JMenuItem(..als HTML"));  
        sub.add(new JMenuItem(..als Source"));  
        menuBar.add(menu);  
        j.setJMenuBar(menuBar);  
        j.setVisible(true);  
    }  
}
```

Menueinträge (Fort.)

- ein Menueintrag kann auch eine Checkbox sein
- eine solche Checkbox stellt einen booleschen Zustand dar

```
import javax.swing.*;  
class Beispiel30_6 {  
    public static void main(String [] args) {  
        JFrame j = new JFrame();  
        j.setBounds(100,100,300,200);  
        JMenuBar menuBar = new JMenuBar();  
        JMenu menu = new JMenu("Datei");  
        JMenu sub = new JMenu("Speichern");  
        menu.add(new JMenuItem("Neu"));  
        menu.add(sub);  
        menu.add(new JCheckBoxMenuItem("zentriert"));  
        sub.add(new JMenuItem(..als HTML"));  
        sub.add(new JMenuItem(..als Source"));  
        menuBar.add(menu);  
        j.setJMenuBar(menuBar);  
        j.setVisible(true);  
    }  
}
```

Menueinträge (Fort.)

- Menueinträge können durch die Methode void `setEnabled(boolean b)` der `MenuItem` Klasse aktiviert und deaktiviert werden
- im Initialzustand sind Menueinträge aktiviert, d.h. auswählbar

```
public static void main(String [] args) {  
    JFrame j = new JFrame();  
    j.setBounds(100,100,300,200);  
    JMenuBar menuBar = new JMenuBar();  
    JMenu menu = new JMenu("Datei");  
    JMenu sub = new JMenu("Speichern");  
    menu.add(new JMenuItem("Neu"));  
    menu.add(sub);  
    menu.add(new JCheckBoxMenuItem("zentriert"));  
    sub.add(new JMenuItem(..als HTML"));  
    sub.add(new JMenuItem(..als Source"));  
    menuBar.add(menu);  
    j.setJMenuBar(menuBar);  
    j.setVisible(true);  
    sub.setEnabled(false);  
}
```

Separatoren

- Menueinträge können durch horizontale Linien (Separatoren) getrennt werden
- dies erfolgt durch den `addSeparator()` Befehl der Klasse `Menu`

```
public static void main(String [] args) {  
    JFrame j = new JFrame();  
    j.setBounds(100,100,300,200);  
    JMenuBar menuBar = new JMenuBar();  
    JMenu menu = new JMenu("Datei");  
    JMenu sub = new JMenu("Speichern");  
    menu.add(new JMenuItem("Neu"));  
    menu.add(sub);  
    menu.addSeparator();  
    menu.add(new JCheckBoxMenuItem("zentriert"));  
    sub.add(new JMenuItem(..als HTML"));  
    sub.add(new JMenuItem(..als Source"));  
    menuBar.add(menu);  
    j.setJMenuBar(menuBar);  
    j.setVisible(true);  
}
```

Shortcuts

- Auswählen von Menueinträge kann durch Tastenkombinationen (Shortcuts) erfolgen
- Shortcuts werden durch Klasse KeyStroke implementiert und mittels setAccelerator dem Menu hinzugefügt

```
public static void main(String [] args) {  
    JFrame j = new JFrame();  
    j.setBounds(100,100,300,200);  
    JMenuBar menuBar = new JMenuBar();  
    JMenu menu = new JMenu("Datei");  
    JMenu sub = new JMenu("Speichern");  
    menu.add(new JMenuItem("Neu"));  
    menu.add(sub);  
    JMenuItem zenItem = new JCheckBoxMenuItem("zentriert");  
    zenItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_Z,  
                                                 KeyEvent.CTRL_DOWN_MASK));  
    menu.add(zenItem);  
    sub.add(new JMenuItem(..als HTML"));  
    sub.add(new JMenuItem(..als Source"));  
    menuBar.add(menu);  
    j.setJMenuBar(menuBar);  
    j.setVisible(true);  
}
```

Menauswahl

Wie reagiert man auf die Auswahl eines Menueintrags?

- bei der Auswahl wird dem JMenuItem ein ActionEvent geschickt
- dem MenuItem kann ein ActionListener hinterlegt werden, der auf dieses ActionEvent reagiert (siehe auch: Drücken eines Button)

```
class JMenuItem {  
    public void addActionListener(ActionListener l);  
}
```

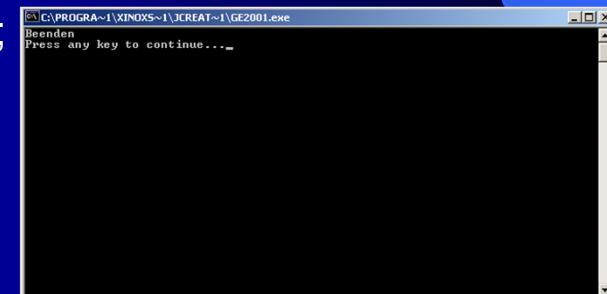
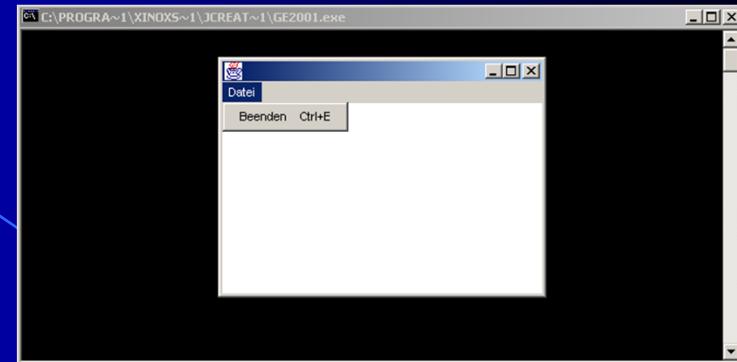
```
interface ActionListener {  
    public void actionPerformed(ActionEvent e);  
}
```

```
class ActionEvent {  
    public String getActionCommand();  
}
```

Go!

Beispiel 31

```
import javax.swing.*;  
import java.awt.*;  
import java.awt.event.*;  
class Beispiel31 {  
    public static void main(String [] args) {  
        JFrame j = new JFrame();  
        j.setBounds(100,100,300,200);  
        JMenuBar menuBar = new JMenuBar();  
        JMenu menu = new JMenu("Datei");  
        JMenuItem endeltem = new JMenuItem("Beenden");  
        endeltem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_E,  
                                         KeyEvent.CTRL_DOWN_MASK));  
        menu.add(endeltem);  
        menuBar.add(menu);  
        j.setJMenuBar(menuBar);  
        j.setVisible(true);  
        endeltem.addActionListener(e -> {  
            System.out.println(e.getActionCommand());  
            j.dispose();  
        })  
    }  
}
```



Go!

Beispiel 32

```
class Beispiel32 extends JFrame implements ActionListener{  
    public Beispiel32() {  
        setBounds(100,100,300,200);  
        JMenuBar menuBar = new JMenuBar();  
        JMenu menu = new JMenu("Action");  
        JMenuItem linksItem = new JMenuItem("Links");  
        linksItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_L,  
                                         KeyEvent.CTRL_DOWN_MASK));  
        JMenuItem rechtsItem = new JMenuItem("Rechts");  
        rechtsItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_R,  
                                         KeyEvent.CTRL_DOWN_MASK));  
        linksItem.addActionListener(this);rechtsItem.addActionListener(this);  
        menu.add(linksItem);menu.add(rechtsItem);  
        menuBar.add(menu);  
        setJMenuBar(menuBar);  
        setVisible(true);  
    }  
    public void actionPerformed(ActionEvent e) {  
        Point p = getLocation();  
        if (e.getActionCommand() == "Links") {p.x = p.x - 10;}  
        else {p.x = p.x + 10;}  
        setLocation(p);  
    }  
}
```

Ist diese Art der
Eventbehandlung
"objektorientiert"?
Was ist das
Problem?

Beispiel 32 (Fort.)

```
...  
JMenuItem linksItem = new JMenuItem("Links");  
JMenuItem rechtsItem = new JMenuItem("Rechts");  
JMenuItem obenItem = new JMenuItem("Oben");  
JMenuItem untenItem = new JMenuItem("Unten");  
  
...  
linksItem.addActionListener(this);  
rechtsItem.addActionListener(this);  
obenItem.addActionListener(this);  
untenItem.addActionListener(this);  
}  
  
public void actionPerformed(ActionEvent e) {  
    Point p = getLocation();  
    if (e.getActionCommand() == "Links") {  
        p.x = p.x - 10;  
    } else {  
        p.x = p.x + 10;  
    }  
    setLocation(p);  
}  
...
```

neu

Diese Art der Verbindung
("mittels Namen") ist **nicht**
objektorientiert!!!

Problem von Beispiel 32

- für alle **MenuItem**s ist der gleiche **ActionListener** ("Sachbearbeiter") installiert, in diesem Fall **this**

```
linksItem.addActionListener(this);  
rechtsItem.addActionListener(this);  
obenItem.addActionListener(this);  
untenItem.addActionListener(this);
```

- der **ActionListener** identifiziert den Aufrufer anhand des Namens

```
public void actionPerformed(ActionEvent e) {  
    Point p = getLocation();  
    if (e.getActionCommand() == "Links") {  
        p.x - p.x - 10,  
    }  
    ...
```

Problem von Beispiel 32 (Fort.)

- Probleme entstehen,
 - wenn sich der Name ändert
 - wenn neue Namen hinzukommen
- Verbindung sollte nicht über Namen, sondern über Objekte erfolgen

Lösung:

jeder Menueintrag bekommt seinen eigenen Sachbearbeiter

Go!

Problem von Beispiel 32 (Fort.)

```
class Beispiel32_2 extends JFrame {  
    class MoveOff implements ActionListener {  
        int mDx;int mDy;  
        public MoveOff(int dx,int dy) {  
            mDx = dx;mDy = dy;  
        }  
        public void actionPerformed(ActionEvent e) {  
            Point p = getLocation();  
            p.x += mDx;p.y += mDy;  
            setLocation(p);  
        }  
    }  
    public Beispiel32_2() {  
        ...  
        JMenuItem linksItem = new JMenuItem("Links");  
        JMenuItem rechtsItem = new JMenuItem("Rechts");  
        JMenuItem obenItem = new JMenuItem("Oben");  
        JMenuItem untenItem = new JMenuItem("Unten");  
        ...  
        linksItem.addActionListener(new MoveOff(-10,0));  
        rechtsItem.addActionListener(new MoveOff(10,0));  
        obenItem.addActionListener(new MoveOff(0,-10));  
        untenItem.addActionListener(new MoveOff(0,10));  
    }  
}
```

es gibt **eine** eigene
ActionListener Klasse,
aber

jeder Menueeintrag hat **sein
eigenes** ActionListener
Objekt

Problem von Beispiel 32 (Fort.)

```
...  
class Beispiel32_2 extends JFrame {  
    class MoveOff implements ActionListener {  
        int mDx;int mDy;  
        public MoveOff(int dx,int dy) {  
            mDx = dx;mDy = dy;  
        }  
        public void actionPerformed(ActionEvent e) {  
            Point p = getLocation();  
            p.x += mDx;p.y += mDy;  
            setLocation(p);  
        }  
    }  
    public Beispiel32_2() {  
        ...  
    }  
    ...  
}
```

Frage: Wessen Methoden werden hier aufgerufen und warum geht das?

Problem von Beispiel 32 (Fort.)

```
...  
class Beispiel32_2 extends JFrame {  
    public void mymove(int x,int y) {  
        Point p = getLocation();  
        p.x += x;  
        p.y += y;  
        setLocation(p);  
    }  
}
```

```
public Beispiel32_2() {  
    setBounds(100,100,300,200);  
    ...  
    linksItem.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            mymove(-10,0);  
        }  
    });  
    rechtsItem.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            mymove(10,0);  
        }  
    });  
    ...  
}
```

eine Methode, die in
jede Richtung das
Fenster verschieben
kann

und

jeder Menueeintrag hat
seine eigene anonyme
Klasse

Go!

Problem von Beispiel 32 (Fort.)

```
class Beispiel32_3 extends JFrame {
```

```
    public void mymove(int dx,int dy) {  
        Point p = getLocation();  
        p.x += dx;  
        p.y += dy;  
        setLocation(p);  
    }
```

```
    public Beispiel32_3() {
```

```
        ...  
        JMenu menu = new JMenu("Action");  
        JMenuItem linksItem = new JMenuItem("Links");  
        JMenuItem rechtsItem = new JMenuItem("Rechts");  
        JMenuItem obenItem = new JMenuItem("Oben");  
        JMenuItem untenItem = new JMenuItem("Unten");  
        ...
```

```
        linksItem.addActionListener(e -> mymove(-10,0));  
        rechtsItem.addActionListener(e -> mymove(10,0));  
        obenItem.addActionListener(e -> mymove(0,-10));  
        untenItem.addActionListener(e -> mymove(0,10));
```

```
}
```

```
...
```

Mit Lambda Ausdrücken
sehr kompakte Schreibweise

```
        linksItem.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                mymove(-10,0);  
            }  
        });
```

wird ersetzt durch

Popup Menus

Popup Menus sind Menus, die kontextabhängig sind, d.h. sie bieten Aktionen, die nur in bestimmten Bereichen des Programms zur Verfügung stehen.

Sie sind nicht permanent vorhanden, sondern erscheinen nur auf Anforderung (Maustaste, spezielle Tastaturtasten, Kombination von beiden).

Der Aufruf und das Erscheinungsbild unterscheidet sich zwischen unterschiedlichen Betriebssystemen und graphischen Oberflächen.



Popup Menus (Fort.)

Popup Menus werden durch die Klasse JPopupMenu implementiert. Sie ist von der Klasse Menu abgeleitet.

```
class JPopupMenu extends JMenu {  
    ...  
    public void show(Component father,int x,int y);  
    ...  
}
```

Die Methode **show** zeigt das Menu auf dem Bildschirm. Es verschwindet automatisch, sobald ein MenuItem ausgewählt oder etwas anderes selektiert wird.

father ist die Komponente, zu der das Popupmenu gehört, x und y beschreiben die Koordinate, an der das Menu erscheinen soll.

Popup Menus (Fort.)

Frage: Wann und wie muss die `show` Methode aufgerufen werden?

Problem: Es kann nicht einfach der Mausklick abgefangen werden, da in unterschiedlichen graphischen Oberflächen die Popup Menus unterschiedlich implementiert sind.

Lösung: Jegliches Mausereignis abfangen und abfragen, ob das "abstrakte" Ereignis zum Aufrufen eines Popup Menus eingetreten ist.

```
class MouseEvent {  
    ...  
    public boolean isPopupTrigger();  
    ...  
}
```

Go!

<https://docs.oracle.com/javase/7/docs/api/javax/swing/JPopupMenu.html>

Beispiel 33

```
class Beispiel33 extends JFrame {  
    JPopupMenu pop = new JPopupMenu();  
    public Beispiel33() {  
        setBounds(100,100,300,200);  
        pop.add("juhu");  
        pop.add(new JMenuItem("doll"));  
  
        enableEvents(AWTEvent.MOUSE_EVENT_MASK);  
        setVisible(true);  
    }  
    protected void processMouseEvent(MouseEvent e) {  
        if (e.isPopupTrigger()) {  
            pop.show(e.getComponent(),e.getX(),e.getY());  
        }  
        super.processMouseEvent(e);  
    }  
    public static void main(String [] args) {  
        new Beispiel33();  
    }  
}
```

Fenster soll auf alle
Mausereignisse reagieren

Soll Popup Menu
angezeigt werden?

Zeige das Popup
Menu an!

Führe das Standard-
verhalten noch aus

Vorlesung 7

Dialoge

Dialoge dienen dazu, Eingaben (in komplexer Form) vom Benutzer anzunehmen und an das Programm weiterzugeben.

Es können zwei Situationen unterschieden werden:

1. die Eingabe ist ***nicht unbedingt notwendig***, d.h. das Programm kann auch ohne die Daten weiterarbeiten (Bsp.: Farbe der Steine im Life-Spiel) → ***nicht-modale Dialoge***
2. die Eingabe ist ***notwendig***, das Programm kann nicht ohne diese Daten sinnvoll weiterarbeiten (Bsp.: Druck Dialog)
→ ***modale Dialoge***

Erzeugen von Dialoge

Dialoge werden in fünf Schritten erzeugt:

1. Ein Dialogfenster wird erzeugt
2. Dem Dialogfenster wird ein Layout-Manager zugeordnet
3. Dem Dialogfenster werden die Dialogelemente hinzugefügt
4. Das Dialogfenster packen (pack-Methode)
5. Das Dialogfenster wird angezeigt (`setVisible(true);`)

Erzeugen von Dialogfenstern

Dialogfenster werden von `JDialog` abgeleitet.

- haben eine Titelleiste
- können keine Menus haben
- können weder maximiert noch in die Taskleiste gelegt werden
- können das Programm blockieren (Möglichkeit modaler Dialoge)

Einfügen von Dialog-Elementen

Die einzelnen Dialog-Elemente werden mittels der add-Methode dem Dialog-Fenster ***logisch*** zugeordnet.

```
class Container {  
    ...  
    public Component add(Component comp);  
    public Component add(Component comp,int pos);  
    public Component add(String pos,Component comp);  
    public void add(Component comp,Object Constraints);  
    public void add(Component comp,Object Constraints,int pos);  
    ...  
}
```

Die logische Zuordnung sagt noch nichts über die Darstellung aus (siehe Layout-Manager). Alle Dialog-Elemente werden sequentiell (hintereinander) angeordnet.

Zuordnung eines Layout-Managers

Frage: Wie kommt man von der logischen, sequentiellen Reihenfolge der Dialogelemente zu einer graphischen Anordnung im Dialogfenster?

Antwort: Ein Layout-Manager berechnet, wie die Dialogelemente geschickt zueinander angeordnet werden können. Dazu gibt man ein *logisches* Layout an ("Element a unterhalb von Element b anordnen").

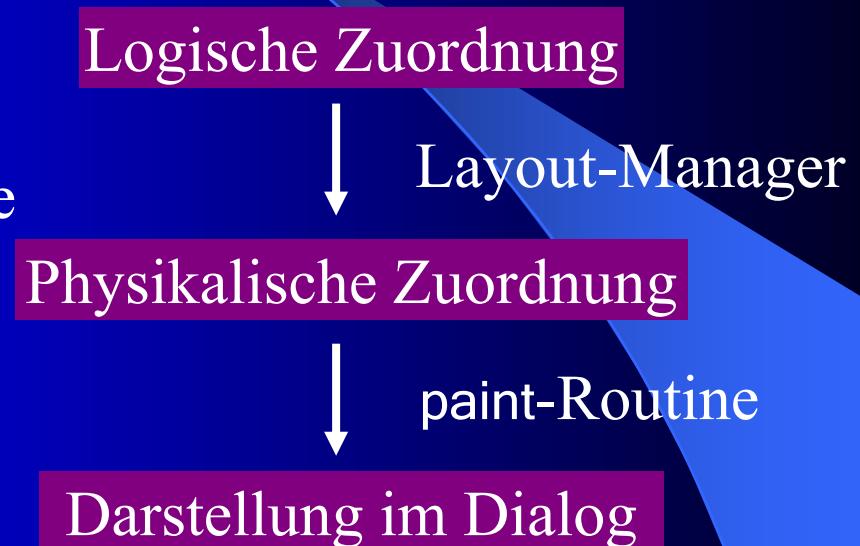
Bei anderen Graphiksystemen müssen die Elemente *physikalisch* zueinander angeordnet werden ("Element a liegt 23 Pixel unterhalb von Element b. Beide haben eine x-Koordinate von 37 Pixel").

Zuordnung eines Layout-Managers (Fort.)

Der Layout-Manager rechnet die logische Zuordnung in die physikalische Zuordnung um.

Vorteil dieses Vorgehens:

- schafft eine neue, abstraktere Ebene
- ist flexibler, so dass es auf unterschiedlichen Systemen funktionieren kann
- Benutzer muss nicht die Dialoge zeichnen, sondern logisch entwerfen



Zuordnung eines Layout-Managers (Fort.)

Die Klasse Container hat eine Methode

```
public void setLayout(LayoutManager mgr);
```

die den angegebenen Layout-Manager für die Darstellung und das Arrangement der Dialog-Elemente festlegt.

Es gibt 5+1 Layout-Manager in AWT und diverse in Swing:

1. FlowLayout
 2. GridLayout
 3. BorderLayout
 4. CardLayout (nicht behandelt)
 5. GridBagLayout (nicht behandelt)
 6. null

FlowLayout-Manager

- Der FlowLayout-Manager ist der einfachste Layout-Manager.
- Alle Dialog-Elemente werden gemäß ihrer logischen Reihenfolge auch physikalisch hintereinander in einer Reihe abgelegt.
- Ist eine Reihe voll, so wird unter dieser eine neue Reihe eröffnet.
- Es gibt drei Konstruktoren:

`public FlowLayout();`

`public FlowLayout(int align);`

`public FlowLayout(int align,int hgap,int vgap);`

- `FlowLayout.LEFT`
- `FlowLayout.RIGHT`
- `FlowLayout.CENTER`

Go!

Beispiel für FlowLayout-Manager

```
class Beispiel34 extends JPanel {  
    public Beispiel34(JFrame f, String title, LayoutManager mgr) {  
        super(f, title);  
        setLayout(mgr);  
        add(new JButton("1"));  
        add(new JButton("Ein seeeeehr großer Button"));  
        add(new JButton("juhu"));  
        pack();  
        setVisible(true);  
    }  
    public static void main(String [] args) {  
        JFrame f = new JFrame();  
        new Beispiel34(f, "ganz einfach", new FlowLayout());  
        new Beispiel34(f, "rechtsbündig", new FlowLayout(FlowLayout.RIGHT));  
        new Beispiel34(f, "linksbündig und mehr Abstand",  
                      new FlowLayout(FlowLayout.LEFT, 20, 50));  
    }  
}
```

Berechnet die Mindestgröße,
damit alle Dialogelemente in
das Fenster passen.

GridLayout-Manager

- Der GridLayout-Manager in einem fest vorgegebenen Gitter (Tabelle) an.
- Alle Dialog-Elemente werden gemäß ihrer logischen Reihenfolge in der Tabelle abgelegt. Dabei wird durch die Zeilen von links nach rechts vorgegangen. Ist eine Zeile voll, wird die nächste unterhalb von links nach rechts belegt.
- Jede Zelle der Tabelle ist gleichgroß.
- Die Elemente werden auf die Zellengröße angepasst.

GridLayout-Manager (Fort.)

- Ist die Anzahl der Zellen nicht groß genug, werden die Zeilen verlängert (es werden mehr Spalten eingefügt). Es werden ***nicht mehr*** Zeilen eingefügt.
- Es gibt zwei Konstruktoren:

```
public GridLayout(int rows,int columns);
```

```
public GridLayout(int rows,int columns,int hgap,int vgap);
```

Go!

Beispiel für GridLayout-Manager

```
class Beispiel35 extends JPanel {  
    public Beispiel35(JFrame f, String title, LayoutManager mgr) {  
        super(tf, title);  
        setLayout(mgr);  
        add(new JButton("1"));  
        add(new JButton("Ein seeeeehr großer Button"));  
        add(new JButton("juhu"));  
        setLocation(200, 200);  
        pack();  
        setVisible(true);  
    }  
    public static void main(String [] args) {  
        JFrame f = new JFrame();  
        new Beispiel35(f, "ganz einfach", new GridLayout(2, 2));  
        new Beispiel35(f, "mit mehr Abstand",  
                      new GridLayout(2, 2, 20, 50));  
        new Beispiel35(f, "zu klein", new GridLayout(1, 2, 20, 50));  
    }  
}
```

BorderLayout-Manager

- Der BorderLayout-Manager kann bis zu fünf Dialogelemente anordnen.
- Die fünf Elemente werden in den vier Himmelsrichtungen und in der Mitte platziert.
- Die Platzierung findet ***unabhängig*** von der ***logischen Reihenfolge*** statt. Der Fokus erfolgt aber nach der ***logischen Reihenfolge***.
- Der Ort wird bei der add-Methode angegeben.

```
public Component add(String pos,Component comp);
```

- Als mögliche Position für pos kommen in Frage:
BorderLayout.NORTH, BorderLayout.SOUTH,
BorderLayout.EAST, BorderLayout.WEST, BorderLayout.CENTER



BorderLayout-Manager (Fort.)

- Wird eine falsche Position angegeben (z.B. "Süden"), wird eine Exception geworfen.
- Werden mehrere Elemente auf eine Position gelegt, wird nur das zuletzt eingetragene Element angezeigt.
- Es gibt zwei Konstruktoren:

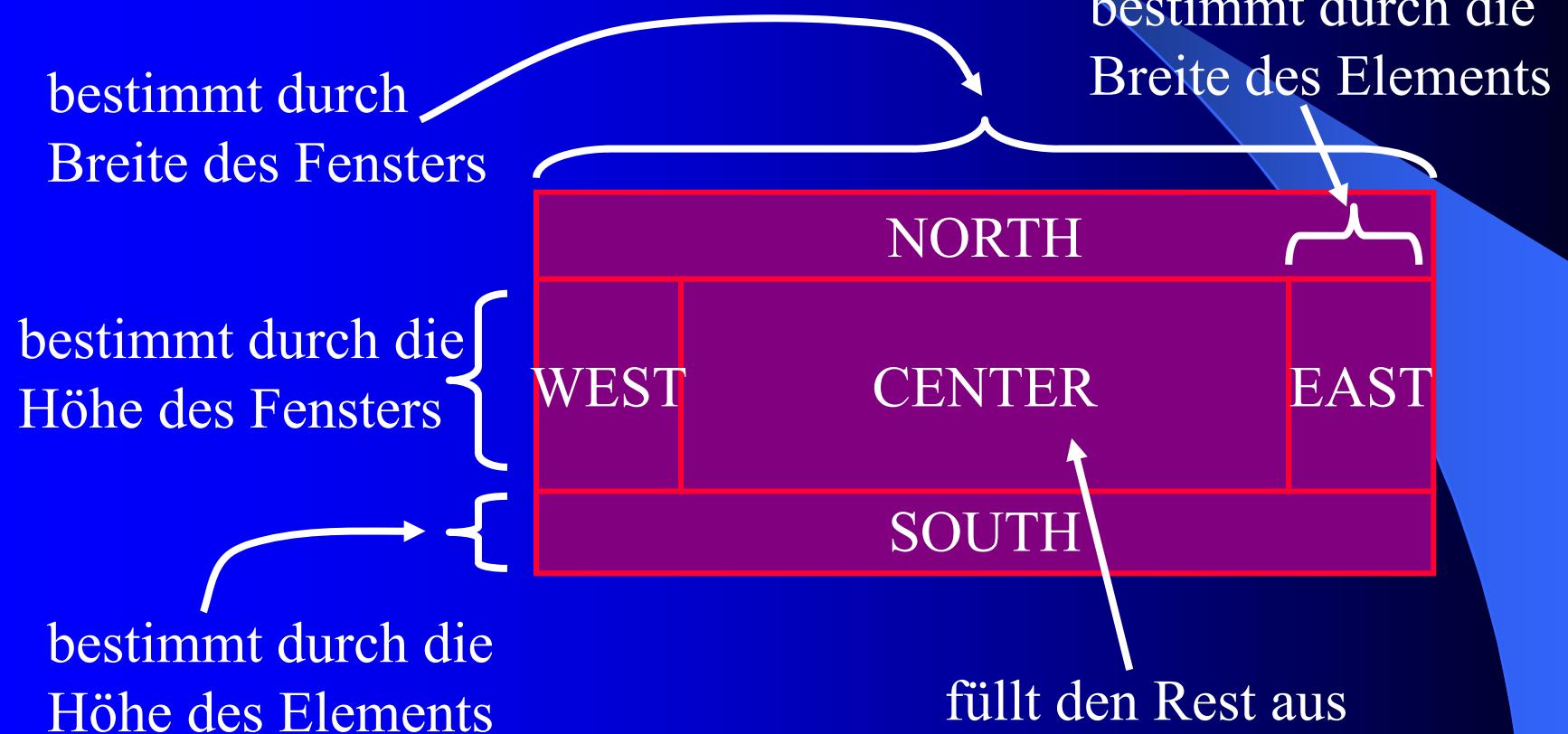
```
public BorderLayout();
```

```
public BorderLayout(int hgap,int vgap);
```

hgap bezeichnet den horizontalen Platz, vgap den vertikalen Platz in Pixel, der zwischen den Elementen freigelassen wird. Im Standardfall wird kein Platz gelassen, d.h. hgap = 0 und vgap = 0.

BorderLayout-Manager (Fort.)

- Wird die Größe des Fensters verändert, verändert sich auch die Größe der Dialogelemente.



Go!

Beispiel für BorderLayout-Manager

```
class Beispiel36 extends JPanel {  
    public Beispiel36(JFrame f, String title, LayoutManager mgr) {  
        super(f, title);  
        setLayout(mgr);  
        add(BorderLayout.NORTH, new JButton("1"));  
        add(BorderLayout.SOUTH, new JButton("Ein seeeeehr großer Button"));  
        add(BorderLayout.WEST, new JButton("Wat is?"));  
        add(BorderLayout.EAST, new JButton("juhu"));  
        add(BorderLayout.CENTER, new JButton("Mitte"));  
        pack();  
        setVisible(true);  
    }  
  
    public static void main(String [] args) {  
        JFrame f = new JFrame();  
        new Beispiel36(f, "ganz einfach", new BorderLayout());  
        new Beispiel36(f, "mit mehr Abstand", new BorderLayout(20, 50));  
    }  
}
```

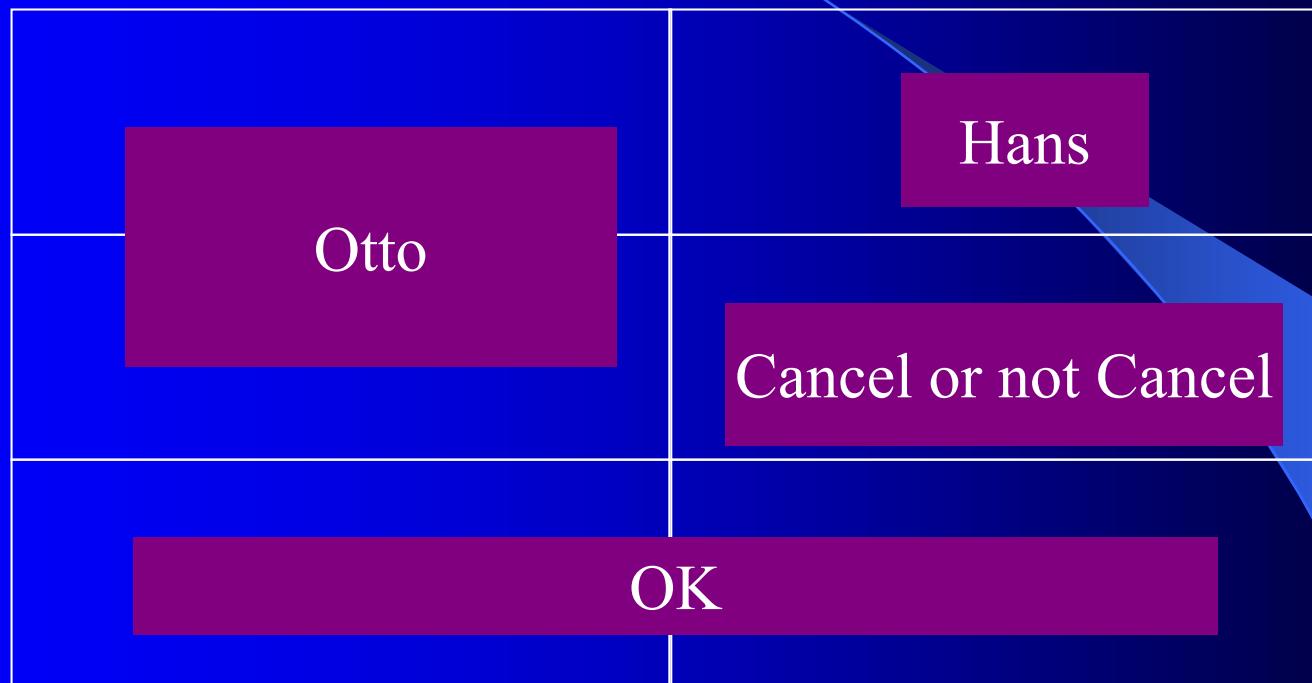
Vorlesung 8

Schachteln von Layout-Manager

- Die Klasse JPanel ist die einfachste konkrete Fensterklasse.
- Sie kann Dialogelemente aufnehmen und sie mittels eines Layout-Managers innerhalb ihrer Grenzen anordnen.
- Sie kann als ein Element einem höher geordnetem Layout-Manager übergeben werden.

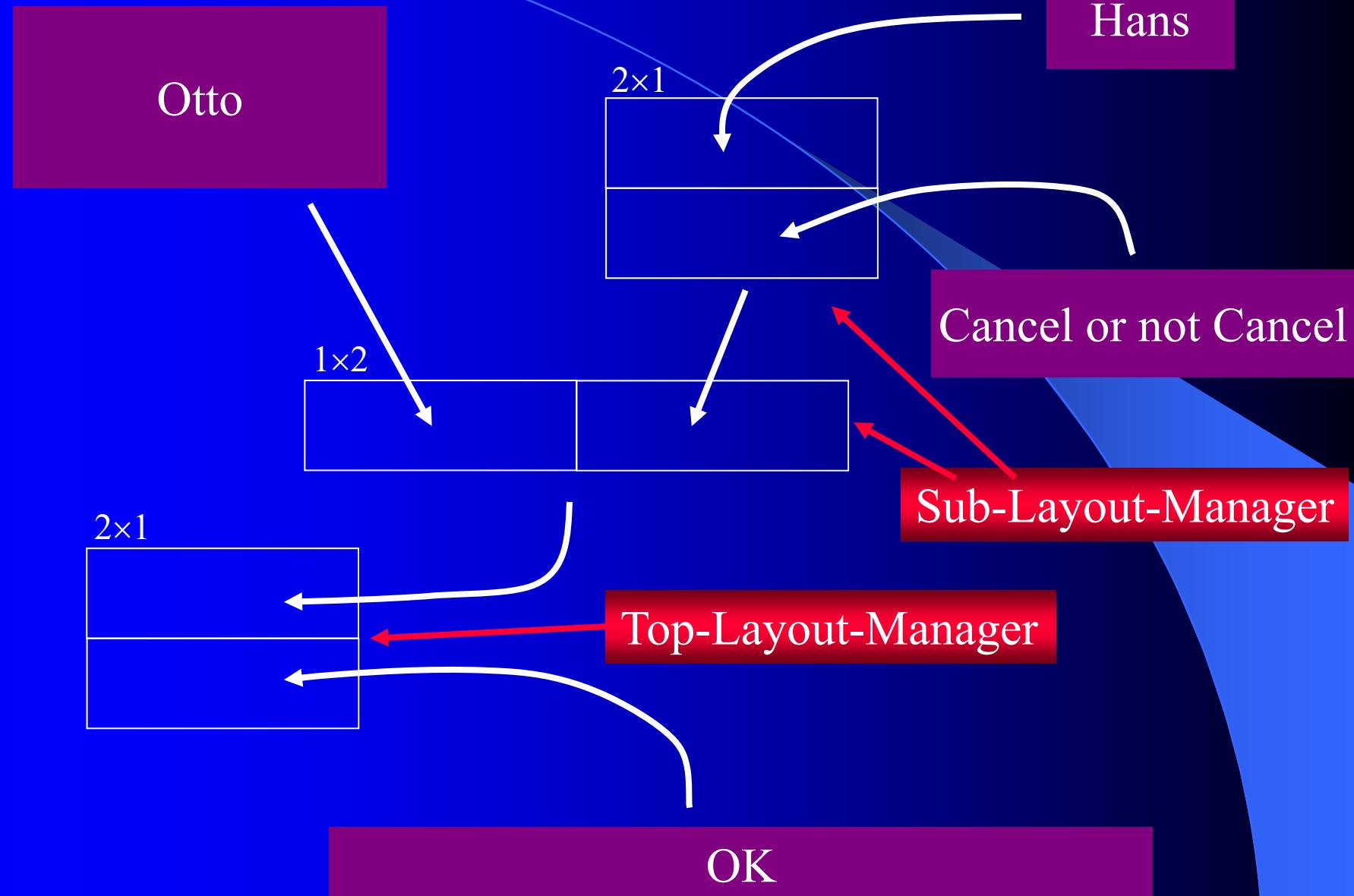
Beispiel (Fort.)

Ziel:



Lösung: Zerlegung in logische Unterstrukturen, d.h. Tabellen,
die in ihren Zellen wieder Tabellen beinhalten

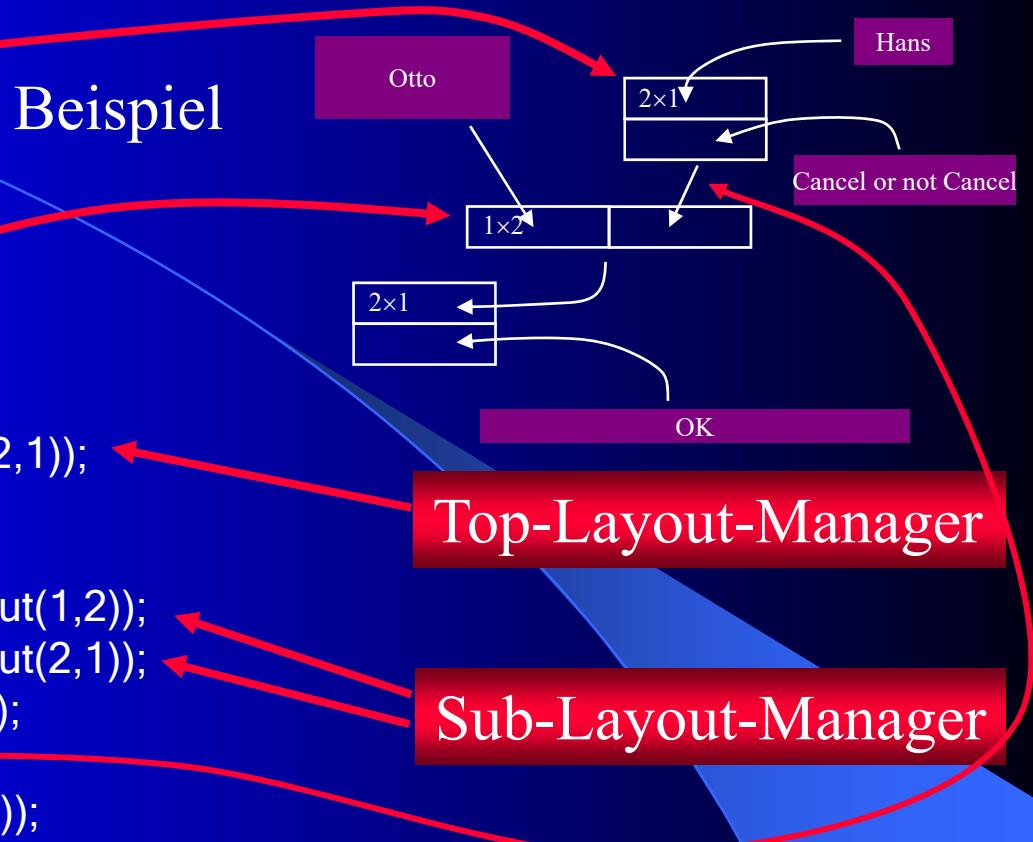
Beispiel (Fort.)



Go!

Beispiel

```
class Beispiel39 extends JDialog {  
  
    public Beispiel39(JFrame f) {  
        super(f);  
        setLayout(new GridLayout(2,1));  
        JPanel p1 = new JPanel();  
        JPanel p2 = new JPanel();  
        p1.setLayout(new GridLayout(1,2));  
        p2.setLayout(new GridLayout(2,1));  
        p1.add(new JButton("Otto"));  
        p1.add(p2);  
        p2.add(new JButton("Hans"));  
        p2.add(new JButton("Cancel or not Cancel"));  
        add(p1);  
        add(new JButton("OK"));  
        pack();  
        setVisible(true);  
    }  
    public static void main(String [] args) throws Exception {  
        new Beispiel39(new JFrame());  
    }  
}
```



Der null-Layout-Manager

- Der null-Layout-Manager ist kein Layout-Manager.
- Keines der Dialogelemente wird angeordnet.
- Durch das Setzen der Größe und der Position der Dialogelemente werden diese physikalisch im Fenster angeordnet.
- Es fehlt somit die abstrakte Schicht des Layouten.
- Diese Möglichkeit sollte nicht verwendet werden, da sie nicht portabel ist.

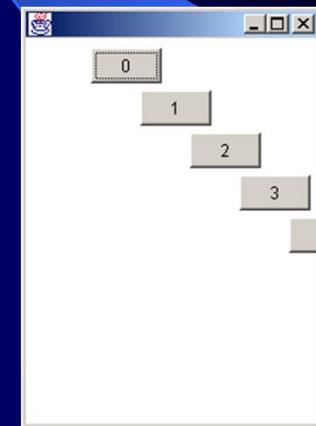
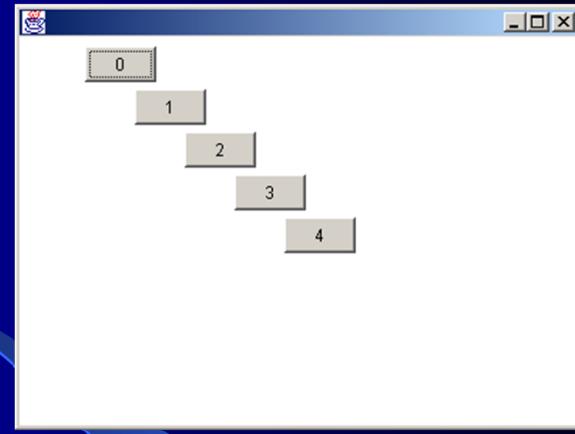
Go!

Beispiel

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

class Beispiel40 extends JDialog {
    public Beispiel40(JFrame f) {
        super(f);
        setSize(400,300);
        setLayout(null);
        for(int i = 0;i < 5;++i) {
            JButton b = new JButton(Integer.toString(i));
            b.setBounds(50+35*i,30+30*i,50,25);
            add(b);
        }
        setVisible(true);
    }

    public static void main(String [] args) throws Exception {
        new Beispiel40(new JFrame());
    }
}
```



Einige Methoden der JDialog Klasse

- Dialoge haben wie normale Frame-Fenster eine Titelzeile, die im Konstruktor oder später durch die Methode

```
public void setTitle(String title);
```

gesetzt werden kann.

- Modale Dialoge blockieren das Vaterfenster solange, wie der Dialog sichtbar ist.
- Im Konstruktor kann das Modal-Flag gesetzt werden. Oder es wird durch den Methoden-Aufruf

```
public void setModal(boolean modal);
```

gesetzt. Der Zustand kann durch

```
public boolean isModal();
```

abgefragt werden.

Einige Methoden der JDialog Klasse (Fort.)

- Sollen Dialogfenster in ihrer Größe nicht verändert werden können, kann durch

```
public void setResizable(boolean resizable);
```

dies unterbunden werden.

- Diese Methode ist auch für Frames verfügbar.
- Ob die Größe eines Fensters oder Dialogs veränderbar ist, kann mittels der Methode

```
public boolean isResizable();
```

abgefragt werden.

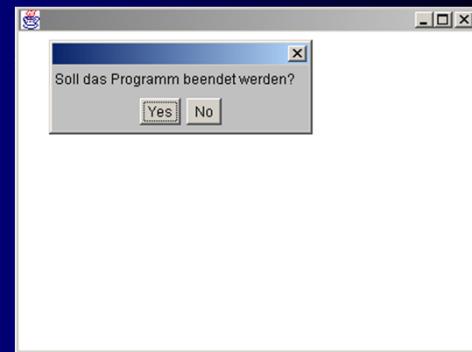
```
class YesNoDialog extends JPanel {    Beispiel
    public boolean m_bResult;
    public YesNoDialog(JFrame owner, String msg) {
        super(owner, "", true); setBackground(Color.lightGray);
        setLayout(new BorderLayout()); setResizable(false);
        Point p = owner.getLocation(); setLocation(p.x + 30, p.y + 30);
        add(BorderLayout.CENTER, new JLabel(msg));
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new FlowLayout(FlowLayout.CENTER));
        add(BorderLayout.SOUTH, buttonPanel); JButton b = new JButton("Yes");
        b.addActionListener(e -> {
            m_bResult = true;
            dispose();
        });
        buttonPanel.add(b); b = new JButton("No");
        b.addActionListener(e -> {
            m_bResult = false;
            dispose();
        });
        buttonPanel.add(b); pack(); setVisible(true);
    }
}
```

Go!

Beispiel (Fort.)

```
...
class Beispiel41 extends JFrame {
    public Beispiel41() {
        setSize(400,300);
        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                YesNoDialog dgl = new YesNoDialog(Beispiel41.this,
                        //((JFrame)e.getSource(),
                        "Soll das Programm beendet werden?");
                if (dgl.m_bResult)
                    dispose();
            }
        });
        setVisible(true);
    }

    public static void main(String [] args) throws Exception {
        new Beispiel41();
    }
}
```



Die JFileChooser Klasse

- Die Klasse **JFileChooser** in Swing dient zur Auswahl von Dateien.
- Sie bietet auch eine Filterfunktionalität an.

```
class JFileChooser {  
    public JFileChooser();  
    public JFileChooser(File currentDir);  
    ...  
    int showOpenDialog(Component father);  
    int showSaveDialog(Component father);  
    ...  
    void addChoosableFileFilter(FileFilter filter);  
    File getSelectedFile();  
}
```

Die JFileChooser Klasse (Forts.)

- **FileFilter** ist eine abstrakte Klasse, von der die konkrete Implementierung **FileNameExtensionFilter** existiert.

```
class FileNameExtensionFilter extends FileFilter {  
    public FileNameExtensionFilter( String description,  
                                    String ... extensions);  
  
    ...  
}
```

Go!

...

```
class Beispiel78 {  
  
    public static void main(String [] args) {  
        JFrame f = new JFrame();  
        f.setSize(200,200);  
        f.setVisible(true);  
  
        JFileChooser chooser = new JFileChooser(new File("."));  
        chooser.addChoosableFileFilter(new FileNameExtensionFilter("JPG & GIF Images",  
            "jpg", "gif"));  
        chooser.addChoosableFileFilter(new FileNameExtensionFilter("Docs",  
            "doc", "docx"));  
  
        if(chooser.showOpenDialog(f) == JFileChooser.APPROVE_OPTION)  
            System.out.println(" ausgewählte Datei : " +  
                chooser.getSelectedFile());  
  
        if(chooser.showSaveDialog(f) == JFileChooser.APPROVE_OPTION)  
            System.out.println(" ausgewählte Datei : " +  
                chooser.getSelectedFile());  
    }  
}
```

Beispiel

starte im aktuellen
Verzeichnis



2 Filter

"JPG & GIF Images",
"jpg", "gif");
"Docs",
"doc", "docx"));

war die
Auswahl
erfolgreich

die ausgewählte Datei

Fenster und Menus in Swing

Anzeige von
Meldungen
und Optionen

Aufbau von
Fenstern in
Fenstern (Multiple
Document Interface =
MDI)

Dient zum über-
lagern mehrerer
Schichten (z.B.
MenuPane und
ContentPane)

Swing	AWT
JFrame	Frame
JWindow	Window
JDialog	Dialog
JOptionPane	
JInternalFrame	
JApplet	Applet
JMenuBar	MenuBar
JMenu	Menu
JMenuItem	MenuItem
JPanel	Panel
JLayeredPane	

Die JOptionPane Klasse

Die JOptionPane Klasse enthält viele statische Methoden, um *Standarddialoge* zu erzeugen. Hier: anzeigen einer Nachricht

```
class JOptionPane {  
    public static void showMessageDialog(Component parent, Object msg);  
    public static void showMessageDialog(Component parent, Object msg,  
                                         String title, int messageType);  
    static final int ERROR_MESSAGE;  
    static final int INFORMATION_MESSAGE;  
    static final int WARNING_MESSAGE;  
    static final int QUESTION_MESSAGE;  
    static final int PLAIN_MESSAGE;  
    ...  
}
```

Go!

Beispiel

```
class Box extends JComponent {  
    Box() { setPreferredSize(new Dimension(100,100)); }  
    public void paintComponent(Graphics g) {  
        g.drawRect(0,0,getWidth()-1 getHeight()-1);  
    }  
}  
  
class Beispiel65 {  
    public static void main(String[] args) throws Exception {  
        JFrame f = new JFrame("ich mache nix");  
        f.setBounds(100,100,400,300);  
        f.setVisible(true);  
        JOptionPane.showMessageDialog(f,"juhu");  
        JOptionPane.showMessageDialog(f,"error","Jetzt kommt der Titel: Dr.",  
                                     JOptionPane.ERROR_MESSAGE);  
        JOptionPane.showMessageDialog(f,"INFO","Jetzt kommt der Titel: Dr.",  
                                     JOptionPane.INFORMATION_MESSAGE);  
        JOptionPane.showMessageDialog(f,new Box(),"Auch 'n Titel: von und zu",  
                                     JOptionPane.WARNING_MESSAGE);  
        JOptionPane.showInputDialog(f,"Frage?","Nix Title",  
                                   JOptionPane.QUESTION_MESSAGE);  
        JOptionPane.showMessageDialog(f,"langweilig","Hier ist nix",  
                                     JOptionPane.PLAIN_MESSAGE);  
    }  
}
```

Der Standard ist
eine Info-Message

beliebige
Component
Objekte sind
zugelassen

Die JOptionPane Klasse (Fort.)

Weitere Standarddialoge der JOptionPane Klasse, um Bestätigungsdialoge anzuzeigen. Auch wieder alle Modal

```
class JOptionPane {  
    public static int showConfirmDialog(Component parent, Object msg);  
    public static int showConfirmDialog(Component parent, Object msg,  
                                       String title, int optionType);  
    public static int showConfirmDialog(Component parent, Object msg,  
                                       String title, int optionType,  
                                       int messageType);  
  
    static final int YES_NO_OPTION;  
    static final int OK_CANCEL_OPTION;  
    static final int YES_NO_CANCEL_OPTION;  
    ...  
    static final int YES_OPTION; ←  
    static final int NO_OPTION;  
    static final int CANCEL_OPTION;  
    static final int OK_OPTION; ← // ist das gleiche wie YES_OPTION  
    static final int CLOSED_OPTION;  
}
```

Go!

```
...
class Beispiel66 {
    public static void pr(int msg) {
        System.out.print("Antwort: ");
        switch (msg) {
            case JOptionPane.YES_OPTION: System.out.println("yes");break;
            // statt JOptionPane.YES_OPTION auch JOptionPane.OK_OPTION
            case JOptionPane.NO_OPTION: System.out.println("no");break;
            case JOptionPane.CANCEL_OPTION: System.out.println("cancel");break;
            case JOptionPane.CLOSED_OPTION: System.out.println("closed");break;
        }
    }
    public static void main(String[] args) throws Exception {
        JFrame f = new JFrame("ich mache nix");
        f.setBounds(100,100,400,300);
        f.setVisible(true);
        pr( JOptionPane.showConfirmDialog(f,"juhu") );
        pr(JOptionPane.showConfirmDialog(f,"ja oder nein","","",
                                         JOptionPane.YES_NO_OPTION));
        pr(JOptionPane.showConfirmDialog(f, new Box(), "nix",
                                         JOptionPane.OK_CANCEL_OPTION));
        pr(JOptionPane.showConfirmDialog(f,"Frage?","Nix Title",
                                         JOptionPane.YES_NO_CANCEL_OPTION,
                                         JOptionPane.WARNING_MESSAGE));
    }
}
```

beliebige
Component
Objekte sind
zugelassen

Der Standard ist eine
Question-Message

Die JOptionPane Klasse (Fort.)

Mit den Standarddialogen der JOptionPane Klasse können auch einfache Daten eingegeben werden

```
class JOptionPane {  
    public static String showInputDialog(Component parent, Object msg);  
  
    public static String showInputDialog(Component parent, Object msg,  
                                         String title, int messageType);  
  
    public static Object showInputDialog(Component parent, Object msg,  
                                         String title, int messageType,  
                                         Icon icon,  
                                         Object[] selectionValues,  
                                         Object initialValue);  
  
    ...  
}
```

Go!

Beispiel

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

class Beispiel67 {

    public static String[] objs = {"juhu","doll","ich fass es nicht"};

    public static void main(String[] args) throws Exception {
        JFrame f = new JFrame("ich mach nix");
        f.setBounds(100,100,400,300);
        f.setVisible(true);
        System.out.println(JOptionPane.showInputDialog(f,"juhu"));

        System.out.println(JOptionPane.showInputDialog(f, new Box(), "", 
                JOptionPane.WARNING_MESSAGE));

        System.out.println(JOptionPane.showInputDialog(f, new Box(), "", 
                JOptionPane.PLAIN_MESSAGE,
                null,objs,objs[2]));
    }
}
```

Der Standard ist eine
Question-Message

beliebige Component
Objekte sind zugelassen

Vorlesung 9

Einfache Dialogelemente in Swing

Swing
JButton
JLabel
JTextField
JPasswordField
JFormattedTextField
JTextArea
JCheckBox
JRadioButton mit ButtonGroup
JList
JComboBox
JSpinner
JScrollBar
JSlider
JProgressBar

JLabel

Ein JLabel dient zur Beschriftung von Dialogboxen.

```
class JLabel {  
    public JLabel();  
    public JLabel(String text);  
    public JLabel(String text,int align);  
    public void setText(String text);  
    public String getText();  
    public void setHorizontalAlignment (int align);  
    ...  
}
```



Go!

Beispiel

```
class Beispiel42 extends JDialog {  
    public Beispiel42() {  
        super(new JFrame());setLayout(new GridLayout(4,1));  
        add(new JLabel("Default"));  
        final JLabel l = new JLabel("Links",SwingConstants.LEFT);  
        add(l);  
        l.addMouseListener(new MouseAdapter() {  
            @Override  
            public void mouseEntered(MouseEvent e) {  
                l.setText("RECHTS");l.setHorizontalTextPosition(SwingConstants.RIGHT);  
            }  
            @Override  
            public void mouseExited(MouseEvent e) {  
                l.setText("Links");l.setHorizontalTextPosition(SwingConstants.LEFT);  
            }  
        });  
        add(new JLabel("Rechts",SwingConstants.RIGHT));  
        add(new JLabel("In der Mitte",SwingConstants.CENTER));  
        pack();  
        setVisible(true);  
    }  
    public static void main(String [] args) throws Exception {  
        new Beispiel42();  
    }  
}
```



Die JPasswordField Klasse

- Die JPasswordField Klasse ist eine Spezialisierung der JTextField Klasse.
- Sie dient zur Eingabe von Passwörtern
- Der eingegeben Text wird nicht direkt angezeigt, sondern alle Zeichen werden durch ein besonderes Zeichen ersetzt.
- Dieses Zeichen kann durch `setEchoChar` geändert werden.
- Der eingegebene Text kann durch `getPassword` abgefragt werden.
- ...

Die JPasswordField Klasse (Fort.)

- ...
- Die Copy- und Cut-Methoden sind ausgeschaltet.
- Die return-Taste löst ein ActionEvent aus.

```
class JPasswordField extends JTextField {  
    public void setEchoChar(char);  
    public char[] getPassword();  
    ...  
}
```

Go!

Beispiel

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

class Beispiel69 extends JFrame {

    public Beispiel69() {
        final JPasswordField field = new JPasswordField();
        field.addActionListener(e -> System.out.println(field.getPassword()));
        add(field);
        pack();
        setVisible(true);
    }

    public static void main(String[] args) {
        new Beispiel69();
    }
}
```

Der ActionListener reagiert
auf die return-Taste

Die JFormattedTextField Klasse

- Die JFormattedTextField Klasse ist von JTextField abgeleitet.
- Sie dient dazu, *formatierten* Text einzugeben und darzustellen.
- Bereits bei der Eingabe wird überprüft, ob der eingegebene Text der Formatierung entspricht
- Sehr sinnvoll für Eingabemasken:
 - Eingabe von Zahlen: keine Buchstaben zulassen
 - Eingabe von Buchstaben: keine Zahlen zulassen
 - Eingabe eines Datums/IP-Adresse: nur Zahlen in einem bestimmten Format zulassen

```
class JFormattedTextField extends JTextField {
```

```
    public JFormattedTextField();
```

```
    public JFormattedTextField(JFormattedTextField.AbstractFormatter);
```

```
    ...
```

abstrakte Klasse:
legt das Format fest

Go!

Beispiel

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

class Beispiel69_2 extends JFrame {

    public Beispiel69_2() {
        final JFormattedTextField text = new JFormattedTextField();
        add(text);
        text.addActionListener(e -> System.out.println(text.getText()));
        pack();
        setVisible(true);
    }

    public static void main(String[] args) {
        new Beispiel69_2();
    }
}
```

Funktioniert wie die
normale JTextField Klasse

Auch das Eventhandling
ist normal

Die JFormattedTextField Klasse (Forts.)

- Ein Format kann über den Parameter vom Typ **JFormattedTextField.AbstractFormatter** angegeben werden
- Die Klasse **MaskFormatter** ist eine mögliche Implementierung der abstrakten **AbstractFormatter** Klasse
- Im Konstruktor wird ein String angegeben, der die Maske beschreibt
- Der String "juhu" würde nur das Wort "juhu" zulassen

nicht gerade
sinnvoll

```
class MaskFormatter extends AbstractFormatter {  
    public MaskFormatter(String) throws ParseError;
```

Die JFormattedTextField Klasse (Forts.)

- In dem String sind folgende "Sonderwerte" erlaubt:

#	Beliebige Ziffer 0, 1, ..., 9
'	Escape Zeichen zur Darstellung dieser Metazeichen
U	Beliebiger Buchstabe: wird in Großbuchstabe umgeformt
L	Beliebiger Buchstabe: wird in einen Kleinbuchstaben umgeformt
A	Buchstabe oder Ziffer
?	Irgendein Zeichen
*	Irgend etwas.
H	Ein Hexadezimalzeichen: 0, 1, ..., 9, A, B, C, D, E, F

- Beispiel: new MaskFormatter("##-##-#####")

mögliche Eingabe:
23-45-2334

Die JFormattedTextField Klasse (Forts.)

- die "Leerstellen" können auch angezeigt werden
- die Methode **setPlaceholderCharacter(char placeholder)** spezifiziert das Zeichen, das für die Platzhalter angezeigt werden soll
- mit **setValidCharacters(String)** können die einzugebenen Buchstaben noch weiter eingeschränkt werden
- Beispiel:

```
MaskFormatter mask =
```

```
    new MaskFormatter("##-UU-#####'U");  
    mask.setPlaceholderCharacter('_');  
    mask.setValidCharacters("ABC123");
```

mögliche Eingabe:
23-BC-2331

Am Ende steht
automatisch ein 'U'

Die Document Klasse

- ein Problem mit dem **MaskFormatter** ist, dass das Format eine feste Anzahl von Eingabezeichen vorschreibt
- dies ist oft zu restriktiv
- soll bei der Eingabe z.B. überprüft werden, ob eine Zahl eingegeben wird,
 - 34536
 - 67
- so kann dazu kein **MaskFormatter** verwendet werden, da die Zahl ein beliebige Länge hat
- jedoch kann ein **Document** Objekt im **JTextField** (und damit auch im **JFormattedTextField**) mittels der Methode **setDocument** registriert werden
 - ...

Die Document Klasse (Forts.)

- ...
- die Document Klasse ist abstrakt
- eine mögliche konkrete Implementierung stellt die Klasse **PlainDocument** dar
- die Methode
`void insertString(int offs, String str, AttributeSet a)`
wird aufgerufen, sobald Text in das JTextField eingetragen wird
- durch das Überlagern dieser Methode kann kontrolliert werden, was in das Textfeld eingetragen wird

Go!

Beispiel

```
class MyDocument extends PlainDocument {  
    public void insertString(int offs, String str, AttributeSet a) throws BadLocationException {  
        try {  
            Integer.parseInt(str);  
        } catch(Exception ex) {  
            Toolkit.getDefaultToolkit().beep();  
            return;  
        }  
        super.insertString(offs,str, a);  
    }  
}
```

ist der bisher eingegebene Text eine Zahl?

Wenn alles ok ist,
Text einfügen

```
class Beispiel69_6 extends JFrame {  
    public Beispiel69_6() {  
        final JTextField text = new JTextField();  
        text.setDocument(new MyDocument());  
        add(text);  
        text.addActionListener(e -> System.out.println(text.getText()));  
        pack();  
        setVisible(true);  
    }  
}
```

Assoziieren des Dokuments
mit dem Textfeld

Die JSlider Klasse

- Die JSlider Klasse ist analog zu der JScrollPane Klasse ein Dialogelement zur quasi analogen Eingabe.
- Ein JSlider hat jedoch eine Anzeigeskala mit grober und feiner Einteilung.
- Ein JSlider hat keine unterschiedliche Schiebergröße, sondern hat immer eine Ausdehnung von 1.

```
class JSlider {  
    public JSlider(int orientation,int min,int max,int value);  
    public int getValue();  
    public int getMinimum();  
    public int getMaximum();  
    final static int VERTICAL;  
    final static int HORIZONTAL;  
    ...  
}
```

Die JSlider Klasse (Fort.)

- Die Breite der groben Einteilung wird mit `setMajorTickSpacing` eingestellt.
- `setMinorTickSpacing` stellt die Breite für die feine Einteilung ein.
- Damit die Ticks und Labels sichtbar werden, müssen sie zusätzlich mittels der Methoden `setPaintTicks` und `setPaintLabels` auf den Bildschirm gebracht werden.

```
class JSlider {  
    ...  
    public void setMajorTickSpacing(int n);  
    public void setMinorTickSpacing(int n);  
    public void setPaintTicks(boolean b);  
    public void setPaintLabels(boolean b);  
}
```

Die JSlider Klasse (Fort.)

- Durch den Aufruf von `setSnapToTicks(true)` wird erreicht, dass der Slider nur noch bei den Ticks einrasten kann.
- Im Gegensatz zu einem Scrollbar wird bei der Veränderung eines Sliders nicht ein `AdjustmentEvent` ausgelöst, sondern ein `ChangeEvent`. Dieses `ChangeEvent` ist aus `javax.swing.event.*` zu importieren.

```
class JSlider {  
    ...  
    public void setSnapToTicks(boolean b);  
    public void addChangeListener(ChangeListener ch);  
}
```

```
interface ChangeListener {  
    public void stateChanged(ChangeEvent e);  
}
```

Die JSlider Klasse (Fort.)

- Die JSlider Klasse bietet zwei Möglichkeiten, den internen Wertezustand abzufragen.
- `getValue` liefert den aktuell eingestellten Wert zurück. Während des Einstellens wird ständig der jeweils aktuelle Wert zurückgeliefert.
- Die Methode `getValuesAdjusting` liefert den Wert `false` zurück, wenn die Änderung abgeschlossen sind, ansonsten `true`.

```
class JSlider {  
    ...  
    public int getValue();  
    public boolean getValuesAdjusting();  
}
```

```

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

interface Doit {
    public void doit(boolean b);
}

class MyButton extends JButton {
    boolean m_bVal;
    String m_Label;
    Doit m_Fct;

    public MyButton(String label,Doit doit) {
        super("set " + label);
        m_Label = label;
        m_Fct = doit;
        m_bVal = true;
        addActionListener(e -> {
            m_Fct.doit(m_bVal);
            setText((m_bVal ? "remove " : "set ") + m_Label);
            m_bVal = !m_bVal;
        });
    }
}

```

Beispiel

Wird später für
ChangeEvent benötigt

Interface zur Abstraktion der Aktion

Button merkt sich seine Aktion

- abstrakte Aktion wird ausgeführt
- Label wird umgesetzt
- m_bVal merkt sich dies

Go!

```
...  
class Beispiel70 extends JFrame {  
    public Beispiel70() {  
        final JSlider SLIDER = new JSlider(JSlider.VERTICAL, 10, 100, 15);  
        SLIDER.setMajorTickSpacing(25);  
        SLIDER.setMinorTickSpacing(5);  
        setLayout(new GridLayout(2,2));  
        add(SLIDER);  
        add(new MyButton("ticks",  
        add(new MyButton("label",  
        add(new MyButton("snap2ticks",
```

Beispiel

Ein Slider wird erzeugt, die Grob- und Feineinteilung wird gesetzt

b -> SLIDER.setPaintTicks(b));
b -> SLIDER.setPaintLabels(b));
b -> SLIDER.setSnapToTicks(b));

Was ist
das hier?

```
SLIDER.addChangeListener(e -> {  
    System.out.println(SLIDER.getValue());  
    if (!SLIDER.getValueIsAdjusting())  
        System.out.println("... that's it");  
}  
);  
pack();  
setVisible(true);  
}  
public static void main(String[] args) throws Exception {  
    new Beispiel70();  
}
```

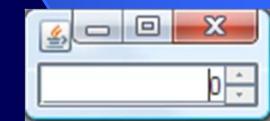
nicht addAdjustmentListener !

Der Slider reagiert
auf Veränderungen

Die JSpinner Klasse

- Die JSpinner Klasse dient zur Auswahl aus einer Liste (ähnlich zu einer Choice).
- Änderungen lösen ein ChangeEvent aus.
- `getValue()` liefert das selektierte Objekt vom Typ Object (!!!) zurück
- Standardmäßig werden Integer Objekte zurückgeliefert (aber als Typ Object)

```
JFrame frame = new JFrame("JSpinner");
final JSpinner spin = new JSpinner();
spin.addChangeListener(e -> System.out.println(spin.getValue()));
frame.add(spin);
frame.pack();
frame.setVisible(true);
```



liefer Object zurück,
ist in Standardfall
aber ein Integer Objekt

Die JSpinner Klasse (Fort.)

- Dem Konstruktor kann ein Objekt, dass das SpinnerModel Interface implementiert übergeben werden
- das SpinnerModel koordiniert, welche Objekte selektiert werden können
- das SpinnerListModel ist eine bereits definierte Implementierung, die Arrays von Objekten an den JSpinner übergibt

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

class MyColor {
    private String mName;
    Color mColor;
    MyColor(String name,Color col) {
        mName = name;  mColor = col;
    }
    public String toString() {      return mName; }
}
```

merkt sich eine Farbe
und dessen Namen

Go!

Die JSpinner Klasse (Fort.)

```
public class Beispiel80 {  
    public static void main(String[] args) throws Exception {  
        MyColor[] cols = { new MyColor("rot",Color.RED),  
                           new MyColor("grün",Color.GREEN),  
                           new MyColor("blau",Color.BLUE)};  
        SpinnerListModel model = new SpinnerListModel(cols);  
        JFrame frame = new JFrame("JSpinner");  
        final JSpinner spin = new JSpinner(model);  
        final JComponent c = new JComponent() {  
            public void paintComponent(Graphics g) {  
                g.drawLine(0,0getWidth(),getHeight());  
            }  
        };  
        spin.addChangeListener(e -> {  
            MyColor col = (MyColor)spin.getValue();  
            c.setForeground(col.mColor);  
            c.repaint();  
        }  
    );  
        frame.setLayout(new GridLayout(2,1));  
        frame.add(spin);frame.add(c);frame.pack();frame.setVisible(true);  
    }  
}
```

ein einfaches Spinner-ListModel und seine Verwendung

eine Komponente, die eine Linie zeichnet

Go!

Ähnlicher Effekt mit JComboBox

```
public class Beispiel80_1 {  
    public static void main(String[] args) throws Exception {  
        MyColor[] cols = { new MyColor("rot",Color.RED),  
                           new MyColor("grün",Color.GREEN),  
                           new MyColor("blau",Color.BLUE)};  
        JFrame frame = new JFrame("JComboBox");  
        final JComboBox<MyColor> spin = new JComboBox<MyColor>(cols);  
        final JComponent c = new JComponent() {  
            public void paintComponent(Graphics g) {  
                g.drawLine(0,0getWidth(),getHeight());  
            }  
        };  
        spin.addActionListener(e -> {  
            MyColor col = (MyColor)spin.getSelectedItem();  
            c.setForeground(col.mColor);  
            c.repaint();  
        }  
    );  
        frame.setLayout(new GridLayout(2,1));  
        frame.add(spin);frame.add(c);frame.pack();frame.setVisible(true);  
    }  
}
```

JComboBox braucht
keine SpinnerModel,
ist aber ein Generic ...

... und reagiert auf den
ActionListener

Die JProgressBar Klasse

- Die JProgressBar Klasse dient zur Anzeige eines Fortschritts.
- Der Fortschritt wird immer in Prozent angezeigt.

```
class JProgressBar {  
    public JProgressBar();  
    public JProgressBar(int orient);  
    public JProgressBar(int min, int max);  
    public JProgressBar(int orient, int min, int max);  
  
    public int getValue();  
    public int getMinimum();  
    public int getMaximum();  
    public void setValue(int);  
  
    final static int VERTICAL;  
    final static int HORIZONTAL;  
    ...  
}
```



Die JProgressBar Klasse (Fort.)

- Mittels der `setStringPainted` Methode kann spezifiziert werden, ob die Anzeige auch noch zusätzlich textuell dargestellt werden soll.

```
class JProgressBar {  
    ...  
    public void setStringPainted(boolean b);  
    ...  
}
```



Go!

Beispiel

```
...
class Beispiel71 extends JFrame {
    public Beispiel71() {
        final JProgressBar progress = new JProgressBar(JProgressBar.VERTICAL,10,300);
        progress.setStringPainted(true);
        setLayout(new FlowLayout());
        JButton next = new JButton("more");
        add(progress);
        add(next);
        next.addActionListener(e -> {
            progress.setValue((progress.getValue() + 5) % progress.getMaximum());
        });
        pack();
        setVisible(true);
    }

    public static void main(String[] args) {
        new Beispiel71();
    }
}
```

Vertikaler ProgressBar
zwischen 10 und 300

Bei Knopfdruck wird
der Fortschritt um 5
Einheiten (NICHT
PROZENT) erhöht

Vorlesung 10

Bitmaps

Bitmaps werden in zwei Schritten auf den Bildschirm gebracht

1. laden des Bildes vom Rechner oder aus dem Netz (oder selber erstellen)
2. malen des Bildes in einen Graphikkontext mittels der verschiedenen `drawImage` Methoden von `Graphics`

Bilder werden durch die Klasse `Image` implementiert.

Laden von Bitmaps

Die Klasse Toolkit bietet mehrere Methoden an, um Bilder einzuladen.

```
class Toolkit {  
    ...  
    public abstract Image getImage(String filename);  
    public abstract Image getImage(URL url);  
    public abstract Image createImage(String filename);  
    public abstract Image createImage(URL url);  
}
```

- `getImage(String filename)` und `createImage(String filename)` laden Bitmaps vom Rechner ein
- `getImage(URL url)` und `createImage(URL url)` laden Bitmaps über das Netz ein

Laden von Bitmaps (Fort.)

getImage speichert Bitmaps zwischen

```
Image i1 = getToolkit().getImage("juhu");
```

```
...
```

```
Image i2 = getToolkit().getImage("juhu");
```

- i1 und i2 werden dasselbe Bild enthalten
- beim 2. Aufruf von getImage("juhu") wird geschaut, ob das Bild zuvor eingeladen wurde
- wenn ja, wird das verwendet
- zwischenzeitliche Änderung an dem Bild **werden nicht** vom Programm **wahrgenommen**

Laden von Bitmaps (Fort.)

`createImage` lädt Bitmaps bei jedem Aufruf neu ein

```
Image i1 = getToolkit().createImage("juhu");
```

```
...
```

```
Image i2 = getToolkit().createImage("juhu");
```

- `i1` und `i2` werden ***nicht*** das selbe Bild enthalten
- beim 2. Aufruf von `createImage("juhu")` wird das Bild erneut aus dem Dateisystem geladen, egal ob das Bild schon zuvor geladen wurde
- zwischenzeitliche Änderung an dem Bild ***werden*** vom Programm ***wahrgenommen***

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

```
interface Loader {  
    public Image load();  
}
```

```
class Beispiel54 extends JFrame {  
    Image img1; Image img2;  
    public Beispiel54(final Loader l, String title) {  
        super(title);  
        setBounds(100, 100, 200, 200);  
        img1 = l.load();  
        addKeyListener(new KeyAdapter() {  
            public void keyPressed(KeyEvent e) {  
                img2 = l.load();  
                repaint();  
            }  
        });  
    }  
    ...
```

Beispiel

Ein Interface, um die Klasse **Beispiel54** unabhängig von der Lademethode zu machen

Das 1. Bild wird sofort geladen

Das 2. Bild wird erst nach einem Tastendruck geladen

Go!

Beispiel

```
...
    add(new JComponent() {
        @Override
        public void paintComponent(Graphics g) {
            g.drawImage(img1,0,0,this);
            if (img2 != null)
                g.drawImage( img2,img1.getWidth(this),
                            img1.getHeight(this),this);
        }
    });
    setVisible(true);
}

public static void main(String [] args) {
    Toolkit t = Toolkit.getDefaultToolkit();
    new Beispiel54(() -> t.getImage("weissschatten_dreh.gif"), "mit getImage");
    new Beispiel54(() -> t.createImage("weissschatten_dreh.gif"), "mit createImage");
}
```

Frage: was ist das hier?

Laden von Bitmaps: ein genauer Blick

- Die Methoden zum Laden von Bildern **blockieren** das Programm **nicht**
- Sie starten das Laden und geben die Programmkontrolle sofort zurück
- Greift man sofort im Anschluss auf das Bild zu, ist es i.d.R. noch nicht vollständig geladen
- Will man warten, bis das Bild eingeladen ist, kann die Klasse MediaTracker helfen

MediaTracker

```
class MediaTracker {  
    public MediaTracker(Component client);  
    public void addImage(Image image, int id);  
    public boolean checkAll();  
    public void waitForAll() throws InterruptedException;  
    ...  
}
```

- `addImage` fügt das Bild `image` unter dem Namen `id` dem `MediaTracker` hinzu
- `checkAll` sagt, ob all Bilder geladen sind (`true`) oder nicht (`false`)
- `waitForAll` wartet solange, bis alle Bilder geladen sind

Go!

Beispiel

```
import javax.swing.*;
import java.awt.*;
class Beispiel56 extends JFrame {
    Image img;
    public Beispiel56(boolean bWait) throws Exception {
        super(bWait ? "mit warten" : "ohne warten");
        img = getToolkit().createImage("SURF.JPG");
        if (bWait) {
            MediaTracker mt = new MediaTracker(this);
            mt.addImage(img,1);
            mt.waitForAll();
        }
        add(new JComponent() {
            public void paintComponent(Graphics g) {
                g.drawImage(img,0,0getWidth(),getHeight(),this);
            }
        });
        setBounds(50,50,img.getWidth(this),img.getHeight(this));
        setVisible(true);
    }
    public static void main(String[] args) throws Exception {
        new Beispiel56(true);
        new Beispiel56(false);
    }
}
```

bWait kontrolliert, ob gewartet werden soll, bis das Bild geladen ist

Hier wird auf Bilddaten zugegriffen

Animation

- Unter ***Animation*** wird im folgenden die Darstellung ***bewegter Bilder*** verstanden.
- Animation wird erreicht durch die ***schnelle aufeinander-folgende*** Darstellung ***einzelner Bilder***.
- Durch die ***Trägheit des menschlichen Auges*** wird diese sequentielle Folge als Bewegung wahrgenommen, wenn die ***Änderungen von einem Bild zum nächsten gering*** sind.

Frage:

Wird die Kontrolle der verschiedenen Bilder in die `paintComponent`-Routine eingebaut?

Animation (Fort.)

Antwort: **NEIN**

Problem: wird die Kontrolle in die `paintComponent`-Routine eingebaut,

- kann die Anwendung nicht mehr auf Benutzer-Events reagieren,
- fängt die Animation immer wieder von vorne an, sobald die `paintComponent`-Routine aufgerufen wird, weil z.B. ein Teil des Fensters verdeckt war und wieder angezeigt wird.

Lösung:

- die Kontrolle muss in eine andere Routine eingebaut werden, die dann die `repaint`-Methode aufruft (`repaint` ist eine Methode aus `JComponent`)
- die `repaint`-Methode ruft dann die `paintComponent`-Routine auf

Beispiel

```
import javax.swing.*;  
import java.awt.*;  
import java.awt.event.*;  
  
class Beispiel57 extends JFrame {  
    Image m_Img;  
    int m_iHeight = 0;    WICHTIG !!!  
    int m_iSleeper;  
    volatile int m_iMaxHeight = 0;  
  
    public Beispiel57(int iSleeper) throws Exception {  
        m_iSleeper = iSleeper;  
        m_Img = getToolkit().createImage("weissschatten.gif");  
        MediaTracker mt = new MediaTracker(this);  
        mt.addImage(m_Img, 1);  
        mt.waitForAll();  
        setBounds(50, 50, m_Img.getWidth(this)*2, m_Img.getHeight(this)*10);  
    }  
}
```

Lädt ein Bild ein und wartet den Vorgang ab

...

Beispiel (Forts.)

```
...  
    addComponentListener(new ComponentAdapter(){  
        public void componentResized(ComponentEvent e) {  
            m_iMaxHeight = 0;  
        }  
    });  
    add(new JComponent() {  
        @Override  
        public void paintComponent(Graphics g) {  
            if (m_iMaxHeight == 0)  
                m_iMaxHeight = getHeight() - m_Img.getHeight(this);  
            g.drawImage(m_Img, 0, m_iHeight, this);  
            g.drawImage(m_Img, m_Img.getWidth(this), m_iMaxHeight-m_iHeight, this);  
        }  
    });  
    getContentPane().setBackground(Color.white);  
    setVisible(true);  
}  
...
```

Ändert sich die Größe des Fensters,
wird die Maximalhöhe zurückgesetzt

malt 2×
das Bild

Go!

Beispiel (Forts.)

...

```
public void anime() throws Exception {  
    while (true) {  
        for(m_iHeight = m_MaxHeight;m_iHeight > 0;m_iHeight-=1) {  
            Thread.sleep(m_Sleeper);  
            repaint();  
        }  
        for(m_iHeight = 0;m_iHeight < m_MaxHeight;m_iHeight+=1) {  
            Thread.sleep(m_Sleeper);  
            repaint();  
        }  
    }  
}
```

m_iHeight kontrolliert
die Sprunghöhe

Nachdem eine neue Sprunghöhe
berechnet worden ist, wird das
Fenster neu gezeichnet

```
public static void main(String[] args) throws Exception {  
    Beispiel57 b = new Beispiel57(2);  
    b.anime();  
}  
}
```

Go!

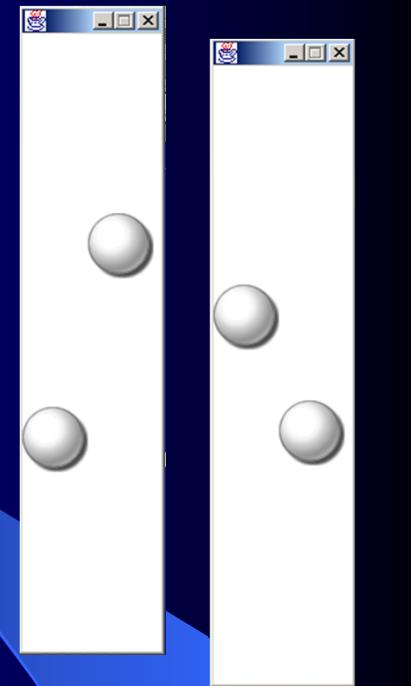
Nebenläufigkeit

Aufgabe: zwei Fenster öffnen, in denen die Bälle auf- und abspringen

Lösungsansatz: zwei Objekte der Klasse Beispiel57 kreieren und die `anime`-Methode aufrufen

```
import java.awt.*;
class Beispiel57_5 extends Frame {
    public static void main(String[] args) throws Exception {
        Beispiel57 b1 = new Beispiel57(2);
        Beispiel57 b2 = new Beispiel57(2);
        b1.anime();
        b2.anime();
    }
}
```

Frage: was wird passieren?



Nebenläufigkeit (Fort.)

```
b1.anime();  
b2.anime();
```

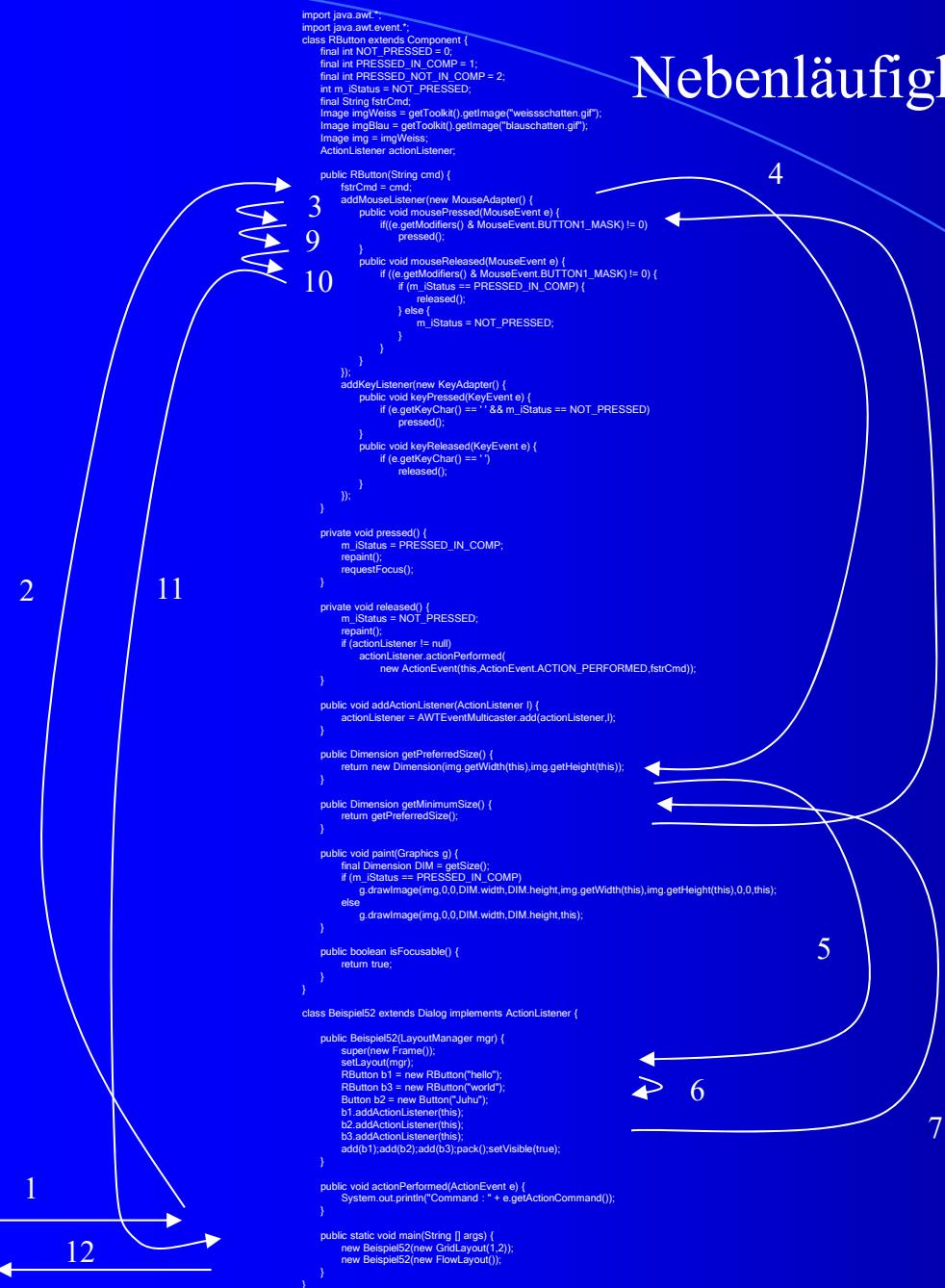
```
}
```

Problem:

- die **anime**-Methode enthält eine Endlosschleife, aus der das Programm nicht mehr zurückkommt,
- somit wird der zweite Aufruf der **anime**-Methode niemals ausgeführt,
- der Programmablauf kann sich nicht "teilen", um zwei Dinge gleichzeitig zu tun

Lösung:

Programmablauf "teilen", um mehrere Sachen gleichzeitig zu tun.



Nebenläufigkeit (Fort.)

'normaler', sprich sequentieller Programmablauf

- alle Schritte kommen hintereinander
- ein neuer Schritt fängt erst an, wenn der alte beendet ist
- alle Schritte können durchgezählt werden

```

import java.awt.*;
import java.awt.event.*;
class RButton extends Component {
    final int NOT_PRESSED = 0;
    final int PRESSED_IN_COMP = 1;
    final int PRESSED_NOT_IN_COMP = 2;
    int m_iStatus = NOT_PRESSED;
    final String fstrCmd;
    Image imgWeiss = Toolkit.getDefaultToolkit().getImage("weisschatten.gif");
    Image imgBlau = Toolkit.getDefaultToolkit().getImage("blauschatten.gif");
    Image imgGrey = Toolkit.getDefaultToolkit().getImage("grau.gif");
    ActionListener actionListener;
    public RButton(String cmd) {
        fstrCmd = cmd;
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                if((e.getModifiers() & MouseEvent.BUTTON1_MASK) != 0)
                    pressed();
            }
            public void mouseReleased(MouseEvent e) {
                if((e.getModifiers() & MouseEvent.BUTTON1_MASK) != 0) {
                    if(m_iStatus == PRESSED_IN_COMP)
                        released();
                    else
                        m_iStatus = NOT_PRESSED;
                }
            }
        });
        addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                if(e.getKeyChar() == ' ' && m_iStatus == NOT_PRESSED)
                    pressed();
            }
            public void keyReleased(KeyEvent e) {
                if(e.getKeyChar() == ' ')
                    released();
            }
        });
        private void pressed() {
            m_iStatus = PRESSED_IN_COMP;
            repaint();
            requestFocus();
        }
        private void released() {
            m_iStatus = NOT_PRESSED;
            repaint();
            if(actionListener != null)
                actionListener.actionPerformed(
                    new ActionEvent(this,ActionEvent.ACTION_PERFORMED,fstrCmd));
        }
        public void addActionListener(ActionListener l) {
            actionListener = AWTEventMulticaster.add(actionListener,l);
        }
        public Dimension getPreferredSize() {
            return new Dimension(img.getWidth(this),img.getHeight(this));
        }
        public Dimension getMinimumSize() {
            return getPreferredSize();
        }
        public void paint(Graphics g) {
            final Dimension DIM = getSize();
            if(m_iStatus == PRESSED_IN_COMP)
                g.drawImage(img.0,0,DIM.width,DIM.height,img.getWidth(this),img.getHeight(this),0,0,this);
            else
                g.drawImage(img.0,0,DIM.width,DIM.height,this);
        }
        public boolean isFocusable() {
            return true;
        }
    }
    class Beispiel52 extends Dialog implements ActionListener {
        public Beispiel52(LayoutManager mgr) {
            super(new Frame());
            setLayout(mgr);
            RButton b1 = new RButton("Hello");
            RButton b3 = new RButton("World");
            Button b2 = new Button("Juhu");
            b1.addActionListener(this);
            b2.addActionListener(this);
            b3.addActionListener(this);
            add(b1);add(b2);add(b3);pack();setVisible(true);
        }
        public void actionPerformed(ActionEvent e) {
            System.out.println("Command : " + e.getActionCommand());
        }
    }
    public static void main(String [] args) {
        new Beispiel52(new GridLayout(1,2));
        new Beispiel52(new FlowLayout());
    }
}

```

The diagram illustrates the execution flow of a program involving multiple threads. It shows a sequence of events labeled 1 through 4a and 4b. Step 1 starts a thread. Step 2a shows the thread performing a task. Step 2b shows another task being performed. Step 3a and 4a show the thread returning to the first task. Step 3b and 4b show it returning to the second task.

Nebenläufigkeit (Fort.)

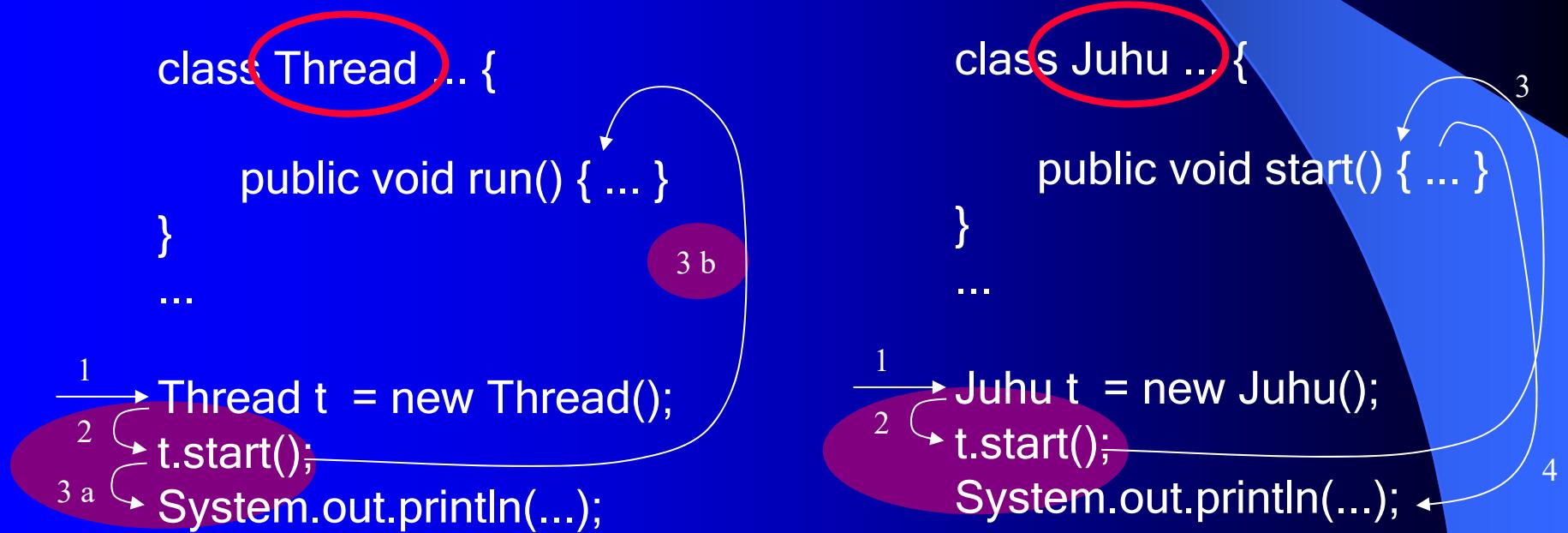
"geteilter", sprich paralleler Programmablauf

- nach einem Schritt kann sich das Programm aufteilen und läuft nebeneinander her
- in einem Faden ("Thread") kommt ein Schritt nach dem anderen (sequentiell, d.h. 2a < 3a)
- die Schritte der Fäden zueinander ist unbekannt (3a < 3b oder 3a > 3b ist zufällig)
- ein Faden kann sich wieder aufteilen

Nebenläufigkeit (Fort.)

Die Klasse Thread erzeugt in Java solche neuen Pfäden.

Die Methode start der Klasse Thread teilt das Programm auf.
Der erste Faden macht nach dem start weiter, der zweite Faden macht im run weiter.



Nebenläufigkeit (Fort.)

```
class Thread ... {  
    public void start();  
    public void run();  
    ...  
}
```

Vorgehensweise für Nebenläufigkeit:

1. eigene Klasse von `Thread` ableiten
 2. die Methode `run` überschreiben
 3. ein Objekt der von `Thread` abgeleiteten Klasse erzeugen
 4. von dem Objekt die `start`-Methode aufrufen
- ⇒ es wird nach der `start`-Methode ***und*** in der `run`-Methode weitergearbeitet

Go!

```
class Beispiel58 extends Thread { Beispiel
```

```
    String m_Msg;
```

```
    public Beispiel58(String msg) {m_Msg = msg;}
```

```
    public void run() {
```

```
        while(true) {
```

```
            System.out.println(m_Msg);
```

```
            try {
```

```
                Thread.sleep((int)(Math.random() * 1500));
```

```
            } catch (InterruptedException e) {
```

```
            }
```

```
        }
```

```
}
```

```
    public static void main(String[] args) throws Exception {
```

```
        Beispiel58 b1 = new Beispiel58("Hello World");
```

```
        Beispiel58 b2 = new Beispiel58("cool...");
```

```
        b1.start();
```

```
        System.out.println("\t... 1. Thread erzeugt ...");
```

```
        b2.start();
```

```
        System.out.println("\t... 2. Thread erzeugt ...");
```

```
        System.out.println("\t... und fertig ...");
```

```
}
```

```
}
```

Endlosschleife

schläft
zufällig bis zu
1,5 Sekunden

1. Thread erzeugen

2. Thread erzeugen

Haupt-Thread
beenden

Nebenläufigkeit (Fort.)

1. Das Programm ist *nicht beendet*, wenn die main-Routine verlassen wird.
2. Das Programm ist erst zu Ende, wenn alle Threads beendet sind.
3. Die Threads werden gleichzeitig abgearbeitet.
4. Wie die Threads zueinander ablaufen, ist nicht vorhersagbar.
5. Threads teilen sich aber die gleichen Variablen.

Go!

Beispiel

```
class Beispiel58_1 extends Thread {  
    static int m_iCnt = 0;
```

```
    public void run() {  
        for(int i = 0;i < 100;++i) {  
            if (m_iCnt % 2 == 0)  
                System.out.print("\t");  
            System.out.println(++m_iCnt);  
        }  
    }
```

beide Threads greifen auf die gleiche Klassenvariable m_iCnt zu

die run-Methode ist nicht atomar, die Threads verschränken sich ineinander
⇒ Vorsicht mit print Anweisungen in mehreren Threads

```
    public static void main(String[] args) throws Exception {  
        Beispiel58_1 b1 = new Beispiel58_1();  
        Beispiel58_1 b2 = new Beispiel58_1();  
        b1.start();  
        b2.start();  
    }  
}
```

Go!

Nochmal: Threads teilen sich die gleichen Variablen, aber ...

- ... eine Methode muss die Variable nicht immer neu auslesen

```
class Beispiel58_1_1 extends Thread {  
    int i = 0;  
  
    public void run() {  
        while(i == 0)    läuft solange, bis sich i ändert  
        ;  
    }  
  
    public static void main(String[] args) throws Exception {  
        Beispiel58_1_1 b = new Beispiel58_1_1();  
        b.start();  
        System.out.println("run läuft noch");  
        Thread.sleep(2000);  
        System.out.println("jetzt sollte gleich Schluss sein");  
        b.i = 10;    hier ändert sich i, folglich sollte der  
    }                Thread sich beenden, aber ...
```

Schlüsselwort: volatile

- ... teilt dem Compiler mit, dass eine Variable in einem anderen Thread geändert werden kann
- Folge: die Methode speichert die Variable nicht zwischen und liest den Wert immer erneut aus dem Speicher aus
- Änderungen werden so weiter gegeben

```
class Beispiel58_1_2 extends Thread {  
    volatile int i = 0;  
  
    public void run() {  
        while(i == 0)  
            ;  
    }  
  
    public static void main(String[] args) throws Exception {  
        ...  
        b.i = 10;    diese Änderung bemerkt run jetzt  
    }  
}
```

jetzt wird i immer
neu ausgelesen

Vorlesung 11

Beenden von Threads

- Das Programm ist beendet, wenn alle Threads (inklusive des Hauptthreads) zu Ende sind.
- Ein einzelner Thread ist zu Ende, wenn die `run`-Methode zu Ende ist.
- Um von "außen" einen Thread zu beenden, muss dem Thread eine Nachricht geschickt werden, auf die der Thread in seiner `run`-Methode reagiert und sich selber beendet.

Threads ***werden nicht*** beendet,
sondern beenden ***sich selber***.

Beenden von Threads (Fort.)

```
class Thread ... {  
    public void interrupt();  
    public boolean isInterrupted();  
    ...  
}
```

- Die `interrupt`-Methode schickt dem Thread die Nachricht, dass er beendet werden soll.
- Diese Nachricht setzt nur ein boolesches, internes Flag, beendet den Thread aber nicht.
- Diese Flag kann mittels der `isInterrupted`-Methode abgefragt werden.
- Die `run`-Methode muss dieses Flag abfragen: nicht zu oft, nicht zu selten.
- Frage: warum nicht zu oft, warum nicht zu selten?

Go!

Beispiel

```
class Beispiel58_2 extends Thread {  
    String m_Msg;  
    public Beispiel58_2(String msg) {m_Msg = msg;}
```

```
public void run() {  
    while(!isInterrupted()) {  
        System.out.println(m_Msg);  
    }  
}
```

Drückt die Nachricht so lange aus, bis das Interrupt Signal kommt

```
public static void main(String[] args) throws Exception {  
    Beispiel58_2 b1 = new Beispiel58_2("Hello World");  
    Beispiel58_2 b2 = new Beispiel58_2("cool...");  
    b1.start();System.out.println("\t... 1. Thread erzeugt ...");  
    b2.start();System.out.println("\t... 2. Thread erzeugt ...");  
    Thread.sleep(1000);  
    b1.interrupt();  
    System.out.println("\t... Schluss mit \"Hello World\" ...");  
    Thread.sleep(100);  
    b2.interrupt();  
    System.out.println("\t... Nix mit \"cool\" ...");  
    System.out.println("\t... Und Tschuess ...");  
}
```

Unterbricht den
1. Thread nach 1
Sekunde

Unterbricht den
2. Thread nach
weiteren 0,1
Sekunde

Beenden von Threads (Fort.)

Probleme mit der interrupt-Methode:

- Die sleep-Methode der Thread-Klasse fängt ebenfalls das Interrupt-Signal ab.
- Wird die interrupt-Methode während der sleep-Methode aufgerufen, so wird
 1. die sleep-Methode abgebrochen
 2. die InterruptedException Exception geworfen
 3. das interne Interrupt-Flag wieder zurückgesetzt.
- Folge: die run-Methode wird nicht beendet

```
public void run() {  
    while(!isInterrupted()) {  
        System.out.println(m_Msg);  
        try {Thread.sleep((int)(Math.random() * 1500));}  
        } catch (InterruptedException e) {}  
    }  
}
```

Lösung:

- In der Exception Behandlung der sleep-Methode nochmals die interrupt-Methode aufrufen, um das Interrupt Flag wieder zu setzen.

```
public void run() {  
    while(!isInterrupted()) {  
        System.out.println(m_Msg);  
        try {  
            Thread.sleep((int)(Math.random() * 1500));  
        } catch (InterruptedException e) {  
            interrupt();  
        }  
    }  
}
```

Ursprungsproblem

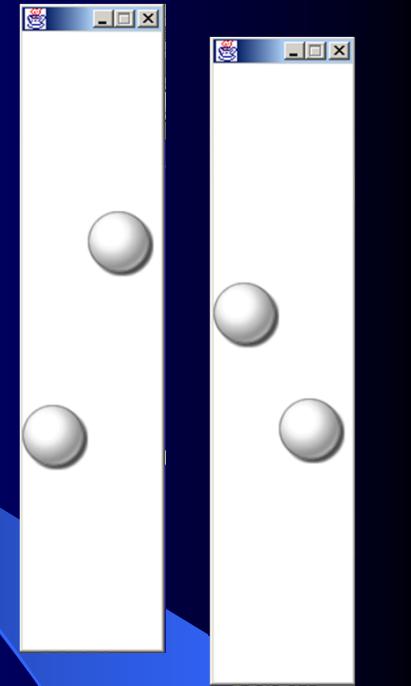
Aufgabe:

zwei Fenster öffnen, in denen die Bälle auf- und abspringen

Lösung:

1. die Klasse mit den beiden Bällen von der Thread-Klasse ableiten
2. in der `run`-Methode die `anime`-Methode aufrufen
(oder den Inhalt der `anime`-Methode gleich in die `run`-Methode kopieren).

Problem: ???



Das Runnable-Interface

Problem:

In Java kann eine Klasse immer nur von maximal einer anderen Klasse abgeleitet sein \Rightarrow man kann eine Klasse nicht gleichzeitig von JFrame und Thread ableiten.

Lösung:

Die eigene Klasse muss das Runnable-Interface implementieren. Einem neuen Thread übergibt man dann ein Objekt dieser eigenen Klasse.

```
class Thread ... {  
    public Thread(Runnable obj);  
    ...  
}
```

Das Runnable-Interface (Fort.)

Allgemeines Schema für das Runnable-Interface:

```
interface Runnable {  
    public void run();  
}
```

Diese Methode soll
nebenläufig ausgeführt
werden

```
class Juhu extends JFrame implements Runnable {  
    ...  
    public void run() { ... }  
    ...  
}  
...  
Juhu j = new Juhu();  
Thread t = new Thread(j);  
t.start();  
...
```

legt ein neues
Juhu Objekt an

legt einen neuen
Thread für das
Juhu Objekt an

startet die Juhu run-
Methode und arbeitet
parallel weiter

Go!

Beispiel

```
class Beispiel58_5 implements Runnable {  
    String m_Msg;  
    public Beispiel58_5(String msg) {m_Msg = msg;}  
    public void run() {  
        while(true) {  
            System.out.println(m_Msg);  
            try {  
                Thread.sleep((int)(Math.random() * 1500));  
            } catch (InterruptedException e) {}  
        }  
    }  
    public static void main(String[] args) throws Exception {  
        Beispiel58_5 b1 = new Beispiel58_5("Hello World");  
        Beispiel58_5 b2 = new Beispiel58_5("cool...");  
        Thread t1 = new Thread(b1);  
        Thread t2 = new Thread(b2);  
        t1.start();  
        System.out.println("\t... 1. Thread erzeugt ...");  
        t2.start();  
        System.out.println("\t... 2. Thread erzeugt ...");  
        System.out.println("\t... Und Tschuess ...");  
    }  
}
```

Thread kann nicht
beendet werden

2 neue Threads
erzeugen

beide Threads
starten

Das Runnable-Interface (Fort.)

Problem:

Wie kann die run-Methode der eigenen Klasse beendet werden?
Das Runnable-Interface hat keine interrupt- oder isInterrupted-Methode.

Lösung:

selber implementieren !

Go!

Beispiel

```
class Beispiel58_6 implements Runnable {  
    String m_Msg;  
    volatile boolean m_bCont = true;  
    public Beispiel58_6(String msg) {m_Msg = msg;}  
    public void nuAberSchluss() {m_bCont = false;}  
    public void run() {  
        while(m_bCont) {  
            System.out.println(m_Msg);  
            try {  
                Thread.sleep((int)(Math.random() * 150));  
            } catch (InterruptedException e) {}  
        }  
    }  
    public static void main(String[] args) throws Exception {  
        Beispiel58_6 b1 = new Beispiel58_6("Hello World");  
        Beispiel58_6 b2 = new Beispiel58_6("cool...");  
        Thread t1 = new Thread(b1);Thread t2 = new Thread(b2);  
        t1.start();System.out.println("\t... 1. Thread erzeugt ...");  
        t2.start();System.out.println("\t... 2. Thread erzeugt ...");  
        Thread.sleep(2000);b1.nuAberSchluss();  
        Thread.sleep(1000); b2.nuAberSchluss();  
        System.out.println("\t... Und Tschuess ...");  
    }  
}
```

Hier muss keine
Exception Behandlung
vorgenommen werden.
Warum nicht?

Sendet den Objekten,
nicht den Threads den
Beenden-Wunsch

Anhalten von Threads

- Manchmal besteht der Wunsch, dass einzelne Threads ihre Berechnungen unterbrechen und anhalten bzw. warten, bis bestimmte Berechnungen fertig sind
- dieses Anhalten kann genau wie das Interrupt nicht von außen erzwungen werden, sondern ...
- ... die run-Methode muss selber entscheiden, ob sie warten möchte
- ähnlich wie mit dem Interrupt kann dies mittels eines booleschen Flags gelöst werden

Beispiel

```
public class Waiting1 implements Runnable {  
    private String m_Msg;  
    private volatile boolean m_bWaiting = false;  
  
    public Waiting1(String msg) {m_Msg = msg;}  
  
    public void run() {  
        try {  
            while(true) {  
                while (m_bWaiting)  
                    ; // busy waiting: do nothing but check again  
                System.out.println(m_Msg);  
                Thread.sleep((int)(Math.random() * 800));  
            }  
        } catch (InterruptedException e) {  
        }  
    }  
    public void stopThisThing() {  
        m_bWaiting = true;  
    }  
    public void restart() {  
        m_bWaiting = false;  
    }  
    ...
```

m_bWaiting wird außerhalb der run Methode gesetzt, aber innerhalb gelesen \Rightarrow volatile

während des Wartens mache immer wieder nichts: SEHR TEUER, GANZ SCHLECHT

setzen des Flags

Beispiel

...

```
public static void main(String[] args) throws Exception {  
    Waiting1 b1 = new Waiting1("Hello World");  
    Waiting1 b2 = new Waiting1("cool...");  
    new Thread(b1).start();  
    System.out.println("\t... 1. Thread erzeugt ...");  
    new Thread(b2).start();  
    System.out.println("\t... 2. Thread erzeugt ...");  
    System.out.println("\t... und fertig ...");  
    Thread.sleep(5000);  
    System.out.println("\t... stop 1. Thread ...");  
    b1.stopThisThing();  
    Thread.sleep(5000);  
    System.out.println("\t... restart 1. Thread ...");  
    b1.restart();  
}
```

Starten der beiden Threads

nach 5 Sekunden ...

... pausieren des 1. Threads

nach weiteren 5 Sekunden ...

... reaktivieren des 1. Threads

Anhalten von Threads (Forts.)

- Das Warten kostet viel Rechenzeit, in der nichts passiert
- Es ist schlecht, ständig und immer wieder das Flag zu testen
- besser ist es, zwischen dem Abfragen des Flags ein wenig zu schlafen

Go!

Beispiel

```
public class Waiting2 implements Runnable {
```

...

```
    public void run() {
        try {
            while(true) {
                while (m_bWaiting)
                    Thread.sleep(20); // NO busy waiting: sleep a little bit
                System.out.println(m_Msg);
                Thread.sleep((int)(Math.random() * 800));
            }
        } catch (InterruptedException e) {
        }
    }
}
```

nur ein bisschen Schlafen (20 ms),
entlasten den Prozessor

Thread.sleep(20); // NO busy waiting: sleep a little bit

System.out.println(m_Msg);
Thread.sleep((int)(Math.random() * 800));

}

} catch (InterruptedException e) {

}

}

...

Diese Lösung ist besser, aber noch nicht gut

Anhalten von Threads (Forts.)

- wenn ein Thread anhalten soll, ist es am Besten, der Thread ruft die `wait()` Methode auf
- daraufhin wird es aus der Thread Scheduler entfernt und kostet gar keine Rechenzeit mehr
- um den Thread von außen wieder zu starten, muss die `notify()` Methode aufgerufen werden
- beide Methoden müssen mit dem Thread synchronisiert werden

Diese Problematik wird ausführlich
in Parallelprogrammierung behandelt

Beispiel

```
public class Waiting3 implements Runnable {  
    ...  
    public void run() {  
        try {  
            while(true) {  
                if (m_bWaiting) {  
                    synchronized (this) {  
                        wait(); // sleep until notify()  
                        m_bWaiting = false;  
                    }  
                }  
                System.out.println(m_Msg);  
                Thread.sleep((int)(Math.random() * 800));  
            }  
        } catch (InterruptedException e) {  
        }  
    }  
    public void stopThisThing() {  
        m_bWaiting = true;  
    }  
    synchronized public void restart() {  
        notify();  
    }  
}
```

wait() muss synchronisiert werden

nach dem Aufwachen,
Flag löschen

wie gehabt

auch notify() muss synchronisiert werden

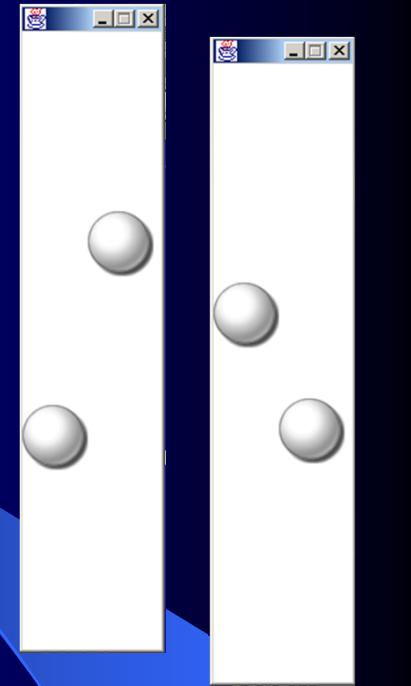
Ursprungsproblem: parallele Animation

Aufgabe:

zwei Fenster öffnen, in denen die Bälle auf- und abspringen

Lösung:

1. Die Klasse mit den beiden Bällen muss das **Runnable**-Interface implementieren.
2. In der **run**-Methode die **anime**-Methode aufrufen (oder den Inhalt der **anime**-Methode gleich in die **run**-Methode kopieren).
3. Im Hauptprogramm zwei Threads anlegen und starten.



Go!

Beispiel

```
import javax.swing.*;  
import java.awt.*;  
class Beispiel59 extends JFrame implements Runnable{  
    ...;  
    public Beispiel59(int iSleeper) throws Exception { ... }  
    public void paintComponent(Graphics g) { ... }  
    public void anime() throws Exception { ... }
```

Alles wie gehabt aus
Beispiel 58: 2 Bälle über
Doublebuffering animieren

```
public void run() {  
    try {  
        anime();  
    } catch (Exception e) {  
    }  
}
```

Die run-Methode ruft
nur die Animation auf

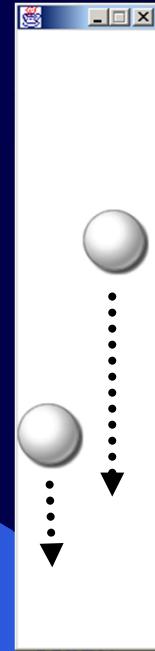
```
public static void main(String[] args) throws Exception {  
    Beispiel59 b1 = new Beispiel59(20);  
    Beispiel59 b2 = new Beispiel59(13);  
    Thread t1 = new Thread(b1);  
    Thread t2 = new Thread(b2);  
    t1.start();  
    t2.start();  
}
```

2 Fenster mit
unterschiedlichen
Geschwindigkeiten

Mehrfache Animation in einem Fenster

Aufgabe:

nur *ein Fenster* öffnen, in denen *mehrere Bälle* auf- und abspringen, die aber *unterschiedliche Geschwindigkeiten* haben



Idee:

Für jeden Ball einen eigenen Thread laufenlassen.

Problem:

Wie kommuniziert das Fenster (in der paintComponent-Methode notwendig) mit den einzelnen Threads?

Mehrfache Animation in einem Fenster (Fort.)

Lösung:

- eigene Dialogkomponente kreieren, die als Hauptbestandteil die Animation beinhaltet
- dazu eine eigene Klasse entwickeln, die
 - das Runnable-Interface implementiert
 - gleich zum Anfang ein Thread startet, um ihre eigene Animation zum Laufen zu bringen

Frage(n):

- Muss die Klasse von irgendeiner Klasse abgeleitet sein?
- Welche Methoden müssen implementiert werden?

Mehrfache Animation in einem Fenster (Fort.)

Antwort:

- Klasse muss von JComponent abgeleitet
- die Methoden der JComponent Klasse
 - public Dimension getMinimumSize()
 - public Dimension getPreferredSize()müssen überschrieben werden
- Klasse muss das Runnable-Interface implementiert
- dazu muss eine
 - public void run()Methode implementiert werden (sie enthält die eigentliche Animation)
- die Klasse muss darauf reagieren können, dass die Größe von außen verändert wird

Mehrfache Animation in einem Fenster (Fort.)

Allgemeine Struktur:

```
class Bounce extends JComponent implements Runnable {
```

```
    Bounce() {
```

```
        ...
```

```
        Thread t = new Thread(this);  
        t.start();
```

```
}
```

startet für sich selber
einen Thread

```
@Override public void paintComponent(Graphics g) { ... }
```

zeichnet den Ball an
der aktuellen Position

```
public void run() {  
    while (true) { ... }  
}
```

enthält die
Animation

```
@Override public Dimension getPreferredSize() { ... }
```

```
@Override public Dimension getMinimumSize() { ... }
```

```
}
```

gibt die Größe der
Komponente an

Mehrfache Animation in einem Fenster (Fort.)

Anwendung der Bounce-Klasse: wird verwendet wie jedes normale Dialogelement

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

class Beispiel60 {

    public static void main(String[] args) throws Exception {
        JFrame f = new JFrame();
        f.setLayout(new GridLayout(1,2));
        f.add(new Bounce());
        f.add(new JButton("Machtnix"));
        f.pack();
        f.setVisible(true);
    }
}
```

Macht für den Anwender
keinen Unterschied, ob
Bounce ein eigenes
Dialogelement ist

Implementierung der Bounce-Klasse

```
class Bounce_1 extends JComponent implements Runnable {  
    Image m_Img;  
    volatile int m_iHeight = 0;  
    volatile int m_iMaxHeight = 0;  
    int m_iWait = 25;  
    volatile int m_ilpress = 0;  
  
    Bounce_1() throws Exception {  
        m_Img = getToolkit().createImage("weissschatten.gif");  
        MediaTracker mt = new MediaTracker(this);  
        mt.addImage(m_Img, 0);  
        mt.waitForAll();  
        Thread t = new Thread(this),  
        t.start();  
        addComponentListener(new ComponentAdapter() {  
            public void componentResized(ComponentEvent e) {  
                m_iMaxHeight = getHeight() - m_Img.getHeight(Bounce_1.this);  
            }  
        });  
    }  
}
```

Lädt das Bild vollständig ein,
d.h. wartet den Ladevorgang ab

Startet einen neuen Thread und
beginnt mit der Animation für sich

Setze die neue Höhe bei Größenänderung

...

Implementierung der Bounce-Klasse (Fort.)

```
public void run() {  
    try {  
        while (true) {  
            for(m_iHeight = m_iMaxHeight;m_iHeight > 0;m_iHeight -= 5) {  
                Thread.sleep(m_iWait);repaint();  
            }  
            m_iHeight = 0; impress(false);  
            for(m_iHeight = 0;m_iHeight < m_iMaxHeight;m_iHeight += 5) {  
                Thread.sleep(m_iWait); repaint();  
            }  
            m_iHeight = m_iMaxHeight; impress(true);  
        }  
    } catch (InterruptedException e) {}  
}  
  
@Override  
public Dimension getPreferredSize() {return getMinimumSize(); }  
  
@Override  
public Dimension getMinimumSize() {  
    return new Dimension(m_Img.getWidth(this),m_Img.getHeight(this)*3);  
}
```

oben und unten wird
der Ball eingedrückt

Größe der Komponente

Implementierung der Bounce-Klasse (Fort.)

```
@Override  
public void paintComponent(Graphics g) {  
    if (m_iMaxHeight == 0)  
        m_iMaxHeight = getHeight()-m_Img.getHeight(this);  
    g.drawImage(m_Img,0,m_iHeight,  
               m_Img.getWidth(this),m_Img.getHeight(this)-m_ilpress(this));  
}  
  
private void impress(boolean bAtBottom) throws InterruptedException {  
    for(m_ilpress = 0;m_ilpress < 20;++m_ilpress) {  
        if (bAtBottom)  
            ++m_iHeight;  
        Thread.sleep(m_iWait);  
        repaint();  
    }  
    for(m_ilpress = 20;m_ilpress > 0;--m_ilpress) {  
        if (bAtBottom)  
            --m_iHeight;  
        Thread.sleep(m_iWait);  
        repaint();  
    }  
}
```

zeichnet den Ball an der aktuellen m_iHeight Position

gleicht die Höhe beim Eindrücken am Boden aus

Mehrfache Animation in einem Fenster (Fort.)

Aufgabe (Wiederholung):

nur *ein Fenster* öffnen, in denen *mehrere Bälle* auf- und abspringen, die aber *unterschiedliche Geschwindigkeiten* haben.

- mehrere Bälle sind kein Problem, da die Bounce-Klasse einfach mehrfach instanziert werden muss
- unterschiedliche Geschwindigkeiten ist auch kein Problem, da jedes einzelne Bounce-Objekt unterschiedlich schnell arbeiten kann (`m_iWait` Variable)

Go!

Mehrfache Animation in einem Fenster (Fort.)

```
class Bounce extends JComponent implements Runnable {  
    int m_iWait = 25;  
    ...  
    public void setWait(int iWait) {  
        m_iWait = iWait;  
    }  
}
```

setzt die Wartezeit
in Millisekunden

```
class Beispiel60_2 {  
    public static void main(String[] args) throws Exception {  
        JFrame f = new JFrame();  
        f.setLayout(new GridLayout(1,2));  
        Bounce b1 = new Bounce();  
        Bounce b2 = new Bounce();  
        f.add(b1);f.add(b2);f.pack();f.setVisible(true);  
        Thread.sleep(5000);  
        System.out.println("nu aber los ...");  
        b1.setWait(10);  
    }  
}
```

erzeugt 2 Bälle

wartet 5 Sekunden und
macht den 1. Ball schneller

Mehrfache Animation in einem Fenster (Fort.)

Aufgabe:

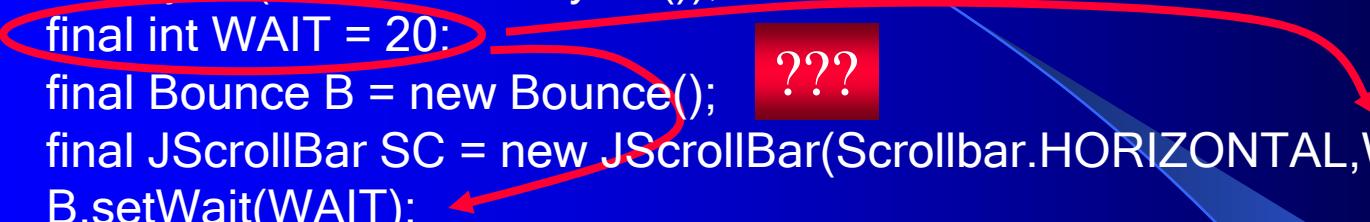
Zu jedem Ball soll es einen JScrollBar geben, mit dem man die Geschwindigkeit einstellen kann.

Lösung:

- Bounce-Objekt und JScrollBar in einem JPanel vereinen
- JScrollBar-Event abfangen und mittels der `setWait`-Methode des Bounce-Objekts die Geschwindigkeit einstellen
- Um diese Kombination mehrfach verwenden zu können, alles in eine Klasse zusammenfassen.
- Von welcher Klasse muss diese Klasse abgeleitet werden, um als Dialog-Element verwendet werden zu können?

Go!

Mehrfache Animation in einem Fenster (Fort.)

```
class ControlBounce extends JPanel {  
    public ControlBounce() throws Exception {  
        setLayout(new BorderLayout());  
        final int WAIT = 20; ???   
        final Bounce B = new Bounce();  
        final JScrollBar SC = new JScrollBar(Scrollbar.HORIZONTAL,WAIT,1,10,30);  
        B.setWait(WAIT);  
        SC.addAdjustmentListener(e -> B.setWait(SC.getValue()));  
        add(BorderLayout.CENTER,B);  
        add(BorderLayout.SOUTH,SC); reagiert auf den Scrollbar  
    }  
}  
  
class Beispiel61 {  
    public static void main(String[] args) throws Exception {  
        JFrame f = new JFrame();f.setLayout(new GridLayout(1,3));  
        f.add(new ControlBounce());  
        f.add(new ControlBounce());  
        f.add(new ControlBounce());  
        f.pack();  
        f.setVisible(true);  
    }  
}
```

3 Bälle erzeugen und dem Fenster zufügen

Vorlesung 12

Komplexe Dialogelemente in Swing

folgende komplexe Dialogelemente gibt es in Swing

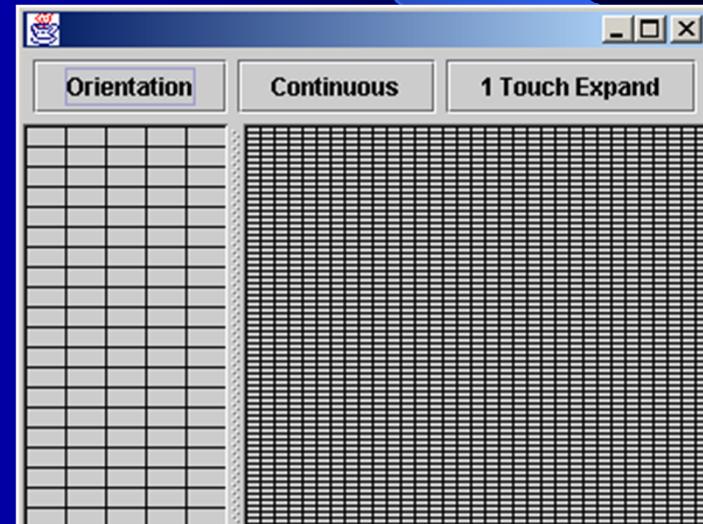
Swing	AWT
JScrollPane	ScrollPane
JSplitPane	
JTabbedPane	
JTable	
JTree	

Die JScrollPane verhält sich ähnlich wie die ScrollPane, hat jedoch mehr Möglichkeiten der Einstellungen.

Die JSplitPane Klasse

- Die JSplitPane Klasse teilt eine *Komponente* in 2 Teile, die entweder *neben-* oder *untereinander* dargestellt werden.
- In den beiden Teilen werden 2 Komponenten dargestellt.
- Die beiden Teile werden durch einen sichtbaren Separator geteilt, der vom Anwender zur Laufzeit verschoben werden kann.

```
class JSplitPane {  
    public JSplitPane(int orient);  
    final int HORIZONTAL_SPLIT;  
    final int VERTICAL_SPLIT;  
    ...  
}
```



Die JSplitPane Klasse (Fort.)

Die beiden darzustellenden Komponenten können dem Konstruktor direkt übergeben werden

```
public JSplitPane(int orient, Component left, Component right);
```

oder nachträglich gesetzt werden durch

```
public void setLeftComponent(Component comp); bzw.
```

```
public void setTopComponent(Component comp);
```

und

```
public void setRightComponent(Component comp); bzw.
```

```
public void setBottomComponent(Component comp);
```

Die JSplitPane Klasse (Fort.)

Der update der beiden Komponenten kann während der Verschiebung des Separators erfolgen oder danach

```
public JSplitPane(int orient, boolean continuousLayout,  
                  Component left, Component right);
```

oder nachträglich gesetzt werden durch

```
public void setContinuousLayout(boolean cont);
```

Der aktuell eingestellte Wert kann mittels der Methode

```
public boolean isContinuousLayout();
```

abgefragt werden.

Die JSplitPane Klasse (Fort.)

Der Separator kann mit Pfeilen versehen werden, die jeweils einen Teil der JSplitPane komplett verdecken, wenn der entsprechende Pfeil angeklickt wird.



Die Methode

```
public void setOneTouchExpandable(boolean cont);
```

setzt diese Pfeile.

Der aktuell eingestellte Wert kann mittels der Methode

```
public boolean isOneTouchExpandable();
```

abgefragt werden.

```
...
class Grid extends JComponent {
    int m_iHorDist;
    int m_iVerDist;
    public Grid(int iHorDist,int iVerDist) {m_iHorDist = iHorDist;m_iVerDist = iVerDist;}
    @Override
    public void paintComponent(Graphics g) {
        final Dimension DIM = getSize();
        for(int i = 0;i < DIM.height;i += m_iHorDist) {
            g.drawLine(0,i,DIM.width-1,i);
        }
        for(int i = 0;i < DIM.width;i += m_iVerDist) {
            g.drawLine(i,0,i,DIM.height-1);
        }
    }
    @Override
    public Dimension getPreferredSize() {
        return new Dimension(10 * m_iHorDist,10 * m_iVerDist);
    }
    @Override
    public Dimension getMiniumSize() {
        Dimension dim = getPreferredSize();
        dim.setSize(dim.width / 2,dim.height / 2);
        return dim;
    }
}
...

```

Beispiel

WICHTIG: paintComponent,
nicht paint überlagern

Die minimale Größe
ist $\frac{1}{4}$ so groß wie die
bevorzugte Größe

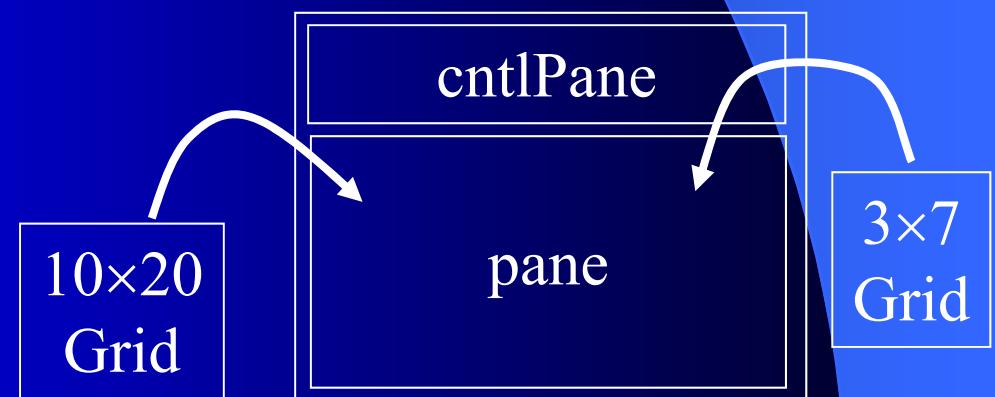
Beispiel (Fort.)

...

```
class Beispiel72 extends JFrame {  
    public Beispiel72() {  
        final JSplitPane pane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,true,  
                                              new Grid(10,20),  
                                              new Grid(3,7));  
  
        JPanel cntlPane = new JPanel();  
        cntlPane.setLayout(new FlowLayout());  
        JButton orient = new JButton("Orientation");  
        JButton cont = new JButton("Continuous");  
        JButton expand = new JButton("1 Touch Expand");  
        cntlPane.add(orient);  
        cntlPane.add(cont);  
        cntlPane.add(expand);  
        setLayout(new BorderLayout());  
        add(BorderLayout.CENTER,pane);  
        add(BorderLayout.NORTH,cntlPane);  
    }  
}
```

erzeugt eine JSplitPane
und 2 Grid's unter-
schiedlicher Größe

3 JButtons zur Ein-
stellung von Eigen-
schaften der JSplitPane



Go!

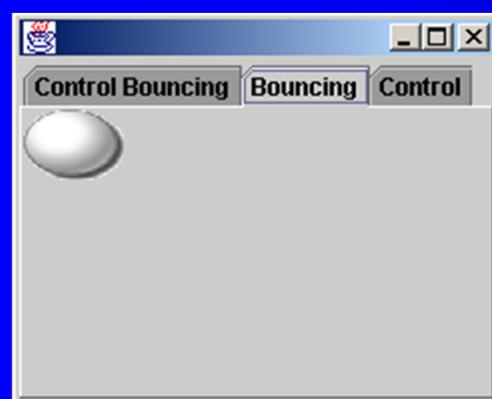
Beispiel (Fort.)

```
orient.addActionListener(e -> {
    if (pane.getOrientation() == JSplitPane.HORIZONTAL_SPLIT)
        pane.setOrientation(JSplitPane.VERTICAL_SPLIT);
    else
        pane.setOrientation(JSplitPane.HORIZONTAL_SPLIT); setzt die Orientierung um
    }
);
cont.addActionListener(e -> pane.setContinuousLayout(!pane.isContinuousLayout()); toggelt den Update-Modus
```

```
expand.addActionListener(e ->
    pane.setOneTouchExpandable(!pane.isOneTouchExpandable()));
pack();
setVisible(true);
}
public static void main(String[] args) {
    new Beispiel72();
}
```

Die JTabbedPane Klasse

- Die JTabbedPane Klasse verwaltet mehrere Registrierungskarten.
- Es wird immer genau eine Karte angezeigt.
- Jede Karte kann wieder eine Komponenten enthalten.
- Jede Karte hat einen Namen.
- Die Namen werden den Karten an einem Rand zugeordnet.
- Durch Auswählen des Namens wird die assozierte Karte in den Vordergrund gebracht.



```
class JTabbedPane {  
    public JTabbedPane();  
    public JTabbedPane(int orient);  
    final int TOP;  
    final int BOTTOM;  
    final int LEFT;  
    final int RIGHT;  
    ...  
}
```



Standard ist TOP

Die JTabbedPane Klasse (Fort.)

- Die Konstruktoren erzeugen noch keine Karten.
- Karten werden mittels der beiden Methoden `addTab` (nacheinander) und `insertTab` (an angegebener Position, startend bei 0) hinzugefügt.

```
class JTabbedPane {  
    public void addTab(String title, Component component);  
    public void addTab(String title, Icon icon,  
                      Component component);  
    public void addTab(String title, Icon icon,  
                      Component component, String tip);  
  
    public void insertTab(String title, Icon icon,  
                         Component component,  
                         String tip, int index)  
    ...  
}
```

Die JTabbedPane Klasse (Fort.)

- Die Position, an denen die Namen zu den Karten stehen, können angefragt und gesetzt werden.

```
class JTabbedPane {  
    public void setTabPlacement(int orientation);  
    public int getTabPlacement();  
    ...  
}
```

- Einzelne Karten können auch deaktiviert werden. Sie können dann nicht mehr ausgewählt werden (Zählen fängt bei 0 an.)

```
class JTabbedPane {  
    public void setEnabledAt(int index, boolean enabled);  
    public boolean isEnabledAt(int index);  
    ...  
}
```

Beispiel

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

class OrientButton extends JButton {
    public OrientButton(String title,final JTabbedPane PANE,final int ORIENT) {
        super(title);
        addActionListener(e -> PANE.setTabPlacement(ORIENT));
    }
}
```

2 eigene JButton
Klassen

```
class Enable extends JButton {
    public Enable(String title,final JTabbedPane PANE,final int INDEX) {
        super(title);
        addActionListener(e -> PANE.setEnabledAt(INDEX,!PANE.isEnabledAt(INDEX)));
    }
}
```

Setzt die Ausrichtung der
Kartennamen gemäß ORIENT

Toggelt das Enable-
Flag für den Eintrag an
der Position "INDEX"

Go!

Beispiel (Fort.)

```
class Control extends JPanel {  
    public Control(JTabbedPane pane) {  
        setLayout(new FlowLayout());  
        add(new OrientButton("top",pane,JTabbedPane.TOP));  
        add(new OrientButton("bottom",pane,JTabbedPane.BOTTOM));  
        add(new OrientButton("left",pane,JTabbedPane.LEFT));  
        add(new OrientButton("right",pane,JTabbedPane.RIGHT));  
        add(new Enable("enable \"Control Bounce\"",pane,0));  
        add(new Enable("enable \"Bounce\"",pane,1));  
    }  
}  
  
class Beispiel73 extends JFrame {  
    public Beispiel73() throws Exception {  
        JTabbedPane pane = new JTabbedPane();  
        pane.addTab("Control Bouncing",new ControlBounce());  
        pane.addTab("Bouncing",new Bounce());  
        pane.addTab("Control",null,new Control(pane),"Ich kontrolliere alles");  
        add(pane);  
        pack();  
        setVisible(true);  
    }  
    public static void main(String[] args) throws Exception {new Beispiel73();}  
}
```

Baut 4 von den
OrientButtons und
2 Enable-Buttons
zusammen

Die JTabbedPane
bekommt 3 Karten

Vorlesung 11 / 2

Die JTable Klasse

- Die JTable Klasse dient dazu, eine Tabelle als Dialogelement zur Verfügung zu stellen.
- Dieses sehr komplexe Element hat eine Vielzahl von Einstellmöglichkeiten.

```
class JTable {  
    public JTable(Object[][] rowData, Object[] columnNames);  
  
    ...  
}
```

- In der einfachsten Form übergibt man der Tabelle bei der Erzeugung die Daten in Form eines 2-dimensionalen Arrays.
- Die Beschriftung der Spalten ist ein 1-dimensionales Array.

Spalte 0	Spalte 1	Spalte 2
Datum : 0x0	Datum : 0x1	Datum : 0x2
Datum : 1x0	Datum : 1x1	Datum : 1x2
Datum : 2x0	Datum : 2x1	Datum : 2x2
Datum : 3x0	Datum : 3x1	Datum : 3x2
Datum : 4x0	Datum : 4x1	Datum : 4x2
Datum : 5x0	Datum : 5x1	Datum : 5x2
Datum : 6x0	Datum : 6x1	Datum : 6x2
Datum : 7x0	Datum : 7x1	Datum : 7x2
Datum : 8x0	Datum : 8x1	Datum : 8x2
Datum : 9x0	Datum : 9x1	Datum : 9x2
Datum : 10x0	Datum : 10x1	Datum : 10x2
Datum : 11x0	Datum : 11x1	Datum : 11x2

Die JTable Klasse (Fort.)

- Ein Objekt der JTable Klasse wird üblicherweise in ein JScrollPane eingebettet, um über größere Tabellen scrollen zu können.
- Die Spaltenbeschriftung wird nur dann aktiviert, wenn die Tabelle in einem JScrollPane eingebettet ist.
- Viele Einstellungen erlauben die unterschiedlichsten Selektionsmöglichkeiten.

```
class JTable {  
    public void setRowSelectionAllowed(boolean flag);  
    public void setColumnSelectionAllowed(boolean flag);  
    public void setSelectionMode(int selectionMode);  
    public void setCellSelectionEnabled(boolean flag);  
    ...  
}
```

Die JTable Klasse (Fort.)

- Auch die Größe der Zellen kann eingestellt werden.

```
class JTable {  
    public void setRowHeight(int newHeight);  
    public void setRowMargin(int newMargin);  
    public void setIntercellSpacing(Dimension newSpacing);  
    ...  
}
```

- Ob zwischen den Zellen Linen gezogen werden sollen (Standard) oder nicht, wird durch die folgenden Methoden geregelt.

```
class JTable {  
    public void setShowGrid(boolean b);  
    public void setShowHorizontalLines(boolean b);  
    public void setShowVerticalLines(boolean b);  
    ...  
}
```

Die JTable Klasse (Fort.)

- Die Methode `setAutoResizeMode` regelt, wie sich die Breite der Spalten verändern soll, wenn der Benutzer eine Spalte in ihrer Breite verändert.

```
class JTable {  
    public void setAutoResizeMode(int mode);  
    final int AUTO_RESIZE_OFF;  
    final int AUTO_RESIZE_LAST_COLUMN;  
    final int AUTO_RESIZE_NEXT_COLUMN;  
    final int AUTO_RESIZE_SUBSEQUENT_COLUMNS;  
    final int AUTO_RESIZE_ALL_COLUMNS;  
    ...  
}
```

Die JTable Klasse (Fort.)

- Der Zugriff auf die einzelnen Elemente erfolgt mittels der Methoden:

```
class JTable {  
    public Object getValueAt(int row, int column);  
    public void setValueAt(Object val, int row, int column);  
    ...  
}
```

- `getValueAt` liefert das Objekt, dass sich in der Zeile `row` und der Spalte `column` befindet.
- `setValueAt` speichert das übergebene Objekt `val` an der angegebenen Position ab.

Go!

Beispiel (Fort.)

```
...
class Beispiel74 extends JFrame {
    public Beispiel74() {
        Object [][] data = new Object[200][3];
        String [] names = new String[data[0].length];
        for(int i = 0;i < data.length;++i) {
            for(int j = 0;j < data[i].length;++j) {
                data[i][j] = "Datum :" +i+"x"+j;
            }
        }
        for(int i = 0;i < names.length;++i) {
            names[i] = "Spalte " + i;
        }
        JTable tab = new JTable(data,names);
        tab.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
        tab.setColumnSelectionAllowed(true);
        tab.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        add(new JScrollPane(tab));
        pack();
        setVisible(true);
    }

    public static void main(String[] args) {new Beispiel74();}
}
```

Erzeugung von Daten und Spaltenbeschriftungen

Tabelle wird erzeugt

JTable wird fast immer mit JScrollPane verwendet

Die JTable Klasse (Fort.)

- Es ist nicht immer sinnvoll, der Tabelle bereits zur Initialisierung alle Daten mitzugeben. So kann der Tabelle anstelle der Daten eine Objekt übergeben werden, dass die Daten bereitstellt, wenn es sie braucht.
- Eine solche Klasse muss das **TableModel** Interface implementieren.

```
class JTable {  
    public JTable(TableModel tm);  
    public void setModel(TableModel tm);  
    public TableModel getModel();  
    ...  
}
```

Die JTable Klasse (Fort.)

- Das TableModel Interface stellt eine Reihe von Methoden zur Verfügung, um der Tabelle Informationen über die Daten zur Verfügung zu stellen.

```
interface TableModel {  
    public int getRowCount();  
    public int getColumnCount();  
    public String getColumnName(int columnIndex);  
    public Class getColumnClass(int columnIndex);  
    public boolean isCellEditable(int rowIndex, int columnIndex);  
    public Object getValueAt(int rowIndex, int columnIndex);  
    public void setValueAt(Object val, int rowIndex, int columnIndex);  
    public void addTableModelListener(TableModelListener l);  
    public void removeTableModelListener(TableModelListener l);  
}
```

Die JTable Klasse (Fort.)

- Um nicht alle Methoden des TableModel Interface implementieren zu müssen, stellt die **abstrakte** Klasse AbstractTableModel eine **Teil-Implementierung** des TableModel Interface da.
- Weiterhin stellt diese Klasse Methoden zur Verfügung, um die TableModelListener zu benachrichtigen, dass sich eine oder mehrere Zellen verändert haben.

```
abstract class AbstractTableModel {  
    ...  
    void fireTableCellUpdated(int row, int column) ;  
    void fireTableDataChanged();  
    ...  
}
```

Beispiel

```
class IntTab extends AbstractTableModel {  
    int [][] m_Data;  
    int m_iResult = 0;
```

```
    public IntTab(int x,int y) {  
        public int getRowCount() {  
        public int getColumnCount() {  
        public String getColumnName(int i) {  
        public boolean isCellEditable(int x,int y) {  
            return true; }
```

```
    m_Data = new int[x][y]; }  
    return m_Data.length; }  
    return m_Data[0].length; }  
    return "Spalte " + i; }  
    return true; }
```

```
    public Object getValueAt(int x,int y) {  
        return m_Data[x][y];  
    }
```

liefert den Wert an Position
(x,y) als String zurück

```
    public void setValueAt(Object val,int rowIndex,int columnIndex) {  
        try {  
            int iNewValue = Integer.parseInt((String)val);  
            m_iResult -= m_Data[rowIndex][columnIndex];  
            m_iResult += iNewValue;  
            m_Data[rowIndex][columnIndex] = iNewValue;  
            fireTableCellUpdated(rowIndex,columnIndex);  
        } catch (Exception e) {}  
    }
```

versucht, das val in einen
Integer zu konvertieren

benachrichtigt
die Listener

Go!

Beispiel (Fort.)

```
class TabWithResult extends JPanel {  
    public TabWithResult(int rows,int columns) {  
        final JLabel RES_LAB = new JLabel("Summe: 0");  
        final IntTab TAB_MOD = new IntTab(rows,columns);  
        setLayout(new BorderLayout());  
        add(BorderLayout.NORTH,RES_LAB);  
        add(BorderLayout.CENTER,new JScrollPane(new JTable(TAB_MOD)));  
        TAB_MOD.addTableModelListener(e ->  
            RES_LAB.setText("Summe: " + TAB_MOD.m_iResult)  
        );  
    }  
}  
  
class Beispiel75 extends JFrame {  
    public Beispiel75() {  
        JTabbedPane pane = new JTabbedPane();  
        pane.addTab(new TabWithResult(10,3),"klein");  
        pane.addTab(new TabWithResult(250,10),"groß");  
        add(pane);  
        pack();  
        setVisible(true);  
    }  
    public static void main(String[] args) throws Exception {new Beispiel75();}  
}
```



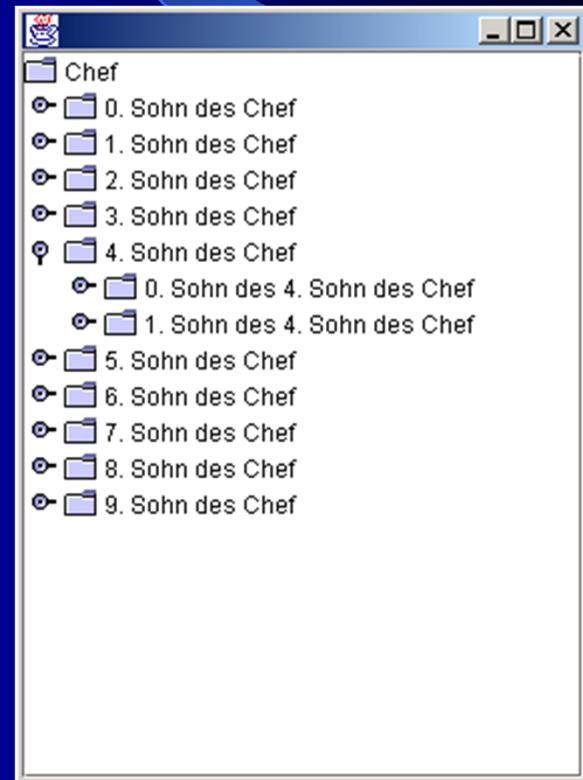
Sobald sich etwas in dem Tabellen Modell geändert hat, wird das Label neu dargestellt

Die JTree Klasse

- Die JTree Klasse dient dazu, baumartige Strukturen darzustellen.
- Solche Strukturen werden z.B. im Explorer verwendet.
- Es wird unterschieden, ob die Wurzel sichtbar oder unsichtbar ist, d.h. die Kinder sofort dargestellt werden.

```
class JTree {  
    public JTree(TreeNode root);  
    public void setRootVisible(boolean b);  
    public boolean isRootVisible();  
    ...  
}
```

- In der einfachsten Form übergibt man dem Baum bei der Erzeugung lediglich den Wurzelknoten.
- Dazu muss der Knoten das Interface TreeNode implementieren.



Die JTree Klasse (Fort.)

- Das TreeNode Interface erwartet eine Reihe von Methoden, die zum Aufbau der Baumstruktur erforderlich sind.
- Der Name im Baum wird durch die `toString` Methode erzeugt.

```
interface TreeNode {  
    public Enumeration children();  
    public boolean getAllowsChildren();  
    public TreeNode getChildAt(int childIndex);  
    public int getChildCount();  
    public int getIndex(TreeNode node);  
    public TreeNode getParent();  
    public boolean isLeaf();  
}
```

Die JTree Klasse (Fort.)

- Um auf das Auseinander- und Zusammenfalten des Baums reagieren zu können, können entsprechende Listener reagiert werden.
- Es wird unterschieden, zwischen dem Zeitpunkt unmittelbar vor der Ausführung und nach der Ausführung (`TreeWillExpandListener` bzw. `TreeExpansionListener`).
- Auf die Selektion einer oder mehrerer Knoten wird mittels des `TreeSelectionListener` reagiert.

```
class JTree {  
    ...  
    public void addTreeExpansionListener(TreeExpansionListener l);  
    public void addTreeSelectionListener(TreeSelectionListener l);  
    public void addTreeWillExpandListener(TreeWillExpandListener l);  
}
```

```
import java.awt.*;  
import java.awt.event.*;  
import java.util.*; ←  
import javax.swing.*;  
import javax.swing.event.*; ← für die Listener  
import javax.swing.tree.*; ← für TreeNode
```

Beispiel (Fort.)

```
class MyNode implements TreeNode {  
    MyNode[] m_Kids = null;  
    MyNode m_Father;  
    String m_Name;  
  
    public MyNode() {  
        this(null,0);  
    }
```

```
    public MyNode(MyNode father,int pos) {  
        m_Father = father;  
        if (m_Father == null)  
            m_Name = "Chef";  
        else  
            m_Name = String.valueOf(pos) + ". Sohn des " + m_Father.toString();  
        m_Kids = new MyNode[(int)(Math.random() * 15)];  
    }
```

...

wird für die Deklaration
von Enumeration benötigt

für TreeNode

Jeder Knoten merkt sich seine Kinder,
seinen Vater und seinen Namen

Jeder Knoten hat Platz um bis
zu 15 Kinder zu speichern

Beispiel (Fort.)

```
...  
public TreeNode getChildAt(int childIndex) {  
    if (m_Kids[childIndex] == null) {  
        m_Kids[childIndex] = new MyNode(this,childIndex);  
    }  
    return m_Kids[childIndex];  
}  
  
public int getChildCount() {  
    return m_Kids.length; }  
public TreeNode getParent() {  
    return m_Father; }  
public boolean isLeaf() {  
    return m_Kids.length == 0; }  
public String toString() {  
    return m_Name; }  
  
public int getIndex(TreeNode kid) {  
    return -1; }  
public boolean getAllowsChildren() {  
    return false; }  
public Enumeration children() {  
    return null; }  
  
} // end of MyNode declaration  
...
```

Die Kinder werden erst erzeugt, wenn auf sie zugegriffen wird

Go!

Beispiel (Fort.)

```
...  
class Beispiel76 extends JFrame {  
    public Beispiel76() {  
        JTree tree = new JTree(new MyNode());  
        tree.addTreeExpansionListener(new TreeExpansionListener() {  
            public void treeCollapsed(TreeExpansionEvent e) {  
                System.out.println("Collapsed " + e.getPath());}  
            public void treeExpanded(TreeExpansionEvent e) {  
                System.out.println("Expanded " + e.getPath());}});  
        tree.addTreeSelectionListener(e -> System.out.println("Selected " + e.getPath()));  
        tree.addTreeWillExpandListener(new TreeWillExpandListener() {  
            public void treeWillExpand(TreeExpansionEvent e) {  
                System.out.println("will expand " + e.getPath());}  
            public void treeWillCollapse(TreeExpansionEvent e) {  
                System.out.println("will collapse " + e.getPath());}});  
  
        add(new JScrollPane(tree));  
        pack();  
        setVisible(true);  
    }  
    public static void main(String[] args) throws Exception {  
        new Beispiel76();  
    }  
}
```

Dem JTree wird der Wurzelknoten übergeben

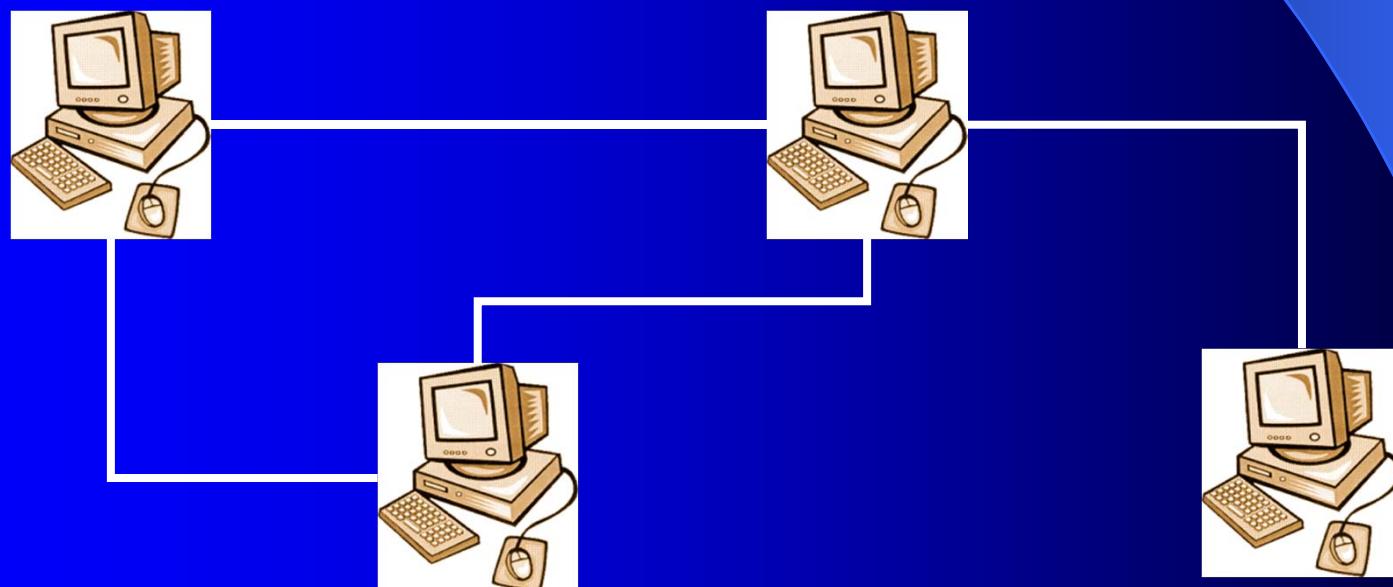
Alle Listener installiert

Verwendung des JTree in der JScrollPane

Vorlesung 13

Netzwerk Programmierung mit Java

- Aufgabe: mehrere Programme sollen über ein Netzwerk miteinander kommunizieren
- hierzu muss festgelegt werden, mittels welcher Protokolle dies geschehen soll
- im folgenden: ausschließlich TCP/IP Protokoll



Netzwerk Programmierung: Grundlagen

- Kommunikation bedeutet hier:
 - zwei Programme auf einem oder zwei Rechnern kommunizieren miteinander
- Fragen:
 - Wie startet die Kommunikation?
 - Wie werden die Rechner adressiert?
 - Wie wird das Programm adressiert?
 - Wie läuft die Kommunikation ab?
 - Wie wird die Kommunikation beendet?

Netzwerk Programmierung: Start der Kommunikation

- oft sieht die Kommunikationsstruktur eine Client-Server Architektur vor
- d.h. auf einem Server läuft ein Programm, dass darauf wartet, dass ein anderes Programm auf einem Client gestartet wird, und
- das entsprechende Programm auf dem Server anspricht
- es obliegt dann dem Server, die angewählte Verbindung durch den Client zu akzeptieren oder abzulehnen



Netzwerk Programmierung: Adressierung des Rechners

- für den Verbindungsauflauf muss der Client die Adresse des Servers kennen
- im TCP/IP Netz hat ein Rechner im Netzwerk eine sogenannte IP-Adresse
- diese IP-Adresse kann in Java durch die Klasse **InetAddress** ermittelt und verarbeitet werden
- dazu muss das Package **java.net** inkludiert werden
- ...

Netzwerk Programmierung: Adressierung des Rechners (Forts.)

- ...
- die eigentliche IP-Adresse besteht aus 4 Zahlen, die jeweils zwischen 0 und 255 (Bsp. 213.165.64.215)
- somit kann (und wird) eine IP-Adresse in 4 Bytes gespeichert
- da für Menschen die Zahlen schlecht zu merken sind, können Rechner mit Namen, sogenannte **Domain Names**, versehen werden
- die Zuordnung der Namen zu den IP-Adressen (**Domain Name System = DNS**) ist nur bedingt eindeutig, d.h.
 - zu einer IP-Adresse kann es mehrere Namen geben
 - zu einem Namen kann es auch mehrere IP-Adressen (oder gar keine geben)
- ...

Netzwerk Programmierung: Adressierung des Rechners (Forts.)

- ...
- die Zuordnung von IP-Adressen zu den Namen und besonders die Auflösung von Namen in Ihre zugehörigen IP-Adressen wird durch DNS-Server vorgenommen
- in Java erledigt dies ebenfalls die Klasse **InetAddress**

```
class InetAddress {  
    public static InetAddress getLocalHost()  
        throws UnknownHostException;  
    public static InetAddress getByName(String host)  
        throws UnknownHostException;  
}
```

Netzwerk Programmierung: Adressierung (Beispiel)

```
import java.net.*;  
  
interface GetInetAddress {  
    public InetAddress getInetAddress() throws UnknownHostException;  
}  
  
public class Socket1 {
```

muss für Sockets
inkludiert werden

```
    public static void eval(GetInetAddress iAdrInterface) {  
        try {  
            InetAddress iAdr = iAdrInterface.getInetAddress();  
            System.out.println(iAdr.getHostName());  
            System.out.println(iAdr.getHostAddress());  
            System.out.println(iAdr.getCanonicalHostName());  
            byte[] ipAdr = iAdr.getAddress();  
            for(int i = 0;i < ipAdr.length;++i)  
                System.out.print(ipAdr[i] + " ");  
        } catch (UnknownHostException e) {  
            System.out.println(e);  
        }  
        System.out.println("\n");  
    }  
    ...
```

eigenes Interface

Repräsentant
der IP-Adresse

Informationen
als String

IP-Adresse als
Byte-Array

Go!

Netzwerk Programmierung: Adressierung (Beispiel Forts.)

versucht, den DNS
Server zu kontaktieren
und löst „gmx.de“ auf

...

```
public static void main(String[] args) {  
    eval(() -> {return InetAddress.getByName("gmx.de");});  
    eval(() -> {return InetAddress.getByName("localhost");});  
    eval(() -> {return InetAddress.getLocalHost();});  
}
```

sollte immer klappen



der eigene Rechner

Netzwerk Programmierung: Adressierung des Programms

- Nach der Adressierung des Rechners muss auch das Programm adressiert werden, mit dem die Kommunikation stattfinden soll
- die Programme erhalten hierzu Nummern, sogenannte Ports
- die Nummern liegen zwischen 0 und 65535
- Ports zwischen 0 und 1023 (einschließlich) sind für Programme mit Superuser Rechten reserviert
- diese Ports sind im Gegensatz zu den IP-Adressen nicht eindeutig, d.h. auf unterschiedlichen Rechner werden i.A. unterschiedliche Programme mit den Ports assoziiert sein
- einige Ports sind jedoch standardisiert
- ...

Netzwerk Programmierung: Adressierung des Programms (Forts.)

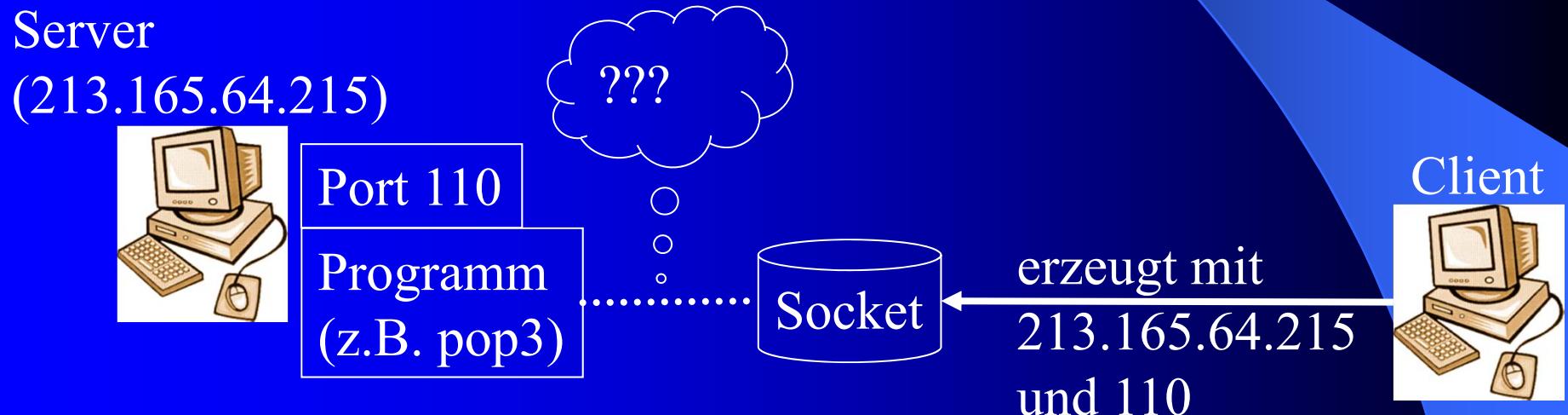
- ...

Name	Port	Beschreibung
echo	7	gibt jede Zeile zurück, die der Client schickt
daytime	13	liefert ASCII-String mit Datum und Uhrzeit
time	37	liefert die aktuelle Uhrzeit als Anzahl der Sekunden seit 1.1.1900
smtp	25	versenden von e-mails
pop3	110	übertragen von Mails
...		

- natürlich ist nicht jeder Port auf jedem Rechner belegt

Netzwerk Programmierung: Socket

- die Kommunikation erfolgt über Sockets (Abstraktion der Verbindung)
- werden durch die Klasse **Socket** realisiert



Netzwerk Programmierung: Socket (Forts.)

- Sockets können auf unterschiedliche Weise erzeugt werden

```
class Socket {  
    public Socket(String host, int port)  
        throws UnknownHostException, IOException;  
}
```

```
    public Socket(InetAddress host, int port)  
        throws IOException;
```

kann keine UnknownHostException werfen, da InetAddress dies bereits getan hat

wandelt host in eine InetAdresse um

Port, um das Programm anzusprechen

Go!

Netzwerk Programmierung: Socket (Beispiel)

```
import java.io.*;
import java.net.*;

public class Socket2 {

    public static void main(String[] args) {
        try {
            InetAddress inet = InetAddress.getByName("localhost");
            for(int i = 0;i < 1024;++i) {
                try {
                    Socket sock = new Socket(inet,i);
                    sock.close();
                    System.out.println("port " + i + " war erfolgreich");
                } catch (IOException e) {
                    System.out.println(i + " " +e);
                }
            }
        } catch (UnknownHostException e) {
            System.out.println(e);
        }
    }
}
```

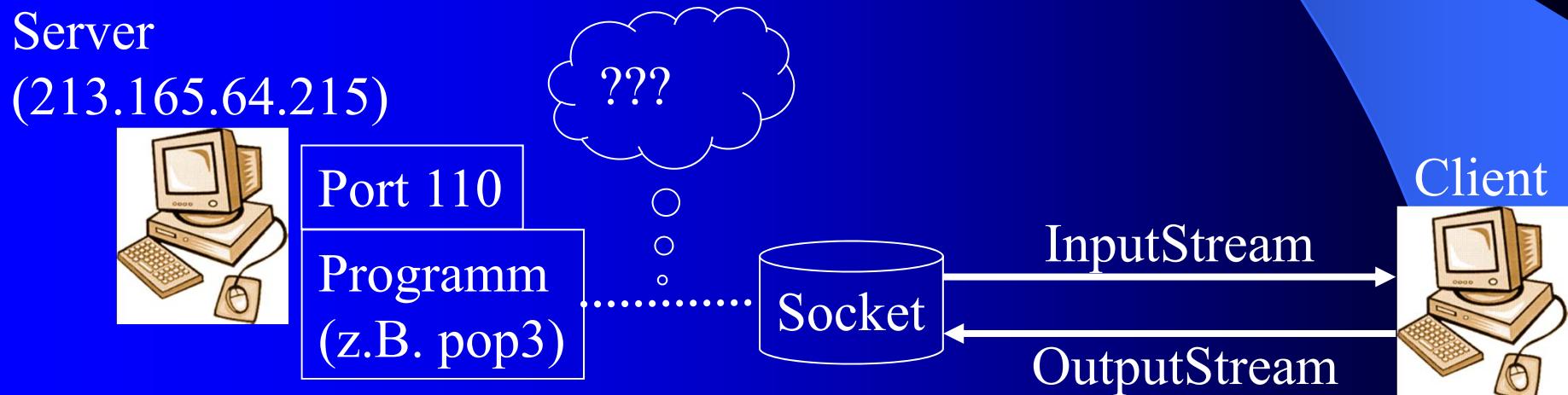
erzeuge die IP-Adresse des lokalen Rechners

versucht die ersten 1024 Ports zu kontaktieren

am Ende muss der Socket wieder geschlossen werden

Netzwerk Programmierung: Socket Kommunikation

- die Kommunikation mittels eines Sockets zwischen den beiden Programmen erfolgt durch Streams
- hierzu stellt ein Socket einen **Input- und einen Outputstream** zur Verfügung
- wie die Kommunikation abläuft, bestimmt der Server



Netzwerk Programmierung: Socket Kommunikation (Forts.)

- diese beiden Streams werden wie folgt generiert:

```
class Socket {  
    public InputStream getInputStream()  
        throws IOException;  
  
    public OutputStream getOutputStream()  
        throws IOException;  
}
```

öffnet den **InputStream**
zum Lesen vom Server

öffnet den **OutputStream**
zum Schreiben an den Server

Go!

Netzwerk Programmierung: Socket Lesen (Beispiel)

```
import java.io.*;
import java.net.*;

public class Socket3 {

    public static void main(String[] args) {
        final int DAY_TIME_PORT = 13;
        try {
            Socket sock = new Socket("localhost",DAY_TIME_PORT);
            InputStream in = sock.getInputStream();
            int len;
            byte[] buf = new byte[100];
            while ((len = in.read(buf)) != -1)
                System.out.write(buf,0,len);
            in.close();
            sock.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

versucht den day_time Port auf dem lokalen Rechner zu kontaktieren

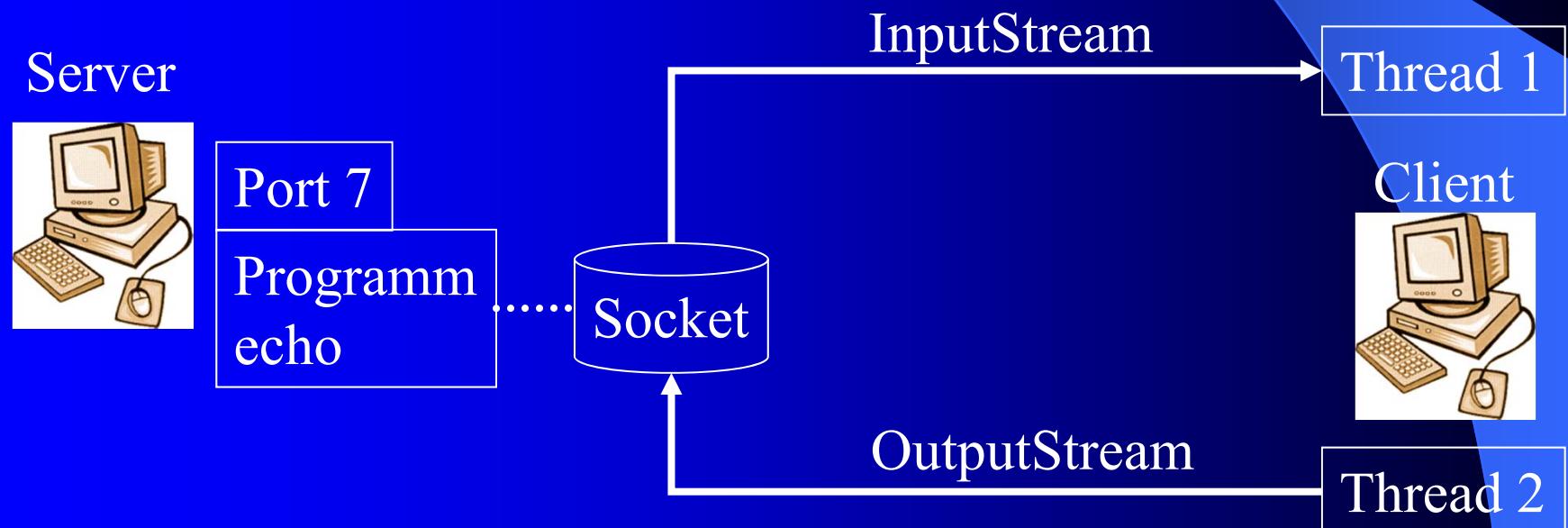
öffnet im Erfolgsfall einen Stream zum Lesen

liest solange in buf ein, bis nichts mehr kommt

Aufräumen am Ende

Netzwerk Programmierung: Socket lesen und schreiben

- soll von einem Socket gleichzeitig gelesen und geschrieben werden, muss das Programm aufgeteilt werden (Threads)
- der eine Thread bedient den InputStream
- der andere Thread bedient den OutputStream



Netzwerk Programmierung: Socket lesen und schreiben (Beispiel)

- Beispiel: echo Port
 - jedes Zeichen, dass dem echo Port geschickt wird, wird wieder zurückgeschickt
 - Aufgabe: von der Konsole sollen Zeichen eingelesen werden
 - diese Zeichen werden an den echo Port geschickt
 - ein anderer Thread liest von dem echo Port die zurückgeschickten Zeichen und stellt sie in einem Fenster da

Netzwerk Programmierung: Socket lesen / schreiben (Beispiel)

```
import javax.swing.*;
import java.io.*;
import java.net.*;

class Output extends JFrame implements Runnable {

    JTextArea m_area = new JTextArea(10,10);
    InputStream m_in;

    public void run() {
        int len;
        byte[] buf = new byte[100];
        try {
            while((len = m_in.read(buf)) != -1) {
                String s = new String(buf,0,len);
                m_area.append(s);
            }
        } catch (IOException e) {
        }
        dispose();
    }
    ...
}
```

das Ausgabefenster
mit einer TextArea

der InputStream
des Sockets

solange der Socket Zeichen
schickt: einlesen und der
TextArea anfügen

wird die Verbindung
unterbrochen, wird
das Fenster zerstört

Netzwerk Programmierung: Socket lesen / schreiben (Beispiel)

```
public Output(InputStream in) {  
    m_in = in;  
    new Thread(this).start(); ←  
    add(new JScrollPane(m_area));  
    pack();  
    setVisible(true);  
}  
  
}  
  
public class Socket4 {  
  
    public static void main(String[] args) {  
        final int PORT = 7;  
        try {  
            Socket sock = new Socket("localhost", PORT);  
            InputStream in = sock.getInputStream();  
            OutputStream out = sock.getOutputStream();  
            ...  
        }  
    }  
}
```

startet einen Thread für sich, um permanent die Zeichen vom **Socket** einzulesen

ein **Socket** für den Echo Service wird erzeugt

Go!

Netzwerk Programmierung: Socket lesen / schreiben (Beispiel)

...

```
new Output(in);
```

das Ausgabefenster

```
BufferedReader conin =
```

```
    new BufferedReader(new InputStreamReader(System.in));
```

```
boolean cont = true;
```

```
while (cont) {
```

```
    String line = conin.readLine();
```

```
    if (line.equals("quit")) {
```

```
        cont = false;
```

```
    } else {
```

```
        out.write(line.getBytes());
```

```
        out.write('\r');
```

```
        out.write('\n');
```

```
    }
```

```
}
```

```
out.close();
```

```
in.close();
```

```
sock.close();
```

```
} catch (Exception e) {
```

```
    System.out.println(e);
```

```
}
```

```
}
```

Zeilen einlesen
von der Tastatur

schicke die eingelesene
Zeile mit einem CR und
einem LF an den Socket

WICHTIG:
aufräumen am Ende

Vorlesung 14

Netzwerk Programmierung: Server / Client Verbindungsaufbau

- Wie wird ein Server (Programm !!!) programmiert, bei dem sich Clients (ebenfalls Programme !!!) über einen **Socket** anmelden können?
 1. Server stellt einen **Socket** zur Verfügung
 2. Server wartet darauf, dass ein Client sich an den **Socket** anmeldet
 3. Server kommuniziert mittels der **Input- und OutputStreams** des **Sockets** mit dem Client
- Server stellt den **Socket** mittels der Klasse **ServerSocket** zur Verfügung, der bei der Konstruktion die Portnummer bekommt

```
class ServerSocket {  
    public ServerSocket(int port) throws IOException;  
    Socket accept() throws IOException;      warten auf Client  
    ...
```

Netzwerk Programmierung: schematischer Ablauf

Zeit Server



ServerSocket(7)

warten(accept)

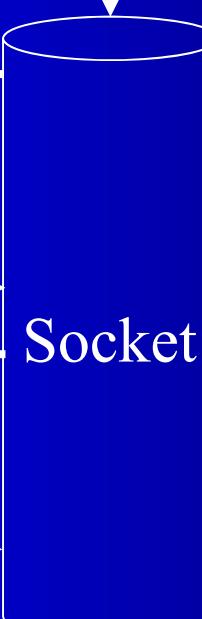
Blockierung
auflösen

schreiben → OutputStream
lesen ← InputStream

close

erzeugt

new Socket(„localhost“, 7)



← InputStream → lesen
→ OutputStream → schreiben

close



Go!

Netzwerk Programmierung: einfacher Socket (Beispiel)

```
public class Server2 {
```

```
    public static void main(String[] args) {
```

```
        final int PORT = 4000;
```

```
        Random rand = new Random();
```

```
        try {
```

```
            ServerSocket servSock = new ServerSocket(PORT);
```

```
            while (true) {
```

```
                Socket sock = servSock.accept();
```

```
                OutputStream out = sock.getOutputStream();
```

```
                try {
```

```
                    while (true)
```

```
                        out.write((char)'a' + rand.nextInt() % 26);
```

```
                } catch (IOException e) {
```

```
                    out.close();
```

```
                    sock.close();
```

```
                }
```

```
            }
```

```
        } catch (IOException e) {
```

```
            System.out.println(e);
```

```
        }
```

```
    }
```

Stelle als Server Port
4000 zur Verfügung

warte auf Client

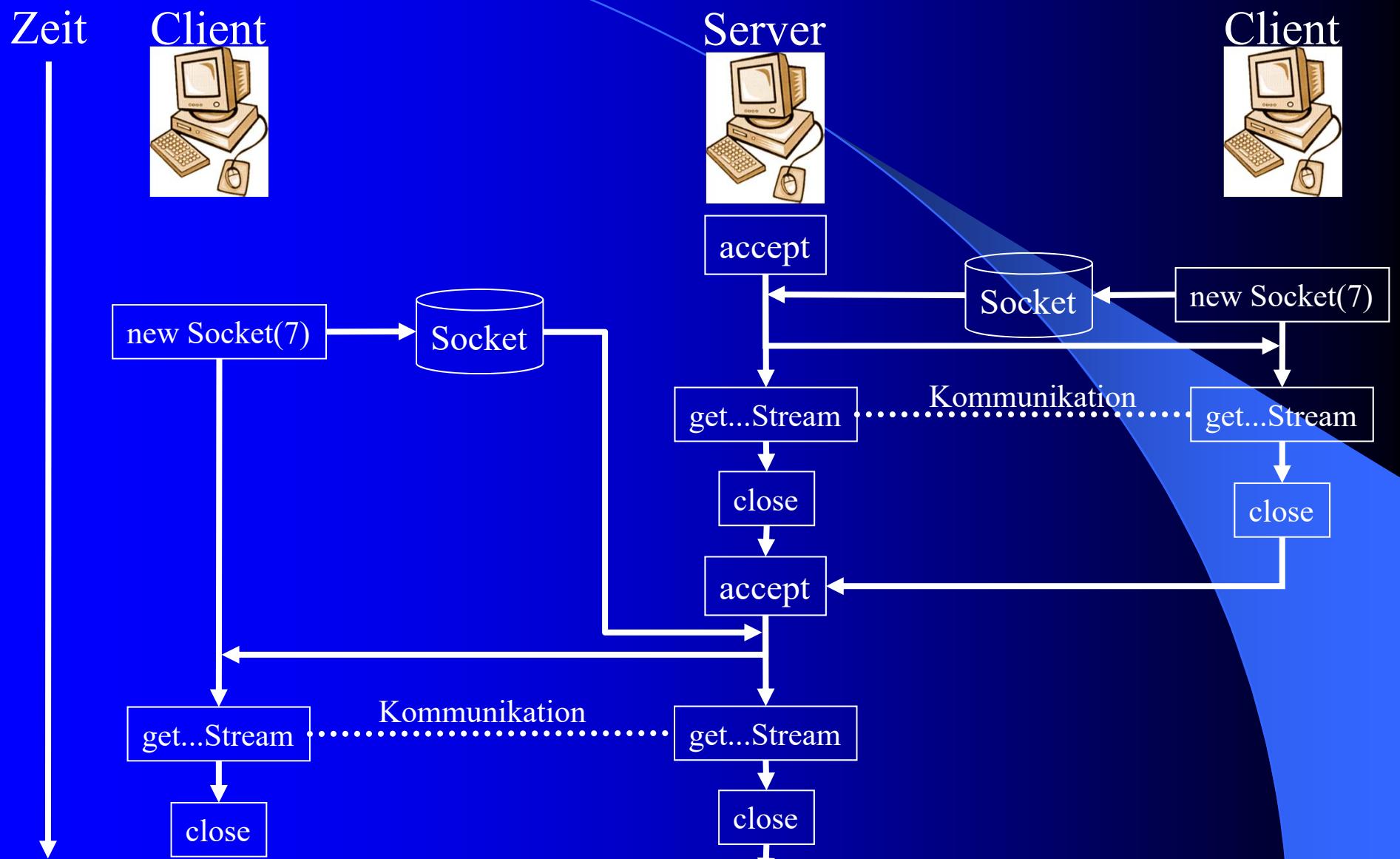
Client hat sich
erfolgreich angemeldet

Beendet der Client die
Verbindung, tritt beim
Schreiben ein Fehler auf

Netzwerk Programmierung: Server Problem

- Problem: Server kann nur einen Client zur Zeit bedienen
- jeder weiterer Client muss warten, bis der vorherige Client die Verbindung zum Server beendet hat
- beendet ein Client die Verbindung zum Server nie, wird der Server die komplette Zeit blockiert
- Folge: kein weiterer Client wird mehr bedient

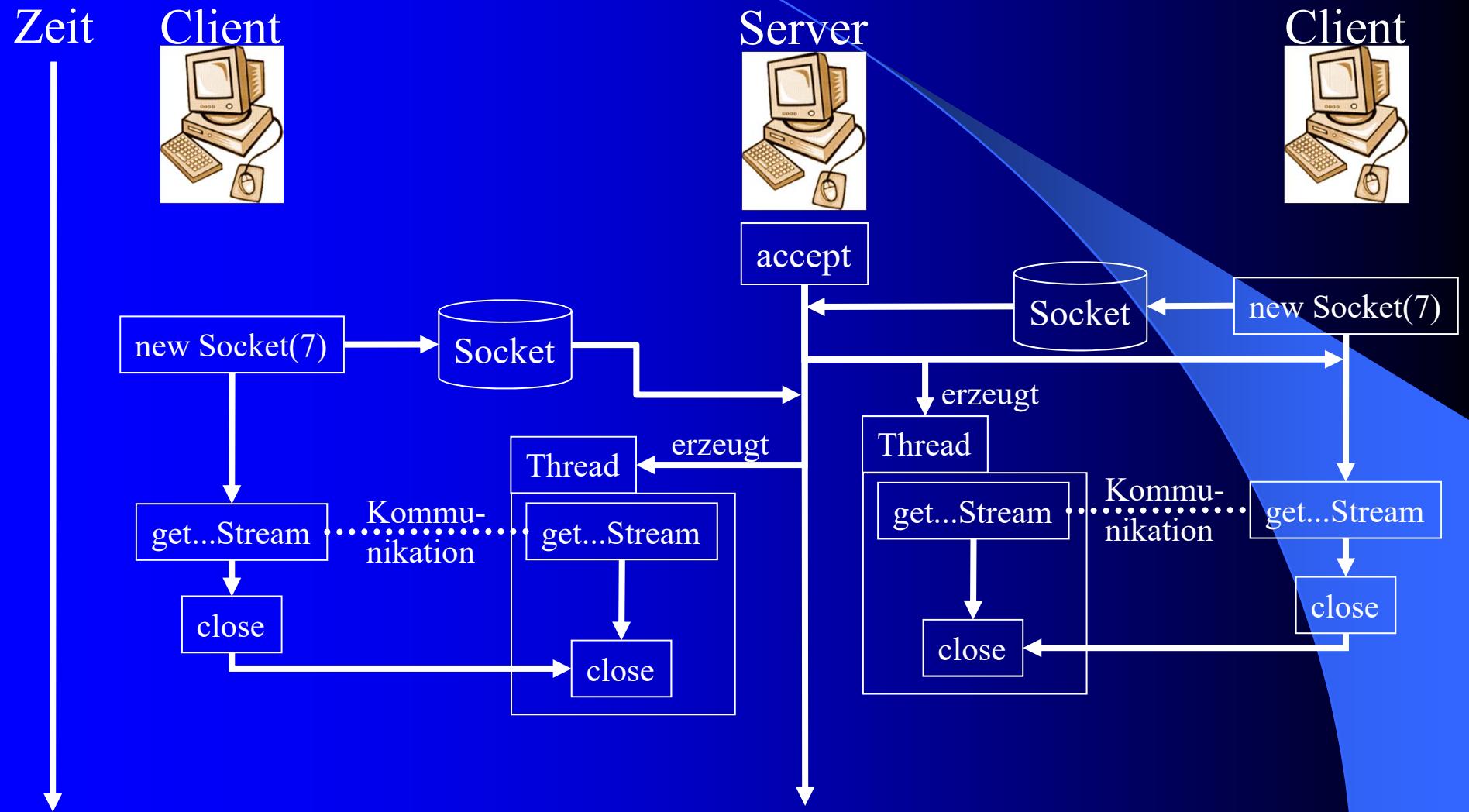
Netzwerk Programmierung: Server Problem (Fort.)



Netzwerk Programmierung: ein Server, mehrere Clients

- Lösung: ein Server muss für jeden Client einen eigenen Socket aufbauen
- dies muss parallel zueinander geschehen, um mehrere Clients parallel bedienen zu können
- dies erfolgt mittels Thread
- in jedem Thread wird ein Socket für exakt einen Client aufgebaut
- in dem Thread läuft die Kommunikation mit den Clients

Netzwerk Programmierung: ein Server, mehrere Clients (Fort.)



Netzwerk Programmierung: ein Server, mehrere Clients (Beispiel)

```
class ClientHandler extends Thread {  
    Socket m_sock;  
    Random rand = new Random();  
  
    public ClientHandler(Socket sock) {  
        m_sock = sock;  
        start();  
    }  
  
    public void run() {  
        try {  
            OutputStream out = m_sock.getOutputStream();  
            try {  
                while (true) {  
                    out.write((char)'a' + rand.nextInt() % 26);  
                }  
            } catch (IOException e) {  
                out.close();  
                m_sock.close();  
            }  
        } catch (IOException e) {}  
    }  
}
```

Der Server startet für jeden Client einen neuen Thread

schickt Zeichen an den Client, bis die Verbindung zusammenbricht (Exception)

wurde die Verbindung unterbrochen, muss der Stream und der Socket geschlossen werden; der Thread wird beendet

Go!

Netzwerk Programmierung: ein Server, mehrere Clients (Beispiel)

...

```
public class Server3 {  
  
    public static void main(String[] args) {  
        final int PORT = 4000;  
        try {  
            ServerSocket servSock = new ServerSocket(PORT);  
            while (true) {  
                Socket sock = servSock.accept();  
                new ClientHandler(sock);  
            }  
        } catch (IOException e) {  
            System.out.println(e);  
        }  
    }  
}
```

Stelle als Server Port
4000 zur Verfügung

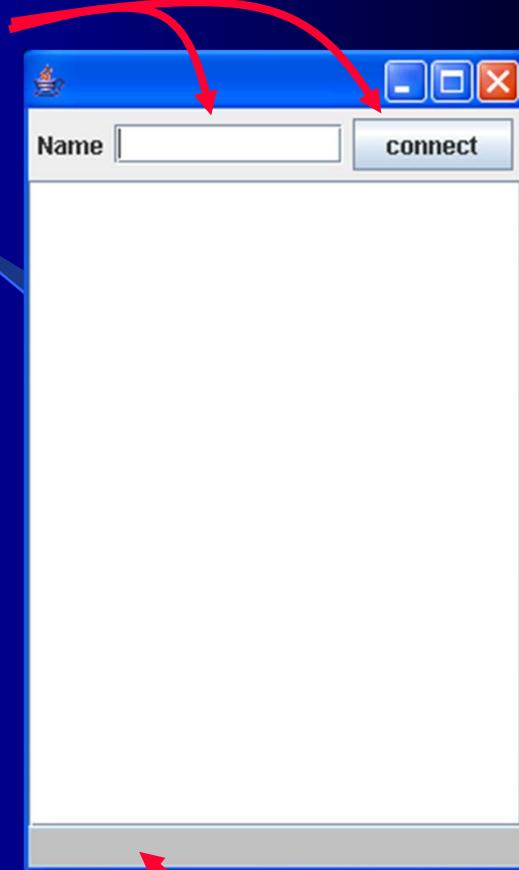
kommt ein neuer Client
hinzug, starte eigenen
neuen Client-Handler

Netzwerk Programmierung: ein Beispiel

- Aufgabe: ein Chat-Server und ein Chat-Client sollen programmiert werden
- Spezifikation Chat-Server:
 - es sollen mehrere Clients gleichzeitig bedient werden
 - ein Client schickt unmittelbar nach der Anmeldung einen String an den Server
 - dieser String wird in der folgenden Session als Name für diesen Client verwendet
 - jeder weitere String, den der Client an den Server schickt, wird an alle Clients zurückgeschickt
 - ...

Netzwerk Programmierung: ein Beispiel (Forts.)

- ...
- Spezifikation Chat-Client:
 - der Chat-Client soll eine graphische Oberfläche haben
 - Anmeldung kann nur mit einem nichtleeren Namen erfolgen
 - erst nach dem Anmelden an den Chat-Server ist es möglich, in dem unteren Bereich seinen Text anzugeben
 - nach einem Return erscheint der Text in allen angemeldeten Chat-Clients im Mittelteil



Netzwerk Programmierung: Chat-Server

```
public class ChatServer {  
    Vector<Socket> m_Clients = new Vector<Socket>();  
  
    class ClientHandler extends Thread {  
  
        Socket m_sock;  
        String m_name;  
        int len;  
        byte[] buf = new byte[100];  
  
        public ClientHandler(Socket sock) {  
            try {  
                m_sock = sock;  
                InputStream in = sock.getInputStream();  
                m_Clients.add(m_sock);  
                len = in.read(buf);  
                m_name = new String(buf, 0, len);  
                System.out.println("new client: " + m_name);  
                start();  
            } catch (IOException e) {  
                System.out.println(e);  
            }  
        }  
    }  
}
```

alle Sockets aller angemeldeten Clients

für jeden Client wird im Server ein neuer eigener Thread gestartet

registriere Socket zum Client

lese Nutzernamen des Clients ein

starte Thread für diesen Client

Netzwerk Programmierung: Chat-Server (Forts.)

...

```
public void run() {  
    try {  
        InputStream in = m_sock.getInputStream();  
        try {  
            while ((len = in.read(buf)) != -1)  
                sendAll(m_name, buf, len);  
        } catch (IOException e) {  
            in.close();  
            m_sock.close();  
        }  
        } catch (IOException e) {  
        }  
        System.out.println("client " + m_name + " removed");  
        m_Clients.remove(m_sock);  
    }  
}
```

lese vom Client
über den Socket

schicke diese
Nachricht an alle
Clients zusammen
mit dem Namen

Client hat die
Verbindung
unterbrochen

Am Ende: aufräumen, sprich
Client-Socket aus der Menge
aller Sockets entfernen

Netzwerk Programmierung: Chat-Server (Forts.)

```
...  
public void sendAll(String sender,byte[] msg,int len) throws IOException {  
    for(int i = 0;i < m_Clients.size();++i) {  
        Socket sock = m_Clients.get(i);  
        OutputStream out = sock.getOutputStream();  
        out.write(sender.getBytes());  
        out.write(':');  
        out.write(' ');  
        out.write(msg,0,len);  
        out.write('\r');  
        out.write('\n');  
    }  
}
```

schicke **msg** unter
Nennung von
sender an alle
Client-Sockets

```
public void waitAndConnect() throws IOException {  
    final int PORT = 4000;  
    ServerSocket servSock = new ServerSocket(PORT);  
    while (true)  
        new ClientHandler(servSock.accept());  
}
```

akzeptiere jede
einkommende
Socketverbindung
und starte neuen
Client Handler

```
public static void main(String[] args) throws IOException {  
    new ChatServer().waitAndConnect();  
}
```

Netzwerk Programmierung: Chat-Client

```
...  
public class ChatClient extends JFrame {
```

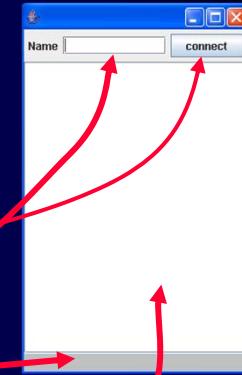
```
    JButton m_connect = new JButton("connect");  
    JTextField m_name = new JTextField(10);  
    JTextField m_message = new JTextField(20);  
    JTextArea m_textarea = new JTextArea(20,20);  
    Socket m_sock;
```

```
    final int PORT = 4000;
```

```
    class Listener implements Runnable {  
        InputStream m_in;
```

```
        public Listener(InputStream in) {  
            m_in = in;  
            new Thread(this).start();  
        }
```

```
    ...
```



horcht ständig den Socket nach
einkommenden Nachrichten ab

das passiert in einem
eigenen Thread

Netzwerk Programmierung: Chat-Client (Fort.)

...

```
public void run() {  
    int len;  
    byte[] buf = new byte[100];  
    try {  
        while((len = m_in.read(buf)) != -1)  
            print(new String(buf,0,len));  
    } catch (IOException e) {  
        print(e.toString());  
        disableMsg();  
    }  
}  
  
private void disableMsg() {  
    m_message.setEnabled(false);  
    m_message.setBackground(Color.LIGHT_GRAY);  
}  
...
```

Deaktivierung des
Message JTextFields

sobald eine Nachricht
reinkommt, drucke sie
mittels der eigenen print
Methode

sollte die
Socketverbindung
unterbrochen werden,
tritt eine Exception auf:
diese wird auch im
Fenster ausgegeben



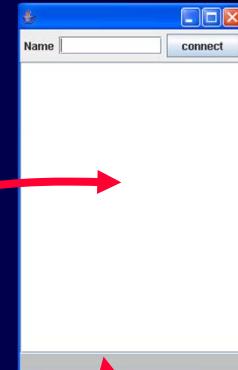
Netzwerk Programmierung: Chat-Client (Fort.)

```
...  
public ChatClient() {  
    JPanel content = new JPanel();  
    content.setLayout(new BorderLayout());  
    add(content);  
    JPanel control = new JPanel();  
    control.setLayout(new FlowLayout());  
    control.add(new JLabel("Name"));  
    content.add(BorderLayout.NORTH,control);  
    control.add(m_name);  
    control.add(m_connect);  
    m_textarea.setEditable(false);  
    content.add(BorderLayout.CENTER,new JScrollPane(m_textarea));  
    content.add(BorderLayout.SOUTH,m_message);  
    disableMsg();  
...}
```

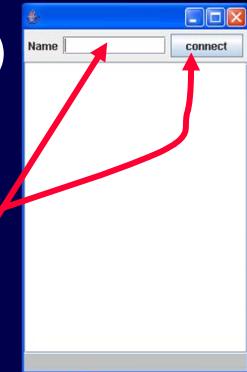
alles

nicht
editierbar

zunächst nicht
editierbar



Netzwerk Programmierung: Chat-Client (Fort.)



Name eingegeben und
gültige Verbindung?

...

```
m_connect.addActionListener(e -> {  
    if (m_name.getText().length() != 0 && connect()) {  
        m_message.setEnabled(true);  
        m_message.setBackground(Color.white);  
        m_connect.setEnabled(false);  
        m_name.setEnabled(false);  
    }  
});  
...
```

enable

disable

Netzwerk Programmierung: Chat-Client (Fort.)

```
...  
    m_message.addActionListener(e -> {  
        try {  
            m_sock.getOutputStream().write(m_message.getText().getBytes());  
            m_message.setText("");  
        } catch (IOException io) {}  
    }  
    );  
};  
  
addWindowListener(new WindowAdapter() {  
    public void windowClosing(WindowEvent e) {  
        try {  
            if (m_sock != null)  
                m_sock.close();  
            dispose();  
        } catch (IOException io) {}  
    }  
});  
pack();  
setVisible(true);  
}  
...  
lösche Message  
Window
```

schicke eingegebenen
Text an den Socket

wird das Fenster geschlossen,
muss der Socket (soweit
vorhanden) geschlossen werden

Go!

Netzwerk Programmierung: Chat-Client (Fort.)

...

```
private boolean connect() {  
    try {  
        m_sock = new Socket("localhost",PORT);  
        new Listener(m_sock.getInputStream());  
        m_sock.getOutputStream().write(m_name.getText().getBytes());  
        return true;  
    } catch (Exception e) {  
        print(e.toString() + '\n');  
        return false;  
    }  
}  
  
private void print(String msg) {  
    m_textarea.append(msg);  
}  
  
public static void main(String[] args) {  
    new ChatClient();  
}
```

baue Verbindung auf,
schicke **m_name** zum
ChatServer

füge den Text dem
Hauptfenster hinzu
und zeichne es neu

