

# **Grundlagen der Programmierung II SS 2005**

**Dr. Michael Thies**

# Ziele der Vorlesung

## Ziele der Vorlesung Grundlagen der Programmierung II

Die Studierenden sollen

- **graphische Bedienungsoberflächen** mit objektorientierten Techniken entwickeln können,
- die Grundlagen **paralleler Prozesse und deren Synchronisation** verstehen und parallele Prozesse in Java programmieren lernen.
- ihre Kenntnisse in der objektorientierten Programmierung in Java festigen und verbreitern.

## Voraussetzungen aus Grundlagen der Programmierung I:

Die Studierenden sollen

- die **Programmentwicklung in Java von Grund auf** erlernen.
- lernen, Sprachkonstrukte sinnvoll und mit **Verständnis** anzuwenden.
- grundlegende **Konzepte der objektorientierten Programmierung** verstehen und anzuwenden lernen. Objektorientierte Methoden haben zentrale Bedeutung im **Software-Entwurf** und in der **Software-Entwicklung**.
- lernen, **Software aus objektorientierten Bibliotheken wiederzuverwenden**.
- **eigene praktische Erfahrungen** in der Entwicklung von **Java-Programmen** erwerben. Darauf bauen größere praktische Entwicklungen in Java oder anderen Programmiersprachen während des Studiums und danach auf.

# Inhalt

<i>Nr. d. Vorl.</i>	<i>Inhalt</i>	<i>Abschnitte in „Java lernen, 2. Auflage“</i>
1	1. Einführung, GUI, Swing (AWT)	10.1
2	2. Zeichenflächen	10.2
3	3. Komponenten erzeugen und platzieren	10.3
4	4. Hierarchisch strukturierte Fensterinhalte	
	5. Ereignisse an graphischen Benutzungsoberflächen	10.3, 11.1
5	Eingabeverarbeitung	11.1
6	6. Beispiel: Ampelsimulation	10.5
7	7. Entwurf von Ereignisfolgen	11.4
8	8. Model/View-Paradigma für Komponenten	—
9	9. Java-Programme in Applets umsetzen	12.1, 12.2
10	10. Parallele Prozesse, Grundbegriffe, Threads	13.1, 13.2
11	11. Unabhängige parallele Prozesse,	13.1, 13.2
12	12. Monitore, Synchronisation gegenseitiger Ausschluss	13.3
13	13. Bedingungssynchronisation im Monitor	
14	14. Verklemmungen, Beispiel: Dining Philosophers	
15	15. Zusammenfassung	

# Literaturhinweise

Elektronisches Skript zur Vorlesung:

- **M. Thies: Vorlesung GP II, 2005, <http://ag-kastens.upb.de/lehre/material/gpii>**
- **U. Kastens: Vorlesung SWE II, 2004, <http://ag-kastens.upb.de/lehre/material/sweii>**
- **U. Kastens: Vorlesung SWE, 1998/99 (aktualisiert), <http://.../swei>**

fast ein Textbuch zur Vorlesung, mit dem Vorlesungsmaterial (in älterer Version) auf CD:

- **J. M. Bishop: Java lernen, Addison-Wesley, 2. Auflage, 2001**
- **J. M. Bishop: Java Gently - Programming Principles Explained, Addison-Wesley, 1997 3rd Edition (Java 2)**

zu allgemeinen Grundlagen der Programmiersprachen in der Vorlesung:

- **U. Kastens: Vorlesung Grundlagen der Programmiersprachen, Skript, 2003 <http://ag-kastens.upb.de/lehre/material/gdp>**
- **D. A. Watt: Programmiersprachen - Konzepte und Paradigmen, Hanser, 1996**

eine Einführung in Java von den Autoren der Sprache:

- **Arnold, Ken / Gosling, James: The Java programming language, Addison-Wesley, 1996.**
- **Arnold, Ken / Gosling, James: Die Programmiersprache Java™, 2. Aufl. Addison-Wesley, 1996**

# Elektronisches Skript: Startseite

Vorlesung Grundlagen der Programmierung 2 SS 2005

http://ag-kastens.upb.de/lehre/material/gpii/ Google

**UNIVERSITÄT PADERBORN**  
Die Universität der Informationsgesellschaft

Fachgruppe Kastens > Lehre > Grundlagen der Programmierung 2 SS 2005

**Vorlesung Grundlagen der Programmierung 2 SS 2005**

**Vorlesungsfolien**

- Kapitelübersicht
- Folienverzeichnis
- Drucken

**Übungsaufgaben**

- Aufgabenblätter
- Drucken

**Organisation**

- Allgemeines
- Aktuelle Hinweise

12.04.2005	Vorlesungsbeginn
12.04.2005	Anmeldung zu den Übungen

**Wissenswertes**

- Ziele
- Literatur
- Java 1.4 Dokumentation im Web
- Inhalt Java Lernen/Java Gently
- Material zu GP1 (WS 2004/2005)
- Material zu SWE2 (SS 2004, AWT statt Swing)

SUCHEN:

Generiert mit Camelot | Probleme mit Camelot? | Geändert am: 01.04.2005

# Elektronisches Skript: Folien im Skript

Grundlagen der Programmierung 2 SS 2005 - Folie 87

http://ag-kastens.upb.de/lehre/material/gpii/folien/Folie87.html

UNIVERSITÄT PADERBORN  
Die Universität der Informationsgesellschaft

Fachgruppe Kastens > Lehre > Grundlagen der Programmierung 2 SS 2005 > Folienverzeichnis >

Hauptseite  
Kapitelübersicht  
Folienverzeichnis  
Vorherige Folie  
Nächste Folie  
Folienpaket drucken

SUCHEN:

## Grundlagen der Programmierung 2 SS 2005 - Folie 87

### Graphische Darstellung von Swing-Komponenten

GP-47

The image displays various Java Swing components arranged in a grid-like fashion. On the left, there are labels for JLabel, JButton (with 'Cancel' and 'OK' buttons), JCheckBox, JRadioButton, JComboBox (with 'Maurit' and 'Lena' options), JList (with 'Compusoft', 'MetaTech', 'Terasoft', and 'Microsoft' items), and JTextField. In the center is a large JScrollPane containing a tiger image. To the right are JPanel (with 'one Two' text), JFrame (a window frame), JOptionPane (a dialog box with 'What is your favorite movie?'), JFileChooser (an 'Open' dialog showing a file list), and JSlider (a vertical slider from -100 to 100).

Autoren: Dr. Michael Thies und Prof. Dr. Uwe Kastens

Generiert mit Camelot | Probleme mit Camelot? | Geändert am: 04.04.2005

Open "http://ag-kastens.upb.de/lehre/material/gpii/folien/Folie87.html" in a new tab

**Ziele:**  
Anschauliche Vorstellung der Komponenten

**in der Vorlesung:**  
Funktion der Komponenten erläutern

**nachlesen:**  
Judy Bishop: Java lernen, 2.Aufl., Abschnitt 10.2

**Verständnisfragen:**

- Welche Komponenten enthalten wiederum Komponenten?
- Welche Komponenten kommen auf Folie 86 vor?

# Elektronisches Skript: Organisation der Vorlesung

Grundlagen der Programmierung 2 SS 2005 – Organisation

http://ag-kastens.upb.de/lehre/material/gpii/organisation.html

UNIVERSITÄT PADERBORN  
Die Universität der Informationsgesellschaft

Fachgruppe Kastens > Lehre > Grundlagen der Programmierung 2 SS 2005 > Organisation

Hauptseite  
Hinweise  
Mein Konto

SUCHE

## Grundlagen der Programmierung 2 SS 2005 - Organisation

### Personen

**Sprechstunde Michael Thies:**

- Di 13.00 - 19.00 P2.303

**Übungsbetreuer:**

- Frank Götz
- Theo Lettmann
- Björn Metzler
- Steffen Priesterjahn

### Termine

#### Vorlesung

- Di 09.15 - 10.45 AudiMax
- Fr 09.15 - 10.45 AudiMax

**Beginn: Dienstag, 12. April 2005**

StudInfo: Grundlagen der Programmierung I, WS 2004 / 2005

Prof. Dr. Stefan Gerlach, Fakultät für Informatik, Universität Paderborn

Kosten einrichten  
Kosten einrichten  
Stellung besetzen

Wissens

Bitte kontaktieren:

Bitte beachten: Dieser Dienst wird für die Bearbeitung von Anfragen für den Kurs einrichten.

Programmierer: Hier sind Ihre Programmierdaten zu sehen

Grundlagen der Programmierung 2 SS 2005 – Organisation

ag-kastens.upb.de/lehre/material/gpii/organisation.html

### Zentralübung

- Fr 13.00 - 13.45 C1

**Beginn: Freitag, 22. April 2005**  
**Ende: Freitag, 3. Juni 2005**

### Übungen

• Gruppe 1	Mo 9 - 11	E4.101	Björn Metzler
• Gruppe 2	Mo 14 - 16	E3.301	Theo Lettmann
• Gruppe 3	Mo 14 - 16	E4.101	Frank Götz
• Gruppe 4	Mo 16 - 18	E3.301	Theo Lettmann
• Gruppe 5	Mo 16 - 18	E4.101	Frank Götz
• Gruppe 6	Di 11 - 13	E4.101	Björn Metzler
• Gruppe 7	Di 14 - 16	E4.101	Björn Metzler
• Gruppe 8	Di 16 - 18	E4.101	Björn Metzler
• Gruppe 9	Mi 9 - 11	E4.101	Steffen Priesterjahn
• Gruppe 10	Mi 11 - 13	E3.301	Steffen Priesterjahn
• Gruppe 11	Mi 11 - 13	E4.101	Theo Lettmann
• Gruppe 12	Mi 16 - 18	E3.301	Steffen Priesterjahn
• Gruppe 13	Mi 16 - 18	E4.101	Theo Lettmann
• Gruppe 14	Do 9 - 11	E4.101	Frank Götz
• Gruppe 15	Do 11 - 13	E3.301	Steffen Priesterjahn
• Gruppe 16	Do 11 - 13	E4.101	Frank Götz

**Beginn: Montag, 18. April 2005**  
**Ende: Donnerstag, 2. Juni 2005**

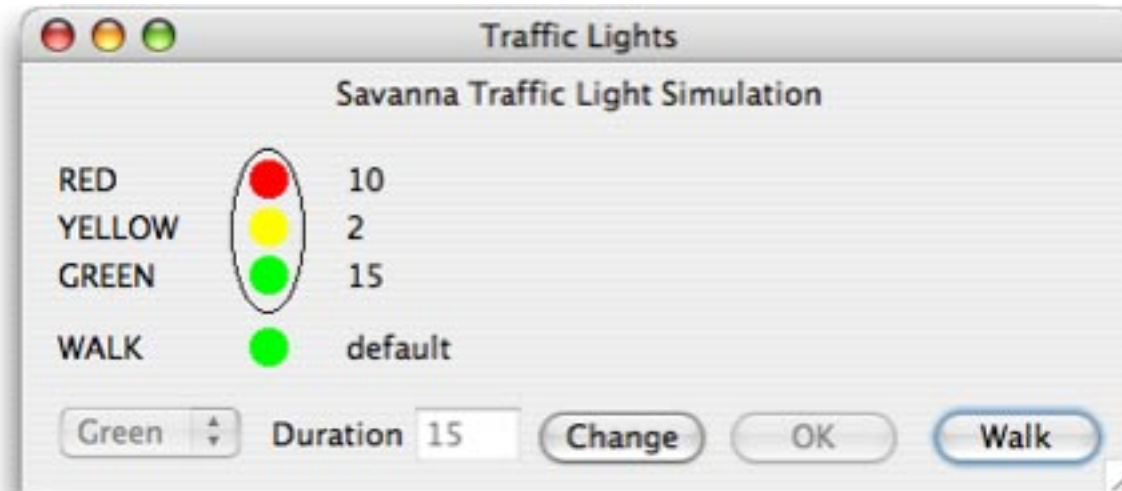
### Klausurtermine (GP1+GP2)

- 1. Klausur Di 02.06.2005 09.00 - 12.00 Sporthalle,

# 1. Einführung in graphische Benutzungsoberflächen

**Graphische Benutzungsoberflächen** (graphical user interfaces, **GUI**) dienen zur

- interaktiven Bedienung von Programmen,
- Ein- und Ausgabe mit graphischen Techniken und visuellen Komponenten



## **Javas Standardbibliothek `javax.swing`**

(Java foundation classes, JFC) enthält wiederverwendbare Klassen zur Implementierung und Benutzung der wichtigsten GUI-Komponenten:

- Graphik
- GUI-Komponenten (siehe GP-87)
- Platzierung, Layoutmanager
- Ereignisbehandlung (`java.awt.event`)
- baut auf dem älteren AWT (abstract windowing toolkit) auf

# Graphische Darstellung von Swing-Komponenten

JLabel JLabel

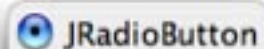
JButton



JCheckBox JCheckBox



JRadioButton



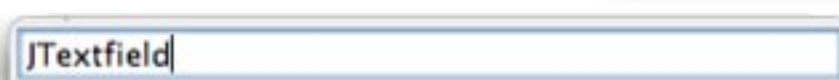
JComboBox



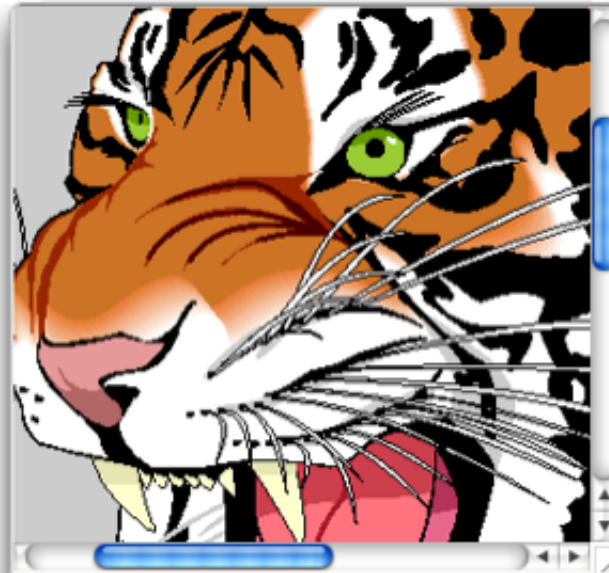
JList



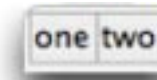
JTextField



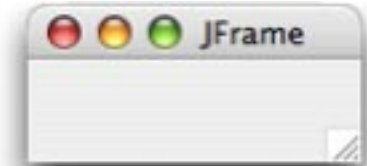
JScrollPane



JPanel



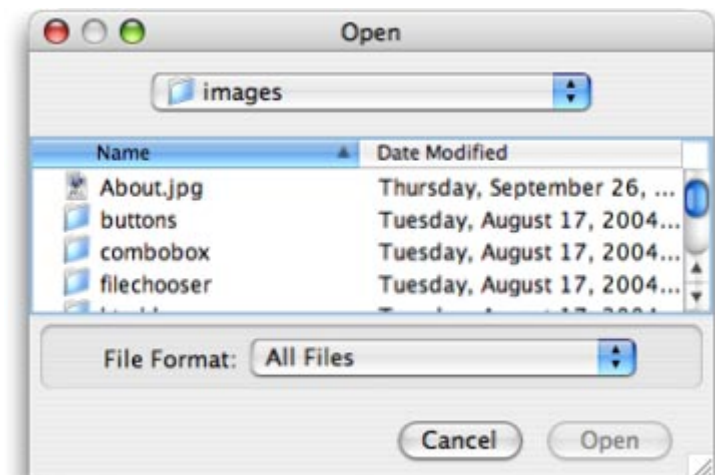
JFrame



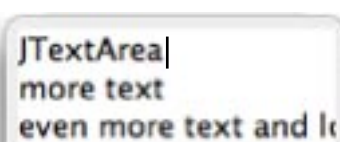
JOptionPane



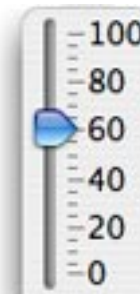
JFileChooser



JTextArea



JSlider



# Klassenhierarchie für Komponenten von Benutzungsoberflächen

Teil der erweiterten Standardbibliothek `javax.swing` (Java foundation classes, JFC)

Klasse in der Hierarchie	Kurzbeschreibung
Component (abstrakt, AWT)	darstellbare Komponenten von Benutzungsoberflächen
Container (abstrakt, AWT)	Behälter für Komponenten
Window (AWT)	Fenster (ohne Rand, Titel, usw.); Wurzel der Objektbäume
Frame (AWT)	Fenster mit Rand, Titel, usw.
JFrame	Swing-Fenster mit Rand, Titel, usw.
JComponent (abstrakt)	darstellbare Swing-Komponenten
JPanel	konkrete Klasse zu Container, Behälter für Komponenten
JScrollPane	Sicht auf große Komponente, 2 Rollbalken
JFileChooser	Fenster zur interaktiven Dateiauswahl
AbstractButton (abstr.)	Komponenten, die auf einfachen Klick reagieren
JButton	Schaltfläche ("Knopf")
JToggleButton	Komponenten mit Umschaltverhalten bei Klick
JCheckBox	An/Aus-Schalter ("Ankreuzfeld"), frei einstellbar
JRadioButton	An/Aus-Schalter, symbolisiert gegenseitigen Ausschluß
JComboBox	Auswahl aus einem Aufklappmenü von Texten
JList	Auswahl aus einer mehrzeiligen Liste von Texten
JSlider	Schieberegler zur Einstellung eines ganzzahligen Wertes
JLabel	Textzeile zur Beschriftung, nicht editierbar
JTextComponent (abstr.)	editierbarer Text
JTextField	einzelne Textzeile
JTextArea	mehrzeiliger Textbereich

# Klassenhierarchie der Swing-Bibliothek

Legende:



Interface

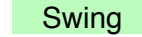


AWT

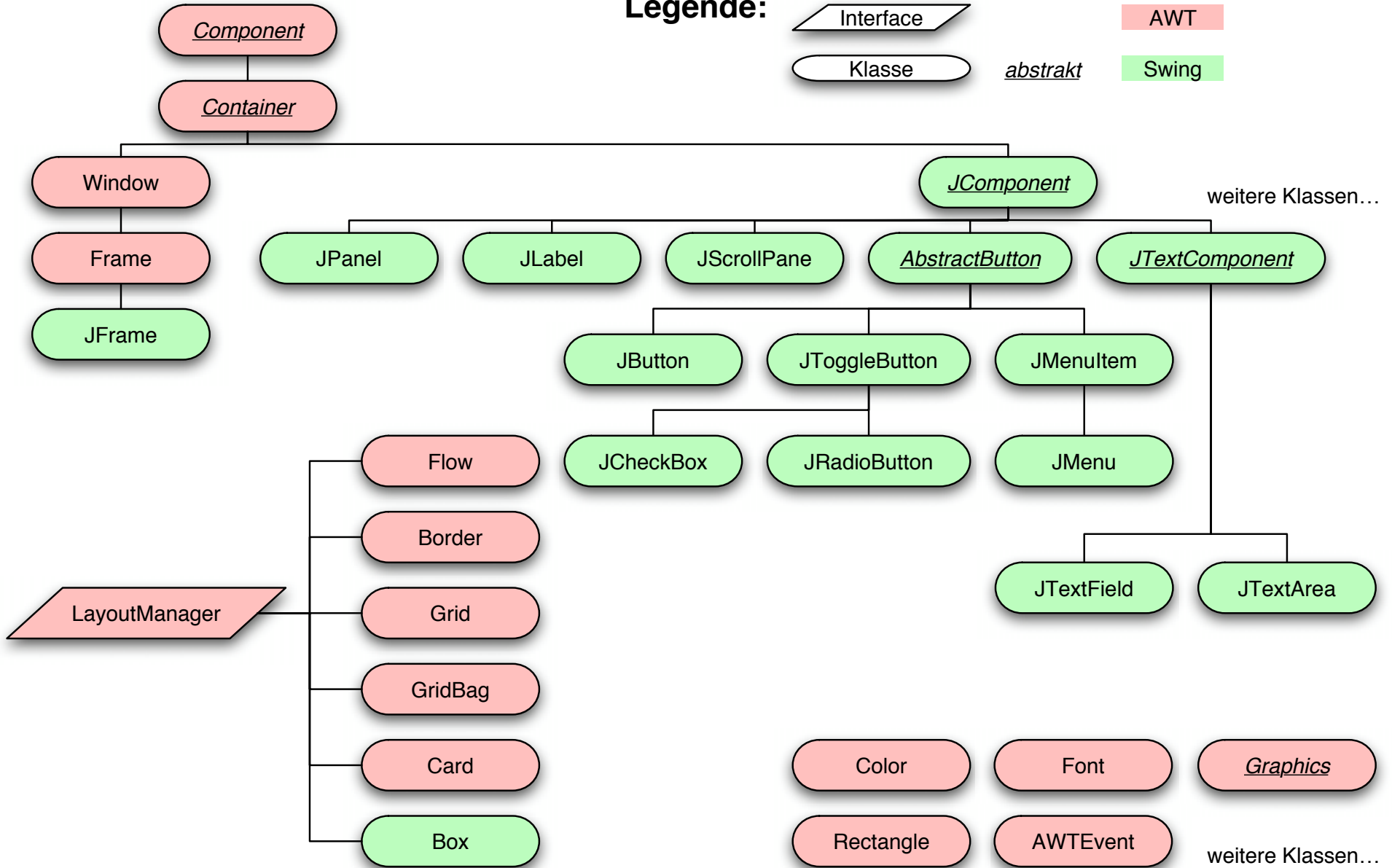


Klasse

*abstrakt*



Swing



weitere Klassen...

weitere Klassen...

weitere Klassen...

analog zu Fig. 10.1 aus  
Java Gently, 3rd ed, p. 385

# Wiederverwendung von Klassen aus Bibliotheken

Die Klasse `javax.swing.JFrame` implementiert **gerahmte Fenster** in graphischen Benutzungsoberflächen (GUI). Sie ist eine Blattklasse in der Hierarchie der GUI-Komponenten:

<code>java.lang.Object</code>	
<code>java.awt.Component</code>	GUI-Komponenten
<code>java.awt.Container</code>	solche, die wieder Komponenten enthalten können
<code>java.awt.Window</code>	Fenster ohne Rahmen
<code>java.awt.Frame</code>	Fenster mit Rahmen (AWT)
<code>javax.swing.JFrame</code>	Fenster mit Rahmen (Swing)

**Methoden** zum Zeichnen, Platzieren, Bedien-Ereignisse Behandeln, etc. sind auf den jeweils passenden Hierarchieebenen implementiert.

In der `abstract class Component` ist die Methode

```
public void paint (Graphics g)
```

definiert, aber nicht ausgefüllt. Mit ihr wird auf der Fläche des Fensters gezeichnet.

**Benutzer definieren Unterklassen von JFrame**, die die Funktionalität der Oberklassen erben. Die Methode `paint` wird **überschrieben** mit einer Methode, die das Gewünschte zeichnet:

```
public class Rings extends JFrame
{
  public Rings () { super("Olympic Rings"); setSize(300, 150); }
  public void paint (Graphics g) { /* draw olympic rings ... */ }
  public static void main (String[] args)
  {
    JFrame f = new Rings(); ...
  }
}
```

# Einige Eigenschaften auf den Ebenen der JFrame-Hierarchie

Klasse	Datenelemente	Ereignisse	Methoden
Component	Location, Size, Bounds, Visible	Key, Mouse, Focus, Component	paint
Container	Layout	Container	add, getComponents, paint
Window	Locale	Window	setVisible, pack, toBack, toFront
Frame	Title, MenuBar, Resizable, IconImage		
JFrame	ContentPane, RootPane, DefaultCloseOperation		

Namenskonventionen:

zum Datenelement *XXX*  
gibt es die Methoden  
**get~~XXX~~** und ggf. **set~~XXX~~**

Ereignisse sind Objekte  
der Klassen **YYEvent**

## Vergleich: Swing und AWT

	Swing	AWT
zu finden im Java-Paket	<code>javax.swing.*</code> optionale, aber standardisierte <u>J</u> ava <u>E</u> xtension seit Java 1.2 bzw. 1.1	<code>java.awt.*</code> Teil der Java Standard Edition seit Java 1.0
Zeichnen der GUI-Komponenten	in Java implementiert (leichtgewichtige Komp.), nur Fenster vom Betriebssystem verwaltet	durch das Betriebssystem, in <i>Peer</i> -Objekten gekapselt (schwergewichtige Komp.)
(visuelle) Rückmeldungen für Benutzeraktionen	in Java implementiert, außer Manipulation ganzer Fenster	durch das Betriebssystem realisiert
Reaktion auf Benutzeraktionen im Programm	in Java implementiert durch sog. <i>Listener</i> -Objekte	in Java implementiert durch sog. <i>Listener</i> -Objekte
angebotene GUI-Komponenten	einfach bis sehr komplex: z.B. Schaltflächen, Tabellen, Baumstrukturen zum Aufklappen	einfach bis mittel: Schnittmenge über verschiedene Betriebssysteme, z.B. Listen, aber keine Tabellen

# Look&Feel-Module in Swing

Es gibt mehrere Sätze von Java-Klassen, die die Swing-Komponenten unterschiedlich grafisch darstellen. Ein vollständiger Satz von Klassen für alle Komponenten bildet ein Look&Feel-Modul:

- Look&Feel eines Programms ist beim Start des Programms frei wählbar oder sogar während der Laufzeit unmittelbar umschaltbar.
- Das generische Java-Look&Feel-Modul *Metal* ist auf jedem System verfügbar.
- Hinzu kommen unterschiedlich genaue Imitationen verschiedener Betriebssysteme.
- **Realisierung:** Look&Feel-Modul enthält eine konkrete Unterklasse für jede abstrakte Look&Feel-Klasse in Swing.



*Metal*



*Aqua (MacOS X)*



*Motif*



*JGoodies  
Plastic XP*

## 2. Zeichenflächen benutzen, Programmschema

```
import javax.swing.JFrame; import java.awt.*;
    // Hauptklasse als Unterklasse von JFrame:
public class GraphicWarning extends JFrame
{   GraphicWarning (String title)                // Konstruktor
    {   super (title);                          // Aufruf des Konstruktors von JFrame

}

    public void paint (Graphics g)              // überschreibt paint in einer Oberklasse
    {   super.paint(g);                          // Hintergrund der Fensterfläche zeichnen

}

    public static void main (String[] args)
    {   JFrame f = new GraphicWarning ("Draw Warning"); // Objekt erzeugen
    }
}
```



# Programmschema: Eigenschaften und Ereignisbehandlung

```

import javax.swing.JFrame; import java.awt.*;
    // Hauptklasse als Unterklasse von JFrame:
public class GraphicWarning extends JFrame
{   GraphicWarning (String title)                // Konstruktor
    {   super (title);                            // Aufruf des Konstruktors von JFrame
        Container content = getContentPane();    // innere Fensterfläche
        content.setBackground (Color.cyan);      // Farbe dafür festlegen
        setSize (35*letter, 6*line);           // Größe festlegen
        setDefaultCloseOperation (EXIT_ON_CLOSE);
        // Verhalten des Fensters: Beim Drücken des Schließknopfes Programm beenden.
        setVisible (true);                      // führt zu erstem Aufruf von paint
    }
    private static final int line = 15, letter = 5; // zur Positionierung
    public void paint (Graphics g)                // überschreibt paint in einer Oberklasse
    {   super.paint(g);                          // Hintergrund der Fensterfläche zeichnen
    }

    public static void main (String[] args)
    {   JFrame f = new GraphicWarning ("Draw Warning"); // Objekt erzeugen
    }
}

```



## Programmschema: paint-Methode ausfüllen

```

import javax.swing.JFrame; import java.awt.*;
    // Hauptklasse als Unterklasse von JFrame:
public class GraphicWarning extends JFrame
{   GraphicWarning (String title)                // Konstruktor
    {   super (title);                            // Aufruf des Konstruktors von JFrame
        Container content = getContentPane();    // innere Fensterfläche
        content.setBackground (Color.cyan);      // Farbe dafür festlegen
        setSize (35*letter, 6*line);            // Größe festlegen
        setDefaultCloseOperation (EXIT_ON_CLOSE);
        // Verhalten des Fensters: Beim Drücken des Schließknopfes Programm beenden.
        setVisible (true);                      // führt zu erstem Aufruf von paint
    }
private static final int line = 15, letter = 5; // zur Positionierung
public void paint (Graphics g) // überschreibt paint in einer Oberklasse
{   super.paint(g); // Rest der Fensterfläche zeichnen (Hintergrund)
    g.drawRect (2*letter, 2*line, 30*letter, 3*line); // auf der Fläche g
    g.drawString ("W A R N I N G", 9*letter, 4*line); // zeichnen und
                                                    // schreiben

public static void main (String[] args)
{   JFrame f = new GraphicWarning ("Draw Warning"); // Objekt erzeugen
}
}

```



# Ablauf des Zeichen-Programms

## 1. `main` aufrufen:

### 1.1. `GraphicWarning`-Objekt erzeugen, Konstruktor aufrufen:

1.1.1 `JFrame`-Konstruktor aufrufen

1.1.2 `Container`-Objekt für innere Fensterfläche abfragen,

1.1.3 Eigenschaften setzen, z. B. Farben,

1.1.4 ggf. weitere Initialisierungen des Fensters

1.1.5 Größe festlegen, `setSize(..., ...)`,

1.1.6 Verhalten des Schließknopfes festlegen, `setDefaultCloseOperation`,

1.1.7 Fenster sichtbar machen, `setVisible(true)`

**parallele Ausführung von (2) initiieren**

1.2 Objekt an `f` zuweisen

1.3 ggf. weitere Anweisungen zur Programmausführung



in Methoden der Oberklassen:

## 2. `Graphics` Objekt erzeugen

2.1 damit erstmals `paint` aufrufen (immer wieder, wenn nötig):

weiter in `paint` von `GraphicWarning`

2.1.1 Methode aus Oberklasse den Hintergrund der Fensterfläche zeichnen lassen

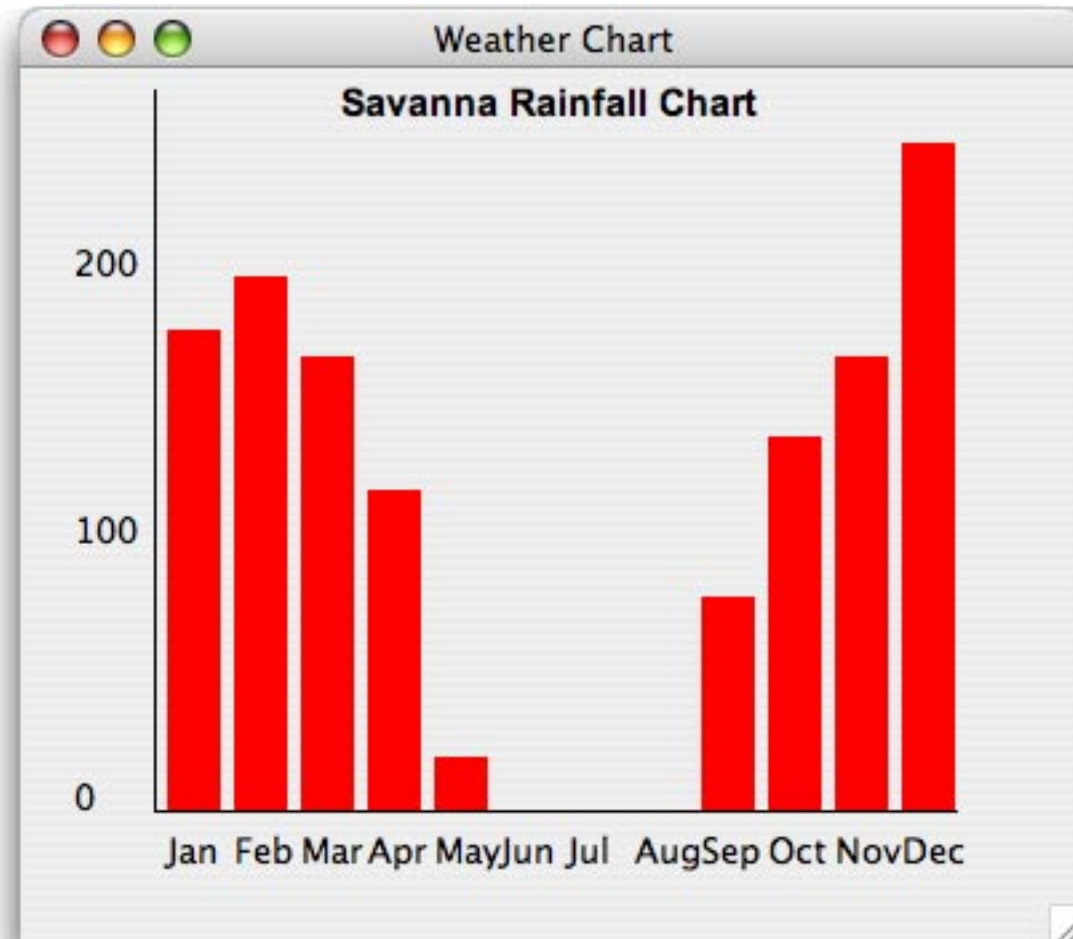
2.1.2 auf der Zeichenfläche des Parameters `g` schreiben und zeichnen

## 3. Schließknopf Drücken

in Methoden der Oberklassen:

3.1 Programm beenden, `System.exit(0)`

# Beispiel: Balkendiagramm zeichnen



## Beispiel: Balkendiagramm zeichnen

```

public void paint (Graphics g)
{
    super.paint(g);                                // Hintergrund zeichnen
    int x = 50, y = 300;                            // Schnittpunkt der Achsen
    int width = 20, gap = 5;                        // Balken und Zwischenraum

    g.drawLine (x, y, x+12*(width+gap), y);         // x-Achse
    g.drawLine (x, y, x, 30);                      // y-Achse

    for (int m = 0; m < 12; m++)                   // Monate an der x-Achse
        g.drawString(months[m], m*(width+gap)+gap+x, y+20);

    for (int i = 0; i < y; i+=100)                 // Werte an der y-Achse
        g.drawString(String.valueOf(i), 20, y-i);

    g.setFont(new Font("SansSerif", Font.BOLD, 14)); // Überschrift
    g.drawString("Savanna Rainfall Chart", 120, 40);

    g.setColor(Color.red);                         // die Balken
    for (int month = 0; month < 12; month++)
    {
        int a = (int) rainTable[month]*10;
        g.fillRect(month*(width+gap)+gap+x, y-a, width, a);
    }
}

private double[] rainTable = new double[12];
private static String months [] = {"Jan", "Feb", "Mar", ..., "Oct", "Nov", "Dec"};

```

### 3. Swing-Komponenten erzeugen und platzieren

Ein einfaches Beispiel für Text und Schaltknöpfe:

Aufgaben zur Herstellung:

#### Aussehen:

- JLabel- und JButton-Objekte generieren
- Anordnung der Komponenten festlegen

#### Ereignisse:

- ein *Listener*-Objekt mit den Buttons verbinden
- *call-back*-Methode für Buttons implementieren



## Komponenten platzieren

Jedes Fenster (**JFrame**-Objekt) besitzt ein **Container**-Objekt, das den Inhalt des Fenster aufnimmt, die sogenannte *content pane*.

Die Klasse **Container** sorgt für die Platzierung der Komponenten, die ein **Container**-Objekt enthält.

Dazu wird für den **Container** ein **LayoutManager** installiert; z. B. mit folgender Anweisung im Konstruktor der Unterklasse von **JFrame**:

```
Container content = getContentPane();  
content.setLayout (new FlowLayout (FlowLayout.CENTER));
```

Ein **LayoutManager** bestimmt die Anordnung der Komponenten nach einer speziellen Strategie; z. B. **FlowLayout** ordnet zeilenweise an.

Komponenten werden generiert und mit der Methode **add** dem **Container** zugefügt, z. B.

```
content.add (new JLabel ("W A R N I N G")); ...  
  
JButton waitButton = new JButton ("Wait");  
content.add (waitButton); ...
```

Die Reihenfolge der **add**-Aufrufe ist bei manchen **LayoutManagern** relevant.

Wird die Gestalt des **Containers** verändert, so ordnet der **LayoutManager** die Komponenten ggf. neu an.

# Programmschema zum Platzieren von Komponenten

```

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

class FlowTest extends JFrame // Definition der Fenster-Klasse
{ FlowTest () // Konstruktor
  { super ("Flow Layout (Centered)"); // Aufruf des Konstruktors von JFrame
    Container content = getContentPane(); // innere Fensterfläche
    content.setBackground (Color.cyan); // Eigenschaften festlegen

    content.setLayout(new FlowLayout(FlowLayout.CENTER));
    // LayoutManager-Objekt erzeugen und der Fensterfläche zuordnen

    content.add(new JButton("Diese")); ... // Komponenten zufügen

    setSize(350,100);
    setDefaultCloseOperation (EXIT_ON_CLOSE);
    // Verhalten des Fensters: Beim Drücken des Schließknopfes Programm beenden.
    setVisible(true);
  }
}

public class LayoutTry
{ public static void main(String[] args)
  { JFrame f = new FlowTest(); // Ein FlowTest-Objekt erzeugen
  }
}

```

# LayoutManager FlowLayout

```

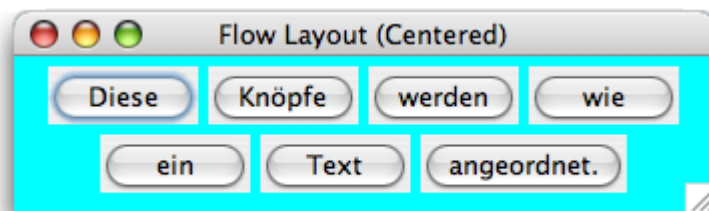
class FlowTest extends JFrame
{
    FlowTest ()
    {
        super("Flow Layout (Centered)");
        Container c = getContentPane(); c.setBackground(Color.cyan);

        c.setLayout(new FlowLayout(FlowLayout.CENTER));

        c.add(new JButton("Diese")); c.add(new JButton("Knöpfe"));
        c.add(new JButton("werden")); c.add(new JButton("wie"));
        c.add(new JButton("ein")); c.add(new JButton("Text"));
        c.add(new JButton("angeordnet."));

        setSize(350,100); setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
}

```



nach  
Ändern  
der Gestalt:



# LayoutManager BorderLayout

```

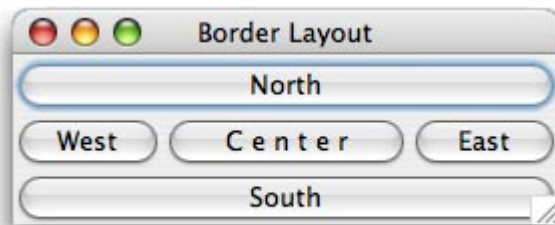
class BorderTest extends JFrame
{
  BorderTest ()
  {
    super("Border Layout");
    Container c = getContentPane(); c.setBackground(Color.cyan);

    c.setLayout(new BorderLayout());

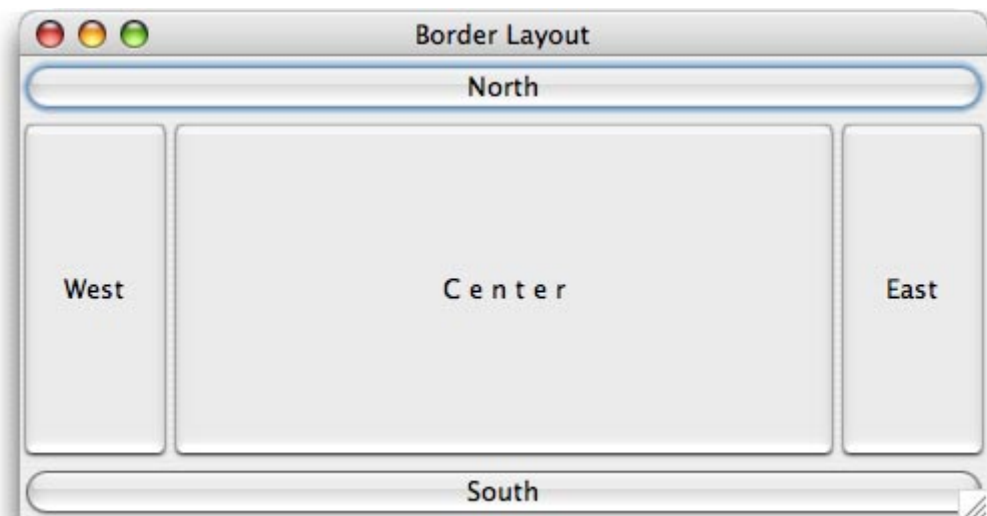
    c.add(new JButton("North"), BorderLayout.NORTH);
    c.add(new JButton("East"), BorderLayout.EAST);
    c.add(new JButton("South"), BorderLayout.SOUTH);
    c.add(new JButton("West"), BorderLayout.WEST);
    c.add(new JButton("C e n t e r"), BorderLayout.CENTER);

    setSize(250,100); setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
  }
}

```



nach  
Ändern  
der Gestalt:



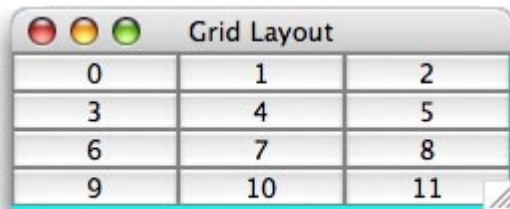
# LayoutManager GridLayout

```
class GridTest extends JFrame
{
    GridTest ()
    {
        super("Grid Layout");
        Container c = getContentPane(); c.setBackground(Color.cyan);

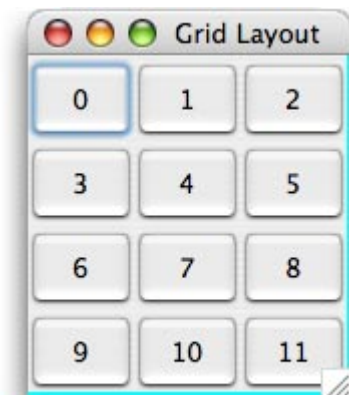
        c.setLayout(new GridLayout(4, 3));

        for (int i = 0; i < 12; i++)
            c.add(new JButton(String.valueOf(i)));

        setSize(250,100); setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
}
```



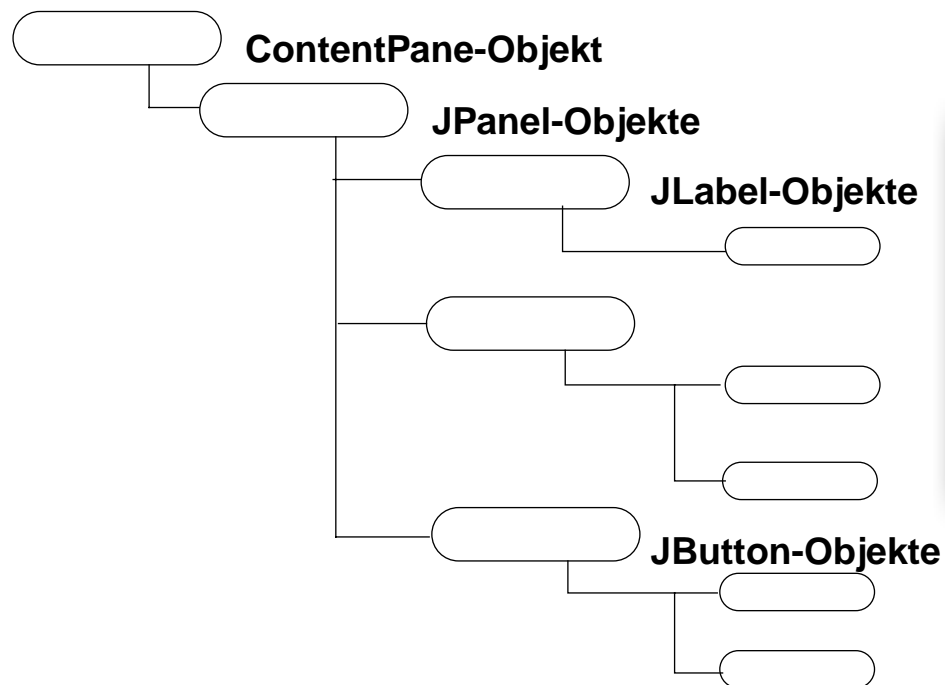
nach  
Ändern  
der Gestalt:



## 4. Hierarchisch strukturierte Fensterinhalte

- **Zusammengehörige Komponenten** in einem Objekt einer `Container`-Unterklasse unterbringen (`JPanel`, `content pane` eines `JFrames` oder selbstdefinierte Unterklasse).
- Anordnung der im `Container` gruppierten Objekte wird dann gemeinsam bestimmt, indem man dem `Container` einen geeigneten `LayoutManager` zuordnet.
- Mit `Container`-Objekten werden beliebig tiefe **Baumstrukturen** von Swing-Komponenten erzeugt. In der visuellen Darstellung sind sie **ineinander geschachtelt**.

JFrame-Objekt



# Programm zu hierarchisch strukturiertem Fenster

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

class LabelContainer extends JPanel           // Klasse zur Herstellung von
{ LabelContainer (String[] words)           // Fließtext aus String-Array
  { super(new FlowLayout(FlowLayout.CENTER)); // Konstr. der Oberklasse
    for (int i = 0; i < words.length; i++)  // legt LayoutManager fest
      add (new JLabel (words[i]));
  }
}

class LayoutComp extends JFrame // Fensterkonstr. erzeugt Komponentenbaum
{ LayoutComp (String title)
  { super(title);
    String[] message = {"Possible", "virus", "detected.", "Reboot",
                       "and", "run", "virus", "remover", "software"};
    JPanel warningText = new LabelContainer (message);

    ... Erzeugung des Komponentenbaumes einfügen ...

    setSize (180, 200); setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main (String[] args)
  { JFrame f = new LayoutComp("Virus Warning"); }
}
```

# Komponentenbaum erzeugen

```
// Text der Warnung im Array message zusammengefasst auf LabelContainer:
JPanel warningText = new LabelContainer (message);

// Überschrift als JPanel mit einem zentrierten JLabel:
JPanel header = new JPanel (new FlowLayout(FlowLayout.CENTER));
header.setBackground(Color.yellow);
header.add (new JLabel ("W a r n i n g"));

// Knöpfe im JPanel mit GridLayout:
JPanel twoButtons = new JPanel (new GridLayout(1, 2));
twoButtons.add (new JButton ("Wait"));
twoButtons.add (new JButton ("Reboot"));

// in der Fensterfläche mit BorderLayout zusammenfassen:
Container content = getContentPane();
content.setBackground(Color.cyan);
content.setLayout(new BorderLayout());
content.add(header, BorderLayout.NORTH);
content.add(warningText, BorderLayout.CENTER);
content.add(twoButtons, BorderLayout.SOUTH);
```

## 5. Ereignisse an graphischen Benutzungsoberflächen

**Interaktion** zwischen Bediener und Programmausführung über **Ereignisse** (*events*):

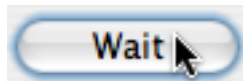
- **Bedien-Operationen lösen Ereignisse aus**, z. B. Knopf drücken, Menüpunkt auswählen, Mauszeiger auf ein graphisches Element bewegen.
- **Programmausführung reagiert auf solche Ereignisse** durch Aufruf bestimmter Methoden

**Aufgaben:**

- **Folgen von Ereignissen** und Reaktionen darauf **planen und entwerfen**  
Modellierung z. B. mit endlichen Automaten oder StateCharts
- **Reaktionen auf Ereignisse** systematisch implementieren  
Swing: Listener-Konzept; Entwurfsmuster „Observer“ (aus AWT übernommen)

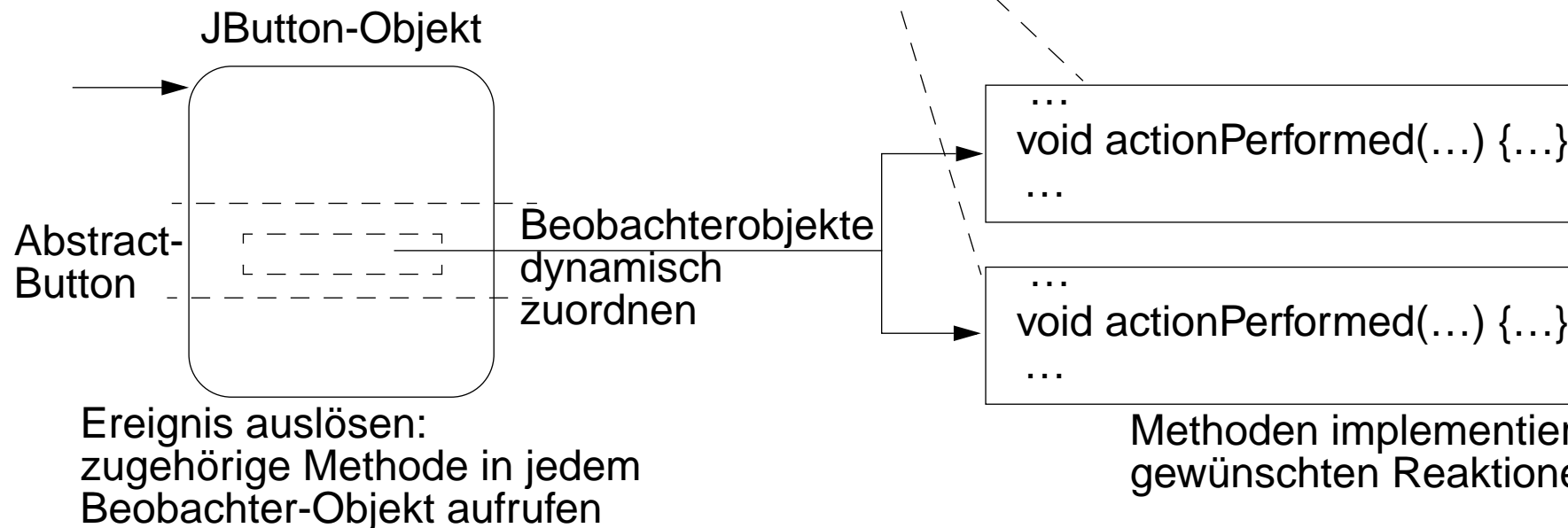
# „Observer“-Prinzip in der Ereignisbehandlung

An Swing-Komponenten werden Ereignisse ausgelöst, z. B. ein ActionEvent an einem JButton-Objekt:



*“click”*

**Beobachter (Listener)** für bestimmte Ereignistypen: Objekte von Klassen, die das zugehörige Interface implementieren



Entwurfsmuster „Observer“: Unabhängigkeit zwischen den Beobachtern und dem Gegenstand wegen Interface und dynamischem Zufügen von Beobachtern.

# Ereignisbehandlung für eine Schaltfläche

Im `java.awt.event` Package gibt es zum Ereignistyp `ActionEvent` ein Interface `ActionListener`:

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

Um auf das Anklicken der Schaltfläche zu reagieren, wird im Programm eine Klasse deklariert, die das **Interface implementiert**. Die **Methode** aus dem Interface wird mit der gewünschten Reaktion **überschrieben**:

```
class ProgramTerminator implements ActionListener
{ public void actionPerformed (ActionEvent e)
  { System.exit (0); }
}
```

Von dieser Klasse wird ein Objekt erzeugt und als Beobachter dem Schaltflächen-Objekt zugefügt:

```
JButton quitButton = new JButton("Quit");
quitButton.addActionListener (new ProgramTerminator());
```

## Programmiertechnik für Listener

Im `java.awt.event` Package gibt es zu jedem Ereignistyp `XXXEvent` ein Interface `XXXListener`:

```
public interface WindowListener extends EventListener
{ void windowActivated (WindowEvent); void windowClosed (WindowEvent);
  void windowClosing (WindowEvent); ... void windowOpened (WindowEvent);
}
```

Eine **abstrakte Klasse `XXXAdapter`** mit leeren Methodenimplementierungen:

```
public abstract class WindowAdapter implements WindowListener
{ public void windowActivated (WindowEvent) { } ...
  public void windowOpened (WindowEvent) { }
}
```

**Anwendungen**, die nicht auf alle Sorten von Methodenaufrufen des Interface reagieren, deklarieren eine Unterklasse und **überschreiben die benötigten Methoden des Adapters**, meist als innere Klasse, um den Zustand eines Objektes zu verändern:

```
class WindowCloser extends WindowAdapter
{ public void windowClosing (WindowEvent e)
  { System.exit (0); }
}
```

Zufügen eines Listener-Objektes zu einer Swing-Komponente:

```
f.addWindowListener (new WindowCloser());
```

# Innere Klassen

Innere Klassen können z. B. als Hilfsklassen zur Implementierung der umgebenden Klasse verwendet werden:

```
class List { ... static class Node { ... } ... }
```

Die `List`-Objekte und `Node`-Objekte sind dann **unabhängig voneinander**.

Es wird nur die Gültigkeit des Namens `Node` auf die Klasse `List` eingeschränkt.

In **inneren Klassen, die nicht `static`** sind, können Methoden der inneren Klasse auf Objektvariable der äusseren Klasse zugreifen. Ein Objekt der inneren Klasse ist dann immer in ein Objekt der äusseren Klasse eingebettet; z. B. die inneren `Listener` Klassen, oder auch:

```
interface Einnehmer { void bezahle (int n); }
```

```
class Kasse // Jedes Kassierer-Objekt eines Kassen-Objekts
{ private int geldSack = 0; // zahlt in denselben Geldsack
```

```
    class Kassierer implements Einnehmer
    { public void bezahle (int n)
      { geldSack += n; }
    }
```

```
    Einnehmer neuerEinnehmer ()
    { return new Kassierer (); }
}
```

# Anonyme Klasse

Meist wird zu der Klasse, mit der Implementierung der Reaktion auf einen Ereignistyp **nur ein einziges Objekt** benötigt:

```
class WindowCloser extends WindowAdapter
{ public void windowClosing (WindowEvent e)
  { System.exit (0); }
}
```

Zufügen eines `Listener`-Objektes zu einer Swing-Komponente:

```
f.addWindowListener (new WindowCloser());
```

Das läßt sich kompakter formulieren mit einer **anonymen Klasse**:

Die Klassendeklaration wird mit der `new`-Operation (für das eine Objekt) kombiniert:

```
f.addWindowListener
( new WindowAdapter ()
  { public void windowClosing (WindowEvent e)
    { System.exit (0); }
  }
);
```

In der `new`-Operation wird der **Name der Oberklasse** der deklarierten anonymen Klasse (hier: `WindowAdapter`) **oder** der **Name des Interface**, das sie implementiert, angegeben!

## Reaktionen auf Buttons

Swing-Komponenten `JButton`, `JTextField`, `JMenuItem`, `JComboBox`,... lösen `ActionEvents` aus.

Sie werden von `ActionListener`-Objekten beobachtet, mit einer einzigen Methode:

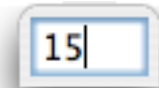
```
public void actionPerformed (ActionEvent e) {...}
```

Beispiel der Virus-Warnung (Abweichung vom Stil im Buch Java Gently!):

```
import javax.swing.*; import java.awt.*; import java.awt.event.*;
class LayoutComp extends JFrame
{ private JButton waitButton, rebootButton; int state = 0;
  LayoutComp (String title)
  { ...
    waitButton.addActionListener      // Listener für den waitButton
      ( new ActionListener ()        // anonyme Klasse direkt vom Interface
        { public void actionPerformed(ActionEvent e)
          { state = 1; setBackground(Color.red); } });
    rebootButton.addActionListener // Listener für den rebootButton
      ( new ActionListener ()
        { public void actionPerformed(ActionEvent e)
          { state = 2; setVisible(false); System.exit(0); } });
  } }
```

Die Aufrufe von `setBackground` und `setVisible` beziehen sich auf das umgebende `LayoutComp`-Objekt — nicht auf das unmittelbar umgebende `ActionListener`-Objekt.

# Eingabe von Texten



**Komponente** `JTextField`: einzeiliger, edierbarer Text

**Ereignisse**: `ActionEvent` (wie bei  `JButton` ) ausgelöst bei der Eingabe von `<Return>`

einige Methoden (aus der Oberklasse `JTextComponent`):

<code>String getText ()</code>	Textinhalt liefern
<code>void setText (String v)</code>	Textinhalt setzen
<code>void setEditable (boolean e)</code>	Edierbarkeit festlegen
<code>boolean isEditable ()</code>	Edierbarkeit abfragen
<code>void setCaretPosition (int pos)</code>	Textcursor positionieren

Typischer `ActionListener`:

```
addActionListener
( new ActionListener ()
  { public void actionPerformed (ActionEvent e)
    { String str = ((JTextField) e.getSource()).getText(); ...
    } });
```

**Eingabe von Zahlen**: [Text in eine Zahl konvertieren](#), Ausnahme abfangen:

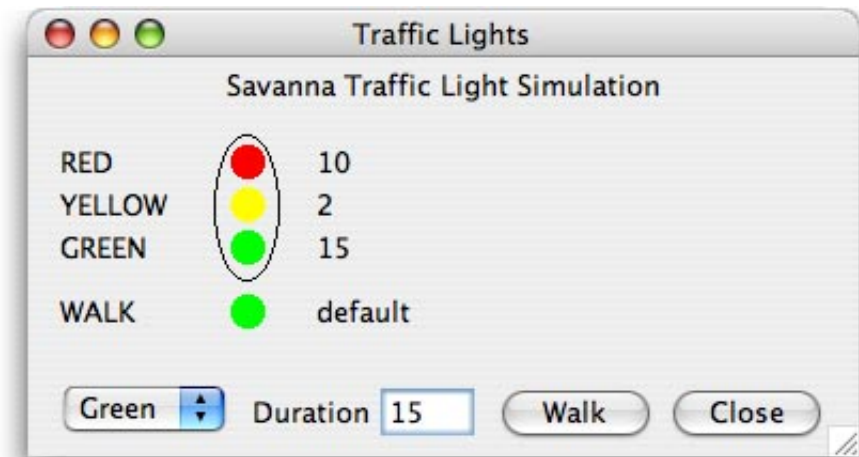
```
int age;
try { age = Integer.parseInt (str); }
catch (NumberFormatException e) { ... } ...
```

## 6. Beispiel: Ampel-Simulation

**Aufgabe:** Graphische Benutzungsoberfläche für eine Ampel-Simulation entwerfen

### Eigenschaften:

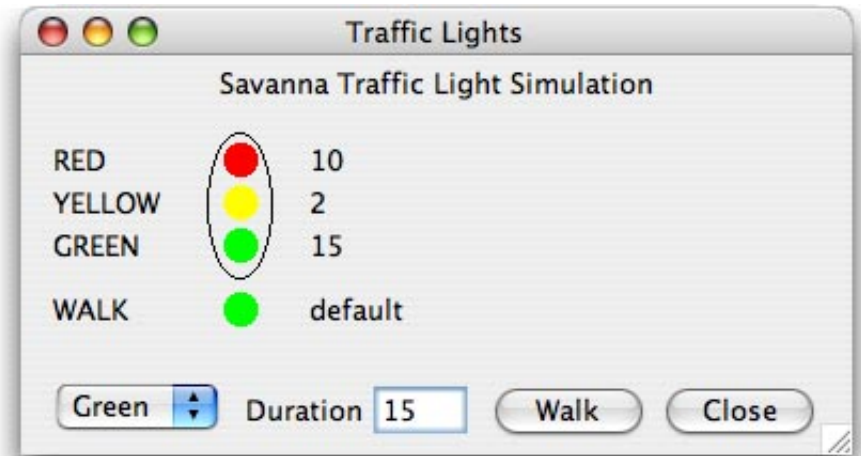
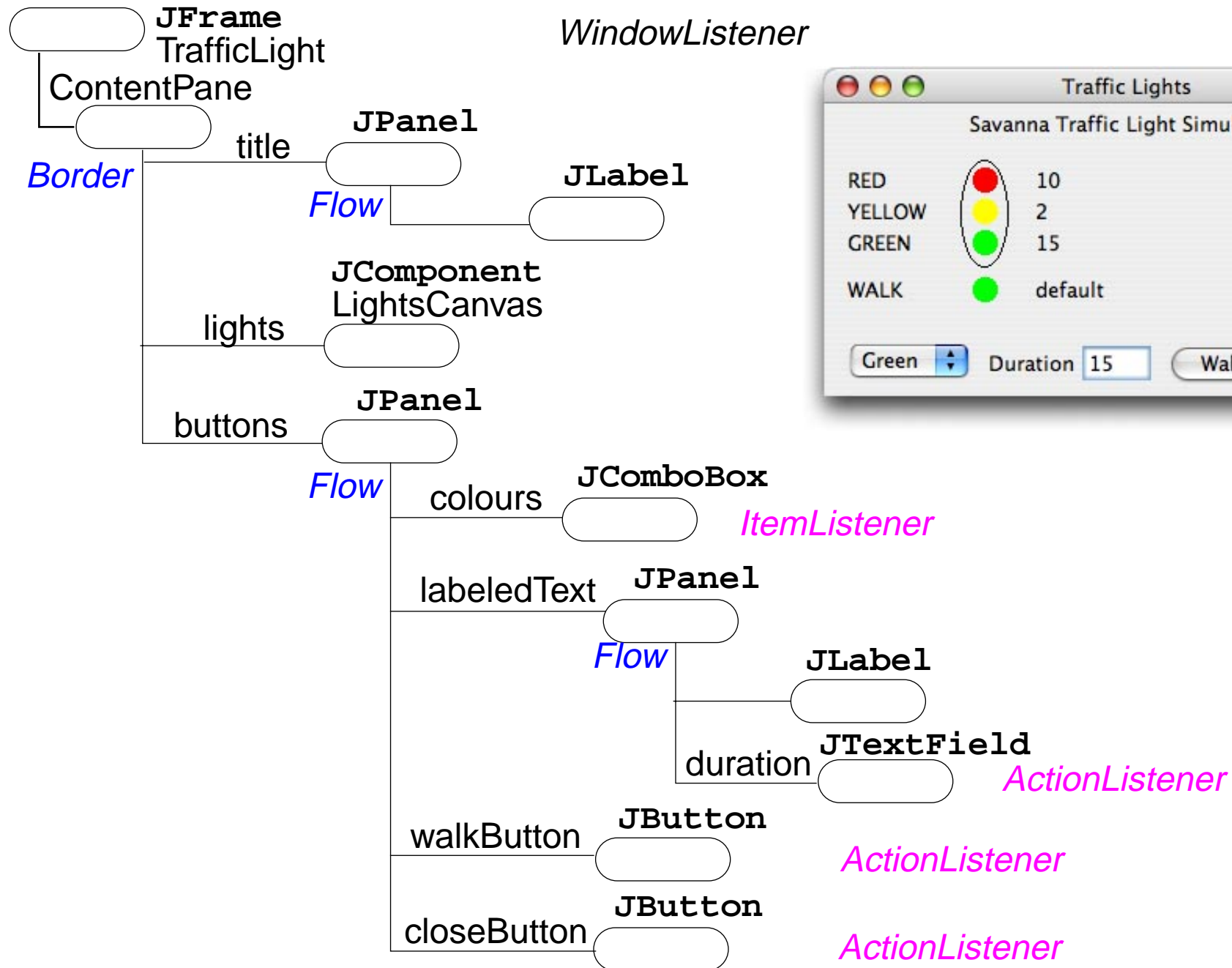
- Ampel visualisieren  
mit Knopf und Licht für Fußgänger  
(später auch animieren)
- Phasenlängen der Lichter  
einzeln einstellbar
- Einstellungen werden angezeigt



### Entwicklungsschritte:

- Komponenten strukturieren
- zeichnen der Ampel (`paint` in eigener Unterklasse von `JComponent`)
- Komponenten generieren und anordnen
- Ereignisbehandlung entwerfen und implementieren

# Objektbaum zur Ampel-Simulation



# Programm zur Ampel-Simulation

Im Konstruktor der zentralen Klasse wird der **Objektbaum** hergestellt:

```
class TrafficLight extends JFrame
{
    // basiert auf: The Traffic Light program by J M Bishop Oct 1997

    // Objektvariable, auf die Listener zugreifen:
    private String[] message = // Phasendauer für jede Lampe als Text:
        { "default", "default", "default", "default" };
    private int light = 0; // die ausgewählte Lampe
    private LightsCanvas lights; // Enthält die gezeichnete Ampel

    public TrafficLight (String wt) // Konstruktor der zentralen Klasse
    {
        super (wt); // Aufbau des Objektbaumes:
        Container cont = getContentPane(); // innere Fensterfläche
        cont.setLayout (new BorderLayout ()); // Layout des Wurzelobjektes

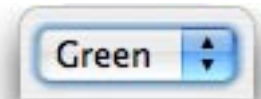
        // Zentrierter Titel:
        JPanel title = new JPanel (new FlowLayout (FlowLayout.CENTER));
        title.add (new JLabel("Savanna Traffic Light Simulation"));
        cont.add (title, BorderLayout.NORTH);

        // Die Ampel wird in einer getrennt definierten Klasse gezeichnet:
        lights = new LightsCanvas (message);
        cont.add (lights, BorderLayout.CENTER);
    }
}
```

# Auswahl-Komponente

Auswahl-Komponenten (JComboBox) lösen `ItemEvents` aus, wenn ein Element ausgewählt wird. Mit der Methode `itemStateChanged` kann ein `ItemListener` darauf reagieren:

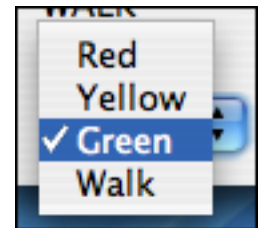
```
String[] lightNames = { "Red", "Yellow", "Green", "Walk" };
JComboBox colours = new JComboBox (lightNames);
```



```
colours.addItemListener
( new ItemListener ()
  { public void itemStateChanged (ItemEvent e)
    { if (e.getStateChange() == ItemEvent.SELECTED)
      { String s = e.getItem().toString();

        if (s.equals("Red"))           light = 0;
        else if (s.equals("Yellow"))   light = 1;
        else if (s.equals("Green"))    light = 2;
        else if (s.equals("Walk"))     light = 3;

      }
    }
  }
);
```



Über den `ItemEvent`-Parameter kann man auf das gewählte Element zugreifen.

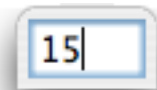
## Eingabe der Phasenlänge

**Eingabe** mit einem `JTextField`. **Reaktion** auf ein `ActionEvent`:

```
JPanel labeledText = new JPanel (new FlowLayout (FlowLayout.LEFT));
// fasst TextField und Beschriftung zusammen
labeledText.add(new JLabel("Duration"));
```

// Eingabeelement für die Phasendauer einer Lampe:

```
JTextField duration = new TextField (3);
duration.setEditable (true);
```



```
duration.addActionListener
( new ActionListener ()
{ public void actionPerformed (ActionEvent e)
{ // Zugriff auf den eingegebenen Text:
// message[light] = ((JTextField) e.getSource()).getText();
// oder einfacher:
message[light] = e.getActionCommand ();
lights.repaint(); // Die Zeichenmethode der gezeichneten Ampel
// wird erneut ausgeführt,
// damit der geänderte Text sichtbar wird.
} });
```

```
labeledText.add (duration);
```

# Button-Zeile

Einfügen der **Button-Zeile** in den Objektbaum:

```

    JButton walkButton = new JButton ("Walk");
                                     // noch keine Reaktion zugeordnet

    JButton closeButton = new JButton ("Close");
    closeButton.addActionListener
    ( new ActionListener ()           // Programm beenden:
      { public void actionPerformed(ActionEvent e)
        { setVisible (false); System.exit (0); }
      }
    );

                                     // Zusammensetzen der Button-Zeile:
    JPanel buttons = new JPanel(new BorderLayout (FlowLayout.CENTER));
    buttons.add (colours);
    buttons.add (labeledText);
    buttons.add (walkButton);
    buttons.add (closeButton);

    cont.add (buttons, BorderLayout.SOUTH);
} // TrafficLight Konstruktor

public static void main (String[] args) { JFrame f = ... }
} // TrafficLight Klasse

```



## Ampel zeichnen und beschriften

Eine Unterklasse der allgemeinen Oberklasse `JComponent` für Swing-Komponenten stellt die **Zeichenfläche** bereit. Die Methode `paint` wird zum Zeichnen und Beschriften überschrieben:

```
class LightsCanvas extends JComponent
{ private String[] msg;

  LightsCanvas (String[] m)                // Die Array-Elemente enthalten die
  { msg = m; }                             // Phasendauern der Lampen als Text. Sie
                                           // können durch Eingaben verändert werden.

  public void paint (Graphics g)
  { super.paint(g);                        // Hintergrund der Komponente zeichnen
    g.drawOval (87, 10, 30, 68);          // darauf: Ampel zeichnen und beschriften
    g.setColor (Color.red);              g.fillOval (95, 15, 15, 15);
    g.setColor (Color.yellow);           g.fillOval (95, 35, 15, 15);
    g.setColor (Color.green);            g.fillOval (95, 55, 15, 15);
    g.fillOval (95, 85, 15, 15);         // walk Lampe ist auch grün

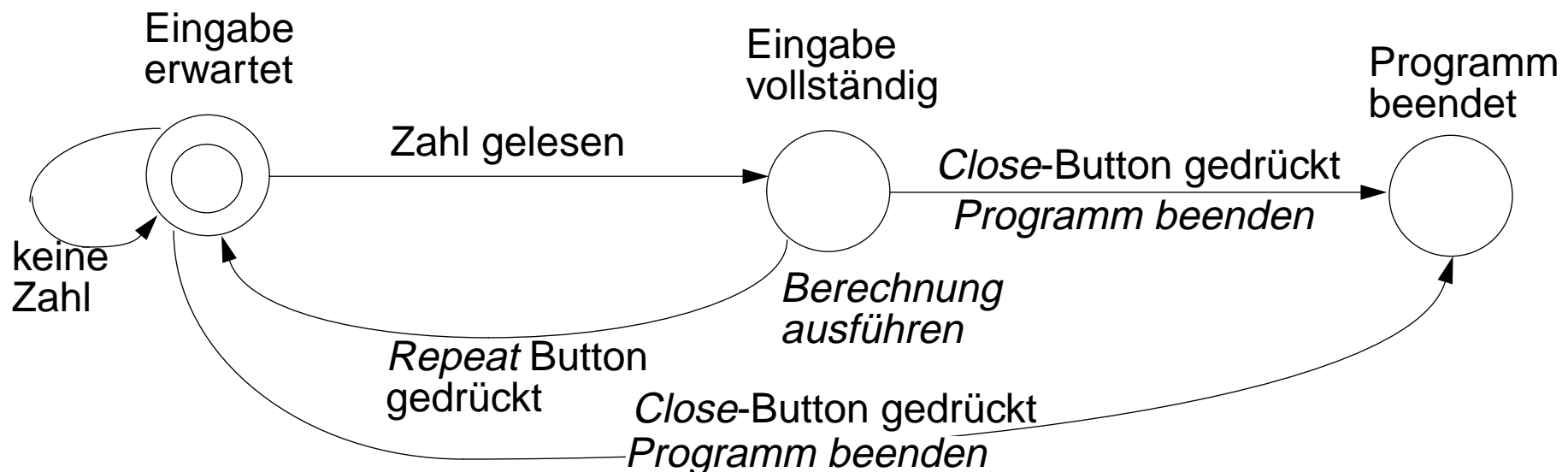
    g.setColor(Color.black);
    g.drawString ("RED", 15, 28); g.drawString ("YELLOW", 15, 48);
    g.drawString ("GREEN", 15, 68); g.drawString ("WALK", 15, 98);
                                           // eingegebene Phasendauern der Lampen:
    g.drawString (msg[0], 135, 28); g.drawString (msg[1], 135, 48);
    g.drawString (msg[2], 135, 68); g.drawString (msg[3], 135, 98);
  } }
```

## 7. Entwurf von Ereignisfolgen

Die zulässigen **Folgen von Bedienereignissen und Reaktionen** darauf müssen für komplexere Benutzungsoberflächen geplant und entworfen werden.

**Modellierung** durch **endliche Automaten** (auch durch *StateCharts*)

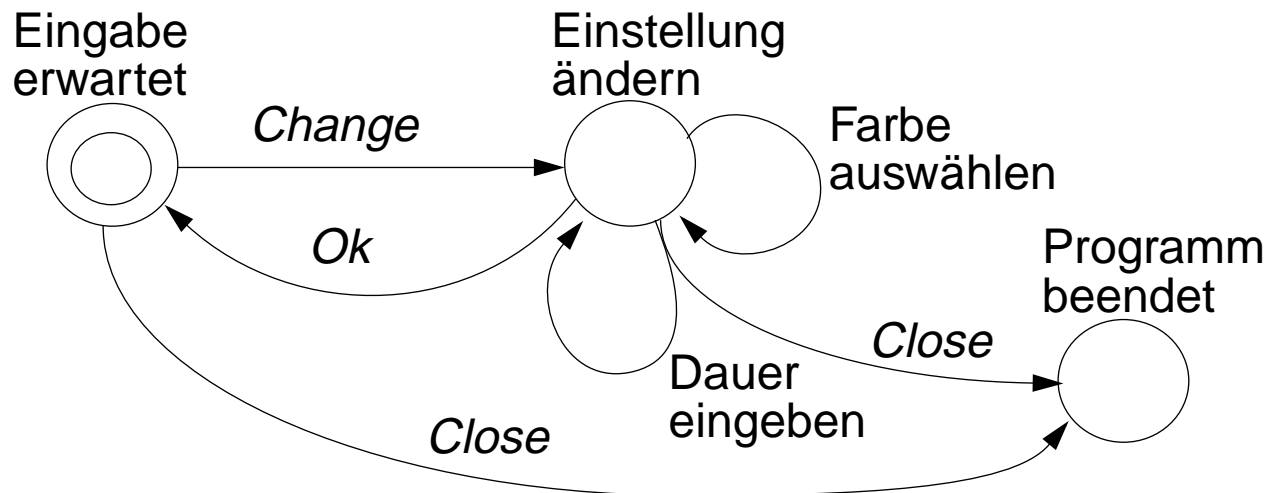
- **Zustände** unterscheiden Bediensituationen (z. B. „Eingabe erwartet“, „Eingabe vollständig“)
- **Übergänge** werden durch Ereignisse ausgelöst.
- **Aktionen** können mit Übergängen verknüpft werden; Reaktion auf ein Ereignis z. B. bei Eingabe einer Phasenlänge Ampel neu zeichnen, und
- **Aktionen** können mit dem Erreichen eines Zustandes verknüpft werden, z. B. wenn die Eingabe vollständig ist, Berechnung beginnen.



# Unzulässige Übergänge

In manchen Zuständen sind **einige Ereignisse nicht als Übergang definiert**. Sie sind in dem Zustand **unzulässig**, z. B. „Farbe auswählen“ im Zustand „Eingabe erwartet“.

**Beispiel:** Ampel-Simulation erweitert um zwei Buttons **Change** und **Ok**:

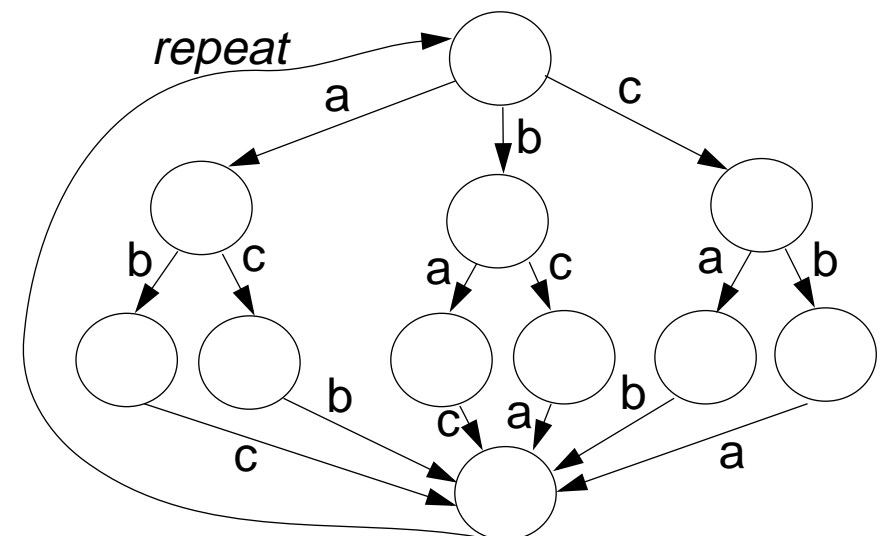
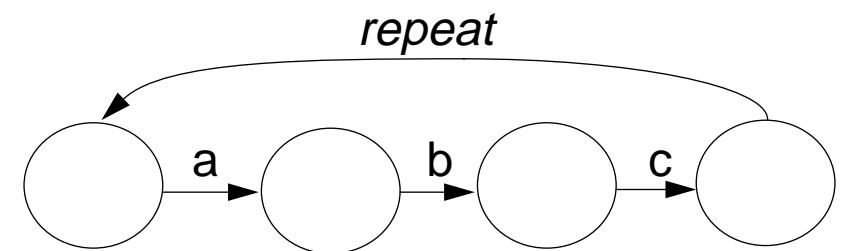
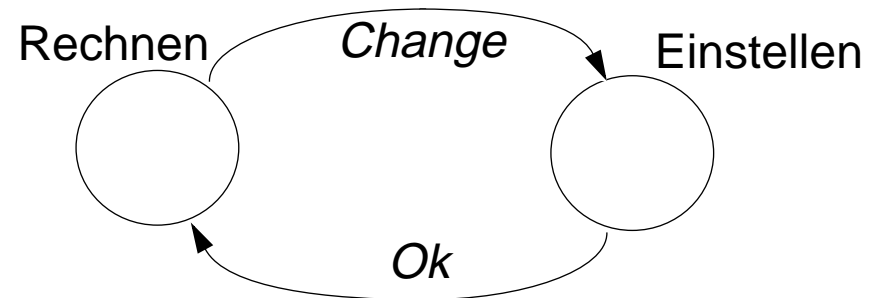


**Robuste Programme** dürfen auch an unzulässigen Ereignisfolgen nicht scheitern. Verschiedene Möglichkeiten für **nicht definierte Übergänge**:

- Sie bleiben **ohne Wirkung**
- Sie bleiben ohne Wirkung und es wird eine **Erklärung** gegeben (Warnungsfenster).
- **Komponenten** werden so **deaktiviert**, dass unzulässige Ereignisse nicht ausgelöst werden können.

# Muster für Ereignisfolgen

- Verschiedene **Bedienungsarten** (mode):  
Vorsicht: Nicht unnötig viele Zustände entwerfen. “Don’t mode me in!”
- Festgelegte **sequentielle Reihenfolge**:  
Vorsicht: Nicht unnötig streng vorschreiben.  
Bediener nicht gängeln.
- **Beliebige Reihenfolge** von Ereignissen:  
Modellierung mit endlichem Automaten ist umständlich (Kombinationen der Ereignisse);  
einfacher mit *StateCharts*.
- **Voreinstellungen** (*defaults*) können Zustände sparen und Reihenfolgen flexibler machen.  
Vorsicht: Nur sinnvolle Voreinstellungen.



## Implementierung des Ereignis-Automaten

**Zustände** ganzzahlig **codieren**; **Objektvariable** speichert den **augenblicklichen Zustand**:

```
private int currentState = initialState;  
private static final int initialState = 0, settingState = 1, ...;
```

Einfache **Aktionen der Übergänge** bleiben in den Reaktionsmethoden der **Listener**;  
Methodenaufruf für den Übergang in einen neuen Zustand zufügen:

```
okButton.addActionListener  
( new ActionListener ()  
  { public void actionPerformed(ActionEvent e)  
    { message[light] = duration.getText();  
      toState (initialState); } }));
```

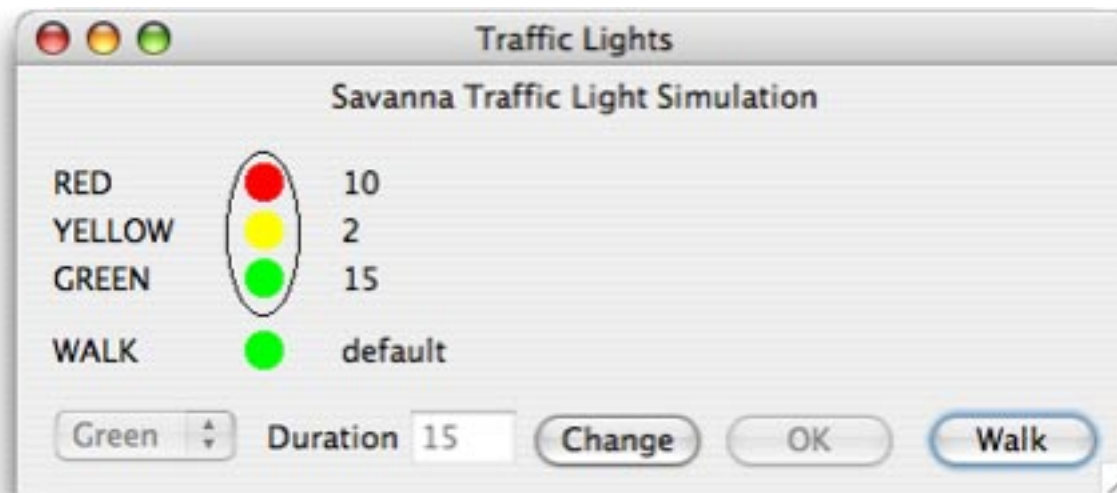
Aktionen der Zustände in **Übergangsmethode** platzieren, z. B. Komponenten (de)aktivieren:

```
private void toState (int nextState)  
{ currentState = nextState;  
  switch (nextState)  
  { case initialState:  
    lights.repaint();  
    okButton.setEnabled(false); changeButton.setEnabled (true);  
    colours.setEnabled (false); duration.setEnabled (false);  
    break;  
    case settingState:  
    okButton.setEnabled(true); changeButton.setEnabled (false); ...  
    break; } }
```

# Ampel-Simulation mit kontrollierten Zustandsübergängen

Zwei Knöpfe wurden zugefügt:

Der **Change-Button** aktiviert die Eingabe, der **Ok-Button** schliesst sie ab.



Die Komponenten zur Farbauswahl, Texteingabe und der Ok-Knopf sind im gezeigten Zustand deaktiviert.

## 8. Model/View-Paradigma für Komponenten

```

class Mitglied
{
  String name;
  Leisure hobby;
  boolean profi;
  String notiz;
  ...
}

```

Darstellung  
der Werte

Änderung  
der Werte

The screenshot shows a dialog box titled 'Mitgliedsdaten' with the following fields:

- Name: Willi Müller
- Hobby: Lesen
- Status:  Vollprofi
- Notiz: Kassenwart 2004, Schriftführer 2003, Sekretär 2002, Hauptversammlung, Beitrag bezahlt, Treuerabatt

Buttons: OK, Cancel

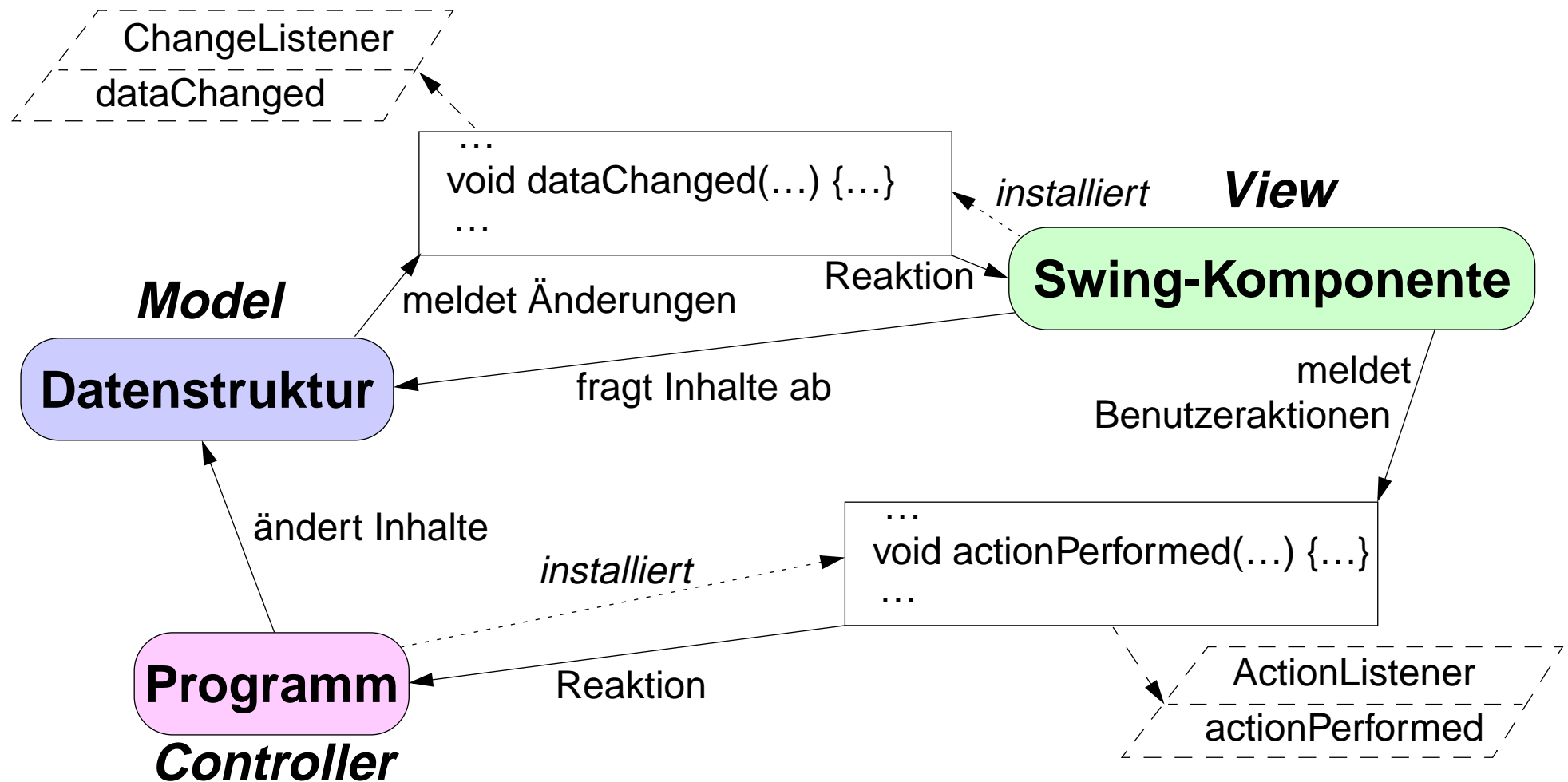
**Model**

**View**

### Wechselwirkung:

- Komponenten der Benutzungsoberfläche (*view*) stellen Inhalte von Datenstrukturen (*model*) des Programms dar
- **Ziel:** Übereinstimmung zwischen Daten und ihrer Darstellung am Bildschirm **automatisch** sicherstellen

# Zusammenarbeit von Model, View und Programm



- Anwendung des "Observer"-Prinzips (wie bei der Ereignisbehandlung): eine (oder mehrere) Komponenten beobachten die Datenstruktur
- Datenstruktur ruft bei jeder Änderung zugehörige Methode in jedem Beobachter-Objekt auf

## Schematischer Aufbau von Model und View

```

class Model
{ // Datenstruktur mit Zugriffsmethoden
  private String data;

  String getData ()
  { return data; }

  void String setData (String s)
  { data = s;
    fireChangeEvent();
  }

  // Verwaltung der Listener
  ChangeListener[] listener;
  int count = 0;

  void addChangeListener
    (ChangeListener cl)
  { listener[count++] = cl; }

  void fireChangeEvent ()
  { for (int i=0; i<count; i++)
    listener[i].dataChanged();
  }
}

```

```

class View extends JComponent
{ private Model model;

  View (Model m)
  { model = m;
    model.addChangeListener(
      new RedrawListener());
  }

  ...

  class RedrawListener
    implements ChangeListener
  { public void dataChanged()
    { String s =
      model.getData();

    ...
    repaint();
  }
}

interface ChangeListener
{ public void dataChanged(); }

```

# Beispiel: Haushaltsbuch

**Aufgabe:** Programm zur Überwachung von Ausgaben im Haushalt entwerfen

## Eigenschaften:

- speichert Geldbetrag (Summe der Ausgaben) für jede Kategorie
- Eingabe von Einzelbuchungen (+/–) und Löschen ganzer Kategorien
- Liste aller Kategorien mit Beträgen wird angezeigt



## Entwicklungsschritte:

- Datenstruktur entwerfen
- Datenstruktur zum Datenmodell (*Model*) für eine Liste (`JList`) erweitern
- Komponenten strukturieren
- Komponenten generieren und anordnen
- Ereignisbehandlung entwerfen und implementieren

## Datenstruktur für Haushaltsbuch

Speicherung der Kategorien und Beträge in zwei parallel belegten Arrays;  
zentrale Operation “*Buchung vornehmen*” und Abfragemethoden:

```
class ChequeBook
{
  private String[] purposes = new String[20];    // Namen der Kategorien
  private int[] amounts = new int[20];         // Geldbeträge dazu
  private int entries = 0;                     // Anzahl Einträge

  void addTransaction (String p, int a)
  {
    if (p.length() == 0 || a == 0) return;     // keine sinnvolle Buchung
    for (int i = 0; i < entries; i++)
    {
      if (purposes[i].equals(p))               // Kategorie bereits vorhanden
      {
        amounts[i] += a; return; }            // Geldbetrag anpassen
    }
    int index = entries; entries += 1;         // neue Kategorie anfügen
    purposes[index] = p; amounts[index] = a;
  }

  int getEntries ()                           // Abfragemethoden: Anzahl Einträge
  { return entries; }

  int getAmount (int index)                   // Geldbetrag eines Eintrags
  { return amounts[index]; }

  String getPurpose (int index)                 // Kategorienname eines Eintrags
  { return purposes[index]; }
}
```

## Model für JList-Komponenten

Das **Interface ListModel** im Package `javax.swing` beschreibt die Anforderungen an das Datenmodell für eine graphisch dargestellte Liste (`JList`-Objekt):

```
public interface ListModel
{ // Verwaltung von Swing-Komponenten, die das Model beobachten:
  void addListDataListener (ListDataListener ldl);
  void removeListDataListener (ListDataListener ldl);
  int getSize (); // Abfrage der Daten durch das JList-Objekt
  Object getElementAt (int index); // Abfrage eines Eintrags
}
```

Die **abstrakte Klasse AbstractListModel** implementiert das An- und Abmelden von **ListDataListener**-Objekten und ergänzt Methoden, um diese zu benachrichtigen:

```
public abstract class AbstractListModel implements ListModel
{ public void addListDataListener (ListDataListener ldl) { ... }
  public void removeListDataListener (ListDataListener ldl) { ... }
  protected void fireContentsChanged (Object src, int from, int to)
  { ... } // ruft Methode contentsChanged() in allen Listenern auf
  protected void fireIntervalAdded (...) { ... }
  protected void fireIntervalRemoved (...) { ... }
}
```

Die **firexxx**-Methoden korrespondieren mit den Methoden im **Interface ListDataListener**: `contentsChanged(ListDataEvent e)`, `intervalAdded(...)`, `intervalRemoved(...)`.

## Model für Haushaltsbuch

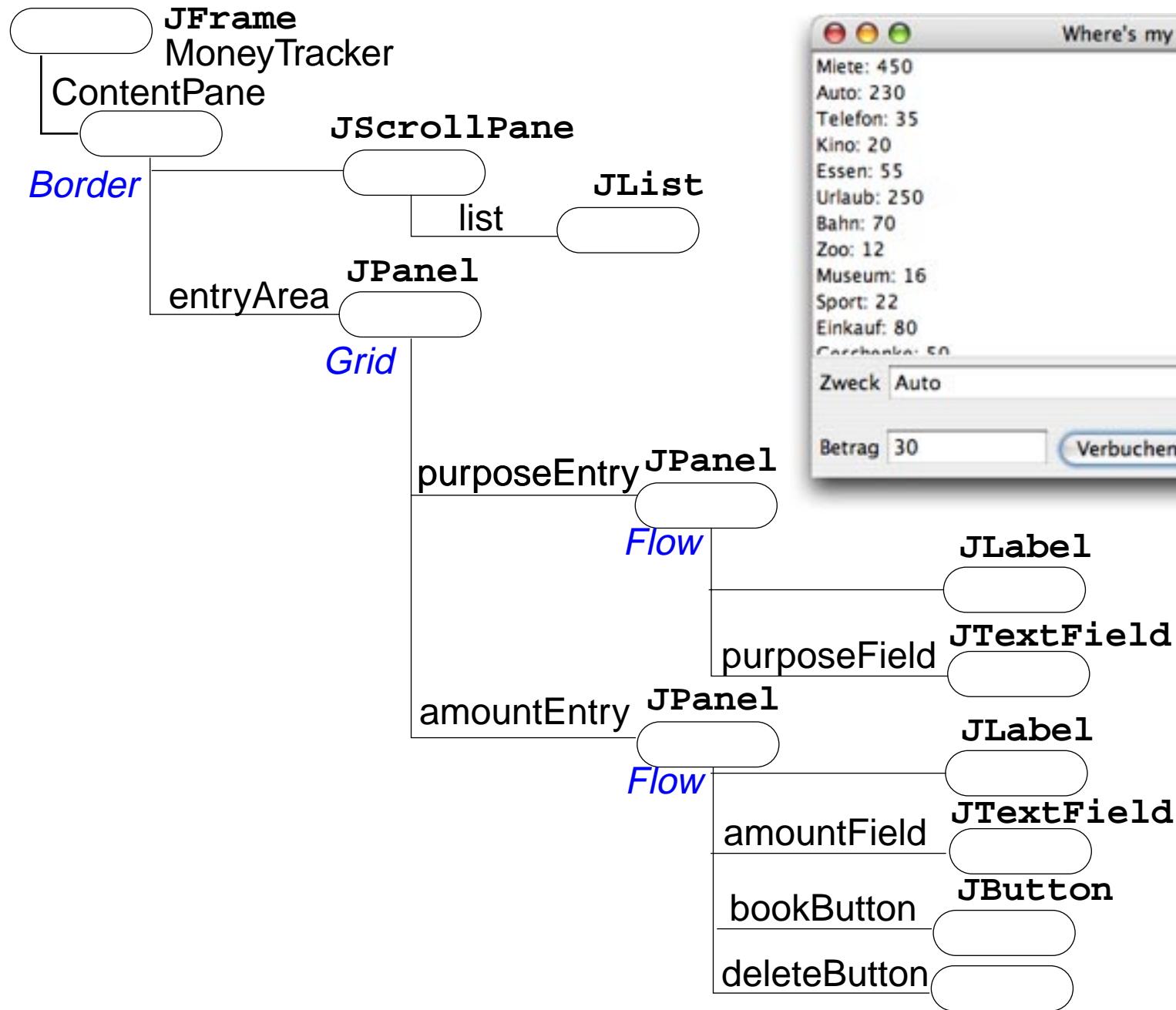
Erweitern die Datenstruktur zum Datenmodell für eine Liste:

- **Abfragemethoden** für `JList`-Objekt hinzufügen.
- **Benachrichtigung** der Listener in den Operationen ergänzen, die die Datenstruktur verändern.

```
class ChequeBook extends AbstractListModel // neue Oberklasse
{ private ...; // Objektvariablen bleiben unverändert
  public int getSize ()
  { return entries; }
  public Object getElementAt (int index)
  { return purposes[index] + ": " + String.valueOf(amounts[index]); }
  void addTransaction (String p, int a)
  { if (p.length() == 0 || a == 0) return; // keine sinnvolle Buchung
    for (int i = 0; i < entries; i++)
    { if (purposes[i].equals(p)) // Kategorie bereits vorhanden
      { amounts[i] += a; // Geldbetrag anpassen
        fireContentsChanged(this, i, i); // Beobachter informieren
        return;
      }
    }
    int index = entries; entries += 1; // neue Kategorie anfügen
    purposes[index] = p; amounts[index] = a;
    fireIntervalAdded(this, index, index); // Beobachter informieren
  } ... } // restliche Abfragemethoden bleiben unverändert
```

Auto: 230  
 Telefon: 35  
 Kino: 20  
 Essen: 55

# Objektbaum zum Haushaltsbuch



*ActionListener*  
*ActionListener*

# Programm zum Haushaltsbuch

```

class MoneyTracker extends JFrame
{
    private ChequeBook listData; private JList list;
    private JTextField purposeField, amountField;

    MoneyTracker (String title, ChequeBook data)
    {
        super(title); listData = data;
        list = new JList(listData);           // Liste erzeugen und mit Model verbinden
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        JPanel purposeEntry = new JPanel(new BorderLayout(...));
        purposeEntry.add(new JLabel("Zweck"));
        purposeField = new JTextField(30);
        purposeEntry.add(purposeField);
        JPanel amountEntry = new JPanel(...); ... // Komponenten einfügen
        JPanel entryArea = new JPanel(new GridLayout(2, 1));
        entryArea.add(purposeEntry); entryArea.add(amountEntry);
        Container content = getContentPane();
        content.setLayout(new BorderLayout());
        content.add(new JScrollPane(list), BorderLayout.CENTER);
        content.add(entryArea, BorderLayout.SOUTH);
        ... setVisible(true); // Eigenschaften des Fensters einstellen
    }

    public static void main (String[] args)
    {
        ChequeBook myMoney = new ChequeBook();
        JFrame f = new MoneyTrackerB("Where's my money?", myMoney);
    } }

```



# Implementierung der Ereignisbehandlung

Methoden aufrufen, die Änderungen am Datenmodell vornehmen:

```
class MoneyTracker extends JFrame
{ private ChequeBook listData; private JList list;
  private JTextField purposeField, amountField;

  MoneyTracker (String title, ChequeBook data)
  { super(title); listData = data;
    list = new JList(listData);           // Liste erzeugen und mit Model verbinden
    ...
    JPanel amountEntry = new JPanel(...); ...           // Komponenten einfügen
    JButton bookButton = new JButton("Verbuchen");
    amountEntry.add(bookButton); ...
    bookButton.addActionListener(new ActionListener()
    { public void actionPerformed (ActionEvent e)       // Reaktion auf Klick
      { String purpose = purposeField.getText();       // Werte auslesen
        String amount = amountField.getText();
        int money;
        try
        { money = Integer.parseInt(amount); } // Geldbetrag in Zahl wandeln
        catch (NumberFormatException nfe)
        { money = 0; } // keine Zahl: 0 annehmen
          listData.addTransaction(purpose, money);       // Modell verändern
        }
      });
    ... setVisible(true); // Rest des Fensters aufbauen
  } ... }
```



# Anpassung der Darstellung durch Renderer-Objekte

```

class MoneyTracker extends JFrame
{
    ...
    MoneyTracker (String title, ChequeBook data, ListCellRenderer renderer)
    {
        super(title); listData = data;
        list = new JList(listData); // Liste erzeugen und mit Model verbinden
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        list.setCellRenderer(renderer); ... // eigenes Renderer-Objekt anschliessen
    }

    public static void main (String[] args)
    {
        ChequeBook myMoney = new ChequeBook();
        JFrame f = new MoneyTracker("Where's my money?", myMoney,
                                    new BalanceRenderer(myMoney));
    }
}

class BalanceRenderer implements ListCellRenderer
{
    private ChequeBook entries;
    BalanceRenderer (ChequeBook data) { entries = data; }
    public Component getListCellRendererComponent (JList list, Object value,
                                                    int index, boolean isSelected, boolean cellHasFocus)
    {
        JLabel lab = new JLabel("** " + value.toString());
        if (isSelected) { ... } else { ... }
        int amount = entries.getAmount(index);
        if (amount < 0) lab.setForeground(Color.green);
        else if (amount > 500) lab.setForeground(Color.red);
        ... lab.setOpaque(true); return lab;
    }
}

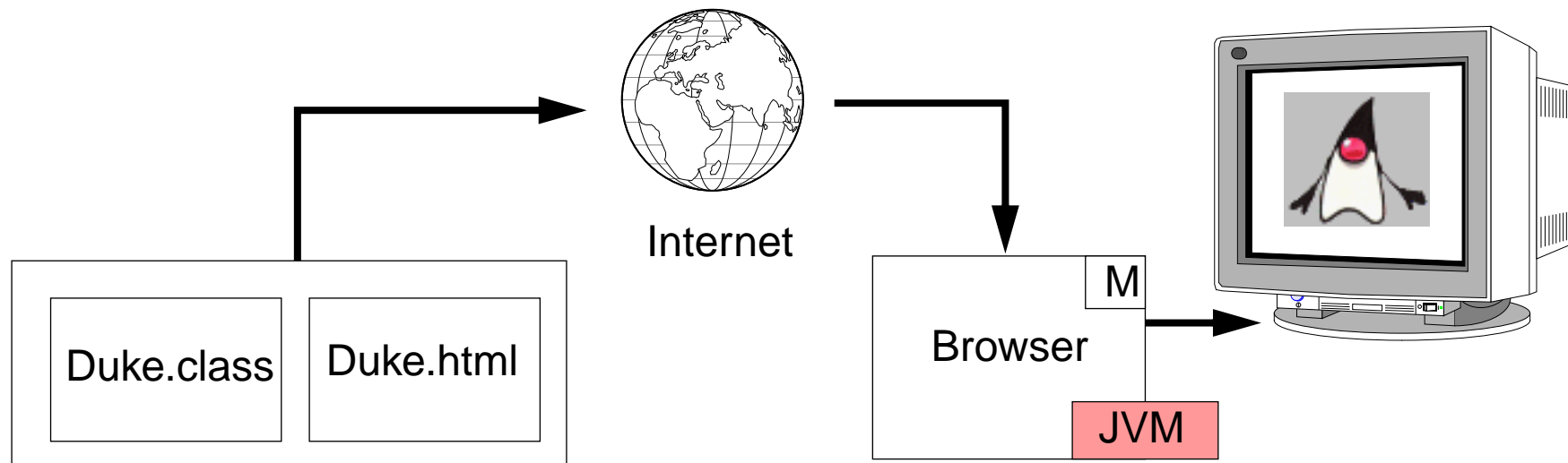
```



## 9. Applets

**Applet** (small application):

- kleines Anwendungsprogramm in Java für eine spezielle Aufgabe,
- an eine WWW-Seite (world wide web) gekoppelt;
- das Applet wird mit der WWW-Seite über das Internet übertragen;
- der Internet Browser (Werkzeug zum Navigieren und Anzeigen von Informationen) arbeitet mit einer Java Virtual Machine (JVM) zusammen; sie führt eintreffende Applets aus.



**Programm (Java-Applet) wird übertragen, läuft beim Empfänger, bewirkt dort Effekte.**

Applets benutzen JVM und Java-Bibliotheken des Empfängers.

# Programmierung von Applets

Applets werden wie Programme mit Swing-Benutzungs Oberfläche geschrieben, aber:

- Die Hauptklasse ist Unterklasse von `JApplet` statt von `JFrame`.
- Es gibt die Methode `main` nicht.
- `System.exit` darf nicht aufgerufen werden.
- Die Methode `public void init ()` tritt an die Stelle des Konstruktors.
- Ein- und Ausgabe mit Swing-Komponenten statt mit Dateien.

Programmschema: Fenster mit Zeichenfläche (GP-88):

```
class WarningApplet extends JApplet
{
    ...
    public void paint (Graphics g)
    { ... auf g schreiben und zeichnen ... }
}
```

Programmschema: Bedienung mit Swing-Komponenten (GP-103):

```
class TrafficLight extends JApplet
{
    public void init ()
    { ... Objektbaum mit Swing-Komponenten aufbauen ... }
    ...
}
```

# Applet-Methoden

Die Klasse `JApplet` liegt in der Hierarchie der graphischen Komponenten:

```

java.awt.Component
  java.awt.Container
    java.awt.Panel
      java.awt.Applet
        javax.swing.JApplet
  
```

`JApplet` definiert Methoden ohne Effekt zum Überschreiben in Unterklassen:

	wird aufgerufen, wenn das Applet ...
<code>void init ()</code>	geladen wird
<code>void start ()</code>	seine Ausführung (wieder-)beginnen soll
<code>void stop ()</code>	seine Ausführung unterbrechen soll
<code>void destroy ()</code>	das Applet beendet wird

Methoden zum Aufruf in Unterklassen von `JApplet`, z. B.

<code>void showStatus (String)</code>	Text in der Statuszeile des Browsers anzeigen
<code>String getParameter (String)</code>	Wert aus HTML-Seite lesen

Weitere Methoden aus Oberklassen, z. B.

<code>void paint (Graphics g)</code>	auf <code>g</code> schreiben und zeichnen (aus <code>Container</code> )
--------------------------------------	---

# Applets zu HTML-Seiten

**HTML** (Hypertext Markup Language):

- Sprache zur Beschreibung formatierter, strukturierter Texte und deren Gestaltung
- Standardsprache zur Beschreibung von Internet-Seiten
- Einfache Sprachstruktur: Marken `<hr>` und Klammern `<ul> ... </ul>`, `<li> ... </li>` mit bestimmter Bedeutung, z. B.

Wir unterscheiden

```
<ul>
  <li>diesen Fall,</li>
  <li>jenen Fall</li>
</ul>
und viele andere Fälle.
<hr>
```

Wir unterscheiden

- diesen Fall,
  - jenen Fall
- und viele andere Fälle.
- 

Ein Applet wird auf einer HTML-Seite aufgerufen, z. B.

```
<title>Catch the Duke!</title>
<hr>
<applet code="CatchM.class" width="300" height="300">
</applet>
```

Beim Anzeigen der HTML-Seite im Browser oder Appletviewer wird das Applet auf einer Fläche der angegebenen Größe ausgeführt.

# Java-Programm in Applet umsetzen

Ein Java-Programm kann man in folgenden Schritten in ein Applet transformieren:

1. Alle Datei-E/A in Benutzung von Swing-Komponenten umsetzen, z. B. `System.out.println(...)` in `g.drawString(...)` in der Methode `paint`.
2. Programm nicht anhalten, `System.exit`-Aufrufe und `close`-Button bzw. Aufrufe von `setDefaultCloseOperation` entfernen.
3. Layoutmanager ggf. explizit wählen, Vorgabe ist `BorderLayout` (wie bei `JFrame`).
4. `javax.swing.JApplet` importieren, Hauptklasse als Unterklasse von `JApplet` definieren
5. Konstruktor durch `init`-Methode ersetzen; darin wird der Objektbaum der Komponenten aufgebaut.
6. `main`-Methode entfernen; die Hauptklasse des Java-Programms entfällt einfach, wenn sie nur die `main`-Methode enthält und darin nur das `JFrame`-Objekt erzeugt und platziert wird.
7. HTML-Datei herstellen mit `<applet>`-Element zur Einbindung der `.class`-Datei.
8. Testen des Applets; erst mit dem `appletviewer` dann mit dem Browser.

siehe Beispiel Ampel-Simulation GP-104 bis 108.

## Parameter zum Start des Applet

Dem Applet können zum Aufruf von der HTML-Seite Daten mitgegeben werden.

Notation: Paare von Zeichenreihen für Parametername und Parameterwert

```
<param name="Parametername" value="Parameterwert">
```

eingesetzt im Applet-Aufruf:

```
<applet code="CoffeeShop.class" width="600" height="200">  
<param name="Columbian" value="12">  
<param name="Java" value="15">  
<param name="Kenyan" value="9">  
</applet>
```

Im Applet auf die Parameterwerte zugreifen:

```
preis = Integer.parseInt (getParameter ("Java"));
```

# Sicherheit und Applets

Beim Internet-Surfen kann man nicht verhindern, dass fremde Applets auf dem eigenen Rechner ausgeführt werden. Deshalb sind ihre **Rechte i. a. stark eingeschränkt**:

Operation	Java Programm	Applet im appletviewer	lokales Applet im Browser	fremdes Applet im Browser
lokale Datei zugreifen	X	X		
lokale Datei löschen	X			
anderes Programm starten	X	X		
Benutzernamen ermitteln	X	X	X	
zum Sender des Applet verbinden	X	X	X	X
zu anderem Rechner verbinden	X	X	X	
Java-Bibliothek laden	X	X	X	
<b>System.exit</b> aufrufen	X	X		
Pop-up Fenster erzeugen	X	X	X	X

Diese Einstellungen der Rechte können im Browser geändert werden.

Man kann auch **signierten Applets** bestimmter Autoren weitergehende Rechte geben.

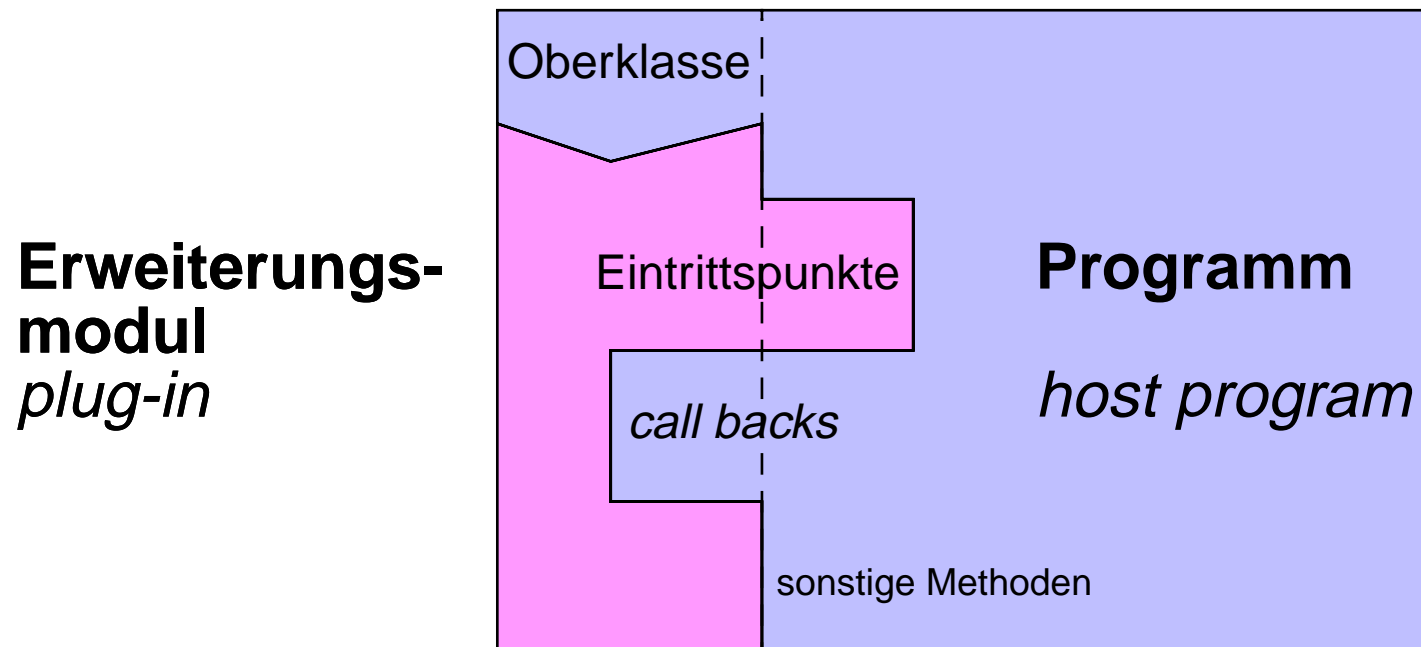
Außerdem wird der **Bytecode** jeder Klasse auf Konsistenz **geprüft**, wenn er in die JVM geladen wird.

# Applets als Beispiel für Erweiterungsmodule (*plug-ins*)

Erweiterungsmodule (*plug-ins*) sind keine selbständigen Programme, sondern werden in ein anderes Programm (*host program*) eingebettet, um dessen Funktionalität zu erweitern.

Zusammenarbeit zwischen Erweiterungsmodul und Programm basiert (in der Regel) auf:

- fest vorgegebener (abstrakter) Oberklasse für das Erweiterungsmodul
- Methoden, die das Modul unbedingt anbieten muss (Eintrittspunkte)
- Methoden, die das Programm speziell für solche Module bereitstellt (*call backs, services*)
- sonstigen Methoden des Programms, die auch zur Verwendung in Erweiterungsmodulen geeignet sind



# 10. Parallele Prozesse

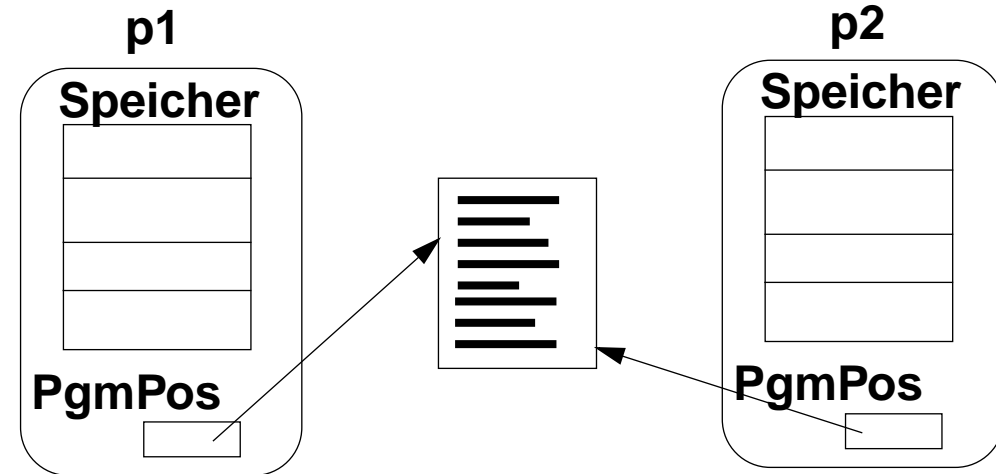
## Grundbegriffe (1)

### Prozess:

**Ausführung** eines sequentiellen Programmstückes in dem zugeordneten Speicher (Adressraum).

**Zustand** des Prozesses: Speicherinhalt und Programmposition

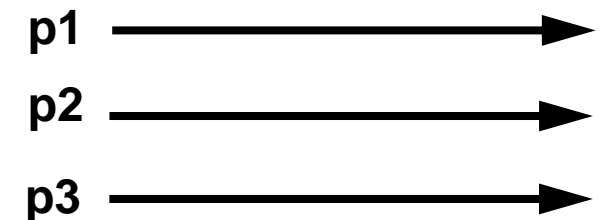
Zustände zweier Prozesse zu demselben Programm



**sequentieller Prozess:** Die Operationen werden eine nach der anderen ausgeführt.

**Parallele Prozesse:** mehrere Prozesse, die **gleichzeitig auf mehreren Prozessoren** ausgeführt werden;

- keine Annahme über relative Geschwindigkeit
- **unabhängige** Prozesse oder
- **kommunizierende** Prozesse:  
p1 liefert Daten für p2 oder  
p1 wartet bis eine Bedingung erfüllt ist



## Grundbegriffe (2)

**verzahnte Prozesse:** mehrere Prozesse, die stückweise **abwechselnd auf einem Prozessor** ausgeführt werden

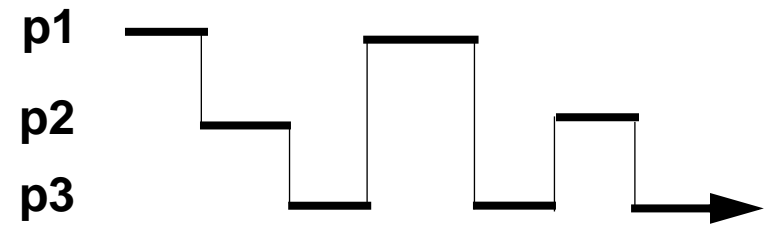
**Prozessumschaltung** durch den Scheduler des Betriebssystems, der JVM oder durch die Prozesse selbst.

**Ziele:**

- Prozessor auslasten, blockierte Prozesse nicht bearbeiten
- Gleichzeitigkeit simulieren

**nebenläufige Prozesse:**

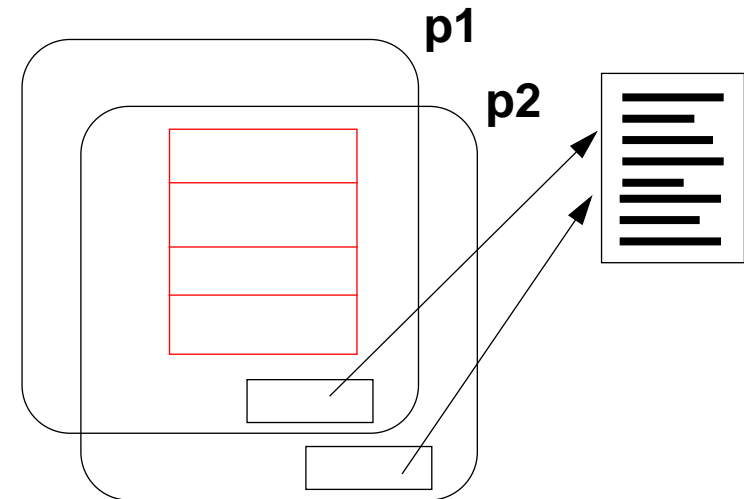
Prozesse, die parallel oder verzahnt ausgeführt werden können



## Grundbegriffe (3)

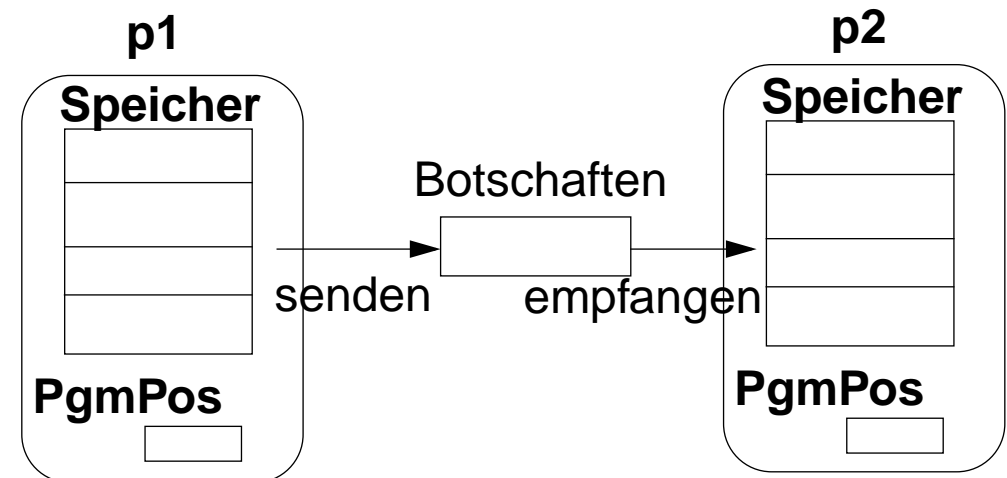
**Threads** (“Fäden“ oder auch “Ausführungsstränge“ der Programmausführung):  
 Prozesse, die parallel oder verzahnt in **gemeinsamem Speicher** ablaufen.

Die **Prozessumschaltung** ist besonders einfach und **schnell**,  
 daher auch **leichtgewichtige Prozesse**



Zugriff auf Variable im  
**gemeinsamen Speicher**

Im Gegensatz zu  
**Prozessen im verteilten Speicher** auf  
 verschiedenen Rechnern,  
**kommunizieren über Botschaften**, statt  
 über Variable im Speicher



# Anwendungen Paralleler Prozesse

- **Benutzungsoberflächen:**  
Die Ereignisse werden von einem speziellen Systemprozess weitergegeben.  
Aufwändige Berechnungen sollten nebenläufig programmiert werden, damit die Bedienung der Oberfläche nicht blockiert wird.
- **Simulation** realer Abläufe:  
z. B. Produktion in einer Fabrik
- **Animation:**  
Veranschaulichung von Abläufen, Algorithmen; Spiele
- **Steuerung** von Geräten:  
Prozesse im Rechner überwachen und steuern externe Geräte, z. B. Montage-Roboter
- **Leistungssteigerung** durch Parallelrechner:  
mehrere Prozesse bearbeiten gemeinsam die gleiche Aufgabe,  
z. B. paralleles Sortieren großer Datenmengen.

# Java Threads erzeugen (1)

Prozesse, die nebenläufig, gemeinsam im Speicher des Programms bzw. Applets ablaufen.

Es gibt 2 Techniken Threads zu erzeugen. Erste Technik:

Eine Benutzerklasse implementiert das Interface `Runnable`:

```
class Aufgabe implements Runnable
{ ...
  public void run ()           vom Interface geforderte Methode run
  {...}                       das als Prozess auszuführende Programmstück
  Aufgabe (...) {...}        Konstruktor
}
```

Der Prozess wird als Objekt der vordefinierten Klasse `Thread` erzeugt:

```
Thread auftrag = new Thread (new Aufgabe (...));
```

Erst folgender Aufruf startet dann den Prozess:

```
auftrag.start();      Der neue Prozess beginnt, neben dem hier aktiven zu laufen.
```

Diese Technik (das Interface `Runnable` implementieren) sollte man anwenden, wenn der **abgespaltene Prozess nicht weiter beeinflusst** werden muss; also einen Auftrag erledigt (Methode `run`) und dann terminiert.

## Java Threads erzeugen (2)

Zweite Technik:

Die Benutzerklasse wird als Unterklasse der vordefinierten Klasse `Thread` definiert:

```
class DigiClock extends Thread
{
    ...
    public void run ()                überschreibt die Thread-Methode run
    {...}                            das als Prozess auszuführende Programmstück
    DigiClock (...) {...}            Konstruktor
}
```

Der Prozess wird als Objekt der Benutzerklasse erzeugt (es ist auch ein `Thread`-Objekt):

```
Thread clock = new DigiClock (...);
```

Erst folgender Aufruf startet dann den Prozess:

```
clock.start();                der neue Prozess beginnt neben dem hier aktiven zu laufen
```


Diese Technik (Unterklasse von `Thread`) sollte man anwenden, wenn der abgespaltene Prozess weiter beeinflußt werden soll; also weitere Methoden der Benutzerklasse definiert und von aussen aufgerufen werden, z. B. zum vorübergehenden Anhalten oder endgültigem Terminieren.

# 11. Unabhängige parallele Prozesse

## Beispiel: Digitale Uhr als Prozess im Applet (1)

Der Prozess soll in jeder Sekunde Datum und Uhrzeit als Text aktualisiert anzeigen.

## Digital Clock Applet



16:36:32

```
class DigiClock extends Thread
```

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

```
{ while (running)
```

```
{ line.setText(timeFormat.format(new Date())); Datum schreiben
```

```
try { sleep (1000); } catch (InterruptedException e) {} Pause
```

```
}
```

```
}
```

Methode, die den Prozeß von außen terminiert:

```
public void stopIt () { running = false; }
```

```
private boolean running = true; Zustandsvariable
```

```
public DigiClock (JLabel t) {line = t;} Label zum Beschreiben übergeben
```

```
private JLabel line;
```

```
private DateFormat timeFormat = DateFormat.getTimeInstance(  
DateFormat.MEDIUM, Locale.GERMANY);
```

```
}
```

Prozess als **Unterklasse** von **Thread**, weil er

- durch Aufruf von `stopIt` terminiert und
- durch weitere **Thread**-Methoden unterbrochen werden soll.

## Beispiel: Digitale Uhr als Prozess im Applet (2)

Der Prozess wird in der `init`-Methode der `JApplet`-Unterklasse erzeugt:

```
public class DigiApp extends JApplet
{
    public void init ()
    {
        JLabel clockText = new JLabel ("--:--:--");
        getContentPane().add(clockText);

        clock = new DigiClock(clockText);
        clock.start();
    }

    public void start ()    { clock.resume(); }
    public void stop ()    { clock.suspend(); }
    public void destroy () { clock.stopIt(); }

    private DigiClock clock;
}

```

Prozess erzeugen  
 Prozess starten  
  
 Prozess fortsetzen  
 Prozess anhalten  
 Prozess terminieren

Prozesse, die in einem Applet gestartet werden,

- sollen angehalten werden (`suspend`, `resume`), solange das Applet nicht sichtbar ist (`stop`, `start`),
- müssen terminiert werden (`stopIt`), wenn das Applet entladen wird (`destroy`).

Andernfalls belasten Sie den Rechner, obwohl sie nicht mehr sichtbar sind.

# Wichtige Methoden der Klasse Thread

```
public void run ();
```

wird überschrieben mit der Methode, die die auszuführenden Anweisungen enthält

```
public void start ();
```

startet die Ausführung des Prozesses

```
public void join () throws InterruptedException;
```

der aufrufende Prozess wartet bis der angegebene Prozess terminiert ist:

```
try { auftrag.join(); } catch (InterruptedException e) {}
```

```
public static void sleep (long millisec) throws InterruptedException;
```

der aufrufende Prozess wartet mindestens die in Millisekunden angegebene Zeit:

```
try { Thread.sleep (1000); } catch (InterruptedException e) {}
```

```
public void suspend ();
```

```
public void resume ();
```

hält den angegebenen Prozess an bzw. setzt ihn fort:

```
clock.suspend(); clock.resume();
```

unterbricht den Prozess u. U. in einem kritischen Abschnitt, nur für **start/stop** im Applet

```
public final void stop () throws SecurityException;
```

nicht benutzen! Terminiert den Prozess u. U. in einem inkonsistenten Zustand

# Beispiel: Ampelsimulation (1)

## Aufgabe:

- Auf einer Zeichenfläche mehrere Ampeln darstellen.
- Der Prozess jeder Ampel durchläuft die Ampelphasen und zeichnet die Lichter entsprechend.
- Vereinfachungen:  
1 Ampel; keine Einstellung der Phasendauer; keine Grünanforderung für Fußgänger.

## Lösung:

- Ampel-Objekte aus einer Unterklasse (`SetOfLights`) von `Thread` erzeugen (Methode 2).
- Ampel-Objekte zeichnen auf gemeinsamer Zeichenfläche (`JComponent`-Objekt).
- `run`-Methode wechselt die Ampelphasen und ruft eine Methode zum Zeichnen der Lichter auf (`drawLights`).
- Eine weitere Methode (`draw`) zeichnet die unveränderlichen Teile des Ampel-Bildes.
- Die `paint`-Methode der Zeichenfläche ruft beide Zeichenmethoden der Ampel-Objekte auf (Delegation der Zeichenaufgabe).

## Prozessklasse der Ampelsimulation

```

class SetOfLights extends Thread
{
    /* Jedes Objekt dieser Klasse zeichnet eine Ampel an vorgegebener
       Position x auf ein gemeinsames JComponent-Objekt area. */

    private JComponent area;                // gemeinsame Zeichenfläche
    private int x;                          // x-Koordinate dieser Ampel
    private int light = 0;                  // aktuelle Ampelphase

    public SetOfLights (JComponent area, int x)
    { this.area = area; this.x = x; }

    public void run ()
    { while (running)                       // drei Ampelphasen wiederholen
      { for (light = 0; light < 3; light++)
        { Graphics g = area.getGraphics();  // Kontext für Zeichenfläche
          drawLights (g);                   // neuen Zustand zeichnen
          g.dispose();                      // Systemressourcen freigeben
          try { sleep(500); }               // nach fester Dauer Zustand wechseln
            catch (InterruptedException e) { }
        }
      }
    }

    public void stopIt () { running = false; } // Prozess von außen anhalten
    private boolean running = true;
    public void draw (Graphics g) {...}      // unveränderliche Bildteile zeichnen
    public void drawLights (Graphics g) {...} // veränderliche Bildteile zeichnen
}

```

## Zeichenmethoden der Ampelsimulation

```
class SetOfLights extends Thread
{
    ...
    public void draw (Graphics g)
    {
        // unveränderliche Teile des Bildes relativ zu x zeichnen und schreiben:
        g.drawOval(x-8, 10, 30, 68); // der Ampelumriss
    }

    public void drawLights (Graphics g)
    {
        // veränderliche Teile des Bildes zeichnen:

        if (light == 0) g.setColor(Color.red); // die 4 Lichter
        else g.setColor(Color.lightGray);
        g.fillOval(x, 15, 15, 15);

        if (light == 1) g.setColor(Color.yellow);
        else g.setColor(Color.lightGray);
        g.fillOval(x, 35, 15, 15);

        if (light == 2) g.setColor(Color.green);
        else g.setColor(Color.lightGray);
        g.fillOval(x, 55, 15, 15);

        g.setColor(Color.green); g.fillOval(x, 85, 15, 15);
    }
}
```

## Applet-Klasse der Ampelsimulation

```

public class TrafficLight extends JApplet
{
    private JComponent area;                // gemeinsame Zeichenfläche
    private int lightsPosition = 105;      // x-Koordinate der nächsten Ampel
    private SetOfLights lights;           // zunächst nur eine Ampel

    public void init ()
    {
        setLayout(new BorderLayout ());
        area =                                // das JComponent-Objekt zum Zeichnen der Ampeln
            new JComponent()                 // anonyme Unterklasse mit 1 Objekt:
            {
                public void paint (Graphics g) // Zeichnen an Ampel-Objekt delegieren
                {
                    lights.draw(g);          // zeichne statischen Teil
                    lights.drawLights(g);    // zeichne veränderlichen Teil
                }
            };
        add (area, BorderLayout.CENTER);

        lights = new SetOfLights(area, lightsPosition); // 1 Ampel-Objekt
        area.repaint();                                // erstes Zeichnen anstoßen
        lights.start();                                // Prozess starten
    }

    // Prozesse des Applets anhalten, fortsetzen, beenden:
    public void stop ()    { lights.suspend(); }
    public void start ()   { lights.resume();  }
    public void destroy () { lights.stopIt();  }
}

```

## 12. Monitore, Synchronisation: Gegenseitiger Ausschluss

Wenn mehrere Prozesse die **Werte gemeinsamer Variablen verändern**, kann eine ungünstige Verzahnung (oder echte Parallelausführung) zu inkonsistenten Daten führen.

Z. B. zwei Prozesse benutzen lokale Variable `tmp` und eine gemeinsame Variable `konto`:

```

p   tmp = konto;                               konto = tmp+10;
q           tmp = konto; konto = tmp+10;

```

**Kritische Abschnitte:** zusammengesetzte Operationen, die gemeinsame Variablen lesen und/oder verändern.

Prozesse müssen kritische Abschnitte unter **gegenseitigem Ausschluss** (*mutual exclusion*) ausführen:

D. h. **zu jedem Zeitpunkt darf höchstens ein Prozess** solch einen kritischen Abschnitt für bestimmte Variablen ausführen. Andere Prozesse, die für die gleichen Variablen mit der Ausführung eines kritischen Abschnitts beginnen wollen, müssen warten.

# Gegenseitiger Ausschluss durch Monitore

## Monitor:

Ein Modul, der Daten und Operationen darauf kapselt.

Die **kritischen Abschnitte** auf den Daten werden als **Monitor-Operationen** formuliert.

Prozesse rufen Monitor-Operationen auf, um auf die Daten zuzugreifen.

Die Monitor-Operationen werden unter gegenseitigem Ausschluss ausgeführt.

## Monitore in Java:

Methoden einer Klasse, die kritische Abschnitte auf Objektvariablen implementieren, können als **synchronized** gekennzeichnet werden:

```
class Bank
{  public synchronized void abbuchen (...)  {...}
   public synchronized void überweisen (...) {...}
   ...
   private int[] konten;
}
```

**Jedes Objekt** der Klasse wirkt dann als **Monitor** für seine Objektvariablen:

**Aufrufe von synchronized Methoden** von mehreren Prozessen für dasselbe Objekt werden unter **gegenseitigem Ausschluss** ausgeführt.

Zu jedem Zeitpunkt kann höchstens 1 Prozess mit **synchronized** Methoden auf Objektvariable zugreifen.

## Beispiel: gegenseitiger Ausschluss durch Monitor

```
class Bank
{
    public synchronized void abbuchen (String name) // kritischer Abschnitt
    {
        int vorher = konto;
        System.out.println (name + " Konto vor : " + vorher);
        // kritische Verwendung der Objektvariablen konto:
        konto = vorher - 10;
        System.out.println (name + " Konto nach: " + konto);
    }
    private int konto = 100;
}

class Kunde extends Thread
{
    Kunde (String n, Bank b) { name = n; meineBank = b; }
    public void run ()
    {
        for (int i = 1; i <= 2; i++) meineBank.abbuchen(name);
    }
    private Bank meineBank; private String name;
}

class BankProgramm
{
    public static void main (String[] args)
    {
        Bank b = new Bank(); // Objekt als Monitor für seine Objektvariable
        new Kunde("ich", b).start(); new Kunde("du ", b).start();
    }
}
```

## 13. Bedingungssynchronisation im Monitor

### Bedingungssynchronisation:

Ein Prozess wartet, bis eine Bedingung erfüllt ist — verursacht durch einen anderen Prozess; z. B. erst dann in einen Puffer schreiben, wenn er nicht mehr voll ist.

### Bedingungssynchronisation im Monitor:

Ein Prozess, der in einer kritischen Monitor-Operation auf eine Bedingung wartet, muss den Monitor freigeben, damit andere Prozesse den Zustand des Monitors ändern können.

### Bedingungssynchronisation in Java:

Vordefinierte Methoden der Klasse `Object` zur Bedingungssynchronisation:  
(Sie müssen aus `synchronized` Methoden heraus aufgerufen werden.)

- `wait()`            **blockiert den aufrufenden Prozess und gibt den Monitor frei**  
– das Objekt, dessen `synchronized` Methode er gerade ausführt.
- `notifyAll()`   **weckt alle in diesem Monitor blockierten Prozesse;**  
sie können weiterlaufen, sobald der Monitor frei ist.
- `notify()`        weckt einen (beliebigen) blockierten Prozess;  
ist für unser Monitorschema nicht brauchbar

Nachdem ein blockierter Prozess geweckt wurde, muß er die **Bedingung, auf die er wartet, erneut prüfen** — sie könnte schon durch schnellere Prozesse wieder invalidiert sein:

```
while (avail < n) try { wait(); } catch (InterruptedException e) { }
```

## Beispiel: Monitor zur Vergabe von Ressourcen

**Aufgabe:** Eine Begrenzte Anzahl gleichartiger Ressourcen wird von einem Monitor verwaltet. Prozesse fordern einige Ressourcen an und geben sie später zurück.

```
class Monitor
{ private int avail;           // Anzahl verfügbarer Ressourcen
  Monitor (int a) { avail = a; }

  synchronized void getElem (int n, int who)    // n Elemente abgeben
  { System.out.println("Client"+who+" needs "+n+",available "+avail);

    while (avail < n)                          // Bedingung in Warteschleife prüfen
    { try { wait(); } catch (InterruptedException e) {}
      }                                          // try ... catch nötig wegen wait()
    avail -= n;
    System.out.println("Client"+who+" got "+n+", available "+avail);
  }

  synchronized void putElem (int n, int who) // n Elemente zurücknehmen
  { avail += n;
    System.out.println("Client"+who+" put "+n+",available "+avail);
    notifyAll();                               // alle Wartenden können die Bedingung erneut prüfen
  }
}
```

## Beispiel: Prozesse und Hauptprogramm zu GP-146

```

import java.util.Random;

class Client extends Thread
{
    private Monitor mon; private Random rand;
    private int ident, rounds, max;
    Client (Monitor m, int id, int rd, int avail)
    {
        ident = id; rounds = rd; mon = m; max = avail;
        rand = new Random(); // Neuer Zufallszahlengenerator
    }

    public void run ()
    {
        while (rounds > 0)
        {
            int m = rand.nextInt(max) + 1;
            mon.getElem (m, ident); // m Elemente anfordern
            try { sleep (rand.nextInt(1000) + 1); }
                catch (InterruptedException e) {}
            mon.putElem (m, ident); // m Elemente zurückgeben
            rounds -= 1;
        }
    }
}

```

```

class TestMonitor
{
    public static void main (String[] args)
    {
        int avail = 20;
        Monitor mon = new Monitor (avail);
        for (int i = 0; i < 5; i++)
            new Client(mon,i,4,avail).start();
    }
}

```

## Schema: Gleichartige Ressourcen vergeben

Ein **Monitor** verwaltet eine endliche Menge von  $k \geq 1$  **gleichartigen Ressourcen**.

**Prozesse** fordern jeweils unterschiedlich viele ( $n$ ) Ressourcen an,  $1 \leq n \leq k$  und geben sie nach Gebrauch wieder zurück.

Die **Wartebedingung** für das Anfordern ist "Sind  $n$  Ressourcen verfügbar?"

Zu jedem Zeitpunkt zwischen Aufrufen der Monitor-Methoden gilt:  
"Die Summe der freien und der vergebenen Ressourcen ist  $k$ ."

Die Monitor-Klasse auf GP-146 implementiert dieses Schema.

### Beispiele:

- Walkman-Vergabe an Besuchergruppen im Museum (siehe [Java lernen, 13.5])
- Taxi-Unternehmen;  $n = 1$

auch **abstrakte Ressourcen**, z. B.

- das Recht, eine Brücke begrenzter Kapazität (Anzahl Fahrzeuge oder Gewicht) zu befahren,

auch **gleichartige Ressourcen mit Identität**:

Der Monitor muß dann die Identitäten der Ressourcen in einer Datenstruktur verwalten.

- Nummern der vom Taxi-Unternehmen vergebenen Taxis
- Adressen der Speicherblöcke, die eine Speicherverwaltung vergibt

# Schema: Beschränkter Puffer

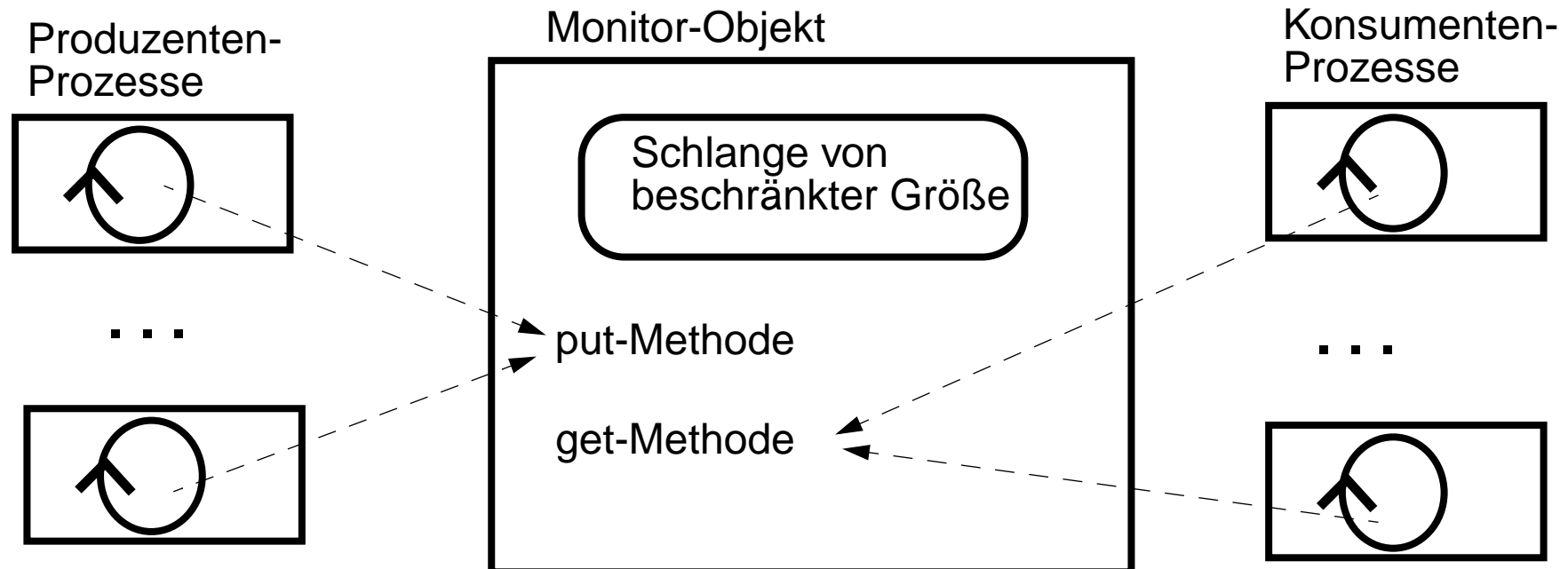
Ein Monitor speichert Elemente in einem **Puffer von beschränkter Größe**.

**Produzenten-Prozesse** liefern einzelne Elemente,  
**Konsumenten-Prozesse** entnehmen einzelne Elemente.

Der Monitor stellt sicher, dass die Elemente in der gleichen **Reihenfolge** geliefert und entnommen werden. (Datenstruktur: Schlange)

Die **Wartebedingungen** lauten:

- in den Puffer **schreiben**, nur wenn er **nicht voll** ist,
- aus dem Puffer **entnehmen**, nur wenn er **nicht leer** ist.



## Monitor-Klasse für “Beschränkter Puffer”

```
class Buffer
{ private Queue buf;          // Schlange der Länge n zur Aufnahme der Elemente
  public Buffer (int n) {buf = new Queue(n); }

  public synchronized void put (Object elem)
  {
    // ein Produzenten-Prozess versucht, ein Element zu liefern
    while (buf.isFull())          // warten bis der Puffer nicht voll ist
      try {wait();} catch (InterruptedException e) {}
    buf.enqueue(elem); // Wartebedingung der get-Methode hat sich verändert
    notifyAll();        // jeder blockierte Prozess prüft seine Wartebedingung
  }

  public synchronized Object get ()
  {
    // ein Konsumenten-Prozess versucht, ein Element zu entnehmen
    while (buf.isEmpty())        // warten bis der Puffer nicht leer ist
      try {wait();} catch (InterruptedException e) {}
    Object elem = buf.first();
    buf.dequeue(); // Wartebedingung der put-Methode hat sich verändert
    notifyAll();   // jeder blockierte Prozess prüft seine Wartebedingung
    return elem;
  }
}
```

## 14. Verklemmungen

Ein **einzelner Prozess ist verklemmt** (in einer *Deadlock*-Situation), wenn er auf eine **Bedingung wartet, die nicht mehr wahr werden kann**.

**Mehrere Prozesse** sind untereinander verklemmt (in einer *Deadlock*-Situation), wenn sie **zyklisch aufeinander warten**;

d. h. die Wartebedingung eines Prozesses kann nur von einem anderen, ebenfalls wartenden Prozess erfüllt werden.

**Verklemmung bei Ressourcenvergabe** kann eintreten, wenn folgende Bedingungen gelten:

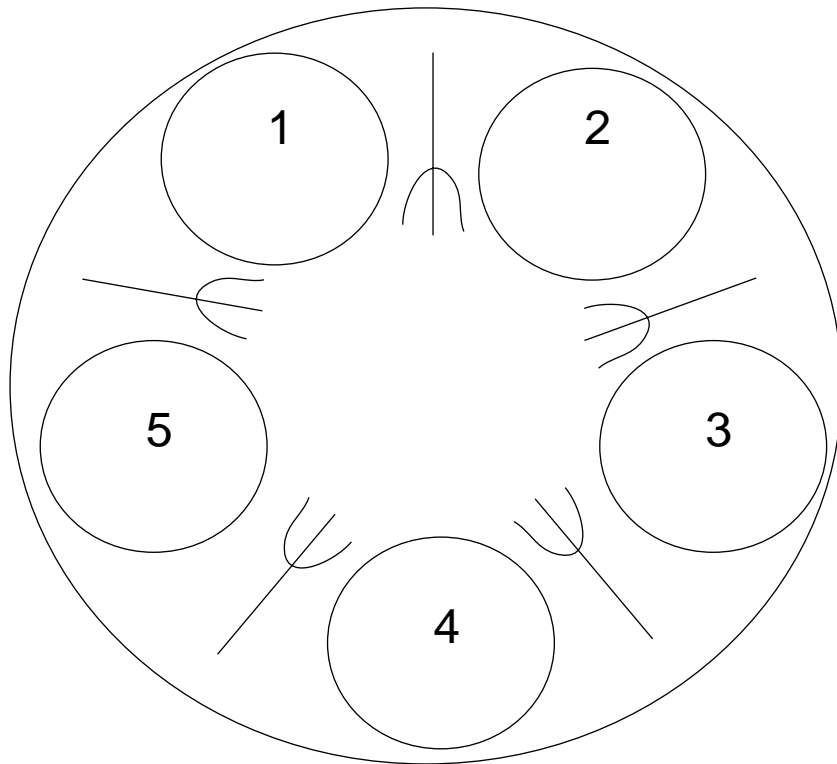
1. Jeder Prozess **fordert mehrmals nacheinander Ressourcen an**.
2. Jeder Prozess benötigt bestimmte Ressourcen **exklusiv** für sich allein.
3. Die **Relation** "Prozess  $p$  benötigt eine Ressource, die Prozess  $q$  hat" ist **zyklisch**.

Beispiel: Dining Philosophers (GP-152)

# Synchronisationsaufgabe: Dining Philosophers

Abstraktion von Prozesssystemen, in denen die Prozesse **mehrere Ressourcen exklusiv** benötigen. (Erstmals formuliert von E. Dijkstra 1968.)

*5 Philosophen sitzen an einem gedeckten Tisch. Jeder beschäftigt sich abwechselnd mit denken und essen. Wenn ein Philosoph Hunger verspürt, versucht er die beiden Gabeln neben seinem Teller zu bekommen, um damit zu essen. Nachdem er gesättigt ist, legt er die Gabeln ab und verfällt wieder in tiefgründig philosophisches Nachdenken.*



Abstraktes Programm für die Philosophen:

```
wiederhole
  denken
  nimm rechte Gabel
  nimm linke Gabel
  essen
  gib rechte Gabel
  gib linke Gabel
```

Es gibt einen Verklemmungszustand:

Für alle Prozesse  $i$  aus  $\{1, \dots, 5\}$  gilt:  
 Prozess  $i$  hat seine rechte Gabel  
 und wartet auf seine linke Gabel.

# Dining Philosophers, Lösung (1)

Lösungsprinzip: Ressourcen (Gabeln) **zugleich anfordern** — nicht nacheinander.

Ein **Monitor** verwaltet die Gabeln als **Ressourcen mit Identität**.

**Monitor-Operation:** Dem Prozess  $i$  die linke und rechte Gabel geben.

**Wartebedingung** dafür: Die beiden Gabeln für den  $i$ -ten Prozess sind frei.

```
class ResourcePairs
{
    // nur Paare aus "benachbarten" Ressourcen werden vergeben
    private boolean[] avail = {true, true, true, true, true};

    synchronized void getPair (int i)
    { while (!(avail[i % 5] & avail[(i+1) % 5]))
        try {wait();} catch (InterruptedException e) {}

        avail[i % 5] = false; avail[(i+1) % 5] = false;
    }

    synchronized void putPair (int i)
    { avail[i % 5] = true; avail[(i+1) % 5] = true;
      notifyAll();
    }
}
```

Der Verklemmungszustand "Jeder Prozess hat **eine** Gabel." kann nicht eintreten.

## Dining Philosophers, Lösung (2)

**Lösungsprinzip:** Die Aussage die den Verklemmungszustand charakterisiert negieren.

*Für alle Prozesse  $i$  aus  $\{1, \dots, 5\}$  gilt: Prozess  $i$  hat seine rechte Gabel und wartet.*

Negation liefert Lösungsidee:

*Es gibt einen Prozess  $i$  aus  $\{1, \dots, 5\}$ : Prozess  $i$  hat seine rechte Gabel nicht oder er wartet nicht.*

**Lösung:** Einer der Prozesse, z. B. 1, nimmt erst die linke dann die rechte Gabel.

Der Verklemmungszustand “Jeder Prozess hat **eine** Gabel.” kann nicht eintreten.

Die Bedingung (3) von GP-151 “zyklisches Warten” wird invalidiert.

Ein **Monitor** verwaltet die Gabeln als **Ressourcen mit Identität**.

**Monitor-Operation:** Dem Prozess  $i$  die  $i$ -te Gabel (**einzeln!**) geben.

**Wartebedingung** dafür: Die  $i$ -te Gabel ist frei.

# Zusammenfassung von GP II

## allgemeine Begriffe und Konzepte

## Java Programmierung

### Graphische Bedienungsflächen

Komponenten-Bibliothek  
 hierarchische Objektstrukturen  
 Eingabekomponenten  
 Observer-Muster  
 Reaktion auf Ereignisse, Folgen

Swing (AWT)  
**LayoutManager**  
**JButton, JTextField, ...**  
 Model/View-Paradigma  
 Applets

### Parallele Prozesse

Grundbegriffe  
 parallel, verzahnt, nebenläufig

Monitore  
 gegenseitiger Ausschluss  
 Bedingungssynchronisation im Monitor

Schema: Vergabe gleichartiger Ressourcen  
 Schema: Beschränkter Puffer  
 Schema: Ressourcenvergabe

Verklemmung

Java-Threads, Prozess-Objekte  
 Interface **Runnable**, Oberklasse **Thread**  
**Thread**-Methoden, Threads in Applets  
 Monitor-Objekte in Java  
**synchronized** Methoden  
 Methoden **wait**, **notifyAll**