Fortgeschrittene Objektorientierung

GUI-Programmieren in Java

Gregor Snelting
Andreas Lochbihler

Universität Karlsruhe
Lehrstuhl Programmierparadigmen

27. Mai 2008

Übersicht

Swing (→ 23)

Swing ist eine Bibliothek von Java

- zur Programmierung von graphischen Benutzeroberflächen nach dem Baukastenprinzip
- Nachfolger des Abstract Window Toolkit (AWT)
- Plattformunabhängig
- Durchgängig objektorientiert

Hello, world!

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

public class SwingHelloWorld {
  public static void main(String[] args) {
    JFrame frame = new JFrame();

    frame.getContentPane().add(new JLabel("Hello, world!"));

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(100, 100);
    frame.setVisible(true);
  }
}
```

1. Fenster erzeugen (`JFrame`)
2. Schriftzug “Hello, world!” einfügen (`JLabel`)
3. Programm soll sich beim Schließen des Fensters beenden
4. Größe festlegen
5. Anzeigen

Hello, world!


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

public class SwingHelloWorld {
  public static void main(String[] args) {
    JFrame frame = new JFrame();

    frame.getContentPane().add(new JLabel("Hello, world!"));

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(100, 100);
    frame.setVisible(true);
  }
}
```

1. Fenster erzeugen (`JFrame`)
2. Schriftzug “Hello, world!” einfügen (`JLabel`)
3. Programm soll sich beim Schließen des Fensters beenden
4. Größe festlegen
5. Anzeigen

Nun eine Schaltfläche

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

public class ButtonXpl {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Button example");
    frame.setLayout(new FlowLayout());

    Container c = frame.getContentPane();
    c.add(new JButton("Click me!"));
    c.add(new JButton("Ignore me!"));

    c.add(new JTextField("Type something! Please!"));

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(200, 150);
    frame.setVisible(true);
  }
}
```

Aber: Noch keine Reaktion auf Benutzeraktionen

Nun eine Schaltfläche


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

public class ButtonXpl {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Button example");
    frame.setLayout(new FlowLayout());

    Container c = frame.getContentPane();
    c.add(new JButton("Click me!"));
    c.add(new JButton("Ignore me!"));

    c.add(new JTextField("Type something! Please!"));

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(200, 150);
    frame.setVisible(true);
  }
}
```

Aber: Noch keine Reaktion auf Benutzeraktionen

Reaktion auf Benutyeraktionen: Events (→ 25)

GUI-Programmierung ist *Event-driven Programming*. Was sind Events?

- Mausklicks
- Mausbewegungen
- Eingaben auf der Tastatur
- ...

Wie bekommt man Events?

⇒ **Registrieren** bei dem Swing Widget mit einem **Event-Listener**

Verschiedene Kategorien von Events:

- Action
- Mouse
- MouseMotion
- Keyboard
- Focus
- ...

ActionListener

```
import javax.swing.*; import java.awt.*; import java.awt.event.*;
public class ActionListenerDemo extends JFrame {
  private JButton button1, button2;
  private JTextField textfield;
  public ActionListenerDemo() {
    super("Button example");

    ButtonListener bl = new ButtonListener();

    button1 = new JButton("Click me!");
    button1.setActionCommand("First button");
    button1.addActionListener(bl);

    button2 = new JButton("Ignore me!");
    button2.setActionCommand("Second button");
    button2.addActionListener(bl);

    textfield = new JTextField("Type something! Please!");

    this.setLayout(new FlowLayout());
    Container c = getContentPane();
    c.add(button1); c.add(button2); c.add(textfield);
  }
  private class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      textfield.setText(e.getActionCommand());
  } }
  public static void main(String[] args) {
    JFrame frame = new ActionListenerDemo();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(200, 150); frame.setVisible(true);
  } }
```

Reaktion auf Klicks:

`ActionListener`
implementieren

`actionPerformed`
definiert Reaktion

`ActionEvent` speichert Informationen zum Event

`addActionListener`
Verknüpfung von Widget und `ActionListener`

`setActionCommand`
setzt `String`-Parameter für ActionEvent

Das Swing-Eventmodell (→25.2)

Eventgetriebene Aktionen in Swing:

1. Swing-Komponente "feuert" Event
2. Aufruf der entsprechenden Methode in allen registrierten Listener
3. Verarbeitung in den Listenern

Verschiedene Klassen für verschiedene Events: Je Event-Typ

- eine Event-Klasse
- eine Listener-Klasse
- eine Registrierungsmethode **`addXXXListener(...)`**

Aufteilung in **Algorithmik**, **Darstellung** und **Aktionsverarbeitung**

Eventtypen und Listener (→25.3.3)

`ActionEvent` Verarbeitet durch `ActionListener`

Unterstützt von `JButton`, `JList`, `JTextField`, `JMenuItem`, `JMenu`, `JPopupMenu`

⇒ `addActionListener(...)`, `removeActionListener(...)`

`MouseEvent` Verarbeitet durch `MouseListener`

Unterstützt von allen Komponenten

⇒ `addMouseListener(...)`, `removeMouseListener(...)`

u.v.a.m. siehe Java-API, Pepper: 25.3.3

Listener-Interfaces und Adapter

```
public interface MouseListener extends EventListener {
  void mouseClicked(MouseEvent e);
  void mouseEntered(MouseEvent e);
  void mouseExited(MouseEvent e);
  void mousePressed(MouseEvent e);
  void mouseReleased(MouseEvent e);
}
```

Oft nur eine Methode benötigt $\Rightarrow$ `MouseAdapter`

```
public class MouseAdapter implements MouseListener ... {
  void mouseClicked(MouseEvent e) {}
  void mouseEntered(MouseEvent e) {}
  void mouseExited(MouseEvent e) {}
  void mousePressed(MouseEvent e) {}
  void mouseReleased(MouseEvent e) {}
... }
```

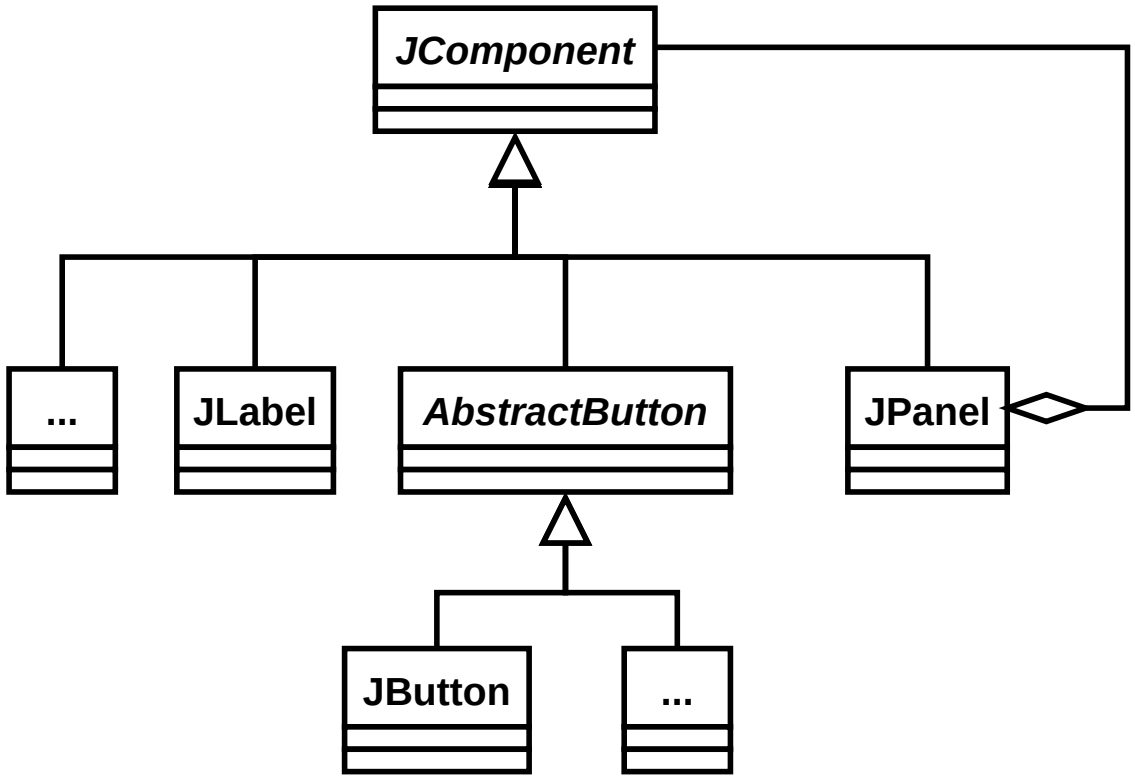
$\Rightarrow$ **Vererben von `MouseAdapter` und Methode überschreiben**

Analog für alle anderen Event-Typen

Hierarchischer Aufbau von GUIs

Basis-Widgets elementare GUI-Komponenten wie
`JButton`, `JTextField`, `JLabel`, ...

Container Behälter für andere Widgets
`JFrame`, `JPanel`, ...



Layout-Manager (→ 24.3.4)

Layout-Manager ordnen Elemente in Containern an:

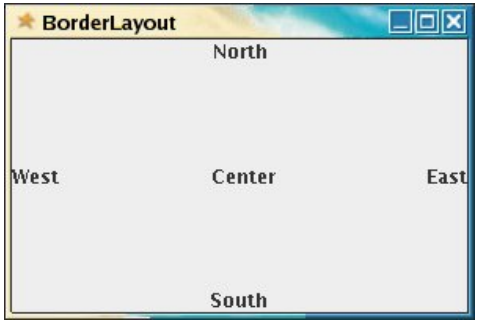
`FlowLayout`	Anordnung nach Einfügereihenfolge von links nach rechts, von oben nach unten
`BorderLayout`	Einfache Positionsangabe (oben, unten, links, rechts, mittig)
`GridLayout`	Tabellenlayout mit fester Spalten- und Zeilenzahl, alle Zellen sind gleich gross
`BoxLayout`	Alles in einer Zeile oder einer Spalte
AbsolutePositioning	Ganz ohne Layout-Manager mühsam und schlecht portabel

Setzen mit `setLayout(new ...Layout(...))`

BorderLayout

```
import javax.swing.*; import java.awt.*;
public class BorderLayout1 {
  public static void main(String[] args) {
    JFrame frame = new JFrame("BorderLayout");
    frame.setLayout(new BorderLayout());

    Container c = frame.getContentPane();
    c.add(new JLabel("North"), BorderLayout.NORTH);
    c.add(new JLabel("South"), BorderLayout.SOUTH);
    c.add(new JLabel("East"), BorderLayout.EAST);
    c.add(new JLabel("West"), BorderLayout.WEST);
    c.add(new JLabel("Center"), BorderLayout.CENTER);

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 200);
    frame.setVisible(true);
  }
}
```

BorderLayout



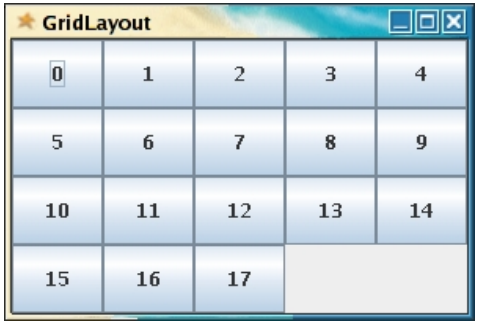

```
import javax.swing.*; import java.awt.*;
public class BorderLayout1 {
  public static void main(String[] args) {
    JFrame frame = new JFrame("BorderLayout");
    frame.setLayout(new BorderLayout());

    Container c = frame.getContentPane();
    c.add(new JLabel("North"), BorderLayout.NORTH);
    c.add(new JLabel("South"), BorderLayout.SOUTH);
    c.add(new JLabel("East"), BorderLayout.EAST);
    c.add(new JLabel("West"), BorderLayout.WEST);
    c.add(new JLabel("Center"), BorderLayout.CENTER);

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 200);
    frame.setVisible(true);
  }
}
```

GridLayout

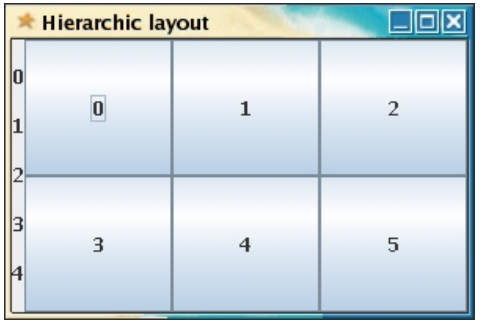
```
import javax.swing.*; import java.awt.*;
public class GridLayout1 {
  public static void main(String[] args) {
    JFrame frame = new JFrame("GridLayout");
    frame.setLayout(new GridLayout(4, 5));

    Container c = frame.getContentPane();
    for (int i = 0; i < 18; i++) {
      c.add(new JButton(String.valueOf(i)));
    }

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 200);
    frame.setVisible(true);
  }
}
```

GridLayout

```
import javax.swing.*; import java.awt.*;
public class GridLayout1 {
  public static void main(String[] args) {
    JFrame frame = new JFrame("GridLayout");
    frame.setLayout(new GridLayout(4, 5));

    Container c = frame.getContentPane();
    for (int i = 0; i < 18; i++) {
      c.add(new JButton(String.valueOf(i)));
    }

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 200);
    frame.setVisible(true);
  }
}
```



Beispiel: Hierarchisches GUI-Design




```
import javax.swing.*; import java.awt.*;
public class HierarchicLayout {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Hierarchic layout");
    frame.setLayout(new BorderLayout());

    JPanel c = new JPanel(new GridLayout(2, 3));
    for (int i = 0; i < 6; i++) {
      c.add(new JButton(String.valueOf(i))); }
    frame.getContentPane().add(c, BorderLayout.CENTER);

    JPanel w = new JPanel();
    w.setLayout(new BoxLayout(w, BoxLayout.Y_AXIS));
    for (int i = 0; i < 5; i++) {
      w.add(Box.createVerticalGlue());
      w.add(new JLabel(String.valueOf(i))); }
    w.add(Box.createVerticalGlue());
    frame.getContentPane().add(w, BorderLayout.WEST);

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 200); frame.setVisible(true);
} }
```

Zusammenfassung GUI-Aufbau

GUI-Aufbau mit Swing:

- GUI-Aufbau aus verschiedenen Komponenten
- Verschachtelte Container für hierarchischen Aufbau
- Layout-Manager zur Anordnung der Elemente

Mehr dazu: Laying Out Components Within a Container

`http://java.sun.com/docs/books/tutorial/uiswing/layout/index.html`

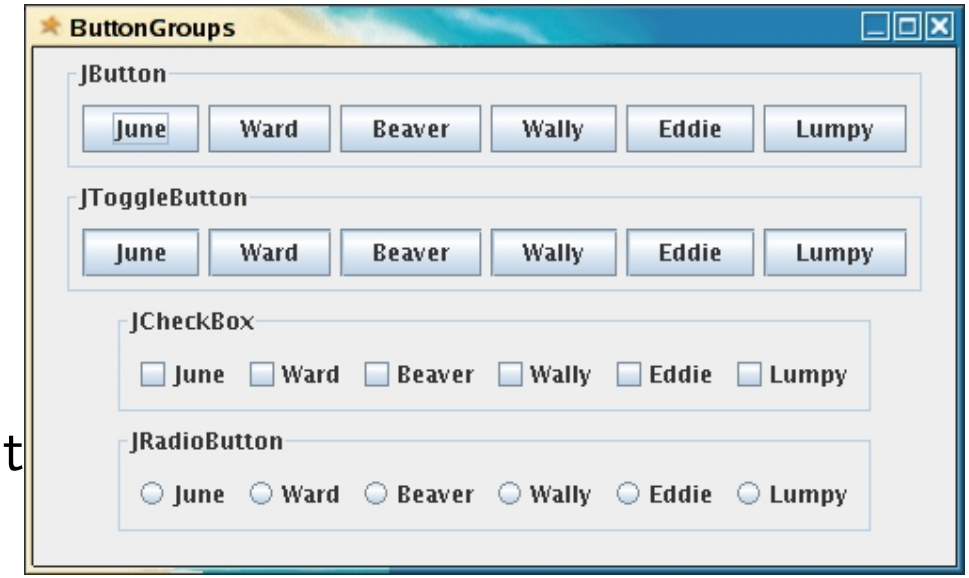
Mehr Schaltflächen

Verschiedene Buttons für verschiedene Anwendungen:

- `JButton`
- `JToggleButton`
- `JCheckBox`
- `JRadioButton`
- Vorgefertige Schaltflächen

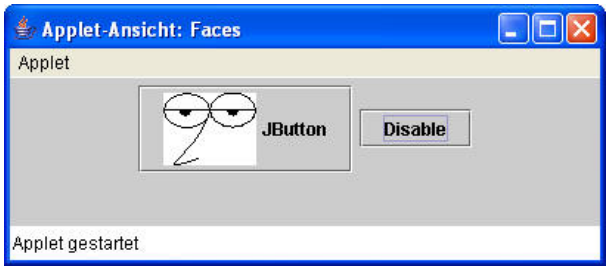
Gruppierung von Schaltflächen mit `ButtonGroup`:

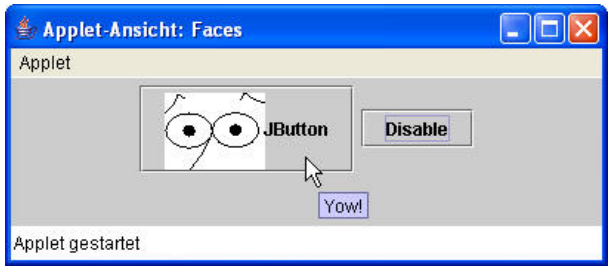
- `JToggleButton`, `JCheckBox`, `JRadioButton`
- Immer maximal einer ausgewählt

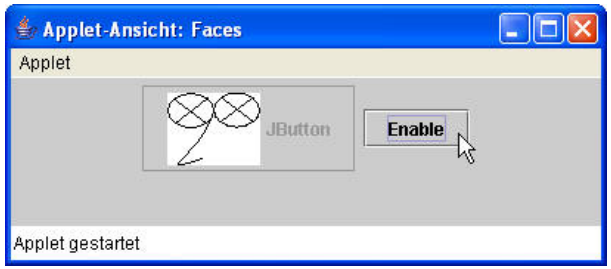


Bilder als `Icon`

Fast alle Widgets lassen sich mit Bildern (`Icon`) hinterlegen. Beispiel aus Thinking in Java:





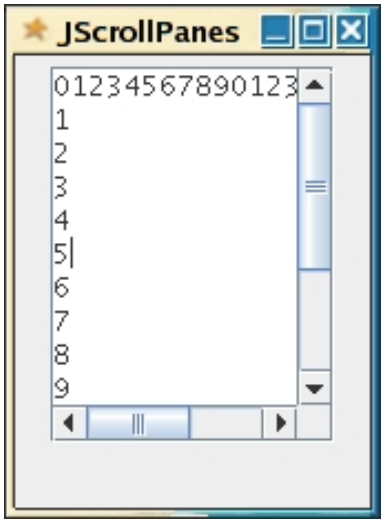


Dieses Beispiel zeigt auch noch einmal schön das Arbeiten mit Events.

Übung: Implementieren Sie das nach!

JScrollPanes

JScrollPanes als Wrapper um ein Widget oder Container:




```
import javax.swing.*; import java.awt.*;
public class JScrollPanes {
  public static void main(String[] args) {
    JFrame frame = new JFrame("JScrollPanes");
    Container cp = frame.getContentPane();
    cp.setLayout(new FlowLayout());

    String content = "01234567890123456789012"
        + "\n1\n2\n3\n4\n5\n6\n7\n8\n9\n0\n1\n2\n3\n4\n5";
    JTextArea t6 = new JTextArea(content, 10, 10);
    cp.add(new JScrollPane(t6, JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                           JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS));

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(170, 230); frame.setVisible(true);
  }
}
```

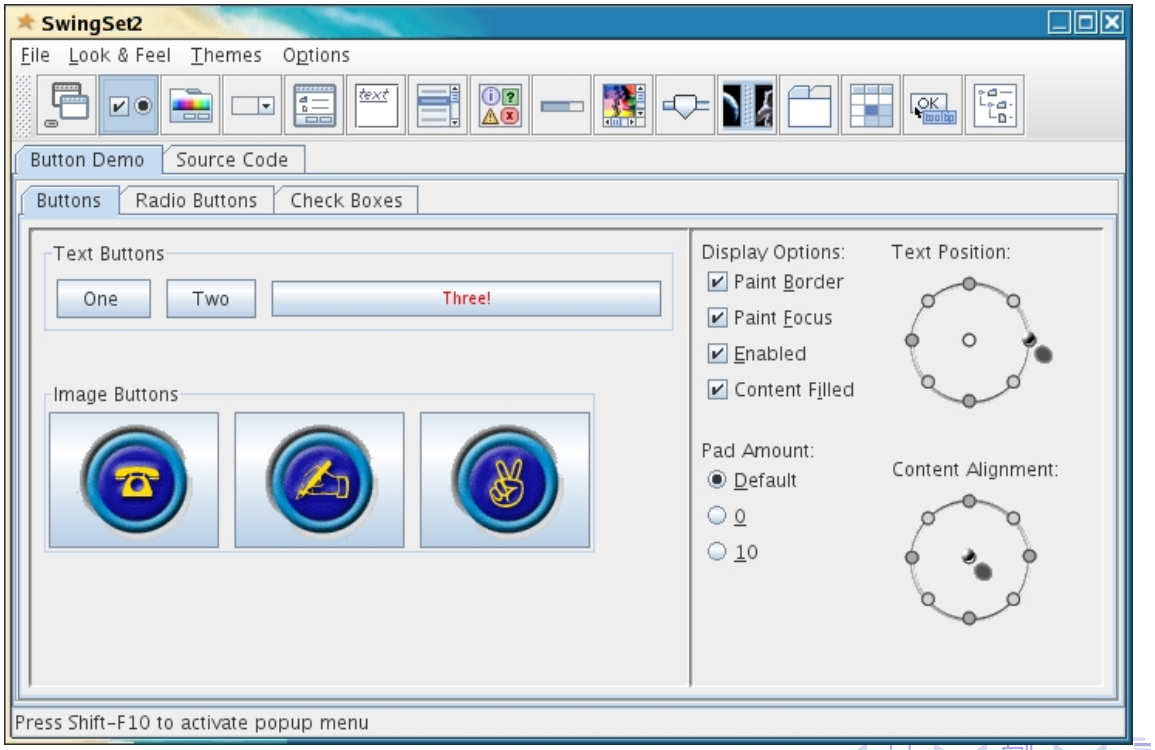
Weitere GUI-Elemente

Oft auch brauchbar:

`JComboBox`	Drop-Down-Box
`JList`	List-Box
`JTextField`	einzeiliges Textfeld
`JTextArea`	mehrzeiliges Textfeld

Java-Demo “SwingSet2” zu Swing:

```
$JAVA_HOME/demo/jfc/SwingSet2
java -jar SwingSet2.jar
```



Komplexere GUI-Elemente von Swing

Swing bietet neben Basis-Bausteinen auch *vorgefertigte* Standard-Komponenten an:

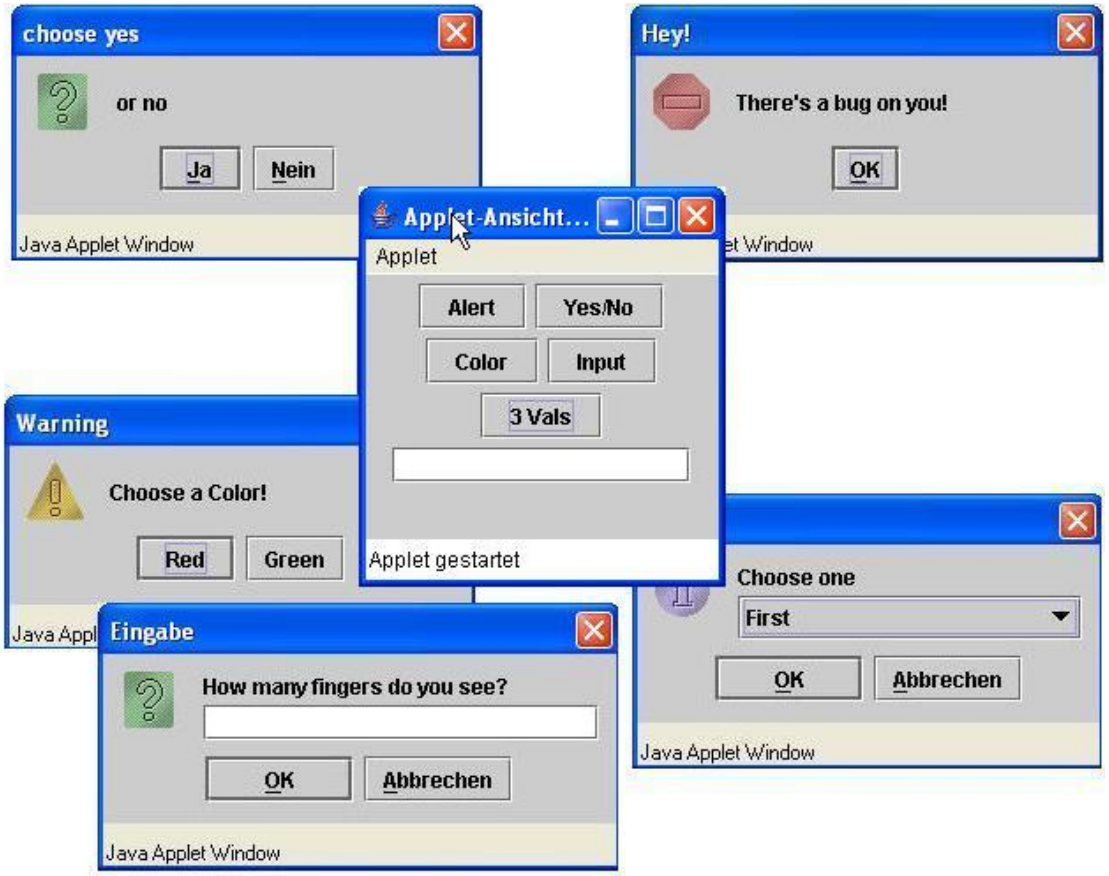
- Message Boxes - Feedback für den Benutzer
- Menüs
- Zeichnen mit Swing

Einfache Popup-Dialoge

Vom Benutzer wird benötigt:

- Einfache Eingaben
- Bestätigungsabfrage
- Auswahl

⇒ Konfigurierbare Standarddialoge aus `JOptionPane`



Message boxes mit JOptionPane

```
public void actionPerformed(ActionEvent e) {
  String id = e.getActionCommand();
  if (id.equals(titles[0])) {
    JOptionPane.showMessageDialog(null, "There's a bug on you!",
                                  "Hey!", JOptionPane.ERROR_MESSAGE);
  } else if (id.equals(titles[1])) {
    JOptionPane.showConfirmDialog(null, "or no", "choose yes",
                                  JOptionPane.YES_NO_OPTION);
  } else if (id.equals(titles[2])) {
    Object[] options = { "Red", "Green" };
    int sel = JOptionPane.showOptionDialog(null, "Choose a Color!",
      "Warning", JOptionPane.DEFAULT_OPTION,
      JOptionPane.WARNING_MESSAGE, null, options, options[0]);
  ...
```

Das Methodenangebot der Utility-Klasse JOptionPane:

showMessageDialog()	reiner Hinweis
showConfirmDialog()	Auswahldialog (ja/nein, links/rechts, ...)
showInputDialog()	Eingabedialog
showOptionDialog()	Frei konfigurierbarer Dialog

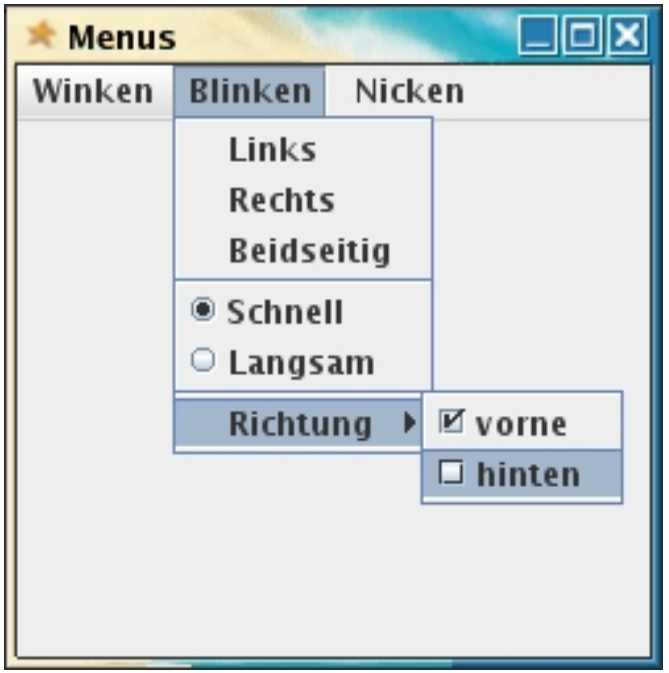
Menüs mit Swing

Menüs sind klassische Kandidaten, um diese als ein *separates GUI-Objekt* zu erzeugen.

Aggregationshierarchie:

- eine `JMenuBar` enthält
- `JMenu`s, diese enthalten
- `JMenuItem`s oder `JMenu`s

Beliebig tief verschachtelbar! `JMenuBar` und `JMenu` verwenden `BoxLayout`



Menü-Beispiel

```
// Menükomponenten erzeugen
JMenuBar menubar = new JMenuBar();
JMenu blMenu = new JMenu("Blinken");
JMenuItem blLinks = new JMenuItem("Links");
JMenuItem blRechts = new JMenuItem("Rechts");
JMenuItem blBeide = new JMenuItem("Beidseitig");
JRadioButtonMenuItem blSchnell = new JRadioButtonMenuItem("Schnell");
JRadioButtonMenuItem blLangsam = new JRadioButtonMenuItem("Langsam");
...
// Verhalten der Komponenten durch ActionListener festlegen
blLinks.addActionListener(...); blRechts.addActionListener(...); ...
...
blMenu.add(blLinks); blMenu.add(blRechts); blMenu.add(blBeide);
blMenu.addSeparator();

// RadioButtons gruppieren
ButtonGroup bg = new ButtonGroup(); bg.add(blSchnell); bg.add(blLangsam);
blSchnell.setSelected(true);
blMenu.add(blSchnell); blMenu.add(blLangsam);
...
menubar.add(blMenu);
getContentPane().add(menubar);
```

Popup-Menüs

Kontext-Menü: JPopupMenu + MouseListener

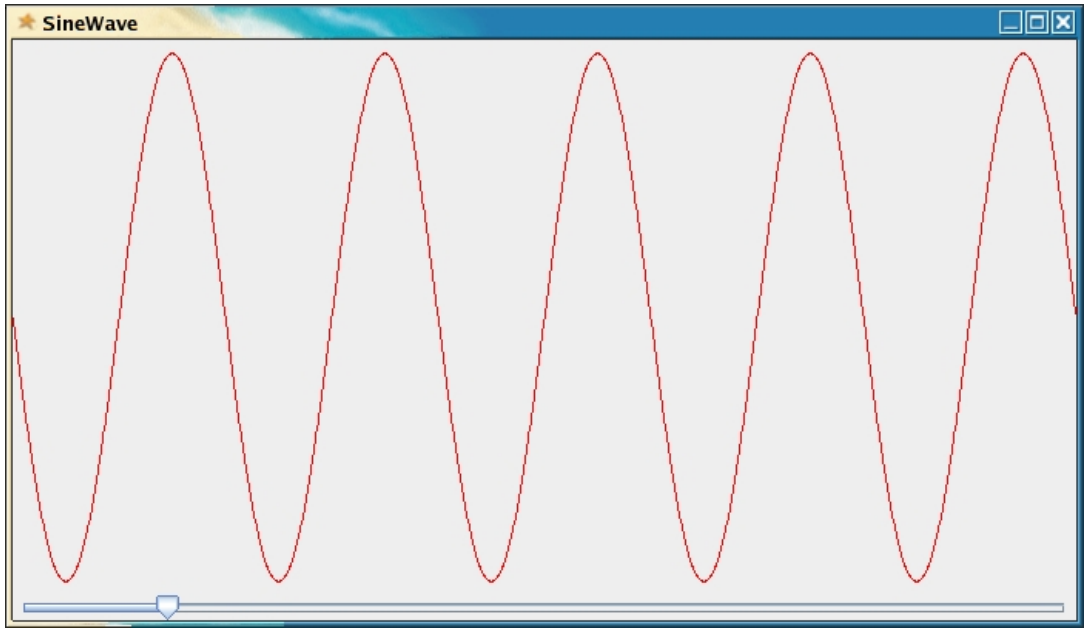

```
JPopupMenu popup = new JPopupMenu();
JMenuItem hither = new JMenuItem("Hither");
JMenuItem yon = new JMenuItem("Yon");
JMenuItem afar = new JMenuItem("Afar");
JMenuItem stayhere = new JMenuItem("Stay here");
...
hither.addActionListener(...); yon.addActionListener(...); ...
popup.add(hither); popup.add(yon); popup.add(afar);
popup.addSeparator(); popup.add(stayhere);

JLabel jb = new JLabel("Click me!", SwingConstants.CENTER);
jb.addMouseListener(new MouseAdapter() {
    @Override
    public void mousePressed(MouseEvent e) {
      if (e.isPopupTrigger())
        popup.show(frame.getContentPane(), e.getX(), e.getY());
    }
  });
frame.getContentPane().add(jb);
```

Zeichnen mit Swing

- Manchmal sind Standard-Widgets nicht genug

$\Rightarrow$ selbst zeichnen

- `JPanel` als Zeichenfläche



1. Überschreiben der `paintComponent(Graphics g)`-Methode
2. `Graphics` bietet Methoden zum Zeichnen an
3. Neuzeichnen mit `repaint()` anfordern

Zeichnen mit Swing

```
class SineDraw extends JPanel {
  private static final int SCALEFACTOR = 200;
  private int cycles; private int points; private double[] sines;
  public SineDraw() { setCycles(5); }
  @Override
  public void paintComponent(Graphics g) {
    super.paintComponent(g);

    int maxWidth = getWidth();  double hstep = (double) maxWidth / (double) points;
    int maxHeight = getHeight();
    int[] pts = new int[points];
    for(int i = 0; i < points; i++)
      pts[i] = (int)(sines[i] * maxHeight / 2 * .95 + maxHeight / 2);

    g.setColor(Color.RED);
    for(int i = 1; i < points; i++) {
      int x1 = (int) ((i - 1) * hstep); int x2 = (int) (i * hstep);
      int y1 = pts[i - 1]; int y2 = pts[i];
      g.drawLine(x1, y1, x2, y2);
    }
  }
  public void setCycles(int newCycles) {
    cycles = newCycles; points = SCALEFACTOR * cycles * 2;
    sines = new double[points];
    for(int i = 0; i < points; i++)
      sines[i] = Math.sin((Math.PI / SCALEFACTOR) * i);
    repaint();
  }
}
```

Übersicht

Trennung von Programmlogik und Darstellung

Grundprinzip des Software Engeneering:
Jede Klasse hat genau ein Geheimnis!

Problem: Verflechtung von Programmlogik und Darstellung

- Algorithmik in der Event-Verarbeitung
- Berechnung von Daten in den graphischen Komponenten

$\Rightarrow$ schwer wartbar

$\Rightarrow$ schwer erweiterbar

$\Rightarrow$ nicht wiederverwendbar

Lösung: Konsequente Trennung

Trennung von Programmlogik und Darstellung

```
class BusinessLogic {
  private int modifier;
  public BusinessLogic(int mod) {
    modifier = mod; }
  public void setModifier(int mod) {
    modifier = mod; }
  public int getModifier() { return modifier; }
  // Some business operations:
  public int calculation1(int arg){
    return arg * modifier;}
  public int calculation2(int arg){
    return arg + modifier;}
}

public class Separation extends JFrame {
  private JTextField t = new JTextField(15);
  private JTextField mod = new JTextField(15);
  private JButton calc1 = new JButton("Calc 1"),
  private JButton calc2 = new JButton("Calc 2");
  private BusinessLogic bl = new BusinessLogic(2);

  public static int getValue(JTextField tf) {
    try {
       return Integer.parseInt(tf.getText());
    } catch(NumberFormatException e) {
      return 0;
    }
  }
```

```
class Calc1L implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // Delegation an BusinessLogic
    int i = bl.calculation1(getValue(t));
    t.setText(Integer.toString(i));
  }
}
...
public Separation() {
  Container cp = getContentPane();
  cp.setLayout(new FlowLayout());
  cp.add(t);
  calc1.addActionListener(new Calc1L());
  calc2.addActionListener(new Calc2L());
  JPanel p1 = new JPanel();
  p1.add(calc1); p1.add(calc2);
  cp.add(p1);
  JPanel p2 = new JPanel();
  p2.add(new JLabel("Modifier:"));
  p2.add(mod); cp.add(p2);
}
...
}
```

Trennung von Programmlogik und Darstellung

Geschäftslogik Das zugrundeliegende Modell für Berechnungen
Variable `modifier` der Klasse `BusinessLogic`
und die `calculate`-Methoden

Darstellung Die Anzeige des Modells
`modifier` dargestellt in `JTextField mod`

Ereignisverarbeitung Verknüpft GUI und Model
$\Rightarrow$ `Listener`

Vorteil dieser Architektur:

Abhängigkeit Geschäftslogik unabhängig von der GUI

- Wiederverwendbar
- Automatisch testbar

Delegation GUI delegiert Berechnungen an die Geschäftslogik

Änderungen am einen Ende unabhängig vom anderen Ende möglich!

Swing & Nebenläufigkeit

Achtung: Swing Applikationen sind immer nebenläufig!

Verschiedene Anzeigen:

- `Initial Value`
- `Initialization complete`
- `Application ready`
- `Done`

Preisfrage:
In welcher Reihenfolge?

```
public class EventThreadFrame extends JFrame {
  private JTextField statusField =
    new JTextField("Initial Value");

  public EventThreadFrame() {
    getContentPane().add(statusField, BorderLayout.NORTH);

    this.addWindowListener(new WindowAdapter() {
        public void windowOpened(WindowEvent e) {
          try { // Simulate initialization overhead
            Thread.sleep(2000);
          } catch (InterruptedException ex) { }
          statusField.setText("Initialization complete");
        }
      });
  }

  public static void main (String[] args) {
    EventThreadFrame etf = new EventThreadFrame();
    etf.setSize(150, 60);
    etf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    etf.setVisible(true);
    etf.statusField.setText("Application ready");
    System.out.println("Done");
  }
}
```

Kontrollieren der Nebenläufigkeit

Des Rätsels Lösung:

1. `Done` wird als erstes auf der Konsole ausgegeben
2. `Application ready` wird sofort überschrieben, ist *gar nicht sichtbar*
3. `statusField` zeigt `Initial Value`
4. `statusField` wechselt nach 2s zu `Initialization complete`

Zwei Threads:

main	führt `main`-Methode aus
EventDispatcher	verarbeitet Events und zeichnet die GUI
$\Rightarrow$	Nebenläufiger Zugriff auf `etf.statusField`

Unsauberkeiten:

- Keine Synchronisation vorhanden
- Nur EventDispatcherThread sollte GUI ändern

invokeLater und invokeAndWait

Einfügen von Aufrufen in die EventQueue mit `SwingUtilities.invokeLater()` und `SwingUtilities.invokeAndWait()`

- `Runnable-Objekt` als Parameter
- `run`-Methode wird später ausgeführt
- `invokeAndWait()` legt aktuellen Thread solange schlafen

```
public static void main(String[] args) {
  final EventThreadFrame elf = new EventThreadFrame();
  etf.setSize(150, 60);
  etf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  etf.setVisible(true);
  SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        elf.statusField.setText("Application ready");
      }
    });
  System.out.println("Done");
}
```

GUI-Lock Ups: Aufwändige Berechnung im EventThread

```
public class GUILock1 extends JFrame {
  public GUILock1() {
    Container c = getContentPane(); c.setLayout(new FlowLayout());
    JButton b = new JButton("Lock Up!"); c.add(b);
    final JTextField tf = new JTextField(5); c.add(tf);
    b.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          int cnt = 0; while (true) { tf.setText("" + cnt++); }
    } });
  } ... }
```

Grundproblem: Aufwändige Aktionen im EventThread $\Rightarrow$ träge GUI
Hier:

1. Die GUI wird normal angezeigt
2. Sobald der Button angeklickt wird, friert die GUI ein
3. `actionPerformed` hindert EventDispatcherThread am Zeichnen

Lösung: Eigene Threads für aufwändige Operationen, nur reine GUI-Updates (z.B. `repaint`) im Event-Thread

GUI Lock Ups: Pushen von Daten

```
class GUILock2 {
  public static void main(String[] args) {
    JFrame f = new JFrame(); final JTextField txt = new JTextField("0");
    f.getContentPane().add(txt); ...
    while (true) {
      SwingUtilities.invokeLater(new Runnable() {
          public void run() {
            // expensive update
            txt.setText("" + (Integer.parseInt(txt.getText()) + 1));
            try { Thread.sleep(1000); } catch (InterruptedException e) {}
    }});
}}}
```

Szenario: Der `main`-Thread produziert kontinuierlich Daten, die in der GUI angezeigt werden sollen.

Idee: Benachrichtigung der GUI bei neuen Werten

`invokeAndWait` Bremst den Rechenkern

`invokeLater` Produktion ist evtl. zu schnell

⇒ Überlauf der EventQueue; Absturz mit `OutOfMemoryError`

Pull von Daten mittels Timer

```
class TimedGUI {
  public static final int DELAY = 50;
  private static int counter = 0;
  public static void main(String[] args) {
    JFrame frame = new JFrame(); final JTextField txt = new JTextField("0");
    frame.getContentPane().add(txt);
    ...
    Timer timer = new Timer(DELAY, new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          txt.setText("" + counter); txt.repaint();
        }
    });
    timer.start();
    while (true) { frame.counter++; }
  }
}
```

Stabilere Lösung: Pull Modell. GUI stellt sich regelmäßig selbst neu dar:

- realisierbar mit `Timer`-Objekt
- dessen Konstruktor bekommt Zeitintervall und ActionListener, der ausgeführt werden soll
- `actionPerformed` implementiert das GUI-Update
- **Achtung:** i.A. werden nur einzelne Komponenten neu gezeichnet, nicht der ganze Frame

Summary

Das war eine (sehr) knappe Einführung in Java Swing. Details:

- Java Tutorial, Swing Trail:
  `http://java.sun.com/docs/books/tutorial/uiswing/in`
- Die `SwingSet2` Demo Applikation (Teil des JDK)
- Swing und Threads:
  http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html
- Es gibt gute und dicke Bücher zu Swing alleine!

Noch eine Warnung zum Schluss

Die Autoren des SubArctic Java ToolKit:

> *It is our basic belief that extreme caution is warranted when designing and building multi-threaded applications, particularly those which have a GUI component. Use of threads can be very deceptive. In many cases they appear to greatly simplify programming by allowing design in terms of simple autonomous entities focused on a single task. In fact in some cases they do simplify design and coding. However, in almost all cases they also make debugging, testing, and maintenance vastly more difficult and sometimes impossible. Neither the training, experience, or actual practices of most programmers, nor the tools we have to help us, are designed to cope with the non-determinism. For example, thorough testing (which is always difficult) becomes nearly impossible when bugs are timing dependent. This is particularly true in Java where one program can run on many different types of machines and OS platforms, and where each program must work under both preemptive or non-preemptive scheduling.*
>
> *As a result of these inherent difficulties,* **we urge you to think twice about using threads in cases where they are not absolutely necessary...**