Programmiersprache

Java

2022 / Teil 8

Detlef Wilkening
www.wilkening-online.de
© 2022

Programmiersprache Java

16 GUI Programmierung....3
16.1 GUI Programmierung in Swing 3
16.2 Ein einfaches Fenster 4
16.3 Ein etwas besseres Fenster....5
16.4 Ein grafisches „Hallo Welt“....6
16.5 Graphics Objekt....8
16.6 Klasse „Color“ 9
16.7 Änderung der Oberfläche in den Windows-Style....10

17 Philosophie der GUI Programmierung 12
17.1 Besitzer des Bildschirms....12
17.2 Kontrollfluss....13

18 Event-Modelle von Swing....14
18.1 Events und Gruppierungen....14
18.2 Internes Event-Modell....15
18.3 Externes Event-Modell....20
18.4 Vergleich der Event-Modelle....32
18.5 Scribble 4 mit Daten-Modell....32

19 Swing Layouts 34
19.1 Swing Klasse „JButton“....35
19.2 Grundlagen....35
19.3 Layouts....36
19.4 Verschachtelte Layouts 39

20 Swing GUI-Elemente....40
20.1 Labels....40
20.2 Text-Felder 41
20.3 Buttons 42
20.4 Radio-Buttons....43
20.5 Check-Boxen....45
20.6 Scroll-Bars und Scroll-Panes 46
20.7 Tabellen....47
20.8 Panels 51
20.9 Menüs....51
20.10 Timer 53

Hinweis: Diese Kapitel sind nicht mehr prüfungs-relevant. Daher diese Themen kommen nicht in der Prüfung vor. Trotzdem empfehle ich Ihnen, auch diese Kapitel durchzuarbeiten. Zum einen macht GUI Programmierung den meisten Studenten sehr viel Spaß, da man sieht, was man programmiert hat. Und zum anderen werden in der GUI Programmierung alle

Sprachelemente benutzt, die Sie in den bisherigen Kapiteln gelernt haben. Sie ist also eine wunderschöne Übung in Vorbereitung auf die Prüfung oder spätere Java Einsätze.

16 GUI Programmierung

Bei vielen Programmen wird heutzutage eine grafische Bedienoberfläche mit Fenstern, Menüs, Maus, uvm. erwartet. Für die Programmierung grafischer Oberflächen gibt es für Java mehrere Bibliotheken, unter anderem die folgenden vier:

- **AWT** (Abstract Window Toolkit) ist seit Java 1.0 Bestandteil von Java, und immer noch in Java enthalten. AWT ist sehr einfach gehalten, war dafür aber sehr kompakt und performant. Trotzdem ist AWT schon lange in die Jahre gekommen, und eigentlich nicht mehr aktuell.
- **Swing** ist seit Java 2 (JDK 1.2) Bestandteil von Java und baut im Kern auf AWT auf. Swing wurde entwickelt, um die Grenzen und Beschränkungen von AWT zu beseitigen, und in Java eine zeitgemäße GUI Bibliothek zu integrieren. Trotzdem Swing zum Teil mittlerweile auch in die Jahre gekommen ist, ist Swing wahrscheinlich immer noch die verbreitetste und am meisten benutzte GUI Bibliothek in Java.
- **Java FX** wurde in Java 8 in Java aufgenommen, und mit Java 11 wieder aus Java entfernt. Seitdem wird Java FX eigenständig und unabhängig weiterentwickelt, und kann als externe GUI Bibliothek genutzt werden. Die Motivation zur Entwicklung von Java FX waren u.a. die Beschränkungen, die AWT und Swing im Bereich Medien und Animation haben, was bei vielen modernen Oberflächen wichtig ist. Außerdem bringt Java FX eine deklarative GUI Beschreibungssprache mit (FXML), und es können Web-Technologien wie CSS für die Gestaltung eingesetzt werden. Der Erfolg von Java FX ist bislang sehr überschaubar, da im Desktop Umfeld Swing sehr etabliert ist, auf mobilen Devices die nativen GUI Frameworks von iOS oder Android die erste Wahl sind, und für die Web-Entwicklung meist ganz andere Technologien (wie z.B. JavaScript) verwendet werden. Außerdem ist Java FX heute nicht mehr Bestandteil von Java.
- **SWT/JFace** soll hier auch noch erwähnt werden. Dies ist eine GUI Bibliothek, die in vielen Dingen mit Swing vergleichbar ist, und im Eclipse Projekt entstanden ist – also nicht Teil von Java ist. Durch den Erfolg von Eclipse als Rich-Client-Plattform ist die Bibliothek SWT/JFace auch noch recht verbreitet in der Java Welt.

Direkter Bestandteil von Java sind heute aber nur die Bibliotheken AWT und Swing – und so schauen wir uns hier Swing als den moderneren der Beiden an.

16.1 GUI Programmierung in Swing

Swing kapselt die Unterschiede zwischen den einzelnen Betriebssystemen, so dass eine mit Java erstellt Anwendung auf allen Plattformen läuft, auf denen eine virtuelle Maschine JVM vorhanden ist. Dies macht Swing u.a. dadurch, dass es das komplette GUI selber zeichnet und sich nicht auf die nativen Widgets des jeweiligen Betriebssystems abstützt. Dieses Vorgehen hat mehrere Konsequenzen:

- Vorteile:
  - Alle Swing Elemente stehen immer auf jeder JVM zur Verfügung, auch wenn das

zugrunde liegende OS gar keine entsprechenden Widgets kennt.
  - Das Look&Feel von Swing Anwendungen kann jederzeit ausgetauscht werden, und dies sogar zur Laufzeit. So ist z.B. auf einem Windows Rechner ein Motif Look&Feel möglich, oder umgekehrt.
- Nachteile:
  - Bestimmte betriebssystem-spezifische Features sind nicht in Swing vorhanden, da sie nur bedingt auf andere Plattformen abbildbar sind - z.B. neue Windows Elemente. Aufgrund der vollständigen Kapselung der unter der JVM liegenden Plattform (OS und Prozessor) besteht außer via JNI auch keine Möglichkeit solche Features anzusprechen.
  - Swing GUIs sind etwas langsamer und speicherfressender als native Widgets – dies ist aber heutzutage meistens nicht relevant.

Die Klassen-Bibliotheken für die grafische Oberfläche haben sich in der Geschichte von Java mehrmals geändert. Beim Umstieg vom JDK 1.0 zum JDK 1.1 wurde u.a. das Event-Modell komplett umgekrempelt. Mit dem JDK 1.2 wurden die alten AWT (Abstract Window Toolkit) Klassen um die neueren viel leistungsfähigeren JFC (Java Foundation Classes) Klassen erweitert, die ein Bestandteil von Swing sind und die alten AWT Klassen als Basis-Klassen beinhalten.

Dieses Kapitel stellt nur eine erste Einführung in die Programmierung von grafischen Oberflächen dar, indem es das berühmte „Hallo Welt“ implementiert. In den folgenden Kapiteln werden einzelne Bereiche der Programmierung von grafischen Oberflächen weiter vertieft.

Achtung – die Kapitel über GUIs mit Swing sind natürlich nur eine kleine Einführung. Das Thema GUI Entwicklung ist viel zu umfangreich, als das es hier umfassend behandelt werden könnte – allein mit Swing kann man ganze Bücher füllen. Diese Einführung versucht die wichtigsten Konzepte von GUI Programmierung und Swing vorzustellen

16.2 Ein einfaches Fenster

Wenn es nur darum geht, ein Fenster auf den Bildschirm zu zaubern, hält sich der Aufwand in Grenzen – wir haben das Programm schon früher kennen gelernt.

```
import javax.swing.JFrame;

public class Appl {

  public static void main(String[] args) {
    JFrame frame = new JFrame("Mein GUI Fenster 1");
    frame.setLocation(20, 20);
    frame.setSize(600, 400);
    frame.setVisible(true);
  }

}
```

In „main“ wird ein Objekt der Swing-Frame Klasse „JFrame“ erzeugt, die für ein einfaches Fenster darstellt – dem Konstruktor wird der Fenstertitel übergeben. Mit ‘setLocation’ wird der Startort, mit ‘setSize’ die Grösse, und mit ‘setVisible’ die Sichtbarkeit gesetzt. Fertig.

Aber diese Minimal-Lösung hat mehrere Nachteile:

- Mit Schliessen des Fensters wird nicht mal das Programm beendet – Sie können es nur auf der Kommandozeile mit [Strg]+[C] abschiessen.
- Das JFrame-Klasse liefert nur ein allgemeines Default-Verhalten, das eigentlich nie reicht. In dem Fenster soll ja schließlich was passieren.

Um mit dem Schliessen des Fensters das Programm zu beenden, muss für das Fenster ein entsprechendes Flag gesetzt werden – dies geschieht mit „setDefaultCloseOperation" und der Konstanten „JFrame.EXIT_ON_CLOSE".

```
import javax.swing.JFrame;

public class Appl {

  public static void main(String[] args) {
    JFrame frame = new JFrame("Mein GUI Fenster 2");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLocation(20, 20);
    frame.setSize(600, 400);
    frame.setVisible(true);
  }

}
```

In einem späteren Kapitel über Event-Handling werden wir das Beenden des Programms beim Schließen des Fensters anders lösen, indem wir uns selber in die Event-Bearbeitung einhängen. Aber das wird nur ein Beispiel zum Verständnis des Event-Handlings in Swing sein – der normale Weg sollte der hier beschriebene mit dem Setzen der Default-Close-Operation sein.

16.3 Ein etwas besseres Fenster

Besser wäre also ein eigenes Fenster, das wir nach unseren Vorstellungen formen (programmieren) können - wir würden das Default-Verhalten von JFrame aber gerne beibehalten. Was macht man dann in Java? Natürlich erben!

Hier ein zweites Programm, das zwar nicht mehr kann als das vorherige, aber schon mal die Vorbereitungen für mehr darstellt.

```
public class Appl {

  public static void main(String[] args) {
    MyFrame frame = new MyFrame("Mein eigenes Fenster");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLocation(20, 20);
    frame.setSize(600, 400);
    frame.setVisible(true);
  }

}
```

```
import javax.swing.JFrame;

public class MyFrame extends JFrame {
```

```
  public MyFrame(String title) {
    super(title);
  }

}
```

Im Prinzip ist es das alte Programm – nur wird statt der Swing Klasse „JFrame“ eine eigene Frame-Klasse „MyFrame“ benutzt, die sich von „JFrame“ ableitet. Da „MyFrame“ keine neue Funktionalität hinzufügt, verhält sich das neue Programm wie das alte.

16.4 Ein grafisches „Hallo Welt“

16.4.1 Das normale grafische „Hallo Welt“

Um ein grafisches „Hallo Welt“ zu erhalten, müssen wir im Fenster auch „Hallo Welt“ ausgeben. Wie macht man das?

- Immer, wenn ein Fenster neu gezeichnet werden muss, wird automatisch die Funktion „paint“ aufgerufen, die daher für eigene Ausgaben überschrieben werden muss – siehe weiter unten und spätere Kapitel.
- Die Funktion „paint“ bekommt ein „Graphics“ Objekt übergeben, das u.a. Funktionen für die Textausgabe in einem Fenster zur Verfügung stellt – siehe Kapitel 16.5.

```
import java.awt.Graphics;
import javax.swing.JFrame;

public class MyFrame extends JFrame {

  public MyFrame(String title) {
    super(title);
  }

  public void paint(Graphics g) {
    super.paint(g);
    g.drawString("Hallo Welt", 50, 50);
  }

}
```

Für die Ausgabe überschreiben wir die Funktion „paint“, die an den Koordinaten „50/50“ den Text „Hallo Welt“ ausgibt - mehr dazu in Kapitel 17 und Kapitel 16.5.

Abb. 16-1 : Ein grafisches „Hallo Welt“ Programm

Achtung - es ist ganz wichtig - nicht nur für die Funktion „paint“ sondern auch für viele weitere Funktionen, die wir zukünftig kennenlernen werden - dass zuerst die überschriebene Funktion der Basis-Klasse (hier „paint“) aufgerufen wird. Das Neu-Zeichnen unseres Fensters besteht ja nicht nur aus der Ausgabe von „Hallo Welt“, sondern noch aus anderen Dingen - und die geschehen natürlich in den Basisklassen, und sollten nicht verhindert werden. Und man sollte zuerst die Super-Funktion aufrufen, da sie ansonsten möglicherweise alle unsere Aktionen wieder zerstört.

Hinweis – warum funktioniert das? Hier ist natürlich wieder Vererbung und Polymorphie im Spiel. Der interne Fenster-Verwaltungs-Mechanismus der JVM kennt unsere Fenster Klasse „MyFrame“ natürlich nicht, aber er arbeitet auf einer Basisklasse von „MyFrame“. In dieser ist die Funktion „paint“ definiert, und die wird intern aufgerufen. Wir haben diese Funktion aber nun überschrieben, und dank Polymorphie landet damit der Aufruf von „paint“ in „MyFrame.paint“.

16.4.2 Probleme mit dem JDK 1.5 und später

Das im vorherigen Kapitel 16.4.1 beschriebene Java-Programm funktioniert mit dem JDK 1.5 und neueren Versionen nicht fehlerfrei - zumindest in den meisten von mir getesteten JDK Versionen. Mit allen möglichen alten JDKs, z.B. 1.2, 1.3, 1.4, 1.4.1, 1.4.2 und auch mit einigen neueren JDK Versionen funktioniert das Programm korrekt.

Was ist denn hier nun das Problem? Ganz einfach – immer wenn das Fenster in irgendeiner Form vergrößert wird (d.h. es wird mit der Maus oder der Tastatur größer gezogen, oder es wird maximiert), dann wird der Inhalt des Fensters nicht neu gezeichnet, d.h. die Paint-Funktion wird nicht aufgerufen.

In der realen Praxis ist dies kein Problem, da dort eigentlich nie direkt das Haupt-Frame für Paint-Aktionen genutzt wird, sondern im Haupt-Frame eigentlich immer weitere GUI-Elemente (z.B. Menüs, Buttons, Panels, usw.) liegen. Und bei denen klappt das alles auch z.B. im JDK 1.5 problemlos.

Problematisch ist der Bug aber für dieses Tutorial und auch andere Lehrbücher. Denn hier sollen die Beispiele natürlich möglichst einfach sein, weshalb im Frame oft keine weiteren Elemente liegen – und damit tritt der Fehler auf.

In der Praxis sollten Sie sich dieses Problems bewusst sein – und die Musterlösungen aus dem Tutorial entsprechend verändern. Eine mögliche Lösung z.B. ist das Frame mit einem eigenen Panel (siehe Kapitel 20.8) zu füllen, und alle Aktionen statt im Frame im Panel unterzubringen. Als Beispiel hier das „Hallo-Welt“ Programm in einer Version mit zusätzlichem Panel, die u.a. auch mit dem JDK 1.5 funktioniert.

```
import javax.swing.JFrame;

public class Appl {

  public static void main(String[] args) {
    MyFrame frame = new MyFrame("Hallo-Welt Fenster fuer das JDK 1.5");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    frame.setLocation(20, 20);
    frame.setSize(600, 400);
    frame.setVisible(true);
  }

}
```

```
import javax.swing.JFrame;

public class MyFrame extends JFrame {

  public MyFrame(String title) {
    super(title);
    getContentPane().add(new MyPanel());
  }

}
```

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class MyPanel extends JPanel {

  public void paint(Graphics g) {
    super.paint(g);
    g.drawString("Hallo Welt", 50, 50);
  }

}
```

16.5 Graphics Objekt

Die Klasse „java.awt.Graphics“ stellt die Schnittstelle eines Swing Programms für die Ausgabe auf dem Bildschirm dar. Die Klasse bietet verschiedenste Element-Funktionen an um Pixel, Linien, Kreise, Rechtecke, Texte, uvm. auf dem Bildschirm auszugeben.

Lassen sie sich von Eclipse einfach mal die Menge an Funktionen anzeigen, bzw. schauen sie in die Java-Hilfe. Bei vielen Funktionen sagt der Name intuitiv was die Funktion macht, und auch die Parameter benötigen kaum Erklärung, von daher sollten einfache Anwendungen kein Problem sein.

Beispiele von Element-Funktionen in „java.awt.Graphics“

Funktion	**Beschreibung**
void drawLine(int x1, int y1, int x2, int y2)	Zeichnet eine Linie
void drawRect(int x, int y, int width, int height)	Zeichnet ein Rechteck
void fillRect(int x, int y, int width, int height)	Zeichnet ein ausgefülltes Rechteck
void drawOval(int x, int y, int width, int height)	Zeichnet eine Elipse
void fillOval(int x, int y, int width, int height)	Zeichnet eine ausgefüllte Elipse
void setColor(Color c)	Setzt die Zeichen-Farbe
void setPaintMode()	Setzt den Zeichen-Mode auf Überschreiben
void setXORMode(Color c1)	Setzt den Zeichen-Mode auf XOR zur übergebenen Farbe

```
import java.awt.Color;
```

```
import java.awt.Graphics;
import javax.swing.JFrame;

public class MyFrame extends JFrame {

  public MyFrame(String title) {
    super(title);
    setSize(380, 340);
  }

  public void paint(Graphics g) {
    super.paint(g);
    g.drawString("Hallo Welt", 20, 60);
    g.fillOval(100, 100, 80, 50);
    g.setColor(Color.RED);
    g.drawRoundRect(40, 200, 300, 40, 20, 20);
    g.setColor(Color.BLUE);
    g.drawLine(50, 150, 300, 300);
  }

}
```

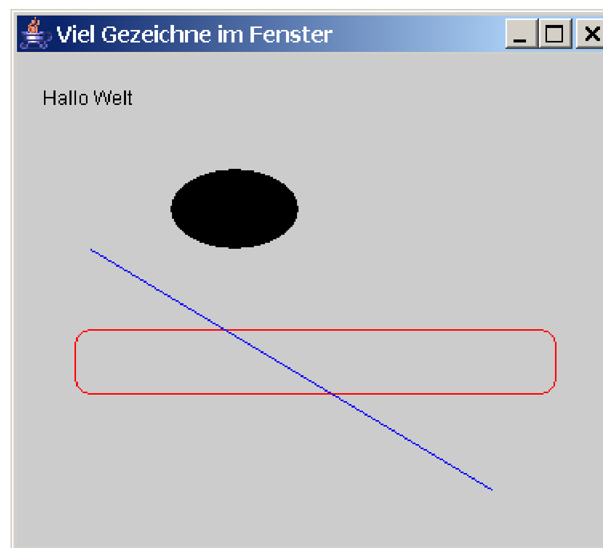

Abb. 16-2 : Ein GUI Programm mit bunter Ausgabe im Fenster

Hinweis - während des normalen Programm-Ablaufs, z.B. bei einer Benutzer-Interaktion oder während einer Berechnung, kann es ohne weiteres notwendig sein direkt auf den Bildschirm zu zeichnen. Auch in diesen Fällen benötigt man ein Graphics Objekt - man kann es sich für jedes Swing Objekt mit der Funktion „getGraphics()“ holen. Eine Anwendung dafür findet sich z.B. im Scribble Programm in Kapitel 18.2.4.

Achtung – das Überschreiben der Paint-Funktion des Haupt-Fensters führt mit dem JDK 1.5 zu Problemen – siehe Kapitel 16.4.2.

16.6 Klasse „Color“

Im letzten Beispiel wurde u.a. die Zeichen-Farbe für das Graphics Objekt auf „rot“ bzw. „blau“ gesetzt - siehe Kapitel 16.5. Für Farben gibt es die Klasse „java.awt.Color“. Sie enthält u.a.

Konstanten für die wichtigsten Farben wie „schwarz – BLACK“, „weiß – WHITE“, „rot – RED“, „blau – BLUE“, „grün – GREEN“, usw.

Objekte der Klasse „Color“ können aber z.B. auch durch die gewünschten RGB Werte (rot, grün, blau) erzeugt werden – die Klasse enthält hierfür verschiedene Konstruktoren.

```
Color c = new Color(128, 0, 128);
```

Die Graphics Klasse enthält die Funktion „setColor“ zum Setzen der Zeichen-Farbe.

Alle Swing Klassen haben die Funktionen „setForeground“ und „setBackgroundColor“, um die Vorder- und Hintergrund-Farbe zu setzen – die Vordergrund-Farbe entspricht der Zeichenfarbe.

16.7 Änderung der Oberfläche in den Windows-Style

Alle GUI Programme in diesem Tutorial werden mit dem normalen Swing-Style erzeugt, d.h. der Oberflächen-Skin ist Swing. Nehmen wir z.B. das GUI-Fenster aus Kapitel 19.4, das obwohl der Screenshot unter Windows XP mit klassischem Windows Skin erzeugt wurde – nicht wirklich nach Windows aussieht:

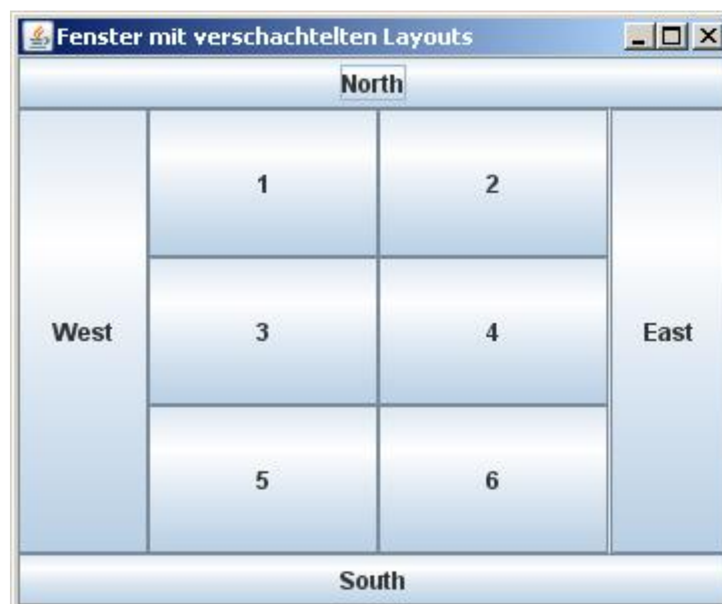

Abb. 16-3 : GUI-Fenster aus Kapitel 19.4 mit Swing-Style

Möglicherweise möchten Sie aber, dass Ihre Programme unter Windows auch einen Windows-Style haben. Das können Sie in Swing erreichen, indem Sie in Ihrem Programm den Style entsprechend setzen. Hier der Quelltext aus Kapitel 19.4 mit verändertem Windows-Style:

```
import javax.swing.JFrame;
import javax.swing.UIManager;

public class Appl {
```

```
   public static void main(String[] args) {
      try {
         UIManager.setLookAndFeel(
            "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
      } catch (Exception e) {
         // Klappt es wohl nicht - auch egal - dann eben mit Swing-Style...
      }

      MyFrame frame = new MyFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setLocation(20, 20);
      frame.setSize(600, 400);
      frame.setVisible(true);
   }

}
```

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.GridLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

@SuppressWarnings("serial")
public class MyFrame extends JFrame {

   public MyFrame() {
      super("Fenster mit verschachtelten Layouts");
      setSize(400, 300);

      JPanel panel = new JPanel(new GridLayout(3, 2));
      panel.add(new JButton("1"));
      panel.add(new JButton("2"));
      panel.add(new JButton("3"));
      panel.add(new JButton("4"));
      panel.add(new JButton("5"));
      panel.add(new JButton("6"));

      Container c = getContentPane();
      c.setLayout(new BorderLayout());
      c.add(new JButton("North"), BorderLayout.NORTH);
      c.add(new JButton("West"), BorderLayout.WEST);
      c.add(new JButton("East"), BorderLayout.EAST);
      c.add(new JButton("South"), BorderLayout.SOUTH);
      c.add(panel, BorderLayout.CENTER);
   }

}
```

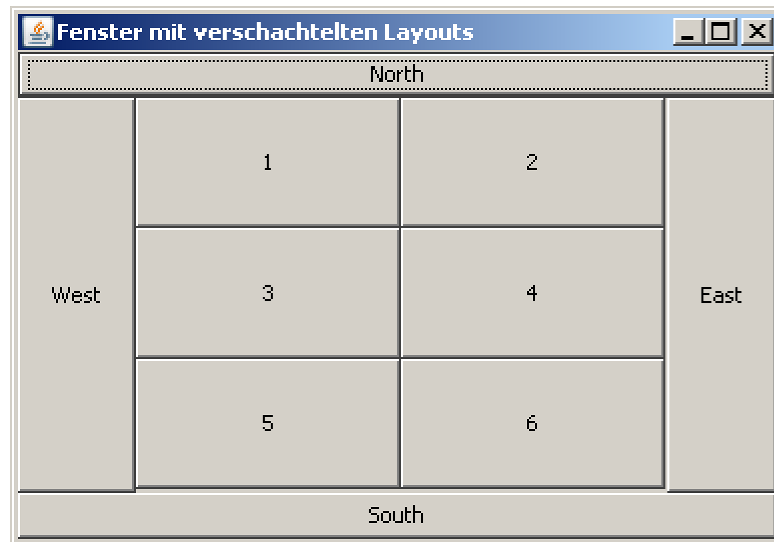

Abb. 16-4 : GUI-Fenster aus Kapitel 19.4 mit Windows-Style

17 Philosophie der GUI Programmierung

Bevor wir weiter gehen, müssen einige Dinge über die Philosophie von GUI Programmierung gesagt werden. Diese Dinge sind ganz unabhängig von Swing oder Java, sondern gelten wohl für alle GUI Bibliotheken.

- Der Bildschirm gehört nicht dem Programm oder einem Fenster, sondern dem Betriebssystem.
- Der Kontrollfluss wird vom GUI vorgegeben und ist stark interaktiv.

17.1 Besitzer des Bildschirms

Im Gegensatz zu alten DOS oder CPM Zeiten unterstützen heute alle modernen Betriebssysteme die Möglichkeit mehrere Programme gleichzeitig auszuführen. Wenn dann diese Programme noch GUI Anwendung sind, wie die meisten Anwendungen heute, hat das einige Konsequenzen für den Zugriff auf den Bildschirm:

- Ein Programm kann nicht einfach beliebig auf den Bildschirm schreiben, da es nicht selbstverständlich ist, dass der Bildschirm ihm gehört, und nicht ein anderes Programm dort seine Ausgaben macht.
- Ein Programm kann seinen Ausgabe-Zustand nicht auf dem Bildschirm abfragen, da mittlerweile ein anderes Programm dort seine Ausgaben gemacht haben kann.
- Ein Programm muss immer in der Lage sein, seine Darstellung zu erneuern, da der Benutzer die Fenster beliebig nach vorne, hinten, seitwärts und sonst was verschieben kann, d.h. ein bislang nicht sichtbarer Teil des Fensters oben liegt und neu gezeichnet werden muss - siehe z.B. Kapitel 18.2.4.

Die letzten beiden Punkte bedingen, dass ein Programm immer ein komplettes *Modell* seiner

Ausgabe enthalten muss. Damit kann ein Programm dann jederzeit seine Darstellung auf dem Bildschirm erneuern, wenn es dazu aufgefordert wird. Dies geschieht via OS, JVM und Paint-Funktion.

17.2 Kontrollfluss

Klassische Programme wurden oft nach dem EVA Prinzip aufgebaut - hierbei ist der Programmfluss relativ einfach - ein Schritt folgt nach dem anderen.

Auch komplexere *klassische* Programme haben eine klare Kontrollflusssteuerung in ihrer Programmlogik. Ein Beispiel dafür ist unser „Tic-Tac-Toe“, dessen „main“ den Kontrollfluss klar vorgibt.

In einem modernen grafischen Benutzerinterface werden viel höhere Anforderungen an ein Programm gestellt. Der Benutzer kann beliebige Aktionen durch die Tasten, Menü, Buttons,... auswählen, er kann das Fenster vergrößern, verkleinern, in den Hintergrund legen bzw. nach vorne holen. Andere Programme können die Umgebung verändern, der Benutzer kann neue Einstellungen vornehmen - und auf all das muss das Programm jederzeit sauber und kontrolliert reagieren.

Damit dies mit einem halbwegs vernünftigen und überschaubaren Programmieraufwand möglich ist, hat sich das Programmiermodell von EVA hin zu einem Event-gesteuertem Modell verändert. Hierbei liegt die Kontrolle nicht mehr primär bei dem Programm, sondern beim Betriebssystem, das bei jedem Ereignis (Event) eine entsprechende Funktion des Programms aufruft – sogenannte „Callback-Funktionen“.

Dieses event-gesteuerte Programmiermodell hat sich in der Praxis sehr bewährt, und findet sich natürlich auch in Java wieder. An jedes mögliche Ereignis kann der Programmierer eine *Funktion ankoppeln*, die bei ihrem Auftritt automatisch ausgeführt wird.

Das ganze ist am Anfang recht ungewohnt, und es sind viele Fragen zu klären:

- Wie kann auf Events reagiert werden, d.h. wie wird eine Callback-Funktion in Java implementiert?
- Wie können Events verhindert oder modifiziert werden können?
- Wie werden Events bei mehreren aufeinanderliegenden grafischen Elementen propagiert?
- Können Events programmgesteuert ausgelöst werden?
- Welche Elemente können Events auslösen, und welche darauf reagieren?
- Wie können Events und die dahinter liegende Benutzerlogik von den Daten und der Programmlogik getrennt werden?

Diese und weitere Fragen müssen von einem Event-Modell beantwortet werden - und dieses Modell gibt damit den Rahmen - oder auch die Philosophie - vor, in dem ein Programmierer ein Programm Design erschaffen und implementieren kann und muss.

Hinweis – damit haben wir natürlich ein Problem mit unserer Tic-Tac-Toe Lösung bzw. auch den darauf aufbauenden Lösungen. In dieser Lösung liegt die Steuerung des Kontrollflusses klar auf der Seite der Spiele-Logik, während uns das GUI Programmier-Modell diese Option nicht lässt. In einem späteren Kapitel werden wir mögliche Lösungen dieses Konflikts diskutieren.

18 Event-Modelle von Swing

Swing kennt zwei Event-Modelle:

- das interne Event-Modell – siehe Kapitel 18.2, und
- das externe Event-Modell – siehe Kapitel 18.3.

Achtung – nicht Java kennt zwei Event-Modelle, sondern Swing. Die Sprache stellt Ausdrucks-Elemente zur Verfügung, in deren Rahmen Bibliotheken und Programme entwickelt werden können. Wie eine Bibliothek (wie z.B. Swing) dann ihre Funktionalität dem Benutzer zur Verfügung stellt – d.h. wie sie benutzt wird – ist eine Frage der Bibliothek und nicht der Sprache. Die Sprache gibt den Rahmen vor indem sie sich bewegen kann.

18.1 Events und Gruppierungen

Ein unerfahrener Programmierer mag sich die Frage stellen: „Welche Events gibt es überhaupt?“

Grundsätzlich ist ein Event eine *elementare* Aktion des Benutzers, die eine Reaktion des Programms zur Folge haben kann. Dazu gehören z.B.:

- Jede Art der Tastatur-Betätigung.
- Jede Maus-Aktion wie Linksklick, Rechtsklick, Linksdoppelklick, Maus bewegen, usw.
- Fenster-Aktionen wie Fenster wird bewegt, vergrößert, verkleinert, minimiert, maximiert, geschlossen, usw.
- Fokus-Aktionen, d.h. ein Element verliert bzw. bekommt den Fokus.
- uvm.

Die exakt vorhandenen Events sind vom jeweiligen grafischen Element abhängig, das den Fokus hat. Z.B. hat ein Baum-Element sicher Events für das Selektieren von Einträgen, für das Auf- und Zuklappen von Teilbäumen, uvm. Diese Events machen aber z.B. für ein einfaches Fenster gar keinen Sinn.

Da alle grafischen Elemente viele Events haben, sind diese in Swing gruppiert. Gerade komplexere Elemente wie Tabellen und Bäume wären sonst sehr unübersichtlich. So gibt es z.B. bei den meisten GUI Elementen drei Gruppen von Maus-Events:

- Mouse
- MouseMotion
- MouseWheel

Hinweis – es kann auch programm-interne Events geben, z.B. Timer-Events – siehe Kapitel 20.10 – aber die meisten Events werden durch externe Aktionen des Benutzers angestoßen, wie Tastatur- oder Maus-Aktionen. Um die Sache einfach zu halten, gehen wir im Folgenden davon aus, dass alle Events von außen angestoßen werden. Ist dies nicht der Fall, durchläuft das Event nur nicht so viele Stufen – die grundsätzliche Abarbeitung in Swing ist aber immer gleich.

18.2 Internes Event-Modell

18.2.1 Der Fluss eines Events

Jedes Event, das der Benutzer auslöst – z.B. das Klicken mit der linken Maus-Taste in ein Fenster – durchläuft mehrere Stufen:

- Das Betriebssystem bildet dabei die unterste Ebene. Es interagiert mit Hilfe von Treibern mit der Hardware, und wird von jeder Aktion unterrichtet. Das Betriebssystem interpretiert die Aktion, und leitet sie gegebenenfalls an das aktive Programm weiter – in unserem Fall die JVM.
- Die JVM kennt das aktuelle Fenster – genau genommen das grafische Element, das den Fokus hält – und ruft eine entsprechende Event-Behandlungs-Funktion auf.
- Das aktive grafische Element hat sich beim Erzeugen automatisch bei der JVM registriert – dies passiert automatisch in den Konstruktoren der Basisklassen - und hat die entsprechende Event-Behandlungs-Funktion möglicherweise überschrieben. So erfährt es von dem Event und kann darauf reagieren.
- Hierbei wird zuerst die Funktion „processEvent“ aufgerufen, die das Event dann auf die jeweiligen Event-Gruppen „process“ Funktionen verteilt - siehe Kapitel 0.

18.2.2 Swings „process“ Funktionen und Event-Klassen

Für jede Gruppe gibt es in den Swing Klassen „process“ Funktionen, die bei den jeweiligen Events aufgerufen werden. Hier ein Teil der „process“ Funktionen der Fenster Klasse „JFrame“:

- protected void processWindowEvent(WindowEvent e)
- protected void processWindowFocusEvent(WindowEvent e)
- protected void processWindowStateEvent(WindowEvent e)
- protected void processKeyEvent(KeyEvent e)
- protected void processMouseEvent(MouseEvent e)
- protected void processMouseMotionEvent(MouseEvent e)
- protected void processMouseWheelEvent(MouseWheelEvent e)

Alle diese Funktionen sind „protected“, da sie nicht von außen aufgerufen werden sollen, aber natürlich zum Überschreiben gedacht sind, d.h. in den abgeleiteten Klassen ansprechbar sein müssen.

Alle „process“ Funktionen bekommen ein Event-Objekt als Parameter, das detailierte

Informationen zum jeweiligen Event enthält. Welche Informationen das jeweils sind, ist vom jeweiligen abhängig.

Die Namen der Event-Klassen entsprechen häufig dem dem der jeweiligen „process“ Funktion ohne Präfix „process“, d.h. die Funktion „processKeyEvent“ z.B. bekommt einen Parameter vom Typ „KeyEvent“.

Die Event-Klassen sind in den verschiedensten Packages definiert. Aber die grundlegenden Event-Klassen wie „WindowEvent“, „KeyEvent“ oder „MouseEvent“ kommen aus „java.awt.event“.

Achtung – im Normallfall sollte eine überschriebene „process“ Funktion zuerst die überschriebene Basis-Klassen Funktion aufrufen. Geschieht dies nicht, schneiden sie die geerbte Event-Behandlung vom Event-Fluß ab – d.h. sie entfällt komplett. Im Normallfall werden sie dies nicht erreichen wollen, ganz im Gegenteil.

Achtung - ein Teil der „process“ Funktionen - wie z.B. „processMouseMotionEvent“ - müssen explizit aktiviert werden. Der Grund ist, dass manche dieser Events sehr häufig passieren können - z.B. Bewegungen der Maus. Je nach Event-Gruppe, Plattform und Anwendung könnte diese aufwändige Event-Verarbeitung zu einer spürbaren Performance-Verschlechterung führen. Auf modernen Hardware-Plattformen sollte dies zwar nicht der Fall sein, aber beim Design von Swing wurde hier auf „*Nummer Sicher*“ gegangen.

„process“ Funktionen können explizit mit der Funktion „enableEvents“ und einer passenden Bitmaske an- und ausgestellt werden - siehe z.B. Kapitel 18.2.4. Implizit werden sie auch mit der Registrierung eines entsprechenden Listener-Objekts aktiviert - siehe Kapitel 18.3.2.

18.2.3 Beispiel „WindowClose“

In Kapitel 16.2 haben wir gelernt wie man ein erreicht, dass das Schließen eines „JFrame“ automatisch auch das Programm beendet. Selbst wenn die dort dargestellt Methode sicher für die meisten Fälle ausreichend und sinnvoll ist, wollen wir hier als Beispiel dies mal selber implementieren.

Dazu muss man wissen, dass das Schließen eines Fensters unter die *normalen* Window-Events fällt.

- Daraus folgt, dass wir in unserer eigenen Frame-Klasse die Funktion „processWindowEvent“ überschreiben müssen.
- Dann wollen wir auf den Event-Typ „WindowEvent.WINDOW_CLOSING“ reagieren, der mit „getID()“ abgefragt werden kann.
- In diesem Fall wollen wir das Programm direkt beenden - dies geht mit „System.exit(0)“.
- Und auch hier gilt - wie in Kapitel 0 und Kapitel 16.4 erklärt - dass natürlich zuerst die Basis-Klassen Funktion „processWindowEvent“ aufgerufen werden sollte.

Alles zusammen ergibt dann das folgende Programm:

```
public class Appl {

  public static void main(String[] args) {
    MyFrame frame = new MyFrame("Fenster mit eigenem Programm-Ende");

    // Nicht mehr notwendig - darum kuemmern wir uns jetzt selbst
    // frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    frame.setLocation(20, 20);
    frame.setSize(600, 400);
    frame.setVisible(true);
  }

}
```

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

public class MyFrame extends JFrame {

  public MyFrame(String title) {
    super(title);
  }

  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0);
    }
  }

}
```

18.2.4 Beispiel „Scribble“

Um das interne Event-Modell näher zu verstehen, wollen wir eine erste Version eines „Scribble“ Programms mit Hilfe des internen Event-Modells implementieren.

Scribble ist quasi ein extrem einfaches Zeichen-Programm, bei dem sie mit der Maus Kurven und Linien in das Fenster zeichnen können.

- Mit dem Drücken einer beliebigen Maus-Taste beginnt das Zeichnen.
- Solange die Maus-Taste gedrückt ist, wird entsprechend den Maus-Bewegungen gezeichnet.
- Mit dem Loslassen der Maus-Taste endet die Zeichen-Aktion.

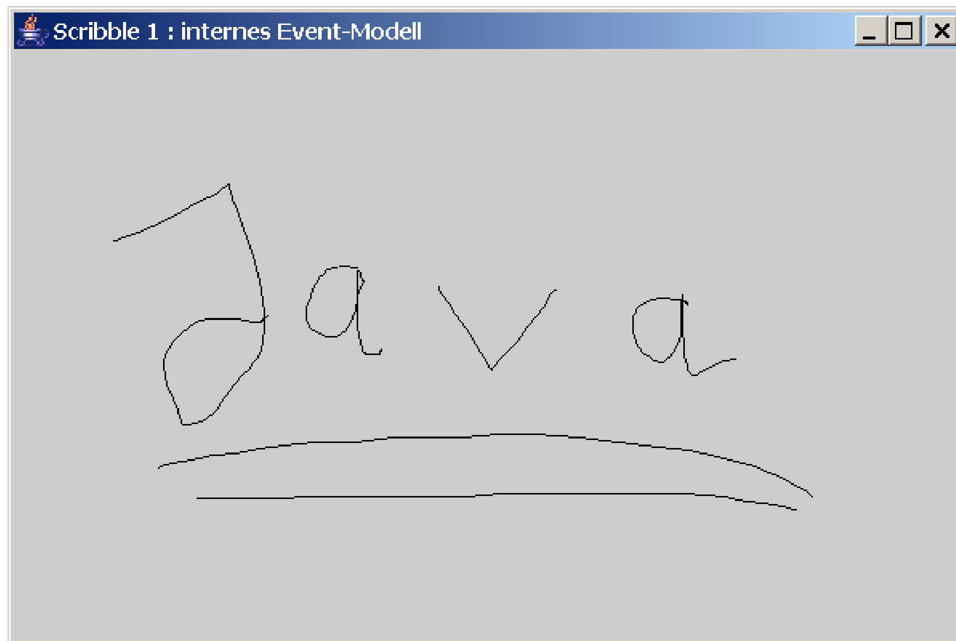

Abb. 18-1 : Ein einfaches Scribble Zeichen-Programm

Als erstes müssen wir uns überlegen, wie das Programm prinzipiell arbeiten soll:

- Die gezeichneten Striche sind Geraden zwischen den Punkten, die für die Maus-Bewegung als Event gemeldet werden.
- Für den ersten Strich benötigen wir also die x/y Koordinate der Maus beim Maus-Klick und nach der ersten Maus-Bewegung. Die x/y Koordinaten der Maus beim Maus-Klick müssen also gespeichert werden - hierzu werden zwei Attribute „lastX“ und „lastY“ in der Fenster-Klasse definiert.
- Für alle weiteren Striche werden die x/y Koordinaten zwischen Ende des letzten Strichs und der jeweils nächsten Maus-Bewegung benötigt - auch hier benutzen wir zur Speicherung die Attribute „lastX“ und „lastY“.
- Das Loslassen der Maus-Taste ist kein relevantes Event, da es an einer Maus-Position passiert, die schon als Maus-Bewegung gemeldet wurde.
- Damit auch nur ein einzelner Maus-Klick gezeichnet wird, muss noch der erste Punkt explizit gezeichnet werden.

Folgende Events sind also für Scribble wichtig:

- Eine beliebige Maus-Taste wurde gedrückt - dies fällt unter die *normalen* Maus-Events, d.h. die Funktion „processMouseEvent“ muss überschrieben werden. Das relevante Event ist das mit der ID „MouseEvent.MOUSE_PRESSED“ - die x/y Koordinaten können mit „getX“ und „getY“ erfragt werden.
- Das Ziehen mit gedrückter Maus-Taste muss verfolgt werden - dies ist eine Drag-Aktion und fällt unter die Maus-Motion-Events, d.h. die Funktion „processMouseMotionEvent“ muss überschrieben werden. Das relevante Event ist hier das mit der ID „MouseEvent.MOUSE_DRAGGED“.

Und so ergibt sich folgender Quelltext für unser „Scribble 1“. Denken sie daran, dass beide Event-Gruppen explizit mit „enableEvents“ aktiviert werden müssen - siehe auch Kapitel 0.

```
import javax.swing.JFrame;

public class Appl {

  public static void main(String[] args) {
    JFrame frame = new ScribbleFrame("Scribble 1 : internes Event-Modell");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLocation(20, 20);
    frame.setSize(600, 400);
    frame.setVisible(true);
  }

}
```

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

public class ScribbleFrame extends JFrame {

  private int lastX;
  private int lastY;

  public ScribbleFrame(String title) {
    super(title);
    enableEvents(
      AWTEvent.MOUSE_EVENT_MASK | AWTEvent.MOUSE_MOTION_EVENT_MASK);
  }

  protected void processMouseEvent(MouseEvent e) {
    super.processMouseEvent(e);
    if (e.getID() == MouseEvent.MOUSE_PRESSED) {
      onMousePressed(e);
    }
  }

  protected void processMouseMotionEvent(MouseEvent e) {
    super.processMouseMotionEvent(e);
    if (e.getID() == MouseEvent.MOUSE_DRAGGED) {
      onMouseDragged(e);
    }
  }

  private void onMousePressed(MouseEvent e) {
    lastX = e.getX();
    lastY = e.getY();
    Graphics g = getGraphics();
    g.drawLine(lastX, lastY, lastX, lastY);
  }

  private void onMouseDragged(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    Graphics g = getGraphics();
    g.drawLine(lastX, lastY, x, y);
    lastX = x;
    lastY = y;
  }

}
```

Hiermit haben sie ein ganz einfaches Scribble programmiert, das aber genau unter der in Kapitel 17.1 beschriebenen Problematik leidet: „Der Bildschirminhalt kann und wird auf Anforderung nicht neu gezeichnet, d.h. er geht unter Umständen verloren."

Probieren sie es aus: Minimieren sie z.B. das Fenster, schieben sie es in den Hintergrund, oder verschieben sie es zum Teil aus dem Bildschirmbereich, und stellen sie es dann wieder her. Der Bildschirm ist dann leer. Das Fenster muss neu gezeichnet werden, und zum Teil wird es das auch, denn um Rahmen, Titelleiste, usw. kümmern sich die Swing Basis-Klassen. Aber unser Inhalt fehlt, da wir uns nicht darum kümmern.

Wir werden dieses Problem in Kapitel 18.5 lösen, aber vorher schauen wir uns das externe Event-Modell an.

18.3 Externes Event-Modell

Das externe Event-Modell baut auf das sogenannte Observer Pattern auf. Von daher macht es Sinn, das Observer-Pattern kurz für sich zu besprechen.

„Design Pattern" bzw. Entwurfsmuster stellen mehr oder weniger fest umrissene **Design-Entwürfe** für konkrete Problem-Situationen dar. Sie enthalten keine exakte Code-Vorgabe, sondern sie erklären das konzeptionelle Klassen-Gerüst und die Verteilung der Aufgaben, um ein Problem zu lösen. Normalerweise finden sie d.h. keinen fertigen Quelltext, der nur noch abgetippt werden muss, wie z. B. bei Algorithmen, sondern sie müssen selbständig den Entwurf in passenden Code für ihr Projekt überführen.

18.3.1 Observer Pattern

18.3.1.1 Problem

Ein häufiges Problem ist, dass parallel mehrere (oft unterschiedliche) Sichten auf einen Daten-Bestand existieren. Wird der Daten-Bestand nun über eine der Sichten geändert, sollen sich alle anderen mit aktualisieren.

Ein sehr großes Beispiel wäre eine Daten-Visualisierung, bei der die Daten sowohl in Form einer Tabelle als auch in Form verschiedener Grafiken angezeigt werden. Werden nun die Daten in einer Sicht manipuliert, so sollen sich die anderen Sichten parallel anpassen

Ein anderes – vielleicht aktuelleres Beispiel – ist die Eclipse. Änderungen im Quelltext werden simultan im Package-Explorer, im Outline und in anderen Sichten angezeigt.

Das grundsätzliche Problem „ich-bin-interessiert-an-Änderungen-der-Daten" ist ein Standard-Problem der Software-Entwicklung und existiert in allen Größen-Ordnungen. Aber wie löst man es verständlich, wart bar, übersichtlich, erweiterbar, wiederverwendbar, usw. in einer objekt-orientierten Sprache? Die Antwort ist das Observer-Pattern.

18.3.1.2 Lösung

- Allgemeine abstrakte Basis-Klasse für die Daten
- Allgemeines Interfaces für die Sichten
- Kopplung nur zwischen der Basis-Klasse und dem Interface
- Benachrichtigungs-Mechanismus wird nur einmal in der Daten-Basis-Klasse implementiert, die konkreten Daten erben ihn.

```
import java.util.ArrayList;
import java.util.Iterator;

public class Observable {

  private ArrayList observers = new ArrayList();

  public void addObserver(Observer obs) {
    observers.add(obs);
  }

  public void notifyObservers() {
    Iterator i = observers.iterator();
    while (i.hasNext()) {
      Observer obs = (Observer)i.next();
      obs.update();
    }
  }

}
```

```
public class Data extends Observable {

  private String date = "";

  public void setDate(String arg) {
    date=arg;
    super.notifyObservers();
  }

  public String getDate() {
    return date;
  }

}
```

```
public interface Observer {

    public void update();

}
```

```
import java.io.*;

public class View1 implements Observer {

  private Data data;

  public View1(Data d) {
    data = d;
    data.addObserver(this);
  }

  public void update() {
    System.out.println("View 1 Daten \"" + data.getDate() + '"');
```

```
  }

  public void editData() {
    System.out.print("View 1 - bitte geben sie neue Daten ein:\n> ");
    try {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader reader = new BufferedReader(isr);
      String in = reader.readLine();
      data.setDate(in);
    }
    catch (Exception x) {
    }
  }

}
```

```
import java.io.*;

public class View2 implements Observer {

  private Data data;

  public View2(Data d) {
    data = d;
    data.addObserver(this);
  }

  public void update() {
    System.out.println("View 2 Daten \"" + data.getDate() + '"');
  }

  public void editData() {
    System.out.print("View 2 - bitte geben sie neue Daten ein:\n> ");
    try {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader reader = new BufferedReader(isr);
      String in = reader.readLine();
      data.setDate(in);
    }
    catch (Exception x) {
    }
  }

}
```

```
public class Appl {

  public static void main(String[] args) {
    Data data = new Data();
    View1 view1 = new View1(data);
    View2 view2 = new View2(data);
    data.setDate("Aus main");
    view1.editData();
    view2.editData();
  }

}
```

mögliche Ausgabe

```
View 1 Daten "Aus main"
View 2 Daten "Aus main"
View 1 - bitte geben sie neue Daten ein:
> Dateneingabe via View1
View 1 Daten "Dateneingabe via View1"
View 2 Daten "Dateneingabe via View1"
View 2 - bitte geben sie neue Daten ein:
```

```
> Und was aus View 2
View 1 Daten "Und was aus View 2"
View 2 Daten "Und was aus View 2"
```

Hinweis – da das Oberver-Pattern ein Standard-Problem in der Software-Entwicklung ist, gibt es schon seit dem JDK 1.0 in der Java Bibliothek die Klasse „java.util.Observer“ und das Interface „java.util.Observable“. Für viele Probleme sind diese Klassen allemal ausreichend.

18.3.2 Listener-Interfaces

Das externe Event-Modell ist im Prinzip die konsequente Anwendung des Observer Patterns auf Events. Jedes Event wird quasi als Daten-Änderung eines Observables betrachtet und an alle registrierten Observer gesendet.

Für jede Event-Gruppe gibt es ein entsprechendes Listener-Interface (das entspricht quasi dem Observer-Interface im Observer-Pattern). Und für jedes Event der Event-Gruppe gibt es eine abstrakte Funktion, die überschrieben werden muss (sie entsprechen quasi jeweils der „update“ Funktion des Observer-Interfaces.

Hier beispielhaft die Definitionen der Listener-Interfaces für die Maus- und Maus-Motion-Events aus dem Package „java.awt.event“:

```
package java.awt.event;

public interface MouseListener implements EventListener {

  public void mouseClicked(MouseEvent e);
  public void mousePressed(MouseEvent e);
  public void mouseReleased(MouseEvent e);
  public void mouseEntered(MouseEvent e);
  public void mouseExited(MouseEvent e);

}
```

```
package java.awt.event;

public interface MouseMotionListener implements EventListener {

  public void mouseDragged(MouseEvent e);
  public void mouseMoved(MouseEvent e);

}
```

Jedes GUI-Element, dass eine Event-Gruppe unterstützt, bietet entsprechende Funktionen an, um Listener beim GUI-Element zu registrieren und wieder abzumelden. So hat die Klasse „JFrame“ u.a. die Funktionen:

- public void addMouseListener(MouseListener)
- public void removeMouseListener(MouseListener)
- public void addMouseMotionListener(MouseMotionListener)
- public void removeMouseMotionListener(MouseMotionListener)

Hinweis – die Listener-Interfaces heißen immer wie die Event-Gruppen mit Postfix „Listener“. Die zugehörigen „add“ und „remove“ Funktionen heißen immer wie die Listener-Interfaces mit

Präfix „add“ bzw. „remove“.

Wir könnten unser Scribble also auch folgendermaßen implementieren:

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

public class ScribbleFrame
  extends JFrame implements MouseListener, MouseMotionListener {

  private int lastX;
  private int lastY;

  public ScribbleFrame(String title) {
    super(title);
    addMouseListener(this);
    addMouseMotionListener(this);
  }

  private void onMousePressed(MouseEvent e) {
    lastX = e.getX();
    lastY = e.getY();
    Graphics g = getGraphics();
    g.drawLine(lastX, lastY, lastX, lastY);
  }

  private void onMouseDragged(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    Graphics g = getGraphics();
    g.drawLine(lastX, lastY, x, y);
    lastX = x;
    lastY = y;
  }

  public void mouseClicked(MouseEvent e) {
  }

  public void mouseEntered(MouseEvent e) {
  }

  public void mouseExited(MouseEvent e) {
  }

  public void mousePressed(MouseEvent e) {
    onMousePressed(e);
  }

  public void mouseReleased(MouseEvent e) {
  }

  public void mouseDragged(MouseEvent e) {
    onMouseDragged(e);
  }

  public void mouseMoved(MouseEvent e) {
  }

}
```

18.3.3 Listener-Adapter-Klassen

Die notwendige Implementierung aller Funktionen der Listener-Interfaces ist häufig nervig, denn in der Praxis werden viele Funktionen leer überschrieben, da das Event für die Anwendung nicht wichtig ist – siehe z.B. Quelltext „Scribble2“ in Kapitel 18.3.2.

Eine Alternative ist hier die Verwendung der Listener-Adapter-Klassen statt der Listener-Interfaces. Die Adapter-Klassen implementieren alle Funktionen des jeweiligen Interfaces mit leeren Funktionen.

Hier beispielhaft die Definitionen der Listener-Adapter-Klassen für die Maus- und Maus-Motion-Events aus dem Package „java.awt.event“:

```
package java.awt.event;

public class MouseAdapter implements MouseListener {

  public void mouseClicked(MouseEvent e) {}
  public void mousePressed(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}

}
```

```
package java.awt.event;

public class MouseMotionAdapter implements MouseMotionListener {

  public void mouseDragged(MouseEvent e) {}
  public void mouseMoved(MouseEvent e) {}

}
```

Das es überhaupt Interfaces gibt liegt daran, dass Java keine Mehrfach-Vererbung von Klassen kennt, und daher z.B. das „Scribble2“ sich nicht von „JFrame“, „MouseAdapter“ und „MouseMotionAdapter“ ableiten kann sondern nur von einer Klasse. Dem gegenüber kann eine Klasse beliebig viele Interfaces implementieren, und kann so mehrere Aufgaben übernehmen.

Hinweis – die Listener-Adapter-Klassen heißen immer wie die Event-Gruppen mit Postfix „Adapter“.

18.3.4 Innere Klassen

Das „Scribble2“ ist erstmal kein wirklicher Fortschritt gegenüber dem „Scribble1“ – eher im Gegenteil, da auch die unbenutzten Interface-Funktionen leer implementiert werden müssen.

Aber das externe Event-Modell hat gegenüber dem internen einen wichtigen Vorteil:

- Im internen Event-Modell muss man immer eine eigene Klasse schreiben, die sich von der normalen Klasse ableitet und die entsprechende „process“ Funktion überschreibt. Eine solche eigene abgeleitete Klasse ist aber spezifisch für das aktuelle Problem und könnte daher nicht wiederverwendet werden. Daher benötigt man dann viele abgeleitete Klassen mit

sehr ähnlichem Code, was sehr unschön ist.
- Im externen Event-Modell dagegen können sie einen Observer beim zu beobachtenden Element registrieren, ohne das sie dieses anpassen müssten.

Um diesen Unterschied zwischen den beiden Event-Modellen noch mal klar zu machen, implementieren wir mit beiden Strategien die gleiche Aufgabe:
- Ausgabe der Maus-Koordinaten auf der Console bei Drücken einer beliebigen Maus-Taste in einem Fenster.
- D.h. konkret soll beim Drücken einer beliebigen Maus-Taste in einem Fenster die möglichst „private“ Element-Funktion „out“ der Klasse „Appl“ mit den x- und y-Koordinaten der Maus aufgerufen werden.
- Die Element-Funktion „out“ der Klasse „Appl“ soll die Koordinaten dann mit einer Meldung auf der Console ausgeben.

Bei der Aufgabe geht es darum, dass ein Event (hier Maus-Taste drücken) eines GUI-Elements (hier das Fenster) für ein Objekt einer anderen Klasse (hier „Appl“) eine Aktion auslöst (d.h. eine Element-Funktion aufruft). Dies ist eine Standard-Aufgabe. Denken sie z.B. an einen Button in einem Fenster. Wird der Button betätigt soll im Fenster irgendwas passieren. Die eigentliche Aktion ist also sicher nicht Bestandteil des Buttons, sondern des Fensters. Das Fenster in unserer Aufgabe entspricht hier also dem Button, das „Appl“ Objekt dem Fenster, und die „out“ Funktion der Aktion. Und da die Aktion ein Teil des Verhaltens des Gesamt-Moduls ist, sollte „out“ möglichst „private“ sein.

Hinweis – die Aufgabe ist absichtlich so gewählt, da sie mit unserem aktuellen Wissen problemlos implementierbar ist, denn wir kennen die Fenster-Klasse „JFrame“ und die Maus-Events, aber z.B. noch keine Buttons.

18.3.4.1 Implementierung mit dem internen Event-Modell

Um die Aufgabe mit dem internen Event-Modell zu lösen, muss:
- eine eigene Fenster-Klasse von „JFrame“ abgeleitet werden,
- die Maus-Events im Konstruktor enabled werden, und
- die entsprechende „process“ Funktion überladen werden.

Außerdem muss die Element-Funktion „out“ des „Appl“ Objekts aufgerufen werden können - dazu muss das „Appl“ in unserer Fenster-Klasse bekannt sein - d.h. implementieren wir ein entsprechendes Attribut „appl“ und setzen dies im Konstruktor, der nun natürlich auch einen solchen Parameter benötigt. Die Funktion „out“ kann hierbei leider nicht „private“ sein.

Der Rest ist Pflicht, und sollte kein Problem mehr sein.

```
import javax.swing.JFrame;

public class Appl {

  public static void main(String[] args) {
    Appl appl = new Appl();
    appl.run();
```

```
  }

  private void run() {
    MyFrame frame = new MyFrame(this);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLocation(20, 20);
    frame.setSize(600, 400);
    frame.setVisible(true);
  }

  public void out(int x, int y) {
    System.out.println("Maus-Pressed an " + x + "/" + y);
  }

}
```

```
import java.awt.AWTEvent;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;

public class MyFrame extends JFrame {

  private Appl appl;

  public MyFrame(Appl a) {
    super("Maus-Pressed Koordinaten auf Console via internem Event-Modell");
    appl = a;
    enableEvents(AWTEvent.MOUSE_EVENT_MASK);
  }

  protected void processMouseEvent(MouseEvent e) {
    super.processMouseEvent(e);
    if (e.getID() == MouseEvent.MOUSE_PRESSED) {
      appl.out(e.getX(), e.getY());
    }
  }

}
```

18.3.4.2 Implementierung mit dem externen Event-Modell

Die Implementierung mit dem externen Event-Modell ist hier ganz einfach.

- Da „Appl“ keine Basis-Klasse hat, können wir „Appl“ direkt von „MouseAdapter“ ableiten, und sind nicht auf das Interface „MouseListener“ angewiesen. Darum brauchen wir auch nur die „process“ Funktion überladen, die uns interessieren.
- Da wir das externe Event-Modell benutzen und keine speziellen Fenster-Fähigkeiten brauchen, können wir direkt die Fenster-Klasse „JFrame“ nehmen.
- Jetzt können wir die „out“ Funktion auch „private“ machen – siehe Aufgaben-Stellung.

```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;

public class Appl extends MouseAdapter {

  public static void main(String[] args) {
    Appl appl = new Appl();
    appl.run();
  }

  private void run() {
```

```
    JFrame frame = new JFrame("M-P Ko auf Console via externem Event-Modell");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLocation(20, 20);
    frame.setSize(600, 400);
    frame.addMouseListener(this);
    frame.setVisible(true);
  }

  public void mousePressed(MouseEvent e) {
    out(e.getX(), e.getY());
  }

  private void out(int x, int y) {
    System.out.println("Maus-Pressed an " + x + "/" + y);
  }

}
```

Die Implementierung mit dem externen Event-Modell ist hier so einfach da:

- „Appl“ keine Basis-Klasse hat.
- Wir nur für ein GUI-Element „mousePressed“ überschreiben müssen.

18.3.4.3 Implementierung mit einer non-static Member-Klasse

Was wäre aber, wenn es zwei Fenster gäbe, für die die „MousePressed“ Events überwacht werden müssen? Das ginge natürlich auch, aber dann müsste die Funktion „mousePressed“ unterscheiden, welches Fenster das Event abgetriggert hat. Prinzipiell funktioniert das, da die Event-Klassen (hier „MouseEvent“) u.a. auch Informationen über das Source-Element enthalten – die Frage, die sich stellt, ist: „Will man das?“

Übertragen auf ein Fenster mit vielen Buttons hieße das z.B., dass die „mousePressed“ Funktion alle Buttons kennen und unterscheiden können muss. Das ergibt weder schönen noch wirklich wartbaren Code. Besser wäre es sicher, pro Button eine eigene kleine unabhängige „process“ Funktion zu haben.

Das Problem ist doch eigentlich, dass man folgendes braucht:

- Eine von „MouseListener“ abgeleitete „kleine“ Klasse.
- Am besten eine spezielle Klasse ohne explizite Basis-Klasse, damit man „MouseAdapter“ benutzen kann, und sie sich nur um die Event-Gruppe kümmern muss.
- Ein Objekt dieser Klasse muss mit dem Objekt der übergeordneten Klasse verbunden sein – in unserem Beispiel z.B. mit dem „Appl“ Objekt, oder mit dem Fenster-Objekt im Button Beispiel.
- Und es wäre schön, wenn sie auf die „private“ Elemente des übergeordneten Objekts zugreifen könnte – z.B. auf „private“ Funktionen – um das Modul-Konzept der übergeordneten Klasse nicht aufzubrechen.

Erinnern sie sich? So was gibt es, und nennt sich „eingebette nicht-static Klasse“, „innere nicht-static Klasse“, „Member-Klasse“ oder auch „Element-Klasse“.

Schreiben wir unser Beispiel mal auf eine Member-Klasse um.

```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;

public class Appl {

  class MouseEventTarget extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      out(e.getX(), e.getY());
    }
  }

  public static void main(String[] args) {
    Appl appl = new Appl();
    appl.run();
  }

  private void run() {
    JFrame frame = new JFrame("M-P Koor. auf Console mit Member-Klasse");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLocation(20, 20);
    frame.setSize(600, 400);
    frame.addMouseListener(new MouseEventTarget());
    frame.setVisible(true);
  }

  private void out(int x, int y) {
    System.out.println("Maus-Pressed an " + x + "/" + y);
  }

}
```

18.3.4.4 Implementierung mit einer lokalen Klasse

Da die Klasse nur an einer Stelle benötigt wird, können wir sie auch zu einer lokalen Klasse machen.

```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;

public class Appl {

  public static void main(String[] args) {
    Appl appl = new Appl();
    appl.run();
  }

  private void run() {
    JFrame frame = new JFrame("M-P Koor. auf Console mit lokaler Klasse");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLocation(20, 20);
    frame.setSize(600, 400);

    class MouseEventTarget extends MouseAdapter {
      public void mousePressed(MouseEvent e) {
        out(e.getX(), e.getY());
      }
    }
    frame.addMouseListener(new MouseEventTarget());

    frame.setVisible(true);
  }
```

```
  private void out(int x, int y) {
    System.out.println("Maus-Pressed an " + x + "/" + y);
  }

}
```

18.3.4.5 Implementierung mit einer anonymen Klasse

Und da auch nur ein Objekt der Klasse benötigt wird, kann man daraus auch eine anonyme Klasse machen.

```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;

public class Appl {

  public static void main(String[] args) {
    Appl appl = new Appl();
    appl.run();
  }

  private void run() {
    JFrame frame = new JFrame("M-P Koor. auf Console mit anonymer Klasse");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLocation(20, 20);
    frame.setSize(600, 400);

    frame.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        out(e.getX(), e.getY());
      }
    });

    frame.setVisible(true);
  }

  private void out(int x, int y) {
    System.out.println("Maus-Pressed an " + x + "/" + y);
  }

}
```

Die Verwendung einer anonymen Klasse – abgeleitet von einer Event-Adapter-Klasse – als Event-Listener-Klasse war bis Java 8 das normale Verfahren zum Event-Handling in Swing. Für funktionale Interfaces werden seit Java 8 Lambda-Ausdrücke bevorzugt.

18.3.4.6 Implementierung mit einem Lambda-Ausdruck

Noch eleganter, ist seit Java 8, die Nutzung eines Lambda-Ausdrucks, wenn das zu implementierende Interface ein funktionales Interface ist. Leider trifft dies auf viele AWT und Swing Interfaces nicht zu, da sie viel älter sind. Wenn Sie aber die Möglichkeit für einen Lambda Ausdruck haben, dann bevorzugen Sie ihn.

18.3.5 Scribble 3 mit anonymen Listener-Klassen

Lassen sie uns jetzt unser Scribble mit dem externen Event-Modell und anonymen Listener-Klassen implementieren.

```
import javax.swing.JFrame;

public class Appl {

  public static void main(String[] args) {
    JFrame frame = new ScribbleFrame("Scrib 3 mit anonymen Listener-Klassen");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLocation(20, 20);
    frame.setSize(600, 400);
    frame.setVisible(true);
  }

}
```

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

public class ScribbleFrame extends JFrame {

  private int lastX;
  private int lastY;

  public ScribbleFrame(String title) {
    super(title);

    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        onMousePressed(e);
      }
    });

    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        onMouseDragged(e);
      }
    });
  }

  private void onMousePressed(MouseEvent e) {
    lastX = e.getX();
    lastY = e.getY();
    Graphics g = getGraphics();
    g.drawLine(lastX, lastY, lastX, lastY);
  }

  private void onMouseDragged(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    Graphics g = getGraphics();
    g.drawLine(lastX, lastY, x, y);
    lastX = x;
    lastY = y;
  }

}
```

18.4 Vergleich der Event-Modelle

Wann nimmt man nun welches Event-Modell?

- Im Normallfall ist das externe Event-Modell vorzuziehen. Im Zusammenspiel mit anonymen Listener-Klassen ist es einfach und unproblematisch zu verwenden.
- Bei der Entwicklung eigener bzw. spezialisierter GUI-Elemente kann es Sinn machen, sich in die interne Event-Verarbeitung einzuklinken. Auf dieser Ebene hat man mehr viel Einfluss und Möglichkeiten, der bei der Entwicklung eigener Elemente notwendig sein kann. Aber man kann auch viel kaputt machen, da ein Eingriff in die interne Event-Verarbeitung eine Operation am offenen Herzen des Elements ist. Man sollte daher wissen, was man macht und was man will. Ansonsten sollte man besser die Finger davon lassen.

18.5 Scribble 4 mit Daten-Modell

Bleibt zum Schluss noch eine Sache zu tun: die versprochene Implementierung eines Scribbles, das auch nach dem Neu-Zeichnen sein Bild darstellt – siehe Kapitel 17.1.

Im Prinzip ist das ganz einfach: wir müssen uns das gezeichnete Bild merken, und es auf Abruf (d.h. in der Funktion „paint“ – siehe Kapitel 16.4) zeichnen. Dieses Problem zwingt uns dazu, uns Gedanken zu machen, was denn unser Scribble-Bild eigentlich ist bzw. wie es aufgebaut ist:

- Jede x/y-Koordinate ist ein Punkt.
- Mit der Maus zeichnet der Benutzer Linien-Züge, d.h. eine Aneinander-Reihung (oder Verkettung) von Linien – sogenannte „Polygone“. Hierbei ist der Extrem-Fall zu beachten, dass ein Polygon auch nur aus einem einzelnen Punkt bestehen kann. Ein Polygon ist also eine Reihe von beliebig vielen, aber mindestens einem Punkt.
- Von diesen Polygonen kann das Scribble beliebig viele enthalten.

Aus dieser Problem-Analyse ergeben sich zwangsläufig folgende notwendige Klassen:

- Eine Klasse für Punkte, die x/y- Koordinaten aufnehmen kann. Hiermit sind wir schnell fertig, da es im Package „java.awt“ eine einfache Klasse „Point“ für Punkte gibt.
- Eine Klasse „Polygon“ für Polygone, die beliebig viele Punkte aufnehmen kann. Um mindestens einen Punkt zu garantieren, wird der erste Punkt direkt im Konstruktor übergeben.
- Und eine Klasse „PaintModel“ für das gesamte Bild, die beliebig viele Polygone aufnehmen kann. Die Klasse heißt „PaintModel“ und nicht „Picture“, da man in der Software-Entwicklung häufig von Modellen redet, wenn strukturierte Daten gemeint sind. Z.B. werden wir bei Swing-Tabellen noch sogenannte „TableModel‘s“ kennen lernen.

```
import java.awt.Graphics;
import java.awt.Point;
import java.util.ArrayList;
import java.util.Iterator;

public class Polygon {

  private ArrayList points = new ArrayList();
```

```
  // Ersten Punkt doppelt einfuegen, damit mindestens
  // zwei Punkte da sind - siehe Funktion 'paint'.
  public Polygon(Point p) {
    points.add(p);
    points.add(p);
  }

  public void addPoint(Point p) {
    points.add(p);
  }

  // Diese Funktion setzt voraus, dass mindestens zwei Punkte
  // vorhanden sind - der Konstruktor garantiert dies.
  // - waere kein Punkt da    => Exception in Zeile (*)
  // - waere nur ein Punkt da => kein Zeichnen, da die  Schleife
  //                              nie betreten wird.
  public void paint(Graphics g) {
    Iterator it = points.iterator();
    Point start = (Point) it.next();                     // (*)
        while (it.hasNext()) {
            Point end = (Point) it.next();
            g.drawLine(start.x, start.y, end.x, end.y);
            start = end;
        }
    }

}
```

```
import java.awt.Graphics;
import java.awt.Point;
import java.util.ArrayList;
import java.util.Iterator;

public class PaintModel {

  private ArrayList polygons = new ArrayList();
  private Polygon actuellPolygon;

  public void addPolygon(Point p) {
    actuellPolygon = new Polygon(p);
    polygons.add(actuellPolygon);
  }

  public void addPoint(Point p) {
    actuellPolygon.addPoint(p);
  }

  public void paint(Graphics g) {
    Iterator it = polygons.iterator();
    while (it.hasNext()) {
      Polygon poly = (Polygon) it.next();
      poly.paint(g);
    }
  }

}
```

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

public class ScribbleFrame extends JFrame {

  private int lastX;
  private int lastY;
```

```
  private PaintModel model = new PaintModel();

  public ScribbleFrame(String title) {
    super(title);

    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        onMousePressed(e);
      }
    });

    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        onMouseDragged(e);
      }
    });
  }

  private void onMousePressed(MouseEvent e) {
    lastX = e.getX();
    lastY = e.getY();
    Graphics g = getGraphics();
    g.drawLine(lastX, lastY, lastX, lastY);
    model.addPolygon(new Point(lastX, lastY));
  }

  private void onMouseDragged(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    Graphics g = getGraphics();
    g.drawLine(lastX, lastY, x, y);
    lastX = x;
    lastY = y;
    model.addPoint(new Point(x, y));
  }

  public void paint(Graphics g) {
    super.paint(g);
    model.paint(g);
  }

}
```

19 Swing Layouts

Für viele GUI-Elemente gibt es in Swing natürlich fertige Klassen, z.B. für verschiedene Arten von Buttons, für Labels, Eingabe-Felder, Listen, Tabellen, Bäume, uvm. Ein Teil dieser Klassen besprechen wir in Kapitel 20. Bevor wir diese Elemente besprechen, sollten wir aber wissen, wie man sie in ein Fenster einfügt und positioniert.

Das ganze hat was von der Henne/Ei-Problematik. Man kann keine Elemente in ein Fenster einfügen, wenn man keine Layouts kennt. Aber wie soll man Layouts erklären, wenn man keine Elemente zum Einfügen hat.

Wir lösen das ganz pragmatisch – in Kapitel 19.1 führen wir ein einfaches GUI-Element ein, einen Button. Danach besprechen wir die Layouts, und nutzen erstmal für alle Beispiele nur Buttons.

Aber warum gibt es überhaupt Layouts? Man könnte doch einfach die GUI Elemente pixelgenau positionieren und fertig!? Dieses Vorgehen hat in der Praxis viele Probleme, da z.B. Schriftgrößen sehr unterschiedlich sind, und damit eine pixelgenaue Postionierung nicht sinnvoll ist. Außerdem können pixelgenaue Positionierungen z.B. sich nicht von alleine an wechselnde Fenster-Größen anpassen. Darum gibt es in Swing Layouts, die die Ausrichtung, Größe und Position selbständig angleichen.

19.1 Swing Klasse „JButton“

Für einen normalen Button gibt es in Swing die Klasse „JButton“ im Package „javax.swing“. Um einen Button mit Text auf der Schaltfäche zu erzeugten, muß der Konstruktor „JButton(String text)“ benutzt werden. Weitere Informationen zur Klasse „JButton“ finden sie in Kapitel 20.3.

Ein GUI-Element wird in das Fenster mit „getContentPane().add“ eingefügt.

```
import javax.swing.*;

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Fenster mit Button");
    getContentPane().add(new JButton("Hallo"));
  }

}
```

Dieser Code erzeugt ein Fenster, dass komplett mit einem Button ausgefüllt ist. Und der Button passt sich automatisch der Größe des Fensters an. Probieren sie es aus!

Abb. 19-1 : Ein Fenster mit einem Button

19.2 Grundlagen

Ein „JFrame“ Fenster unterteilt sich in den sogenannten Client-Bereich und den Rest. Der Rest sind die Titelzeile, die Rahmen, möglicherweise Menü und Statusleiste, usw. Der innere Bereich steht dem Programm zur Verfügung und nennt sich Client-Bereich.

Für die Verwaltung des Client-Bereichs enthält ein „JFrame“ Fenster ein sogenantes „Content-Pane“ Objekt - vom Typ her ist es ein „java.awt.Container“.

- Zugreifen kann man auf die Content-Pane mit der Funktion „getContentPane()“.
- In diese Content-Pane können GUI-Elemente mit „add“ eingefügt werden.
- Ohne Layout verwaltet die Content-Pane nur das zuletzt eingefügte Element. Im folgenden Programm z.B. enthält das Fenster nur den Button mit dem Text „3“.

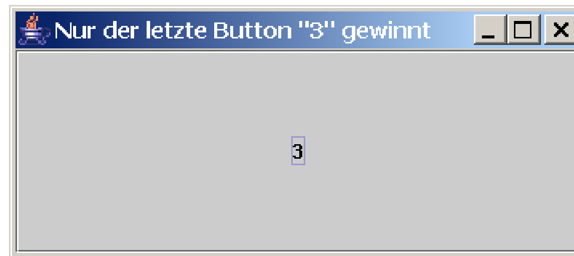

Abb. 19-2 : Die Content-Pane enthält nur den letzten Button

```
import javax.swing.*;

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Nur der letzte Button \"3\" gewinnt");
    getContentPane().add(new JButton("1"));
    getContentPane().add(new JButton("2"));
    getContentPane().add(new JButton("3"));
  }

}
```

Hinweis – GUI-Elemente im Sinne von Swing sind alle Objekte, die von „java.awt.Component“ abgeleitet sind.

19.3 Layouts

Layouts sind Klassen, die mehrere GUI Elemente verwalten können, und diese aneinander positionieren. Folgende Layouts sind u.a. Teil der Swing Bibliothek:

- Border-Layout
- Flow-Layout
- Grid-Layout

Swing enthält noch viel mehr Layout-Klassen, aber diese drei sind mit Abstand am einfachsten zu benutzen und decken schon viele Anwendungs-Fälle ab. Aus Zeitmangel werden wir uns daher auf diese drei beschränken.

19.3.1 Border-Layout

Das Border-Layout teilt sich in einen großen mittleren Bereich (CENTER) und vier Seiten-

Bereiche (NORTH, SOUTH, WEST und EAST) auf. Beim Hinzufügen eines Elements muss der Ziel-Bereich angegeben werden – dafür existieren entsprechende Konstanten in der Klasse „BorderLayout".

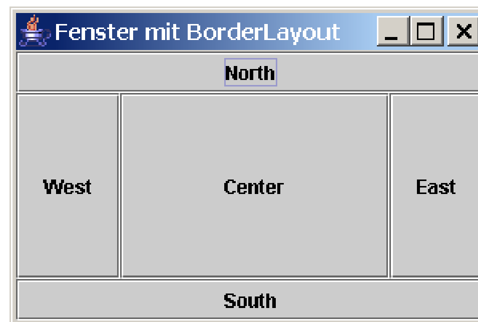

Abb. 19-3 : Fenster mit Border-Layout

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

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Fenster mit BorderLayout");
    setSize(300, 200);

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

}
```

Beim Border-Layout müssen nicht alle Bereiche gesetzt werden. Sind welche unbesetzt, dann werden die anderen einfach vergrößert – bis der Client-Bereich abgedeckt ist.

19.3.2 Flow-Layout

Das Flow-Layout ordnet Komponenten zeilenweise von links nach rechts und von oben nach unten an, wobei es preferredSize für jede Komponente verwendet. Es werden so viele Komponenten wie möglich in eine Zeile gesetzt, bevor eine neue Zeile begonnen wird.

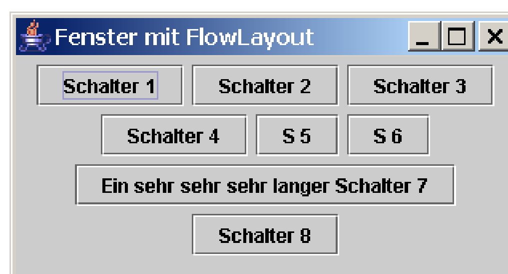

Abb. 19-4 : Fenster mit Flow-Layout

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

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Fenster mit FlowLayout");
    setSize(320, 170);

    Container c = getContentPane();
    c.setLayout(new FlowLayout());
    c.add(new JButton("Schalter 1"));
    c.add(new JButton("Schalter 2"));
    c.add(new JButton("Schalter 3"));
    c.add(new JButton("Schalter 4"));
    c.add(new JButton("S 5"));
    c.add(new JButton("S 6"));
    c.add(new JButton("Ein sehr sehr sehr langer Schalter 7"));
    c.add(new JButton("Schalter 8"));
  }

}
```

19.3.3 Grid-Layout

Das Grid-Layout fügt Komponenten in ein Gitter von Zellen, bestehend aus Zeilen und Spalten, ein. Es vergrößert die Komponenten auf den in der Zelle verfügbaren Platz. Jede Zelle hat dieselbe Größe. Das Gitter ist einheitlich. Wenn Sie die Größe eines Grid-Layout-Containers verändern, vergrössert Grid-Layout die Zellen im Rahmen des für den Container verfügbaren Platzes auf das größtmögliche Maß.

Abb. 19-5 : Fenster mit Grid-Layout

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

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Fenster mit GridLayout");
    setSize(320, 170);

    Container c = getContentPane();
    c.setLayout(new GridLayout(2, 3));
```

```
    c.add(new JButton("1"));
    c.add(new JButton("2"));
    c.add(new JButton("3"));
    c.add(new JButton("4"));
    c.add(new JButton("5"));
    c.add(new JButton("6"));
  }

}
```

19.4 Verschachtelte Layouts

Um Komponenten *„tiefer zu layouten“*, können Panels als Komponenten in ein Layout eingeführt werden. Panels können ihrerseits wieder ein Layout haben und Komponenten aufnehmen. Panels sind Objekte vom Typ „javax.swing.JPanel“ und auch ganz normale GUI Elemente – siehe auch Kapitel 20.8.

Das folgende Beispiel zeigt ein Fenster mit Border-Layout, bei dem im Center-Bereich ein Grid-Layout integriert ist.

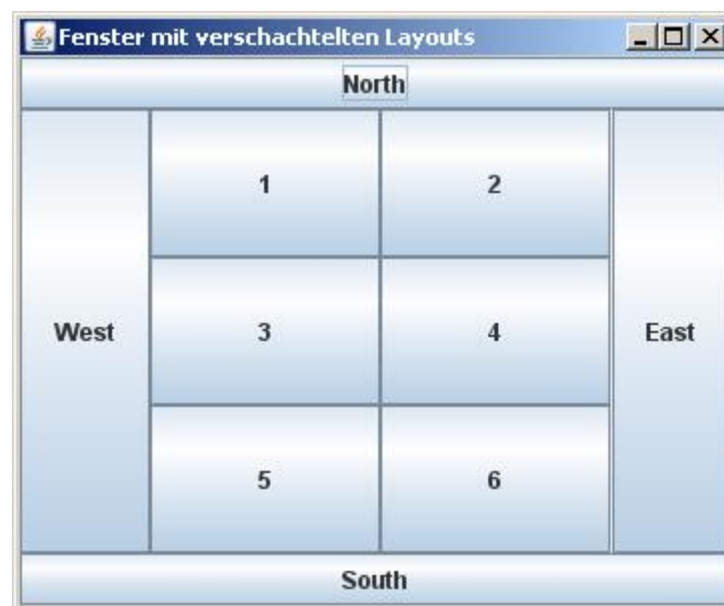

Abb. 19-6 : Fenster mit verschachtelten Layouts

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

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Fenster mit verschachtelten Layouts");
    setSize(400, 300);

    JPanel panel = new JPanel(new GridLayout(3, 2));
    panel.add(new JButton("1"));
    panel.add(new JButton("2"));
    panel.add(new JButton("3"));
    panel.add(new JButton("4"));
```

```
    panel.add(new JButton("5"));
    panel.add(new JButton("6"));

    Container c = getContentPane();
    c.setLayout(new BorderLayout());
    c.add(new JButton("North"), BorderLayout.NORTH);
    c.add(new JButton("West"), BorderLayout.WEST);
    c.add(new JButton("East"), BorderLayout.EAST);
    c.add(new JButton("South"), BorderLayout.SOUTH);
    c.add(panel, BorderLayout.CENTER);
  }

}
```

20 Swing GUI-Elemente

In diesem Kapitel werden kurz einige Swing GUI-Elemente vorgestellt. In Swing gibt es natürlich noch viel mehr fertige GUI-Elemente – hier können nur ein paar kleine wichtige GUI-Elemente vorgestellt werden. Und vor allem können all diese GUI-Elemente viel mehr, als hier beschrieben wird. Die Kapitel stellen nur kurze Einführungen in die wichtigsten Grund-Funktionalitäten der GUI-Elemente dar.

20.1 Labels

Die Klasse „javax.swing.JLabel" ist eine Klasse für einfache Labels (Beschriftungen).

Wichtige Element-Funktionen:

Konstruktor JLabel ()	Erzeugt ein einfaches Label.
Konstruktor JLabel (String text)	Erzeugt ein Label mit Text.
void setText(String text)	Setzt den Text des Labels neu.
String getText()	Gibt den Text des Labels zurück.
void setEnabled(boolean b)	Aktiviert bzw. deaktiviert das Label.

Das Beispiel ist ein einfaches Fenster mit Label.

Abb. 20-1 : Fenster mit Label

```
import javax.swing.*;

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Fenster mit Label");
    getContentPane().add(new JLabel("Ich bin ein Label"));
  }
```

```
}
```

20.2 Text-Felder

Die Klasse „javax.swing.JTextField“ ist eine Klasse für Text-Eingabe-Felder – auch Edit-Felder genannt.

Wichtige Element-Funktionen:

Konstruktor JTextField()	Erzeugt ein leeres Text-Feld.
Konstruktor JTextField (String text)	Erzeugt ein Text-Feld mit Text.
void setText(String text)	Setzt den Text neu.
String getText()	Gibt den Text zurück.
void setEnabled(boolean b)	Aktiviert bzw. deaktiviert das Text-Feld.

Wichtige Events:

- Caret
- Key

Name:	**Caret**
Beschreibung:	Wird aktiviert, wenn das Caret bewegt wird.
Listener-Interface:	javax.swing.event.CaretListener
Eine Funktion:	void caretUpdate(CaretEvent e)
Adapter-Klasse:	---
Event-Klasse:	javax.swing.event.CaretEvent

Name:	**Key**
Beschreibung:	Wird aktiviert, wenn die Tastatur benutzt wird.
Listener-Interface:	java.awt.event.KeyListener
Drei Funktionen:	void keyPressed(KeyEvent e) void keyReleased(KeyEvent e) void keyTyped(KeyEvent e)
Adapter-Klasse:	java.awt.event.KeyAdapter
Event-Klasse:	java.awt.event.KeyEvent

Das Beispiel zeigt automatisch bei jeder Änderung des eingegebenen Textes im unteren Label die Anzahl von Zeichen des Textes an.

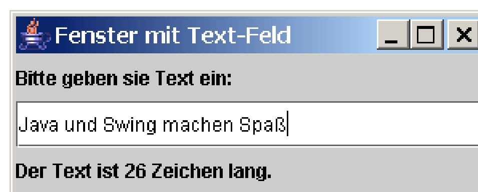

Abb. 20-2 : Fenster mit Text-Feld und automatischer Längen-Angabe

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

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

public class MyFrame extends JFrame {

  private JTextField field = new JTextField();
  private JLabel output = new JLabel();

  public MyFrame() {
    super("Fenster mit Text-Feld");

    onTextChanged();

    field.addCaretListener(new CaretListener() {
      public void caretUpdate(CaretEvent e) {
        onTextChanged();
      }
    });
    field.addKeyListener(new KeyAdapter() {
      public void keyTyped(KeyEvent e) {
        onTextChanged();
      }
    });

    Container c = getContentPane();
    c.setLayout(new GridLayout(3, 1));
    c.add(new JLabel("Bitte geben sie Text ein:"));
    c.add(field);
    c.add(output);
  }

  private void onTextChanged() {
    String text = field.getText();
    output.setText("Der Text ist " + text.length() + " Zeichen lang.");
  }

}
```

Hinweis – es werden sowohl „Update-Caret“ als auch „Key-Typed“ Events überwacht, da je nach Benutzung von „Entf“, „Backspace“, dem Clipboard, usw. unterschiedliche Events getriggert werden, und daher die Überwachung eines Events nicht ausreicht.

20.3 Buttons

Die Klasse „javax.swing.JButton“ ist eine Klasse für normale Buttons, auch „Push-Buttons“ genannt.

Wichtige Element-Funktionen:

Konstruktor JButton()	Erzeugt einen einfachen Button mit leerer Schaltfläche.
Konstruktor JButton(String text)	Erzeugt einen Button mit Text auf der Schaltfläche.
void setText(String text)	Setzt den Text der Schaltfäche neu.
String getText()	Gibt den Text der Schaltfäche zurück.
void setEnabled(boolean b)	Aktiviert bzw. deaktiviert den Button.

Wichtige Events:

- Action

Name: **Action**

Beschreibung: Wird aktiviert, wenn der Button betätigt wird.
Listener-Interface: java.awt.ActionListener
  Eine Funktion: void actionPerformed(ActionEvent e)
Adapter-Klasse: ---
Event-Klasse: java.awt.ActionEvent

Das Beispiel ist ein Fenster mit einem Button „Klick mich“. Wird der Button angeklickt, so ändert sich der Text auf „Aua“, und bei jedem weiteren Klick wird ein weiteres „aua“ hinzugefügt.

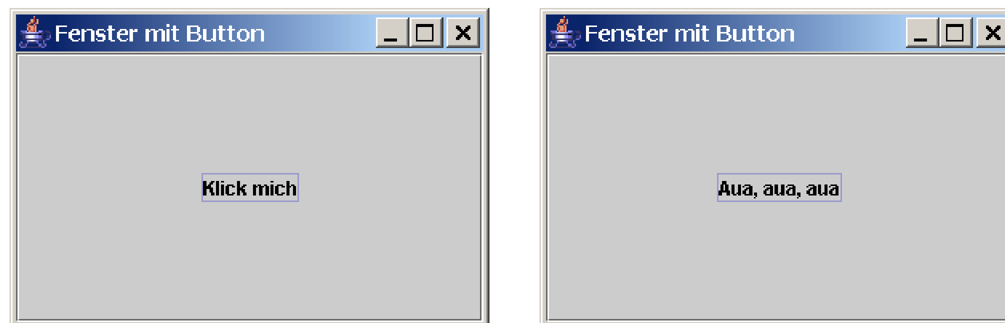

Abb. 20-3 : Fenster mit Button im initialen Zustand und nach dreimaliger Betätigung

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

public class MyFrame extends JFrame {

  private JButton button;

  public MyFrame() {
    super("Fenster mit Button");

    button = new JButton("Klick mich");
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        onButtonClick();
      }
    });

    getContentPane().add(button);
  }

  private void onButtonClick() {
    if (button.getText().equals("Klick mich")) {
      button.setText("Aua");
    }
    else {
      button.setText(button.getText() + ", aua");
    }
  }

}
```

20.4 Radio-Buttons

Die Klasse „javax.swing.JRadioButton“ ist eine Klasse für Radio-Buttons, d.h. Buttons die gruppiert auftreten, und von denen nur einer pro Gruppe selektiert sein kann.

Default-mässig ist erstmal jeder Radio-Button seine eigene Gruppe, d.h. alle Radio-Buttons schalten unabhängig voneinander an und aus. Sollen mehrere Radio-Buttons einander automatisch deselektieren, so müssen sie einer gemeinsamen Button-Gruppe zugeordnet werden – hierfür gibt es in Swing die Klasse „javax.swing.ButtonGroup“. Mit „add“ können Radio-Button einer Gruppe hinzugefügt werden.

Wichtige Element-Funktionen:

Konstruktor JRadioButton()	Erzeugt einen unselektierten Radio-Button ohne Text.
Konstruktor JRadioButton(String text)	Erzeugt einen unselektierten Radio-Button mit Text.
Ko. JRadioButton(String text, boolean b)	Erzeugt einen Radio-Button mit Text und entsprechender Selektion.
void setText(String text)	Setzt den Text neu.
String getText()	Gibt den Text zurück.
void setEnabled(boolean b)	Aktiviert bzw. deaktiviert den Radio-Button.
void setSelected(boolean b)	Selektiert bzw. deselektiert den Radio-Button.
boolean isSelected()	Gibt zurück, ob der Radio-Button selektiert ist.

Wichtige Events:

- Action – siehe Kapitel 20.3

Das Beispiel zeigt ein Fenster mit vier Radio-Buttons, die einer Gruppe zugeordnet sind. Ausserdem ist am unteren Rand ein Label vorhanden, das den jeweils selektieren Radio-Button explizit angibt – hierfür ist im Beispiel für jeden Radio-Button ein „ActionListener“ gesetzt.

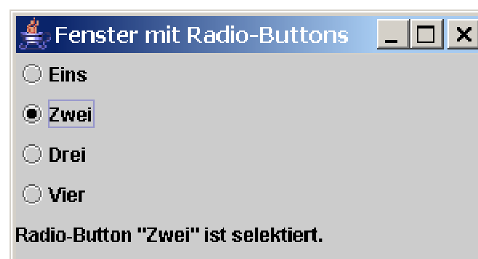

Abb. 20-4 : Fenster mit einer Gruppe von Radio-Buttons

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

public class MyFrame extends JFrame {

  private JRadioButton[] buttons = new JRadioButton[] {
    new JRadioButton("Eins", true),
    new JRadioButton("Zwei"),
    new JRadioButton("Drei"),
    new JRadioButton("Vier"),
  };
  private ButtonGroup group = new ButtonGroup();
  private JLabel output = new JLabel();

  public MyFrame() {
```

```
    super("Fenster mit Radio-Buttons");

    onSelectionChanged(buttons[0]);

    Container c = getContentPane();
    c.setLayout(new GridLayout(buttons.length+1, 1));
    for (int i=0; i<buttons.length; i++) {
      buttons[i].addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          onSelectionChanged((JRadioButton)e.getSource());
        }
      });
      group.add(buttons[i]);
      c.add(buttons[i]);
    }
    c.add(output);
  }

  private void onSelectionChanged(JRadioButton button) {
    output.setText("Radio-But. \"" + button.getText() + "\" ist selektiert.");
  }

}
```

Achtung – diese Gruppierung via „ButtonGroup" ist eine rein logische Gruppierung, und hat überhaupt nichts mit der Anordnung der Radio-Buttons im Fenster zu tun.

20.5 Check-Boxen

Check-Boxes sind spezielle Buttons, die meistens zwei Stati haben – diese werden im GUI durch Boxen mit Check-Häkchen dargestellt. In Swing werden sie durch die Klasse „javax.swing.JCheckBox" repräsentiert.

Wichtige Element-Funktionen:

Konstruktor JCheckBox()	Erzeugt eine unselektierte Check-Box ohne Text.
Konstruktor JCheckBox (String text)	Erzeugt eine unselektierte Check-Box mit Text.
Kon. JCheckBox (String text, boolean b)	Erzeugt eine Check-Box mit Text und entsprechender Selektion.
void setText(String text)	Setzt den Text neu.
String getText()	Gibt den Text zurück.
void setEnabled(boolean b)	Aktiviert bzw. deaktiviert die Check-Box.
void setSelected(boolean b)	Selektiert bzw. deselektiert die Check-Box.
boolean isSelected()	Gibt zurück, ob die Check-Box selektiert ist.

Wichtige Events:
- Action – siehe Kapitel 20.3

Das Beispiel zeigt ein Fenster mit drei Check-Boxen. Werden sie selektiert bzw. deselektiert, so wird automatisch ihre Vordergrund-Farbe auf „grün" bzw. „rot" gesetzt.

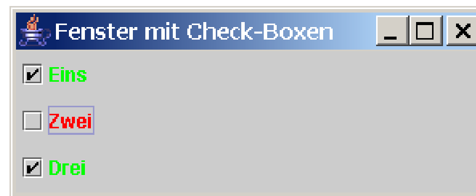

Abb. 20-5 : Fenster mit drei Check-Boxen

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

public class MyFrame extends JFrame {

  private JCheckBox[] buttons = new JCheckBox[] {
    new JCheckBox("Eins", true),
    new JCheckBox("Zwei"),
    new JCheckBox("Drei")
  };

  public MyFrame() {
    super("Fenster mit Check-Boxen");

    onSelectionChanged(buttons[0]);

    Container c = getContentPane();
    c.setLayout(new GridLayout(buttons.length, 1));
    for (int i=0; i<buttons.length; i++) {
      buttons[i].addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          onSelectionChanged((JCheckBox)e.getSource());
        }
      });
      onSelectionChanged(buttons[i]);
      c.add(buttons[i]);
    }
  }

  private void onSelectionChanged(JCheckBox button) {
    button.setForeground(button.isSelected() ? Color.GREEN : Color.RED);
  }

}
```

20.6 Scroll-Bars und Scroll-Panes

Möglicherweise ist dem ein oder anderen aufgefallen, dass unsere Fenster noch ein kleines Problem haben – macht man sie zu klein, so verschwinden Elemente. Die Layouts haben Grenzen, sobald ein Element so klein würde, dass es nicht mehr sinnvoll darstellbar wäre.

Möchte man in einem solchen Fall Scroll-Bars bekommen, so muss man sich nicht selber darum kümmern, sondern muss eine sogenannte Scroll-Pane zwischen Container und den enthaltenden Elementen legen. Im Extremfall ist dies sogar für ein einzelnes GUI-Element nötig, z.B. Tabellen – siehe Kapitel 20.7.2.

Ein Scroll-Pane ist eine Fläche, die Scrollbars zur Verfügung stellt – in Swing dargestellt durch

die Klasse „javax.swing.JScrollPane“ – ein Beispiel findet sich in Kapitel 20.7.2. Sowohl für die horizontale als auch für die vertikale Richtung können die Scrollbars auf verschiedene Arten unabhängig voneinander immer, nie, automatisch auf- und ausgeblendet werden – siehe auch das Interface „ScrollPaneConstants“.

Bemerkung – hier sieht man wieder eine ganz zentrale OO-Philosophie, die sich natürlich auch in Swing wiederfindet: Baue keine „ich-kann-alles“ Klassen, sondern zerlege das Problem in viele kleine Komponenten, die man überschreiben und/oder wiederverwenden kann.

20.7 Tabellen

Die Swing Tabellen-Klasse „javax.swing.JTable“ ist sehr leistungsfähig. Man kann quasi alles beeinflussen – bis hin zum Erscheinungs-Bild und den Editier-Möglichkeiten einer einzelnen Zelle. Für alle eigenständigen Aufgaben gibt es eigene Interfaces und Klassen, von denen man sich ableiten kann, und damit die Default-Einstellungen überschreiben kann.

Leider sind die Tabellen dadurch nicht immer einfach in ihrer Programmierung – und ein wirklich umfassender Einstieg wäre fast ein eigenes Buch. Von daher beschränkt sich dieses Kapitel auf die wesentlichen Themen.

20.7.1 Eine einfache Tabelle

Um eine einfache Tabelle zu erstellen, braucht es nicht viel. Man erzeugt ein Objekt der Klasse „javax.swing.JTable“ und übergibt dem Konstruktor die Anzahl an Zeilen und Spalten – im Beispiel 4 Zeilen und 3 Spalten.

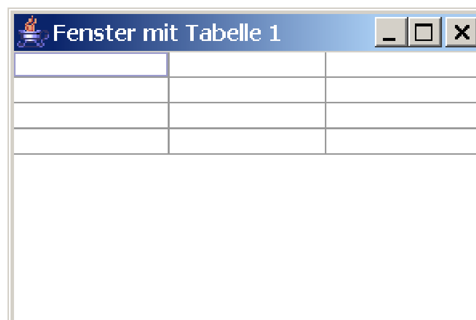

Abb. 20-6 : Fenster mit einfacher 4x3 Tabelle

```
import javax.swing.*;

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Fenster mit Tabelle 1");

    JTable table = new JTable(4, 3);
    setContentPane(table);
  }
```

```
}
```

20.7.2 Tabellen mit Scrollbars

Leider hat eine solche Tabelle ein Problem, das man erst bei einer größeren Anzahl an Zeilen bzw. Spalten sieht – darum hier eine Tabelle mit 40 Zeilen und 30 Spalten. Wie man sieht, hat die Tabelle keine Scrollbars.

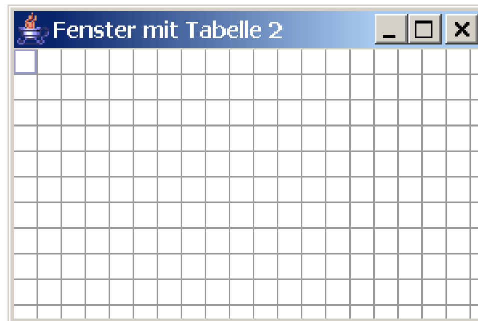

Abb. 20-7 : Fenster mit 40x30 Tabelle ohne Scrollbar

```
import javax.swing.*;

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Fenster mit Tabelle 2");

    JTable table = new JTable(40, 30);
    setContentPane(table);
  }

}
```

Wie wir in Kapitel 20.6 gelernt haben, müssen wir ein Scroll-Pane einsetzen. Wir machen dies hier in einer ganz primitiven Variante, ohne die Scrollbars hier wirklich optimal zu unterstützen.

Abb. 20-8 : Fenster mit 40x30 Tabelle mit Scrollbar

```
import javax.swing.*;

public class MyFrame extends JFrame {
```

```
  public MyFrame() {
    super("Fenster mit Tabelle 3");

    JTable table = new JTable(40, 30);
    JScrollPane scrollPane = new JScrollPane(table);
    setContentPane(scrollPane);
  }

}
```

20.7.3 Tabellen-Inhalt aus Arrays

Als wichtigstes fehlt aber noch der Inhalt der Tabellen – bislang sind sie leer und damit nicht besonders hilfreich.

Um Inhalte in eine Tabelle anzuzeigen, muss ein Tabellen-Modell erstellt werden – siehe Kapitel 20.7.4. Da dies für einfache Tabellen-Anzeigen sehr aufwändig ist, gibt es zwei „Hilfs“ Konstruktoren in „JTable“, die Arrays bzw. den Java-Container „java.util.Vector“ erwarten. Im Hintergrund baut das „JTable“ aus dem Array oder demContainer ein Default-Tabellen-Modell auf – siehe „javax.swing.table.DefaultTableModel“.

Im Beispiel bekommt „JTable“ 2 Arrays übergeben:

- Parameter 1 ist ein zwei-dimensionales Array, das den Inhalt der Tabelle enthält – typisiert ist er auf „Object[ ][ ]“. Da sich jedes Objekt immer in einen String wandeln lässt, kann dieses Array immer angezeigt werden.
- Parameter 2 ist ein ein-dimensionales Array, das die Spalten-Überschriften enthält. Auch es ist auf „Object[ ]“ typisiert.

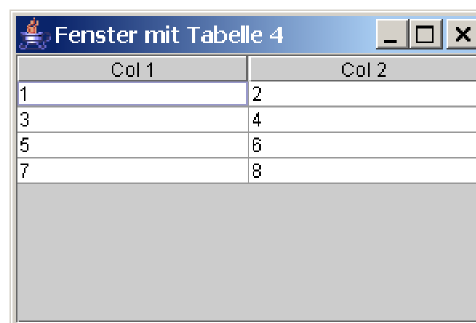

Abb. 20-9 : Fenster mit Tabelle, dessen Inhalt aus einem Array stammt

```
import javax.swing.*;

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Fenster mit Tabelle 4");

    String[][] values = {{"1","2"}, {"3","4"}, {"5","6"}, {"7","8"}};
    String[] titles = { "Col 1", "Col 2"};
    JTable table = new JTable(values, titles);
    setContentPane(new JScrollPane(table));
  }
```

```
}
```

20.7.4 Tabelle mit Tabellen-Modell

Für komplexere Inhalte, Interaktion, und weitreichende Einfluss Möglichkeiten muss der Tabelle ein Tabellen-Modell mitgegeben werden – d.h. das Objekt muss das Interface „javax.swing.table.TableModel" implementieren. In diesem Interface ist die minimale Schnittstelle zwischen Tabellen-Anzeige und Tabellen-Daten beschrieben, so z.B. Funktionen für die Anzahl an Zeilen und Spalten, für die Inhalte, und einiges mehr.

In der Praxis werden nicht alle diese Beeinflussungs-Möglichkeiten benötigt – d.h. gibt es die abstrakte Klasse „javax.swing.table.AbstractTableModel", die für einige Funktionen Default-Implementierungen anbietet. So müssen nur noch minimal drei Funktionen überschrieben werden.

Das folgende Beispiel nutzt genau diese Klasse, leitet sich von „AbstractTableModel" ab, und überschreibt den minimalen Satz an notwendigen Funktionen.

Fenster mit Tabelle 5

A	B	C
0 / 0	0 / 1	0 / 2
1 / 0	1 / 1	1 / 2
2 / 0	2 / 1	2 / 2
3 / 0	3 / 1	3 / 2
4 / 0	4 / 1	4 / 2
5 / 0	5 / 1	5 / 2
6 / 0	6 / 1	6 / 2
7 / 0	7 / 1	7 / 2
8 / 0	8 / 1	8 / 2

Abb. 20-10 : Fenster mit Tabelle mit Tabellen-Modell

```
import javax.swing.*;

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Fenster mit Tabelle 5");

    JTable table = new JTable(new MyTableModel());
    setContentPane(new JScrollPane(table));
  }

}
```

```
import javax.swing.table.*;

class MyTableModel extends AbstractTableModel {

  public int getRowCount() {
    return 15;
  }

  public int getColumnCount() {
    return 3;
  }
```

```
  public Object getValueAt(int arg0, int arg1) {
    return "" + arg0 + " / " + arg1;
  }

}
```

20.8 Panels

Auch die in Kapitel 19.4 vorgestellten Panels „javax.swing.JPanel“ sind ganz normale Gui-Elemente. Häufig werden sie einfach als eine Art GUI-Container in Verbindung mit Layouts genutzt.

Aber sie können auch als einfache GUI-Elemente benutzt werden, die z.B. eigene Zeichnungen enthalten – natürlich via Ableiten und Überschreiben der „paint“ Funktion.

20.9 Menüs

Um an ein „JFrame“ Fenster ein Menü anzuhängen, müssen drei Klassen benutzt werden:

- „javax.swing.JMenuBar“ repräsentiert das komplette Menü, d.h. die Menü-Zeile im Fenster.
- „javax.swing.JMenu“ repräsentiert einzelne Menüs, die Items, Seperatoren und Sub-Menüs enthalten können. Sub-Menüs sind wiederrum nur ganz normale Menüs – sie lassen sich halt verschachten.
- „javax.swing.JMenuItem“ sind einzelne Menü-Einträge, die vom Benutzer angewählt werden können.

Wichtige Element-Funktionen von „JMenuBar“

Konstruktor JMenuBar()	Erzeugt eine leere Menü-Zeile.
void add(JMenu m)	Fügt das Menü ans Ende der Menü-Zeile an.

Wichtige Element-Funktionen von „JMenu“

Konstruktor JMenu(String text)	Erzeugt ein leeres Menü mit dem übergebenen Text.
void add(JMenuItem item)	Fügt den Menü-Eintrag ans Ende des Menüs an. Hier dürfen auch „JMenu“ Objekte übergeben werden, da diese von „JMenuItem“ abgeleitet sind.
void add(String text)	Fügt einen Menü-Eintrag mit dem übergebenen Text ans Ende des Menüs an. Kurzform für „add(new MenuItem(text))“.
void addSeparator()	Fügt einen Separator ans Ende des Menüs an.
void setText(String text)	Setzt den Text des Menüs neu.
String getText()	Gibt den Text des Menüs zurück.
void setEnabled(boolean b)	Aktiviert bzw. deaktiviert das Menü.

Wichtige Element-Funktionen von „JMenuItem“

Konstruktor JMenuItem(String text)	Erzeugt einen Menü-Eintrag mit dem übergebenen Text.
void setEnabled(boolean b)	Aktiviert bzw. deaktiviert den Menü-Eintrag

Wichtige Events von „JMenuItem:
- Action – siehe Kapitel 20.3

Um ein Menü ohne spezielle Features zu erstellen, muß man also:
- eine Menü-Zeile erstellen,
- mindestens ein Menü an die Menü-Zeile anhängen,
- Menü-Einträge, Separatoren und Unter-Menüs an das Menü anhängen, und
- die Menü-Einträge mit Action-Listener versehen.

Hinweis – um Menü-Einträge selektiert, d.h. mit einem Häkchen versehen, darzustellen muss statt eines „JMenuItem“ ein Objekt der abgeiteten Klasse „JCheckBoxMenuItem“ genommen werden. Mit „void setSelected(boolean)“ kann die Selektion gesetzt, und mit „boolean isSelected()“ abgefragt werden.

Das Beispiel ist ein „JFrame“ Fenster mit einer Menü-Zeile, die zwei Menüs enthält. Im Bild ist das erste Menü zu sehen – es besteht aus zwei Einträgen, einem Separator und einem Sub-Menü. Das Sub-Menü enthält drei Einträge, wobei der erste deaktiviert und der dritte selektiert ist. Das zweite Menü „Menü-2“ enthält nur einen Eintrag, an dem aber ein Action-Listener hängt.

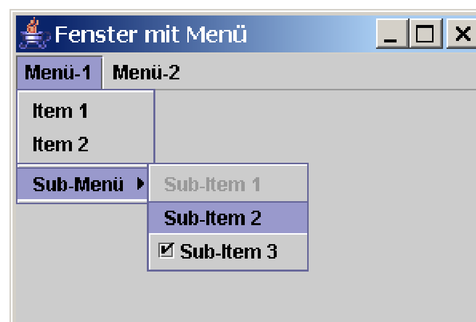

Abb. 20-11 : Fenster mit Menü

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

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Fenster mit Menü");

    JMenuItem subItem1 = new JMenuItem("Sub-Item 1");
    JMenuItem subItem2 = new JMenuItem("Sub-Item 2");
    JMenuItem subItem3 = new JCheckBoxMenuItem("Sub-Item 3");

    subItem1.setEnabled(false);
    subItem3.setSelected(true);

    JMenu submenu = new JMenu("Sub-Menü");
    submenu.add(subItem1);
    submenu.add(subItem2);
    submenu.add(subItem3);
```

```
    JMenu menu1 = new JMenu("Menü-1");
    menu1.add("Item 1");
    menu1.add("Item 2");
    menu1.addSeparator();
    menu1.add(submenu);

    JMenu menu2 = new JMenu("Menü-2");
    JMenuItem item3 = new JMenuItem("Item 3");
    item3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        setTitle("Item 3 wurde betätigt");
      }
    });
    menu2.add(item3);

    JMenuBar menubar = new JMenuBar();
    menubar.add(menu1);
    menubar.add(menu2);

    setJMenuBar(menubar);
  }

}
```

20.10 Timer

Für periodisch wiederkehrende Aufgaben gibt es in Swing eine Timer-Klasse „javax.swing.Timer“. Gestartet ruft sie in regelmäßigen Zeit-Intervallen – diese können in Milli-Sekunden definiert werden – die gesetzten Action-Listener auf.

Hinweis –bei positiven Intervall-Werten muss der Timer explizit gestartet werden – siehe Tabelle und Beispiel.

Wichtige Element-Funktionen:

Konstr. Timer(int delay, ActionListener l)	Erzeugt einen Timer mit dem Zeit-Intervall „delay“ in Milli-Sekunden und dem übergebenen Action-Listener.
void start()	Startet den Timer.
void stop()	Stoppt den Timer.

Wichtige Events:

- Action – siehe Kapitel 20.3

Das Beispiel erzeugt ein Fenster, dass jede halbe Sekunde die Farbe von „grün“ nach „rot“ und wieder zurück wechselt. Außerdem wird die aktuelle Farbe in der Titelzeile des Fensters angezeigt.

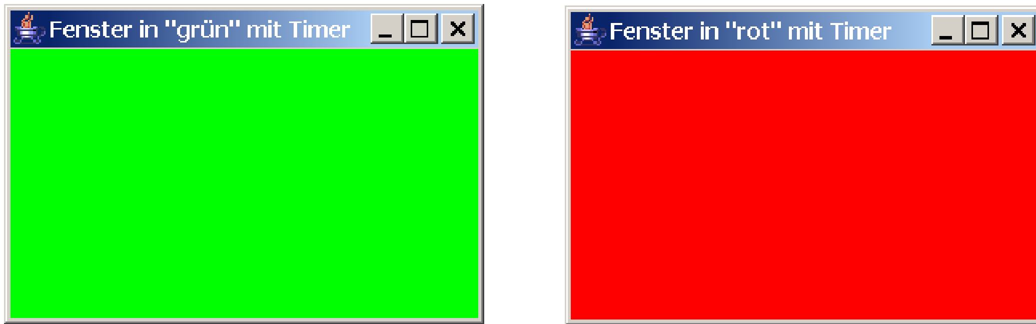

Abb. 20-12 : Fenster in grün und in rot – gesteuert von einem Timer

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

public class MyFrame extends JFrame {

  public MyFrame() {
    onTimer();

    Timer timer = new Timer(500, new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        onTimer();
      }
    });
    timer.start();
  }

  private void onTimer() {
    if (getContentPane().getBackground() == Color.GREEN) {
      getContentPane().setBackground(Color.RED);
      setTitle("Fenster in \"rot\" mit Timer");
    }
    else {
      getContentPane().setBackground(Color.GREEN);
      setTitle("Fenster in \"grün\" mit Timer");
    }
  }

}
```