GUI-Programmierung mit Java-Swing

Michael Dienert

18. Oktober 2010

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Was ist Swing?

Dieser Text enthält eine Anleitung, um mit Hilfe der *Swing*-Klassen von Java Programme mit grafischer Benutzeroberfläche zu schreiben.

Swing ist die zusammen mit Java 1.2 eingeführte, neue Bibliothek für Komponenten grafischer Benutzeroberflächen, sogenannte GUI-Komponenten. GUI ist die Abkürzung für **G**rafical **U**ser **I**nterface. Da Swing inzwischen die bei Java 1.0 und Java 1.1 verwendete GUI-Bibliothek **AWT** fast völlig abgelöst hat [1] , sollte man als Einsteiger in die Programmierung grafischer Benutzerschnittstellen gleich mit Swing beginnen.

Einen wichtigen Unterschied zwischen Swing und der alten AWT-Bibliothek muss man aber beachten: das AWT hat alle grafischen Komponenten mit Hilfe des jeweiligen Host-Betriebssytems gezeichnet. Was zur Folge hatte, dass die Anwendungen auf verschiedenen Plattformen (Solaris, Mac, Unix/Motif und ein weiteres, unbedeutendes Betriebssystem) auch verschieden aussahen. Ein weiterer Nachteil war, dass das AWT natürlich nur den kleinsten, gemeinsamen Nenner an GUI-Komponenten der drei Host-Betriebsysteme darstellen konnte. Komplexe Dialogelemente wie Bäume, Tabellen oder Registerkarten waren im AWT unbekannt, mit Swing lassen sie sich leicht realisieren.

Swing verwendet keine Routinen der Host-Betriebssysteme, sondern alle GUI-Komponenten werden von Swing selbst gezeichnet. Das jeweilige, sogenannte **Look-And-Feel** des Host-Systems wird dabei von Swing *nachgestellt*. Das hat folgende Auswirkungen:

- Swing-Anwendungen sehen auf jeder Plattform gleich aus.
- Swing bringt eigene, komfortable Komponenten wie z.B. die Klassen JTable für Tabellen oder JTree zur Darstellung hierarchischer Strukturen mit.
- Da das Aussehen der Oberfläche (Look-And-Feel) von Java selbst erzeugt wird, lässt es sich *während* der Laufzeit eines Programms umschalten. Man kann z.B. auswählen zwischen **Metal** (das Java-eigene Look-And-Feel), **Motif** (Motif ist eine beliebte GUI unter X-Windows), **Windows** und **Macintosh**. Aus rechtlichen Gründen lässt sich aber z.B. auf einem Mac kein Windows Look-And-Feel erzeugen und umgekehrt. Und auf einem Unix-Rechner sind nur Motif und Metal erlaubt.

[1] Bei der Programmierung von Applets sollte man noch das AWT verwenden, da die aktuellen Browser Swing noch nicht unterstützen.

- Da Swing die Host-Look-And-Feels nur nachbildet, sind kleine Abweichungen erkennbar.
- Wo soviel Licht ist, da ist natürlich auch Schatten: das Nachbilden der Look-And-Feels kostet viel Rechenzeit und Speicher, sprich Swing-Programme galten bisher als träge. Sun hat aber inzwischen nachgebessert und auf modernen Rechnern ist die Grösse des Hauptspeichers selbst für Swing-Programme mehr als ausreichend.

1.0.1 Ein sehr gutes Beispiel

Ein gutes, mit Hilfe von Swing geschriebenes Programm ist der Programmiereditor *jEdit*. Auf den ersten Blick scheint jEdit nur ein einfacher Editor mit GUI zu sein. Was jEdit aber auszeichnet ist seine Erweiterbarkeit mit Plugins, die aus jEdit eine vollständige IDE (Integrated Development Environment) machen. Ausserdem bringt jEdit den Java-Quellcode-Interpreter *BeanShell* mit. Dies erlaubt, Scripte für jEdit in Java zu schreiben.

1.0.2 Eine sehr schnelle Alternative zu Swing

Neben Sun ist die Firma IBM stark an der Java-Weiterentwicklung beteiligt (konkurierend). IBM hat mit *Eclipse* eine universelle IDE geschaffen, die nicht mit Swing programmiert ist sondern mit dem SWT (). SWT zeichnet nicht alle Oberflächenelemente mit der JavaVM, sondern verwendet wo immer möglich das darunterliegende Host-Betriebssystem.

SWT Programme sind deshalb deutlich schneller als Swing Programme. Allerdings ist SWT nicht mehr vollständig plattformunabhängig
[2] .

1.1 Schreibweisen von Klassen und Objekten in diesem Text

Alle Java-Klassen- und Objektnamen sowie Java-Schlüsselwörter sind in diesem Text in dieser `Schreibmaschinenschrift` gesetzt.

Ausserdem habe ich folgende Konvention eingeführt:

Bezieht sich der Text auf ein *Objekt* einer Klasse, wird als Objektname der Klassenname beginnend mit einem *Kleinbuchstaben* verwendet.

Ein Beispiel: die Klasse `JButton` dient zur Erzeugung von Schaltflächen auf einer grafischen Benutzeroberfläche. Um jetzt nicht immer umständlich schreiben zu müssen: *ein Objekt der Klasse* `JButton` , benutze ich einfach den Ausdruck `jButton`.

1.2 Im Text eingebettete Java-Beispiele

Java-Quelltextzeilen werden durch die vielen Schlüsselwörter oft sehr lang. Damit sie trotzdem in den schmalen, zweispaltigen Fliesstext eingebettet werden können, habe

[2] Am schnellsten und effektivsten ist es natürlich immer noch, vollständig auf eine GUI zu verzichten und eine Kommandozeilenanwendung zu schreiben. Leider hat man die Shell-Unterstützung bei der Java-Entwicklung bisher vernachlässigt, so dass das etwas mühsam ist.

ich manchmal einige zusätzliche Zeilenumbrüche in den Quelltext eingefügt. Der Java-Compiler wird diese ignorieren, aber der besseren Lesbarkeit wegen, sollten Sie diese Schreibweise nicht 1:1 übernehmen.

2 Die Klasse `JFrame`

Um die Swing-Klassenbibliothek zu erforschen, schreiben wir gleich ein Programm, mit dem wir die wichtigsten Swing-Komponenten ausprobieren können.

Da unser Programm eine grafische Benutzerführung erhalten soll, sollte sich nach dem Start ein neues Fenster, das Haupt- oder *Anwendungsfenster*, öffnen.

Hierzu gibt es die Klasse `JFrame`. Indem wir unsere Klasse von `JFrame` erben lassen, haben wir ohne viel Arbeit ein komplettes Anwendungsfenster zur Verfügung.

Anhang D zeigt den hierzu nötigen Quelltext. `LAF0.java` erweitert nicht nur die Klasse `JFrame`, sondern implementiert noch die Schnittstellen `ActionListener` und `WindowListener`.

Zunächst wird nur die Implementierung von `WindowListener` verwendet: mit dem Schliessen des Fensters wird auch die Anwendung komplett beendet. Dafür sorgt die Methode `windowClosing(WindowEvent)`.
Das Beenden des Programms beim Wegklicken des Hauptfensters kann man seit Java2-Version 1.3 alternativ auch mit folgender Zeile, die im Konstruktor unserer Fensterklasse stehen sollte, bewerkstelligen:

```
this.setDefaultCloseOperation(
               JFrame.EXIT_ON_CLOSE);
```

Das Programm ist noch ziemlich langweilig: es zeigt nur ein leeres Fenster. Um das Fenster mit Leben zu füllen, müssen ein paar Menues, Radioknöpfe, Textfelder usw. hinein.

Die Frage ist jetzt: wo sollen diese Komponenten "festgemacht" und wie sollen sie angeordnet werden?

3 Was sind Komponenten?

Alle Elemente einer grafischen Oberfläche unter Java haben als Oberklasse die Klasse `Component`. Man bezeichnet daher alle diese Elemente als *Komponenten*.

Hier folgt eine kleine Auswahl an Swing-Komponenten:

`jButton` Ein Druckknopf. Ein `jButton` kann mit einem Text beschriftet und/oder mit einem *Icon* versehen werden. Ein `jButton` ist wie alle Komponenten eine *Event-Quelle*. Ein gedrückter `jButton` erzeugt beim Loslassen ein `actionEvent`. Mit der Methode

```
jButton.addActionListener(ActionListener)
```

kann man für jeden `jButton` festlegen, welche Ereignisabhörer in diesem Fall informiert werden sollen. Weiteres zu diesem Thema folgt in Kap. 6.

`jCheckBox` Ein kleines Quadrat mit evtl. nebenstehender Beschriftung. Das Quadrat kann mit der Maus ausgewählt werden. Ein ausgewähltes Quadrat wird abhängig vom Look-And-Feel optisch markiert. Selbstverständlich erzeugt auch diese Komponente Ereignisobjekte.

Methode	Beschreibung
`void setBorder(Border border)`	Festlegen des Rahmens einer Komponente und dessen Aussehen.
`void setToolTipText(String text)`	Setzen eines Hilfetexts der erscheint, wenn der Mauszeiger über der Komponente steht.

`void setMinimumSize(Dimension minSize)`
`void setMaximumSize(Dimension maxSize)`
`void setPreferredSize(Dimension prefSize)`
	Grösse einer Komponente festlegen

`void setBackground(Color bg)`	Farbe festlegen
`Graphics getGraphics()`	Liefert den Grafikkontext um auf der Komponente zeichnen zu können.
`void setAccelerator(KeyStroke kS)`	Tastenkombination setzen; **diese Methode gibt es nur bei `menuItem`'s**!

Tabelle 1: Die allerwichtigsten Methoden der Klasse `JComponent`

`jTextField` Um Texteingaben vornehmen zu können, dienen Objekte vom Typ `JTextField`. Steht der Cursor im Textfeld und wird dann die ENTER-Taste betätigt, wird ein Event-Objekt erzeugt. Mit der Methode `getText()` kann man den Inhalt des Textfelds auslesen.

Eine elegantere Methode interaktiv Eingaben vom Benutzer entgegenzunehmen bietet die Klasse `JOptionPane`. Im Kapitel 8 ist diese Klasse näher beschrieben.

Die Tabelle 2 im Anhang enthält eine umfangreichere Liste an nützlichen Swing Komponenten.

3.1 Die Methoden der Klasse JComponent

Alle Komponenten (`jComponents`) besitzen eine eine grosse Anzahl nützlicher Methoden. In der Tabelle 1 ist ein kleiner Auszug der wichtigsten Methoden von `JComponent` aufgelistet.

3.2 Beschriftung von Komponenten

Der Beschriftungstext einer Komponente wird einfach als `string` Ihrem Konstruktor übergeben. Beginnt dieser `string` mit `<html>`, wird der nachfolgende Text als html-Quelltext interpretiert!

4 Was ist ein Container?

Die Klasse `Container` erbt direkt von der Klasse `Component` (siehe Kap. 3). Ein `container` ist also ein grafisches Element, eben eine Komponente. Das Besondere an Container-Objekten ist aber, dass sie wiederum andere Komponenten *enthalten* dürfen.

Und für alle Swing-Komponenten gilt: sie sind Unterklassen der Klasse `Container`. D.h. alle Swing-Komponenten dürfen ineinander geschachtelt werden.

4.1 Der Container eines `jFrame`

Auch `JFrame` ist eine Unterklasse von `Container`. Allerdings kann man in `JFrame`-Objekte nicht direkt weitere Swing- Komponenten einfügen, da ein `jFrame` bereits eine Komponente vom Typ `JRootPane` enthält.
Mit der Methode

```
jFrame.getContentPane()
```

kann man sich jedoch das Container-Objekt eines `jFrame` (eigentlich ja das der `jRootPane`) geben lassen. In dieses Container-Objekt können anschliessend die Komponenten eingefügt werden. Hierzu muss man die Methode

```
contentPane.add(Component)
```

der Klasse `Container` verwenden. Zur Erinnerung: `jFrame` und `contentPane` sind beides Objekte (sie beginnen ja mit einem Kleinbuchstaben). Ihr Name ist daher *beliebig*. Ich habe hier nur denselben Namen wie Ihre Klasse verwendet um deutlich zu machen, von welchem *Typ* die Objekte sind.
Hier der ganze Vorgang nochmal als komplettes Beispiel. Das Schlüsselwort `this` steht für ein Objekt der Klasse JFrame. D.h. das Beispiel steht innerhalb der Klassendefinition unseres Anwendungsfensters und dort wiederum im Konstruktor:

```
Container contentPane =
   this.getContentPane();

          ...

JButton sueden = new JButton("Sueden");
contentPane.add(sueden);
```

Abbildung 1 zeigt den Aufbau eines Swing-Fensters. Das `Container`-Objekt heisst `contentPane`. Oberhalb dieser Fläche kann evtl. noch eine Menueleiste platziert werden. Dazu muss man zunächst ein `JMenuBar`-Objekt erzeugen und dieses dann mit der Methode

```
jFrame.setJMenuBar(JMenuBar)
```

im Fenster plazieren.

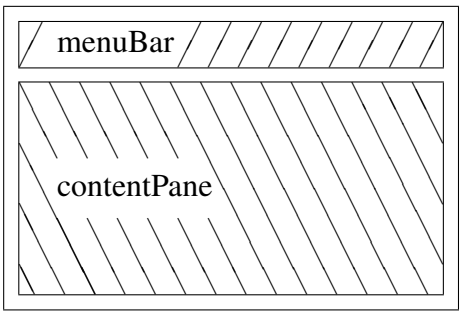

Abbildung 1: Der Inhalt eines Swing-Fensters

Weiter oben habe ich behauptet, dass alle Swing-Komponenten nicht vom Betriebssystem des Hostrechners, sondern von Java selbst gezeichnet werden. Dabei habe ich verschwiegen, dass das nicht für die Klasse `JFrame`, sowie `JDialog, JWindow` und `JApplet` gilt! Diese vier Klassen sind die *einzigen* Klassen der Swing-Bibliothek, die mit Hilfe des jeweiligen Host-Betriebssytems gezeichnet werden. Man erkennt das daran, dass beim Umschalten des Look-And-Feels der *äusserste* Rahmen des `JFrame`-Fensters unverändert bleibt.

Das Innere eines `jFrame` dagegen wird von einem `JRootPane`-Objekt verwaltet. Diese `jRootPane` und alles, was darin enthalten ist, wird nun vollständig von Java sebst gezeichnet.

5 Der Layout Manager

In Java gibt es eine Reihe sog. Layout Manager. Das sind Klassen, die alle das Interface `LayoutManager` implementieren. Objekte dieser Klassen legen fest, wie die Komponenten innerhalb eines Containers angeordnet werden sollen.

Dabei bleibt die Anordung der Komponenten zueinander erhalten, wenn die Grösse des Containers etwa vom Benutzer mit der Maus verändert wird. Der LayoutManager sorgt dann dafür, dass alles neu angeordnet wird.

Von den vielen Layout Managern wollen wir hier nur zwei erforschen, die Klassen `BorderLayout` und `FlowLayout`:

- Beim `FlowLayout` verhalten sich die Komponenten wie geschriebener Text: sie werden von links nach rechts und oben nach unten angeordnet.
- Beim `BorderLayout` sind die Komponenten angeordent wie die vier Himmelsrichtungen plus eine Komponente in der Mitte. Das ist das Default-Layout, wenn kein Layout Manager angegeben wird!

Um nun eine Komponente an einer gewünschten Stelle im Container platzieren zu können, muss man der Methode `add()` weitere Parameter mitgeben. Hier ein Beispiel, wie das Layout der `contentPane` gesetzt wird und wie anschliessend Komponenten darin platziert werden:

```
contentPane.setLayout(new BorderLayout(10,10));
       ...
JButton sueden = new JButton("Sueden");
contentPane.add(sueden, BorderLayout.SOUTH);
       ...
```

Mit den vielen Eigenschaften und Gestaltungsmöglichkeiten von `BorderLayout` muss man an einem Beispiel selbst experimentieren. Im Anhang finden Sie hierzu einige Fragen und Aufgaben.

6 Unsere Komponenten lösen Ereignisse aus

Bis hier sind wir so weit, dass wir unsere `contentPane` mit Komponenten füllen können.

Aber damit die Komponenten nicht nur unser Anwendungsfenster schmücken, sondern auch zu Bedienungselementen des Programms werden, müssen die Ereignisobjekte, die die Komponenten erzeugen, bei einem `ActionListener`-Objekt *registriert* werden.

Hier nochmals eine kleine Wiederholung der Ereignis-Abarbeitung in Java:

- In Java erzeugen *alle* Objekte der Klasse `Component` und ihrer Unterklassen, also alle GUI-Komponenten, Ereignisobjekte. Sie werden deshalb *Ereignisquellen* genannt.

  Ein `jButton` z.B. erzeugt ein `ActionEvent`-Objekt, wenn er gedrückt und wieder losgelassen wird. Und auch ein `jMenuItem` erzeugt ein `actionEvent`, wenn es mit der Maus ausgewählt wird.

- Jede Ereignisquelle hat wiederum eine Liste, auf der alle `EventListener`-Objekte (=Ereignisabhörer) stehen, die informiert werden sollen, wenn ein Ereignis auftritt.

  In unserem Beispielprogramm ist das ein Objekt unserer Klasse `LAF` selbst, da `LAF` das `ActionListener`-Interface implementiert. Abbildung 2 zeigt ein Klassendiagramm von `LAF`.

  Um nun innerhalb einer Klasse auf ein Objekt *der Klasse selbst* Bezug nehmen zu können, kann man keinen Objektnamen angeben. Man kann ja nicht vorher wissen, wie das Objekt heissen wird. Stattdessen verwendet man das Schlüsselwort `this` als Referenz auf ein *Objekt* der Klasse.

  Um nun einen Ereignisabhörer auf die Liste zu setzen, man sagt auch um einen Ereignisabhörer zu *registrieren*, verwendet man z.B. für `actionEvents` die Methode `addActionListener(ActionListener)`.

  Diese Methode besitzen alle Objekte, die `actionEvents` erzeugen können, wie z.B. ein `jButton` oder ein `jMenuItem`. Hier ein Beispiel für das Registrieren eines Listeners für eine Schaltfläche:

  ```
  jButton.addActionListener(ActionListener)
  ```

- Tritt nun ein Ereignis auf, wird automatisch ein Ereignisobjekt erzeugt. Z.B. ein `actionEvent` bei einem Klick auf einen `jButton`. Dieses wird dann der Methode `actionPerformed(ActionEvent)` des registrierten `eventListener` übergeben.

  Da wir diese Methode beim Implementieren des `EventListener`-Interfaces selbst schreiben mussten, können wir damit festlegen, was passieren soll, wenn z.B. ein bestimmter `jButton` gedrückt wird.

- Möglicherweise haben wir nun aber mehrere Komponenten, die `ActionEvent`-Objekte erzeugen. Und jedesmal wird nur die eine Methode

  ```
  actionPerformed(ActionEvent)
  ```

  aufgerufen.
  Nun muss man also herausfinden, *welche* der Komponenten das `actionEvent` erzeugt hat. Dabei hilft uns die Methode

  ```
  actionEvent.getActionCommand().
  ```

  Sie liefert z.B. die *Beschriftung* eines `jButton` (als `string`) , der angeklickt wurde.
  Man kann die String-Objekte, die `getActionCommand()` liefert, für jede Ereignisquelle mit der Methode

```
setActionCommand(String)
```

selbst festlegen. Das ist z.B. für `JTextField`-Objekte zwingend erforderlich, da sonst als `actionCommand` der Inhalt des Textfelds zurückgegeben wird und der ist natürlich variabel.

Legt man sie nicht extra fest, wird per Default die *Beschriftung* der Ereignisquelle verwendet.

Da das jetzt reichlich viel Theorie war, folgen ein paar Beispielzeilen:

1. Erzeugen eines `jButtons` und registrieren des Ereignisabhörers:

```
JButton sueden = new JButton("Sueden");
contentPane.add(sueden, BorderLayout.SOUTH);
sueden.addActionListener(this);
```

2. Hier wird ein `CheckBox`-Objekt erzeugt, der Namens-String des Ereignis-Objekts geändert und ein Ereignisabhörer registriert:

```
JCheckBox westen =
          new JCheckBox("West-JCheckBox");
contentPane.add(westen, BorderLayout.WEST);
westen.setActionCommand("checkBox");
westen.addActionListener(this);
```

3. Hier ein Code-Ausschnitt der zeigt, wie man innerhalb der Methode `actionPerformed(ActionEvent)` mit `if`-Anweisungen herausfiltert, welche Komponente das Ereignis ausgelöst hat:

```
public void
actionPerformed(ActionEvent ev) {
 if (ev.getActionCommand().
          equals("Sueden")){
    System.out.println(
               "Sueden gedrueckt!");
 }
               ...
               ...

 else if (ev.getActionCommand().
                equals("checkBox")){
    System.out.println("
               checkBox betaetigt");
 }
}
```

7 Menueleisten

In Abb. 1 ist ausser der `contentPane` noch eine Menueleiste gezeigt. Wie *alles* andere in Java, ist auch die Menueleiste ein Objekt. Und zwar ein Objekt der Klasse `JMenuBar`.
Mit der Methode

```
jFrame.setJMenuBar(JMenueBar)
```

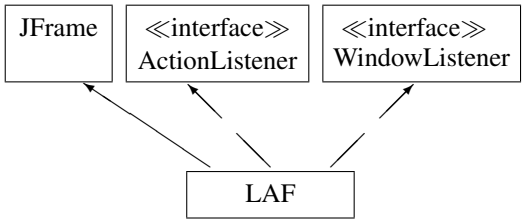

Abbildung 2: Die Klasse `LAF` implementiert zwei Schnittstellen

wird die Menueleiste oben ins Anwendungsfenster eingefügt.
Zuvor muss man die Menueleiste aber aus einzelnen Menues zusammenbauen: Eine Menueleiste kann mehrere Objekte vom Typ `JMenu` enthalten. Mit der Methode

```
jMenuBar.add(JMenu)
```

werden diese Menueobjekte in die Leiste eingefügt.
Schliesslich enthält jedes Menue Menueeinträge, sog. `menuItems`, also Objekte der Klasse `JMenuItem`. Und diese werden ins Menue mit der Methode

```
jMenu.add(JMenuItem)
```

eingefügt.

Was sind nun `jMenuItem`'s? Eigentlich kennen wir derartige Objekte schon: sie sind nämlich sehr eng verwandt mit `jButton`'s: beide haben dieselbe Oberklasse `AbstractButton`. Das heisst, `jMenuItem`'s sind im Grunde nichts Anderes als *Schaltflächen*, nur werden sie eben mit der Maus über die Menueleiste bedient. Das Registrieren eines `eventListener`'s funktioniert deshalb auch genau gleich wie bereits im Kap. 6 beschrieben.

8 Dialogmenues

Um Meldungen an den Benutzer auszugeben oder Benutzereingaben entgegenzunehmen, bietet die Java-Klassenbibliothek vorkonfigurierte Dialogfenster an. Diese sind alle Objekte der Klasse `JOptionPane`. Den grundsätzlichen Aufbau eines solchen Fensters zeigt Abb. 3.

Um nun ein solches Dialogfenster zu erzeugen, rufen Sie einfach eine der vier folgenden *statischen* Methoden der Klasse `JOptionPane` auf:

static int showConfirmDialog(...) Diesen Dialog muss man bestätigen. Je nach Wahl der Options-Buttons liefert die Methode z.B. bei YES-NO-CANCEL die Werte 0,1 oder 2 zurück.

static String showInputDialog(...) Diese Methode liefert den eingegebenen String zurück.

static void showMessageDialog(...) Hier wird nichts zurückgeliefert.

static int showOptionDialog(...) dieser Methode übergibt man ein Array mit Optionen (z.B. ein String-Array) und erhält die Indexnummer des gewälten Array-Elements zurück.

Jenachdem welchen Dialog sie erzeugen möchten, kann *message* einfach eine Meldung an den Benutzer oder auch ein Fragetext sein. Das Textfeld gibt es nur bei einem Input Dialog.
Sehr bequem und schnell kann man das für den Optionsdialog nötige String-Array so erzeugen:

```
String[] a = {"null", "eins","zwei","drei"};
```

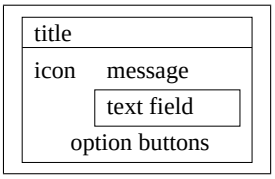

Abbildung 3: Struktur der `JOptionPane`-Dialogfenster

Hier ein Code-Beispiel für einen Input Dialog:

```
String eingabe =
  JOptionPane.showInputDialog(
     this,
     "hurra! Sie haben den Input"+
     "Button geklickt!"+
     "\n geben Sie bitte eine ganze"+
     "zahl ein:",
     "ich bin ein inputDialog",
     JOptionPane.PLAIN_MESSAGE
  );
```

A Wichtige Swing-Komponenten

Siehe Tabelle 2.

B Aufgaben

1. Kompilieren und starten Sie das Beispiel LAF0.java. Funktioniert das Beenden des Programms duch Klicken des Fenster-Schliessen-Knopfs?
2. Nun soll das Beenden der VM beim Schliessen des Fensters mit der Methode

   ```
   this.setDefaultCloseOperation(
               JFrame.EXIT_ON_CLOSE);
   ```

   erfolgen. Die Zeile `System.exit(0);` kann man anschliessend einfach herauskommentieren. Dieser Programmteil muss genau einmal abgearbeitet werden. An welcher Stelle muss er daher im Programm platziert werden?

   Da wir nun keine `windowEvents` mehr auswerten müssen, kann man jetzt das implementieren des Interface `WindowListener` entfernen.
3. Die `contentPane` des Fensters soll ein *BorderLayout* erhalten. Um das Layout sichtbar zu machen, sollen fünf `jButton`'s darin angeordnet werden. Dazu sind folgende Schritte notwendig:

Komponente	Beschreibung
`JButton`	Schaltfläche. Gekennzeichnet mit Icon oder Beschriftung, erzeugt `actionEvent`.
`JCheckBox`	Kontrollkästchen. Gekennzeichnet mit Icon oder Beschriftung, erzeugt `actionEvent`.
`JRadioButton`	Optionsschalter. Nur in Verbindung mit `buttonGroup`! Gekennzeichnet mit Icon oder Beschriftung, erzeugt `actionEvent`.
`JLabel`	Fester Text.
`JTextField`	Einzeilige Texteingabe. Gekennzeichnet mit Beschriftung, erzeugt `actionEvent`.
`JPasswordField`	Einzeilige Texteingabe. Angezeigt werden '*' statt Buchstaben. Gekennzeichnet mit Beschriftung, erzeugt `actionEvent`.
`JTextArea`	Mehrzeilige Texteingabe. Unterstützt Cut, Copy und Paste. Erzeugt `documentEvent`.
`JSlider`	Schieberegler. Erzeugt `changeEvent`.
`JScrollBar`	Horizontale und Vertikale Verschiebebalken. Erzeugt `adjustmentEvent`.
`JList`	Liste von Strings. Erzeugt `listSelectionEvent`.
`JComboBox`	Drop-Down-Liste. Ähnlich `JList`, aber Elemente auswählbar. Erzeugt `actionEvent` → `getSelectedItem()` liefert gewähltes Element.
`JTree`	Darstellung von Baumstrukturen. Sehr komplex.
`JTable`	Tabelle. Vollständig editierbar. Sehr komplex.
`JToolBar`	Werkzeugleiste. Mit der Methode `add(Component)` können Komponenten hineinplatziert werden.
`JProgressBar`	Fortschrittsanzeige. Erzeugt `changeEvent`.
`JPanel`	Rahmenloses Teilfenster innerhalb des Anwendungsfensters.
`JScrollPane`	Teilfenster mit Scrollbars. Diese werden nur angezeigt, wenn der Inhalt des Fensters grösser ist als der Fensterausschnitt.
`JSplitPane`	Ein horizontal oder vertikal geteiltes Teilfenster.
`JTabbedPane`	Karteikartenstapel. Mit `addTab(String, Component)` können neue Karteikarten hinzugefügt werden. Erzeugt `changeEvent`.
`JTextPane`	Eingabe und Darstellung von formatiertem (html und rtf) Text. Cut, Copy und Paste, mit Textcursor (Caret), mächtiger als `JTextArea`.
`JOptionPane`	Interaktive Message-Boxen. Siehe Kap. 8.
`JFrame`	Anwendungsfenster. Siehe Kap. . Erzeugt `windowEvent`.
`JInternalFrame`	Nebenfenster
`JDesktopPane`	Behälter für `jInternalFrame`'s.
`JDialog`	Erzeugt Dialogfenster ähnlich `JOptionPane`. Die Dialogfenster können aber frei gestaltet werden. Erzeugt `windowEvent`.
`JMenuBar`	Menueleiste eines `jFrame`'s. Wird mit `jFrame.setJMenuBar(JMenuBar)` oben im Anwendungsfenster platziert.
`JMenu`	Menue in einer Menueleiste. Wird mit `jMenuBar.add(JMenu)` in die Menueleiste aufgenommen.
`JMenuItem`	Menueeintrag in einem Menue. Wird mit `menu.add(JMenuItem)` ins Menue aufgenommen. Erzeugt `actionEvent`.
`JPopupMenu`	Kontextmenue.

Tabelle 2: Die allerwichtigsten Swing-Komponenten

(a) Erzeugen Sie als erstes ein Container-Objekt, das die `contentPane` unseres Anwendungsfensters enthält.

(b) Das Layout dieses Objekts soll dann als `BorderLayout` eingerichtet werden.

(c) Erzeugen Sie 5 `JButton`-Objekte und fügen Sie diese in das `Container`-Objekt ein.

4. Wie ändern sich die Grössenverhältnisse der Schaltflächen, wenn Sie die Grösse des `jFrame`'s ändern?

5. Beim Drücken eines `jButton`'s soll auf der Konsole eine Meldung ausgegeben werden, welcher Knopf gedrückt wurde.

   (a) Registrieren Sie für jeden `jButton` einen `actionListener`.

   (b) Schreiben Sie den Rumpf der Methode `actionPerformed(ActionEvent)`. Mit `actionEvent.getActionCommand()` soll ermittelt werden, welcher `jButton` das Ereignis ausgelöst hat.

6. Ersetzen Sie den mittleren `jButton` durch ein `jTextField`. Was muss ein Anwender unseres Programms tun, damit das Textfeld ein `actionEvent` erzeugt? Geben Sie beim Auftreten eines solchen Ereignisses den Inhalt des Text-Felds aus. Testen Sie auch die Klasse `JPasswordField`.

7. Der westliche `jButton` soll durch eine `jCheckBox` ersetzt werden, die mit "West-jCheckBox" beschriftet ist. Wird die CheckBox betätigt, soll die Methode `actionEvent.getActionCommand()` den Wert "checkBox" liefern.

8. Der nördliche `jButton` soll durch ein `jMenuItem` ersetzt werden.

9. Fügen sie dem Anwendungsfenster eine Menueleiste mit einem Dateimenue hinzu. Das Dateimenue soll die Punkte "Neu", "Oeffnen", "Schliessen", "Speichern" und "Speichern unter" enthalten.

   Jedesmal wenn ein Menupunkt gewählt wurde, soll auf der Konsole eine entsprechnde Meldung ausgegeben werden.

10. Ähnlich wie eine Menueleiste sind auch Werkzeugleisten, `JToolBar`-Objekte aufgebaut. Wie man ein `JToolBar`-Objekt erzeugt, sollen Sie diesmal in der API-Dokumentation von Sun nachlesen. Erzeugen Sie dann ein solches Objekt mit *vertikaler Orientierung* und fügen darin drei `jButton`'s mit der Methode `add` ein. Die Knöpfe sollen mit "Motif", "Metal" und "Speichern" beschriftet sein. Registrieren Sie für jeden der Knöpfe einen Ereignisabhörer. Was passiert beim Betätigen des Knopfs "Speichern" ? Warum ist das so?

11. Beim Betätigen der Knöpfe Motif bzw. Metal soll das Look-and-Feel der Anwendung umgeschaltet werden. Dies erreicht man durch Aufruf der Methode

```
setLAF("javax.swing.plaf.
        metal.MetalLookAndFeel");
```

bzw.

```
setLAF("com.sun.java.swing.plaf.
        motif.MotifLookAndFeel");
```

Abbildung 4: Die Methode `setLAF(String)`

```
public void setLAF(String laf){

   try{
      //LAF umschalten
      UIManager.setLookAndFeel(laf);
      SwingUtilities.
         updateComponentTreeUI(this);
   }
   catch (
      UnsupportedLookAndFeelException e){
      System.err.println(e.toString());
   }
   catch (ClassNotFoundException e) {
      System.err.println(e.toString());
   }
   catch (InstantiationException e) {
      System.err.println(e.toString());
   }
   catch (IllegalAccessException e) {
      System.err.println(e.toString());
   }
}
```

Die Methode `setLAF(String)` ist aber keine Methode aus der Java-Klassenbibliothek, sonden Sie müssen diese selbst in Ihren Source-Code schreiben. Da an dieser Stelle zuwenig Platz ist um Ihnen die dazu nötigen Hintergrundinformationen zu beschreiben, ist die die Methode im Anhang (Abb. 4) abgedruckt.

12. Nun sollen noch vier weitere Knöpfe in die Werkzeugleiste eingebaut werden. Mit diesen Knöpfen sollen jeweils folgende Optionsdialoge geöffnet werden:

    - Confirm Dialog
    - Input Dialog
    - Message Dialog
    - Option Dialog

    Falls bei einem Dialog der Benutzer etwas eingeben oder bestätigen muss, soll der Wert dieser Eingabe gleich wieder mit einem Message Dialog ausgegeben werden.

13. Damit die Werkzeugleiste schöner wird, soll die Grösse aller `jButton`'s einheitlich gesetzt werden. Vergleiche hierzu Tabelle 1 und die API-Doc.

C Die Methode `setLAF(String)`

Mit dieser Methode (Abb. 4) kann man das Look-And-Feel eines `JFrame`'s setzen.
Dies ist eine selbtgeschriebene Methode und keine Methode aus der Java-Klassenbibliothek.

D Beispielprogramm

Die Abb. 5 zeigt den Quelltext eines leeren Anwendungsfensters.

Abbildung 5: Das Code-Gerüst für ein Anwendungsfenster in Java

```
/*
 * @(#)LAF0.java 0.01 10.feb.2002
 *
 * By Michael Dienert. This is Freeware.
 *
 */

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

class LAF0 extends JFrame
    implements ActionListener, WindowListener{

    //implementieren des WindowListener interfaces

    public void windowClosing(WindowEvent e) {
        dispose();
        System.exit(0);
    }

    public void windowClosed(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}

    //implementieren des ActionListener interfaces

    public void actionPerformed(ActionEvent event) {
    }

    //konstruktor

    public LAF0() {
        super("Ein Swing-Beispiel");
        this.addWindowListener(this);
        this.setSize(600,400);
        setVisible(true);
    }

    //die unvermeidliche main-methode
    public static void main(String args[]) {
        new LAF0();
    }
```