Einführung in die GUI Programmierung mit Java

Christian Knauer

1

GUI - Programmierung

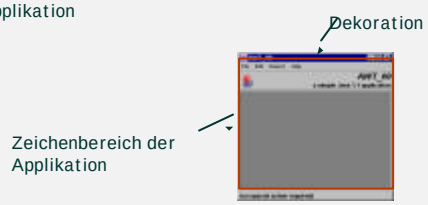
GUI =
Graphical User Interface =
Grafische Benutzeroberfläche

- Graphische Darstellung der Anwendungsdaten und Interaktionsmöglichkeiten in Fenstern
- Steuerung der Anwendung durch Maus- und Tastaturaktionen auf GUI-Elementen in den Fenstern (Widgets, Komponenten): Buttons, Menüs, Eingabefelder, Dialogboxen, Slider, Scrollbars, ...

2

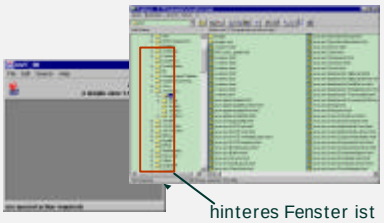
Grafische Benutzeroberflächen ...

- Display Manager
- Window Manager
- Applikation



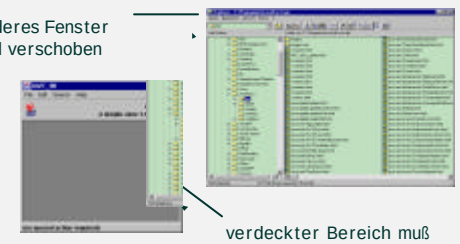
3

... Grafische Benutzeroberflächen ...



4

... Grafische Benutzeroberflächen



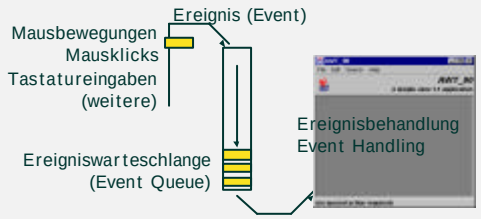
5

Aufbau grafischer Oberflächen

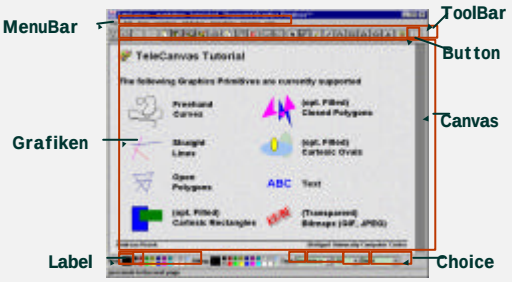
- hierarchische Anordnung von Komponenten
- Anordnung der Komponenten mithilfe eines "Layout-Manager"
- Komponenten haben "Funktionalität", reagieren auf Events
- "Programmieren" von Oberflächen bedeutet Erstellen von Event-Behandlungsroutinen

6

Ereignisorientierte Programmierung



Oberflächen-Komponenten



GUI-Programmierung mit Java

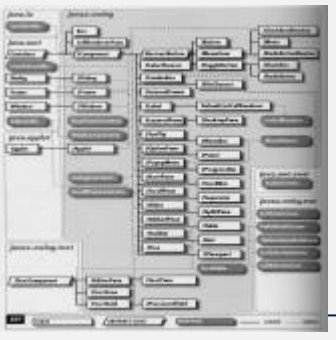
- Java 1.0: Abstract Windowing Toolkit
  - Bestandteil der Java API
- Java 2: Swing (leichtgewichtige Komponenten)
  - basiert auf AWT
  - Bestandteil der Java Foundation Classes (JFC)

AWT vs. Swing

- Schwergewichtige Komponenten (AWT):
  Komponente hat plattformspezifische Peerobjekte, die seine Funktionalität erbringen
  - nur kleinster gemeinsamer Nenner umsetzbar
  - Konsistenzprobleme
  - Performanzprobleme
- Leichtgewichtige Komponenten (Swing):
  Vollständige Implementierung der Komponenten in Java (mit konfigurierbarem "Look and Feel")
  - plattformunabhängig
  - größere Funktionalität
  - verbesserte Performanz

Swing

javax.swing.*



Swing Features

- Vielzahl von Komponenten
- gleiches Aussehen auf allen Plattformen
- beliebig geformte Komponenten
- Schaltflächen o.ä. mit Rastergrafiken
- frei wählbare Rahmen um Komponenten
- automatisches „Double Buffering“ (abschaltbar)
- „Tool Tips“
- Bedienung mittels Tastatur
- Unterstützung von Sprachanpassungen (Lokalisierungen)
- konfigurierbares Look-and-Feel
- „Model-View-Controller“-Ansatz (MVC)

13

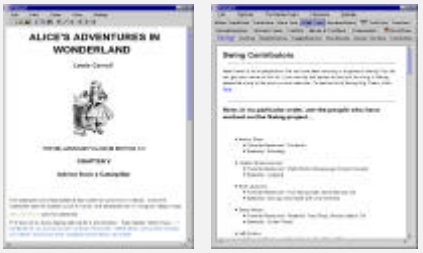
Swing Komponenten …

14

… Swing Komponenten …

15

… Swing Komponenten



16

Konfigurierbares Look-and-Feel

- regelt
  - Aussehen,
  - Bedienung und
  - Verhalten von Benutzeroberflächen
- in Swing implementiert
  - Windows
  - Motif
  - „Cross Plattform“ (Metal)

17

Ein erstes Swing-Programm

18

Programm

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

public class SwingApplication {
    private static String labelPrefix = "Number of button clicks: ";
    private int numClicks = 0;
    public Component createComponents() {
        final JLabel label = new JLabel(labelPrefix + "0    ");
        JButton button = new JButton("I'm a Swing button!");
        button.setMnemonic(KeyEvent.VK_I);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                numClicks++;
                label.setText(labelPrefix + numClicks);
            }});
      JPanel pane = new JPanel();
        pane.setBorder(BorderFactory.createEmptyBorder(30, 30, 10, 30));
        pane.setLayout(new GridLayout(0, 1));
        pane.add(button);
        pane.add(label);
        return pane; }
    public static void main(String[] args) {
        JFrame frame = new JFrame("SwingApplication");
        SwingApplication app = new SwingApplication();
        Component contents = app.createComponents();
        frame.getContentPane().add(contents, BorderLayout.CENTER);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }});
        frame.pack();
        frame.setVisible(true);}
```


Ausgabe

Import benötigter Pakete ...

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

public class SwingApplication {
    private static String labelPrefix = "Number of button clicks: ";
    private int numClicks = 0;
    public Component createComponents() {
        final JLabel label = new JLabel(labelPrefix + "0    ");
        JButton button = new JButton("I'm a Swing button!");
        button.setMnemonic(KeyEvent.VK_I);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                numClicks++;
                label.setText(labelPrefix + numClicks);
            }});
      JPanel pane = new JPanel();
        pane.setBorder(BorderFactory.createEmptyBorder(30, 30, 10,
      30));
        pane.setLayout(new GridLayout(0, 1));
        pane.add(button);
        pane.add(label);
        return pane; }
    public static void main(String[] args) {
        JFrame frame = new JFrame("SwingApplication");
        SwingApplication app = new SwingApplication();
        Component contents = app.createComponents();
        frame.getContentPane().add(contents, BorderLayout.CENTER);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }});
        frame.pack();
        frame.setVisible(true);}
```


... Import benötigter Pakete

```
import javax.swing.*;

import java.awt.*;
import java.awt.event.*;

public class SwingApplication {
    ...
```

- die Pakete aus **`java.awt.*`** und **`java.awt.event.*`** werden unter anderem für die Ereignisbehandlung benötigt

~~Erzeugen/Initialisieren eines Top-Level Containers ...~~

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

public class SwingApplication {
    private static String labelPrefix = "Number of button clicks: ";
    private int numClicks = 0;
    public Component createComponents() {
        final JLabel label = new JLabel(labelPrefix + "0    ");
        JButton button = new JButton("I'm a Swing button!");
        button.setMnemonic(KeyEvent.VK_I);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                numClicks++;
                label.setText(labelPrefix + numClicks);
            }});
      JPanel pane = new JPanel();
        pane.setBorder(BorderFactory.createEmptyBorder(30, 30, 10,
      30));
        pane.setLayout(new GridLayout(0, 1));
        pane.add(button);
        pane.add(label);
        return pane; }
    public static void main(String[] args) {
        JFrame frame = new JFrame("SwingApplication");
        SwingApplication app = new SwingApplication();
        Component contents = app.createComponents();
        frame.getContentPane().add(contents, BorderLayout.CENTER);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }});
        frame.pack();
        frame.setVisible(true);}
```


... Erzeugen des Top-Level Containers ...

```
public class SwingApplication {
    ...
    public static void main(String[] args) {
        ...
        JFrame frame = new JFrame("SwingApplication");
```

- jedes Programm hat mindestens einen Container
- Container können weitere Komponenten enthalten und anordnen
- Top-level Container in Swing: **`JFrame, JDialog, JWindow, JApplet`**
- Top-level Container stellen die Infrastruktur bereit, die Komponenten zum Zeichnen und zur Ereignisbehandlung benötigen

… Initialisieren des Top-Level Containers …

```
public class SwingApplication {
    ...
    public static void main(String[] args) {
        ...
        JFrame frame = new JFrame("SwingApplication");
        ...
        Component contents = app.createComponents();
        frame.getContentPane().add(contents, BorderLayout.CENTER);
        frame.addWindowListener(...);
```

- der Top-Level Container **frame** enthält den Container **contents** der wiederum die Basiskomponenten der Anwendung enthält
- diese Basiskomponenten werden in der Prozedur **createComponents** erzeugt und in den Container **contents** eingefügt
- Ereignisbehandlung für den Top-Level Container **frame** initialisieren

… Anzeigen des Top-Level Containers

```
public class SwingApplication {
    ...
    public static void main(String[] args) {
        ...
        frame.getContentPane().add(contents, BorderLayout.CENTER);
        frame.addWindowListener(...);
        frame.pack();
        frame.setVisible(true);
```

- Anzeigen des Top-Level Containers

Erzeugen des Labels/Buttons …

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

public class SwingApplication {
   private static String labelPrefix = "Number of button clicks: ";
   private int numClicks = 0;
   public Component createComponents() {
       final JLabel label = new JLabel(labelPrefix + "0    ");
       JButton button = new JButton("I'm a Swing button!");
       button.setMnemonic(KeyEvent.VK_I);
       button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                numClicks++;
                label.setText(labelPrefix + numClicks);
            }});
       JPanel pane = new JPanel();
       pane.setBorder(BorderFactory.createEmptyBorder(30, 30, 10,
30));
       pane.setLayout(new GridLayout(0, 1));
       pane.add(button);
       pane.add(label);
       return pane; }
   public static void main(String[] args) {
       JFrame frame = new JFrame("SwingApplication");
       SwingApplication app = new SwingApplication();
       Component contents = app.createComponents();
       frame.getContentPane().add(contents, BorderLayout.CENTER);
       frame.addWindowListener(new WindowAdapter() {
           public void windowClosing(WindowEvent e) {
               System.exit(0);
           }});
       frame.pack();
       frame.setVisible(true);}
```


… Erzeugen des Labels …

```
public class SwingApplication {
   private static String labelPrefix = "Number of button clicks: ";
    ...
    public Component createComponents() {
       final JLabel label = new JLabel(labelPrefix + "0    ");
       JButton button = new JButton("I'm a Swing button!");
       ...
```

- Erzeugen einer Basiskomponente **label** vom Typ **JLabel**

… Erzeugen des Buttons …

```
public class SwingApplication {
    ...
    public Component createComponents() {
        final JLabel label = new JLabel(labelPrefix + "0    ");
       JButton button = new JButton("I'm a Swing button!");
       button.setMnemonic(KeyEvent.VK_I);
       button.addActionListener(...);
       ...
```

- Erzeugen einer Basiskomponente **button** vom Typ **JButton**
- Definiert die Taste I als Mnemonic für diesen Button
- Ereignisbehandlung für die Komponente **button** initialisieren

… Erzeugen eines Containers für Label und Button

```
public class SwingApplication {
    ...
    public Component createComponents() {
        ...
        button.addActionListener(...);
        JPanel pane = new JPanel();
        ...
        pane.setLayout(new GridLayout(0, 1));
        pane.add(button);
        pane.add(label);
```

- Erzeugen eines Containers **pane** vom Typ **JPanel**
- Erzeugen eines **Layout Managers** für den Container
- Hinzufügen des Buttons und des Labels zu dem Container
- Damit werden diese Komponenten vom Layout Manager des Containers kontrolliert. Der Layout Manager eines Containers bestimmt die Grösse und Position jeder Komponente in dem Container.

Ereignisbehandlung ...

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

public class SwingApplication {
    private static String labelPrefix = "Number of button clicks: ";
    private int numClicks = 0;
    public Component createComponents() {
        final JLabel label = new JLabel(labelPrefix + "0    ");
        JButton button = new JButton("I'm a Swing button!");
        button.setMnemonic(KeyEvent.VK_I);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                numClicks++;
                label.setText(labelPrefix + numClicks);
            }});
        JPanel pane = new JPanel();
        pane.setBorder(BorderFactory.createEmptyBorder(30, 30, 10,
30));
        pane.setLayout(new GridLayout(0, 1));
        pane.add(button);
        pane.add(label);
        return pane; }
    public static void main(String[] args) {
        JFrame frame = new JFrame("SwingApplication");
        SwingApplication app = new SwingApplication();
        Component contents = app.createComponents();
        frame.getContentPane().add(contents, BorderLayout.CENTER);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }});
        frame.pack();
        frame.setVisible(true);}
```


... Ereignisbehandlung für den Button ...

```
public class SwingApplication {
    private static String labelPrefix = "Number ... ";
    private int numClicks = 0;
    ...
    public Component createComponents() {
      ...
      button.setMnemonic(KeyEvent.VK_I);
      button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            numClicks++;
            label.setText(labelPrefix + numClicks);
        }
      });
```

- Registriert einen Action Listener für die Komponente **`button`**
- Ein Action Listener ist ein Objekt, welches das Interface **`ActionListener`** implementiert, ibs. eine Methode **`actionPerformed()`** hat
- Clickt der Benutzer auf den Button, so wird die Methode **`actionPerformed()`** des Action Listener Objektes aufgerufen

... Ereignisbehandlung für den Top-Level Container

```
public class SwingApplication {
    ...
    public static void main(String[] args) {
        ...
        frame.getContentPane().add(contents, BorderLayout.CENTER);
        frame.addWindowListener(
            new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            });
```

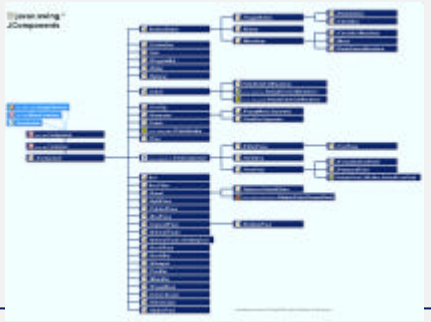
- Registriert einen Window Listener für die Komponente **`frame`**.
- Ein Window Listener ist ein Objekt, welches das Interface **`WindowListener`** implementiert, ibs. eine Methode **`windowClosing()`** hat.
- Clickt der Benutzer auf den Close-Button eines Fensters, in dem eine Komponente dargestellt wird, so wird die Methode **`windowClosing()`** aller Window Listener Objekte aufgerufen, die bei dieser Komponente registriert sind.

Komponenten und die Komponentenhierarchie

Komponentenhierarchie

- GUI aufgebaut aus Komponenten
- Container (spezielle Komponenten) können weitere Komponenten enthalten und anordnen
- Container, die auf oberster Hierarchiestufe verwendbar sind, basieren auf AWT-Containern (schwergewichtig)
- Top-level Container in Swing:
  `JFrame, JDialog, JWindow, JApplet`
- leichtgewichtige Komponenten erben von **`JComponent`**

Swing Klassenhierarchie unter `JComponent`



Methoden von JComponent

```
Border getBorder();
void SetBorder(Border);
boolean
isDoubleBuffered();
void
setDoubleBuffered(boolean
);
void
setToolTipText(String);
```

37

Das Beispielprogramm

erzeugt Komponenten

- `frame` vom Typ `JFrame`
- `panel` vom Typ `JPanel`
- `button` vom Typ `JButton`
- `label` vom Typ `JLabel`

38

Top-Level Container

- Können weitere Komponenten enthalten und anordnen
- Stellen die Infrastruktur bereit, die Komponenten zum Zeichnen und zur Ereignisbehandlung benötigen
- Top-Level Container in Swing: `JFrame`, `JDialog`, `JWindow`, `JApplet`

`JFrame`

`JDialog`

39

Container Komponenten

- Können weitere Komponenten enthalten und anordnen
- Bestimmen die Grösse und Position jeder enthaltenen Komponente
- Container in Swing: `JScrollPane`, `JTabbedPane`, …

`JScrollPane`

`JTabbedPane`

40

Basiskomponenten

- stellen elementare Informationen dar
- erhalten Eingaben vom Benutzer
- Basiskomponenten in Swing: `JComboBox`, `JTextField`, `JTable`

`JTextField`

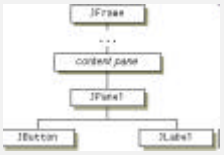
`JTable`

41

Basiskomponente JButton

42

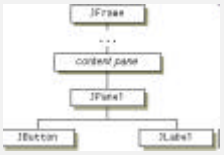
Komponentenhierarchie Beispielprogramm



- Diagramm der Container und Komponenten, die vom Programm erzeugt und benutzt werden
- Wurzel ist immer ein Top-level Container
- Der Top-level Container ist mit einem Fenster assoziiert. In diesem Fenster werden seine (sichtbaren) Nachfolgerkomponenten dargestellt.

43

Die content pane



- Jeder Top-level Container enthält eine Container Komponente, die *content pane*
- Die *content pane* enthält (direkt oder indirekt) alle sichtbaren Kompomenten der GUI des Fensters mit dem der Top-level Container assoziiert ist (Ausnahme: **`MenuBar`**)

44

Komponenten zu Containern hinzufügen

- Mit der **`add()`** Methode einer Container Komponente werden Komponenten zu dem Container hinzugefügt (und damit zu Kindern des Containers in der Komponentenhierarchie)
- Ein Argument der **`add()`** Methode spezifiziert die hinzuzufügende Komponente
- Manchmal muss man der **`add()`** Methode weitere Argumente übergeben, um z.B. das Layout der hinzugefügten Komponente im Container zu spezifizieren

45

Aus dem Beispiel

```
frame = new JFrame(...);
button = new JButton(...);
label = new JLabel(...);
pane = new JPanel();
pane.add(button);
pane.add(label);
frame.getContentPane().add(pane, BorderLayout.CENTER);
```

- Das **`JPanel`** **pane** wird zentriert in seinem Container (dessen content pane) dargestellt

46

Layout Management

47

Layout Manager ...

- Jeder Container hat einen Layout Manager
- Der Layout Manager eines Containers bestimmt die Grösse und Position jeder Komponente in dem Container
- Alle **`JPanel`** Objekte benutzen standardmässig den **`FlowLayout`** Manager
- Alle content panes (die Container in **`JApplet`**, **`JDialog`**, **`JFrame`**) benutzen standardmässig den **`BorderLayout`** Manager

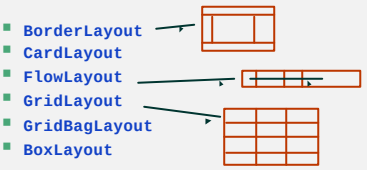
48

… Layout Manager …

- Informationen für Layout Manager müssen nur beim Erzeugen eines `JPanel`'s oder bei Hinzufügen von Komponenten zu content panes angegeben werden
- Die Argumente der `add()` Methode beim Hinzufügen von Komponenten zu einem Panel oder einer contentpane variieren für die verschiedenen Layout Manager (→ Dokumentation)
- Der Layout Manager einer Komponente kann mit der Methode `setLayout()` geändert werden
- Layoutmanager kann auf `null` gesetzt werden, wenn explizites Platzieren gewünscht wird

… Layout Manager

- Verfügbare Layout Manager
  - `BorderLayout`
  - `CardLayout`
  - `FlowLayout`
  - `GridLayout`
  - `GridBagLayout`
  - `BoxLayout`



Ereignisbehandlung

Ereignisse in einer GUI

- Eingaben des Benutzers
  - Mausbewegung
  - Mausklick
  - Tastendruck
- Nachrichten des Hostbetriebssystems
  - Komponente wird sichtbar
  - Komponente erhält den Fokus
- Timer

Benachrichtigung über Ereignisse

- Komponenten (sogar beliebige Objekte) können von für sie relevanten/interessanten Ereignissen benachrichtigt werden
- Die Objekte müssen sich dazu bei der Komponente anmelden, damit diese die geforderten Ereignisse weitermeldet
- Objekte können sich bei mehreren Komponenten anmelden und bei einer Komponente können mehrere Objekte als „Zuhörer“ angemeldet sein

Darstellung von Ereignissen in Swing

- Ein Ereignis wird durch ein Objekt repräsentiert
- Verschiedene Ereignistypen werden durch Objekte unterschiedlichen Typs repräsentiert (alle erweitern die Klasse `AWTEvent`)
- Ein Ereignisobjekt beinhaltet Informationen über das Ereignis sowie die Quelle des Ereignisses
  - Zugriff auf Quelle mit `Object getSource();`
  - Zufriff auf Typ mit `int getID();`
- Ereignisquellen sind üblicherweise Komponenten

Benachrichtigung über Ereignisse in Swing …

- Tritt das (physikalische) Ereignis ein, so
  - erzeugt die Java-Runtime für die betroffene Komponente ein Ereignisobjekt, und
  - Ruft bei allen an dieser Komponente für den Ereignistyp registrierten Event Listenern die Event Handler Methode der registrierten Objektes auf.
  - Dabei wird das Ereignisobjekt als Parameter übergeben.

55

… Benachrichtigung über Ereignisse in Swing

- Objekte die von einer Komponente über ein bestimmtes Ereignis (**Event**) X benachrichtigt werden wollen, müssen
  - ein passendes Interface (`XListener`) implementieren, d.h. eine Methode zur Verfügung stellen, die das Ereignis behandeln kann (**Event Handler**)
  - sich als **Event Listener** bei der Komponente (**Event Source**) anmelden (`addXListener()`)

56

Beispiel: Anklicken eines Buttons

`myActionListener.actionPerformed(event)`

`JButton button`

Der Mausklick veranlasst den angeklickten `JButton`, ein Objekt `event` vom Typ `ActionEvent` zu erzeugen.

Dieses wird an alle bei dem Button registrierten `ActionListener` Objekte geschickt.

`ActionEvent event`

Hat alle Informationen bzgl. des Ereignisses und seiner Quelle, sowie eine Beschreibung der ausgeführten Aktion.

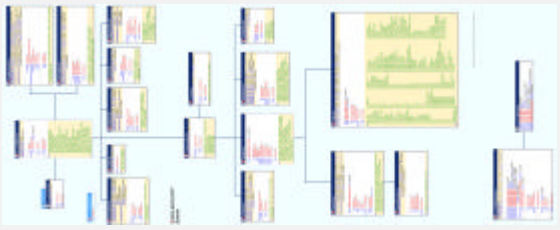
`MyListenerClass myActionListener`

implementiert die Event Handler Methode `actionPerformed()` in der festgelgt ist, was beim Eintreten des Events passieren soll.

Sie wird von dem Button bei dem das Objekt als `ActionListener` registriert ist aufgerufen.

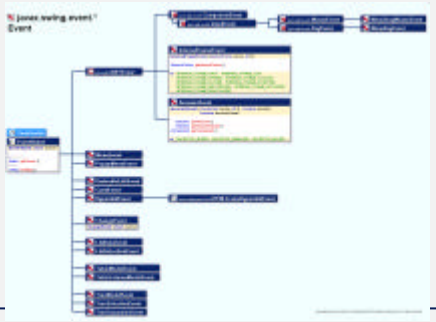
57

AWT Events



58

Swing Events



59

Implementierung von Event Listenern …

- Event Listener Objekte müssen Instanzen von Klassen sein, die entweder das passende Listener Interface implementieren oder von einer Klasse abgeleitet sind, die das tut

```
public class myActionListenerClass implements ActionListener {
…
   public void actionPerformed(ActionEvent e) {
       ...//code that reacts to the action...
   }
…
}
```

60

… Implementierung von Event Listenern

- Event Listener Objekte besitzen damit die zu dem entsprechenden Ereignis passende Event Handler Methode

```
public class myActionListenerClass implements ActionListener {
…
   public void actionPerformed(ActionEvent e) {
       ...//code that reacts to the action...
   }
…
}
```

61

Erzeugung von Event Listenern

- Event Listener Objekte müssen instanziiert werden

```
public class myActionListenerClass implements ActionListener {
   …
}
…
MyActionListenerClass myActionListener
                              = new MyActionListenerClass();
…
someComponent.addActionListener(myActionListener);
```

62

Registrierung von Event Listenern

- Event Listener Objekte müssen bei den entsprechenden Komponenten registriert werden

```
public class MyActionListenerClass implements ActionListener {
   …
}
…
MyActionListenerClass myActionListener
                              = new MyActionListenerClass();
…
someComponent.addActionListener(myActionListener);
```

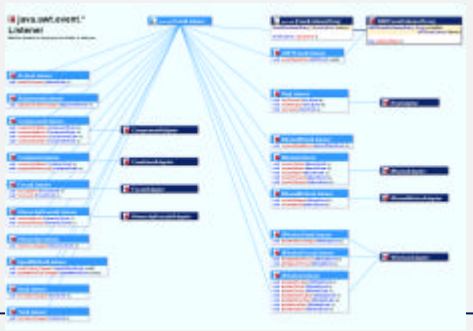
63

Verfügbare Listener Interfaces (Auszug)

- Für jeden Ereignistyp existiert ein eigenes Interface.
  - **`ActionListener`**: Benutzer klickt einen Button an, drückt die Enter-Taste in einem Textfeld oder wählt einen Menüeintrag aus
  - **`WindowListener`**: Benutzer schliesst ein Fenster
  - **`MouseListener`**: Benutzer klickt eine Maustaste während der Mauszeiger über einer Komponente ist
  - **`MouseMotionListener`**: Benutzer bewegt die Maustaste über einer Komponente
  - **`ComponentListener`**: Komponente wird sichtbar
  - **`FocusListener`**: Komponente bekommt den Focus

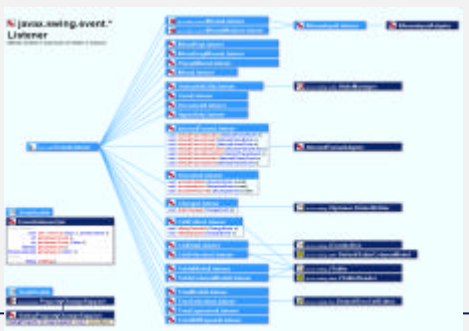
64

Verfügbare Listener Interfaces AWT



65

Verfügbare Listener Interfaces Swing



66

Event Listener als anonyme Klassen

- Implementierung von Event Handlern oft als interne (anonyme) Klassen

```
public Component createComponents() {
    ...
    button.setMnemonic(KeyEvent.VK_I);
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
          numClicks++;
          label.setText(labelPrefix + numClicks);
      }
    });
```


Ereignis Adapter …

- Einigen Listener Interfaces beinhalten mehrere Event Handler Methoden
  - `MouseListener: mousePressed, mouseReleased, mouseEntered, mouseExited, mouseClicked`
- Auch wenn man nicht an allen Ereignissen interessiert ist, müssen in einer Event Listener Klasse alle diese Methoden implementiert sein (ggf. mit leerem Rumpf)

  → *schwer lesbarer Code*

… Ereignis Adapter …

- Beispiel: MyClass ist nur an Mausklicks interessiert

```
public class MyClass implements MouseListener {
    ...
someObject.addMouseListener(this);
    ...
    /* Empty method definitions. */
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}

    public void mouseClicked(MouseEvent e) {
...//Event handler implementation goes here...
    }
}
```


… Ereignis Adapter …

- Für jedes Event Listener Interface mit mehr als einer Event Handler Methode gibt es eine Adapterklasse mit leeren Methodenrümpfen
- Um einen Adapter zu benutzen erweitert man die jeweilige Adapterklasse (anstatt das entsprechende Interface zu implementieren) und überschreibt die relevanten Event Handler

… Ereignis Adapter

- Beispiel: MyClass ist nur an Mausklicks interessiert

```
public class MyClass extends MouseAdapter {
    ...
    someObject.addMouseListener(this);
    ...

    public void mouseClicked(MouseEvent e) {
    ... //Event Handler Implementierung ...
    }
}
```


Ereignis Adapter als anonyme Klassen

- Kombination der beiden Techniken möglich

```
...
someObject.addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent e) {
    ...// Event Handler Implementierung ...
        }
});
...
```

Aktionen

- Ein **Action** Objekt ist ein **ActionListener**
- Zusätzlich kann es den Zustand von **ActionEvent** Sources **JToolBar, JButton**, etc. verwalten
- Ein **Action** Objekt wird mit der Methode **setAction()** zu einer Komponente hinzugefügt
- Der Zustand der Komponente wird dem Zustand des **Action** Objektes angeglichen
- Das Action Objekt wird als **ActionListener** bei der Komponente registriert
- Wenn sich der Zustand des **Action** Objektes ändert, wird der Zustand der Komponente ebenfalls geändert

73

Erzeugen von Aktionen

- Um ein **Action** Objekt zu erzeugen, instanziiert man eine Unterklasse der Klasse **AbstractAction**
- Die Unterklasse muss die Methode **actionPerformed()** implementieren, da das **Action** Objekt auch als **ActionListener** fungiert

```
class LeftAction extends AbstractAction {
   ...
   public void actionPerformed(ActionEvent e) { ... }
   ...
}
...
Action leftAction = new LeftAction();
...
```

74

Beispiel: Koppeln von GUI Komponenten

- Ein Button in einem Toolbar und ein Menüeintrag sollen die gleiche Funktionalität haben

```
class LeftAction extends AbstractAction { ... }
...
Action leftAction = new LeftAction();
...
button   = new JButton(leftAction);
menuItem = new JMenuItem(leftAction);
...
```

75

Ereignisbehandlung und Threads

- Die Ereignisbehandlung erfolgt immer im gleichen Thread, dem **Event Dispatching Thread**
  - stellt sicher, dass ein Event Handler terminiert, bevor ein anderer aufgerufen wird
- Auch die Zeichenroutinen der Komponenten laufen im Event Dispatching Thread
  - während ein Event Handler abgearbeitet wird ist die GUI „eingefroren“ (kann sich nichtaktualisieren und reagiert nicht auf Benutzereingaben)
- Event Handler dürfen (sollen) nur wenig Rechenzeit verbrauchen
  - Langfristige Berechnungen müssen asynchron in einem extra Thread ausgeführt werden

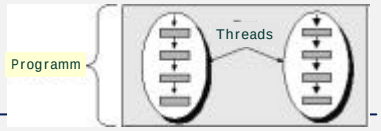
76

Threads

77

Was sind Threads?

- leichtgewichtige Prozesse
  - teilen sich Ressourcen des Programms
  - sequentieller Kontrollfluss innerhalb eines Threads
- ein Programm besteht aus mehreren Threads
  - laufen parallel
  - erledigen verschiedene Aufgaben
  - jeder Thread hat seinen eigenen Kontext
    - Programmzähler
    - Stack



78

Wofür kann man Threads verwenden?

- rechenintensive GUI Initialisierung, z.B. Laden von Bildern, etc. → GUI startet schneller
- rechenintensive Prozesse aus dem Event Dispatching Thread auslagern → GUI bleibt reaktiv
- wiederholte Ausführung des gleichen Prozesses in regelmässigen Abständen
- warten auf Nachrichten/Daten anderer Programme

79

Erzeugen von Threads

- Low-Level API
  - erweitern der Klasse `java.lang.Thread` und überschreiben der `run()` Methode oder
  - implementieren des Interface `java.lang.Runnable`, (damit ibs. implementieren einer `run()` Methode)
  - anschliessend: Instanziieren der entsprechenden Klasse und Aufruf der `start()` Methode an der Instanz
- High-Level API
  - `java.util.Timer`
    - führt eine Aufgabe wiederholt oder nach einer vorgegebenen Verzögerung aus
    - der entsprechende Code wird im Thread der Instanz eines `java.util.TimerTask` Objekts ausgeführt

80

Low-Level API: `java.lang.Thread`

- die Klasse `java.lang.Thread` bildet die Grundlage für Threads in Java
  - stellt ein Thread-API zur Verfügung
  - definiert elementare Eigenschaften/elementares Verhalten von Threads
    - starten, schlafen, aktiv werden, ...
    - Priorität haben, ...
  - implementieren eines Threads mit Hilfe der Klasse `Thread` durch erweitern der Klasse und überschreiben ihrer `run()` Methode mit dem Code der von dem Thread ausgeführt werden soll
  - starten des Threads durch Aufruf der `start()` Methode an einer Instanz der erweiterten Klasse

81

Low-Level API: `java.lang.Runnable`

- implementieren des Interfaces `java.lang.Runnable` mit einer Klasse; damit ibs. implementieren einer `run()` Methode mit dem Code der von dem Thread ausgeführt werden soll
- übergeben einer Instanz dieser Klasse beim Instanziieren eines Thread Objekts

```
public class Clock extends Applet implements Runnable {
    private Thread clockThread;
    public void start() {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
    public void run() { ... }
    ...
}
```

82

High-Level API: Erzeugen von Threads

- erweitern der Klasse `java.util.TimerTask` und überschreiben den `run()` Methode mit dem gewünschten Code
- erzeugen eines Threads durch instanziieren eines `java.util.Timer` Objektes
- instanziieren der erweiterten `TimerTask` Klasse
- anmelden dieser Instanz beim `Timer` Objekt mittels `schedule`
  - Angabe der anfänglichen Verzögerung
  - Angabe der Wiederholungsrate

83

Timer Threads: Code verzögert ausführen

```
public class Reminder {
    Timer timer;
    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(),
                       seconds*1000 // delay
                       );
    }
    class RemindTask extends TimerTask {
        public void run() {
            System.out.println("Time's up!");
            timer.cancel(); //Terminate the timer thread
        }
    }
    public static void main(String args[]) {
        new Reminder(5);
    }
}
```

84

Timer Threads: Code wiederholt ausführen

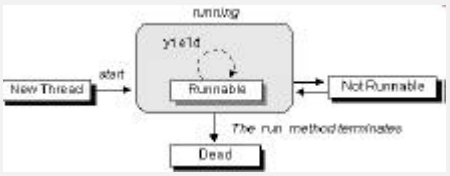
```
public class Reminder {
    Timer timer;
    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(),
                       seconds*1000 // initial delay
                       1000         // repeat
                       );
    }
    ...
```

High-Level API: Anhalten von Threads

- ein Programm läuft solange Timer Threads aktiv sind
- Abbrechen eines Timer Threads durch
  - Aufruf der `cancel()` Methode des Timer Objekts
  - erzeugen des Timer Threads als Daemon mittels `new Timer(true);`
    - wenn nur noch Daemons-Threads aktiv sind, terminiert ein Programm
  - aufrufen von `System.exit()` beendet das Programm und alle seine Threads

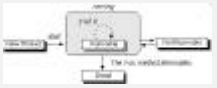
Der Lebenszyklus eines Threads

- ein Thread kann in einem der folgenden Zustände sein
  - ***neu*** (New Thread)
  - ***lauffähig*** (Runnable)
  - ***nicht lauffähig*** (Not runnable)
  - ***tot*** (Dead)



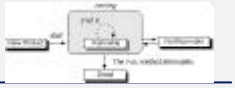
Neue Threads

- unmittelbar nachdem ein Thread Objekt instanziiert wurde ist der assoziierte Thread im Zustand ***neu***
- für diesen Thread wurden noch keine Systemressourcen angefordert
- nur die `start()` Methode des Objektes kann aufgerufen werden.
- alle anderen Methoden liefern eine `IllegalThreadStateException`



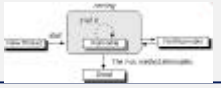
Lauffähige Threads

- wird die `start()` Methode eines Thread-Objekts aufgerufen, so geschieht folgendes:
  - anfordern von Systemressourcen für den Thread
  - anmelden des Threads beim Scheduler des JRE
  - aufrufen der `run()` Methode des Threads
- nach der Rückkehr der `start()` Methode ist der Thread im Zustand ***lauffähig***
  - der Thread wartet nun darauf, vom Scheduler des Java Runtime Systems die CPU zugeteilt zu bekommen (lauffähig ≠ laufend)



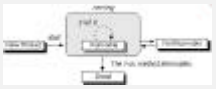
Nicht lauffähige Threads ...

- ein Thread wechselt in den Zustand ***nicht lauffähig***, falls
  - seine `sleep()` Methode aufgerufen wird
  - er die `wait()` Methode aufruft, um auf eine Bedingung zu warten
  - bei einer I/O-Operation blockiert wird
- ein ***nicht lauffähiger*** Thread erhält vom Scheduler des JRE keine Rechenzeit zugeteilt



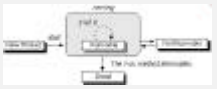
… Nicht lauffähige Threads

- damit ein Thread wieder in den Zustand ***lauffähig*** wechseln kann, müssen Bedingungen erfüllt sein, die davon abhängen, warum er in den Zustand ***nicht lauffähig*** gewechselt hat
  - wenn seine `sleep()` Methode aufgerufen wurde, muss die angegebene Zeit verstrichen sein
  - wenn er mittels `wait()` auf eine Bedingung wartet, muss ihn ein anderes Objekt mittels `notify()` oder `notifyAll()` benachrichtigen
  - wenn er bei einer I/O-Operation blockiert wurde, muss diese Operation abgeschlossen sein



Tote Threads

- ein Thread wechselt in den Zustand ***tot***, wenn seine `run()` Methode terminiert
- wenn ein Thread im Zustand ***tot*** ist,
  - wird er wird beim Scheduler des JRE abgemeldet
  - werden seine Systemressourcen wieder freigegeben



Abfrage des Threadzustands

- der aktuelle Zustand eines Threads kann mit der Methode `isAlive()` aus der Klasse `Thread` festgestellt werden
  - `isAlive()==true`
    - der Thread wurde gestartet und ist nicht tot, ist also entweder im Zustand ***lauffähig*** oder ***nicht lauffähig***
  - `isAlive()==false`
    - der Thread ist entweder im Zustand ***neu*** oder ***tot***
  - man kann damit nicht zwischen den Zuständen ***lauffähig*** oder ***nicht lauffähig*** bzw. ***neu*** oder ***tot*** unterscheiden

Scheduling

- alle Threads im System müssen sich eine CPU teilen
  - Verteilen der Threads auf die CPU nennt man Scheduling
- in Java: preemptives Scheduling mit fester Priorität
  - jeder Thread hat eine feste, ganzzahlige Priorität zwischen `MIN_PRIORITY` und `MAX_PRIORITY` (Konstanten aus der Klasse `Thread`)
  - wenn mehrere Threads im Zustand ***lauffähig*** sind, so wird die CPU dem Thread mit der höchsten Priorität zugeteilt
  - preemptives Scheduling: wenn ein Thread, dessen Priorität höher ist als die des CPU-Threads, in den Zustand ***lauffähig*** übergeht, so verdrängt er den CPU-Thread
  - Threads mit der gleichen Priorität verdrängen sich nicht (kein „Time-slicing“)
  - der CPU-Thread kann die CPU freiwillig an einen Thread gleicher Priorität abgeben (mittels `yield()`)
  - haben alle Threads im System die gleiche Priorität, wählt der Scheduler den CPU-Thread in einem Round-Robin Verfahren

Synchronisation von Threads …

- Threads dürfen nicht gleichzeitig auf gemeinsam genutzte Ressourcen zugreifen (→ Inkonsistenzen möglich)
  - Codeabschnitte in denen auf dasselbe Objekt von verschiedenen Threads zugegriffen wird, heissen kritisch
  - Blöcke oder Methoden können kritisch sein; sie werden mit dem Schlüsselwort `synchronized` gekennzeichnet
  - für jedes Objekt mit `synchronized` Methoden existiert ein Lock
  - wird eine `synchronized` Methode eines Objektes ausgeführt, so sperrt der ausführende Thread das Lock des Objektes
  - wenn die Methode terminiert, so löscht der ausführende Thread das Lock des Objektes
  - wenn ein anderer Thread eine `synchronized` Methode eines gesperrten Objekts aufruft, wird er blockiert
  - Setzen und Löschen von Locks geschieht atomar im JRE

… Synchronisation von Threads …

- Threads müssen ihre gemeinsame Arbeit koordinieren
  - ein Thread kann mit einem Aufruf der `wait()` Methode in einem kritischen Codeabschnitt
    - das Lock des von ihm gesperrten Objekts löschen
    - sich daraufhin blockieren
  - ein Thread kann mit einem Aufruf der `notify()` Methode in einem kritischen Codeabschnitt
    - das Lock des von ihm gesperrten Objekts löschen
    - einen beliebigen Thread der auf das Lock dieses Objekts wartet aufwecken

… Synchronisation von Threads: Beispiel …

```
public class Application  {

   public static void main(String[] args) {

        Fork   singleFork = new Fork();

        Philosopher plato = new Philosopher(singleFork);
        plato.start();

        Philosopher zeno  = new Philosopher(singleFork);
        zeno.start();
}
```


Synchronisation von Threads: Beispiel …

```
public class Philosopher extends Thread {
   private Fork fork;

   public Philisopher(Fork fork) { this.fork = fork; }

   private void think() { ... }
   private void eat() { ... }

   public void run() {
       while (true) {
             think(); fork.get(); eat(); fork.put();
   }
}
...
```


… Synchronisation von Threads: Beispiel

```
public class Fork  {
   private boolean isInUse = false;

   public synchronized void get() {
        while ( isInUse ) {
                try { wait(); } catch ( InterruptedException e ) {}
        }
        isInUse = true;
   }
   public synchronized void put() {
        isInUse = false;
        notifyAll();
   }
}
```


Probleme bei der Threadsynchronisation

- ein System heisst fair, wenn jeder Thread die Ressourcen bekommt, die er braucht; in fairen Systemen gibt es keine
  - Startvation : es gibt Threads die permanent blockiert sind, da eine gemeinsam genutzte Ressource von anderen Threads nicht freigegeben wird
  - Deadlocks : Threads warten gegenseitig auf Ressourcen die jeweils der andere blockiert

Threads in Swing

- Swing-Komponenten sind nicht thread-sicher; d.h., Zugriff auf Komponenten von mehreren Threads aus nicht möglich ohne besondere Synchronisationsmaßnahmen
- es existiert ein Thread für Ereignisbehandlung, das Neuzeichnen und den Zugriff auf die GUI Komponenten: Event Dispatching Thread
- Single-Thread Rule: Sobald eine Swing-Komponente realisiert ist, muss sämtlicher Code, der den Zustand der Komponente abfragt oder ändert, im Event Dispatching Thread ausgeführt werden
  (die Methoden `pack()` und `setVisible(true)` ***realisieren*** ein top-level Fenster und alle seine Komponenten)

Swing und Threadprogrammierung

- `javax.swing.Timer` Objekte
  - erzeugen Threads in denen Aufgaben wiederholt oder nach einer vorgegebenen Verzögerung ausgeführt werden können
- `SwingWorker` Objekte
  - erzeugen einen Thread, in dem rechenintensive Aufgaben ausgeführt werden können
  - nach Abarbeitung des Threads kann zusätzlicher Code im Event Dispatching Thread ausgeführt werden
- Packet `SwingUtilities`
  - `invokeLater`() und `invokeAndWait`() können Code im Event Dispatching Thread ausführen
  - `isEventDispatchingThread`() liefert true wenn der Code im Event Dispatching Thread ausgeführt wird

Klasse `javax.swing.Timer`

- ein **`javax.swing.Timer`** Objekt löst nach einer angegebenen Verzögerung einen (oder mehrere) **`ActionEvents`** aus
- der Timer kann so eingestellt werden, dass er die Ereignisse in regelmässigen Intervallen erzeugt
- der entsprechende Code wird *im Event Dispatching Thread* ausgeführt

Animation mittels `Timer` …

- die Komponente die die Animation darstellen soll muss das Interface **`ActionListener`** implementieren, um auf **`ActionEvents`** reagieren zu können, die von einem **`Timer`** Objekt erzeugt werden

```
public class Animation extends JFrame
                       implements ActionListener {
    ...
    Animation() {
        ...
    }
}
```


… Animation mittels `Timer` …

- das Interface **`ActionListener`** beinhaltet die Methode **`actionPerformed`** als Event Handler für **`ActionEvents`**

```
public class Animation extends JFrame
                       implements ActionListener {
    ...
    public void actionPerformed(ActionEvent e) {
       ...
    }
}
```


… Animation mittels `Timer` …

- die Komponente die die Animation darstellen soll hat ein **`Timer`** Objekt **`timer`**
- die Variable **`frame`** gibt an, wie oft der Timer aufgerufen wurde
- ihr Wert wird im Label **`label`** angezeigt

```
public class Animation extends JFrame
                       implements ActionListener {
    int    frame;
    Timer  timer;
    JLabel label;
    ...
    Animation() {
        ...
    }
}
```


… Animation mittels `Timer` …

- im Konstruktor der Komponente wird ein **`Timer`** Objekt instanziiert.
- das erste Argument des Konstruktors gibt an, in welchem Abstand (in ms) die Ereignisse erzeugt werden.
- das erste Argument des Konstruktors gibt an, an welches Objekt die Ereignisse geschickt werden

```
public class Animation extends JFrame
                       implements ActionListener {
    ...
    Animation() {
        ...
        timer = new Timer(100, this);
        ...
    }
}
```


… Animation mittels `Timer` …

- im Hauptprogramm wird die Komponente erzeugt und realisiert
- dann wird der Timer gestartet

```
public class Animation extends JFrame
                       implements ActionListener {
    ...
    public static void main(String args[]) {
        animation = new Animation();
        animation.pack();
        animation.setVisible(true);
        animation.timer.start();
    }
}
```

… Animation mittels Timer

- nach Ablauf der Verzögerung wird vom Timer ein **ActionEvent** an die bei ihm registrierten Event Listener geschickt, also die Methode **actionPerformed()** der Komponente aufgerufen
- diese Methode zählt die Variable **frame** hoch und passt die Beschriftung von **label** an

```
public class Animation extends JFrame
                        implements ActionListener {
    ...
    public void actionPerformed(ActionEvent e) {
       frame++;
       label.setText("Frame " + frame);
    }

}
```


Klasse SwingWorker

- beim Instanziieren einer von **SwingWorker** abgeleiteten Klasse wird ein Thread erzeugt
- dieser Thread wird mit der Methode **start()** gestartet
- die (überschriebene) Methode **construct()** enthält den Code der in dem Thread ausgeführt wird. Diese Methode gibt ein **Object** zurück
- auf das Objekt kann mit der Methode **get()** zugegeriffen werden (Vorsicht: der Aufruf blockiert)
- der Thread kann mit der Methode **interrupt()** angehalten werden (damit **get()** nicht blockiert)
- die (überschriebene) Methode **finished()** wird *im Event Dispatching Thread* aufgerufen nachdem **construct()** beendet ist

Bilder laden mit SwingWorker …

- der Event Handler **actionPerformed()** muss eine Aufgabe ausführen, die längere Zeit in Anspruch nehmen kann
- diese Aufgabe soll von einem separaten Thread ausgeführt werden, damit die GUI nicht blockiert wird
- der Thread wird beim Instanziieren der Klasse **SwingWorker** erzeugt

```
public void actionPerformed(ActionEvent e) {
  ...
  loadImage(imagedir + filename); // kann länger dauern
  ...
}
...
private void loadImage(final String img) {
    final SwingWorker worker = new SwingWorker() {...};
    worker.start();
}
```


… Bilder laden mit SwingWorker …

- die Methode **construct()** enthält den Code der ein Icon lädt
- im Beispiel wird der Rückgabewert dieser Methode nicht verwendet
- der Thread in dem **construct()** läuft wird mit der Methode **start()** gestartet

```
private void loadImage(final String img) {
    final SwingWorker worker = new SwingWorker() {
        ImageIcon icon = null;
        public Object construct() {
            icon = new ImageIcon(getURL(img)); return null;
        }
        public void finished() {...}
    };
    worker.start();
}
```


… Bilder laden mit SwingWorker

- die Methode **finished()** aufgerufen nachdem **construct()** beendet ist
- diese Methode läuft im *Event Dispatching Thread*, daher können GUI Komponenten gefahrlos manipuliert werden

```
private void loadImage(final String img) {
    final SwingWorker worker = new SwingWorker() {
        ImageIcon icon = null;
        public Object construct() {...}
        public void finished() { button.setIcon(icon); }
    };
    worker.start();
}
```


SwingUtilities.invokeLater()

- kann von jedem Thread aufgerufen werden, um Code im Event Dispatching Thread auszuführen
- der Code muss in der **run()** Methode eines **Runnable()** Objektes enthalten sein
- dieses **Runnable()** Objekt wird als Argument an **invokeLater()** übergeben

```
...
Runnable updateComponent = new Runnable() {
   public void run() { component.doSomething(); }
};
SwingUtilities.invokeLater(updateComponent);
...
```

SwingUtilities.invokeAndWait()

- ähnlich zu **invokeLater()**
- *Unterschied:* Der Funktionsaufruf kehrt erst zurück, wenn der übergebene Code im Event Dispatching Thread ausgeführt wurde
- *Vorsicht:* Gefahr des Blockierens

Verschiedenes

Einstellen des "Look-and-Feel"

```
public class Hello extends JFrame implements ActionListener {
  public Hello () {
    super("Hello, Swing!");
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      UIManager.setLookAndFeel (
         UIManager.getCrossPlatformLookAndFeelClassName()
      );
    } catch (Exception Signal) { /* nop */ };
   JPanel Pane = new JPanel();
    ...
   };
   ...
};
```


Weiterführende Informationen

Literatur

- K.A. Mughal, R.W. Rasmussen: A Programmer's Guide to Java Certification. Addison-Wesley 2000.
- D. Flanagan: Java Foundation Classes in a Nutshell. O'Reilly 1999.
- H.M. Deitel, P.J. Deitel, S.E. Santry: Advanced Java 2 Platform—How to Program. Prentice Hall 2002.
- K. Topley: Core Swing Advanced Programming. Prentice Hall 2000.

Weitere Informationen

- Online-Dokumentation:
  http://java.sun.com/j2se/1.3/docs/api/
- Demos:
  http://java.sun.com/products/javawebstart/demos.html
- Tutorial:
  http://java.sun.com/docs/books/tutorial/uiswing/start/swingTour.html