

17 Graphische Benutzeroberflächen

Eine graphische Benutzer-Oberfläche (**GUI**) ist i.A. aus mehreren Komponenten zusammen gesetzt, die einen (hoffentlich) **intuitiven Dialog** mit der Benutzerin ermöglichen sollen.

Idee:

- ▶ Einzelne Komponenten bieten der Benutzerin Aktionen an.
- ▶ Ausführen der Aktionen erzeugt **Ereignisse**.
- ▶ Ereignisse werden an die dafür zuständigen Listener-Objekte weiter gereicht **Ereignis-basiertes Programmieren**.

Ereignisse

Ereignisse

- ▶ Maus-Bewegungen und -Klicks, Tastatureingaben etc. werden von der Peripherie registriert und an das ↑Betriebssystem weitergeleitet.
- ▶ Das Java-Laufzeitsystem nimmt die Signale vom Betriebssystem entgegen und erzeugt dafür AWTEvent-Objekte.
- ▶ Diese Objekte werden in eine AWTEventQueue eingetragen Producer!
- ▶ Die Ereignisschlange verwaltet die Ereignisse in der Reihenfolge, in der sie entstanden sind, kann aber auch mehrere ähnliche Ereignisse zusammenfassen. . .
- ▶ Der AWTEvent-Dispatcher ist ein weiterer Thread, der die Ereignis-Schlange abarbeitet Consumer!

- ▶ Abarbeiten eines Ereignisses bedeutet:
 1. Weiterleiten des **AWTEvent**-Objekts an das Listener-Objekt, das vorher zur Bearbeitung solcher Ereignisse **angemeldet** wurde;
 2. Aufrufen einer speziellen Methode des Listener-Objekts.
- ▶ Die Objekt-Methode des Listener-Objekts hat für die Reaktion des Applets zu sorgen.

GUI-Frameworks

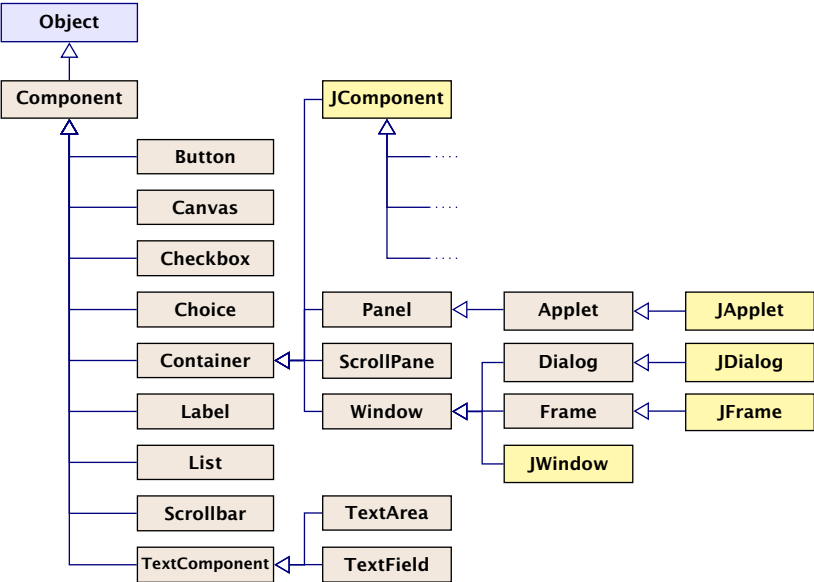
AWT, **A**bstr**W**indowing **T**oolkit.

- ▶ nutzt GUI-Elemente des Betriebssystems
- ▶ gut für Effizienz
- ▶ Anwendungen sehen auf verschiedenen Systemen unterschiedlich aus (kann Vorteil aber auch Nachteil sein)
- ▶ unterstützt üblicherweise nur Elemente die auf den meisten Systemen verfügbar sind
- ▶ funktioniert mit Applets

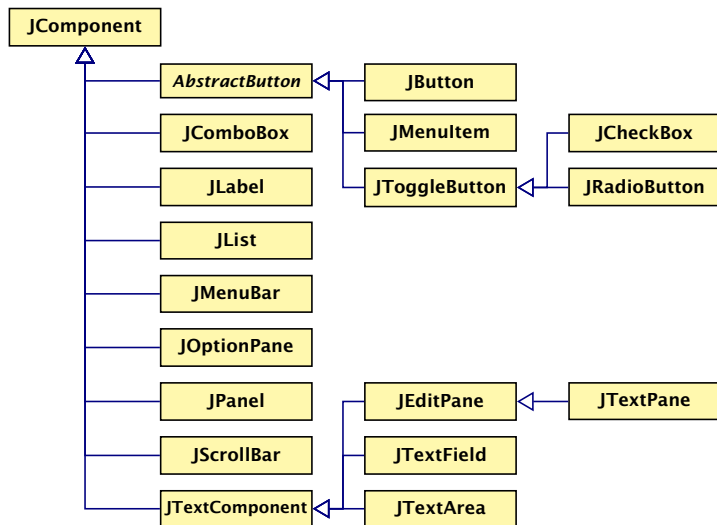
Swing

- ▶ fast alle GUI-Elemente sind in Java implementiert
- ▶ Anwendungen sehen überall gleich aus; (aber **skinnable**)
- ▶ reichhaltigere Sammlung von Elementen

Elemente in AWT und Swing



Elemente in AWT und Swing



Ein Button

```
1 import java.awt.*; import java.awt.event.*;
2 import javax.swing.*;
3 public class FirstButton extends JFrame implements
4                               ActionListener {
5     JLabel label;
6     JButton button;
7     public FirstButton() {
8         setLayout(new FlowLayout());
9         setSize(500,100);
10        setVisible(true);
11        setFont(new Font("SansSerif", Font.BOLD, 18));
12        label = new JLabel();
13        label.setText("This is my first button :-)");
14        add(label);
15        button = new JButton("Knopf");
16        button.addActionListener(this);
17        add(button);
18        revalidate();
19    }
```

"FirstButton.java"

Ein Button

```
20     public void actionPerformed(ActionEvent e) {
21         label.setText("Damn - you pressed it ...");
22         System.out.println(e);
23         remove(button);
24         // layout manager recalculates positions
25         revalidate();
26         repaint();
27     }
28     static class MyRunnable implements Runnable {
29         public void run() {
30             new FirstButton();
31         }
32     }
33     public static void main(String args[]) {
34         SwingUtilities.invokeLater(new MyRunnable());
35     }
36 } // end of FirstButton
```

"FirstButton.java"

Erläuterungen

- ▶ Wir erzeugen einen `JFrame`; ein normales Fenster mit Menüleiste, etc.
- ▶ Wir setzen Größe (`setSize`) des Frames, und machen ihn sichtbar (`setVisible`).
- ▶ `setLayout` kommt später...
- ▶ Der Frame enthält zwei weitere Komponenten:
 - ▶ ein `JButton`
 - ▶ ein `JLabel`
- ▶ Objekte dieser Klassen besitzen eine Aufschrift...
- ▶ Die in den Labels verwendete Schriftart richtet sich nach der des umgebenden `Containers` (zumindest in der Größe); deshalb wählen wir eine Schrift für den Frame

Erläuterungen

- ▶ Die Objekt-Methoden:

```
void add(Component c)
```

```
void add(Component c, int i)
```

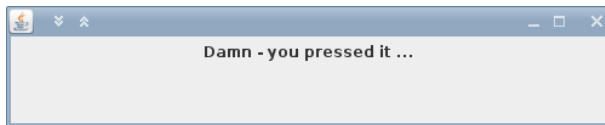
... fügen die Komponente `c` zum `Container JFrame` hinten (bzw. an der Stelle `i`) hinzu.

- ▶ `public void addActionListener(ActionListener listener)` registriert ein Objekt `listener` als das, welches die von der Komponente ausgelösten `ActionEvent`-Objekte behandelt, hier: der `JFrame` selber.
- ▶ `ActionListener` ist ein `Interface`. Für die Implementierung muss die Methode `void actionPerformed(ActionEvent e)` bereitgestellt werden.

Erläuterungen

- ▶ Die Methode `actionPerformed(ActionEvent e)` ersetzt den Text des Labels und entfernt den Knopf mithilfe der Methode `remove(Component c)`; anschließend muss der `Container` validiert und ggf. neu gezeichnet werden.
- ▶ Beim Drücken des Knopfs passiert das Folgende:
 1. ein `ActionEvent`-Objekt `action` wird erzeugt und in die Ereignisschlange eingefügt.
 2. Der `AWTEvent`-Dispatcher holt `action` wieder aus der Schlange. Er identifiziert den Frame `f` selbst als das für `action` zuständige Listener-Objekt. Darum ruft er `f.actionPerformed(action)`; auf.
- ▶ Wären `mehrere` Objekte als `Listener` registriert worden, würden sukzessive auch für diese entsprechende Aufrufe abgearbeitet werden.

Ein Button



Mehrere Knöpfe

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 public class SeveralButtons extends JFrame implements
6                               ActionListener {
7     JLabel label;
8     JButton butA, butB;
```

"SeveralButtons.java"

Mehrere Knöpfe

```
9      public SeveralButtons() {
10         setLayout(new FlowLayout());
11         setSize(500,100);
12         setVisible(true);
13         setFont(new Font("SansSerif", Font.BOLD, 18));
14         label = new JLabel();
15         label.setText("Press key...");
16         add(label);
17         butA = new JButton("Knopf A");
18         butA.setActionCommand("1");
19         butA.addActionListener(this);
20         add(butA);
21         butB = new JButton("Knopf B");
22         butB.setActionCommand("2");
23         butB.addActionListener(this);
24         add(butB);
25     }
```

Der Default-Wert für ActionCommand in diesem Fall ist der Text des Buttons (d.h. "Knopf A" bzw. "Knopf B"), d.h. wir könnten eine Fallunterscheidung durchführen ohne explizit mit `setActionCommand()` unterschiedliche Werte für die Buttons zu setzen.

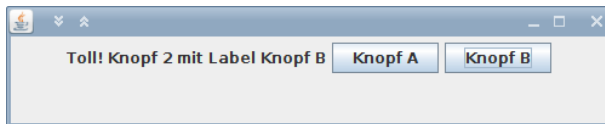
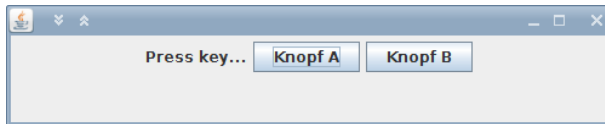
"SeveralButtons.java"

Mehrere Knöpfe

```
26     public void actionPerformed(ActionEvent e) {
27         if (e.getActionCommand().equals("1")) {
28             label.setText("Toll! Knopf 1 mit Label "+
29                 ((JButton)e.getSource()).getText());
30         } else {
31             label.setText("Toll! Knopf 2 mit Label "+
32                 ((JButton)e.getSource()).getText());
33         }
34         System.out.println(e);
35     }
36     static class MyRunnable implements Runnable {
37         public void run() {
38             new SeveralButtons();
39         }
40     }
41     public static void main(String args[]) {
42         SwingUtilities.invokeLater(new MyRunnable());
43     }
44 }
```

"SeveralButtons.java"

Mehrere Knöpfe



Alternativen

Wo kann man EventListener platzieren?

1. In der Klasse, die das Widget enthält (wie bei uns).
 - ▶ Widgets teilen sich Eventfunktionen (z.B. `ActionPerformed()`). Fallunterscheidung notwendig.
 - ▶ Die Widgets sind nicht von der Ereignisverarbeitung getrennt.
2. In einer/mehreren anderen Klasse.
 - ▶ Trennung von Ereignisverarbeitung und graphischen Elementen.
 - ▶ Bei einer Klasse Fallunterscheidungen erforderlich; mehrere Klassen führen evt. zu sehr viel Code
 - ▶ Zugriffe auf private Elemente?
3. Inner Class
4. Anonymous Inner Class

Inner Class

```
1 public class OuterClass {  
2     private int var;  
3     public class InnerClass {  
4         void methodA() {};  
5     }  
6     public void methodB() {};  
7 }
```

- ▶ Instanz von **InnerClass** kann auf alle Member von **OuterClass** zugreifen.
- ▶ Wenn **InnerClass** **static** deklariert wird, kann man nur auf statische Member zugreifen.
- ▶ Statische innere Klassen sind im Prinzip normale Klassen mit zusätzlichen Zugriffsrechten.
- ▶ Nichtstatische innere Klassen sind immer an eine konkrete Instanz der äußeren Klasse gebunden.

Beispiel – Zugriff von Außen

Um von außen ein Objekt der inneren Klasse zu erzeugen, muss man erst ein Objekt der äußeren Klasse generieren.

Dann erzeugt man ein Objekt der Klasse z.B. durch `objOuter.new InnerClass()`, wenn `InnerClass` einen Defaultkonstruktor hat.

```
1 class OuterClass {
2     private int x = 1;
3     public class InnerClass {
4         void show() {
5             System.out.println("x = " + x);
6         }
7     }
8     public void showMeth() {
9         InnerClass b = new InnerClass();
10        b.show();
11    } }
12 public class TestInner {
13     public static void main(String args[]) {
14         OuterClass a = new OuterClass();
15         OuterClass.InnerClass x = a.new InnerClass();
16         x.show();
17         a.showMeth();
18    } }
```

"TestInner.java"

Beispiel – Zugriff von Außen

Normalerweise erzeugt man keine Objekte einer inneren Klasse von außen. Stattdessen bietet häufig die äußere Klasse eine Funktion, die ein Objekt der inneren Klasse zur Verfügung stellt.

```
1 class OuterClass {
2     private static int x = 1;
3     public static class InnerClass {
4         void show() {
5             System.out.println("x = " + x);
6         } }
7     public void showMeth() {
8         InnerClass b = new InnerClass();
9         b.show();
10 } }
11 public class TestInnerStatic {
12     public static void main(String args[]) {
13         OuterClass a = new OuterClass();
14         OuterClass.InnerClass x =
15             new OuterClass.InnerClass();
16         x.show();
17         a.showMeth();
18 } }
```

Häufig (vgl. verkettete Liste mit innerer Klasse `ListElem`) werden innere Klassen aber auch nur zur Datenkapselung eingesetzt und sind dann privat.

Eine Anwendung von öffentlichen, inneren Klassen sind z.B. **Adapterklassen** (vgl. `Iterator`).

"TestInnerStatic.java"

Local Inner Class

Eine **lokale, innere Klasse** wird innerhalb einer Methode deklariert:

```
1 public class OuterClass {  
2     private int var;  
3     public void methodA() {  
4         class InnerClass {  
5             void methodB() {};  
6         }  
7     }  
8 }
```

- ▶ Kann zusätzlich auf die **finalen** Parameter und Variablen der Methode zugreifen.

Man kann nicht von außen auf die Klasse zugreifen. Deshalb machen modifier wie **private**, **public** keinen Sinn und sind auch nicht erlaubt.

Beispiel – Iterator

```
1 interface Iterator<T> {  
2     boolean hasNext();  
3     T next();  
4     void remove(); // optional  
5 }
```

- ▶ Ein Iterator erlaubt es über die Elemente einer Kollektion zu iterieren.
- ▶ Abstrahiert von der Implementierung der Kollektion.
- ▶ `hasNext()` testet, ob noch ein Element verfügbar ist.
- ▶ `next()` liefert das nächste Element (falls keins verfügbar ist wird eine `NoSuchElementException` geworfen).
- ▶ `remove()` entfernt das zuletzt über `next()` zugegriffene Element aus der Kollektion.

Falls die Kollektion das Entfernen von Elementen nicht erlaubt, bleibt `remove()` unimplementiert und liefert bei Aufruf eine Exception.

Beispiel – Iterator

`curr` zeigt auf das Element, das beim letzten Aufruf von `next()` zurückgegeben wurde.

```
1 public class TestIterator {
2     Integer[] arr;
3     TestIterator(int n) {
4         arr = new Integer[n];
5     }
6     public Iterator<Integer> iterator() {
7         class MyIterator implements Iterator<Integer> {
8             int curr = arr.length;
9             public boolean hasNext() { return curr>0;}
10            public Integer next() {
11                if (curr == 0)
12                    throw new NoSuchElementException();
13                return arr[--curr];
14            }
15        }
16        return new MyIterator();
17    }
}
```

"TestIterator.java"

Beispiel – Iterator

Anwendung des Iterators:

```
18     public static void main(String args[]) {
19         TestIterator t = new TestIterator(10);
20         Integer i = null;
21         for (Iterator<Integer> iter = t.iterator();
22             iter.hasNext(); i = iter.next()) {
23             System.out.println(i);
24         }
25     }
26 }
```

"TestIterator.java"

In diesem Fall wird nur 10 mal null ausgegeben...

Anonymous Inner Classes

Der Anwendungsfall für lokale, innere Klassen ist häufig:

- ▶ eine Methode erzeugt genau ein Objekt der inneren Klasse
- ▶ dieses wird z.B. an den Aufrufer zurückgegeben

Anonyme Innere Klasse:

- ▶ **Ausdruck** enthält Klassendeklaration, und instanziiert ein Objekt der Klasse
- ▶ man gibt **ein** Interface an, das implementiert wird, oder **eine** Klasse von der geerbt wird
- ▶ die Klasse selber erhält keinen Namen

Beispiel - Iterator

```
public Iterator<Integer> iterator() {  
    return new Iterator<Integer>() {  
        int curr = arr.length;  
        public boolean hasNext() { return curr>0;}  
        public Integer next() {  
            if (curr == 0)  
                throw new NoSuchElementException();  
            return arr[--curr];  
        }  
    };  
}
```

"IteratorAnonymous.java"

Mehrere Knöpfe – Andere Klasse(n)

```
1 import javax.swing.*;
2 import java.awt.*; import java.awt.event.*;
3
4 class ListenerA implements ActionListener {
5     JLabel label;
6     ListenerA(JLabel l) { label = l; }
7     public void actionPerformed(ActionEvent e) {
8         label.setText("To11! Knopf 1 mit Label "+
9             ((JButton)e.getSource()).getText());
10 } }
11 class ListenerB implements ActionListener {
12     JLabel label;
13     ListenerB(JLabel l) { label = l; }
14     public void actionPerformed(ActionEvent e) {
15         label.setText("To11! Knopf 2 mit Label "+
16             ((JButton)e.getSource()).getText());
17 } }
```

"SeveralButtonsOther.java"

Mehrere Knöpfe – Andere Klasse(n)

```
19 public class SeveralButtonsOther extends JFrame {
20     private JLabel label;
21     private JButton butA, butB;
22
23     public SeveralButtonsOther() {
24         setLayout(new FlowLayout());
25         setSize(500,100);
26         setVisible(true);
27         setFont(new Font("SansSerif", Font.BOLD, 18));
28         label = new JLabel();
29         label.setText("Press key...");
30         add(label);
31         butA = new JButton("Knopf A");
32         butA.addActionListener(new ListenerA(label));
33         add(butA);
34         butB = new JButton("Knopf B");
35         butB.addActionListener(new ListenerB(label));
36         add(butB);
37     }
```

"SeveralButtonsOther.java"

Mehrere Knöpfe – Andere Klasse(n)

```
38     public static void main(String args[]) {
39         SwingUtilities.invokeLater(
40             new Runnable() {
41                 public void run() {
42                     new SeveralButtonsOther();
43                 }
44             }
45         );
46     }
47 }
```

"SeveralButtonsOther.java"

Mehrere Knöpfe – Inner Class

```
1 import javax.swing.*;
2 import static javax.swing.SwingUtilities.*;
3 import java.awt.*; import java.awt.event.*;
4
5 public class SeveralButtonsInner extends JFrame {
6     private JLabel label;
7     private JButton butA, butB;
8     public class listenerA implements ActionListener {
9         public void actionPerformed(ActionEvent e) {
10             label.setText("Toll! Knopf 1 mit Label "+
11                 ((JButton)e.getSource()).getText());
12         } }
13     public class listenerB implements ActionListener {
14         public void actionPerformed(ActionEvent e) {
15             label.setText("Toll! Knopf 2 mit Label "+
16                 ((JButton)e.getSource()).getText());
17         } }
```

"SeveralButtonsInner.java"

Mehrere Knöpfe – Inner Class

```
18     public SeveralButtonsInner() {
19         setLayout(new FlowLayout());
20         setSize(500,100);
21         setVisible(true);
22         setFont(new Font("SansSerif", Font.
23             label = new JLabel();
24             label.setText("Press key...");
25         add(label);
26         butA = new JButton("Knopf A");
27         butA.addActionListener(new listenerA());
28         add(butA);
29         butB = new JButton("Knopf B");
30         butB.addActionListener(new listenerB());
31         add(butB);
32     }
33     public static void main(String args[]) {
34         invokeLater(()->new SeveralButtonsInner());
35     }
```

Wenn man innere Klassen benutzt, hat man automatisch Zugriff auf die privaten Variablen der umgebenden Klasse. In der Variante 2 musste man extra Konstruktoren schreiben, damit die Listenerklassen eine Referenz auf `label` erhielten.

Nachteil bei inneren Klassen: Code für Layout und Ereignisverarbeitung ist nicht getrennt.

"SeveralButtonsInner.java"

Die Syntax in Zeile 34 wird später erklärt.

Mehrere Knöpfe – Anonymous Class

```
1 import javax.swing.*;
2 import java.awt.*; import java.awt.event.*;
3 import static javax.swing.SwingUtilities.*;
4 public class SeveralButtonsAnonymous extends JFrame {
5     JLabel label;
6     JButton butA, butB;
7     public static void main(String args[]) {
8         invokeLater(()->new SeveralButtonsAnonymous());
9     }
10    public SeveralButtonsAnonymous() {
11        setLayout(new FlowLayout());
12        setSize(500,100);
13        setVisible(true);
14        setFont(new Font("SansSerif", Font.BOLD, 18));
15        label = new JLabel();
16        label.setText("Press key...");
```

"SeveralButtonsAnonymous.java"

Mehrere Knöpfe – Anonymous Class

```
17     add(label);
18     butA = new JButton("Knopf A");
19     butA.addActionListener(new ActionListener() {
20         public void actionPerformed(ActionEvent e) {
21             label.setText("Toll! Knopf 1 mit Label "+
22                 ((JButton)e.getSource()).getText());
23         }
24     });
25     add(butA);
26     butB = new JButton("Knopf B");
27     butB.addActionListener(new ActionListener() {
28         public void actionPerformed(ActionEvent e) {
29             label.setText("Toll! Knopf 2 mit Label "+
30                 ((JButton)e.getSource()).getText());
31         }
32     });
33     add(butB);
34 } }
```

"SeveralButtonsAnonymous.java"

Diskussion

Für größere Projekte ist Variante 2 vorzuziehen, da sie kleinere Klassen erlaubt, und eine saubere Trennung zwischen Ereignisbehandlung und graphischer Ausgabe ermöglicht.

Der Umweg über Innere Klassen vermeidet Fallunterscheidungen aber macht den Code recht unübersichtlich.

Weitere Alternative: Lambda-Ausdrücke/Methodenreferenzen

Lambda-Ausdrücke

Ein **funktionales Interface** ist ein Interface, das **genau** eine Methode enthält.

```
interface Runnable {  
    void run();  
}
```

Ein **Lambda-Ausdruck** ist das Literal eines Objektes, das ein funktionales Interface implementiert. Z.B.:

Syntax:

- ▶ allgemein
(%Parameterliste) -> {...}
- ▶ nur **return**-statement/eine Anweisung (bei **void**-Funktion)
(%Parameterliste) -> %Ausdruck
- ▶ nur genau ein Parameter
a -> {...}

Die formalen Parameter eines Lambda-Ausdrucks, müssen keine Typbezeichner enthalten. Der notwendige Typ wird durch den Kontext bei der Verwendung des Lambda-Ausdrucks ermittelt.

Beispiele

Bei Namenskollisionen verhalten sich die beiden Codestücke leicht unterschiedlich. Insbesondere falls der Name `Foo` schon verwendet wird...

```
Runnable r = () -> {System.out.println("Hello!");};
```

ist (im Prinzip) äquivalent zu

```
class Foo implements Runnable {  
    void run() {  
        System.out.println("Hello!");  
    }  
}  
Runnable r = new Foo();
```

Beispiele

Der Konstruktor `Eval` bekommt ein input array `arr`, ein Ergebnisarray `res` derselben Größe, und ein Funktionsobjekt `f`. Für jeden Eintrag `arr[i]` wird die Funktion ausgewertet und das Ergebnis nach `res[i]` geschrieben.

```
1 interface Func<T> {
2     T func(int arg);
3 }
4 public class Eval<T> {
5     void eval(Func<T> f, int[] arr, T[] res) {
6         for (int i=0; i<arr.length; i++) {
7             res[i] = f.func(arr[i]);
8         }
9     }
10    public static void main(String args[]) {
11        int[] a = {1,2,3,4,5};
12        Integer[] b = new Integer[5];
13        new Eval<Integer>().eval(x->x*x, a, b);
14        for (int i=0; i<5; i++) {
15            System.out.print(b[i]+",");
16        }
17    }
18 }
```

Hier steht `x->x*x` für ein Funktionsobjekt, das ein `int` nach `Integer` abbildet. Falls man in `main`, `Integer` durch `Double` ersetzt, steht `x->x*x` für ein Funktionsobjekt, das ein `int` nach `Double` abbildet, d.h., der Typ ist vom Kontext abhängig. (Nach der Ersetzung liefert der Compiler einen Fehler, da `int` nicht nach `Double` konvertierbar ist)

Beispiel - Überladen

```
1 interface Func1 {
2     String func(String arg);
3 }
4 interface Func2 {
5     int func(int arg);
6 }
7 interface Func3 {
8     String func(int arg);
9 }
10 public class Test {
11     static void foo(Func1 f) { }
12     static void foo(Func2 f) { }
13     static void foo(Func3 f) { }
14     public static void main(String args[]) {
15         foo(x->x);
16     }
17 }
```

"TestLambda.java"

Beispiele

```
Interface Block<T> {  
    void apply(T t);  
}
```

```
Interface Function<T> {  
    T map (T t);  
}
```

```
Function<Block<String>> twice  
    = b -> t -> { b.apply(t); b.apply(t); }  
Block<String> print2  
    = twice.map(s -> {System.out.println(s);});  
print2.apply("hello");
```

```
final List<String> list = new ArrayList<>();  
Block<String> adder2  
    = twice.map(s -> {list.add(s);});  
adder2.apply("world");  
System.out.println(list);
```


Methodenreferenzen

An der Stelle, an der ein Lambda-Ausdruck möglich ist, kann man auch eine **Methodenreferenz** einer passenden Methode angeben.

Beispiel:

- ▶ Klasse **ClassA** verfügt über statische Methode **boolean less(int a, int b)**.
- ▶ Das **Funktionsinterface Iface** verlangt die Implementierung einer Funktion, die zwei **ints** nach **boolean** abbildet.
- ▶ Außerdem existiert Funktion **sort(int[] a, Iface x)**.
- ▶ Dann sortiert der Aufruf:

```
int[] arr = {5,8,7,2,11};  
sort(arr, ClassA::less);
```

gemäß der durch **less** vorgegebenen Ordnung.

Wenn man eine Objektmethode benutzen möchte müsste man **x::less** schreiben, wobei **x** ein Objekt der Klasse **ClassA** ist.

Vorausgesetzt die Funktion **sort** ist so implementiert, dass sie tatsächlich sortiert...

Mehrere Knöpfe – Lambda/Methodenreferenz

```
1 import javax.swing.*;
2 import static javax.swing.SwingUtil;
3 import java.awt.*; import java.awt
4 class EventHandler {
5     private JLabel label;
6     EventHandler(JLabel l) { label = l; }
7     public void actionA(ActionEvent e) {
8         label.setText("Toll! Knopf 1 mit Label
9             "+e.getActionCommand());
10    }
11    public void actionB(ActionEvent e) {
12        label.setText("Toll! Knopf 2 mit Label
13            "+e.getActionCommand());
14    }
15 }
16 public class SeveralButtonsLambda extends JFrame {
17     JLabel label;
18     JButton butA, butB;
19     EventHandler handler;
```

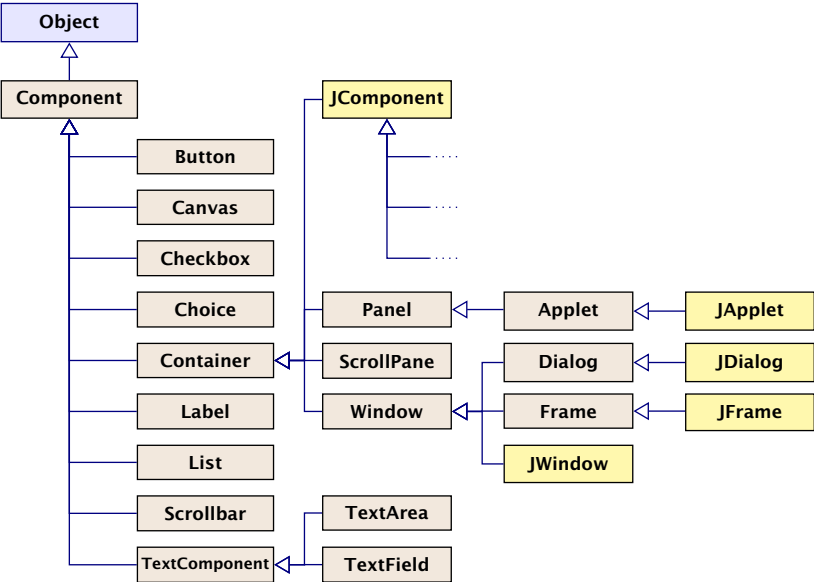
Wir sammeln die Funktionen für die Ereignisbehandlung in einer externen Klasse. Diese benötigt eine Referenz, um auf `label` zugreifen zu können. Diese übergeben wir im Konstruktor.

Mehrere Knöpfe – Lambda/Methodenreferenz

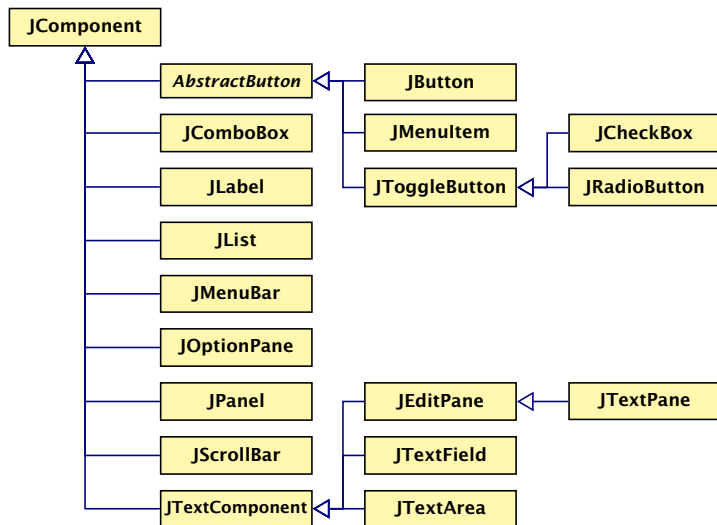
```
1 public static void main(String args[]) {
2     invokeLater(()->new SeveralButtonsLambda());
3 }
4 public SeveralButtonsLambda() {
5     setLayout(new FlowLayout());
6     setSize(500,100); setVisible(true);
7     setFont(new Font("SansSerif", Font.BOLD, 18));
8     JLabel = new JLabel();
9     JLabel.setText("Press key..."); add(JLabel);
10    handler = new EventHandler(JLabel);
11    JButton butA = new JButton("Knopf A");
12    butA.addActionListener(handler::actionA);
13    add(butA);
14    JButton butB = new JButton("Knopf B");
15    butB.addActionListener(
16        e -> JLabel.setText("Toll! Knopf 2 mit Label "
17            + e.getActionCommand()));
18    add(butB);
19 }
```

Das ist eigentlich recht nett, funktioniert aber nur für EventListener, die **Funktionsinterfaces** sind. Im Prinzip erzeugen wir das Listener-Objekt als Anonyme Klasse über eine sehr angenehme Syntax.

Elemente in AWT und Swing



Elemente in AWT und Swing



Elemente:

- (J)Label Zeigt eine Textzeile.
- (J)Button Einzelner Knopf um Aktion auszulösen.
- (J)Scrollbar Schieber zur Eingabe kleiner `int`-Zahlen.

Beispiel – Scrollbar

```
1 import javax.swing.*;
2 import java.awt.*; import java.awt.event.*;
3 public class ScalingFun extends JFrame
4         implements AdjustmentListener {
5     private JScrollbar scrH, scrW;
6     private JComponent content;
7     private Image image;
8     private int width, height;
9
10    class MyComponent extends JComponent {
11        MyComponent(int w, int h) {
12            setPreferredSize(new Dimension(w,h));
13        }
14        public void paintComponent(Graphics page) {
15            int l = getWidth()/2 - width/2;
16            int r = getHeight()/2 - height/2;
17            page.drawImage(image,l,r,width,height,null);
18        } }
```

"ScalingFun.java"

Beispiel – Scrollbar

```
ScalingFun() { // Konstruktor
    image = Toolkit.getDefaultToolkit().
                getImage("al-Chwarizmi.png");
    // wait for image to load...
    while (image.getHeight(null) == -1);
    int h = height = image.getHeight(null);
    int w = width = image.getWidth(null);
    setLayout(new BorderLayout());
    scrH=new JScrollBar(JScrollBar.VERTICAL,h,50,0,h+50);
    scrH.addAdjustmentListener(this);
    add(scrH,"West");
    scrW=new JScrollBar(JScrollBar.HORIZONTAL,w,50,0,w+50);
    scrW.addAdjustmentListener(this);
    add(scrW,"South");
    setVisible(true);
    add(content = new MyComponent(w,h));
    pack();
}
```

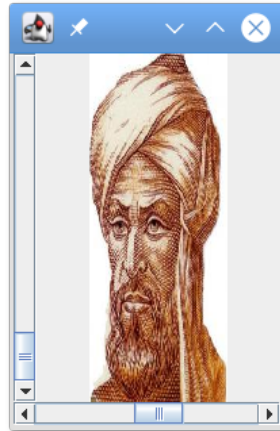
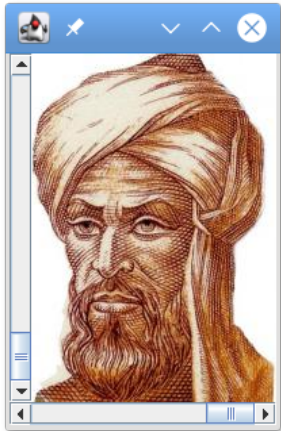
"ScalingFun.java"

Beispiel – Scrollbar

```
public void adjustmentValueChanged(AdjustmentEvent e) {
    Adjustable s = e.getAdjustable();
    int value = e.getValue();
    if (s == scrH) height = value;
    else width = value;
    revalidate();
    repaint();
}
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> new ScalingFun());
}
} // end of ScalingFun
```

"ScalingFun.java"

Scrollbar



Erläuterungen

- ▶ Ein `JScrollbar`-Objekt erzeugt `AdjustmentEvent`-Ereignisse.
- ▶ Entsprechende Listener-Objekte müssen das Interface `AdjustmentListener` implementieren.
- ▶ Dieses verlangt die Implementierung einer Methode `void adjustmentValueChanged(AdjustmentEvent e);`
- ▶ Der Konstruktor legt zwei `JScrollbar`-Objekte an, eines horizontal, eines vertikal.

Dafür gibt es in der Klasse `JScrollbar` die `int`-Konstanten `HORIZONTAL` und `VERTICAL`.

Erläuterungen

- ▶ Der Konstruktor `JScrollbar(int dir, int init, int slide, int min, int max);` erzeugt eine `JScrollbar` der Ausrichtung `dir` mit Anfangsstellung `init`, Breite des Schiebers `slide`, minimalem Wert `min` und maximalem Wert `max`.

Aufgrund der Breite des Schiebers ist der **wirkliche** Maximalwert `max - slide`.

- ▶ `void addAdjustmentListener(AdjustmentListener adj);` registriert das `AdjustmentListener`-Objekt als Listener für die `AdjustmentEvent`-Objekte der Scrollbars.

- ▶ Um `AdjustmentEvent`-Objekte behandeln zu können, implementieren wir die Methode

```
AdjustmentValueChanged(AdjustmentEvent e);
```

- ▶ Jedes `AdjustmentEvent`-Objekt verfügt über die Objekt-Methoden:

```
public AdjustmentListener getAdjustable();  
public int getValue();
```

...mit denen das auslösende Objekt sowie der eingestellte `int`-Wert abgefragt werden kann.

Bleibt, das Geheimnis um **Layout** und **West** bzw. **South** zu lüften...

- ▶ Jeder Container, in den man weitere Komponenten schachteln möchte, muss über eine Vorschrift verfügen, wie die Komponenten anzuordnen sind.
- ▶ Diese Vorschrift heißt **Layout**.

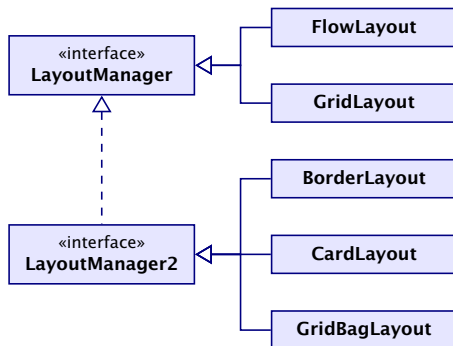
Zur Festlegung des Layouts stellt **Java** das Interface **LayoutManager** zur Verfügung sowie nützliche implementierende Klassen...

- ▶ Eine davon ist das `BorderLayout`.
- ▶ Mithilfe der `String`-Argumente:

```
BorderLayout.NORTH = "North",  
BorderLayout.SOUTH = "South",  
BorderLayout.WEST = "West",  
BorderLayout.EAST = "East", und  
BorderLayout.CENTER = "Center"
```

kann man **genau eine** Komponente am bezeichneten Rand bzw. der Mitte positionieren.

Einige Layoutmanager



Layout Manager

FlowLayout: Komponenten werden von links nach rechts zeilenweise abgelegt; passt eine Komponente nicht mehr in eine Zeile, rückt sie in die nächste.

BorderLayout: Die Fläche wird in die fünf Regionen **North**, **South**, **West**, **East** und **Center** aufgeteilt, die jeweils von einer Komponente eingenommen werden können.

CardLayout: Die Komponenten werden wie in einem Karten-Stapel abgelegt. Der Stapel ermöglicht sowohl den Durchgang in einer festen Reihenfolge wie den Zugriff auf spezielle Elemente.

GridLayout: Die Komponenten werden in einem Gitter mit gegebener Zeilen- und Spalten-Anzahl abgelegt.

GridBagLayout: Wie **GridLayout**, nur flexibler, indem einzelne Komponenten auch mehrere Felder des Gitters belegen können.

Ereignisse

- ▶ Komponenten erzeugen Ereignisse;
- ▶ Listener-Objekte werden an Komponenten für Ereignis-Klassen registriert;
- ▶ Ereignisse werden entsprechend ihrer Herkunft an Listener-Objekte weitergereicht.

Ereignisse

- ▶ Jedes `AWTEvent`-Objekt verfügt über eine `Quelle`, d.h. eine Komponente, die dieses Ereignis erzeugt.
`public Object getSource()` (der Klasse `java.util.EventObject`) liefert dieses Objekt.
- ▶ Gibt es verschiedene Klassen von Komponenten, die Ereignisse der gleichen Klasse erzeugen können, werden diese mit einem geeigneten Interface zusammengefasst.

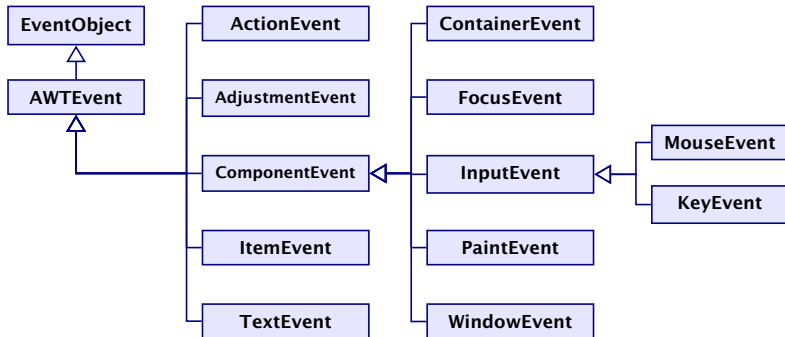
Beispiele:

<i>Ereignisklasse</i>	<i>Interface</i>	<i>Objektmethode</i>
<code>ItemEvent</code>	<code>ItemSelectable</code>	<code>getItemSelectable()</code>
<code>AdjustmentEvent</code>	<code>Adjustable</code>	<code>getAdjustable()</code>

Ereignisse

- ▶ Eine Komponente kann Ereignisse **verschiedener** **AWTEvent**-Klassen erzeugen.
- ▶ Für jede dieser Klassen können getrennt Listener-Objekte registriert werden...
- ▶ Man unterscheidet zwei Sorten von Ereignissen:
 1. **semantische** Ereignis-Klassen — wie **ActionEvent** oder **AdjustmentEvent**;
 2. **low-level** Ereignis-Klassen — wie **WindowEvent** oder **MouseEvent**.

Überblick – Eventklassen

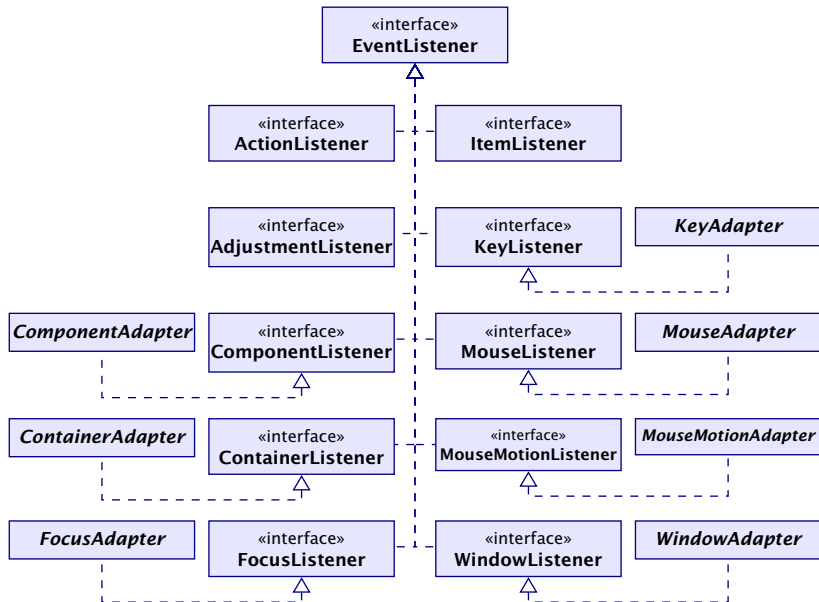


Listeners

- ▶ Zu jeder Klasse von Ereignissen gehört ein Interface, das die zuständigen Listener-Objekte implementieren müssen.
- ▶ Manche Interfaces verlangen die Implementierung **mehrerer** Methoden.
- ▶ In diesem Fall stellt **Java Adapter**-Klassen zur Verfügung.
- ▶ Die Adapterklasse zu einem Interface implementiert sämtliche geforderten Methoden auf **triviale** Weise ;-)
- ▶ In einer Unterklasse der Adapter-Klasse kann man sich darum darauf beschränken, nur diejenigen Methoden zu implementieren, auf die man Wert legt.

Überblick – Eventklassen

Die Adapterklassen sind abstrakt, obwohl sie keine abstrakten Methoden enthalten.



Beispiel – Ein MouseListener

- ▶ Das Interface `MouseListener` verlangt die Implementierung der Methoden:
 - ▶ `void mousePressed(MouseEvent e);`
 - ▶ `void mouseReleased(MouseEvent e);`
 - ▶ `void mouseEntered(MouseEvent e);`
 - ▶ `void mouseExited(MouseEvent e);`
 - ▶ `void mouseClicked(MouseEvent e);`
- ▶ Diese Methoden werden bei den entsprechenden Maus-Ereignissen der Komponente aufgerufen.
- ▶ Unser Beispielprogramm soll bei jedem Maus-Klick eine kleine Kreisfläche malen...

Beispiel – Ein MouseListener

```
1 import javax.swing.*;
2 import java.awt.*; import java.awt.event.*;
3
4 public class Mouse extends JFrame {
5     private Image buffer;
6     private JComponent comp;
7     private Graphics gBuff;
8
9     Mouse() {
10         setSize(500,500);
11         setVisible(true);
12         buffer = createImage(500,500);
13         gBuff = buffer.getGraphics();
14         gBuff.setColor(Color.orange);
15         gBuff.fillRect(0,0,500,500);
16         comp = new MyComponent();
17         comp.addMouseListener(new MyMouseListener());
18         add(comp);
19     }
```

"Mouse.java"

Beispiel – Ein MouseListener

```
20     class MyMouseListener extends MouseAdapter {
21         public void mouseClicked(MouseEvent e) {
22             int x = e.getX(); int y = e.getY();
23             System.out.println("x:"+x+",y:"+y);
24             gBuff.setColor(Color.blue);
25             gBuff.fillOval(x-5,y-5,10,10);
26             repaint();
27         }
28     }
29     class MyComponent extends JComponent {
30         public void paintComponent(Graphics page) {
31             page.drawImage(buffer,0,0,500,500,null);
32         }
33     }
34     public static void main(String args[]) {
35         SwingUtilities.invokeLater(() -> new Mouse());
36     }
37 }
```

"Mouse.java"

Erläuterungen

- ▶ Wir wollen nur die Methode `mouseClicked()` implementieren. Darum definieren wir unsere `MouseListener`-Klasse `MyMouseListener` als Unterklasse der Klasse `MouseListener`.
- ▶ Die `MouseEvent`-Methoden:

```
public int getX();    public int getY();
```

liefern die Koordinaten, an denen der Mouse-Klick erfolgte...
- ▶ an dieser Stelle malen wir einen gefüllten Kreis in den Puffer.
- ▶ Dann rufen wir die Methode `repaint()` auf, um die Änderung sichtbar zu machen...

Beispiel – MouseListener

