

Programmieren II

Events (Ereignisse)



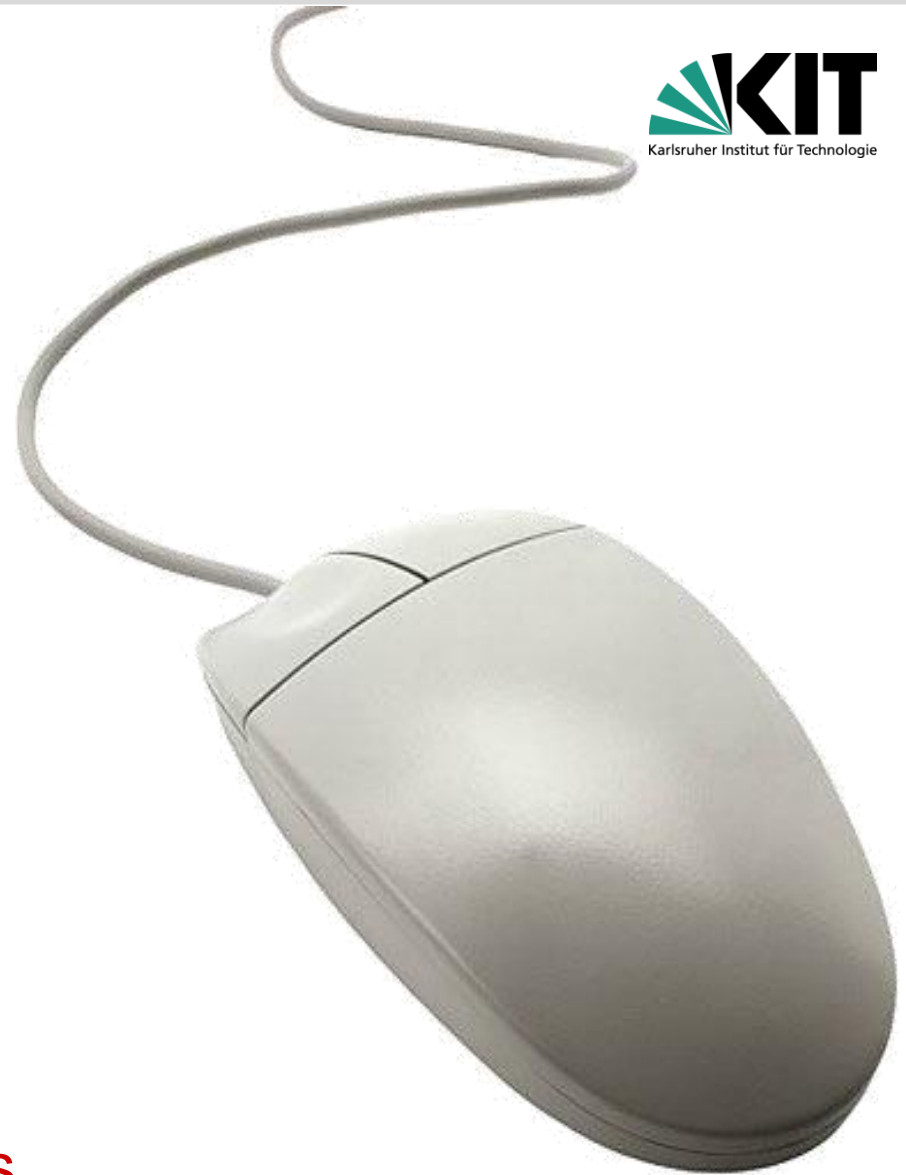
Heusch 16.6 (2. Bd)
Ratz 15

Institut für Automation und angewandte Informatik

```
final List<String> allResults = new ArrayList<String>();  
final Map<String, Integer> typeWordResultCount = new HashMap<String, Integer>();  
final Map<String, Integer> typePoints = new HashMap<String, Integer>();  
evaluation.put(type, typePoints);  
  
for (final Sheet sheet : this.sheets) {  
    final String sheetResult = sheet.getPlayerInput(type);  
    if (sheetResult.startsWith(start) && this.isValidWord(sheetResult, type)) {  
        validWordCountForType++;  
        allResults.add(sheetResult);  
    }  
}
```

Event-Modelle in Java

- Graphische Anwendungen werden über Events gesteuert.
- Sie machen nichts, solange der Benutzer nicht die Maus bewegt, einen Button anklickt oder eine Taste drückt.
- Es gibt unterschiedliche Eventmodelle bei Java 1.0 und Java 1.1.
- **Wir verwenden nur noch das Event-Modell 1.1.**



Das Java Event-Modell 1.1 (1)

- Das Java Event-Modell 1.1 basiert auf dem Konzept des „**Event-Listeners**“.
- Ein Event-Listener ist ein Objekt, das Events abfangen möchte.
- Ein Objekt, das Events generiert (eine Event-Quelle, „event source“),
 - pflegt Listen von Listnern, die über auftretende Events informiert werden wollen. Diese Listen unterscheiden sich nach dem Typ des Events, auf das reagiert werden soll.
 - stellt Methoden bereit, um Listener in diese Liste einzutragen bzw. sie wieder aus ihr zu entfernen

Das Java Event-Modell 1.1 (2)

- Verschiedene Event-Typen werden jeweils durch *eigene Java-Klassen* repräsentiert.
- Jedes Event ist eine Subklasse von `java.util.EventObject`

```
■ java.util.EventObject
    |
    +--java.awt.AWTEvent
        |
        +--java.awt.event.ActionEvent
        +--java.awt.event.AdjustmentEvent
        +--java.awt.event.ComponentEvent
        .
        . |
        . +--java.awt.event.FocusEvent
        . +--java.awt.event.WindowEvent
    +-- ...
```

Das Java Event-Modell 1.1 (3)

- Wird ein Event ausgelöst, benachrichtigt die Event-Quelle alle in der entsprechenden Liste eingetragenen Listener-Objekte, dass ein Event aufgetreten ist.
- Eine Event-Quelle benachrichtigt ein Event-Listener-Objekt durch den Aufruf einer Methode und die Übergabe eines Event-Objektes.
- Damit eine Quelle eine Methode des Listeners aufrufen kann, müssen alle Listener die verlangte Methode implementieren.
- Das wird dadurch sichergestellt, dass alle Event-Listener eines bestimmten Event-Typs ein entsprechendes Interface implementieren, z.B.

```
interface ActionListener
```

mit seiner (einzigen) Methode

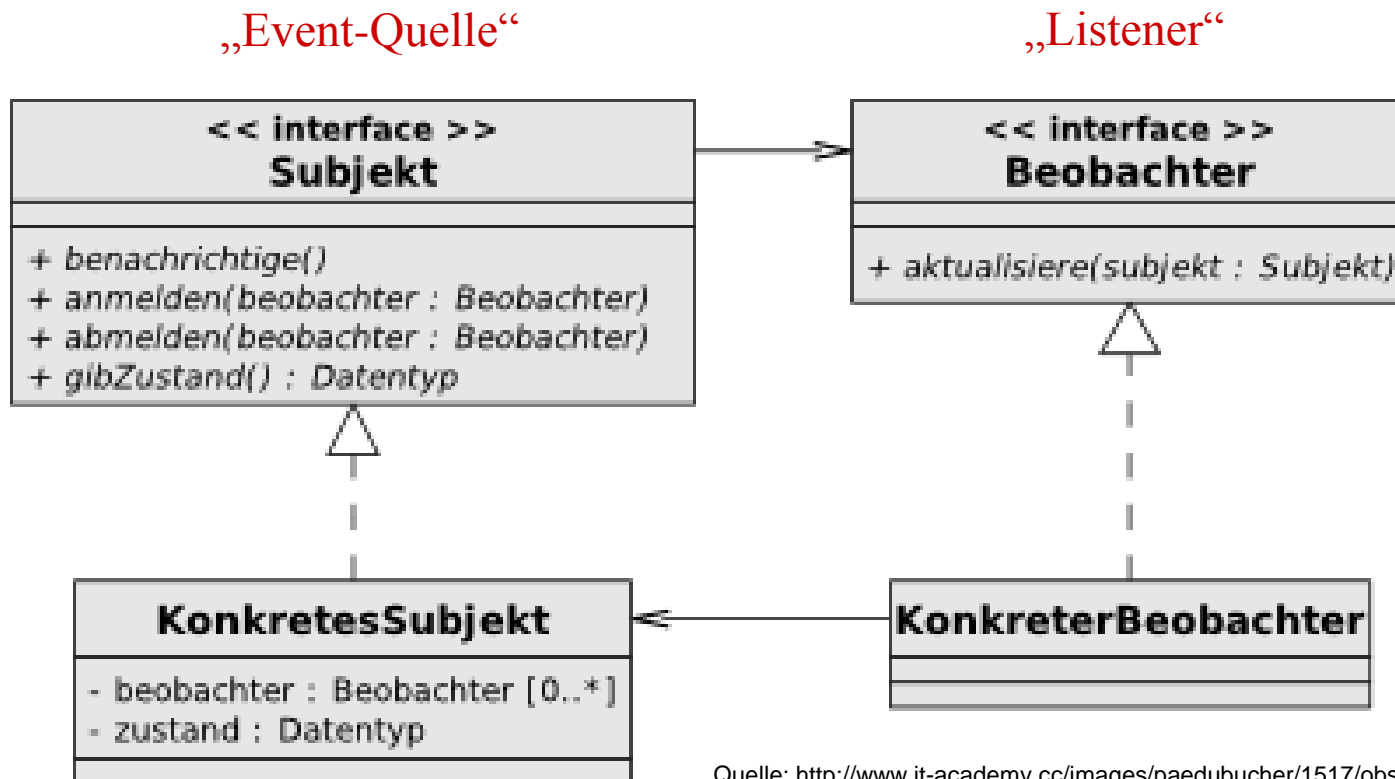
```
void actionPerformed(ActionEvent e)
```

Listener-Interfaces in java.awt.event

- ActionListener
- AdjustmentListener
- AWTEventListener
- ComponentListener
- ContainerListener
- FocusListener
- HierarchyBoundsListener
- HierarchyListener
- InputMethodListener
- ItemListener
- KeyListener
- MouseListener
- MouseMotionListener
- MouseWheelListener
- TextListener
- WindowFocusListener
- WindowListener
- WindowStateListener

Java Events 1.1 und Design-Patterns (1)

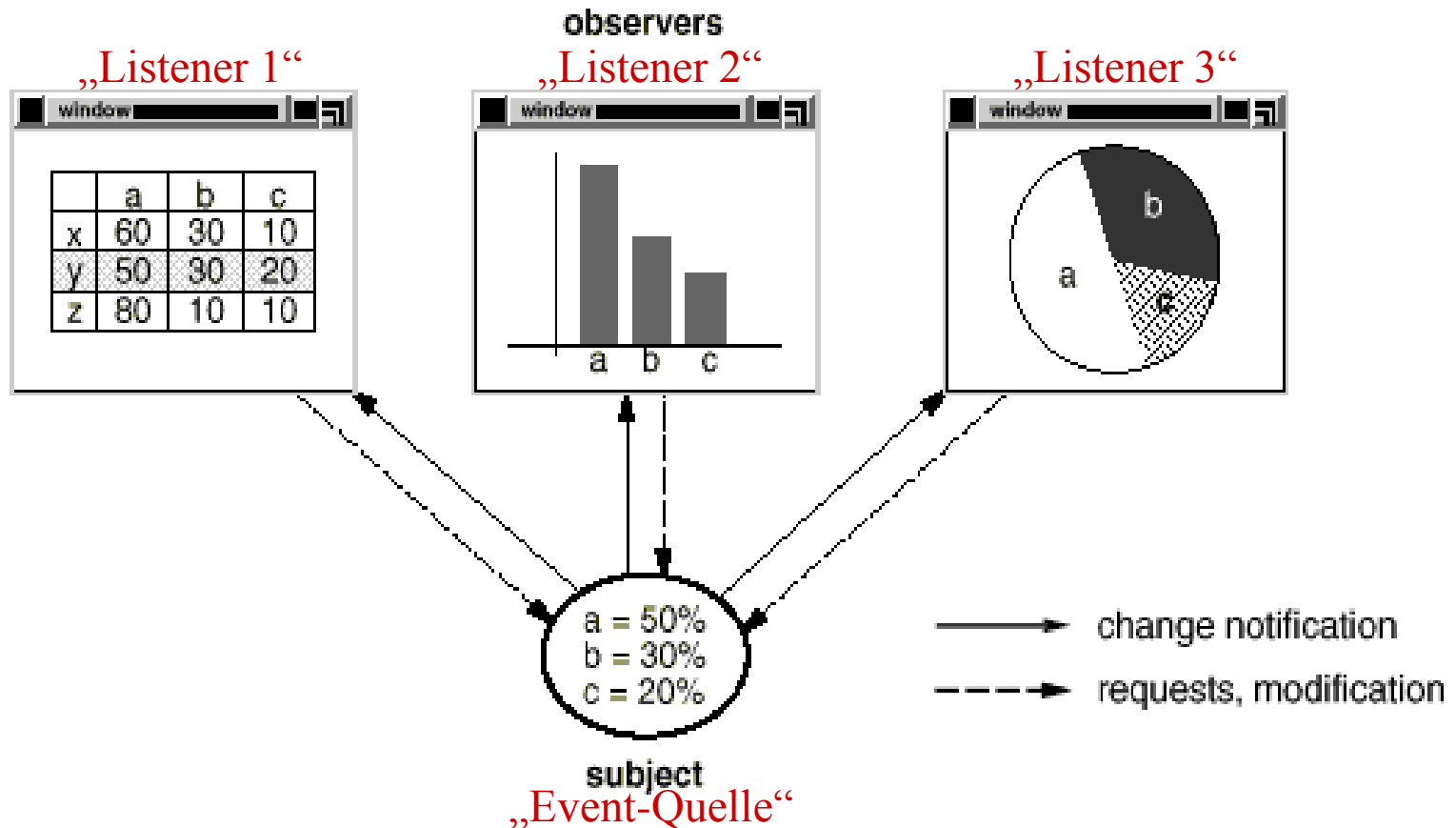
- Die Java-Events entsprechen dem allgemeinen objektorientierten Entwurfsmuster (Design Pattern) „**Observer**“ (s. Vorlesung Softwareengineering)



Quelle: <http://www.it-academy.cc/images/paedubucher/1517/observer.png>

Java Events 1.1 und Design-Patterns (2)

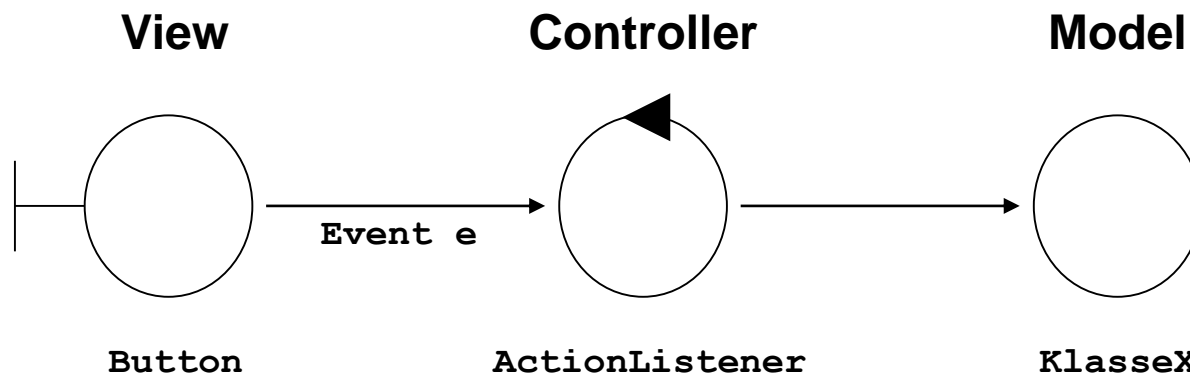
- Beispiel für mehrere Observer auf einem Subject:



Quelle und weitere Infos: <http://userpages.umbc.edu/~tarr/dp/lectures/Observer.pdf>

Java Events 1.1 und Design-Patterns (3)

- Das Java 1.1-Eventmodell eignet sich dank der der Entkopplung von GUI und Eventhandling gut für die Implementierung des „**Model-View-Controller**“-Patterns.
- Trennung von
 - Datenmodell (Model)
 - Darstellung im GUI (View, z.B. Swing-Komponente)
 - Vermittlung zwischen beiden durch Controller (z.B. `ActionListener`)



Eigenschaften des 1.1-Modells

■ Hohe Flexibilität

- Durch die Registrierung von Listenern kann man Eventströme frei steuern, je nachdem, wie man sie braucht.
- Eine saubere Trennung des GUI vom restlichen Programmcode wird ermöglicht.
- Man kann auch zur Laufzeit Listener hinzufügen, entfernen oder austauschen.

■ Geringer Overhead

- Ein Objekt erhält ausschließlich die Event-Typen, die es auch verarbeitet bzw. weiterleitet.

■ Einfachere Event-Identifizierung

- Events sind sofort anhand ihrer Klasse identifizierbar. Jeder Event-Typ wird von einer eigenen Listener-Methode behandelt, so dass eine weitere Typprüfung entfallen kann.

Beispiel

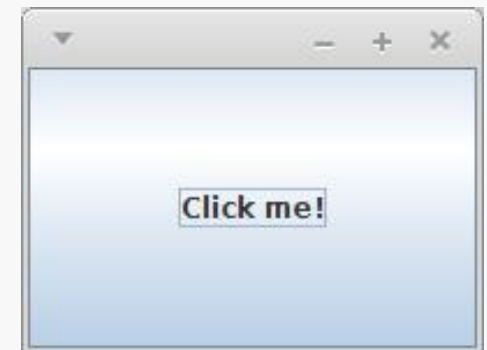
```
import java.awt.Toolkit;
import java.awt.event.*;
import javax.swing.*;

public class Beep extends JFrame implements ActionListener {
    JButton button = new JButton("Click me!");

    public Beep() {
        this.add(this.button);
        this.button.addActionListener(this);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(200, 150);
        this.setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        Toolkit.getDefaultToolkit().beep();
    }

    public static void main(String[] args) {
        new Beep();
    }
}
```



Weiteres Beispiel (1)

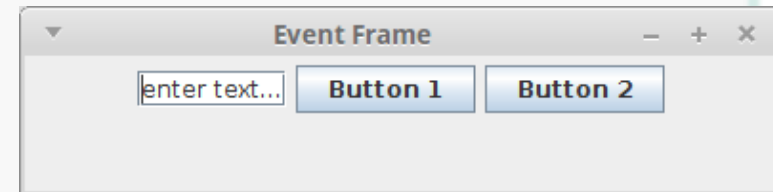
```
import java.awt.FlowLayout; import java.awt.event.*; import javax.swing.*;

public class ButtonEventDemo implements ActionListener {
    JFrame f = new JFrame("Event Frame");
    JTextField text = new JTextField("enter text...");
    JButton button1 = new JButton("Button 1");
    JButton button2 = new JButton("Button 2");

    public ButtonEventDemo() {
        this.f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.f.getContentPane().setLayout(new FlowLayout());
        this.f.add(this.text);
        this.text.addActionListener(this);
        this.f.add(this.button1);
        this.button1.addActionListener(this);
        this.f.add(this.button2);
        this.button2.addActionListener(this);
        this.f.setSize(400, 100);
        this.f.setVisible(true);
    }

    public static void main(String[] args) {
        ButtonEventDemo demo = new ButtonEventDemo();
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == this.text) System.out.println("Text changed");
        else System.out.println(((JButton) e.getSource()).getText() + " pressed.");
    }
}
```



Weiteres Beispiel (2)

- Oder ganz „generisch“:

```
public void actionPerformed(ActionEvent e) {  
    if (e.getSource() == component1) {  
        // Aktion 1  
    }  
    if (e.getSource() == component2) {  
        // Aktion 2  
    }  
    if (e.getSource() == component3) {  
        // Aktion 3  
    }  
    // Bis alle Events abgearbeitet sind ...  
}
```

- Mit `getSource()` erhält man die Objektreferenz auf die GUI-Komponente, für die das Event ausgelöst wurde, z.B. auf den Button, der gerade angeklickt wurde.

Events mit anonymen inneren Klassen behandeln

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

public class EventsWithAnonymousClasses extends JFrame {

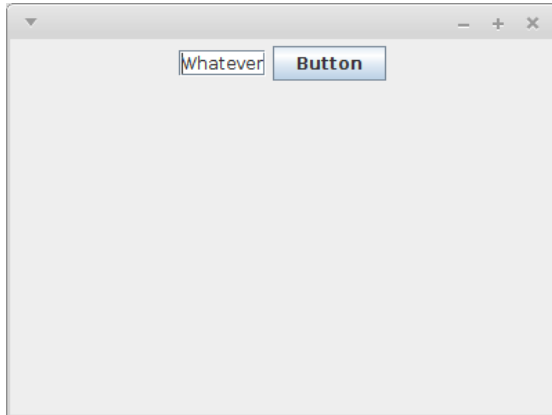
    JTextField text = new JTextField("Whatever");
    JButton button = new JButton("Button");

    EventsWithAnonymousClasses() {
        this.setLayout(new FlowLayout());
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.add(this.text);
        this.add(this.button);
        this.button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                button.setText(text.getText());
            }
        });
        this.setSize(400, 300);
        this.setVisible(true);
    }

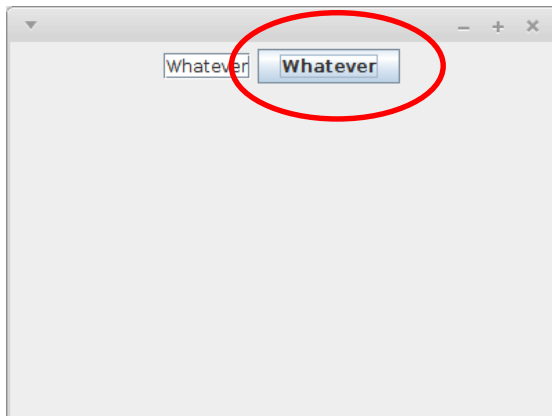
    public static void main(String[] args) {
        new EventsWithAnonymousClasses();
    }
}
```

Die innere Klasse kann auf alle Attribute der Klasse und alle Methoden zugreifen - allerdings nicht auf lokale Variablen des Konstruktors (es sei denn, dass diese Variablen als **final** deklariert sind)!

Screenshot zum vorherigen Beispiel



Fenster nach dem Starten der Applikation



Fenster nach dem Klicken des Buttons

Verschiedene Arten des Event-Handling (1)

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

public class Beep1 extends JFrame
    implements ActionListener {

    JButton button = new JButton("Click me!");

    public Beep1() {
        this.add(this.button);
        this.button.addActionListener(this);

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

    public void actionPerformed(ActionEvent e){
        Toolkit.getDefaultToolkit().beep();
    }
}
```

**hier: Methode des Listener-Interfaces
in der Klasse selbst implementiert**

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

public class Beep2 extends JFrame {
    JButton button = new JButton("Click me!");

    public Beep2() {
        this.add(this.button);
        this.button.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e){
                    Toolkit.getDefaultToolkit().beep();
                }
            });

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

hier: in anonymer innerer Klasse

Verschiedene Arten des Event-Handling (2)

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

public class Beep3 extends JFrame {
    JButton button = new JButton("Click me!");

    public Beep3() {
        this.add(this.button);
        this.button.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    Toolkit.getDefaultToolkit().beep();
                }
            }
        );
        this.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
        this.setSize(200, 150);
        this.setVisible(true);
    }
}
```

```
class ActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        Toolkit.getDefaultToolkit().beep();
    }
}
```

hier: in separater Klasse

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

public class Beep4 extends JFrame {
    ActionListener ac;
    JButton button = new JButton("Click me!");

    public Beep4() {
        this.add(this.button);
        this.ac = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Toolkit.getDefaultToolkit().beep();
            }
        };
        this.button.addActionListener(this.ac);
        this.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
        this.setSize(200, 150);
        this.setVisible(true);
    }
}
```

```
class ActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        Toolkit.getDefaultToolkit().beep();
    }
}
```

hier: mit Referenz auf Listener-Objekt der
sep. Klasse, das dann mehrfach nutzbar wäre

Listener implementieren vs. anonyme Event-Klassen

Listener implementieren

- Unterscheidung nach auslösender Komponente
- Alles in einer Methode bzw. wenigen Methoden
- → geht schneller bei kleinen GUIs mit wenigen Events
- → wird bei größeren GUIs extrem schnell unübersichtlich

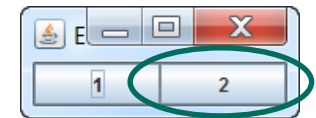
Anonyme Event-Klasse


- keine Unterscheidung der auslösenden Komponente nötig
- Eine Methode je Event
- Ein Listener lässt sich an mehrere Komponenten hängen
- → etwas mehr Aufwand bei kleinen GUIs
- → übersichtlicher bei großen GUIs, da Events sich auf das Programm aufteilen lassen

Event-Verarbeitung: Daten „durchreichen“ (1)

- Drücken eines Buttons löst ein Event aus
 - ➔ Wie kann ich feststellen, welcher Button es war?
 - ➔ Allgemeiner: wie kann ich für Events „Daten“ mitgeben?
- Möglichkeit 1: Button-Text (vgl. Folie 12)
 - Nur geeignet falls Texte unterschiedlich

```
// (...)  
  
JButton btn1 = new JButton( "1" );  
JButton btn2 = new JButton( "2" );  
  
ActionListener listener = new ActionListener() {  
    public void actionPerformed((ActionEvent e) {  
        String text = ((JButton)e.getSource()).getText();  
        System.out.println( text );  
    }  
};  
  
btn1.addActionListener( listener );  
btn2.addActionListener( listener );  
  
// (...)
```



 Ausgabe bei Klick
2

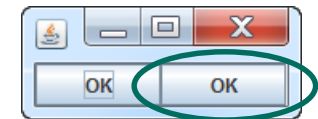
Event-Verarbeitung: Daten „durchreichen“ (2)

- Möglichkeit 2: Objektreferenz (vgl. Folie 13)
 - Nur anwendbar, wenn Referenzen gespeichert werden (bspw. in Instanzvariablen)

```
public class EventData extends JFrame {

    private JButton btn1 = new JButton( "OK" );
    private JButton btn2 = new JButton( "OK" );

    public EventData() {
        // (...)
        ActionListener listener = new ActionListener() {
            public void actionPerformed( ActionEvent e ) {
                if ( e.getSource() == btn1 ) {
                    System.out.println( "First OK" );
                }
                else if ( e.getSource() == btn2 ) {
                    System.out.println( "Second OK" );
                }
            }
        };
        btn1.addActionListener( listener );
        btn2.addActionListener( listener );
        // (...)
    }
}
```



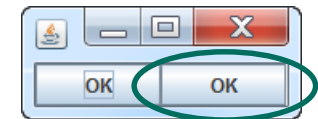
```
> Ausgabe bei Klick
Second OK
```

Event-Verarbeitung: Daten „durchreichen“ (3)

- Möglichkeit 3: separate `ActionListener`-Instanzen
 - Meistens anwendbar
 - bläht jedoch Code auf
 - kann ggf. leicht zu redundantem Code führen

```
// (...)
JButton btn1 = new JButton( "OK" );
JButton btn2 = new JButton( "OK" );

btn1.addActionListener( new ActionListener() {
    public void actionPerformed((ActionEvent e) {
        System.out.println( "Button1" );
    }
});
btn2.addActionListener( new ActionListener() {
    public void actionPerformed((ActionEvent e) {
        System.out.println( "Button2" );
    }
});
// (...)
```



```
> Ausgabe bei Klick
Button2
```

Event-Verarbeitung: Daten „durchreichen“ (4)

■ Möglichkeit 4: *ActionCommand*

- Nur anwendbar, falls sinnvolle Command-Namen vorhanden

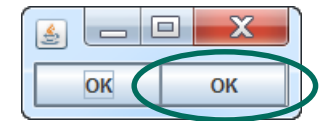
```
// (...)


JButton btn1 = new JButton( "OK" );
JButton btn2 = new JButton( "OK" );
btn1.setActionCommand( "Button1" );
btn2.setActionCommand( "Button2" );

ActionListener listener = new ActionListener() {
    public void actionPerformed((ActionEvent e) {
        String cmd = e.getActionCommand();
        System.out.println( cmd );
    }
};

btn1.addActionListener( listener );
btn2.addActionListener( listener );

// (...)
```



 Ausgabe bei Klick
Button2

Event-Verarbeitung: Daten „durchreichen“ (5)

■ Möglichkeit 5: Button-Name

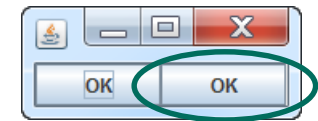
- Nur anwendbar, falls sinnvolle Namen vorhanden
- Nutzt von Component geerbte Methoden


`void setName (String name)` und `String getName ()`

```
// (...)
JButton btn1 = new JButton( "OK" );
JButton btn2 = new JButton( "OK" );
btn1.setName( "Button1" );
btn2.setName( "Button2" );

ActionListener listener = new ActionListener() {
    public void actionPerformed((ActionEvent e) {
        String text = ((JButton)e.getSource()).getName();
        System.out.println( text );
    }
};

btn1.addActionListener( listener );
btn2.addActionListener( listener );
// (...)
```



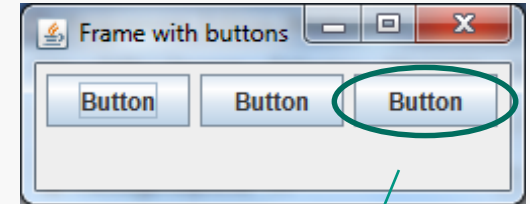
 Ausgabe bei Klick
Button2

Erweitertes Beispiel für Button-Name

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

public class MultipleButtonName2 {
    public MultipleButtonName2() {
        JFrame frame = new JFrame();
        JPanel jp = new JPanel();
        ActionListener listener = new ActionListener() {
            public void actionPerformed( ActionEvent e ) {
                String buttonName = ((JButton)e.getSource()).getName();
                System.out.println( buttonName );
            }
        };
        for ( int i = 1; i <= 3; i++ ) {
            final JButton button = new JButton( "Button" );
            jp.add( button );
            button.setName( "#" + i );
            button.addActionListener( listener );
        }
        frame.add(jp); frame.setTitle("Frame with buttons"); frame.setSize(250, 100);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); frame.setVisible(true);
    }

    public static void main( String[] args ) { new MultipleButtonName2(); }
}
```



```
Ausgabe bei Klick
#3
```

Erweiterung und Modifikation des Beispiels *MultiButton* in Folie 34 des Kapitels „GUI/Swing“

Weitere Möglichkeit: Vererbung (1)

- Möglichkeit 6: Unterklasse von JButton
 - Ermöglicht Erzeugen beliebiger weiterer Attribute

```
public class MyJButton extends JButton {  
    private int myNumber;  
  
    public MyJButton(String text, int myNumber) {  
        super(text);  
        this.myNumber = myNumber;  
    }  
  
    public int getMyNumber() {  
        return myNumber;  
    }  
  
    public void setMyNumber( int myNumber ) {  
        this.myNumber = myNumber;  
    }  
}
```

Weitere Möglichkeit: Vererbung (2)

```
// (...)

MyJButton btn1 = new MyJButton( "OK", 1 );
MyJButton btn2 = new MyJButton( "OK", 2 );

ActionListener listener = new ActionListener() {
    public void actionPerformed( ActionEvent e ) {
        int num = ((MyJButton)e.getSource()).getMyNumber();
        System.out.println( Integer.toString( num ) );
    }
};

btn1.addActionListener( listener );
btn2.addActionListener( listener );

// (...)
```



Doku und Tutorial

■ Javadoc:

- <http://docs.oracle.com/javase/6/docs/api/index.html?overview-summary.html>
- <http://docs.oracle.com/javase/6/docs/api/javax/swing/package-frame.html>

■ Event-Tutorial:

- Lesson: Writing Event Listeners

<http://docs.oracle.com/javase/tutorial/uiswing/events/index.html>