Eventhandling

- Dient der Kommunikation der Controlls mit der eigentlichen Anwendung.
- Komponenten (Button, Label, TextField, Panel …) erzeugen Events als Reaktion auf Benutzeraktivitäten (Nutze drückt Butten → Actionevent).
- Es gibt Events, die von allen Komponenten erzeugt werden können, (z.Bsp.: MouseEvents) und es gibt spezielle Events, die nur von bestimmten Komponenten erzeugt werden (Scrolling Events von Scrollbars)

Eventhandling

- Innerhalb der Eventbehandlungsroutinen findet sich die Funktionalität der Anwendung als Reaktion auf Benutzeraktivitäten.
- Aktivitäten des Benutzers bewirken in java über die Virtuelle Maschine den automatischen Aufruf von vordefinierten Funktionen.
- Diese Funktionen sind in Interfaces oder elementaren Klassen vordefiniert und werden vom Anwendungsentwickler überschrieben.

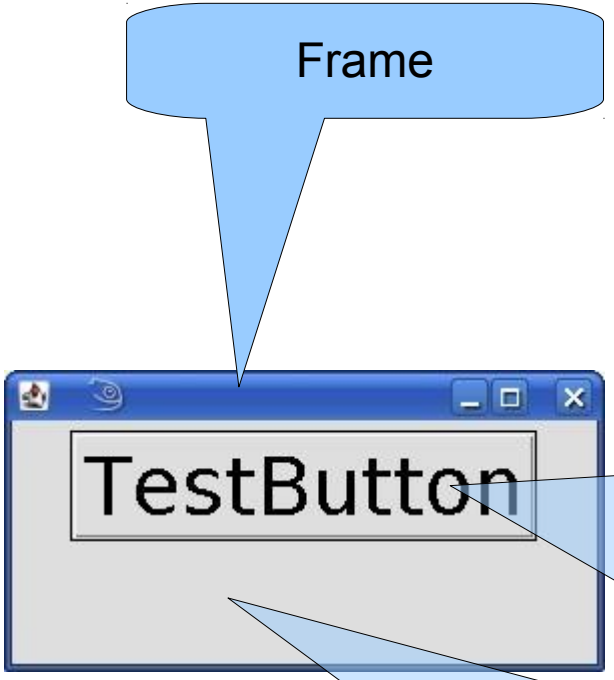
Eventhandling

- Prinzipiell gibt es in grafischen Benutzeroberflächen zwei verschiedene Verfahren für das Eventhandling
  - Delegation model
  - Eventhandler model
- Die Arbeit mit dem delegation model ist in java deprecated. Es gibt aber Bibliotheken zur Oberflächenprogrammierung, die dieses Modell nutzen. Deshalb ist es einer Betrachtung wert.

Delegation model (nur zur Information)

- Für jedes Event gibt es eine spezielle Methode in Component (action, mouseMove, keyDown), die **überschrieben** werden kann. Sie wird bei Eintreten des Ereignisses aufgerufen. Sie gibt true oder false zurück.
- Da auch alle Container von Component erben, kann auch in den umgebenden Containern die Eventmethode überschrieben werden.
- Die Defaultimplementation führt lediglich „return false;“ aus.
- Der Returnwert false bewirkt die Delegation des Events innerhalb der Objekthierarchie, jeweils an den enthaltenden Container.
- Der Returnwert true signalisiert, dass das Event behandelt worden ist, es wird nicht weiter delegiert.

Delegationmodel (java 1.0)




```
class MyButton extends Button
{
   boolean flag;
   MyButton(String label,boolean flag)
   {
      super(label);this.flag=flag;
   }
   @Override
   public boolean action(Event what, Object who)
   {
     System.out.println("action MyButton\n"
                                  +who+" Event: "+what);
     return flag;
   }
}
```

```
class Event10 extends Panel
{
    @Override
    public boolean action(Event what, Object who)
    {
      System.out.println("action Panel\n"+who+" Event: "+what);
      return false;
    }
...
```

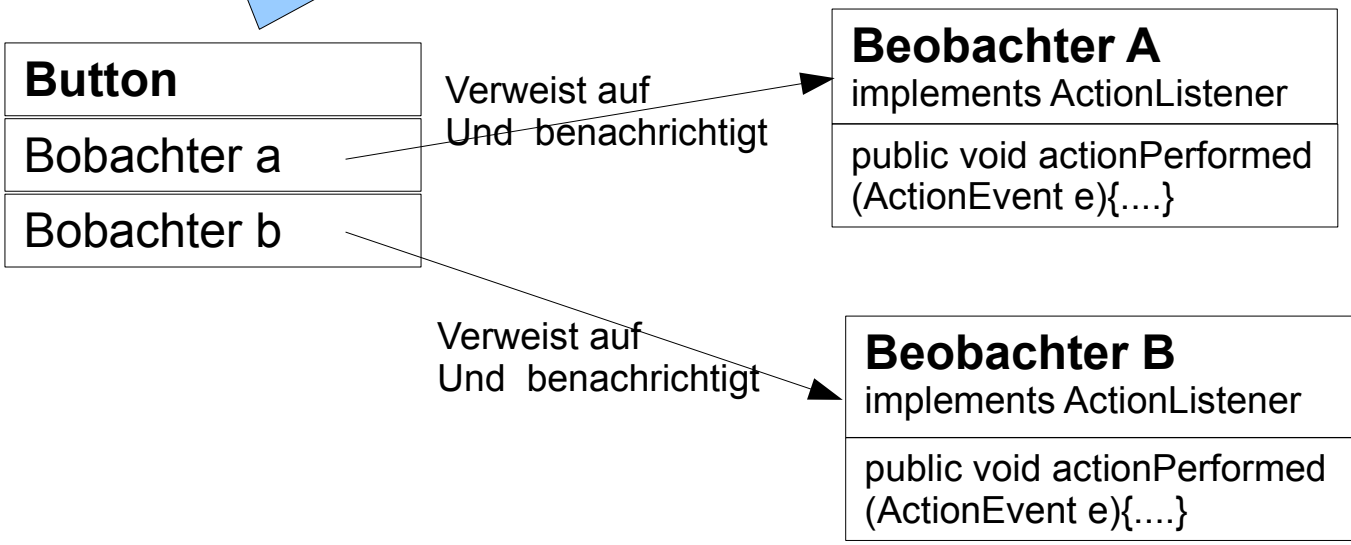
```
import java.awt.*;
import java.awt.event.*;

class Event10 extends Panel
{

  class MyButton extends Button
  {
   boolean flag;
   MyButton(String label,boolean flag)
   {
    super(label);this.flag=flag;
   }
   @Override
   public boolean action(Event what, Object who)
   {
     System.out.println("action MyButton\n"+who+" Event: "+what);
     return flag;
   }
  }
```

```
    public Event10(boolean flag)
    {
       setFont(new Font("System",Font.PLAIN,40));
       MyButton b1=new MyButton("TestButton",flag);
       add(b1);
    }

    @Override
    public boolean action(Event what, Object who)
    {
      System.out.println("action Panel\n"
                              +who+" Event: "+what);
      return false;
    }

    public static void main(String args[])
    {
      Frame F=new Frame();
      Event10 p=new Event10(Boolean.parseBoolean(args[0]));
      F.add(p);
      F.setSize(300,150);
      F.setVisible(true);
      F.addWindowListener(. . .);
    }
}
```

Eventhandler (Observer Pattern)

Der Button ist das Subjekt des Szenarios. Er kann mehrere Beobachter verwalten. Die Beobachter müssen bestimmte Eigenschaften aufweisen.

Beobachter A und Beobachter B können dabei Objekte verschiedener Klassen sein. Sie müssen aber Methoden zur Benachrichtigung, die einem Interface festgeschrieben sind, Implementieren.



Das Subjekt ruft bei Zustandsänderung diese Methoden der Beobachter auf.

Ein Button ist ein solches Subjekt. Er kann mehrere ActionListener (Beobachter) verwalten, die er beim Betätigen des Buttons benachrichtigt. Die Benachrichtigung erfolgt durch Aufruf der Methode actionPerformed, die im Interface ActionListener deklariert ist und in den Klassen der Listenerobjekte implementiert sein muss.

- Eventhandler sind Objekte.
- Die Klasse eines Eventhandlers muss bestimmte Methoden zur Eventbehandlung implementieren.
- Diese Methoden sind in Interfaces (ActionListener, MouseMotionListner...) festgeschrieben.
- Einer Component können beliebig viele Handler (Beobachter) zu jedem Event, das diese Component erzeugen kann, zugeordnet werden.
- Die Zuordnung erfolgt über Methoden

```
void add<event_type>Listener(<event_type>Listener L);
```

<event_type>
hier nicht im Sinne
generischer Typen!

- Über remove...Listener können Listener auch wieder entfernt werden.
- Der Mechanismus findet nicht nur bei GUI Components Anwendung. Dieses Konzept des Observerpatterns wird auch beispielsweise vom XML Sax-Parser verwendet, man spricht von Event getriebenen Programmen.
- Zu Listenern, die mehr als eine Methode implementieren gibt es eine Adapterclass, die alle Methoden des Interfaces als leere Methoden implementiert.
- Durch Ableiten einer Adapterclass und Überschreiben der event handling Methoden werden Listener gebaut.
- Vor allem inner classes kommen hier viel zum Einsatz.

Verschiedene Techniken

Listener können auf vielerlei Weise programmiert werden. Welche Variante zum Einsatz kommt, hängt von vielen Faktoren und der konkreten Situation ab.

- Implementation des ListenerInterfaces in der eigen Klasse
- Implementation des ListenerInterfaces in eienr anderen Klasse
- Verwendung von inner Classes

Erläuterung zu den Beispielen

- In den nachfolgenden Beispielen wird das Eventlistening an Hand der MouseMotion Events erläutert.
- Es wereden die MouseDragged Events behandelt, die ausgelöst werden, wenn die Maus mit gedrückter Taste bewegt wird.
- Es wird die Mousepostion dabei aufgenommen und in posx und posy gespeichert.
- An der aktuellen Stelle posx und posy wird dann ein Schriftzug ausgegeben.
- Man hat dn Eindruck, dass der Schriftzug dem Maus folgt.

Anwendung implementiert Listenerinterface

- Wenn die Anwendungsklasse (abgeleitet von Panel) ein Listenerinterface implementiert, kann das Objekt der eigenen Klasse (this) als Eventlistener verwendet werden.
- Im nachfolgenden Beispiel wird das demosntriert.

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

public class Mouse1 extends Panel implements MouseMotionListener
{
  int PosX=20,PosY=20;
  String S;
  public Mouse1(String S)
  {
    this.S=S;
    setFont(new Font("sanserif",Font.BOLD,24));
    addMouseMotionListener(this);
  }

  public void paint(Graphics g)
  {
    g.drawString(S,PosX,PosY);
  }

  public static void main(String args[])
  {
      . . .
  }
}
```

Alle Funktionen des Interfaces müssen implementiert werden

```
public void mouseDragged(MouseEvent e)
{
  PosX=e.getX(); PosY=e.getY();
  repaint();
}
public void mouseMoved  (MouseEvent e)
{
}
```

Implementation des Interfaces in der eigenen Klasse

Inner classes

- 4 Arten inner classes
  - Nested top level class
  - Memberclass
  - Local class
  - Anonymous class
- Für jede inner class wird ein gesondertes .class-file erzeugt. Der Name wird aus dem Namen der umgebenden Klasse, dem $-Zeichen und dem Namen der Inneren Klasse gebildet. Bei anonymous classes wird eine fortlaufende Nummer angehängt.

Memberclass

- Die Class wird wie ein Member (eine Methode oder Instancevariable) in die Hauptklasse integriert.
- Die Methoden der Memberclass haben Zugang zu den Membern der umgeben Klasse, auch zu den privaten Membern.
- Bei gleichnamigen Membern in der inneren und äußeren Klasse kommt es zur Überdeckung.

Memberclass

```
public class Mouse2 extends Panel
{
  int PosX=20,PosY=20;
  String S;
  public Mouse2(String S)
  {
    this.S=S;
    addMouseMotionListener(new myMouseMotionListener());
  }

  // Member Class
  class myMouseMotionListener extends MouseMotionAdapter
  {

    public void mouseDragged(MouseEvent e)
    {
      PosX=e.getX(); PosY=e.getY(); repaint();
    }
  }

  public void paint(Graphics g)  {g.drawString(S,PosX,PosY); }

  public static void main(String args[])  {     . . .   }
}
```

Basisklasse, hier **MouseMotionAdapter**

Anmerkung : Zugang zu überdeckten Membern der outer class

```
public class InnerOuter
{
   private String s="Outer";

   class Inner
   {
     String s;
     Inner()
     {
       this.s="Inner";
     }

     String getInner(){return s;}
     String getOuter(){return InnerOuter.this.s;}
   };
   Inner createInner(){return new Inner();}

   public static void main(String args[])
   {
     InnerOuter o=new InnerOuter();
     Inner i=o.createInner();
     System.out.println(i.getInner());
     System.out.println(i.getOuter());
   }
}
```

Local Class

- Local classes residieren, wie lokale Variablen, innerhalb von Funktionen.
- Sie haben auch Zugang zu den Instanzvariablen der umgeben Klasse.
- Sie haben (eingeschränkten) Zugang zu Variablen der umgebenden Funktion.
- Zugriff auf Variablen der umgebenden Funktion sollte vermieden werden, er ist ohnehin auf Variable, die als final gekennzeichnet sind beschränkt.

Local Class

Erläuterung dazu: Objekte der localClass haben oft eine viel längere Lebenszeit als die Funktion, in der sie definiert sind und erzeugt werden. Somit haben auch lokale Variable der umgebenden Funktion oft eine viel kürzere Lebenszeit als die Objekte der LocalClass.

Bei der Erzeugung der Objekte der Local Class werden von allen als final gekennzeichneten Variablen Kopien erzeugt und diese in Form eines Bundle als hidden Argument übergeben.

Local class

```
public class Mouse3 extends Panel
{
  int PosX=20,PosY=20;
  String S;
  public Mouse3(String S)
  {
    this.S=S;
    class myMouseMotionListener extends MouseMotionAdapter
    {
      public void mouseDragged(MouseEvent e)
      {
        PosX=e.getX(); PosY=e.getY(); repaint();
      }
    }

    addMouseMotionListener(new myMouseMotionListener());
  }

  public void paint(Graphics g){g.drawString(S,PosX,PosY);}

  public static void main(String args[]) {  . . .  }
}
```

Anonymous Class

- Sie ist in ihren Einschränkungen der local class sehr ähnlich
- Diese Klasse mutet zunächst am Merkwürdigsten an, wird nachher aber sehr viel verwendet.
- Im Grunde wird ein Objekt einer Basisklasse mit new erzeugt und in einem unmittelbar anschließendem Funktionskörper können Member zugefügt oder Methoden überschrieben werden.
- Es ist, wie die Bestellung einer Pizza Tonno, aber bitte mit zusätzlich Spinat.

Anonymous Class

```
public class Mouse4 extends Panel
{
  int PosX=20,PosY=20;
  String S;
  public Mouse4(String S)
  {
    this.S=S;
    // anonymous Class
    addMouseMotionListener(new MouseMotionAdapter()
                      {
                        public void mouseDragged(MouseEvent e)
                        {
                         PosX=e.getX(); PosY=e.getY(); repaint();
                        }
                      });
  }

  public void paint(Graphics g)  {    g.drawString(S,PosX,PosY);  }

  public static void main(String args[])  {  . . .  }
}
```

Hier wird das Objekt der Basis-klasse erzeugt und die Methode **MouseDragged** überschrieben

Nested top Level Class

- Sytaktisch entspricht sie der Memberclass, nur dass ihr der modiifier static vorangestellt ist.
- Bedingt durch static hat sie keinen Zugang den Variablen der umgebenden Klasse.
- Im Grunde gleicht sie einer separaten Klasse, die als nicht public Klasse in der selben Datei programmiert wurde.
- Werte, der umgebenden Klasse, die benötigt werden, müssen als Erzeugungsparameter übergeben werden

Nested top level class

```
public class Mouse5 extends Panel
{
  int PosX=20,PosY=20;
  String S;
  public Mouse5(String S)
  {
    this.S=S;
    addMouseMotionListener(new myMouseMotionListener(this));
  }
  // nested top level Class
  static class myMouseMotionListener extends MouseMotionAdapter
  {
    Mouse5 M5;
    myMouseMotionListener(Mouse5 M5)
    {
      this.M5=M5;
    }
    public void mouseDragged(MouseEvent e)
    {
      M5.PosX=e.getX(); M5.PosY=e.getY(); M5.repaint();
    }
  }
  public void paint(Graphics g)  {    g.drawString(S,PosX,PosY);  }
  public static void main(String args[])  { . . .  }
}
```

Non public class in same file

```
  // non public Class
class myMouseMotionListener extends MouseMotionAdapter
{
  Mouse6 M6;
  myMouseMotionListener(Mouse6 M6)
  {
    this.M6=M6;
  }

  public void mouseDragged(MouseEvent e)
  {
    M6.PosX=e.getX(); M6.PosY=e.getY(); M6.repaint();
  }
}

public class Mouse6 extends Panel
{
// wie bei nested top level class
. . .
}
```

Listener	Adapter	Methods	Notes
ActionListener		actionPerformed	
AjustmentListener		adjustmentValueChanged	Scrollevents
ComponentListener	ComponentAdapter	componentHidden componentMoved componentResized componentShown	
ContainerListener	ContainerAdapter	componentAdded componentRemoved	
FocusListener	FocusAdapter	focusGained focusLost	
ItemListener		itemStateChanged	
KeyListener	KeyAdapter	keyPressed keyRelesed keyTyped	

MouseListener	MouseAdapter	mouseClicked mouseEntered mouseExited mousePressed mouseRelesed
MouseMotionListener	MouseMotionAdapter	mouseDragged mouseMoved
TextListener		textValueChanged
WindowListener	WindowAdapter	windowActivated windowClosed windowClosing windowDectivated windowDeiconified windowIconified windowOpened

Zwei Strategien

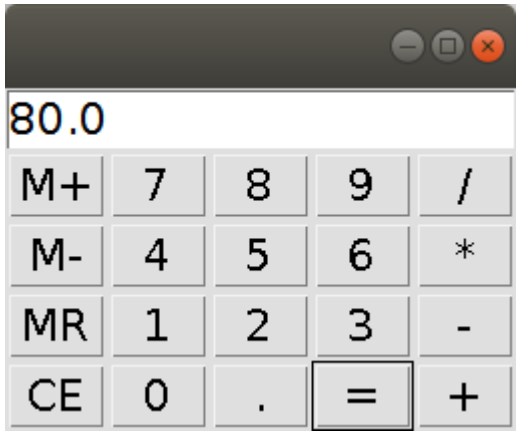
- Ein Eventhandler für alle Events gleichen Eventtyps in dem abgefragt wird, welche Komponente der Auslöser des Events war (z.Bsp. evnt.getSource()==B1). Davon Abhängig wird die Aktion ausgeführt. Als Listenerobjekt wird hier oft this verwendet.
- Jede Komponente bekommt ihren eigenen Eventhandler für jedes Event. Abfragen sind hier nicht nötig. Diese Variante ist unter Nutzung von inner classes interessant.
- Mischformen bilden in vielen Fällen die optimale Variante

Lösungsvorschlag für Taschenrechner

- Es können Gruppen von Buttons mit dem selben Handler verbunden werden.
- Eine Gruppe könnten die Zifferntasten bilden, sie werden alle gleich behandelt.
- Für Operationstasten sollte ein Handler benutzt werden.
- Ein Array mit dem Elementetyp ActionListener kann Referenzen auf die jeweiligen Listener enthalten und so in einer Schleife den Buttons zugeordnet werden.

Zuordnung der Tasten und Listener

```
String lbls[]={
 "M+","7","8","9","/",
 "M-","4","5" ,"6" ,"*" ,
 "MR","1" ,"2" ,"3" ,"-" ,
 "CE","0" ,"." ,"=" ,"+" };
ActionListener lstnrs[]={
 ml,  nl, nl, nl, nx,
 ml,  nl, nl, nl, nx,
 ml,  nl, nl, nl, nx,
 ce,  nl, dl, nx,  nx};
```



Eine solche Konstruktion erlaubt die sehr effiziente Erzeugung der Buttons in einer Schleife.