Grundlagen der Informatik für Ingenieure I

10. Eventhandling in Java

10.1 Ereignisse (*Events vs. Exceptions*)

- Java unterscheidet drei Arten von Ereignissen:
  - ***Errors:*** Interne Fehler in der *JVM* bzw. vom Benutzerprogramm zur Laufzeit nicht behandelbare Fehler
  - ***Exceptions:*** alle anderen Ereignisarten, ausgenommen Interaktionsereignisse mit dem *GUI*, unterteilt in
    - *Runtime Exceptions*
    - *I/O-Exceptions*
    - sonstige *Exceptions*
  - ***Events:*** Interaktionsereignisse mit dem *GUI* (dieses Kapitel)

    Errors und Exceptions: Siehe Kapitel 12

10.2 *Eventhandling* in Java

- *Eventhandling* in Java ist Teil des *AWT-package*.
- Es dient der Kommunikation eines Programmsystems mit der “Benutzeroberfläche”, dem *User-Interface (UI; GUI)*.
- Es handelt sich hierbei z. B. um
  - *Keyboard* - Betätigungen
  - *Mouse* - Bewegungen oder *Mouse*-Clicks
  - Interaktionen mit der Graphischen Benutzeroberfläche.
- Wir müssen in unserem Programm eine Konstruktion vorsehen, die aufpasst, ob ein solches Ereignis eintritt.
- Das ist Aufgabe der ***"Listener".***

10.3 Listener

- Das *AWT-Package* stellt uns, geordnet nach *Eventklassen*, ***“Listener”*** zur Verfügung.
- ***Listener*** sind ***Interfaces***, die durch ihre Definition das ***Handling*** mit ***Events*** vorgeben.
- Beispiele für ***Listener*** sind:

```
MouseListener
ActionListener
KeyListener
```

- Damit das Laufzeitsystem weiß, dass wir einen solchen Dienst in Anspruch nehmen wollen, müssen wir ***Listener*** bei der jeweiligen ***AWT***-Komponenten (Button, Checkbox, ...) anmelden (registrieren).

10.3 Listener

- Für jeden ***Listener*** gibt es eine ***add-method()*** mit der man ihn registrieren lassen kann, z. B.:

```
addMouseListener(...)
addMouseMotionListener(...)
addKeyListener(...)
```

- Zu jedem ***Listener*** existiert eine entsprechende ***Event class***,
  - z. B. zum ***MouseListener*** die Klasse ***MouseEvent*** oder zum ***ActionListener*** die Klasse ***ActionEvent***
  - In diesen Klassen sind Methoden definiert, die man bei der Behandlung der zugehörigen ***Events*** benötigt.
  - Tritt ein ***Event*** auf, wird ein Objekt der zugehörigen ***Event class*** erzeugt und der entsprechenden ***Listener*** - Methode übergeben.

10.4 Adapter

- Bei der Verwendung eines ***Interfaces*** muss dieses vollständig implementiert werden, auch dann, wenn man nur einen Teil der Funktionalität ausnutzen will. Aus diesem Grund gibt es für viele *Listener* ***Adapter***.
- ***Adapter*** sind (abstrakte) Klassen! Sie implementieren vollständig - im Sinne von Methodendefinitionen, u. U. auch mit leerem Rumpf - die entsprechenden *Interfaces* z. B.

```
public abstract class MouseAdapter.....
       implements MouseListener {...}
```

- Wenden wir *Adapter* an, brauchen wir nur die Methoden zu implementieren *(overriding)*, die wir zur Lösung unseres Problems benötigen.

10.5 Beispiel: Buttons

◆ Beispiel für die Anwendung eines ***Listeners***:

```
// button actions

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

public class ButtonActionsTest extends java.applet.Applet {

    Button redButton, blueButton, greenButton;

    class RedButtonListener implements ActionListener {
        public void actionPerformed( ActionEvent e ) {
            setBackground( Color.red );
        }
    }
```


10.5 Beispiel: Buttons

◆ Beispiel für die Anwendung eines ***Listeners*** (cont.):

```
    class BlueButtonListener implements ActionListener {
        public void actionPerformed( ActionEvent e ) {
            setBackground( Color.blue );
        }
    }

    class GreenButtonListener implements ActionListener {
        public void actionPerformed( ActionEvent e ) {
            setBackground( Color.green );
        }
    }
```

10.5 Beispiel: Buttons

◆ Beispiel für die Anwendung eines Listeners (cont.):

```
    public void init() {
        setFont( new Font( "Helvetiva", Font.BOLD, 72 ) );
        setBackground( Color.white );
        setLayout( new FlowLayout( FlowLayout.CENTER, 10, 10 ) );

        redButton = new Button( "Red" );
        RedButtonListener redListener = new RedButtonListener();
        redButton.addActionListener( redListener );
        add( redButton );

        blueButton = new Button( "Blue" );
        BlueButtonListener blueListener
                            = new BlueButtonListener();
        blueButton.addActionListener( blueListener);
        add( blueButton );

        greenButton = new Button( "Green" );
        GreenButtonListener greenListener
                           = new GreenButtonListener();
        greenButton.addActionListener( greenListener);
        add( greenButton );
    }
}
```


10.5 Beispiel: Buttons

◆ Zum Beispiel:

- Es wird das Interface ***ActionListener*** verwendet
- Der ***ActionListener*** enthält nur eine Methode:

  ```
  actionPerformed( ActionEvent e )
  ```

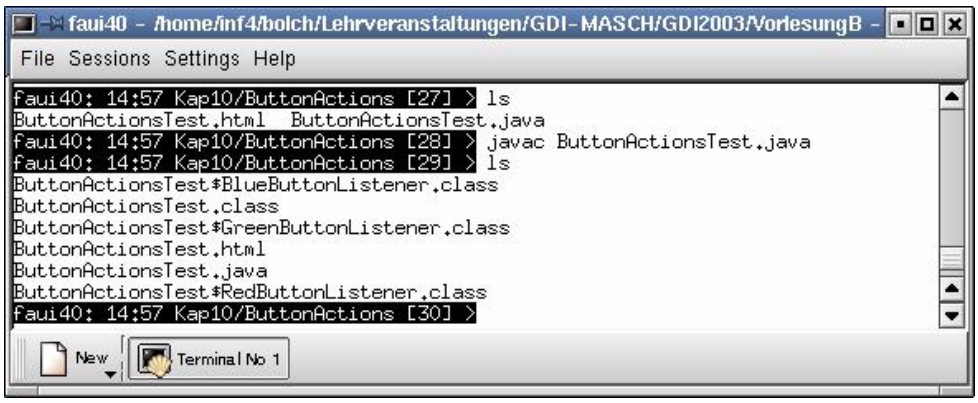
- Zu ***actionPerformed*** gehört die Eventklasse ***ActionEvent.***
- Mit der ***add-method():***

  ```
  addActionListener( ... )
  ```

  wird der ***ActionListener*** bei dem jeweiligen ***Button*** registriert
- Die ***ButtonListener*** wurden innerhalb einer anderen Klasse (der *Applet*-Klasse) definiert.
- Man nennt solche Klassen ***Inner Classes***.

10.5 Beispiel: Buttons

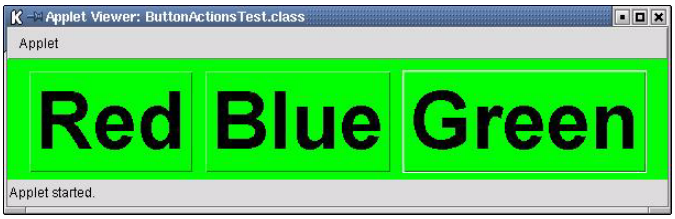
◆ Compilieren mit ***javac:***



- Es werden für die äußere Klasse ***ButtonActionTest*** und für die 3 ***Inner Classes*** jeweils eine ***class***-Datei (Byte-Code) erzeugt.

10.5 Beispiel: Buttons

◆ Ergebnis mit Appletviewer:



10.6 Anonymous Inner Classes

- Eine Variante der ***Inner Class*** ist die ***Anonymous Inner Class***
- Beispiel: Definition von ***Eventhandler-Classes*** als ***Anonymous Inner Classes:***
  - Es wird für jeden ***Button*** ein Eventhandler-Objekt *"**ButtonListener**"* erzeugt.
  - Die Definition erfolgt "innerhalb" einer Klasse als sog. ***Anonymous Inner Classes.***
  - Der Compiler generiert jeweils ein separates ***.class-file.***
  - Zur Laufzeit ist dadurch die Zuordnung ***Event/Eventhandler*** gegeben.

10.6 Anonymous Inner Classes

- Beispiel: ButtonActionsInnerAnonym

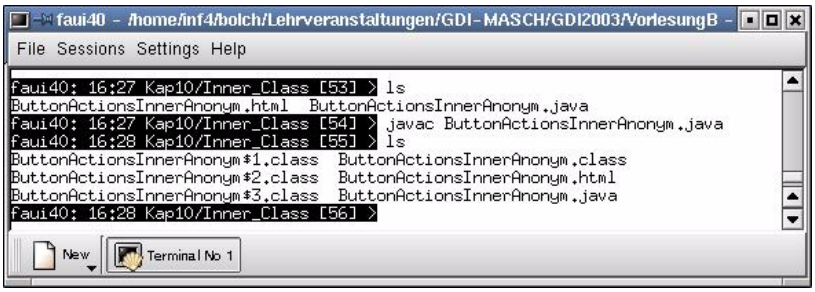
```
import java.awt.*;
import java.awt.event.*;

public class ButtonActionsInnerAnonym extends java.applet.Applet {

    Button redButton, blueButton, greenButton;

    public void init() {

        setFont( new Font( "Helvetiva", Font.BOLD, 72 ) );
        setBackground( Color.white );
        setLayout( new FlowLayout( FlowLayout.CENTER, 10, 10 ) );

        redButton = new Button( "Red" );
        add( redButton );

        blueButton = new Button( "Blue" );
        add( blueButton );

        greenButton = new Button( "Green" );
        add( greenButton );
```

10.6 Anonymous Inner Classes

```
        redButton.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent e ){
                setBackground( Color.red );
            }
        } );

        blueButton.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent e ){
                setBackground( Color.blue );
            }
        } );

        greenButton.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent e ){
                setBackground( Color.green );
            }
        } );
    }
}
```


10.6 Anonymous Inner Classes

◆ Compilieren mit ***javac:***

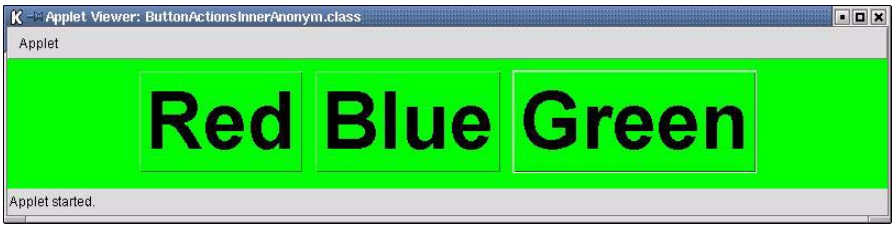


- Es wird für jede ***Anonymous Inner Class*** eine *class*-Datei erzeugt. die zur Unterscheidung mit Ziffern versehen werden.

- Durch Einführung der ***Anonymous Inner Classes*** wird der Code kürzer und übersichtlicher, durch deutliche Trennung von der Erzeugung der Buttons und der Registrierung (Anmeldung) der ***ActionListeners*** bei den Buttons.

10.6 Anonymous Inner Classes

◆ Ergebnis mit Appletviewer:



10.7 Mouse- und Key - Eventhandling

◆ *Mouse- und Key-Listener*:

Listener Interface	Events	Method Definition
MouseListener	mouse down	public void mousePressed(MouseEvent e)
	mouse up	public void mouseReleased(MouseEvent e)
	mouse enter	public void mouseEntered(MouseEvent e)
	mouse exit	public void mouseExited(MouseEvent e)
	mouse clicks	public void mouseClicked(MouseEvent e)
		(a mouse click is a press followed by a release in the same location)
MouseMo-tionListener	mouse move	public void mouseMoved(MouseMotionEvent e)
	mouse drag	public void mouseDragged(MouseMotionEvent e)
KeyListener	key down	public void keyReleased(KeyEvent e)
	key up	public void keyPressed(KeyEvent e)
	key typed	public void keyTyped(KeyEvent e)
		(a key typed is a key down followed by the same key up)

10.7 Mouse- und Key - Eventhandling

◆ Ein Beispiel mit vollständiger Implementation des Interfaces im Applet:

```
/* draw lines at each click and drag */

import java.awt.Graphics;
import java.awt.Color;
import java.awt.Point;
import java.awt.event.*;

public class Lines extends java.applet.Applet
   implements MouseListener, MouseMotionListener {

  final int MAXLINES = 10;
  private Point starts[] = new Point[ MAXLINES ]; // starting points
  private Point ends[] = new Point[ MAXLINES ];    // endingpoints
  private Point anchor;    // start of current line
  private Point currentpoint; // current end of line
  private int currline = 0; // number of lines

public void init() {
    setBackground( Color.white );
  // register event listeners
    addMouseListener( this );
    addMouseMotionListener( this );
}
```


10.7 Mouse- und Key - Eventhandling

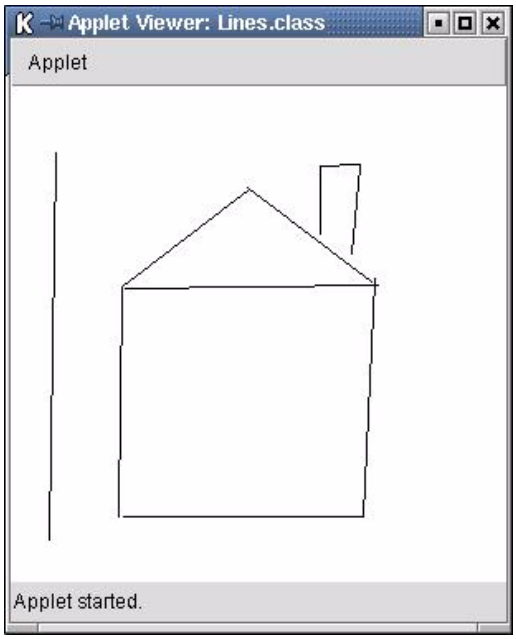
```
  // needed to satisfy listener interfaces

public void mouseMoved( MouseEvent e ) {}
public void mouseClicked( MouseEvent e ) {}
public void mouseEntered( MouseEvent e ) {}
public void mouseExited( MouseEvent e ) {}

public void mousePressed( MouseEvent e ) {
  if ( currline < MAXLINES )
    anchor = new Point( e.getX(), e.getY() );
  else
    System.out.println( "Too many lines." );
  }

public void mouseReleased( MouseEvent e ) {
  if ( currline < MAXLINES )
    addline( e.getX(), e.getY() );
  }

public void mouseDragged( MouseEvent e ) {
  if (currline < MAXLINES) {
    currentpoint = new Point( e.getX(), e.getY() );
    repaint();
  }
}
```

10.7 Mouse- und Key - Eventhandling

```
void addline( int x, int y ) {
    starts[ currline ] = anchor;
    ends[ currline ] = new Point( x, y );
    currline++;
    currentpoint = null;
    anchor = null;
    repaint();
}

public void paint( Graphics g ) {

        // draw existing lines

  for ( int i = 0; i < currline; i++ ) {
    g.drawLine( starts[i].x, starts[i].y, ends[i].x, ends[i].y );
  }

        // draw current line

  g.setColor( Color.blue );
  if ( currentpoint != null )
    g.drawLine( anchor.x, anchor.y, currentpoint.x, currentpoint.y );
  }
}
```


10.7 Mouse- und Key - Eventhandling

◆ Ergebnis mit Appletviewer:



10.7 Mouse- und Key - Eventhandling

◆ Ein Beispiel mit Testausgaben auf die Java-Console des Browsers:

```
/* press a key then use arrows to move it around */

import java.awt.Graphics;
import java.awt.Color;
import java.awt.Event;
import java.awt.Font;
import java.awt.event.*;

public class Keys extends java.applet.Applet
  implements KeyListener {
  private char currkey;
  private int currx;
  private int curry
  private boolean notmove = true;

  public void init() {
    currx = ( this.getWidth() / 2 ) -8;  //default
    curry = ( this.getHeight() / 2 ) -16;
    setBackground( Color.white );
    setFont( new Font( "Helvetica", Font.BOLD, 36 ) );
    addKeyListener( this );
  }
```


10.7 Mouse- und Key - Eventhandling

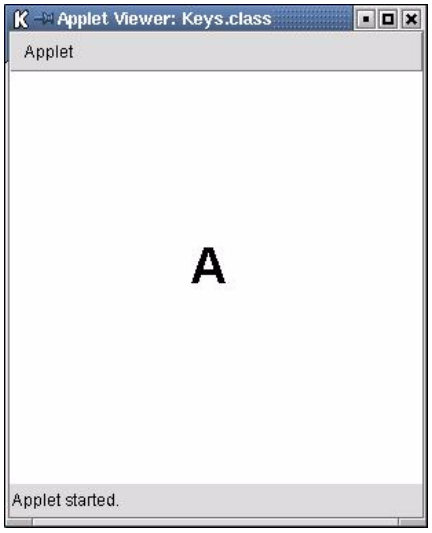
```
  public void keyReleased( KeyEvent e)  { }
  public void keyTyped( KeyEvent e ) { }

  public void keyPressed( KeyEvent e ) {

    if ( e.getKeyCode() == KeyEvent.VK_DOWN ) {
      curry += 5;
          notmove = false;
    }
    if ( e.getKeyCode() == KeyEvent.VK_UP ) {
      curry -= 5;
      notmove = false;
    }
    if ( e.getKeyCode() == KeyEvent.VK_LEFT ) {
      currx -= 5;
      notmove = false;
    }
    if ( e.getKeyCode() == KeyEvent.VK_RIGHT ) {
      currx += 5;
      notmove = false;
    }
```

10.7 Mouse- und Key - Eventhandling

```
    if ( notmove ) {
       currkey = e.getKeyChar();
    }
    notmove = true;
    repaint();
  }

  public void paint(Graphics g) {
    if ( currkey != 0 ) {
      g.drawString( String.valueOf( currkey ), currx, curry );
    }
  }
}
```


10.7 Mouse- und Key - Eventhandling

- Ergebnis mit Appletviewer:



10.8 Listener (Übersicht)

Listener	Methods
ActionListener	actionPerformed(ActionEvent e)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent e)
ComponentListener	componentResized(ComponentEvent e)
	componentMoved(ComponentEvent e)
	componentShown(ComponentEvent e)
	componentHidden(ComponentEvent e)
ContainerListener	componenentAdded(ContainerEvent e)
	componenentRemoved(ContainerEvent e)
FocusListener	focusGained(FocusEvent e)
	focusLost(FocusEvent e)
ItemListener	itemStateChanged(ItemEvent e)
TextListener	textValueChanged(TextEvent e)

10.8 Listener (Übersicht)

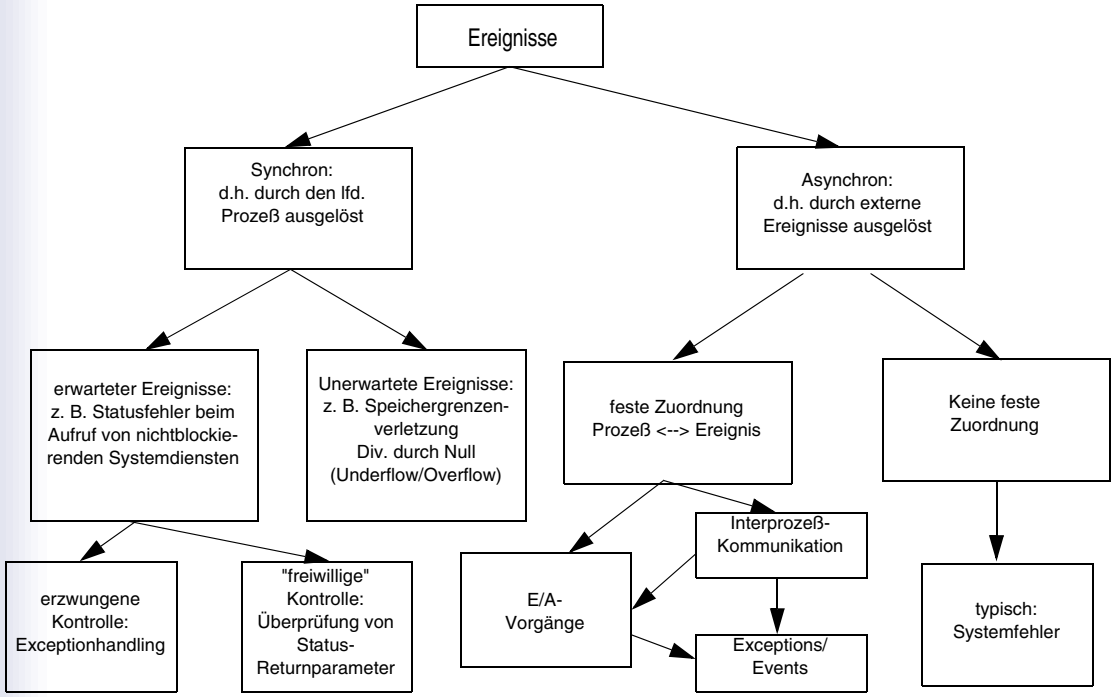
10.8 Listener (Übersicht)

Listener	Methods
WindowListener	windowActivatet(WindowEvent e)
	windowClosed(WindowEvent e)
	windowClosing(WindowEvent e)
	windowDeactivatet(WindowEvent e)
	windowDeiconified(WindowEvent e)
	windowIconified(WindowEvent e)
	windowOpened(WindowEvent e)

◆ Die Methoden der *Event-classes* finden Sie unter ...java.awt.event.

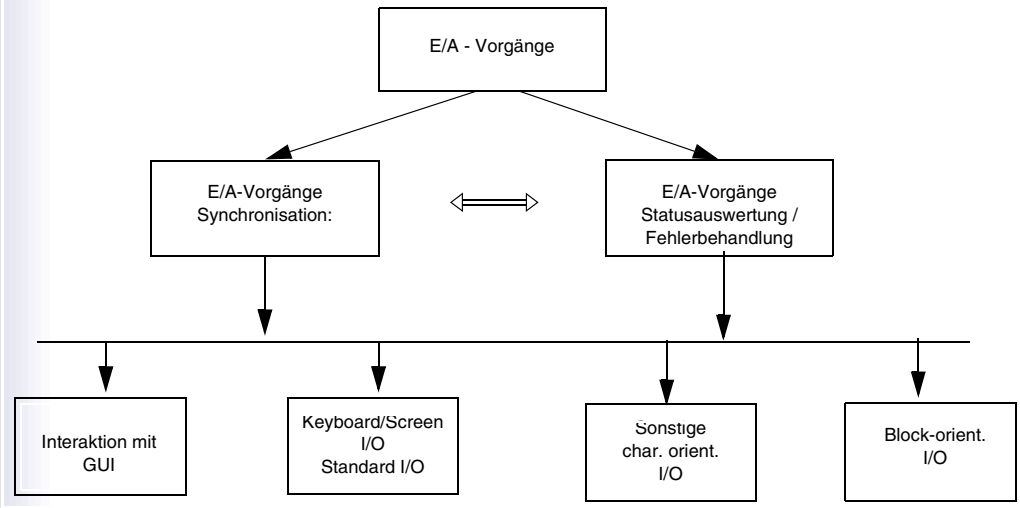
10.9 Anhang: Ereignisse (*Events vs. Exceptions*)

- Allgemeine Klassifikation von Ereignissen(Überblick)



10.9 Anhang: Ereignisse (*Events vs. Exceptions*)

- Allgemeine Klassifikation von Ereignissen(Überblick):



10.9 Anhang: Ereignisse (*Events vs. Exceptions*)

- Abgesehen von einigen wenigen “harten” Fehlerklassen, die das Betriebssystem veranlassen, Programmsysteme zu terminieren, müssen robuste Programmsysteme allen potentiell möglichen Ausnahmesituationen begegnen können.
  - Zu diesen “harten” Fehlern (Laufzeitfehlern; direkt vom Programm ausgelöst) gehören
    - Speichergrenzenverletzungen (Segmentation Violation)
    - Arithmetische Fehler
      - Division durch Null
      - Overflow/Underflow (wird heute weitgehend toleriert - Programm muß diese Klasse von Fehlern vermeiden.)
- Die Klasse der “harten” Fehler auf Betriebssystemebene führen zum *Crashen* des Systems.

10.9 Anhang: Ereignisse (Events vs. Exceptions)

10.9 Anhang: Ereignisse (*Events vs. Exceptions*)

- In der Java-Umgebung werden solche Fehler durch den Interpreter “abgefangen”.
  - Eine *“Null-Pointer-Exception”* wäre potentiell eine Speichergrenzenverletzung.
  - *Arithmetic Exception* (Div. durch Null)

  Darüberhinaus generiert die Java Laufzeitumgebung *(JVM)* weitere Laufzeitfehler, wenn objekt- bzw. java-spezifische Ereignisse auftreten, die erst zur Laufzeit erkannt werden können, wie z. B.:
  - *Security Exception*
  - *Illegal Argument Exception*
  - *Illegal Thread State Exception*

  ...(Siehe spätere Kapitel)

10.9 Anhang: Ereignisse (*Events vs. Exceptions*)

- Sämtliche anderen (asynchronen) Ereignisse können nur dann auftreten, wenn vom Programm her irgendwann Vorgänge angestoßen wurden, die zu solchen Ereignissen führen können. Typischerweise werden diese Ereignisse vom Programm durch Auswertung von Statusinformationen (häufig Returnparameter) behandelt.
- Zu diesen Ereignisklassen gehören:
  - Synchronisierung von E/A-Vorgängen mit Prozessen
  - Fehlersituationen im Zusammenhang mit E/A-Vorgängen
  - Fehlersituationen im Zusammenhang mit anderen Systemaufrufen
  - Prozeßkommunikation
  - *Signalhandling*
- In einigen seltenen Fällen kann man auch Ereignisse bewußt durch Maskieren ignorieren; z. B. einige "Unix-*Signale*".
- Die Zuordnung externer (asynchroner) Ereignisse zu Aktivitätsträgern (Prozesse) wird vom Betriebssystem vorgenommen.
- Auch diese Ereignisklassen werden durch die *Java Virtual Machine* “objektorientiert aufbereitet” an das Java-Programmsystem weitergegeben.