Ereignisbehandlung und Exceptions in Java

PROG 2: Einführung in die Programmierung für Wirtschaftsinformatiker

Steffen Helke

Technische Universität Berlin
Fachgebiet Softwaretechnik

22. April 2013

Übersicht

- Wiederholung: Layout-Manager & GUI-Komponenten
- Ereignisbehandlung in Java
- Exceptions

Teil I der Vorlesung PROG 2

Entwicklung grafischer Schnittstellen

Graphical User Interface

– Layout-Manager –

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Möglichkeiten des Layouts

Intention

- Anordnung von GUI-Komponenten innerhalb eines Fensters
- feste Position für dynamisch veränderliche Fenster ungeeignet

Umsetzung

- automatische Anordnung mit Hilfe von Layout-Managern
- Festlegung mit *setLayout*, Elemente mit *add* hinzufügen

Beispielmanager

- *FlowLayout:* Elemente nebeneinander
- *GridLayout:* Elemente im Gitter
- *BorderLayout:* Elemente in vordefinierten Bereichen
- *CardLayout:* mehrere Unterdialoge im Fenster
- *GridBagLayout:* *GridLayout* um Bedingungsobjekte erweitert

FlowLayout

Einordnung

- einfachster Layout-Manager von AWT
- Anordnung von links nach rechts, von oben nach unten
- Reihenfolge beim Hinzufügen entscheidend
- Verändern der Fenstergröße $\Rightarrow$ neue Anordnung

Umsetzung

```
FlowLayout(int align, int hgap, int vgap)
```

- align: zentriert, links- oder rechtsbündig

```
FlowLayout.CENTER, FlowLayout.LEFT, FlowLayout.RIGHT
```

- hgap: horizontaler Abstand zwischen Komponenten
- vgap: vertikaler Abstand zwischen Komponenten

GridLayout

Matrixartige Anordnung

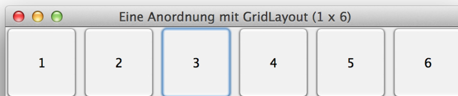
- Konstruktor

```
new GridLayout(rows, cols)
```

- Elemente zeilenweise hinzufügen
- bei zu wenig Elementen, bleiben Felder leer
- bei zu viel Elementen, wird Spaltenzahl erhöht

  $\Rightarrow$ flexible Handhabung bei $\text{rows} = 0$ oder $\text{cols} = 0$

Beispiel: 6 Buttons im GridLayout




```
public class GridLayoutBsp extends JFrame {
  private JButton buttons[];
  private final String names[] = {"1","2","3","4","5","6" };
  private GridLayout gridLayout;

  public GridLayoutBsp() {
    super( "Eine Anordnung mit GridLayout (1 x 6)" );
    gridLayout = new GridLayout( 1, 6, 5, 5 );
    setLayout( gridLayout );
    buttons = new JButton[ names.length ];
    for (int count = 0;count < names.length; count++) {
              buttons[count] = new JButton(names[count]);
              add(buttons[count]);
    }
}}
```

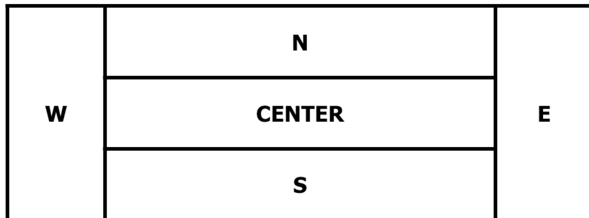
BorderLayout

Anordnung mit 5 definerbaren Bereichen

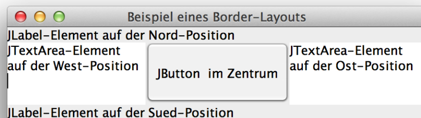
- *NORTH, WEST, SOUTH, EAST, CENTER*
- Hinzufügen zu einem Bereich mit

```
container.add(element, BorderLayout.WEST)
```

- für leere Bereiche gilt: Höhe oder Breite ist 0
- automatische Breitenberechnung für Randbereiche, der Rest wird dem Zentrum zugeordnet

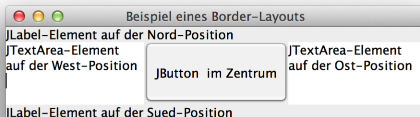


Beispiel: 5 GUI-Elemente im BorderLayout



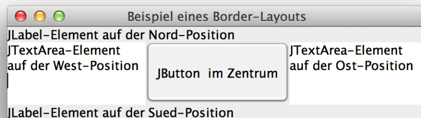

```
public class BorderLayoutBsp extends JFrame {
 private JButton button;
 private JTextArea textAreaWest, textAreaEast;
 private JLabel labelNorth, labelSouth;

 public BorderLayoutBsp() {
  super("Beispiel eines BorderLayouts");
  labelNorth = new JLabel("JLabelElement auf NordPosition");
  add(labelNorth, BorderLayout.NORTH);







 }}
```

Beispiel: 5 GUI-Elemente im BorderLayout



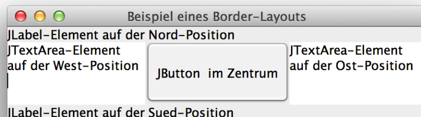

```
public class BorderLayoutBsp extends JFrame {
 private JButton button;
 private JTextArea textAreaWest, textAreaEast;
 private JLabel labelNorth, labelSouth;

 public BorderLayoutBsp() {
  super("Beispiel eines BorderLayouts");
  labelNorth = new JLabel("JLabelElement auf NordPosition");
  add(labelNorth, BorderLayout.NORTH);
  labelSouth = new JLabel("JLabelElement auf SuedPosition");
  add(labelSouth, BorderLayout.SOUTH);





 }}
```

Beispiel: 5 GUI-Elemente im BorderLayout




```
public class BorderLayoutBsp extends JFrame {
 private JButton button;
 private JTextArea textAreaWest, textAreaEast;
 private JLabel labelNorth, labelSouth;

 public BorderLayoutBsp() {
  super("Beispiel eines BorderLayouts");
  labelNorth = new JLabel("JLabelElement auf NordPosition");
  add(labelNorth, BorderLayout.NORTH);
  labelSouth = new JLabel("JLabelElement auf SuedPosition");
  add(labelSouth, BorderLayout.SOUTH);
  textAreaWest = new JTextArea(textWest,9,12);
  add(textAreaWest, BorderLayout.WEST);



 }}
```

Beispiel: 5 GUI-Elemente im BorderLayout




```
public class BorderLayoutBsp extends JFrame {
 private JButton button;
 private JTextArea textAreaWest, textAreaEast;
 private JLabel labelNorth, labelSouth;

 public BorderLayoutBsp() {
  super("Beispiel eines BorderLayouts");
  labelNorth = new JLabel("JLabelElement auf NordPosition");
  add(labelNorth, BorderLayout.NORTH);
  labelSouth = new JLabel("JLabelElement auf SuedPosition");
  add(labelSouth, BorderLayout.SOUTH);
  textAreaWest = new JTextArea( textWest,9,12);
  add(textAreaWest, BorderLayout.WEST);
  textAreaEast = new JTextArea( textEast,9,12);
  add(textAreaEast, BorderLayout.EAST);
  button = new JButton("JButton im Zentrum");
  add(button, BorderLayout.CENTER); }}}
```

GritBagLayout

Einordnung

- sehr flexibel, aber auch kompliziert
- Umgang mit Elementen unterschiedlicher Größe
- Reihenfolge des Hinzufügens ist nicht für Layout relevant

Koordinaten zur Anordnung

- rechteckiges Gitter von Zellen
- Zellen können unterschiedlich groß sein
- flexible Platzierung, z.B. können GUI-Elemente über mehrere Zellen gehen

Syntax

```
layout = new GridBagLayout();
```

```
constraints = new GridBagConstraints();
```

Constraints im GritBagLayout

- In welcher Spalte liegt der linke Rand eines GUI-Elementes?

```
int gridx/gridy
```

- Über wie viele Zeilen erstreckt sich das Element?

```
int gridwidth/gridheight
```

- An welcher Kante der Zelle wird das Dialogelement befestigt?

```
int anchor
```

- Was passiert bei Größenveränderung?

```
int fill
```

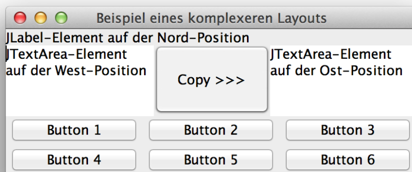
- Wie ist die Mindestgröße definiert?

```
int ipadx/ipady
```

- Layout-Schachtelung: z.B. nur eine Zelle mit GridLayout

  $\Rightarrow$ **Umsetzung beliebig komplexer Layouts möglich!**

Beispiel: Geschachteltes Layout mit JPanel




```
public class PanelFrame extends JFrame {
 private JPanel buttonJPanel;
 private JButton buttons[];
 ...
 private JLabel labelNorth;

 public PanelFrame() {
  super("Beispiel eines komplexeren Layouts");
  labelNorth = new JLabel("JLabelElement auf NordPosition");
  add(labelNorth, BorderLayout.NORTH);
  ...
  buttonJPanel = new JPanel();
  buttonJPanel.setLayout(new GridLayout(2,3));
  buttons = new JButton[6]; ...
  add(buttonJPanel, BorderLayout.SOUTH); }}
```

Weitere Layoutmanager

AWT

- *CardLayout:* Kartenstapel, bei dem nur das oberste Element sichtbar ist

Swing

- *BoxLayout:* Boxen werden hintereinander dargestellt und auf gleicher Höhe ausgerichtet
- *OverlayLayout:* Elemente werden übereinander positioniert
- *JTabbedPane:* Elemente sind durch Reiter identifizierbar, Element mit aktiven Reiter wird dargestellt
- *GroupLayout:* Vertikales und horizontales Layout werden unabhängig behandelt

Teil I der Vorlesung PROG 2

Entwicklung grafischer Schnittstellen

Graphical User Interface

– Komponenten in einer GUI –

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Ausgabe von Text mit Labeln in einer GUI

Features

- Setzen von Text, Position, Verbindung mit Icons
- Tooltips (Hilfeanzeige, wenn Mauszeiger überm Label ist)

Beispiel

```
lab = new JLabel(); // Label erzeugen
lab.setText("Darzustellender Text");
lab.setIcon(BspIcon);
lab.setHorizontalTextPosition(SwingConstants.CENTER);
lab.setVerticalTextPosition(SwingConstants.BOTTOM);
lab.setToolTipText("Label bei Mousezeiger");
```

Ein- und Ausgabe von Text mit Textfeldern

Features

- realisiert durch *JTextField*
- kann editierbar oder schreibgeschützt gesetzt werden
- Betätigen der *ENTER*-Taste löst Ereignis aus
- *JPasswordField* ist Erweiterung für Passwörter (Zeichen werden versteckt)

Beispiel

```
t = new JTextField("Unveränderlicher Text",21);
// false = unveränderbar
t.setEditable(false);
```

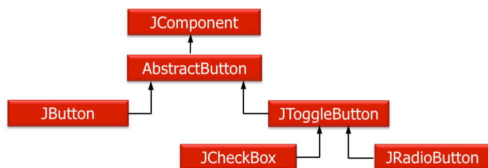
Buttons in einer GUI

Klassischer Button

- mit *JButton*

Wechselnde Button

- Zustand bleibt (*CheckBox*)
- Deaktiviert aktuellen Zustand (*RadioButton*)



Beispiel

```
private JButton button1, button2
button1 = new JButton("Nur Text als Label");
button2 = new JButton("Beispiel mit Icon",bspIcon);
```

CheckButtons und RadioButtons

CheckButton

➜ Erzeugung mit *JCheckBox* analog zu *JButton*

RadioButton

➜ zweistufige Erzeugung

1. Erstellung einzelner Elemente

```
private JRadioButton button1, button2;
button1 = new JRadioButton("Bold",false);
button2 = new JRadioButton("BoldItalic",true);
```

2. Zusammenfassung zu logischer Gruppe

```
private ButtonGroup radioGroup;
radioGroup = new ButtonGroup();
radioGroup.add(button1);
radioGroup.add(button2);
```

Teil I der Vorlesung PROG 2

Entwicklung grafischer Schnittstellen

Graphical User Interface

– Ereignisbehandlung in einer GUI –

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Ereignisbehandlung (Event handling)

Intention

- Festlegung, was passieren soll, wenn GUI-Eingaben erfolgen

Konzepte zur Umsetzung

- Erzeugung von Ereignissen (*Events*, z.B. Maus klicken)
- Belauschen der Ereignisse mit speziellen Objekten (*Listener*)
- Definition von *Eventcode* (Ereignisbehandlung)

Prinzipielles Vorgehen

1. *Events* klassifizieren
2. *Listener* definieren und mit GUI-Objekten assoziieren
3. *Listener* erkennt *Event*, ermittelt Kategorie und führt *Eventcode* aus

Beispiel: Maus-Ereignisbehandlung

Beteiligte Klassen/Interfaces

- Ereignisklasse: *MouseEvent*
- Listener-Interface: *MouseListener*
- Registrierungsmethode: *addMouseListener*
- Mögliche Ereignisquelle: *Component*

Methoden zur Ereignisbehandlung aus MouseListener

- *mouseClicked*: Maustaste gedrückt und losgelassen
- *mouseEntered*: Mauszeiger im Bereich einer Komponente
- *mouseExited*: Mauszeiger außerhalb des Komponentenbereichs
- *mousePressed*: Maustaste gedrückt
- *mouseReleased*: Maustaste losgelassen

Entwurfsmuster zur Ereignisbehandlung

Varianten

1. Fensterklasse implementiert erforderliche Interfaces für EventListener und registriert sich selbst bei Ereignisquellen
2. Definition von lokalen oder anonymen Klassen in der Fensterklasse, um EventListener zu implementieren
3. Trennung von GUI-Code und Ereignisbehandlung in komplett separaten Klassen
4. Überlagerung spezieller Methoden der Komponentenklasse, die für Empfangen/Verteilen von Nachrichten erforderlich sind

Trennung von GUI-Code und Ereignisbehandlung ist im Sinne des MVC-Patterns für größere Programme unbedingt zu empfehlen!

Variante 1: Entwurf zur Ereignisbehandlung

Fenster-Klasse implementiert EventListener-Interface

⇒ einfacher Zugriff auf alle Methoden in einer Klasse

⇒ Nachteil: unübersichtlich und viele leere Methoden

```
class MyFrame extends Frame implements KeyListener {
  public static void main(String[] args) { ... }
  public MyFrame() {
          ...;  addKeyListener(this);
          }
  public void keyPressed(KeyEvent event) {
          if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
              setVisible(false);
              dispose();
              System.exit(0);
            }
          }
  public void keyReleased(KeyEvent event) {  }
  public void keyTyped(KeyEvent event) {  }
}
```

Adapter-Klassen zur Ereignisbehandlung

Intention

- Häufig sind nicht alle Ereignisbehandlungen aus einem EventListener-Interface für die Anwendung interessant
- *Adapterklassen* implementieren für zugehörige Interfaces leere Methoden
- Eigene *Listener*-Klassen können von Adapter-Klassen erben und nur für die Anwendung interessante Methoden definieren

```
public abstract class WindowAdapter
    implements WindowListener, ... {
      public void windowOpened (WindowEvent e) {  };
      public void windowClosing (WindowEvent e) {  };
      public void windowClosed (WindowEvent e) {  };
      public void windowActivated (WindowEvent e) { };
      ...
  }
```

Variante 2: Entwurf zur Ereignisbehandlung

Fenster-Klasse enthält lokale Klassen

- ➜ Kapselung der Event-Funktionalität in einer lokalen Klasse, die von *Adapterklasse* erbt
- ➜ Vorteil: nur benötigte Methoden zu implementieren, aber Ereignisquelle/Ereignisbearbeitung in der selben Klasse

```
class MyFrame extends Frame {
  public static void main(String[] args) { ... }
  public MyFrame() {
        ...;
        addKeyListener(new MyKeyListener());}
  private class MyKeyListener extends KeyAdapter {
        public void keyPressed(KeyEvent event) {
          if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
           setVisible(false);dispose();System.exit(0);
  }}}}
```

Variante 2: Entwurf zur Ereignisbehandlung

Fenster-Klasse enthält anonyme Klasse

- ➜ Kapselung der Event-Funktionalität in einer anonymen Klasse, die von *Adapterklasse* erbt
- ➜ Vorteil: weniger Code, aber noch weniger Kapselung, keine Trennung von GUI und Anwendungslogik

```
class MyFrame extends Frame {
  public static void main(String[] args) { ... }
  public MyFrame() { ...;
    addKeyListener(
      new KeyAdapter() {
          public void keyPressed(KeyEvent event) {
            if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
              setVisible(false);dispose();System.exit(0);}}
          });
  }
```

Variante 3/4: Entwurf zur Ereignisbehandlung

Trennung von GUI- und Anwendungscode

- Implementierung der Ereignisbehandlung in zwei separaten Klassen ⇒ bessere Modularisierung des Codes
- Verbindung durch Aufruf der Methode *addKeyListener*

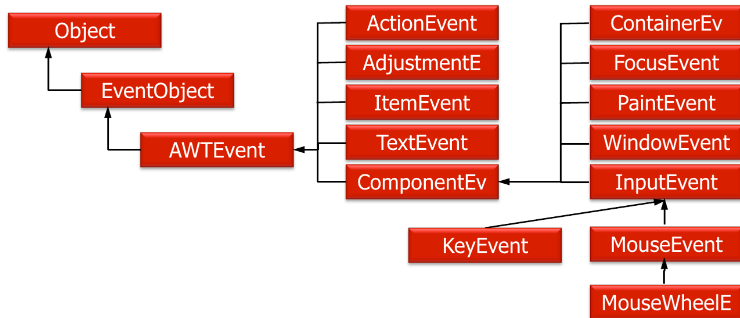
Überlagerung Event-Handler in den Komponenten

- jede Ereignisquelle besitzt Methoden, die für Aufbereiten/Verteilen von Nachrichten zuständig sind
- Weiterreichen von Nachrichten mit der Methode *processEvent*
- Verteilung der Nachrichten anhand ihres Typs an spezialisierte Methoden (Name wird vom Typ der zugehörigen Ereignisklasse abgeleitet)

Klassifikation von Ereignissen in Java

Wie unterscheidet man die vielen GUI-Events?

- Ereignistypen in java.awt.event und javax.swing.event

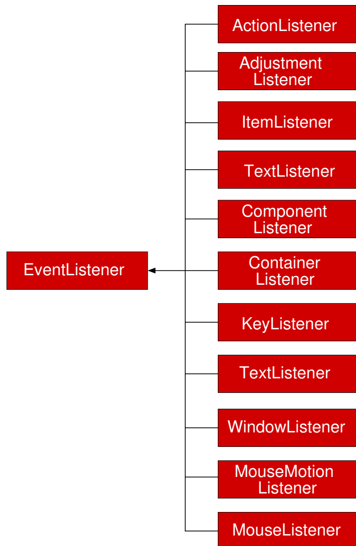


- Event-erzeugte Objekte benötigen passende Listener-Objekte

Klassifikation der EventListener in Java

Zuordnung Event und Listener

- jedes *JComponent*-Objekt besitzt Variable *listenerList* (Referenz auf Objekt der Klasse *EventListenerList*)
- *listenerList* verwaltet alle Event-Listener
- tritt Event auf, sucht JVM daraus passendes Listener-Objekt
- z.B. *MouseEvent* ⇒ *MouseListener* oder *MouseMotionListener*



Teil I der Vorlesung PROG 2

Entwicklung grafischer Schnittstellen

Graphical **U**ser **I**nterface

Ausnahmebehandlung
(Exception Handling)

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Ausnahmen (Exceptions)

Intention

- Konzept zur *kontrollierten Reaktion auf Laufzeitfehler* (robuste Softwareentwicklung)
- Mechanismen zur sauberen Trennung von Programmcode und Code zur Fehlerbehandlung

Arten von Laufzeitfehlern

1. logische Programmierfehler (z.B. Division durch 0)
2. fehlerhafte Bedienung eines korrekten Programms (z.B. kein Leserecht für ausgewählte Datei)
3. Probleme im Java-Laufzeitsystem

Vorbereitungen für eine Ausnahmebehandlung

Vorgehensweise

1. *Erstellung eines Fehlermodells*: Identifikation von Ausnahmesituationen, Festlegung angemessener Reaktionen
2. *Fehlererkennung*: an diversen Programmstellen, manchmal eindeutig, häufig aber nur Vermutungen
3. *Fehlerbehandlung*: an anderer Stelle im Programm umgesetzt

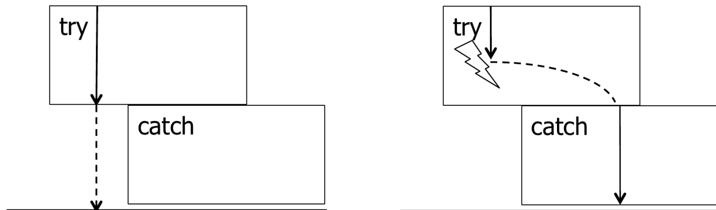
Varianten der Fehlerbehandlung

- *Maskieren*: Verbergen/Abschwächen eines Fehlers (z.B. durch Wiederholen gescheiterter Aktionen)
- *Tolerieren*: Fehleranzeige und Benutzerreaktion abwarten (z.B. Web-Site nicht erreichbar)
- *Wiederherstellen*: Rekonstruktion konsistenter Systemzustände

Java-Prinzip zur Ausnahmebehandlung

Try-Catch-Konzept

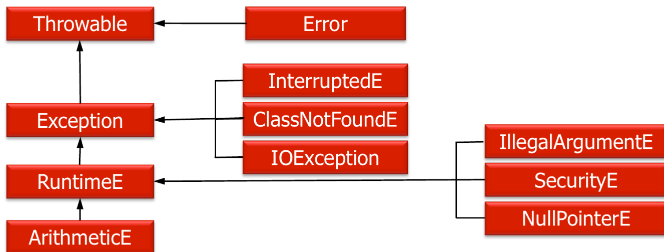
- Unterteilung des Codes in *try*- und *catch*-Blöcke
- (potenziell fehlerhafter) Programmcode in *try*-Blöcken
- Fehlerbehandlung in *catch*-Blöcken
- nur im Fehlerfall wird *catch*-Block abgefragt



Auszug der Exception-Hierarchie

Java-Unterstützung

- Klassen zur Modellierung von Exceptions verfügbar
- Auftreten eines Fehlers ⇒ Erzeugen eines passenden Exception-Objekts



Abstrakte Formulierung von Try und Catch

```
try {
        Programmfragment mit potenziellen Fehlern
    }
```

Abstrakte Formulierung von Try und Catch

```
try {
        Programmfragment mit potenziellen Fehlern
    }

catch (Exception-Art 1 e1) {
        Fehlerbehandlung 1
    }
...
catch (Exception-Art n en) {
        Fehlerbehandlung n
    }
```

Abstrakte Formulierung von Try und Catch

```
try {
        Programmfragment mit potenziellen Fehlern
    }

catch (Exception-Art 1 e1) {
        Fehlerbehandlung 1
    }
...
catch (Exception-Art n en) {
        Fehlerbehandlung n
    }

finally {
        Abschlussaktivitaet (falls noetig)
    }
```

Hinweis: Nur die erste passende catch-Anweisung wird ausgeführt, alle nachfolgenden (bis auf *finally*) werden ignoriert!

Finally-Blocks

Intention

- Aufräumarbeiten nach Abschluss, um nicht mehr benötigte Ressourcen freizugeben
- gemeinsamen Code aus *catch*-Anweisungen und *try*-Anweisung zusammenfassen (redundanten Code vermeiden)

Wann wird ein finally-Block ausgeführt?

1. Exception ist nicht aufgetreten und *try*-Block wurde erfolgreich ausgeführt
2. Exception ist aufgetreten und passende *catch*-Anweisung wurde ausgeführt
3. Exception ist aufgetreten, aber keine passende *catch*-Anweisung gefunden

Deklaration von möglichen Exceptions

Verpflichtung

- Deklaration möglicher Exceptions ist verpflichtend, wenn sie nicht im *try-catch*-Block einer Methode behandelt werden

⇒ Erweiterung der Methodensignatur mit *throws*-Statement

```
return_type Methodenname (Parameter)
            throws Exception1 ,... ,ExceptionN
```

Ausgenommen sind ...

- Exceptions, die überall im Programm auftreten können, z.B. Division durch 0 oder nicht definierte Array-Zugriffe
- verpflichtende Behandlung würde Code unleserlich machen

⇒ betrifft alle Unchecked Exceptions in Java, z.B. die Subklassen von *RuntimeException* und *Error*

Auswertung einfacher try-catch-Blöcke

Regeln

1. erste passende *catch*-Anweisung wird genommen, alle anderen *catch*-Anweisungen ignoriert
2. keine passende *catch*-Anweisung ⇒ Abbruch des Programms

```
class ButtonsListener implements ActionListener {
 private String laf;
 public void actionPerformed(ActionEvent e) {
   if(e.getSource() == metalButton) {
       laf = "javax.swing.plaf.metal.MetalLookAndFeel"; }
   ...
   try { UIManager.setLookAndFeel(laf); }
   catch (UnsupportedLookAndFeelException ue) {
       System.err.println(ue.toString()); }
   catch (ClassNotFoundException ce) {
       System.err.println(ce.toString()); } ...
```

Auswertung geschachtelter try-catch-Blöcke

Aufbau

- Methode mit *try-catch*-Block ruft weitere Methoden mit eigenen *try-catch*-Blöcken auf
- Stack-basierte Implementierung, zuletzt geöffneter *try-catch*-Block bildet oberstes Element auf dem Stack
- Verlassen eines *try-catch*-Blockes ⇒ Entfernen des *try-catch*-Blocks vom Stack

Regeln

1. Exceptionbehandlung: Suche nach passender *catch*-Anweisung im gesamten Stack mit oberstem Element beginnend
2. kein passender *catch*-Block ⇒ Abbruch des Programms

Fehlerverfolgung in try-catch-Blöcken

Problem

- Fehlerverfolgung und Debugging in verschachtelten *try-catch*-Blöcken schwierig

Hilfsmittel zur Ausgabe

- Fehlerbeschreibung einer Exception

```
String getMessage()
```

- Informationen zur Stack-Darstellung

```
StackTraceElement[] getStackTrace()
```

- Ausgabe von Exception und Trace

```
void printStackTrace()
```

- Zeile, Methoden- oder Klassenname der Exception

```
getLineNumber(), getMethodName(), getClassName()
```

Wie benutze ich Exceptions richtig?

1. Einsatz von möglichst spezifischen Exceptions, keine sinnlosen Fehlerbehandlungen, wie z.B.

```
catch(Throwable e) {System.err.println("Exception");}
```

2. leere *catch*-Blöcke vermeiden
3. *RuntimeExceptions* nicht deklarieren oder abfangen, sondern durch geeigneten Code das Auftreten verhindern
4. bereits definierte Fehlerklassen nutzen und nur sparsam neue Fehlerklassen einführen
5. Dokumentation unter Verwendung von @throws aus Javadoc

```
/**
* @throws NewException
*     Erklaerender Text
**/
public void method (...) throws NewException { ... }
```

RuntimeException vs. Error

RuntimeException

- Programmierfehler, die vermeidbar sind (z.B. Division durch 0)

Error

- Probleme, die eigentlich nicht auftreten sollten (z.B. Fehler in der JVM)