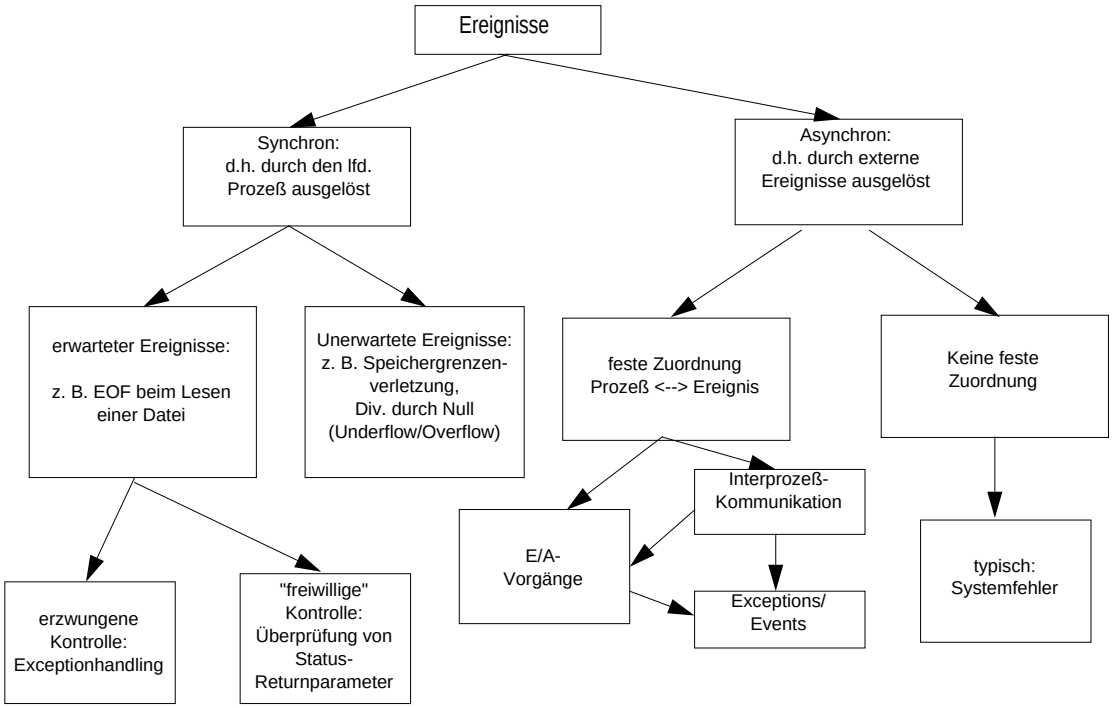
Grundlagen der Informatik für Ingenieure I

12. Exceptions

12 Exceptions

- Java unterscheidet drei Arten von Ereignissen:
  - ***Events:*** Interaktionsereignisse mit dem *GUI* (siehe Kapitel 10)
  - ***Errors:*** Interne schwerwiegende Fehler in der *JVM* bzw. vom Benutzerprogramm zur Laufzeit nicht behandelbare Fehler
  - ***Exceptions:*** alle anderen Ereignisarten, unterteilt in
    - *Runtime Exceptions*
    - *I/O-Exceptions*
    - sonstige *Exceptions*
- Dieses Kapitel befasst sich mit Exceptions und Errors

12.1 Klassifikation von Ereignissen



12.1 Klassifikation von Ereignissen

- Abgesehen von einigen wenigen “harten” Fehlerklassen, die zur Beendigung des Programms führen, 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
- Die Klasse der “harten” Fehler auf Betriebssystemebene führen zur Beendigung des Programms.

12.1 Klassifikation von Ereignissen

- In der Java-Umgebung werden solche "harten" Fehler durch den Interpreter “abgefangen”.
  - Eine *“Null-Pointer-Exception”* (z.B. für eine Objektvariable wird kein Objekt mit new erzeugt) wäre potentiell eine Speicher-grenzenverletzung.
  - *Arithmetic Exception* (Div. durch Null)
  - Darüberhinaus generiert die Java Laufzeitumgebung *(JVM)* weitere Lauf-zeitfehler, wenn objekt- bzw. java-spezifische Ereignisse auftreten, die erst zur Laufzeit erkannt werden können, wie z. B.:
    - *Illegal Argument Exception*
    - *Illegal Thread State Exception*

12.1 Klassifikation von Ereignissen

- Sämtliche anderen (asynchronen) Ereignisse sind **Exceptions** und können nur dann auftreten, wenn vom Programm her irgendwann Vorgänge angestoßen wurden, die zu solchen Ereignissen führen können.
- 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

12.1 Klassifikation von Ereignissen

- In einigen seltenen Fällen kann man auch Ereignisse bewußt 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.

12.1 Klassifikation von Ereignissen

- Java unterscheidet drei Arten von Ereignisse:
  - ***Events:*** Interaktionsereignisse mit dem *GUI* (Kapitel 10)
  - ***Errors:*** Interne schwerwiegende Fehler in der *JVM* bzw. vom Benutzerprogramm zur Laufzeit nicht behandelbare Fehler
  - **Exceptions:** alle anderen Ereignisarten, unterteilt in
    - *Runtime Exceptions*
    - *I/O-Exceptions*
    - sonstige *Exceptions*
    -

12.2 Exceptions

- Dieser Abschnitt widmet sich der Behandlung der *Exceptions* und *Errors*:

  ◆ *Errors:*

  - *Errors* sind “interne” Fehlersituationen, z. B. in der JavaVM, auf die ein Anwendungsprogramm nicht geeignet reagieren kann - sie sollten eigentlich nicht auftreten.

  ◆ *Exceptions:*

  - Exceptions (Ausnahmen) werden durch Ausnahmesituationen bei der Programmausführung ausgelöst.
  - Sie können "aufgefangen" werden und damit Programmabrüche verhindert werden.

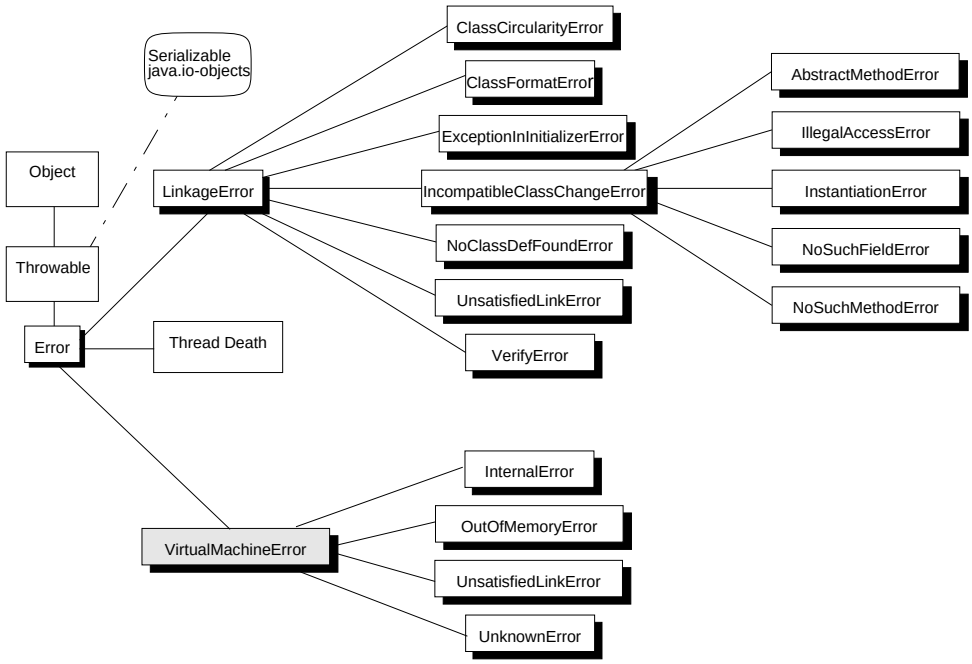
12.2 Exceptions

◆ Exceptions können eingeteilt werden in:

- allgemeine *Exceptions*
- *Runtime Exceptions*
- *I/O - Exceptions*
  Diese Klasse ist im Rahmen dieser Vorlesung von besonderer Bedeutung. (Kapitel 13: Java-I/O-System)

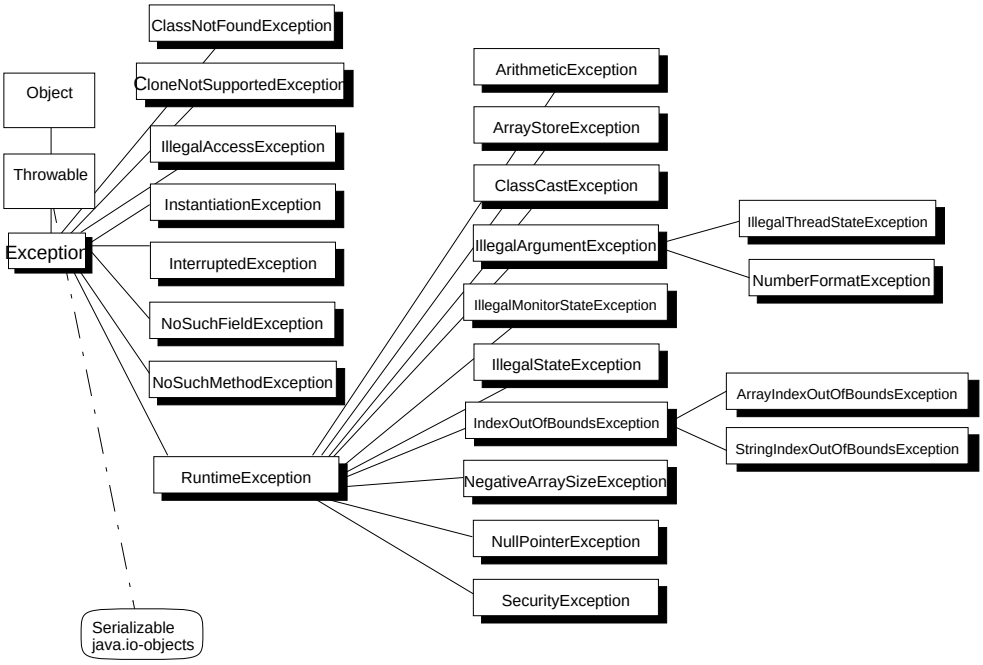
12.2 Exceptions

- *Error* - Klassenhierachie im package *java.lang:*



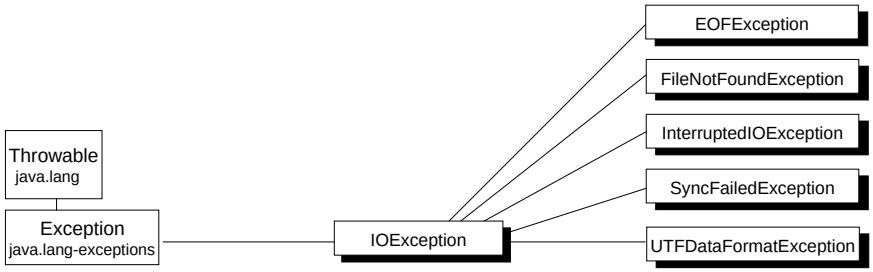
12.2 Exceptions

- *Exception* - Klassenhierachie im package *java.lang*:



12.2 Exceptions

- *I/O - Exceptions:*



12.2 Exceptions

- In Java können *Exceptions* “geworfen”*(thrown*) werden, d.h. sie werden nicht berücksichtigt und es werden keine geigneten Maßnahmen ergriffen.
- Ein gegen Fehlersituationen robustes Programm “fängt” *Exceptions* (nicht alle!) auf *(catch)* und leitet geeignete Maßnahmen ein.
- Im Gegensatz zu den meisten anderen Programmiersprachen, **erzwingt** der Java-Compiler die Behandlung einer Reihe möglicher *Exceptions.*
- In einem objektorientierten System ist (natürlich) eine *Exception* ein Objekt der *Throwable class* als Superklasse.

12.3 Exceptionhandling

- *Exceptionhandling* wird strukturiert:
  - in einen Teil, der den Code einschließt, der eine *Exception* auslösen könnte ***(try clause)***:
    - Zum Beispiel:

```
try { Thread.sleep( 1000 ) }
```

oder:

```
try {
    while ( numbytes <= myBuffer.length ) {
            myInputStream.read( myBuffer );
            numbytes++;
    }
}
```


12.3 Exceptionhandling

- ...und einen zweiten Teil ***(catch clause)***, der den Code enthält, der die Maßnahmen einleitet, die bei Auftreten des entsprechenden Ereignisses notwendig sind:
  - Zum Beispiel;

```
try { Thread.sleep( 1000 ) }
catch( InterruptedException e ) {}
```

oder:

```
try {
    while ( numbytes <= myBuffer.length ) {
            myInputStream.read( mybuffer );
            numbytes++;
        }
}
catch ( IOException e ) {
    System.out.println( “Fehlermeldung” )
    //..retten was zu retten ist - Code
}
```

12.3 Exceptionhandling

- Eine Exception kann "geworfen" werden mit "***throws":***

  - Beispiel:

```
public static void main( String args[] )
throws java.io.IOEXceptions {

    int eingabeWert;

    BufferedReader instream = new BufferedReader( new
            InputStreamReader( System.in ) );

    eingabeWert = Integer.parseInt( instream.readLine() );
}
```

  - Bei ***readLine()*** kann IO-Fehler auftreten, muß behandelt werden mit ***try*** oder geworfen werden mit ***throws*** wie hier im Beispiel.

12.3 Exceptionhandling

- Ein vollständiges Beispiel: *Keyboard-Input*

```
import java.io.*;

//  KeyboardInput: Zeilenweise Eingabe einer Zahl als float


class  KeyboardInput {

  private BufferedReader instream;

  KeyboardInput() {

    instream = new BufferedReader( new
                          InputStreamReader( System.in ) );
  }
```

12.3 Exceptionhandling

```
public float getFloat()
     throws java.io.IOException {

     //   Falsche Eingabe kann 2 x wiederholt werden, bevor NaN als
     //   Wert zurueckgegeben wird.

     float floatValue = Float.NaN;
     String inputLine;
     int tryCounter = 0;

     System.out.println( "Geben Sie eine Zahl als float ein:" );

     while ( tryCounter < 3 ) {
       try {
            inputLine = instream.readLine();
            floatValue = Float.parseFloat( inputLine );
       }
       catch( NumberFormatException e ) {
            tryCounter++;
            System.out.println ( "Wrong Format, try again:" );
            continue;
       }
       break;
     }
     return floatValue;
   }
}
```


12.3 Exceptionhandling

◆ Testprogramm:

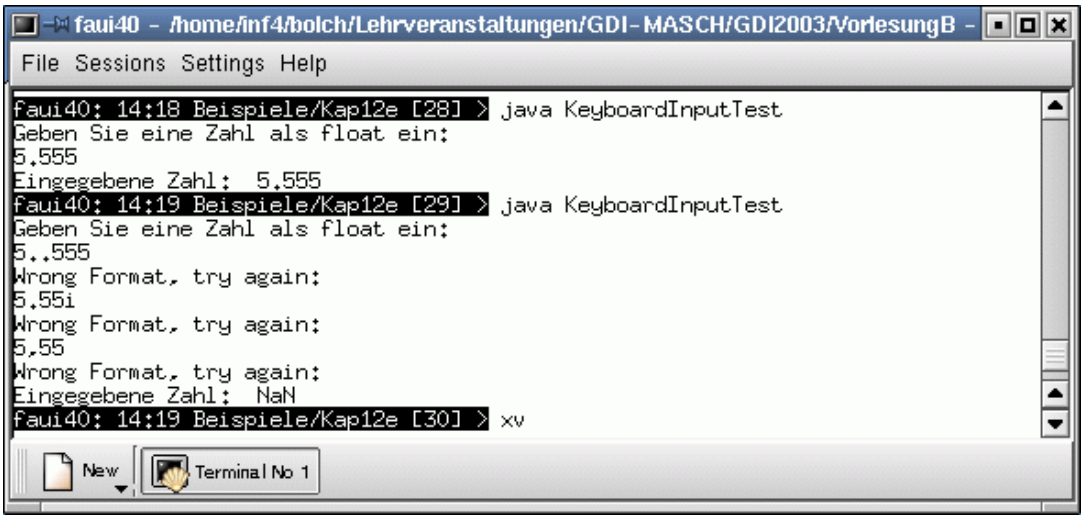
```
class KeyboardInputTest {

    public static void main( String args[] )
    throws java.io.IOEXception {

        KeyboardInput eingabe = new KeyboardInput();
        System.out.println( "Eingegebene Zahl:  "
                            +  eingabe.getFloat() );
    }
}
```

12.3 Exceptionhandling

- Ergebnis:



12.3 Exceptionhandling

- Man kann auch auf mehrere *Exceptions* reagieren.
- Dabei ist darauf zu achten, dass es auch im *Exceptionhandling* Hierachien gibt. So ist z. B. **`EOFException`** eine Untermenge von **`IOException`**.
- Die *catch-clauses* müssen so geordnet werden, daß die *Clause*, die die geringste Menge *Exceptions* abdeckt, zuerst behandelt wird.

```
try { protected code }

catch( FileNotFoundException eFileNotFound ) {
...
{
catch( EOFException eEndOfFile ) {
...
{
catch( IOException e ) {
// Behandlung aller sonstigen IO-Exceptions
...
{
```

12.3 Exceptionhandling

- Die ***finally clause*** dient dazu “abschließende” Arbeiten, die unabhängig von gerade aufgetretenen Typ einer *Exception* notwendig sind, zusammenzufassen:

```
try { protected code }
catch ( FileNotFoundException eFileNotFound ) {
...
{
catch ( EOFException eEndOfFile ) {
...
{
catch ( IOException e ) {
// Behandlung aller sonstigen IO-Exceptionms
...
{
finally {
    file.close();
}
```