Barry Linnert
Institut für Informatik

Freie Universität Berlin

Programmieren

Barry Linnert

Sommersemester 2020

Gliederung der heutigen Vorlesung

- Motivation und Konzept der Ausnahmebehandlung
- Ausnahmebehandlung in Java
  - Klasse *Throwable*
  - Ausnahmen auslösen
  - Ausnahmen behandeln
  - Ausnahmen weiterreichen
  - Ausnahmen definieren
  - Ausnahmen nachverfolgen
- Zusammenfassung

Freie Universität Berlin

Ausnahmebehandlung

MOTIVATION

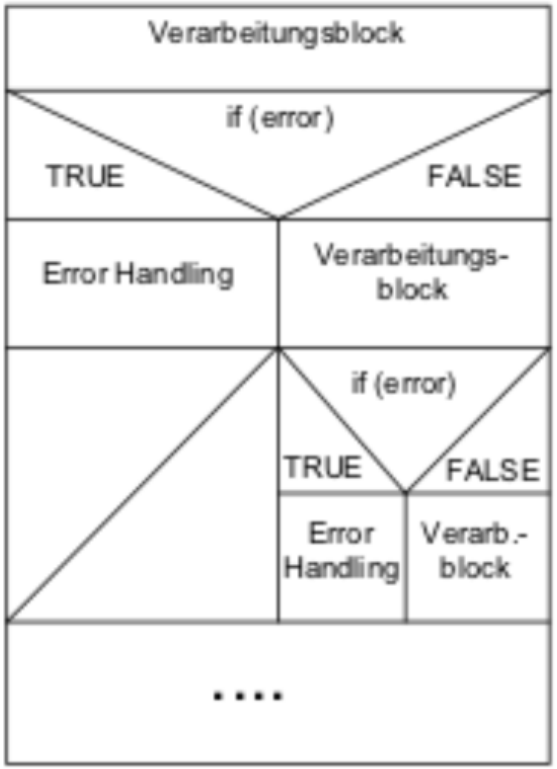
Was passiert, wenn bei einer Suche ein Element nicht gefunden wird?

Lösung 1:

- Wir können unsere Suchoperation so definieren, dass die null Konstante zurückgegeben wird, wenn ein Element nicht gefunden wird.

```
int suche(int[] a, int was){
   int n = a.length;
   for (int i = 0; i < n; i++)
       if (a[i] == was)
          return i;
   return -1;
}
```

“Klassische Fehlerbehandlung”



Mögliche Probleme

- Der Benutzer muss das Verhalten genau kennen, ansonsten ist der Fehler nicht verständlich.
- Vor lauter Fehlerbehandlung sieht man eigentliches Programm nicht mehr.
- Abfragen des Fehlercodes kann vergessen werden.
- Abfragen des Fehlercodes wird oft aus Bequemlichkeit nicht durchgeführt.
- Funktion kann neben Fehlercode kein anderes Ergebnis mehr liefern.

Was passiert, wenn bei einer Suche ein Element nicht gefunden wird?

Lösung 2:

- Wir zwingen den Benutzer, den Fall zu behandeln, sonst wird sein Programm nicht übersetzt.

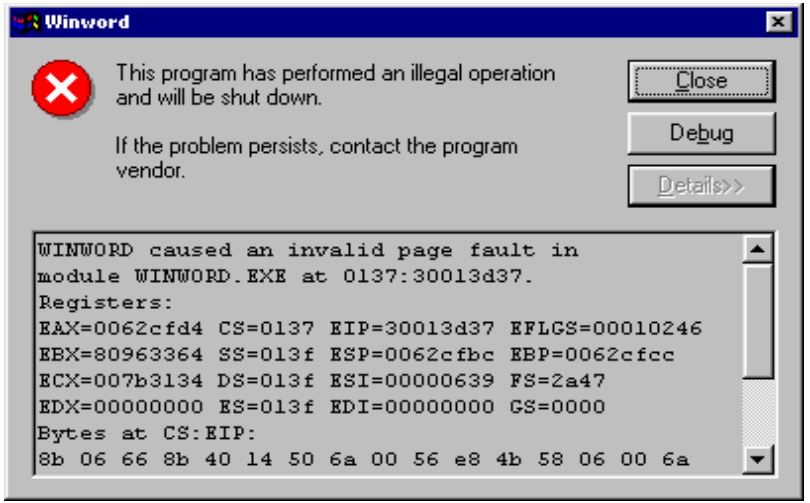
In **Programmiersprachen ohne Ausnahmebehandlung** muss beispielsweise das Ergebnis einer erfolglosen Suche durch einen vereinbarten Rückgabecode signalisiert werden.

Programmiersprachen mit Ausnahmebehandlung können Ausnahmen abgefangen und gesondert behandelt werden.

Fehler vs. Ausnahme

- Ein ***Fehler*** liegt vor, wenn der Programmablauf unerwartet (aus Gründen, die innerhalb oder außerhalb des Verantwortungsbereichs der Entwicklerin bzw. des Entwicklers liegen können) vom vorgesehenen Ablauf abweicht.

- ***Ausnahmen*** (engl.: *exceptions*) sind Sonderfälle, mit denen die Programmiererin bzw. der Programmierer oder das Programmiersystem aber rechnet.



Terminologie

- Zum sicheren (bzw. defensiven) Programmieren gehört es, Ausnahmen so weit wie möglich **abzufangen** (engl.: to **catch**).

- Ausnahmen können durch Laufzeitfehler (weitere Beispiele nächste Folie) verursacht, aber auch vom Programmierer bewusst veranlasst werden.

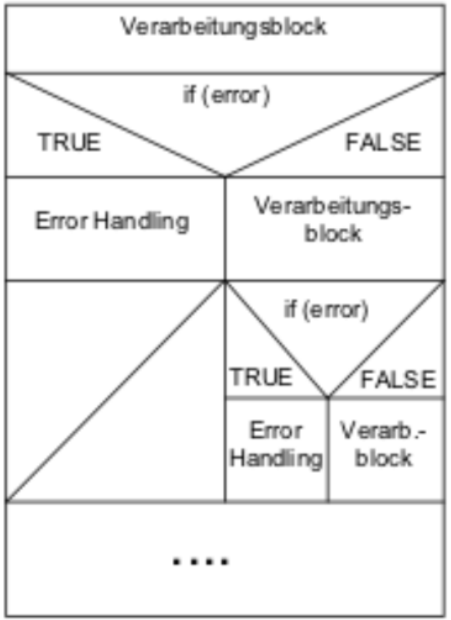
- Das **Auslösen** einer Ausnahme wird als (engl.) „to **throw** an exception“ bezeichnet.

Beispielursachen für Ausnahmen (Exceptions)

- Der Zugriff auf eine Referenz, die auf null zeigt
- Ungenügende Systemressourcen, wie ein Mangel an Speicherplatz
- Verletzung der Array-Grenzen
- Ein- und Ausgabeoperationen
- Versuch, in eine schreibgeschützte Datei zu schreiben
- Division durch 0

Ausnahmebehandlung (*engl.* Exception Handling)

- Ziel des Exception Handling ist es, **normalen und fehlerbehandelnden Code** übersichtlich **zu trennen** und Ausnahmesituationen sicher zu behandeln.



Klassische Fehlerbehandlung vs. Exception Handling

Ausnahmebehandlung

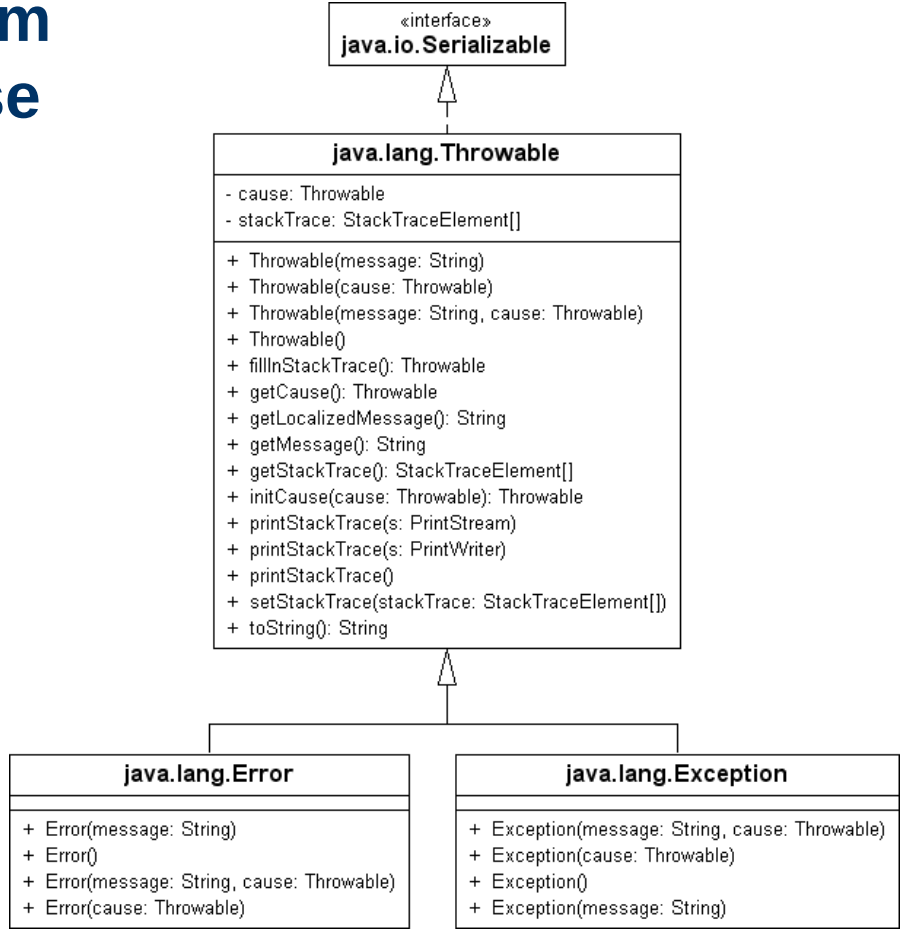
AUSNAHMEBEHANDLUNG (EXCEPTION HANDLING) IN JAVA

Konstruktiver Umgang mit Laufzeitfehlern

Sprachkonstrukte in Java ermöglichen es,

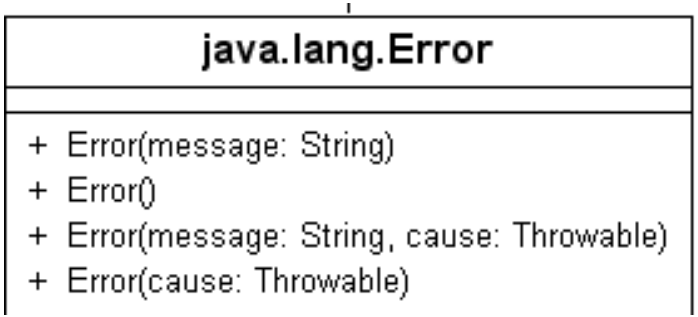
- das Auftreten von Ausnahmesituationen zu überwachen,
- dass im Fall einer Ausnahmesituation, bei deren Auftreten der normale Verlauf der Programmabarbeitung unterbrochen wird, vom Laufzeitsystem ein Ausnahme-Objekt erzeugt wird und
- die Kontrolle an spezielle Programmteile übergeben wird, die versuchen, den Laufzeitfehler aufzufangen und zu behandeln.

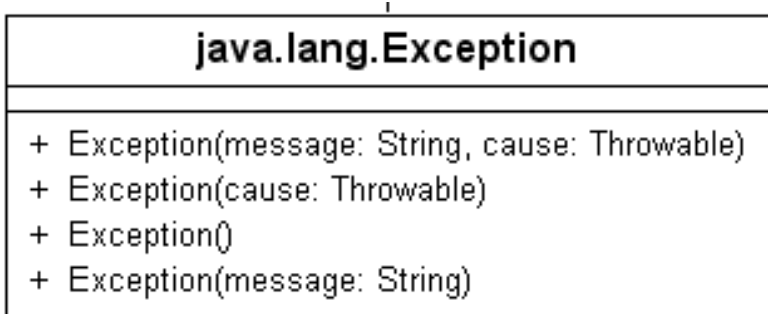
UML-Diagramm der Oberklasse *Throwable*



UML-Diagramm der wichtigen Oberklasse Throwable

Die Klasse `Error` und `Exception`





Die Klasse `Error`

- Ausnahmen der Klasse Error sollten zur Laufzeit eines Java-Programms eigentlich gar nicht auftreten. Ein Programm sollte in der Regel nicht versuchen, einen solchen Fehler abzufangen.

Die Klasse `Exception`

- Normalerweise lösen Java-Programme Ausnahmen aus, die von der Klasse Exception abstammen. Es handelt sich um Ausnahmen, die das Programm zur Laufzeit behandeln kann.

Beispiele für Exceptions (1)

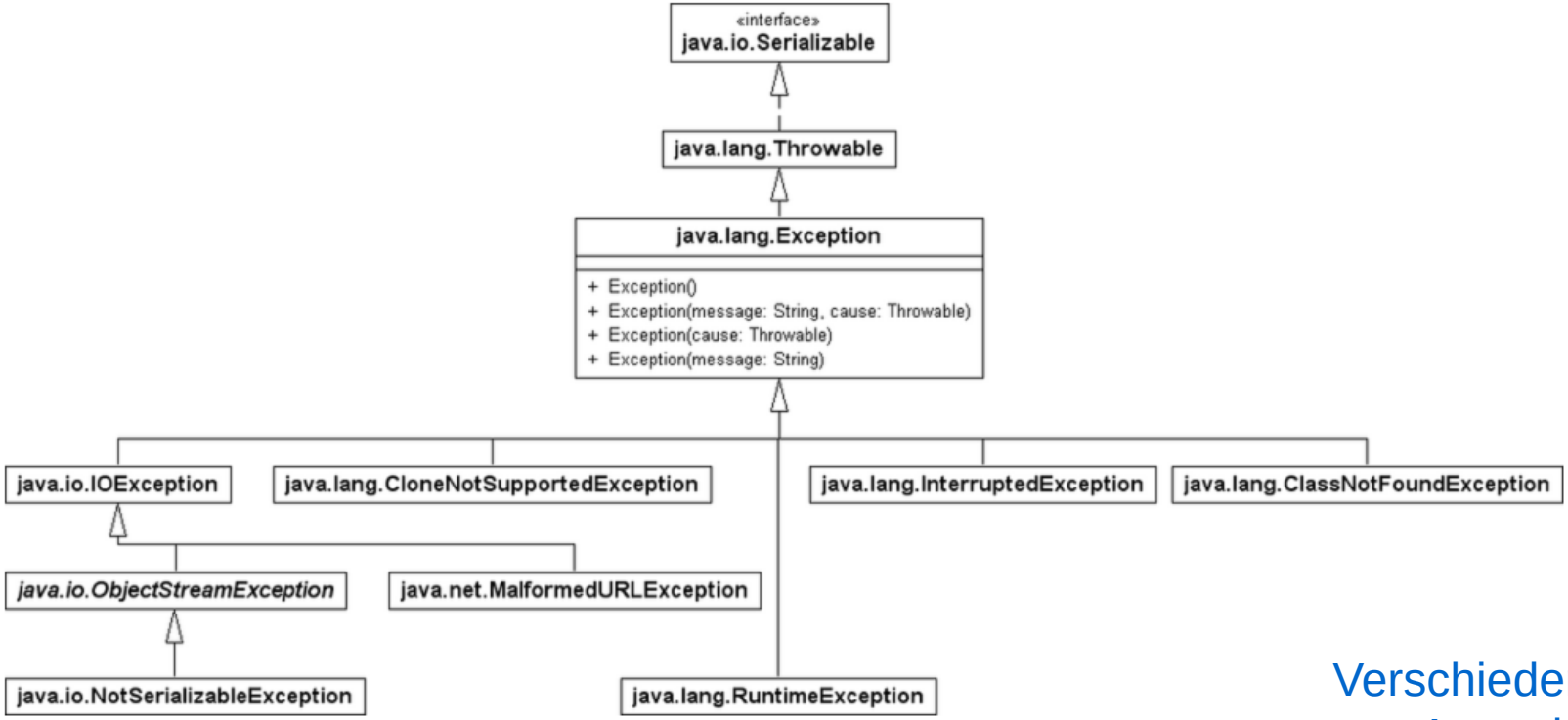
- *Exceptions vom Typ Error*

Exception	Ursache
`VirtualMachineError`	Die virtuelle Maschine ist defekt oder hat keine Ressourcen
`AnnotationFormatError`	Annotationen im Bytecode sind fehlerhaft
`AssertionError`	Eine Aussage wird als falsch bewertet
`IOError`	Eingabe- oder Ausgabemechanismen weisen ernsthafte Fehler auf

- *Exceptions, die von der Klasse Exception abgeleitet sind:*

Exception	Ursache
`CertificateException`	Probleme mit Zertifikaten sind aufgetreten
`CloneNotSupportedException`	Ein Objekt sollte kopiert werden, welches das Cloning aber nicht unterstützt
`DataFormatException`	Ein Datenformat ist fehlerhaft

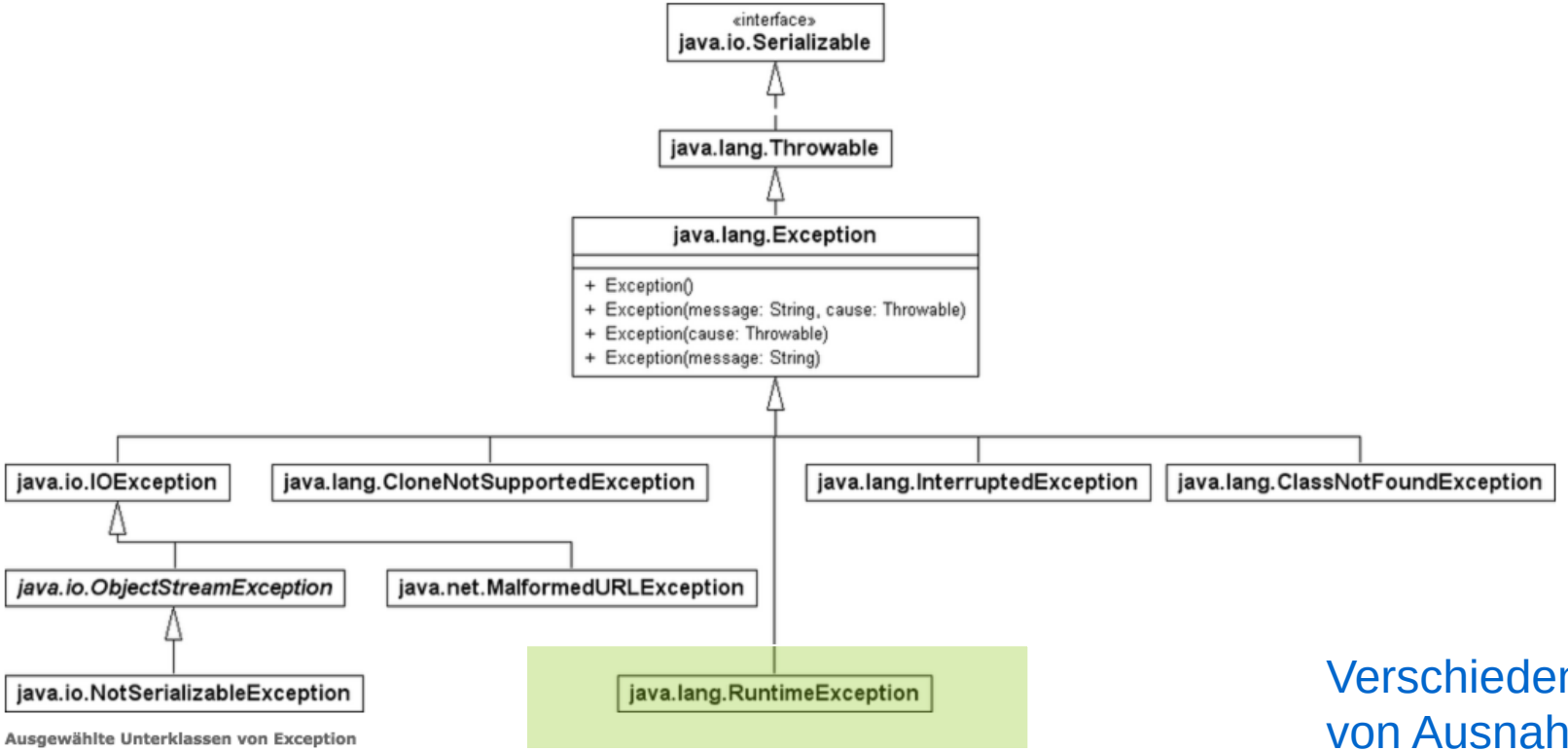
Ausgewählte Unterklassen von *Exception*



Ausgewählte Unterklassen von Exception

Verschiedene Arten von Ausnahmen

Ausgewählte Unterklassen von *Exception*



Ausgewählte Unterklassen von Exception

Verschiedene Arten von Ausnahmen

Beispiele für Exceptions (2)

- *Exceptions, die von der Klasse RuntimeException abgeleitet s*ind:

Exception	Ursache
`ArithmeticException`	Ein Integer-Wert wurde durch Null dividiert
`IndexOutOfBoundsException`	Auf ein Feld mit ungültigem Index wurde zugegriffen
`ClassCastException`	Cast wegen fehlender Typverträglichkeit nicht möglich
`NullPointerException`	Versuchter Zugriff auf ein Datenfeld oder eine Methode über die `null`-Referenz

Unterscheidung von checked und unchecked Exceptions

Unchecked Exceptions	Checked Exceptions
Klassen `RuntimeException` und `Error` sowie davon abgeleitete Klassen	alle anderen

- **Checked Exceptions** sind Exceptions die von der Programmiererin bzw. vom Programmierer behandelt werden müssen, und bei denen dies auch vom Compiler überprüft (checked) wird.

- **Unchecked Exceptions**, werden von der Programmiererin bzw. vom Programmierer nicht behandelt und ein Programm wird vom Compiler dahingehend nicht überprüft (unchecked).

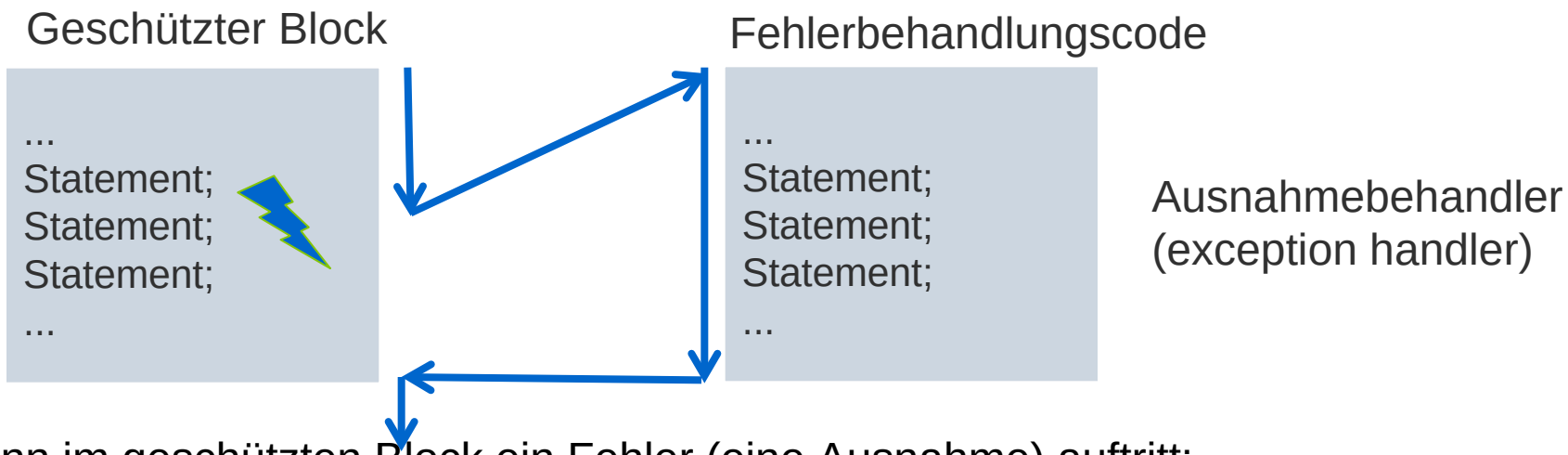
Zwischenfazit: Ausnahmebehandlung in Java

- Ausnahmen (Exceptions) sind unerwartet auftretende Laufzeitfehler.

- Eine Ausnahme in Java ist ein Objekt, das eine Instanz der Klasse `Throwable` (oder einer ihrer Unterklassen) ist.

- *Beispiele*
  - Division durch 0 `( ArithmeticException )`
  - Lesen über Arraygrenzen hinweg `( IndexOutOfBoundsException )`
  - Lesen über das Ende einer Datei hinaus `( EOFException )`

Ansatz der Ausnahmebehandlung in Java



Wenn im geschützten Block ein Fehler (eine Ausnahme) auftritt:

- Ausführung des geschützten Blocks wird abgebrochen
- Fehlerbehandlungscode wird ausgeführt
- Programm setzt nach dem geschützten Block fort

`try`-Anweisung

- Das Exception Handling wird in Java durch eine **try-Anweisung** umgesetzt.

- In Java wird ein Exception Handler durch einen catch-Block realisiert, d.h. eventuell auftretende Exceptions können durch einen oder mehrere catch-Blocks behandelt werden.

- Vorteile
  - Fehlerfreier Fall und Fehlerfälle sind sauberer getrennt
  - Man kann nicht vergessen, einen Fehler zu behandeln
  - (Compiler prüft, ob es zu jeder möglichen Ausnahme einen Behandler gibt)

Struktur der `try`-Anweisung

```
try
{
. . . . .
}
catch (Exceptiontyp1 name1)
{
. . . . .
}
catch (Exceptiontyp2 name2)
{
. . . . .
}
. . .
```

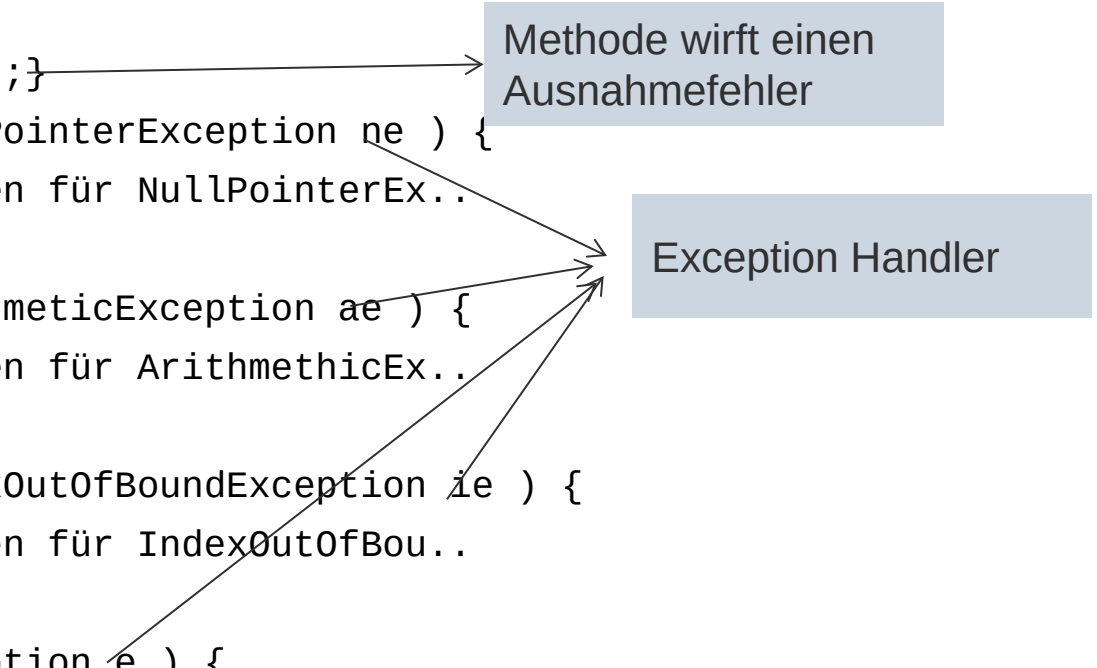
try-Block. (geschützter Block)
Das ist der normale Code,
in dem Fehler auftreten können.

catch-Block 1. (Ausnahmebehandler)
Fängt Fehler der Klasse
Exceptiontyp1

catch-Block 2. (Ausnahmebehandler)
Fängt Fehler der Klasse
Exceptiontyp2

Beispiel der `try`-Anweisung

```
try {
  riskyMethod ( );}
    catch ( NullPointerException ne ) {
    // Anweisungen für NullPointerEx..
    }
    catch ( ArithmeticException ae ) {
    // Anweisungen für ArithmethicEx..
    }
    catch ( IndexOutOfBoundException ie ) {
    // Anweisungen für IndexOutOfBou..
    }
    catch ( Exception e ) {
    // behandelt alle anderen Fehler
    } ...
```



Struktur der `try`-Anweisung mit `finally`-Block

```
try {... }
catch (XYZException e) {
   Out.println(e.getMessage() + ", error code = ", + e.getErrorCode());
}
catch (NullPointerException e) {
...
}
catch (Exception e) {
...
} finally {
...
}
```

finally-Block ist optional, wenn mindestens ein catch-Block da ist. Der finally-Block wird in jedem Fall durchlaufen, egal ob ein Fehler aufgetreten ist oder nicht.

Zweck des finally-Blocks

```
try {
FileStream s = new FileStream(...);
...
...
...
s.close();
} catch (...) {
...
}
```

Falsch

Datei wird im Fehlerfall nicht geschlossen

```
FileStream s;
try {
s = new FileStream(...);
...
...
...
} catch (...) {
...
} finally {
s.close();
}
```

Richtig

Datei wird auf jeden Fall geschlossen

try-catch-finally-Anweisung

- Ausführung der `finally`-Anweisung:
  - Es gab keine Ausnahme!
  - Es gab eine Ausnahme und diese wurde in einer catch-Anweisung behandelt!
  - Es gab eine Ausnahme, aber keine catch-Anweisung war für diesen Fehler zuständig!
- *In allen drei Fällen wird die Finally-Anweisung ausgeführt!*

Der multi-catch-Block

- Die Struktur der try-Anweisung kann verkürzt werden, indem mehrere catch-Blöcke zu einem einzigen catch-Block (= **multi-catch-Block**) zusammengefasst werden.

- Dieser **multi-catch-Block** beinhaltet mehrere Ausnahmetypen, die von ihm gefangen werden können.

- Realisiert wird das Fangen mehrerer Ausnahmetypen mit der binären Oder-Verknüpfung (|) in der Parameterliste des catch-Blocks.

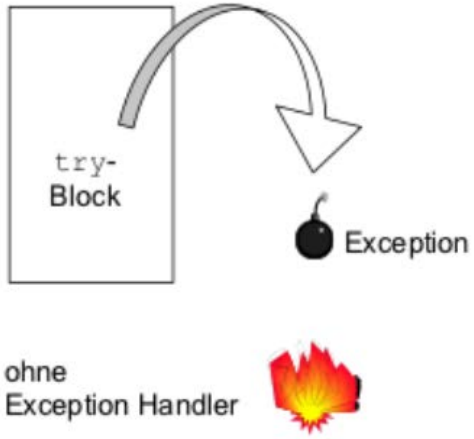
Struktur eines try-Blocks mit multi-catch-Block

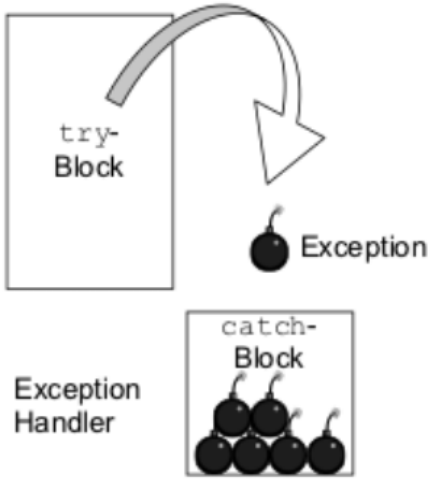
```
try
{
 // Quelltext, in dem Fehler auftreten können
}
catch (Exceptiontyp1 | Exceptiontyp2 ex)
{
 // Dieser catch-Block fängt sowohl Fehler der Klasse
 // Exceptiontyp1 als auch der Klasse Exceptiontyp2 ab
}
finally
{ // Dinge, die stets gemacht werden müssen }
```

Werfen und Fangen von Exceptions

- Die Generierung eines Exception-Objektes und die Übergabe mit `throw` an die Java-VM werden als das Auslösen (Werfen) einer Exception bezeichnet. Das Exception-Objekt enthält Informationen über den aufgetretenen Fehler.

Falls kein Handler vorhanden ist, wird das Programm von der Java-VM abgebrochen.





Falls ein Handler gefunden wird, werden die Anweisungen des Handlers als nächstes ausgeführt und das Programm wird nach den Handlern fortgesetzt.

Beispiel: Exception generieren, auswerfen und fangen

```
public class MyClass {

   public static void main (String[] args) {
      try {
         MyException ex = new MyException();
         throw ex;
      }
      catch (MyException e) {
         System.out.println (e.getMessage());
      }
   }
}
```

Dieser try-Block ist untypisch, da in ihm nur eine Exception zu Demonstrationszwecken geworfen wird.

Anweisungen unterhalb einer throw-Anweisung werden nie abgearbeitet.

Kurze Zusammenfassung: Exceptions

- Das Grundkonzept der Behandlung von Ausnahmen in Java folgt dem Schema:
  - Try: Ausprobieren
  - Throw: Auslösen
  - Catch: Auffangen

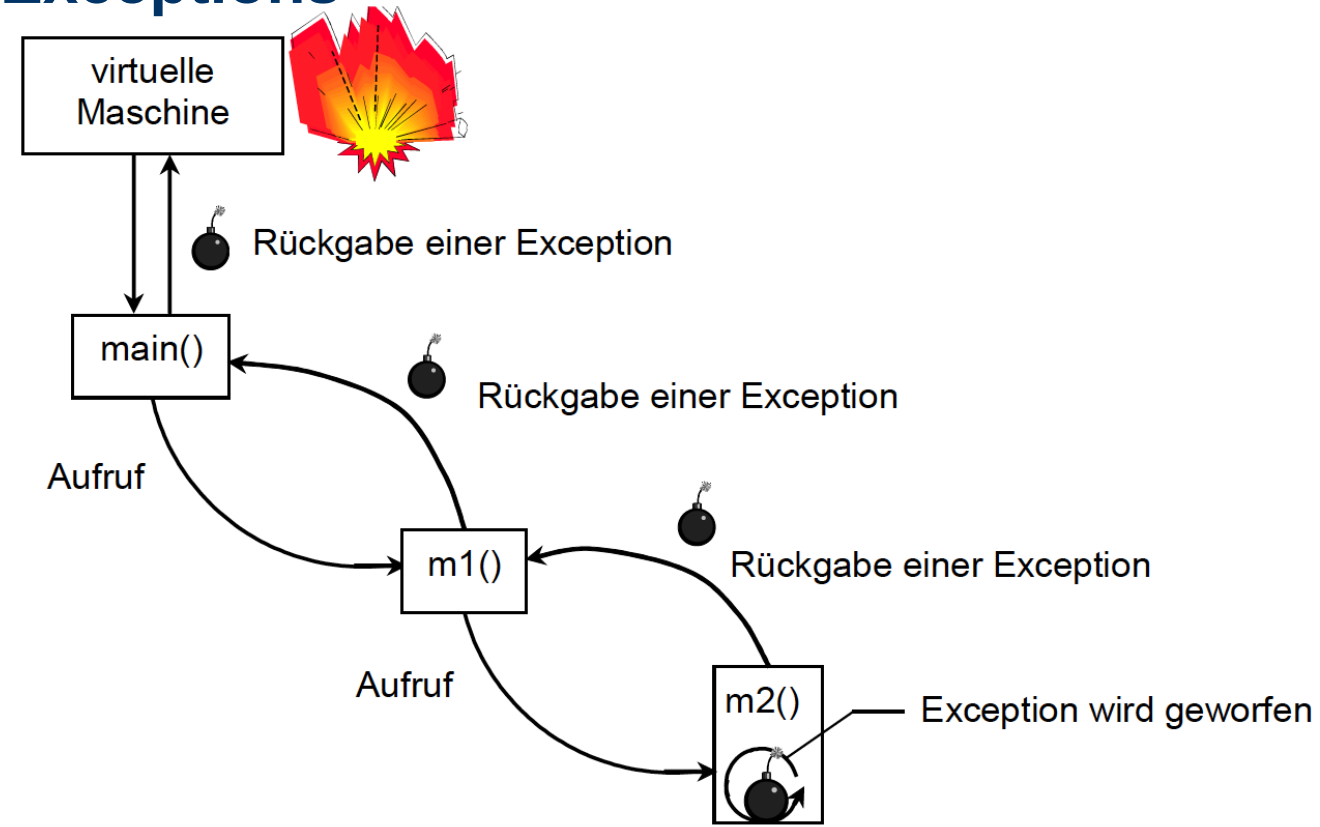
- Beim Auftreten einer Ausnahme wird die Ausführung des Programms unterbrochen und ein Ausnahmeobjekt "geworfen" (mit `throw`).

- Es wird nach einer passenden Ausnahmebehandlung gesucht, die das Ausnahmeobjekt "auffängt" (`catch`).

- Wird keine Ausnahmebehandlung gefunden, wird das Programm durch Aufruf der Methode `System.exit` abgebrochen.

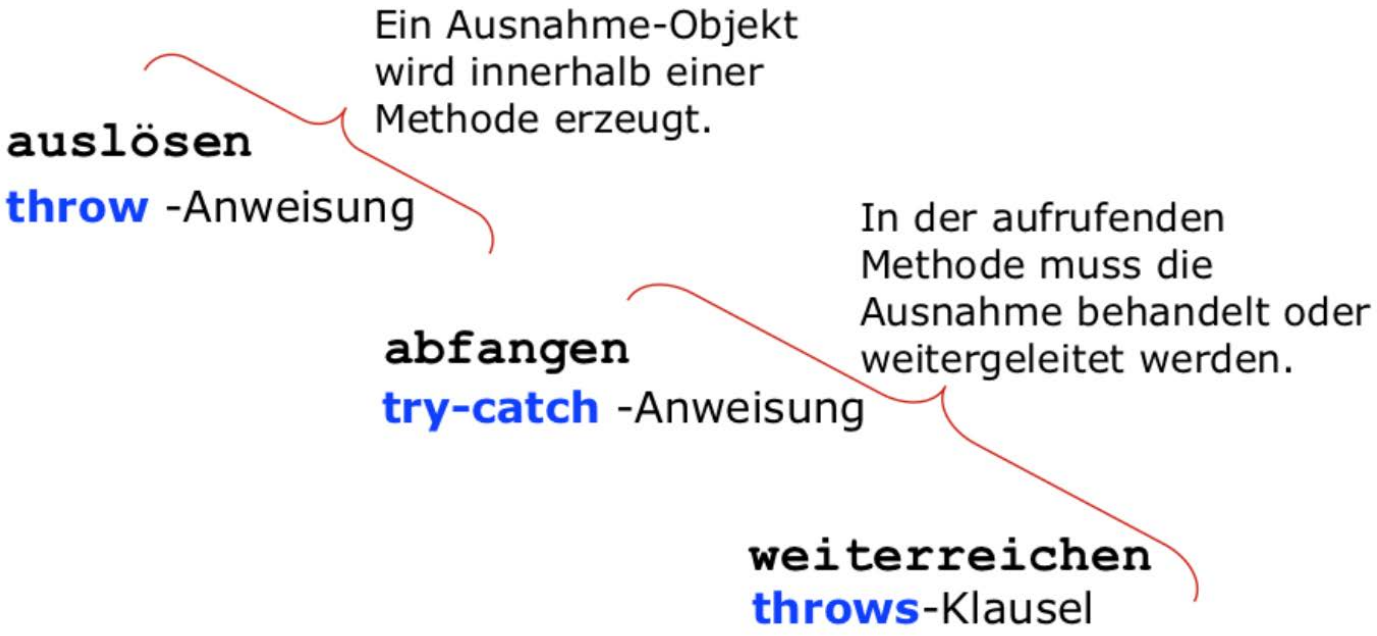
Propagieren von Exceptions

- Exceptions können auch an den Aufrufer weiter-geleitet werden.



Alternatives Ausnahmebehandlungsschema

- Ausnahmen müssen grundsätzlich abgefangen oder weitergereicht werden.



Beispiel: Datumsprüfung

```
import java.util.Date;
import java.text.*;

public class DatumEingabe {

   public Date pruefeDatum (String datum) throws ParseException {
      DateFormat df = DateFormat.getDateInstance();
      df.setLenient (false);
      Date d = df.parse (datum);
      return d;
   }
...
```

Eine auf die Rechnerlokation abgestimmte Instanz der Klasse DateFormat wird erzeugt.

Strenge Datumsprüfung einschalten

Datum überprüfen und in ein Date-Objekt wandeln

Die Methode parse() wirft eine ParseException, wenn in datum kein gültiges Datum steht.

Ausnahmen ankündigen – die throws-Klausel

- Eine Methode kann nur die Checked Exceptions auslösen, die sie in der throws-Klausel angegeben hat. Unchecked Exceptions hingegen kann sie immer werfen.

- Soll also die Exception außerhalb einer Methode verarbeitet werden, muss die Methodendeklaration wie folgt erweitert werden:

```
[Zugriffsmodifikatoren] Rueckgabewert Methodenname ([Parameter])
throws Exceptionname1 [, Exceptionname2, . . . .]
```

parse

```
public Date parse(String source)
           throws ParseException
```

Parses text from the beginning of the given string to produce a date. The method may not use the entire text of the given string.

See the `parse(String, ParsePosition)` method for more information on date parsing.

Parameters:

`source` - A `String` whose beginning should be parsed.

Returns:

A `Date` parsed from the string.

Throws:

`ParseException` - if the beginning of the specified string cannot be parsed.

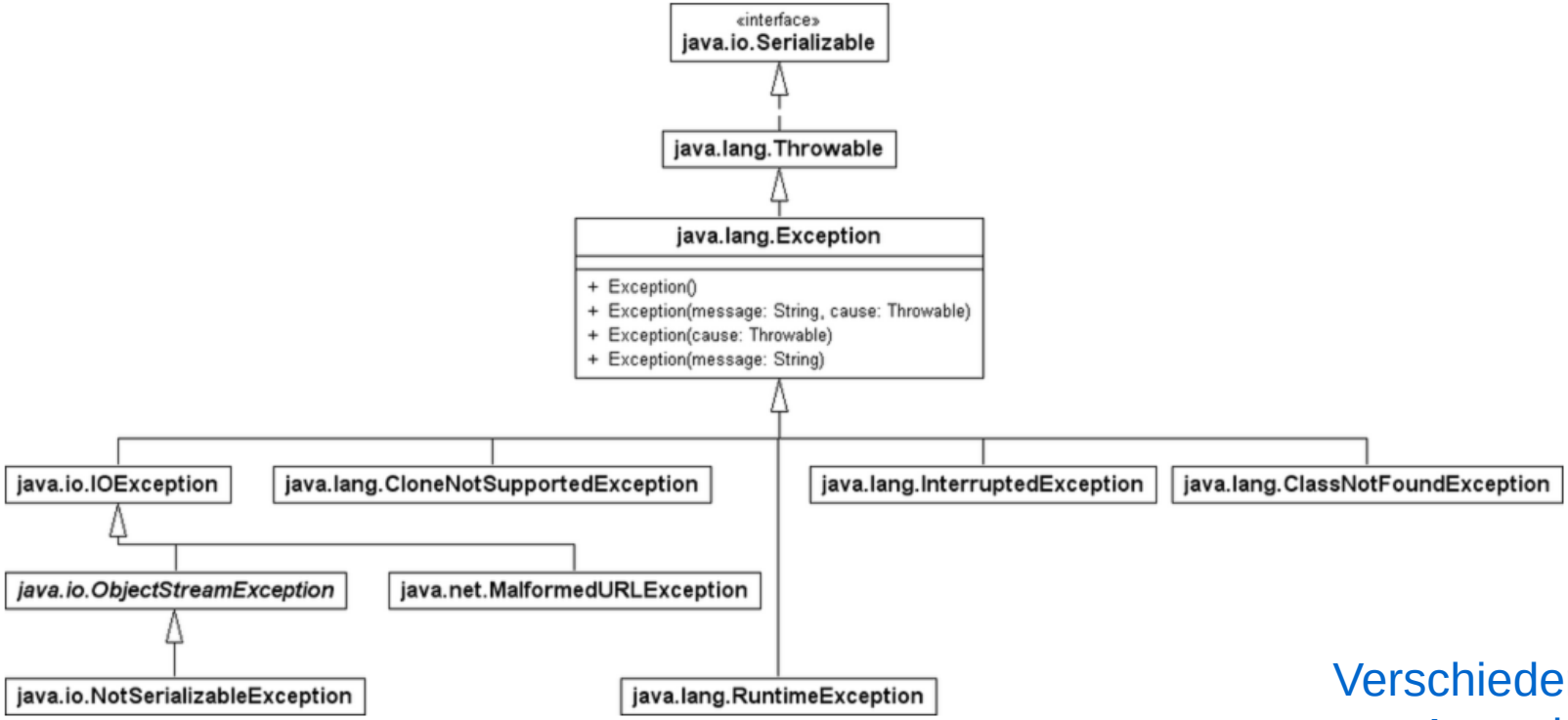
https://docs.oracle.com/javase/7/docs/api/java/text/DateFormat.html

```
...
 public static void main (String[] args) {
   DatumEingabe v = new DatumEingabe();
   String[] testdaten = {"13.06.2013", "13.13.2013"};
   Date datum = null;
   for (int i = 0; i < testdaten.length; i++) {
      try {
         datum = v.pruefeDatum (testdaten [i]);
         System.out.println ("Eingegebenes Datum ist ok:\n” + datum);}
      catch (ParseException e) {
          System.out.println ("Eingegebenes Datum ist nicht ok:\n"
          + testdaten [[i]);}
   }
 }
}
```

Ausgabe

```
Eingegebenes Datum ist ok:
Thu Jun 13 00:00:00 CEST 2013
Eingegebenes Datum ist nicht ok:
13.13.2013
```

Die Exception-Hierarchie



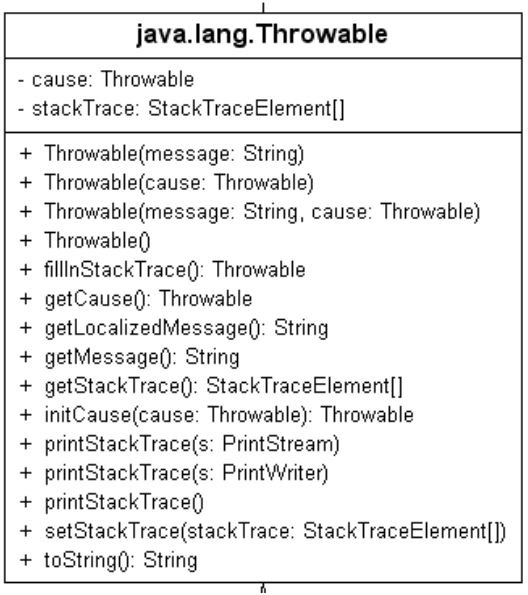
Ausgewählte Unterklassen von Exception

Verschiedene Arten von Ausnahmen

Definition einer eigenen Ausnahmeklasse (exception)

- Es wird eine neue spezielle Klasse definiert, wenn Ausnahmen ganz bestimmte spezifische Eigenschaften haben, die im **Klassenbaum der Exceptions** noch nicht vertreten sind.

- Die neue Klasse wird von der Klasse Exception abgeleitet ist.



Beispiel: MyException definieren

```
class MyException extends Exception {

   public MyException() {

      super ("Fehler ist aufgetreten!");
   }
}
```

Aufruf des Konstruktors der Klasse Exception.
Ihm wird ein String mit dem Fehlertext übergeben, der in einem Datenfeld der Klasse Throwable gespeichert wird.

Beispiel: Erfolglose Suche

- Ausnahmeklasse für erfolglose Suche:

```
class NichtGefunden extends Exception{...}
```

- Anwendung in der Suche-Methode

```
int suche( int [] a, int was) throws NichtGefunden{
   for (int i = 0; i < a.length; i++)
      if (a[i] == was) return i;
   throw new NichtGefunden ( ) ;
}
```

Die Ausnahme wird Bestandteil der Methoden-Signatur

Vorteile des Exception-Konzeptes

- Eine saubere **Trennung** des Codes **in** "**normalen**" **Code** und **in Fehlerbehandlungscode**.

- Der Compiler prüft, ob checked Exceptions vom Programmierer abgefangen werden. Damit werden **Nachlässigkeiten** beim Programmieren **bereits zur Kompilierzeit** und nicht erst zur Laufzeit **entdeckt**.

- Das Propagieren einer **Exception** erlaubt, diese **auch in einem umfassenden Block oder** einer **aufrufenden Methode** zu **behandeln**.

- Da Exception-Klassen in einem Klassenbaum angeordnet sind, können – **je nach Bedarf** – **spezialisierte Handler oder generalisierte Handler geschrieben** werden.

Hinweise zur Anwendung

- Ausnahmebehandlung nicht zur Behandlung normaler Programmsituationen einsetzen!
- Ausnahmebehandlung nicht in zu kleinen Einheiten durchführen!
- Auf keinen Fall Ausnahmen "abwürgen" oder "ignorieren” z. B. durch triviale Fehlermeldungen!
- **Ausnahmen zu propagieren ist keine Schande!**

Nachverfolgung von Ausnahmen

- Java bietet hierfür innerhalb der Klasse Throwable – und damit in allen davon abgeleiteten Unterklassen – die Methode printStackTrace() an.

- Um den genauen Ort des Fehlers festzustellen, kann man e.printStackTrace() aufrufen.

- Diese Methode ermöglicht einen Einblick in die Fehlerursache durch Auflistung
  - der Klasse der Ausnahme samt Fehlertext sowie
  - den sogenannten Stack-Trace, der die Aufrufreihenfolge der Methoden mitsamt deren Klassen umfasst

Beispiel

```
import java.io.FileInputStream;


public class StackTraceExample {
   public static void main(String[] args) {
      try {
         String fileName = null;
         // Daten einlesen
         FileInputStream inStream = new FileInputStream (fileName);
      }
      catch (Exception ex) { ex.printStackTrace();
      }
   }
}
```

Ausgabe

```
java.lang.NullPointerException
at java.io.FileInputStream.<init>(FileInputStream.java:134)
at java.io.FileInputStream.<init>(FileInputStream.java:97)
at StackTraceExample.main(StackTraceExample.java:8)
```

Exceptions vs. Assertations

- Exceptions werden von einem operationell eingesetzten Programm zur Laufzeit geworfen. Assertions hingegen werden zum Testen von Programmen verwendet. Sie überprüfen beim Testen das Vorliegen bestimmter Bedingungen.

- Assertions sind beim Testen eingeschaltet. Sie müssen im im operationellen Betrieb eines Programms abgeschaltet werden.

Freie Universität Berlin

Ausnahmebehandlung

ZUSAMMENFASSUNG

Sie sollten wissen...

- Welche Idee hinter dem Ansatz der Ausnahmebehandlung steht?
- Wie Ausnahmen grundsätzlich in Java abgefangen werden?
- Wie Ausnahmen weitergereicht werden?
- Wie bestehen Methoden bestehender Ausnahmeklasse überladen werden können?
- Wie sie eine eigene Ausnahme-Klasse definieren können?
- Welcher Unterschied zwischen Exceptions und Assertations besteht?