



# Programmierungsmethodik 1

# Programmiertechnik

## Ausnahmebehandlung

- Beziehungen zwischen Klassen
- Basisklasse Object
- Rekursion

Ausblick für heute

- In Programmen sollen Fehler "kommuniziert" werden.
- Fehler sollen abgefangen werden können und nicht zum Programmabsturz führen.
- Es soll eigene Fehlerklassen und –behandlungen definiert werden.
- Fehler (und andere Programmereignisse) sollen protokolliert werden

- Einführung
- Exception-Typen
- catch und finally
- Exceptions werfen
- Logging

Einführung

- Programm
  - darf bei Fehlern nicht unkontrolliert abstürzen oder gar Daten zerstören
  - muss Fehler erkennen und behandeln
  - Beispiele für Fehlertypen
    - falsche Eingabe-Daten (Bedienfehler)
    - inkonsistente (Programm-)Zustände (logische Fehler)

- Erinnerung

```
Scanner scanner = new Scanner(System.in);  
scanner.nextInt();  
→ Eingabe: "Zahl"
```

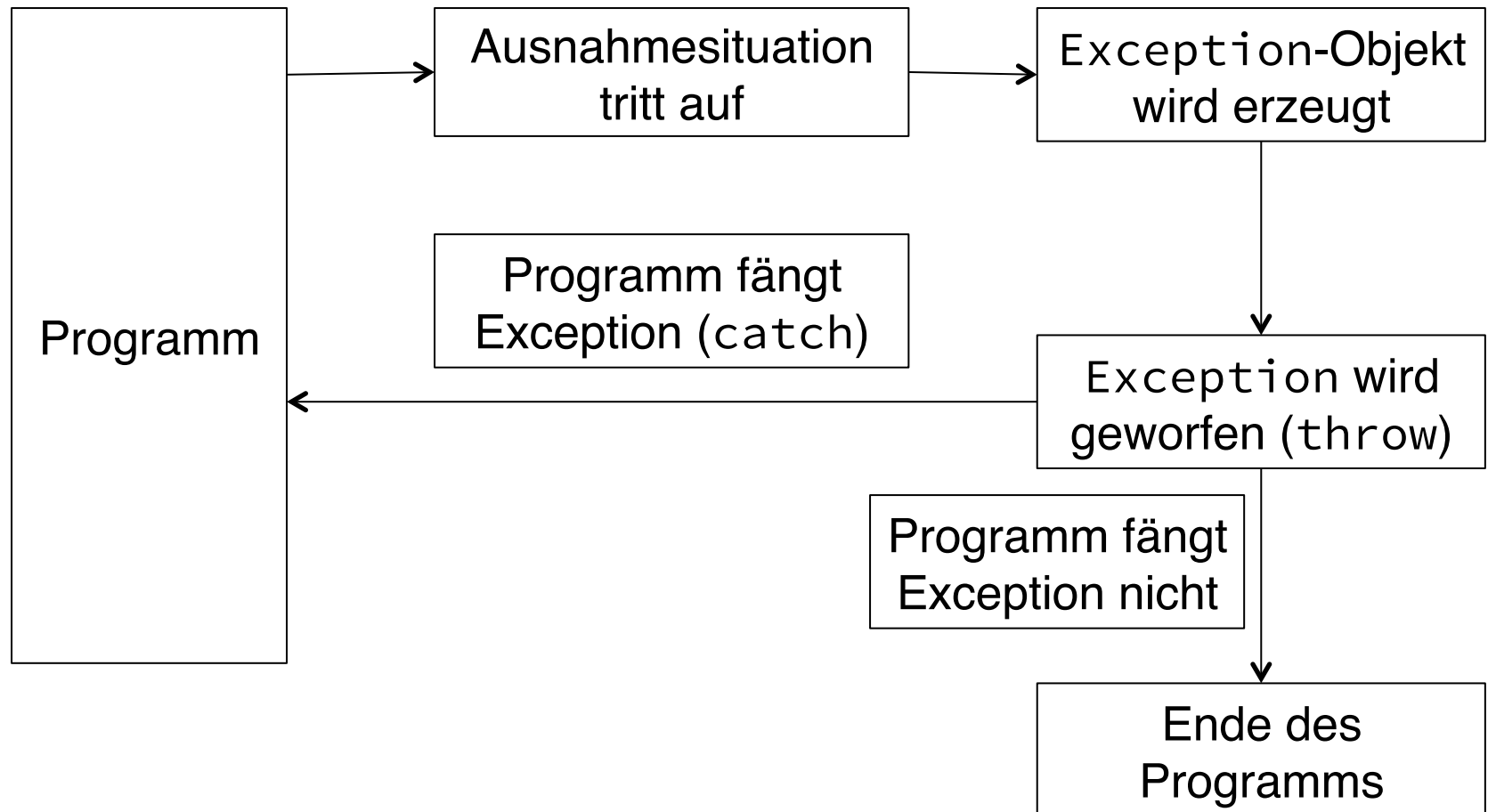
*Exception in thread "main"*

*java.util.InputMismatchException*

- Programmiersprache sollte dafür sorgen, dass ...
  - Code für die Fehlerbehandlung vom eigentlichen Anwendungscode (Programmlogik) getrennt wird.
  - Fehlerbehandlung erzwungen werden kann.
  - die Aufgabe der Fehlerbehandlung an andere Methoden weitergegeben werden kann.



- Ausnahmezustände (Fehler) werden als `Exception` bezeichnet
- sind als Klassen implementiert
  - konkreter Fehler = Objekt
- Grundprinzip:
  - Laufzeitfehler oder eine vom Entwickler gesetzte Bedingung lösen eine `Exception` aus (`throw`)
  - `Exception` wird von der Methode, in der sie ausgelöst wurde
    - behandelt (`catch`)
    - oder an den Aufrufer der Methode weitergegeben
  - wird die `Exception` in einer Methode weder behandelt noch weitergegeben
    - Programmabbruch und zur Ausgabe einer Fehlermeldung



- fehlerhafter Code

```
double[] doubleArray = { 1.0, 2.0, 3.0 };  
System.out.println(doubleArray[5]);
```

- Ausgabe:

```
Exception in thread "main"  
    java.lang.ArrayIndexOutOfBoundsException: 5
```

- da Exception nicht gefangen: Abbruch des Programms!

- Ansatz
  - falls `Exception` auftritt
  - wird diese aufgefangen
  - verarbeitet
  - Programm läuft kontrolliert werden
- Umsetzung in Java
  - Umschließen des Codes mit einem `try-catch-Block`

- Beispiel

```
try {
```

```
    double[] doubleArray = { 1.0, 2.0, 3.0 };
```

```
    System.out.println(doubleArray[5]);
```

```
} catch (Exception e) {
```

```
    System.out.println("Exception: " + e.getMessage());
```

```
}
```

- Ausgabe

```
Exception: 5
```

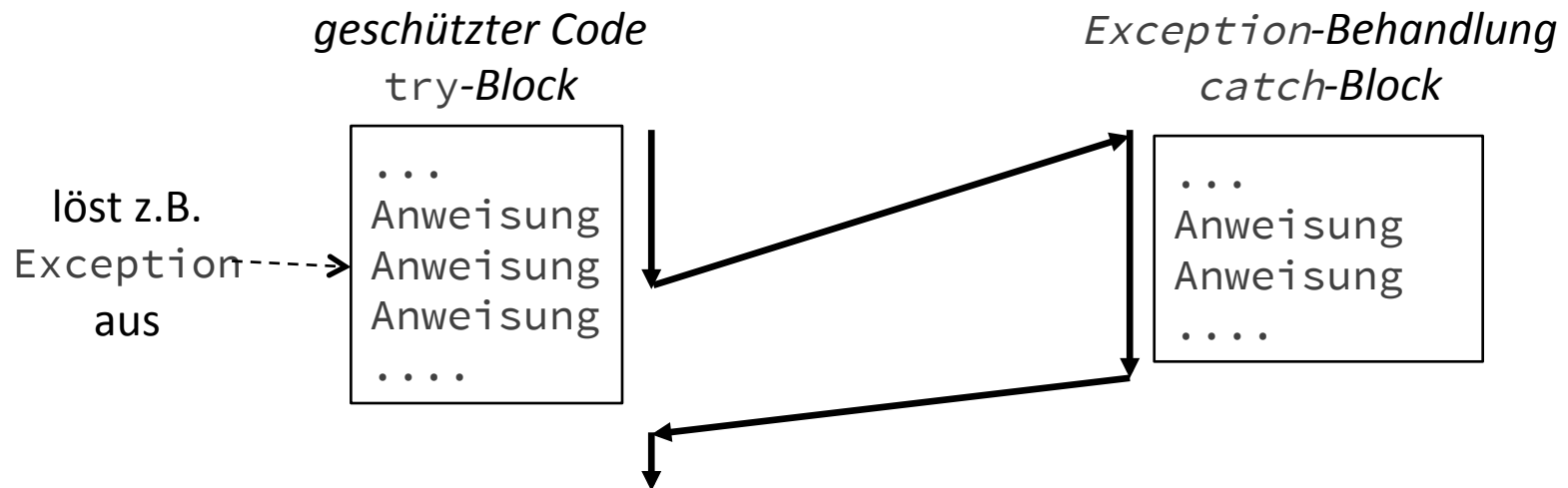
- Programm läuft weiter

- ab Ende des catch-Blocks

Fehler tritt beim Auswerten  
dieses Ausdrucks auf,  
System.out... wird nicht mehr  
abgearbeitet



- ausgelöste Exception im try-Block:
  - Ausführung des try-Blocks wird abgebrochen
  - catch-Block wird durchlaufen (Fehlerbehandlung)
  - Programm wird nach dem try-Block fortgesetzt
- keine Exception ausgelöst:
  - try-Block wird ordnungsgemäß durchlaufen
  - catch-Block wird nicht ausgeführt



- geschützter Code
  - für diesen Block ist explizit das Abfangen und Behandeln von Fehlern "eingeschaltet" → try-Block

- Syntax

```
try {  
    //geschützter Block: "Normalfall"  
    <Anweisung>  
    ...  
} catch (<Exception-Typ> <Variablenname>){  
    // Exception-Behandlung mit aufgetretener  
        // Exception als Parameter  
    // → nur Reaktion auf Fehler */  
    <Anweisung>  
    ...  
}
```

- Normalfall und Fehlerbehandlung sind sauber getrennt!



- Ganzzahl von Benutzer eingeben lassen

```
Scanner scanner = new Scanner(System.in);  
int zahl = scanner.nextInt();
```
- Problem: Laufzeitfehler, falls Eingabestring nicht zu einer Integerzahl konvertiert werden kann!

- API-Dokumentation der Scanner-Methode `nextInt`:

*Throws:*

- *`InputMismatchException` - if the next token does not match the Integer regular expression, or is out of range*
- *`NoSuchElementException` - if input is exhausted*
- *`IllegalStateException` - if this scanner is closed*

- falls Methode Exception werfen könnte
  - Angabe in der Methodensignatur
  - Schlüsselwort throws in der
  - Methode kann eine Exception dieses Typs auslösen
- Beispiel (Methode, die einen Eintrag aus einem Array zurückliefert)

```
public int getWertInArray(int index)
    throws IndexOutOfBoundsException {
    ...
}
```

- daher folgender Umgang mit dem Scanner-Beispiel
  - Code für die Benutzereingabe als geschützten Block deklarieren
  - NumberFormatException behandeln

```
int m;  
try {  
    Scanner scanner = new Scanner(System.in);  
    m = scanner.nextInt();  
    scanner.close();  
} catch (NumberFormatException exceptionObjekt) {  
    // Fehler aufgetreten  
    System.out.println("Fehler bei Eingabe");  
}
```

eigentlich müssen natürlich  
alle möglichen Exceptions  
gefangen werden

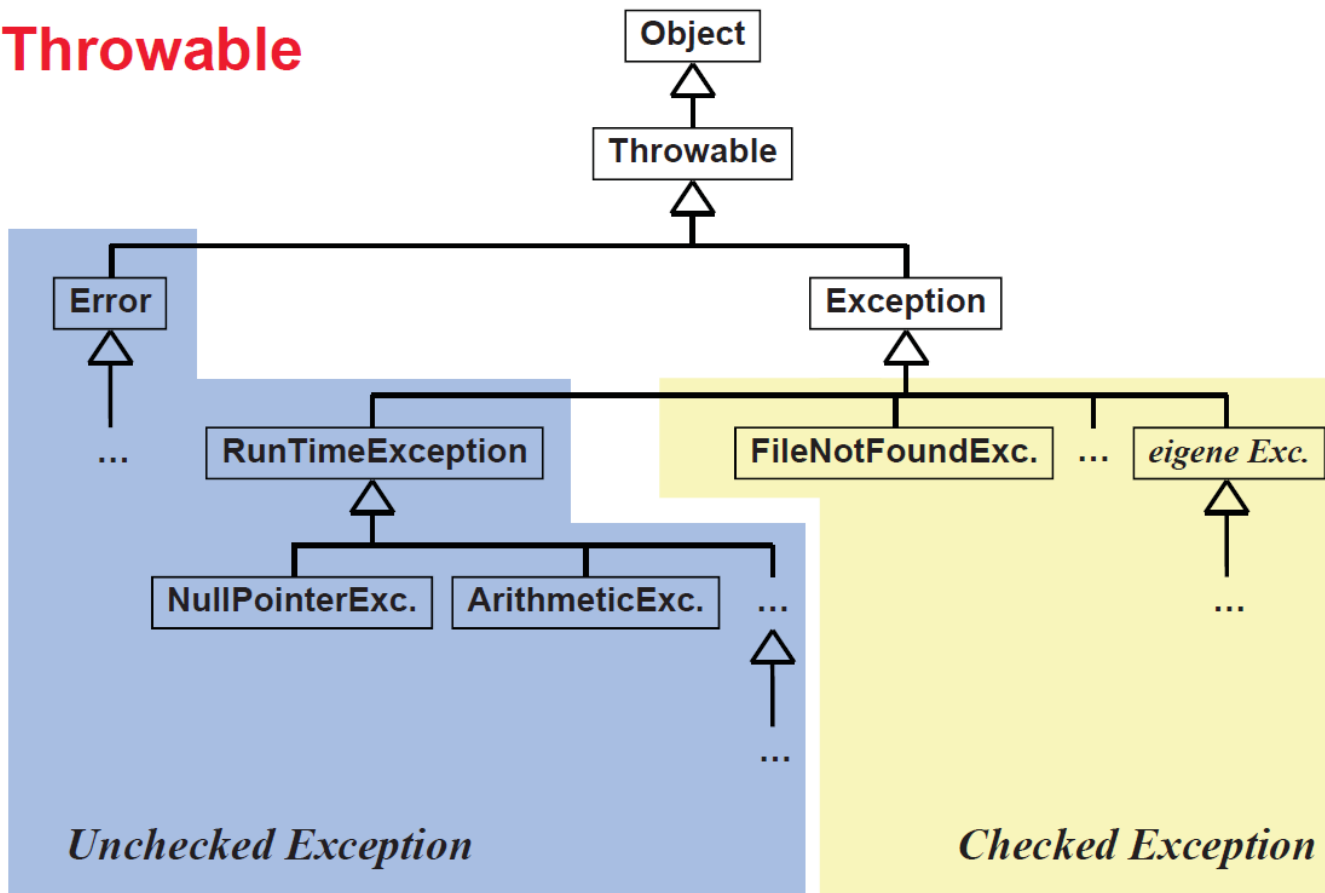
- Gegeben ist die folgende Methodensignatur einer fiktiven Klasse  
Funktion:

```
void auswerten(double x) throws InvalidNumberException;
```

- Schreiben Sie einen sicheren Aufruf der Methode, der mögliche  
Exceptions beim Methodenaufruf verarbeitet

Exception-Typen

## Throwable



- Unchecked Exceptions
- Laufzeitfehler, Systemausnahmen
- werden automatisch von der Java-VM ausgelöst, Beispiele:
  - Division durch 0: `ArithmeticException`
  - Zugriff über Null-Pointer: `NullPointerException`
  - Indexüberschreitung: `IndexOutOfBoundsException`
- können explizit im eigenen Code behandelt werden

- geprüfte Exceptions, Benutzerausnahmen
- spezielle vordefinierte Exceptions
  - z.B. `FileNotFoundException`
- eigene vom Programmierer ausgelöste Exceptions
- müssen explizit im Code behandelt werden
  - Compilerprüfung



- Klasse alle Exceptions sind von der Klasse Exception abgeleitet
  - zumindest indirekt
- Fehlerinformationen in jedem Exception-Objekt:

```
class Exception extends Throwable {  
    // Konstruktor mit Übergabe der Fehlermeldung  
    Exception (String message);  
    // Liefern der Fehlermeldung  
    String getMessage();  
    // Ausgeben der Methodenaufkette auf der Konsole  
    void printStackTrace()  
    ...  
}
```

- Implementieren Sie einen neuen Exception-Typ.
- Die Exception soll geworfen werden, wenn eine Stromversorgung unter einen Schwellwert fällt.
- Die Exception soll sowohl den Schwellwert als auch den aktuellen Spannungswert als Meldung ausgeben können.

catch und finally

- mehrere catch-Blöcke für einen try-Block definierbar
  - Unterscheidung von verschiedenen Fehlerklassen ist möglich

```
try {  
    <Anweisung>  
    ...  
} catch (<Exception-Typ1> <Variablenname1>){  
    <Anweisung>  
    ...  
} catch (<Exception-Typ2> <Variablenname1>){  
    <Anweisung>  
    ...  
}
```

- Exception-Typen der catch-Blöcke werden der Reihe nach mit einer ausgelösten Exception verglichen
  - also: Reihenfolge ist entscheidend!
- erster kompatibler catch-Block gilt
  - nachfolgende und eventuell ebenso passende catch-Blöcke werden ignoriert

# Beispiel: Benutzereingaben



```
int eingabe;
try {
    Scanner scanner = new Scanner(System.in);
    // nextInt() löst evtl. Exception aus!
    eingabe = scanner.nextInt();
    // eingabe <= 0 --> ebenfalls Exception auslösen!
    if (eingabe <= 0) {
        scanner.close();
        throw new NumberFormatException("Wert < 0");
    }
    scanner.close();
} catch (NumberFormatException exception) {
    // Fehlerbehandlung für falsche Eingabe
    System.out.println("Ungültige Eingabe: " +
exception.getMessage());
} catch (IllegalStateException exception) {
    System.out.println("Scanner nicht offen: " +
exception.getMessage());
}
```

- noch einmal zum Scanner

```
int eingabe;  
try {  
    Scanner scanner = new Scanner(System.in);  
    // nextInt() löst evtl. Exception aus!  
    eingabe = scanner.nextInt();  
    // eingabe <= 0 --> ebenfalls Exception auslösen!  
    if (eingabe <= 0) {  
        scanner.close();  
        throw new NumberFormatException("Wert < 0");  
    }  
    scanner.close();  
}
```

falls hier eine Exception  
auftritt, wird die Ressource  
scanner nicht wieder  
geschlossen!

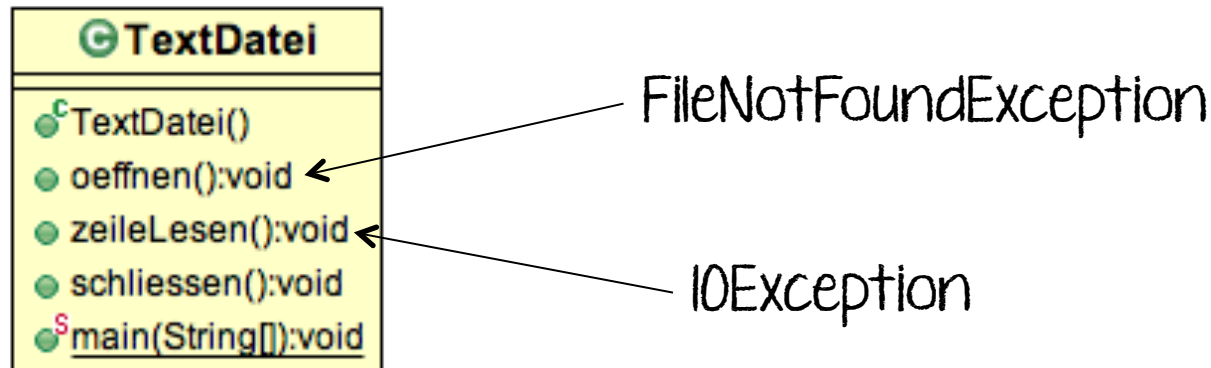
- häufig: "Aufräumen" notwendig unabhängig davon, ob Exception auftritt
  - Ressourcen freigeben
  - Dateien schließen
  - Zustand speichern
- Lösung: optionaler `finally`-Block am Ende der Liste der `catch`-Blöcke
- Syntax

```
try {...}
catch(...) {...}
catch(...) {...}
...
finally {...}
```



- `finally`-Block wird immer ausgeführt, d.h. auch in folgenden Fällen:
  - normales Ende des `try`-Blocks
  - Ende durch `return`-Anweisung im `try`-Block
  - Exception im `try`-Block, passender `catch`-Block gefunden und durchlaufen
  - Exception im `try`-Block, kein passender `catch`-Block

- Schreiben Sie ein Code-Fragment mit folgendem Verhalten:
- zunächst öffnen Sie eine TextDatei (oeffnen)
- dann lesen Sie eine Zeile aus der Textdatei (zeileLesen)
- dann schließen Sie die Datei wieder (schliessen)
- jede der Methoden soll nur einmal aufgerufen werden



Exceptions werfen

- catch-or-throw-Regel: jede `CheckedException` muss
  - entweder behandelt (`catch`)
  - oder weitergegeben (`throw`) werden
- kein geeigneter `catch`-Block in einer Methode definiert:
  - Exception automatisch an den Aufrufer der Methode weitergegeben
  - kann bei `UncheckedExceptions` zu einem Programmabbruch führen
- mögliches Weitergeben von Exceptions muss (bei `CheckedExceptions`) der aufrufenden Methode mitgeteilt werden
  - über die Methodensignatur (Schnittstelle!)

```
<Ergebnistyp> <Methodenname>(...)  
    throws <Exception-Typ1>, <Exception-Typ2>, ... { ... }
```

- Exception wird explizit durch eine `throw`-Anweisung ausgelöst:

```
throw <Exception-Objekt>;
```

- Beispiel:

```
throw new NumberFormatException("Keine Zahl");
```

- bricht „normale“ Programmausführung (`try`-Block) sofort ab
- es wird von innen nach außen in allen umgebenden `try`-/`catch`-Anweisungen nach passendem `catch`-Block gesucht
- startet den `catch`-Block und übergibt `Exception`-Objekt als Argument
- nach Beendigung des `catch`-Blocks wird die Ausführung nach dem zugehörigen `try`-Block fortgesetzt
  - falls im `catch`-Block kein Abbruch mit `return` erfolgte

# Beispiel: Benutzereingaben



```
int eingabe;
try {
    Scanner scanner = new Scanner(System.in);
    // nextInt() löst evtl. Exception aus!
    eingabe = scanner.nextInt();
    // eingabe <= 0 --> ebenfalls Exception auslösen!
    if (eingabe <= 0) {
        scanner.close();
        throw new NumberFormatException("Wert < 0");
    }
    scanner.close();
} catch (NumberFormatException exception) {
    // Fehlerbehandlung für falsche Eingabe
    System.out.println("Ungültige Eingabe: " +
exception.getMessage());
}
```

- Vordefinierte Exception-Klassen ok, sofern angemessen:
  - `IllegalArgumentException`: Methode erhält unzulässigen Parameter
  - `IndexOutOfBoundsException`: Index nicht im zulässigen Bereich
  - `BufferOverflowException`: zu viele Elemente in Speicherstruktur
  - `MissingResourceException`: Vorgabe fehlt
  - `NoSuchElementException`: Zugriff auf nicht existierendes Element
  - `UnknownElementException`: Unbekanntes Element
  - ...



- neue, eigene Exception-Klassen vorteilhaft, wenn spezielle Fehlersituation vorliegt, z.B.:

```
/**  
 * Eigene Exception-Klasse für die Behandlung eines  
 * Benutzer-Abbruchwunsches  
 */  
class UserCancelledException extends Exception {  
    ...  
}
```

- Entwickeln Sie eine eigene Exception `DivisionDurchNullException`.
- Schreiben Sie eine Methode `division`, die zwei Fließkommazahlen durcheinander teilt und das Ergebnis zurückgibt.
- Die Methode soll eine `DivisionDurchNullException` werfen (falls durch 0 geteilt wird).

Logging

- häufig sinnvoll: Ablauf eines Programms protokollieren
  - vergleiche: Tracing
- wichtige Meilensteine im Programm werden festgehalten
- z.B.:
  - Datei erfolgreich geöffnet
  - Fehler bei Datenbankabfrage
  - Nutzer xy hat falsches Passwort angegeben
  - ...

- Klasse Logger aus Package `java.util.logging`
- Erzeugen eines Logger-Objektes  
`Logger logger = Logger.getLogger(<Name des Loggers>);`
- Als Name kann z.B. der Klassenname verwendet werden  
`TextDatei.class.getName()`

- Logging eines Ereignisses
  - Methode `log(Level level, String msg)`
- Level gibt an, wie schwerwiegend das Ereignis ist
  - `Level.INFO`: allgemeine Informationen (Tracing, Debugging)
  - `Level.SEVERE`: schwerwiegendes Ereignis
  - ...
- Beispiel:  

```
logger.log(Level.INFO, "Datei geöffnet.");
```

- Überladene Variante von log

```
log(Level level, String msg, Throwable thrown)
```

- Beispiel

```
Exception exception = new NullPointerException();  
logger.log(Level.SEVERE, "Null-Pointer", exception);
```

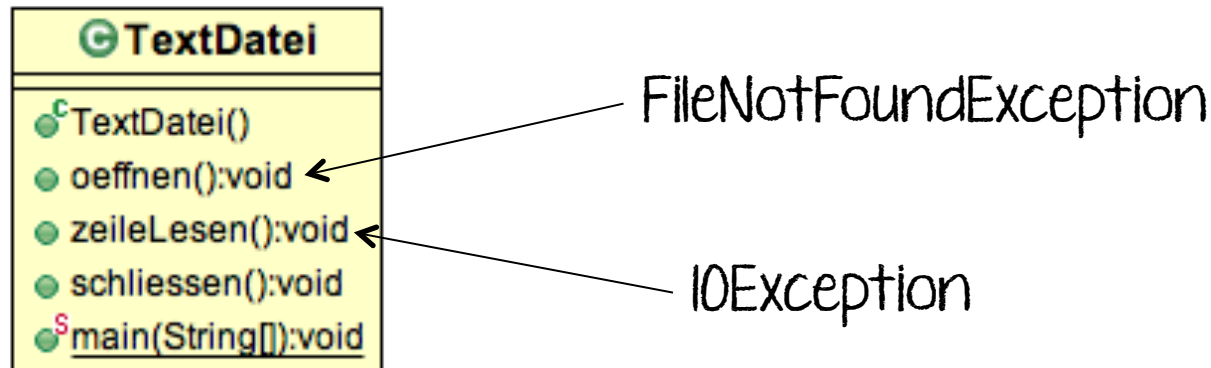
- Standard: Konsole
- weitere Handler registrierbar
  - Beispiel: Datei

```
FileHandler fileHandler = new FileHandler("logdatei.txt");
logger.addHandler(fileHandler);
```
  - Ablage im XML-Format



- `java.util.logging` gibt es erst seit Java 1.4
- daher haben sich Alternativen etabliert
- funktionieren ähnlich
- Beispiele
  - Log4J (<http://logging.apache.org/log4j/2.x/>)
  - Logback (<http://logback.qos.ch/>)
  - Slf4J (<http://www.slf4j.org/>)

- Setzen Sie für die Klasse `TextDatei` Logging um
- Kümmern Sie sich nicht um die Implementierung der eigentlichen Datei-Funktionalität



- Einführung
- Exception-Typen
- catch und finally
- Exceptions werfen
- Logging