

# Programmieren II: Java

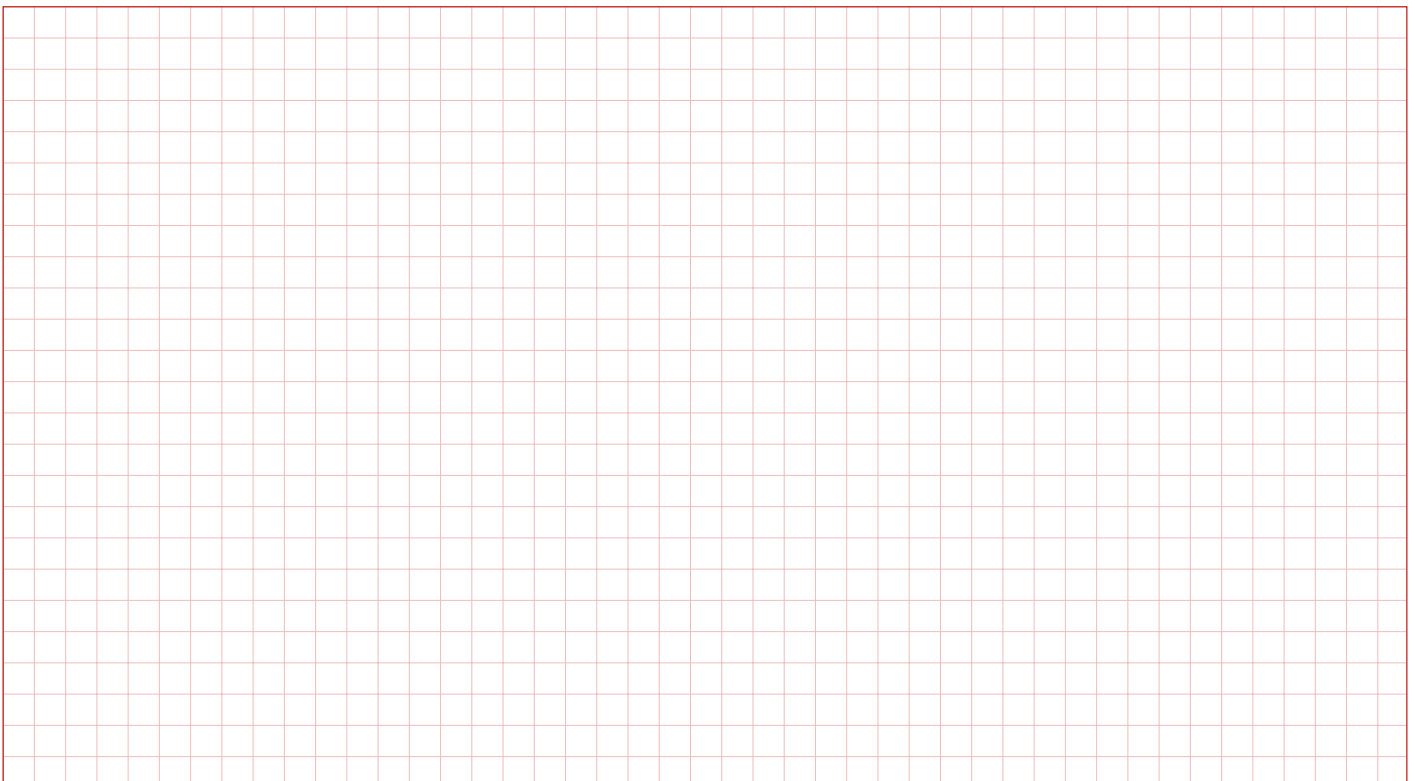
Ausnahmebehandlung

Prof. Dr. Christopher Auer

Sommersemester 2024



## Notizen



**Ausnahmen behandeln**

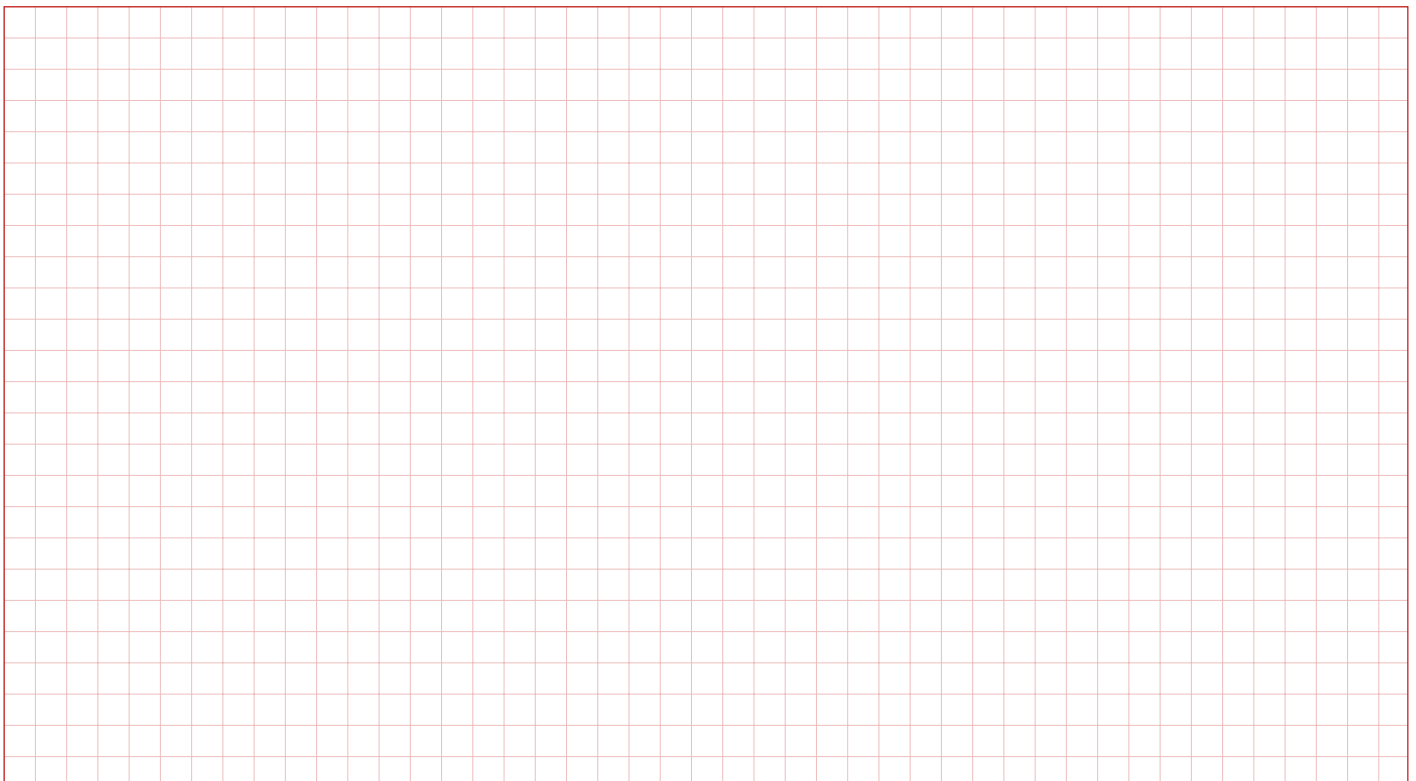
**Hierarchie der Ausnahmen**

**Ausnahmen auslösen**

**Eigene Ausnahmen definieren**

**Zusammenfassung**

## **Notizen**

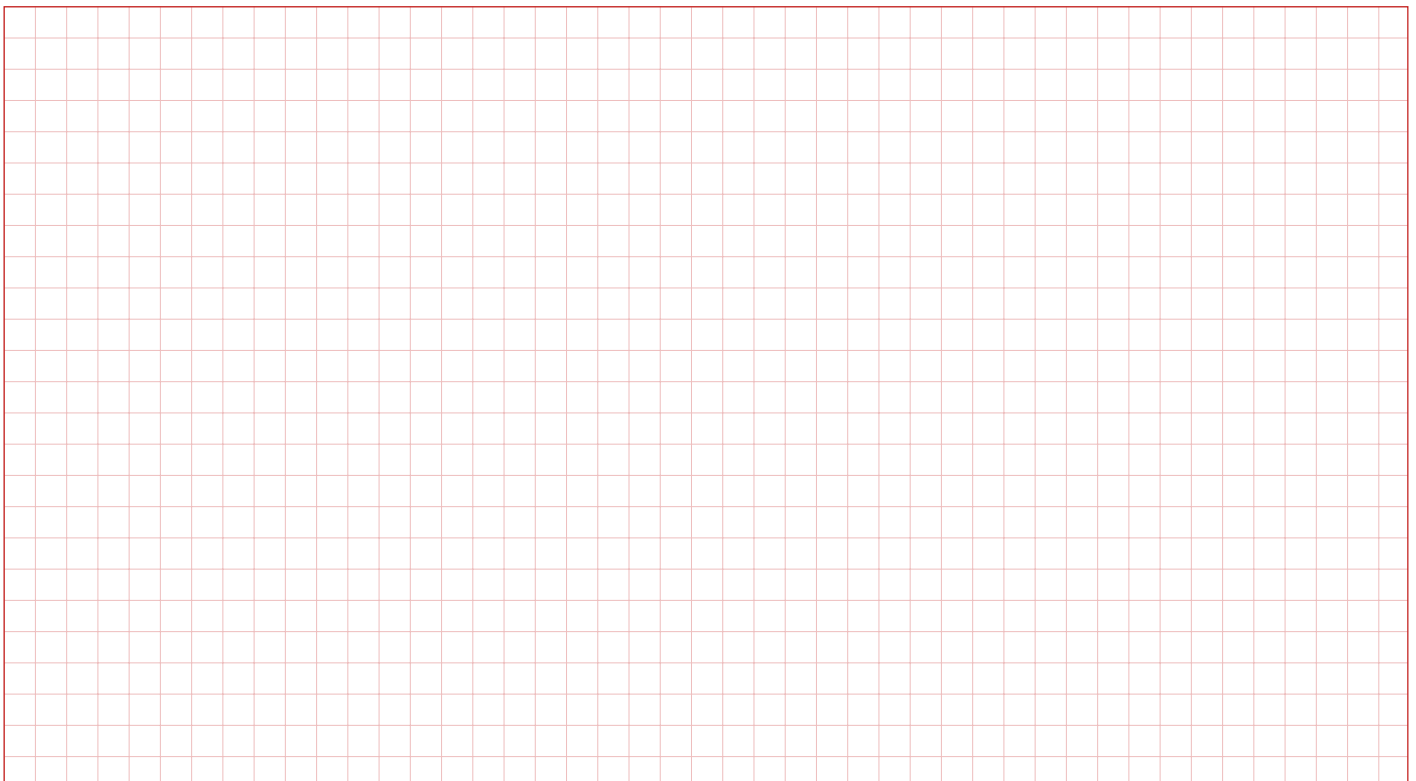


# Inhalt

## Ausnahmen behandeln

- Ausnahmebehandlung bisher
- Ausnahmebehandlung in Java
- Auffangen mehrerer Ausnahmetypen
- Geprüfte und ungeprüfte Ausnahmen
- throws-Deklaration
- Wo und wie Ausnahmen fangen?

## Notizen



# Inhalt


## Ausnahmen behandeln

## Ausnahmebehandlung bisher

## Notizen

## Ausnahmebehandlung bisher


- ▶ Ausnahmebehandlung bisher **nicht vorhanden**

```
13  runNoExceptionHandlingExample  
14 Scanner scanner = new Scanner(System.in);  
16 int i = scanner.nextInt();  
17 out.printf("%d*d = %d%n", i, i, i*i);  
19 scanner.close();
```

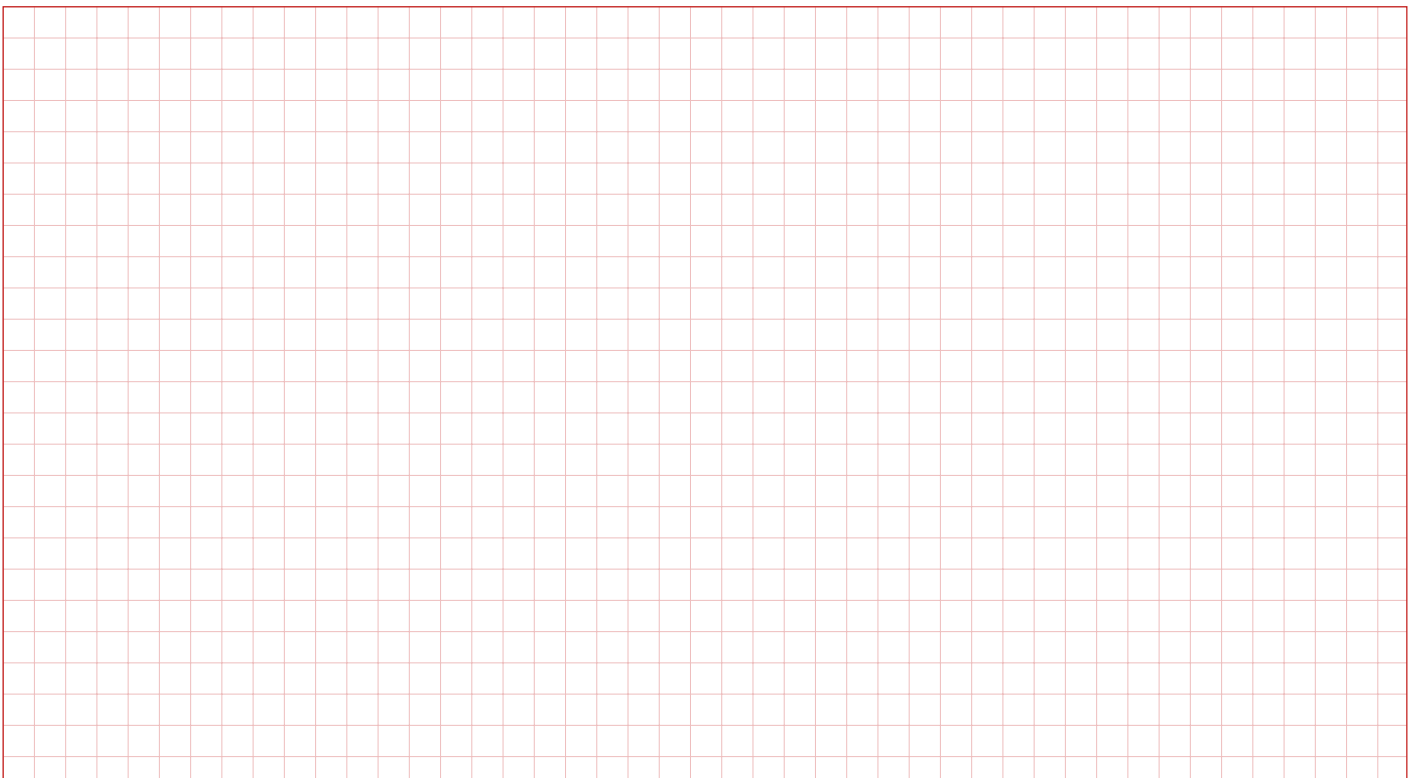
 ExceptionHandlingExamples.java

```
2  
2*2 = 4 // OK
```

```
vier // FEHLER
```

- ▶  `InputMismatchException`: vier konnte nicht in Zahl gewandelt werden
- ▶ Programmabbruch

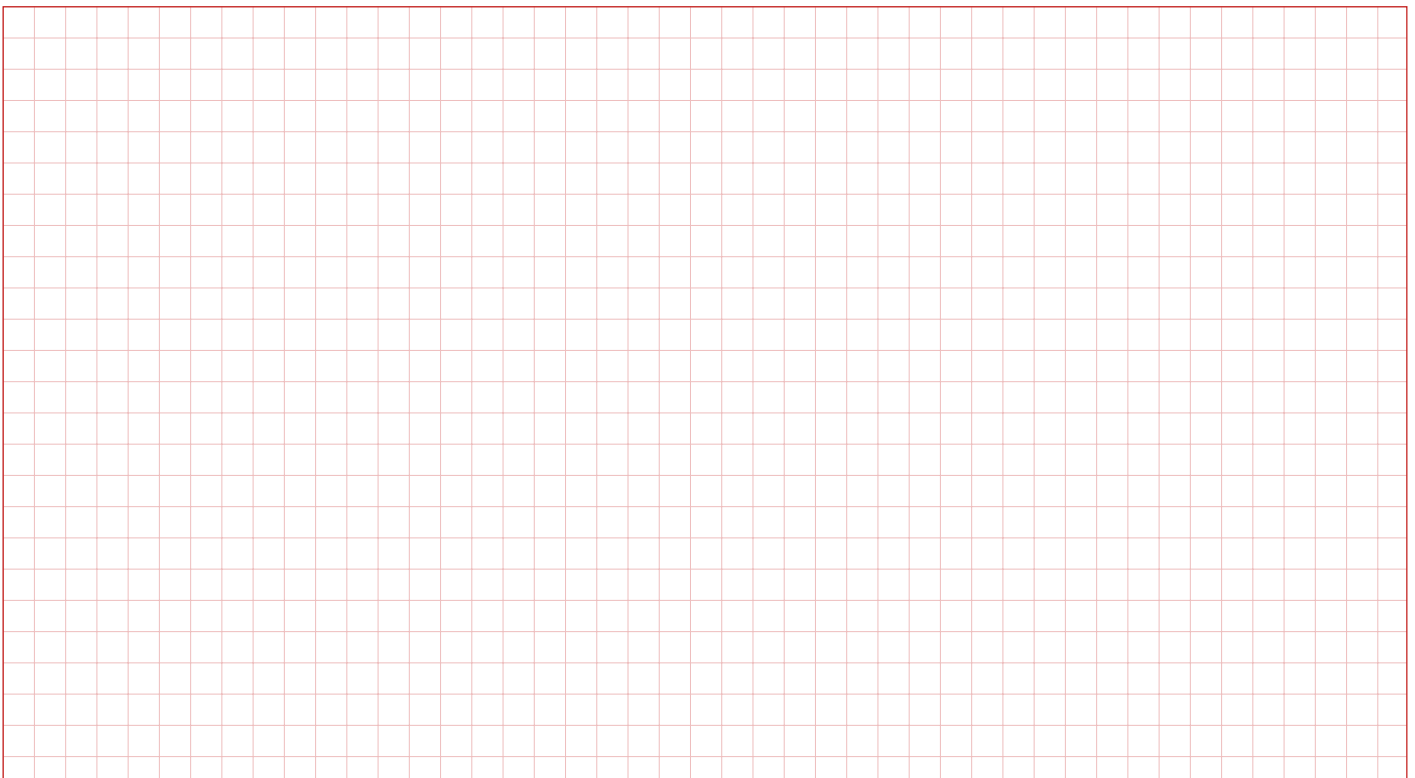
## Notizen



# Ausnahmen

- ▶ **Exception (Ausnahme)**: Abweichung vom erwarteten Programmfluss
  - ▶ **Logischer Fehler** z.B. Programmierfehler (↗ `NullPointerException`)
  - ▶ **Bedienfehler** z.B. oben (↗ `InputMismatchException`)
  - ▶ **Problem in der JVM** z.B. kein Speicher mehr (`OutOfMemoryException`)
  - ▶ **Sicherheit** z.B. keine Zugriffsrechte (↗ `AccessDeniedException`)
  - ▶ ...
- ▶ **Reaktion auf Ausnahmen**
  - ▶ **Logischer Fehler**: Programmabbruch (**Bugfix!**)
  - ▶ **Bedienfehler**: Nutzerhinweis
  - ▶ **Problem in der JVM**: Programmabbruch, evtl. Hinweis/Rettung
  - ▶ **Sicherheit**: Nutzerhinweis
- ▶ Es ist **wichtig** auf Ausnahmen **passend zu reagieren!**

## Notizen



## Ausnahmebehandlung in C

- ▶ Ausnahmebehandlung in C
- ▶ Funktion `fopen` **öffnet Datei** zum Lesen/Schreiben

```
FILE* fopen(const char* path, const char* mode);
```

### Rückgabewert

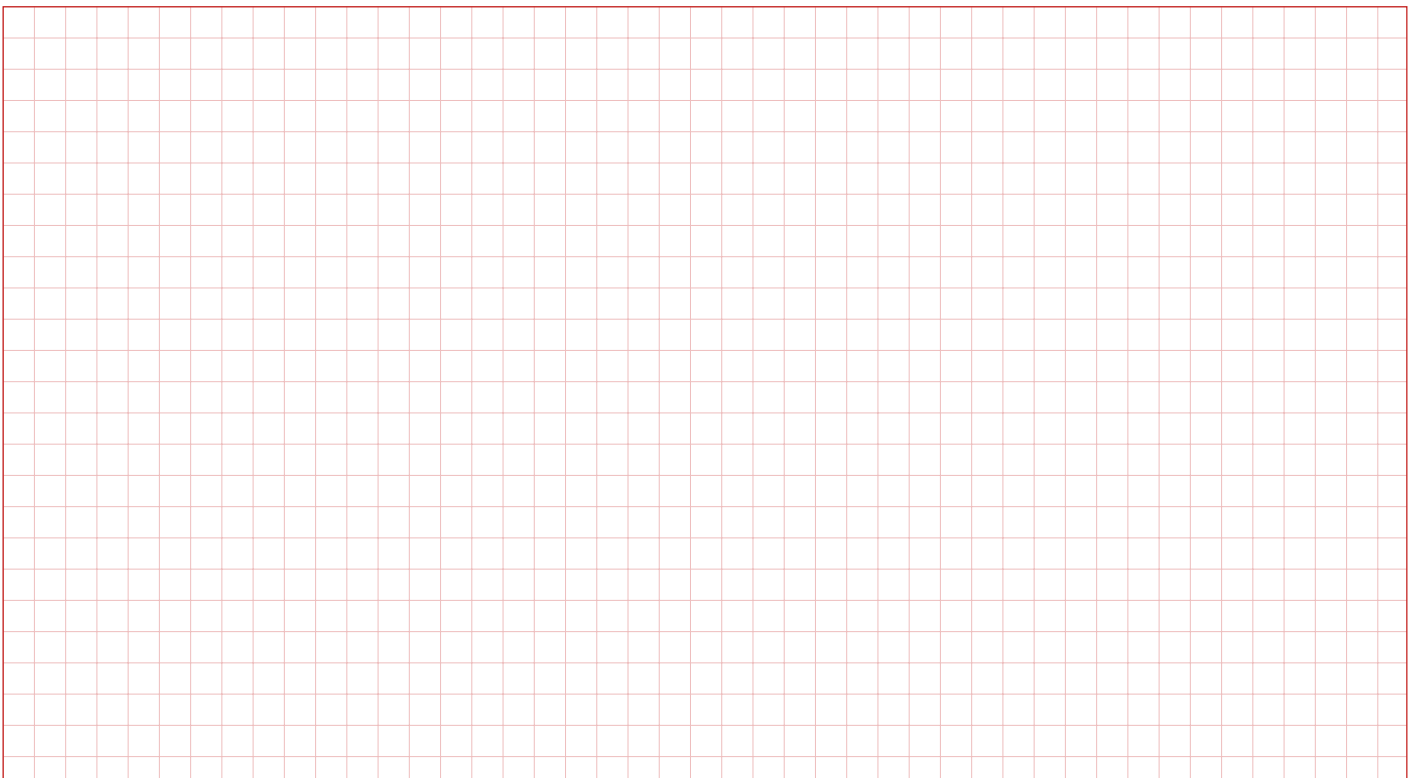
- ▶ **Erfolgreich:** Handle
- ▶ **Fehler**
  - ▶ NULL
  - ▶ **Fehlergrund:** globale Variable `errno`
- ▶ Funktion `fwrite` **schreibt** in Dateistrom

```
size_t fwrite(const void* ptr, size_t size, size_t nitems,  
FILE* stream);
```

### Rückgabewert

- ▶ **Erfolgreich:** Anzahl geschriebener Einträge
- ▶ **Fehler:**
  - ▶ `< nitems`
  - ▶ **Fehlergrund:** über `feof()` ermitteln

## Notizen



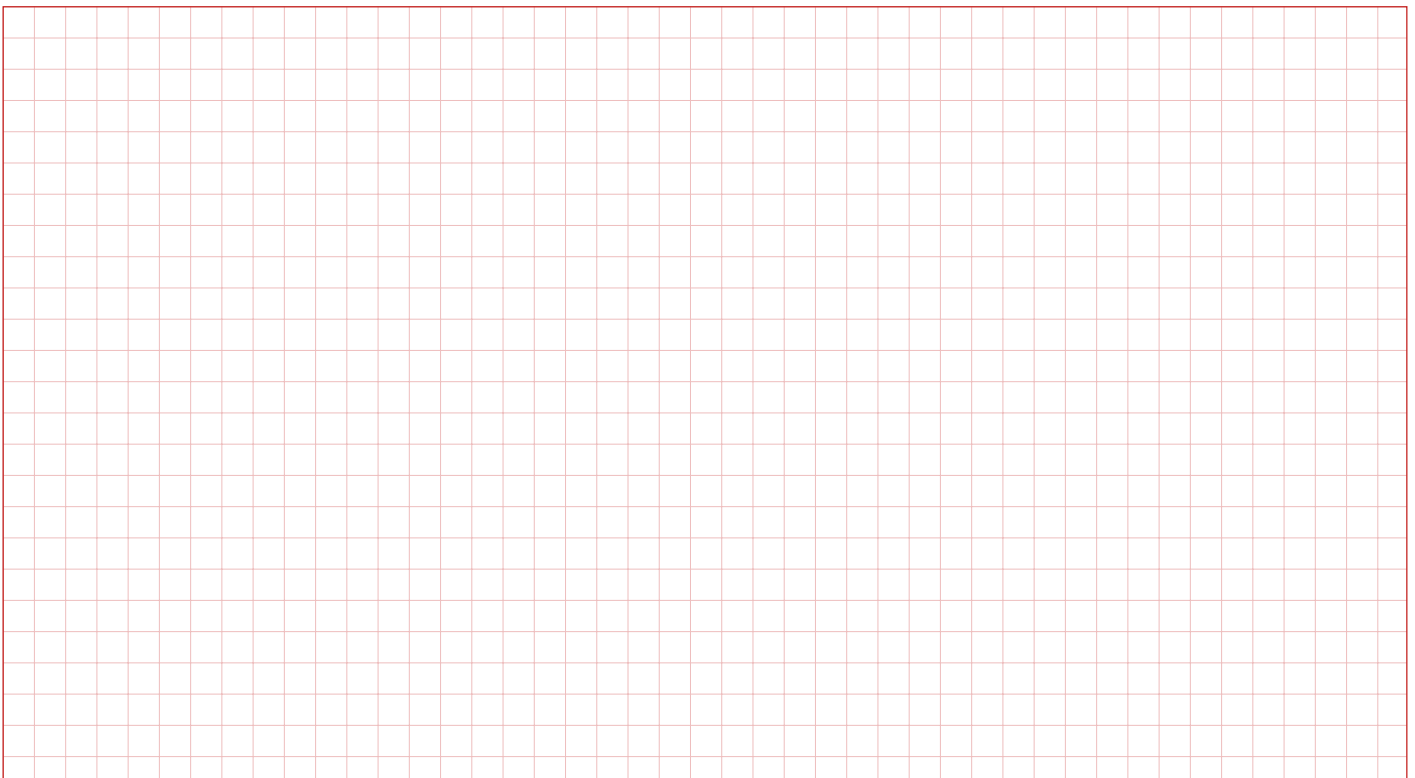
## Ausnahmebehandlung in C

- ▶ Ausnahmebehandlung über Rückgabewerte/globale Variablen
  - ▶ Rückgabewert hat mehrdeutige Semantik
  - ▶ Fehlergrund muss „irgendwo/wie“ codiert werden
  - ▶ Vermischung Ausnahmebehandlung mit Rest

```
FILE* f = fopen("test.txt", "w");  
if (f==NULL)  
    // Ausnahmebehandlung  
else {  
    size_t r = fwrite(..., ntimes, f);  
    if (r < ntimes){  
        // Ausnahmebehandlung  
    }  
}
```

- ▶ Wie werden Fehler weitergegeben?
- ▶ Wie skaliert der Ansatz?
- ▶ ...

## Notizen





# Inhalt

## Ausnahmen behandeln

- Ausnahmebehandlung in Java
  - Fangen einer Ausnahme
  - Fangen mehrerer Ausnahmen
  - finally-Block
  - Nicht gefangene Ausnahmen
  - Zusammenfassung

## Notizen

A large rectangular area filled with a fine grid of light gray lines, intended for taking handwritten notes. The grid consists of approximately 30 columns and 40 rows of small squares.

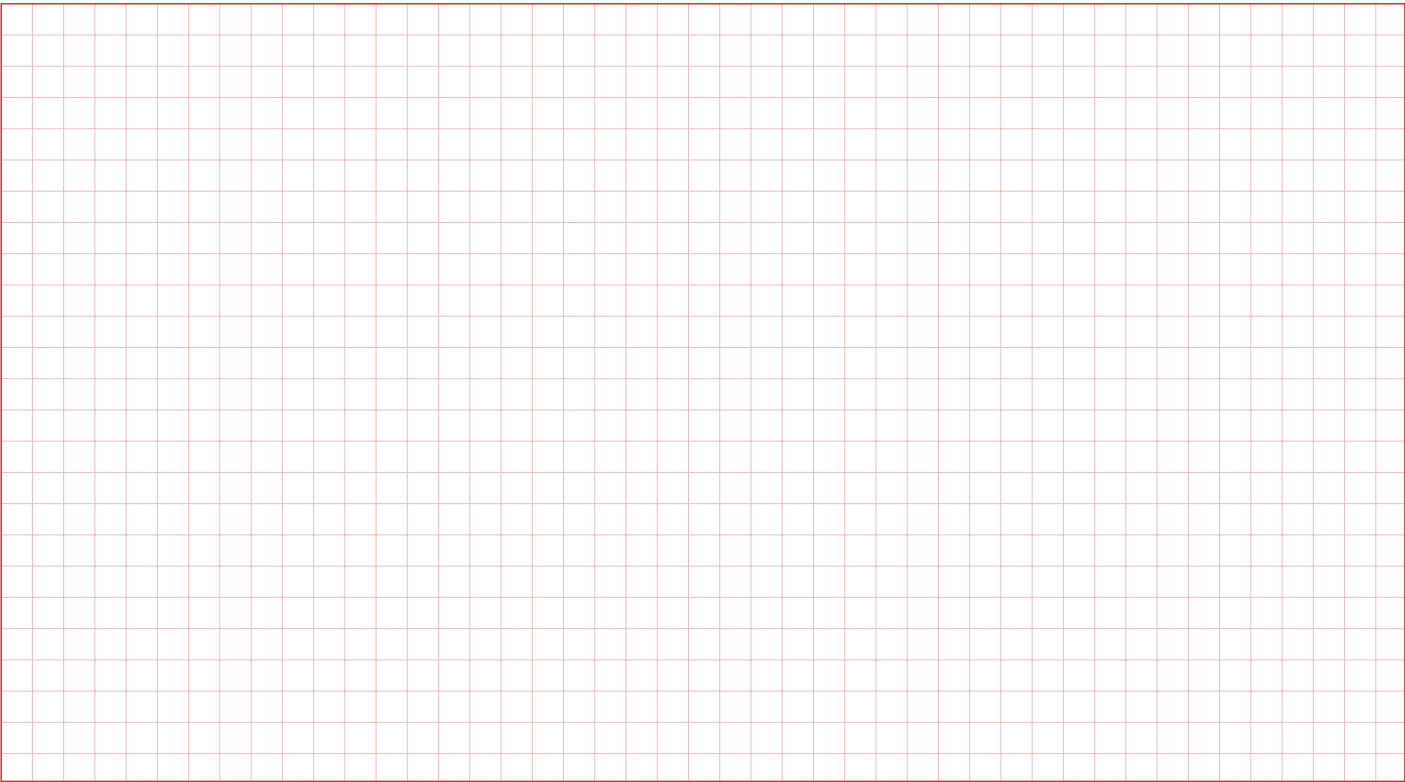
# Inhalt

## Ausnahmen behandeln

### Ausnahmebehandlung in Java

- Fangen einer Ausnahme
- Fangen mehrerer Ausnahmen
- finally-Block
- Nicht gefangene Ausnahmen
- Zusammenfassung

## Notizen

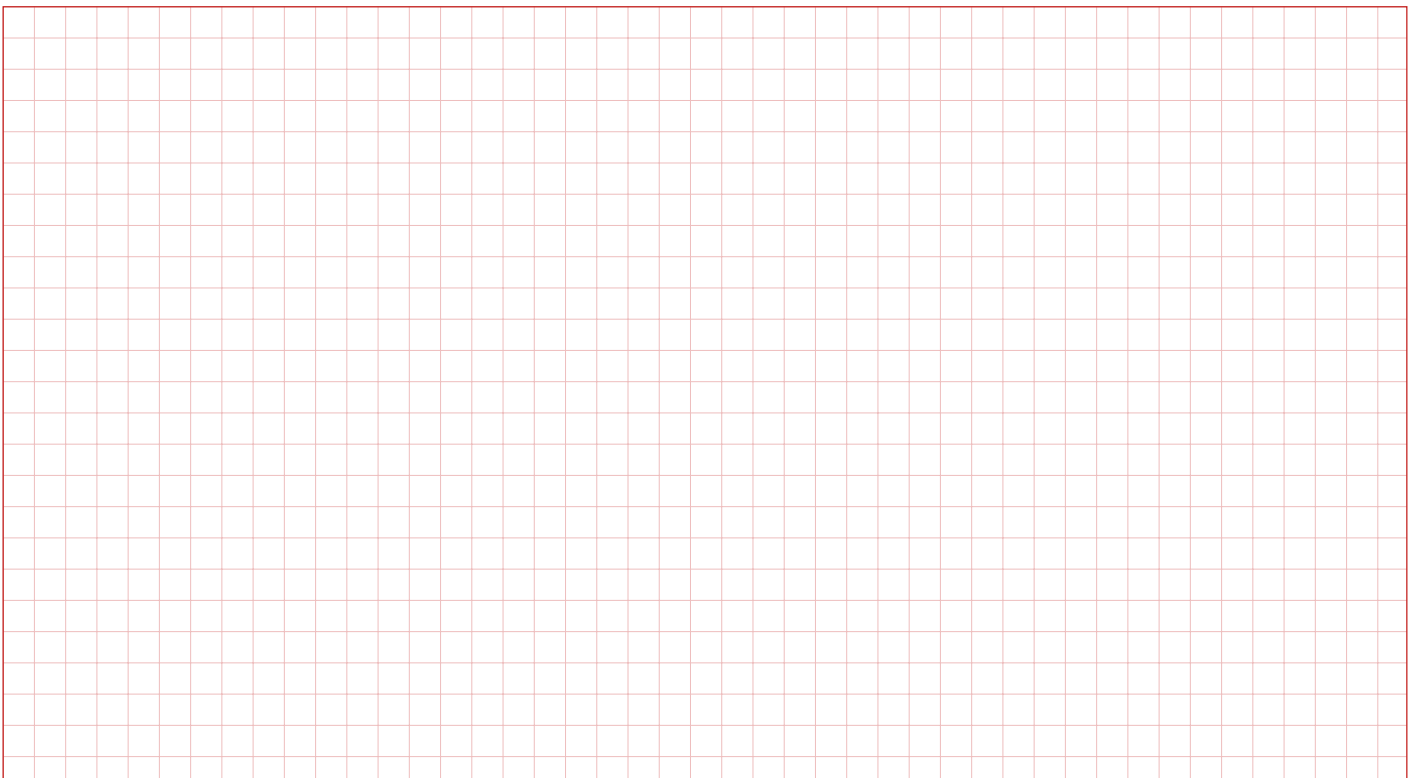


## Grundidee

```
try {  
    // Logik  
} catch (Ausnahme a){  
    // Ausnahmebehandlung  
}
```


- ✓ Blöcke: Klare Trennung zwischen Logik und Ausnahmebehandlung
- ✓ Ausnahmen sind Objekte: Erweiterbarer objektorientierter Ansatz
- ✓ Rückgabewerte haben eindeutige Semantik
- ✓ Klare Schnittstellen für Weitergabe
- ✓ Noch mehr Vorteile (später)
- ✗ Auch ein paar Nachteile (später)

## Notizen



## Beispiel

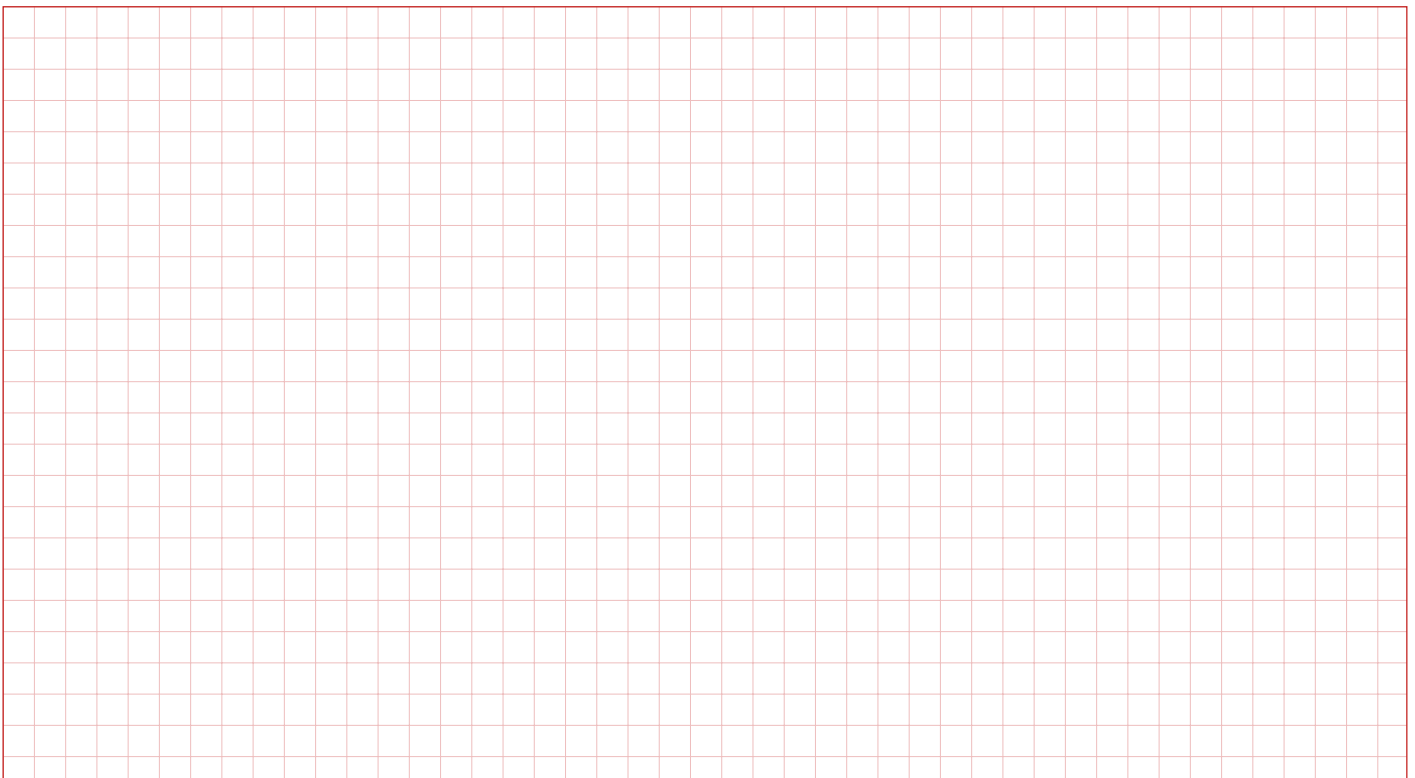
- Beispiel von vorher mit (etwas) Ausnahmebehandlung

```
25  runSomeExceptionHandlingExample
26 try{
27     Scanner scanner = new Scanner(System.in);
28     int i = scanner.nextInt();
29     out.printf("%d*d = %d%n", i, i, i*i);
30     scanner.close();
31 } catch (InputMismatchException exception){
32     err.printf("Zahl erwartet%n");
33 }
```

 ExceptionHandlingExamples.java

```
vier
Zahl erwartet
```

## Notizen

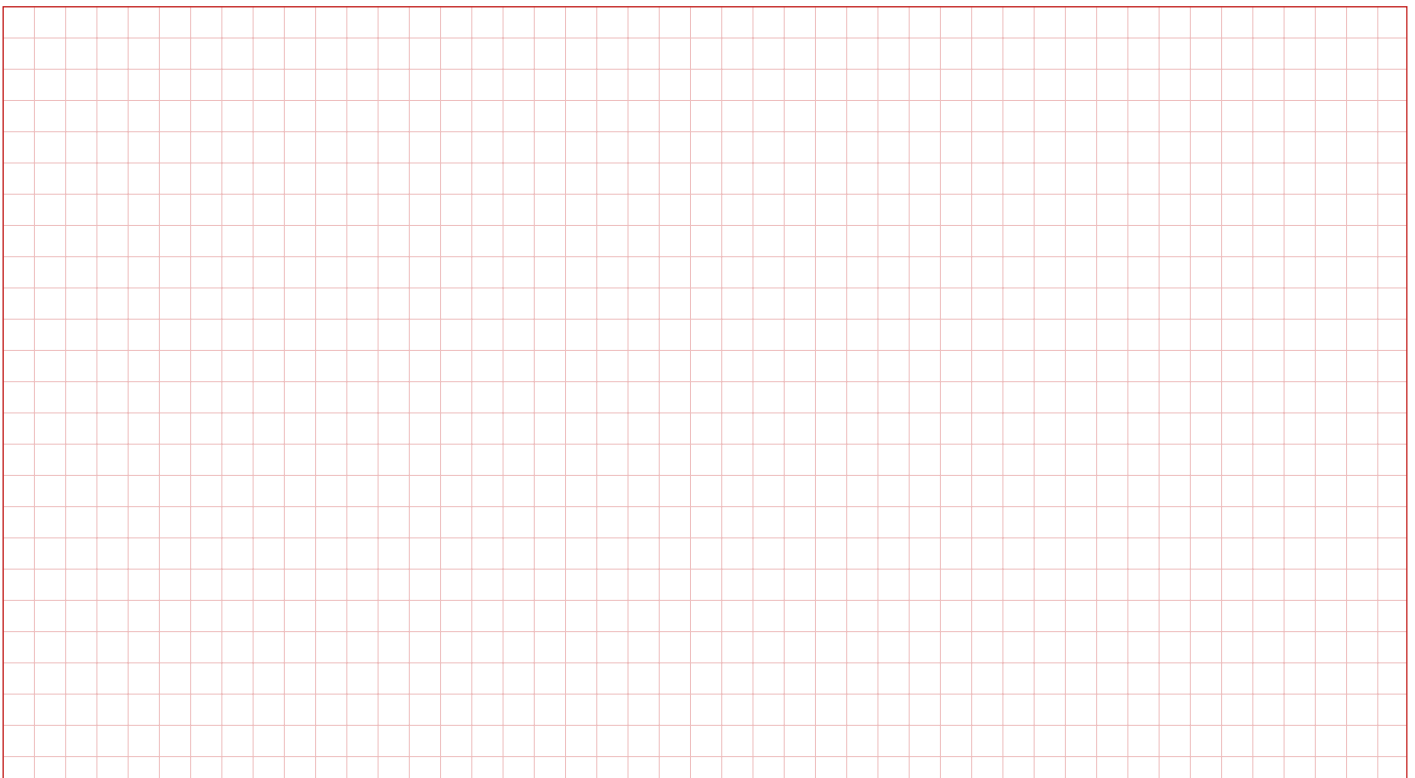


## Beispiel


```
try {  
    // ...  
    int i = scanner.nextInt();  
    // ...  
} catch (InputMismatchException exception){  
    err.printf("Zahl erwartet%n");  
}
```

- ▶ **try-Block**
  - ▶ Beinhaltet **Logik** von vorher
  - ▶ Wird bis `scanner.readInt()` ausgeführt
- ▶ **catch-Block**
  - ▶ Definiert **welche** Ausnahme **gefangen** wird
  - ▶ Beinhaltet **Ausnahmebehandlung**
- ▶ **Unschön**: Programmabbruch bei fehlerhafter Eingabe

## Notizen



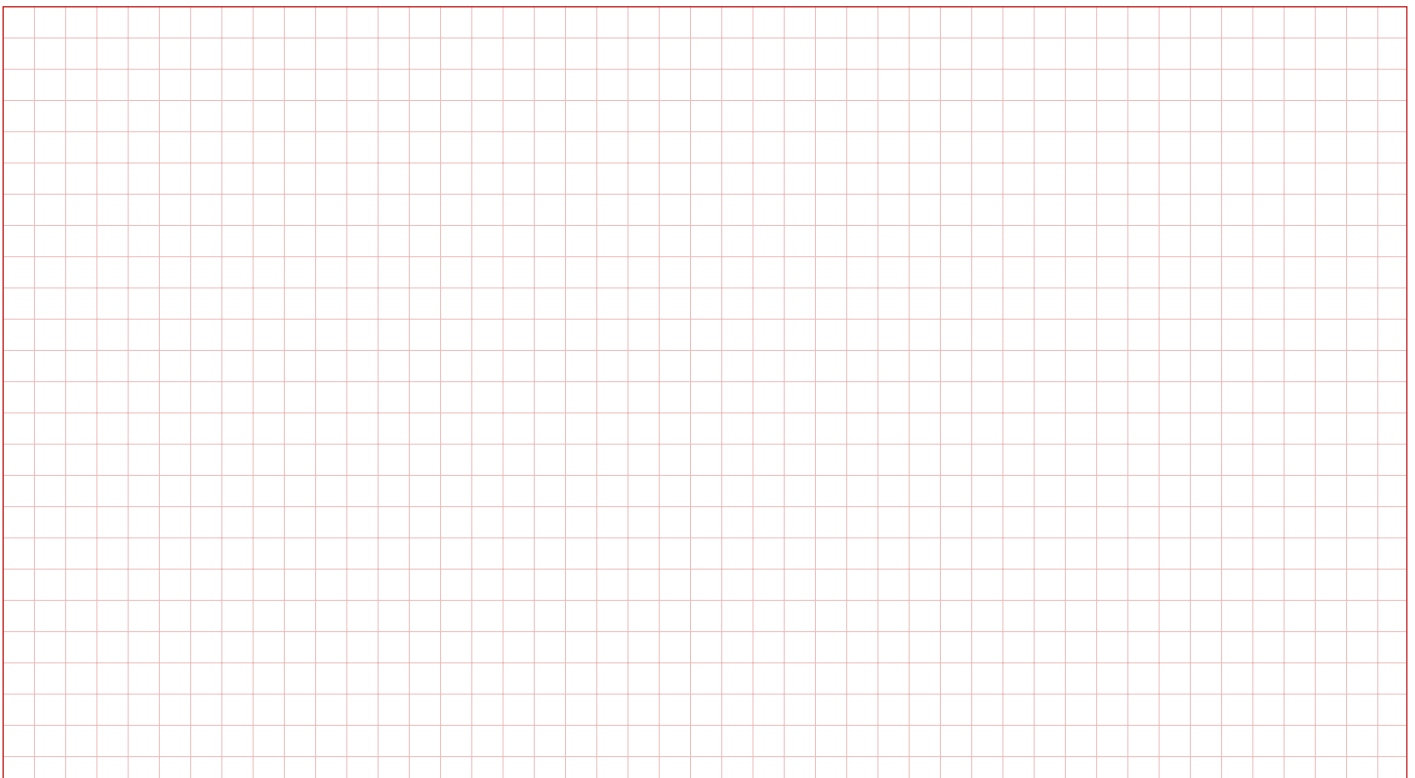
## Beispiel (verbessert)

```
39  runSomeExceptionHandlingExample2
40 boolean valid = false;
41 Scanner scanner = new Scanner(System.in);
42 do {
43     try{
44         int i = scanner.nextInt();
45         out.printf("%d*d = %d%n", i, i, i*i);
46         valid = true;
47     } catch (InputMismatchException exception){
48         err.printf("Bitte ganze Zahl eingeben!%n");
49         scanner.nextLine();
50     }
51 } while (!valid);
52 scanner.close();
```

 ExceptionHandlingExamples.java

```
vier
Bitte ganze Zahl eingeben!
4
4*4 = 16
```

## Notizen



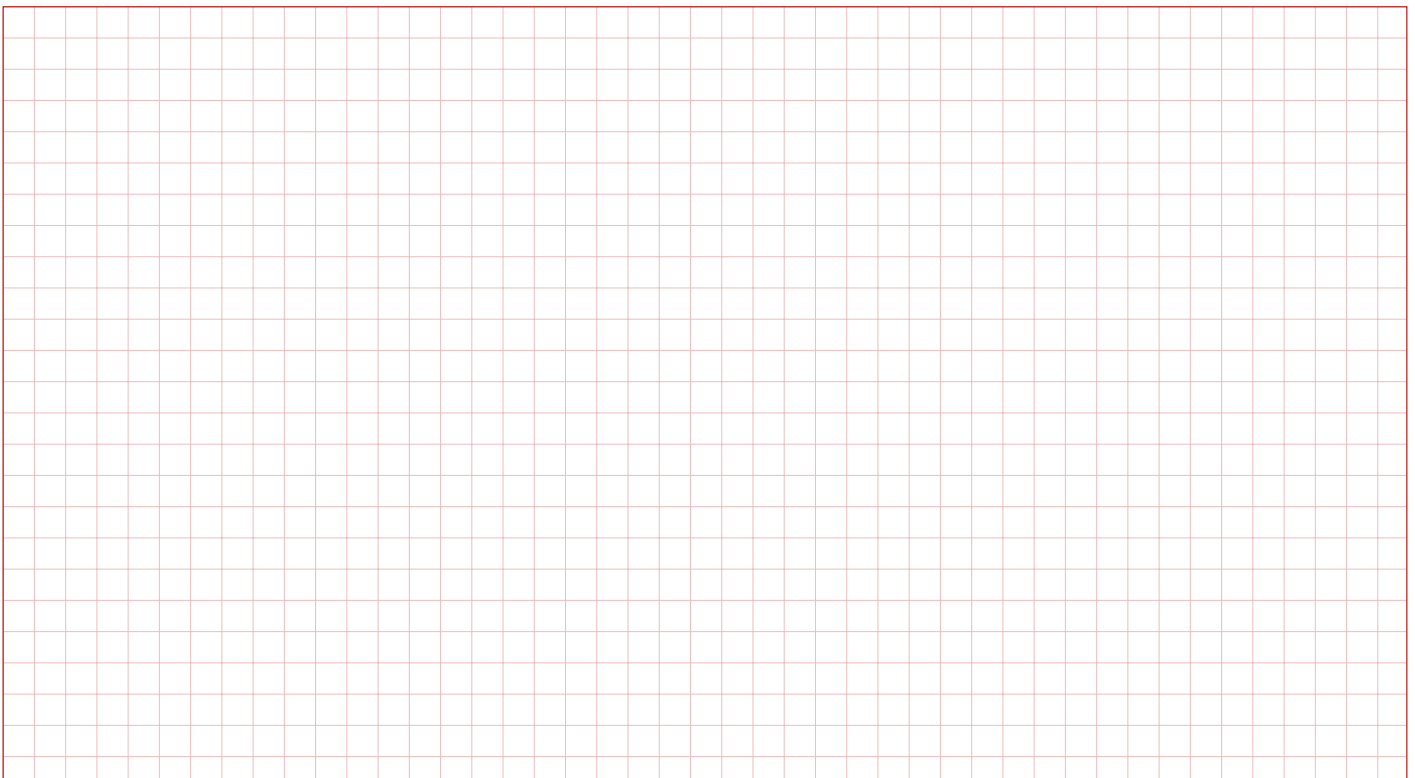
## Beispiel (verbessert)

```
boolean valid = false;
do {
    try{
        int i = scanner.nextInt();
        // ...
        valid = true;
    } catch (InputMismatchException exception){
        err.printf("Bitte ganze Zahl eingeben!\n");
    }
} while (!valid);
```

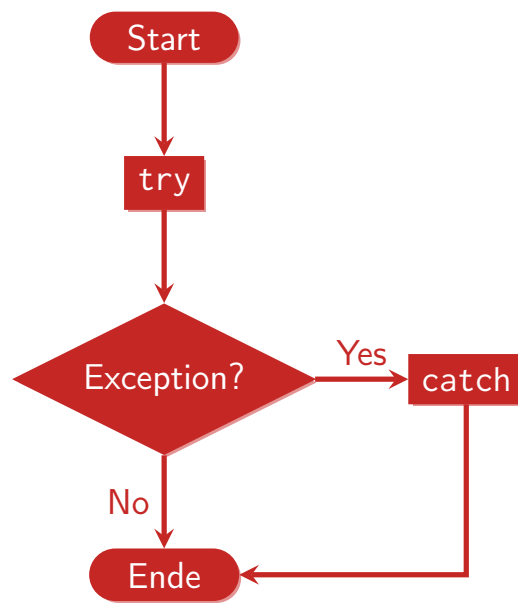
### ► Beobachtungen

- **Ausnahme** bricht **while**-Schleife nicht ab
- **Ausnahme** wird also wirklich gefangen
- Anweisungen in **try**-Block **nach Ausnahme** werden **nicht ausgeführt**
- **catch** muss **nicht** zu **Programmabbruch** führen

## Notizen



## Flussdiagramm: Ausnahmebehandlung (einfacher Fall)



## Notizen



# Inhalt

## Ausnahmen behandeln

### Ausnahmebehandlung in Java

- Fangen einer Ausnahme
- Fangen mehrerer Ausnahmen
- finally-Block
- Nicht gefangene Ausnahmen
- Zusammenfassung

## Notizen

A large rectangular area filled with a light gray grid pattern, intended for taking notes. The grid consists of small squares, approximately 20 columns wide and 30 rows high.

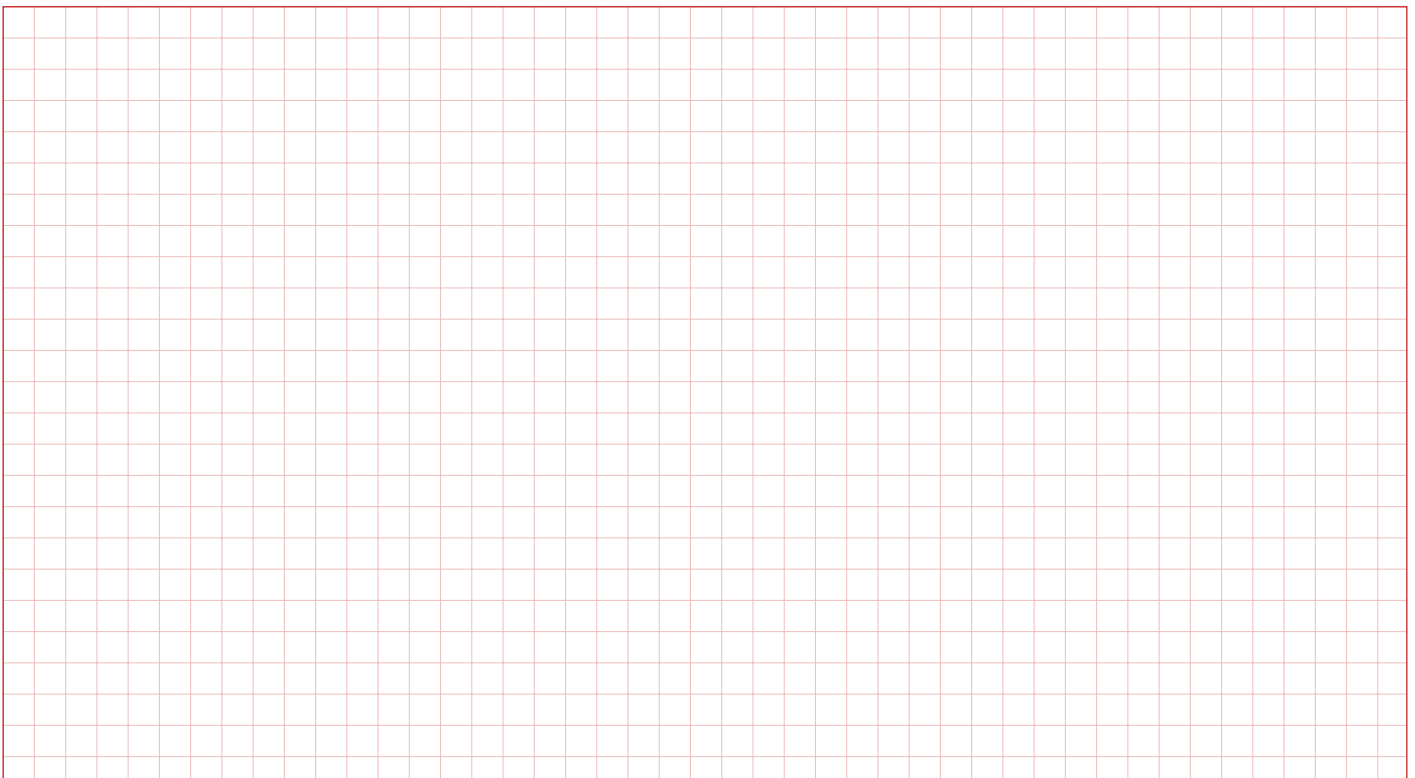
- ▶ Was passiert wenn Nutzer Eingabe **vorzeitig** beendet?

```
$ gradle runSomeExceptionHandlingExample2  
<Ctrl-D>/<Ctrl-Z> FEHLER
```


↗ `NoSuchElementException`

- ▶ **Andere** Ausnahme
- ▶ Die sollten wir **auch** noch fangen...

## Notizen



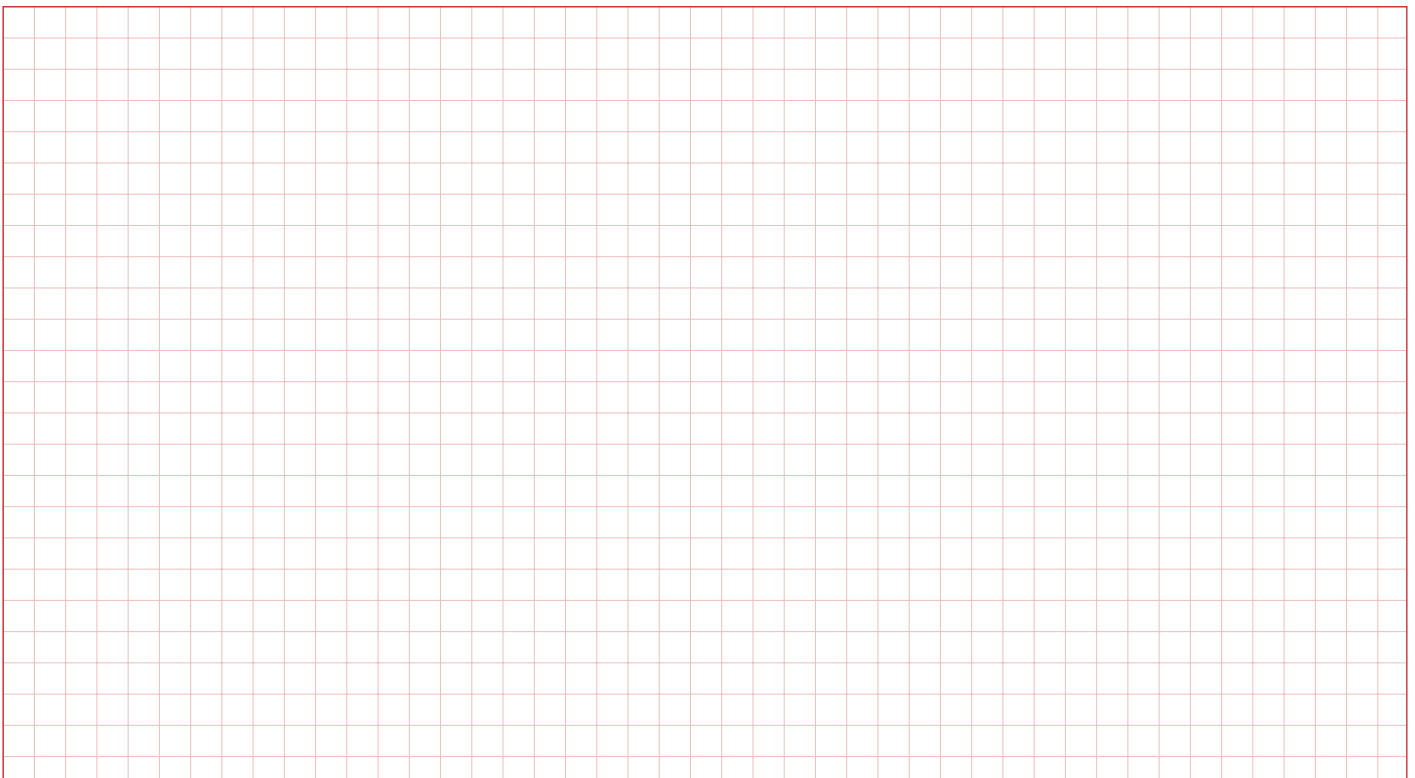
## Beispiel (nochmals verbessert)

```
59  runMoreExceptionHandlingExample
60 try{
61     int i = scanner.nextInt();
62     out.printf("%d*d = %d%n", i, i, i*i);
63     scanner.close();
64 } catch (InputMismatchException exception){
65     err.printf("Bitte ganze Zahl eingeben!%n");
66 } catch (NoSuchElementException exception){
67     err.printf("Kann nichts mehr lesen!%n");
68 }
```

 ExceptionHandlingExamples.java

```
<Ctrl-D>/<Ctrl-Z>
Kann nichts mehr lesen!
```

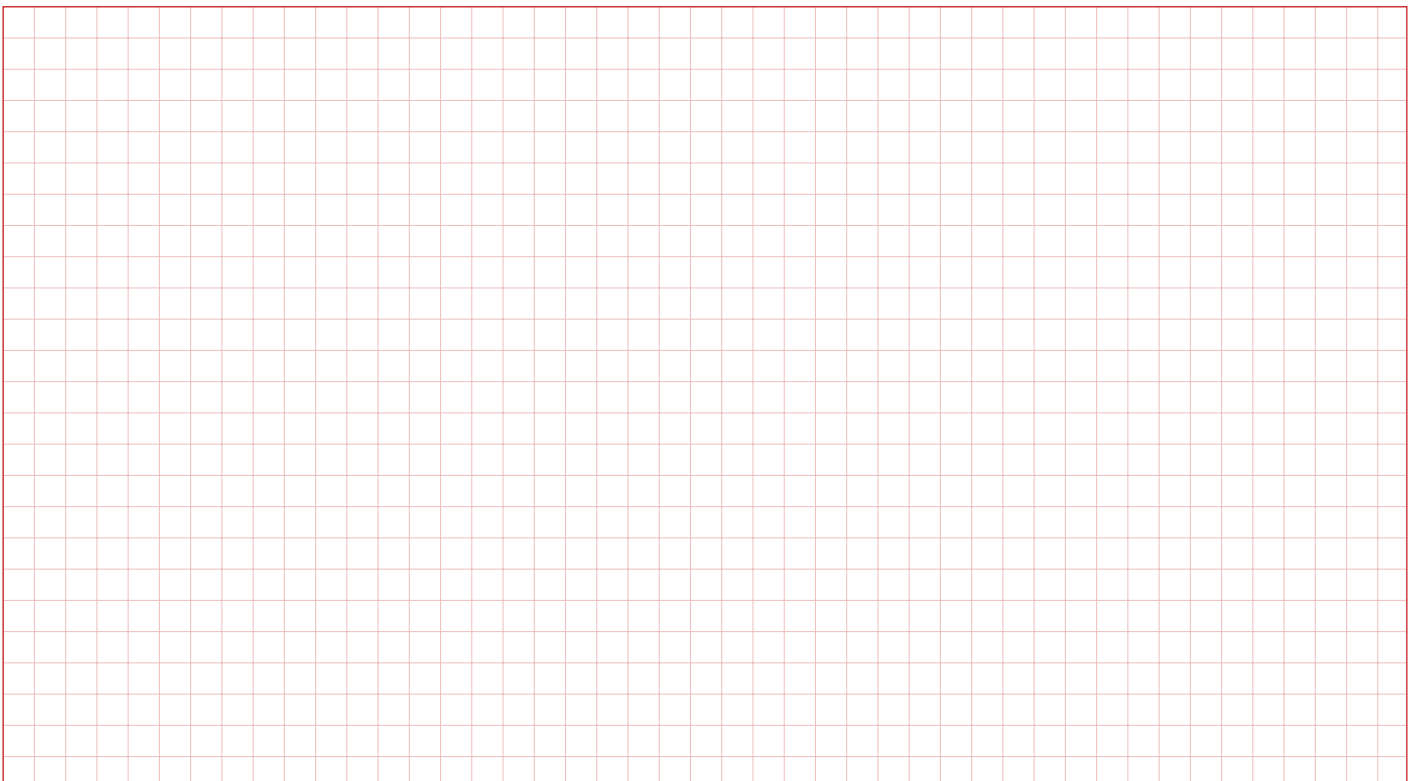
## Notizen



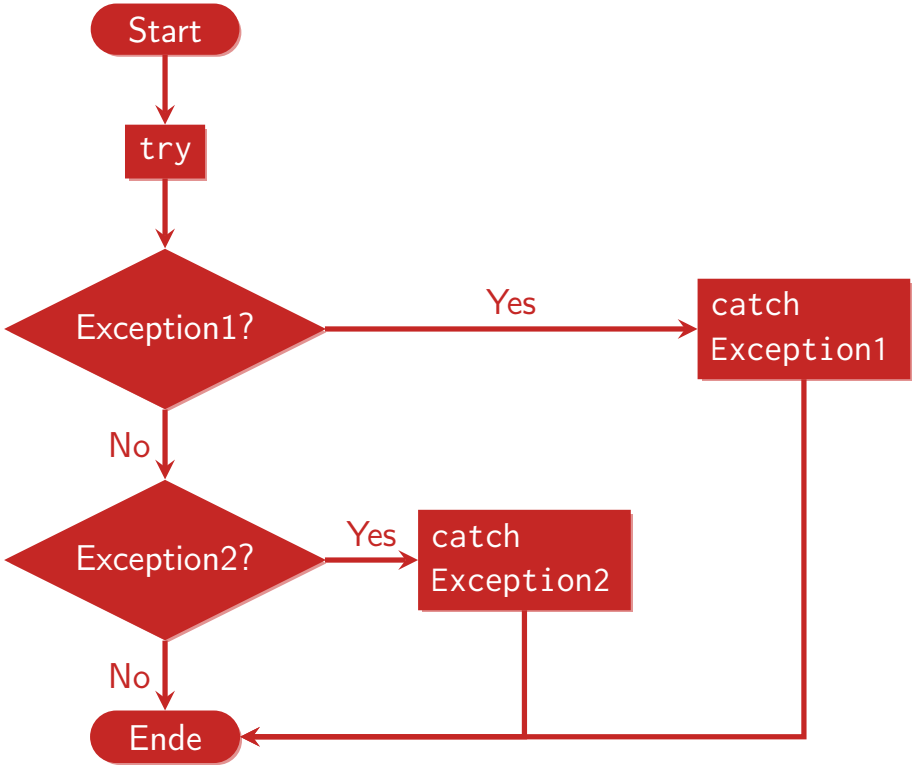
```
try{
    int i = scanner.nextInt();
    // ...
} catch (InputMismatchException exception){
    // ...
} catch (NoSuchElementException exception){
    // ...
}
```

- ▶ Mehrere **catch**-Blöcke möglich
- ▶ Es wird nur **der eine** **catch**-Block ausgeführt der zutrifft

## Notizen



## Flussdiagramm: Ausnahmebehandlung mehrerer Ausnahmen



## Notizen

# Inhalt

## Ausnahmen behandeln

### Ausnahmebehandlung in Java

- Fangen einer Ausnahme
- Fangen mehrerer Ausnahmen
- finally-Block
- Nicht gefangene Ausnahmen
- Zusammenfassung

## Notizen

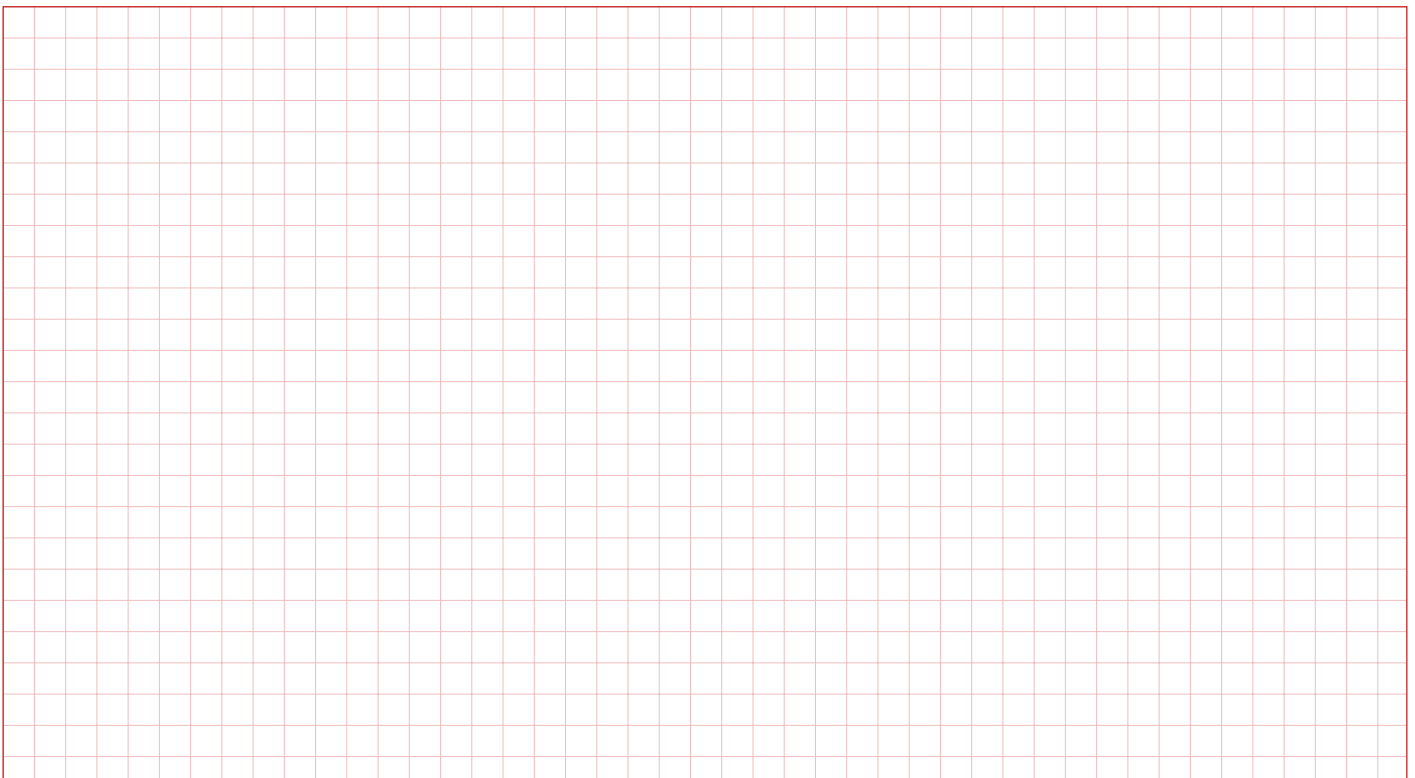
A large rectangular area filled with a fine grid of light gray lines, intended for taking handwritten notes. The grid consists of approximately 30 columns and 40 rows of small squares.

## Schließen von scanner

```
Scanner scanner = new Scanner(System.in);  
scanner.close();
```

- ▶ [↗](#) **Scanner** reserviert bei Erstellung **Betriebssystem-Ressourcen**
- ▶ Werden **evtl. nicht** von Garbage Collector freigegeben
- ▶ `scanner.close()` gibt Ressourcen frei
- ▶ **Problem**: `scanner.close()` nur wenn **keine** Ausnahme
- ▶ **Idee**: Einfach in alle **catch**-Blöcke einfügen!

## Notizen

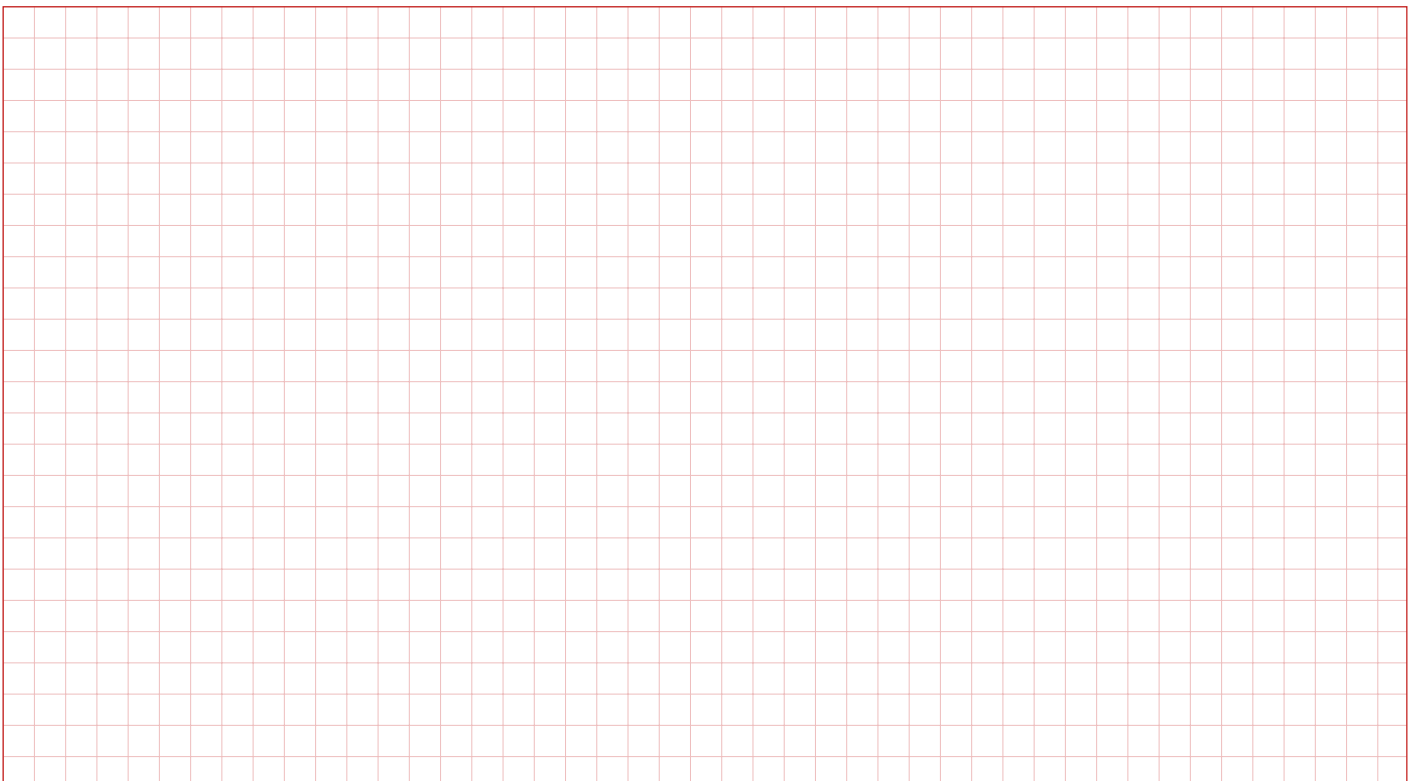


## Schließen von scanner

```
try{
    // ...
} catch (InputMismatchException e){
    scanner.close();
} catch (NoSuchElementException e){
    scanner.close();
}
```

- ▶ **Keine** gute Idee
  - ▶ Code-Dopplung
  - ▶ Fehleranfällig
  - ▶ **Noch** ein Problem (s. unten)

## Notizen



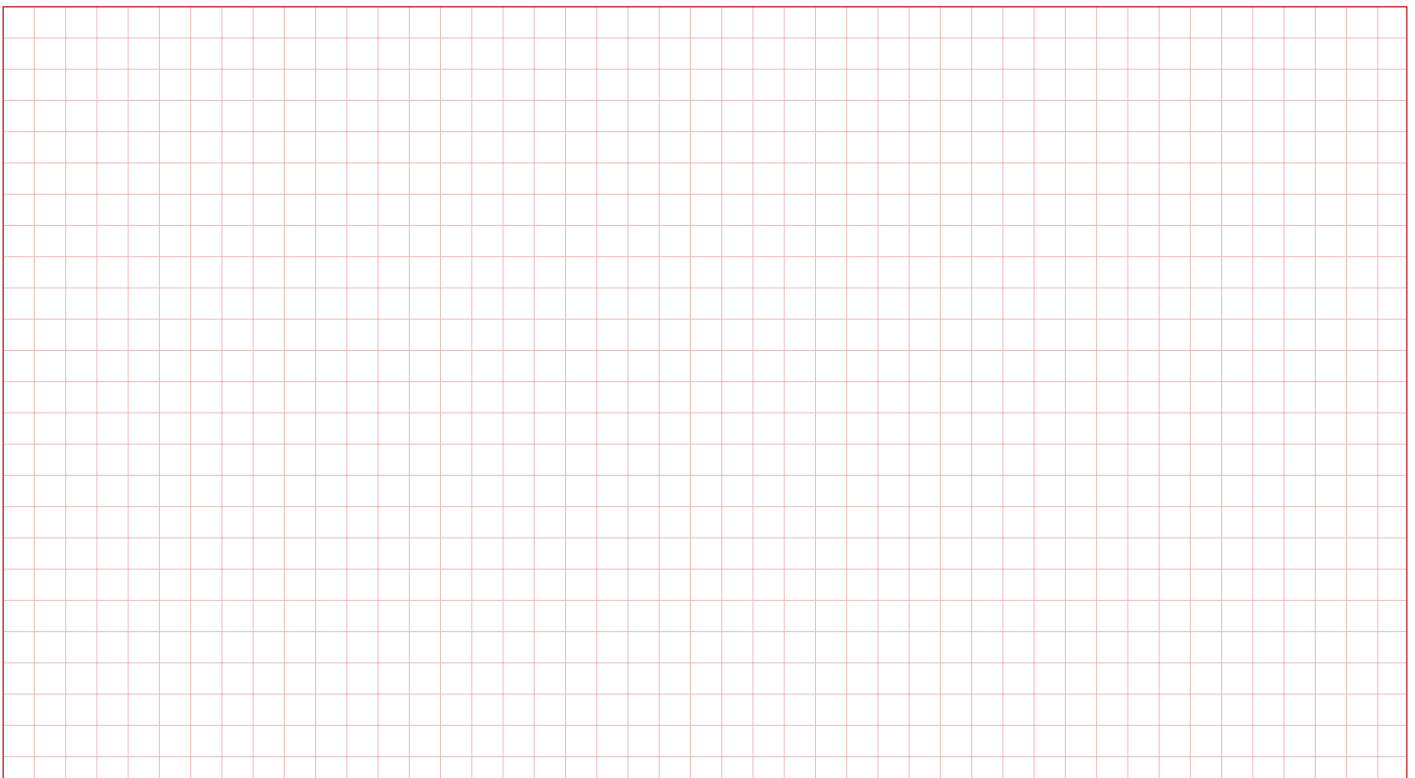


## Schließen von scanner


```
try{
    // ...
} catch (InputMismatchException e){
} catch (NoSuchElementException e){
}
scanner.close();
```

- ▶ Prinzipiell möglich
- ▶ Problem(e)
  - ▶ ↗ `Scanner.nextInt` kann noch ↗ `IllegalStateException` werfen
  - ▶ `try`-Block: Andere Operationen können ganz andere Ausnahmen werfen
- ▶ Wir benötigen etwas das definitiv immer ausgeführt wird
  - ▶ Wenn alles gut geht
  - ▶ Bei einer gefangenen Ausnahme
  - ▶ Bei einer nicht gefangenen Ausnahme
- ▶ Antwort: `finally`-Block

## Notizen

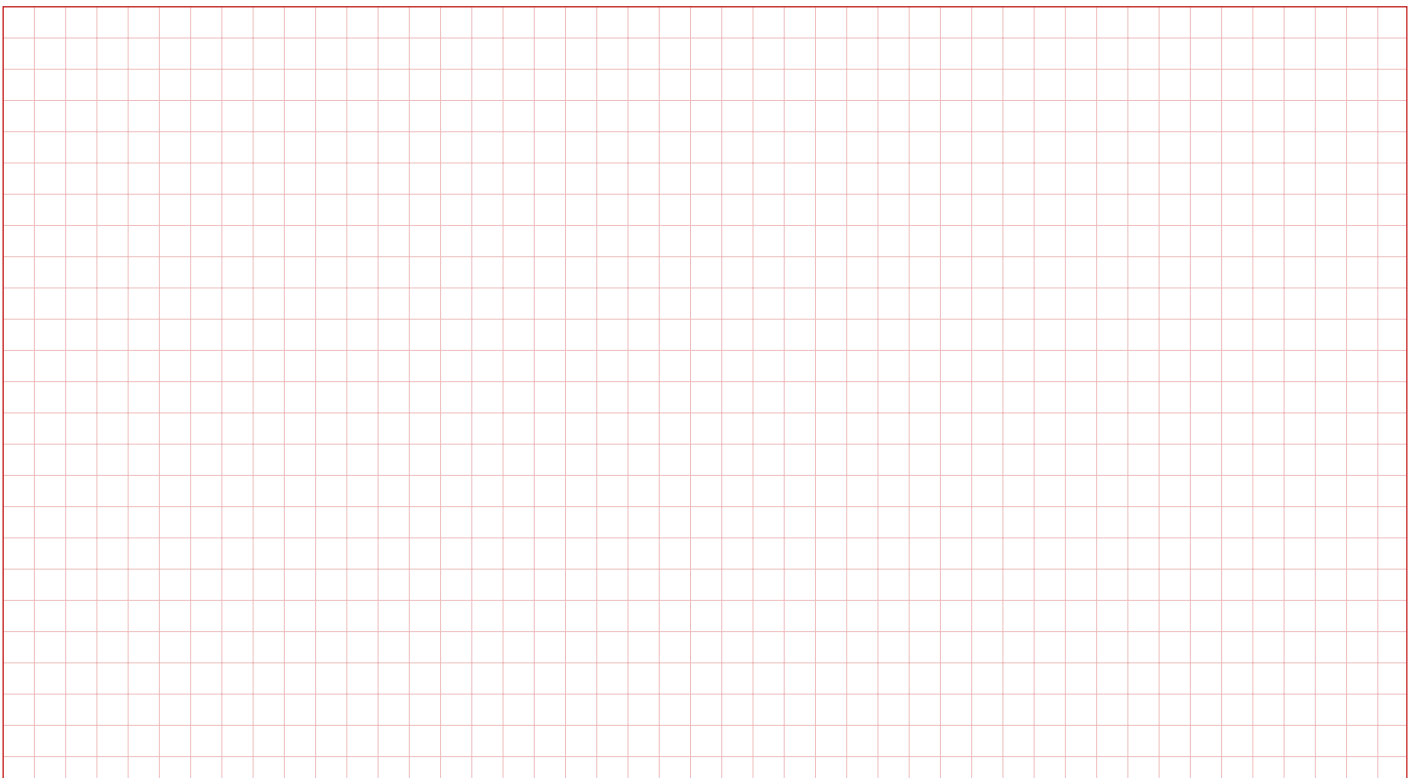


## Beispiel (noch mehr verbessert)

```
75  runFinallyExceptionHandlingExample  
76 try{  
77     int i = scanner.nextInt();  
78     out.printf("%d*d = %d%n", i, i, i*i);  
80 } catch (InputMismatchException exception){  
81     err.printf("Bitte ganze Zahl eingeben!%n");  
83 } catch (NoSuchElementException exception){  
84     err.printf("Kann nichts mehr lesen!%n");  
86 } finally {  
87     scanner.close();  
88     System.out.printf("Scanner geschlossen!%n");  
89 }
```

 ExceptionHandlingExamples.java

## Notizen



## Beispiel (noch mehr verbessert)

### ► Korrekte Eingabe

```
2
2*2 = 4
Scanner geschlossen!
```

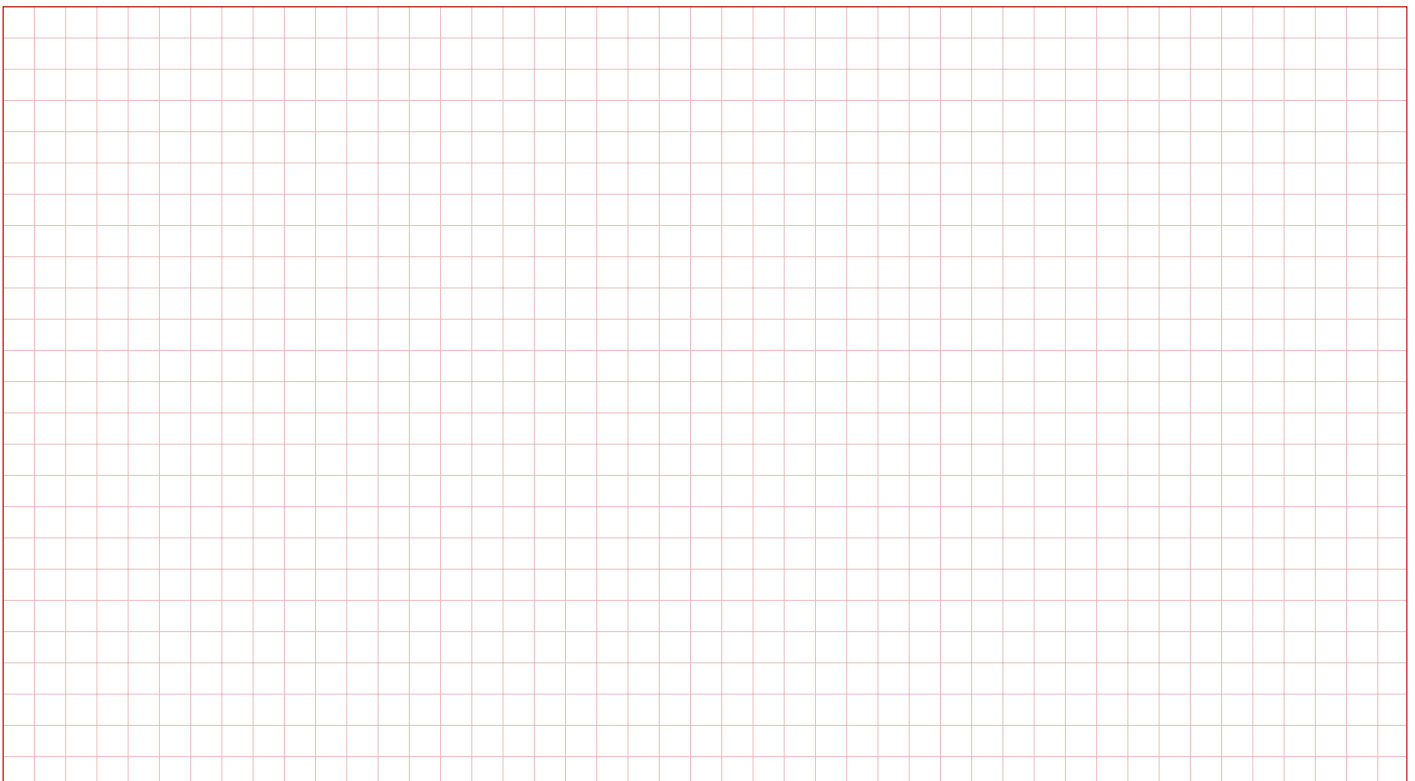
### ► Falsche Eingabe

```
vier
Bitte ganze Zahl eingeben!
Scanner geschlossen!
```

### ► Vorzeitiger Abbruch

```
<Ctrl-D>/<Ctrl-Z>
Kann nichts mehr lesen!
Scanner geschlossen!
```

## Notizen

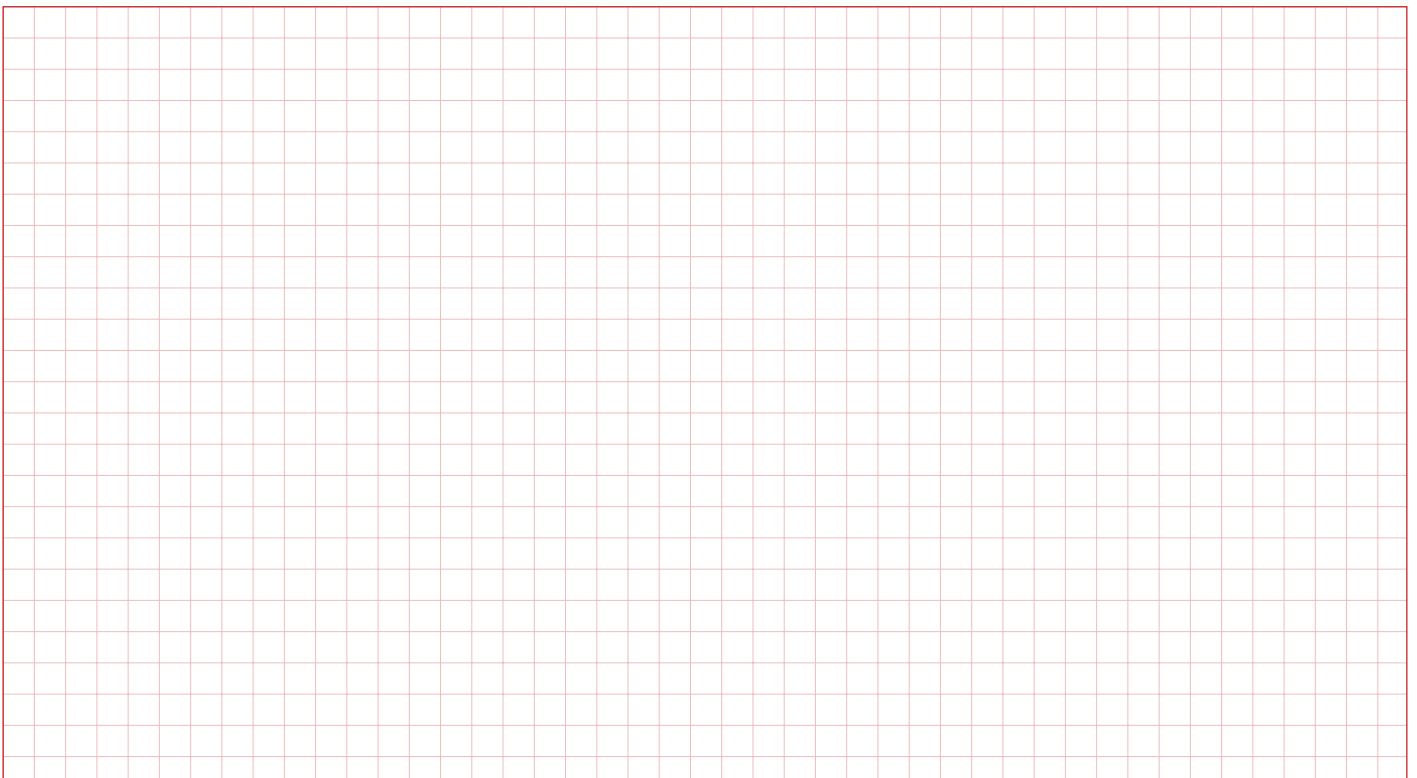


## finally

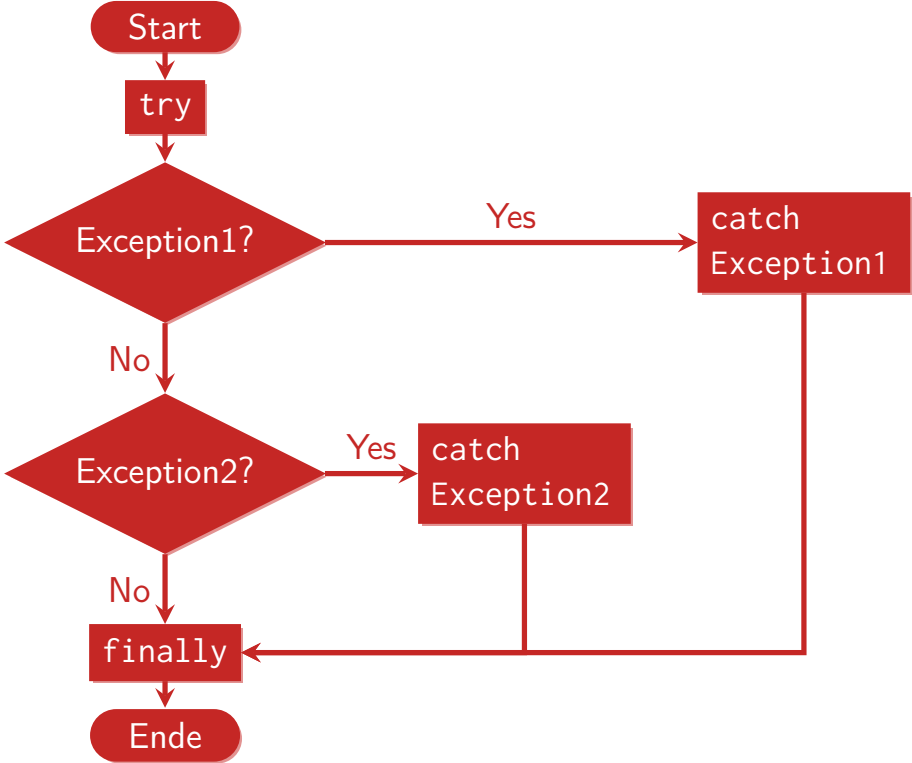
```
try{
    int i = scanner.nextInt();
    // ...
} catch (InputMismatchException exception){
    // ...
} catch (NoSuchElementException exception){
    // ...
} finally {
    scanner.close();
}
```

- ▶ **finally**-Block wird **immer** ausgeführt
- ▶ Auch bei [IllegalStateException](#) oder anderen (noch unbekannten) **Ausnahmen**
- ▶ Benutzen für **Aufräumaktionen**
- ▶ Für **Ressourcen-Management** gibt es noch eine **elegantere** Lösung (**später**)

## Notizen



## Flussdiagramm: Ausnahmebehandlung mit finally



## Notizen

# Inhalt

## Ausnahmen behandeln

### Ausnahmebehandlung in Java

- Fangen einer Ausnahme
- Fangen mehrerer Ausnahmen
- finally-Block
- Nicht gefangene Ausnahmen
- Zusammenfassung

## Notizen

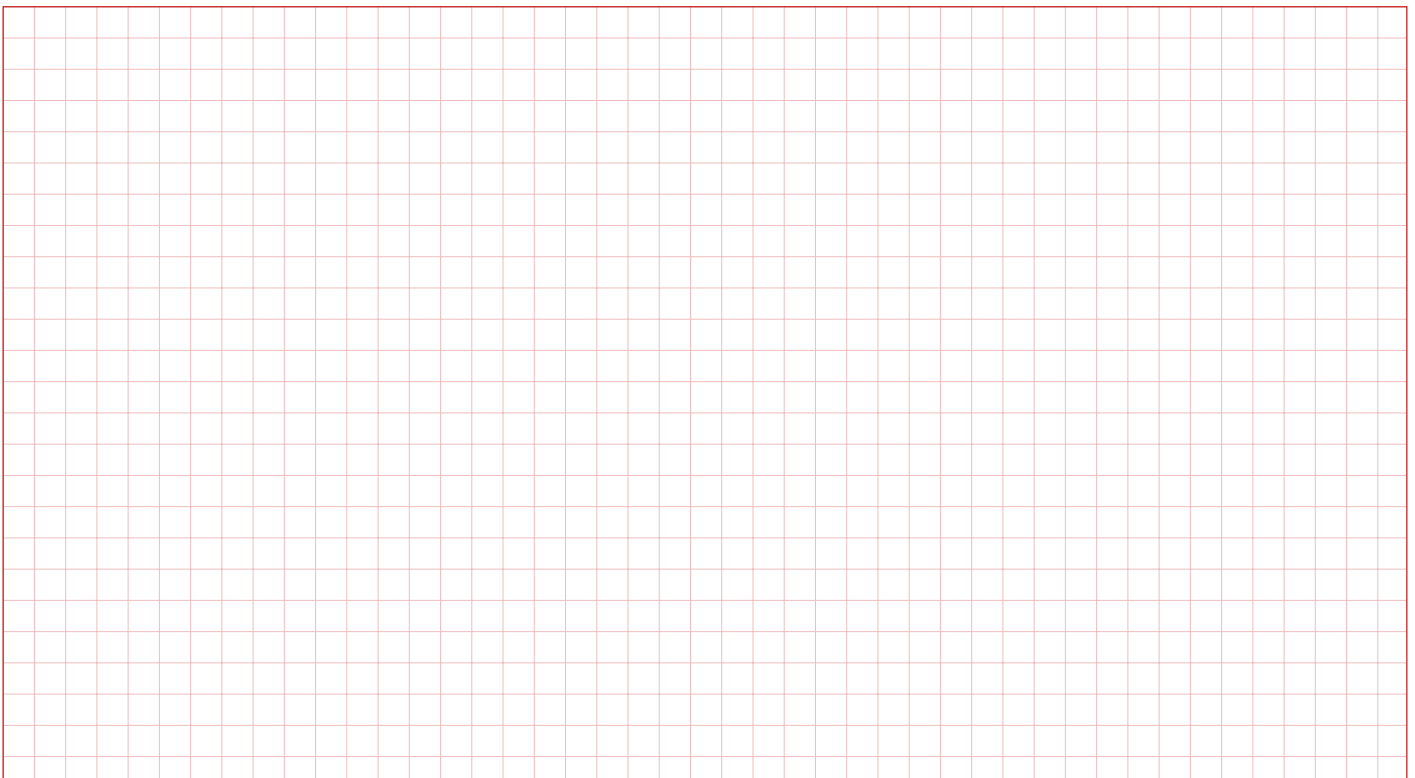
A large rectangular area filled with a fine grid of light gray lines, intended for taking handwritten notes. The grid consists of approximately 30 columns and 40 rows of small squares.

# Nicht gefangene Ausnahmen

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

- ▶ Was ist mit Ausnahmen, die **nicht** durch **catch** **gefangen** werden?
- ▶ Erkenntnis bisher: **finally** wird **garantiert** ausgeführt
- ▶ Ausnahmen ohne **passenden catch-Block** werden nach **finally** **weitergereicht**, bis
  - ▶ ein **catch-Block** sie fängt
  - ▶ sie bei der JVM (meist main) ankommt; dann **Fehlermeldung**

## Notizen



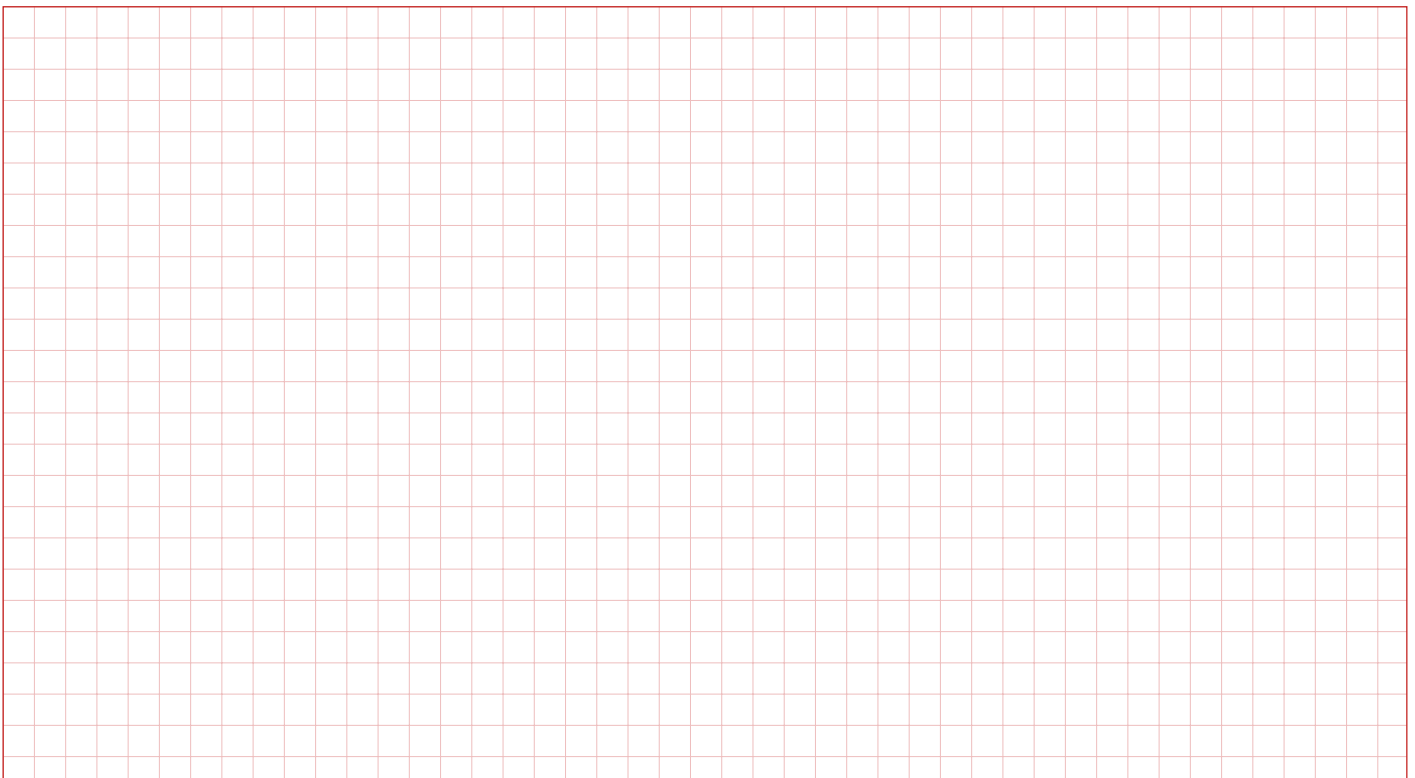
## Beispiel

### ► Beispiel von vorher als Methode

```
94 public static void readAndSquare(Scanner scanner) {
95     try{
96         int i = scanner.nextInt();
97         out.printf("%d*d = %d%n", i, i, i*i);
99     } catch (InputMismatchException exception){
100         err.printf("Bitte ganze Zahl eingeben!%n");
102     } catch (NoSuchElementException exception){
103         err.printf("Kann nichts mehr lesen!%n");
105     } finally {
106         scanner.close();
107         System.out.printf("Finally ausgeführt!%n");
108     }
109 }
```

📄 ExceptionHandlingExamples.java

## Notizen





## Beispiel

- ▶ Aufruf der Methode `readAndSquare` mit geschlossenem [Scanner](#)

```
114 try {  
115     Scanner scanner = new Scanner(System.in);  
116     scanner.close(); // provoziert Exception  
117     readAndSquare(scanner);  
118 } catch (IllegalStateException exception){  
119     err.println("Methode hat IllegalStateException geworfen");  
120 }
```

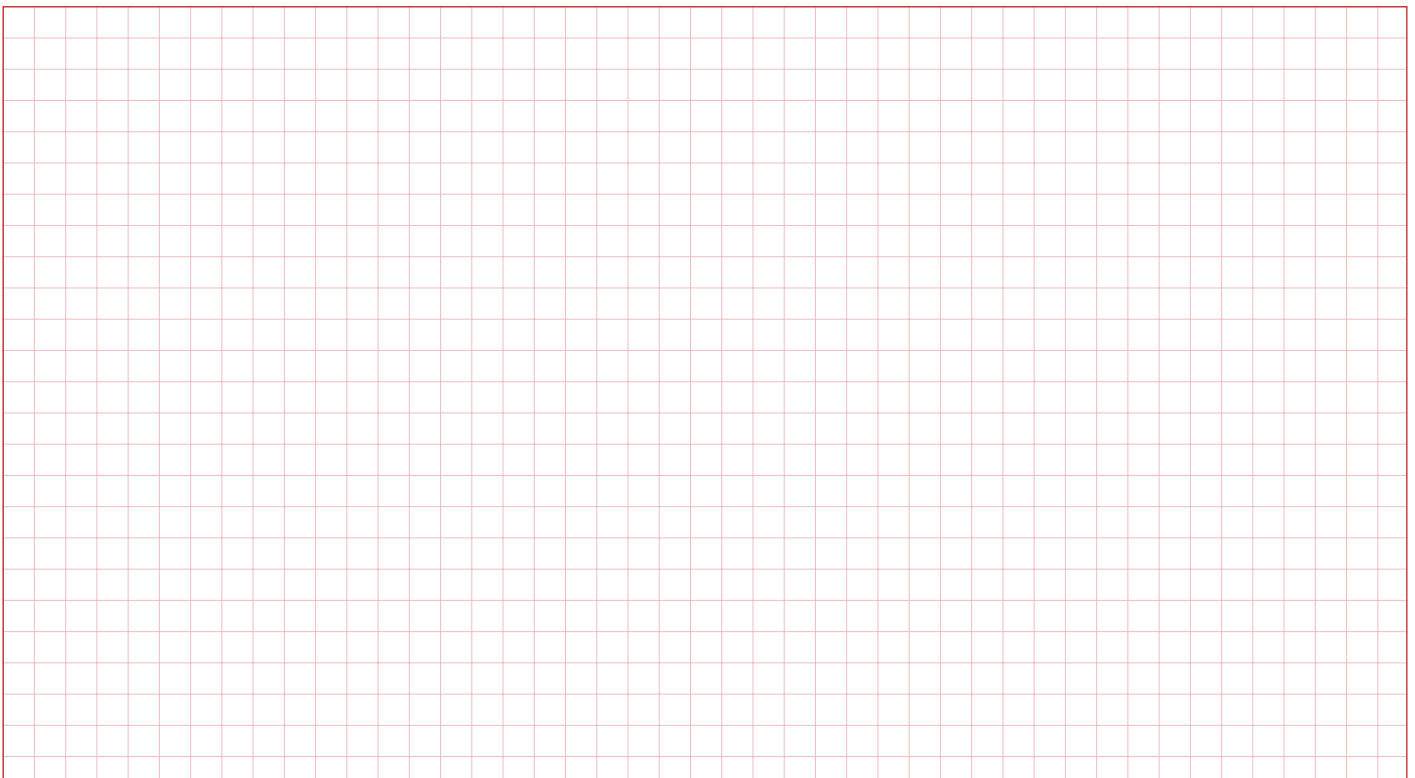
[ExceptionHandlingExamples.java](#)

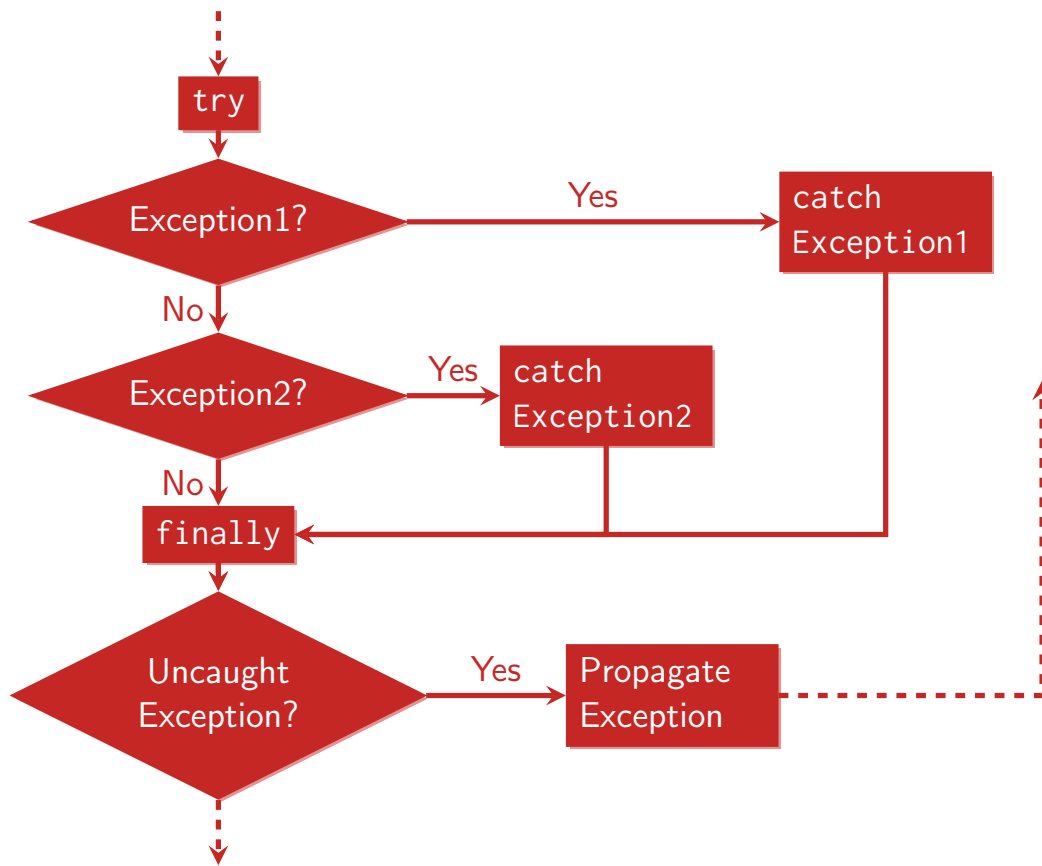
- ▶ `scanner.close` provoziert [IllegalStateException](#) in `readAndSquare`

Finally ausgeführt!  
Methode hat `IllegalStateException` geworfen

- ▶ Methode wird in **aufrufender Methode** gefangen
- ▶ **Wenn nicht**: Weitergabe an nächste Methode, usw.

## Notizen





## Notizen

# Inhalt

## Ausnahmen behandeln

### Ausnahmebehandlung in Java

- Fangen einer Ausnahme
- Fangen mehrerer Ausnahmen
- finally-Block
- Nicht gefangene Ausnahmen
- Zusammenfassung

## Notizen

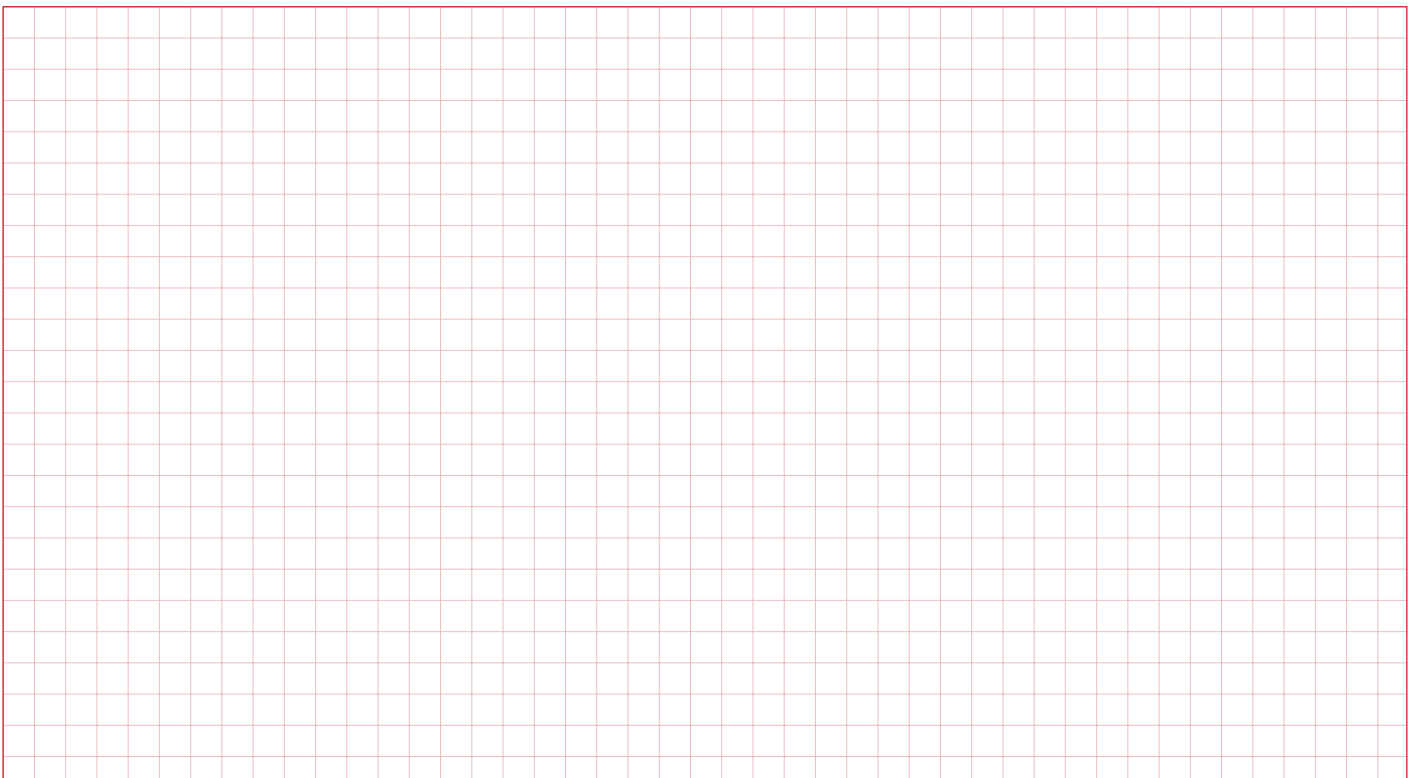
A large rectangular area filled with a fine grid of light gray lines, intended for taking handwritten notes. The grid consists of approximately 30 columns and 40 rows of small squares.

## Zusammenfassung

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

- ▶ **Ausnahmebehandlung** in Java
  - ▶ **Trennung** von Logik und Ausnahmebehandlung
  - ▶ **Keine Überladung** der Semantik von Rückgabewerten
- ▶ **try-Block**
  - ▶ **Enthält Logik**, die Ausnahmen erzeugen kann
  - ▶ Bei Ausnahme **Abbruch**
- ▶ **catch-Block**
  - ▶ Definiert zu **fangende** Ausnahme
  - ▶ **Mehrere catch-Blöcke** möglich
  - ▶ Beinhaltet **Ausnahmebehandlung**
- ▶ **finally-Block**:
  - ▶ Wird **immer** ausgeführt
  - ▶ Nur **einer** möglich
- ▶ **Nicht gefangene** Ausnahmen werden **weitergereicht**

## Notizen



# Inhalt

## Ausnahmen behandeln

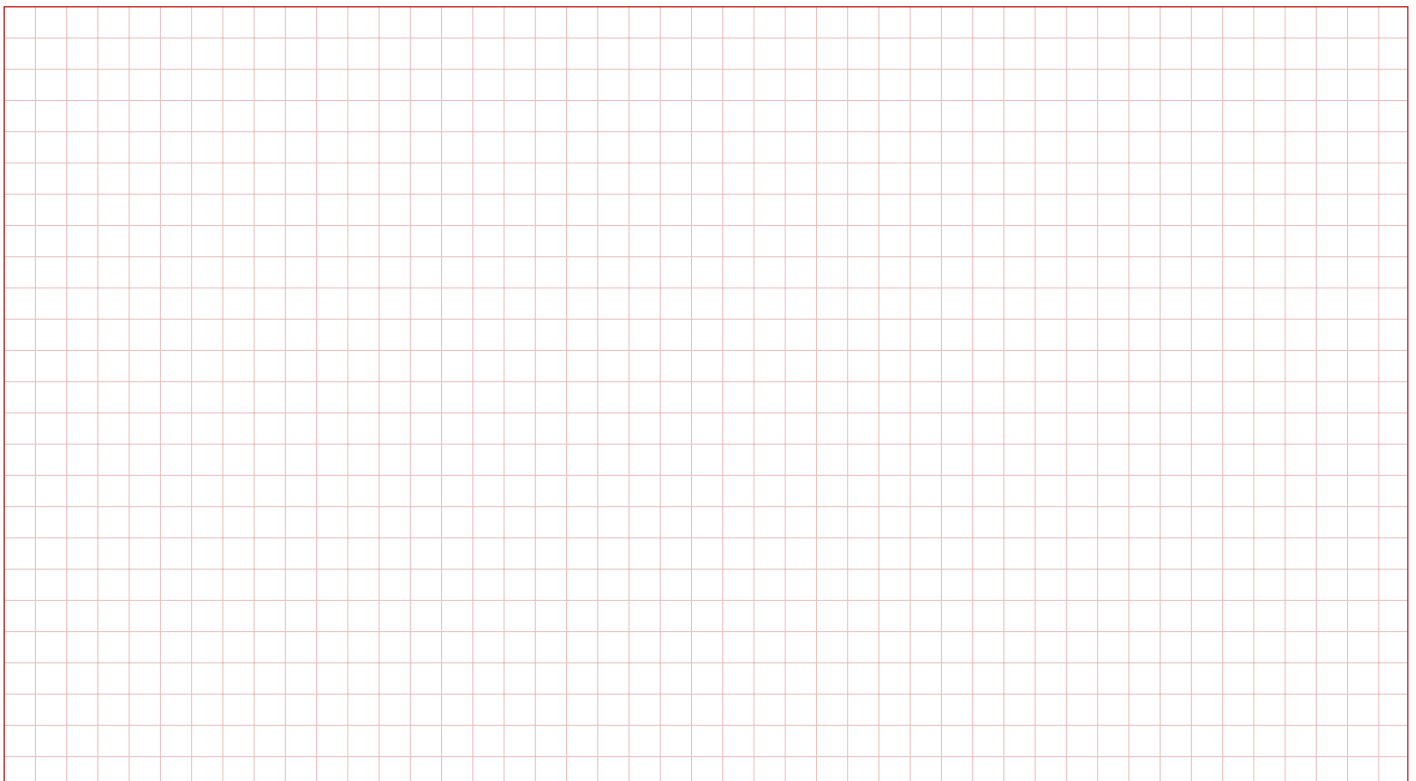
Auffangen mehrerer Ausnahmetypen

Multi-catch

Ausnutzen der Hierarchie

Zusammenfassung

## Notizen



# Inhalt

## Ausnahmen behandeln

Auffangen mehrerer Ausnahmetypen

Multi-catch

Ausnutzen der Hierarchie

Zusammenfassung

## Notizen

A large rectangular area filled with a fine grid of light gray lines, intended for taking handwritten notes. The grid consists of approximately 30 columns and 40 rows of small squares.

## Multi-catch

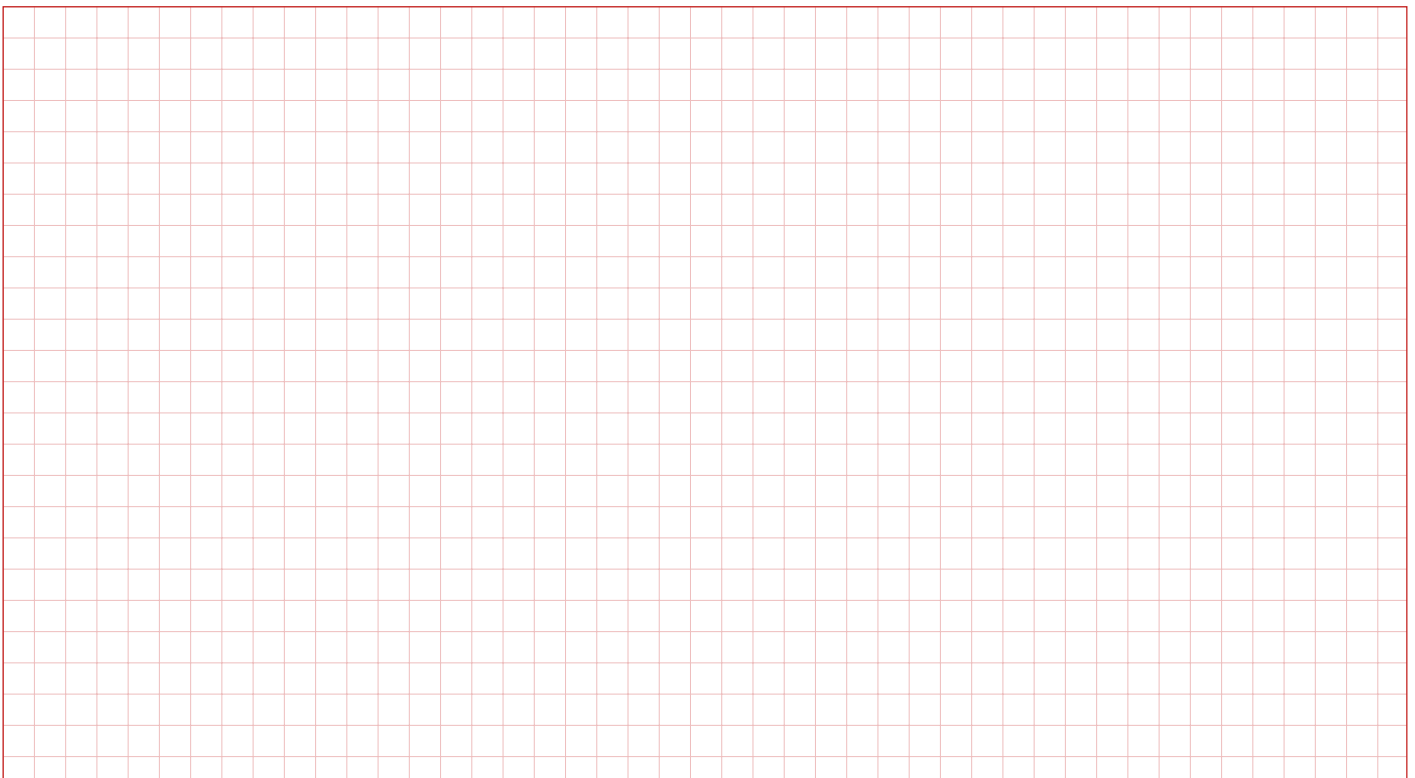
- ▶ Wir wollen noch die [IllegalStateException](#) behandeln:

```
try { ... }  
catch (InputMismatchException exception){  
    err.printf("Bitte ganze Zahl eingeben!\n");  
}  
catch (NoSuchElementException exception){  
    err.printf("Kann nichts mehr lesen!\n");  
}  
catch (IllegalStateException exception){  
    err.printf("Kann nichts mehr lesen!\n");  
} finally { ... }
```

- ▶ **Problem**

- ▶ Behandlung von [NoSuchElementException](#) und [IllegalArgumentException](#) ist gleich
- ▶ Code-Dopplung

## Notizen



## Multi-catch

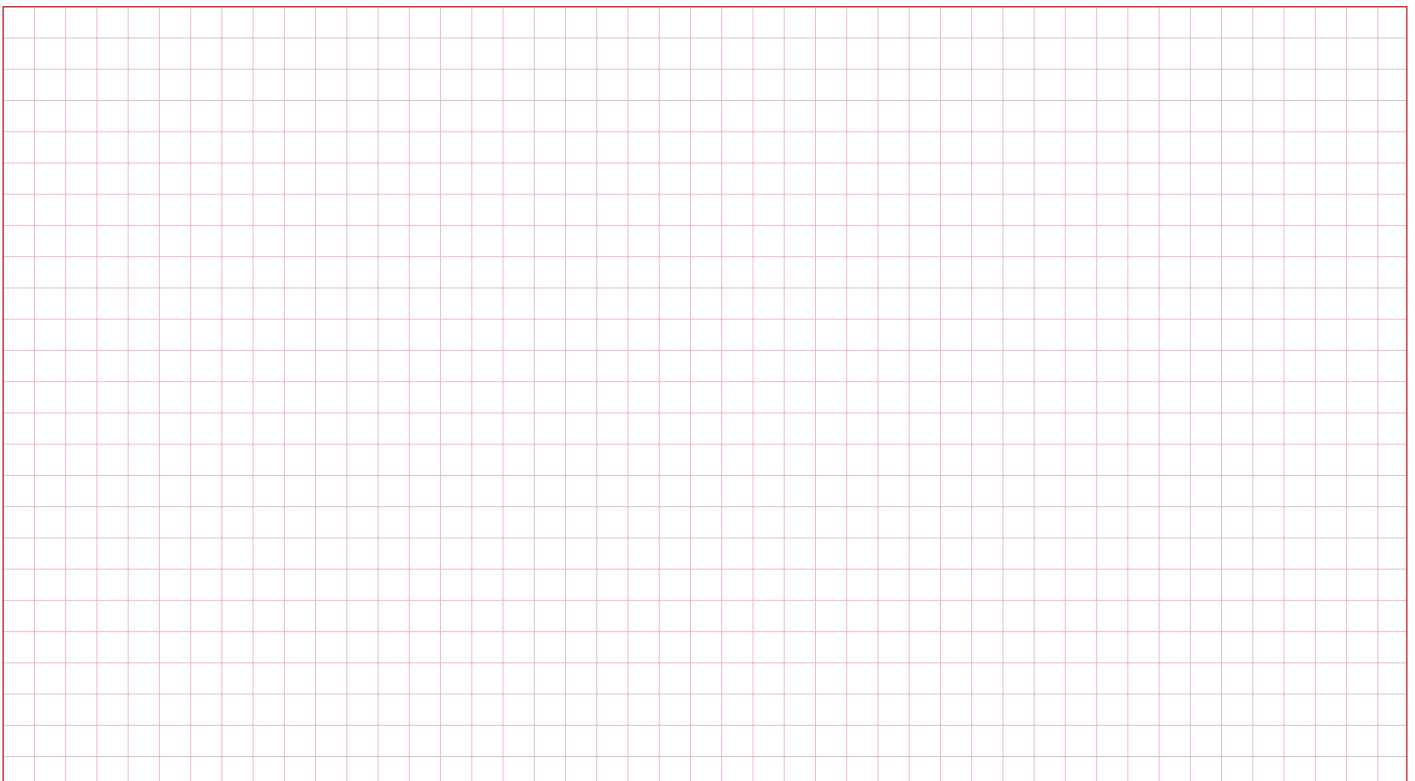
- ▶ Ein **catch** kann **mehrere** Ausnahmetypen behandeln
- ▶ Allgemeine **Syntax**

```
catch (Exception1 | Exception2 | ... | ExceptionN e){  
}
```

- ▶ **Bedingung:** Keine zwei Ausnahmetypen dürfen in einer „ist ein“-Beziehung stehen (später mehr)
- ▶ **Anwendung** auf Beispiel

```
try { ... }  
catch (InputMismatchException exception){ ... }  
catch (NoSuchElementException | IllegalStateException exception){  
    err.printf("Kann nichts mehr lesen!\n");  
} finally { ... }
```

## Notizen





# Inhalt

## Ausnahmen behandeln

Auffangen mehrerer Ausnahmetypen

Multi-catch

Ausnutzen der Hierarchie

Zusammenfassung

## Notizen

A large rectangular area filled with a light gray grid pattern, intended for taking notes. The grid consists of small squares, with a slightly thicker border around the perimeter.

## Welches catch wird verwendet?

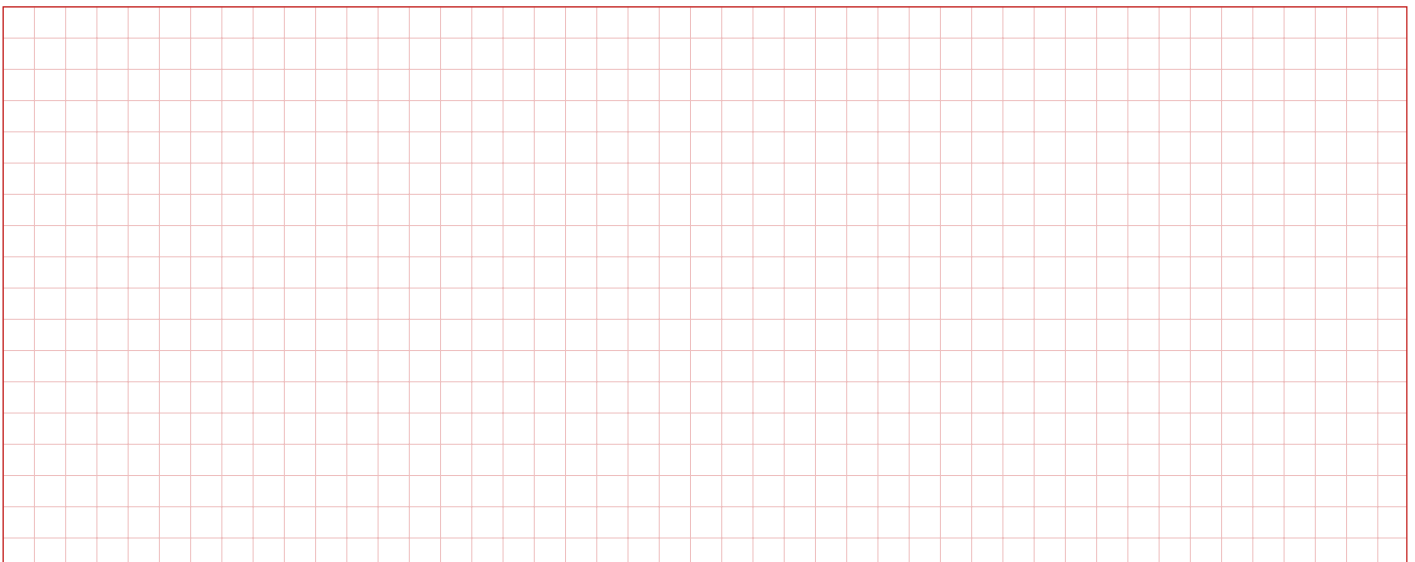
- ▶ Wie entscheidet die JVM welches **catch** verwendet wird?
- ▶ Ausnahmen sind **ganz normale** Java-Klassen mit einer **Hierarchie**
- ▶ Ausnahmen stehen in „ist ein“-Beziehung



- ▶ „ist ein“-Beziehung definiert **welches catch** verwendet wird

## Notizen

- InputMismatchException ist eine NoSuchElementException ist eine RuntimeException, usw.
- IllegalStateException ist *nicht* kompatibel mit NoSuchElementException und InputMismatchException.

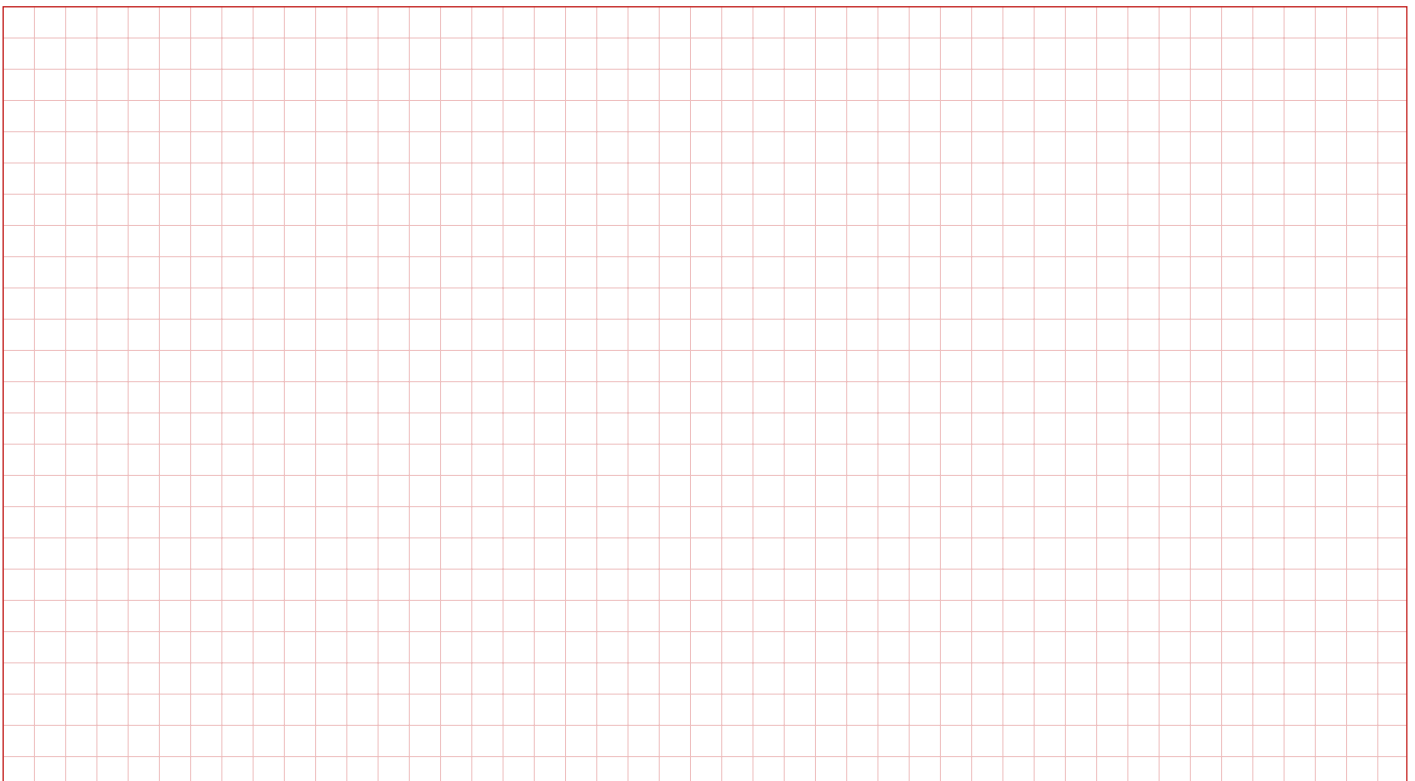


## Welches catch wird verwendet?

```
try { ... }  
catch (E1 e1) {...}  
catch (E2 e2) {...}  
...  
catch (En en) {...}
```

- ▶ **Informeller** Algorithmus wenn Ausnahme vom **Objektyp** E geworfen wird
- ▶ E ist ein E1? **Ja**: Führe **catch**-Block aus
- ▶ **Nein**: E ist ein E2? **Ja**: Führe **catch**-Block aus
- ▶ ...
- ▶ **Nein**: E ist ein En? **Ja**: Führe **catch**-Block aus
- ▶ **Nein**
  - ▶ Führe **finally** aus
  - ▶ Gib Ausnahme **weiter**

## Notizen



## Beispiel: Welches catch wird verwendet?

```
try { ... }  
catch (InputMismatchException exception){ 1 }  
catch (NoSuchElementException exception){ 2 }  
catch (IllegalStateException exception){ 3 }
```

### ► [InputMismatchException](#)

1. InputMismatchE ist InputMismatchE: **Ja**

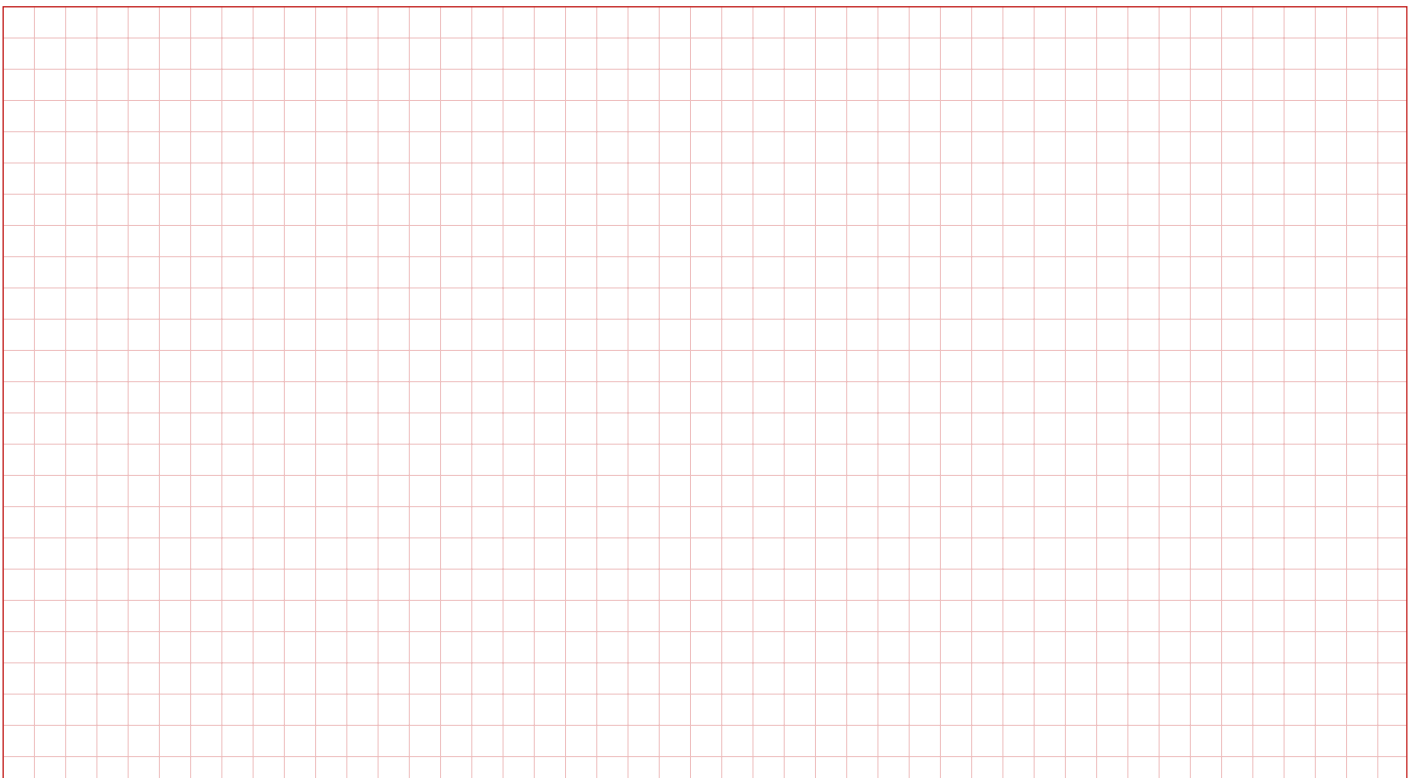
### ► [NoSuchElementException](#)

1. NoSuchElementException ist InputMismatchE: **Nein**
2. NoSuchElementException ist NoSuchElementException: **Ja**

### ► [IllegalStateException](#).

1. IllegalStateException ist InputMismatchE: **Nein**
2. IllegalStateException ist NoSuchElementException: **Nein**
3. IllegalStateException ist IllegalStateException: **Ja**

## Notizen

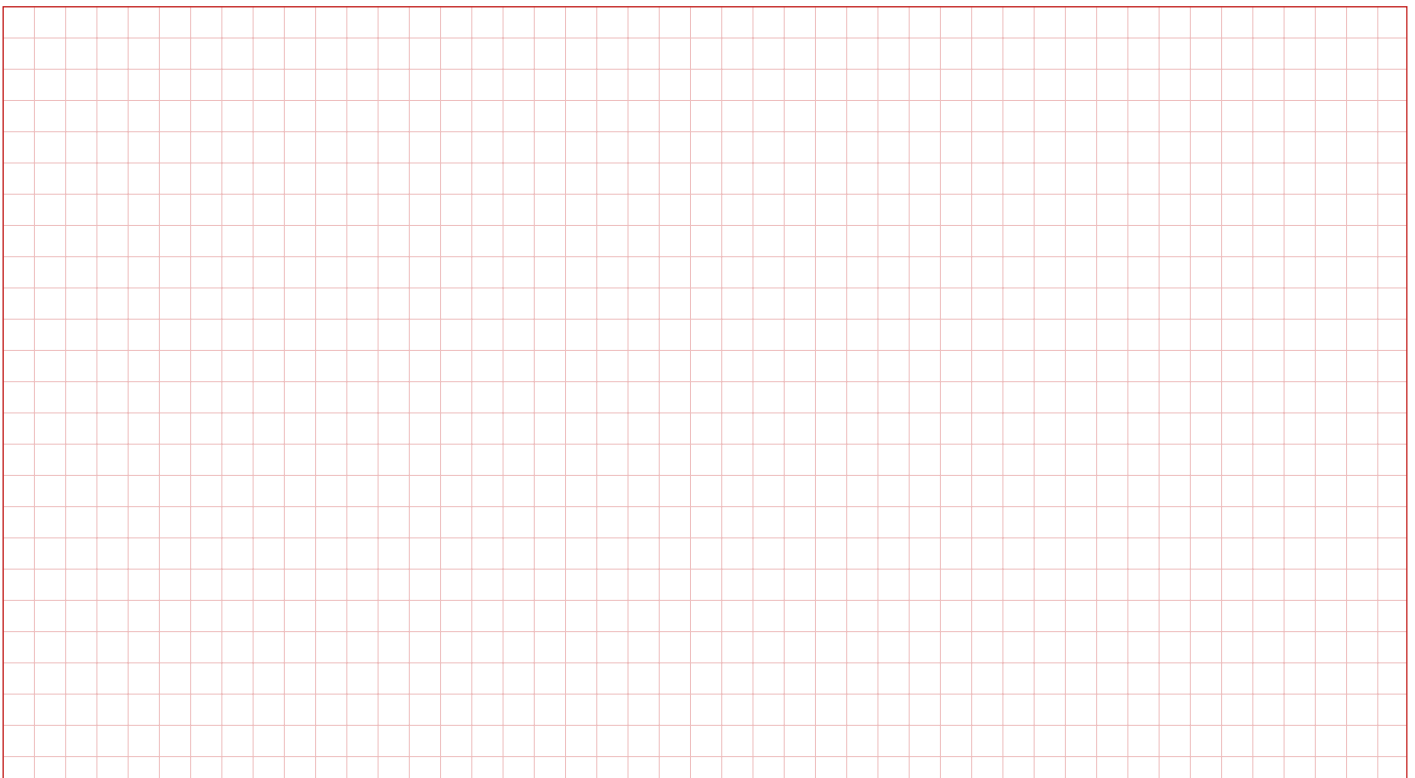


## Beispiel: Welches catch wird verwendet?

```
try { ... }  
catch (NoSuchElementException exception){ 1 }  
catch (InputMismatchException exception){ 2 } FEHLER  
catch (IllegalStateException exception){ 3 }
```

- ▶ InputMismatchE und NoSuchElementException wurden **vertauscht**
- ▶ **Zur Erinnerung:** InputMismatchE ist eine NoSuchElementException
- ▶ ☒ **InputMismatchException**
  1. InputMismatchE ist NoSuchElementException: **Ja**
- ▶ ☒ **NoSuchElementException**
  1. NoSuchElementException ist NoSuchElementException: **Ja**
- ▶ **Beobachtung:**
  - ▶ InputMismatchE wird **immer** in **catch**-Block 1 behandelt
  - ▶ **catch**-Block 2 wird **nie** ausgeführt
- ▶ Java-Compiler **merkt das** und gibt **Compiler-Fehler**: „Unreachable catch block“

## Notizen

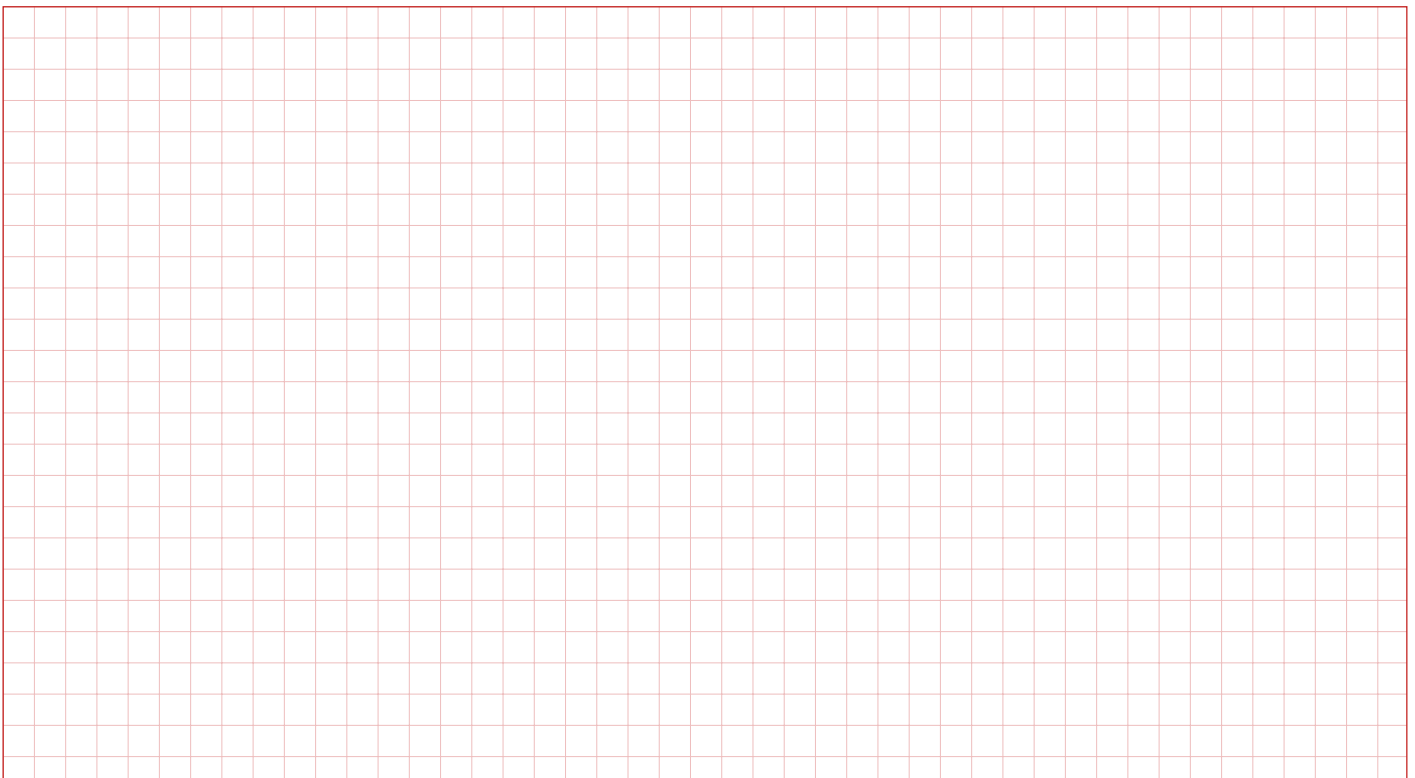


## Beispiel: Welches catch wird verwendet?

```
try { ... }  
catch (Throwable exception){ }
```

- ▶ Jede Ausnahme leitet von [Throwable](#) ab (später mehr)
- ▶ Damit: Jeder Ausnahmetyp ist ein [Throwable](#)
- ▶ `catch (Throwable)` fängt somit jede Ausnahme
- ▶ Aber: Ausnahmebehandlung sollte so spezifisch wie möglich sein
- ▶ Obiger Code nur für
  - ▶ Debugging
  - ▶ Tests
  - ▶ „Wegwerfprogramme“

## Notizen



## Allgemeine Regel

```
try { ... }  
catch (E1 e1) {...}  
catch (E2 e2) {...}  
...  
catch (En en) {...}
```

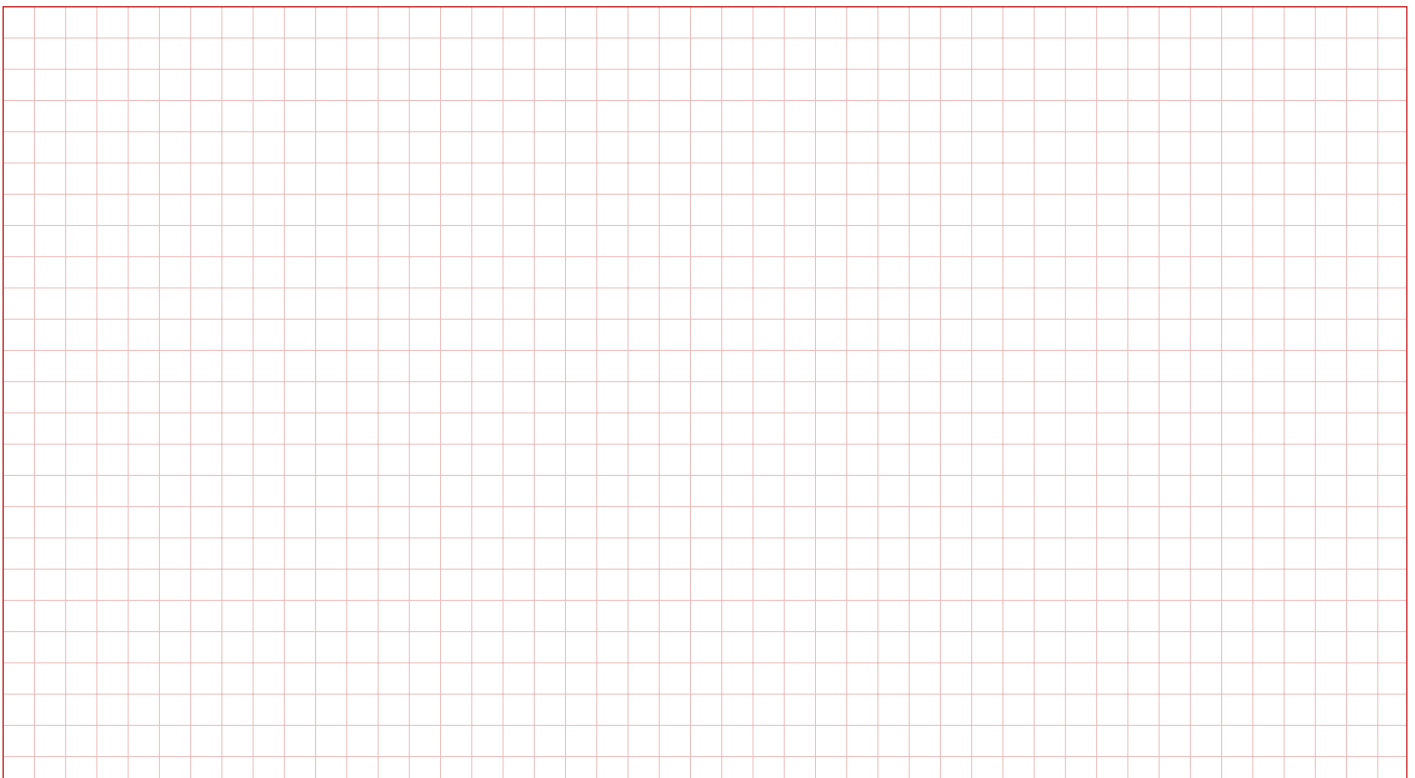
- ▶ **Allgemeine Regel:** Für  $i < j$  darf **nicht** gelten
  - ▶  $E_j$  ist ein  $E_i$
  - ▶ **Sonst:** **catch**-Block von  $E_j$  wird **nie** ausgeführt

```
try { ... }  
catch (E1 | E2 | ... | En e)
```

- ▶ **Regel:** Für  $i \neq j$  darf **nicht** gelten
  - ▶  $E_j$  ist ein  $E_i$  (oder **umgekehrt**)
  - ▶ **Sonst:** Redundanz
- ▶ **Beispiel**

```
catch (InputMismatchE | NoSuchElementException e) FEHLER
```

## Notizen



# Inhalt

## Ausnahmen behandeln

Auffangen mehrerer Ausnahmetypen

Multi-catch

Ausnutzen der Hierarchie

Zusammenfassung

## Notizen

A large rectangular area filled with a fine grid of light gray lines, intended for taking handwritten notes. The grid consists of approximately 30 columns and 40 rows of small squares.



# Zusammenfassung

## ► Herangehensweise

- Welche **Ausnahmen** sollen **unterschiedlich** behandelt werden?

```
catch (...) { ... }  
catch (...) { ... }  
...  
catch (...) { ... }
```

- Welche Ausnahmen sollen **gleich** behandelt werden?
- Für **gleich behandelte Ausnahmen** seien E1, E2, ..., En die Klassen
  - Sortiere durch „ist ein“-Beziehung abgedeckte **Klassen** aus
  - Leiten die Klassen von **gemeinsamer** Klasse BaseE ab,...
  - so dass **nicht zu viel** gefangen wird? (Bsp.: [Throwable](#))

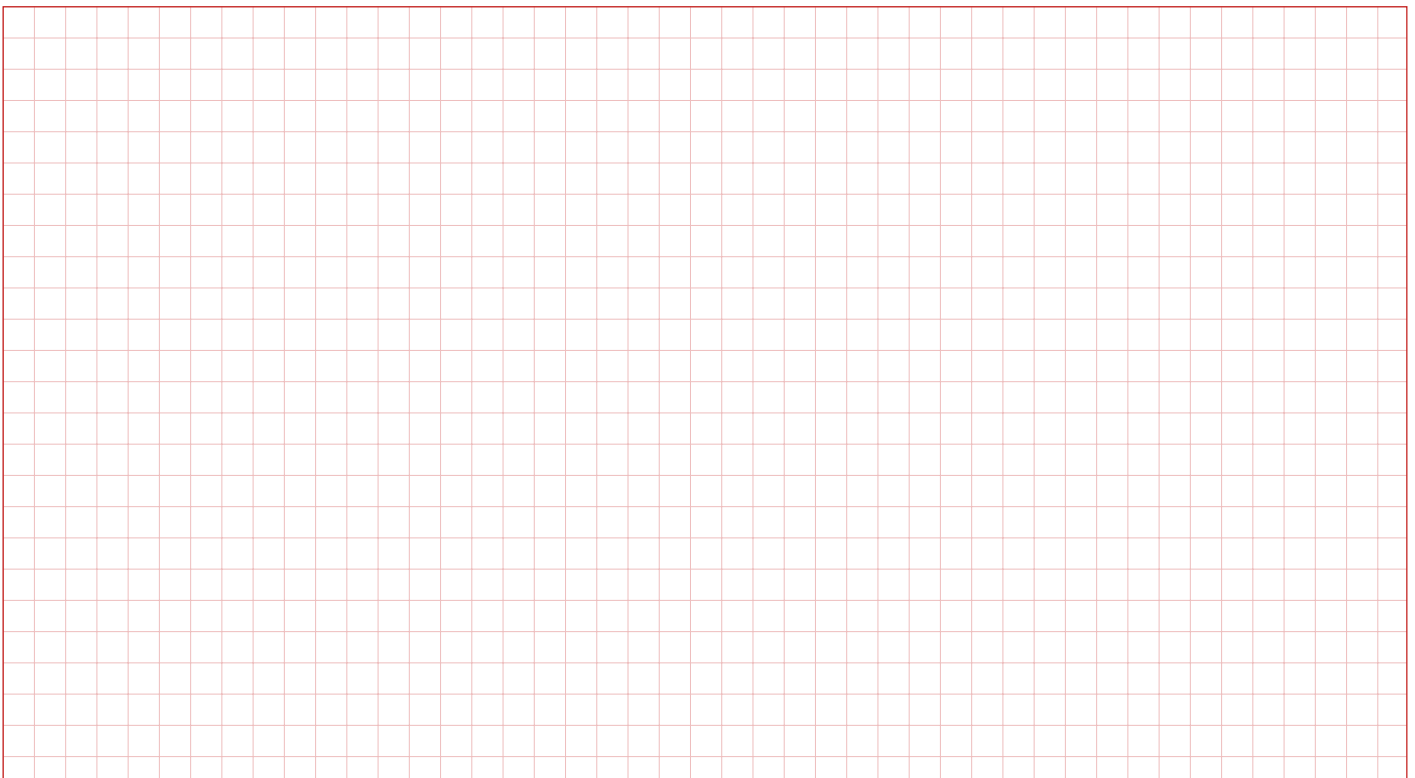
```
catch (BaseE e)
```

- Wenn **nicht**

```
catch (E1 | E2 | ... | En e)
```

- Oder **Mischung** aus beidem

## Notizen



# Inhalt

## Ausnahmen behandeln

## Geprüfte und ungeprüfte Ausnahmen

## Ungeprüfte Ausnahmen

## Geprüfte Ausnahmen

## Zusammenfassung

# Notizen

# Geprüfte und ungeprüfte Ausnahmen

- ▶ Beispiel bisher: Ausnahmen **konnten**, aber **mussten nicht** behandelt werden

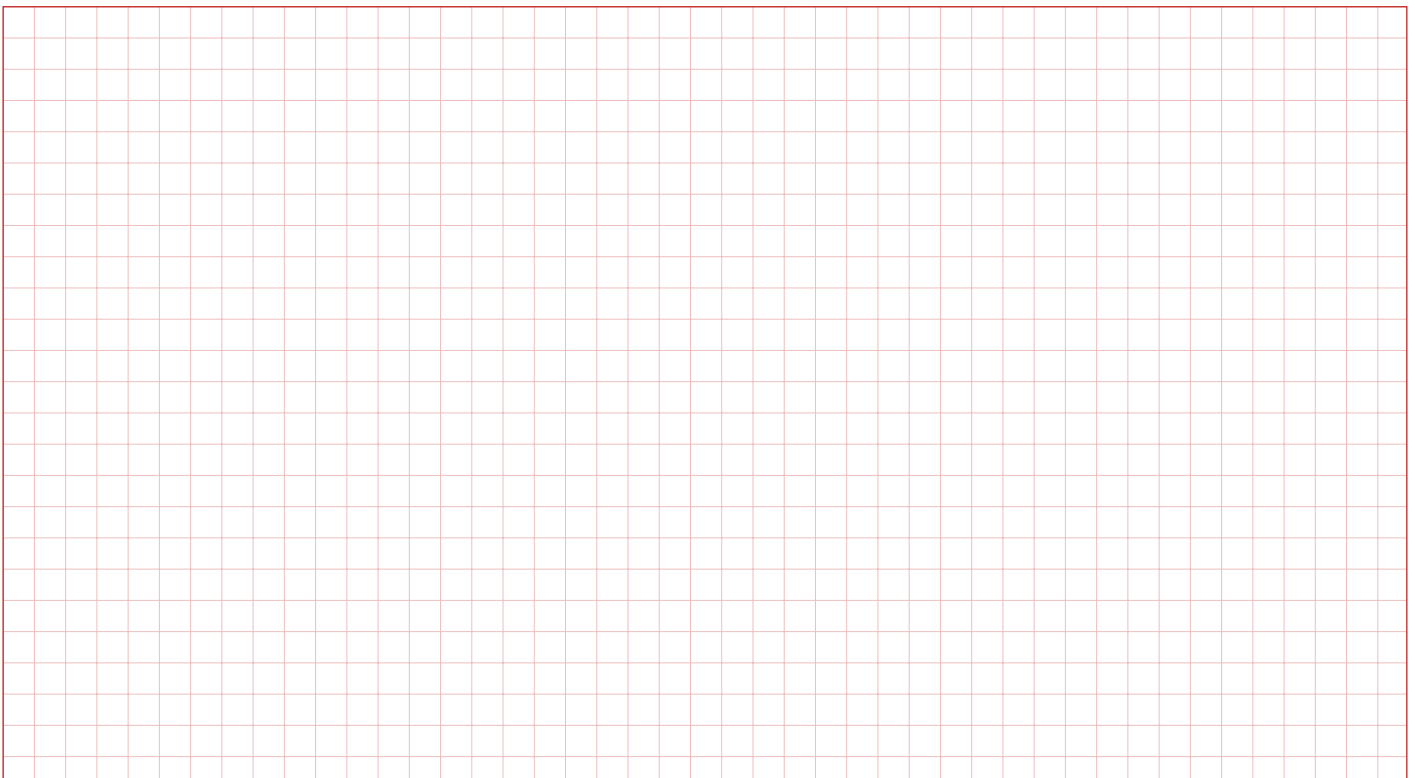
```
scanner.readInt(); // auch ohne try-catch gültig
```

- ▶ Manche Ausnahmen **müssen** behandelt werden

```
File.createTempFile("java1-", "txt"); FEHLER
```

- ▶ Erstellt temporäre Datei
- ▶ Wirft [IOException](#) bei Fehler
- ▶ Fehler: „IOException must be caught or declared to be thrown“
- ▶ Was ist der **Unterschied** zwischen [IOException](#) und [IllegalStateException](#)?
  - ▶ [IOException](#) ist **geprüfte Ausnahme** („checked exception“)
  - ▶ [IllegalStateException](#) ist **ungeprüfte Ausnahme** („unchecked exception“)

## Notizen



# Inhalt

## Ausnahmen behandeln

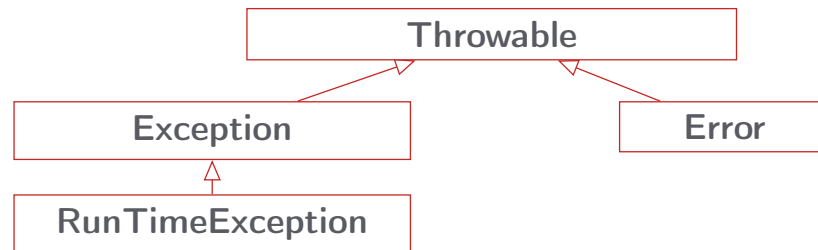
## Geprüfte und ungeprüfte Ausnahmen

## Ungeprüfte Ausnahmen

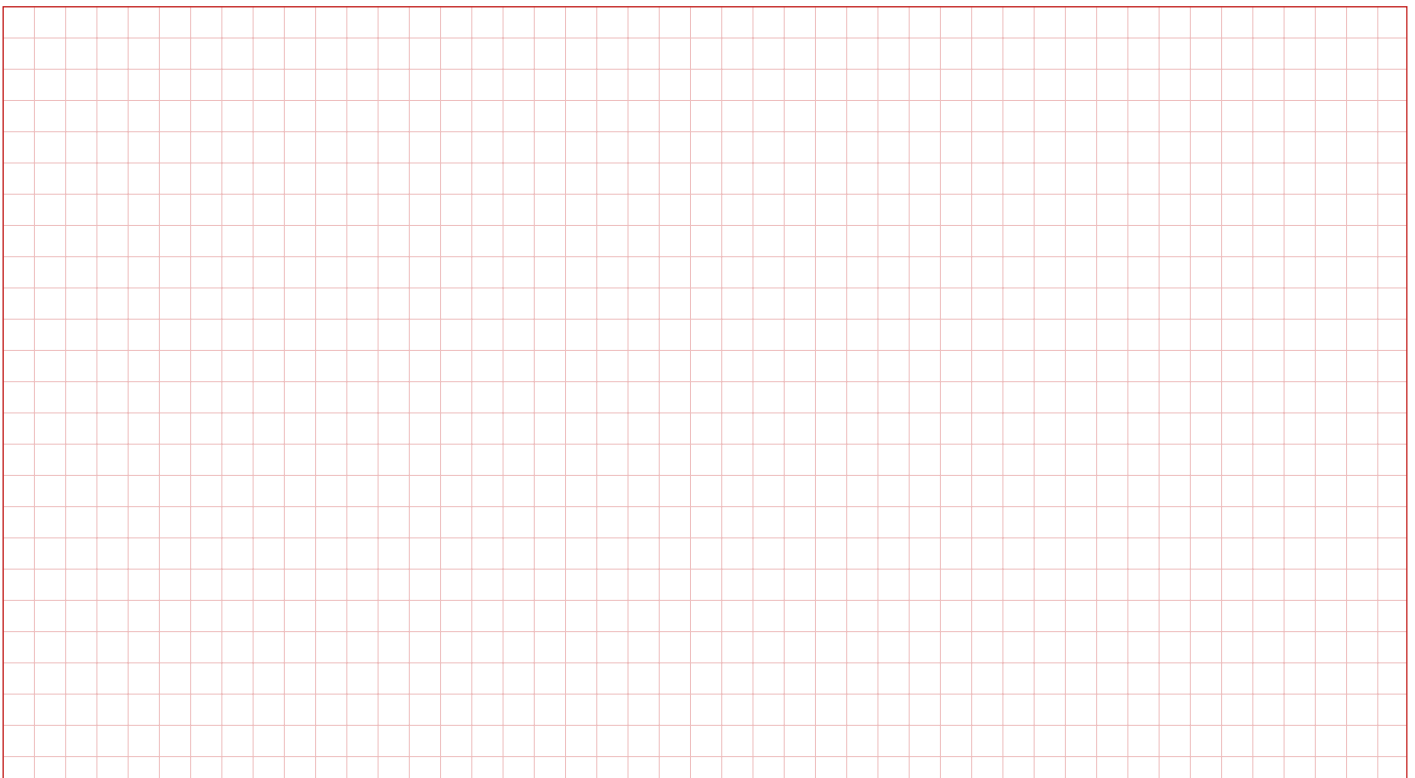
## Notizen

# Geprüft und ungeprüfte Ausnahmen

- ▶ Warum gibt es diese **Unterscheidung**?
- ▶ **Annahme**: Alle Ausnahmen wären geprüft
  - ▶ Sehr viele **try-catch**-Blöcke
  - ▶ Oder: Sehr **lange Methoden-Signaturen** (später)
- ▶ Es gibt zwei Arten von **ungeprüften Ausnahmen**
  - ▶ [RuntimeException](#)
  - ▶ [Error](#)



## Notizen



# Ungeprüfte Ausnahmen

## ▶ [RuntimeException](#) s sind ungeprüft

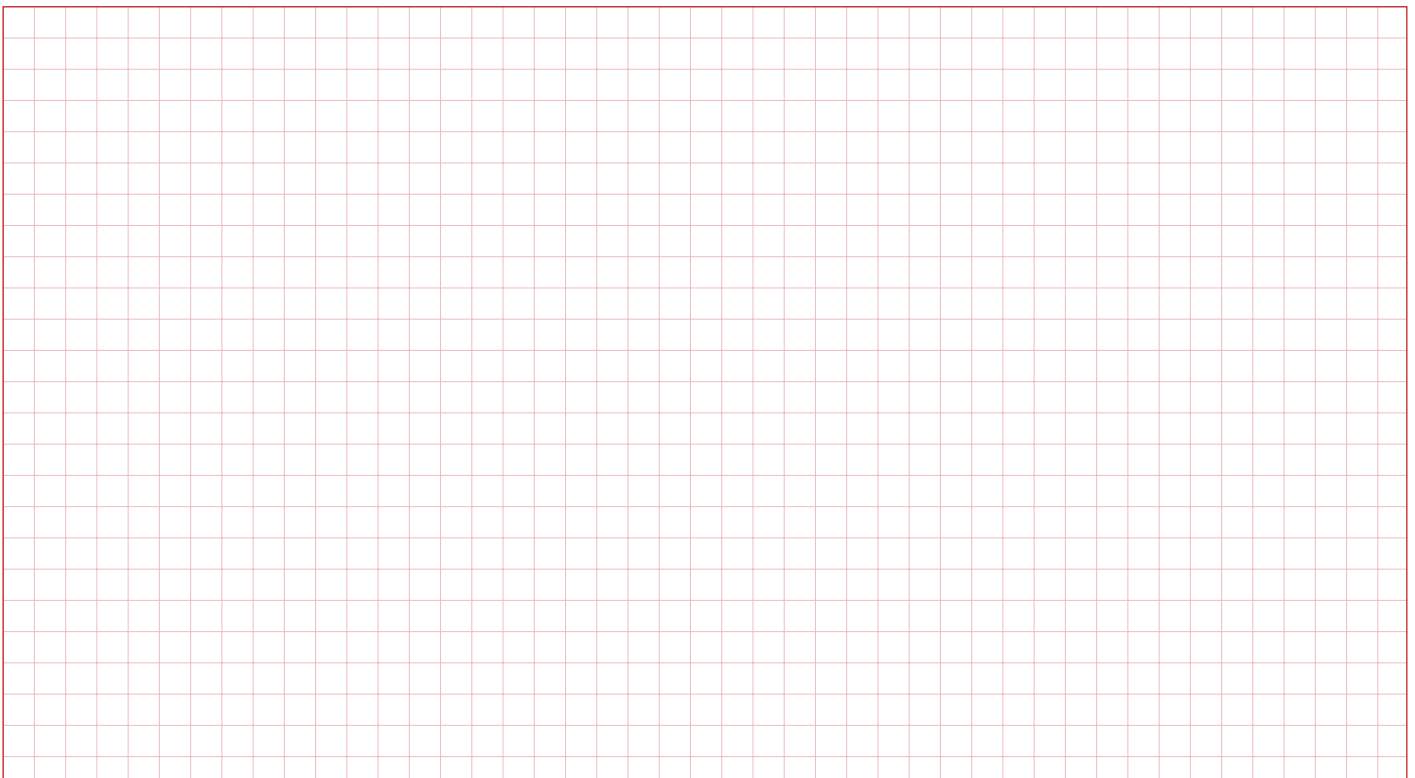
- ▶ Leiten von [RunTimeException](#) ab
- ▶ Fehler, die Programmierer vermeiden könnte
- ▶ Beispiele
  - ▶ [ArrayIndexOutOfBoundsException](#), [NullPointerException](#), [ClassCastException](#), ...
  - ▶ [NoSuchElementException](#)

```
if (scanner.hasNext())  
    i = scanner.nextInt();
```

## ▶ [Error](#) s sind ungeprüft

- ▶ Leiten von [Error](#) ab
- ▶ „Harte Fehler“ der JVM
- ▶ Von Programmierer nicht direkt vermeidbar
- ▶ Beispiele: [OutOfMemoryError](#), [StackOverflowError](#), [IOError](#) (`!= IOException`), ...

## Notizen



# Inhalt

## Ausnahmen behandeln

## Geprüfte und ungeprüfte Ausnahmen

## Geprüfte Ausnahmen

# Notizen

## Geprüfte Ausnahmen

- ▶ Geprüfte Ausnahmen
  - ▶ „Alle anderen“
  - ▶ Nicht-☑ Error und Nicht-☑ RuntimeException
- ▶ Beispiel ☑ IOException

```
File.createTempFile("java1-", "txt"); FEHLER
```

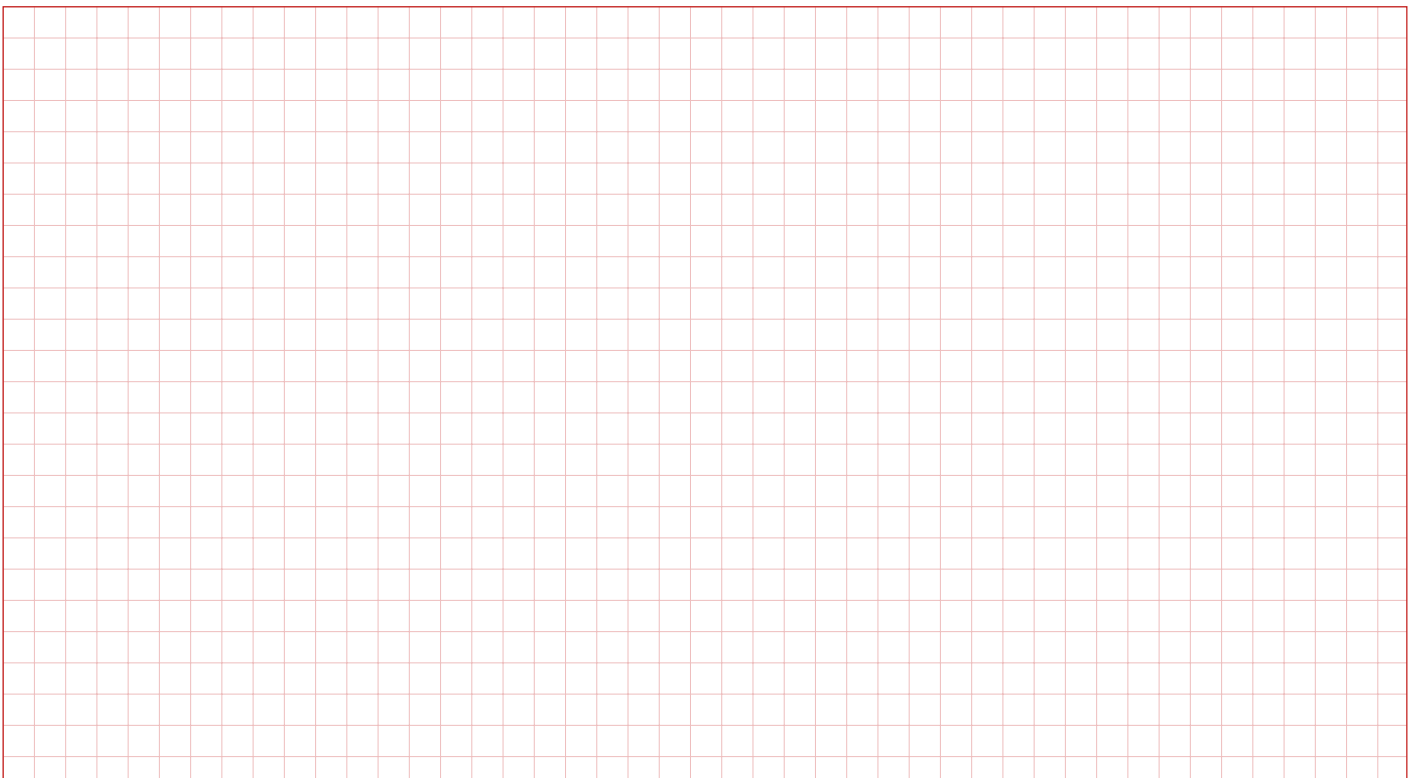
- ▶ Fehler: „IOException must be caught or declared to be thrown“
- ▶ Wir haben also **zwei** Möglichkeiten
  - ▶ Fangen (kennen wir schon)

```
12 try {  
13     File.createTempFile("java1-", "txt");  
14 } catch (IOException exception){  
15     out.println("Kann Datei nicht erstellen");  
16 }
```

☑ CheckedExceptionExamples.java

- ▶ „Must be declared to be thrown“ ???

## Notizen





# Inhalt


## Ausnahmen behandeln

## Geprüfte und ungeprüfte Ausnahmen

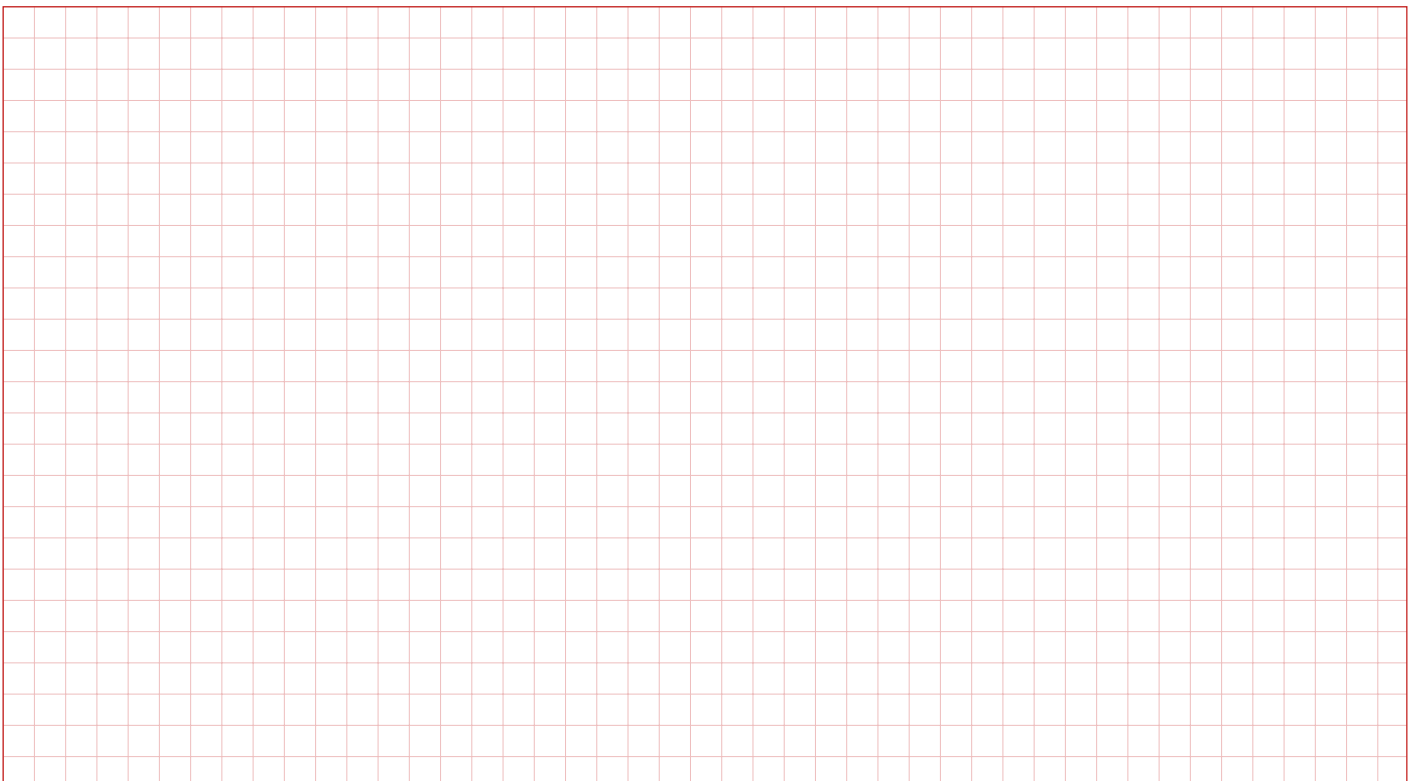
## Zusammenfassung

## Notizen

# Zusammenfassung

- ▶ `RuntimeException`
  - ▶ Müssen **nicht** gefangen werden
  - ▶ Können von Programmierer **vermieden** werden (Prüfung auf **null**, etc.)
  - ▶ Signalisieren **Fehler durch Programmierer**
  - ▶ Programm kann i.d.R. nach Behandlung **nicht fortgeführt** werden
- ▶  **Error**
  - ▶ **harte** Fehler der JVM
  - ▶ **Nicht direkt** vermeidbar
  - ▶ **Keine** sinnvolle **Fortführung** möglich
- ▶ **Geprüfte Ausnahmen**
  - ▶ „Alle anderen“
  - ▶ **Müssen** behandelt werden
  - ▶ Signalisieren zu **erwartende Fehler** (z.B. falsche Benutzereingabe)
  - ▶ Programm kann i.d.R. nach Behandlung **fortgeführt** werden

## Notizen



# Inhalt

## Ausnahmen behandeln

- throws-Deklaration
  - Hinweise zu throws
  - Regeln beim Überschreiben

## Notizen

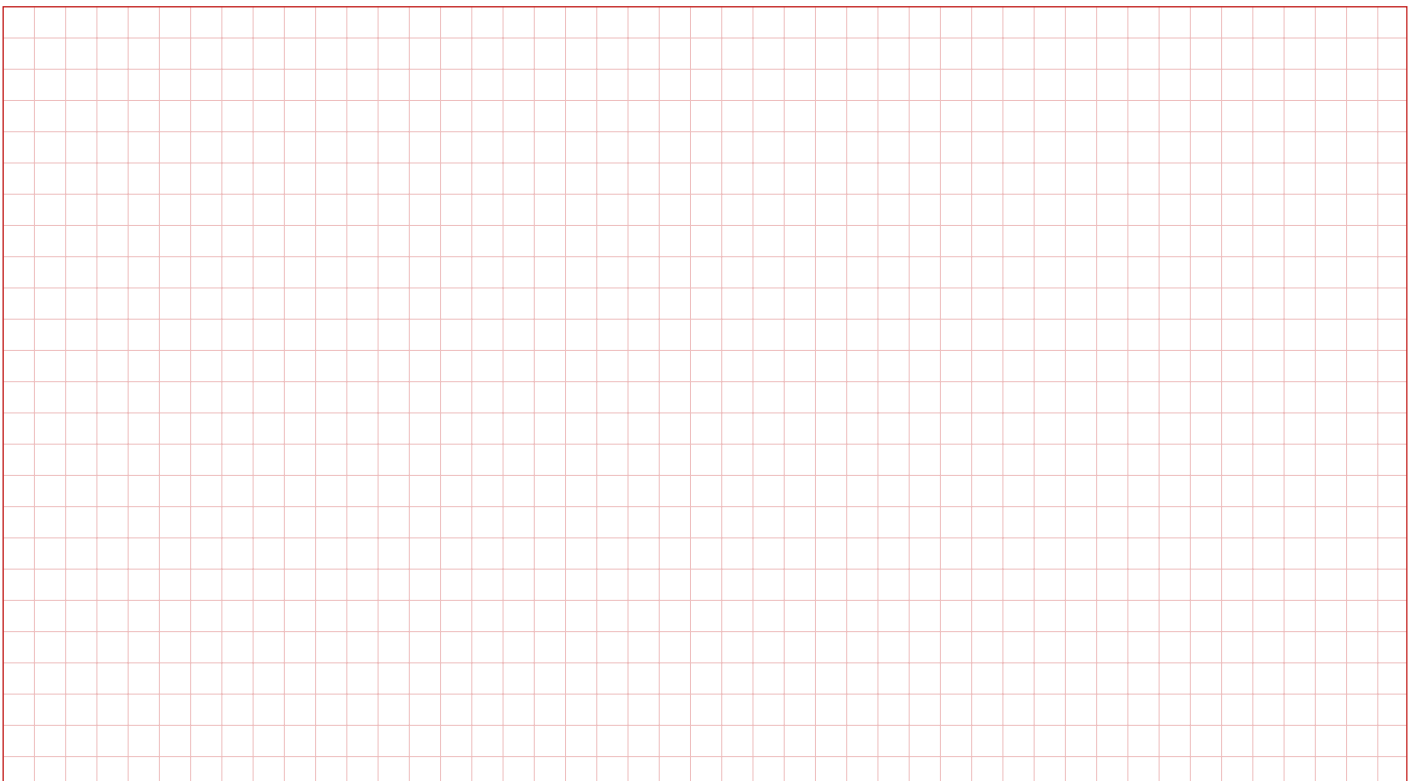
A large rectangular area filled with a fine grid of light gray lines, intended for taking handwritten notes. The grid consists of approximately 30 columns and 40 rows of small squares.

## throws-Deklaration

```
public static void createTempFile(){  
    File.createTempFile("java1-", "txt"); FEHLER  
}
```

- ▶ **Fehler**: „IOException must be caught or declared to be thrown“
- ▶ **Alternative** zu **catch**: Ausnahme **weiterreichen**
- ▶ **Deklaration** von geworfenen Ausnahmen in Methodendeklaration mit **throws**

## Notizen



## throws-Deklaration

```
12 public static File generateTempFile()  
13     throws IOException {  
14     return File.createTempFile("java1-", "txt");  
15 }
```

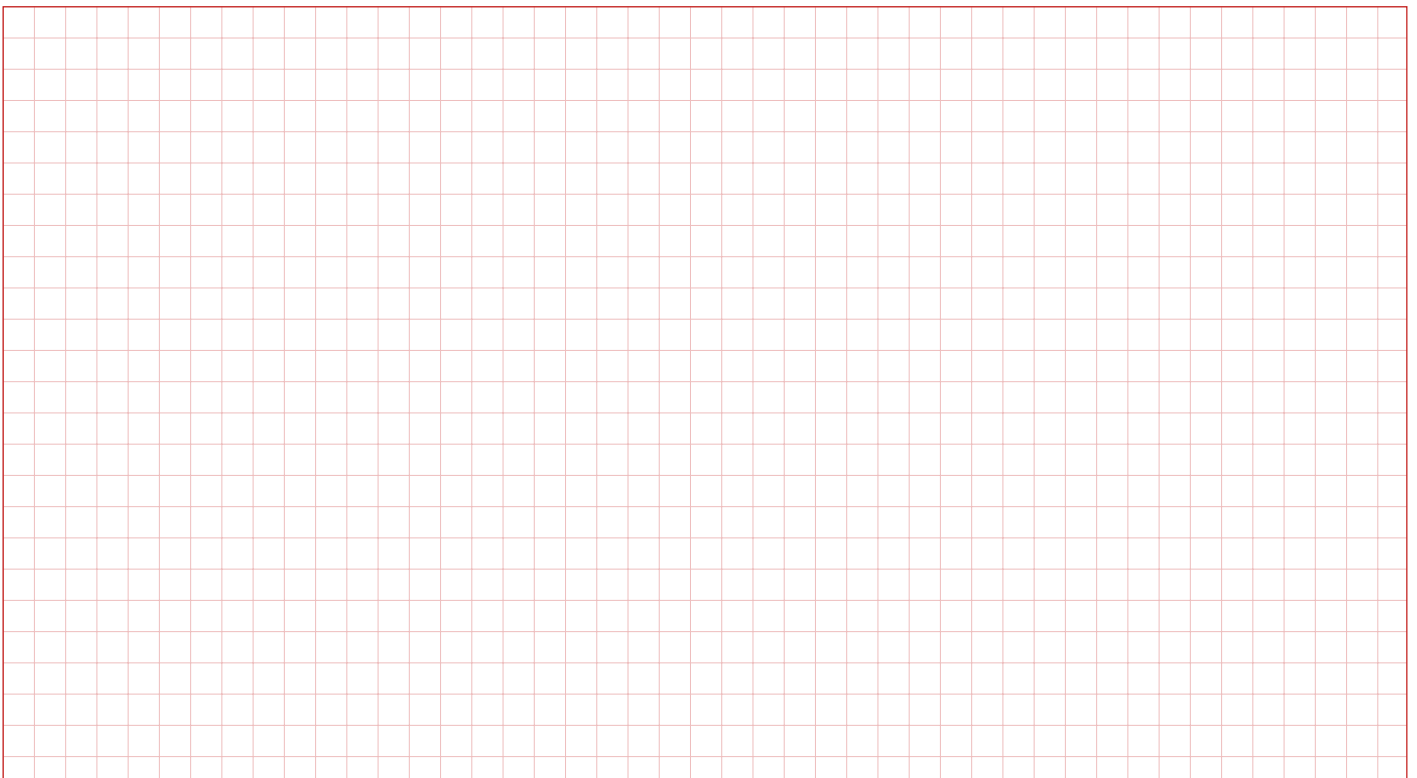
ThrowsExamples.java

- ▶ **Schlüsselwort:** **throws**
- ▶ **Kein Fehler** mehr
- ▶ **Deklariert:** Bei Aufruf dieser Methode **muss** mit [IOException](#) gerechnet werden
- ▶ Fehler muss von **Aufrufer** behandelt werden

```
try{  
    File f = generateTempFile();  
} catch (IOException) { ... }
```

- ▶ **Oder:** Wieder **throws** bei **Aufrufer-Methode**

## Notizen



# Inhalt

## Ausnahmen behandeln

throws-Deklaration

Hinweise zu throws

Regeln beim Überschreiben

## Notizen

A large rectangular area filled with a fine grid of light gray lines, intended for taking handwritten notes. The grid consists of approximately 30 columns and 40 rows of small squares.

## throws-Deklaration

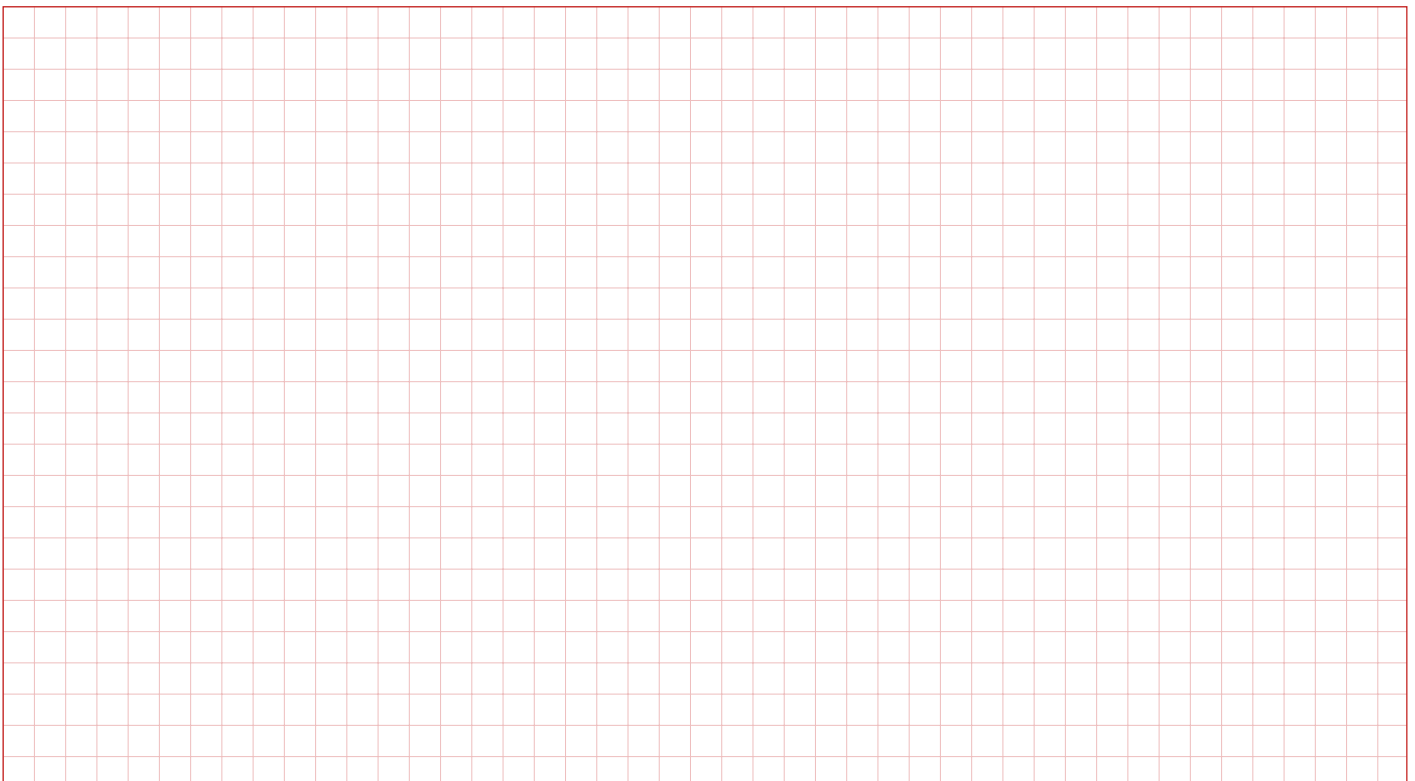
- ▶ Ungeprüfte Ausnahmen
  - ▶ Können mit **throws** weitergereicht werden
  - ▶ Sollten zumindest **dokumentiert** werden (JavaDoc)
- ▶ **JavaDoc** mit @throws Typ Beschreibung

```
/**  
 * ...  
 * @throws IOException if file could not be created  
 */
```

- ▶ **Konstruktoren** können auch geworfene Ausnahmen deklarieren

```
public TempFileManager()  
    throws IOException{  
    File.makeTempFile("java1-", "txt");  
}
```

## Notizen



## throws-Deklaration

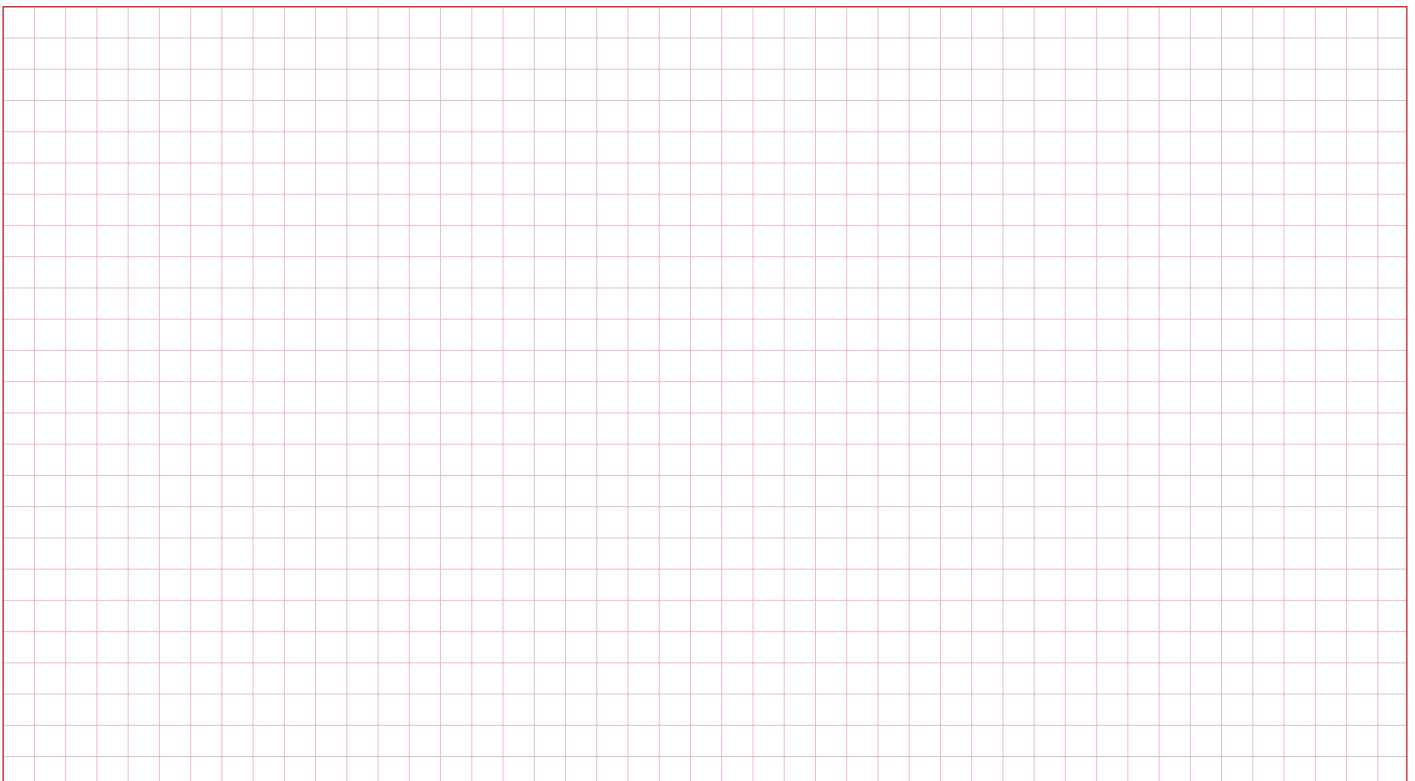
- Auch **mehrere** Ausnahmen sind möglich

```
19 public static int readFirstByte(String path)
20     throws FileNotFoundException, IOException {
21
22     var file = new File(path);
23     var in = new FileInputStream(file); // FileNotFoundException
24     int b = in.read(); // IOException
25     in.close(); // IOException
26     return b;
27 }
28 }
```

ThrowsExamples.java

- Aufrufer muss **jede** (geprüfte) Ausnahme behandeln

## Notizen





# Inhalt

## Ausnahmen behandeln

throws-Deklaration

Hinweise zu throws

Regeln beim Überschreiben

## Notizen

A large rectangular area filled with a light gray grid pattern, intended for taking notes. The grid consists of small squares and covers most of the lower half of the page.

# Regeln beim Überschreiben

```
void createTempFile() throws IOException { }
```

- ▶ Es gilt das **Substitutionsprinzip**
  - ▶ **throws**-Deklaration beibehalten

```
@Override void createTempFile() throws IOException{ }
```

- ▶ Ausnahmen **spezialisieren**

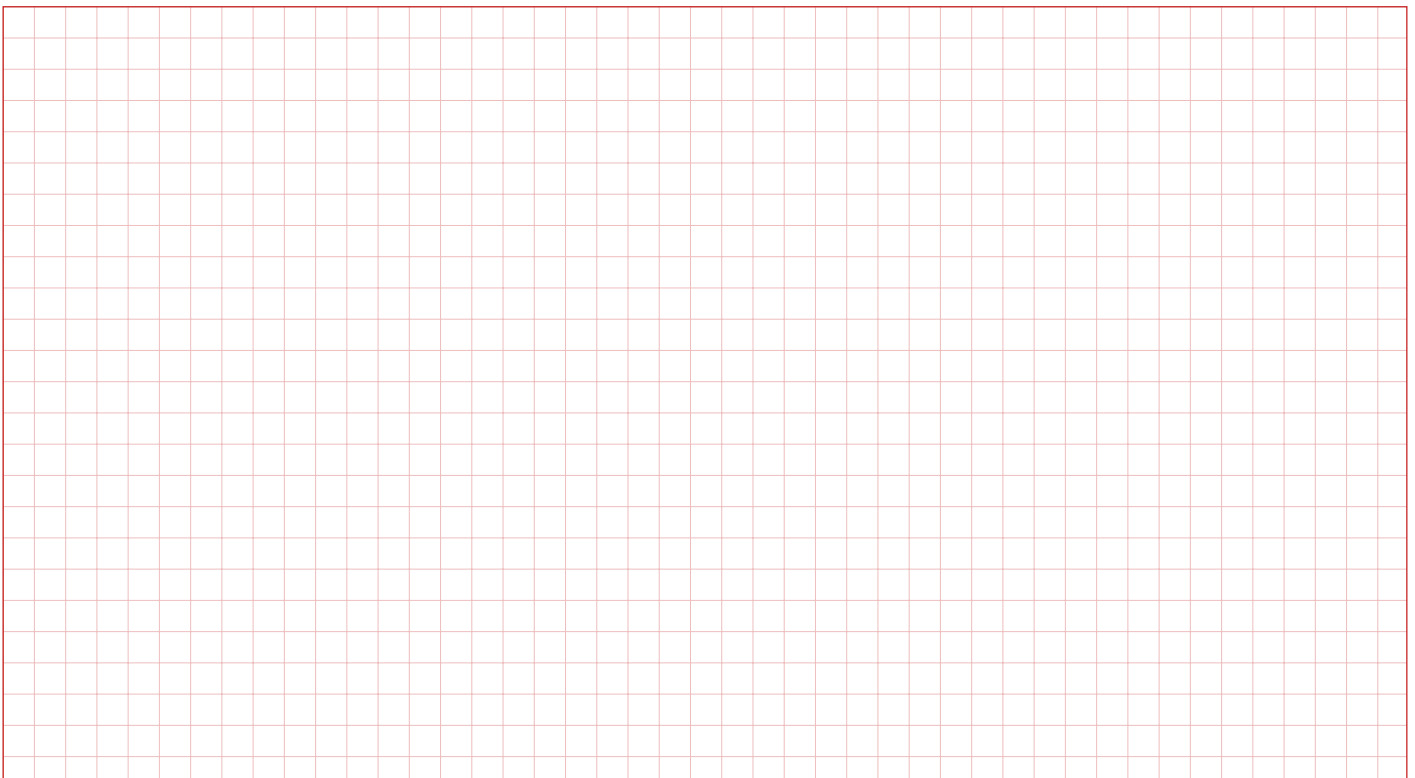
```
@Override void createTempFile() throws FileAlreadyExistsException{ }
```

wobei [FileAlreadyExistsException](#) von [IOException](#) abstammt

- ▶ Ausnahmen **weglassen**

```
@Override void createTempFile() { }
```

## Notizen



## Überschriebene Konstruktoren

```
public TempFileManager(String s) throws IOException{  
    File.makeTempFile(s, "txt");  
}
```

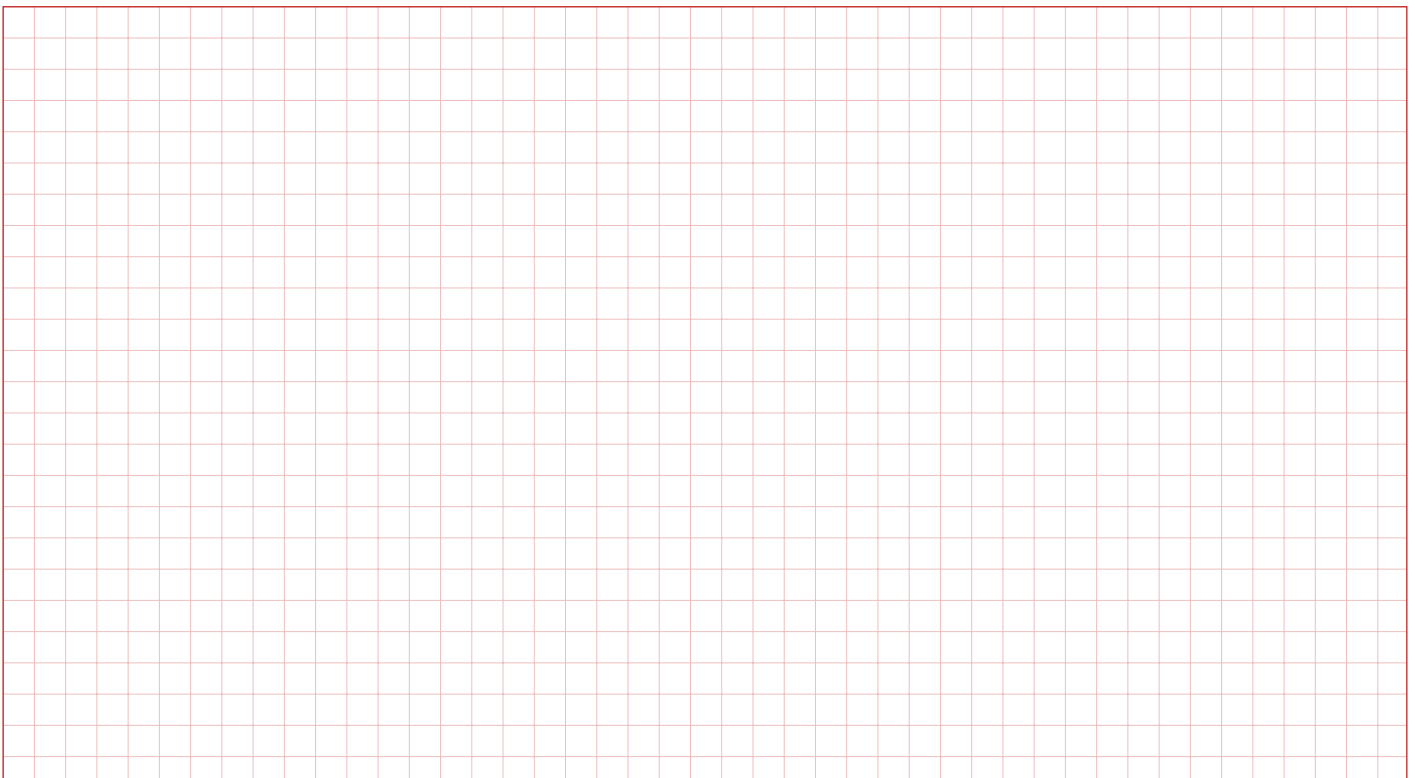
- ▶ Ausnahmen bei Konstruktorenrufe über **this()**, **super()**
- ▶ können **nicht** mit **try-catch** gefangen werden
- ▶ **Grund: this()/super()** müssen **erste Anweisung** sein

```
public TempFileManager(){  
    try {  
        this("java1-"); // FEHLER  
    } catch (IOException e){ }  
}
```

- ▶ Konstruktor **muss geworfene Ausnahme** deklarieren

```
public TempFileManager() throws IOException {  
    this("java1-");  
}
```

## Notizen



# Inhalt

## Ausnahmen behandeln

## Wo und wie Ausnahmen fangen?

# Notizen

## Wo und wie Ausnahmen fangen?

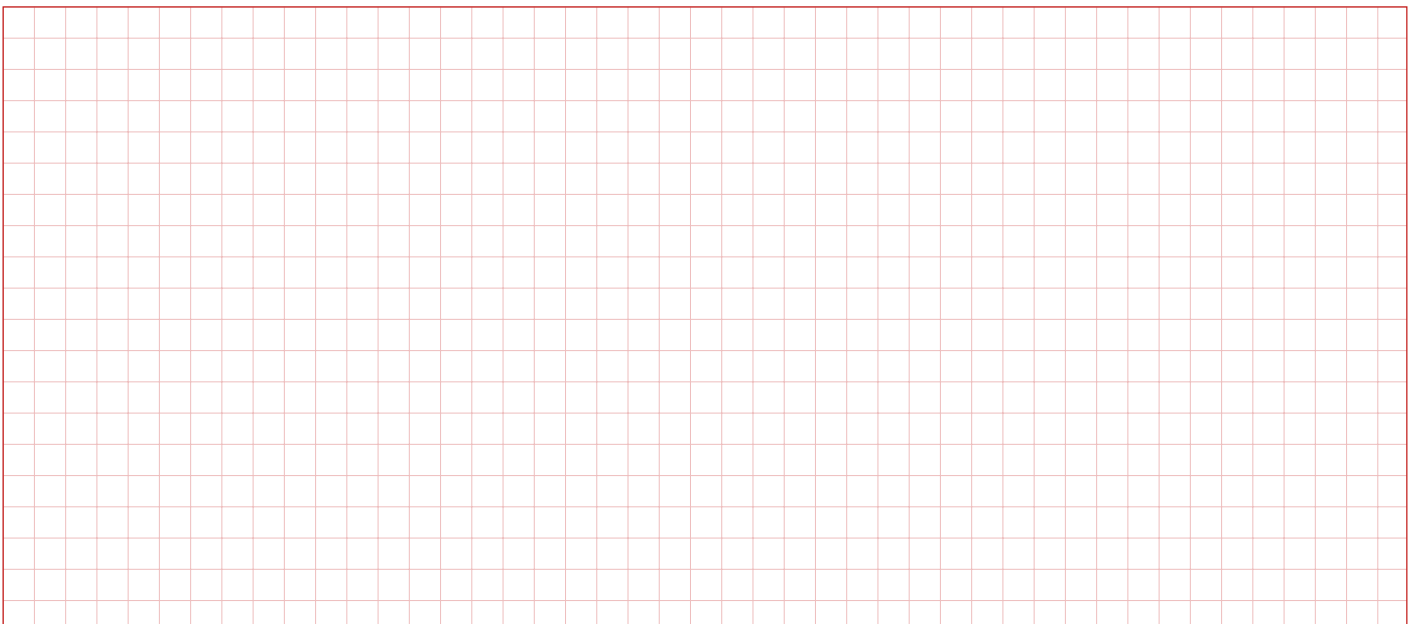
- ▶ Wo fängt man Ausnahmen und **was** macht man dann damit?
- ▶ Das hängt vom **Kontext** ab!
- ▶ Beispiel: [FileNotFoundException](#)

```
String readFile(String path)  
    throws FileNotFoundException
```

- ▶ **Benutzer** gibt Dateipfad an
  - ▶ Behandlung in **UI/CLI**
  - ▶ **Hinweis** an Nutzer
  - ▶ Danach Programm **fortfahren**
- ▶ Öffnen von **Konfigurationsdatei**
  - ▶ Behandlung beim **Laden der Konfiguration**
  - ▶ Evtl. in **Log** schreiben
  - ▶ **Defaultwerte** setzen und fortfahren
- ▶ Öffnen von **wichtiger Resource**
  - ▶ Behandeln beim **Laden der Resource**
  - ▶ Fehler auf UI/CLI
  - ▶ Evtl. Programmabbruch

## Notizen

- UI/CLI heißt auf der Benutzeroberfläche die graphisch sein kann (GUI) oder auf der Kommandozeile („command-line interface“; CLI)

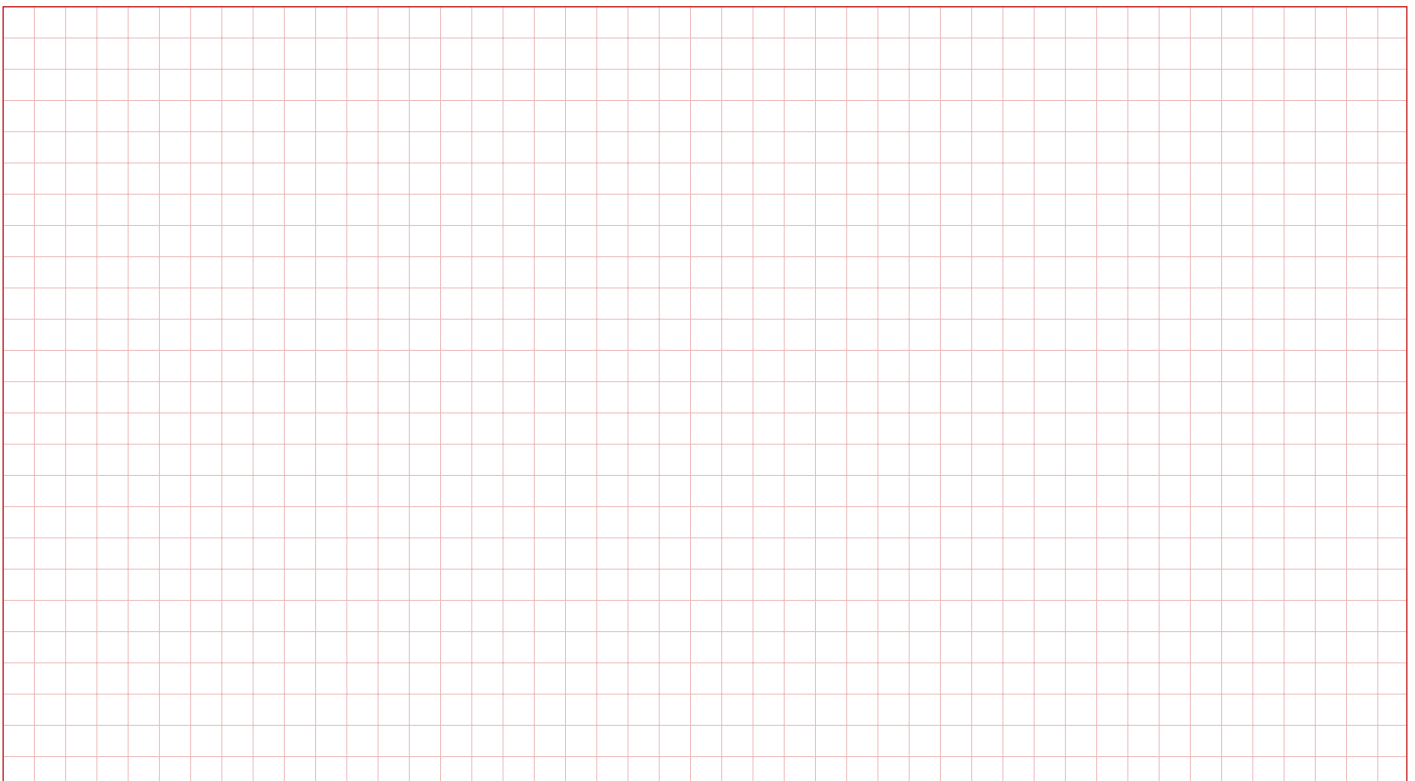


## Wo fängt man Ausnahmen

```
try {  
    String content = readFile(path);  
} catch (FileNotFoundException e){  
    showDialog("Datei existiert nicht! Bitte korrigieren!");  
}
```

- ▶ Als **Regel** gilt: Ausnahme dann behandeln wenn
  - ▶ Kontextinformationen **angemessene Behandlung** erlaubt
- ▶ **Sonst**: Weiterreichen
- ▶ **Praxis**
  - ▶ Leichter **gesagt** als **getan**
  - ▶ Oft eigenes **Modul/Framework/Konzept** zur Ausnahmebehandlung

## Notizen



# Inhalt

## Hierarchie der Ausnahmen

## Ausnahmen des JDK

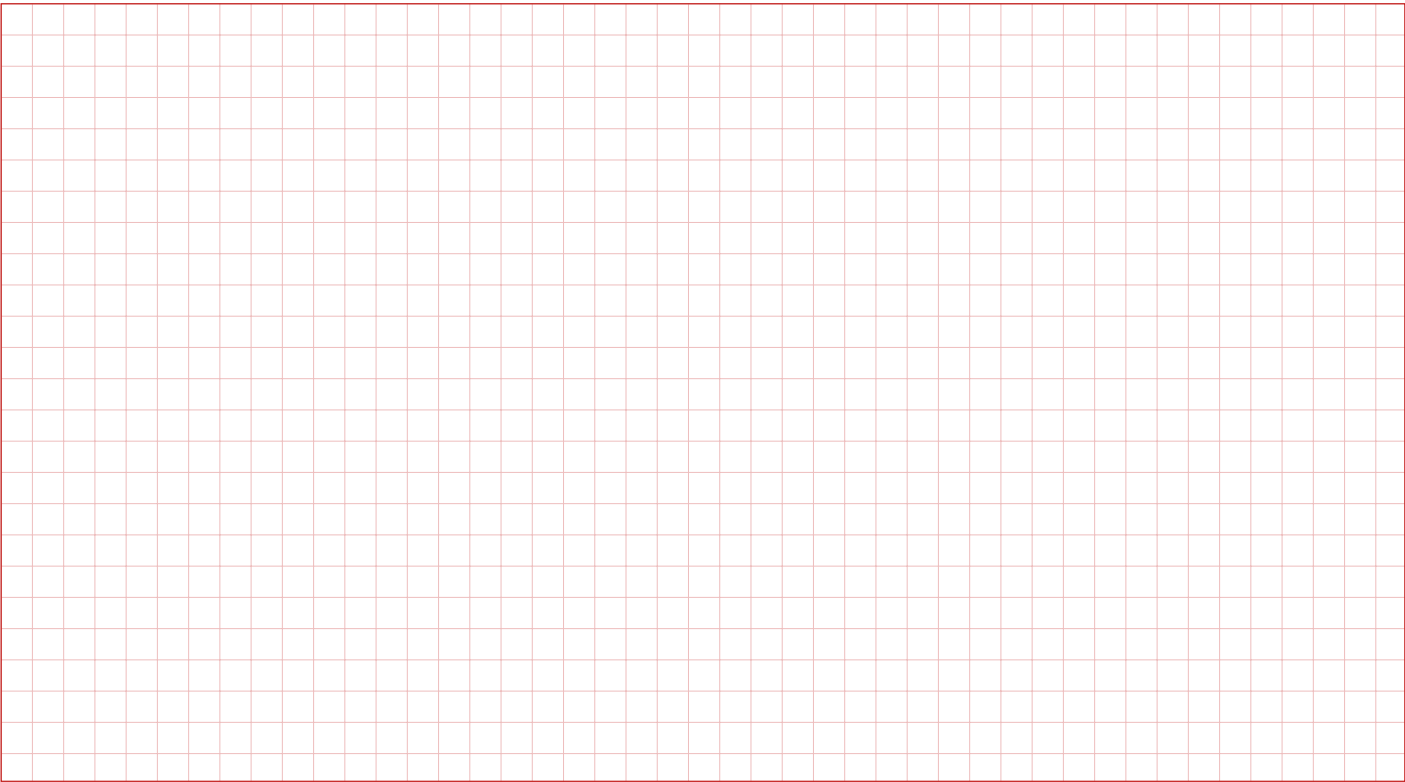
# Notizen

# Inhalt

## Hierarchie der Ausnahmen

- Ausnahmen des JDK
  - Klassenhierarchie des JDK
  - Die Klasse Throwable

## Notizen





# Inhalt

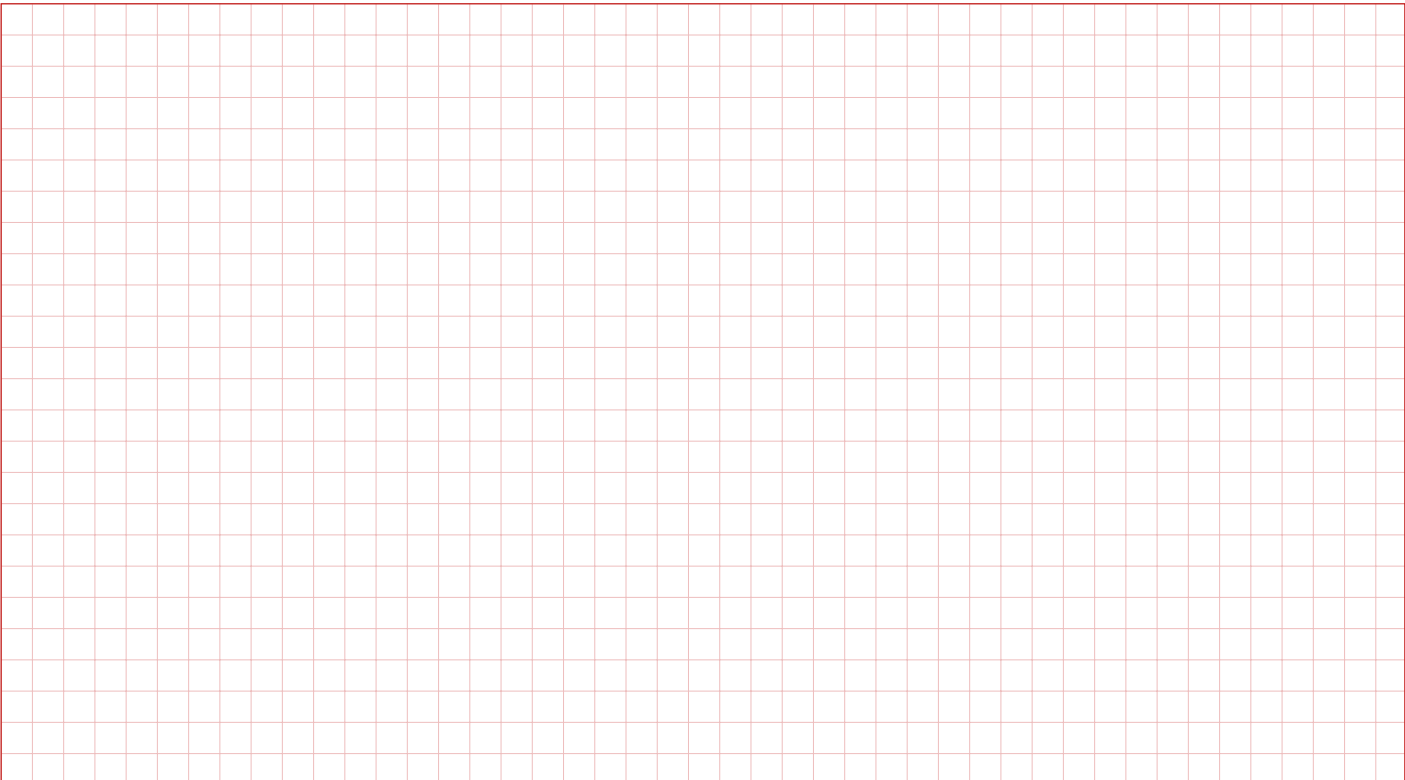
## Hierarchie der Ausnahmen

Ausnahmen des JDK

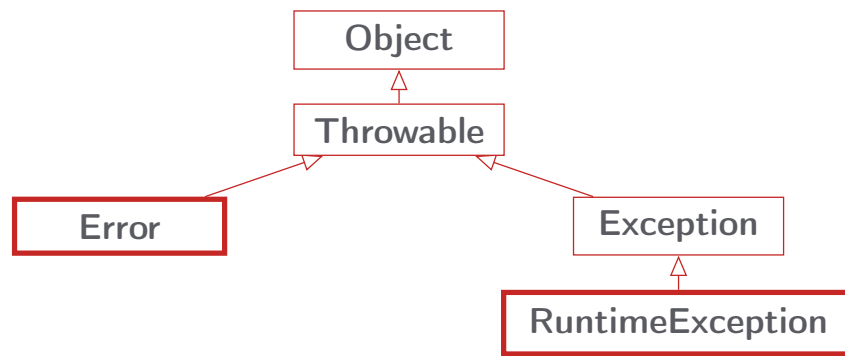
Klassenhierarchie des JDK

Die Klasse Throwable

## Notizen

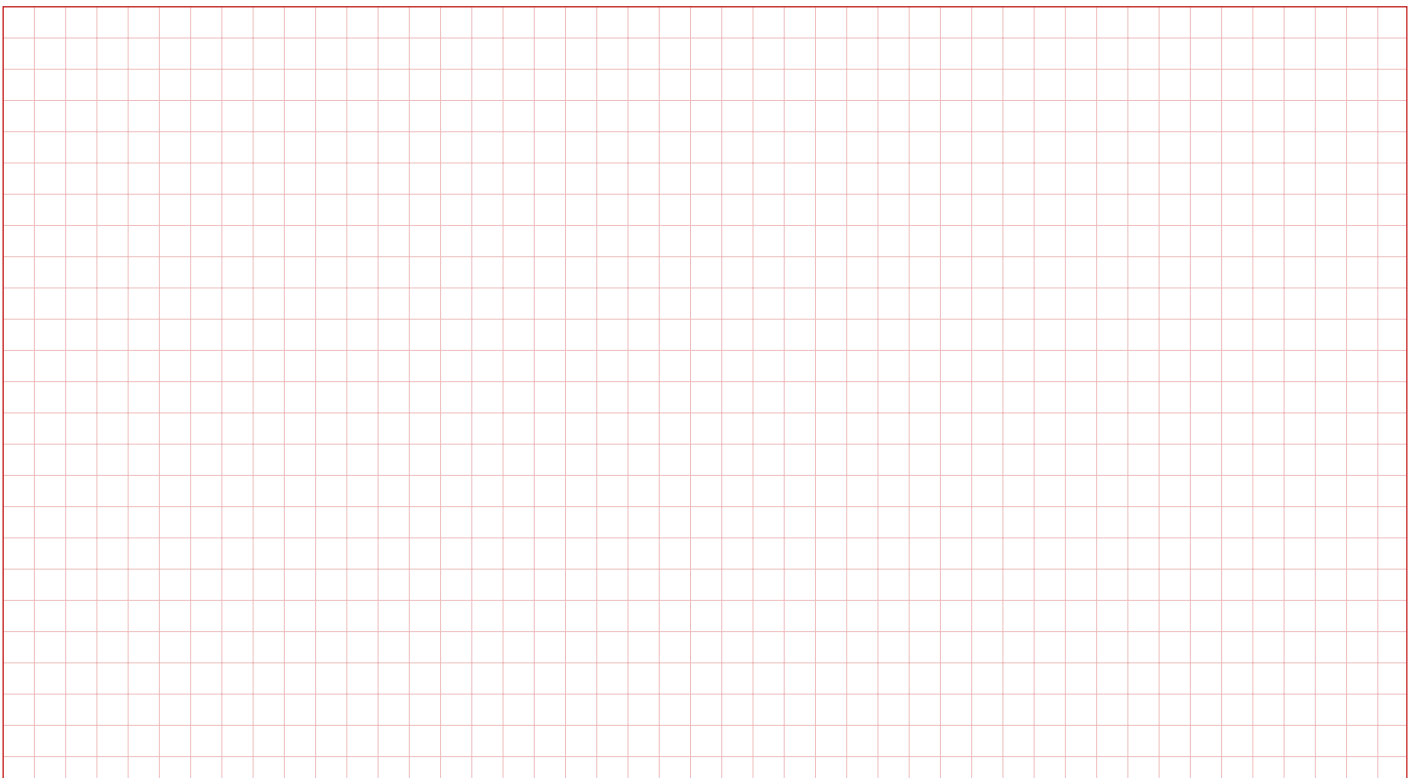


## Ausnahmen des JDK

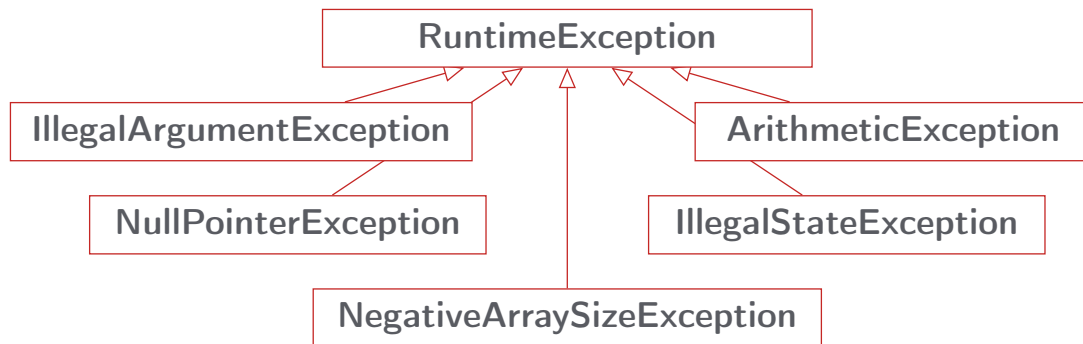


- ▶ Basis-Ausnahmeklassen des JDK
- ▶ **Ungeprüfte Ausnahmen** (dicker Rahmen): [☞ Error](#), RuntimeException und alles was davon ableitet
- ▶ **Geprüfte Ausnahmen**: Alles was...
  - ▶ [☞ Throwable](#) ableitet...
  - ▶ und **nicht** von [☞ Error](#) oder RuntimeException ableitet

## Notizen

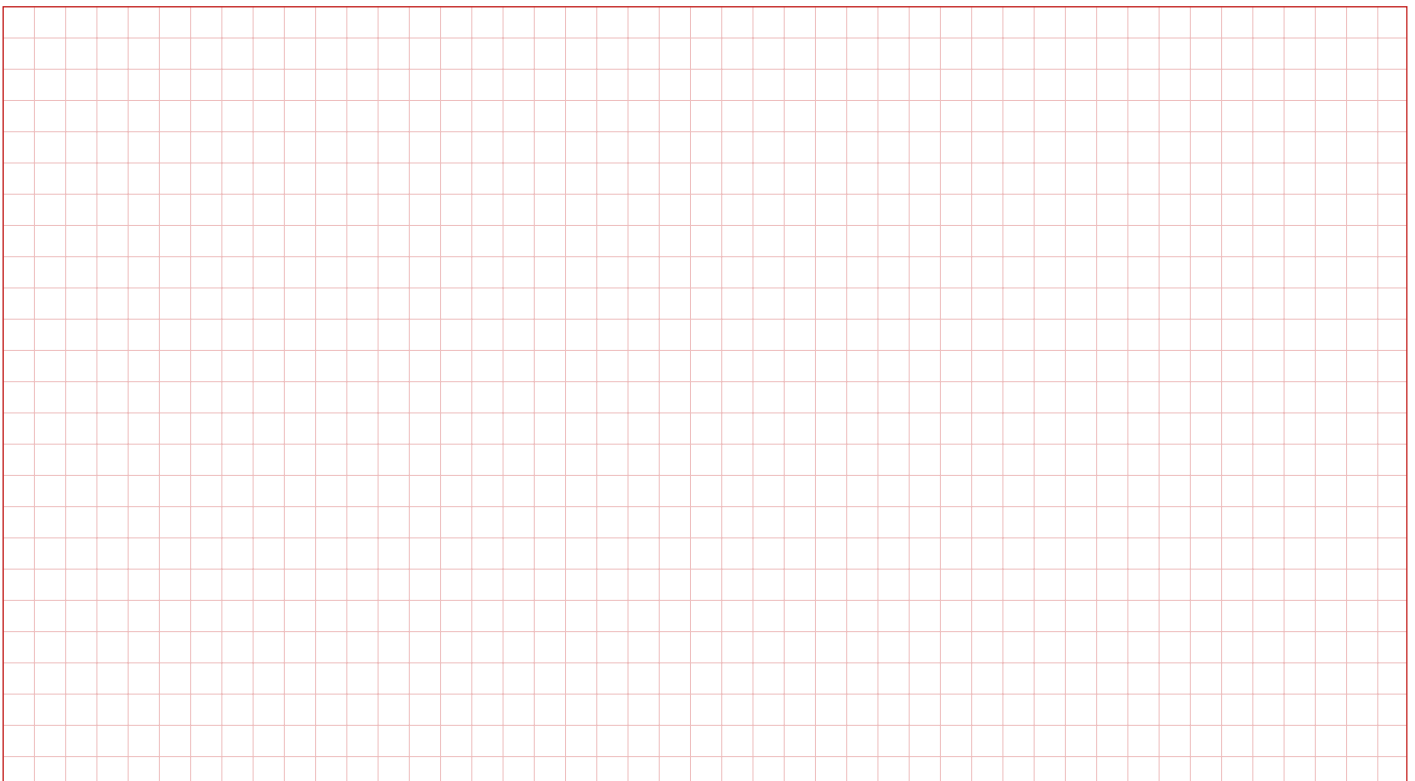


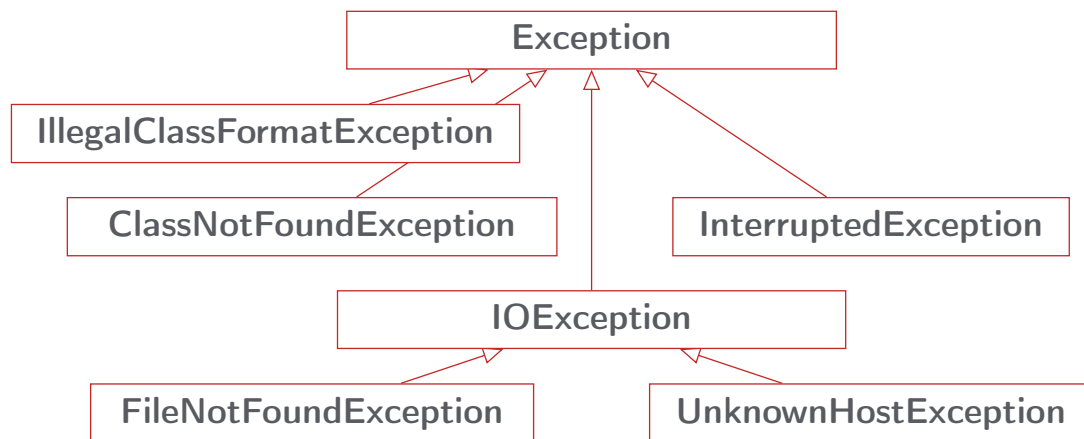
## RunTimeExceptions



- ▶ Nur ein **kleiner Ausschnitt!**
- ▶ Ausnahmen, die i.d.R. **vermieden werden können**

## Notizen





- ▶ Nur ein **kleiner Ausschnitt!**
- ▶ Ausnahmen durch **fehlerhafte Nutzereingabe**
- ▶ Meist **nicht Schuld** des Programmierers
- ▶ **Müssen** behandelt werden

## Notizen

# Inhalt

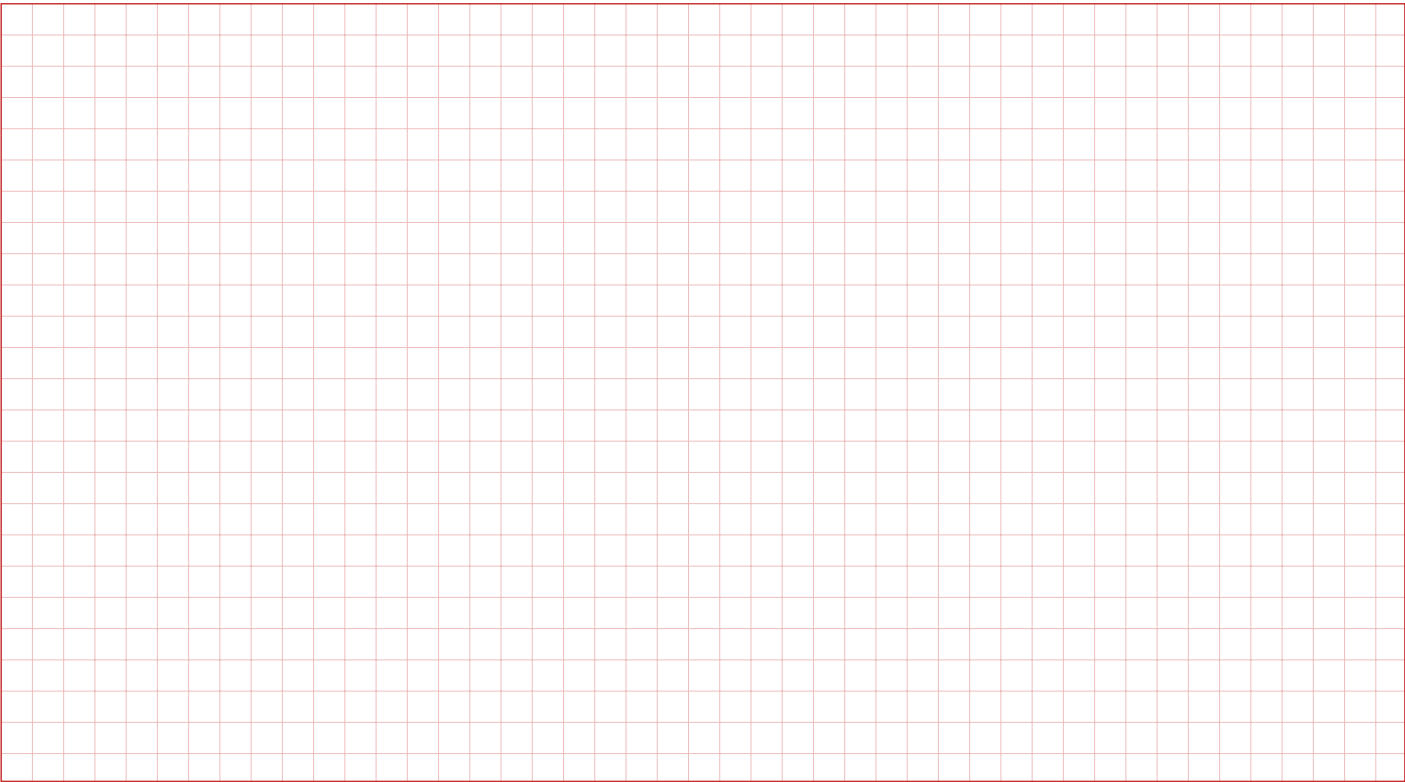
## Hierarchie der Ausnahmen

Ausnahmen des JDK

Klassenhierarchie des JDK

Die Klasse Throwable

## Notizen



## Die Klasse throwable

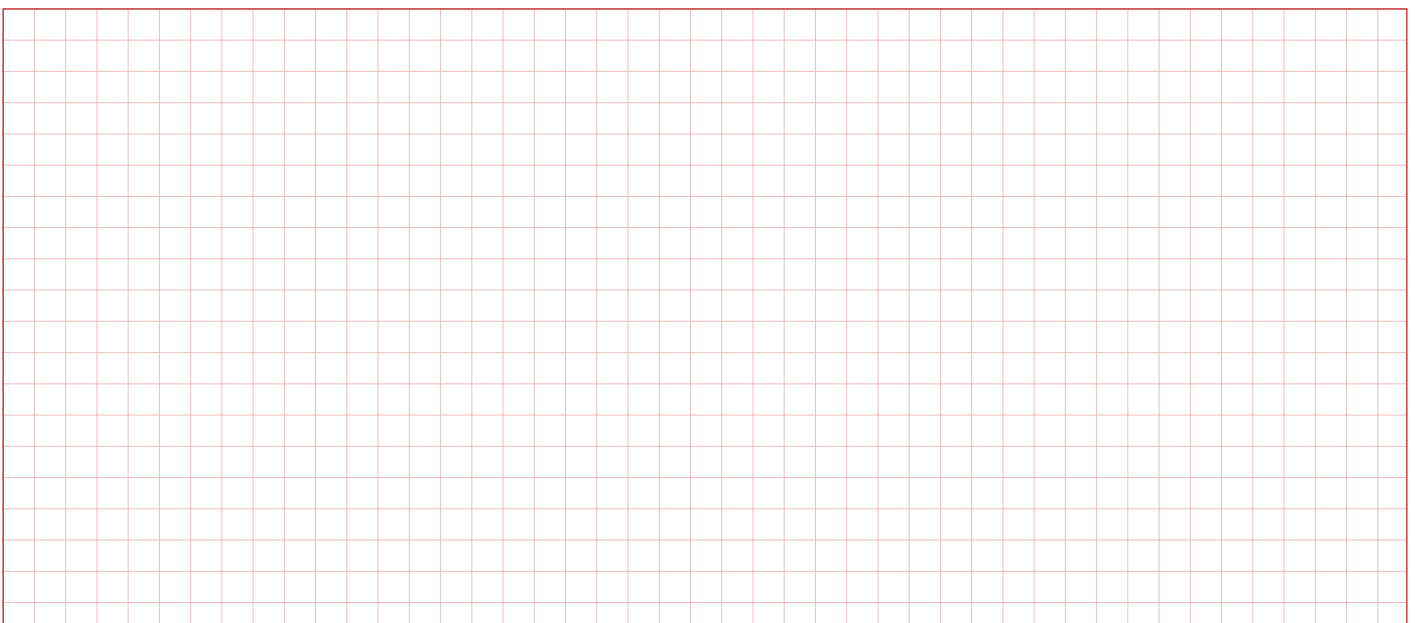
### Throwable

```
+ Throwable()
+ Throwable(message : String)
+ Throwable(message : String, cause : Throwable)
+ Throwable(cause : Throwable)
+ getMessage(): String
+ getCause(): Throwable
+ printStackTrace()
...
```

- ▶ **Basisklasse** aller Ausnahmen
- ▶ message — Nachricht der Ausnahme (möglichst **aussagekräftig**)
- ▶ cause — geschachtelte Ausnahme (später)
- ▶ [Exception](#)/[RuntimeException](#)/...implementieren obige **Konstruktoren**
- ▶ printStackTrace gibt den **Aufrufpfad** bis zur Ausnahme aus


## Notizen

- Siehe auch <https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Throwable.html>




## Beispiel: Throwable

- `printThrowableInfo` gibt Informationen zu [Throwable](#) aus

```
11  runPrintThrowableInfo  
12 public static void printThrowableInfo(Throwable t) {  
13     out.printf("Type: %s\n", t.getClass().getSimpleName());  
14     out.printf("Message: %s\n", t.getMessage());  
15     out.printf("Cause: %s\n", t.getCause());  
16     t.printStackTrace();  
17 }
```

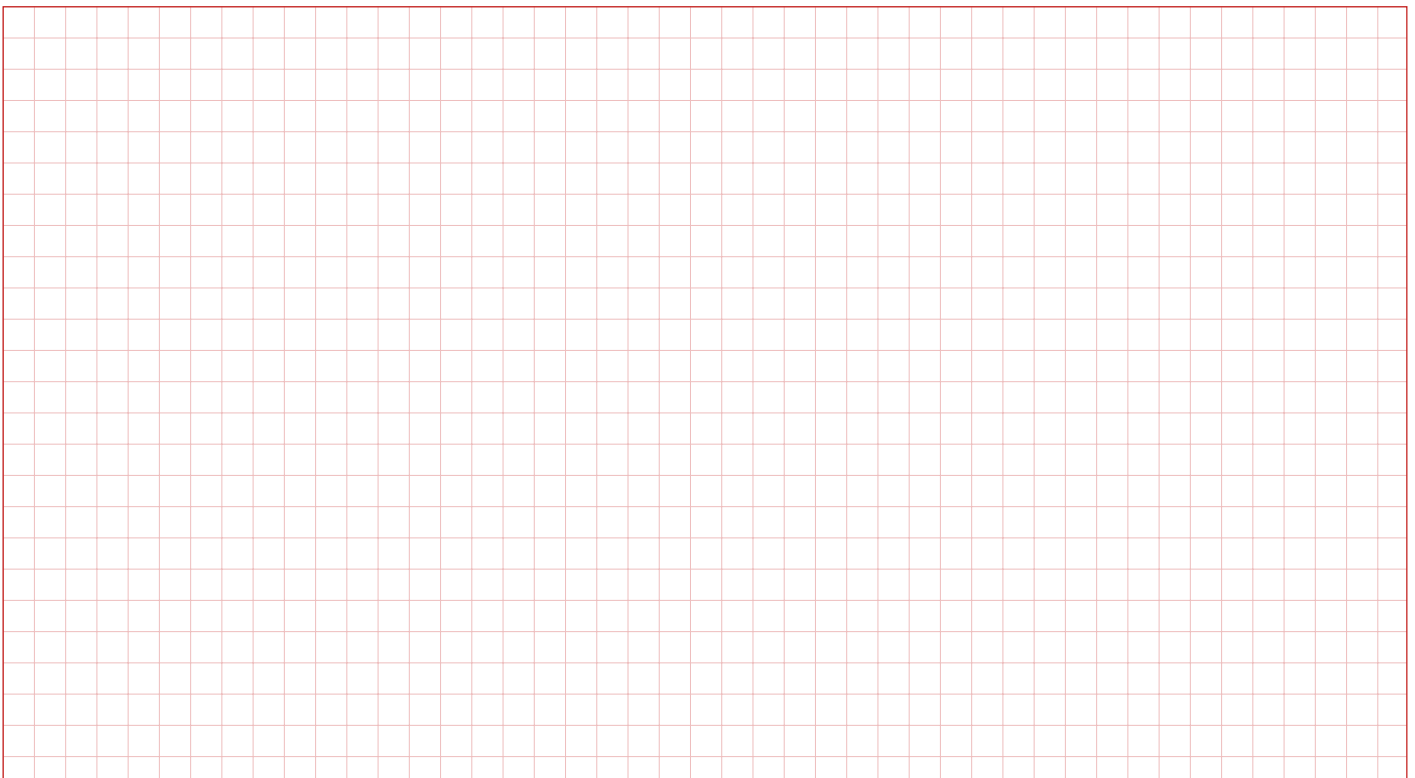
 `ThrowableExamples.java`

- Aufruf

```
22  runPrintThrowableInfoExample  
23 try {  
24     var examFile = new File("/home/auer/java1-exam.txt");  
25     var in = new FileInputStream(examFile);  
26 } catch (IOException exception){  
27     printThrowableInfo(exception);  
28 }
```

 `ThrowableExamples.java`

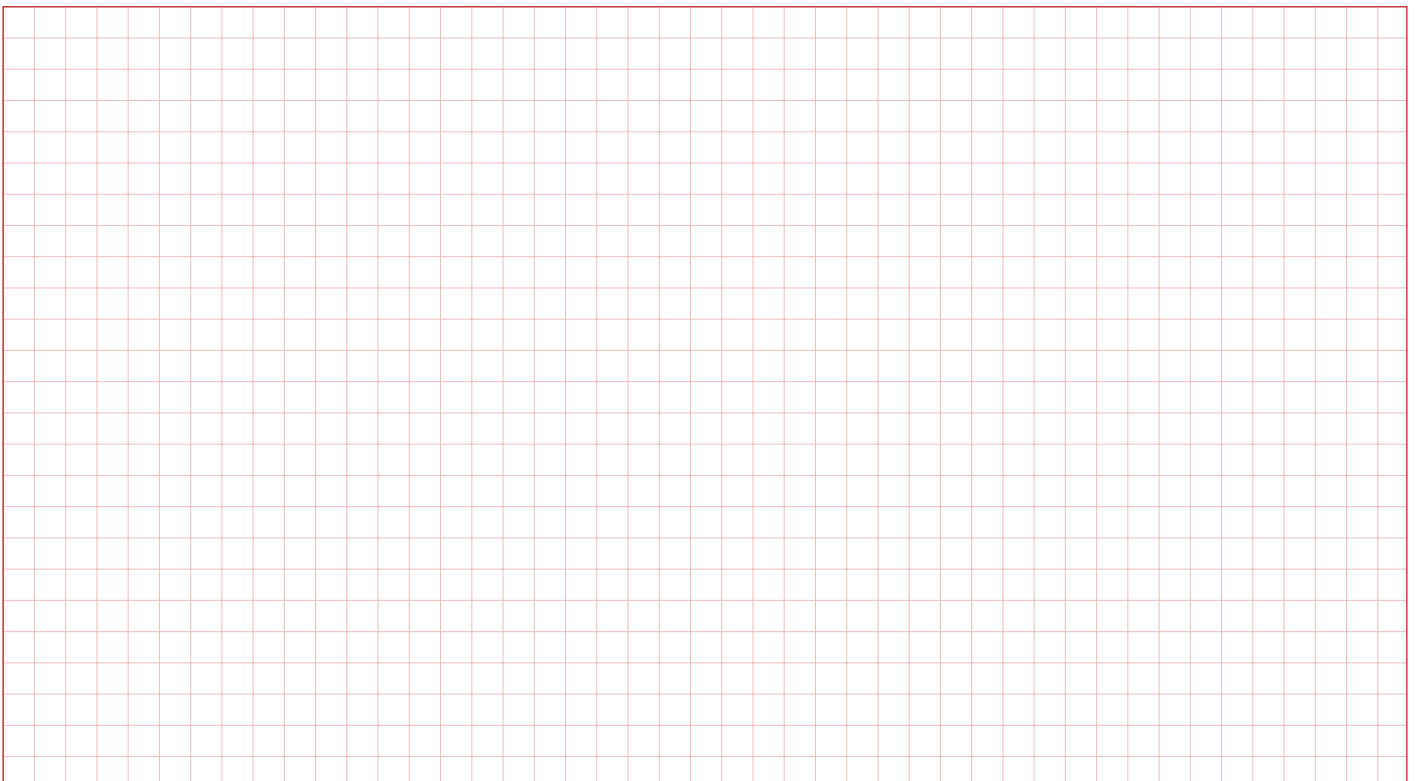
## Notizen



## Beispiel: Throwable

```
Type: FileNotFoundException
Message: /home/auer/java1-exam.txt (No such file or directory)
Cause: null
java.io.FileNotFoundException: /home/auer/java1-exam.txt (No such file or directory)
  at java.base/java.io.FileInputStream.open0(Native Method)
  at java.base/java.io. FileInputStream.open(FileInputStream.java:213)
  ...
```

## Notizen





# Inhalt

## Ausnahmen auslösen

Ausnahmen auslösen

Vordefinierte Ausnahmen des JDK

## Notizen

A large rectangular area filled with a fine grid of light gray lines, intended for taking handwritten notes. The grid consists of approximately 30 columns and 40 rows of small squares.

# Inhalt

## Ausnahmen auslösen

### Ausnahmen auslösen

- Neue Ausnahmen auslösen
- Ausnahmen weiterreichen
- Geschachtelte Ausnahmen
- Zusammenfassung

## Notizen

A large rectangular area filled with a fine grid of light gray lines, intended for taking notes. The grid is approximately 30 columns wide and 35 rows high.

# Inhalt

## Ausnahmen auslösen

## Ausnahmen auslösen


## Neue Ausnahmen auslösen

# Notizen

## Neue Ausnahmen auslösen

```
throw new ImportantException(...);
```

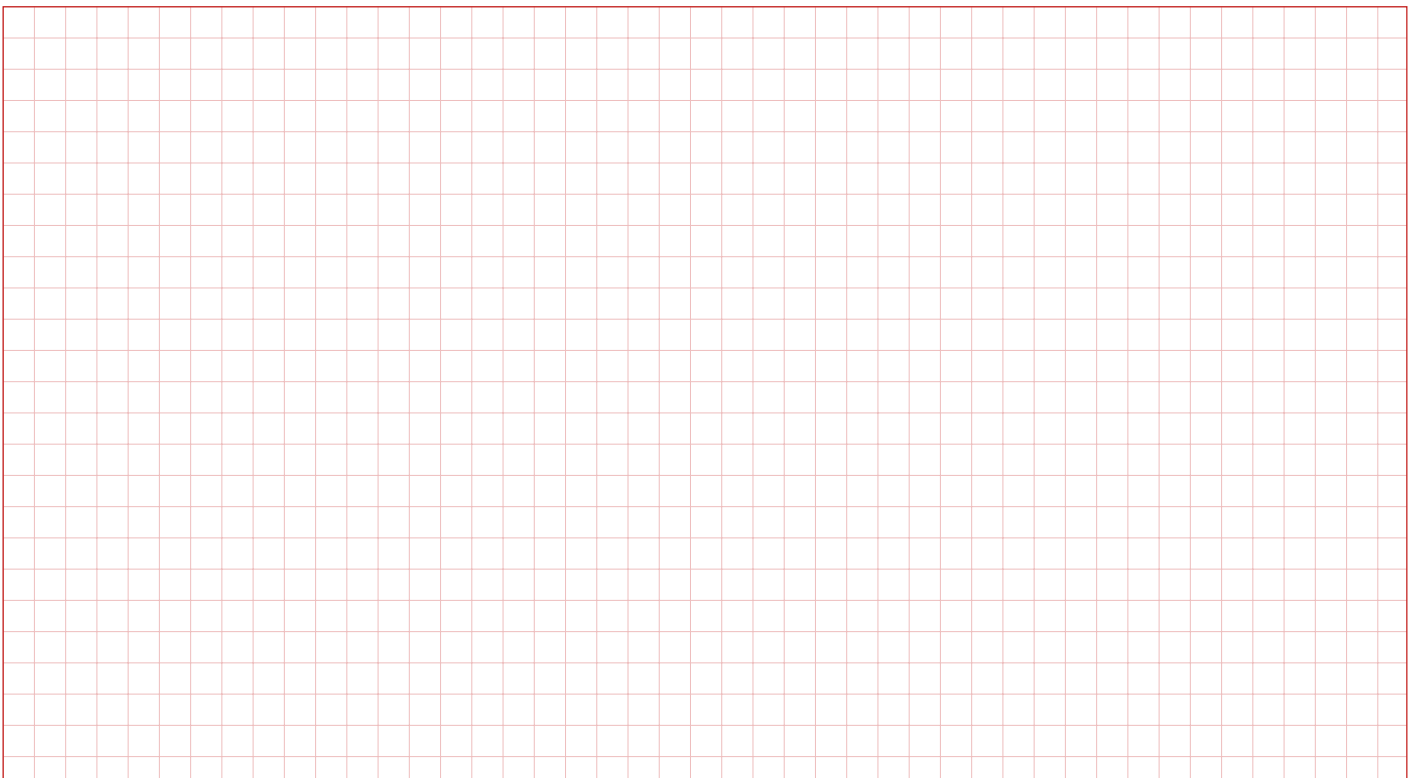
- ▶ **Schlüsselwort:** `throw`
- ▶ **Argument:** Instanz von `Throwable` oder abgeleiteter Klasse
- ▶ **Beispiel** (Fakultät berechnen)

```
20  runFactorial  
21 public static long factorial(int n) {  
22     if (n < 0)  
23         throw new IllegalArgumentException("n must not be negative");  
25     // n! berechnen und zurückgeben
```

`ThrowExamples.java`


- ▶ Löst `IllegalArgumentException` bei negativem n aus
- ▶ Ausnahme enthält (verhältnismäßig) **aussagekräftige Nachricht**
- ▶ `IllegalArgumentException` ist **nicht geprüft**

## Notizen



## Neue Ausnahmen auslösen

### ► Aufruf

```
35  runThrowExceptionExample  
36 try {  
37     factorial(-5);  
38 } catch (IllegalArgumentException exception){  
39     printInfo(exception);  
40 }
```

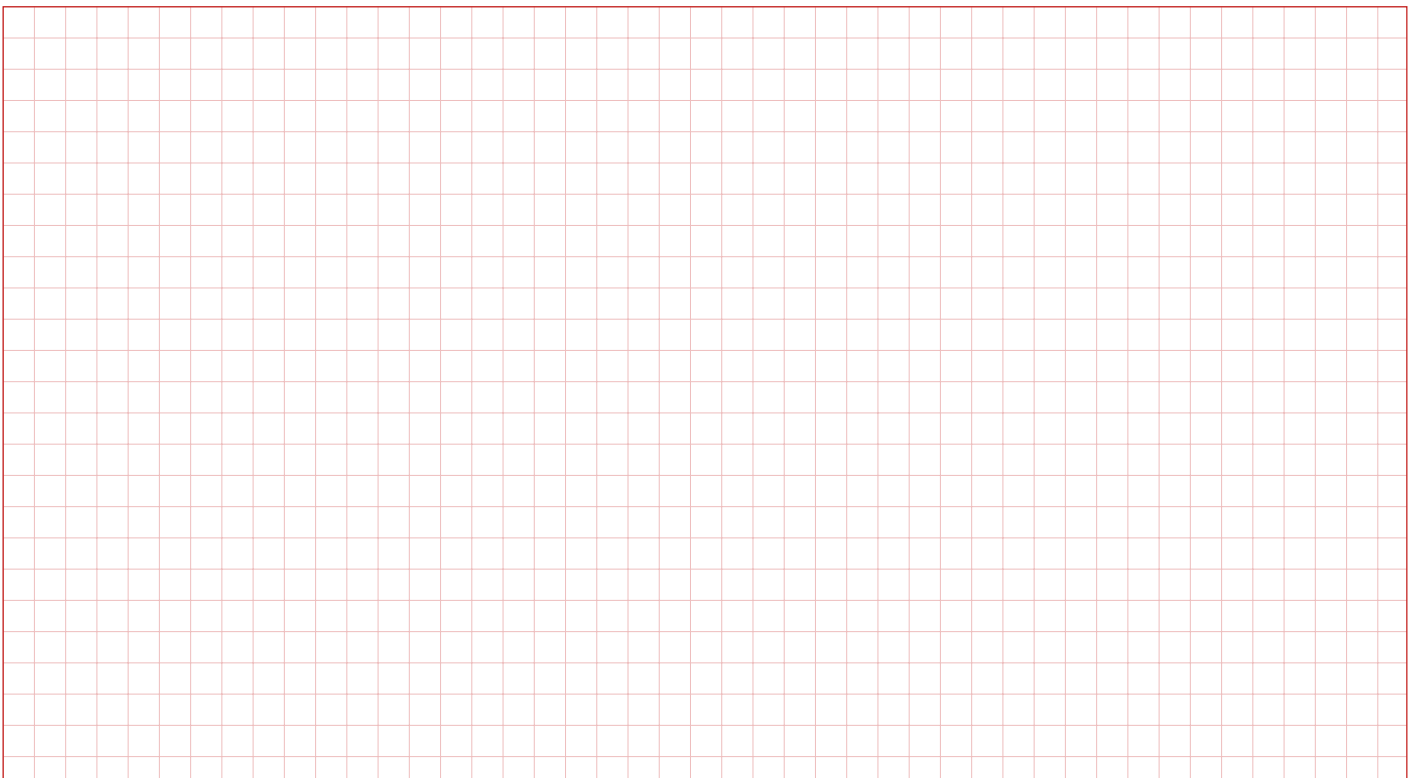
 ThrowExamples.java

printInfo gibt Informationen zu Exception aus

### ► Ausgabe

```
Type: IllegalArgumentException  
Message: n must not be negative  
Cause: null
```

## Notizen



# Inhalt

## Ausnahmen auslösen

## Ausnahmen auslösen

## Ausnahmen weiterreichen

## Geschachtelte Ausnahmen

## Notizen

## Ausnahmen weiterreichen

- ▶ Beispiel von vorher: liest erstes Byte aus Datei

```
45 public static int readFirstByte(String path)
46     throws FileNotFoundException, IOException {
```

ThrowExamples.java

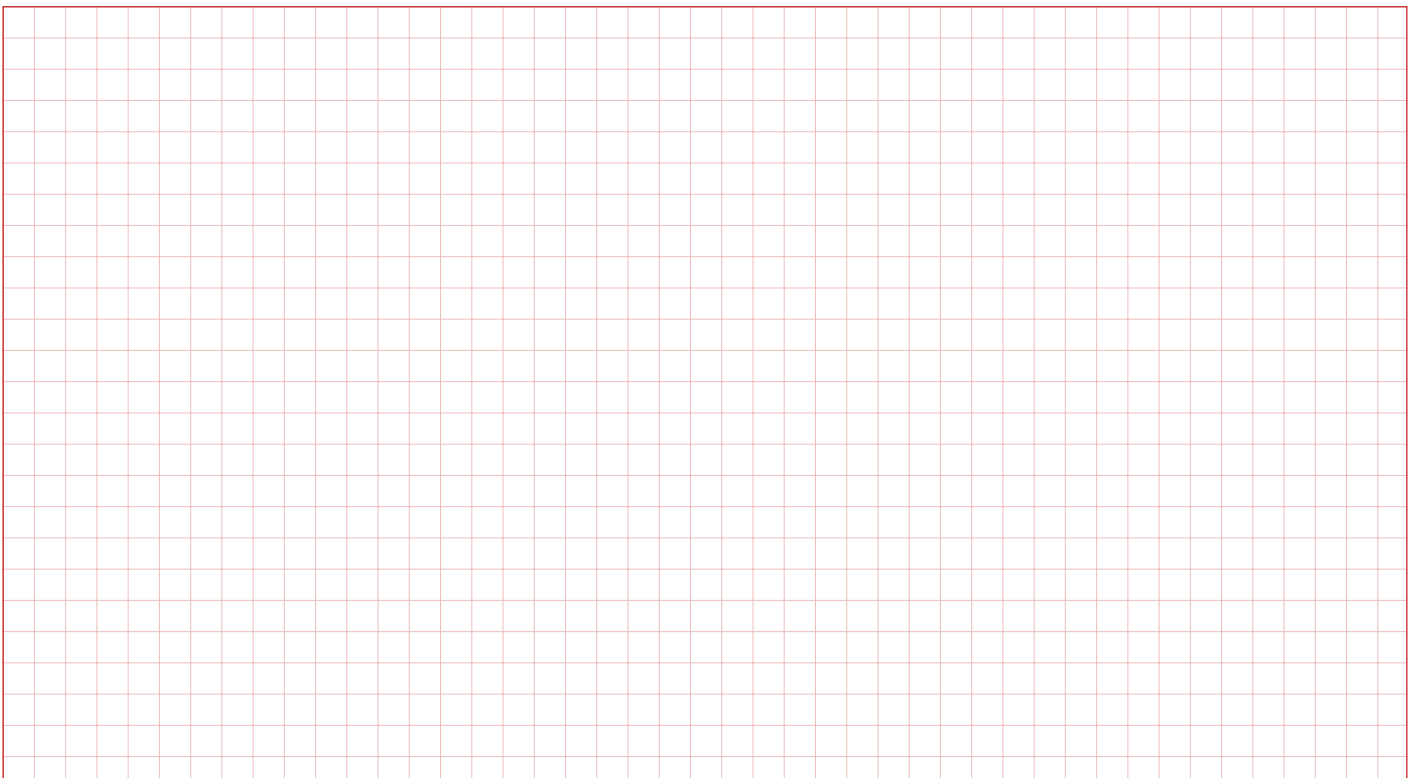
- ▶ Aufrufer kann auftretende Ausnahmen „weiterwerfen“ („rethrow“)

```
try {
    ...
} catch (ImportantException e){
    // Ausnahmebehandlung
    throw e; // weiterwerfen
}
```


- ▶ Ist ImportantException **geprüft** muss Methodendeklaration erweitert werden

```
void callingMethod() throws ImportantException
```

## Notizen

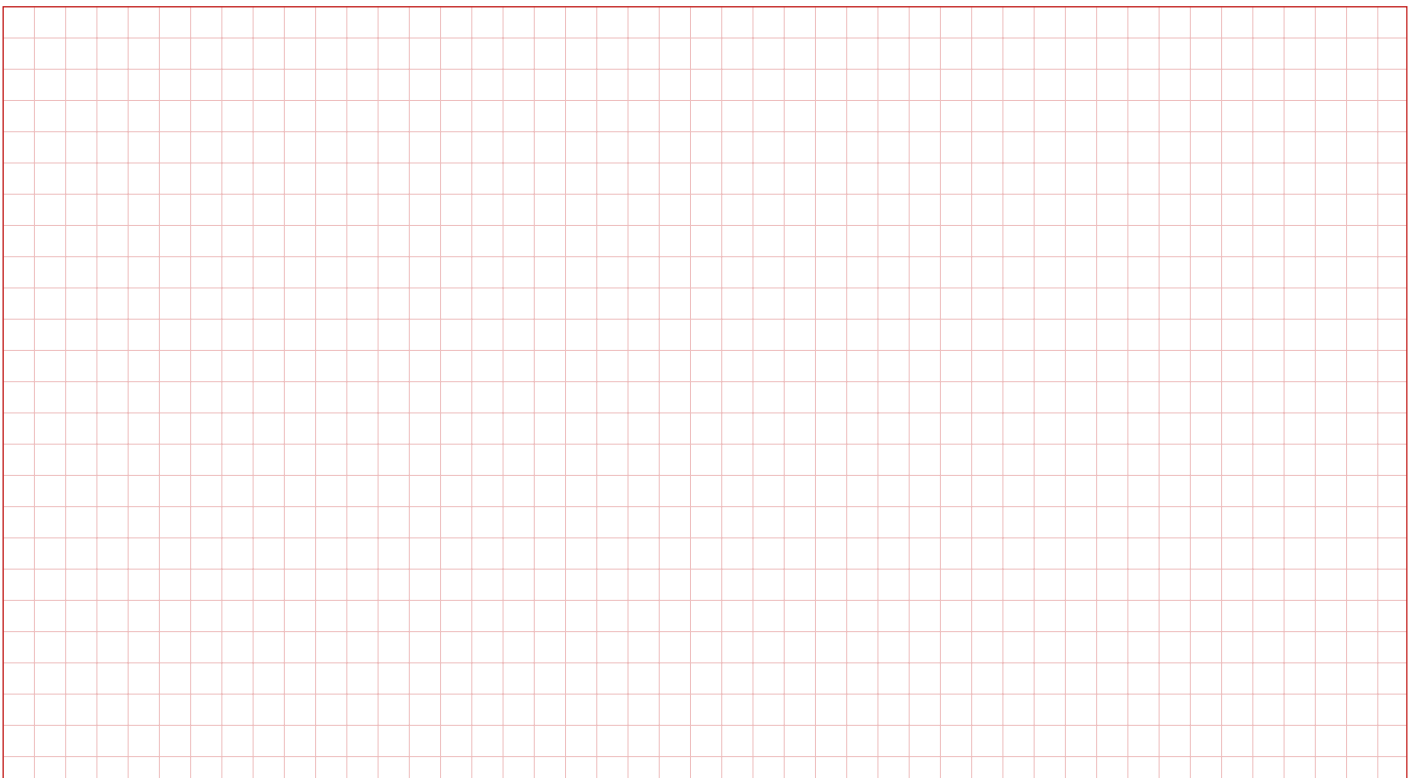


## Ausnahmen weiterreichen

```
57  runRethrowExceptionExample  
58 public static void rethrowExceptionExample()  
59     throws FileNotFoundException, IOException {  
60     try {  
61         readFirstByte("/home/auer/java1-exam.txt");  
62     } catch (FileNotFoundException e){  
64         printInfo(e); // "Ausnahmebehandlung"  
65         throw e;  
67     } catch (IOException e){  
69         printInfo(e); // "Ausnahmebehandlung"  
70         throw e;  
72     }  
73 }
```

 ThrowExamples.java

## Notizen





## Ausnahmen weiterreichen

Type: FileNotFoundException

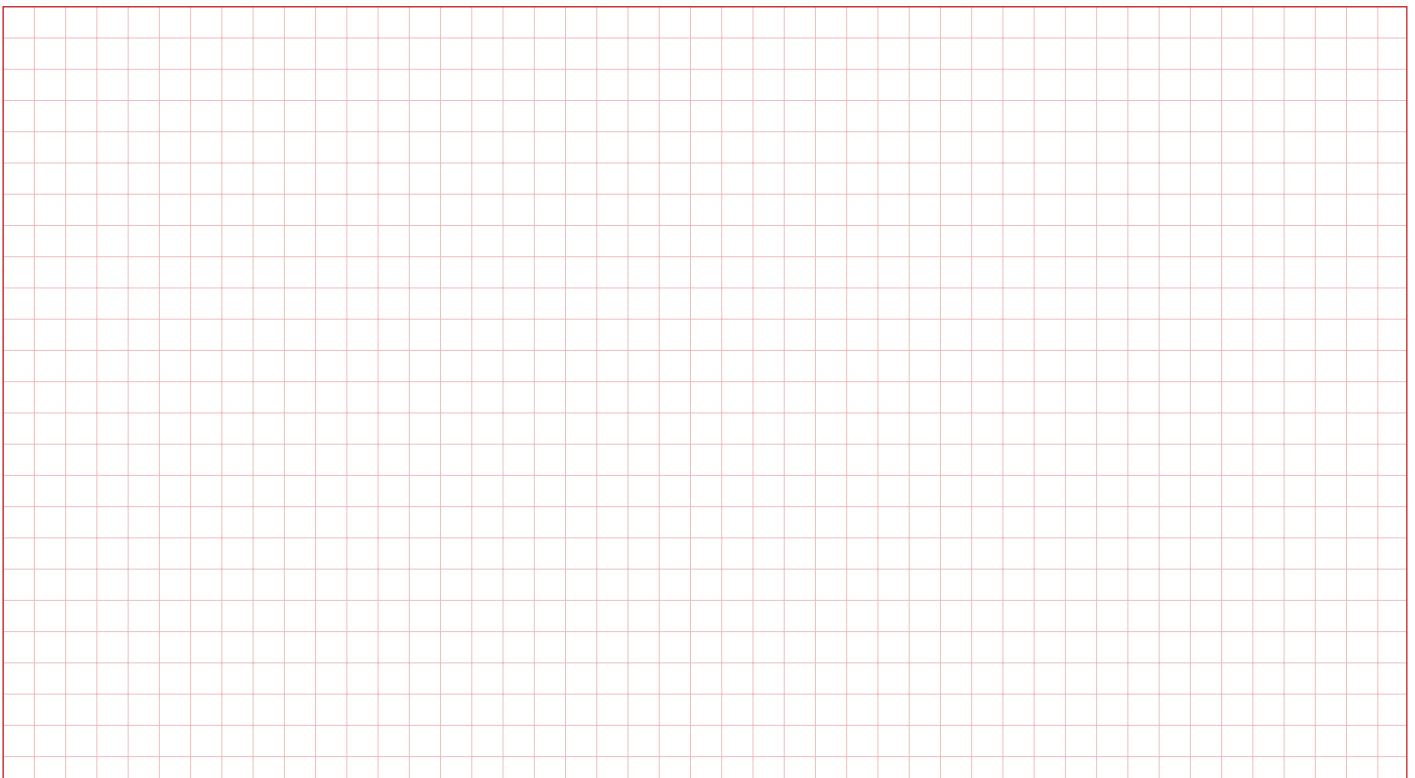
Message: /home/auer/java1-exam.txt (No such file or directory)

Cause: null

**FEHLER** java.io.FileNotFoundException: ...

- ▶ Nach Ausnahmebehandlung (hier nur Ausgabe) wird [↗ FileNotFoundException](#) weitergereicht
- ▶ Wann macht man so etwas?
  - ▶ Wenn Ausnahme **unverändert weiter propagiert** werden soll...
  - ▶ aber trotzdem im **Aufrufer behandelt** wird

## Notizen



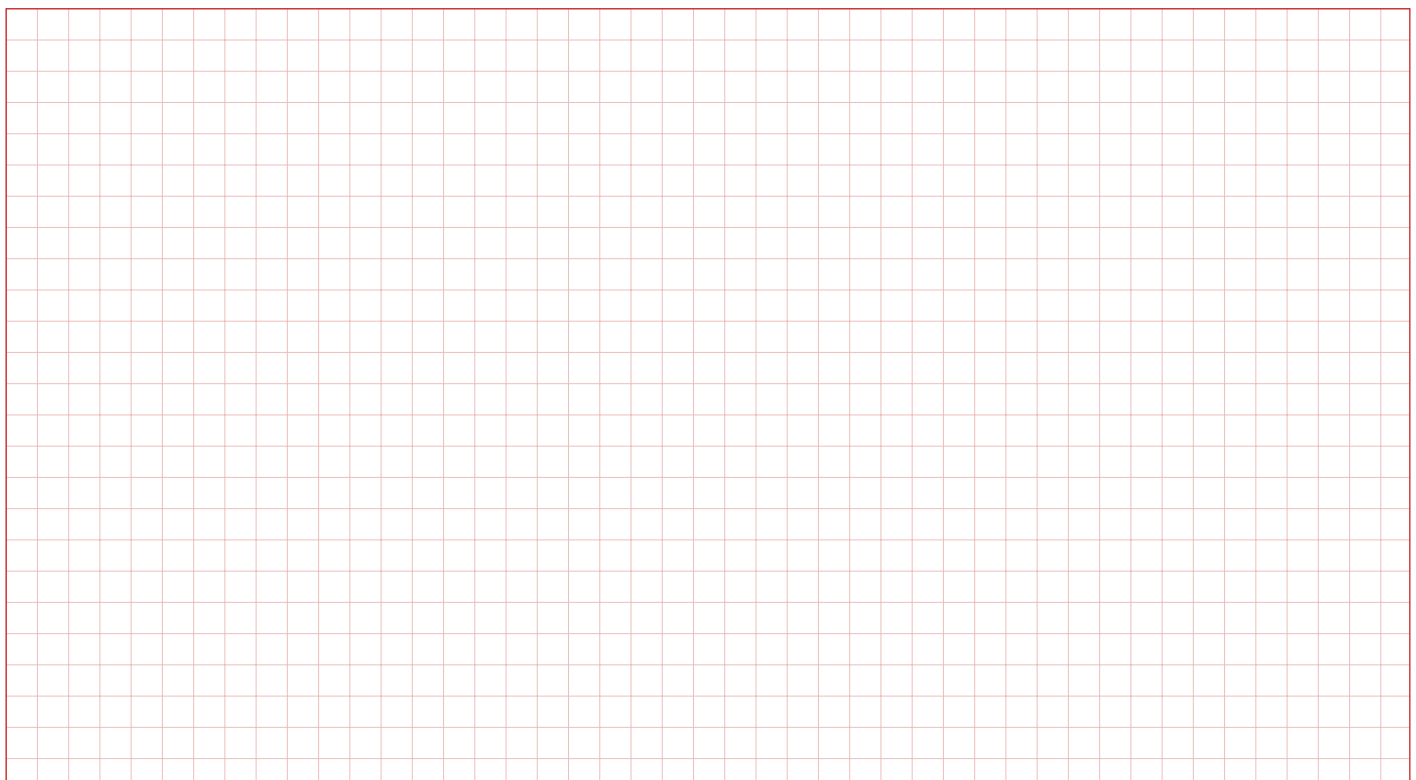
# Inhalt

## Ausnahmen auslösen

## Ausnahmen auslösen

## Geschachtelte Ausnahmen

# Notizen



## Geschachtelte Ausnahme weitergeben

- ▶ Wieder Beispiel von vorher

```
45 public static int readFirstByte(String path)
46     throws FileNotFoundException, IOException {
```

ThrowExamples.java

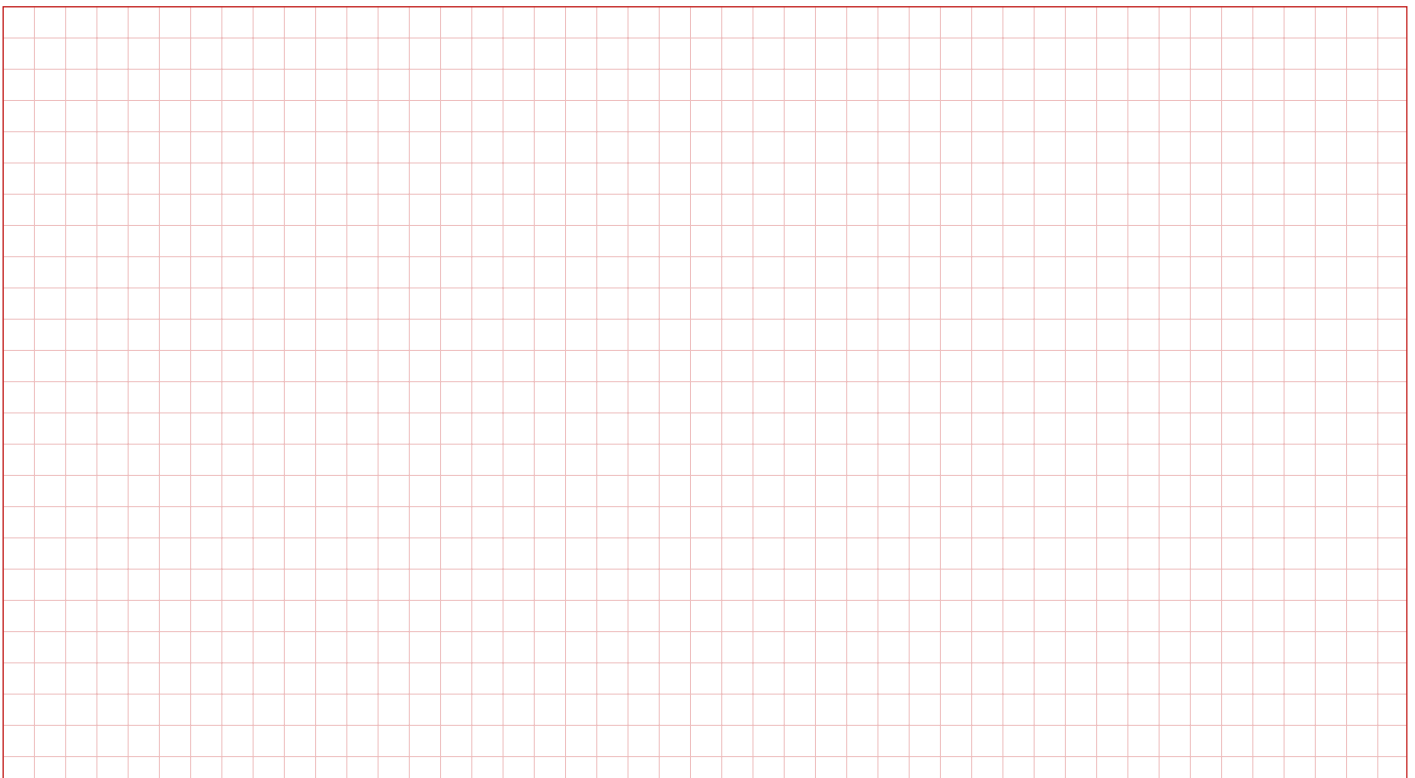
- ▶ Geworfene Ausnahmen sollen...
  - ▶ in [UncheckedIOException](#) geschachtelt werden
  - ▶ Dadurch von **geprüft** in **ungeprüfte** Ausnahme umgewandelt werden
- ▶ **Schachteln** einer Ausnahme

```
catch (ImportantException e){
    throw new NestingException("...", e);
}
```


NestingException: message=..., cause=

ImportantException: ...

## Notizen

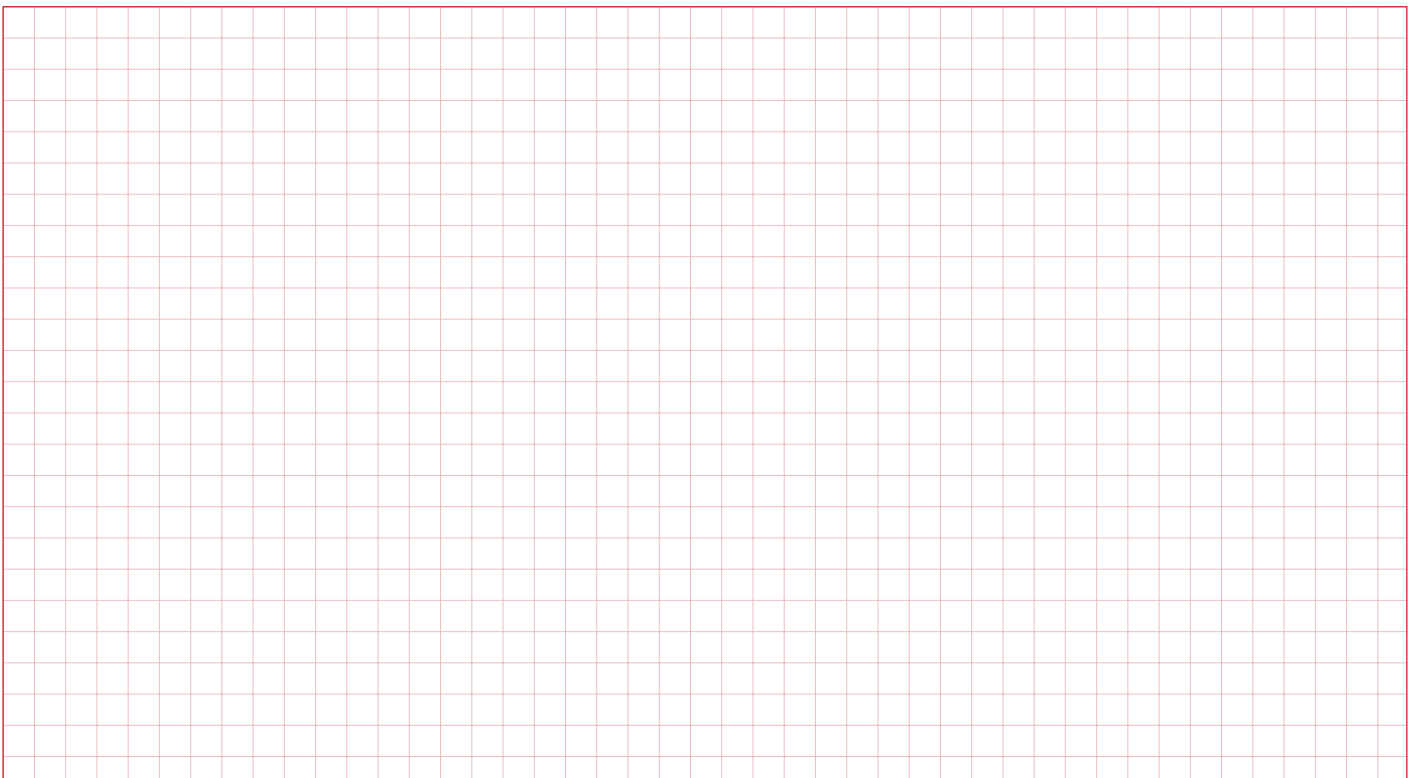


## Geschachtelte Ausnahmen

```
80  runNestExceptionExample  
81 try {  
82     readFirstByte("/home/auer/java1-exam.txt");  
83 } catch (FileNotFoundException e){  
84     var nestingE =  
85         new UncheckedIOException("Wrong path", e);  
86     printInfo(nestingE);  
87     throw nestingE;  
88 } catch (IOException e){  
89     var nestingE =  
90         new UncheckedIOException("File cannot be read", e);  
91     printInfo(nestingE);  
92     throw nestingE;  
93 }  
94 }
```

 ThrowExamples.java

## Notizen



# Geschachtelte Ausnahmen

## ► Ausgabe

```
Type: UncheckedIOException
Message: Wrong path
Cause: java.io.FileNotFoundException: /home/auer/java1-exam.txt (No such file ↔
      or directory)
FEHLER java.io.UncheckedIOException: ...
```

- [FileNotFoundException](#)/[IOException](#) (geprüft) ist in [UncheckedIOException](#) (ungeprüft) geschachtelt
- Zuerst **geprüfte Ausnahme**, dann **ungeprüft**
- Wann macht man so etwas?
  - Wenn **aufrufende Methode** Ausnahmen **nicht in Methodendeklaration** aufnehmen kann/will (**throws**)
  - Wenn **Programmier-Fehler angezeigt** werden soll (ungeprüfte Ausnahme)
- Funktioniert natürlich genauso **umgekehrt**
  - **(Un)geprüft** in **(un)geprüft** schachteln

## Notizen

- Es gibt mehrere Gründe warum eine Methode geprüfte Ausnahmen nicht in die Methodendeklaration aufnehmen will:
  - Zum Beispiel haben sich schon viele Ausnahmen in der Methodendeklaration aufgesammelt. So können sich zu den beiden Ausnahmen in `readFirstByte` noch weitere Ausnahmen hinzugesellen. Dies wird irgendwann unübersichtlich und unhandlich in Deklaration und Verwendung. Dann macht es Sinn die Ausnahmen in eine einzige Ausnahme zu schachteln.
  - Handelt es sich um eine überschriebene Methode, so gelten Einschränkungen welche Ausnahmen in die Methodendeklaration aufgenommen werden können. Auch hier kann es Sinn machen auftretende Ausnahmen in einen kompatiblen Ausnahmetypen zu schachteln.



## Geschachtelte Ausnahmen

```
void manyExceptions()  
    throws E1, E2, E3{  
    ...  
}
```

```
void evenMoreExceptions()  
    throws E1, E2, E3, E4, E5{  
    manyExceptions();  
    ...  
}
```

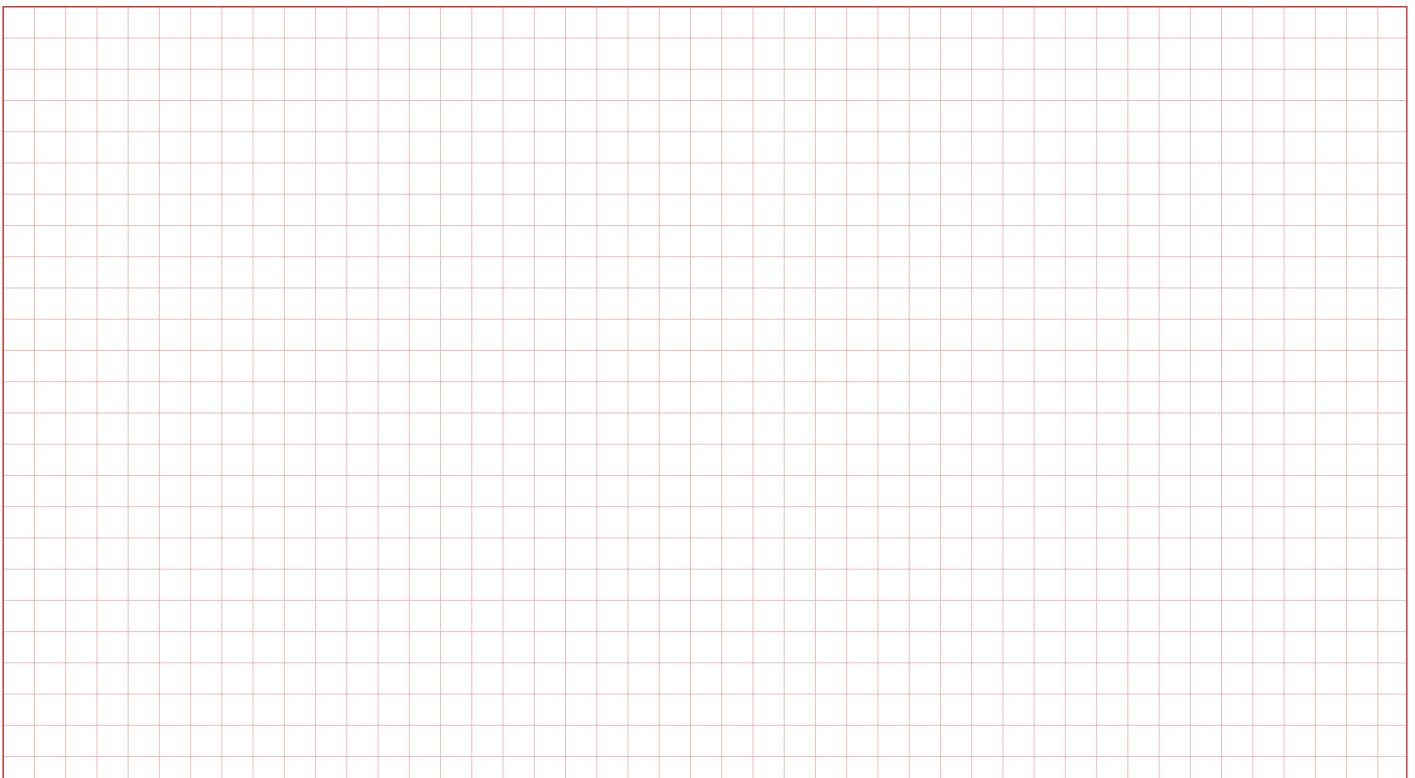
- ▶ Ausnahmen in **Methodendeklarationen** können sich **ansammeln**
  - ▶ manyExceptions wirft E1, E2, E3
  - ▶ evenMoreExceptions ruft manyExceptions auf und wirft E4, E5
- ▶ **Problem**: unübersichtlich, unhandlich
- ▶ **Lösungsansatz**: geschachtelte Ausnahmen
  - ▶ Verpacke E1, ..., E5 in neue Ausnahme E
  - ▶ **Beispiel** für E1

```
catch (E1 e1){ throw new E(e1); }
```

- ▶ Signatur wird wieder **klein**

```
void lessExceptions() throws E
```

## Notizen



# Inhalt

## Ausnahmen auslösen

## Ausnahmen auslösen

## Zusammenfassung

## Notizen

## Zusammenfassung

- ▶ **Neue** Ausnahme erzeugen mit **throw**

```
throw new IOException("File not found");
```

- ▶ **Weiterreichen** von Ausnahmen

```
catch (ImportantException e){  
    throw e;  
}
```

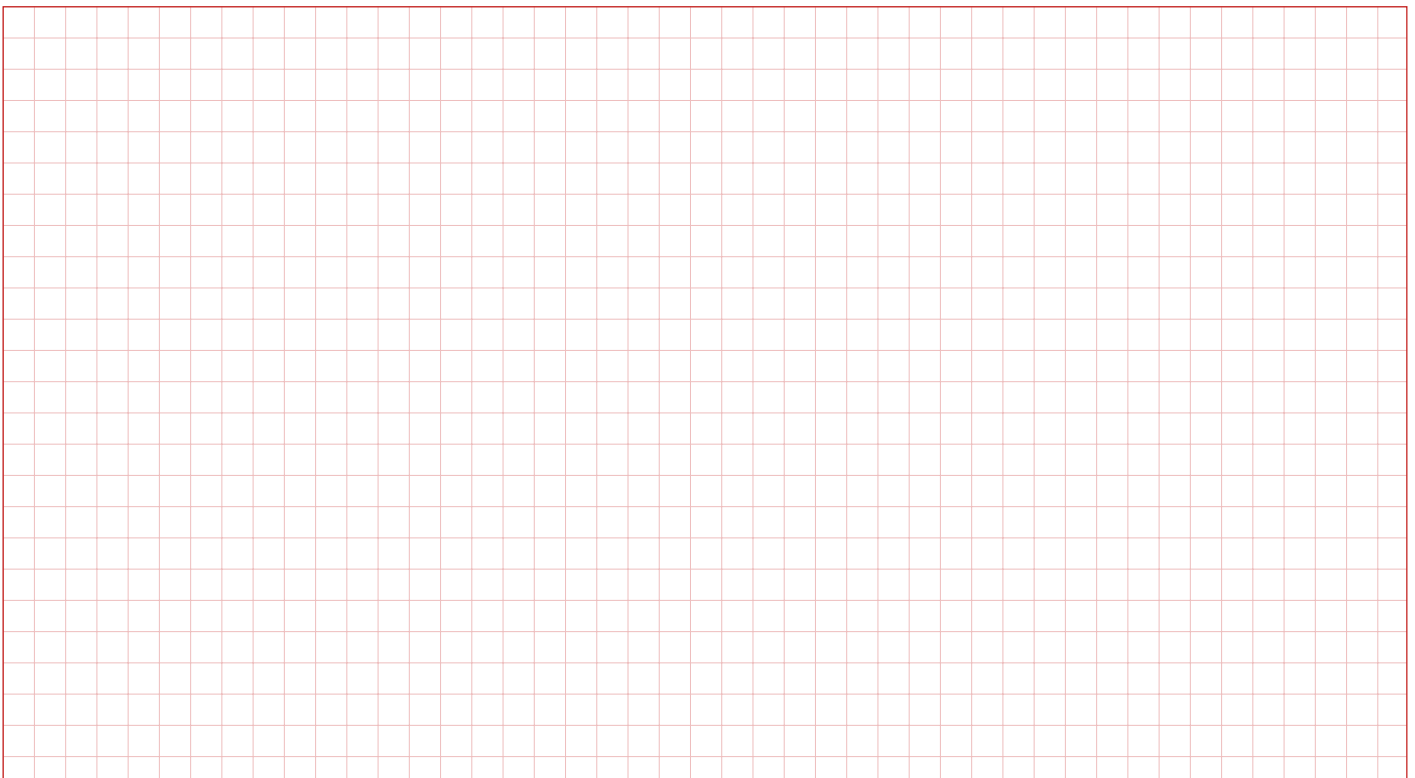
- ▶ **Schachteln** von Ausnahmen

```
catch (ImportantException e){  
    throw new NestingException(e);  
}
```

Bei

- ▶ **Umwandlung** von Ausnahmen
- ▶ **Verkleinern** von Methodensignaturen

## Notizen



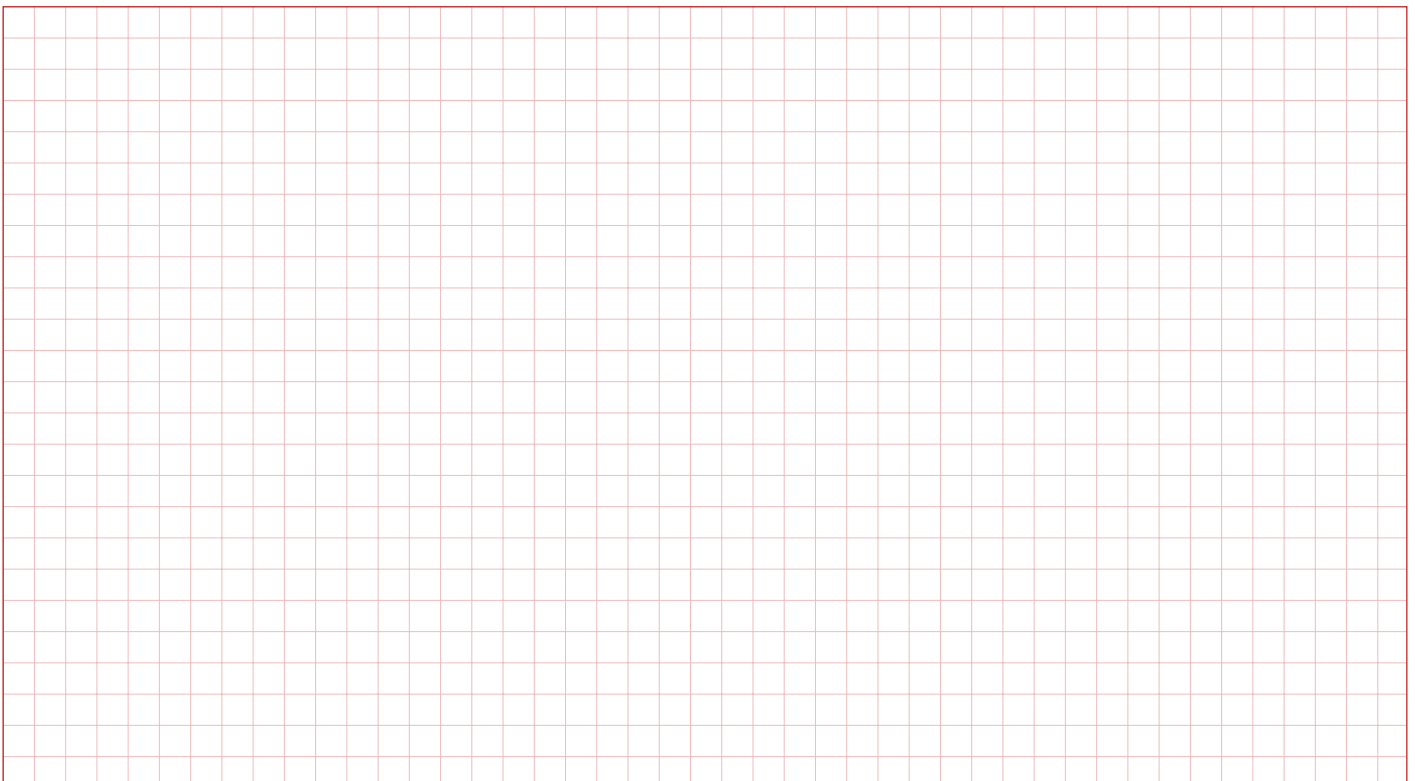


# Inhalt

## Ausnahmen auslösen

Vordefinierte Ausnahmen des JDK

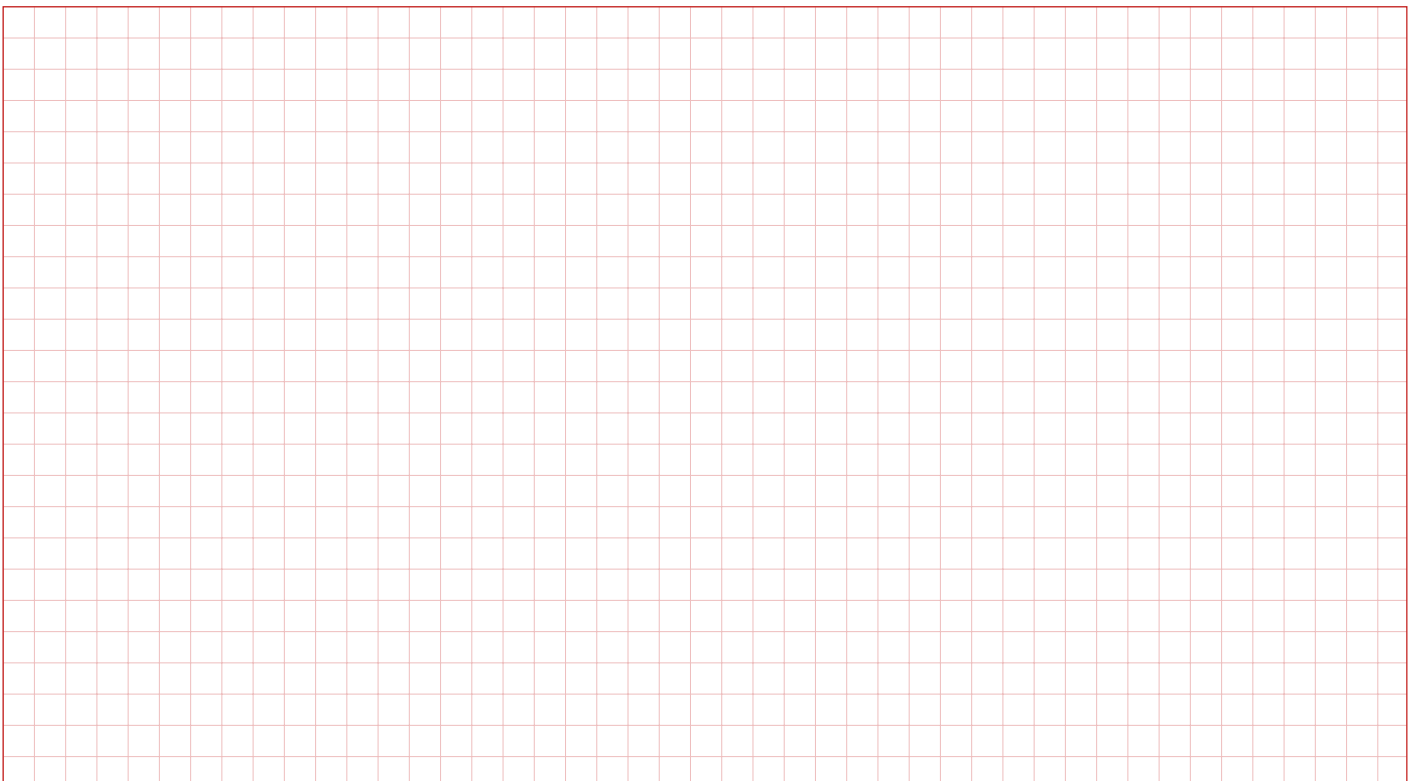
## Notizen



# Vordefinierte Ausnahmen des JDK


- ▶ JDK kommt mit **vordefinierten** Ausnahmen
- ▶ Geprüfte Ausnahmen: <https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Exception.html>
- ▶ Ungeprüfte Ausnahmen: <https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/RuntimeException.html>

## Notizen



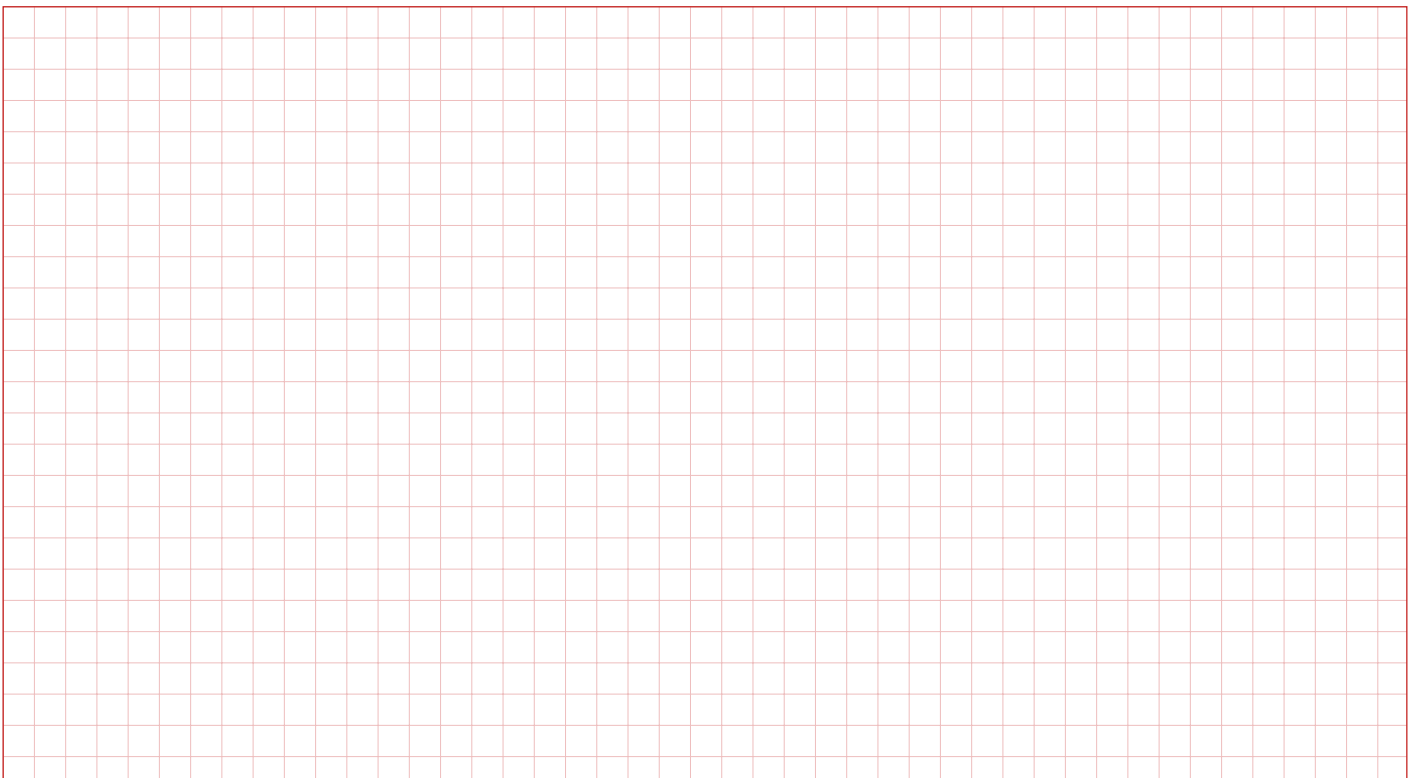
# IllegalArgumentException

- ▶ [IllegalArgumentException](#) extends `RuntimeException`
- ▶ Ungeprüft
- ▶ Wann werfen?
  - ▶ Inkorrektter Wert für `Methodenparameter`, z.B., außerhalb von Wertebereich, `null`, etc.
- ▶ Beispiel

```
20  runFactorial  
21 public static long factorial(int n) {  
22     if (n < 0)  
23         throw new IllegalArgumentException("n must not be negative");  
25     // n! berechnen und zurückgeben
```

 `ThrowExamples.java`

## Notizen

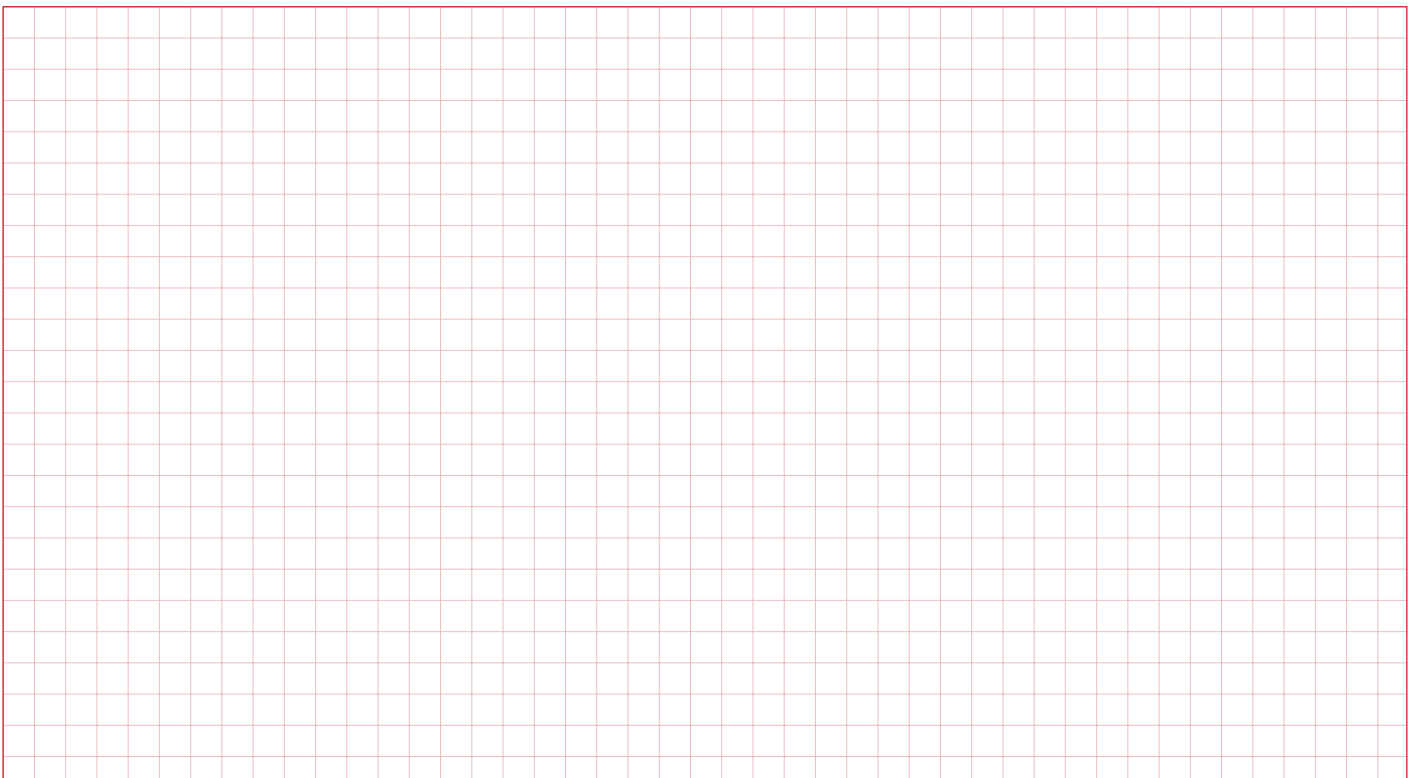


# IllegalStateException

- ▶ [↗](#) `IllegalStateException` extends `RuntimeException`
- ▶ Ungeprüft
- ▶ Wann werfen?
  - ▶ Methode kann im **aktuellen Zustand** des Objekts **nicht aufgerufen** werden
- ▶ Beispiele
  - ▶ [↗](#) `Scanner.readInt()` kann nicht aufgerufen werden wenn der **unterliegende Datenstrom** bereits **geschlossen ist**
  - ▶ Siehe **Übung** zu „Game of Thrones“-Charakteren

```
public void fight(){
    if (!alive)
        throw new IllegalStateException("Dead!");
    // ...
}
```

## Notizen

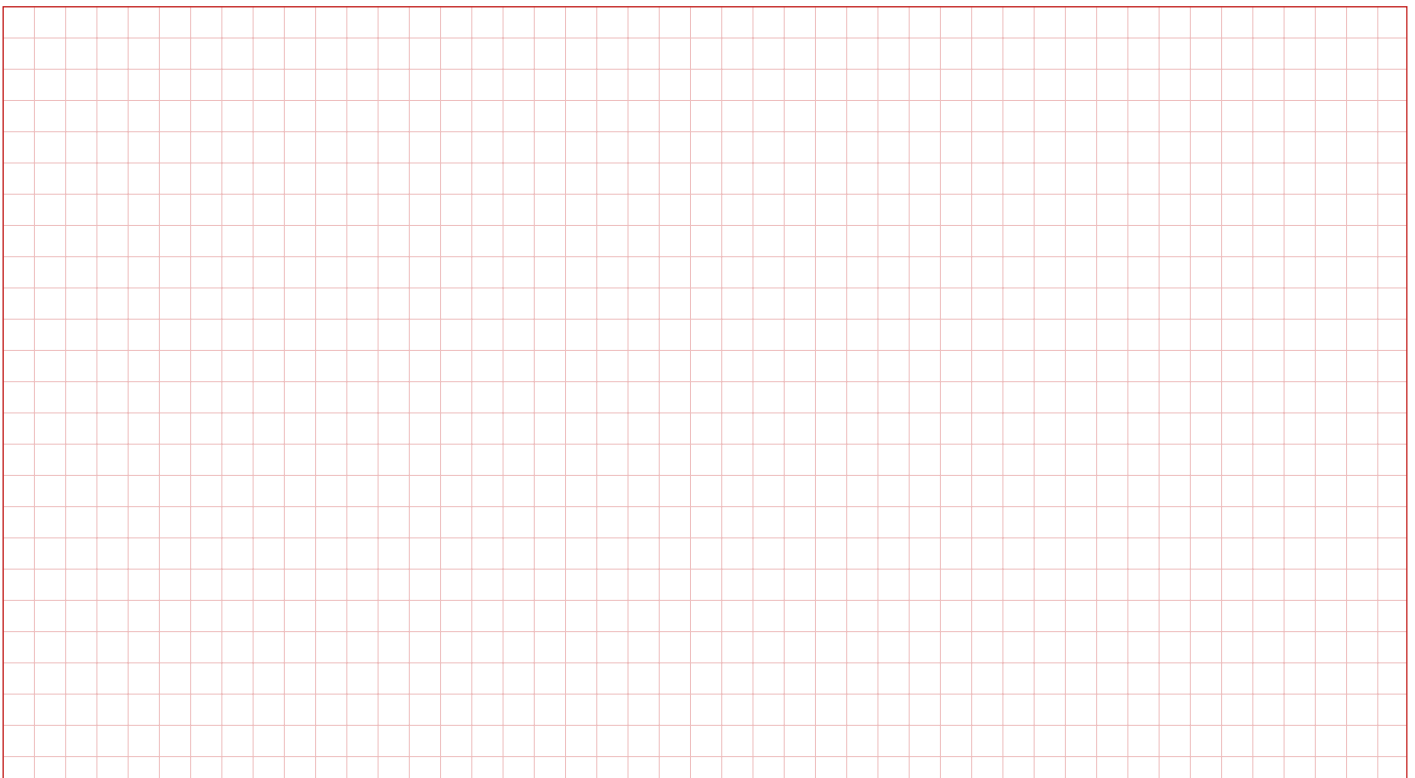


# UnsupportedOperationException

- ▶ [↗ UnsupportedOperationException](#) extends `RuntimeException`
- ▶ Ungeprüft
- ▶ Wann werfen?
  - ▶ Wenn Methode für Klasse **keinen Sinn** macht/**nicht unterstützt** wird
  - ▶ Meist bei **geerbten Methoden**
- ▶ **Beispiel** aus unserem „Spiel“: MagicWand

```
@Override
public Consumable split(int n) {
    throw new
        UnsupportedOperationException("Cannot split wand");
}
```

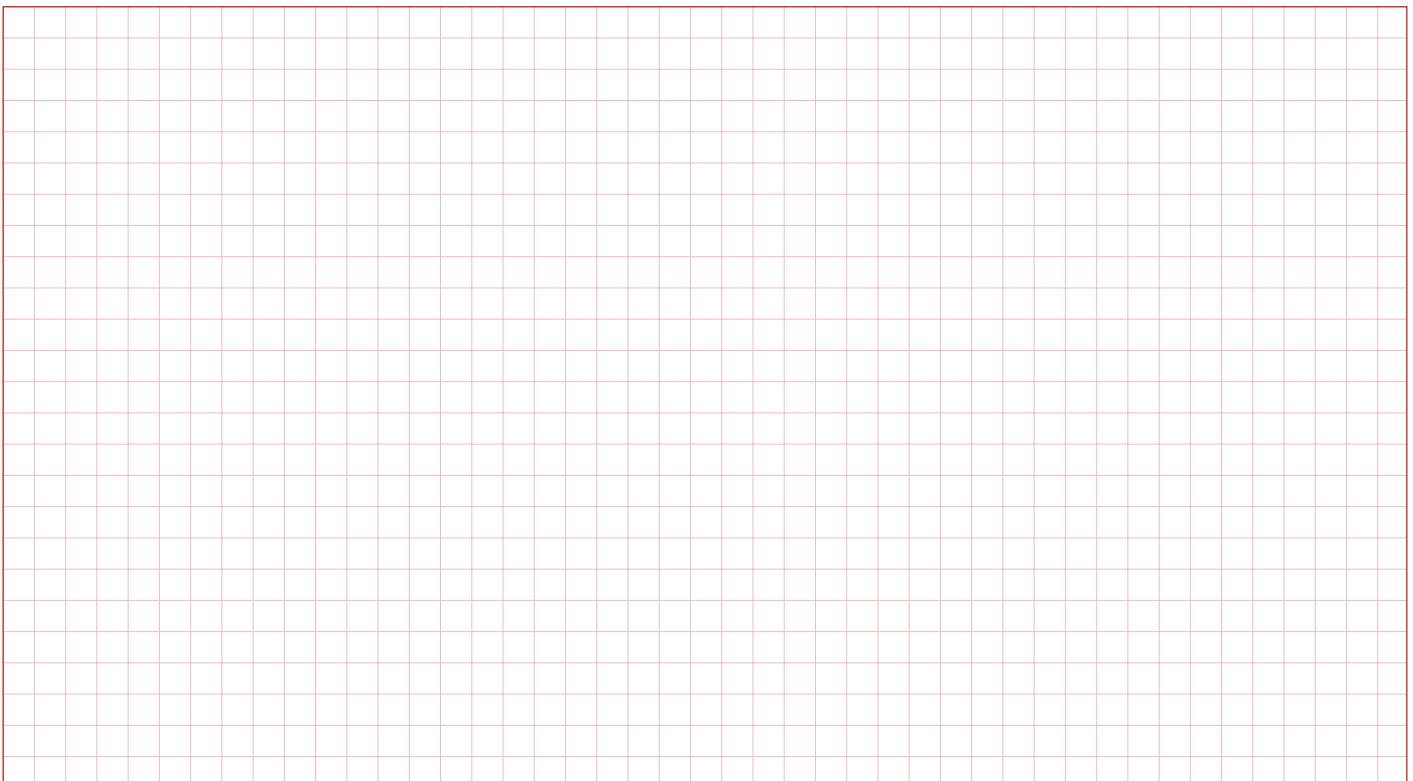
## Notizen



# IndexOutOfBoundsException

- ▶ `IndexOutOfBoundsException` extends `RuntimeException`
- ▶ Ungeprüft
- ▶ Wann werfen?
  - ▶ Wenn Zugriffs-Index „irgendeiner Art“ außerhalb des gültigen Bereichs ist
  - ▶ Meist bei geerbten Methoden
- ▶ Beispiele
  - ▶ `String.charAt(int)`
  - ▶ `ArrayList.get(int)`/`ArrayList.remove(int)`
  - ▶ `Merchant.buy(int)` sollte `IndexOutOfBoundsException` werfen

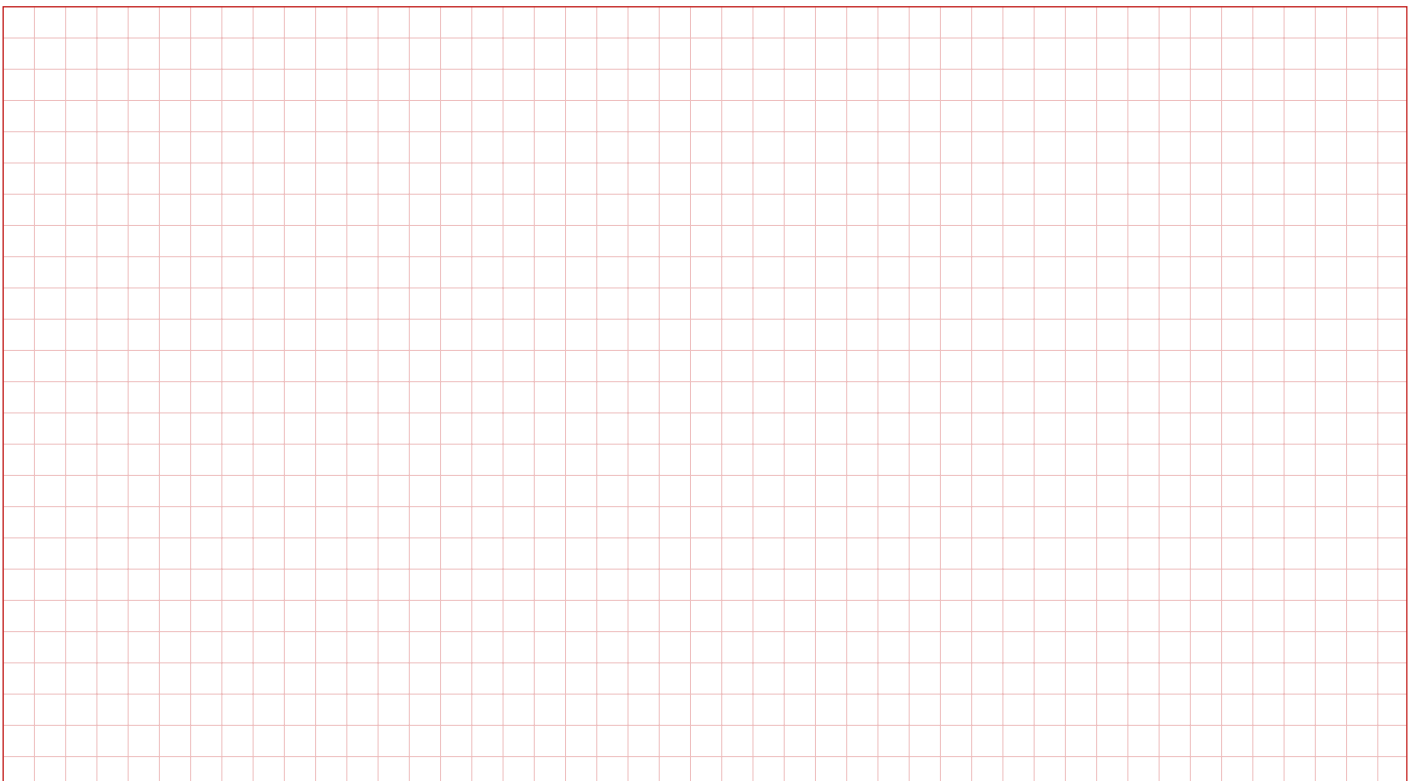
## Notizen



# IOException

- ▶ [IOException](#) extends `Exception`
- ▶ Geprüft ([UncheckedIOException](#) ungeprüft)
- ▶ Wann werfen?
  - ▶ Bei Ausnahmen die mit I/O (Datenstromein-/ausgabe) zu tun haben
- ▶ [IOException](#) hat viele spezifische Unterklassen
- ▶ Beispiele
  - ▶ Datei nicht gefunden: [FileNotFoundException](#)
  - ▶ Hostname nicht auflösbar: [UnknownHostException](#)
  - ▶ Genereller Fehler beim Lesen/Schreiben: [IOException](#)

## Notizen



## Automatisch generierte Ausnahmen

- ▶ Manche Ausnahmen werden i.d.R. **nicht selbst** erzeugt
- ▶ Werden von JVM **automatisch erzeugt**
- ▶ [↗ ArithmeticException](#) — Fehler in arithmetischen Berechnungen

```
int p = 1;  
int q = 0;  
int x = p/q;
```

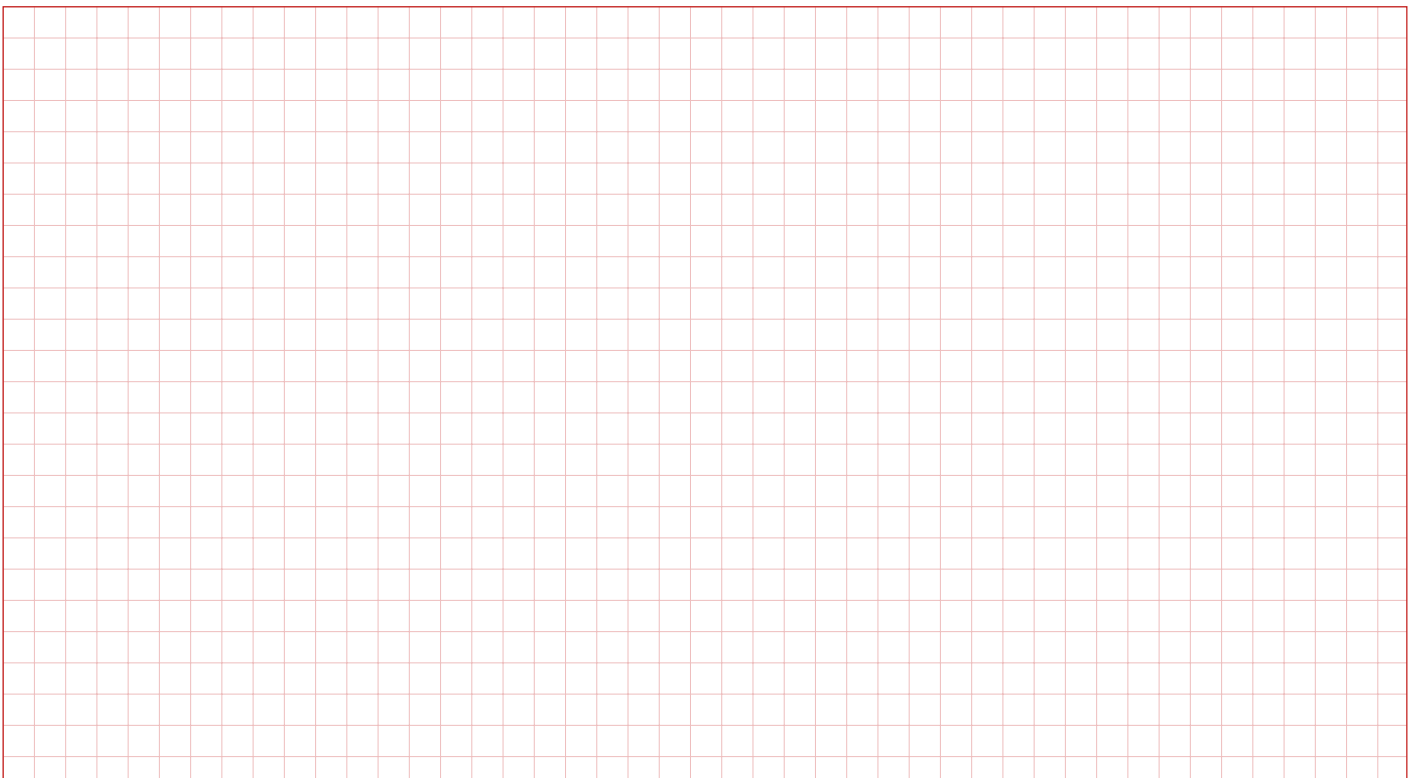
- ▶ [↗ NullPointerException](#) — Zugriff auf **null**-Referenz
  - ▶ Kann mit **sinnvoller** Nachricht selbst geworfen werden
  - ▶ Beispiel:

```
String s = null;  
char c = s.charAt(0);
```

- ▶ [↗ ClassCastException](#) — ungültiger Cast

```
String s = "I'm Geralt of Rivia!";  
Player p = (Player) s;
```

## Notizen





# Inhalt

- Eigene Ausnahmen definieren
  - Eigene Ausnahmeklassen definieren

## Notizen

A large rectangular area filled with a fine grid of light gray lines, intended for taking handwritten notes. The grid consists of approximately 30 columns and 40 rows of small squares.

# Inhalt

## Eigene Ausnahmen definieren

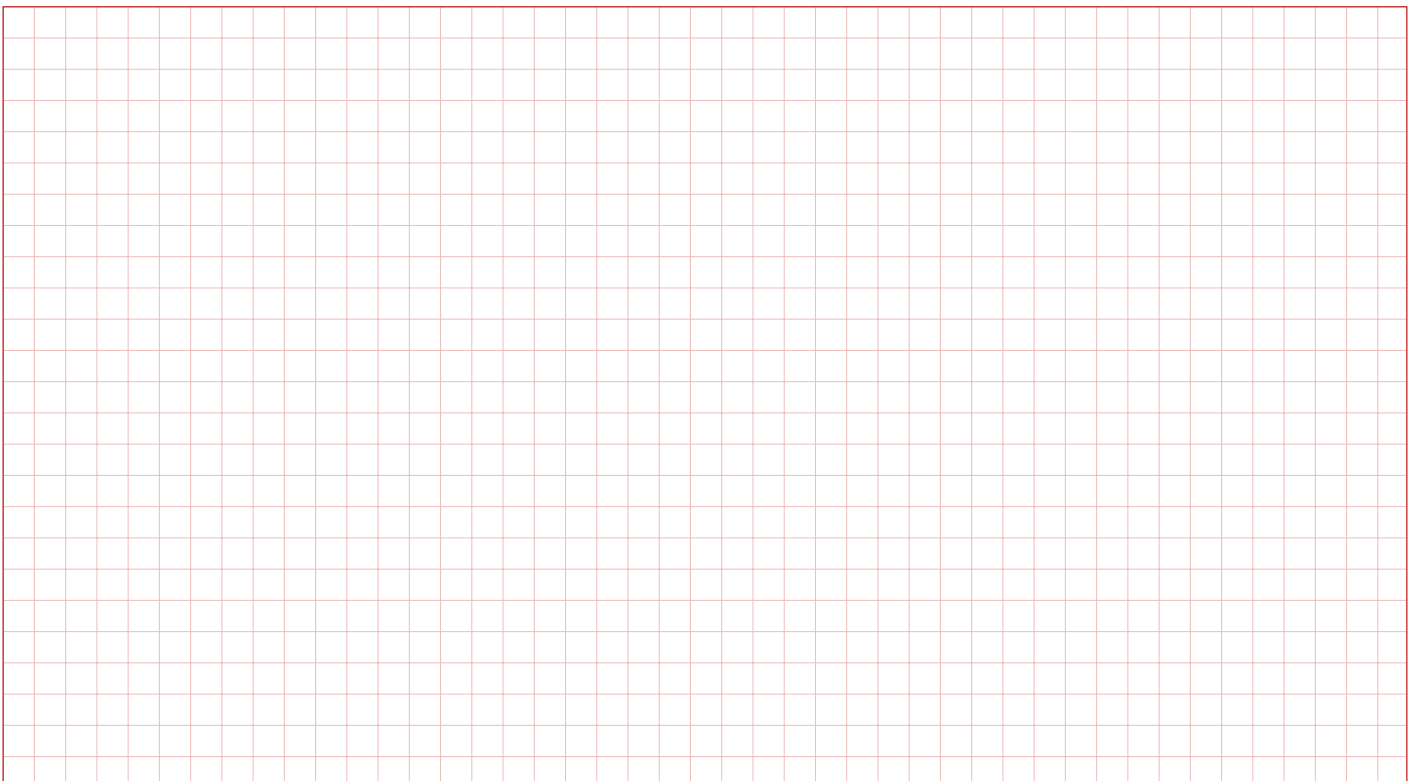
- Eigene Ausnahmeklassen definieren

  - Beispiel: Consumable

  - Geprüfte oder ungeprüfte Ausnahmen

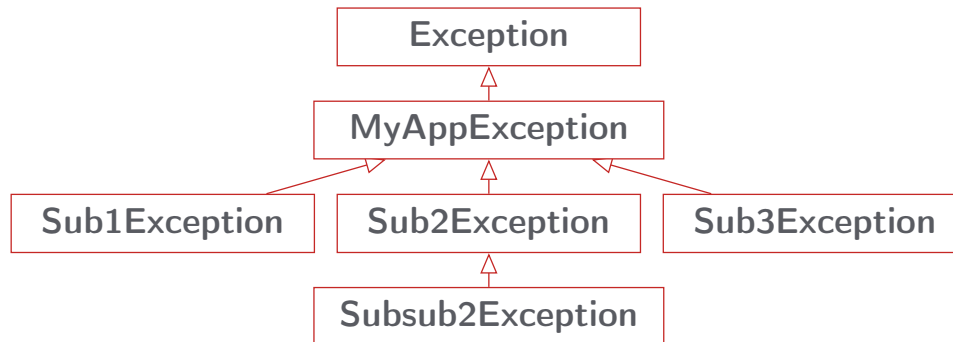
  - Zusammenfassung

## Notizen



## Eigene Ausnahmeklassen definieren

- ▶ Manchmal sind **vordefinierte JDK-Ausnahmen** zu **unspezifisch**
- ▶ Eigene Ausnahmeklassen definieren
  - ▶ **Geprüft**: von `Exception` ableiten
  - ▶ **Ungeprüft**: von `RuntimeException` ableiten
- ▶ **Besser**: Hierarchie an Ausnahmeklassen definieren



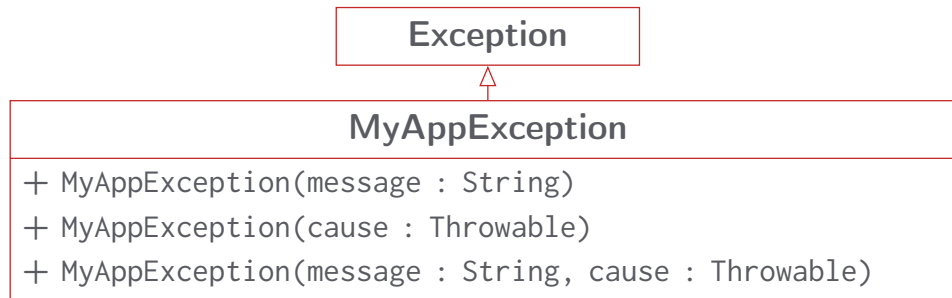
- ▶ `MyAppException` — **Basisklasse** für alle Ausnahmen der Anwendung
- ▶ Ableitungen für **Komponenten/spezielle Ausnahmen**

## Notizen



## Ableiten von (Runtime)Exception

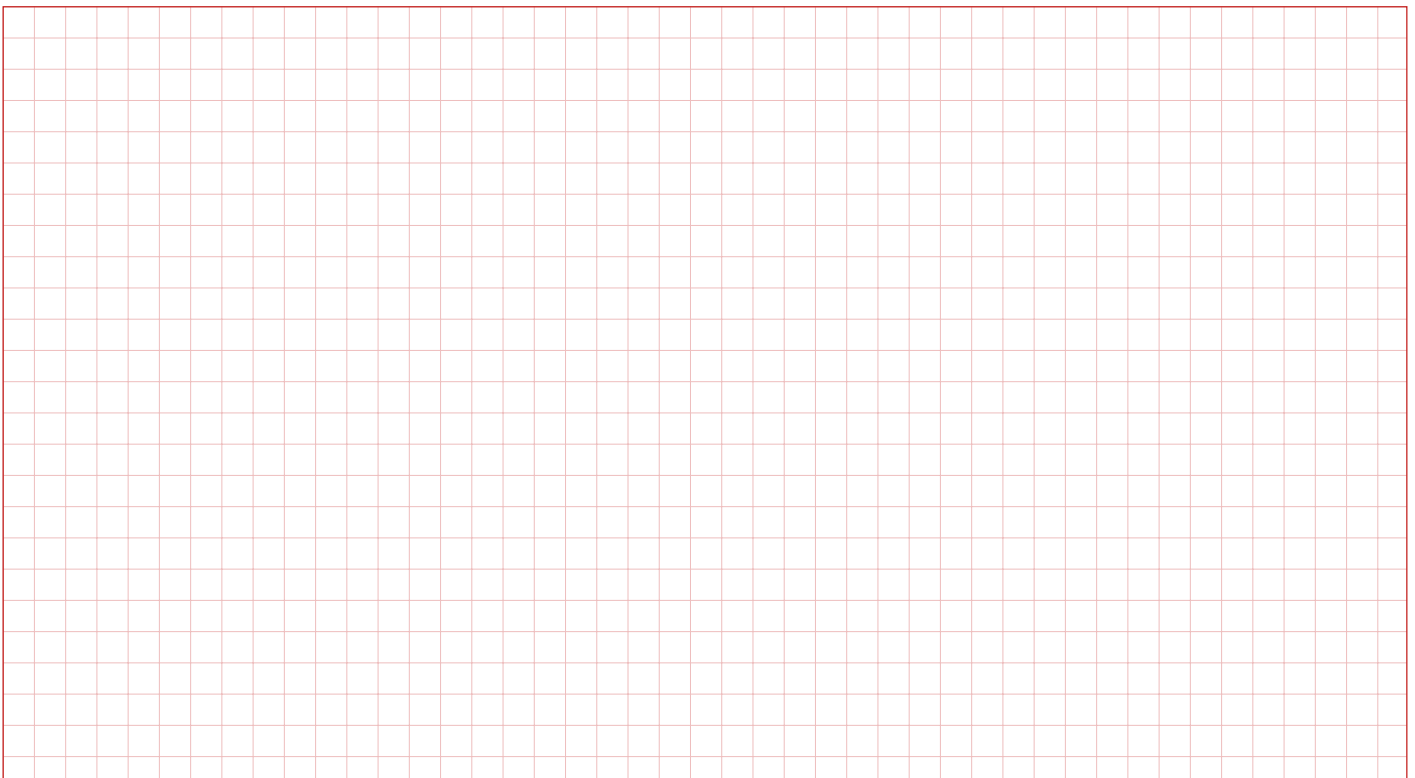
- ▶ Eigene Ausnahmeklassen sehen **immer ähnlich aus**



- ▶ **Ableiten** von (Runtime)Exception
- ▶ Implementieren der **drei obigen Konstruktoren**, z.B.

```
public MyAppException(String message, Throwable cause){
    super(message, cause);
}
```

## Notizen



# Inhalt

## Eigene Ausnahmen definieren

Eigene Ausnahmeklassen definieren

Beispiel: Consumable

Geprüfte oder ungeprüfte Ausnahmen

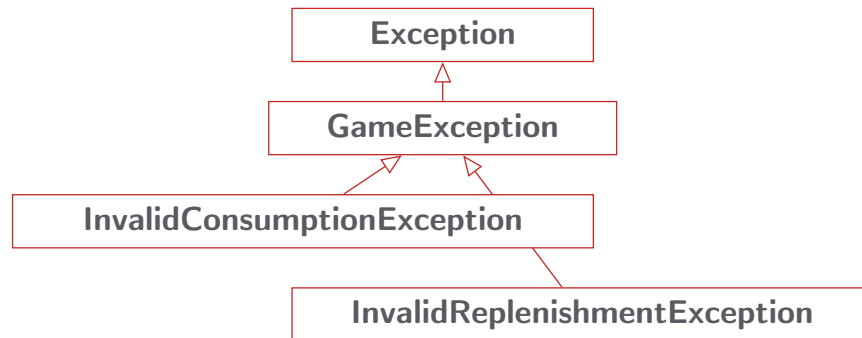
Zusammenfassung

## Notizen

A large rectangular area filled with a light gray grid pattern, intended for taking notes. The grid consists of small squares, approximately 20 columns wide and 30 rows high.

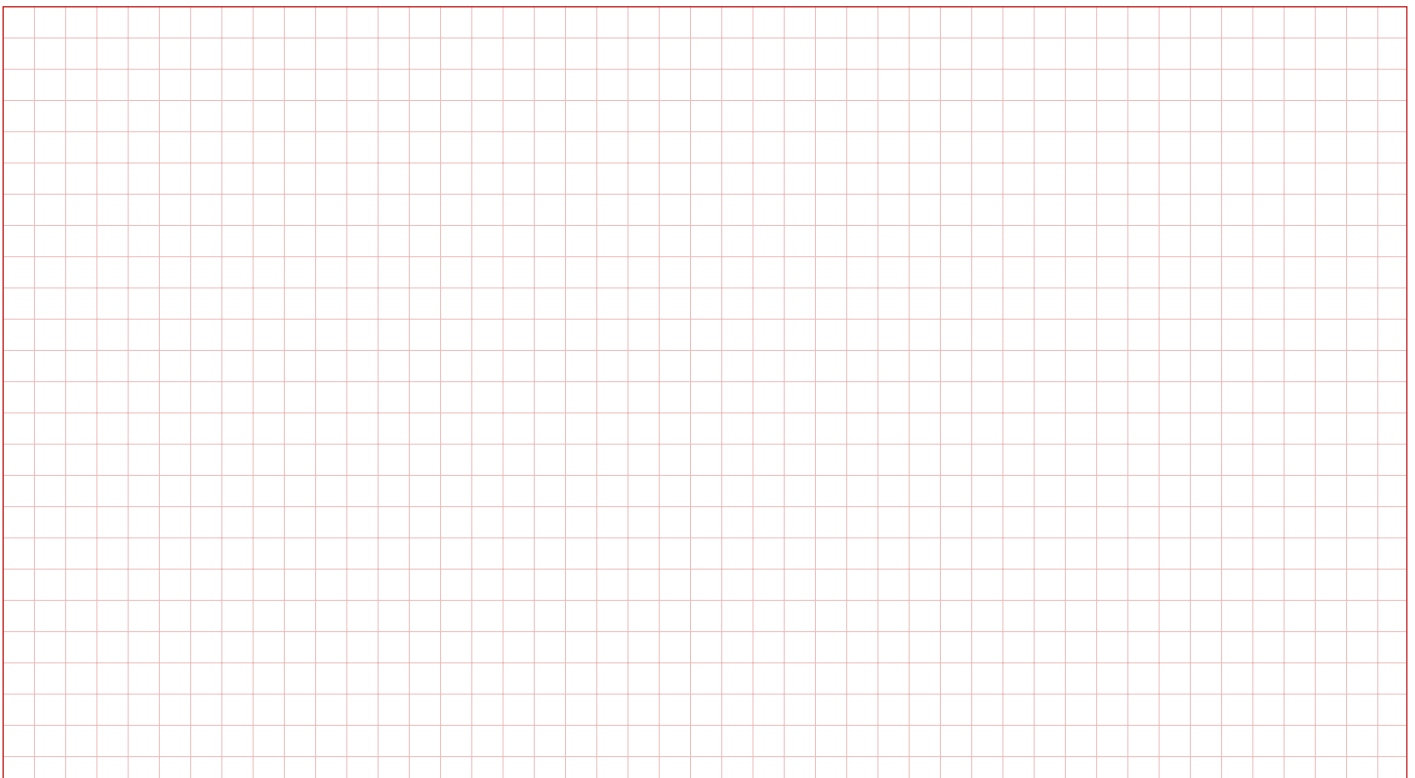
# Ausnahmen für unser Spiel

- ▶ Wir wollen Ausnahmen für unser Spiel **definieren**
  - ▶ **Basisklasse**: GameException
  - ▶ Für Ausnahmen in Consumable-Interface: InvalidConsumptionException
  - ▶ Für Ausnahmen in Replenishable-Interface: InvalidReplenishmentException
  - ▶ **Hierarchie**



- ▶ Noch **weitere Ausnahmen** für GameCharacter und Co. möglich

## Notizen



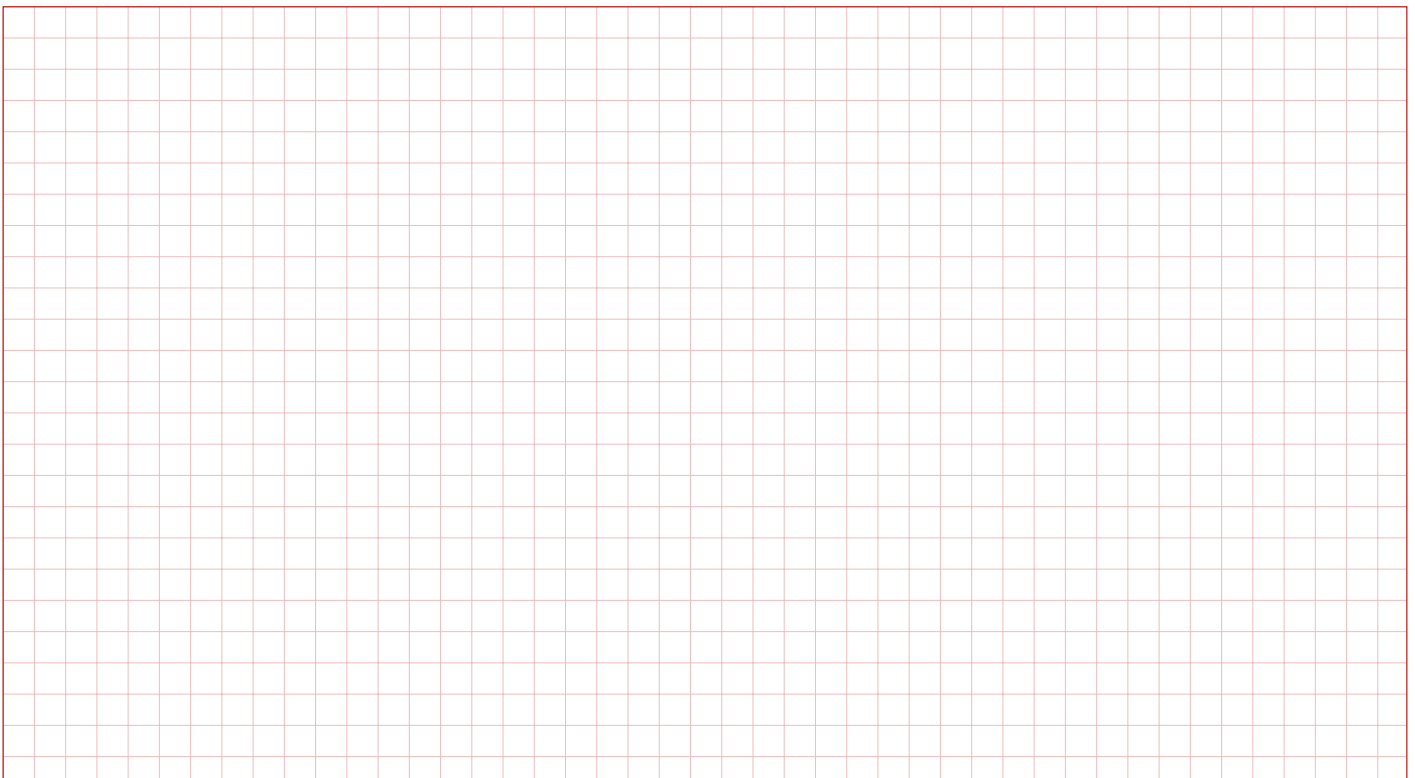
## Ausnahmen für unser Spiel

### ► GameException

```
4 public class GameException extends Exception{
6     private static final long serialVersionUID
7         = -3573120189683177891L;
9     public GameException(String message){ super(message); }
11    public GameException(Throwable cause){ super(cause); }
13    public GameException(String message, Throwable cause){
14        super(message, cause);
15    }
17 }
```

game/GameException.java

### Notizen



## Ausnahmen für unser Spiel

- ▶ Implementierung von `InvalidConsumptionException` und `InvalidReplenishmentException` **ähnlich** zu `GameException`

- ▶ `InvalidConsumptionException`

```
4 public class InvalidConsumptionException
5     extends GameException {
```

📄 `game/InvalidConsumptionException.java`

- ▶ In Implementierungen von **interface** `Consumable` erzeugt

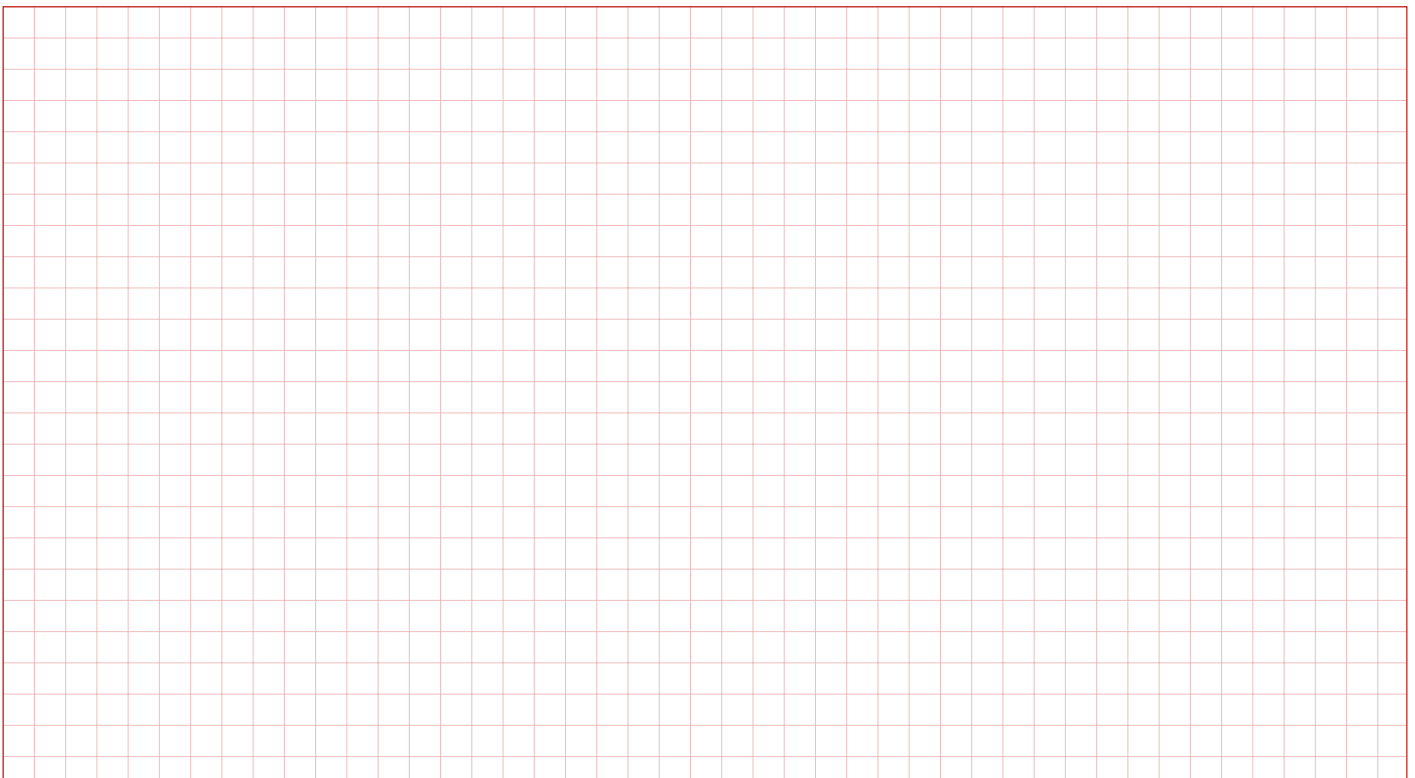
- ▶ `InvalidReplenishmentException`

```
5 public class InvalidReplenishmentException
6     extends GameException {
```

📄 `game/InvalidReplenishmentException.java`

- ▶ In Implementierungen **interface** `Replenishable` erzeugt

## Notizen





## Ausnahmen für unser Spiel

- Modifizierte Variante von Consumable

```
4 public interface Consumable {  
5     int unitsLeft();  
6     void consume(int n)  
7         throws InvalidConsumptionException;  
8 }
```

game/Consumable.java

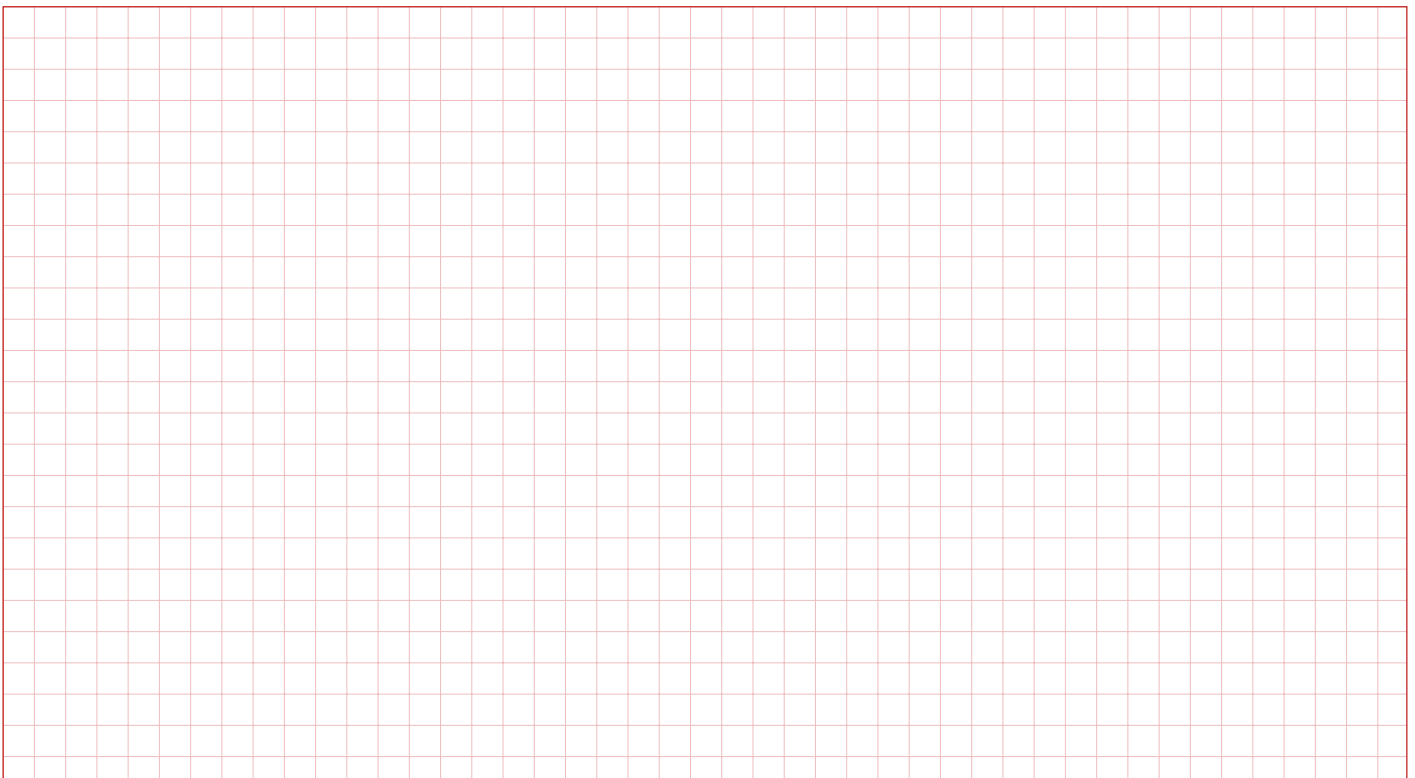
- Modifizierte Variante von Replenishable

```
5 public interface Replenishable{  
6     int maximumUnits();  
7     void replenish(int n)  
8         throws InvalidReplenishmentException;  
9 }
```

game/Replenishable.java

- **Beachte:** Auch **interface**-Methoden können **geworfene Ausnahmen** definieren

## Notizen



## Ausnahmen für unser Spiel

- ▶ Bottle implementiert Consumable und Replenishable
- ▶ Consumable.consume

```
35 public void consume(int n)
36     throws InvalidConsumptionException {
37     if (sipsLeft < n)
38         throw new InvalidConsumptionException("Empty!");
39     sipsLeft -= n;
40 }
```

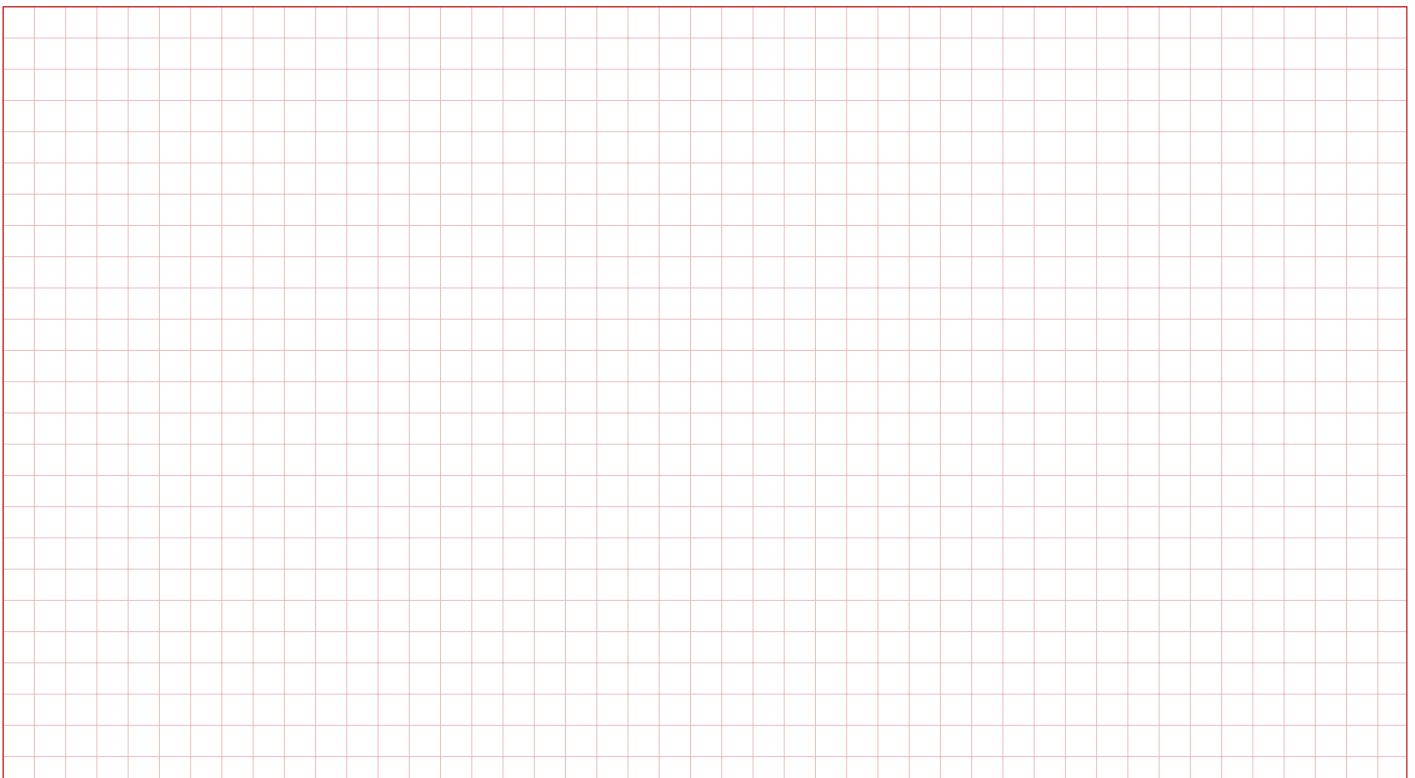
game/Bottle.java

- ▶ Replenishable.replenish

```
20 public void replenish(int n)
21     throws InvalidReplenishmentException {
22     if (sipsLeft + n > volume)
23         throw new InvalidReplenishmentException("Full!");
24     sipsLeft += n;
25 }
```

game/Bottle.java

## Notizen



## Ausnahmen für unser Spiel: Test


### ► Test von Bottle

```
Bottle bottle = new Bottle(0,100);  
bottle.consume(1); // FEHLER
```

„Unhandled exception InvalidConsumptionException“

### ► Zur Erinnerung: GameException **extends** Exception (geprüft)

### ► Nächster Versuch

```
18  runCustomExceptionExample  
19 try {  
20     Bottle bottle = new Bottle(0, 100);  
21     bottle.consume(1);  
22 } catch (InvalidConsumptionException exception){  
23     printInfo(exception);  
24 }
```

 CustomExceptionExamples.java


```
Type: InvalidConsumptionException  
Message: Empty!
```

## Notizen



## Ausnahmen für unser Spiel: Test

- ▶ Noch ein Test

```
30  runCustomExceptionExample2
31 try {
32     Bottle bottle = new Bottle(1, 100);
33     bottle.consume(1);
34     bottle.replenish(150);
35 } catch (GameException exception){
36     printInfo(exception);
37 }
```

 CustomExceptionExamples.java

Type: InvalidReplenishmentException  
Message: Full!

- ▶ **Beachte:** `catch` fängt `GameException`
- ▶ **Vorteil** einer Ausnahmen-Hierarchie
  - ▶ Basisklasse für **grobe Ausnahmebehandlung**
  - ▶ Unterklassen für **konkrete Ausnahmebehandlung**

## Notizen



# Inhalt

## Eigene Ausnahmen definieren

## Eigene Ausnahmeklassen definieren

## Geprüfte oder ungeprüfte Ausnahmen

## Notizen

## Geprüfte oder ungeprüfte Ausnahmen

```
try{
    bottle.consume(1);
} catch (InvalidConsumptionException exception){ ... }
```

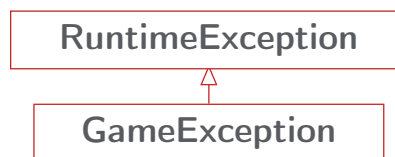
### ► Beobachtungen

- **Etwas lästig**: Aufrufe auf consume/replenish brauchen **try-catch**
- **Außerdem**: Ausnahmen könnten vermieden werden

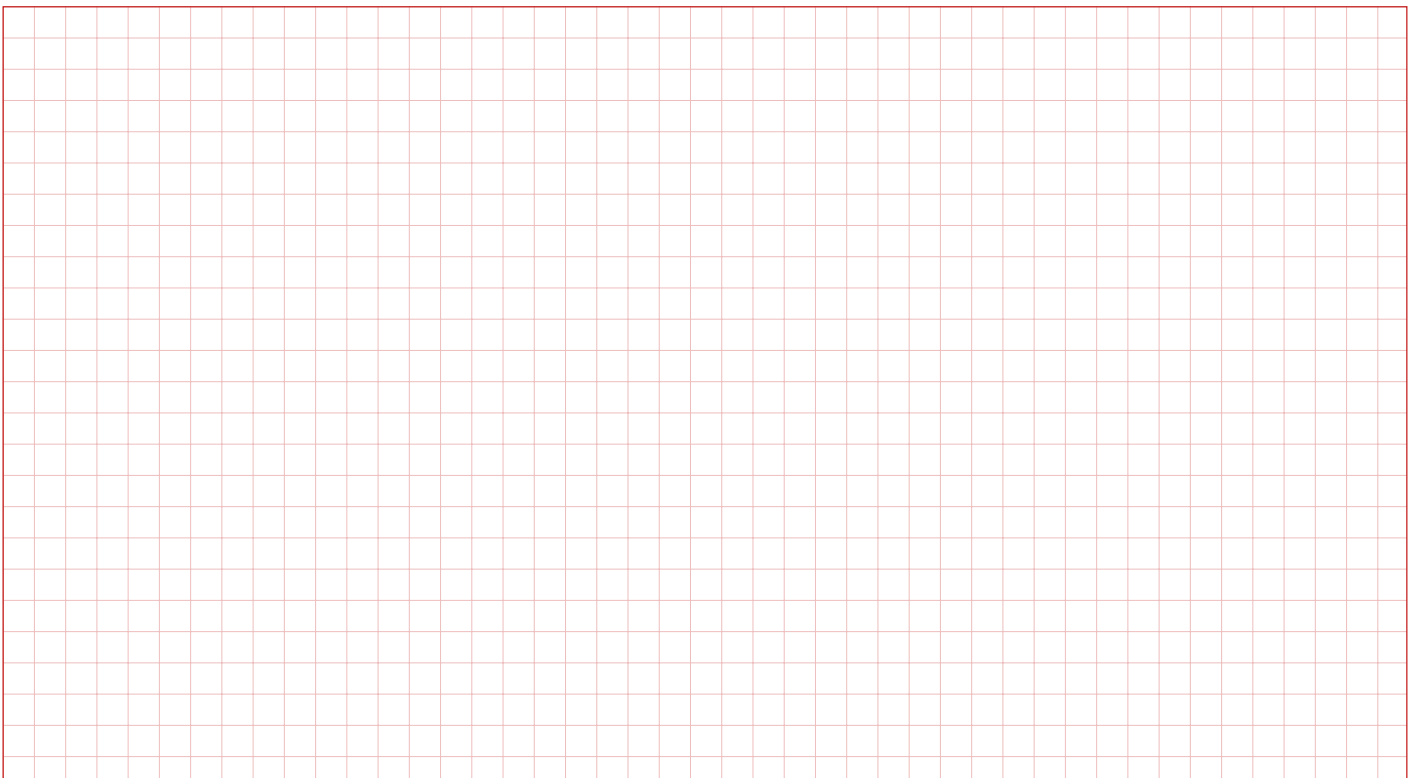
```
if (bottle.unitsLeft() >= 1)
    bottle.consume(1);
```

- Spricht für **ungeprüfte** Ausnahmen

- **Alternative**: GameException **extends** RuntimeException



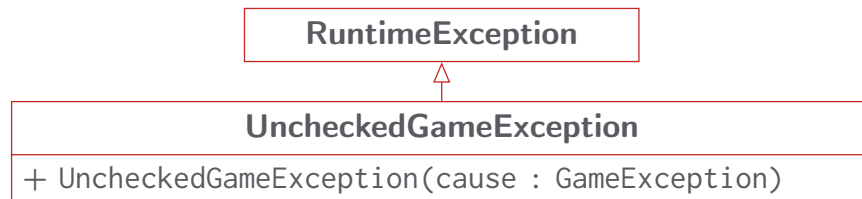
## Notizen



# Geprüfte oder ungeprüfte Ausnahmen

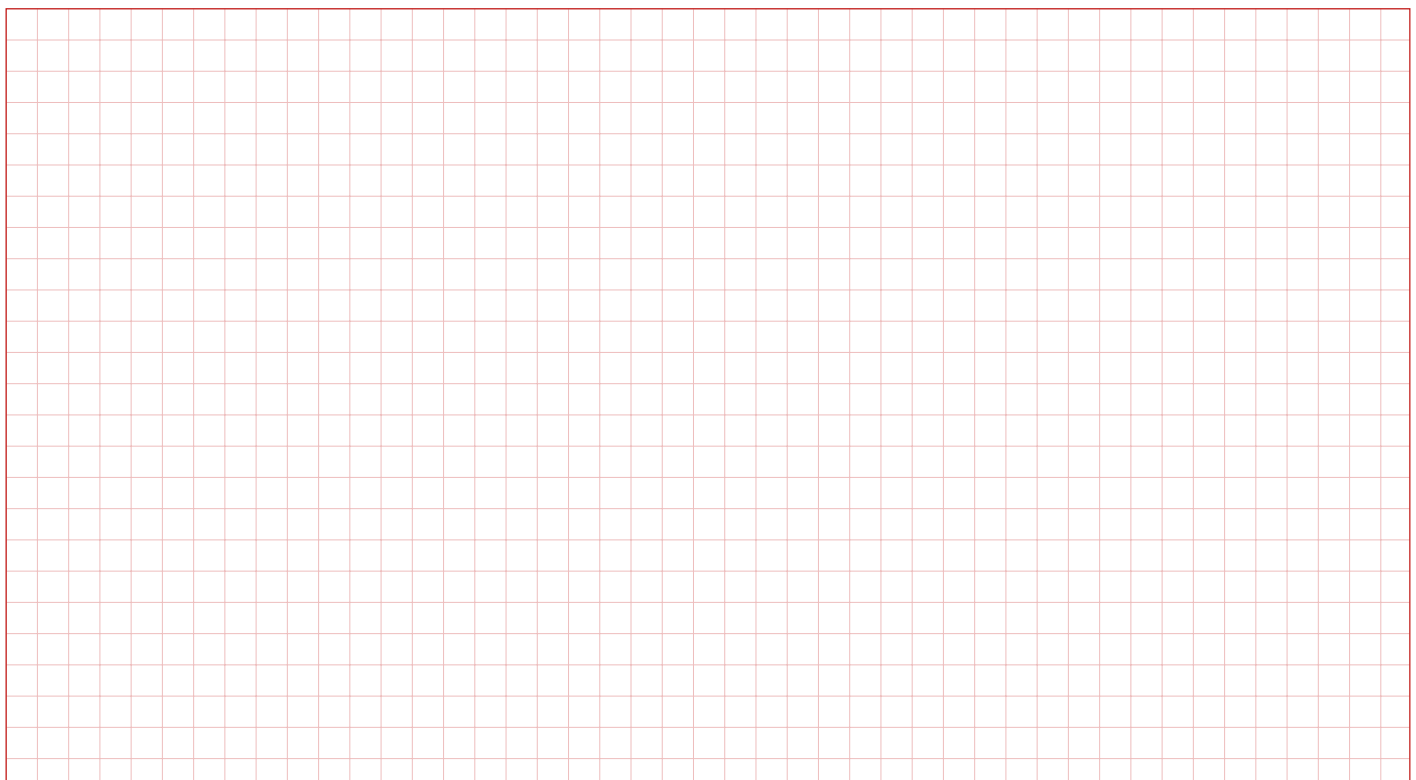
```
try{  
    bottle.consume(1);  
} catch (InvalidConsumptionException exception){ ... }
```

- ▶ Entscheidung ob **geprüft** oder **ungeprüft** muss für **Basisklasse** fallen
- ▶ Oder:
  - ▶ **Zwei Hierarchien** über zwei Basisklassen
  - ▶ „Konversion“ über Schachtelung



- ▶ (Tendenz in vielen Bibliotheken: **ungeprüfte Ausnahmen**)

## Notizen



# Inhalt

## Eigene Ausnahmen definieren

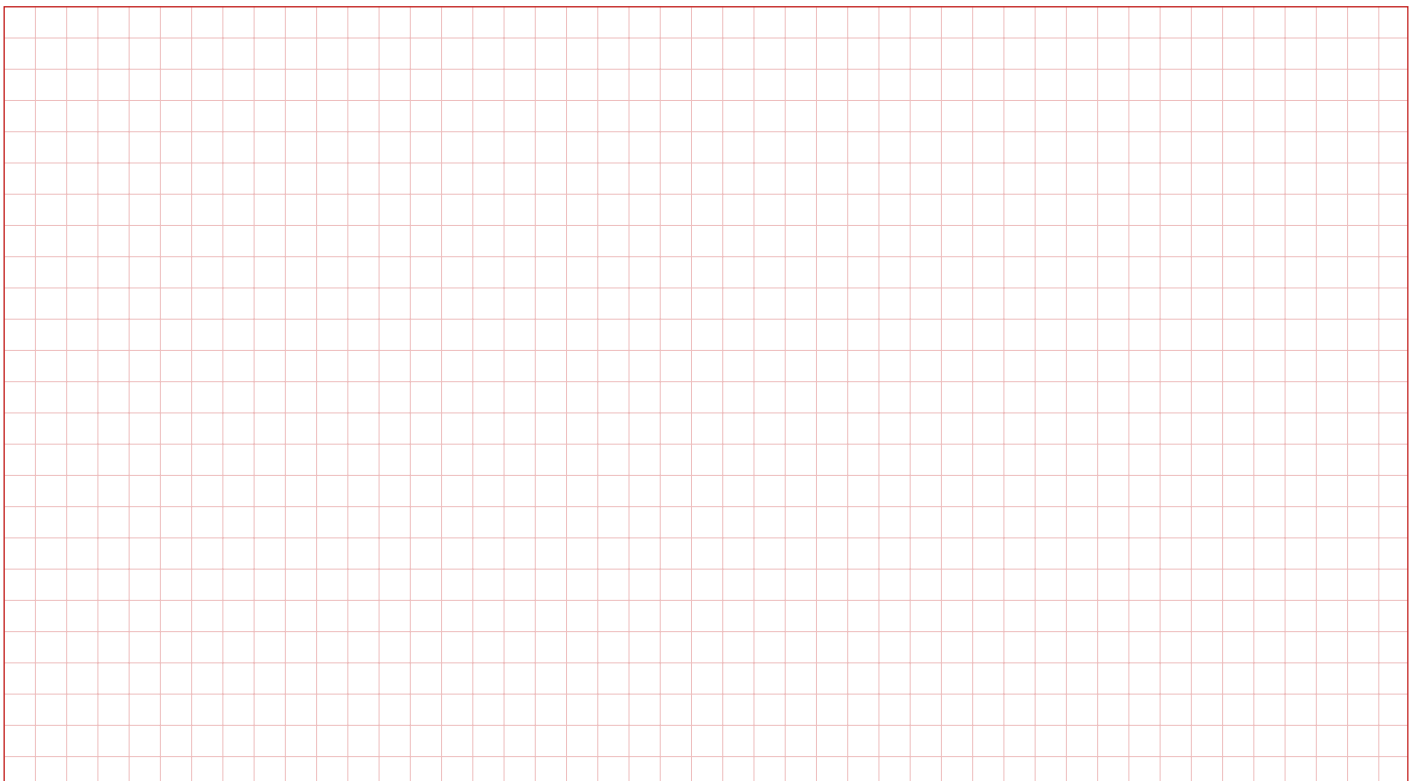
### Eigene Ausnahmeklassen definieren

Beispiel: Consumable

Geprüfte oder ungeprüfte Ausnahmen

Zusammenfassung

## Notizen

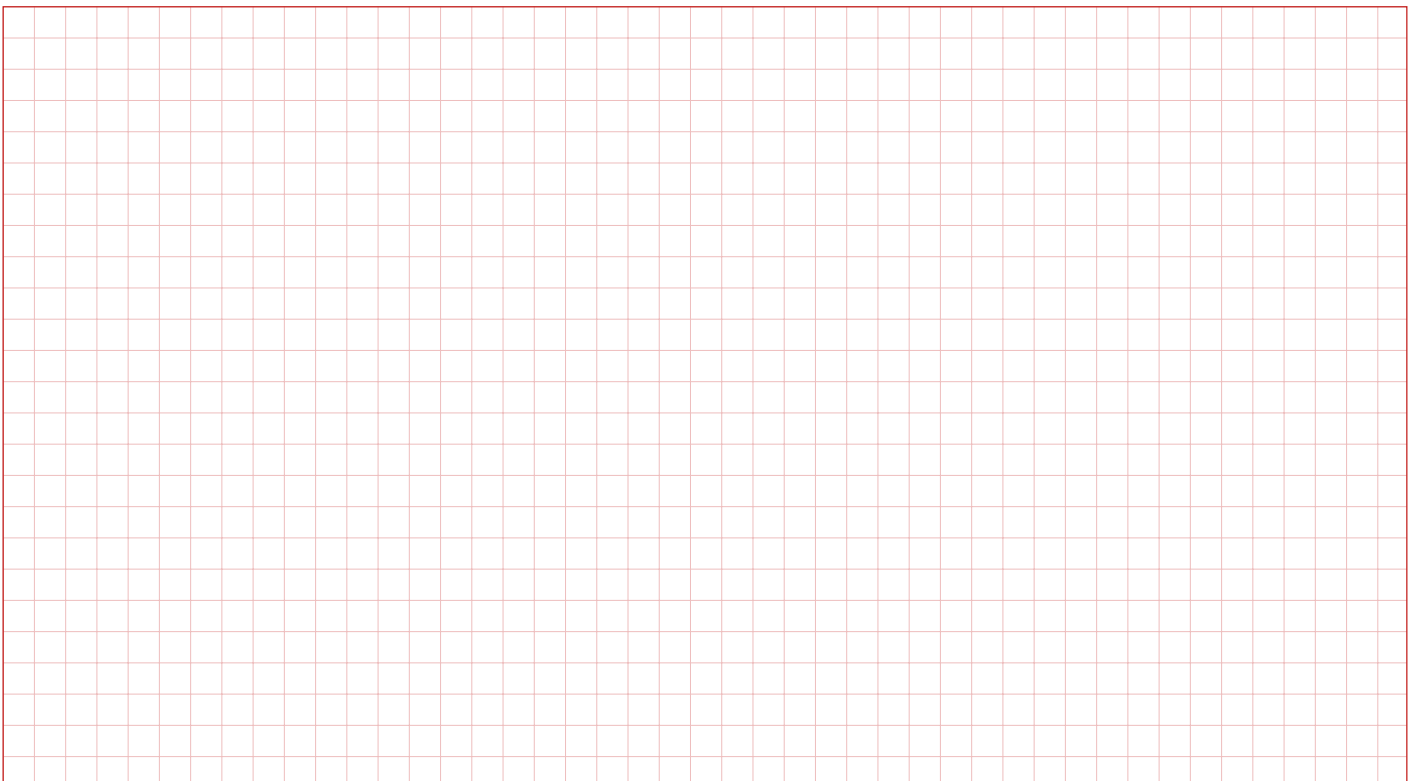




# Zusammenfassung

- ▶ Eigene Ausnahmeklassen ableiten von
  - ▶ `Exception` für geprüfte Ausnahmen
  - ▶ `RuntimeException` für ungeprüfte Ausnahmen
- ▶ Hierarchie von Ausnahmeklassen bilden
  - ▶ Basisklasse für Anwendung
  - ▶ Ableitungen für Komponenten/spezielle Ausnahmen
- ▶ Entscheidung geprüft oder ungeprüft
  - ▶ Für Basisklasse(n)
  - ▶ Oder: Geschachtelte Ausnahmen

## Notizen



# Inhalt

## Zusammenfassung

122

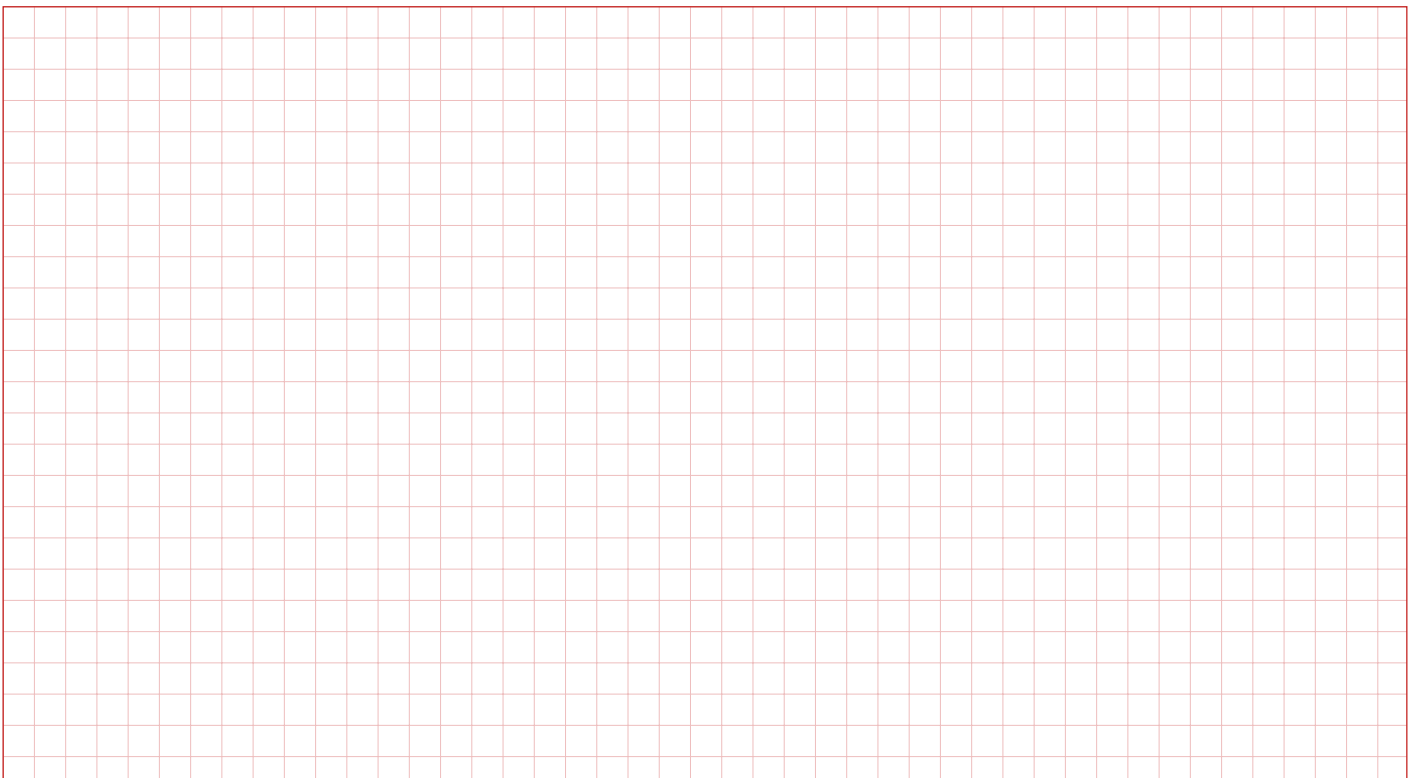
## Notizen

# Zusammenfassung

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

- ▶ **Ausnahmebehandlung** in Java
  - ▶ **try**-Block — Logik
  - ▶ **catch**-Block — Ausnahmebehandlung (**mehrfach**)
  - ▶ **finally**-Block — wird **immer zum Schluss** ausgeführt
- ▶ **Geprüfte Ausnahmen**
  - ▶ **Müssen** behandelt werden
  - ▶ Programm i.d.R. **fortführbar**
  - ▶ **Beispiel**: falsche Nutzereingabe
- ▶ **Ungeprüfte Ausnahmen**
  - ▶ **Müssen nicht** behandelt werden
  - ▶ Programm i.d.R. **nicht fortführbar**
  - ▶ **Beispiel**: Programmierfehler, Fehler in JVM

## Notizen



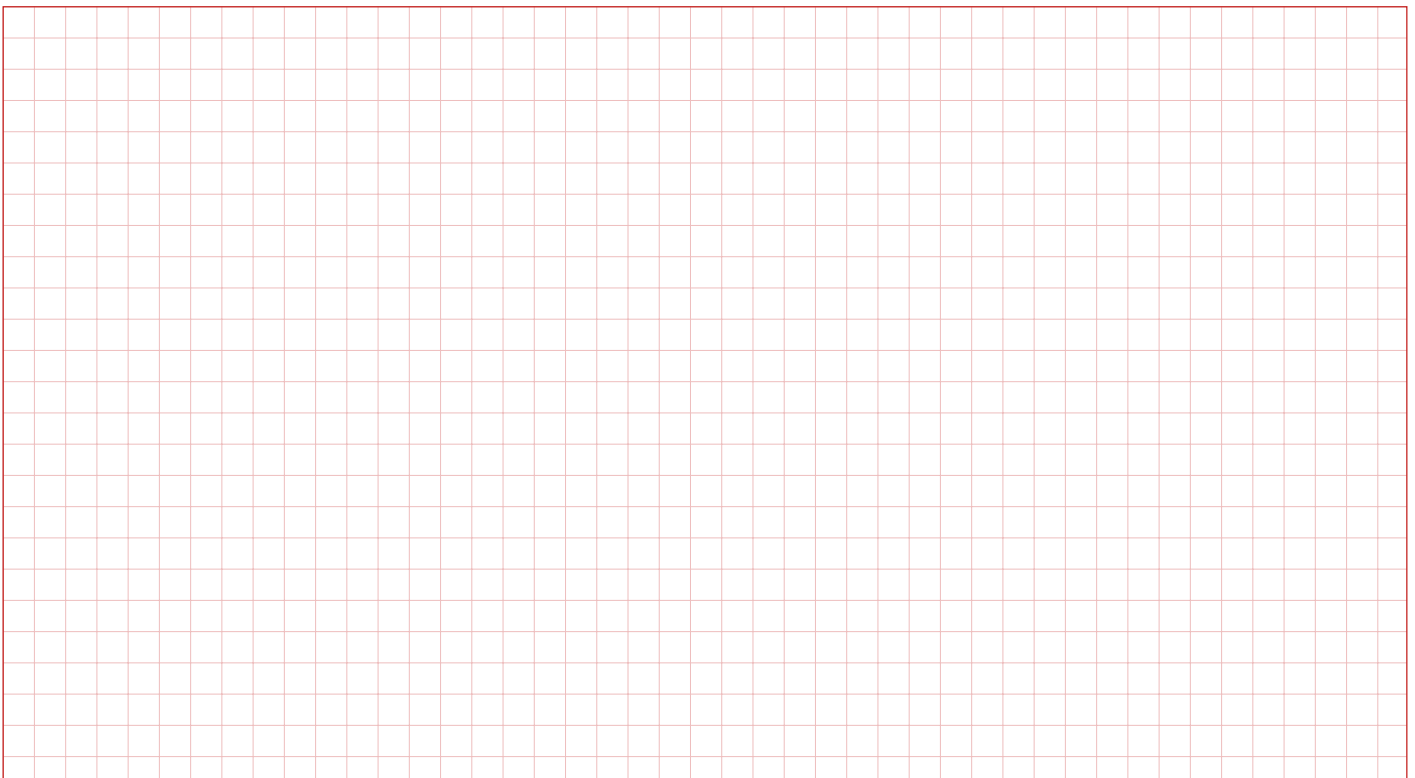
# Zusammenfassung

- ▶ **throws** deklariert geworfene Ausnahmen

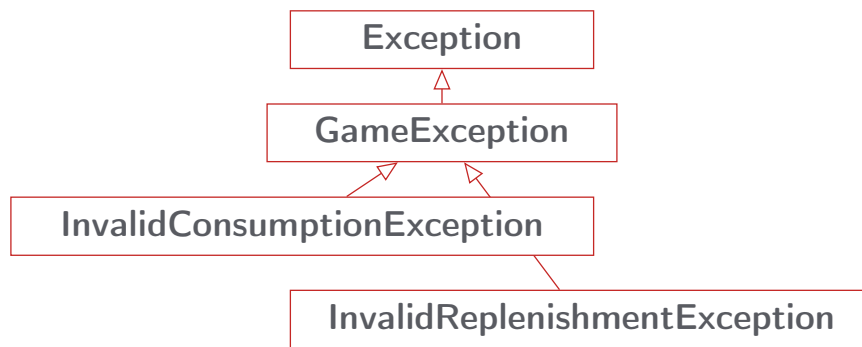
```
void doSomething() throws ImportantException
```

- ▶ Kein **try-catch** mehr nötig
- ▶ Auftretende Ausnahmen werden **weitergeleitet**
- ▶ **Überschreiben**: **Weglassen** und **spezialisieren** erlaubt
- ▶ Ausnahmen auslösen
  - ▶ Mit **throw new** ImportantException(...)
  - ▶ Nachricht muss **aussagekräftig** sein
  - ▶ **Geschachtelte** Ausnahmen: **throw new** NestingException(nested);
  - ▶ **Weiterleiten**: **throw** exception;

## Notizen



## Zusammenfassung



- ▶ JDK enthält viele **Standard-Ausnahmen**: diese nutzen!
- ▶ Für **eigene Anwendungen**
  - ▶ **Ausnahmen-Hierarchie** definieren
  - ▶ **Basisklasse(n)**
  - ▶ Design-Entscheidung: **geprüft** oder **ungeprüft**

## Notizen