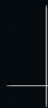


Ausnahmen in Java



Gliederung

- 1) Ausnahmen mit try und catch
- 2) geprüfte Ausnahmen
 - a) try-catch-Behandlung
 - b) throws im Methodenkopf angeben
- 3) ungeprüfte Ausnahmen
- 4) Ausnahmen auffangen
 - a) leere catch-Blöcke
 - b) Wiederholung abgebrochener Bereiche
 - c) Abschlussbehandlung mit finally
- 5) Klassenhierarchie der Ausnahmen
 - a) Eigenschaften des Exception-Objekts
 - b) Basistyp Throwable
- 6) try mit Ressourcen

1) Ausnahmen mit try und catch

```
try {  
    // Programmcode, der eine Ausnahme ausführen kann  
}  
catch ( ... ) {  
    // Programmcode zum Behandeln der Ausnahme  
}  
// Es geht ganz normal weiter, denn die Ausnahme wurde behandelt
```

2) geprüfte Ausnahmen

```
Scanner scanner = new Scanner (System.in);  
double zahl = scanner.nextDouble();
```

- kann InputMismatchException auslösen
- geprüfte Ausnahme
- zwei Möglichkeiten, den Fehler zu behandeln
 - try-catch
 - throws

2) geprüfte Ausnahmen

a) try-catch-Behandlung

- try-Anweisung gefolgt von einem *try-Block*
- Kombination mit *catch-Klausel*
- `catch (InputMismatchException e)` **deklariert *Exception-Handler***
- **fängt alles auf, was vom Typ `InputMismatchException` ist**
- nach dem Auffangen geht das Programm ganz normal weiter

2) geprüfte Ausnahmen

```
try {  
    Scanner scanner = new Scanner (System.in);  
    zahl = scanner.nextDouble();  
}  
catch (InputMismatchException e) {  
    System.err.println("Leider wurde keine Zahl eingegeben.")  
}
```

2) geprüfte Ausnahmen

b) throws im Methodenkopf angeben

- try-Block mit angehängtem catch-Block: **Auffangen** problematischer Blöcke
- throws im Methodenkopf angeben: **Weiterleiten** der Ausnahme an Aufrufer
- Methode zeigt an, dass sie bestimmte Ausnahmen nicht selbst behandelt
- tritt Ausnahme auf, wird Methode abgebrochen
- Aufrufer muss sich um Ausnahme kümmern

2) geprüfte Ausnahmen

```
public static double eingabe() throws InputMismatchException {  
    Scanner scanner = new Scanner (System.in);  
    double zahl = scanner.nextDouble();  
    return zahl;  
}
```


2) geprüfte Ausnahmen

```
public static void main( String[] args ) {  
    try {  
        eingabe()  
    }  
    catch ( InputMismatchException e ) {  
        System.err.println("Leider wurde keine Zahl eingegeben");  
    }  
}
```

3) ungeprüfte Ausnahmen

- RuntimeException wird zum Beispiel ausgelöst, wenn...
 - ...ganzzahlige Division durch 0 versucht wird
 - ...ungültige Indexwerte beim Zugriff auf Arrays angegeben werden
- Denkfehler des Programmierers
- Programm sollte normalerweise Auffangen und Behandeln nicht versuchen
- RuntimeException Unterklasse von Exception zeigt Programmierfehler auf, die behoben werden müssen

3) ungeprüfte Ausnahmen

Unterklasse von RuntimeException	Was den Fehler auslöst
ArithmeticException	ganzzahlige Division durch 0
ArrayIndexOutOfBoundsException	Indexgrenzen werden missachtet
ClassCastException	Typumwandlung nicht möglich
EmptyStackException	Stapelspeicher ist leer
IllegalArgumentException	Methoden melden falsche Argumente
IllegalMonitorStateException	Thread möchte warten, hat aber den Monitor nicht
NullPointerException	Meldet einen der häufigsten Programmierfehler
UnsupportedOperationException	Operationen sind nicht gestattet

4) Ausnahmen auffangen

a) leere catch-Blöcke

- Ausnahmen müssen von einem catch-Block oder per throw-Anweisung gefangen bzw. nach oben weitergeleitet werden
- catch-Block kann sinnvolle Behandlung beinhalten oder leer sein
- leerer catch-Block unterdrückt die Ausnahme „heimlich“
- mindestens `System.err.println(e)` oder Loggen der Ausnahme

4) Ausnahmen auffangen

b) Wiederholung abgebrochener Bereiche

- bisher keine von Java unterstützte Möglichkeit, an den Punkt zurückzukehren, der die Ausnahme ausgelöst hat
- im Fall einer fehlerhaften Eingabe Wiederholung erwünscht
- while-Schleife als Behelfslösung

4) Ausnahmen auffangen

```
while ( true ) {  
    try {  
        Scanner scanner = new Scanner (System.in);  
        zahl = scanner.nextDouble();  
        break;  
    }  
    catch (InputMismatchException e) {  
        System.err.println("Leider wurde keine Zahl eingegeben.")  
    }  
}
```


4) Ausnahmen auffangen

c) Abschlussbehandlung mit finally

- optimale Behandlung von Ausnahmen mit finally
- finally wird immer ausgeführt – mit oder ohne Ausnahme zuvor
- *TCFTC* (try-catch-finally-try-catch) → nicht schön!

4) Ausnahmen auffangen

```
try {  
    RandomAccessFile f = new RandomAccessFile("file.gif", "r");  
    f.seek(6);  
    System.out.printf("%s x %s Pixel%n", f.read() + f.read() * 256, f.read() + f.read() * 256);  
    f.close();  
}  
  
catch (FileNotFoundException e) {  
    System.err.println("Datei ist nicht vorhanden!");  
}  
  
catch (IOException e) {  
    System.err.println("Allgemeiner Ein-/Ausgabefehler!");  
}
```



Datenstrom wird eventuell
nicht geschlossen!

4) Ausnahmen auffangen

```
RandomAccessFile f = null;
```

```
try {
```

```
    f = new RandomAccessFile("file.gif", "r");
```

```
    f.seek(6);
```

```
    System.out.printf("%s x %s Pixel%n", f.read() + f.read() * 256, f.read() + f.read() * 256);
```

```
} catch (FileNotFoundException e) {
```

```
    System.err.println("Datei ist nicht vorhanden!");
```

```
} catch (IOException e) {
```

```
    System.err.println("Allgemeiner Ein-/Ausgabefehler!");
```

```
} finally {
```

```
    if ( f != null )
```

```
        try { f.close(); } catch (IOException e) { }
```

```
}
```

lokale Variable mit sehr
großem Radius

in eigenem try-catch-Block
→ TCFTC

5) Klassenhierarchie der Ausnahmen

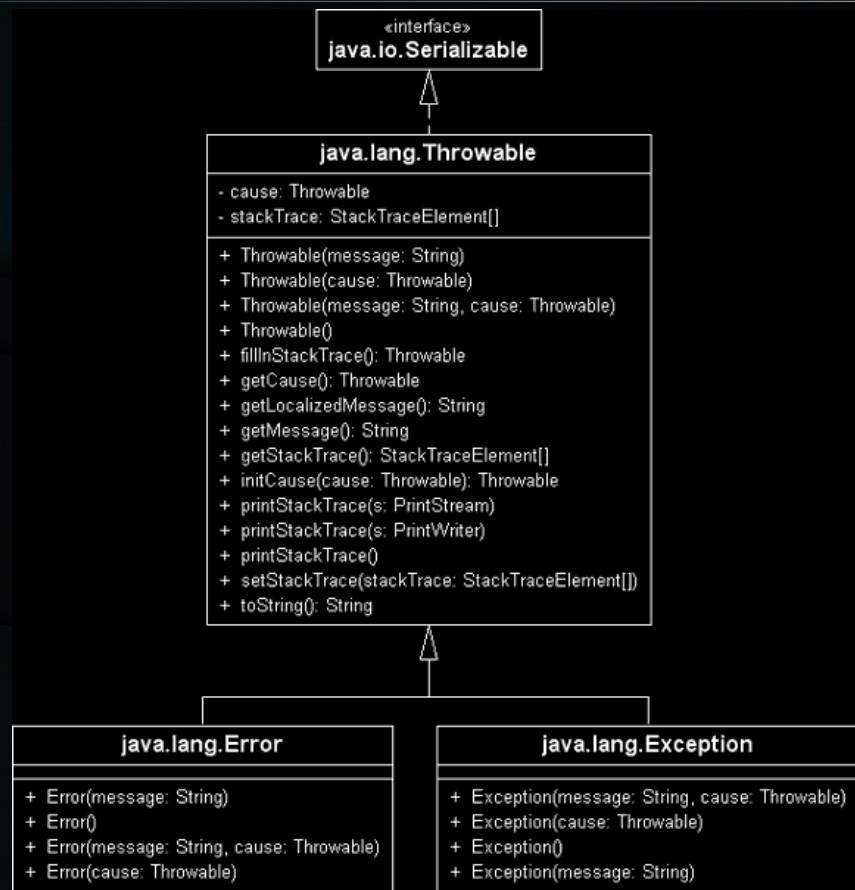
a) Eigenschaften des Exception-Objekts

- Ausnahme ist Objekt vom Typ einer Klasse
- Exception-Klasse ist von anderen Ausnahmeklassen abgeleitet
- Exception-Objekt ist reich an Informationen
 - Art der Ausnahme
 - Fehlernachricht
- Abfrage und Ausgabe mit `printStackTrace()`

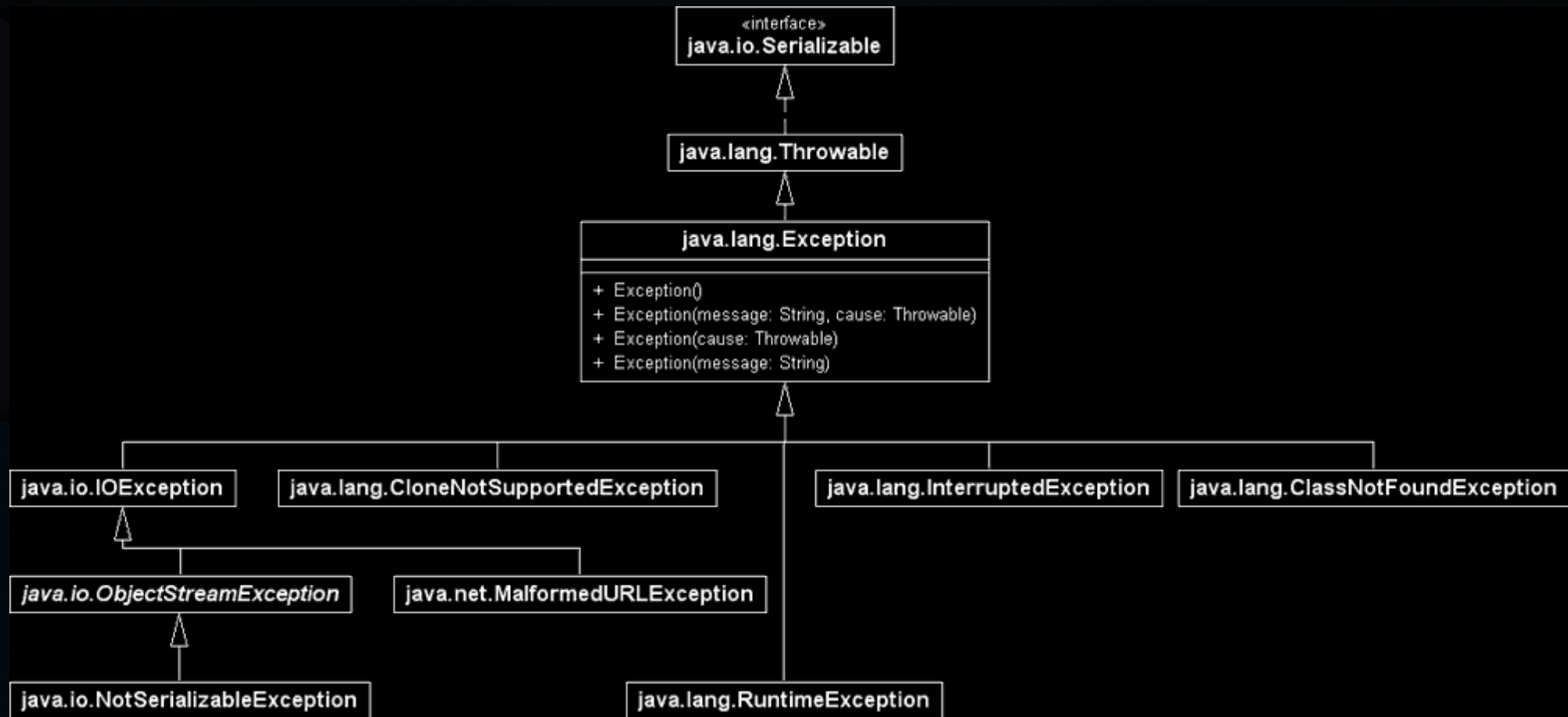
5) Klassenhierarchie der Ausnahmen

b) Basistyp Throwable

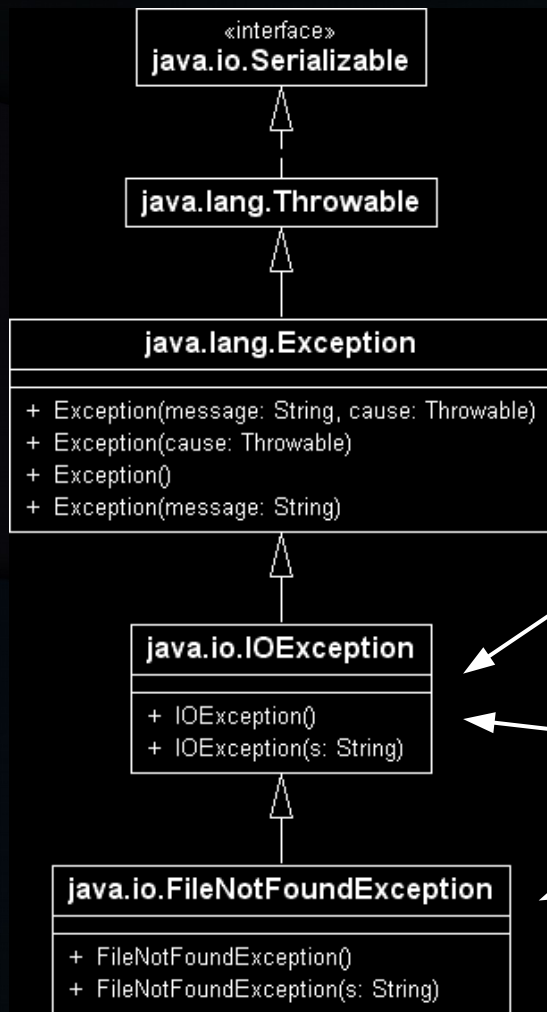
- jede Ausnahme direkt oder indirekt von `java.lang.Throwable` **abgeleitet**
- Throwable **vererbt viele nützliche Methoden** an `java.lang.Error` und an `java.lang.Exception`
- Error **behandelt schwerwiegende Ausnahmen**
- im Folgenden ausschließlich Betrachtung von `Exception` und ihre Unterklassen



5) Klassenhierarchie der Ausnahmen



5) Klassenhierarchie der Ausnahmen



es ist problematisch,
alles mit Exception
abzufangen

es reicht aus, eine
FileNotFoundException
mit einer IOException
aufzufangen

6) try mit Ressourcen

- Garbage-Collector bereinigt ausschließlich Speicher automatisch
- Dateien, Datenbankverbindungen (uvm.) müssen auch geschlossen werden, um Ressourcen freizugeben
- try-catch-finally gibt (mit viel Quellcode) Ressourcen frei, allerdings
 - Variablen müssen außerhalb vom try-Block deklariert werden, um für finally zugänglich zu sein
 - zusätzliches try-catch nötig, um Ressource zu schließen

6) try mit Ressourcen

- Garbage-Collector bereinigt ausschließlich Speicher automatisch
- Dateien, Datenbankverbindungen (uvm.) müssen auch geschlossen werden, um Ressourcen freizugeben
- try-catch-finally gibt (mit viel Quellcode) Ressourcen frei, allerdings
 - Variablen müssen außerhalb vom try-Block deklariert werden, um für finally zugänglich zu sein
 - zusätzliches try-catch nötig, um Ressource zu schließen

6) try mit Ressourcen

- *Ressourcentypen*, die `java.lang.AutoCloseable` implementieren, lassen sich vereinfacht schließen: *try mit Ressourcen*
- dient dem *Automatic Resource Management*, daher auch *ARM-Block*
- Ein-/Ausgabeklassen wie `Scanner`, `InputStream` und `Writer`
- erweiterte Syntax mit Ressourcenspezifikation: `try (...) {...}` statt `try {...}`

6) try mit Ressourcen

- ausgeschriebene Implementierung:

```
InputStream in = ClassLoader.getResourceAsStream ( "datei.txt" );  
{  
    Scanner res = new Scanner ( System.in );  
    try { System.out.println( res.nextLine() ) }  
    finally { res.close(); }  
}
```

6) try mit Ressourcen

- try mit Ressourcen

```
InputStream in = ClassLoader.getResourceAsStream ( "datei.txt" );  
try ( Scanner res = new Scanner ( System.in ) ) {  
    System.out.println( res.nextLine() );  
}
```

6) try mit Ressourcen

- in den runden Klammern wird Ressource spezifiziert, die später automatisch geschlossen wird
- direkte Initialisierung von Variablen, etwa mit einem Konstruktor- oder Methodenaufruf
- lokale AutoCloseable-Variable ist nur in dem Block gültig
- AutoCloseable-Typen:
 - hauptsächlich im java.io-Paket (Reader, Writer, ...)
 - einige aus java.sql-Paket

Quellen

https://openbook.rheinwerk-verlag.de/javainsel/07_001.html#u7.1

https://openbook.rheinwerk-verlag.de/javainsel/07_002.html#u7.2

https://openbook.rheinwerk-verlag.de/javainsel/07_003.html#u7.3

https://openbook.rheinwerk-verlag.de/javainsel/07_006.html#u7.6