

## 6. Java Fehler und Ausnahmen

Fehler, Systemausnahmen, Benutzerausnahmen, Behandeln von Ausnahmen, Spezialfall Ressourcen

## Fehler und Ausnahmen in Java

Fehler und Ausnahmen unterbrechen die normale Programmausführung abrupt und stellen ein *nicht geplantes Ereignis* dar.



Ausnahmen sind böse, oder doch nicht?

- Java ermöglicht es, solche Ereignisse abzufangen und zu behandeln (als Alternative zum Programmabsturz).
- Nicht behandelte Fehler und Ausnahmen werden durch den Aufrufstapel hochgereicht.

129

130

## Fehler (Errors)



Fehler treten in der virtuellen Maschine von Java auf und sind *nicht reparierbar*.

Beispiele

- Kein Speicher mehr verfügbar
- Zu hoher Aufrufstapel (→ Rekursion)
- Fehlende Programmbibliotheken
- Bug in der virtuellen Maschine
- Hardwarefehler

Hier ist nichts mehr zu machen

## Ausnahmen (Exceptions)

Ausnahmen werden von der virtuellen Maschine oder vom Programm selbst ausgelöst und können meist behandelt werden um die *Normalsituation wiederherzustellen*



Aufwischen und neu einschenken

Beispiele

- Dereferenzierung von `null`
- Division durch 0
- Schreib/Lesefehler (Dateien)
- Businesslogik Fehler

131

132

## Arten von Ausnahmen

### Systemausnahmen (runtime exceptions)

- Können überall auftreten
- *Können* behandelt werden
- Ursache: Bug im Programm

### Benutzerausnahmen (checked exceptions)

- Müssen deklariert werden
- *Müssen* behandelt werden
- Ursache: Unwahrscheinliches, aber prinzipiell mögliches Ereignis

## Beispiel einer Systemausnahme

```
1 import java.util.Scanner;
2 class ReadTest {
3     public static void main(String[] args){
4         int i = readInt("Number");
5     }
6     private static int readInt(String prompt){
7         System.out.print(prompt + ": ");
8         Scanner input = new Scanner(System.in);
9         return input.nextInt();
10    }
11 }
```

Eingabe: Number: asdf

133

134

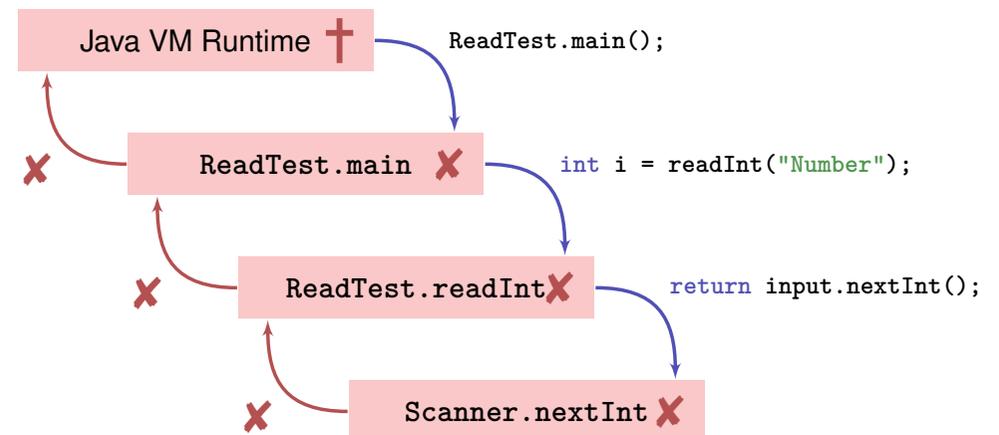
## Nicht behandelte Fehler und Ausnahmen

Das Programm stürzt ab und hinterlässt auf der Konsole eine "Aufrufstapelzurückverfolgung" 😊 (ab jetzt: *Stacktrace*). Darin sehen wir, wo genau das Programm abgebrochen wurde.

```
Exception in thread "main" java.util.InputMismatchException
    [...]
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at ReadTest.readInt(ReadTest.java:9)
    at ReadTest.main(ReadTest.java:4)
```

⇒ Forensische Nachforschungen mit Hilfe dieser Information.

## Ausnahme propagiert durch Aufrufstapel



135

136

## Stacktraces verstehen

Ausgabe:

```
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextInt(Scanner.java:2117)
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at ReadTest.readInt(ReadTest.java:9)
    at ReadTest.main(ReadTest.java:4)
```

*Eine unpassende Eingabe ...*

*... in Methode `readInt` auf Zeile 9 ...*

*... aufgerufen durch Methode `main` auf Zeile 4.*

137

## Stacktraces verstehen

```
1 import java.util.Scanner;
2 class ReadTest {
3     public static void main(String[] args){
4         int i = readInt("Number");
5     }
6     private static int readInt(String prompt){
7         System.out.print(prompt + ": ");
8         Scanner input = new Scanner(System.in);
9         return input.nextInt();
10    }
11 }
```

at ReadTest.readInt(ReadTest.java:9)  
at ReadTest.main(ReadTest.java:4)

138

## Systemausnahme: Bug im Programm?!

Wo ist der Fehler?

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    return input.nextInt();
}
```

*Nicht garantiert, dass als nächstes ein `int` anliegt.*

⇒ Die Scanner Klasse bietet ein Test dafür an

139

## Systemausnahme: Bugfix!

Erst prüfen!

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    if (input.hasNextInt()){
        return input.nextInt();
    } else {
        return 0; // or do something else ...?!
    }
}
```

140

## Erste Erkenntnis: Oft keine Ausnahmesituation

Oft sind die "Sonderfälle" gar kein besonderes Ereignis, sondern absehbar. Hier sollten *keine* Ausnahmen verwendet werden!



Kinder kippen Becher um.  
Man gewöhnt sich daran.

Beispiele

- Falsche Credentials beim Einloggen
- Leere Pflichtfelder in Eingabemasken
- Nicht verfügbare Internet-Ressourcen
- Timeouts

## Zweite Erkenntnis: Ausnahmen verhindern

Statt eine Systemausnahme abzuwarten *aktiv verhindern*, dass diese überhaupt auftreten kann.



Problem gelöst.

Beispiele

- Usereingaben frühzeitig prüfen
- Optionale Typen verwenden
- Timeout Situationen voraussehen
- Plan B für nicht verfügbare Ressourcen

## Arten von Ausnahmen

**Systemausnahmen**  
(runtime exceptions)

- Können überall auftreten
- *Können* behandelt werden
- Ursache: Bug im Programm

**Benutzerausnahmen**  
(checked exceptions)

- Müssen deklariert werden
- *Müssen* behandelt werden
- Ursache: Unwahrscheinliches, aber prinzipiell mögliches Ereignis

## Beispiel einer Benutzerausnahme

```
private static String[] readFile(String filename){
    FileReader fr = new FileReader(filename);
    BufferedReader bufr = new BufferedReader(fr);
    ...
    line = bufr.readLine();
    ...
}
```

**Compiler Fehler:**

```
./Root/Main.java:9: error: unreported exception FileNotFoundException; must be caught or declared to be
    FileReader fr = new FileReader(filename);
    ~~~~~
./Root/Main.java:11: error: unreported exception IOException; must be caught or declared to be thrown
    String line = bufr.readLine();
    ~~~~~
```

2 errors

## Kurzer Blick in die Javadoc

### readLine

```
public String readLine()
    throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

#### Returns:

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

#### Throws:

IOException - If an I/O error occurs

#### See Also:

Files.readAllLines(java.nio.file.Path, java.nio.charset.Charset)

145

## Warum eine Benutzerausnahme?

Folgende Situationen rechtfertigen Benutzerausnahmen:

- Fehlerfall ist *unwahrscheinlich aber prinzipiell möglich* – und kann durch geeignete Massnahmen zur Laufzeit behoben werden können.

Der Aufrufer einer Methode mit einer deklarierten Benutzerausnahme wird gezwungen, sich damit zu beschäftigen – behandeln oder weiterreichen.

146

## Behandeln von Ausnahmen

```
private static String[] readFile(String filename){
    try{
        FileReader fr = new FileReader(filename);
        BufferedReader bufr = new BufferedReader(fr);
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    } finally {
        // close resources
    }
}
```

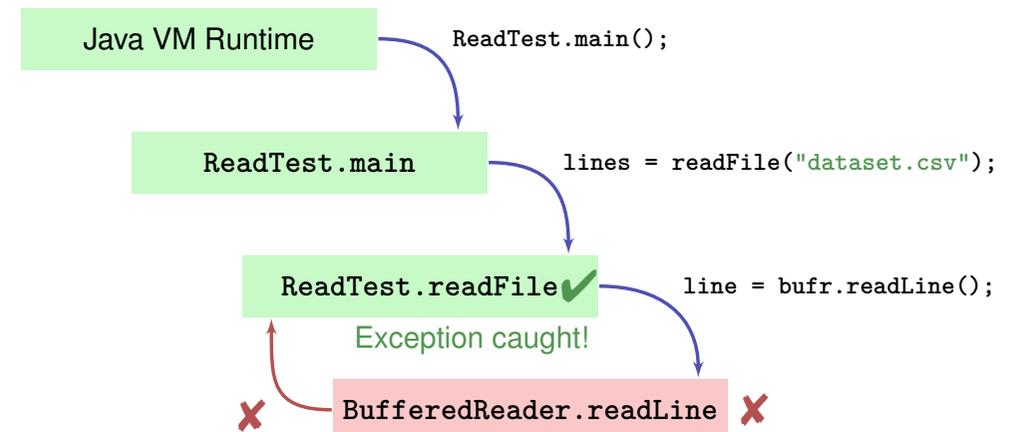
*Geschützter Bereich*

*Massnahmen zur Wiederherstellung der Normalsituation*

*Wird in jedem Fall am Schluss ausgeführt, immer!*

147

## Behandlung von Ausnahmen: Propagieren stoppen!



148

## Finally: Ressourcen schliessen!

In Java müssen *Ressourcen* unbedingt geschlossen werden nach Gebrauch. Ansonsten wird Speicher nicht freigegeben.

Ressourcen:

- Dateien
- Datenströme
- GUI Elemente
- ...



149

## Try-With-Resources Anweisung

Spezifische Syntax, um Ressourcen *automatisch* zu schliessen:

```
private static String[] readFile(String filename){
    try (FileReader fr = new FileReader(filename);
        BufferedReader bufr = new BufferedReader(fr)) {
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    }
}
```

*Ressourcen werden hier geöffnet*

*Ressourcen werden hier automatisch geschlossen*

150

## 7. Funktionale Konzepte in Java

Funktionale Programmierung, Lambda Ausdrücke, Datenströme, Pipelines

## Funktionale vs. Imperative Programmierung

Imperative Konzepte

- Ausführen von Anweisungen
- Zustand (z.B. Felder)
- Änderbare Datentypen
- Fokus auf Datenstrukturen
- Fokus auf das "wie"

Funktionale Konzepte

- Evaluierung von Ausdrücken
- Kein Zustand
- Immutable Datentypen
- Fokus auf Datenströme
- Fokus auf das "was"

151

152

## Beispiel: Einlesen von Dateien - Imperativ

```
try (BufferedReader br=new BufferedReader(new FileReader("data.csv"))){
    LinkedList<Measurement> result = new LinkedList<>();
    br.readLine();
    String line ;
    while ((line = br.readLine()) != null){
        Measurement m = new Measurement(line);
        result .add(m);
    }
    return result ;
}
```

153

## Beispiel: Einlesen von Dateien - Funktional

```
try (Stream<String> stream = Files.lines(Paths.get("data.csv"))) {
    return stream.skip(1).map(Measurement::new).collect(toList());
}
```

154

## Streams - Datenströme

In Java sind *Streams* die Basis für funktionale Programmierung.  
Quellen von Datenströmen:

- Dateien
- Arrays
- Datenstrukturen
- ...

### Beispiel

```
Stream<String> stream = Files.lines (...)
```

155

## Operationen auf Datenströmen: Map

**Map:** Anwenden von Funktionen auf den einzelnen Elementen des Streams

- Mathematische Berechnungen
- Erstellen neuer Objekte basierend auf den existierenden Elementen.
- ...

### Beispiel

```
map(Measurement::new)
```

156

## Operationen auf Datenströmen: Reduce

**Reduce:** Aggregation der einzelnen Elemente eines Datenstroms zu einem einzelnen Wert.

- Statistische Aggregationen
- Ablegen der Elemente in Datenstrukturen
- ...

### Beispiel

```
collect (toList ())
```

157

## Beispiel: Suchen von Daten - Imperativ

```
List<Measurement> data = readCsvData();
Coordinate ref = readCoordinate();

for (Measurement m : data){
    if (m.position.near(ref)){
        System.out.println(m.originalLine);
    }
}
```

158

## Beispiel: Suchen von Daten - Funktional

```
List<Measurement> data = readCsvData();
Coordinate ref = readCoordinate();
```

```
data.stream()
    . filter (m -> ref.near(m.position))
    . forEach(System.out::println);
```

159

## Operationen auf Datenströmen: Filter

**Filter:** Herausfiltern einzelner Elemente des Streams.

- Illegale Werte entfernen
- Auswahl von Werten basierend auf Anfragen
- ...

### Beispiel

```
filter (m -> ref.near(m.position))
```

160

## Operationen auf Datenströmen: Seiteneffekte

**Seiteneffekte:** Der nicht-funktionale Aspekt: Ausführen von beliebigen Operationen aufgrund einzelner Elemente.

- Input/Output
- Datenstrukturen updaten
- ...

### Beispiel

```
forEach(System.out::println)
```

161

## Funktionalität als Parameter

Operationen auf dem Stream haben *Funktionalität* (Code) als Parameter, anstelle von *Daten*

Möglichkeiten, Funktionalität zu übergeben (statt Daten):

- "lose" Code-Stücke
- Referenzen auf Methoden
- Referenzen auf Konstruktoren

Wie können wir dies bewerkstelligen?

162

## Lambda Ausdrücke

Lambda Ausdrücke sind im Wesentlichen Methoden ohne Namen.

Normale Methode

```
double discriminant(double a, double b, double c){  
    return b*b - 4*a*c;  
}
```

Gleichwertiger Lambda Ausdruck

```
(double a, double b, double c) -> {  
    return b*b - 4*a*c;  
}
```

163

## Lambda Ausdrücke

Lambda Ausdruck

```
(double a, double b, double c) -> {  
    return b*b - 4*a*c;  
}
```

Ohne explizite Typdeklaration der Parameter

```
(a, b, c) -> {  
    return b*b - 4*a*c;  
}
```

Mit einem einzelnen Ausdruck statt einem Block

```
(a, b, c) -> b*b - 4*a*c
```

164

## Lambda Ausdruck im Beispiel

### Beispiel

```
filter (m -> ref.near(m.position))
```

- Die Methode `filter` erwartet eine Methode als Parameter, welche einen `Measurement` entgegennimmt, und einen `boolean` zurueckgibt.
- `m` ist ein Parameter vom Typ `Measurement` ✓
- `ref.near(m.position)` ist ein einzelner `boolean`Ausdruck ✓
- Die Variable `ref` aus dem definierenden Kontext ist zugänglich, solange sie *effektiv* konstant (`final`) ist.

165

## Referenzen auf Methoden

Um eine Methode auf einem Objekt *auszuführen*, schreiben wir:  
`object.method()`

Um eine *Referenz* auf eine Methode auf einem Objekt anzugeben, schreiben wir:

```
object::methode
```

166

## Referenzen auf statische Methoden

Um eine statische Methode *auszuführen*, schreiben wir:

```
Clazz.method()
```

Um eine *Referenz* auf eine statische Methode anzugeben, schreiben wir:

```
Clazz::method
```

167

## Referenz auf eine Methode im Beispiel

### Beispiel

```
forEach(System.out::println)
```

- Die Methode `forEach` erwartet eine Methode, welche nichts zurück gibt und ein Argument vom Typ `Measurement` akzeptiert.
- Die Methode `println` auf Objekt `out` erfüllt diese Eigenschaften. ✓

168

## Referenzen auf Konstruktoren

Um einen Konstruktor einer Klasse *auszuführen*, schreiben wir:

```
newClazz()
```

Um eine *Referenz* auf einen Konstruktor anzugeben, schreiben wir:

```
Clazz::new
```

169

## Referenz auf einen Konstruktor im Beispiel

### Beispiel

```
map(Measurement::new)
```

- Die Methode `map` erwartet eine Methode, welche ein Objekt eines gewissen Datentyps zurückgibt (egal welcher) und ein Argument vom Typ `String` akzeptiert.
- Der Konstruktor der Klasse `Measurement` erfüllt diese Eigenschaft. ✓

170

## Vor- und Nachteile von Funktionaler Programmierung

- |  |   |
|--|---|
| <ul style="list-style-type: none"><li>■ Weniger fehleranfällig</li><li>■ Einfacher zu unterhalten</li><li>■ Ermöglicht elegante Programmierung</li><li>■ Nicht abhängig von spezifischer Architektur</li></ul> | <ul style="list-style-type: none"><li>■ Neue Sprachkonzepte zu lernen</li><li>■ Ausführungsdetails unbekannt</li><li>■ Aufgesetzt auf imperativer Sprache</li></ul> |
|--|---|

171