12. 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.

Fehler (Errors)

Hier ist nichts mehr zu machen

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

Beispiele

- Kein Speicher mehr verfügbar
- Zu hoher Aufrufstapel ($\rightarrow$ Rekursion)
- Fehlende Programmbibliotheken
- Bug in der virtuellen Machine
- Computer-Hardwarefehler

Ausnahmen (Exceptions)

Ausnahmen werden von der virtuellen Machine oder vom Programm selbst ausgelösst 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

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("Zahl");
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: `Zahl: asdf`

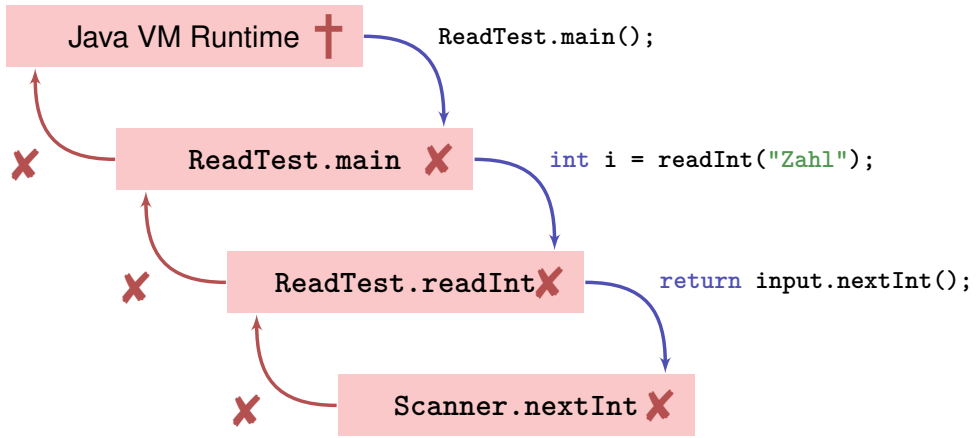
Nicht behandelte Fehler und Ausnahmen

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



Stacktraces verstehen

Eine unpassende Eingabe ...

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)
```

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

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

Stacktraces verstehen

```
1  import java.util.Scanner;
2  class ReadTest {
3      public static void main(String[] args){
4          int i = readInt("Zahl");
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)
```


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
⇒ Optionale Datentypen ermöglichen guten Stil

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

Problem gelösst.

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

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)`

Warum eine Benutzerausnahme?

Situation rechtfertigt die Benutzerausnahme:

- 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.

Behandeln von Ausnahmen

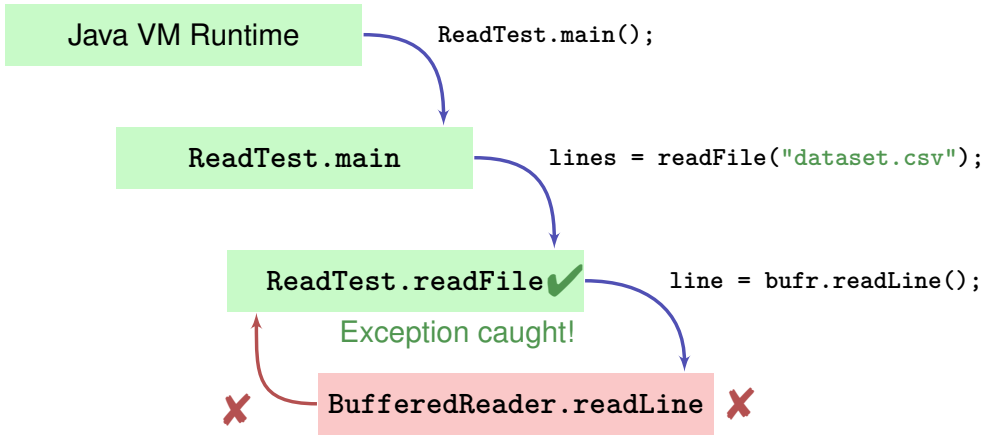
```
private static String[] readFile(String filename){
    try{
        FileReader fr = new FileReader(filename);
        BufferedReader bufr = new BufferedReader(fr);
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    } finally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

Geschützter Bereich

Massnahmen zur Wiederherstellung der Normalsituation

Wird in jedem Fall am Schluss ausgeführt, immer!

Behandlung von Ausnahmen: Propagieren stoppen!



Finally: Resourcen schliessen!

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

Resourcen:

- Dateien
- Datenströme
- GUI Elemente
- ...

Try-with-resources Anweisung

Spezifische Syntax an, 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
    }
}
```

Resourcen werden hier geöffnet

Resourcen werden hier automatisch geschlossen

13. Sortieren

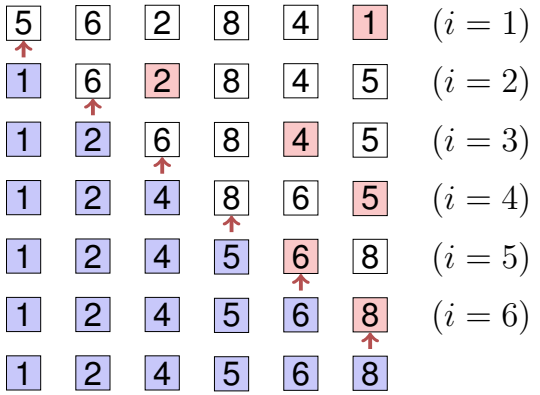
Einfache Sortierverfahren, Quicksort

Problemstellung

Eingabe: Ein Array $A = (A[1], ..., A[n])$ der Länge n.

Ausgabe: Eine Permutation A' von A, die sortiert ist: $A'[i] \leq A'[j]$ für alle $1 \leq i \leq j \leq n$.

Sortieren durch Auswahl



- Iteratives Vorgehen wie bei Bubblesort.
- Auswahl des kleinsten (oder grössten) Elementes durch direkte Suche.

Algorithmus: Sortieren durch Auswahl

Input : Array $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output : Sortiertes Array A

```
for i ← 1 to n − 1 do
    p ← i
    for j ← i + 1 to n do
        if A[j] < A[p] then
            p ← j;
    swap(A[i], A[p])
```

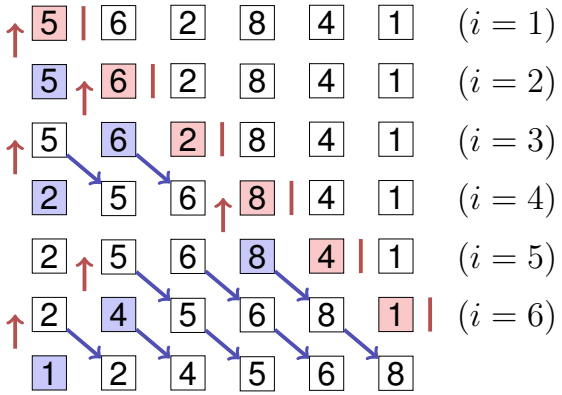

Analyse

Anzahl Vergleiche im schlechtesten Fall: $\Theta(n^2)$.

Anzahl Vertauschungen im schlechtesten Fall: $n - 1 = \Theta(n)$

Anzahl Vergleiche im besten Fall: $\Theta(n^2)$.

Sortieren durch Einfügen



- Iteratives Vorgehen: $i = 1...n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen, ggfs. Verschiebung nötig.

Sortieren durch Einfügen

? Welchen Nachteil hat der Algorithmus im Vergleich zum Sortieren durch Auswahl?

! Im schlechtesten Fall viele Elementverschiebungen.

? Welchen Vorteil hat der Algorithmus im Vergleich zum Sortieren durch Auswahl?

! Der Suchbereich (Einfügebereich) ist bereits sortiert. Konsequenz: binäre Suche möglich.

Algorithmus: Sortieren durch Einfügen

Input : Array $A = (A[1], \ldots, A[n])$, $n \geq 0$.
Output : Sortiertes Array A

```
for i ← 2 to n do
    x ← A[i]
    p ← BinarySearch(A[1...i − 1], x); // Kleinstes p ∈ [1, i] mit A[p] ≥ x
    for j ← i − 1 downto p do
        A[j + 1] ← A[j]
    A[p] ← x
```


Analyse

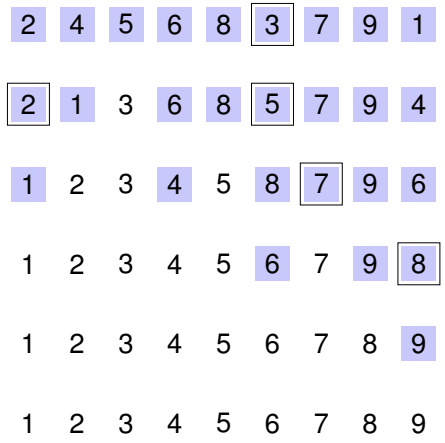
Anzahl Vergleiche im schlechtesten Fall:
$\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \mathcal{O}(n \log n)$.

Anzahl Vergleiche im besten Fall: $\Theta(n \log n)$.[7]

Anzahl Vertauschungen im schlechtesten Fall: $\sum_{k=2}^{n}(k-1) \in \Theta(n^2)$

[7] Mit leichter Anpassung der Funktion BinarySearch für das Minimum / Maximum: $\Theta(n)$

Quicksort (willkürlicher Pivot)



Algorithmus Quicksort($A[l, \ldots, r]$

Input : Array A der Länge n. $1 \leq l \leq r \leq n$.
Output : Array A, sortiert zwischen l und r.

```
if l < r then
    Wähle Pivot p ∈ A[l, ..., r]
    k ← Partition(A[l, ..., r], p)
    Quicksort(A[l, ..., k − 1])
    Quicksort(A[k + 1, ..., r])
```

Analyse: Anzahl Vergleiche

Bester Fall. Pivotelement = Median; Anzahl Vergleiche:

$$T(n) = 2T(n/2) + c \cdot n,\ T(1) = 0 \quad \Rightarrow \quad T(n) \in \mathcal{O}(n \log n)$$

Schlechtester Fall. Pivotelement = Minimum oder Maximum; Anzahl Vergleiche:

$$T(n) = T(n-1) + c \cdot n,\ T(1) = 0 \quad \Rightarrow \quad T(n) \in \Theta(n^2)$$

Analyse (Randomisiertes Quicksort)

Theorem

Im Mittel benötigt randomisiertes Quicksort $\mathcal{O}(n \cdot \log n)$ Vergleiche.

Praktische Anmerkungen

Rekursionstiefe im schlechtesten Fall: $n - 1$[8]. Dann auch Speicherplatzbedarf $\mathcal{O}(n)$.

Kann vermieden werden: Rekursion nur auf dem kleineren Teil. Dann garantiert $\mathcal{O}(\log n)$ Rekursionstiefe und Speicherplatzbedarf.

[8] Stack-Overflow möglich!

Praktische Anmerkungen

Für den Pivot wird in der Praxis oft der Median von drei Elementen genommen. Beispiel: Median3$(A[l], A[r], A[\lfloor l + r/2 \rfloor])$.