

## 18. Java Input/Output

---

Benutzereingaben/Konsolenausgabe, Datei Ein- und Ausgabe (I/O)

### Benutzereingaben (die halbe Wahrheit)

```
public class Main {
    public static void main(String[] args) {
        Out.print("Number: ");
        int i = In.readInt();
        Out.print("Your number: " + i);
    }
}
```

Anscheinend passiert nicht viel!

```
Number: spam
Your number: 0
```

### Benutzereingaben (die halbe Wahrheit)

- z.B. einer Zahl: `int i = In.readInt();`
- Unsere `In` Klasse stellt diverse solche Methoden zur Verfügung.
- Manche dieser Methoden müssen mit falschen Eingaben umgehen können: Was passiert mit `readInt()` bei folgendem Input?

"spam"

### Benutzereingaben (die ganze Wahrheit)

- z.B. einer Zahl, mittels der Klasse `Scanner`

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Out.print("Number: ");
        Scanner input = new Scanner(System.in);
        int i = input.nextInt();
        Out.print("Your number: " + i);
    }
}
```

Was passiert bei folgendem Input?

"spam"

## Benutzereingaben (die ganze Wahrheit)

```
Number: spam
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at Main.main(Main.java:7)
    at TestRunner.main(TestRunner.java:330)
```

Oh! Wir schauen uns das im naechsten Kapitel an...

476

## Also: Benutzereingabe/Konsolenausgabe

- Lesen von Eingaben via dem Eingabestrom **System.in**
- Schreiben von Ausgaben via dem Ausgabestrom **System.out**

478

## Konsolenausgabe

- Bisher kannten Sie: **Out.print("Hi")** oder **Out.println("Hi")**

- Ohne unsere **Out** Klasse:

```
System.out.print("The answer is: ");
System.out.println(42);
System.out.println("What was the question?!");
```

Dies ergibt die folgende Ausgabe:

```
The answer is: 42
What was the question?!
```

477

## Dateien (zeilenweise) Lesen/Schreiben

- Dateien können mittels der Klasse **java.io.FileReader** byte für byte gelesen werden
- Um ganze Zeilen zu lesen, verwenden wir ausserdem einen **java.io.BufferedReader**
- Dateien können mittels der Klasse **java.io.FileWriter** byte für byte geschrieben werden
- Um ganze Zeilen zu schreiben, verwenden wir ausserdem einen **java.io.BufferedWriter**

479

## Dateien (zeilenweise) Lesen

```
import java.io.FileReader;
import java.io.BufferedReader;

public class Main {
    public static void main(String[] args) {
        FileReader fr = new FileReader("gedicht.txt");
        BufferedReader bufr = new BufferedReader(fr);
        String line;
        while ((line = bufr.readLine()) != null){
            System.out.println(line);
        }
    }
}
```

... deshalb nun ...

## Dateien (zeilenweise) Lesen

Wir erhalten folgenden Kompilier-Fehler:

```
./Main.java:6: error: unreported exception FileNotFoundException;
                must be caught or declared to be thrown
    FileReader fr = new FileReader("gedicht.txt");
                        ^
./Main.java:9: error: unreported exception IOException; must be
                caught or declared to be thrown
    while ((line = bufr.readLine()) != null){
                        ^
```

2 errors

Offensichtlich müssen wir mehr verstehen zum Thema  
"Exceptions"

## 19. 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.

484

# Fehler (Errors)



Hier ist nichts mehr zu machen

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

485

# 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

486

# 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

487

## 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

488

## Nicht behandelte Fehler und Ausnahmen

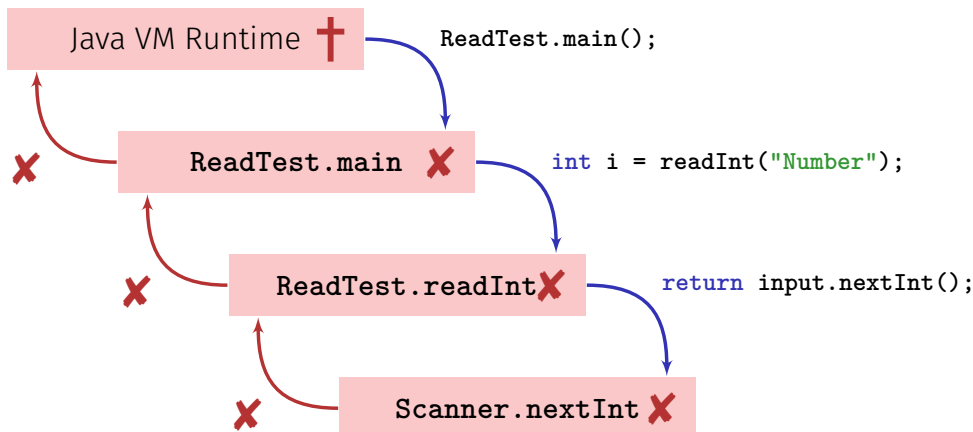
Das Programm stürzt ab und hinterlässt auf der Konsole eine "Aufrufstapelzurückverfolgung" ☺ (ab jetzt: **Stacktrace**). Darin sehen wir, wo genau das Programm abgebrochen wurde.

```
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.

489

## Ausnahme propagiert durch Aufrufstapel



490

## 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.

491

## 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)

492

## 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 ...?!
    }
}
```

494

## 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

493

## Erste Erkenntnis: Oft keine Ausnahme

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

495

## Zweite Erkenntnis: Ausnahmen verhindern



Problem gelöst.

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

496

## 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

497

## 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 thrown
    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

498

## 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\(java.nio.file.Path, java.nio.charset.Charset\)](#)

499

## 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.

500

## 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
    }
}
```

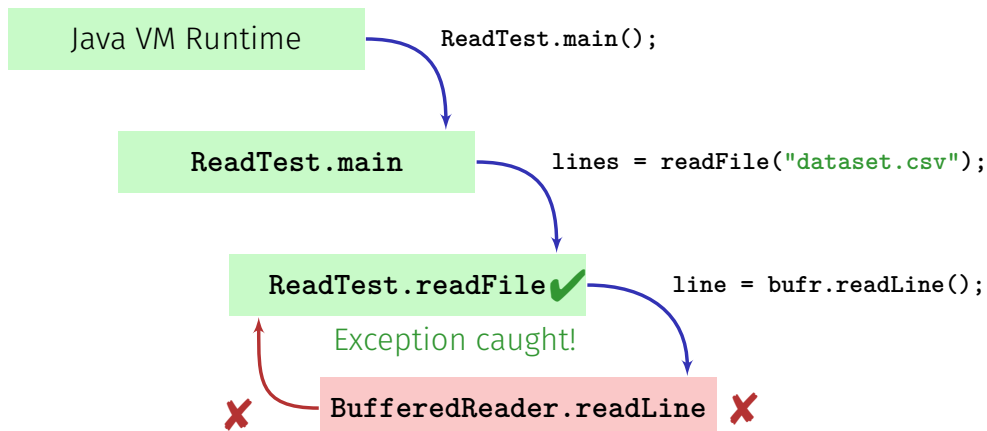
← Geschützter Bereich

← Massnahmen zur Wiederherstellung der Normalsituation

← Wird in jedem Fall am Schluss ausgeführt, immer!

501

## Behandlung von Ausnahmen: Propagieren stoppen!



502

## Finally: Ressourcen schliessen!

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

Ressourcen:

- Dateien
- Datenströme
- GUI Elemente
- ...



503



# 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  
    }  
}
```

Resources werden hier geöffnet

Resources werden hier automatisch geschlossen