252-0027
Einführung in die Programmierung

11. Arbeiten mit Dateien

Manuela Fischer, Malte Schwerhoff

Departement Informatik
ETH Zürich

11. Arbeiten mit Dateien

11.1 Arbeiten mit Dateien

11.2 Eingabe: Dateien lesen

11.3 Ausgabe: Dateien schreiben

Persistente Daten ➜ Dateien

- Beispiel: Aus Code Expert exportierte Kursdaten

cx.csv

```
1  Legi,Username,Lastname,Firstname,Email,TotalScore,EarnedXP,Groupname,Project 1: Guess a Number-Task 1: First Version,Projec
2  59-935-541,hhrdsjdukljr,Khorath,Lukas,hhrdsjdukljr@inf.ethz.ch,8,0,unknown,1,1,1,1,1,1,1,1
3  52-128-754,aseekxuggvry,D'Vorak,Anna,aseekxuggvry@inf.ethz.ch,8,0,Students,1,1,1,1,1,1,1,1
4  03-063-714,nkdwgnjymuoa,T'Rok,Matteo,nkdwgnjymuoa@inf.ethz.ch,8,0,Students,1,1,1,1,1,1,1,1
5  97-029-284,vgmcipbhhnot,K'Vok,Sophie,vgmcipbhhnot@inf.ethz.ch,8,0,Students,1,1,1,1,1,1,1,1
6  18-751-488,kdkvejzkduyz,Ghorath,Elias,kdkvejzkduyz@inf.ethz.ch,8,0,Students,1,1,late submitted,1,1,1,1,1
7  80-685-813,htnnjrqnkmyl,Qorath,Lara,htnnjrqnkmyl@inf.ethz.ch,8,0,Students,1,1,1,1,1,1,1,1
8  38-347-738,bzdqaghenwxj,Shorak,Jonas,bzdqaghenwxj@inf.ethz.ch,8,0,Students,1,1,1,1,1,1,1,1
9  02-542-246,jfqlgqquxmsb,Vokar,Léa,jfqlgqquxmsb@inf.ethz.ch,7.9565,0,Students,late submitted,late submitted,1,1,1,1,0.9565,1
10 92-769-538,qcomtfljyyrd,K'Rath,Noah,qcomtfljyyrd@inf.ethz.ch,6,0,Students,1,1,1,1,1,not submitted,not submitted,1
11 89-032-722,sfibdcqswmtu,Gorath,Emma,sfibdcqswmtu@inf.ethz.ch,6,0,Students,1,1,1,1,1,1,not submitted,not submitted
12 80-036-020,zlxatfqzlccy,T'Ghor,David,zlxatfqzlccy@inf.ethz.ch,6,0,Students,1,1,1,1,1,not submitted,not submitted,1
13 52-317-639,jqueuavpsnsw,Zorak,Charlotte,jqueuavpsnsw@inf.ethz.ch,5,0,Students,1,1,1,1,1,not submitted,not submitted,not sub
14 23-016-528,ezsmsnmcidxb,B'Nar,Gabriel,ezsmsnmcidxb@inf.ethz.ch,5,0,Students,1,1,1,1,1,not submitted,not submitted,not submi
15 91-938-360,yfvmgsbgavux,K'Vokar,Clara,yfvmgsbgavux@inf.ethz.ch,5,0,Students,1,1,1,1,1,not submitted,not submitted,not submi
```

- Textdateien (unser Fokus), aber auch Binärdateien (Bilder, …)

Beispiel: Temperaturen

- Datei enthält (Tages-)Temperaturen
- Ziel: Programm, das die Differenz zwischen aufeinanderfolgenden Temperaturen ausgibt

Datei

```
16.2   23.5
19.1 7.4  22.8

18.5  -1.8 14.9
```

Ausgabe

```
16.2 to 23.5, change = 7.3
23.5 to 19.1, change = -4.4
19.1 to 7.4, change = -11.7
7.4 to 22.8, change = 15.4
22.8 to 18.5, change = -4.3
18.5 to -1.8, change = -20.3
-1.8 to 14.9, change = 16.7
```

Arbeiten mit Dateien

Ermöglicht durch Klasse `File`

- Importieren der Klasse `File`: `import java.io.File;`
- Datei-Handle («file handle») erstellen
  - `File file = new File("example.txt");`
  - Objekt vom Typ `File`, das für (potenziell nicht existierende) Datei steht
  - Datei wird **nicht** erstellt
- Datei kann dann via Datei-Handle bearbeitet werden

```
import java.io.File;
File file = new File("example.txt");
```

File-Methoden

Name der Methode	Beschreibung der Methode
`exists()`	Gibt `true` zurück, falls Datei existiert, sonst `false`
`canRead()`	Gibt `true` zurück, falls diese Datei gelesen werden kann, sonst `false`
`getName()`	Gibt den Namen dieser Datei zurück
`length()`	Gibt die Dateigrösse, in Bytes, zurück
`delete()`	Löscht die Datei
`renameTo(file)`	Benennt die Datei zum Namen eines Datei-Handles `file` um

Beispiel

Erstellen eines Datei-Handles, **nicht** einer Datei!

```
import java.io.File;

File file = new File("example.txt"); // file handle

// delete file if too large
if (file.exists() && file.length() > 1000) {
  file.delete();
}
```

11. Arbeiten mit Dateien

11.1 Arbeiten mit Dateien

11.2 Eingabe: Dateien lesen

11.3 Ausgabe: Dateien schreiben

Scanner

- `Scanner` importieren (bereits bekannt):
  - `import java.util.Scanner;`
- `Scanner`-Objekt konstruieren mit Angabe der Eingabequelle:
  - Von der Konsole (bereits bekannt):
    ```
    Scanner consoleScanner = new Scanner(System.in);
    ```
  - Aus einer Datei (neu):
    ```
    Scanner fileScanner = new Scanner(file);
    ```
    Datei-Handle
  - Von einem String (später mehr):
    ```
    Scanner stringScanner = new Scanner(myString);
    ```

Lesen aus Datei mit Scanner

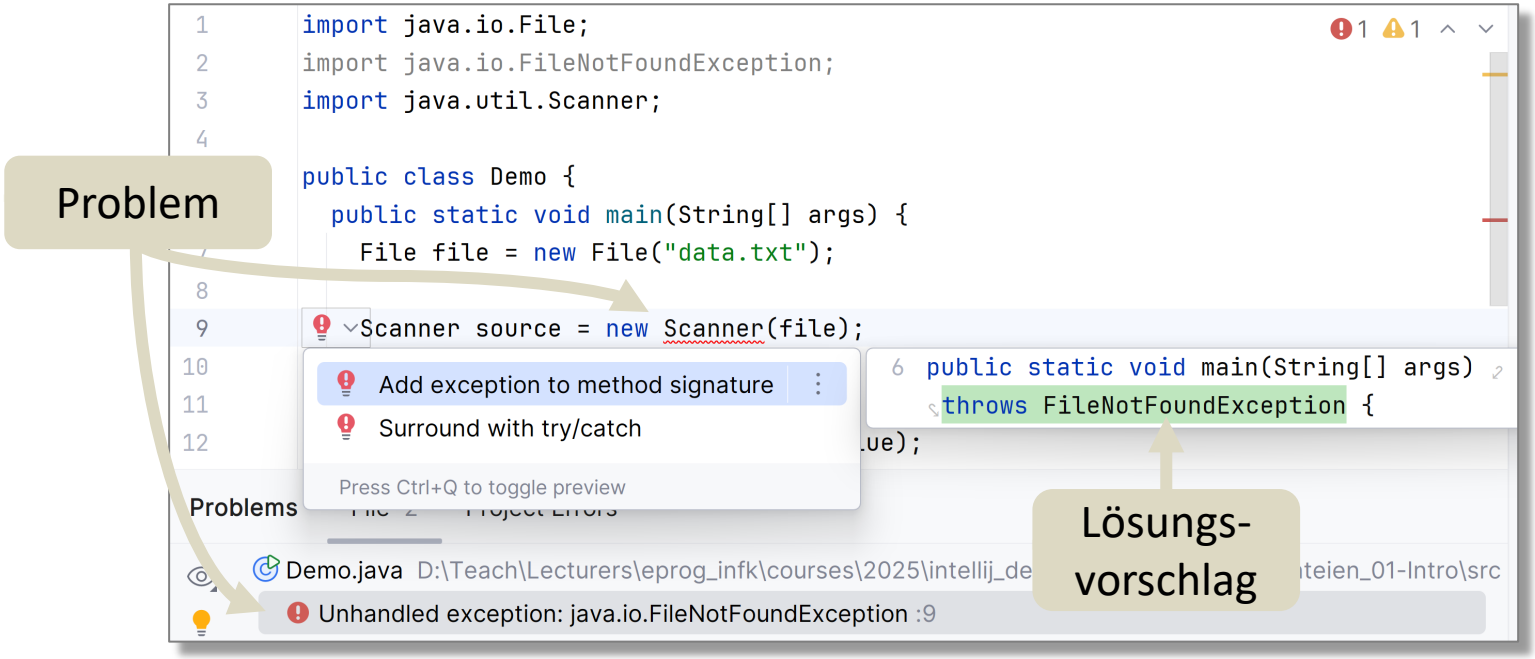
```
import java.io.File;                    // for files
import java.util.Scanner;               // for scanner

File file = new File("input.txt");      // file handle
Scanner scanner = new Scanner(file); // scanner for file
int number = scanner.nextInt();         // read int from file
```

Alternativ in einer Zeile:

```
Scanner scanner = new Scanner(new File("input.txt"));
```

Noch nicht ganz…



- Unbehandelte Ausnahme («unhandled exception»)

Dokumentation: Scanner

Scanner

```
public Scanner(File source)
        throws FileNotFoundException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:

`source` - A file to be scanned

Throws:

`FileNotFoundException` - if source is not found

Was bedeutet das?

Vorschau: Exceptions («Ausnahmen»)

- Gewisse Operationen können zu Laufzeitfehlern führen
  - Sie **werfen** eine Exception («`throw an exception`»)
  - Methode mit einer solchen Operation muss entweder
    a. **Fangen** («**Catch**»): Auf Laufzeitfehler reagieren (try/catch-Block) *oder*
    b. **Werfen** («**Throw**»): Exception weiterwerfen (an Aufrufer delegieren)

- Methode muss das (Weiter-)Werfen einer Exception ankündigen durch **throws-Dekalration** («**throws declaration**») :
  - `throws ExceptionType` nach Parameterliste und vor Methodenrumpf

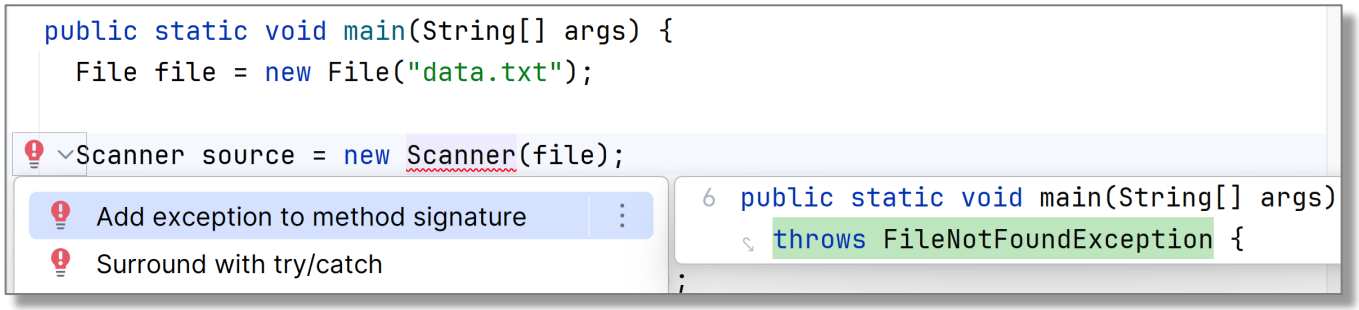
Zwei Optionen: Catch oder Throw

Falls Laufzeitfehler durch Code-Zeile(n) möglich, muss entweder

a) Die entsprechende Exception gefangen werden (durch Umschliessen der Zeile(n) mit einem try/catch-Block)

später

b) Die umliegende Methode deklarieren, dass sie entsprechende Exception werfen kann



`throws`-Ankündigung einer Methode

```
… returnType method(...) throws exceptionType1, … {
   …
}
```

- Deklaration, dass die Methoden beim Aufruf eine Exception werfen kann
  - z.B. durch Nutzen von `Scanner`-Methoden/Konstruktoren
- Wer eine solche Methode aufruft, muss nun entweder
  - Sich selbst um mögliche Exceptions kümmern
  - Das (Weiter-)Werfen der Exception deklarieren (→ an Aufrufer delegieren)

Lesen aus Datei mit Scanner und throws

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Test {
  public static void main(String[] args)
      throws FileNotFoundException {

    Scanner f = new Scanner(new File("input.txt"));
    int number = f.nextInt();
  }
}
```

Die von `main()` geworfenen Exceptions werden letztendlich vom System gefangen (→ Programmabbruch)

Wieso hatten wir bisher keine `throws`?

- Nicht alle Eingabequellen für `Scanner` können Exceptions verursachen

Scanner

```
public Scanner(InputStream source)
```

Constructs a new Scanner that produces values scanned from the specified input stream. Bytes from the stream are converted into characters using the underlying platform's default charset.

- Zudem müssen gewisse Exception-Typen nicht deklariert werden
  - Werden «unchecked exceptions» genannt (Gegenteil: «checked exceptions»)
  - Beispiele: `NullPointerException`, `ArrayIndexOutOfBoundsException`

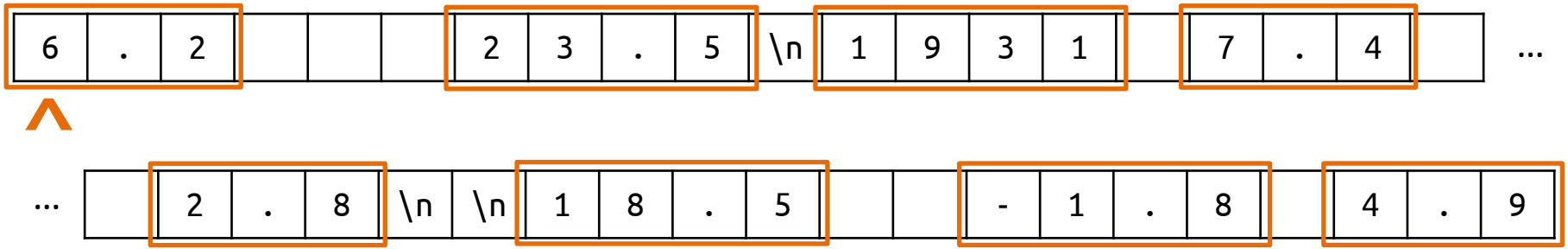
Zurück zur Eingabe: Cursor

Datei

```
6.2   23.5
1931 7.4   2.8

18.5   -1.8 4.9
```

- Eingabe ist eine Folge von Zeichen
  - \n steht für neue Zeile, ist aber nur ein weiteres Zeichen in der Datei
- Scanner zerlegt Eingabe in Tokens (getrennt durch Leerraum)
- Eingabe-Cursor bestimmt aktuelle Position im Zeichenstrom



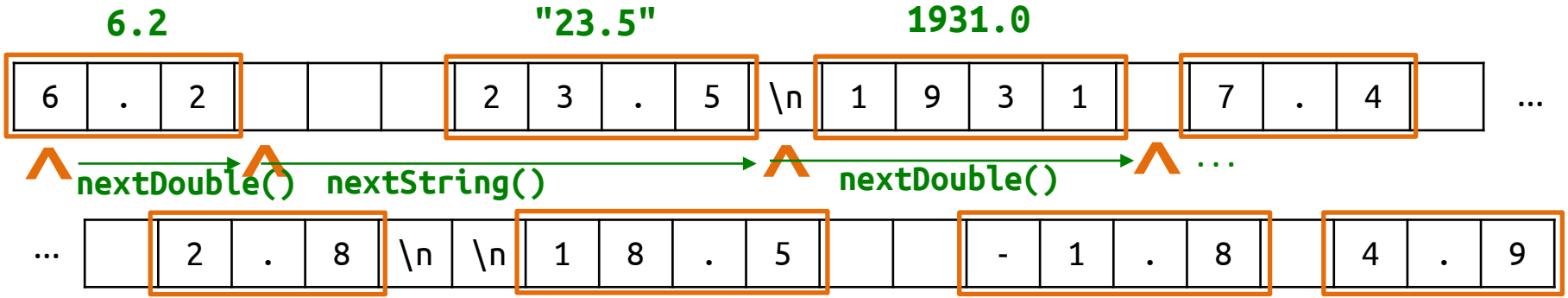
Konsumieren von Tokens

Datei

```
6.2   23.5
1931 7.4   2.8

18.5  -1.8 4.9
```

- Bei Aufruf von `nextInt()`, `nextDouble()`, …
  - Nächstes Token (Leerraum ignoriert) wird gelesen und zurückgegeben
  - Eingabe-Cursor wurde zum Ende des Tokens bewegt (= Token konsumiert)
  - Laufzeitfehler bei falscher Form (mehr später)
  - Datei bleibt unverändert



Zurück zum Beispiel

- Datei enthält (Tages-)Temperaturen
- Ziel: Programm, das die Differenz zwischen aufeinanderfolgenden Temperaturen ausgibt

Datei

```
16.2   23.5
19.1 7.4  22.8

18.5  -1.8 14.9
```

Ausgabe

```
16.2 to 23.5, change = 7.3
23.5 to 19.1, change = -4.4
19.1 to 7.4, change = -11.7
7.4 to 22.8, change = 15.4
22.8 to 18.5, change = -4.3
18.5 to -1.8, change = -20.3
-1.8 to 14.9, change = 16.7
```

Temperaturen: Code-Gerüst

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Temperatures {
  public static void main(String[] args)
      throws FileNotFoundException {

    Scanner in = new Scanner(new File("temps.txt"));
    // … see next slide …
  }
}
```

Temperaturen: Versuch 1

```
double previous = in.nextDouble(); // read first value
for (int i = 1; i <= 7; i++) {     // read seven values
  double next = in.nextDouble();
  System.out.println("change: " + (next - previous));
  previous = next;
}
```

- Funktioniert nur mit genau acht Temperaturen
  - Zusätzliche Werte werden nicht gelesen …
  - … und bei weniger Werten gibt es einen Laufzeitfehler

```
Exception in thread "main" java.util.NoSuchElementException
```

Scanner-Exceptions

- `NoSuchElementException`
  - Versuch, über das Ende der Datei hinauszulesen

```
Exception in thread "main" java.util.NoSuchElementException
    at java.util.Scanner.throwFor(Scanner.java:838)
      at java.util.Scanner.next(Scanner.java:1347)
        at Temperatures.main(Temperatures.java:13)
```

Problem in dieser Zeile

- `InputMismatchException`
  - Beim Versuch, einen Typ zu lesen, für den das Token nicht gültig ist
  - z.B. `hi` oder `1.234` mit `nextInt()` oder `1.2.3` mit `nextDouble()`

Scanner-Exceptions vermeiden

- Scanner-Methoden zum Prüfen des Inputs
  - Konsumieren keine Eingabe, liefern nur Information über nächstes Token
  - Praktisch, um Laufzeitfehler zu verhindern
  - Nicht nur für Dateien, auch für Konsole und String

Name der Methode	Beschreibung der Methode
`hasNext()`	returns `true` if there is a next token
`hasNextInt()`	returns `true` if there is a next token and it can be read as an `int`
`hasNextDouble()`	returns `true` if there is a next token and it can be read as a `double`

hasNext-Methode

```
Scanner in = new Scanner(new File("input.txt"));

while (in.hasNext()) { // end of file not yet reached
  String token = in.next();
  System.out.print("next token is: ");
  System.out.println(token);
}
```

Token existiert garantiert

Temperaturen: Versuch 2

```
double previous = in.nextDouble();

while (in.hasNext()) { // end of file not yet reached
  double next = in.nextDouble();
  System.out.println("change: " + (next - previous));
  previous = next;
}
```

- Funktioniert nur mit validen Zahlen: `InputMismatchException` bei Zeichenfolgen, die nicht als `double` interpretiert werden können

Temperaturen: Versuch 3

```
double previous = in.nextDouble();

while (in.hasNextDouble()) { // next token is a double
  double next = in.nextDouble();
  System.out.println("change: " + (next - previous));
  previous = next;
}
```

- Funktioniert nur falls
  - Datei nicht-leer ist und
  - Erstes Token als `double` interpretiert werden kann

Temperaturen: Versuch 4

```
if (in.hasNextDouble()) { // first token is a double
  double previous = in.nextDouble();

  while (in.hasNextDouble()) { // next token is a double
    double next = in.nextDouble();
    System.out.println("change: " + (next - previous));
    previous = next;
  }
}
```

Temperaturen: Nicht-numerische Tokens

Datei

```
16.2   23.5
Dienstag 19.1   Mittwoch 7.4   Do.Wert: 22.8
^

18.5  -1.8  (@Clark Messfehler???) 14.9
```

- Programm bricht bei erstem nicht-numerischen Token ab
- Möglichkeiten für weiteres Vorgehen
  1. Exception an Benutzer melden – Datei muss «sauber» sein
  2. Nicht-numerische Tokens aktiv ignorieren

Temperaturen: Nicht-numerische Tokens ignorieren

```
double previous = 0;

while (in.hasNext() && !in.hasNextDouble()) input.next();

if (in.hasNextDouble()) previous = input.nextDouble();

while (in.hasNext()) { // end of file not yet reached
  if (in.hasNextDouble()) { // next token is a double
    double next = in.nextDouble();
    System.out.println("change: " + (next - previous));
  } else { // next token is not a double
    in.next(); // ignore unwanted token
  }
}
```

Beispiel: Arbeitsstunden

- Datei enthält Arbeitsstunden: Eine Zeile pro Person mit ID, Name und Sequenz von Arbeitsstunden

Datei

```
123 Paula 12.5 8.1 7.6 3.2
456 Erich 4.0 11.6 6.5 2.7 12
789 Steffie 8.0 8.0 8.0 8.0 7.5
```

- Schreiben Sie ein Programm, das die Gesamtarbeitsstunden pro Person berechnet

Ausgabe

```
Paula (ID#123) worked 31.4 hours (7.85 hours/day)
Erich (ID#456) worked 36.8 hours (7.36 hours/day)
Steffie (ID#789) worked 39.5 hours (7.9 hours/day)
```

Arbeitsstunden: Code-Gerüst

```
// Imports omitted

public class HoursWorked {
  public static void main(String[] args)
      throws FileNotFoundException {

    Scanner in = new Scanner(new File("hours.txt"));
    // … see next slide …
  }
}
```

Arbeitsstunden: Versuch 1

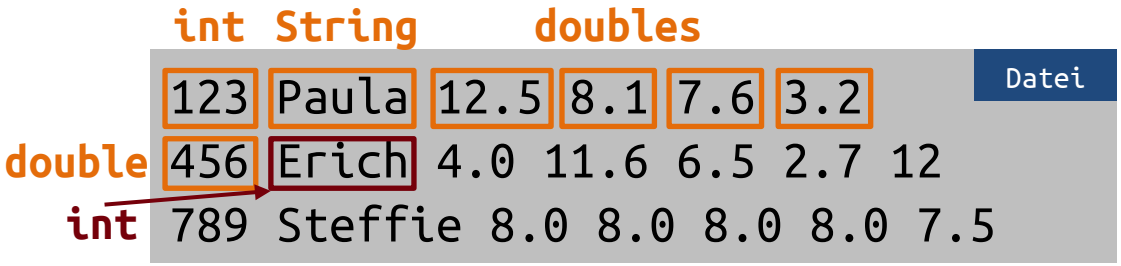
```
while (in.hasNext()) {
  int id = in.nextInt();        // read ID
  String name = in.next();      // read name

  double totalHours = 0.0;
  int days = 0;

  while (in.hasNextDouble()) { // read working hours
    totalHours += in.nextDouble();
    days++;
  }

  System.out.print(name + " " + id + " " + totalHours + " ");
  System.out.println(totalHours/days);
}
```

Arbeitsstunden: Versuch 1




```
Paula (ID#123) worked 487.4 hours (97.48 hours/day)
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:840)
        at java.util.Scanner.next(Scanner.java:1461)
        at java.util.Scanner.nextInt(Scanner.java:2091)
        at HoursWorked.main(HoursBad.java:9)
```

- Zeilenumbruch wird ignoriert → Personalnummer der 2. Person wird als Arbeitsstunden der 1. Person interpretiert

Hybrider Ansatz für zeilenbasierte Dateien

1. Eingabedatei zeilenweise einlesen
   - Scanner (mit Datei als Quelle) kann ganze Zeile der Datei als String einlesen

Name der Methode	Beschreibung der Methode
`nextLine()`	returns next entire line of input
`hasNextLine()`	returns `true` if there are any more lines of input to read (always true for console input)

2. Einzelne Zeilen in Tokens zerlegen
   - Scanner (mit Zeilen-String als Quelle) kann Zeile in Tokens zerlegen (wie gehabt)

Hybrider Ansatz: Grundgerüst

Scanner 1: Liest Datei und zerlegt sie in Zeilen

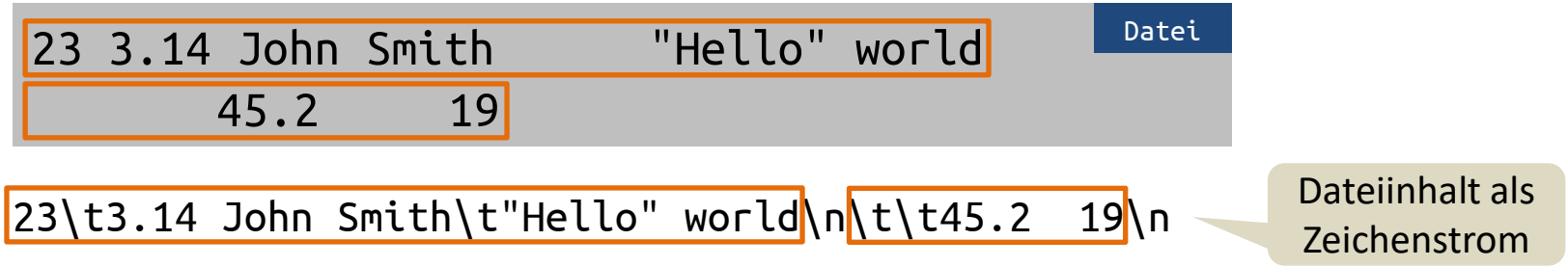
```
Scanner fileScanner = new Scanner(new File("file.txt"));

while (fileScanner.hasNextLine()) { // not last line yet
  String line = fileScanner.nextLine(); // read next line
  Scanner lineScanner = new Scanner(line); // scanner for line
  // … work with line via lineScanner …
}
```

Scanner 2: Liest Zeile und zerlegt sie in Tokens

Hybrider Ansatz: Scanner 1 (Zeilen von Datei)

- `nextLine`-Methode liest eine Zeile bis zum nächsten `\n` (oder Dateiende)
  - An Anrufer zurückgegebener String enthält *kein* `\n`
  - `\n` wird aber konsumiert



Beispiel hybrider Ansatz: Wörter pro Zeile

input.txt	Output
The quick brown fox jumps over the lazy dog.	Line has 6 words Line has 3 words

```
Scanner fileScanner = new Scanner(new File("input.txt"));
while (fileScanner.hasNextLine()) {         // not last line yet
  String line = fileScanner.nextLine();     // read next line
  Scanner lineScanner = new Scanner(line);  // scanner for line
  int wordCount = 0;
  while (lineScanner.hasNext()) {           // not last token yet
    lineScanner.next();                     // read/ignore next token
    wordCount++;
  }
  System.out.println("Line has " + count + "words");
}
```

Zurück zu Arbeitsstunden: Lösung

```
Scanner fileScanner = new Scanner(new File("hours.txt"));

while (fileScanner.hasNextLine()) {
  String line = fileScanner.nextLine();

  processLine(line); // … see next slide …
}
```

Arbeitsstunden: Lösung

```
public static void processLine(String line) {
  Scanner in = new Scanner(line); // line scanner

  int id = in.nextInt();          // read ID
  String name = in.next();        // read name
  double totalHours = 0.0;
  int days = 0;
  while (in.hasNextDouble()) {    // read working hours
    totalHours += in.nextDouble();
    days++;
  }
  System.out.print(name + " " + id + " " + totalHours + " ");
  System.out.println(totalHours/days);
}
```

Einziger Unterschied zum 1. Versuch (Seite 33)

Warnung

- Nie gleichen Scanner für Zeilen und für Tokens verwenden!
  - Ein Scanner für Zeilen mit `nextLine`
  - Ein zweiter Scanner für Token-Verarbeitung (`nextInt()` usw.)

- Bei *einem* Scanner für beides
  - Beim Token-Scannen wird `\n` stehengelassen …
  - … was beim Zeilen-Scannen als leere Zeile interpretiert wird

Nur *ein* Scanner – was passiert?

Datei

```
23 3.14
John Smith      "Hello" world
       45.2     19
```

```
Scanner input = new Scanner(new File("input.txt"));
System.out.println(input.nextInt());     // outputs 23
System.out.println(input.nextDouble());  // outputs 3.14
System.out.println(input.nextLine());    // no output!
```

Lesen der leeren Zeile

```
23\t3.14\nJohn Smith\t"Hello" world\n\t\t45.2  19
```

Was passiert hier?

```
Scanner console = new Scanner(System.in);
System.out.print("Enter your age: ");
int age = console.nextInt();
System.out.print("Enter your name: ");
String name = console.nextLine();
System.out.print(name + " is " + age + " years old.");
```

```
Enter your age: 32
Enter your name: is 32 years old.
```

`nextLine()` für Tokens mit Leerzeichen

- Beispiel: Namen einer Datei vom Benutzer erfragen
- Name könnte Leerzeichen enthalten ➜ `nextLine()`

```
Scanner console = new Scanner(System.in);
System.out.println("Type a file name: ");
String fileName = console.nextLine();
File fileHandle = new File(fileName);
if (!fileHandle.exists()) {
  System.out.println("File not found!");
} else {
  Scanner inputFile = new Scanner(fileHandle);
  …
```

11. Arbeiten mit Dateien

11.1 Arbeiten mit Dateien

11.2 Eingabe: Dateien lesen

11.3 Ausgabe: Dateien schreiben

PrintStream

- Importieren der `PrintStream`-Klasse: `import java.io.PrintStream;`
- Erlaubt Datenausgabe (unter anderem) in Datei via Datei-Handle

```
import java.io.File;
import java.io.PrintStream;
File file = new File("output.txt");
PrintStream fileOutput = new PrintStream(file);
```

- Ausgabemethoden von `System.out` funktionieren auch für `PrintStream` :
  - `fileOutput.print()` statt `System.out.print()`
  - `fileOutput.println()` statt `System.out.println()`

Beispiel

```
// Imports omitted

File file = new File("output.txt");
PrintStream fileOutput = new PrintStream(file);

for (int i = 1; i <= 2; i++) {
  fileOutput.print("Hello world ");
  fileOutput.print(i);
  fileOutput.println("!");
}
```

Auch hier Exceptions möglich

```
Hello world 1!
Hello world 2!
```

Wurde in die Datei geschrieben

In Eclipse sichtbar (evtl. nötig: Ansicht mit Taste F5 aktualisieren)

Dateiausgabe

- `new PrintStream(fileHandle)` öffnet die Datei (zum Schreiben)
  - Falls Datei nicht existiert, wird sie erstellt
  - Falls Datei existiert, wird sie überschrieben
- Zwei mögliche Fehler
  1. Gleiche Datei für Eingabe (mit `Scanner`) und Ausgabe (mit `PrintStream`)
     - `PrintStream` leert/überschreibt die Eingabedatei (kein Inhalt mehr)
  2. Datei mehrfach öffnen – z.B. im Rumpf einer Schleife
     - Datei wird wiederholt (zum Schreiben) geöffnet, alter Inhalt wird überschrieben
     - Nur die letzte Ausgabe befindet sich am Schluss in der Datei

`System.out` und `PrintStream`

- `System.out` (Objekt für Konsolenausgabe) ist ein `PrintStream`
- Ermöglicht z.B. Methoden, die parametrisch in der Datensenke sind

```
void writeToLog(String message, PrintStream sink) {
  // … Nachricht formatieren …

  sink.println(message);
}
```

```
writeToLog("Password reset for user 'mschwerhoff'", System.out);
```

```
writeToLog("Password reset for user 'mschwerhoff'", fileStream);
```

Abschluss: Dateien schliessen

- Best Practice: Dateien nach getaner Arbeit schliessen (andernfalls *kann* es zu technischen Problemen kommen)
  - Eingabe: `scanner.close()`
  - Ausgabe: `printstream.close()`
- Alternative: try-with-resource-Block

Für EProg nicht relevant, aber in der Praxis

```
try (Scanner in = new Scanner(new File("input.txt"))) {
  // Scanner «in» hier nutzen …
} // … wird anschliessend automatisch geschlossen
```

```
try (PrintStream out = new PrintStream(new File("output.txt"))) {
  // Stream «out» hier nutzen …
} // … wird anschliessend automatisch geschlossen
```