Objektorientierte Programmierung

Kapitel 10: Ein-/Ausgabe

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

http://www.informatik.uni-halle.de/~brass/oop18/

Inhalt

1. Kommandozeile
2. Dateien
3. Streams
4. Reader/Writer
5. Formatierung
6. Scanner

Motivation/Übersicht

- Offensichtlich sind Programme nur sinnvoll, wenn sie mit ihrer Umwelt interagieren:
  - Mit Menschen (Benutzern) über Bildschirm, Tastatur, Maus, Touchscreen,
  - mit Dateien auf dem Computer (gespeichert auf Festplatte, CD/DVD, USB-Stick, Speicherkarte),
  - mit angeschlossenen Geräten (z.B. Audio-Ausgabe, Drucker, Schalten von Zündkreisen bei Feuerwerken),
  - mit anderen Programmen auf dem gleichen Rechner oder über das Netzwerk (z.B. Webserver).
- Die Möglichkeiten zur Ein-/Ausgabe sind nicht in die Sprache Java selbst eingebaut, sondern werden über Bibliotheksklassen ermöglicht, die mit Java mitgeliefert werden (Java API).

Lernziele

- Man muss nicht alle Bibliotheksklassen mit allen ihren Methoden auswendig lernen.
- Man muss aber eine Vorstellung davon haben, was es gibt, und zügig nähere Information finden können.

  Dazu sollte man sich in der offiziellen Dokumentation zurechtfinden und ein gutes Lehrbuch kennen. Natürlich kann man sich mit Google Inspiration holen, aber am Ende muss man wirklich von der Korrektheit des erstellten Programms überzeugt sein. Dazu braucht man eigenes Denken und Verstehen, sowie verläßliche Dokumentation, nicht obskure Webseiten.

- Je länger man programmiert, desto mehr hat man schon verwendet, und kann es ohne langes Nachschauen anwenden.

  Gerade Ein-/Ausgabe ist ein Bereich, den man häufig benötigt. Ein Arbeitgeber (oder auch ein Übungsleiter in einer folgenden Vorlesung) hat Anspruch darauf, dass man Standardaufgaben zügig erledigen kann.

Kommandozeilen-Argumente (1)

- Wenn man Programme über eine Textschnittstelle (Terminal-Fenster, ssh, shell, Eingabeaufforderung, etc.) startet, gibt man in der Kommandozeile Argumente an, z.B.

```
cp Hello.java Test.java
```

- Unter Linux/UNIX kopiert dieser Befehl die Datei "`Hello.java`" in die Datei "`Test.java`".
- An erster Stelle steht der Name des Programms: "`cp`".

  Die ausführbare Datei (mit den Maschinenbefehlen) wird über den Suchpfad gefunden (wird mit "`echo $PATH`" angezeigt, probieren Sie auch "`which cp`").
- Danach folgenden die Argumente für dieses Programm, im Beispiel "`Hello.java`" als erstes Argument, und "`Test.java`" als zweites Argument.

  Der Kopier-Befehl muss ja wissen, welche Datei er kopieren soll.

Kommandozeilen-Argumente (2)

- Java-Programme können auf die Kommandozeilen-Argumente über den Parameter der `main`-Methode zugreifen:

```
class MyCP {
    public static void main(String[] args) {
        ...
    }
}
```

- Wenn man dieses Programm mit

```
java MyCP Hello.java Test.java
```

aufruft, ist `args.length == 2`, und es steht

  - "`Hello.java`" in `args[0]`,
  - "`Test.java`" in `args[1]`.

Kommandozeilen-Argumente (3)

- Die Argumente werden durch ein oder mehrere Leerzeichen vom Programmnamen und untereinander getrennt.

  Dies macht der Kommandozeilen-Interpreter. Dies ist keine Funktion von Java.

- Auch wenn man mehr Leerzeichen zwischen die Argumente schreibt, ändern sich die Werte in `args` nicht:

```
java MyCP      Hello.java                Test.java
```

  `args[0]` ist noch immer "`Hello.java`" und `args[1]`" wie vorher "`Test.java`". Die umgebenen Leerzeichen werden nicht Teil der Argumente.

- Wenn ein Argument Leerzeichen enthalten soll, müssen Sie es in Anführungszeichen einschliessen:

```
java MyCP 'Das ist das erste Argument' '2. Arg.'
```

  Die Anführungszeichen stehen nicht in `args[0]` und `args[1]` (nur Begrenzer).

Kommandozeilen-Argumente (4)

Beispiel:

- “Hello, World” mit Kommandozeilen-Argument (optional):

```
class HelloArg {
    public static void main(String[] args) {
        String name;
        if(args.length > 0)
            name = args[0];
        else
            name = "World";
        System.out.println("Hello, " +
                               name + "!");
    }
}
```

Kommandozeilen-Argumente (5)

- Dieses Programm kann in folgender Form ausgeführt werden:

  ```
  java HelloArg Nina
  ```

  Es gibt dann "`Hello, Nina!`" aus.

- Man kann das Programm aber auch ohne Argument aufrufen:

  ```
  java HelloArg
  ```

  Dann gibt es "`Hello, World!`" aus.

  Über `args.length` wird getestet, ob das Programm mit Argument aufgerufen wurde oder nicht.

- Nicht ganz in Ordnung ist, dass das Programm keine Fehlermeldung ausgibt, wenn es mit mehr als einem Argument aufgerufen wird.

  Die überzähligen Argumente werden einfach ignoriert. Es ist aber üblich, in diesem Fall eine Kurzanleitung auszugeben und das Programm zu beenden.

Kommandozeilen-Argumente (6)

- Die Benutzung von Werten aus der Kommandozeile, die direkt beim Starten des Programms angegeben werden, ist eine einfache Möglichkeit, Eingaben zu realisieren.
- Dies wird mächtig zusammen mit Konstrukten des Kommando-Interpreters (Shell), z.B. beim Auflösen von Wildcards wie "*", oder mit Shell-Skripten (Batch-Dateien).

  Auch das Programm `make` ist nützlich, um Befehle auszuführen, die abhängige Dateien erzeugen oder aktualisieren. Klassisch wurde es zum Aufruf des Compilers verwendet, aber es gibt auch viele andere Einsatzmöglichkeiten. Es kann auch Java-Programme aufrufen, mit zu erzeugender Datei in `args`.

- Wenn man das Programm in einer IDE ausführt, können Kommandozeilen-Argumente etwas umständlicher sein.

  Bei Eclipse eingeben unter "`Run`→`Run Configurations`→`Arguments`".

Exit-Codes (1)

- Es gibt auch eine einfache Kommunikationsmöglichkeit in der anderen Richtung (vom Java-Programm zur Shell bzw. zum aufrufenden Prozess):

```
System.exit(int n);
```

  beendet das Programm mit dem "exit code"/"status" `n`.

  Der Wert `0` bedeutet "ok", mit kleinen positiven Werten (bis 255) kann man verschiedene Fehler anzeigen. Negative Werte sollte man besser vermeiden. Wenn `main` normal zu Ende läuft, wird automatisch `0` geliefert, bei Abbruch durch Exception `1`. Unter Linux/UNIX enthält die Shell-Variable `$?` hinterher den Exit-Code (`echo $?`), bei Windows `%ERRORLEVEL%`.

- Die Klasse `System` bietet noch weitere Möglichkeiten.

  Z.B. Umgebungsvariablen abfragen, Java System Properties (Eigenschaften) abfragen/setzen, Laufzeit-Messungen.

  [http://docs.oracle.com/javase/7/docs/api/java/lang/System.html]

Exit-Codes (2)

```
class Square {
    public static void main(String[] args) {
        if(args.length != 1) {
            System.out.println(
                "Aufruf: java Square <zahl>");
            System.exit(2);
        }
        try {
            int n = Integer.parseInt(args[0]);
            System.out.println(n + "^2 = " + (n*n));
        } catch(NumberFormatException e) {
            System.out.println("Arg. ist kein int");
            System.exit(3);
        }
    }
}
```

Inhalt

1. Kommandozeile
2. Dateien
3. Streams
4. Reader/Writer
5. Formatierung
6. Scanner

Dateien (1)

- Die Klasse `File` im Paket `java.io` dient zur Repräsentation von Informationen über Dateien und Verzeichnisse.

  [http://docs.oracle.com/javase/7/docs/api/java/io/File.html]

- Beispiel:

```
import java.io.File;
class TestFile {
    public static void main(String[] args) {
        File f = new File("xyz.txt");
        if(f.exists())
            System.out.println("Datei existiert");
        else
            System.out.println("Gibt's nicht!");
    }
}
```

Dateien (2)

- Bei der Erzeugung wird im `File`-Objekt zunächst nur der Dateiname gespeichert.

  Man kann auch einen ganzen Pfad angeben, und es gibt auch Konstruktoren mit getrennten Argumenten für Verzeichnis und Datei.

- Die Klasse hat dann aber viele Methoden, mit denen man Eigenschaften der Datei (oder des Verzeichnisses) abfragen kann, oder auch Datei oder Verzeichnis anlegen kann.

- Die Klasse versucht, die Unterschiede zwischen Pfadnamen in verschiedenen Betriebssystemen zu verstecken, so dass Java-Programme möglichst portabel sind.

  Linux/Unix verwenden das Trennzeichen "/" und kein Laufwerk.

  Bei Windows ist das Trennzeichen "\", mit Laufwerk Präfix.

- Achtung: In String-Konstanten "\" doppelt schreiben: "\\".

Inhalt

1. Kommandozeile
2. Dateien
3. Streams
4. Reader/Writer
5. Formatierung
6. Scanner

Ein-/Ausgabe-Ströme: Übersicht (1)

- Man spricht von Eingabe-"Strömen" (engl. "Streams"), weil man genauen Quelle der Daten abstrahieren will:
  - von der Tastatur,
  - aus einer Datei,
  - über das Netzwerk,
  - aus einem String oder Zeichenarray (Variable),
  - von einem anderen Programm.
- Alle diese Datenquellen sollen mit der gleichen Schnittstelle angesprochen werden (entsprechend: Ausgabeströme).
- Die Java-Bibliothek ist auch so gemacht, dass vieles "zusammengesteckt" werden kann: Man kann leicht aus einem Eingabestrom einen machen mit Zusatzdiensten (s.u.).

Ein-/Ausgabe-Ströme: Übersicht (2)

- Da Java intern mit Unicode (UTF-16) arbeitet, muss man unterscheiden zwischen
  - `InputStream`: Quelle von Bytes / Bytestrom

    Bytes sind 8-Bit Einheiten und entsprechen Zahlen im Bereich 0 bis $255 = 2^8 - 1$.
  - `Reader`: Quelle von Zeichen / Zeichenstrom

    Die `char`-Werte in Java sind UTF-16 Codes (16 Bit) und entsprechen Zahlen von 0 bis $65\,535 = 2^{16} - 1$.
- Da man aus einem `InputStream` einen `Reader` machen kann, behandeln wir ihn hier zuerst.

  Dazu muss man die Codierung der Zeichen kennen, z.B. UTF-8 oder ISO 8859-1.
- Die gleiche Unterscheidung gibt es für Ausgaben zwischen `OutputStream` (Bytes) und `Writer` (Zeichen).

Input-Streams: Eingabe von Bytes (1)

- Das Programm hat bereits Zugriff auf einen `InputStream`, nämlich `System.in`.
- Normalerweise kommt diese Eingabe von der Tastatur, aber die Shell (Kommandointerpreter, Eingabeaufforderung) bietet Möglichkeiten, diese "Standard Eingabe" umzulenken:
  - Die Eingabe kann von einer Datei kommen:

    ```
    java MyProg <Test.txt
    ```

  - Die Eingabe kann Ausgabe eines anderen Programms sein:

    ```
    ls | java MyProg
    ```

  - Die Eingabe kann auch im Shellscript / Batch-Datei selbst stehen, das unser Programm aufruft:

    ```
    java MyProg <<ENDE
    ...
    ENDE
    ```

Input-Streams: Eingabe von Bytes (2)

- Bisher haben wir `System.in` nur benutzt, um die Eingabe für einen `Scanner` zu bekommen.

  Dies ist ein Beispiel, wie Funktionalität zusammengesteckt werden kann. Allerdings ist ein `Scanner` nicht selbst ein `InputStream`. Solche Möglichkeiten gibt es auch (s.u.).

- Man kann allerdings auch direkt Methoden der Klasse `InputStream` (im Paket `java.io`) benutzen.

  [http://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html]

- Die Methode `read` liefert das nächste Byte:

```
int   read()
      Reads the next byte of data from the input
      stream.
```

Input-Streams: Eingabe von Bytes (3)

- Wenn man die Methode `read()` aufruft, können verschiedene Dinge passieren:
  - Man bekommt sofort ein Byte (Zahl im Bereich `0` bis `255`)
  - man bekommt `-1`, um das Dateiende anzuzeigen,
  - der Aufruf erzeugt eine `IOException` (Fehler),
  - das Programm wartet auf eine Eingabe (Aufruf "blockiert").

    Anschließend erhält man dann eine der anderen drei Möglichkeiten. Man braucht diesen Fall im Programm also nicht extra zu behandeln, der Aufruf der Methode dauert nur etwas länger. Gelegentlich kann es aber Situationen geben, in denen Programme gegenseitig aufeinander warten (z.B. bei Kommunikation über eine Netzwerkverbindung). Wenn das Programm eine graphische Schnittstelle hat, muss die auch regelmäßig bedient werden (sonst wirkt es "abgestürzt").

Input-Streams: Eingabe von Bytes (4)

- Es auch eine Methode, um gleich ein ganzes Array von Bytes zu lesen:

```
int   read(byte[] b)
      Reads some number of bytes from the input stream
      and stores them into the buffer array b.
```

  Dies ist ein Beispiel für eine überladene Methode: Sie heisst genauso, unterscheidet sich aber durch den Parameter.

- Diese Methode liefert die Anzahl der gelesenen Bytes, bzw. `-1`, falls es kein Byte mehr gab und das Dateiende erreicht ist.

  Es wird nicht unbedingt das ganze Array gefüllt. Üblich ist, falls der Benutzer schon Zeichen eingegeben hat, nur diese Zeichen zu liefern, und nicht zu blockieren und auf weitere Zeichen zu warten. Wenn man mehr Zeichen will, muss man es noch einmal aufrufen. Die Dokumentation sagt dazu aber nichts aus (läßt unterschiedliche Implementierungen zu).

Input-Streams: Eingabe von Bytes (5)

- Es gibt schließlich noch eine Variante von `read`, mit der man ein Stück (Abschnitt) eines Arrays füllen kann:

```
int  read(byte[] b, int off, int len)
     Reads up to len bytes of data from the input
     stream into an array of bytes.
```

  Das erste gelesene Byte wird nach `b[off]` gespeichert. Der Rückgabewert ist wieder die Anzahl der tatsächlich gelesenen Bytes. Diese Methode ist nützlich, wenn man über das Netzwerk oder vom Benutzer nur die gerade verfügbaren Bytes bekommen hat, und erneut lesen will (dann blockierend), um eine bekannte Anzahl Bytes zu erhalten. Wenn man als neuen Offset den alten plus die gelesenen Bytes wählt, werden die nächsten Bytes direkt im Anschluss in das Array geschrieben. Das macht man in einer Schleife.

- Mit der Methode `available()` kann man eine Schätzung bekommen, wie viele Bytes ohne Warten verfügbar sind.

Input-Streams zum Lesen von Dateien (1)

- Die Klasse `InputStream` hat selbst keinen Konstruktor (sie ist "abstrakt").

  Sie dient nur als gemeinsame Schnittstelle für unterschiedliche Arten von Eingabeströmen.

- Objekte speziellerer Klassen, z.B. `FileInputStream`, können aber einer `InputStream`-Variablen zugewiesen werden:

```
InputStream in = new FileInputStream("test.txt");
```

- Hiermit wird die angegebene Datei eröffnet, und man hat einen Eingabestrom, aus der Datei liest.

- Es gibt noch weitere Arten spezieller Eingabeströme, z.B. den `ByteArrayInputStream`.

  Dieser liest die Daten aus einer Variablen vom Typ `byte[]`.

Input-Streams zum Lesen von Dateien (2)

- Die Klasse `FileInputStream` ist eine "Subklasse" (oder "Unterklasse") der Klasse `InputStream`.

  Es gibt später ein eigenes Kapitel über Unterklassen.

- Das bedeutet: Die Menge der `FileInputStream`-Objekte ist eine Teilmenge der Menge der `InputStream`-Objekte.
- Also: `FileInputStream`s sind spezielle `InputStream`s.
- Daher kann man alles, was man mit einem allgemeinen `InputStream` machen kann, insbesondere auch mit einem `FileInputStream` machen.

  Z.B. kann man die `read`-Methoden auch für `FileInputStream`-Objekte aufrufen, und man kann dem Konstruktor des Scanners statt einem `InputStream` auch ein `FileInputStream`-Objekt übergeben.

Input-Streams zum Lesen von Dateien (3)

- Die Klasse `FileInputStream` hat mehrere Konstruktoren, u.a.
  - mit einem Dateinamen als `String` (wie oben gezeigt),
  - mit einem `File`-Objekt (enthält auch einen Pfadnamen).
- Die Konstruktoren erzeugen eine `FileNotFoundException`, wenn es die Datei nicht gibt.

  Man muss diese Exception mit `catch` behandeln (oder deklarieren, dass `main` sie weitergeben kann).
- Es empfiehlt sich, am Ende die Datei (den Eingabestrom) mit der Methode `close()` wieder zu schließen.

  Geöffnete Dateien belegen Resourcen des Betriebssystems, die begrenzt sein können (z.B. maximal 15 Dateien gleichzeitig offen für einen Prozess). Beim Programmende werden die Dateien automatisch geschlossen, aber mindestens wenn das Programm länger läuft, sollte man die Dateien schließen.

Input-Streams zum Lesen von Dateien (4)

```
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

class DateiLesen {
    public static void main(String[] args) {
        InputStream in;
        try {
            in = new FileInputStream("test.txt");
        } catch(FileNotFoundException e) {
                System.out.println(e.getMessage());
                return;
        }
```

Input-Streams zum Lesen von Dateien (5)

```
            Scanner scan = new Scanner(in);
            while(scan.hasNext()) {
                int n = scan.nextInt();
                System.out.println(n);
            }

            try {
                in.close();
            } catch(IOException e) {
                System.out.println(e.getMessage());
                return;
            }
        }
    }
```

Output-Streams: Ausgabe von Bytes (1)

- Das Gegenstück zur Ausgabe ist ein `OutputStream`.

  [http://docs.oracle.com/javase/7/docs/api/java/io/OutputStream.html]

- Er hat u.a. eine Methode zur Ausgabe eines Bytes:

  ```
  void write(int b)
      Writes the specified byte to this output stream.
  ```

  Falls der übergebene `int`-Wert größer als 255 ist, werden die oberen 24 Bit einfach ignoriert.

- Die Methode kann eine `IOException` auslösen, die behandelt werden muss.

- Es gibt wie bei der Eingabe auch Varianten mit einem Byte-Array und einem Abschnitt aus einem Byte-Array.

Output-Streams: Ausgabe von Bytes (2)

- Es ist möglich, dass mit `write` ausgegebene Daten noch in einem Zwischenspeicher stehen (also nicht wirklich sofort ausgegeben wurden). Mit der Methode `flush()` kann man die gepufferten Daten wirklich schreiben lassen.

  Wenn man z.B. einen Prompt ausgibt und danach etwas liest, kann das wichtig sein. Auch wenn ein Programm abbricht (z.B. wegen eines Fehlers) kann es verwirrend sein, wenn eigentlich ausgegebene Daten noch nicht auf dem Bildschirm oder in der Datei stehen. Das ist aber selten.

- Es gibt einen `FileOutputStream` für Dateien.

  Zu verwenden wie der `FileInputStream` (s.o.).

- Man sollte mindestens Datei-Ausgabeströme am Ende mit `close()` schließen.

Print-Streams: Ausgabe von Daten (1)

- System.out bietet mehr Möglichkeiten als ein normaler OutputStream, es ist ein PrintStream.

  [http://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html]

- PrintStream ist eine spezielle Art von OutputStream.

- Wenn man einen gewöhnlichen OutputStream hat, kann man daraus einen PrintStream machen:

```
OutputStream out =
        new FileOutputStream("ausgabe.txt");
PrintStream p = new PrintStream(out, false);
```

  Mit dem zweiten Parameter (autoflush) kann man festlegen, ob die Ausgabe am Ende jeder Zeile wirklich durchgeführt wird, und nicht ggf. im Puffer verbleibt (genaue Regeln siehe Doku). Für Dateien braucht man das eher nicht. Es gibt noch eine Reihe weiterer Konstruktoren, man muss nicht über den fileOutputStream gehen, sondern kann direkt den Dateinamen angeben.

Print-Streams: Ausgabe von Daten (2)

- U.a. hat ein `PrintStream` zusätzliche Methoden zur Ausgabe verschiedener Datentypen:
  - `void println()`
  - `void println(boolean x)`
  - `void println(char x)`
  - `void println(char[] x)`
  - `void println(double x)`
  - `void println(float x)`
  - `void println(int x)`
  - `void println(long x)`
  - `void println(Object x)`
  - `void println(String x)`

Print-Streams: Ausgabe von Daten (3)

- Der `PrintStream` übersetzt also Aufrufe auf höherer Ebene in das Schreiben von Bytes in den benutzen `OutputStream`.

  Ganz ähnlich wie die Klasse `Scanner` für Eingaben.

- `PrintStream` ist auch dadurch bequemer als `OutputStream`, weil es `IOException` abfängt.

  Die Methode `write` von `OutputStream` kann `IOException` erzeugen, die Methoden von `PrintStream` können das nicht.

- Man muss die Aufrufe der `PrintStream`-Methoden daher nicht in einem `try`-Block einschließen.

  Man kann mit der folgender Methode prüfen, ob bei den intern verwendeten Operationen auf dem `OutputStream` eine Exception aufgetreten ist:

  ```
  boolean checkError()
          Flushes the stream and checks its error state.
  ```

Ausgabe-Umleitung

- Der Kommandointerpreter (die Shell) bietet die Möglichkeit, die Standard-Ausgabe des Programms umzuleiten:
  - in eine Datei:

    ```
    java MyProg arg1 arg2 >ausgabe.txt
    ```

  - in die Standard-Eingabe eines anderen Programms:

    ```
    java MyProg arg1 arg2 | more
    ```

- Das Java-Programm ist davon nicht beeinflusst.
- Es ist üblich, Fehlermeldungen nicht auf `System.out` auszugeben, sondern auf `System.err`.

  Dies ist ein zweiter `PrintStream`, die Ausgabe erfolgt normalerweise auch auf den Bildschirm. Dadurch können die wichtigen Meldungen nicht unbemerkt in der Datei verschwinden, wenn die normale Ausgabe umgeleitet ist. (Man kann auch den Standard-Fehlerkanal umleiten, aber dann macht man es bewusst.)

Inhalt

1. Kommandozeile
2. Dateien
3. Streams
4. Reader/Writer
5. Formatierung
6. Scanner

Reader/Writer: Einführung (1)

- Die technische Basis (Dienst des Betriebssystems) ist die Ein-/Ausgabe von Bytes: `InputStream`/`OutputStream`.

  Da das Betriebssystem nichts über die Bedeutung der Daten weiss, ist das angemessen. Z.B. werden auch Bilder oder Maschinenbefehle (ausführbare Programme) in Dateien gespeichert (binäre Daten im Ggs. zu Textdateien).

- Häufig will man aber Texte (Folgen von Zeichen) eingeben oder ausgeben.

- Heute ist Unicode sehr verbreitet, und für Java Standard. Man muss aber unterscheiden zwischen
  - dem Zeichenvorrat mit den "Code Points" (Nummern) der Zeichen (d.h. der Zeichentabelle)
  - der Codierung dieser 16 Bit oder 21 Bit Zahlen als Folge von Bytes in einer Datei.

Reader/Writer: Einführung (2)

- Als Codierung gibt es viele Möglichkeiten, z.B.
  - ISO Latin 1 (ISO/IEC 8859-1)

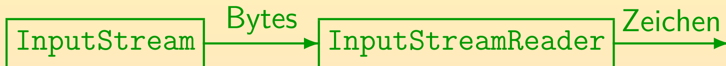
    Dies benötigt ein Byte pro Zeichen. Damit können natürlich nicht alle Unicode-Zeichen repräsentiert werden, aber es reicht für die in Deutschland üblichen Zeichen. Das Euro-Symbol wurde allerdings erst später eingeführt, das gibt es nur in der neueren Variante 8859-15.
  - UTF-8

    Die benötigt 1–3 Bytes pro Zeichen. Die klassischen ASCII-Zeichen benötigen 1 Byte, die deutschen Umlaute 2.
  - UTF-16

    Dies belegt 2 Byte für jedes Zeichen der "Basic Multilingual Plane" (alle normalen Zeichen für die meisten Weltsprachen). Sehr seltene Zeichen brauchen 4 Byte. Es gibt noch zwei unterschiedliche Varianten mit niederwertigem oder höherwertigem Byte der 16 Bit zuerst (mit "Byte Order Mark" ist automatische Erkennung möglich).

Reader/Writer: Einführung (3)

- Mit den Reader/Writer-Klassen kann man sich die Umwandlung zwischen Bytes und Zeichen abnehmen lassen:



- Ein InputStreamReader ist ein spezieller Reader, der seine Eingabe aus einem InputStream bezieht.

  [http://docs.oracle.com/javase/7/docs/api/java/io/InputStreamReader.html]

- Man kann den gewünschten Zeichensatz (Codierung) beim Erzeugen eines Objektes dieser Klasse angeben:

```
InputStreamReader in =
    new InputStreamReader(System.in, "UTF-8");
```

  Zeichensatz-Namen sind u.a.: "US-ASCII", "ISO-8859-1", "UTF-8", "UTF-16", "UTF-16BE" ("big endian", höherwertiges Byte/Ende zuerst), "UTF-16LE".

Reader/Writer: Einführung (4)

- Man muss nicht unbedingt einen Zeichensatz (Codierung) angeben, dann wird der systemabhängige Default verwendet:

```
InputStreamReader in =
    new InputStreamReader(System.in);
```

- Es gibt eine Klasse `CharSet` (im Paket `java.nio.charset`) mit vielen weiteren Informationen zur Zeichencodierung.

  Ein Objekt dieser Klasse kann man statt des Zeichensatz-Namens bei der Erzeugung des `InputStreamReader`-Objektes angeben.
  [http://docs.oracle.com/javase/7/docs/api/java/nio/charset/Charset.html]

- Wenn man ein Objekt vom Typ `InputStreamReader` hat, kann man den verwendeten Zeichensatz (Codierung) abfragen:

```
System.out.println(in.getEncoding());
```

Reader: Methoden (1)

- Die Klasse `Reader` (im Paket `java.io`) bietet praktisch die gleichen Methoden wie `InputStream` an, nur steht hier der Typ `char` statt `byte`:

  [http://docs.oracle.com/javase/7/docs/api/java/io/Reader.html]

  - `int read()`: Liest ein Zeichen.

    Der Rückgabetyp ist `int`, weil man noch den zusätzlichen Wert `-1` für das Dateiende benötigt. Normale Zeichen werden als Zahlen im Bereich `0` bis 65 535 geliefert.

  - `int read(char[] cbuf)`: Liest Zeichen in ein Array.

    Es werden die gelesene Anzahl Zeichen zurückgeliefert, oder `-1` am Dateiende, wenn gar keine Zeichen mehr gelesen wurden. Das Array wird nicht unbedingt vollständig gefüllt, selbst wenn das Dateiende noch nicht erreicht ist (z.B. nur eine Zeile).

  - `int read(char[] cbuf, int off, int len)`

Reader: Methoden (2)

- Weitere Methoden der Klasse `Reader`:
  - `boolean ready()`: Gibt es schon Daten?

    Falls `true`, wird der nächste Leseaufruf nicht blockieren. Dies ist ein Unterschied zu `inputStream`: Dort liefert `available()` eine Anzahl Bytes.
  - `void close()`: Schließt Zeichenstrom.

    Dabei werden eventuell belegte Resourcen freigegeben. Anschließend kann man natürlich nicht mehr von dem Objekt lesen.
- Alle genannten Methoden können eine `IOException` liefern.

  Das ist eine “checked exception”, d.h. man muss die Methoden-Aufrufe in einen `try`-Block einschließen (mit einem passenden `catch`), oder man muss deklarieren, dass die aktuelle Methode (`main`) eine `IOException` liefern kann.
- Da `InputStreamReader` ein spezieller `Reader` ist, hat er auch alle diese Methoden.

Writer: Methoden (1)

- Die Klasse `Writer` (im Paket `java.io`) erlaubt entsprechend das Schreiben von Zeichen:

  [http://docs.oracle.com/javase/7/docs/api/java/io/Writer.html]

  - `void write(int c)`: Schreibt ein Zeichen.

    Das zu schreibende Zeichen steht in den unteren 16 Bit des übergebenen Wertes, die oberen 16 Bit werden ignoriert. Selbstverständlich kann man auch direkt Werte vom Typ `char` übergeben, aber so passt das `write` zum `read` (man braucht keinen expliziten Typ-Cast, wenn man das gelesene Zeichen wieder ausgeben will). Auch Rechnen mit den Codes erfordert keinen Cast.

  - `void write(char[] cbuf)`: Schreibt Zeichen aus Array.

    Hier muss das Array ganz gefüllt sein.

  - `void write(char[] cbuf, int off, int len)`

    Hier kann man angeben, welches Stück des Arrays gefüllt ist.

Writer: Methoden (2)

- Natürlich kann man auch Zeichenketten ausgeben:
  - `void write(String str)`: Zeichenkette ausgeben.
  - `void write(String str, int off, int len)`:
    Stück einer Zeichenkette ausgeben.
- Ausgabe mit Rückgabe des Objektes:
  - `Writer append(char c)`: Zeichen ausgeben.

    `out.append(c)` bewirkt das gleiche wie `out.write(c)` und liefert `out` zurück, so kann man Ketten `out.append(...).append(...)` schreiben.
  - `Writer append(CharSequence csq)`

    Eine Zeichenkette (vom Typ `String`) ist eine spezielle `CharSequence`, kann also hier verwendet werden. Aber z.B. auch `StringBuilder`.
  - `Writer append(CharSequence csq, int start, int end)`

Writer: Methoden (3)

- Weitere Methoden:
  - `void flush()`: Puffer leeren.

    Nach den `write()`- und `append()`-Aufrufen können die ausgegebenen Zeichen noch in einen Zwischenspeicher (Puffer) stehen. Diese Maßnahme dient der Effizienzsteigerung, so dass z.B. nicht einzelne Zeichen ans Betriebssystem übergeben werden müssen. Wenn man sicher gehen will, dass zumindest Java die Zeichen nicht mehr puffert, muss man `flush()` aufrufen. Eventuell hat das Betriebsssystem allerdings nochmal einen Puffer, so dass nicht 100% garantiert ist, dass die Daten tatsächlich anschließend auf der Festplatte stehen.
  - `void close()`: Ausgabe schließen.

    Dies macht automatisch ein `flush()`. Natürlich kann man anschließend die `write()`-Methoden nicht mehr aufrufen.
- Alle Methoden können eine `IOException` erzeugen.

Spezielle Writer

- `FileWriter`: Ausgabe in Datei.

  [http://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html]

  Das kann man natürlich auch mit einem `OutputStreamWriter` machen, wenn der `OutputStream` in eine Datei ausgibt. Tatsächlich ist `FileWriter` ein spezieller `OutputStreamWriter`. Aber hier kann man direkt im Konstruktor ein `File`-Objekt oder einen Dateinamen (`String`) angeben, und muss nicht erst manuell den `OutputStream` erzeugen.

- `StringWriter`: Ausgabe in Zeichenkette.

  [http://docs.oracle.com/javase/7/docs/api/java/io/StringWriter.html]

  Mit der Methode `toString()` kann man die bisher durch die Ausgaben erstellte Zeichenkette abfragen.

- `PrintWriter`: Ausgabe primitiver Datentypen.

  [http://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html]

  Dies hat die vom `PrintStream` bekannten Methoden `println()` etc.

Spezielle Reader

- `FileReader`: Eingabe von Datei.

  [http://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html]

  Siehe `FileWriter`: Bei Objekterzeugung gibt man Dateinamen oder `File` an.

- `StringReader`: Eingabe aus Zeichenkette.

  [http://docs.oracle.com/javase/7/docs/api/java/io/StringReader.html]

  Man gibt beim Erzeugen des Objektes eine Zeichenkette (`String`) an, aus der die Eingabe gelesen wird.

- `BufferedReader`: Eingabe mit Pufferung.

  [http://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html]

  Bei der Erzeugung des Objektes gibt man einen anderen `Reader` als Quelle der Zeichen an. Der `BufferedReader` liest aus dieser Quelle immer größere Stücke in einen internen Zwischenspeicher (Puffer), aus dem man dann die einzelnen Zeichen holen kann. Dies dient der Effizienzsteigerung. Außerdem gibt es eine Methode, um ganze Zeilen zu lesen (siehe nächste Folie).

Eingabe mittels BufferedReader (1)

- In den meisten Java-Kursen wird vermutlich die Eingabe mit einem `BufferedReader` gemacht (nicht mit `Scanner`).

  Mir schien der `Scanner` einfacher zu sein, aber die Tutoren sind nach Korrektur der Hausaufgaben anderer Meinung. Den `BufferedReader` gibt es seit Java Version 1.1, den `Scanner` erst seit 1.5, was eventuell auch die größere Verbreitung des `BufferedReader`s erklärt.

- Um einen `BufferedReader` zu erzeugen, muss man schon einen `Reader` haben. Also erzeugt man aus `System.in` (einem `InputStream`) zuerst einen `InputStreamReader`.

- Das geht in einer Anweisung so:

```
BufferedReader inBuf = new BufferedReader(
                 new InputStreamReader(System.in));
```

Eingabe mittels BufferedReader (2)

- Der BufferedReader erlaubt es, die nächste Eingabezeile zu lesen. Dabei muss IOException behandelt werden:

```
try {
    String eingabe = inBuf.readLine();
    int n = Integer.parseInt(eingabe);
    System.out.println("n^2 = " + n*n);
} catch(NumberFormatException e) {
    System.out.println(
        "Eingabeformat falsch");
} catch(IOException e) {
    System.out.println(
        "Fehler beim Einlesen");
}
```

Inhalt

1. Kommandozeile
2. Dateien
3. Streams
4. Reader/Writer
5. Formatierung
6. Scanner

Ausgabe-Formatierung (1)

- Ausgabeformatierung wird u.a. in folgenden Fällen benötigt:
  - Bei Gleitkomma-Zahlen möchte man häufig die Anzahl der Nachkomma-Stellen vorgeben.
  - Wenn man eine tabellarische Ausgabe haben will, müssen alle Werte einer Spalte die gleiche Breite aufweisen.
  - Für Zeit-/Datumswerte gibt es viele verschiedene Formate.
- Z.B. wird so ein `double`-Wert `x` mit 6 Stellen insgesamt und 2 Nachkomma-Stellen ausgegeben:

```
System.out.format("%6.2f", x);
```

- Oder Formatierungs-Ergebnis als String bestimmen:

```
String s = String.format("%6.2f", x);
```

Ausgabe-Formatierung (2)

- Format-Spezifikationen sind in der Klasse `Formatter` (Paket `java.util`) beschrieben.

  [http://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html]

- Formatspezifikation beginnen mit einem Prozentzeichen und enden mit einem Buchstaben, der Datentyp und Ausgabeformat anzeigt, z.B.
  - `f`: Gleitkommazahl mit Dezimalpunkt/Komma.
  - `e`: Gleitkommazahl in wissenschaftlicher Notation.
  - `d`: Ganze Zahl in Dezimalschreibweise.
  - `x`: Ganze Zahl in Hezadezimalschreibweise.
  - `c`: Zeichen (gemäß Unicode).
  - `s`: String oder beliebiger in String umzuwandelnder Typ.

Ausgabe-Formatierung (3)

- Zwischen dem %-Zeichen und dem Buchstaben für die Konvertierung kann man Folgendes angeben:
  - einen Argumentindex,

    Das braucht man sehr selten (wenn man mehrere auszugebene Werte umordnen will, z.B. bedeutet 1$ das erste Argument).
  - Zusatzoptionen ("flags"), z.B. 0 für führende Nullen, - für linksbündige Ausgabe, + für ggf. auch +-Vorzeichen,
  - die (minimale) Ausgabebreite (Anzahl Zeichen),
  - einen Punkt "." und die Anzahl Nachkommastellen.

    Natürlich nur bei Gleitkomma-Zahlen.
- Zeichen außerhalb einer Format-Spezifikation werden einfach in die Ausgabe kopiert.

  Wenn man ein Prozentzeichen braucht, muß man es verdoppeln: "%%".

Ausgabe-Formatierung (4)

- Es ist besonders praktisch, dass die `format`-Funktion beliebig viele Werte beliebiger Typen akzeptiert. Beispiel:

```
System.out.format("%20s: %3d.%02d Euro%n",
                  bez, euro, cent);
```

  - Dies gibt zuerst String in der Variablen `bez` aus,

    Rechtsbündig in einem Feld der Breite 20, d.h. links mit Leerzeichen aufgefüllt, so dass es 20 Zeichen werden. Linksbündig mit "`%-20s`".

  - dann einen Doppelpunkt "`:`" und ein Leerzeichen,
  - die ganze Zahl in der Variablen `euro`,

    nach links mit Leerzeichen zur Breite 3 aufgefüllt,

  - zweistellig den `cent`-Betrag, ggf. links mit Nullen aufgefüllt,
  - den String "` Euro`",
  - den plattform-spezifischen Zeilenumbruch.

Inhalt

1 Kommandozeile

2 Dateien

3 Streams

4 Reader/Writer

5 Formatierung

6 Scanner

Scanner (1)

- Die Klasse `Scanner` im Paket `java.util` erlaubt
  [http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html]
  - einfache Eingaben von Datenwerten (z.B. `int`),
  - Aufspaltung der Eingabe in Stücke mit wählbaren Trennern,
  - vereinfachten Umgang mit Exceptions (keine `IOException`).
- Wie schon oft benutzt, kann man einen Scanner anlegen, der die Eingabe von einem `InputStream` liest:

```
Scanner scan = new Scanner(System.in);
```

- Falls erwünscht, kann man explizit eine Codierung wählen:

```
Scanner scan = new Scanner(System.in, "UTF-8");
```

- Es gibt viele weitere Möglichkeiten für Datenquellen, z.B. `File`, `Reader`, `String`.

Scanner (2)

- Die Eingabe wird in Stücke aufgespalten, zwischen denen "Trenner" (engl. "delimiter") stehen.
- Solange man das Muster für die Trenner nicht umgestellt hat, ist es Leerplatz (engl. "whitespace").

  Eine Folge von Leerzeichen, Tabulatorzeichen, Zeilenumbrüchen, "Form Feed".
- Wenn man mit

```
int n = scan.nextInt();
```

  eine ganze Zahl lesen will,
  - darf der Nutzer zuerst Leerplatz eingeben (ignoriert),

    Man kann natürlich auch direkt mit einer Ziffer starten.
  - dann ein optionales Vorzeichen und eine Ziffernfolge,
  - danach muss wieder Leerplatz folgen.

Scanner (3)

- Wenn statt einer Ziffernfolge z.B. Buchstaben eingegeben werden, gibt es eine "`InputMismatchException`".

  Exception auch aus Paket "`java.util`". Das gleiche passiert, wenn direkt nach der Ziffernfolge ein Zeichen kommt, das nicht ins Zahlformat passt, z.B. ein "`;`". Als fortgeschrittener Nutzer kann man das Muster für die Trenner umstellen.

- Man kann aber mit der Methode `hasNextInt()` vorher prüfen, ob das folgende Stück der Eingabe als Zahl interpretiert werden kann.

- Falls nicht, sollte man eine Fehlermeldung ausgeben, und mit `nextLine()` die aktuelle Eingabezeile "weglesen" (überspringen): Beispiel auf nächster Folie.

  Die fehlerhaften Zeichen stehen ja noch in der Eingabe. Wenn man sie nicht entsorgt, bevor man einen neuen Versuch startet, erhält man eine Endlosschleife.

Scanner (4)

- Beispiel: Eingabe einer ganzen Zahl mit Möglichkeit zur Korrektur falscher Eingaben:

```
System.out.print("Bitte ganze Zahl eingeben: ");
while(!scan.hasNextInt()) {
    scan.nextLine(); // Eingabezeile entsorgen
    System.out.println("Keine Zahl!");
    System.out.print("Neue Eingabe: ");
}
int n = scan.nextInt();
```

- Falls der Benutzer `Ctrl+D` eingibt (Dateiende), erhält man eine `NoSuchElementException`.

  Die Exception stammt auch aus dem Paket `java.util`. Alternativ kann man vor `nextLine()` mit `hasNextLine()` testen, ob es noch eine Zeile gibt.

Scanner (5)

- Allgemein kann es mit `Scanner` gelegentlich ein unerwartetes Verhalten geben, weil Daten noch im Puffer stehen:
  - Wenn man mit `nextInt()` eine Zahl liest und dann mit `nextLine()` eine Zeile, bekommt man nicht eine neue Zeile, sondern den Rest von der Zeile mit der Zahl.

    Ein Aufruf von `nextInt()` entfernt nur die Ziffern aus der Eingabe, der anschließende Zeilenumbruch steht noch da. Wenn man noch eine Zahl liest, ist das kein Problem, weil das nächste `nextInt()` Leerplatz vor seinen Ziffern wegliest. Wenn man nun aber mit `nextLine()` eine ganze Zeile lesen will, bekommt man zuerst eine leere Zeichenkette (den Rest der aktuellen Zeile). Man muss `nextLine()` dann nochmals aufrufen. Eigentlich ist `nextLine()` auch eher für Aufräumarbeiten gedacht, so wie oben gezeigt. Es passt nicht in das wortbasierte Konzept des Scanners. Wenn man zeilenweise Eingabe will, ist ein `BufferedReader` üblich.

Scanner (6)

- Methoden der Klasse Scanner (Auszug):
  - `boolean hasNext()`: Gibt es noch ein "Wort"?
  - `String next()`: Liefert nächstes Wort (Token).

    Das Wort wird bei allen `next`-Methoden aus der Eingabe entfernt (ist dann gelesen).
  - `boolean hasNextInt()`: Kann nächstes Wort als `int` interpretiert werden?

    Für diese und die folgende Methode gibt es auch Varianten für viele andere Datentypen: `boolean`, `byte`, `short`, `long`, `float`, `double`, `BigDecimal`, `BigInteger`. Beispiel: `hasNextDouble()`.
  - `int nextInt()`: Liefert nächstes Wort als `int`-Wert.
  - `String nextLine()`: Liest bis zum Ende der aktuellen Zeile.

    Der zurückgelieferte String enthält die weggelesenen Zeichen, aber nicht das Zeilenende. Das ist aber auch weggelesen.

Scanner (7)

- Das Zahlformat, das der `Scanner` akzeptiert, hängt von den Einstellungen für Sprache und Region ("Locale") ab.
- Bei deutscher Spracheinstellung darf man in ganzen Zahlen Dreier-Gruppen von Ziffern durch "`.`" trennen (Tausender):

  `1.234.567` (= 1 234 567)
- Statt des in Java-Konstanten verwendeten Dezimalpunktes muss man das in Deutschland übliche Komma "`,`" schreiben.
- Wenn man das nicht will, kann man auf die amerikanische Schreibweise umschalten:

```
scan.useLocale(Locale.US);
```

[http://docs.oracle.com/javase/7/docs/api/java/util/Locale.html]

Man braucht: `import java.util.Locale;` Es gibt auch `Locale.GERMANY`.

Scanner (8)

- Der Scanner fängt `IOException` ab, dadurch muss man diese Exception nicht selbst behandeln.

  Falls die Exception auftritt, nimmt der Scanner an, dass das Ende der Eingabe erreicht ist. Wenn man prüfen will, ob eine `IOException` aufgetreten ist, kann man die letzte solche Exception mit folgender Methode abfragen:

  `IOException ioException()`

  Wenn kein solcher Fehler aufgetreten ist, wird `null` zurückgeliefert.

- Wenn man beim Erzeugen des Scanners eine Datei geöffnet hat (z.B. durch Übergabe eines `File`-Objektes), sollte man diese am Ende wieder schließen mit der Methode

  ```
  void close()
  ```

  Dies schließt den `Scanner` und die benutzte Datenquelle.