

Java

Matthias Bethke bethke@linguistik.uni-erlangen.de

Linguistische Informatik
Universität Erlangen-Nürnberg

Sommersemester 2006

- Kenntnisstand der Teilnehmer?
- Klausur oder Übungsaufgaben?
- Entwicklungsumgebung: EMACS/vi + Kommandozeile oder Eclipse?

- Allgemeine Textmanipulation (StringBuffer, Vector, Hashtable, ...)
- Tokenizer
- Reguläre Ausdrücke
- Korpuslinguistisches: Frequenzlisten, KWIC
- Parsen und Ausgeben von Markup-Sprachen (HTML, XML)
- Grafische Oberflächen?

EMACS ist neben **vi** der Standardeditor unter Linux.

- Start aus der Shell: `emacs filename &`
- Öffnen einer Datei: `C-x C-f` (d.h. `Strg` und `x` zusammen drücken, dann `Strg` und `f` zusammen)
- Speichern der aktuellen Datei: `C-x C-s`
- Syntax-Highlighting anschalten: `A-x font-lock-mode` (d.h. `Alt+x`, dann „font-lock-mode“ eingeben, `Return`)

Java-Programme werden in Textdateien mit der Erweiterung „.java“ gespeichert. In jeder Datei darf nur eine **public**-Klasse vorkommen, die genauso heißen muss wie die Datei.

Achtung: es wird zwischen Groß- und Kleinschreibung unterschieden!

Übersetzen aus dem *Quellcode* in den *Bytecode* der virtuellen Maschine („VM“):

```
javac MeinProgramm.java
```

Ausführen des Programms durch Starten der VM mit dem Namen der kompilierten Klasse:

```
java MeinProgramm
```

Beispiel: HelloWorld

Das üblicherweise erste Programm in jeder Programmiersprache!

```
public class HelloWorld {  
    public static void main(String [] args) {  
        System.out.println ("Hello, □world!");  
    }  
}
```

Geben Sie das Programm mit dem EMACS ein, speichern Sie es, und führen Sie es nach der Übersetzung in Bytecode aus!

Elemente von Computerprogrammen

- Variable = Datenfelder
- Konstante = Datenwerte
- Anweisungen = Aktionen
- Dateien zur Ein- und Ausgabe von Daten

Ausführung in den meisten Programmiersprachen (auch Java) in der Reihenfolge der Anweisungen im Programm.

Datentypen (1)

Datentypen geben an, welche Art von Daten in einer Variablen oder Konstanten gespeichert ist.

- Speicherbedarf und Genauigkeit in Java unabhängig vom Computersystem
- Unterscheidung: *primitive Datentypen* und *Objekt-Typen*
- Primitive Typen kennen keine Attribute und Methoden (Verwaltungsaufwand, Geschwindigkeit!)

Datentypen (2)

Primitive Datentypen

- ganze Zahlen: **byte** , **short** , **int** , **long**
- Fließkommazahlen: **float** , **double**
- logische Werte: **boolean** (**true** oder **false**)
- Unicode-Zeichen **char**

Objekt-Typen

- Unicode-Zeichenketten (~ Texte): **String**
- Alle als Klassen definierten Typen

- Einfache Ganzzahlen haben den Typ **int** und werden dezimal interpretiert, Suffix „L“ für Typ **long**. Z.B.: 42, 1234L
- Oktalzahlen mit Präfix „0“; Hexadezimalzahlen mit „0x“. Z.B.: 020, 0xfe.
- Zahlen mit Dezimalpunkt oder Exponent („E“) sind vom Typ **double**, Suffix „F“ für Typ **float**. Z.B.: 3.14159, 1.0E9, 12.345F
- Logische Konstanten: **true** und **false**
- Zeichenkonstanten in Apostrophen, z.B.: 'a'
- Zeichenkettenkonstanten in Anführungszeichen. Z.B. "abc", "Irgendein_□Text"

Variablen- und Konstantendeklaration

- Ein Datenfeld: `typ name;`
- Mehrere Variable: `typ name1 , name2 , name3;`
- Wertzuweisung: `name = wert;`
- Deklaration mit Anfangswert:
`typ name = wert;`
- Das Wort **final** vor der Typangabe macht aus der Variablen eine Konstante. Der Wert kann dann nachträglich nicht mehr verändert werden. Konventionell werden Konstanten mit Blockbuchstaben bezeichnet.
Beispiel: **final double** PI = 3.14159;

- Objektorientierte Programmierung: kombiniert Daten und Funktionen in Objekten.
- Eine *Klasse* stellt eine Art abstrakten Bauplan für *Objekte* dar.
- Klassen haben Eigenschaften oder *Attribute* und können damit verschiedene Aktionen oder *Methoden* ausführen.
- Von einer Klasse können beliebig viele konkrete Objekte erzeugt werden, die auch *Instanzen* der Klasse genannt werden.

Z.B. ist „Matthias“ ein Objekt der Klasse „Mensch“, Attribute könnten „Größe“ oder „Gewicht“ sein, Methoden etwa „sprechen“ oder „essen“.

Anlegen von Objekten

- Eine Variable eines Objekttyps kann *Referenzen* auf Objekte dieses Typs enthalten:

```
ClassName name;
```

z.B. `String s;`

- Es können direkt Konstante zugewiesen werden, z.B.

```
String s = "Hallo!";
```

- Anlegen neuer Objekte mit **new**:

```
ClassName name = new ClassName;
```

z.B. `String s = new String("Hallo!");`

Ausdrücke und Operatoren

Mit einem oder mehreren Operatoren verbundene Variable oder Konstante heißen *Ausdruck*. Die wichtigsten Operatoren:

- Mathematische: + - * / \%
- Logische: < > <= >== == != ! \&\& ||
- String-Operatoren: + equals¹
- Objekt-Operatoren: **new** .

¹Eigentlich eine Methode, wird hier nur wegen der Ähnlichkeit mit dem '=='-Operator in anderen Sprachen mit aufgeführt

Ausdrücke und Operatoren (Beispiele)

```
int i=42, j=2; k=3, x;  
String s, t="Hallo";  
boolean b;  
    x = i * j - k * j    // Operator-Vorrang  
    x = (i+j)*k        // Klammerung  
    x = i % (j*k)      // a modulo (b mal c)  
    b = j > k  
    s = new String(t)  
    b = s == t         // false , ungleiche Objekte  
    b = s.equals(t)   // true , gleicher Text  
    b = (i>j) && (i<k)
```

Arrays

Ein *Array* ist eine Menge von Variablen gleichen Typs. Arrays haben eine feste Länge und jedes Element ist über einen *Index* ansprechbar.

Anlegen einer Variablen, die eine Referenz auf ein Array speichern kann:

```
typ [] name;
```

Anlegen eines neuen Arrays mit n Elementen:

```
name = new typ[n];
```

oder in einem:

```
typ [] name = new typ[n];
```

z.B.:

```
String [] POS = new String[7];
```

Array-Initialisierung und -Indizierung

Array-Variable anlegen und Anfangswerte zuweisen (*Dimension* des Arrays ergibt sich aus den Werten):

```
String [] POS = {"Noun", "Verb", "Adjective", "Adverb", "Determiner", "Preposition", "Pronoun", "Particle"};
```

Einzelnes Array-Element ansprechen:

```
String s = new String(POS[0]);  
System.out.println (POS[5]);
```

Indizes laufen immer von 0 bis $n-1$! Array-Elemente neu belegen:

```
POS[1] = "Test";  
POS[9] = "Blah"; // FEHLER!
```

Ein Block von Anweisungen

- wird in geschweifte Klammern eingeschlossen
- wird als eine Einheit ausgeführt (z.B. abhängig von einer Bedingung)
- kann beliebig viele weitere Blöcke enthalten

Anstelle eines Blocks kann meistens auch eine einzelne Anweisung stehen.

Schleifen 1: *for*

Eine *for*-Schleife ist besonders geeignet für die Verarbeitung von Daten in Arrays. Z.B. Ausgabe der Kommandozeilenparameter:

```
public static void main(String [] args)
{
    for(int i=0; i<args.length; ++i) {
        System.out. println (args [ i ] );
    }
}
```

Besteht aus: Initialisierung ($i = 0;$), Laufbedingung ($i < \text{args} . \text{length} ;$) und finaler Anweisung ($++i$).

Schleifen 2: *while-do* und *do-while*

Eine Bedingung (**boolean**-Variable oder Ausdruck) wird geprüft und ein Block von Anweisungen so lange ausgeführt, wie die Bedingung **true** ist:

```
boolean running=true, done=false;
```

```
while(!done) {  
    // do something  
}
```

```
do {  
    // something  
} while(running);
```

for vs. while

Die beiden folgenden Schleifen tun genau das selbe:

```
for(int i=0; i<args.length; ++i) {  
    System.out. println ( args [ i ] );  
}
```

```
int i=0;  
while(i<args.length) {  
    System.out. println ( args [ i ] );  
    ++i;  
}
```

Bedingungen 1: *if-else*

„*if*“ prüft eine beliebige Bedingung und führt den zugehörigen Block aus, wenn die Bedingung **true** ist, andernfalls den *else*-Block, sofern vorhanden.

```
boolean b=true;  
int x=1, y=2;  
if(b) {  
    System.out.println ("b_ist_true!");  
} else {  
    System.out.println ("b_ist_false!");  
}  
if(x<y) {  
    System.out.println ("x_ist_kleiner_als_y!");  
}
```

Bedingungen 2: *switch-case*

Ein *switch*-Block erlaubt eine mehrfache Verzweigung in Abhängigkeit vom Wert einer Variablen:

```
int note = 2;
switch(note) {
    case 1:
        System.out. println ("sehr_gut");
        break;
    case 2:
        System.out. println ("gut");
        break;
    default:
        System.out. println ("Nicht_so_gut");
}
```

Anmerkungen zu *switch-case*

- Jeder mögliche Fall sollte mit einem *case*-Statement behandelt werden.
- Nicht explizit aufgeführte Möglichkeiten können in einem **default**-Zweig behandelt werden.
- Mehrere Möglichkeiten, die die gleiche Aktion auslösen sollen, können durch Weglassen des **break**-Statements zusammengefasst werden:

```
switch(antwort) {  
    case 'j':  
    case 'J':  
        System.out.println ("Ja");  
        break;  
}
```

Eine einfache Klasse

```
class Rechteck {  
    private double breite , hoehe;  
    Rechteck() {  
        breite = hoehe = 1;  
    }  
    Rechteck(double b, double h) {  
        setBreite (b);  
        setHoehe(h);  
    }  
    public double flaeche() {  
        return breite * hoehe;  
    }  
    public void setHoehe(double h) { hoehe = h; }  
    public void setBreite (double b) { breite = b; }  
}
```

Anmerkungen zur einfachen Klasse (1)

- Attribute und Methoden können sowohl **private** als auch **public** sein (später mehr zu **protected**, **abstract** und un spezifiziert)
- Als **static** deklarierte Methoden können aufgerufen werden, ohne vorher ein Objekt der Klasse anzulegen. Z.B.
`Integer.parseInt(String)`
- *Konstruktoren* werden um Anlegen eines neuen Objekts aufgerufen. Sie heissen immer wie die Klasse und haben keinen Rückgabetyt.

Anmerkungen zur einfachen Klasse (2)

- Es können mehrere Konstruktoren mit unterschiedlicher Parameterliste existieren, von denen der Compiler den zum Aufruf passenden auswählt. Z.B.:
Rechteck r1 = Rechteck () → erste Variante
- Methoden innerhalb einer Klasse können andere Methoden direkt aufrufen (z.B. setBreite (b) in Rechteck (**double** , **double**)
- *Zugriffsmethoden* wie setHoehe (**double**) stellen im Gegensatz zu **public**-Attributen sicher, dass ungültige Attributwerte zurückgewiesen werden können.

Namenskonventionen (1)

Namenskonventionen für Zugriffsmethoden nach dem *JavaBeans*-Standard

- Schreiben: `setXxxx()`
- Lesen: `getXxxx()` bzw. `isXxxx()` für **boolean**-Attribute

Namenskonventionen (2)

Weitere empfohlene Namenskonventionen:

- *Klassennamen* beginnen mit Großbuchstaben und bestehen aus Groß- und Kleinbuchstaben und Ziffern (z.B. HelloWorld, Button2).
- *Konstanten* beginnen mit einem Großbuchstaben und bestehen nur aus Großbuchstaben, Ziffern und Underlines (z.B. MAX_SPEED).
- Alle anderen Namen (Datenfelder, Methoden, Objekte, Packages etc.) beginnen mit einem Kleinbuchstaben und bestehen aus Groß- und Kleinbuchstaben und Ziffern. (z.B. openFileDialog, button2, addActionListener).

Tips zum Debugging (1)

Zum Finden von Fehlerursachen ohne richtigen *Debugger* (Programm zum Analysieren eines anderen Programms zur Laufzeit) bieten sich zusätzliche Ausgaben mittels `System.out.println()` an:

```
int i=20;
while(i > 0) {
    System.out.println (args[i]);
}
```

```
int i=20;
while(i > 0) {
    System.out.println (args[i]);
    System.out.println (i);
}
```

Tips zum Debugging (2)

Tipparbeit spart eine **static**-Methode in der Klasse, wo Meldungen ausgegeben werden sollen:

```
void irgendwas() {  
    int i=20;  
    while(i > 0) {  
        System.out. println ( args[ i ] );  
        p(i);  
    }  
}  
  
static void p(String s) {  
    System.out. println (s);  
}
```

Übungen zu einfachen Klasse (1)

- 1 Schreiben Sie eine Programm, das zwei Rechteck-Objekte erzeugt: ein Standardquadrat und eins beliebiger Dimension.
- 2 Geben Sie die Fläche der beiden Rechtecke aus.
Tip: Zahlen werden von `println()` automatisch in Text gewandelt.
- 3 Fügen Sie zwei Methoden hinzu, mit denen Breite und Höhe abgefragt werden können.
- 4 Erweitern Sie die `set*`-Funktionen um eine Abfrage, die nur positive Werte zulässt und andernfalls eine Fehlermeldung ausgibt.
Tip: `System.err.println()`

Übungen zu einfachen Klasse (2)

- 5 Schreiben Sie eine Methode, die den Umfang des Rechtecks berechnet und probieren Sie sie aus.
- 6 Schreiben Sie eine neue Klasse für eine andere Figur (Kreis, Ellipse, Dreieck, ...), die ähnliche Methoden bereitstellt.

Beispiel Kochrezept

```
Wasser w = new Wasser(3000);
Topf topf = new Topf(4);
Herd herd = new Herd();
Pasta p = new Pasta(300);
topf.put(w); // Methode "put" muss Objekt d.
// Klasse Wasser akzeptieren

topf.put(new Salz(10)); // anonymes Objekt der Klasse Salz
herd.aufPlatte (1, topf);
herd.an(1);
while(topf.inhalt().Temperatur < 100) {
    herd.wait(60000);
}
topf.put(p);
System.wait(p.kochZeit);
herd.aus(1);
topf.abgießen(); // Unicode-Zeichen in Bezeichnern
    erlaubt
```

- Erweiterung und/oder Spezialisierung einer Klasse durch eine andere, wobei nur Unterschiede im Verhalten implementiert werden.
- Beispiel: Mensch → Student
- Unterklassen (*subclasses*) fügen ihrer Oberklasse (*superclass*) Methoden/Attribute hinzu oder ändern sie ab. Der Rest wird *geerbt*.
- Alle Klassen in Java erben automatisch von der Klasse *Object*.
- **protected**-Attribute verhalten sich nach außen wie **private**, nur für Unterklassen wie **public**.

Vererbung: Oberklasse *Mensch*

```
class Mensch {  
    protected float gewicht;  
    private int alter;  
    Mensch() {  
        gewicht=3.5F; alter=0;  
    }  
    Mensch(float g, int a) {  
        gewicht=g; alter=a;  
    }  
    public void laufen(int r) { /* ... */ }  
    public void essen(float menge) {  
        gewicht += menge; /* gewicht = gewicht + menge */  
    }  
}
```

Vererbung: Unterklasse *Student*

```
class Student extends Mensch {  
    private String matrikelNum;  
    static final int MAX_BEER = 10;  
    Student(String mn, float g, int a) {  
        super(g,a);  
        matrikelNum = new String(mn);  
    }  
    public void lernen(float zeit) {  
        gewicht -= zeit/50;  
    }  
    public boolean trinken(int flaschen) {  
        gewicht += flaschen / 2;  
        return flaschen < MAX_BEER;  
    }  
}
```

- Standard-Ein- und Ausgabe: `System.in` bzw. `System.out`
- Verschiedene Reader-Klassen stellen Ein- und Ausgabefunktionen zur Verfügung.
- Häufigste (linguistische) Anwendung: zeilenweises Lesen
 - `FileReader` öffnet eine mit Namen angegebene Datei
 - `InputStreamReader` liest aus bereits offenen Inputstreams wie `System.in`
 - `BufferedReader` stellt Pufferung und zeilenweises Lesen für einen existierenden Reader zur Verfügung

Beispiel: Datei lesen

```
try {  
    BufferedReader in = new BufferedReader(new FileReader("text"));  
    String str;  
    while ((str = in.readLine()) != null) {  
        // ...  
    }  
    in.close();  
} catch (IOException e) {  
    System.err.println("Read_error!");  
}
```

Exceptions (1)

- Sprach-Unterstützung für Fehlerbehandlung
- Allgemein: **try** { <statements > }
catch(<exception >) { [statements] } ...
[**finally** { <statements > }]
- Anweisungen im **try**-Block „werfen“ Fehler als Unterklassen von Exception bzw. Throwable, **try/catch** fängt sie ab.
- **finally**-Block wird *immer* ausgeführt, unabhängig von Auftreten eines Fehlers.
- Reihenfolge der **catch**-Blöcke: speziellere Exceptions zuerst!

Hierarchie der Throwable-Klassen

Sehr unvollständig, s. Klassendokumentation!

- Error – Schwerwiegender Fehler
- Exception – Ausnahmebedingung
 - IOException – Ein-/Ausgabefehler
 - FileNotFoundException
- InterruptedException
- RuntimeException – Laufzeitfehler
 - NullPointerException
 - IndexOutOfBoundsException
 - ArrayIndexOutOfBoundsException

Auslösen einer Exception

Methoden müssen sämtliche ausgelösten Exceptions vorab deklarieren:

```
class SomeClass {  
    public void myMethod(int foo)  
        throws IOException  
    {  
        if ( error ) {  
            throw new IOException("Dumm_␣gelaufen");  
        }  
    }  
}
```

Exceptions (2)

Eigene Exceptions können als Unterklasse existierender deklariert werden.

```
// Trivial , nur zur Vergabe eines deskriptiven Namens  
class MyException extends Exception { }
```

Werden von benutzten Methoden geworfene Exceptions nicht abgefangen, müssen diese wiederum in der eigenen Methode deklariert werden:

```
void myMethod(void) throws IOException {  
    // I/O-Operationen ohne try/catch-Blöcke  
}
```

Abstrakte Klassen

- Nur als Oberklassen fuer (i.d.R. mehrere) andere Klassen geeignet.
- Keine direkte Erzeugung von Objekten möglich.
- Dienen zur Vereinheitlichung von Schnittstellen bei gleichzeitiger Vererbung gemeinsamer Teile.
- Nicht implementierte Methoden werden mit ihrer *Signatur* (Name, Rückgabetyt und Parameter) angegeben und als **abstract** gekennzeichnet.
- Wenn eine oder mehrere Methoden **abstract** sind, muss die ganze Klasse ebenfalls als **abstract** deklariert sein.

Beispiel: Abstrakte Klassen

```
abstract class Mensch {  
    private double gewicht;  
    void trinken(double menge) {  
        // Diese Tätigkeit ist bei allen Menschen gleich  
        gewicht += menge;  
    }  
    // Arbeiten tun alle etwas unterschiedliches  
    abstract Object arbeiten(double zeit);  
}  
class Student extends Mensch {  
    Object arbeiten(double zeit) {  
        return new Hirn(zeit);  
    }  
}
```

Interfaces

- Ein Interface stellt quasi eine vollständig abstrakte Klasse dar.
- Vorteil: eine Klasse kann mehrere Interfaces implementieren (Ersatz für die Mehrfachvererbung in C++).

```
interface MyInterface {  
    public void myMethod1();  
    public Object myMethod2(int a);  
}  
class Foo implements MyInterface {  
    public void myMethod1() { /* ... */ }  
    public Object myMethod2(int a) { /* ... */ }  
}
```

- Problem: Speichern von Daten unter nichtnumerischen Indizes.
 - Allgemein: mit Namen assoziierte Informationen
 - Linguistik: *Frequenzlisten* („Wörter zählen“)
- Arrays: möglich, aber extrem langsam.
- Lösung: Hashtables
 - bilden große Schlüsselräume auf kleine ab.
 - bieten kurze Suchzeiten relativ unabhängig von Schlüssellänge und Tabellengröße.
- Java-Datentypen: `Hashtable` bzw. `HashMap`

- Implementiert das Interface Map, d.h. bildet eindeutige Schlüssel auf Werte ab.
- Wichtige Methoden:
 - `Object get(Object key)` – Wert unter Schlüssel `key` lesen
 - `Object put(Object key, Object value)` – Wert `value` unter Schlüssel `key` speichern.
 - `Object remove(Object key)` – Wert unter `key` löschen
 - **boolean** `containsKey(Object key)` – testen, ob `key` in der Map existiert.

HashMap: Beispiel

```
import java.util.*;
HashMap h = new HashMap();
h.put("test","hallo");
h.put("foo","bar");
System.out.println (h.get("foo"));
```

HashMaps nehmen *beliebige Objekte* sowohl als Schlüssel als auch als unter selbigen zu speichernde Werte!

HashMap: Aufgabe

- 1 Schreiben Sie ein Programm, das zeilenweise Strings von `System.in` liest.
- 2 Benutzen Sie jede Zeile als Schlüssel einer HashMap (**import** `java.util.*`; nicht vergessen!) und setzen Sie den zugehörigen Wert auf den leeren String
- 3 Besorgen Sie sich am Schluss ein Array der enthaltenen Keys (`h.keySet().toArray()`) und geben Sie es aus.

- Interface für Klassen, die eine Sammlung anderer Objekte darstellen.
- Implementation kann doppelte Einträge erlauben oder auch verbieten.
- Enthält u.a. Methoden zur Abfrage der Größe (`size()`), hinzufügen/entfernen von Objekten (`add()/remove()`) und diverse Mengenoperationen (Schnittmenge, Vereinigungsmenge, etc.)
- Methode `iterator()` stellt ein `Iterator`-Objekt zur Verfügung, mit dem Elemente einzeln abgerufen werden können.

Iterator (1)

- Ebenfalls ein Interface, es sind wiederum verschiedene Implementierungen möglich.
- Stellt drei Methoden zum iterieren über Mengen, Listen etc. zur Verfügung:
 - **boolean** hasNext () gibt **true** zurück, wenn noch weitere Elemente vorhanden sind.
 - **Object** next () gibt das nächste Element zurück
 - **void** remove () entfernt das letzte von next () zurückgegebene Objekt.

Iterator (2)

Iteratoren erlauben das Bearbeiten aller Elemente einer Menge, ohne Details über die Implementation der Menge kennen zu müssen:

```
Iterator iter = ... // Iterator aus einem Mengenobjekt besorgen
while( iter.hasNext() ) {
    Object o = iter.next();
    // Objekt bearbeiten
}
```

Mehr zu Datenströmen (1)

- Datenströme beinhalten alle Arten von Dateien, außerdem *Sockets* (hauptsächlich für Datenströme über Netzwerke)
- Java kennt zwei unterschiedliche Arten von *Datenströmen*:
 - `InputStream`, `OutputStream` und `RandomAccessFile` für *byte-orientierte Dateien*.
 - `Reader` und `Writer` für zeichen- und zeilenorientierte *Textdateien*. Sie können Zeichensatzumwandlung vom Systemstandard nach Unicode und umgekehrt durchführen.

Mehr zu Datenströmen (2)

- Die grundlegenden Datenstrom-Klassen werden meist mit Hilfe spezialisierterer Klassen benutzt, die bestimmte *Arten* der Ein-/Ausgabe implementieren.
- `BufferedInputStream`, `BufferedOutputStream`: puffern Daten im Speicher für effizienteres Lesen/Schreiben.
- `ObjectInputStream`, `ObjectOutputStream`, `DataInputStream`, `DataOutputStream`: erlauben das Lesen/Schreiben von Binärdaten bzw. ganzen Objekten.
- `BufferedReader`, `BufferedWriter`: puffern Reader/Writer wie oben.

Beispiel (1): Lesen von einem Webserver

```
String location =  
    "http://www.linguistik.uni-erlangen.de/";  
URL url = new URL(location);  
InputStreamReader instream =  
    new InputStreamReader(url.openStream(),"8859_1");  
BufferedReader in = new BufferedReader(instream);
```

Beispiel (2): Pipes

Zum Weiterverarbeiten der Ausgabe eines Programms lässt sich diese als `InputStream` auslesen.:

```
BufferedReader progout(String cmd)
throws IOException
{
    // Neuen Prozess erzeugen und "cmd" darin ausführen
    Process p = Runtime.getRuntime().exec(cmd);
    // Ausgabe als InputStream
    InputStream in = p.getInputStream();
    // Aus Effizienzgründen sowohl InputReader als auch
    // BufferedReader anlegen
    return new BufferedReader(new InputStreamReader(in));
}
```

Die *Writer*-Klassen

Anlegen wie beim *Reader*:

```
BufferedWriter out =  
    new BufferedWriter(new FileWriter("myfile"));
```

Append-Modus, d.h. geschriebene Daten werden an eine bestehende Datei angehängt:

```
FileWriter fw = new FileWriter("myfile",true);
```

OutputStreamWriter: für Spezialfälle, z.B. Umcodierung in anderen Zeichensatz:

```
OutputStreamWriter sw =  
    new OutputStreamWriter("myfile","8859_5");
```

Methoden der *Writer*-Klassen

- `write (int)` – schreibt ein einzelnes Zeichen in den niederwertigen Bits des **int** (**char** casten!)
- `write (String)`
- `write (char [] , offset , length)` – schreibt das `char`-Array, optional *length* Zeichen ab *offset*.
- `newLine ()` – schreibt ein Zeilenende-Zeichen.
- `flush ()` – stellt sicher, dass Daten im Puffer auf das Medium geschrieben wurden (bei `close ()` automatisch!)
- `close ()` – schließt die Datei.

Aufgabe: Datei kopieren

- Schreiben Sie eine Klasse, die per `InputStream/OutputStream` Daten zeichenweise aus einer Datei liest und in eine andere schreibt.
- Probieren Sie das Programm mit einer großen Datei (z.B. `/usr/bin/emacs`) aus.
- Fügen Sie `Buffered*`-Objekte zur Pufferung hinzu und probieren Sie es noch einmal.

Reguläre Ausdrücke (1)

Regulär: sind Grammatiken einer *regulären Sprache* der Chomsky-Hierarchie (\rightarrow Hausser, Kap. 8)

Def. *Atom*: ein einzelnes Zeichen oder ein durch runde Klammern zusammengefasster Unterausdruck. Symbole und Ausdrücke in der gebräuchlichsten Notation:

- $.$: ein beliebiges Zeichen.
- $*$: vorhergehendes Atom beliebig oft, auch 0 mal.
- $+$: vorhergehendes Atom beliebig oft, mind. einmal.

Reguläre Ausdrücke (2)

- $?$: vorhergehendes Atom 0- oder einmal.
- $[...]$: ein beliebiges in den Klammern enthaltenes Zeichen.
- $[^...]$: ein beliebiges Zeichen *außer* den eingeklammerten.
- $[a-z]$: ein beliebiges Zeichen im Bereich *zwischen* a und z.
- $(...)$: enthaltenen Ausdruck zu einem Atom zusammenfassen.

Reguläre Ausdrücke (3)

- |: Der Ausdruck links *oder* der rechts des senkrechten Strichs.
- ^: Zeilenanfang (intelligent interpretiert, steht also zwischen anderen Atomen fuer sich selbst!).
- \$: Zeilenende (dito).

Besonderheit in Java: `String.matches()` sowie `Pattern` implizieren ein „`^ ... $`“ um den abzustimmenden Ausdruck.

Einfaches Patternmatching

```
String s = "irgendwas";  
if(s.matches("muster")) {  
    System.out.println ("\" + s + "\" passt!");  
}
```

Nachteil: "muster" wird bei jedem Abpassen erneut „kompiliert“, d.h. in einen Automaten verwandelt, der das entsprechende Muster erkennen kann.

Effizienter wiederholbares Patternmatching

```
import java. util . regex . * ;  
String s = "irgendwas";  
Pattern p = Pattern.compile("muster");  
Matcher m = p.matcher(s);  
if (m.matches()) {  
    System.out. println ( "\ " + s + "\ " + "passt!" );  
}
```