Algorithmen & Programmierung für Ingenieure

Dr.-Ing. Ulf Döring

TU Ilmenau
Fakultät für Informatik und Automatisierung
FG Graphische Datenverarbeitung

Vorabversion vom 2. Oktober 2020

Teil I

Struktogramme & Java-Grundlagen

Heutige Themen

- Organisatorisch:
  - Semesterplanung: Vorlesungen/Übungen/Tutorien
    → zunächst hybrid, später ggf. auch nur online
    → Infos zunächst hier:
    `https://www.tu-ilmenau.de/gdv/lehre/ws2020/aup`
    später auch:
    `https://moodle2.tu-ilmenau.de/course/view.php?id=3127`
  - Stand am 2. Oktober 2020: 42. KW (erste Vorlesungswoche) nur im Audimax Großübung (Donnerstag, 15. Okt.) und Vorlesung (Freitag, 16. Okt.); Gruppenübungen und Tutorien (im Zusebau, Z1021) erst ab 43. KW
  - AuP - Wozu?
- Inhaltlich:
  - Allgemeiner Umgang mit Eclipse (Livedemo)
  - Darstellung von Algorithmen mit Struktogrammen
  - Einführung zu Variablen in Java
  - Einführung zu Methoden in Java

AuP - Wozu?

- Weil es Spaß macht.
- Weil man muss.
- Weil es nützt. Beispiele:
  - Wie entwickeln sich die Berufsbilder für Ingenieure?
  - Möchte ich wirklich die Datei mit den Messdaten per Hand auswerten/bearbeiten?
  - Will ich den Informatikern alles glauben, was sie mir sagen?

Muss letztlich jeder für sich entscheiden.

Umfrage 1

Welche **Programmiersprache(-n)** wird (zumindest etwas) beherrscht?

Bitte im Chat antworten.

Eclipse, Einrichtung, erstes Programm, ...

Algorithmen vs. Daten

Was ist ein Algorithmus allgemein?

Ein Algorithmus ist eine **(eindeutige)** Handlungsvorschrift zur Lösung einer Aufgabe.

Im Kontext AuP:

Ein Algorithmus beschreibt **(eindeutig)** die systematische Verarbeitung von **Daten** (Informationen).
→ Ziel ist die maschinelle Ausführung.

Beispiele:

- Kaffeekochen mit Kaffeemaschine
- Verhalten an einer Straßenkreuzung
- Berechnung des Mittelwertes einer Messreihe (ohne Ausreißer)
- ...

Variablen zur Speicherung von Daten

- Variablen in Java nur nutzbar, wenn ihr **Typ** bekannt ist
  → Größe des Speicherbereichs & Art der Auswertung
- allg. Variablendefinition **ohne** Initialisierung: `Typ Name;`
  mit Initialisierung: `Typ Name = Anfangswert;`
- wichtige Typen und Bsp. zur Definition (wird später noch vertieft):
  - Ganzzahlen: `byte, short, int, long`
    ```
    int i = 3;          // typisch: int
    ```
  - Gleitkommazahlen: `float, double`
    ```
    double pi = 3.14; // typisch: double
    ```
  - Zeichenketten: `String`
    ```
    String s = "Hallo";
    ```
  - Wahrheitswerte: `boolean`
    ```
    boolean b = true;
    ```

Struktogramme (1)

Wozu? Warum nicht gleich Java?

- keine strenge Syntaxprüfung,
- für viele Anfänger anschaulicher,
- beliebig genau,
- standardisiert, siehe DIN 66261, *Sinnbilder für Struktogramme nach Nassi-Shneiderman*

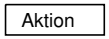
Grundidee:

- visuell gut erfassbare Blockstruktur (Rechtecke),
- streng hierarchisch gegliedert (Matrjoschka-artige Verschachtelungen),
- Blöcke können Texte aber auch z.B. Java-Anweisungen enthalten
- nur wenige Blockarten (Arten von Arbeitschritten)

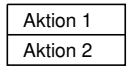
Struktogramme (2)

Arten von Struktogrammblöcken:

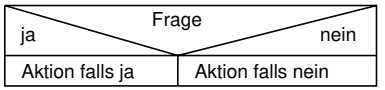
- elementarer Schritt (nicht weiter unterteilt), z.B. Eingabe, Ausgabe, Stop, Berechnung ...



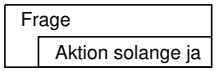
- Sequenz (Folge von Schritten)



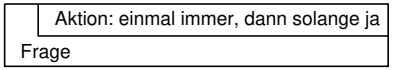
- bedingte Ausführung (Verzweigung)



- Wiederholung (Schleife)
  - 1. kopfgesteuert:



  - 2. fußgesteuert:



Struktogramme (3)

Sonstiges:

- am Anfang meist **Eingabe(n)**,
- im Struktogramm (oft auch am Ende) **Ausgabe(n)**,
- Beginn der Abarbeitung beim obersten Block,
- Sequenzen werden von oben nach unten abgearbeitet,
- **Initialisierung** von Variablen ist grundsätzlich wichtig,
- **Stop** des Algorithmus, wenn letzter Befehl der äußeren Sequenz erreicht ist, oder wenn im Aktionsblock beschrieben (z.B. durch Wörter wie *Stop* oder *Halt*)

Methoden (1)

- oft ähnlich Funktionen in der Mathematik (wenn sie etwas zurückgeben, z.B. Ergebnis einer Summen- oder Produktberechnung)
- Anwendungsbeispiele:

```
int sum = summe(3, 4); // danach speichert sum die 7
```

```
System.out.println(produkt(3, 4)); // Ausgabe von 12
```

- Definitionsbeispiele:

```
int summe(int x, int y){
    return x+y; // platzsparend
}
```

```
int produkt(int x, int y){
  int p = x*y; // mit temporärer Hilfsvariable p,
  return p;    // etwas komplizierter, aber auch OK
}
```

Methoden (2)

verallgemeinerte Definitionsbeispiele für 1, 0 und 2 Parameter in der Parameterliste sowie mit und ohne Rückgabe eines Rückgabewertes:

-
```
Rückgabetyp methodenname(Parametertyp parametername){
    return Rückgabewert;
}
```
-
```
Rückgabetyp methodenname(){
    Aktion 1
    Aktion 2
    return Rückgabewert;
}
```
-
```
void methodenname(Parametertyp1 parametername1,
                  Parametertyp2 parametername2){
  Aktion 1
  Aktion 2
  Aktion 3 // keine Werterückgabe, deshalb void
}
```


Methoden (3)

Wenn nichts zurückgegeben werden soll, dann ist `void` an Stelle des Rückgabetyps zu schreiben.

```
/* Dies ist ein Beispiel für eine Methode, welche Ausgaben
   auf der Konsole macht. Sie nimmt einen String entgegen
   und gibt nichts zurück. */

void sopl(String x){
    System.out.println(x);
}
```

Vor dem Rückgabetyp (oder `void`) können auch sogenannte Modifikatoren wie `public` oder `static` in **beliebiger Reihenfolge** stehen. Siehe z.B. die `main`-Methode.

```
static public void main(String[] a){ // Parametername ist frei wählbar
    // hier steht der Quellcode zum Programmstart ...
}
```

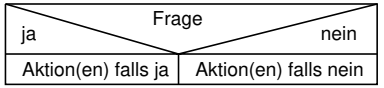

Heutige Themen

Kontrollstrukturen: *Wie wird der Programmfluss in Java gesteuert?*

- Verzweigungen bzw. optionale Ausführung per `if/else`,
- verschiedene Arten von Schleifen:
  - `while`-Schleife,
  - `for`-Schleife und
  - `do/while`-Schleife,
- Anwendung von `break`- und `continue`-Befehlen in Schleifen,
- Mehrfachverzweigungen per `switch/case` sowie
- jeweils Livedemo, Hinweise auf häufige Fehler, Beispiele für Einsatz von Variablen und Methoden

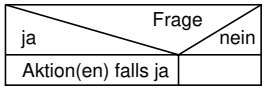
Verzweigungen (1)

- Verzweigungen werden verwendet, wenn es:
  - 2 alternative Abarbeitungspfade gibt



oder

  - Code nur optional (bedingt) aufgerufen werden soll



- In Java dienen `if` und (optional) `else` zum Verzweigen.
  - Beispiel für 2 alternative Abarbeitungspfade:

```
if (temperatur>limit)    // Vergleich 2er Variablen
    alarmLichtAn();      // Aufruf einer Methode falls true
else
    alarmLichtAus();     // bei false Aufruf einer anderen Methode
```

  - Beispiel für optionalen Code

```
if (zuWenigKraftstoff())    // Test mittels Methodenaufruf
    anhaltenUndMotorAus();  // Methodenaufruf nur falls true
```

Verzweigungen (2)

- In einem `if`- oder `else`-Zweig steht **genau ein** Befehl

```
if (ampelIstRot())
    halteAn();
else
    fahreWeiter();
```

oder ein **Befehlsblock**, d.h. eine Liste von 0 bis n Befehlen in geschweiften Klammern.

```
if (genuegendKraftstoff()){ // 0 Befehle falls true
}
else {                      // 2 Befehle falls false
    halteAn();
    macheMotorAus();
}
```

- Häufiger Anfängerfehler:

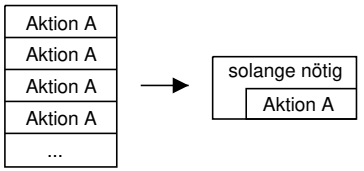
```
if (ampelIstRot());
    halteAn();
```

Hier wird der leere Befehl ausgeführt, wenn die Ampel rot ist. Angehalten wird dann immer. In Eclipse führt die Formatierung per Ctrl-Shift-F zu:

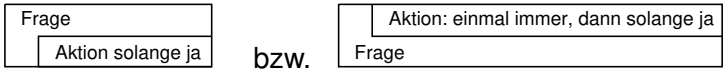
```
if (ampelIstRot())
    ;               // hier erkennt man das Problem besser
halteAn();          // angehalten wird stets (nicht nur bei roter Ampel)!
```


Allgemeines zu Schleifen

- erlauben wiederholte Abarbeitung von Code (vermeiden langer Sequenzen mit gleichen Aktionen),



- können kopfgesteuert (`while` und `for`) oder fußgesteuert (`do-while`) sein,

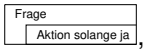


- nutzen oft sogenannte Laufvariablen:

```
for (int i=0; i<10; i=i+1)
    System.out.print(i+" ");
```

im obigen Beispiel zur Ausgabe der Ziffern 0 bis 9 auf der Konsole.

while-Schleife (1)

- kopfgesteuert 

, kann also Codeausführung im Innern verhindern,
- hat in den runden Klammern z.B.:
  - eine `boolean`-Variable,

```
while (wasserstandIstZuHoch)
    pumpeWasserAb();
```

  - einen Vergleich von Werten,

```
while (getWasserstand() > erlaubterHoechstwert)
    pumpeWasserAb();
```

  - oder den Aufruf einer Methodenaufruf, welche einen `boolean`-Wert zurückgibt

```
while (wasserstandZuHoch())
    pumpeWasserAb();
```

- Im Inneren hat die `while`-Schleife genau einen Befehl (siehe oben) oder einen Block von Befehlen:

```
int i=0; // weiteres Laufvariablenbeispiel
while (i<10){
    System.out.print(i+" ");
    i=i+1;
}
```


while-Schleife (2)

- Häufiger Anfängerfehler:

```
while (luftdruckImSchlauchZuNiedrig());
    pumpeLuftInSchlauch();
```

Hier wird der leere Befehl ausgeführt (also eigentlich **nichts** getan), solange der Luftdruck zu niedrig ist. Sollte dann (warum auch immer) genug Luft im Schlauch sein, wird noch „etwas“ mehr hineingepumpt.
In Eclipse führt die Formatierung per Ctrl-Shift-F zu:

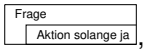
```
while (luftdruckImSchlauchZuNiedrig())
    ;                           // ungewollte Untätigkeit
pumpeLuftInSchlauch();          // ungewollte Extraportion
```

- auch oft falsch:

```
int i = 1;
int sum = 0;
while (i<=100) {  // berechne die Summe von 1 bis einschließlich 100
    sum = sum+i;
}
```

`i` wird in der Schleife nicht geändert. D.h. wenn zu Beginn der Schleife tatsächlich `i<=100` ist, dann bleibt die Bedingung auch stets `true`.
Dann handelt es sich um eine **Endlosschleife**.
→ Erhöhung von `i` muss auch in die Schleife, z.B. `i = i+1;` **(aber wohin?!)**

for-Schleife (1)

- kopfgesteuert 

, kann also Codeausführung im Innern verhindern,
- hat in den runden Klammern 3 Bereiche:
  - Initialisierung,
    ```
    int i=1 // z.B. zur Initialisierung der Laufvariablen
    ```
  - Test, d.h. etwas, was einen `boolean` ergibt (oder leer ist)
    ```
    i<=100 // z.B. ob Laufvariable noch im Bereich ist
    ```
  - Code, der nach dem letzten inneren Befehl ausgeführt wird
    ```
    i = i+1 // hier nicht so leicht vergessbar wie bei while!
    ```
- Die 3 Bereiche werden durch 2 Semikolons voneinander getrennt:

```
int sum=0;
for (int i=0; i<=100; i = i+1)
    sum = sum+i;
System.out.print(sum); // gibt 5050 auf der Konsole aus
```


for-Schleife (2)

- Jeder Bereich kann leer sein.
- Extremfall: alle Bereiche sind leer.

```
for (;;)
    System.out.println("noch eine Ausgabe auf der Konsole");
```

- dies ist auch eine Endlosschleife:

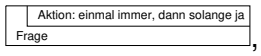
```
for (;true;)
    System.out.println("noch eine Ausgabe auf der Konsole");
```

- (leider) ist auch folgendes erlaubt:

```
for (int i=1, sum=0; i<=100; sum=sum+i,
     System.out.println("sum:"+sum), i=i+1);
```

Im 1. Bereich darf auch mehr als eine Variable initialisiert werden und im 3. Bereich dürfen sogar mehrere (beliebige) Befehle stehen. Die Trennung erfolgt jeweils durch ein Komma.

do-while-Schleife

- fussgesteuert 

, kann also **beim ersten Durchlauf** die Codeausführung im Innern **NICHT** verhindern,
- Nach der Fortsetzungsbedingung muss ein Semikolon stehen!

```
static public int calcSum(int from, int to){
    int sum=0;
    int i=from;
    do {
        sum = sum+i;
        i = i+1;
    } while (i<=to); // Compiler fordert hier das Semikolon
    return sum;
}
```

- Empfehlung: im Allgemeinen lieber kopfgesteuerte Schleifen verwenden (sicherer, insbesondere die `for`-Schleife)

break, continue und return in Schleifen (1)

- Aufruf von `break` → Schleife wird verlassen (dann Befehle unter der Schleife abarbeiten)
- Aufruf von `continue` in `for` → Schleife wird im 3. Bereich fortgesetzt:

```
for (int i=0; i<7; i=i+1){
    if (i==3)
        continue; // geht zu i=i+1
    if (i==5)
        break;    // verlässt die Schleife
}
```

- Aufruf von `continue` in `while` → Schleife wird mit Test der Bedingung fortgesetzt, ähnliches Beispiel:

```
int i=0;
while ( i<7 ){
    if (i==3)
        continue; // geht zum Test i<7, ist das ein Problem?
    if (i==5)
        break;    // würde die Schleife verlassen
    i = i+1;
}
```

break, continue und return in Schleifen (2)

- Aufruf von `continue` in `do while` →
  Schleife wird mit Bedingungstest fortgesetzt, ähnliches Beispiel:

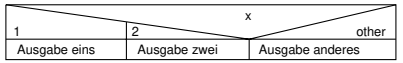
```
int i=0;
do {
    if (i==3)
        continue; // geht zum Test i<7, ist das ein Problem?
    if (i==5)
        break;    // würde die Schleife verlassen
    i = i+1;
} while ( i<7 );
```

- Aufruf von `return` →
  beendet die Methode und damit auch die Schleife:

```
static int machWas(int i){ // Wert in i wird von außen vorgegeben
    do {
        if (i%2==0)   // gerade?
            continue; // geht zum Test i<7, ist das ein Problem?
        if (i==5)
            return i;    // würde die Schleife verlassen
        i = i+1;
    } while ( i<7 );
    return i; // Frage: welche Werte könnte machWas zurückgeben?
}
```


switch-case (Mehrfachverzweigung) (1)

- wie `if else`, erlaubt aber mehr Alternativen, z.B.



- Es wird nur die Alternative angesprungen, die den passenden Wert hat. Analog zu `else` kann hier `default` die Restmenge behandeln. Bsp:

```
public static String asRoman_1(int i){
    switch (i){
        case  1: return "I";
        case  2: return "II";
        case  3: return "III";
        default: return "todo("+i+")";
    } // da jede Alternative die Methode beendet, steht hier kein return
}
```

- selbes Verhalten wie oben (diesmal ohne `default`-Zweig):

```
static public String asRoman_2(int i){
    switch (i){
        case  1: return "I";
        case  2: return "II";
        case  3: return "III";
    }
    return "todo("+i+")";
}
```


switch-case (Mehrfachverzweigung) (2)

- selbes Verhalten wie vorher, aber `break`-Befehl ist wichtig, um am Ende eine Alternative nicht die Befehle der nächsten abzuarbeiten (Durchlauf verhindern):

```
static public String asRoman_3(int i){
    String ret = "todo("+i+")";   // default
    switch (i){
        case  1: ret = "I";   break; // raus aus switch (hin zum return)
        case  2: ret = "II";  break; // raus aus switch (hin zum return)
        case  3: ret = "III"; break; // raus aus switch (hin zum return)
    }
    return ret;
}
```

- mittels Durchlauf lassen sich auch mehrere Werte zusammen verarbeiten:

```
public static String klassifiziere(int i){
    String ret = "unbekannt("+i+")";   // default
    switch (i){
        case  1:                                  // durchlaufen lassen
        case  2: ret = "eins oder zwei"; break;   // Durchlauf beenden
        case  3:                                  // durchlaufen lassen
        case  4: ret = "drei oder vier"; break;   // Durchlauf beenden
    }
    return ret;
}
```

- häufiger Fehler: `break` vergessen

Heutige Themen

- Organisatorisches
- Klassendefinition
- Programmierfehler
- Arrays:
  - Wozu?
  - Erzeugung,
  - Lesen und Schreiben von Elementen,
  - Methoden zur Verarbeitung von 1d-Feldern,
  - mehrdimensionale Felder,
  - häufige Fehler

Definition einer Klasse (vereinfacht)

- Klassennamen **sollen** mit einem großen Buchstaben anfangen.
- Aktuell beschränken wir uns auf folgendes:
  - äußerer Klassenrahmen ist stets
    ```
    public class Klassenname {
    }
    ```
  - Zwischen den geschweiften Klammern werden Methoden definiert.
  - Methoden haben stets die Modifikatoren `static public` (Reihenfolge egal).
  - Eine Methode ist stets `void main(String[] a)` (Name des Parameters frei wählbar). Sie startet das Programm (z.B. die Tests).
  - Die Klasse ist immer in einer Datei namens `Klassenname.java`
- Später (Abschnitt „OOP“ = „objektorientierte Programmierung“) nutzen wir wesentlich mehr Möglichkeiten zur Definition von Klassen!

Arten von Fehlern, Finden/Vermeiden von Fehlern

Arten:

- Compilierfehler (Compiler „meckert“)
- Laufzeitfehler (Absturz, ungewollte Endlosschleife ...)
- logische Fehler = das Programm tut nicht (immer) was es soll

Finden/Vermeiden:

- auf Compilermeldungen achten (rote Markierungen am Rand der IDE) **und** Compilerhinweise lesen/verstehen/umsetzen lernen
- Debugger benutzen
- viel üben, üben, üben... (insbesondere auch um später ohne eine IDE Fehler zu erkennen - z.B. in Klausur;-)
- geschriebenen Code **systematisch** testen

Allgemeines zu Arrays/Feldern, Erzeugung (1d)

Verwendung:

- nutzbar als Variable mit „beliebig vielen“ Elementen (gleichen Typs),
- Speichern von Zahlenfolgen (z.B. Meß- oder Testwerten) in 1d-Feldern,
- Speichern von 2d- oder 3d-Bilddaten (in mehrdimensionalen Feldern),
- ...

Erzeugungsbeispiele (1d):

- Feld mit konstanten Gleitkommazahlen (z.B. zum Testen):
  ```
  double[] testeDiese = { 0.0, -0.1, 3.1415, 7, .5 };
  ```
- Feld zur Speicherung von n Ganzzahlen (automatisch mit 0 gefüllt):
  ```
  int n = 1000;
  long[] werte = new long[n]; // auch: new long[1000]
  ```
- (noch) nicht existierendes `int`-Feld:
  ```
  int[] gibtEsNochNicht = null;
  ```


Beispiele für das Lesen/Schreiben von Felddaten

- Test auf Existenz:
  ```
  if (werte == null) // existiert nicht
      return false;  // Verarbeitung nicht möglich
  ```
- Test auf Leersein:
  ```
  if (werte.length == 0) // ist leer
      return false;  // Verarbeitung sinnlos
  ```
  Felder speichern ihre Länge in der Konstanten `length` (ist nur lesbar).
- Addition der einzelnen Feldelemente:
  ```
  for (int i=0; i<werte.length; i++)
      sum = sum + werte[i];  // Lesen via Index 0..n-1
  ```
  Zugriff auf negative Indices oder solche `>=` der Feldlänge führt zum Absturz!
- alle Elemente um eine Position nach vorn kopieren und 0 in letztes schreiben:
  ```
  for (int i=1; i<werte.length; i++)
      werte[i-1] = werte[i];  // um eins nach vorn
  werte[werte.length-1] = 0; // letztes mit 0 füllen
  ```

Methoden zur Verarbeitung von 1d-Feldern

- typisches allgemeines Beispiel:

Eingabe: Feld f
accu = neutrales Element
für alle Elemente e in f
verarbeite e (z.B. "aufsammeln" in accu)
Ausgabe: accu

- Summe, Produkt, Minimum:

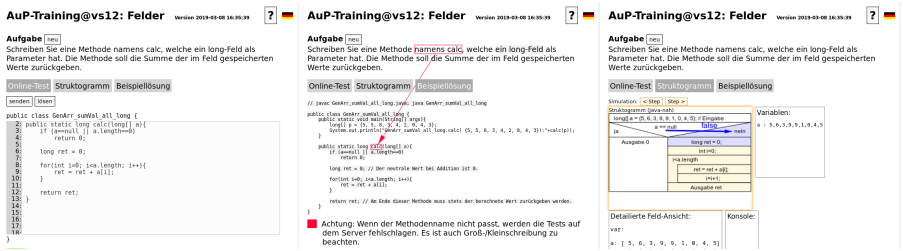
Eingabe: Feld f
accu = 0
für alle Elemente e in f
accu = accu + e
Ausgabe: accu

Eingabe: Feld f
accu = 1
für alle Elemente e in f
accu = accu * e
Ausgabe: accu

Eingabe: Feld f
accu = 1000000
für alle Elemente e in f
accu>e ja / nein
accu = e
Ausgabe: accu

Eingabe: Feld f
accu = erstes Element
für alle weiteren Elemente e in f
accu>e ja / nein
accu = e
Ausgabe: accu

Online-Aufgabe(n) zum Üben solcher Methoden

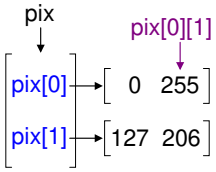


siehe `https://vs12.prakinf.tu-ilmenau.de/AuP/jsp/genArrTest3.jsp`

Mehrdimensionale Felder

- 2d-`int`-Feld mit konstanten Werten (z.B. 2x2-Grauwertbild):

```
int[][] pix = {{ 0, 255}, { 127, 206}};
```




```
int[] zeile1 = pix[0]; // inneres Array ist ein 1d-Array
int[] zeile2 = pix[1]; // == { 127, 206 }
int pixelObenRechts  = zeile1[1]; // Element des inneren Arrays
int pixelUntenRechts = pix[1][1]; // == 206  (so auch OK)
```

- Feld zur Speicherung einer 4x4 Matrix `m` (automatisch mit 0 gefüllt):

```
double[][] m = new double[4][4];
```

- Hauptdiagonale von `m` mit 1 beschreiben, um Einheitsmatrix zu erhalten:

```
for (int i=0; i<4; i++)
    m[i][i] = 1;
```


weitere Beispiele, Hinweise ...

Im Merkzettel gibt es mehrere Abschnitte, welche sich mit 1d- und mehrdimensionalen Felder beschäftigen. Dort sind allgemeine Hinweise, aber auch viele Beispiele mit Anmerkungen zu finden.

Empfehlung: Beispiele in Eclipse nachvollziehen, mit dem Quellcode experimentieren und bei Problemen nachfragen (z.B. in Tutorien oder Seminaren).

siehe

`https://moodle2.tu-ilmenau.de/mod/resource/view.php?id=81953`

Häufige Fehler beim Arbeiten mit Feldern

- Absturz, wenn Index bis **einschließlich n** läuft (Zugriff auf das erste nicht existierende Element), z.B.:

```
for (int i=0; i<=werte.length; i++) // richtig: i<werte.length
    summe = summe + werte[i];
```

```
double d = werte[werte.length]; // richtig: werte[werte.length-1];
```

- Absturz, da auf nicht existierendes Feld zugegriffen wird (auf seine Länge oder einen beliebigen Index), z.B.:

```
if (werte.length==0 || werte==null) // falsche Reihenfolge!!!
    return false;            // richtig: 1. Existenz, 2. Länge
```

- Absturz, da das Minimum mit dem 1. Feldelement initialisiert wird, das Feld aber leer ist:

```
int min = werte[0]; // wenn Feld leer, existiert keinen Index 0
```

- Absturz, da bei (unbekannten) mehrdimensionalen Feldern auf ein nicht existierendes inneres Feld zugegriffen wird
- Absturz, da bei (unbekannten) mehrdimensionalen Feldern die inneren Felder ungleich lang sind, aber stets nur die Länge des 1. inneren Feldes beachtet wird

Heutige Themen

- Organisatorisches
  - Tutorien,
  - Materialien in Moodle
- Kommentare in Java,
- Bezeichner (Namen für Variablen, Methoden, Klassen etc)
- Variablen (bisherige Arten, Deklaration, Initialisierung, Sichtbarkeit, Konvertierung, Wertebereiche...),
- Operatoren (Arten, Auswertungsreihenfolge, Beispiele)

Kommentare in Java

- Kommentare werden vom Java-Compiler ignoriert.
- Sie sollen helfen die Implementierung auch später noch zu verstehen.
- In IDEs (**I**ntegrated **D**evelopment **E**nvironment) werden Kommentare oft in einer extra Farbe eingefärbt (z.B. hellgrün).
- **Zeilenkommentare** starten mit `//` und gehen bis zum Ende der Zeile.

```
double x = 13;  // x-Position, startet links
```

- **Blockkommentare** starten mit `/*` und enden mit `*/` .

```
int a = 7;  /* Hier beginnt der Blockkommentar
int b = 8; dieser Befehl wird ignoriert
der folgende aber nicht mehr! */ int c = 9;
```

- Blockkommentare können zwischen „Code-Elementen“ stehen, z.B.:

```
/*Typ:*/double/*Name:*/ x = /* init. mit */ 13;
```


Bezeichner

- dienen als Namen für Variablen, Methoden, Klassen ...,
- dürfen nicht mit einer Ziffer anfangen,
- sollten möglichst nur aus:
  - ASCII-Buchstaben (a bis z sowie A bis Z),
  - Ziffern (0 bis 9) sowie
  - dem Unterstrich ("_") bestehen,
- sollten für **v**ariablen und **m**ethoden mit einem kleinen Buchstaben beginnen, für **K**lassen jedoch mit einem großen!

Namenslänge:

- Faustregel: je kleiner der Einsatzbereich, desto kleiner der Name (z.B. i für Zählvariable in kurzer Schleife), denn zu lange Bezeichner schlecht für ein schnelles Erfassen der Code-Struktur
- Bezeichner möglichst als „camel**C**ase**W**ort“ schreiben (jedes neue innere Wort beginnt mit einem Großbuchstaben) oder als „bezeichner_mit_unterstrichen“ (Worte durch Unterstriche trennen)

Bisherige Arten von Variablen

- lokale Variablen
  - können bei Deklaration initialisiert werden (optional),
  - müssen aber vor dem 1. Lesezugriff initialisiert werden,
  - werden am Methodenende verworfen
    (nur return-Werte „überleben“ das Methodenende!)
- Parametervariablen
  - werden bei Definition nicht initialisiert (immer erst beim Aufruf!)
  - werden dann in Methode wie lokale Variablen behandelt

```
public class ABC {
    public static int addiere(int x, int y){ // Parametervar. x und y
        return x+y;
    }
    public static void main(String[] a){ // Parametervariable a
        int y = 17;                 // lokale Variable y
        int s = addiere(y, 4);      // lokale Variable s
        System.out.println(s);
    }
}
```


Sichtbarkeit von Variablen

- Bei einer Befehlssequenz sind die Variablen erst nach ihrer Deklaration sichtbar:

```
// hier sind weder s noch t bekannt bzw. beide sind nicht sichtbar
int s = 3; // hier ist t immer noch nicht sichtbar (nur s)
int t = 4; // ab hier ist auch t sichtbar
```

- Parametervariablen sind nur in der Methode sichtbar:

```
public static int addiere(int a, int b){
    return a+b; // a und b sind hier sichtbar
}
// hier ist weder a noch b sichtbar!
```

- In Schleifen oder Verzweigungen deklarierte Variablen sind außerhalb nicht sichtbar (Sichtbarkeit endet am jeweiligen Blockende).

```
for (int i=0; i<5; i++){
   // hier ist i sichtbar
}
// hier ist i nicht sichtbar!
if (doIt()){
    int x = 3;
    System.out.println(x); // hier ist x sichtbar
}
else {
    System.out.println("bla"); // hier ist x nicht sichtbar
}
// hier ist x nicht sichtbar!
```


Konvertierung / Ergebnistypen

mit Strings:

- `String` + irgendwas → `String`
- irgendwas + `String` → `String`
- **Tipp:** `""+zahl` wandelt die Zahl in einen `String`

mit Zahlentypen:

- `double` * zahlentyp → `double` (auch bei anderer Rechenoperation)
- zahlentyp * `double` → `double` (auch bei anderer Rechenoperation)
- **Beachte:** Zieltypen sind nicht entscheidend!

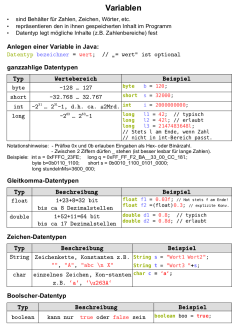
```
double x = 1/2;       // 0    da Integer-Division!
double y = 1/2.0;     // 0.5
double z = 1.0/2;     // 0.5
```

- automatisch „von klein nach groß“ :
  `byte` → `short` → `int` → `long` → `float` → `double`
  z.B. `byte b=3; short s=b;`
- sonst muss Typkonvertierung erzwungen werden, z.B.:
  `short s=3; byte b = (byte)s; // vgl. Zufallszahlen`

Wertebereiche, Initialisierungsmöglichkeiten etc

siehe Merkblatt!

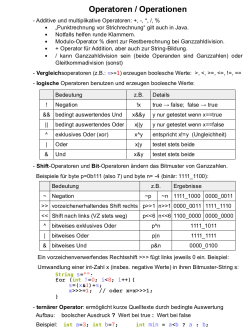
https://moodle2.tu-ilmenau.de/mod/resource/view.php?id=81953



Arbeit mit Operatoren

siehe Merkblatt!

`https://moodle2.tu-ilmenau.de/mod/resource/view.php?id=81953`



Operatoren / Operationen

- Additive und multiplikative Operatoren: `+, -, *, %`
  - „Punktrechnung vor Strichrechnung" gilt auch in Java.
  - Notfalls helfen runde Klammern.
  - Modulo-Operator `%` dient zur Restberechnung bei Ganzzahldivision.
  - `+` Operator für Addition, aber auch zur String-Bildung.
  - `/` kann Ganzzahldivision sein (beide Operanden sind Ganzzahlen) oder Gleitkommadivision (sonst)
- **Vergleichsoperatoren** (z.B. `<=>=`) erzeugen boolesche Werte: `>, <, >=, <=, !=, ==`
- **logische** Operatoren benutzen und erzeugen boolesche Werte:

	Bedeutung	z.B.	Details
!	Negation	!x	true ↔ false; false ↔ true
&&	bedingt auswertendes Und	x&&y	y nur getestet wenn x==true
||	bedingt auswertendes Oder	x||y	y nur getestet wenn x==false
^	exklusives Oder (xor)	x^y	entspricht x!=y (Ungleichheit)
|	Oder	x|y	testet stets beide
&	Und	x&y	testet stets beide

- **Shift**-Operatoren und **Bit**-Operatoren ändern das Bitmuster von Ganzzahlen.

Beispiele für byte p=0b1111 (also 7) und byte m=-4 (binär: 1111_1100):

	Bedeutung	z.B.	Ergebnisse	
~	Negation	~p	~m	1111_1000	0000_0011
>>	vorzeichenerhaltendes Shift rechts	p>>1	m>>1	0000_0011	1111_1110
<<	Shift nach links (VZ bleibt weg)	p<<3	m<<3	1102_0000	0000_0000
^	bitweises exklusives Oder	p^m		1111_1011	
|	bitweises Oder	p|m		1111_1111	
&	bitweises Und	p&m		0000_0100	

Ein vorzeichenerhaltendes Rechtsshift `>>` fügt links jeweils 0 ein. Beispiel:

Umwandlung einer int-Zahl x (inkrl. negativer Werte) in ihren Bitmuster-String s:

```
String s= "";
for (int i=0; i<32; i++) {
    s=(x&1)+s;
    x>>>=1;  // oder x=x>>>1;
}
```

- **ternärer Operator**: ermöglicht kurze Quelltexte durch bedingte Auswertung
  Aufbau: `boolscher Ausdruck ? Wert bei true : Wert bei false`
  Beispiel: `int x=3; int z=7;  int MIN = x<z ? x : z;`

- **Prioritäten**: ohne Klammerung werden die Operatoren entsprechend den in Java definierten Prioritäten ausgewertet, z.B. `&&` vor `||`. **Tipp: Im Zweifelsfall immer Klammern setzen!** Vgl. auch die Java-Dokumentation z.B. bei Oracle:
  https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html
  wo z.B. zu sehen ist, dass Punktrechnung („multiplicative" operators) eine höhere Priorität als Strichrechnung („additive" operators) hat. Ebenso ist „logical AND" höher als „logical OR" priorisiert.
  Eine Zuweisung hat stets die geringste Priorität.
- Die **Auswertungsreihenfolge** (von links nach rechts) bei 2 Operanden sowie bei mehreren Operatoren gleicher Priorität ist wichtig, z.B. wenn Methodenaufrufe verwendet werden:

```
if (isWinner(a) && isPrime(a)) // && hat 2 Operanden
    return true;
```

Die rechenzeitintensivere Methode sollte hier zuletzt ausgeführt werden (ggfs. Zeitersparung, wenn die „schnellere" erste Berechnung bereits `false` ist).

Auch bei Tests von z.B. Arrays ist die Reihenfolge wichtig:

```
if (arr.length==0 || arr==null) // || hat 2 Operanden
```

wird z.B. abgekürzt, wenn arr gleich `null` ist. Korrekt wäre:

```
if (arr==null || arr.length==0)
```

Die „von links nach rechts"-Regel zeigt sich auch hier:

```
System.out.println("a+b="+a+b); // sicher anerwartet, da 3+1=11
System.out.println("a-b="+a-b); // Compilerfehler, wegen String-b
```

- **Kombination von Berechnung und Zuweisung** wie bei `+=` gibt es auch für andere Operationen: `-=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=`
  Es sei x eine Variable, op ein „allgemeiner" Operatorsymbol und a ein Operand.

dann ist	x op= a;	ausführlich schreibbar als	x = x op a;	
z.B. ist	a *= 1;	gleichbedeutend mit	a = a * 1;	
und	x /= a;	gleichbedeutend mit	x = x / a;	
und	b |= ampelRot();	gleichbedeutend mit	b = b	ampelRot();	

- Anwendung von Wissen aus der TI:
  - Bsp. Umsetzung Karnaugh-Plan

x32 \ x01	00	10	11	01
00	0	0	0	0
01	0	1	1	0
11	0	1	1	0
10	0	0	0	0

```
return s0 && (x2 || !x3);
```

- Beim Increment und Decrement per `++` bzw. `--` gibt es zu beachten:
  - `i++` erhöht den Wert in i um 1, nutzt dann aber den **alten** (kleineren) Wert (z.B. für Ausgabe, in Berechnung oder Zuweisung). Auswirkung:
    `i=i+1;` // **Dies verändert den Wert in i nicht!**
    `i++;` // Dies ist der übliche Befehl, um eine Variable um 1 zu erhöhen!
  - Auch `++i` erhöht den Wert in i, nutzt dann aber dann den **neuen** (höheren) Wert, d.h. `i++=i;` würde funktionieren, ist aber unnötig umständlich.

Heutige Themen

- Organisatorisches
  - Tutorien und Seminare diese Woche
  - Bedarf an virtueller Gruppenarbeit?
- Arbeit mit Informationsquellen
- rekursive Methodenaufrufe (VT/NT, Vergleich mit Schleifen)
- Ansätze zum Testen (z.B. temporäres interaktives Debuggen vs. systematischer automatisierter Nachweis der Fehlerfreiheit)

Empfohlene externe Informationsquellen

- Java-Dokumentation (Original bei Oracle), siehe z.B.
  `https://docs.oracle.com/javase/7/docs/api/java/lang/String.html`
- „Java ist auch eine Insel" (frei online oder auch als Buch), siehe
  `http://openbook.rheinwerk-verlag.de/javainsel`

- Angebote der Uni-Bibliothek (z.B. Lehrbuchsammlung):
  - *Sattler*: „Algorithmen und Datenstrukturen"
  - *Ullenboom*: „Java ist auch eine Insel"
  - *Mössenböck*: „Sprechen Sie Java?"
  - *Ratz+...*: „Grundkurs Programmieren in Java"
  - *Krüger+Hansen*: „Handbuch der Java Programmierung"
  - ...
- Stackoverflow (`https://stackoverflow.com`) stackoverflow
- sonstiges via Suchmaschine der Wahl

Buch erst eine Zeit testen, dann ggfs. kaufen (aber **nicht „durchlesen"!**).

Rekursive Methodenaufrufe

Rekursiver Methodenaufruf: Eine Methode ruft sich direkt selbst oder indirekt über eine andere Methode (oder mehrere andere Methoden) auf.

Rekursive Methodenaufrufe:

- sind nicht ungewöhnlich,
- ermöglichen manchmal **elegante** (intuitive/kurze) Lösungen, z.B.:
  - Traversierung von Bäumen (spätere Vorlesung),
  - Fakultätsberechnung (Idee: $n! = n \cdot (n-1)!$),
- sind durchaus **„gefährlich"**,
  → gezielter Rekursionsabbruch im Programm vs. **Programmabsturz**,
- sind grundsätzlich gleichmächtig mit Schleifen,
  → jede Rekursion ist in eine Iteration umwandelbar und umgekehrt.

Es ist zu klären, was ist „gefährlicher" ist:
Endlosschleife bei Iteration oder Absturz bei Rekursion?

einführendes Beispiel: Fakultätsberechnung

Grundidee:

$$n! = \underbrace{1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot (n-1)}_{= (n-1)!} \cdot n$$
$$= (n-1)! \cdot n$$
$$fakRek(n) = fakRek(n-1) * n \qquad \text{// hergeleitetes Umsetzungsmuster}$$

Die direkte Umsetzung der Idee ist aber problematisch!

```
public static int fakRek(int n) {
   return fakRek(n-1)*n;  // "endlose" Rekursion -> Programmabsturz
}
```

Es wird unbedingt ein gezielter **Rekursionsabbruch** benötigt!

```
public static int fakRek(int n) {
   if (n<=1) // keine Rekursion für negative Zahlen sowie 0 und 1
       return 1; // triviale Fälle: 1! == 0! == 1
   return fakRek(n-1)*n;
}
```


einführendes Beispiel: Fakultätsberechnung (2)

```
public static int fakRek(int n) {
   if (n<=1) // Abbruchbedingung
       return 1; // hier endet die Rekursion
   return fakRek(n-1)*n;
}
```

Wertetabelle zur Veranschaulichung der Rekursion
für `fakRek(4)` = 1*2*3*4 = 24:

Aufruf	n	Rückgabe
① **fakRek(4)**	4	⑧ fakRek(3)*4 = ? * 4 = **?**, **fakRek(4)** ist nun bekannt. Fertig!
② fakRek(3)	3	⑦ fakRek(2)*3 = ? * 3 = ?, fakRek(3) ist nun bekannt.
③ fakRek(2)	2	⑥ fakRek(1)*2 = ? * 2 = ?, fakRek(2) ist nun bekannt.
④ fakRek(1)	1	⑤ 1 , fakRek(1) ist nun bekannt.

Abarbeitungsreihenfolge : ①, ②, ③, ④, ⑤, ⑥, ⑦, ⑧

Beispiel: Quersummenberechnung

Die Quersumme einer Zahl ist die Summe ihrer (Dezimal-)Ziffern.

```
public static int qsum(int x) {
   if (x<=9) // Abbruchbedingung
       return x; // hier endet die Rekursion
   return qsum(x/10) + x%10;
}
```

Wertetabelle zur Veranschaulichung der Rekursion
`qsum(3624)` = 3+6+2+4 = 15:

Aufruf	x	Rückgabe
① **qsum(3624)**	3624	⑧ qsum(362)+4 = ? + 4 = **?**, **qsum(3624)** ist nun bekannt. Fertig!
② qsum(362)	362	⑦ qsum(36) +2 = ? + 2 = ?, qsum(362) ist nun bekannt.
③ qsum(36)	36	⑥ qsum(3) +6 = ? + 6 = ?, qsum(36) ist nun bekannt.
④ qsum(3)	3	⑤ 3 , qsum(3) ist nun bekannt.

Abarbeitungsreihenfolge : ①, ②, ③, ④, ⑤, ⑥, ⑦, ⑧

Frage: Ist `qsum` für alle `int`-Werte sinnvoll?

Beispiel: Fibonacci-Zahlen

- Zahlenreihe, welche Vermehrung von z.B. Kaninchenpaaren simuliert.
- Zum Zeitpunkt $t = 0$ gibt es 1 Paar (z.B. 1 Kaninchenpaar, Annahme: es gibt immer bzgl. dem Geschlecht passende Paare).
- Zum Zeitpunkt $t = 1$ gibt es auch 1 Paar, nun ist es geschlechtsreif.

- Ab dem Zeitpunkt $t = 2$ berechnet sich die Anzahl der nun lebenden Paare aus:
  - der Anzahl der Tiere zum Zeitpunkt $t - 1$ (denn alle diese Tiere **leben weiter**, keines der Tiere stirbt)
  - plus der Anzahl zum Zeitpunkt $t - 2$ (diese sind alt genug und bekommen 2 Kinder pro Paar, d.h. ihre Anzahl verdoppelt sich durch **Neugeburten**).

```
public static int fibo(int t) {
   if (t<0)   // negative Zeit -> keine Tiere vorhanden
       return 0;
   if (t<2) // Abbruchbedingung, Rekursionsende
       return 1; // zum Zeitpunkt 0 und 1 gibt es nur 1 Paar.
   return fibo(t-2)+fibo(t-1);
}
```

Zeitpunkt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Anzahl	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597

Warum Absturz und nicht eine Art von Endlosschleife?

Iteration (Schleifen): Speicherung der aktuellen Werte (z.B. Laufvariable und Fakultät) *in den selben* lokalen und/oder Parametervariablen.

Rekursion (Methodenaufrufe): Speicherung dieser aktuellen Werte pro Aufruf *in neuen* „Instanzen“ dieser Variablen auf dem sogenannten „call stack“ oder „(Aufruf-)Stack“.

Gefahr: Abbruch durch **Stackoverflow** schon bei einigen tausend Aufrufen!

praktisches Beispiel zum Austesten, Aufruf z.B. in `main` per `count(1);`

- Pro Aufruf ein weiterer Eintrag auf dem **Stack** mit den aktuellen Werten der Parameter und lokalen Variablen sowie der Rücksprungadresse.
- Je nach Java-Installation gibt es eine maximale Stackgröße (bzw. Verschachtelungstiefe). Bei Überschreitung → **Stackoverflow** (Abbruch des Programms durch eine „Exception“).

```
public static void count(int n){
    if (n>1000000)
        return; // Abbruch wird üblicherweise nie erreicht
    System.out.println(n); // z.B. bis 10821, dann Abbruch
    count(n+1); // bis zum Abbruch durch Stackoverflow!!!
} // Hinweis: Wird die Methode beendet, wird auch der Stackeintrag freigegeben.
```


Testen in AuP

Was gehört zum Testen in AuP?

- Systematischer Nachweis der Korrektheit: Tut das Programm das, was es laut Aufgabenstellung soll? → „**Verifikation** des Programms“
- Lokalisieren des Fehlers (**Fehlerfindung**, wenn eine Abweichung festgestellt wird),
- **Hinterlegung** des Fehlerfindungswissens (z.B. für automatisierte Verifikation):
  - Sammlung kritischer/repräsentativer Fälle,
  - jeder Zeit sollen Tests gestartet werden können, (z.B. nach Änderungen der Software, z.B. „Optimierungen“)
  - **Ziel:** sichere Vermeidung von alten Fehlern!

Empfehlung: Stets auch kritische Prüfung/Abwägung, ob Aufgabenstellung geeignet ist, das entsprechende Problem zu lösen.
→ „**Validierung** des Programms bzgl. Einsatzzweck“

Bewährte Wege um (in AuP) Software zu testen

Was und *wozu*:

- Debuggen im Kopf: Erfahrung/Übung sehr wichtig! *(Fehlerfindung)*,
- Debuggen in IDE/Eclipse *(Fehlerfindung)*,
- Ausgaben via `System.out.println()` (temporär zur *(Fehlerfindung)* oder auch zur Protokollierung, falls später Fehlerfindung nötig ist),
- gezielte Sammlung von Testfällen (pro Methode), z.B. in `main` oder speziellen „Testmethoden“ *(Hinterlegung)*,
- Zufallswerte und/oder spezielle (kritische) Werte zum Testen benutzen *(Verifikation/Fehlerfindung)*,
- Speichern von kritischen/repräsentativen Testwerten (ggfs. auch zugehörige Ergebniswerte) in Arrays und Abarbeiten der Testwerte in Schleifen *(Hinterlegung)*,
- Ergebnisse von verschiedenen Methoden, die das gleiche tun sollen, vergleichen *(Verifikation)*.

Faustregeln zum Erstellen von Testfällen

Kriterien für eine gute Testwertsammlung sind z.B.:

- Sie sollte Werte enthalten (aufsammeln), die schon zu Fehlern führten.
- Ihre Werte erzwingen, dass jeder „Codepfad“ (z.B. in `if/else`- oder `switch/case`-Verzweigungen) durchlaufen wird.
- Ihre Werte erzeugen Spezialfälle, die zum Absturz führen würden (z.B. nicht existierende Objekte (OOP) oder Arrays).
- Ihre Werte erzeugen numerisch problematische Spezialfälle (z.B. Division durch 0 oder Wurzel aus einer negativen Zahl).
- Außerdem lohnt sich oft die Erzeugung leerer Arrays (z.B. wegen Absturz bei negativem oder zu großem Index).

Hauptproblem: kombinatorische Vielfalt aus Obigem eigentlich nötig.

Pragmatische Lösung: Zusätzlich „intelligent“ zufällig gewählte Testwerte verwenden (z.B. für Testläufe über Nacht).

Heutige Themen

- Anwendungsbeispiele für die `Math`-Klasse
- Nützliche Programmiermuster
  - Erzeugung von Zufallszahlen
  - Runden auf bestimmte Stellenanzahl
  - Serialisierung von 1d-„Elementen“ in einen String
  - Serialisierung eines 2d-Feldes in einen String
- Weitere Beispiele für Algorithmen (teils rekursiv und iterativ)
  - GGT
  - Quadratwurzel einer positiven Zahl

Oft genutzte Konstanten und Methoden von Math

- siehe z.B.:
  - Java-Dokumentation (Original bei Oracle):
    `https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html`
  - Merkzettel (in Moodle):
    `https://moodle2.tu-ilmenau.de/mod/resource/view.php?id=71016`
- Konstanten:
  - `PI`, z.B.: `double agl_rad = agl_deg*Math.PI/180; // Umrechnung Deg->Rad`
  - `E` , z.B.: `double e = Math.E; // für Kurzschreibweisen in Formeln`
- Methoden:
  - `pow`, z.B.: `double y = Math.pow(x, 1.0/3); // 3. Wurzel aus x`
  - `sin`, z.B.: `double sinX = Math.sin(x*Math.PI/180); // falls x in Grad`
  - ...

Erzeugung von Zufallszahlen - Würfelbeispiel

Die **Generator**-Methode `Math.random()` liefert stets „nur“ `double`-Werte [0...1). Mit typischen Programmiermustern (z.B. in parametrisierten Methoden) erfolgt die Transformation vom Bereich [0...1) in den Zielbereich bzw. die Zielmenge.

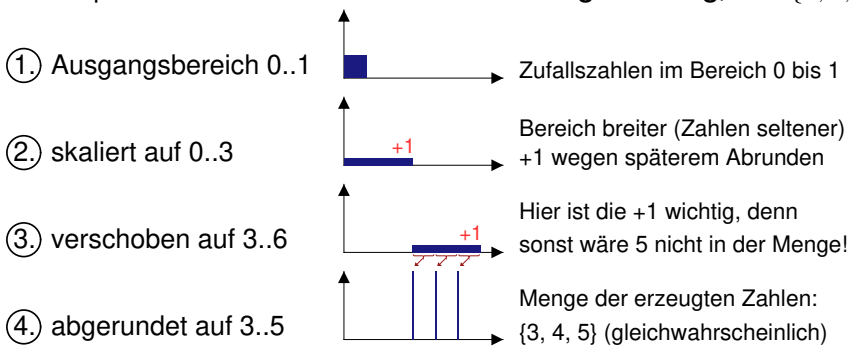
```
// Ermittlung der Zufallszahlen von 1 bis 6 per "Würfel"
int z1 = (int)(Math.random()*6+1); // per direkter Berechnung
int z2 = wuerfeln();               // per Aufruf einer Methode
```

Passende Methoden findet man in Java-Bibliotheken oder schreibt sie selbst.

```
public static int wuerfeln(){    // direkt implementierte Version
    return (int)(Math.random()*6+1); // etwas schneller als wuerfeln_2
}

public static int wuerfeln_2(){ // Version, welche rand nutzt
    return rand(1, 6); // siehe Methodendefinition unten
}

public static int rand(int von, int bis){
    return (int)(Math.random()*(bis-von+1)+von);
}
```


Erzeugung von ganzzahligen Zufallszahlen - Erklärung

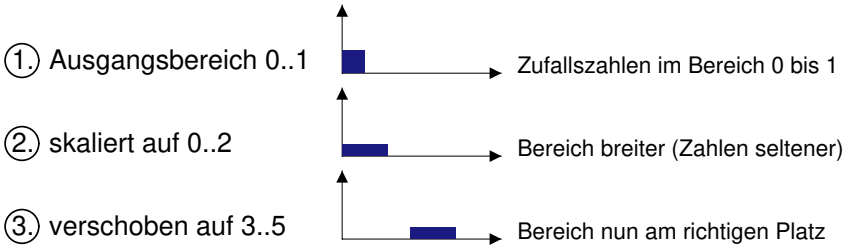
Zahlenbeispiel zur **Intervalltransformation** und **Mengenbildung**, Ziel: $\{3, 4, 5\}$




```
public static int zufallAusfuehrlich(int von, int bis){
    // im Beispiel sei von=3 und bis=5
    double z = Math.random(); // 0 bis 0.999999...
    int w = bis-von+1;        // "Breite", hier w=5-3+1=3
    z *= w;                   // Aufspreizung : 0 bis 2.999999...
    z += von;                 // Verschieben  : 3 bis 5.999999...
    return (int)z;            // Mengenbildung: 3 oder 4 oder 5
}
```

Erzeugung von Gleitkommazufallszahlen - Erklärung

Zahlenbeispiel zur **Intervalltransformation**, Ziel: [3, 5**)** (hier keine Mengenbildung)




```
public static double zufallAusfuehrlich(double von, double bis){
    // im Beispiel sei von=3 und bis=5
    double z = Math.random(); // 0 bis 0.999999...
    double w = bis-von;       // "Breite", hier w=5-3=2
    z *= w;                   // Aufspreizung : 0 bis 1.999999...
    z += von;                 // Verschieben  : 3 bis 4.999999...
    return z;                 // Interval mit praktisch [3, 5]
}
```

Weitere Beispiele: siehe Merkzettel, z.B.:
zufällige Felder, robustere Parameternutzung

Runden auf bestimmte Stellenzahl

Zahlenbeispiel zum **Runden**, Ziel hier: 2 Nachkommastellen
→ aus 12.34567 soll z.B. 12.35 werden

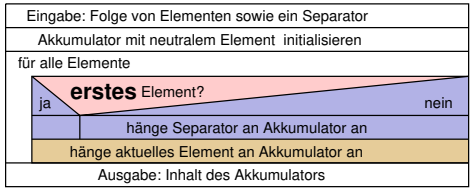
```
static public double runde_ausfuehrlich(double d, int n){
    // im Beispiel ist d=12.34567 und n=2
    double f = Math.pow(10, n); // passenden Faktor zu n berechen
                                // hier ist f = 100
    double o = 0.5; // Offset für korrektes Auf-/Abrunden
    if (d<0)
       o = -0.5; // negatives d braucht negativen Offset
    double g = d*f+o; // g = 1235.067 ( = 1234.567 + 0.5)
    long h = (long)g; // h = 1235 (Nachkommastellen "abhacken")
    double a = h/f;   // a = 12.35 ("Zurückverschieben")
    return a; // auf n Stellen gerundete Zahl zurückgeben
}
```

Hinweis: Zur formatierten Ausgabe bzw. Umwandlung in formatierte Zeichenketten sind u.U. noch Leerzeichen oder Nullen anzuhängen!

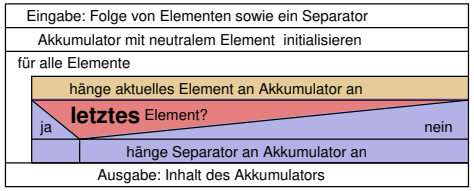
Beispiel: `double r = runde(12.30456, 2);` hier wird `r` zu `12.3` und so auch in Strings oder Ausgaben geschrieben

Serialisierung von 1d-„Elementen“ in einen String

vorgeschaltetes Separatoreinfügen (einfacher)



nachgeschaltetes Separatoreinfügen (schwieriger)



vorgeschaltetes Einfügen hat einen sehr simplen Test (Beispiel für 1d-Array)

```
static public String toString(int[] a, String sep){
    String s = ""; // Initialisierung mit neutralem Element (leerer String)
    for (int i=0; i<a.length; i++){ // Schleife über alle Feldelemente
        if (i>0)    // erst nach dem ersten Element muss angehangen werden
            s+=sep; // vor allen weiteren muss jedoch der Separator sein
        s+=a[i];    // dann kommt das eigendliche Element
    }
    // nun sind in s alle Elemente gespeichert
    return s;  // s wird zurückgegeben
}
```

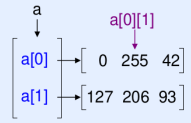
nachgeschaltetes Einfügen hat einen etwas schwierigeren Test (Beispiel für 1d-Array)

```
        s+=a[i];          // zuerst kommt das eigendliche Element
        if (i+1<a.length) // nur vor dem letzten Element wird angehangen
            s+=sep;       // nach allen anderen muss der Separator stehen
```

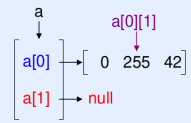

Serialisierung eines 2d-Feldes in einen String

Beispiel: Rückgabe eines 2d-Arrays als Java-Quellcode-String

```
public static String toJavaSrcString(double[][] a){
    if(a==null) // Spezialfall: Feld a ist nicht existent!
        return "null"; // und Schluss!

    String s="{";
    for (int i=0; i<a.length; i++){
        if (i>0) // vorgeschaltet
            s+=",\n";

        if (a[i]==null){ // Spezialfall: inneres Feld ist nicht existent, z.B. a[1]==null
            s+="  null";   // in s vermerken
            continue;    // und nächstes untersuchen...
        }

        s+="  {";  // existierendes inneres Feld startet mit {
        for (int j=0; j<a[i].length; j++){

            if (j>0) // vorgeschaltet
                s+=", ";

            s += a[i][j]; // "endlich mal" ein double-Wert
        }
        s+="}"; // existierendes inneres Feld endet mit }
    }
    return s+"}";
}
```



// zum Testen: `int[][] a = {{0, 255, 42},{127, 206, 93}};`



// zum Testen: `int[][] a = {{0, 255, 42}, null};`

Größter gemeinsammer Teiler (GGT): rekursiv

- Wenn t Teiler von x und y (2 nicht neg. Zahlen) ist, dann teilt t auch die Differenz.
- Der ggT kann nicht größer als die Differenz sein.

Lösungsidee: Schrittweise Verkleinerung des Problems durch Ersetzen der größeren der beiden Zahlen durch die Differenz.

```
public static int ggT_rek(int x, int y) {
    if (x<=0 || y<=0) return -1;               // Fehler: ggT nicht definiert
    if (x==y)         return x;                // fertig (Rekursionsabbruch)
    if (x>y)          return ggT_rek(x-y, y  ); // x durch Differenz ersetzen
                      return ggT_rek(x  , y-x); // y durch Differenz ersetzen
}
```

Verbesserungsidee: Insbesondere wenn eine Zahl viel größer als die andere ist, muss sehr oft die kleinere von der größeren abgezogen werden → effizienter ist es, gleich den Rest der Division als „letztliche Differenz“ zu verwenden.

```
public static int ggT_rek2(int x, int y) { // effizienter
    if (x==0)       return y;                // fertig (Rekursionsabbruch)
    if (y==0)       return x;                // fertig (Rekursionsabbruch)
    if (x<0 || y<0) return -1;               // Fehler: ggT nicht definiert
    if (x==y)       return x;                // fertig (Rekursionsabbruch)
    if (x>y)        return ggT_rek2(x%y, y  ); // spart x/y-1 Durchläufe
                    return ggT_rek2(x  , y%x); // spart y/x-1 Durchläufe
}
```


Größter gemeinsammer Teiler (GGT): iterativ

```
public static int ggT_itr(int x, int y) { // effizienter kurzer Code
    do {   // Annahme x>y (dann funktioniert %), sonst Tausch: r=x, x=y, y=r
        int r = x % y;      // Differenz in Hilfsvariable speichern
        x = y;              // größere wird zur kleineren
        y = r;              // kleinere wird zur Differenz
    } while (y > 0);        // stoppe, wenn y (die Differenz) <= 0 ist
    return x;               // Die größere Zahl (x) ist dann der ggT.
}
```

Frage: Wo und bei welchen Parameterwerten tritt im obige Quellcode ein Problem (Exception bzw. Programmabbruch) auf.

Quadratwurzel einer positiven Zahl (Algor. v. Heron)

Die Wurzelberechnung hat eine spezielle Methode zum Rekursionsstart:

```
static public double sqrt(double x){ // typische Rekursionsstartmethode
    return sqrt(x, 1.0, x, 1e-6);
}
```

- Die eigentliche rekursiv aufgerufene Methode hat 4 Parameter, wobei `f` und `eps` nicht verändert werden (werden nur „durchgereicht“).
- `f` repräsentiert das obige `x` als Flächeninhalt (wird nicht geändert).
- Es gilt stets `b*h=f` (Breite * Höhe = Fläche).
- Wenn `b` gleich `h` ist (Quadrat), dann wurde die Wurzel „genau“ berechnet.
- Ein Gleichheitstest mit wählbarem Epsilon `eps` ermöglicht schnelleren Abbruch.

```
static public double sqrt(double b, double h, double f, double eps){
    if (b<0||h<0||f<0||eps<0)
        return -1; // Fehlerfall
    if (Math.abs(b-h)<=eps)
        return b;  // Rekursionsabbruch
    b = (b+h)/2;   // b und h nähern sich weiter an (b wird kleiner)
    return sqrt(b, f/b, f, eps); // neues h = f/b wichtig für f=h*b
}
```


Teil II

Objektorientierte Programmierung (OOP)

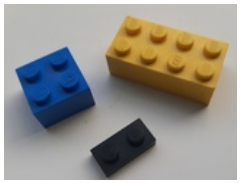
Heutige Themen

- Organisatorisches
  - ab jetzt (bis auf Widerruf) Vorlesung mittwochs um 15Uhr,
  - ab KW22 Tutoriumsaufgaben stärker auf aktuelle Inhalte bezogen, z.B. letzte Woche T8 (Math-Klasse, Zufall, Histogramme, Formatierung) und diese Woche T9 (Beispiel für den schrittweisen Aufbau einer Box-Klasse)
  - Folien V8 auch schon online (2. Foliensatz zu OOP)
- OOP - Wozu? (Motivation)
- Allgemeines zur OOP (Grundidee, Ziele)
- Variablentypen
  - wo definiert: Objektvariablen, Klassenvariablen, ...
  - Inhalt: primitive Datentypen (direkte Speicherung) **vs.** Referenzdatentypen (Zeiger auf Speicherbereich)
- Arbeit mit Referenzdatentypen (Erzeugen, Kopieren, „Löschen“)
- Konstruktoren (Erzeugen von Objekten)

OOP - Wozu?

Warum sollte man anders programmieren, als es bisher gezeigt wurde?

Ein möglicher Grund: Unsere bisherige Art Variablen zu definieren skaliert nicht (ist für „kleine“ Probleme/Programme ok, jedoch nicht für größere)!



Beispiel: Bauteilverwaltung

```
int    breite_1 = 1;
int    hoehe_1  = 1;
int    laenge_1 = 2;
String farbe_1  = "schwarz";

int    breite_2 = 2;
int    hoehe_2  = 3;
int    laenge_2 = 2;
String farbe_2  = "blau";

int    breite_3 = 2;
int    hoehe_3  = 3;
int    laenge_3 = 4;
String farbe_3  = "gelb";
```

← wäre noch ok

nicht mehr praktikabel →

```
...

int    breite_77 = 4;
int    hoehe_77  = 1;
int    laenge_77 = 6;
String farbe_77  = "grau";

...

int    breite_4856 = 1;
int    hoehe_4856  = 3;
int    laenge_4856 = 8;
String farbe_4856  = "blau";

...
```


Grundidee der OOP

- Viele Dinge/Zusammenhänge/Abläufe/... der realen Welt lassen sich als „Objekte“ beschreiben - aus Sicht der Programmierung bedeutet das:
  - als **Daten**, die ein solches Objekt beschreiben (**Objektvariablen**)
  - als **Methoden**, die mit diesen Daten arbeiten (**Objektmethoden**: lesen, transformieren und/oder schreiben Objektvariablen).
- Zusammengehörige **Daten** und **Methoden** werden in einer **Klasse** („**Blaupause**“, „**Objektfabrik**“, „**Generator**“) beschrieben.

  Eine Klasse dient zur Erzeugung von beliebig vielen **Objekten** („**Variablenmengen**“, „**Dingen**“, „**Instanzen**“).

  Vorteil: Einmal in Klassendefinition beschrieben lassen sich beliebig viele Objekte erzeugen (per **new**-Operator, wurde im Bild unten z.B. dreimal verwendet).

genau einmal im Speicher

Klasse: Sensor
int Messpunktnummer float aktuellerWert

0- bis n-mal im Speicher (hier: drei Objekte)

Objekt: sensor_1	Objekt: sensor_2	Objekt: sensor_3
4643 23.1	4645 25.7	4711 19.8

Ziele der OOP

- Verbergen von Details: andere Progammier sollen wissen **WAS** gemacht wird, aber nicht **WIE** → **Kapselung**, Vorteile:
  - **Zugriffsschutz**: einfache Steuerung des Zugriffs auf Daten und Methoden mittels Schlüsselwörtern (`public`, `protected`, „ohne“, `private`)

    Beispiel: Zugriff auf Daten über Methoden erzwingbar (z.B. eine Objektvariable ist `proteced` und ihre Lesemethode ist `public`)
  - **Wiederverwendung**: einmal geschriebene (und getetstete, dokumentierte) Klassen lassen sich beliebig oft woanders einsetzen → **Zeiteinsparung** beim Entwickeln, weitere Tests bei Nutzung in verschiedenen Kontexten (Qualitätssicherung!)
- Klassen, die sich nur wenig unterscheiden (z.B. LKW und PKW) können **Quellcode „teilen“** (z.B. gemeinsamer Quellcode in Klasse Fahrzeug zusammenfassen + PKW und LKW erben den Quellcode von Fahrzeug) → Aufbau von Klassenhierarchien → **Übersichtlichkeit**, Zeiteinsparung

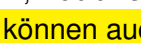
Arten von Variablen: 1. Wo/Rolle/„Lebensdauer“

Klassen können Variablen enthalten

- Klassenvariablen (Schlüsselwort **static** als Modifier), existieren genau einmal pro Klasse
- Objektvariablen (Normalfall), existieren einmal pro Objekt (d.h. 0...∞ mal)
- Beispiel:

```
public class Vars {
    public int ov=1; // Objektvariable, eigener Wert für jedes Objekt
    public int ow=0; // Objektvariable, eigener Wert für jedes Objekt
    public static int kv=0;// Klassenvariable, gleich für alle Objekte

    public int m(int pv){  // Parametervariable, nur in Methode bekannt
        int lv=2*pv;       // lokale Variable, nur in Methode bekannt
        return lv;         // Wert kann aber nach außen gebbar
    } // Parameter- und lok.Variablen existieren nur bei Methodenaufruf!
}     // (auf dem Stack, verschwinden beim Beenden der Methode / return)
```


Arten von Variablen: 2. Was speichern sie?

- **primitive Datentypen**: `int`, `double`, `boolean` ...
  Variablen sind nie `null`, sind gut für schnelleres Rechnen, ... siehe:
  `http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_02_003.htm`
- **Referenztypen**: Verweise auf Objekte, z.B. Felder primitiver Datentypen, `String`, `Integer`, `Double` ...
  Variablen von Referenztypen können auch `null` sein! 

  siehe:
  `http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_03_005.htm`

  Spezialfall: Wrapperklassen für primitiven Datentypen, Beispiele:

primitiv	Wrapper
long	**L**ong
double	**D**ouble
boolean	**B**oolean
char	**Character**
int	**Integer**

Referenzdatentypen (I)

- Objektreferenzen verweisen auf Objekte  oder `null` 

  (wie Zeiger in C oder C++)
- **Anlegen von Objekten** (Erzeugen) erfolgt entsprechend der Klassendefinition („**Blaupause**“) per **new**-Operator und **Konstruktor**), z.B.:

```
new Double(1);
```

  Erzeugt nur ein Objekt, jedoch keine Referenz darauf!!! 1
- **Anlegen von Variablen** (zur Speicherung von Referenzen auf Objekte) erfolgt wie bei primitiven Datentypen (**int i** : 1. **Typ**, 2. **Name**), z.B.:

```
Double d1; // enthält null als default-Wert und NICHT 0.0!
```

  Erzeugt nur eine (Referenz-)Variable auf ein nicht existierendes Objekt!!!

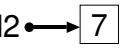
  d1 

- $\rightarrow$ Beide obige Befehle sind untypisch. Meist wird die Kombination aus dem Variable- und Objektanlegen verwendet ...

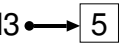
Referenzdatentypen (II): Kopieren der Referenz per =

- Variablen von Objekten sind immer Referenzen, speichern also „Zeiger“ auf den Speicherbereich, wo die Objektdaten liegen

```
Double d2 = new Double(7); // hat Referenz zum Double-Objekt mit 7
```

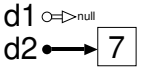
  d2 

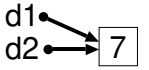

```
Double d3 = new Double(5); // hat Referenz zum Double-Objekt mit 5
```

  d3 

- Zuweisung wie `o1 = o2;` kopiert nur den Zeiger, d.h. o1 und o2 zeigen auf das selbe Objekt, Beispiel:

```
d1=d2; // d1 und d2 zeigen nun beide auf das Double-Objekt mit 7
```

  vorher: 

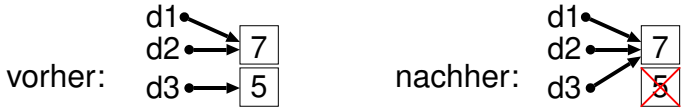
  nachher: 

Referenzdatentypen (III): automatisches Löschen

- Wenn ein Objekt keine Variablen mehr hat, die auf es zeigen, dann wird es automatisch gelöscht (bei der **Garbage Collection**).

```
d3=d2; // wie d1 und d2 zeigt nun auch d3 auf den Double mit 7
```

Nichts zeigt mehr auf den Double mit der 5
→ dieses Objekt kann nun automatisch gelöscht werden.

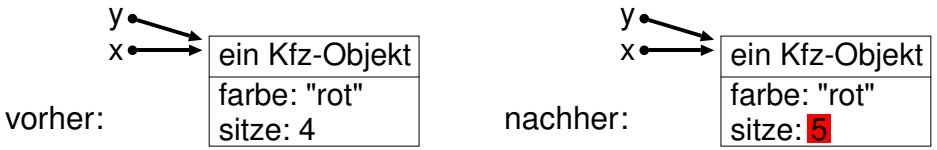


- Wie kann auch der Double 7 zum Löschen freigegeben werden?
  - Programm beenden,
  - `d1`, `d2` und `d3` sind lokale Variablen (also in einer Methode) und die Methode wird beendet,
  - `d1=null; d2=null; d3=null;`
  - ...

Referenzdatentypen (IV): Ändern von Objektdaten

Merke: Zeigen `o1` und `o2` auf das selbe Objekt, dann wirken sich Änderungen in `o1` direkt auf `o2` aus!

```
public class Kfz {
    int sitze = 4;         // Objektvariable, eigener Wert jedes Objektes
    String farbe = "rot"; // Objektvariable, eigener Wert jedes Objektes

    public static void main(String[] args){
        Kfz x = new Kfz();  // erzeugt mit Vorbelegung 4 und rot
        Kfz y = x;          // y zeigt auf selbes Kfz wie x!
        System.out.println("Sitze vorher : "+x.sitze);
        y.sitze = 5;        // Änderung des Objektes, auf das y zeigt
        System.out.println("Sitze nachher: "+x.sitze); // ändert auch x!
    }
}
```



Konstruktoren I

- pro Konstruktoraufruf wird ein Objekt erzeugt
- ein **Konstruktor** ist vergleichbar mit einer Methode, aber:
  - für ihn wird kein Rückgabetyp angegeben (auch nicht `void`)
  - Konstruktoren heißen stets so wie die Klasse
- wichtigster Zweck ist die individuelle Initialisierung der **Objektvariablen** des erzeugten Objektes (z.B. per Konstruktor-**Parameter**)
- Beispiel:

```
public class Kfz {
    int sitze = 4;         // Objektvariable, eigener Wert jedes Objektes
    String farbe = "rot"; // Objektvariable, eigener Wert jedes Objektes
    int km=0;              // Objektvariable, eigener Wert jedes Objektes
    public Kfz(){ // default-Konstruktor, stets parameterlos
        // hier könnten z.B. obige Werte geändert werden
    }
    public Kfz(String f, int sitze){ // Konstruktor mit 2 Parametern
        farbe=f; // die Objektvariable farbe erhält den Parameter f als Wert
        this.sitze = sitze; // this. ist wichtig, sonst sinnlos: sitze=sitze;
        // km bleibt hier bei 0
    }
}
```


Konstruktoren II

Beispiel mit verschiedenen Konstruktoren

gezeigte Programmiermuster:
- Zählen der bisher generierten Objekte
- fortlaufende Id pro Objekt
- Definition alternativer Konstruktoren

```
public class Box {
    // Objektvariablen: individuell für jedes Objekt
    double breite = 0.0;
    double hoehe = 0.0;
    double tiefe = 0.0;
    String serNo = "NN";

    // Klassenvariable (nur einmal pro Klasse!), eignet sich als Zähler
    private static long genBoxes = 0; // Anzahl der erzeugten Boxen

    public Box(){ // default-Konstruktor, stets parameterlos
        genBoxes++; // Aktualisierung der Klassenvariable
        serNo="box"+genBoxes;
    }
    public Box(double volumen){ // dient als Konstruktor für einen Würfel
        genBoxes++; // Aktualisierung der Klassenvariable
        serNo="box"+genBoxes;
        breite = Math.pow(volumen, 1.0/3);  // 3. Wurzel des Volumens
        hoehe  = breite;
        tiefe  = breite;
    }
    public Box(double grundfl, double h){ // dient als Konstruktor für einen Quader
        genBoxes++; // Aktualisierung der Klassenvariable
        serNo="box"+genBoxes;
        breite = Math.sqrt(grundfl); // Quader hat stets eine quadrat. Grundfl.
        hoehe  = h;
        tiefe  = breite;
    }
}
```

Heutige Themen

- Zugriffskontrolle auf Methoden und Variablen
- Klassenmethoden vs. Objektmethoden
- Klassen als abstrakte Datentypen, Objekte als Parameter, unveränderliche Objekte (immutable)
- Packages : „Ordnungshilfe“ für Klassen
  - zuerst: Methode (Folge von Anweisungen)
  - dann: Klasse (Menge von Methoden und Variablen und ...)
  - nun auch: Packages (hierarchisch angeordnete Mengen von Klassen)
- ausgewählte von Java bereitgestellte Klassen (Object, String)
- Getter und Setter: Objektmethoden zum Lesen und Schreiben von Objektvariablen
- Vererbung (Schnittstellen, (Unter-)Klassen)

Zugriffssteuerung / Sichtbarkeitskontrolle

Bisher meist benutzte „**Modifizierer**“ waren `public` und `static`. Neben `public` existieren weitere Möglichkeiten, den Zugriff auf „**Members**“ zu steuern.

siehe `https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html`

4 Berechtigungsstufen auf „Class Members“

- die „Member“ einer Klasse sind ihre Methoden und Variablen
- **public**: Zugriff von beliebiger Stelle aus erlaubt
- **protected**: Zugriff nur aus eigener Klasse, Klassen des selben Packages oder Unterklassen
- **package-private**: Zugriff nur aus eigener Klasse oder Klassen des selben Packages, **default**-Berechtigung (falls keiner der 3 anderen Modifier)
- **private**: Zugriff nur aus eigener Klasse

Modifier	Klasse	selbes Package	Unterklasse	beliebig
`public`: „öffentlich“	ja	ja	ja	ja
`protected`: „geschützt“	ja	ja	ja	nein
ohne Modifier	ja	ja	nein	nein
`private`: „privat“	ja	nein	nein	nein

Zuordnung zu Objekten oder Klassen

Bisher meist benutzte „**Modifizierer**“ waren `public` und `static`. Um mit einzelnen Objekten zu arbeiten, wird der Modifier `static` weggelassen, denn:

Der Modifier **`static`** macht aus Methoden **Klassenmethoden**. Außerhalb von Methoden definierte Variablen werden per `static` zu **Klassenvariablen**.

siehe `https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html`

Auf welche Member kann direkt zugegriffen werden? direkt: nur durch Angabe des Namens

Art der Methode/Variable	auf Klassenvar. oder -meth.	auf Objektvar. oder -meth.
von Klassenmethode	ja	**nein!!!**
von Objektmethode	ja	ja

Häufiger Fehler:

```
public class Pkw {
    int sitze = 4;        // Objektvariable, eigener Wert jedes Objektes
    public void sitzeRaus(){ // Objektmethode
        sitze = 0; // ok, direkter Zugriff von Objektmethode auf Objektvariable ist möglich
    }
    public static void sitzeRaus_statisch(){ // Klassenmethode
       sitze=0; // FEHLER! da direkter Zugriff von Klassenmethode auf Objektvariable
    } // "error: non-static variable sitze cannot be referenced from a static context"
}
```


Zuordnung zu Objekten oder Klassen (2)

Ausführlicheres Beispiel (inkl. indirektem Zugriff zur Problemlösung):

```
public class Pkw {
    int sitze = 4; // Objektvariable, eigener Wert jedes Objektes (jeder Pkw hat
                   // seine eigene Sitzanzahl)

    public void sitzeRaus(){ // Objektmethode
        sitze = 0; // ok, direkter Zugriff von Objektmethode auf die diesem
                   // Objekt gehörenden Objektvariablen ist immer möglich
    }

    public static void sitzeRaus_statisch(){ // Klassenmethode
       sitze=0; // FEHLER! direkter Zugriff von der Klassenmethode auf
                // die Objektvariable sitze ist nicht möglich
    }

    public static void sitzeRaus_statisch(Pkw p){ // Klassenmethode
        p.sitze=0; // ok, mit p. wird festgelegt, zu welchem Objekt die Sitze
                   // gehören. Dies ist ein indirekter Zugriff über die
                   // Referenzvariable p.
    }

    public static void main(String[] args){ // auch main ist eine Klassenmethode,
                                            // also keinem Objekt zugeordnet!
        Pkw x = new Pkw();  // erzeugt ein Pkw-Objekt mit Vorbelegung: 4 Sitze
        System.out.println("Sitzanzahl : "+x.sitze); // ok, da indirekter Zugriff
        System.out.println("Sitzanzahl : "+  sitze); // FEHLER! direkter Zugriff
    }
}
```

Jede Klasse definiert einen neuen Datentyp!

Klassen lassen sich wie primitive Datentypen verwenden, z.B.:

- Deklaration einer Variable (Parameter, lokal, Klassenvar., Objektvar.) per **Typ Name**,
- Initialisierung einer Variable (lokal, Klassenvar., Objektvar.) per **Typ Name** = **Wert**,
- als Parameter einer Methode, z.B. per Variablenname (**Name**)

Beispiele:

```
Double d1; // enthält null als default-Wert und NICHT 0.0!
Kfz    k1; // enthält null als default-Wert
double p1; // enthält 0.0 als default-Wert
```

```
Double d2 = new Double(7); // hat Referenz zum Double-Objekt mit 7
Kfz    x  = new Kfz();     // erzeugt ein neues Kfz, x zeigt drauf
Kfz    y  = x;             // y zeigt auf selbes Kfz wie x!
double p2 = 7;             // speichert den Wert 7 direkt
```

```
System.out.println(x);  // Ausgabe des Kfz x auf der Konsole
System.out.println(p1); // Ausgabe des Werts der double-Variable
```


Parameterübergabe: call by value

Programmiersprachen haben für die Parameterübergabe verschiedene Konzepte, insbesondere:

- call by value (**Kopie des Wertes**)
  - in der Parametervariable wird eine Kopie des Wertes gespeichert
  - keine Auswirkung auf Originalvariable bei primitiven Datentypen
- call by reference (**Zeiger auf Wert**)
  - die Parametervariable ist die Speicheradresse des Werts
  - Änderungen erfolgen direkt in der ursprünglichen Variablen
  - kennen wir schon (Feld als Parameter oder `String` als Parameter, denn jeder `String` ist ein Objekt!)
- **in Java: stets call by value, ABER:** Variablen, die Objekte speichern, sind stets Referenzen
- implizit in Java also auch call by reference

Objekte als Parameter in Methoden

```
public class Pkw {
    int sitze = 4;        // Objektvariable, eigener Wert jedes Objektes
    String farbe = "rot"; // Objektvariable, eigener Wert jedes Objektes

    public static void sitzeRaus_statisch(Pkw p){ // Klassenmethode
        p.sitze=0;   // p. referenziert die Objektvariable (indirekter Zugr.)
    }  // p wurde als Parameter übergeben.

    public void umspritzen(String farbe){ // Obj.var. gleichnamig mit Param.
        this.farbe = farbe; // this. referenziert die Objektvariable
    }  // this ist das Schlüsselwort für die Referenz aufs aktuelle Objekt.

    public static void main(String[] args){ // Bsp.: mache x und y identisch
        Pkw x = new Pkw();  // erzeugt mit Vorbelegung 4 und rot
        Pkw y = x;          // y zeigt auf SELBES Kfz wie x!
        System.out.println("Sitze vorher : "+x.sitze);
        sitzeRaus_statisch(y);  // Änderung des Objektes, auf das y zeigt
        System.out.println("Sitze nachher: "+x.sitze); // ändert auch x
        x.umspritzen("blau");                          // Änderung von x
        System.out.println("Farbe nachher: "+y.farbe); // ändert auch y
    }
}
```

```
Sitze vorher : 4
Sitze nachher: 0
Farbe nachher: blau
```


Das String Mysterium (I) - Das Experiment

Allgemein gilt: Werden Objekte einer Methode als Parameter übergeben und dort geändert, dann ist die Änderung auch außerhalb der Methode sichtbar (Referenzdatentyp!).

Folgendes Experiment zeigt: Das gilt z.B. für Strings nicht! Warum???

```
public class StringTest {
    public static void m1(String x){ // x ist+bleibt ein lokaler Parameterwert
        x = x+"1";
        System.out.println("x in   m1: "+x);
    }
    public static String m2(String x){ // x ist ein lokaler Parameterwert
        x = x+"2";
        System.out.println("x in   m2: "+x);
        return x;            // neuer Wert von x kann außerhalb genutzt werden
    }
    public static void main(String[] args){
        String s="A";  System.out.println("s zu Beg.: "+s);
               m1(s);  System.out.println("s nach m1: "+s);
               m2(s);  System.out.println("s nach m2: "+s); // Rückgabe verworfen
          s = m2(s);   System.out.println("s nach m2: "+s); // Rückgabe genutzt
    }
}
```

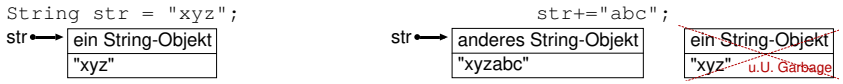
```
s zu Beg.: A   // s zeigt auf "A"
x in   m1: A1  // der lokale Parameter x zeigt erst auf "A", dann auf "A1"
s nach m1: A   // s wurde nicht geändert (zeigt immer noch auf A)
x in   m2: A2  // der lokale Parameter x zeigt erst auf "A", dann auf "A2"
s nach m2: A   // s wurde nicht geändert (zeigt immer noch auf A)
x in   m2: A2  // wie oben
s nach m2: A2  // s enthält nun den Rückgabewert (Referenz auf "A2")
```

Das String Mysterium (II) - Die Antwort

In Java sind Strings stets **unveränderliche** (immutable) Zeichenketten.

siehe `https://docs.oracle.com/javase/tutorial/java/data/strings.html`

- Bei Änderungen (z.B. Anhängen von Zeichen wie `s+="abc"`) werden stets neue Zeichenketten angelegt (erkennbar in Eclipse: andere ID).



- Häufige Änderungen (z.B. schrittweiser Aufbau eines sehr langen Strings) führen zu langsamen Code.
- Lösung: Nutzung von `StringBuilder`-Objekten beschleunigt den Code wieder (z.B. durch Vermeidung von „Umkopieren“)

  siehe `https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html`

Auch Wrapper-Klassen (z.B. `Double`, `Integer`, `Long`) sind immutable. Jede Änderung des Wertes lässt die Referenz auf ein anderes Objekt zeigen!

Packages: Die Java-Package-Hierarchie

Es gibt tausende von Klassen, siehe z.B. **hier**. Zur besseren Verwaltung von Klassen stellt Java „Pakete“ bereit. Klassen können Packages zugeordnet werden. Packages können in eine Package-Hierarchie eingeordnet werden.

Wichtige Schlüsselworte:

- **package:** Einordnung einer Klasse in ein bestimmtes Package
- **import:** Anmeldung der Nutzung eines Packages, ermöglicht im Quellcode die Nutzung von `File` statt `java.io.File`
- Eine Hilfsklasse `SimpleFileHandler` zur Dateiarbeit könnte z.B. so anfangen:

```
package org.tuil.gdv.wtv;   // SimpleFileHandler wird hier eingeordnet
import java.io.File;        // Klasse File        aus dem Package java.io wird genutzt
import java.io.IOException; // Klasse IOException aus dem Package java.io wird genutzt
public class SimpleFileHandler { // hier fängt die eigentliche Klassendefinition an
   ...
```

Alle Klassen des Packages `java.lang` werden stets automatisch importiert, z.B. `System`, `Math`, `String`, `Double`.

siehe `https://docs.oracle.com/javase/7/docs/api/java/lang/package-summary.html`

Wäre das nicht so, dann müsste man z.B. explizit schreiben **oder** dann im Quelltext stets z.B.

```
import java.lang.String;
java.lang.String s="";
```


java.lang.Object: Die Wurzelklasse aller Java-Klassen

Jede Klasse ist automatisch von `java.lang.Object` abgeleitet. `Object` ist somit die Wurzel der Java-**Klassenhierarchie**.

Beispiele für Methoden in `Object` (Alle Objekte **erben** diese!) :

- **String toString():** Umwandlung in einen `String` (z.B. für Ausgaben)
- **int hashCode():** wird u.a. in `toString()` benutzt
- **Class getClass():** wird u.a. in `toString()` benutzt

siehe `https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#toString()`

Vererben bedeutet:

- Alle Unterklassen besitzen automatisch alle Methoden der vererbenden Klasse, d.h. sie sind dort „einfach“ nutzbar (ABER: **Modifier beachten!**).
- Bei Bedarf lässt sich der ursprüngliche Methodeninhalt mit eigenem Code **überschreiben, ABER** die Änderung wirkt sich nur auf Objekte der aktuellen Klasse und ihre Kinder aus!

java.lang.String: Speicherung von Zeichenketten

siehe `https://docs.oracle.com/javase/7/docs/api/java/lang/String.html`

- Erzeugen eines Strings + Speichern in Referenzvariable

```
String s1 = new String("eine Zeichenkette"); // via Konstruktor
String s2 = "eine andere Zeichenkette";      // "automatisch"
```

- Länge eines Strings

```
int len1 = s1.length(); // Methode! ergibt hier 17
```

- Zugriff auf einzelne Zeichen per Index (0...Länge-1)

```
char c = s1.charAt(2);  // ergibt hier 'n'
```

- Vergleich des **Inhalts** von String-Objekten (z.B. für alphab. Sortierung)

```
boolean b = s1.equals(s2);  // false, da inhaltlich ungleich
int i = s1.compareTo(s2);   // alphabetische Ordnungsrelation
                            // hier: i = 'Z'-'a' = 0x5A-0x61 = -7
                            // allg. für compareTo: i<0 -> s1<s2
                            //                      i=0 -> s1=s2
                            //                      i>0 -> s1>s2
```

Die Codes der ersten sich unterscheidenden Zeichen werden verglichen.

java.lang.String: Speicherung von Zeichenketten (2)

- Problem: Vergleich per ==
  Experiment: **Referenzvergleich** vs. **Inhaltsvergleich**

```
String x = "ABC";
String y = "AB"+(Math.random()<1?"C":"D"); // Zufallswert ist immer [0..1)
           // y hat nun auch immer den Inhalt "ABC", aber andere Referenz
System.out.println(x==y); // gibt false aus, denn
                          // x und y zeigen auf verschiedenen Speicher
System.out.println(x.equals(y));  // gibt true aus, denn es findet ein
                                  // zeichenweiser Inhaltsvergleich statt
```

  == vergleicht also nur die „Zeiger“ auf den Speicher. Zeigen sie auf den selben Speicher (das selbe Objekt), dann ergibt der Vergleich `true`, ansonsten `false`.

Bei Strings müssen **Inhaltsvergleiche** (das ist meistens gewollt) also stets per **`equals`** oder **`compareTo`** gemacht werden!

Explizite Referenzvergleiche will man selten machen.

Getter und Setter

- **Getter** sind meist öffentliche (`public`) Methoden zum Lesen von Objekt- oder Klassenvariablen.

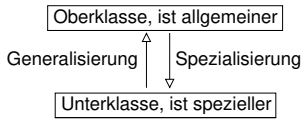
```
private String farbe = "rot";     // Objektvariable mit default-Wert
                                  // kein Zugriff von außerhalb der Klasse, da private!
public String getFarbe(){         // Getter-Methode, erlaubt lesenden Zugriff von überall
    return farbe;
}
```

- **Setter** sind oft öffentliche (public) Methoden zum Schreiben von Objekt- oder Klassenvariablen, sollten also die Parameter prüfen!

```
public void setFarbe_typisch(String f){ // erlaubt schreibenden Zugriff von überall
    if ("rot".equals(f) || "gelb".equals(f) || "grün".equals(f) ) // Inhaltsprüfung
        farbe = f; // setzt nur erlaubte Werte (z.B. bei Ampel)
}

public void setFarbe_unchecked(String f){ // ungesicherter Schreibzugriff von überall
    farbe = f; // sogar null wird nicht abgewehrt!
}

public Ampel setFarbe_profi(String f){  // Rückgabe des Objektes
    if ("rot".equals(f) || "gelb".equals(f) || "grün".equals(f) )
        farbe = f;
    return this; // Nutzung z.B. Ampel a = new Ampel().setFarbe_profi("gelb");
}

public boolean setFarbe_response(String f){ // Rückgabe von true, wenn Farbe ok
    if ("rot".equals(f) || "gelb".equals(f) || "grün".equals(f) ){
        farbe = f;
        return true;
    }
    return false; // Nutzung z.B. if (!a.setFarbe("gelblich")) return;
}
```

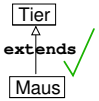

Vererbung via extends und implements

- Member-Methoden und -Variablen werden von der **Oberklasse** an die **Unterklasse** (alias “abgeleitete Klasse”) vererbt
- **Spezialisierung**: Unterklassen sind spezieller als Oberklassen (neue Variablen oder Methoden kommen hinzu)
- **Generalisierung**: Code-einsparung möglich, wenn ähnliche Klassen zu Unterklassen einer gemeinsammen (generelleren) Oberklasse gemacht werden (gleicher Code und gleiche Variablen wird in die Oberklasse verlagert und von dort vererbt)

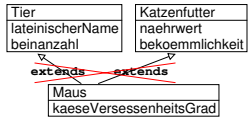


- Schlüsselwort: **`extends`** steht zwischen Klassenname und Oberklassenname; Beispiel (Maus ist ein spezielles Tier):

```
public class Maus extends Tier {
```



- **Beachte**: in Java keine Mehrfachvererbung, d.h. eine Klasse kann nicht Variablen und/oder Methoden von mehr als einer Klasse gleichzeitig erben (in C++ geht das z.B.)!



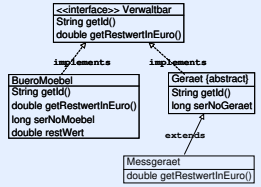
- ABER: **Interfaces** können beliebig viele geerbt werden per Schlüsselwort **`implements`**, zum Beispiel:

```
public class String implements Serializable, Comparable {
```


Beispiel zu Interfaces

Interfaces legen fest, welche **Methoden** Klassen mit diesem Interface zu implementieren haben. Tun die Klassen es nicht, dann lassen sich keine Objekte dieser Klassen erzeugen. **Bsp:** Angenommen eine Verwaltungssoftware basiert darauf, beliebige Objekte mit der `Verwaltbar`-Schnittstelle verwalten zu können.

```
public interface Verwaltbar { // legt 2 zu definierende Methoden fest
    public String getId();
    public double getRestwertInEuro();
}

public class BueroMoebel implements Verwaltbar { // implementiert beide Methoden
    long serNoMoebel;             // hier würden noch viele andere
    double restWert;              // Moebel-Objektvariablen+Methoden definiert werden
    public String getId(){
        return "M"+serNoMoebel;
    }
    public double getRestwertInEuro(){
        return restWert;
    }
}

public abstract class Geraet implements Verwaltbar {  // implementiert nur eine Methode der Verwaltbar-Schnittstelle
    long serNoGeraet;                                  // etwas nicht vollständig implementiertes ist "abstract"
    public String getId(){          // hier würden noch viele andere Geraet-Objektvariablen+Methoden definiert werden
        return "G"+serNoGeraet;
    }
}
```



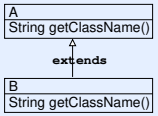
Von `BueroMoebel` lassen sich Objekte erzeugen, von `Geraet` jedoch nicht, da für `Geraet` `getRestwertInEuro` nicht implementiert ist. Deshalb ist diese Klasse explizit mit dem Schlüsselwort `abstract` gekennzeichnet. Die speziellere Klasse `Messgeraet` behebt das Problem, so dass `Messgeraet`-Objekte erzeugbar sind.

```
public class Messgeraet extends Geraet {    // hier ist die Verwaltbar-Schnittstelle vollständig implementiert
    public double getRestwertInEuro(){  // hier würden noch viele andere Messgeraet-Objektvar.+Methoden definiert werden
        return SpezialMessgeraeteDatenbank.getWertInEuro(serNoGeraet);
    }
}
```

Beispiel zu Klassen: Überladen vs. Überschreiben

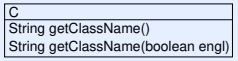
Überschreiben (Overriding) ist das Implementieren eines *neuen Methodeninhalts* in einer **Unterklasse**. Die Signatur der Methode (Name, Parameterliste, Rückgabetyp) bleibt dabei gleich.

```
public class A {
    public String getClassName(){
        return "A";
    }
}
public class B extends A {
    public String getClassName(){
        return "B und nicht A";
    }
}
```



Überladen (Overloading) ist das mehrfache Auftreten desselben Methodennamens **in einer Klasse**. Die Parameterlisten dieser Methoden müssen sich unterscheiden!

```
public class C {
    public String getClassName(){
        return "C";
    }
    public String getClassName(boolean eng){
        if (eng)
            return "I am an object of class C.";
        return "Ich bin ein Objekt der Klasse C";
    }
}
```



Teil III

Anwendung von OOP, Algorithmen und dynamische Datenstrukturen

Heutige Themen

- Suche in Feldern
  - sequentiell
  - binär
- Aufwand von Algorithmen
  - Komplexitätsklassen
  - O-Notation
- Listen
  - grundsätzlicher Aufbau
  - objektorientierte Umsetzung
  - Eigenschaften/Aufwand

Sequentielle (bzw. lineare) Suche in einem **Feld** (1)

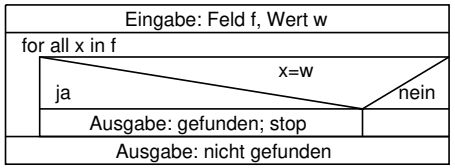
- Aufgabe: Überprüfe, ob ein zu suchender Wert im Feld enthalten ist. Teste dabei ein Element nach dem anderen.

  Beispiel: | 2 | 7 | 5 | 8 | 1 | 4 | 1 | 9 | 3 | „Suche die 4!“
- typische Lösung: `for`-Schleife über alle Feldelemente, in der jeweils verglichen wird.
- Bei linearer Suche ist es grundsätzlich egal, ob der Feldinhalt sortiert oder unsortiert ist.
- Fallunterscheidungen zur Aufwandsabschätzung (bei n Elementen):
  - **bester Fall** (best case, bc): 1 Suchschritt (sofort gefunden)
  - **schlechtester Fall** (worst case, wc): n Suchschritte (erst am Ende gefunden oder nie gefunden)
  - **im Mittel** (average case, ac): ca. n/2 Suchschritte (wenn Element enthalten ist) oder n (wenn es nicht enthalten ist)

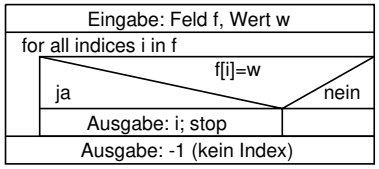
Sequentielle (bzw. lineare) Suche in einem **Feld** (2)

Ausgabe-/Rückgabevarianten: Ob vs. Wo

- **Ob:** reiner Test; typischer Methodenname: `contains`



- **Wo:** Positionssuche; typischer Methodenname: `indexOf`



Beachte: Obiger Algorithmus gibt bei **Mehrfachauftreten** des gesuchten Wertes stets den 1. passenden Index aus, wenn i von 0 schrittweise erhöht wird.

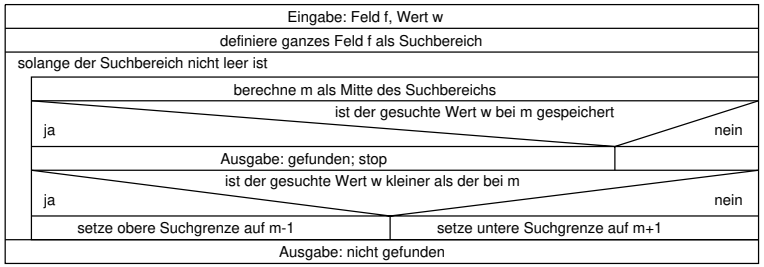
Frage: Wie kann stets der letzte passende Index zurückgegeben werden?

Binäre Suche in einem **Feld**: allgemein

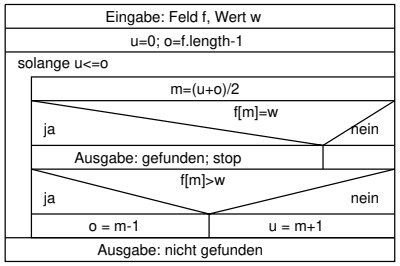
- Aufgabe: Überprüfe, ob ein zu suchender Wert im Feld enthalten ist. Wende pro Suchschritt „teile und herrsche“ an.
- Für binäre Suche muss der Feldinhalt sortiert sein!
- Beispiel: effizientes Raten einer Zahl aus einem Bereich
- Fallunterscheidungen zur Aufwandsabschätzung (bei n Elementen):
  - **bester Fall** (best case, bc): 1 Suchschritt (sofort gefunden)
  - **schlechtester Fall** (worst case, wc): $\approx \log_2(n)$ Suchschritte (erst am Ende gefunden oder nie gefunden)
  - **im Mittel** (average case, ac): zwischen $< \log_2(n)$ Suchschritte (wenn Element enthalten ist) bzw. $\approx \log_2(n)$ (wenn es nicht enthalten ist)

Binäre Suche in einem **Feld**: abstrakt

Struktogramme zum Enthaltenseinstest; typ. Methodenname: `contains`



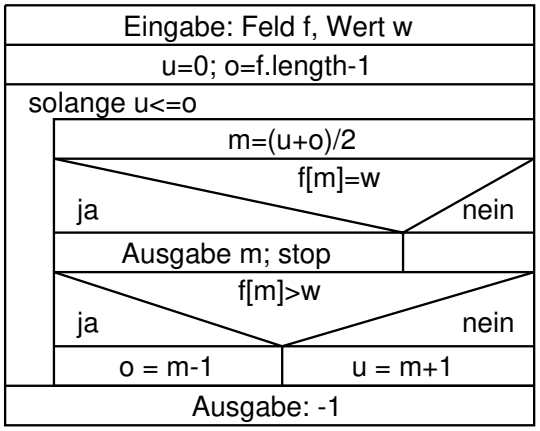
„prosaisch“



„implementierungsfreundlich“

Binäre Suche in einem **Feld**: konkreter

Struktogramm zur Positionssuche; typ. Methodenname: `indexOf`



Dieser Algorithmus liefert in höchstens ca. $\log_2(n)$ Schritten **einen** Index, an dem der Wert w im Feld f steht, oder -1, wenn der Wert nicht im Feld enthalten ist.

Suche: Zusammenfassung

- verschiedene Umsetzungsvarianten für Suche im **Feld**, z.B.:
  - sequenziell vs. binär: wesentlich für Geschwindigkeit (binär aber nur wenn **sortiert!**)
  - Test auf Enthaltensein vs. Positions-/Indexfindung
  - Typ der gesuchten Daten (z.B. int vs. String vs. irgendwelche Objekte) → müssen ordenbar sein (z.B. numerisch, lexikographisch), um effizient (binär) suchen zu können (OOP: `Comparable`-Interface)
- Suchaufwand für sequenzielle und binäre Suche ist datenabhängig und teilweise sehr unterschiedlich

Aufwandsabschätzung

- Speicherplatz oder Rechenzeit; hier: Rechenzeit
- ungefähre Schätzung
- bezieht sich typischerweise auf (hinreichend) große Datenmengen (asymptotische Analyse)
- Bildung von sog. Komplexitätsklassen (O-Notation) durch Rechenregeln:
  - konstante Faktoren fallen weg, z.B. $17 \cdot n \mathrel{\hat{=}} O(n)$
  - dementsprechend ist bei Logarithmen die Basis irrelevant, z.B. $\log_2(n) \approx 3 \cdot \log_{10}(n) \rightarrow O(\log_2(n)) = O(\log_{10}(n)) = O(\log(n))$
  - bei Polynomen gilt nur der Term mit dem größten Exponenten $(0.01 \cdot n^3 + 1000 \cdot n^2 \mathrel{\hat{=}} O(n^3))$
- Abschätzung ungeeignet bei garantiert kleinen Datenmengen

Typische Komplexitätsklassen (sortiert nach Aufwand)

$O(1)$	konstanter Aufwand	„beste“ Klasse
$O(\log n)$	logarithmischer Aufwand	oft nur etwas schlechter als konstanter Aufwand
$O(n)$	linearer Aufwand	schon problematisch für sehr große Datenmengen
$O(n \cdot \log n)$	$n \log n$-Aufwand	etwas schlechter als linearer Aufwand
$O(n^2)$	quadrat. Aufwand	unproblematisch bei kleinen Datenmegen
$O(n^3)$	kubischer Aufwand	
$O(n^k)$	allg. polynomialer Aufwand	mit $k > 3$ schlechter als kubisch
$O(2^n)$	exponentieller Aufwand	typisch für „binär-kombinatorische“ Probleme *)
$O(n!)$	exponentieller Aufwand	typisch für allgemeine kombinatorische Probleme *)
$O(n^n)$	exponentieller Aufwand	ebenso möglich *)

*) schon bei kleinen Problemen oft nicht vollständig lösbar („exponentielle Explosion“)

„Gedankenstütze“ zum Vergleich exponentieller Aufwände:
$2^n = 2 * 2 * 2 * 2 \ldots * 2$; $n! = 1 * 2 * 3 * 4 \ldots * n$; $n^n = n * n * n * n \ldots * n$

O(n) vs. O(log n) für große n

Beispiel: Anzahl der etwa benötigten Suchschritte (wc) bei **binärer** Suche:

n	n	Suchschritte: $log_2(n)$
1T	1_000	10
1Mio	1_000_000	20
1Mrd	1_000_000_000	30
1Bil	1_000_000_000_000	40

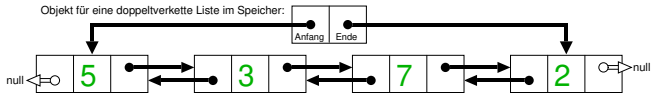
Listen als dynamische Datenstrukturen

- Listen speichern einzelne Listeneinträge (Listenelemente) in einer bestimmten Reihenfolge (ähnlich Arrays, aber **schnell änderbar!**).
- Listen gehören zu den dynamischen Datenstrukturen, weil sie mit der zu speichernden Anzahl von Daten „wachsen“.

- Es gibt **einfachverkette** Listen:

(einfach am Anfang einzufügen, „schwer“ in der Mitte und am Ende)

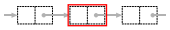
und **doppeltverkettete** Listen:



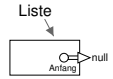
(einfach am Anfang und am Ende einzufügen, „schwer“ in der Mitte)

Umsetzung (allg. Beispiel)

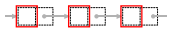
- ein **Knotenobjekt** pro Listeneintrag (bzw. Listenelement)



- Knotenobjekte werden bei Bedarf **dynamisch** angehängt, eingefügt, gelöscht...
- Jedes Listenobjekt speichert die Referenz auf den 1. Knoten (Anfang, Kopf, Head) direkt (zu beginn `null`, da Liste leer ist).



- pro Knoten wird gespeichert:
  - Wert („Nutzlast“)

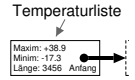


bzw.

  - Verweis auf den nächsten Knoten (bzw. `null`, falls es keinen nächsten gibt) **einfach verkettete Liste**

  - optional auch Verweis auf vorherigen Knoten (bzw. `null`, falls es keinen vorherigen gibt) **doppelt verkettete Liste**

- Kandidaten für **nicht** dynamisch gespeicherte bzw. ermittelte Listendaten (im Listenobjekt „gecacht“, **müssen aktuell gehalten werden!**):



  - Länge der Liste (Anzahl der Knoten, sonst durchzählen mit $O(n)$)
  - minimaler und maximaler Wert, sonst Bestimmung mit $O(n)$)

Vergleich: Arrays vs. Listen

	Array	**Liste**
Anwendungsbeispiel	double[] arr;	ArrayList<**Double**> lis;
statisch vs. dynamisch	statisch (feste Länge)	dynamisch (beliebig anpassbar)
direkter Zugriff auf ein Element	per Index in O(1) ⊕	im worst case Durchhangeln mit $O(n)$ ⊖
Größenänderung möglich	nein ⊖	ja ⊕
Aufwand zur Speicherverwaltung und Datenstrukturierung	klein (ein zusammenhängender Speicherbereich)	größer (u.U. sehr viele einzelne Bereiche bzw. Knotenobjekte)
Aufwand bei (ggfs. „simulierter“) Größenänderung	O(n), Kopieren der bisherigen Elemente! ⊖	O(1) ⊕
Aufwand der Einfügeoperation	O(n), Verschieben von bis zu n Elementen! ⊖	O(1) wenn Vorgänger bekannt ist ⊕

OOP: Innere Klassen (Klassendefinitionen in Klassen)

- Klassen können Klassen enthalten (Elementklassen)

  Warum hier sinnvoll? Eine Liste ohne Knoten kann Sinn machen (leere Liste), ein Knoten aber nicht ohne **seine** Liste.

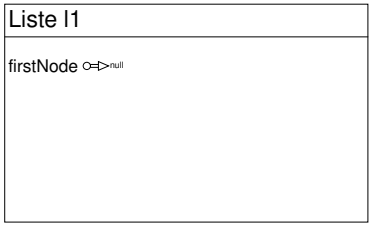
- Innere Klassen werden in der selben java-Datei wie die Top-Level-Klasse (äußer(st)e Klasse) gespeichert.
- **Beispiel:** Listenklasse mit innerer Knotenklasse:

```
public class Liste {

    public class Node { // innere Klasse
        Node nextNode;  // Objektvariable in Node
        int value;      // Objektvariable in Node
        public Node(Node n, int v){
            nextNode = n;
            value = v;
        } // ... und weitere Methoden/Konstruktoren des Knotens
    }

    Node firstNode; // Objektvariable in Liste

    public void addValue(int x){
        firstNode = new Node(firstNode, x);
    } // ... und weitere Methoden/Konstruktoren/Variablen der Liste
}
```

Beispiel zur Erzeugung einer Liste I

- Erzeugen einer Liste und Speichern **(des Zeigers darauf)** in `l1` könnte in `main` so aussehen:

```
Liste l1 = new Liste();
```



Zunächst existiert nur die Liste. Sie enthält keine Knoten. Das einzige, was sie enthält, ist die Referenz auf den 1. Knoten (und da der nicht existiert hat `firstNode` den Wert `null`).

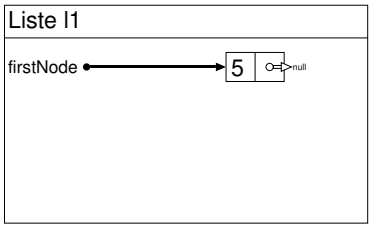
Beispiel zur Erzeugung einer Liste II

- Anhängen eines Knotens mit dem Wert 5

```
l1.addValue(5);
```

in `addValue` wird ausgeführt:

```
firstNode = new Node(null, 5);
```



Zunächst wird das neue Objekt erzeugt und dann die Referenz darauf in `firstNode` gespeichert. Nun zeigt `firstNode` also auf den gerade erzeugten Knoten.

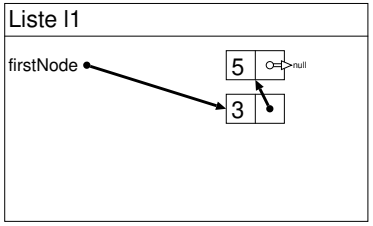
Beispiel zur Erzeugung einer Liste III

- Anhängen eines Knotens mit dem Wert 3

```
l1.addValue(3);
```

in `addValue` wird ausgeführt:

```
firstNode = new Node(<Zeiger auf Knoten mit 5>, 3);
```



Zur Speicherung der 3 wird ein neuer Knoten erzeugt und die Referenz darauf wieder in `firstNode` gespeichert. Der neue Knoten wird also immer **vorn** angehängt. Der „alte“ Zeiger auf den 1. Knoten wurde aber vorher im Konstruktor von `Node` benutzt, um die Knoten zu verketten.

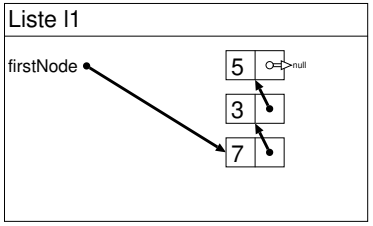
Beispiel zur Erzeugung einer Liste IV

- Anhängen eines Knotens mit dem Wert 7

```
l1.addValue(7);
```

in `addValue` wird ausgeführt:

```
firstNode = new Node(<Zeiger auf Knoten mit 3>, 7);
```



Wieder werden die zu speichernde Zahl und der bisherige Zeiger auf den ersten Knoten im Konstruktor von `Node` verwendet, um einen neuen Knoten zu erstellen. `firstNode` speichert schließlich die Referenz auf diesen neuen Knoten.

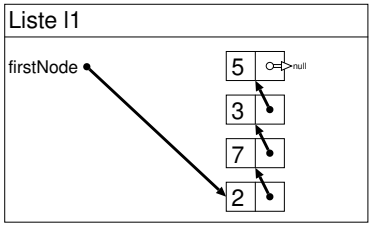
Beispiel zur Erzeugung einer Liste V

- Anhängen eines Knotens mit dem Wert 2

```
l1.addValue(2);
```

in `addValue` wird ausgeführt:

```
firstNode = new Node(<Zeiger auf Knoten mit 7>, 2);
```



Nach der Erstellung des neuen ersten Knotens zeigt `firstNode` auf ihn. `firstNode` kann also dazu benutzt werden, die Liste vom Kopf her zu durchlaufen.

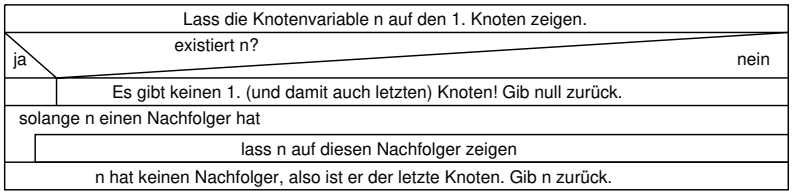
Beispiel zum Durchlaufen einer Liste I

- Ermitteln des letzten Knotens einer Liste, z.B. Aufruf in `main` für eine existierende Liste `l1`:

```
Node last = l1.getLastNode();
```

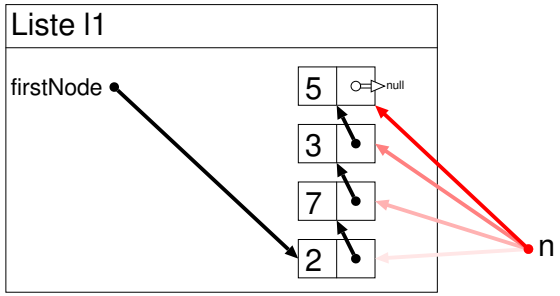
wobei `getLastNode()` in der Klasse `Liste` wiefolgt definiert ist:

```
public Node getLastNode(){
    Node n = firstNode;      // firstNode NICHT ändern!!!
    if (n==null)
        return n;            // erster (und letzter) sind null
    while(n.nextNode!=null) // solanger letzter nicht erreicht
        n = n.nextNode;      // setze n einen Knoten weiter
    return n;                // n ist jetzt der letzte Knoten
}
```



Beispiel zum Durchlaufen einer Liste II

- `n` zeigt also beim Durchlaufen immer auf den jeweils aktuellen Knoten



Anwendungen des „Listedurchlaufen“-Programmiermusters für z.B.:

- Länge der Liste zurückgeben
- Summe der Werte einer Liste zurückgeben
- letzten Wert oder `null` zurückgeben
  (gute Idee für „Faule;-)“: `getLastNode()` verwenden!)

Listenoperationen und entsprechender Aufwand

Aktion	Aufwand	Erläuterung
Erstellung	O(1)	nur nicht dynamische Anteile
Anhängen eines neuen Elementes (ohne Verweis auf den Endknoten)	O(n)	„Durchhangeln“ bis zum Ende
Anhängen eines neuen Elementes (mit Verweis auf den Endknoten)	O(1)	neuer Knoten direkt in bisherigen Endknoten einhängbar
Länge feststellen (ohne direkte Speicherung der Länge)	O(n)	schrittweises Durchzählen
Länge feststellen (mit direkter Speicherung der Länge)	O(1)	eine Leseoperation (aber: pro Anhängen / Einfügen / Löschen muss die Länge aktualisiert werden)
an k-ter Stelle einfügen	O(k)	Durchhangeln zum k-ten Knoten
bei sortierter Liste an passender Stelle einfügen	O(n)	Durchhangeln bis Stelle gefunden

Verständnisfragen zu Listen

Frage	Antwort
Kann man in einer Liste sequentiell Suchen?	ja, Durchhangeln von Knoten zu Knoten und jeweils vergleichen - typischer Fall
Kann man in einer Liste binär Suchen?	ja, aber mit O(n) Aufwand, (kein direkter Zugriff auf Median wie beim Array möglich)!
Wenn eine einfachverkette Liste ihre Daten in k-Knoten speichert, wie viele Knoten braucht dann eine doppeltverkette Liste zur Speicherung derselben Daten?	k, Verkettung ist hier egal

Heutige Themen

- Begriffe:
  - Dynamische Datenstrukturen
  - Parametrisierte Datentypen
- Binärbäume:
  - Aufbau, Begriffe
  - Ausgeglichenheit
  - Operationen / Aufwand
  - Traversierung
- Beispiele für vordefinierte „Container“-Klassen:
  - ArrayList (Liste mit Zugriff via Index in $O(1)$)
  - TreeMap (Binärbaum mit Suchen in $O(log_n)$)
  - TreeSet (Menge mit Suchen in $O(log_n)$)

Eigenschaften von Datenstrukturen

- statisch vs. dynamisch

  Statische Datenstrukturen haben eine fixe Größe, dynamische passen sich (bzw. den von ihnen benutzten Speicher) hingegen dem jeweiligen Bedarf dynamisch an (weiteren hinzufügen, ungenutzten freigeben).

  Beispiel: Array = statisch vs. Liste = dynamisch
  Frage: In welcher Hinsicht könnten auch Arrays dynamisch genutzt werden?
  - bei Bedarf neu anlegen und alten Inhalt umkopieren oder
  - größer anlegen als anfangs gebraucht und „Füllstandsvariable“ verwenden oder
  - 1d-Array von 1d-Arrays (innere Arrays zu Beginn meist noch `null`) oder ...

- fest definierter Typ vs. (typ-)parametisiert

  Es ist ein Vorteil, wenn der Typ der zu speichernden / zu bearbeitenden Daten wählbar ist (flexibler einsetzbare Datenstruktur, z.B. „Container“-Klassen in Java: Angabe des Typs in <spitzen Klammern>).

  Liste von Strings dann genauso möglich wie Liste von Double-Werten oder Liste von Autos

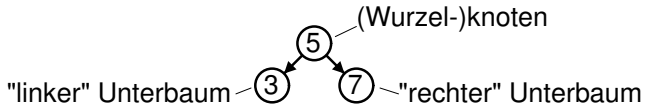
Wozu Bäume?

- Auch Bäume speichern Daten - in bestimmten Fällen aber wesentlich effizienter als Arrays oder Listen.
- Kombination der besten Eigenschaften von Arrays und Listen
- vereinen Dynamik der Datenstruktur einer Liste und Möglichkeit der binären Suche mit $O(log_n)$ wie in einem Array
  $\rightarrow$ **hier: stets Binärbäume**

  alternativ z.B. in Computergrafik: Quadtree, Octree
- Einfügen und Löschen (d.h. dynamsiche Anpassbarkeit) wie bei Liste mit $O(1)$ möglich (wenn die Position gegeben ist, z.B. via Referenzvariable), bzw. mit $O(log_n)$, wenn vorher die passende Position gesucht werden muss

Aufbau von Binärbäumen, Begriffe (I)

- Wurzelknoten, ist am Beginn (leerer Baum) `null`
- Hierarchie, jeder Knoten hat bis zu 2 Nachfolgeknoten (Unterbäume), welche bei Bedarf dynamisch erzeugt werden

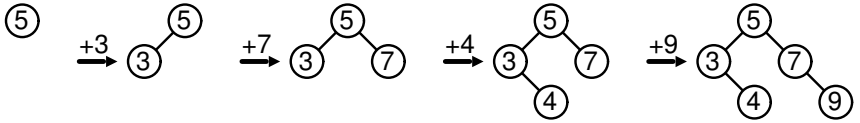


- Soll in einem Baum ein neuer Wert gespeichert werden, dann wird der Baum **von der Wurzel aus** durchlaufen, bis ein passender Platz im Baum gefunden ist.
- **Einsortierregel: "kleinere nach links"**

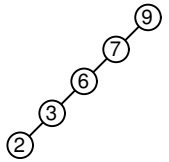
  Ist im Knoten k ein Wert x gespeichert und soll in k (bzw. seinen Unterbäumen) ein Wert y eingefügt werden und $y < x$, dann wird y im linken Unterbaum gespeichert, bei $y > x$ aber im rechten.

Aufbau von Binärbäumen, Begriffe (II)

- Struktur ist reihenfolgeabhängig
  Beispiel: Einfügen von 5, 3, 7, 4, 9 in dieser Reihenfolge



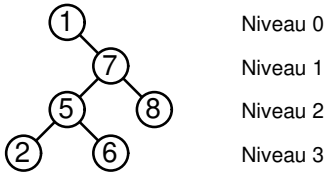
- ein Binärbaum kann (ohne Pflege) zu Liste degenerieren
  Beispiel: Einfügen von 9, 7, 6, 3, 2 etc in dieser Reihenfolge



- Schlussfolgerung: Um $O(n)$ Aufwand pro Suchen/Einfügen zu vermeiden, muss ein Binärbaum gepflegt werden!

Aufbau von Binärbäumen, Begriffe (III)

- Knoten mit der selben Entfernung zur Wurzel haben das selbe **Niveau**



- **Blätter** haben keine Kinder (2, 6, 8)
- **Höhe** des Baumes: max. Niveau+1 = 3+1 = 4
- **innere Knoten**: 5, 7 (und 1, die Wurzel)
- jeder (Kind-)Knoten ist per **Kante** mit seinem Elternknoten verbunden
  → Wie hängen Knotenanzahl x und Kantenanzahl y zusammen?
  $$x = y + 1$$
- **Pfad**: Reihe der Knoten und Kanten von der Wurzel zu einem bestimmten Knoten; Weg auf dem der Wert gefunden wird

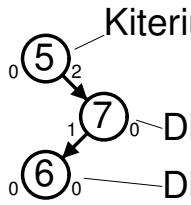
Ausgeglichenheit (I)

- Pflege durch Sichern der Ausgeglichenheit = „Balanciertheit“
- sinnvolle Regel für Ausgeglichenheit (**AVL-Bäume**):

  Für jeden Knoten unterscheiden sich die Höhen des linken und des rechten Teilbaums um höchstens 1.

- zur Einhaltung der Regel sind ggfs. Ausgleichsoperationen nach Einfügen und Löschen nötig
- Aufwand: jeweils nur **lokaler Umbau**, d.h. der Umbau „kostet“ nur $O(1)$
- Ergebnis: statt $O(n)$ pro Suche oder Einfügen kann der Knoten stets in $O(log_n)$ gefunden werden, d.h. der **Pflegeaufwand lohnt sich!**
- strengeres Kriterium: „Niveau der Knoten mit einem oder keinem Unterbaum unterscheidet sich um höchstens 1“ → *vollständig balancierter Binärbaum*
- *vollständiger Binärbaum*: alle Blätter im selben (voll besetzten!) Niveau
  → dieser Baum hat stets $2^{Höhe} - 1$ Knoten sowie $2^{Höhe-1}$ Blätter

Ausgeglichenheit (II)



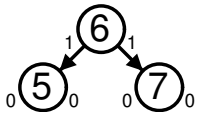
Kiterium verletzt, da Differenz größer als 1 ist

Differenz ist 1, d.h. Kriterium ist erfüllt

Differenz ist 0, d.h. Kriterium ist erfüllt

- Durch eine Rotation (lokaler Umbau des Baumes, Aufwand $O(1)$) kann das Kriterium wieder erfüllt werden.
- Der Median (also der nach Sortierung mittelste Wert) der drei Knoten sollte zur lokalen Wurzel werden (hier 6), die anderen entsprechend Einfügeregel zum linken und rechten Kind.

Ergebnis des Umbaus:

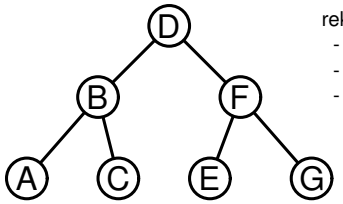


Operationen auf Binärbäumen

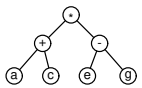
Operation/Aktion	Aufwand	Erläuterung
Suche eines Elements (im ausgeglichenen Baum mit n Elementen)	$O(log_n)$	Höhe des ausgeglichenen Binärbaums ist log_n
Suche eines Elements (unausgeglichener Baum mit n Elementen)	$O(n)$	Höhe des unausgeglichenen Binärbaums ist im worst case n
Einfügen eines Elements (im ausgeglichenen Baum mit n Elementen)	$O(log_n)$	Höhe des ausgeglichenen Binärbaums ist log_n
Einfügen eines Elements (unausgeglichener Baum mit n Elementen)	$O(n)$	Höhe des unausgeglichenen Binärbaums ist im worst case n
Aufbau aus n Elementen (mit Ausgleich)	$O(nlog_n)$	n mal mit Aufwand $O(log_n)$
Aufbau aus n Elementen (ohne Ausgleich)	$O(n^2)$	n mal mit Aufwand $O(n)$

Traversierung (systematisches Durchlaufen)

Die **Traversierung** eines Baumes beschreibt das systematische Aufsuchen (Abarbeiten, Durchlaufen,...) der Knoten eines Baumes.
Je nach Traversierungsart ergeben sich unterschiedliche Reihenfolgen.



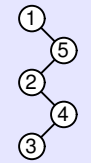
rekursive Abarbeitungsreihenfolge pro Knoten
- Inorder: links, Wert, rechts
- Preorder: Wert, links, rechts
- Postorder: links, rechts, Wert



Art	Besuchsreihenfolge	Anwendungsbeispiel
Inorder	A, B, C, D, E, F, G	z.B. sortierte Ausgabe
Preorder	D, B, A, C, F, E, G	s.o.: `mul(add(a, c), sub(e, g))`
Postorder	A, C, B, E, G, F, D	s.o.: `a c add e g sub mul` (z.B. Postscript)
Levelorder	D, B, F, A, C, E, G	niveauweise, d.h. „Breite zuerst“

Verständnisfragen zu Bäumen

Frage	Antwort
Kann man in Binärbäumen sequentiell Suchen?	ja, siehe Traversierung; z.B. inorder
Kann man in Binärbäumen binär Suchen?	ja, dafür sind sie da!
Wie viele Knoten lassen sich in einem Binärbaum der Höhe 3 maximal speichern?	7, denn die Niveaus 0 bis 2 haben max. 1+2+4 Knoten bzw. $2^3 - 1 = 7$
Ist ein Binärbaum eine dynamische Datenstruktur?	ja, seine Knoten werden nur bei Bedarf erzeugt, auch Löschen ist mögl.

Angenommen folgender Code wird genutzt um einen Binärbaum `biBa` zu füllen:
`for (int i=1, s=1; n>0; i+=--n*s, s*=-1)`
`  biBa.add(i);`
Geben Sie die Reihenfolge der eingefügten Elemente an, wenn n=5 ist. Mit welchem Aufwand erfolgt der Aufbau, wenn Ausgleichsoperationen eingespart werden?
	Reihenfolge bei n=5: 1 5 2 4 3;
	Baum entartet zu einer Liste, deshalb Aufbau in $O(n^2)$
	 

Beispiele für „Container“-Klassen I: Allg. & ArrayList

Speicherung von Objekten einer beliebigen Klasse, Parameter: Klassenname in spitzen Klammern → typparametrisierte Datenstrukturen oder auch **„abstrakte Datentypen“**

- sind u.a. in Java-Klassenbibliothek enthalten
- stellen typische Container wie Menge, Liste, Baum bereit
- lassen sich beliebig kombinieren, z.B.: für ein „Mapping“ von Name auf Liste der Telefonnummern: `TreeMap<String, ArrayList<Long>>`

java.util.ArrayList

- siehe `https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html`
- Liste: mit $O(1)$ für Zugriff per Index, Länge erfragen und „nahezu“ $O(1)$ für das Anhängen, $O(n)$ für Suchen

```
ArrayList<Double> messwerte = new ArrayList<Double>();
messwerte.add(1.7); // Anfügen ans Ende
double mw0 = messwerte.get(0); // 1. Wert via Index lesen
int idxOfX = messwerte.indexOf(x); // 1. Auftreten von x
```


Beispiele für „Container“-Klassen II: TreeMap

java.util.TreeMap

- siehe `https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html`
- **Map**: Abspeicherung von „Schlüssel→Wert“-Paaren (bzw. key → value), jeweils z.B. hinzufügen, löschen, erfragen
- `TreeMap` ist die binärbaumartige Umsetzung einer `Map`
- Bsp.: Ein Container zum „Mappen“ von Geräte-ID auf die zugehörige Messwertliste wäre `TreeMap<Long, ArrayList<Double>>`. ID ist der Schlüssel (*key*) zur Suche der Position im Baum. Im Knoten ist aber auch die Messwerteliste als zugehöriger Wert (*value*) hinterlegt.
- ausgeglichener Baum: deshalb $O(log_n)$ für die typischen Operationen, welche auf Schlüsselsuche basieren (einfügen, löschen, suchen), und $O(n)$ für wertbasierte Suche wie `containsValue` (hier ist sequentielle Suche nötig!)

```
TreeMap<String, Integer> namensStatistik = new TreeMap<String, Integer>();
Integer numHugo = namensStatistik.get("Hugo"); // Anzahl der Hugos
numHugo = numHugo==null ? 1 : (numHugo+1); // Anzahl um 1 erhöhen
namensStatistik.put("Hugo", numHugo); // neue Zahl in Map speichern
int numNames = namensStatistik.size(); // Anzahl der Namen erfragen
```


Beispiele für „Container“-Klassen III: TreeSet

java.util.TreeSet

- Doku:siehe `https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html`
- Set: **Menge** von Objekten einer beliebigen Klasse (Reihenfolge egal) : hinzufügen, löschen, erfragen
- `TreeSet` ist binärbaumartige Umsetzung eines `Set`
- Bsp.: Ein Container zur Speicherung einer Menge von ganzzahligen Schlüsseln (z.B. int-IDs) wäre `TreeSet<Integer>` - beachte: `Integer` ist die Wrapperklasse ist für `int`

```
TreeSet<Integer> schonBearbeitet = new TreeSet<Integer>();
```

```
if(schonBearbeitet.contains(z))
   return; // wir wollen mit der Zahl z nur einmal was tun
// mache was mit z
  ...
schonBearbeitet.add(z); // vermerke die Bearbeitung von z
```


Am Ende: Schöne Bescherung

Was ist ist zu tun? Dem Weihnachtsmann ist sein alter #-Schlitten kaputt gegangen und er muss die Geschenke in einen hoch modernen @-Schlitten umladen (den wollte seine Frau ihm eigenlich erst zu Ostern schenken).
Leider ist ringsum Schneematsch und er muss beim Umladen alle Pakete auf dem =-Tisch ablegen, ABER: es darf NIE ein größeres auf ein kleineres Paket gelegt werden!

```
import java.util.ArrayList;
public class HanoiPakete {
    ArrayList<String> turmA = new ArrayList<String>();
    ArrayList<String> turmB = new ArrayList<String>();
    ArrayList<String> turmC = new ArrayList<String>();
    int h = 0;
    int wait_ms = 0;
    String nix = "";
    public HanoiPakete(String[] pak, int ms, String nix){
        h = pak.length;
        wait_ms = ms;
        this.nix=nix;
        for (int i=0; i<h;i++)
            turmA.add(pak[i]);
        show();
        while(true){ // endlose Iteration zweier Rekursionen...
            hanAnim(h, turmA, turmC, turmB);
            hanAnim(h, turmB, turmC, turmA);
        }
    }
    public void show(){
        for (int i=h-1; i>=0; i--) // zeilenweise von oben beginnend
            System.out.println(     (turmA.size()>i ? turmA.get(i):nix)
                                +" "+(turmB.size()>i ? turmB.get(i):nix)
                                +" "+(turmC.size()>i ? turmC.get(i):nix));
        System.out.println("######### ======== @@@@@@@@\n");
    }
    public void hanAnim(int n, ArrayList<String> f, ArrayList<String> t, ArrayList<String> h){
        if (n==1){
            if (f.size()>0) // direkte Bewegung von f nach t
                t.add(f.remove(f.size()-1));
            try {Thread.sleep(wait_ms);}catch(Exception e){}
            show();
            return;
        }
        hanAnim(n-1, f, h, t);    // um eins kleineren Turm von f nach h
        hanAnim(1  , f, t, h);
        hanAnim(n-1, h, t, f);    // um eins kleineren Turm von h nach t
    }
    public static void main(String[] a){
        String[] p = {"Eisenbahn", " Fahrrad ", "  Puppe  ", "   Tee   ", "    *    "};
        new HanoiPakete(p, 1000, "          ");
    }
}
```

```
    *
   Tee
  Puppe
 Fahrrad
Eisenbahn
######### ======== @@@@@@@@
```

Heutige Themen

- AuP-Planung bis zur Prüfung (+x)
- Sortieralgorithmen
  - **Warum** behandlen wir Sortieralgorithmen?
  - **Wie** betrachten wir Sortieralgorithmen?
  - **Welche** Sortieralgorithmen betrachten wir?
  - **Worauf** beschränken wir uns?
  - Stabilität und Laufzeit
  - Mergesort (MS)
  - Insertionsort (IS), Selektionsort (SS), Bubblesort (BS) als interaktive Tests + Eigenschaften und Beispiele
- allgemeinere Anwendungsbeispiele zum Sortieren
  - Nutzung eines Binärbaums
  - Klausurauswertung
  - Telefonbuch etc

Planung der nächsten Wochen bzgl. AuP

... wenn es soweit ist, dann wird hier was Hilfreiches stehen ;-) ...

Warum behandeln wir Sortieralgorithmen?

- Fach “**Algorithmen** und Programmierung” soll auch zeigen, wie ähnliche Algorithmen (hier: Zweck ist das Sortieren) verglichen werden können
- Auswahlkriterien wie:
  - zu erwartende Geschwindigkeit (z.B. O-Notation)
  - Programmieraufwand
- Zusammenspiel von Algorithmen und Datenstrukturen
- Möglichkeiten des Effizienzgewinns
- **Erkenntnisse übertragbar** auf Algorithmen für andere Aufgaben (Lösung eigener praktischer Probleme im Arbeitsleben)
- Programmierbeispiele: Java, OOP, Vergleichstests...

Abstraktionsniveau / Verständnistiefe

Wie werden die Verfahren betrachtet / wie sollten sie verstanden werden?

- anschaulich (an geg. Beispiel)
- Struktogramm mit Prosa
- Struktogramm mit java-nahen Texten
- Java-Implementierung als Methode(n)
- Java-Implementierung als separate Klasse
- Vergleich von allg. Eigenschaften (z.B. Komplexität)
- Vergleich von Sortierabläufen/Zwischenergebnissen

Überblick über ausgewählte Verfahren

- Insertionsort (kann schnell sein)
- Selectionsort (das “dümmste”)
- Bubblesort (anschaulich, kann schnell sein)
- Mergesort (garantiert nicht langsam, aber relativ komplex)
- Quicksort (sehr selten langsam, aber relativ komplex)
  $\rightarrow$ wird in Vorlesung nicht näher betrachtet

Grundsätzlich Mögliches vs. unsere Annahmen

- Grundsätzlich werden Datensätze/Objekte anhand von **Schlüsselwerten** sortiert.

  Unsere Annahme/Einschränkung: Wir betrachten nur die Schlüsselwerte $\rightarrow$ wir sortieren einfach Zahlen oder Zeichenketten (bzw. alles was mit $<$, $==$, $>$ oder `compareTo()` vergleichbar ist).
- Von den möglichen Implementierungsvarianten betrachten wir meist nur eine bestimmte, allgemein gibt es weitaus mehr.
- In den Beispielen wird stets mit Arrays gearbeitet. Allgemein können aber auch andere Datenstrukturen sortiert werden.
- In den Beispielen findet das Umsortieren meist **“inplace”** statt, d.h. ein gegebenes Array wird direkt zum Sortieren genutzt (möglichst keine weiteren Arrays, die Eingabe wird dadurch aber quasi zerstört)!

Stabilität und Laufzeiten der Sortieralgorithmen

Ein Sortierverfahren ist stabil, wenn es beim Sortieren die Reihenfolge der Datensätze mit gleichem Schlüssel **nie** ändert.
Bei einem instabilen Sortierverfahren **kann** sich deren Reihenfolge ändern.

Per Experiment nur (sicher) zeigbar, dass ein Algorithmus instabil sortiert. Für allg. Aussage: Algorithmus analysieren/verstehen!
Annahme: Datensatz ist sortiert nach Vornamen und soll nun nach Namen sortiert werden; **2 Experimente**:

Meyer	Anna
Holzer	Fred
Meyer	Hans

stabil danach stets:

Holzer	Fred
Meyer	Anna
Meyer	Hans

zulässige Schlussfolgerung aus dem Experiment

Anna bleibt vor Hans, d.h. diese Sortierung **kann** stabil sein

Meyer	Anna
Holzer	Fred
Meyer	Hans

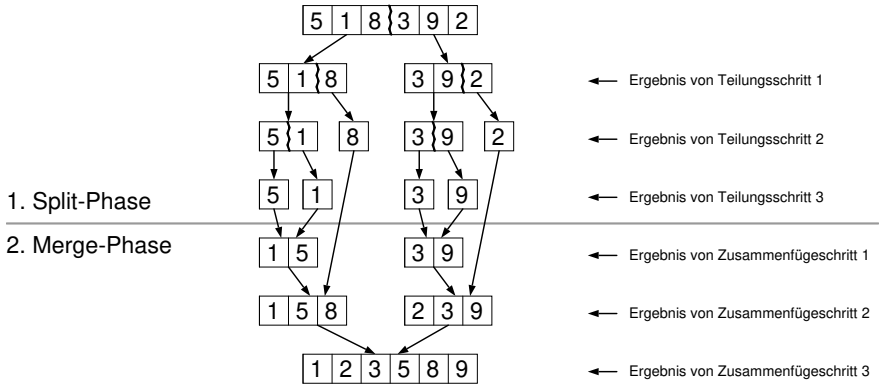
instabil danach **evtl.**:

Holzer	Fred
Meyer	Hans
Meyer	Anna

Anna nun nach Hans, d.h. diese Sortierung **ist instabil**

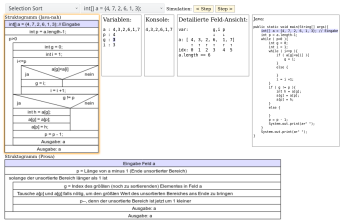
Verfahren	Stabilität	Average Case (AC)	Worst Case (WC)	Best Case (BC)
SelectionSort	instabil	$O(n^2)$	$O(n^2)$	$O(n^2)$
InsertionSort	stabil	$O(n^2)$	$O(n^2)$	$O(n)$
BubbleSort	stabil	$O(n^2)$	$O(n^2)$	$O(n)$
MergeSort	stabil	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
QuickSort	instabil	$O(n \log n)$	$O(n^2)$	$O(n \log n)$

Abarbeitungsbeispiel für Merge-Sort



- Split-Phase besteht aus $O(\log n)$ Schritten, wenn n die Anzahl der zu sortierenden Elemente ist. Grund: Teilung immer in der Mitte
- Merge-Phase besteht ebens aus $O(\log n)$ Schritten, da gleicher Ablauf wie Teilung, nur umgekehrte Reihenfolge
- pro Schritt $O(n)$ Aufwand
- insgesamt also stets $O(n \log n)$

Experimente mit verschiedenen Sortieralgorithmen



https://moodle2.tu-ilmenau.de/mod/resource/view.php?id=93804

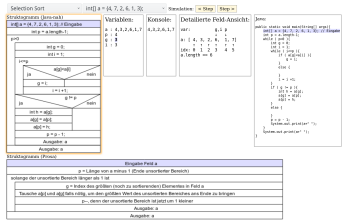
Auf obiger (auch in Moodle verlinkten) HTML-Seite lassen sich interaktive Experimente mit den Sortierverfahren machen:

- Insertion-Sort (IS),
- Selection-Sort (SS) und
- Bubble-Sort (BS).

allgemeine Experimentiertipps:

- verschiedene *Algorithmen* und *zu sortierende Felder* per Auswahlbox einstellbar,
- Basis der interaktiven Tests bilden wieder **Struktogramme** (Wiederholung ist wichtig;-),
- für Tests in Eclipse den Beispiel-Java-Source-Code evtl. anpassen

Experimente mit verschiedenen Sortieralgorithmen



https://moodle2.tu-ilmenau.de/mod/resource/view.php?id=93804

Aufgabenstellung für eigene Experimente (jeweils für IS, SS und BS):

- Wie erfolgt die Sortierung der Zahlen 5, 1, 8, 3, 9, 2?
  Protokollieren Sie die Ergebnisse der Durchläufe tabellarisch auf Papier.
- Zählen Sie dabei jeweils die Vertauschungen von Feldelementen und/oder die Lese- und Schreibzugriffe.
- Was passiert, wenn ein schon sortiertes Feld sortiert werden soll?
- Versuchen Sie zu erklären, warum sich die auf Folie 7 gelisteten Eigenschaften ergeben (Laufzeit, Stabilität).

Abarbeitungsbeispiel für Bubble-Sort

D	#V	Feld nach Durchlauf						Anmerkung
0		5	1	8	3	9	2	initialer Zustand
1	3	1	5	3	8	2	9	nach 1. Durchlauf (mit 3 Vertauschungen) ist 9 oben (fertig)
2	2	1	3	5	2	8	9	nach 2. Durchlauf (mit 2 Vertauschungen)ist auch 8 an ihrem Platz
3	1	1	3	2	5	8	9	2 ist wieder einen Platz nach unten „gebubbelt“ und 5 am Platz
4	1	1	2	3	5	8	9	eigentlich sind alle am Platz, aber finaler Testdurchlauf fehlt noch
5	0	1	2	3	5	8	9	keine Vertauschung mehr gemacht → fertig!

phänomenologische Betrachtungen und Schlussfolgerungen:

- **pro Durchlauf werden alle „Nachbarwerte“ im unsortierten Bereich von links beginnend verglichen und wenn nötig getauscht** (im 1.Durchlauf: $n-1$ Vergleiche)
- pro Durchlauf „bubbelt“ der jeweils größte Wert nach „oben“ (ans Ende des unsortierten Bereichs) → ist dann fertig → d.h. ein **fertig** sortierter Bereich baut sich von rechts auf
- je Wert und Durchlauf sind mehrere Schritte nach rechts möglich, nach links immer nur einer → falls kleinstes Feldelement zu beginn ganz rechts → n Durchläufe nötig (WC).
- pro Durchlauf $O(n)$ Schritte → n Durchläufe haben $O(n^2)$ → WC
- bei initial „richtig“ sortiertem Feld: BS braucht nur 1 Durchlauf → $O(n)$ → BC
- nach fertiger Sortierung immer noch ein Durchlauf nötig → letzte Zeile stets doppelt

Abarbeitungsbeispiel für Insertion-Sort

D	#V	Feld nach Durchlauf						Anmerkung
0		5	1	8	3	9	2	initialer Zustand, Sichtweise: „{5} ist sortiert“
1	1	1	5	8	3	9	2	1 vor die 5 eingefügt, nun ist {1, 5} sortiert
2	0	1	5	8	3	9	2	8 ist bzgl.{1, 5} schon ok → kein Verschieben → nun {1, 5, 8} ok
3	2	1	3	5	8	9	2	3 muss weiter vorn eingefügt werden (vor 5) → nun {1, 3, 5, 8} ok
4	0	1	3	5	8	9	2	9 ist bzgl. {1, 3, 5, 8} ok → kein Verschieben → nun {1, 3, 5, 8, 9} ok
5	4	1	2	3	5	8	9	2 muss vor 3 eingefügt werden, dazu 4 Verschiebungen, alles fertig

phänomenologische Betrachtungen und Schlussfolgerungen:

- zu Beginn gilt der 1. Wert als sortierter Bereich (wie jeder einelementige Bereich)
- der sortierte (nicht zwangsläufig fertig sortierte!) Bereich liegt immer links und wächst pro Durchlauf um jeweils 1 Element
- **pro Durchlauf wird das Element, das nach dem sortierten Bereich liegt, an die richtige Position im sortierten Bereich eingefügt**
- pro Duchlauf 1 bis n Schritte, es sind stets n Schritte nötig (es sind stets n Elemente zu prüfen) → $O(n)$ im BC bis $O(n^2)$ im WC möglich

Abarbeitungsbeispiel für Selection-Sort

D	#V	Feld nach Durchlauf						Anmerkung
0		5	1	8	3	9	2	initialer Zustand, noch zu sortieren: X=0...n-1 (bzgl. Indices)
1	1	5	1	8	3	2	9	hat 9 als Maximum in X gewählt und ans Ende getauscht, X=0...n-2
2	1	5	1	2	3	8	9	hat 8 als Maximum in X gewählt und ans Ende getauscht, X=0...n-3
3	1	3	1	2	5	8	9	hat 5 als Maximum in X gewählt und ans Ende getauscht, X=0...n-4
4	1	2	1	3	5	8	9	hat 3 als Maximum in X gewählt und ans Ende getauscht, X=0...n-5
5	1	1	2	3	5	8	9	hat 2 als Maximum in X gewählt... → X=0...n-6 → X=0...0 → fertig

phänomenologische Betrachtungen und Schlussfolgerungen:

- **selektiert jeweils den größten Wert des noch zu sortierenden Bereiches und tauscht ihn ans Ende dieses Bereichs**
- startet mit dem gesamten Feld als unsortiertem Bereich
- pro Durchlauf wächst der **fertig** sortierte Bereich am Feldende jeweils um 1
- noch zu sortierender Bereich jeweils um 1 kleiner → fertig bei Größe=1
- $n-1$ mal das Maximum suchen → $n-1$ Durchläufe mit je $O(n)$ Vergleichen
- Gesamtaufwand also stets quadratisch! → $O(n^2)$ egal ob WC oder BC!

Binärbaum oder Liste zum Sortieren nutzen

- da eine **Inorder-Traversierung eines Binärbaums** eine sortierte Reihenfolge ergibt, kann auch dadurch sortiert werden, dass:
  1. ein Binärbaum aufgebaut wird → durch Einfügen aller zu sortierender Elemente (nun kann schon binär gesucht werden)
  2. bei Bedarf: Inorder-Traversierung zur Erzeugung einer sortierten Ausgabe oder sortierten Datenstruktur, z.B. Speicherung in sortiertem Feld oder sortierter Liste
- auch Nutzung von **Listen zum Sortieren** möglich, z.B.:
  - Einfügen an richtiger Position in sortierte Liste in $O(n)$ → `insValueSorted`, arbeitet ähnlich wie ein Durchlauf bei Insertionsort
  - rekursives Zusammenfügen von je zwei sortierten Listen in $O(n)$ → `concatSorted`, arbeitet ähnlich der Mergephase bei Mergesort

siehe nächste Folie

Klausursortierung nach dem Einsammeln

Der „real gelebte Sortieralgorithmus“ (einmal pro Semester):

- nach dem Einsammeln: unsortierter großer Stapel, z.B. 300 Klausuren
- jeweils die oberste Klausur nehmen und in den Handstapel einsortieren
- wenn der Handstapel zu groß wird (ca 15 Klausuren) neuen Handstapel anfangen
- jeweils 2 ca gleich große Handstapel wie bei Mergesort zusammenfügen

```
15  15     15  15     15  15     15  15  ... (ca 20 Stapel)
  30         30         30         30     ... (ca 10 Stapel)
       60                    60            ... (ca  5 Stapel)
                  ...

                 300
```


Telefonbuch aufbauen und später pflegen

Annahmen:

- aus n Telefonnummern (alternativ Dokumenten-IDs, Kfs-Nummern etc) soll ein sortiertes Verzeichnis aufgebaut werden, z.B. $n = 30$ *Mio*
- Speicherung in Array → ok, da 30Mio viel kleiner als 2Mrd ist (max. Arraygröße)

typische Wahl von Algorithmen für Arbeitschritte

- Wahl von Mergesort für initiale Sortierung (garantiert $O(nlog_n)$, denn worst case bei $O(n^2)$-Algorithmen wäre für $n = 30$ *Mio* bereits sehr lang!
- beim Pflegen:
  - Einfügen einzelner Einträge per Insertionsort mit $O(n)$
  - Einfügen einer Menge sortierter Einträge per Mergesort-Phase2 (zusammenfügen) mit $O(n)$
  - auch Löschen von 1 Eintrag oder sortierter Menge in je $O(n)$
  - Ändern eines Eintrages: Finden per binärer Suche in $O(log_n)$ dann mit O(1) ändern

Alternative: Nutzung eines Binärbaums statt eines Arrays, dann typischerweise nach Aufbau in $O(nlog_n)$ jeweils Einzelzugriffe per $O(log_n)$

Heutige Themen

- Bisherige Musterarten (allgemein)
- Algorithmenmuster - Wozu?
- Übersicht über Algorithmenmuster
- **Greedy** - ständig gierig mit oft nur mäßigem Ergebnis
- **Generate and Test** - sehr allgemeines Muster
- **Brute Force** - doof aber ausdauernd
- **Backtracking** - intelligent, systematisch, kann Entscheidungen korrigieren
- **Teile und Herrsche** - clevere Problemreduzierungen

Beispiele für bisherige Arten von Mustern

- **Syntax**muster (Java), z.B.:

```
Variablendefinition inkl. Initialisiertung:    Typ Name = Wert;
    z.B. int a = 7;  String s = null;  Haus h1 = baueHaus();

Aufbau des for-Schleifenkopfes:
for( Initialisierung; Test ; Befehle vor weiteren Tests)
    z.B. for( int i=0;i<n;i++)   for( int i=n-1;i>=0;i=i-1)
```

- **Programmier**muster, z.B.:

```
Bestimmung ob Zahl ungerade ist als boolean: zahl%2==1
    z.B. if (z%2==1) System.out.println(z+" ist ungerade");

Tauschen zweier Werte: Hilfsvariable = w1; w1=w2; w2=Hilfsvariable;
    z.B. int h=a; a=b; b=h;   String s=x; x=y; y=s;
```

- **Test**muster, z.B.:
  - Protokollierung der Abarbeitung per System.out.println
  - Debuggen in Eclipse
  - Aufbau einer Testdatenbank für Ein- und Ausgaben einzelner Methoden
- **Formatier**muster
- ...

Algorithmenmuster - Wozu?

- beschreiben **übergeordnete Problemlösungsstrategien**, z.B. für Optimierungsprobleme bzw. Auswahlprobleme (Wegfindung, Packen, Parameterwerte)
- lassen sich auch in vielen bisher behandelten Algorithmen wiedererkennen, jedoch teilweise fließende Übergänge (nicht immer genau abgrenzbar)
- Kenntnis hilft Lösungsstrategien **für eigene Probleme** zu entwerfen bzw. abzuwägen
- für Auswahl mögliches **Kriterium: Ziel**
  - irgendeine Lösung finden („quick and dirty“) vs.
  - eine (optimale) Lösung finden vs.
  - alle Lösungen finden (systematisches Vorgehen), z.B. 7 Jahre für 27x27-Damen-Problem (siehe QR-Code)
- auch **„Eleganz vs. Effizienz“**, z.B. rekursive vs. iterative Berechnung oder **Größe des Suchraums** bzw. nötige Lösungsgeschwindigkeit

Wichtige Algorithmenmuster - Übersicht

- Greedy-Algorithmen

“gierige” Vorgehensweise, bei der stets nur die **lokal „beste“ Lösung** genutzt wird, führt allgemein nicht zur global besten Lösung (wenn überhaupt eine gefunden wird)

- Generate and Test

eine **Generator-Methode** generiert jeweils eine neue potentielle Lösung und eine **Test-Methode** untersucht dann die Qualität bis eine oder genügend Lösungen gefunden sind

- Brute-Force

mit relativ einfachem Code alle potentiellen Lösungen durchtesten, auch wenn es sehr lange dauert (sehr simpler Generator, der oft auch **„sehr sinnlose“ Lösungsvorschläge** generiert)

- Backtracking

solange das Abbruchkriterium nicht erfüllt ist und weitere potentielle Lösungen vorhanden sind, werden diese systematisch untersucht (zielstrebiges „Generate and Test“), dabei werden oft schon **getroffene Teillösungen wieder rückgängig** gemacht

- Divide and Conquer

teilt Datenmenge wiederholt (oft rekursiv, aber nicht notwendigerweise!), um schnell ans Ziel zu gelangen bzw. das zu lösende Problem stetig zu verkleinern und so besser zu beherrschen

Greedy - dumme „gierige“ Algorithmen (1)

ein Packproblem als Beispiel für Kosten/Nutzen-Probleme

- zur Verfügung stehen 4 Objektarten (A bis D) mit folgenden Eigenschaften:

Objektart:	A	B	C	D	Anmwerkung
Nutzen	30	35	24	12	z.B. Warenwert in Euro, sollen maximiert werden
Kosten	6	7	4	3	z.B. Volumen in Liter, Summe darf das Budget nicht überscheiten
N:K	5	5	6	4	ist evtl. auch ein interessantes Kriterium

- Budget (mögliche Kosten) sei 10 Liter (alternativ auch 10kg, ...)
- beste Lösung wäre: A+C (Nutzen=**54**, Kosten 10)
- Greedy1: Kriterium ist „höchster Nutzen“
  1. B → Restbudget = 10 - 7 = 3, Nutzen:35
  2. D → Restbudget = 3 - 3 = 0, Nutzen:**47**
- Greedy2: Kriterium ist „niedrigste Kosten“
  1. D → Restbudget = 10 - 3 = 7, Nutzen:12
  2. D → Restbudget = 7 - 3 = 4, Nutzen:24
  3. D → Restbudget = 4 - 3 = 1, Nutzen:**36**
- Greedy3: Kriterium ist „höchstes Nutzen:Kosten-Verhältnis“
  1. C → Restbudget = 10 - 4 = 6, Nutzen:24
  2. C → Restbudget = 6 - 4 = 2, Nutzen:**48** ← N:K hier am besten, aber < 54

Greedy - dumme „gierige“ Algorithmen (2)

Greedy-Grundidee: Wähle in jedem Schritt die jeweils beste Teillösung aus (im Beispiel: eines der Objekte), welche noch ins (Rest-)Budget passt.
Im Beispiel wurde mit keinem der Kriterien das Optimum gefunden! ← *typisch*

- typisches Einsatzziel: eine **einigermaßen gute Lösung schnell finden**, z.B. als Startpunkt für dann folgende Verbesserungsversuche, bis Zeit abgelaufen ist (z.B. in zeitkritischen Systemen)
- bei Lösungsfindung wächst Lösung schrittweise (z.B. jeweils Auswahl der nächsten Wegrichtung oder des nächsten einzupackenden Gegenstandes)
- verschiedenste Auswahlkriterien für Teilentscheidungen implementierbar

anderes Beispiel für Greedy: Wegfindung zur „Bergspitze“

- lokal wird Richtung der maximalen Steigung (Gradient) gewählt
  Annahme: hier geht es am schnellsten zur Spitze
- auch *Hill-Climbing-Strategie* genannt
- lokal werden keine Kosten betrachtet
- „Bergspitze“ könnte z.B. das Maximum einer Gütefunktion beim Optimieren sein (Hill-Climbing bzw. Gradientenverfahren zur Optimierung)

Generate and Test: ein sehr allgemeines Muster

- bewusste Trennung zwischen:
  - dem **Erzeugen** einer potentiellen Lösung und
  - dem **Bewerten** der Qualität dieser potentiellen Lösung
- beide Schritte werden **abwechselnd aufgerufen**
- **Erzeugung** unterschiedlich aufwändig, z.B.:
  - **simpel** (dann typischerweise oft schlechte Lösungsvorschläge), z.B. Brute-Force
  - **aufwändiger** (dann typischerweise oft bessere Lösungsvorschläge), z.B. Backtracking
- Ende kann z.B. erreicht sein, wenn:
  - die **erste Lösung** gefunden wurde,
  - *n* oder **alle Lösungen** gefunden wurden
- häufige **Basis** für neue Lösungen beim Generieren:
  - **letzte** potentielle Lösung (systematische Suche per Brute-Force oder Backtracking),
  - bisher **beste** Lösung (Suche in deren Nähe per Zufall und/oder Gradient),
  - Suche in der Nähe der *n* **besten** Lösungen (z.B. genetische Algorithmen),
  - manchmal auch gänzlich **zufällig** (genetische Algorithmen: reine Mutation)

Brute-Force - versuche „einfach alles“

wieder Packproblem als Beispiel

- zur Verfügung stehen 4 Objektarten (A bis D) mit folgenden Eigenschaften:

Objektart:	A	B	C	D
Nutzen	30	35	24	12
Kosten	6	7	4	3

- systematisches Durchtesten, z.B. einfach „0 oder 1 A“, „0 oder 1 B“, „0, 1 oder 2 C“, „0, 1, 2, oder 3 D“ (mehr geht jeweils nicht beim gegebenen Budget von 10)

```
int max=0;
String sel="";
for (int a=0; a<=1; a++)
    for (int b=0; b<=1; b++)
        for (int c=0; c<=2; c++)
            for (int d=0; d<=3; d++){ // Generierung fertig, nun Test
                if (a*6+b*7+c*4+d*3 <= 10) {// notwendige Bedingung
                    int quality = a*30 + b*35 + c*24 + d*12;
                    if (max<quality){
                        max=quality;
                        sel="A:"+a+" B:"+b+" C:"+c+" D:"+d+" -> "+quality;
                    }
                }
            }
System.out.println("best:"+sel);
```

- `best:A:1 B:0 C:1 D:0 -> 54` ← Optimum wurde nach 48 generate&test-Zyklen gefunden → **langsamer als Greedy** aber **gründlich!**

Backtracking - „auch mal einen Schritt zurück gehen“

wieder Packproblem als Beispiel

```
public class PackBackTrack {
    public static int[] n = { 30, 35, 24, 12 };   // Nutzen, z.B. Euro
    public static int[] k = {  6,  7,  4,  3 };   // Kosten, z.B. Euro, Kg, Liter
    public static int[] a = {  0,  0,  0,  0 };   // Anzahl
    public static int budget = 10;

    public static void main(String[] args){
        int num=0;
        int max=0;
        String sel="";
        while (generate()){
            int quality = n[0]*a[0] + n[1]*a[1] + n[2]*a[2] + n[3]*a[3];   // Test
            int sumKost = k[0]*a[0] + k[1]*a[1] + k[2]*a[2] + k[3]*a[3];   // nur zum Debugen (wird nie>10)
            System.out.println("test("+(++num)+"):"+"A:"+a[0]+" B:"+a[1]+" C:"+a[2]+" D:"+a[3]
                                                  +" -> "+quality+" k:"+sumKost);
            if (max<quality){ //  bestes finden und merken
                max=quality;
                sel="A:"+a[0]+" B:"+a[1]+" C:"+a[2]+" D:"+a[3]+" -> "+quality;
            }
        }
        System.out.println("best:"+sel);
    }

    public static boolean generate(){ // beachtet notwendige Bedingung (budget)
        for (int i=0; i<a.length; i++){ // beginnt bei niedrigster Stelle
            if (budget>=k[i]){  // Anzahl um 1 erhöhbar
                budget-=k[i];   // ja, Budget reicht -> aktualisieren
                a[i]++;         // Anzahl um 1 erhöhen
                return true;    // neuer Lösungsvorschlag kann getestet werden
            }
            budget+=a[i]*k[i];  // zurück ins Budget
            a[i]=0;             // zurück auf 0-Wert
        }                       // per i++ weiter mit nächster Stelle
        return false;           // kein neuer Vorschlag möglich!
    }
}
```

`best:A:1 B:0 C:1 D:0 -> 54` ← Optimum nach 12 generate&test-Zyklen gefunden

→ **langsamer als Greedy** aber **gründlich** und **schneller als Brute-Force**!

Backtracking - Tests sparend

Ausgabe zum Packproblem

```
test(1):A:1 B:0 C:0 D:0 -> 30 k:6
test(2):A:0 B:1 C:0 D:0 -> 35 k:7
test(3):A:0 B:0 C:1 D:0 -> 24 k:4
test(4):A:1 B:0 C:1 D:0 -> 54 k:10
test(5):A:0 B:0 C:2 D:0 -> 48 k:8
test(6):A:0 B:0 C:0 D:1 -> 12 k:3
test(7):A:1 B:0 C:0 D:1 -> 42 k:9
test(8):A:0 B:1 C:0 D:1 -> 47 k:10
test(9):A:0 B:0 C:1 D:1 -> 36 k:7
test(10):A:0 B:0 C:0 D:2 -> 24 k:6
test(11):A:0 B:0 C:1 D:2 -> 48 k:10
test(12):A:0 B:0 C:0 D:3 -> 36 k:9
best:A:1 B:0 C:1 D:0 -> 54
```

Wertung (hier und allgemein):

- Hier: Im Gegensatz zur Brute-Force-basierten Implementierung verlassen nur valide Lösungsvorschläge den Generator (siehe Ausgabe, kein k ist dort größer als das Budget von 10).
- Allgemein: Schon bei diesem kleinen Problem macht die Back-Tracking-basierte Implementierung deutlich weniger Tests. Bei größeren zu optimierenden Parametermengen ist die prozentuale Einsparung meist noch wesentlich deutlicher (viel weniger generate&test-Zyklen).
- Die Implementierung von Brute-Force ist jedoch meistens leichter zu verstehen als Backtracking.

Wozu Tests sparen: In der Produktoptimierung werden oft aufwändige Simulationen als Test der generierten Produktparameter verwendet. Diese Simulationen können sehr aufwändig sein, z.B. basierend auf dem Parametersatz die Form des Produkts berechnen, für Finite-Elemente-Methode „vernetzen“ und dann die FEM berechen. Es lohnt sich dort also möglichst nur sinnvolle Parametersätze zu generieren, um Tests und damit Zeit zu sparen.

Divide&Conquer: macht große Probleme beherrschbar

Grundidee: ursprüngliches Problem (bzw. ursprüngliche Datenmenge) wird schrittweise **aufgespalten** und so zu einem kleineren Problem oder mehreren kleineren Problemen, welche besser zu **beherrschen** sind.

- Aufspaltung kann z.B. sein:
  - Abspaltung eines Trivialproblems, z.B. Faktor bei der **Fakultät**sberechnung,
  - Abspaltung von nahezu der Hälfte des Problems, z.B. bei **binärer Suche**, wo jeweils nur in einer Hälfte weitergesucht wird,
  - Aufspaltung in zwei kleinere (jedoch nur am Ende triviale) Probleme, z.B. sind bei **Mergesort** beide Teilmengen schrittweise per Halbierung weiter zu zerlegen.
- Ablauf kann sein:
  - **rekursiv: Absturzgefahr** bei zu großer Rekursionstiefe!
    Ist besonders bei Problemaufteilungen „eleganter“, siehe z.B. **Baumtraversierung** oder Mergesort. **Anwendung nur wenn Stacktiefe garantiert groß genug ist!**
  - **iterativ:** keine Gefahr eines Stacküberlaufs
    Wird eher bei schrittweisen Abspaltung eines Trivialproblems vom ursprünglichen Problem, wie z.B. bei der Fakultätsberechnung angewendet. Aber auch die iterative Berechnung der **Fibonacci**-Zahlen ist relativ leicht zu verstehen, obwohl jeweils Aufteilung in 2 gleichartige Probleme erfolgt.

Heutige Themen

- Begriffe: Eigenschaften von Algorithmen
  - terminieren
  - determiniert
  - deterministisch
- Polynomberechnung, Hornerschema + Wichtigkeit von Tests
  - naiv: $O(n^2)$
  - „typisch“: $O(n)$
  - Hornerschema: auch $O(n)$, aber ca 50% weniger Multiplikationen
- Nullstellensuche in 1d (univariat): $y = f(x)$, $f(x_0) = 0$
  - Allgemeines
  - Aufgabe aus Sicht des Algorithmus
  - Generate & Test

Eigenschaften eines Algorithmus

Ein Algorithmus ist:

- terminierend

wenn er für jede beliebige Eingabe nach endlich vielen Schritten endet.
(**vs. Programmierung einer Endlosschleife** → nicht terminierend)

- deterministisch

bzw. hat einen deterministischen Ablauf, wenn er bei gleichen Eingaben stets den gleichen Ablauf und damit das gleiche Ergebnis aufweist.
(**vs. Einsatz von Math.random()** → nicht deterministisch)

- determiniert

bzw. er hat ein determiniertes Ergebnis, wenn gleiche Eingaben stets zu gleichen Ausgaben führen, evtl. aber auf unterschiedlichen Wegen!
(**z.B. int a=Math.random()>.5?-1:+1; return Math.abs(a);**)

Ansätze ein Polynom zu berechnen

- **Annahme:** Koeffizienten seien in einem **Array** gegeben, wobei der Index den jeweiligen Exponenten e für x^e angibt → bei Index 0 ist der konstante Anteil gespeichert, denn $x^0 = 1$ bzw. unabhängig vom x-Wert.
- schnell implementiert:

```
double polySimpel(double x, double[] coeff){
    double y=0;
    for (int e=0; e<coeff.length; e++)
        y = y + coeff[e]*Math.pow(x,e);
    return y;
}
```

- naiv:

```
double polyNaiv(double x, double[] coeff){
    double y=0;
    for (int e=0; e<coeff.length; e++){
        double xHochE = 1;
        for (int i=0;i<e;i++)
            xHochE *= x;
        y += coeff[e]*xHochE;
    }
    return y;
}
```

2 ineinander verschachtelte Schleifen → mit n als Länge des Koeffizienten-Arrays ergibt sich als **Aufwand** $O(n^2)$.

Ansätze ein Polynom zu berechnen (2)

- oft gesehen: **Potenzieren von x am Schleifenende**

```
double polyTypisch(double x, double[] coeff){
    double y=0;
    double xHochE = 1;
    for (int e=0; e<coeff.length; e++){
        y += coeff[e]*xHochE;
        xHochE *= x;
    }
    return y;
}
```

→ $2n$ Multiplikationen und n Additionen → $O(n)$

- nach dem Hornerschema lassen sich ca 50% der Multiplikationen einsparen

```
double polyHorner(double x, double[] coeff){
    double y=0;
    for (int e=coeff.length-1; e>=0; e--)
        y = y*x + coeff[e];
    return y;
}
```

entspricht dem Ausklammern, z.B.:

$$4*x*x*x-3*x*x+7*x+5 = ((4*x-3)*x+7)*x+5$$

→ nur n Multiplikationen und n Additionen → $O(n)$
→ mathematische Umformung kann helfen Zeit zu sparen, evtl. sogar genauer zu rechnen

Ansätze ein Polynom zu berechnen (3): Messungen

Testmethode zur Zeitmessung und ihre Nutzung in `main`:

```
public static void main(String[] a){
    double[] c2 = {0.004, -0.03, -0.001, 0.1, 1.4, -1.3, -1.1, 0.7};
    int num = 10_000_000;
    System.out.println("simpel :"+getCalcTimeInMs(1, num, c2));
    System.out.println("naiv   :"+getCalcTimeInMs(2, num, c2));
    System.out.println("typisch:"+getCalcTimeInMs(3, num, c2));
    System.out.println("Horner :"+getCalcTimeInMs(4, num, c2));
}

public static long getCalcTimeInMs(int func, int numTests, double[] c){
    long beg = System.currentTimeMillis();
    double f = 0.00003;
    double s = 0;
    switch (func){
        case 1: for (int i=0; i<numTests; i++) s+= polySimpel (i*f, c); break;
        case 2: for (int i=0; i<numTests; i++) s+= polyNaiv   (i*f, c); break;
        case 3: for (int i=0; i<numTests; i++) s+= polyTypisch(i*f, c); break;
        case 4: for (int i=0; i<numTests; i++) s+= polyHorner (i*f, c); break;
    }
    // System.out.println("func"+func+" sum="+s);
    return System.currentTimeMillis()-beg;
}
```

Ansätze ein Polynom zu berechnen (4) Vergleich

typische Ausgabe:

```
simpel :5217
naiv   :226
typisch:152
Horner :225
```

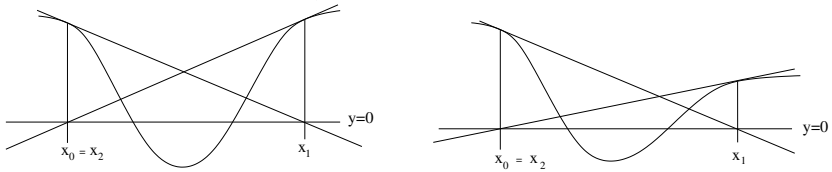
Schlussfolgerungen:

- Nutzung von **Math.pow() ist sehr zeitintensiv!**
  → weiterer Grund für `x*x` statt `Math.pow(x,2)`
- gezeigte Java-Implementierung war auf dem Testrechner für Horneransatz und naiven Ansatz immer etwa gleichwertig
- trotz mehr Rechenoperationen war der „typische“ Ansatz mit 2 Multiplikationen pro Durchlauf im Test am schnellsten

Konkrete Messungen und Vergleiche auf dem Zielsystem (-systemen) zur Überprüfung der theoretischen Überlegungen in der Praxis sind **wichtig**!

Nullstellenfindung (univariat): Allgemeines

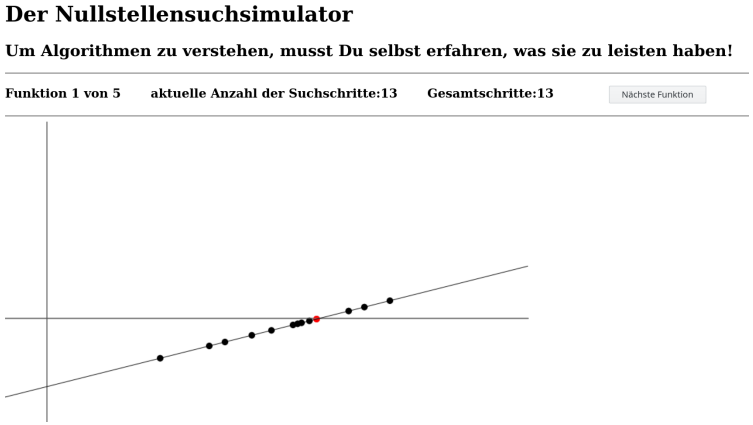
- Annahmen/Varianten:
  - Funktion y=f(x) ist **„Blackbox“** → Ableitung steht nicht zur Verfügung (könnte aber approximiert werden)
  - Schritt für Schritt wird die Funktion **„abgesampelt“**, d.h. für interessante x jeweils ein $f(x)$ berechnet
  - Funktionswerte sind möglicherweise für manche x **nicht berechenbar**
  - Mehrfachlösungen sind typisch: Wahl von „eine Lösung“ vs. „beste Lösung“ vs. „alle/viele/keine Lösungen“ (**Steuerbarkeit der Lösungssuche/-rückgabe** wichtig!)
  - interaktiv/**zeitkritisch** vs. zeitunkritisch
- **Robustheit** ist auch wichtig! 2 Beispiele, bei denen das Newtonverfahren versagt (gleiche Punkte werden immer wieder getestet):



→ **Speicher** für schon untersuchte Punkte könnte das verhindern; wäre auch z.B. sinnvoll, um „Übersichtswissen“ zu haben und nächstes x besser zu wählen

Aufgabenstellung aus Sicht des Algorithmus

Menschliche Intuition ist oft nicht leicht algorithmisch zu fassen (Heuristiken). Hilfreich kann die Einnahme der Sicht des Algorithmus sein, hier also das schrittweise Absampeln einer **unbekannten** Funktion $y = f(x)$ um die Nullstellen zu finden.



Umsetzung per Generate & Test

- Nullstellensuche typischerweise als „**Generate** & **Test**“ implementiert
- **Generate:** Erzeugen einer x-Position, an der gesampelt (getestet) werden soll
- **Test:** Berechnung des Funktionswertes $y = f(x)$ und Auswertung, ob es sich dabei um eine Nullstelle handelt
- **Vergleich** $y == 0$ ist möglich, aber **meist sinnlos!!!**
- Programmiermuster für Nutzung bei **Nullstellentests** (bzw. allgemeiner Vergleichen von Gleitkommazahlen):
  `Math.abs(y) < eps` mit z.B. `static double eps=1e-7;`
- allgemeiner zum **Vergleichen** von 2 Gleitkommazahlen `a` und `b`:
  `Math.abs(a-b) < eps`
- **Tests** verlaufen oft ähnlich. Hauptunterschied der Verfahren liegt im **Generate**-Teil.
- 2 sehr gegensätzliche Strategien bei **Generate** nutzbar:
  - Tiefe zuerst (punktuell schnell in die Tiefe, wie Greedy) → z.B. Newton-Verfahren
  - Breite zuerst (Suchbereich gleichmäßig untersuchen; Überblick bewahren, alles findbar, wie Brute Force) → z.B. rekursive Bisektion des jeweils größten Bereichs
- allgemein ist die Mischung beider Extreme am besten (robust & schnell, steuerbar)

Variablen

- sind Behälter für Zahlen, Zeichen, Wörter, etc.
- repräsentieren den in ihnen gespeicherten Inhalt im Programm
- Datentyp legt mögliche Inhalte (z.B. Zahlenbereiche) fest

Anlegen einer Variable in Java:

```
Datentyp bezeichner = wert;  // „= wert" ist optional
```

ganzzahlige Datentypen

Typ	Wertebereich	Beispiele
byte	-128 … 127	`byte b = 120;`
short	-32.768 … 32.767	`short s = 32000;`
int	-2^{31} … $2^{31}-1$, d.h. ca. ±2Mrd.	`int i = 2000000000;`

long	-2^{63} … $2^{63}-1$	`long l1 = 42; // typisch`
		`long l2 = 42L; // erlaubt`
		`long l3 = 2147483648L;`
		`// Stets L oder l am Ende, wenn Zahl`
		`// nicht in int-Bereich passt.`

Notationshinweise: - Präfixe 0x und 0b erlauben Eingaben als Hex- oder Binärzahl.
- Eine führende 0 macht Zahlen zu Oktalzahlen! 010 ist also acht statt zehn!
- Zwischen 2 Ziffern dürfen _ stehen (ist besser lesbar für lange Zahlen).

Beispiele: int a = 0xFFFC_23FE; long q = 0xFF_FF_F2_BA__33_00_CC_18L;
byte b=0b0110_1100; short s = 0b0010_1100_0101_0000; byte elf = 013;
long stundeInMs=3600_000; int compilerFehler = 018; // nur Ziffern 0 bis 7 erlaubt

Gleitkomma-Datentypen

Typ	Beschreibung	Beispiele

float	1+23+8=32 bit
	bis ca 8 Dezimalstellen
		`float f1 = 0.03f; // Hat f oder F am Ende!`
		`float f2 =(float)0.3; // explizite Konv.`

double	1+52+11=64 bit
	bis ca 17 Dezimalstellen
		`double d1 = 0.8; // typisch`
		`double d2 = 0.8d; // erlaubt`

Zeichen-Datentypen

Typ	Beschreibung	Beispiele
String	Zeichenkette, Konstanten z.B. "", "A", "abc \n X"	`String s = "Wort1 Wort2";` `String t = "Wort3 "+s;`
char	einzelnes Zeichen, Konstanten z.B. 'a', '\u263A'	`char c = 'a';`

Boolscher-Datentyp

Typ	Beschreibung	Beispiele
boolean	kann nur true oder false sein	`boolean boo = true;`

Felder / Arrays (1)

- enthalten beliebig viele Werte oder Felder (siehe mehrdimensionale Felder weiter hinten)
- Elementzugriff über den Index, d.h. 0… Feldlänge-1, der in eckigen Klammern **[ ]** steht, lesen und schreiben ist möglich
- haben ohne Initialisierung den Wert `null`, Vorsicht: Zugriff führt zu Absturz!
- werden mit vorgegebenen Werten initialisiert oder per `new` mit vorgegebener Länge erzeugt (automatisches typabhängiges Füllen)
- können nach Erzeugung in der Länge nicht mehr geändert werden
- Länge kann per bezeichner.**length** gelesen werden (aber nicht geändert!)

Beispiele für das Anlegen von eindimensionalen Feldern in Java:

a) mit vorgegebenen Werten

```
Datentyp[] bezeichner = {}; // leeres Feld; „Datentyp“ muss bekannt sein
long    [] longArr    = {}; // leeres long-Feld

int[] ganzzahlFeld = {1, 2}; // 2 Ganzzahlen
double[] glkoFeld = {1.0, 2.0}; // 2 Gleitkommazahlen
char[] charFeld = {'a', 'b', 'c'}; // 3 Zeichen
double[] dblArr = {0.0, d1, 7.9}; // 3 Gleitkommazahlen, davon 1 aus Variable

int[] intArr1 = null; // das "Nicht-"Feld; Zugriff führt zum Absturz
int[] intArr2; // hat wie intArr1 null als Wert (ist nicht zugreifbar)
intArr2 = {3,-4,17};  // nachträgliche Initialisierung ist möglich
```

b) mit vorgegebener Länge (Werte z.B. 0 bei Zahlen, "" bei Strings, false bei boolean)

```
Datentyp[] bezeichner = new Datentyp[0]; // allg. Schema für leeres Feld
Datentyp[] bezeichner = new Datentyp[n]; // allg. Schema für Feldlänge n

int n1 = 0;
int n2 = n1+1;
long[] longArr0 = new long[0];   // leeres long-Feld, da Länge=0
long[] longArr1 = new long[n1];  // leeres long-Feld, da n1=0
long[] longArr2 = new long[n2];  // long-Feld mit der Länge 1, da n2=1
```

Beispiele für den Lese- bzw. Schreibzugriff auf eindimensionale Felder:

```
System.out.println(dblArr[2]); // 7.9, denn Index 2 liest an der 3. Position
dblArr[2] = 3.3 + glkoFeld[0]; // überschreibt 7.9 mit 4.3   (=3.3+1.0)
dblArr[2] += 10; // überschreibt 4.3 mit 14.3   (=4.3+10.0)

System.out.print("dblArr hat "+ dblArr.length + " Elemente:");
for (int i=0; i<dblArr.length; i++)
    System.out.print(" "+dblArr[i]);
System.out.println();  // gehe in die neue Zeile

// fülle ein int-Feld mit den ersten 7 Quadratzahlen (beginne mit 0²)
int[] iArr = new int[7];
for (int i=0; i<iArr.length; i++) // Schleife über alle Feldelemente
    iArr[i]=i*i;
iArr = new int[]{1, 8, 27}; // Überschreiben mit den ersten 3 Kubikzahlen
String s = "{ "; // Beginn der Umwandlung von iArr in einen String
for (int i=0; i<iArr.length; i++) // Schleife über alle Feldelemente
    s = s + iArr[i]+" ";
s = s + "}"; // nun ist in s das Feld gespeichert -> gut für Ausgaben...
```

Kommentare / Ablaufsteuerung

- Jeder Befehl wird per Semikolon abgeschlossen.
- Ohne spezielle Steuerbefehle wird stets der nächste Befehl ausgeführt (Sequenz).
- Durch Befehle zur bedingten Abarbeitung (Verzweigung per **if/else,** Schleifen oder **switch/case**) kann der lineare Programmablauf gezielt geändert werden.
- Kommentare sollten genutzt werden, um den Quellcode zu erläutern.

Zeilenkommentar per //

- Alles von // bis zum Zeilenende wird von Java ignoriert.

Bereichskommentar per /* */

- Zwischen dem Beginn **/*** und dem Ende ***/** kann beliebiger Text stehen, welcher von Java ignoriert wird.
- Anfang und Ende können in der gleichen oder auch verschiedenen Zeilen sein.

Verzweigung per if / else:

- Ein boolescher Wert.wird ausgewertet (ggfs. vorher berechnet). Ist er wahr, dann werden die Befehle im **if**-Zweig ausgeführt, ansonsten die Befehle im **else**-Zweig.
- Nach dem Schlüsselwort **if** steht die Bedingung in runden Klammern. Dann folgt genau ein Befehl oder ein Block von Befehlen, welcher in geschweiften Klammern steht. Der **else**-Zweig ist optional. Beispiele:

```
if (true /* Dies ist eine konstanter Wahrheitswert */)
    a = a + 1;  // a wird stets um eins erhöht

if (true){      // gleicher Effekt wie oben
    a = a + 1;  // ein Befehlsblock kann 0, 1 oder beliebig
}               // viele Befehle (Anweisungen) enthalten
/* else  auskommentiert, da der else-Zweig hier sinnlos ist
    a = a - 1;
 */

if (a<10) {
    a = a + 1;
}
else {   // Hier geht es weiter wenn a>=10 ist.
    a = a * 3;
    System.out.println("a ist jetzt "+a);
}
```

Häufiger Fehler:
```
if (a<10); // Zwischen ) und ; ist ein leerer Befehl!
    a=a+1; // a wird immer erhöht. Der Test ist egal!
```
Korrektur: Das ; nach der runden Klammer entfernen! Dann wird a nur noch erhöht, wenn es kleiner als 10 ist.

while-Schleife:

- Nach dem Schlüsselwort **while** steht eine Bedingung in runden Klammern.
- Ist diese Bedingung wahr, dann wird der Befehl oder Befehlsblock (beliebig viele Befehle in geschweiften Klammern) ausgeführt, der nach der schließenden runden Klammer steht.
- Testen und Befehlsausführung erfolgt solange, bis die Bedingung nicht mehr wahr ist. Beispiel:

```
int i=0;
while (i<5) { // Befehlsblock mit 2 Befehlen
    System.out.println(i);
    i++;
}
```

- **Häufige Fehler:**

  - Semikolon nach der schließenden Klammer: **while (a<10);** // leerer Befehl in Schleife Wenn a<10, dann werden der leere Befehl und der Test endlos ausgeführt, denn a ändert seinen Wert nicht und die Bedingung bleibt immer wahr.

  - Bedingung ändert sich nie, obwohl in der Schleife Befehle ausgeführt werden. Zum Beispiel wenn eine Zählvariable am Schleifenende nicht weitergezählt wird.

for-Schleife:

- Nach dem Schlüsselwort **for** stehen runde Klammern. Zwischen diesen beiden Klammern stehen stets 2 Semikolons, welche 3 Bereiche voneinander abgrenzen::
  - Initialisierung, enthält z.B. die Initialisierung der Laufvariable: int i=1
  - Test, enthält eine Bedingung, ist sie wahr, dann wird die Schleife weiter ausgeführt (wie bei while-Schleife)
  - Befehl, der stets nach erreichen des Schleifenendes ausgeführt wird, z.B. i++
- Beispiel:

```
for (int i=0; i<5; i++)  // ein Befehl
    System.out.println(i);
```

do-while-Schleife:

- Nach dem Schlüsselwort **do** steht ein Befehl oder ein Befehlsblock (beliebig viele Befehle in einem Paar geschweifter Klammern). Dann folgt das Schlüsselwort **while** sowie eine Bedingung in runden Klammern.
- Beachte: do-while testet erst, nachdem die Schleife durchlaufen wurde! Man nennt sie (im Gegensatz zu while- und for-Schleife) auch „nicht abweisende Schleife“. Beispiel:

```
int i=0;
do { // Befehlsblock mit 2 Befehlen
    System.out.println(i);
    i++;
} while (i<5);
```

spezielle Befehle in Schleifen:

- **break**: verlässt die Schleife; Programmfortsetzung mit Befehlen **nach der Schleife**

- **continue**: Programmfortsetzung am Ende der Schleife bzw. mit dem Test

```
int n=5;
for (int i=1; i<=n; i++){
    if (i==2)
        continue; // zum Schleifenende (hier: i++)
    if (i==4)
        break;
}
```

```
int i=1;
while (i<=n){
    if (i==2){
        i++;          // sonst Endlosschleife!
        continue;     // gehe zum Test
    }
    if (i==4)
        break;
    i++;
}
```

```
i=1;
do{
    if (i==2){
        i++;          // sonst Endlosschleife!
        continue;     // gehe zum Test
    }
    if (i==4)
        break;
    i++;
} while (i<=n);
```

switch-case-Verzweigung:

- kann mehrere if- bzw. if-else-Anweisungen ersetzen, wenn sie den selben Wert testen

```
for (int i=0; i<=3; i++){
    String s="nix";
    if      (i==1) s="eins";
    else if (i==2) s="zwei";
    else if (i==3) s="drei";
    System.out.println(i+" "+s);
}
```

gibt aus:
0 nix
1 eins
2 zwei
3 drei

- **Wichtig**: Zwischen case und Doppelpunkt müssen **Konstanten** stehen! Per Schlüsselwort ***final*** lassen sich Variablen zu Konstanten machen: final int MYPI = 3;

```
for (int i=0; i<=3; i++){
    String s="nix";
    switch(i){ // teste i
       case 1 : s="eins"; break;
       case 2 : s="zwei"; break;
       case MYPI : s="drei"; break;
    }
    System.out.println(i+" "+s);
}
```

gibt aus:
0 nix
1 eins
2 zwei
3 drei

- **break** verlässt die switch-case-Verzweigung. Ohne break-Anweisung würden auch die Befehle der anderen Cases ausgeführt werden (bis zum nächsten break).
- Der **default**-case ist optional und kann verwendet werden, um alle nicht aufgelisteten Fälle zu behandeln. Er kann an beliebiger Position aufgeführt werden, z.B.:

```
for (int i=0; i<=3; i++){
    String s;
    switch(i){ // teste i
       case 1 : s="eins"; break;
       case 2 : s="zwei"; // hier mal ohne break
       case 3 : s="drei"; break;
       default: s="nix";  break;
    }
    System.out.println(i+" "+s);
}
```

gibt aus:
0 nix
1 eins
2 drei
3 drei

- In switch(i) lassen sich statt i beliebige int-Werte oder automatisch zu int konvertierbare Werte (z.B. char, byte, short) verwenden. Ab Java 7 sind sogar Strings zulässig!

```
Sting[] x={"DREI", "two", "aha", "uno"};

for (int i=0; i<x.length; i++){
    String s;
    switch(x[i]){ // teste x[i]
       case "uno" : s="eins"; break;
       case "two" : s="zwei"; break;
       case "DREI": s="drei"; break;
       default    : s="nix";  break;
    }
    System.out.println(x[i]+" "+s);
}
```

gibt aus:
DREI drei
two zwei
aha nix
uno eins

Methoden

- Methoden bieten eine Möglichkeit Programmcode mehrfach zu nutzen.
- Bei der Definition ist festzulegen:
  - Rückgabetyp der Methode (z.B. `int` oder **`void`**, falls keine Rückgabe erfolgt)
  - Name der Methode (erstes Zeichen ist ein kleiner Buchstabe)
  - Parameterliste (in runden Klammern, mit 0 bis n durch Komma getrennten Parameterdefinitionen; pro Parameter: jeweils Typ und lokaler Parametername)
  - auszuführende Befehle in geschweiften Klammern
- Rückgabetyp, Name und Parameterliste bilden die sogenannte Signatur.
- Methoden werden zunächst stets als **`public static`** markiert (weiteres dann beim Thema OOP = „**O**bjekt**o**rientierte **P**rogrammierung“).
- Bekannteste Methode ist die **`main`**-Methode, mit der ein Programm gestartet werden kann. Dazu muss sie wie folgt definiert sein:

```
public static void main(String[] arrayName){
    // beliebige Befehle
}
```

- Aufgerufen (gestartet) wird eine Methode durch den Namen und eine passende Parameterliste (Anzahl und Typ der Parameter müssen stimmen!).
- Beendet wird eine Methode nach der Ausführung ihres letzten Befehls oder nach Ausführung eines **`return`**-Befehls. Dann wird die Programmabarbeitung nach dem Methodenaufruf fortgesetzt.
- Ist der Rückgabetyp **`void`**, dann muss nach **`return`** stets direkt das Semikolon stehen. Andernfalls steht zwischen return und Semikolon der Rückgabewert passenden Typs (z.B. als Konstante, Variable oder Ergebnis einer Methode).
- **Wichtig**: Methoden dürfen nie in anderen *definiert* werden! Der Aufruf anderer Methoden ist hingegen ganz normal.

Rekursion

- Methoden können sich auch selbst aufrufen. Dies kann direkt (siehe Beispiel zur Fakultätsberechnung) oder indirekt geschehen (PingPong-Beispiel):

```
public static long fak(int n){
    if (n<=1)
        return 1;       // Abbruch der Rekursion
    return n*fak(n-1);
}

public static String ping(int n){
    return "ping("+n+") "+pong(n+1);
}

public static String pong(int n){
    if (n>=11)
        return "pong("+n+")"; // Abbruch der Rekursion
    return "pong("+n+") "+ping(n+1);
}
```

- **Häufige Fehler:**
  - Abbruchbedingung vergessen (oder falsch definiert)

```
public static long fak_endlos(int n){
    return n*fak_endlos(n-1);  // Abbruch durch Stackoverflow!!!
}
```

  - zu viele Aufrufsebenen; je nach Java-Installation gibt es eine maximale Verschachtelungstiefe

```
public static void count(int n){
    if (n>1000000)
        return;    // Abbruch wird üblicherweise nie erreicht
    System.out.println(n); // z.B. bis 10821, dann Abbruch
    count(n+1);  // Abbruch durch Stackoverflow!!!
}
```

Boolsche Variablen/Ausdrücke

- **Gewinnung boolscher Werte:** Konstante, aus Variable, aus Variablenfeld, Vergleichsergebnis, Boolsche Operation, Methodenrückgabewert
- **Beispiele zur Nutzung**/Gewinnung (beliebig mischbar)**:**
  - Initialisierung von Variablen:

```
boolean ok=true;   // init. mit Konstante
boolean nok=!ok;   // init. mit neg.Variable
```

  - Methodenparameter:

```
boolean skipZero=true;   // init. mit Konstante
multipilyArrayValues(array, skipZero);
```

  - Verzweigung per if:

```
if (n>=11)              // Vergleichsergebnis
    if (true)           // Konstante
        ok=isPrim(a);   // Methodenrückgabewert
```

  - Schleifenabbruch:

```
while (!true);              // Endlosschleife? nein!
for ( ;weiterMachen(n); )   // Abbruch wenn Methode
                            // false zurückgibt
```

  - komplexere Beispiele:

```
public static boolean isPosAndSmall(int n){
    boolean isP  = n>=1;            // aus Vergleichsergebnis
    boolean isS  = n<10;            // aus Vergleichsergebnis
    boolean isPS = isP && isS;      // aus boolscher Operat.
    return isPS;                    // als Rückgabewert
}

public static boolean isPS(int n){    // Kurzfassung von
    return n>0 && n<10;               // isPosAndSmall
}

boolean[] isPrim = new boolean[9];    // Default-Werte: false
for ( int i=1; i<isPrim.length; i++)
    isPrim[i] =  i==1 || i==2 || i==3 || i==5 || i==7;
System.out.println("ist keine Primzahl(5):" + !isPrim[5]);
```

Operatoren / Operationen

- Additive und multiplikative Operatoren: +, -, *, /, %
  - „Punktrechnung vor Strichrechnung“ gilt auch in Java.
  - Notfalls helfen runde Klammern.
  - Modulo-Operator % dient zur Restberechnung bei Ganzzahldivision.
  - + Operator für Addition, aber auch zur String-Bildung.
  - / kann Ganzzahldivision sein (beide Operanden sind Ganzzahlen) oder Gleitkommadivision (sonst).

- **Vergleich**soperatoren (z.B.: `n>=1`) erzeugen boolesche Werte: >, <, >=, <=, !=, ==

- **logische** Operatoren benutzen und erzeugen boolesche Werte:

ungleich!

	Bedeutung	z.B.	Details
!	Negation	!x	true → false; false → true
&&	bedingt auswertendes Und	x&&y	y nur getestet wenn x==true
||	bedingt auswertendes Oder	x||y	y nur getestet wenn x==false
^	exklusives Oder (xor)	x^y	entspricht x!=y (Ungleichheit)
|	Oder	x|y	testet stets beide
&	Und	x&y	testet stets beide

- **Shift**-Operatoren und **Bit**-Operatoren ändern das Bitmuster von Ganzzahlen.
  Beispiele für byte p=0b111 (also 7) und byte n= -4 (binär: 1111_1100):

	Bedeutung	z.B.		Ergebnisse	
~	Negation	~p	~n	1111_1000	0000_0011
>>	vorzeichenerhaltendes Shift rechts	p>>1	n>>1	0000_0011	1111_1110
<<	Shift nach links (VZ stets weg)	p<<6	n<<6	1100_0000	0000_0000
^	bitweises exklusives Oder	p^n		1111_1011	
|	bitweises Oder	p|n		1111_1111	
&	bitweises Und	p&n		0000_0100	

Ein vorzeichenverwerfendes Rechtsshift >>> fügt links jeweils 0 ein. Beispiel:
Umwandlung einer **int**-Zahl **x** (insbes. negative Werte) in ihren Bitmuster-String s:

```
String s="";
for (int i=0; i<8; i++){ // zeigt die 8 untersten Bits
   s=(x&1)+s;
   x>>>=1; // bzw. x=x>>>1; schiebt von links eine Null ein
}          // dadurch wäre auch do{...}while(x!=0); möglich
```

- **ternärer Operator**: ermöglicht kurze Quelltexte durch bedingte Auswertung
  Aufbau: boolscher Ausdruck **?** Wert bei true **:** Wert bei false
  Beispiel: `int a=3; int b=7;  int min = a<b ? a : b;`

- **Prioritäten**: ohne Klammerung werden die Operationen entsprechend den in Java definierten Prioritäten ausgewertet, z.B. && vor ||. **Tipp: im Zweifelsfall immer Klammern setzen!** Vgl. auch die Java-Dokumentation z.B. bei Oracle:
  https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html
  wo z.B. zu sehen ist, dass Punktrechnung („multiplicative“ operators) eine höhere Priorität als Strichrechnung („additive“ operators) hat. Ebenso ist „logical AND“ höher als „logical OR“ priorisiert.
  Eine Zuweisung hat stets die geringste Priorität.

- Die **Auswertungsreihenfolge** (von links nach rechts) bei 2 Operanden sowie bei mehreren Operatoren gleicher Priorität ist wichtig, z.B. wenn Methodenrückgabe werte verwendet werden:

```
if (isNice(n) && isPrime(n)) // && hat 2 Operanden
    return true;
```

Die rechenzeitintensivere Methode `isPrime` sollte hier zuletzt ausgeführt werden (ggfs. Zeiteinsparung, wenn die „schnellere“ erste Berechnung bereits **`false`** ist).
Auch bei Tests von z.B. Arrays ist die Reihenfolge wichtig:

```
if (arr.length==0 || arr==null) // || hat 2 Operanden
```

wird z.B. abstürzen, wenn **`arr`** gleich **`null`** ist. Korrekt wäre:

```
if (arr==null || arr.length==0)
```

Die „von links nach rechts“-Regel zeigt sich auch hier:

```
System.out.println("a+b="+a+b); // sicher unerwartet, da 1+1=11
System.out.println("a-b="+a-b); // Compilerfehler, wegen String-b
```

- **Kombination von Berechnung und Zuweisung** wie bei += gibt es auch für andere Operationen: -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=
  Es sei v eine Variable, o= ein „allgemeines“ Operatorsymbol und x ein Operand,

dann ist	v o= x;	ausführlich schreibbar als	v = v o x;
z.B. ist	a += 1;	gleichbedeutend mit	a = a + 1;
und	x /= a;	gleichbedeutend mit	x = x / a;
und	b |= ampelRot();	gleichbedeutend mit	b = b | ampelRot();

- Anwendung von Wissen aus der TI:
  - Bsp. Umsetzung Karnaugh-Plan

x32 \ x01	00	10	11	01
00	0	1	1	0
01	0	1	1	0
11	0	1	1	0
10	0	0	0	0

```
return x0 && (x2 || !x3);
```

- Beim Increment und Decrement per ++ bzw. -- gibt es zu beachten:
  - **i++** erhöht den Wert in i um 1, nutzt dann aber den **alten** (kleineren) Wert (z.B. für Ausgabe, in Berechnung oder Zuweisung). Auswirkung:
    i = i++; // **Dies verändert den Wert in i nicht!**
    i++; // Dies ist der übliche Befehl, um eine Variable um 1 zu erhöhen!
  - Auch **++i** erhöht den Wert in i, nutzt dann aber dann den **neuen** (höheren) Wert, d.h. i = ++i; würde funktionieren, ist aber unnötig umständlich.

interessante Methoden/Konstanten der Math-Klasse

- siehe z.B. https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html

Name	Bedeutung	z.B.	Details
E	2.718281828459045		double-Wert
PI	3.141592653589793		double-Wert
abs	Absolutwert	abs(x)	x<0 ? -x : x
min	Minimum 2er Werte	min(x,y)	x<y ? x : y
max	Maximum 2er Werte	max(x,y)	x>y ? x : y
sin	Sinus (in Radiant)	sin(x)	Wert ggfs. umrechnen!
cos	Kosinus (in Radiant)	cos(x)	Wert ggfs. umrechnen!
atan2	Arkustangens	atan2(dy,dx)	kein Problem mit Div. durch 0
pow	Hoch	pow(b,e)	Basis b hoch Exponent e
sqrt	Quadratwurzel	sqrt(x)	NaN bei negativem x!
random	Zufallszahl	random()	Bereich [0...1)

- diverse Beispiele:

```
double min = Math.min(a, Math.min(b,c)); // Minimum 3er Werte
double wrz = Math.sqrt(a<0 ? 0 : a);  // Test vermeidet NaN („not a
           // number“) bei Rechenungenauigkeiten, z.B. wenn a=-1e-17
double inDeg = Math.atan2(dy,dx)*180/Math.PI; // Ergebnis in Grad
```

- Beispiele zur Ermittlung von Zufallszahlen:

```
public static int rndAusführlich(int v, int b){
    // im Beispiel sei v=6 und b=8
    double z = Math.random(); // 0 bis 0.999999...
    int w = b-v+1;            // „Breite“, hier w=8-6+1=3
    z *= w;                   // 0 bis 2.999999...
    z += v;                   // 6 bis 8.999999...
    return (int)z;            // 6 oder 7 oder 8
}

public static int rnd(int v, int b){
    return (int) (Math.random()*(b-v+1)+v);
}

public static double rndAusführlich(double v, double b){
    // im Beispiel sei v=6 und b=8
    double z = Math.random(); // 0 bis 0.999999...
    int w = b-v;              // „Breite“, hier w=8-6=2
    z *= w;                   // 0 bis 1.999999...
    z += v;                   // 6 bis 7.999999...
    return z;                 // 6 bis 7.999999...
}

public static double rnd(double v, double b){
    return Math.random()*(b-v)+v;
}
```

- Verbesserte/robustere/erweiterte Beispiele

```
public static int rndAusführlich(int v, int b){
    // falls b<v oder negative Zufallszahlen gewünscht sind
    double z = Math.random(); // 0 bis 0.999999...
    int w = Math.abs(b-v)+1;  // garantiert positive Breite
    z *= w;                   // wegen +1 als Gleitkommazahl
                              // eigentlich noch zu breit
    int r = (int)z;           // Abschneiden der Nachkommastellen,
                              // jetzt passt die Breite
    return Math.min(b,v)+r;   // Minimum vermeidet Fehler falls b<v
}

public static int rnd(int v, int b){
    return (int)(Math.random()*(Math.abs(b-v)+1))+Math.min(b,v);
}

public static int rnd2(int v, int b){
    return (int)(Math.random()*((b<v?(v-b):(b-v))+1))+(b<v?b:v);
}

public static double rnd( double v, double b){
    return Math.random()*Math.abs(b-v)+Math.min(b,v);
}

public static int[] rndArrX(int vLen, int bLen, int vVal, int bVal){
    int[] a = new int[ rnd(vLen, bLen) ]; // Absturz bei neg. Länge
    for (int i=0; i<a.length; i++)
        a[i]=rnd(vVal, bVal);
    return a;
}

public static int[] rndArr(int vLen, int bLen, int vVal, int bVal){
    // negative Feldlängen könnten z.B. zu 0 werden -> kein Absturz
    int[] a = new int[ rnd(vLen<0?0:vLen, bLen<0?0:bLen) ];
    for (int i=0; i<a.length; i++)
        a[i]=rnd(vVal, bVal);
    return a;
}
```

- Beispiele zum Runden auf Nachkommastellen:

```
public static double runde2(double d){ // Runden auf 2 Nachkommastellen
    return ((long)(d*100+(d<0?-.5:.5)))/100.0; // siehe ternärer Operator
}

public static double runde2_ausfuehrlich(double d){ // z.B. d=12.34567
    double o = 0.5;       // Offset für korrektes Auf-/Abrunden
    if (d<0)
        o = -0.5;         // negatives d braucht negativen Offset
    double g = d*100+o;   // g = 1235.067  ( = 1234.567 + 0.5)
    long   h = (long)g;   // h = 1235      (Nachkommastellen nun weg)
    double a = h/100.0;   // a = 12.35     („Zurückverschieben“)
    return a;             // auf 2 Stellen gerundete Zahl zurückgeben
}

public static double runde(double d, int n){ // Runden auf n NK-Stellen
    double f = Math.pow(10, n);          // Faktor für n berechen
    return ((long)(d*f+(d<0?-.5:.5)))/f; //+oder-.5 zum Auf-/Abrunden
}
```

Felder / Arrays (2)

- mehrdimensionale Felder sind Felder von Feldern
- Elementzugriff pro Dimension wieder über die Indices, d.h. jeweils 0… Feldlänge-1
- Indices stehen jeweils einzeln in eckigen Klammern **[ ]**, Reihenfolge ist wichtig!
  Anfang: Index im äußersten Feld (verweist im mehrdim. Feld stets auf ein inneres Feld)
  Ende: Index eines innersten Feldes (erst hier wird z.B. auf ein int oder float verwiesen)
- innere Felder können im Allg. **unterschiedliche Längen haben oder auch `null` sein**
- auch mehrdimensionale Felder haben ohne Initialisierung den Wert `null`
- werden mit vorgegebenen Werten (Verwendung von {} und Kommas) initialisiert oder per `new` mit vorgegebenen Längen erzeugt (automatisches typabhängiges Füllen)
- innere Felder können auch ausgetauscht oder gelöscht werden

Beispiele für das Anlegen von mehrdimensionalen Feldern in Java:

a) mit vorgegebenen Werten

```
Datentyp[][] bezeichner = {}; // leeres 2d-Feld; „Datentyp“ muss bekannt sein
long    [][] longArr0   = {}; // leeres 2d-long-Feld

int[][] ganzzahlFeld = {{11, 12},{21, 22}}; // 2+2 Ganzzahlen
int[][][] cube = {{{0, 1},{2, 3}},{{4, 5},{6, 7}}}; // 2*2*2=8 Ganzzahlen
double[][] glkoFeld = {{1.0, 2.0}, {3.0}}; // 2+1 Gleitkommazahlen

String[][] strFeld = {{"ja","nein"}, {"yes","no"}}; // 2+2 Zeichenketten
char[][] charFeld = {{‘a‘, ‘b‘, ‘c‘},  {‘A‘, ‘B‘, ‘C‘}}; // 3+3 Zeichen

long[][] longArr1 = null; // das “Nicht-“Feld; Zugriff führt zum Absturz
long[][] longArr2; // hat wie longArr1 null als Wert (ist nicht zugreifbar)
longArr2 = new long[][]{{3, 17}};  // nachträgliche Initialisierung ist
                                   // möglich aber etwas umständlich
```

b) mit vorgegebener Länge (Werte z.B. 0 bei Zahlen, “” bei Strings, false bei boolean)

```
Datentyp[][] bezeichner = new Datentyp[0][]; // allg. Schema leeres 2d-Feld
Datentyp[][] bezeichner = new Datentyp[m][n]; // allg. Schema für Rechteck
Datentyp[][] bezeichner = new Datentyp[n][n]; // allg. Schema für Quadrat
Datentyp[][][] bezeichner = new Datentyp[n][n][n]; // allg. Schema für Würfel

int m = 3;
int n = m+1;
long[][][] longArr = new long[m][m][m]; // Datenvolumen mit 27 long-Werten
int[][] intArr = new int[m][n];  // 2d-Feld mit 3*4 int-Werten
```

Beispiele für den Lese- bzw. Schreibzugriff auf mehrdimensionale Felder:

```
System.out.println(strFeld[1][0]); // gibt yes aus

strFeld[1] = new String[]{"oui","non"};
                            // überschreibt englische Texte (komplettes
                            // inneres Feld wird ausgetauscht)
System.out.println(strFeld[1][0]); // gibt oui aus

System.out.println(“strFeld speichert “+ strFeld.length + “ Sprachen:“);
for (int i=0; i<strFeld.length; i++){
    for (int j=0; j<strFeld[i].length; j++)
         System.out.print(“ “+strFeld[i][j]);
    System.out.println();  // gehe in die neue Zeile
}
```

```
// Rückgabe eines 3d-Arrays als Java-Quellcode-String
public static String toStr(double[][][] a){
    if(a==null) // Spezialfall und Fehlerbehandlung
        return "null"; // nicht initialisiert!
    String s="{";
    for (int i=0; i<a.length; i++){
        if (i>0)
            s+=",\n";
        if (a[i]==null){
            s+="null";
            continue;
        }
        s+="  {";
        for (int j=0; j<a[i].length; j++){
            if (j>0)
                s+=",\n";
            if (a[i][j]==null){
                s+="null";
                continue;
            }
            s+="    {";
            for (int k=0; k<a[i][j].length; k++){
                if (k>0)
                    s+=", ";
                s += a[i][j][k];
            }
            s+="}\n";
        }
        s+="}\n";
    }
    return s+"}";
}

// Abstände zum Ursprung in Normvolumen [-1..+1]^3
// n ist die Anzahl der Samples auf einer Seite, z.B.:
//    bei n=2 pro Koordinate: -1.0, -0.5, 0.0, +0.5, +1.0
public static double[][][] normDistTo0(int n){
    if(n<=0) // Spezialfall und Fehlerbehandlung
        return new double[][][]{0}; // nur Ursprung
    int l = 2*n+1;
    double f = 2.0/(l-1);
    double[][][] a = new double[l][l][l];
    for (int x=0; x<a.length; x++){ // 0..l-1
        double px =x*f-1.0; // -1..1
        double pxq=px*px;   // 0..1
        for (int y=0; y<a[x].length; y++){
            double py =y*f-1.0; // -1..1
            double pyq=py*py;   // 0..1
            for (int z=0; z<a[x][y].length; z++){
                double pz=z*f-1.0; // -1..1
                a[x][y][z] = Math.sqrt(pxq + pyq + pz*pz);
            }
        }
    }
    return a;
}
```