Freie Universität Berlin

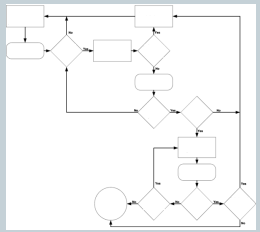
Pro Informatik: Objektorientierte Programmierung

GRUNDLANGEN DER BERECHENBARKEIT

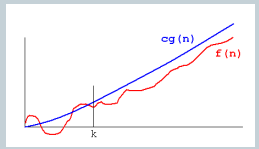
$$\{P\}\ S\ \{Q\}$$

FORMALE VERFAHREN ZUR SPEZIFIKATION UND VERIFIKATION IMPERATIVER PROGRAMME

KONZEPTE IMPERATIVER UND OBJEKTORIENTIERTER PROGRAMMIERUNG



PROGRAMMIERMETHODIK



ANALYSE VON LAUFZEIT UND SPEICHERBEDARF

Informationen zur Vorlesung

Freie Universität Berlin

Vorlesungen:

Mo-Fr 9-12 Uhr c.t. (es gibt Pausen)
Räume siehe Veranstaltungswebseite
Fragen an Marco Block, *block@inf.fu-berlin.de*

Übungen:

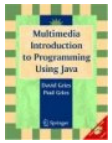
Mo-Fr 14-17 Uhr (Abid Hussain, *hussain@inf.fu-berlin.de*)
Rechnerräume: K44 und K46

Übungzettel gibt es nach Bedarf am Ende einer Vorlesung. Abgabetermin ist darauf entsprechend vermerkt.

Literatur:

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*"
Springer-Verlag 2007

Buch zur Vorlesung



Gries D., Gries P.: "*Multimedia Introduction to Programming Using Java*"
Springer-Verlag 2005

Saake G., Sattler K.U.: "*Algorithmen und Datenstrukturen - Eine Einführung mit Java*"
dpunkt-Verlag 2004

weitere Literatur wird angegeben...

Informationen zur Vorlesung

Mailingliste:

http://lists.spline.inf.fu-berlin.de/mailman/listinfo/oo_2008

- wichtige Änderungen oder Mitteilungen

Forum:

http:// www.java-uni.de

Das Java-Forum dient als Diskussionsplattform.

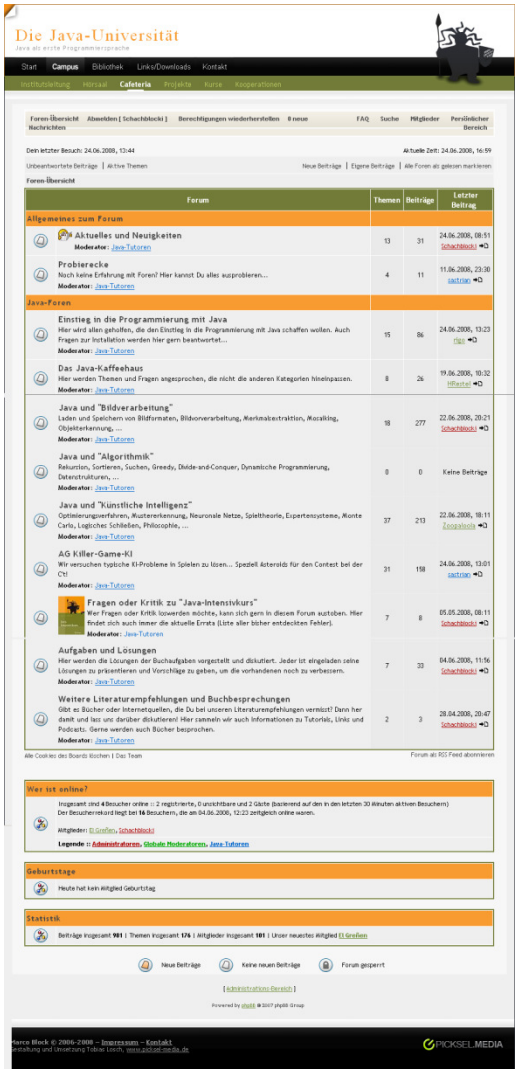
- das virtuelle Tutorium, helft Euch gegenseitig

Veranstaltungswebseite:

http://www.inf.fu-berlin.de/lehre/SS08/PI01/oo1.html

- Vorlesungszeiten, -orte
- Vorlesungsfolien, Übungszettel und Material zur Vorlesung

Bitte in die Mailingliste eintragen und im Forum registrieren...



Informatik-Studium an der Freien Universität Berlin

Inhalt:

- Studieren im Fachbereich Mathematik/Informatik an der Freien Universität Berlin
- Projektauswahl am Fachbereich

Vorlesung und Tutorium:

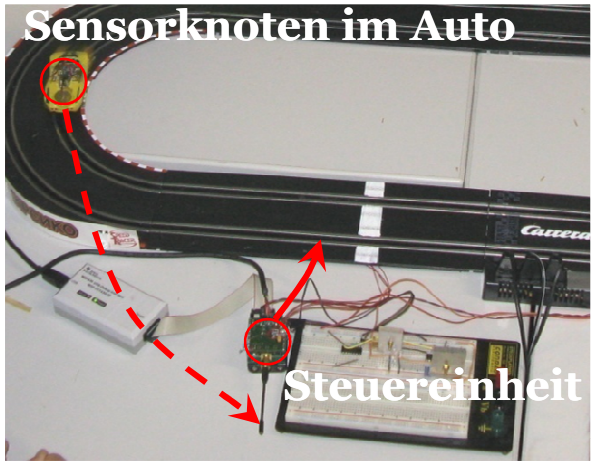
- Vorlesungsstart wird mit Kürzel angegeben: Beginn 9 Uhr c.t. (9:15)
  c.t. (*cum tempore*) mit Zeit oder „Akademisches Viertel “
  s.t. (*sine tempore*) ohne Zeit
  m.c.t (*magno cum tempore*) unüblich „9:30“
  mm.c.t (*maximo cum tempore*) unüblich „9:45“
- Vorlesung ein oder zwei Termine (á 2 Stunden)
- Tutorium 2 Stunden zur Aufarbeitung des Vorlesungsstoffs, Vorrechnen von Übungsaufgaben und Abgabe der Übungszettel
- Tutoren leiten den Übungsbetrieb
- Zum Ende eine Klausur

Informatik-Studium an der Freien Universität Berlin

Sensornetzforschung im Projekt FeuerWhere

MSB430-H Sensorknoten im Auto

- 3D Beschleunigungs-Sensor
- Filterung der Messdaten im Auto
- Funken (868 MHz) der Querbeschleunigungsdaten



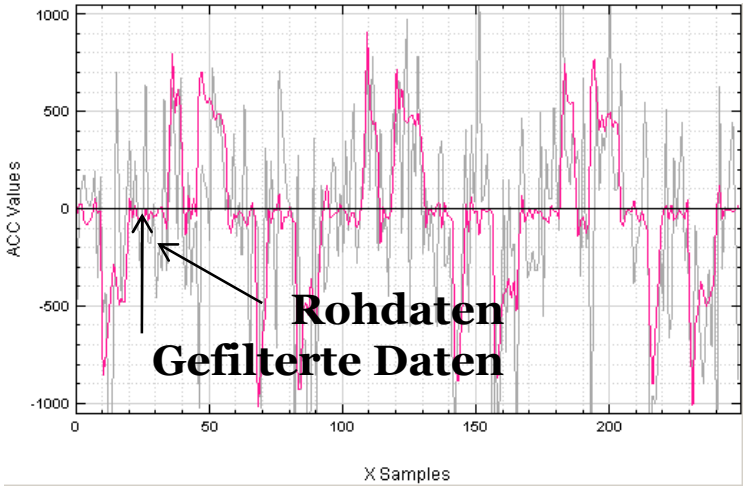
Carrera-Versuchsaufbau LNDW 2008

MSB430-H Steuereinheit

- Auswertung der Querbeschleunigungsdaten
- Zeitkritische Steuerung der Geschwindigkeit
- Zeitmessung der Runden an der Strecke

Reale Anwendung

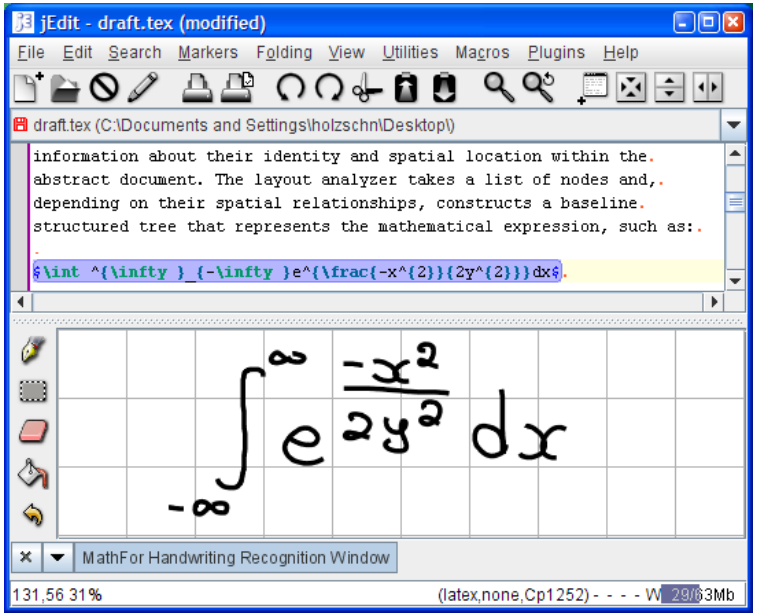
- Projekt FeuerWhere
- Bewegt sich Einsatzkraft noch?
- Ereigniserkennung (Schritt, Sturz, etc.)

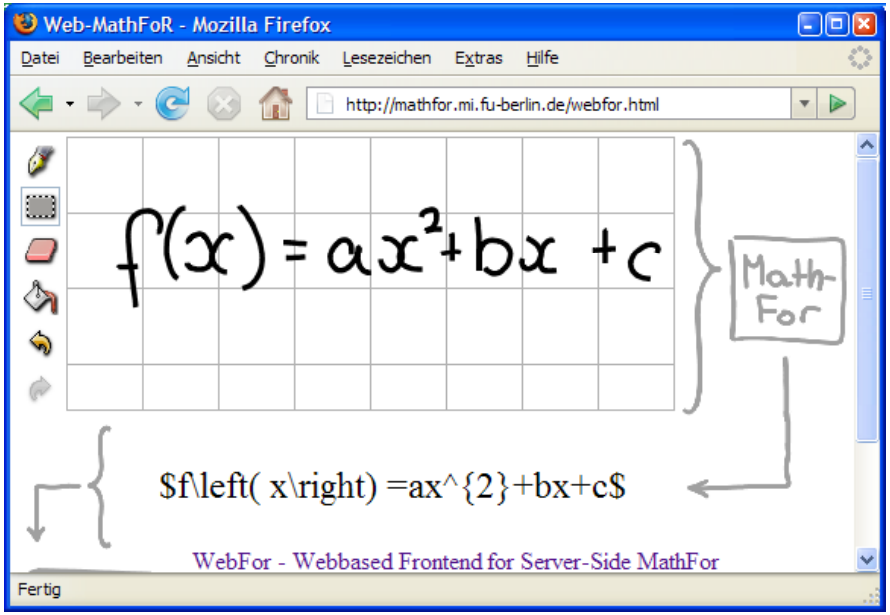


Filterung der Rohdaten im Auto
(weighted moving average)

Erkennung von mathematischen Formeln

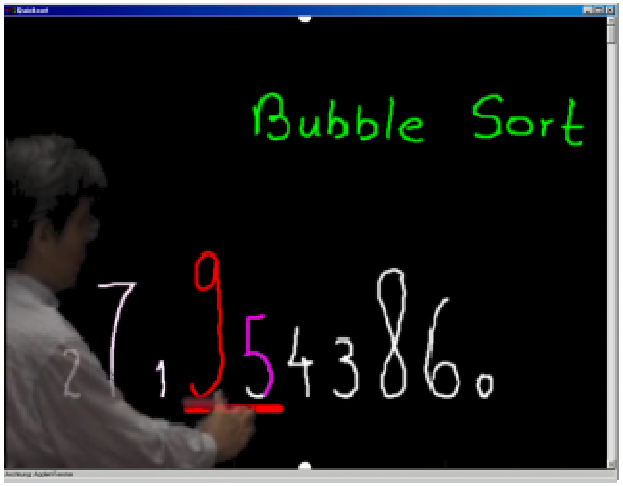
Formeln schreiben, automatisch erkennen und anwenden.
http://mathfor.mi.fu-berlin.de/pmwiki/pmwiki.php





Elektronische Kreide

Schreiben auf der elektronischen Tafel, z.B. in Vorlesungen. Mehr Informationen auf:
http://www.e-kreide.de/



Vorlesungen anschließend mehrfach anschauen. Es gibt auch eine Mitschrift. Beispiele zu Vorlesungen von Prof.Dr.Raúl Rojas finden sich z.B. hier:

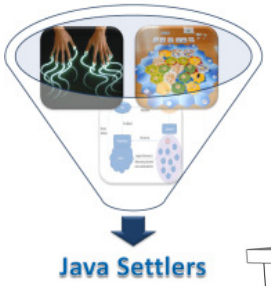
http://www.marco-block.de
(dann unter Rubrik **Lehre**)

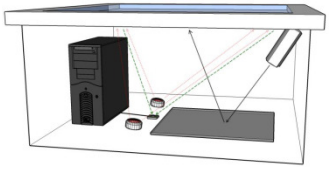
Informatik-Studium an der Freien Universität Berlin

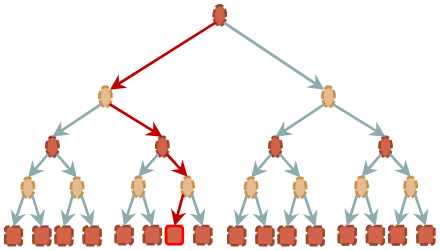
Freie Universität Berlin

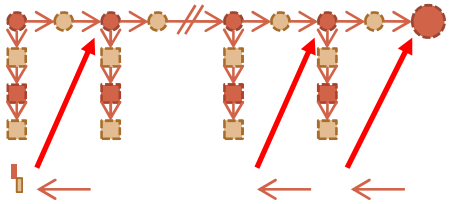
Künstliche Intelligenz in der Spieleprogrammierung

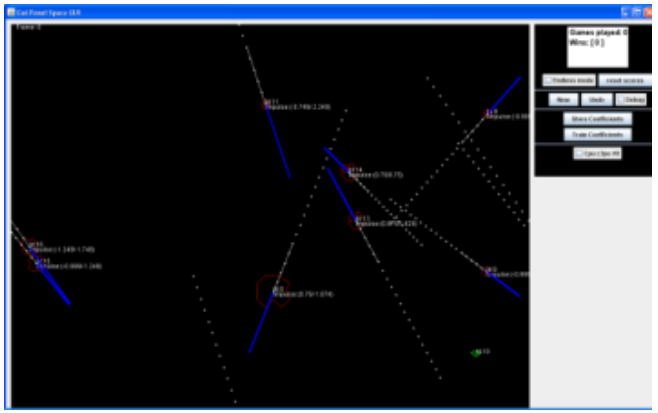
Schachprogrammierung, Brettspiele, Siedler von Catan, Poker, Asteroids, ...
http://www.java-uni.de/





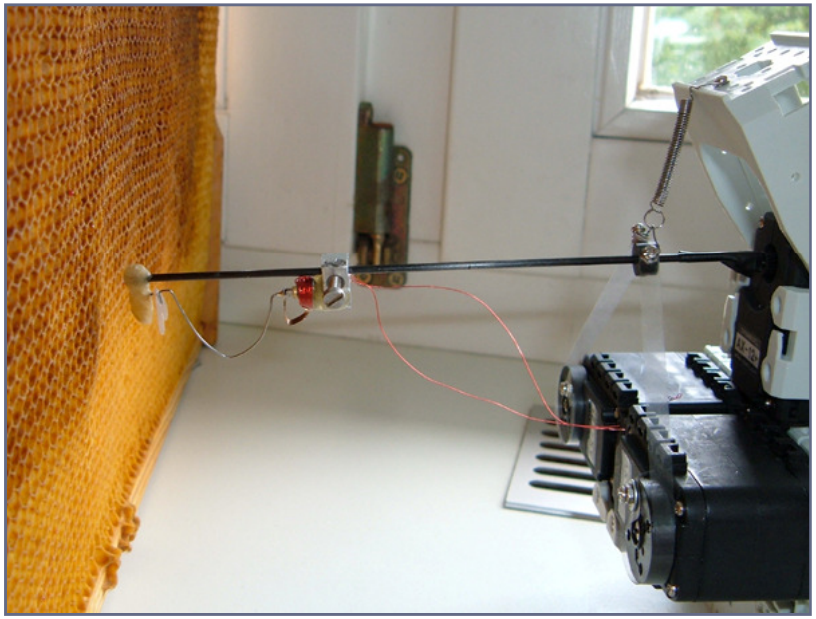


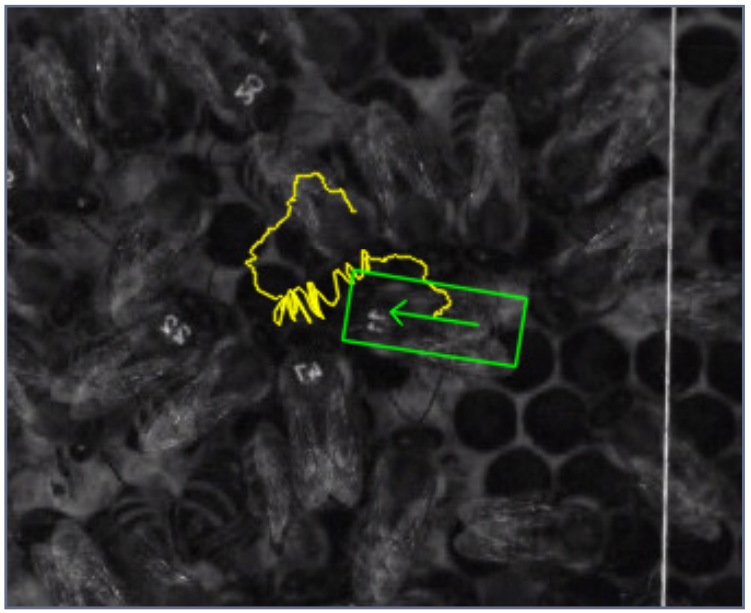




Robobiene

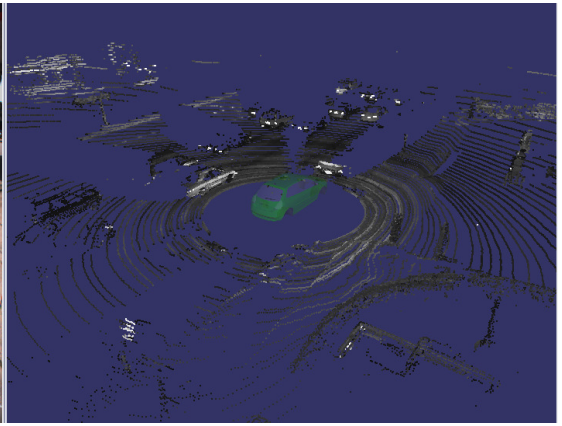
Bienen tracken und Forschung an einer simulierten Roboterbiene.
http://www.robobee.eu





Autonome Fahrzeuge

SpiritOfBerlin fährt ohne Fahrer.
http://robotics.mi.fu-berlin.de/pmwiki/pmwiki.php


Freie Universität Berlin

Kleine Auswahl von Projekten am Fachbereich:

▪ Sensornetzforschung	Norman Dziengel (dziengel@inf.fu-berlin.de)
▪ Formelerkennung	Ernesto Tapia (tapia@inf.fu-berlin.de)
▪ Elektronische Kreide	Christian Zick (zick@inf.fu-berlin.de)
▪ Künstliche Intelligenz in Spielen	Marco Block (block@inf.fu-berlin.de)
▪ Robobiene	Tim Landgraf (langraf@inf.fu-berlin.de)
▪ Autonome Fahrzeuge	Fabian Wiesel (wiesel@inf.fu-berlin.de)

Ansprechpersonen

Einstieg in die Programmierung mit Java

Inhalt:

- Primitive und zusammengesetzte Datentypen
- Methoden (Prozeduren und Funktionen)

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007

Installation und Vorbereitung

1. Download von der Sun-Seite und Installation (Java 1.6)
   http://java.sun.com/

2. Entwicklungsumgebung für Java
   - NotePad++ (http://notepad-plus.sourceforge.net/de/site.htm)
   - **Eclipse**, NetBeans, JCreator, JBuilder
   - (... es gibt eine sehr große Auswahl)

verwenden wir am Anfang der Vorlesung

3. Testen des Javasystems
   - In der Konsole prüfen, ob der Compiler vorhanden ist

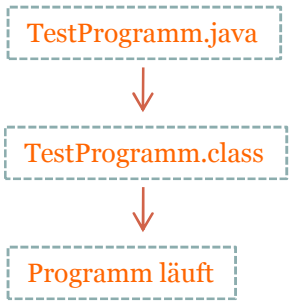
```
c:\>javac
```

   - Ein Testprogramm im Editor schreiben und mit dem Namen TestProgramm.java speichern

```
public class TestProgramm{}
```

   - Compilieren und Ausführen des Testprogramms

```
c:\>javac  TestProgramm.java

c:\>java TestProgramm
```



Primitive Datentypen und ihre Wertebereiche

Wahrheitswerte
- boolean

Zeichen, Symbole
- char

Zahlen
- byte, short, int, long, float, double

Datentyp	Wertebereich	BIT
boolean	true, false	8
char	0 bis 65.535	16
byte	-128 bis 127	8
short	32.768 bis 32.767	16
int	-2.147.483.648 bis 2.147.483.647	32
long	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807	64
float	+/- 1,4E-45 bis +/- 3,4E+38	32
double	+/- 4,9E-324 bis +/-1,7E+308	64

Variablen und Konstanten I

Deklaration von Variablen:

<Datentyp> <Name>;

Beispiel:

```
boolean a;
```

Zuweisung von Werten:

<Datentyp> <Name> = <Wert>;

Beispiel:

```
int b;
b = 7;
boolean a = true;
char c, d, e;
```

Sprechende Bezeichner verwenden:

```
boolean istFertig;
double kursWert;
int schrittZaehler;
```

Variablen und Konstanten II

Beschränkungen für Variablenbezeichnungen:

Variablen beginnen mit einem Buchstaben, Unterstrich oder Dollarzeichen (nicht erlaubt sind dabei Zahlen). Nach dem ersten Zeichen dürfen aber sowohl Buchstaben als auch Zahlen folgen. Operatoren und Schlüsselwörter dürfen nicht als Variablennamen verwendet werden.

Reservierte Schlüsselwörter:

abstract, assert, boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, enum, extends, false, final, finally, float, for, future, generic, goto, if, implements, import, inner, instanceof, int, interface, long, native, new, null, operator, outer, package, private, protected, public, rest, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, true, try, var, void, volatile, while

Hinweis:

Java ist textsensitiv, was bedeutet, dass auf Groß- und Kleinschreibung geachtet werden muss.

Beispiel:

```
boolean istFertig;
istFERTIG = true;
```

Variablen und Konstanten III

Konstanten:

Variablen, die während des Prorgammablaufs unverändert bleiben sollen, deklariert man als Konstanten.

Beispiel **pi** als Variable:

```
double pi = 3.14159;
```

Deklaration von Konstanten:

final <Datentyp> <NAME>;

Konvention: Großbuchstaben verwenden, für bessere Lesbarkeit des Programms

Beispiel **PI** als Konstante

```
final double PI = 3.14159;
```

Primitive Datentypen und Ihre Operationen I

boolean:

Bezeichnet einen Wahrheitswert.

```
boolean b;
b = false;
```

Das logische UND:

Die Operation UND wird mit dem Symbol && geschrieben.

B1	B2	B1 UND B2
0	0	0
0	1	0
1	0	0
1	1	1

Beispiel:

```
boolean a, b, c;
a = true;
b = false;
c = a && b;
```

Welchen Wert trägt c?

Primitive Datentypen und Ihre Operationen II

Das logische ODER:

Die Operation ODER wird mit dem Symbol || geschrieben.

B1	B2	B1 ODER B2
0	0	0
0	1	1
1	0	1
1	1	1

Beispiel:

```
boolean a, b, c;
a = true;
b = a && a;
c = b || a;
```

Welchen Wert trägt c?

Primitive Datentypen und Ihre Operationen III

Das logische NICHT:

Die Operation NICHT wird mit dem Symbol ! geschrieben.

B1	NICHT B1
0	1
1	0

Beispiel:

```
boolean a=true, b=false, c, d;
c = a && b;
d = (a||b) && !c
```

Welchen Wert trägt d?

Primitive Datentypen und Ihre Operationen IV

char:

Bezeichnet ein Zeichen oder Symbol.

```
char d =`7`;
char e = `b`;
```

Relationale Operatoren (Vergleichsoperatoren):

Es existieren folgende Vergleichsoperatoren: == (gleich), != (ungleich), <, >, <= und >=. Vergleichsoperatoren liefern einen boolean.

```
boolean d, e, f, g;
char a, b, c;

a = `1`;
b = `1`;
c = `5`;

d = a == b;
e = a !=b;
f = a < c;
g = c >= b;
```

Primitive Datentypen und Ihre Operationen V

int.:

Der ganzzahlige Datentyp int wird wahrscheinlich von allen am häufigsten verwendet.

byte, short, long sind äquivalent zu verwenden

```
int a, b = 0;
a = 10;
```

Wertebereich:

$$\{-2^{31}, -2^{31}+1, \ldots, -1, 0, 1, 2, \ldots, 2^{31}-1\}$$

Der kleinste und größte darstellbare Wert existiert als Konstante

```
int minimalerWert = Integer.MIN_VALUE;
int maximalerWert = Integer.MAX_VALUE;
```

Was passiert, wenn man diesen Wertebereich verlässt (Overflow)? Beispielsweise mit:

$2^{31}-1+2$ oder 2147483647+2 oder Integer.MAX_VALUE+2

Achtung Overflow!

Wir erwarten als Ergebnis den Wert 2147483649 erhalten aber stattdessen −2147483647.

Antwort: Der Wert landet wieder im int-Bereich! Die interne Darstellung ist zyklisch. Also aufpassen, ein unerkannter Overflow könnte zu falschen Ergebnissen führen.

Primitive Datentypen und Ihre Operationen V

int - Operationen:

Zahlen lassen sich z.B. addieren, subtrahieren, multiplizieren und dividieren. Es gibt aber zwei verschiedene Divisionen.

a) Ganzahliges Teilen ohne Rest (DIV, symbolisiert durch ‚/')

b) Den Rest aus der ganzzahligen Teilung ermitteln (MOD, symbolisiert durch ‚%')

```
int a = 29, b, c;
b = a/10;
c = a%10;

int d=0, e=1;
d = d + e;
e = e - 5;
d = d + 1;
```

a=ganz * b + rest
a/b = ganz
a%b = rest

Kurzschreibweise für Operationen:

```
int d=0, e=1;
d += e;
e -= 5;
d += 1;
d++;
```

Primitive Datentypen und Ihre Operationen V

float, double:

Repräsentieren gebrochene Zahlen oder Gleitkommazahlen.

```
5.      4.3      .00000001      -2.87
```

Java verwendet eine wissenschaftliche Darstellung für Gleitkommazahlen, um längere Zahlen besser lesen zu können

```
1E-8
```

E steht für **Exponent** und die nachfolgende Dezimalzahl gibt an, um wieviele Stellen der Dezimalpunkt verschoben werden muss. Für unser Beispiel ergibt sich eine Verschiebung nach links um **8 Stellen**, also:

$1E-8 = 1*10^{-8} = 0.00000001.$

Double und float können nicht alle realen Werte annehmen und müssen deshalb als **Näherungen** verstanden werden. Beispielsweise wollen wir folgende Berechnung vornehmen:

```
1/3 = 0.333333...
```
(usw.)

Als Typ double `1/3 = 0.3333333333333333`

Schauen wir uns die Darstellung eines double genauer an:

```
1.56E15
```

Einen Teil kennen wir bereits, der Teil hinter dem E ist der Exponent. Der Teil vor dem E ist die **Mantisse**. Eine Mantisse kann auf **16 Zeichen** genau angegeben werden. Ein Wert, z.B. `3.14159265358979324` wird gerundet zu `3.141592653589793`.

Primitive Datentypen und Ihre Operationen V

float, double - Operationen:

Die möglichen Operationen sind die gleichen, wie sie bei int vorgestellt wurden, lediglich die beiden Teilungsoperatoren verhalten sich anders.

```
double a = 2;
double b = 2.2;
float c  = 3;
float d  = 3.3f;

float e, f;
e = d%c;
e = d/c;

f = c*(d/2.4f)+1;
f += 1.3f;
```

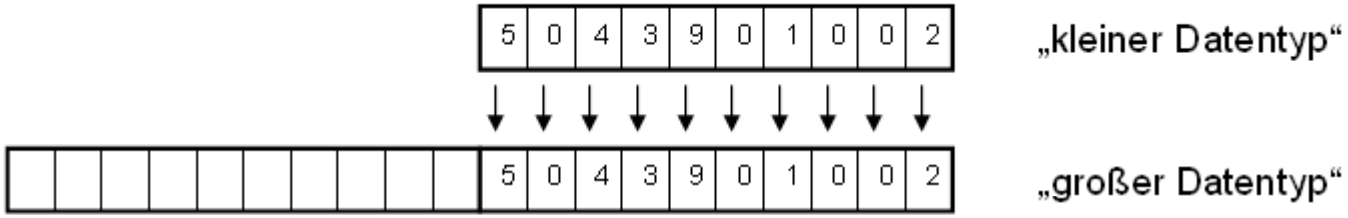
e=3.3%3 liefert 0.29999995 statt 0.3 (Rundungsfehler)

Casting und Typumwandlungen I

Zwei unterscheidbare Fälle:

Problemlos

Fall 1) Kleiner Datentyp wird in einen größeren geschrieben.



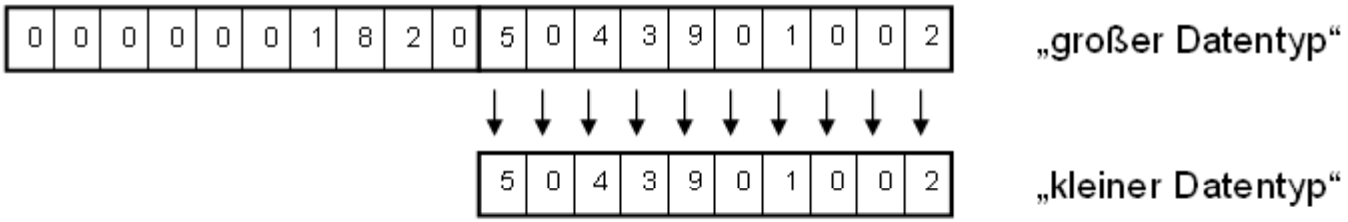

```
byte a = 28;
int b;
b = a;

float f = 2.3f;
double d;
d = f;
```

Casting und Typumwandlungen II

Zwei unterscheidbare Fälle:

Fall 2) Großer Datentyp wird in einen kleineren geschrieben.

Datenverlust mögich!




```
float f;
double d = 2.3;
f = d;
```

```
float f;
double d = 2.3;
f = (float)d;
```

Casting und Typumwandlungen III

Implizite Typumwandlungen:

byte < short < int < long
float < double

Datentypen sind für Operation entscheidend:

```
int a=5, b=7;
double erg = a/b;
System.out.println("Ergebnis (double) von "+a+"/"+b+" = "+erg);
```

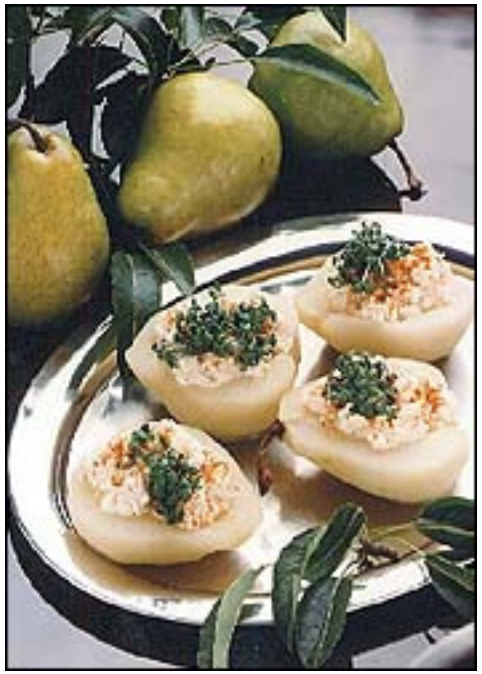
```
c:\>java Berechnung
Ergebnis (double) von 5/7 = 0.0
```

Der DIV-Operator für zwei int-Werte liefert nur den ganzzahligen Wert ohne Rest zurück. Sollte einer der beiden Operatoren ein double sein, so wird die Funktion ausgeführt, die auf mehrere Stellen nach dem Komme den richtigen Wert für die Division berechnet.

```
int a=5, b=7;
double erg = (double)a/b;
System.out.println("Ergebnis (double) von "+a+"/"+b+" = "+erg);
```

```
c:\>java Berechnung
Ergebnis (double) von 5/7 = 0.7142857142857143
```

Methoden der Programmerstellung



Rezept: „Birnen mit Käse gefüllt“

Zutatenliste:

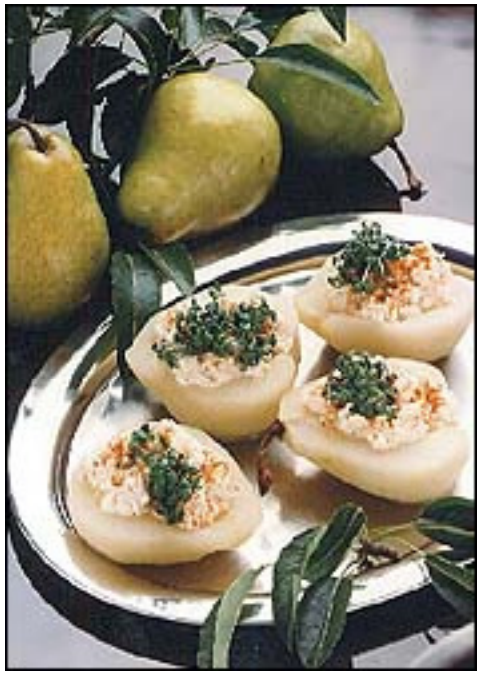
Menge	Maß	Zutat
2	Stück	Birnen
½	Stück	ausgepresste Zitrone
200	g	Schafskäse
3	EL	Sahne
1	Prise	Salz
1	Prise	Paprika
2	EL	Kresse

Birnen waschen, halbieren, Kerngehäuse entfernen und in wenig Wasser mit Zitronensaft garen.

Inzwischen Ziegen- oder Schafskäse zerdrücken und mit Sahne verrühren, mit Salz und Paprika würzen. Käse in die Birnenhälften füllen und mit Kresse garnieren.

Variation: Roquefort mit Dosenmilch verrühren und mit etwas Kirschwasser abschmecken. In die Birnenhälften einfüllen. Mit Salatblättern, Kräutern oder Maraschinokirschen garnieren.

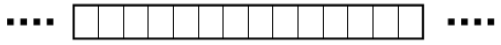
Methoden der Programmerstellung I



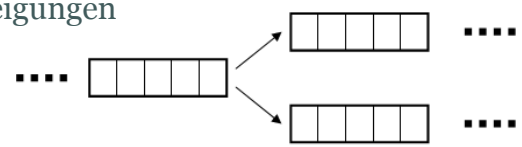
```
1.  Wasche die Birnen
2.  -> falls die Birnen noch nicht sauber sind, gehe zu 1

3.  Halbiere die Birnen und entferne die Kerngehäuse
4.  Gare die Birnen mit wenig Wasser und Zitronensaft in einem Topf
5.  wenn Variante 1 gewünscht gehe zu 6 ansonsten zu 13

6.  Variante 1:
7.     Zerdrücke den Schafskäse und verrühre ihn mit Sahne
8.     Würze die Käsemasse mit Salz und Paprika
9.        -> schmecke die Masse ab , falls Salz oder Paprika fehlen gehe zu 8
10.    Fülle die Käsemasse in die fertiggegarten Birnenhälften
11.    Garniere die Birnenhälften mit Kresse
12.    FERTIG

13. Variante 2:
14.    Verrühre Roquefort mit Dosenmilch und etwas Kirschwasser
15.    Fülle die Masse in die fertiggegarten Birnenhälften
16.    Garniere alles mit Salatblättern, Kräutern oder Maraschinokirschen
17.    FERTIG
```

Methoden der Programmerstellung II

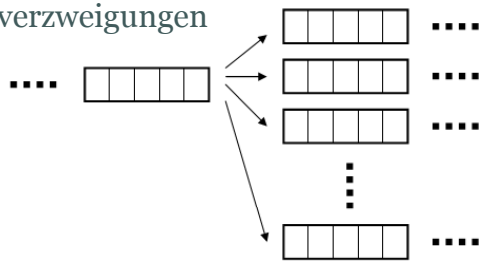
Sequentieller Ablauf



Verzweigungen



Mehrfachverzweigungen

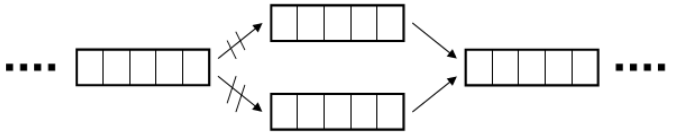


Sprünge

Schleifen

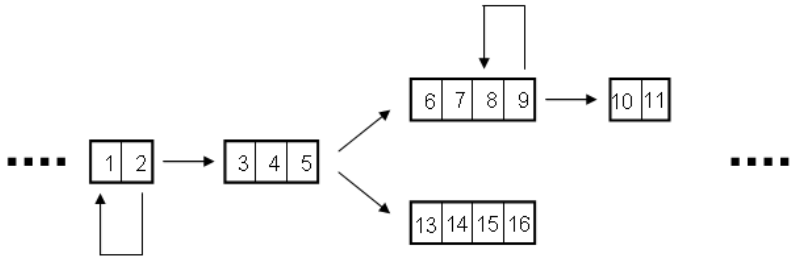
Mehrfachschleifen

Parallelität



Methoden der Programmerstellung II

Birnenrezept als Kombination




```
1.  Wasche die Birnen
2.  -> falls die Birnen noch nicht sauber sind, gehe zu 1

3.  Halbiere die Birnen und entferne die Kerngehäuse
4.  Gare die Birnen mit wenig Wasser und Zitronensaft in einem Topf
5.  wenn Variante 1 gewünscht gehe zu 6 ansonsten zu 13

6.  Variante 1:
7.     Zerdrücke den Schafskäse und verrühre ihn mit Sahne
8.     Würze die Käsemasse mit Salz und Paprika
9.        -> schmecke die Masse ab , falls Salz oder Paprika fehlen gehe zu 8
10.    Fülle die Käsemasse in die fertiggegarten Birnenhälften
11.    Garniere die Birnenhälften mit Kresse
12.    FERTIG

13. Variante 2:
14.    Verrühre Roquefort mit Dosenmilch und etwas Kirschwasser
15.    Fülle die Masse in die fertiggegarten Birnenhälften
16.    Garniere alles mit Salatblättern, Kräutern oder Maraschinokirschen
17.    FERTIG
```

Programme in Java

Einfaches Programm ProgrammEins.java

```
public class ProgramEins{
   public static void main(String[] args){
      System.out.println(„Endlich ist es soweit! Mein erstes Programm
                          läuft…“);
   }
}
```

```
c:\>javac ProgrammEins.java

c:\>java ProgrammEins
Endlich ist es soweit! Mein erstes Programm lõuft...
```

Probleme mit deutschem Zeichensatz

Programmieren mit einem einfachen Klassenkonzept

Inhalt:

- Programme in Java
- Kommentare
- Sequentielle Anweisungen
- Verzweigungen
- Schleifentypen
- Sprunganweisungen
- Funktionen in Java

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007

Programmieren mit einem einfachen Klassenkonzept

Programm als Klasse

Jedes Programm sehen wir erstmal als eine Klasse an. MeinErstesProgramm zeigt das Gerüst für ein Programm in Java.

Name des Programms mit Großbuchstaben

```
public class MeinErstesProgramm{
   public static void main(String[] args){
      //HIER KOMMEN DIE ANWEISUNGEN DES PROGRAMMS HIN
   }
}
```

An diese Stelle schreiben wir das Programm.

Kommentare

Verschiedene Möglichkeiten, um Programme zu kommentieren.

```
// Ich bin ein hilfreicher Kommentar in einer Zeile

/* Falls ich mal Kommentare über mehrere Zeilen
   hinweg schreiben möchte, so kann ich das
   mit diesem Kommentarsymbol tun
*/

public class Kommentierung{      // ich kann auch hier stehen
   public static void main(String[] args){

      /* An diese Stelle schreiben wir die
         Programmanweisungen */

   }
}
```

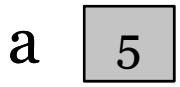
Sequentielle Anweisungen

Programm Sequentiell.java mit sequentiellen Anweisungen

```
public class Sequentiell{
   public static void main(String[] args){
      int a=5;     // Anweisung 1
      a=a*2;       // Anweisung 2
      a=a+10;      // Anweisung 3
      a=a-5;       // Anweisung 4
   }
}
```

Nach einer Anweisung steht ein „;“.

Sequentielle Anweisungen werden der Reihenfolge nach von oben nach unten verarbeitet.



Programmieren mit einem einfachen Klassenkonzept

Sequentielle Anweisungen

Ausgabe am Ende der Berechnung zur Überprüfung.

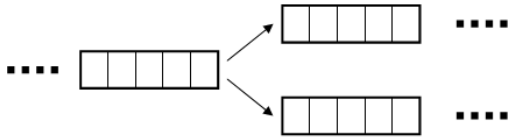
```
public class Sequentiell{
   public static void main(String[] args){
      int a=5;     // Anweisung 1
      a=a*2;       // Anweisung 2
      a=a+10;      // Anweisung 3
      a=a-5;       // Anweisung 4

      System.out.println("a hat den Wert: "+a);
   }
}
```

Anweisung zur Ausgabe im Konsolenfenster

Nach Ausführung des Programms erhalten wir folgende Ausgabe:

```
C:\>javac Sequentiell.java

C:\>java Sequentiell
a hat den Wert: 15
```

Verzweigungen



Wenn eine Bedingung erfüllt ist, dann führe eine einzelne Anweisung aus:

if (<Bedingung>) <Anweisung>;

```
if (x<0) x = -x;
```

Es können auch mehrere Anweisungen ausgeführt werden:

```
if (<Bedingung>) {
   <Anweisung_1>;
   <Anweisung_2>;
    ...
   <Anweisung_n>;
}
```

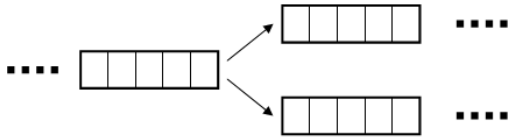
Anweisungsblock in { }

Erweiterung zu „wenn-dann-ansonsten“

if (<Bedingung>) <Anweisung_1>;
else <Anweisung_2>;

```
if (x<y)
   z = x;
else
   z = y;
```

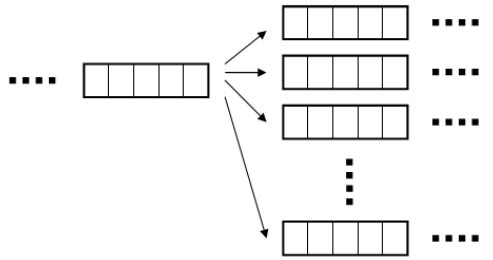
Verzweigungen II



if-Verzweigungen lassen sich verschachteln:

```
if (<Bedingung_1>)
    <Anweisung_1>;
else  if (<Bedingung_2>)
    <Anweisung_2>;
else  if (<Bedingung_3>)
    <Anweisung_3>;
else  <Anweisung_4>;
```

Mehrfachverzweigungen lassen sich mit **switch** realisieren



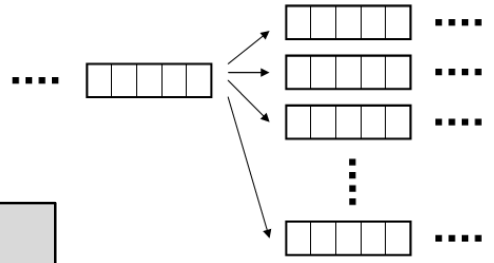
```
switch (<Ausdruck>) {
   case <Konstante1>:
       <Anweisung1>;
   break;

   case <Konstante2>:
       <Anweisung2>;
   break;

   default:
       <Anweisung3>;
}
```

Beschränkungen für Bedingungen z.B. „a==4“.
Im Ausdruck lassen sich nur primitive Datentypen (char, bytem short, int) auf Inhalt überprüfen.

Programmieren mit einem einfachen Klassenkonzept

Verzweigungen III

Die Zahlen von 0 bis 4 werden in die switch-Verzweigung gegeben.



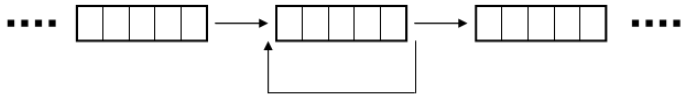
Beispiel für **switch**-Verzweigung:

```
for (int i=0; i<5; i++){
   switch(i){
      case 0:
         System.out.println("0");
         break;
      case 1:
      case 2:
         System.out.println("1 oder 2");
         break;
      case 3:
         System.out.println("3");
         break;
      default:
         System.out.println("hier landen alle anderen...");
         break;
   }
}
```

Als Ausgabe für i=0,...,4 erhalten wir:

```
C:\Java>java Verzweigung
0
1 oder 2
1 oder 2
3
hier landen alle anderen...
```

Verschiedene Schleifentypen



Warum Schleifen?

```
System.out.println("1 zum Quadrat ist "+(1*1));
System.out.println("2 zum Quadrat ist "+(2*2));
System.out.println("3 zum Quadrat ist "+(3*3));
System.out.println("4 zum Quadrat ist "+(4*4));
System.out.println("5 zum Quadrat ist "+(5*5));
System.out.println("6 zum Quadrat ist "+(6*6));
```

Eine **for**-Schleife initialisiert zu Beginn eine Variable und führt den folgenden Anweisungsblock solange aus, erhöht oder verringert dabei die Schleifenvariable, bis die Bedingung nicht mehr erfüllt ist:

```
for (<Startwert>; <Bedingung>; <Schrittweite>)
    <Anweisung>;
```

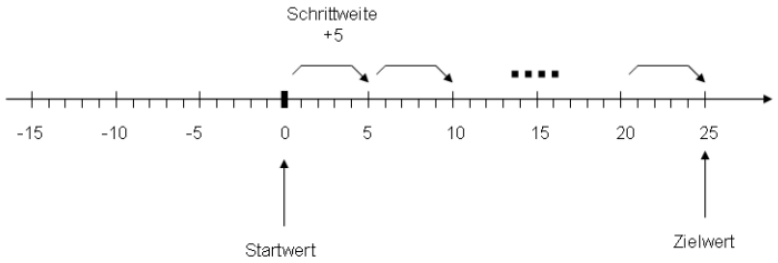
Ausgabe der quadrierten Werte:

```
for (int i=1; i<=1000; i=i+1){
   System.out.println(i+" zum Quadrat ist "+(i*i));
}
```

Verschiedene Schleifentypen II

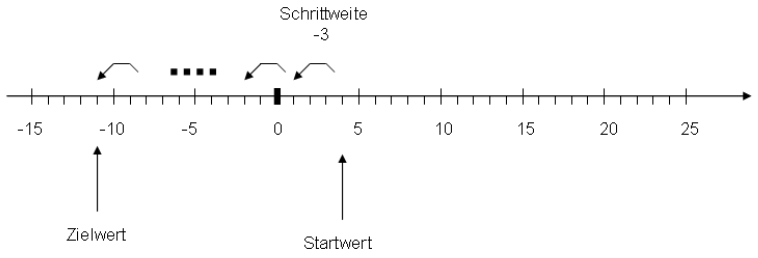
Beispiel 1:

```
for (int i=0; i<=25; i=i+5){
   System.out.println("Aktueller Wert für i ist "+i);
}
```

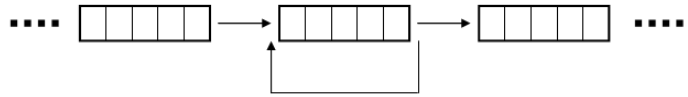


Beispiel 2:

```
for (int i=4; i>=-11; i=i-3){
   System.out.println("Aktueller Wert für i ist "+i);
}
```



Verschiedene Schleifentypen III



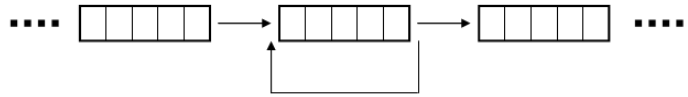
While-Schleifen, wenn nicht bekannt ist, wieviele Durchläufe benötigt werden. Führe die Anweisungen solange aus, wie die Bedingung wahr ist:

```
while (<Bedingung>)
    <Anweisung>;
```

Beispiel: Ausgabe der quadrierten Werte mit Hilfe der while-Schleife:

```
int i=1;
while (i<=1000){
   System.out.println(i+" zum Quadrat ist "+(i*i));
   i=i+1;
}
```

Verschiedene Schleifentypen IV



Do-While-Schleifen, wenn die Anweisungen mindestens einmal ausgeführt werden sollen.

```
do {
      <Anweisung>;
} while (<Bedingung>) ;
```

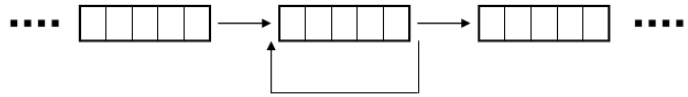
Beispiel für den Einsatz der do-while-Schleife:

```
int i=0;
do{
   i++;
   System.out.println("Wert von i: "+i);
} while (i<5);
```

Als Ausgabe erhalten wir:

```
C:\Java>java Schleifen
Wert von i: 1
Wert von i: 2
Wert von i: 3
Wert von i: 4
Wert von i: 5
```

Verschiedene Schleifentypen V



Wenn wir die Bedingung so ändern, dass sie von vornherein nicht erfüllt ist

```
int i=0;
do{
   i++;
   System.out.println("Wert von i: "+i);
} while (i<0);
```

Als Ausgabe erhalten wir:

```
C:\Java>java Schleifen
Wert von i: 1
```

Das war auch zu erwarten, da die Überprüfung der Bedingung erst nach der ersten Ausführung stattfindet.

Sprunganweisungen I

Mit **break** können wir Schleifen beenden.

```
int wert;
for (int i=0; i<=100; i++){
   wert = (i*i) + 2;
   System.out.println("Wert i="+i+", Funktionswert von i=" + wert);
   if (wert>100)
      break;
}
```

Als Ausgabe erhalten wir:

```
C:\Java>java Spruenge
Wert i=0, Funktionswert von i=2
Wert i=1, Funktionswert von i=3
Wert i=2, Funktionswert von i=6
Wert i=3, Funktionswert von i=11
Wert i=4, Funktionswert von i=18
Wert i=5, Funktionswert von i=27
Wert i=6, Funktionswert von i=38
Wert i=7, Funktionswert von i=51
Wert i=8, Funktionswert von i=66
Wert i=9, Funktionswert von i=83
Wert i=10, Funktionswert von i=102
```

Sprunganweisungen II

In Programmen lassen sich **Marken** unterbringen, die es ermöglichen auch aus mehreren Schleifen herauszuspringen.

<Marke>:

mit **break<Marke>** können wir an diese Stelle springen.

```
SprungZuI:              // Sprungmarke
for (int i=0; i<=2; i++){
   System.out.println("... jetzt sind wir hier bei i");
   SprungZuJ:           // Sprungmarke
   for (int j=0; j<=2; j++){
      System.out.println("... jetzt sind wir hier bei j");
      for (int k=0; k<=2; k++){
         System.out.println("... jetzt sind wir hier bei k");
         if (k==1)
            break SprungZuI;
      }
   }
}
System.out.println("hier sind wir...");
```

Sprunganweisungen III

Als Ausgabe erhalten wir:

```
C:\Java>java Spruenge
... jetzt sind wir hier bei i
... jetzt sind wir hier bei j
... jetzt sind wir hier bei k
... jetzt sind wir hier bei k
hier sind wir...
```

Jetzt ändern wir das Programm so, dass wir zu der Sprungmarke **SprungZuJ** springen.

```
C:\Java>java Spruenge
... jetzt sind wir hier bei i
... jetzt sind wir hier bei j
... jetzt sind wir hier bei k
... jetzt sind wir hier bei k
... jetzt sind wir hier bei i
... jetzt sind wir hier bei j
... jetzt sind wir hier bei k
... jetzt sind wir hier bei k
... jetzt sind wir hier bei i
... jetzt sind wir hier bei j
... jetzt sind wir hier bei k
... jetzt sind wir hier bei k
hier sind wir...
```

Sprunganweisungen IV

Die Anweisung **continue** beendet nicht die aktuelle innere Schleife wie , sondern springt zum Start der Schleife zurück, verändert entsprechend die Variable um die angegebene Schrittweite und setzt die Arbeit fort.

Beispiel:

```
public class Sprunganweisungen{
   public static void main(String[] args){
      for (int i=0; i<3; i++){
         System.out.println("Schleife i="+i+", Code 1");
         for (int j=0; j<3; j++){
            System.out.println("   Schleife j="+j+", Code 1");
            if (j==1){
               System.out.println("   continue-Anweisung");
               continue;
            }
            System.out.println("   Schleife j="+j+", Code 2");
         }
         if (i==1){
            System.out.println("continue-Anweisung");
            continue;
         }
         System.out.println("Schleife i="+i+", Code 2");
      }
   }
}
```

Sprunganweisungen IV

Als Ausgabe erhalten wir:

```
C:\Java>java Sprunganweisungen
Schleife i=0, Code 1
   Schleife j=0, Code 1
   Schleife j=0, Code 2
   Schleife j=1, Code 1
   continue-Anweisung
   Schleife j=2, Code 1
   Schleife j=2, Code 2
Schleife i=0, Code 2
Schleife i=1, Code 1
   Schleife j=0, Code 1
   Schleife j=0, Code 2
   Schleife j=1, Code 1
   continue-Anweisung
   Schleife j=2, Code 1
   Schleife j=2, Code 2
continue-Anweisung
Schleife i=2, Code 1
   Schleife j=0, Code 1
   Schleife j=0, Code 2
   Schleife j=1, Code 1
   continue-Anweisung
   Schleife j=2, Code 1
   Schleife j=2, Code 2
Schleife i=2, Code 2
```

Auch die Anweisung **continue** lässt sich mit einer Sprungmarke versehen.

Funktionen in Java (Motivation) I

Angenommen wir haben eine schöne Ausgabe für Zahlen programmiert und möchten nach jedem Berechnungsschritt diese Ausgabe ausführen. Momentan würden wir es noch so schreiben:

```
// Ausgabe.java
public class Ausgabe{
   public static void main(String[] args){
      int a=4;

      System.out.println();
      System.out.println("*****************************************");
      System.out.println("*** Wert der Variable a ist "+a);
      System.out.println("*****************************************");
      System.out.println();

      a=(a*13)%12;

      System.out.println();
      System.out.println("*****************************************");
      System.out.println("*** Wert der Variable a ist "+a);
      System.out.println("*****************************************");
      System.out.println();

      a+=1000;

      System.out.println();
      System.out.println("*****************************************");
      System.out.println("*** Wert der Variable a ist "+a);
      System.out.println("*****************************************");
      System.out.println();
   }
}
```

Funktionen in Java (Motivation) I

Um Redundanz zu vermeiden lagern wir diese Zeilen in eine Funktion aus. Aus der Mathematik wissen wir, dass Funktionen auch ein Ergebnis liefern. Falls, wie in unserem Fall, kein Rückgabewert existiert, dann schreiben wir als Rückgabewert das Schlüsselwort **void**.

```
public static <Rückgabewert> Funktionsname (Parameter) {
    // Funktionskörper
}
```

Beispiel:

```
public class AusgabeFunktion{
   public static void gebeAus(int a){     // neue Funktion
      System.out.println();
      System.out.println("*****************************************");
      System.out.println("*** Wert der Variable a ist "+a);
      System.out.println("*****************************************");
      System.out.println();
   }

   // main-Funktion
   public static void main(String[] args){
      int a=4;
      gebeAus(a);
      a=(a*13)%12;
      gebeAus(a);
      a+=1000;
      gebeAus(a);
   }
}
```

Daten laden und speichern

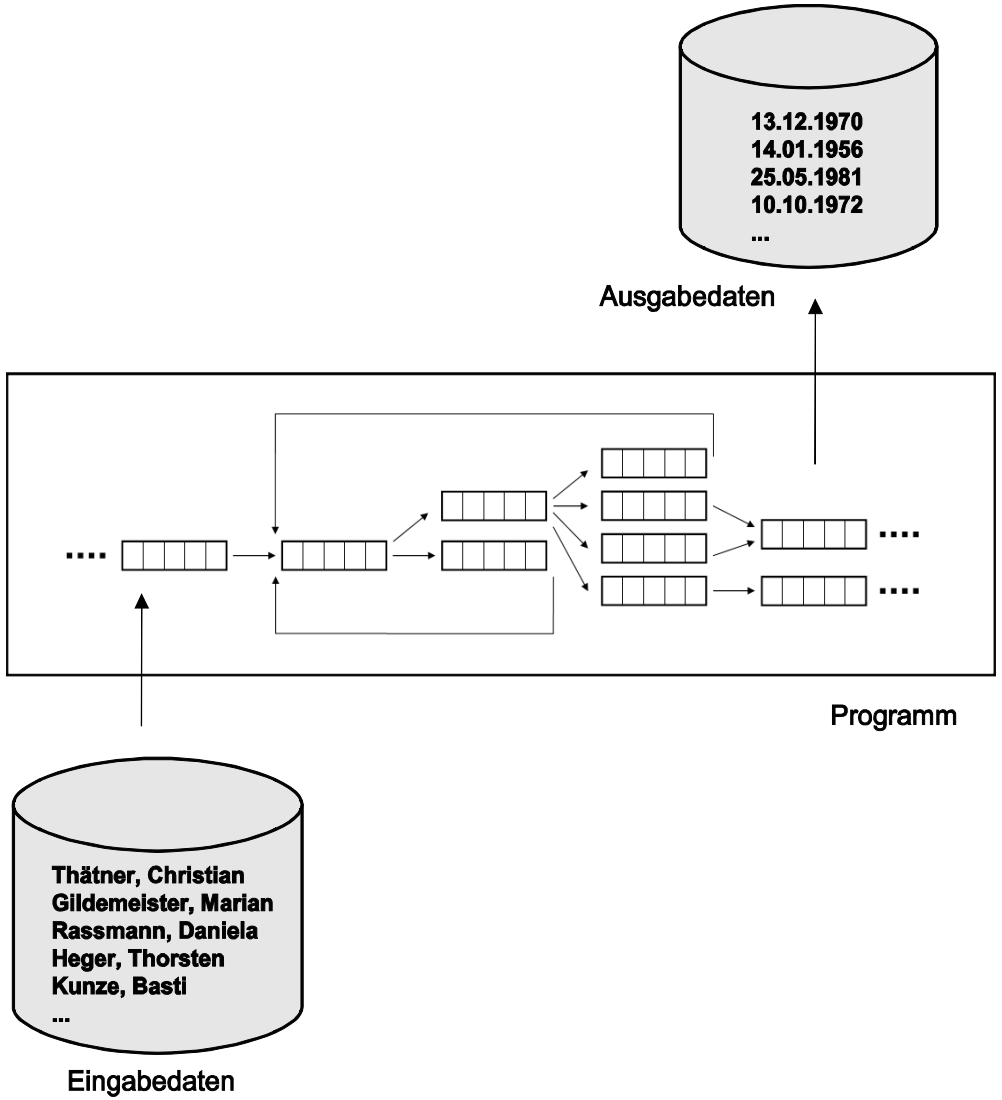
Inhalt:

- Externe Programmeingaben
- Daten laden und speichern
- Daten von der Konsole einlesen

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007

Daten laden und speichern

Datenbanken



Daten laden und speichern

Externe Programmeingaben I

Die einfachste Methode, einem Programm ein paar Daten mit auf dem Weg zu geben, ist die Übergabe von Parametern auf der Kommandozeile.

```
// MirIstWarm.java
public class MirIstWarm{
   public static void main(String[] args){
      System.out.println("Mir ist heute zu warm, ich mache nix :).");
   }
}
```

Mit der Ausgabe:

```
C:\>javac MirIstWarm.java
C:\>java MirIstWarm
Mir ist heute zu warm, ich mache nix :).
```

Externe Programmeingaben II

Die einfachste Methode, einem Programm ein paar Daten mit auf dem Weg zu geben, ist die Übergabe von Parametern auf der Kommandozeile:

```
// MeineEingaben.java
public class MeineEingaben{
   public static void main(String[] args){
      System.out.println("Eingabe 1: >"+args[0]+"< und");
      System.out.println("Eingabe 2: >"+args[1]+"<");
   }
}
```

Mit der Eingabe:

```
C:\>javac MeineEingaben.java
C:\>java MeineEingaben Hallo 27
Eingabe 1: >Hallo< und
Eingabe 2: >27<
```

In diesem Fall war es wichtig zu wissen, wie viele Eingaben wir erhalten haben. Sollten wir auf einen Eintrag in der Stringliste zugreifen, die keinen Wert erhalten hat, dann passiert folgendes:

```
C:\>java MeineEingaben Hallo
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 1
at MeineEingaben.main(MeineEingaben.java:5)
```

Externe Programmeingaben III

Dieser Fehler lässt sich vermeiden, wenn wir zuerst die Anzahl der übergebenen Elemente überprüfen.

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

Beispiel:

```
C:\>java MeineEingaben Hallo 27 und noch viel mehr!
Eingabe 0: >Hallo<
Eingabe 1: >27<
Eingabe 2: >und<
Eingabe 3: >noch<
Eingabe 4: >viel<
Eingabe 5: >mehr!<
```

Daten laden und speichern

Aktuelles Lernziel

Ein Lernziel an dieser Stelle wird sein, die im ersten Augenblick unverständlich erscheinenden Programmteile, einfach mal zu verwenden.

Wir müssen am Anfang einen Mittelweg finden zwischen dem absoluten Verständnis für jede Programmzeile und der Verwendung einer gegebenen Teillösung.

Daten laden und speichern

Daten aus einer Datei einlesen I

```
import java.io.*;
public class LiesDateiEin{
   public static void main(String[] args){
      // Dateiname wird übergeben
      String filenameIn = args[0];
      try{
         FileInputStream fis    = new FileInputStream(filenameIn);
         InputStreamReader isr  = new InputStreamReader(fis);
         BufferedReader bur     = new BufferedReader(isr);

         // die erste Zeile wird eingelesen
         String sLine = bur.readLine();

         // lies alle Zeilen aus, bis keine mehr vorhanden sind
         // und gib sie nacheinander aus
         // falls von vornherein nichts in der Datei enthalten
         // ist, wird dieser Programmabschnitt übersprungen
         int zaehler = 0;
         while (sLine != null) {
            System.out.println("Zeile "+zaehler+": "+sLine);
            sLine = bur.readLine();
            zaehler++;
         }
         // schließe die Datei
         bur.close();
      } catch (IOException eIO) {
         System.out.println("Folgender Fehler trat auf: "+eIO);
      }
   }
}
```

Daten aus einer Datei einlesen II

Verwenden könnten wir **LiesDateiEin.java**, z.B. mit der Datei **namen.dat**. Den Inhalt einer Datei kann man sich auf der Konsole mit dem Befehl **type** anschauen:

```
C:\>type namen.dat
Harald Liebchen
Gustav Peterson
Gunnar Heinze
Paul Freundlich

C:\>java LiesDateiEin namen.dat
Zeile 0: Harald Liebchen
Zeile 1: Gustav Peterson
Zeile 2: Gunnar Heinze
Zeile 3: Paul Freundlich
```

Unser Programm kann eine Datei zeilenweise auslesen und gibt das eingelesene gleich auf der Konsole aus.

Daten in eine Datei schreiben I

Um Daten in eine Datei zu speichern, schauen wir uns mal folgendes Programm an:

```
import java.io.*;
public class SchreibeInDatei{
   public static void main(String[] args){
      // Dateiname wird übergeben
      String filenameOutput = args[0];
      try{
         BufferedWriter myWriter =
            new BufferedWriter(new FileWriter(filenameOutput, false));

         // schreibe zeilenweise in die Datei filenameOutput
         myWriter.write("Hans Mueller\n");
         myWriter.write("Gundel Gaukel\n");
         myWriter.write("Fred Feuermacher\n");

         // schliesse die Datei
         myWriter.close();
      } catch (IOException eIO) {
         System.out.println("Folgender Fehler trat auf: "+eIO);
      }
   }
}
```

Jetzt testen wir unser Programm und verwenden zur Überprüfung die vorher besprochene Klasse **LiesDateiEin.java**.

Daten in eine Datei schreiben II

Ausgabe:

```
C:\>java SchreibeInDatei namen2.dat

C:\>java LiesDateiEin namen2.dat
Zeile 0: Hans Mueller
Zeile 1: Gundel Gaukel
Zeile 2: Fred Feuermacher
```

Wir stellen fest: Es hat funktioniert!

Daten laden und speichern

Daten von der Konsole einlesen

Wir sind nun in der Lage, Daten beim Programmstart mitzugeben und Dateien auszulesen, oft ist es aber wünschenswert eine Interaktion zwischen Benutzer und Programm zu haben. Beispielsweise soll der Benutzer eine Entscheidung treffen oder eine Eingabe machen.

Das ist mit der Klasse **BufferedReader** schnell realisiert:

```
import java.io.*;
public class Einlesen{
   public static void main(String[] args){
      System.out.print("Eingabe: ");
      try{
         InputStreamReader isr   = new InputStreamReader(System.in);
         BufferedReader bur      = new BufferedReader(isr);

         // Hier lesen wir einen String ein:
         String str = bur.readLine();

         // und geben ihn gleich wieder aus
         System.out.println(str);
      } catch(IOException e){}
   }
}
```

Beispiel:

```
C:\>java Einlesen
Eingabe: Ich gebe etwas ein 4,5 a 1/2
Ich gebe etwas ein 4,5 a 1/2
```

Einfache Datenstrukturen

Inhalt:

- Arrays
- Matrizen
- Conways „Game of Life“

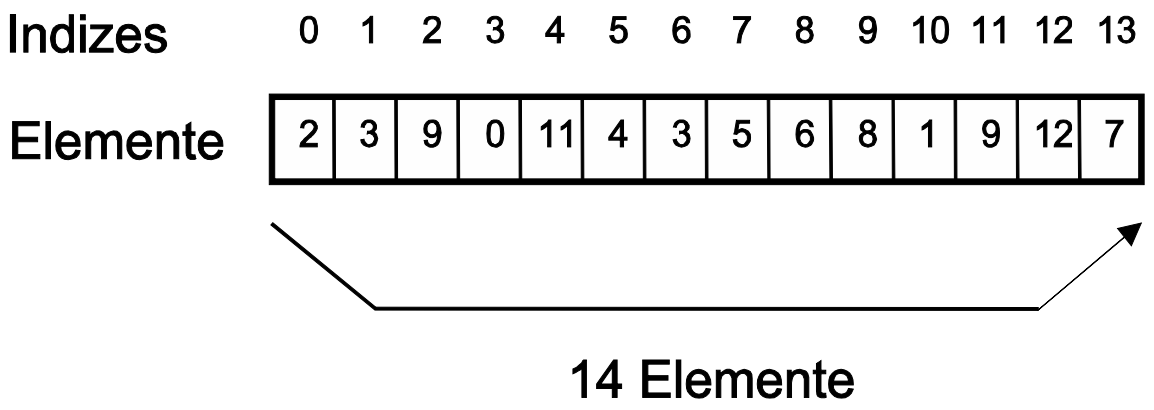
Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007

Listen von Datentypen: Arrays

Nehmen wir an, wir möchten nicht nur einen **int**, sondern viele davon verwalten. Dann könnten wir es, mit dem uns bereits bekannten Wissen, in etwa so bewerkstelligen:

```
int a, b, c, d, e, f;
a=0;
b=1;
c=2;
d=3;
e=4;
f=5;
```

Sehr aufwendig und unschön. Es gibt eine einfachere Möglichkeit mit dem **Array** (Liste).



Einfache Datenstrukturen

Erzeugung von Arrays

Erzeugen eines int-Arrays mit k Elementen:

<Datentyp>[] <name>;
<name> = new <Datentyp>[k];

Oder in einer Zeile:

<Datentyp>[] <name> = new <Datentyp>[k];

Zugriff auf die Elemente und Initialisierung der Variablen:

```
int[] a = new int[2];
a[0] = 3;
a[1] = 4;
```

a = [3, 4]

Sollten wir schon bei der Erzeugung des Arrays wissen, welchen Inhalt die Elemente haben sollen, dann können wir das so vornehmen („Literale Erzeugung“):

```
int[] a = {1,2,3,4,5};
```

a = [1, 2, 3, 4, 5]

Beliebte Fehlerquelle bei Arrays

Daran müssen wir uns gewöhnen und es ist eine beliebte **Fehlerquelle**. Es könnte sonst passieren, dass wir z.B. in einer Schleife alle Elemente durchlaufen möchten, auf das -te Element zugreifen und einen Fehler verursachen:

```
int[] a = new int[10];

for (int i=0; i<=10; i++)
   System.out.println("a["+i+"]="+a[i]);
```

Wir erhalten folgende Fehlermeldung:

```
C:\Java>java Array
a[0]=0
a[1]=0
a[2]=0
a[3]=0
a[4]=0
a[5]=0
a[6]=0
a[7]=0
a[8]=0
a[9]=0
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at Array.main(Array.java:5)
```

Wir erinnern uns:

Auch beim Einlesen von der Konsole gab es diese Fehlermeldung:

```
// MeineEingaben.java
public class MeineEingaben{
   public static void main(String[] args){
      System.out.println("Eingabe 1: >"+args[0]+"< und");
      System.out.println("Eingabe 2: >"+args[1]+"<");
   }
}
```

Mit der Eingabe:

```
C:\>java MeineEingaben Hallo
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 1
at MeineEingaben.main(MeineEingaben.java:5)
```

Vereinfachte for-Schleifennotation

Um Fehler zu vermeiden gibt es seit Java 1.5 die folgende vereinfachte for-Schleifennotation:

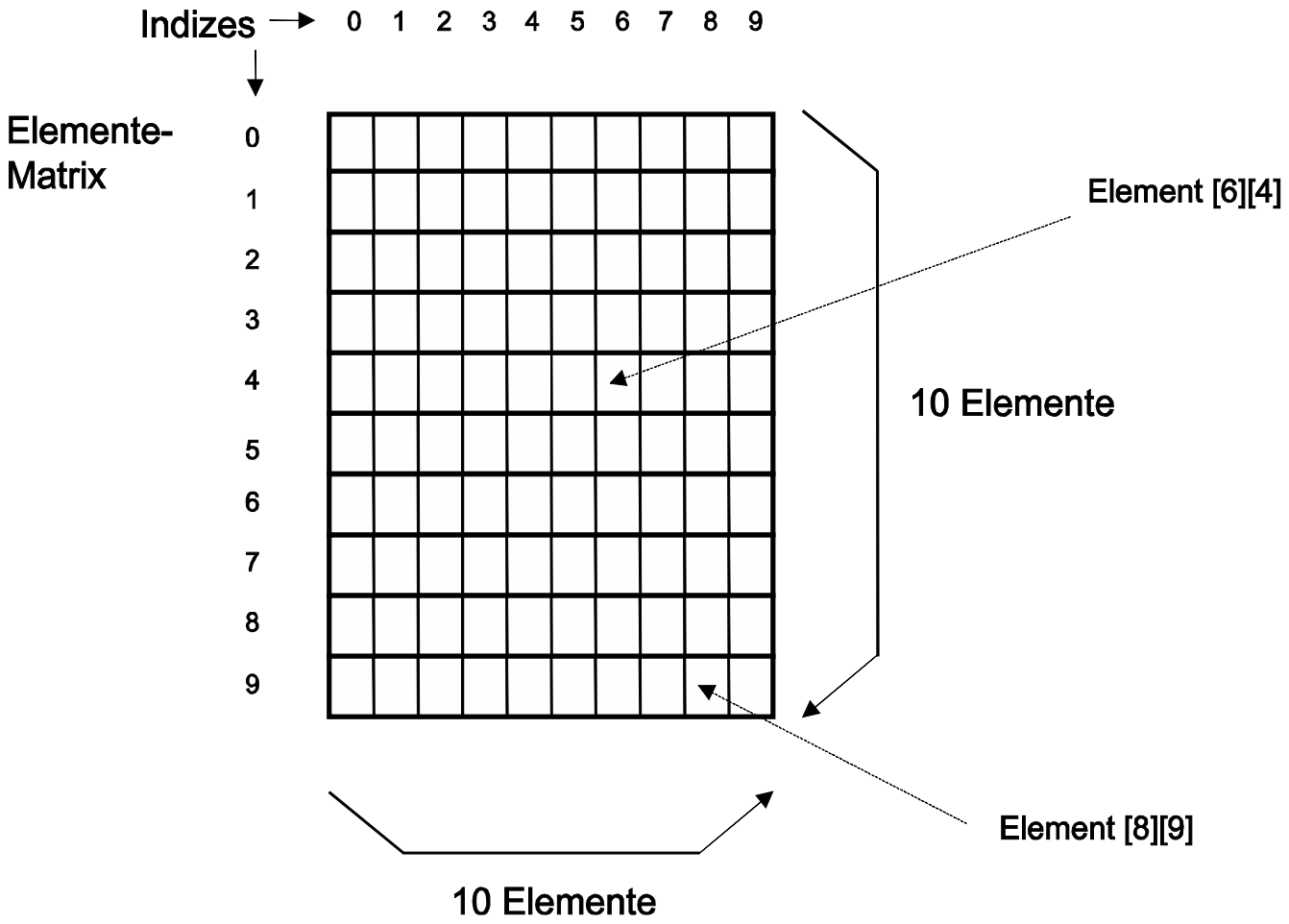
```
for (<Typ>  <Variablenname> : <Ausdruck>)
    <Anweisung>;
```

Lies: „Für jedes **x** aus der Liste **werte**“:

```
int[] werte = {1,2,3,4,5,6};    // Literale Erzeugung

// Berechnung der Summe
int summe = 0;
for (int x : werte)
   summe += x;
```

Hilft „IndexOutOfBoundsExceptions“ zu vermeiden.

Matrizen und multidimensionale Arrays



Matrizen und multidimensionale Arrays

Wir erzeugen Matrizen, indem wir zum Array eine Dimension dazu nehmen:

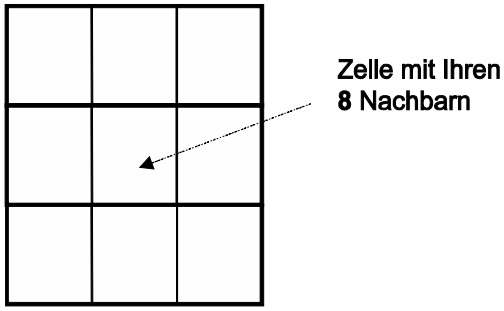
```
int[][] a = new int[n][m];
a[4][1] = 27;
```

Auf diese Weise können wir sogar noch mehr Dimensionen erzeugen:

```
int[][][][] a = new int[k][l][m][n];
```

Conways Game of Life I

Man stelle sich vor, die Welt bestünde nur aus einer 2-dimensionalen Matrix. Jeder Eintrag, wir nennen ihn jetzt mal Zelle oder Zellulärer Automat, kann zwei Zustände annehmen, er ist entweder lebendig oder tot. Jede Zelle interagiert mit ihren 8 Nachbarn.



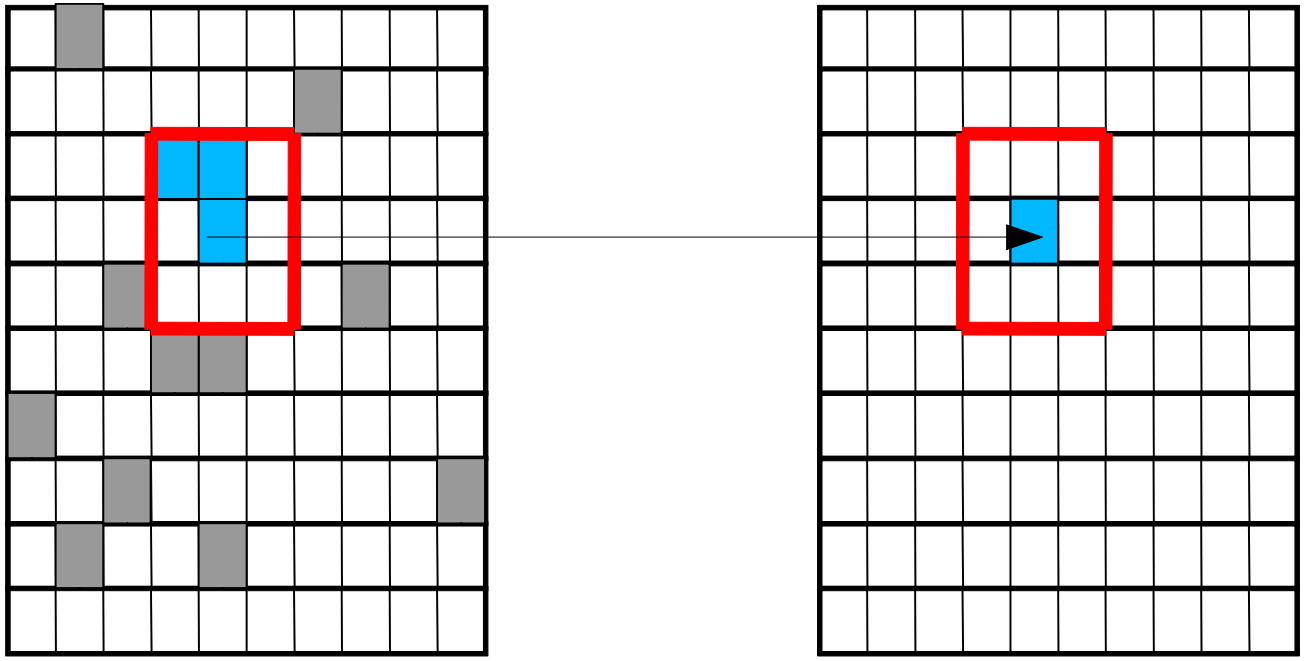
Diese Interaktion unterliegt den folgenden vier Regeln:

1. jede lebendige Zelle, die weniger als zwei lebendige Nachbarn hat, stirbt an Einsamkeit
2. jede lebendige Zelle mit mehr als drei lebendigen Nachbarn stirbt an Überbevölkerung
3. jede lebendige Zelle mit mit zwei oder drei Nachbarn fühlt sich wohl und lebt weiter
4. jede tote Zelle mit genau drei lebendigen Nachbarn wird wieder zum Leben erweckt.

Conways Game of Life II

Die zweite Generation wird durch die Anwendung der vier Regeln auf jede der Zellen erzeugt. Es wird geprüft, ob Zellen lebendig bleiben, sterben oder neu entstehen.

Einfache Implementierung:



Conways Game of Life III

Das Programm **GameOfLife.java** beinhaltet neben der main-Funktion zwei weitere Methoden.

Zum einen eine kleine Ausgabefunktion:

```
import java.util.Random; // erläutern wir später
public class GameOfLife{

   public static void gebeAus(boolean[][] m){
      // Ein "X" symbolisiert eine lebendige Zelle
      for (int i=0; i<10; i++){
         for (int j=0; j<10; j++){
            if (m[i][j])
               System.out.print("X ");
            else
               System.out.print("  ");
         }
         System.out.println();
      }
   }
```

Conways Game of Life IV

und zum anderen eine Funktion, die die Elemente der Nachbarschaften zählt. Dazu verwenden wir einen kleinen Trick:

```
// Wir nutzen hier die Tatsache aus, dass Java einen Fehler erzeugt,
// wenn wir auf ein Element außerhalb der Matrix zugreifen
public static int zaehleUmgebung(boolean[][] m, int x, int y){
   int ret = 0;
   for (int i=(x-1);i<(x+2);++i){
      for (int j=(y-1);j<(y+2);++j){
         try{
            if (m[i][j])
               ret += 1;
         }
         catch (IndexOutOfBoundsException e){}
      }
   }
   // einen zuviel mitgezaehlt?
   if (m[x][y])
      ret -= 1;

   return ret;
}
```

Conways Game of Life V

In der -Methode werden zwei Matrizen für zwei aufeinander folgende Generationen bereitgestellt. Exemplarisch werden die Zellkonstellationen einer Generation, gemäß den zuvor definierten Regeln, berechnet und ausgegeben.

```
public static void main(String[] args){
      // unsere Welt soll aus 10x10 Elemente bestehen
      boolean[][] welt     = new boolean[10][10];
      boolean[][] welt_neu = new boolean[10][10];

      // ***********************************************************
      // Erzeugt eine zufällige Konstellation von Einsen und Nullen
      // in der Matrix welt. Die Chancen liegen bei 50%, dass eine
      // Zelle lebendig ist.
      Random generator = new Random();
      double zufallswert;
      for (int i=0; i<10; i++){
         for (int j=0; j<10; j++){
            zufallswert = generator.nextDouble();
            if (zufallswert>=0.5)
               welt[i][j] = true;
         }
      }
      // ***********************************************************
```

Einfache Datenstrukturen

Conways Game of Life VI

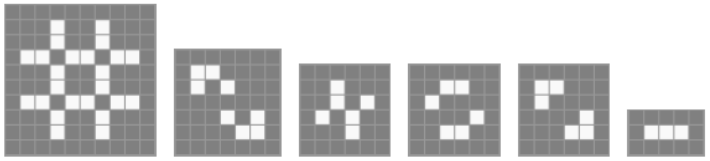
```
      // ********************************************************
      // Ausgabe der ersten Generation
      System.out.println("Generation 1");
      gebeAus(welt);

      int nachbarn;
      for (int i=0; i<10; i++){
         for (int j=0; j<10; j++){
            // Zaehle die Nachbarn
            nachbarn = zaehleUmgebung(welt, i, j);

           if (welt[i][j]){
               // Regel 1, 2:
               if ((nachbarn<2) || (nachbarn>3))
                  welt_neu[i][j] = false;

               // Regel 3:
               if ((nachbarn==2) || (nachbarn==3))
                  welt_neu[i][j] = true;
            }
            else {
               // Regel 4:
               if (nachbarn==3)
                  welt_neu[i][j] = true;
            }
         }
      }
      // Ausgabe der zweiten Generation
      System.out.println("Generation 2");
      gebeAus(welt_neu);
   }
}
```

Methode zur Ermittlung der Anzahl der Nachbarn wird aufgerufen.

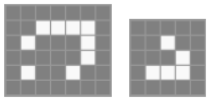
Auswahl besonderer Muster

Für den interessierten Leser ist hier eine kleine Sammlung besonderer Zellkonstellationen zusammengestellt:

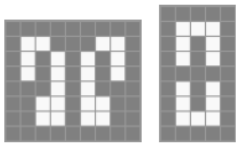
Zyklische Muster



Gleiter



Interessante Startkonstellationen



Debuggen und Fehlerbehandlungen

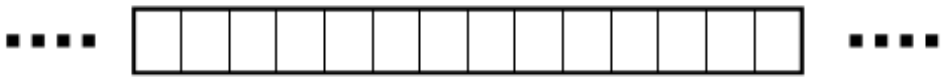
Inhalt:

- Das richtige Konzept
- Exceptions
- Zeilenweises Debuggen und Breakpoints
- Verifikation imperativer Programme (Hoare-Kalkül)

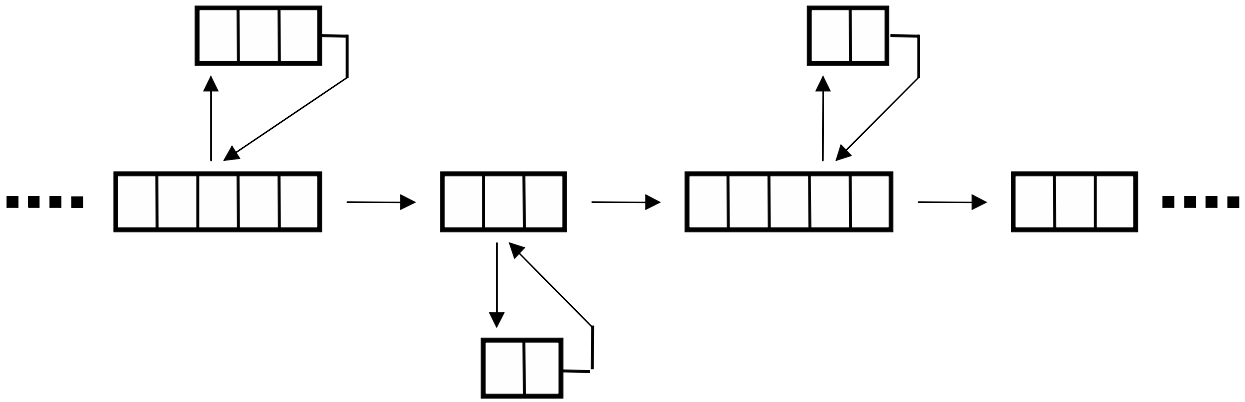
Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007

Das richtige Konzept

Es sollte bei der Entwicklung darauf geachtet werden, dass nicht allzu viele Programmzeilen in eine Funktion gehören.

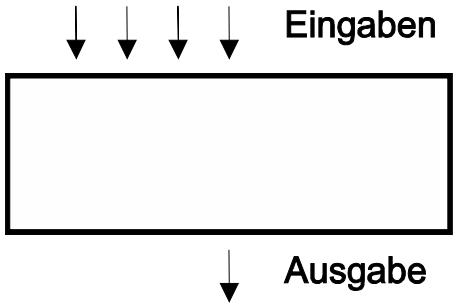


Besser ist es, das Programm zu gliedern und in Programmabschnitte zu unterteilen. Zum einen wird damit die Übersicht gefördert und zum anderen verspricht die Modularisierung den Vorteil, Fehler in kleineren Programmabschnitten besser aufzuspüren und vermeiden zu können.



Constraints I

Ein wichtiges Werkzeug der Modularisierung stellen sogenannte Constraints (Einschränkungen) dar. Beim Eintritt in einen Programmabschnitt (z.B. eine Funktion) müssen die Eingabeparameter überprüft und damit klargestellt werden, dass der Programmteil mit den gewünschten Daten arbeiten kann. Das gleiche gilt für die Ausgabe.



Constraints II

Kommentare sind hier unverzichtbar und fördern die eigene Vorstellung der Funktionsweise eines Programmabschnitts. Als Beispiel schauen wir uns mal die Fakultätsfunktion an:

```
public class Fakultaet{
   /*
     Fakultätsfunktion liefert für i=1 .. 19 die entsprechenden
     Funktionswerte i! = i*(i-1)*(i-2)*...*1

     Der Rückgabewert liegt im Bereich 1 .. 121645100408832000

     Sollte eine falsche Eingabe vorliegen, so liefert das Programm
     als Ergebnis -1.
   */
   public static long fakultaet(long i){
      // Ist der Wert ausserhalb des erlaubten Bereichs?
      if ((i<=0)||(i>19))
         return -1;

      // Rekursive Berechnung der Fakultaet
      if (i==1)
         return 1;
      else
         return i*fakultaet(i-1);
   }

   public static void main(String[] args){
      for (int i=0; i<15; i++)
         System.out.println("Fakultaet von "+i+" liefert "+fakultaet(i));
   }
}
```

Exceptions in Java I

Wenn ein Fehler während der Ausführung eines Programms auftritt, wird ein Objekt einer Fehlerklasse Exception) erzeugt. Da der Begriff Objekt erst später erläutert wird, stellen wir uns einfach vor, dass ein Programm gestartet wird, welches den Fehler analysiert und wenn der Fehler identifizierbar ist, können wir dieses Programm nach dem Fehler fragen und erhalten einen Hinweis, der Aufschluss über die Fehlerquelle gibt.

Schauen wir uns ein Beispiel an:

Die Zahl in einer Zeichenkette wird in einen Wert umgewandelt.

```
public class ExceptionTest{
   public static void main(String[] args){
      int d = Integer.parseInt(args[0]);
      int k = 10/d;
      System.out.println("Ergebnis ist "+k);
   }
}
```

Auf den ersten Blick ist kein Fehler erkennbar, aber ein Test zeigt schon die Fehleranfälligkeit des Programms.

Exceptions in Java II

Auf den ersten Blick ist kein Fehler erkennbar, aber ein Test zeigt schon die Fehleranfälligkeit des Programms:

```
C:\>java ExceptionTest 2
Ergebnis ist 5

C:\>java ExceptionTest 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionTest.main(ExceptionTest.java:5)

C:\>java ExceptionTest d
Exception in thread "main" java.lang.NumberFormatException: For input
string: "d"
       at java.lang.NumberFormatException.forInputString(Unknown Source)
       at java.lang.Integer.parseInt(Unknown Source)
       at java.lang.Integer.parseInt(Unknown Source)
       at ExceptionTest.main(ExceptionTest.java:4)
```

Zwei Fehlerquellen sind hier erkennbar, die Eingabe eines falschen Typs und die Eingabe einer 0, die bei der Division einen Fehler verursacht. Beides sind für Java wohlbekannte Fehler, daher gibt es auch in beiden Fällen entsprechende Fehlerbezeichnungen **NumberFormatException** und **ArithmeticException**.

Exceptions in Java III

Um nun Fehler dieser Art zu vermeiden, müssen wir zunächst auf die Fehlersuche gehen und den Abschnitt identifizieren, der den Fehler verursacht. Wir müssen uns dazu die Frage stellen: In welchen Situationen (unter welchen Bedingungen) stürzt das Programm ab? Damit können wir den Fehler einengen und den Abschnitt besser lokalisieren. Dann müssen wir Abhängigkeiten überprüfen und die Module identifizieren, die für die Eingabe in diesen Abschnitt zuständig sind.

Angenommen, wir haben einen Bereich lokalisiert und wollen diesen nun beobachten, dazu verwenden wir die **try-catch-Klausel**.

```
try {
   <Anweisung>;
   ...
   <Anweisung>;
} catch (Exception e) {
   <Anweisung>;
}
```

Die try-catch Behandlung lässt sich als *„Versuche dies, wenn ein Fehler dabei auftritt, mache das.“* lesen.

Exceptions in Java IV

Um unser Programm **ExceptionTest .java** vor einem Absturz zu bewahren, wenden wir diese Klausel an und testen das Programm:

```
public class ExceptionTest2{
   public static void main(String[] args){
      try{
         int d = Integer.parseInt(args[0]);
         int k = 10/d;
         System.out.println("Ergebnis ist "+k);
      } catch(Exception e){
         System.out.println("Fehler ist aufgetreten...");
      }
   }
}
```

Wie testen nun die gleichen Eingaben:

```
C:\>java ExceptionTest 2
Ergebnis ist 5

C:\>java ExceptionTest 0
Fehler ist aufgetreten...

C:\>java ExceptionTest d
Fehler ist aufgetreten...
```

Einen Teilerfolg haben wir nun schon zu verbuchen, da das Programm nicht mehr abstürzt. Der Bereich in den geschweiften Klammern nach dem Schlüsselwort **try** wird gesondert beobachtet.
Sollte ein Fehler auftreten, so wird die weitere Abarbeitung innerhalb dieser Klammern abgebrochen und der Block nach dem Schlüsselwort **catch** ausgeführt.

Exceptions in Java IV

Dummerweise können wir nun die Fehler nicht mehr eindeutig identifizieren, da sie die gleiche Fehlermeldung produzieren. Um die Fehler aber nun eindeutig abzufangen, lassen sich einfach mehrere catch-Blöcke mit verschiedenen Fehlertypen angeben:.

```
try {
      <Anweisung>;
      ...
      <Anweisung>;
   } catch(Exceptiontyp1 e1){
      <Anweisung>;
   } catch(Exceptiontyp2 e2){
      <Anweisung>;
   } catch(Exceptiontyp3 e3){
      <Anweisung>;
}
```

Exceptions in Java V

Wenden wir die neue Erkenntnis auf unser Programm an:

```
public class ExceptionTest3{
   public static void main(String[] args){
      try{
         int d = Integer.parseInt(args[0]);
         int k = 10/d;
         System.out.println("Ergebnis ist "+k);
      } catch(NumberFormatException nfe){
         System.out.println("Falscher Typ! Gib eine Zahl ein ...");
      } catch(ArithmeticException ae){
         System.out.println("Division durch 0! ...");
      } catch(Exception e){
         System.out.println("Unbekannter Fehler aufgetreten ...");
      }
   }
}
```

Bei den schon bekannten Eingaben liefert das Programm nun folgende Ausgaben:

```
C:\>java ExceptionTest3 2
Ergebnis ist 5

C:\>java ExceptionTest3 0
Division durch 0! ...

C:\>java ExceptionTest3 d
Falscher Typ! Gib eine Zahl ein ...
```

Fehlerhafte Berechnungen aufspüren I

Beim Programmieren wird leider ein nicht unwesentlicher Teil der Zeit mit dem Aufsuchen von Fehlern verbracht. Das ist selbst bei sehr erfahrenen Programmierern so und gerade, wenn die Projekte größer und unübersichtlicher werden, sind effiziente Programmiertechniken, die Fehler vermeiden, unabdingbar. Sollte sich aber doch ein Fehler eingeschlichen haben, gibt es einige Vorgehensweisen, die die zum Auffinden benötige Zeit auf ein Mindestmaß reduzieren.

Beispiel: Berechnung der Zahl PI

Gottfried Wilhelm Leibniz gab 1682 für die Näherung von PI/4 eine Berechnungsvorschrift an, die als **Leibniz-Reihe** bekannt ist. Am Ende der Berechnung müssen wir das Ergebnis also noch mit 4 multiplizieren, um eine Näherung für PI zu erhalten.

Die Vorschrift besagt:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots$$

Fehlerhafte Berechnungen aufspüren II

Der Nenner wird also immer um 2 erhöht, während das Vorzeichen jeden Schritt wechselt. Eine Schleife bietet sich zur Berechnung an:

```
public class BerechnePI{
   static int max_iterationen = 100;
   static public double pi(){
      double PI = 0;
      int vorzeichen = 1;
      for (int i=1; i<=max_iterationen*2; i+=2){
         PI += vorzeichen*(1/i);
         vorzeichen *= -1;
      }
      return 4*PI;
   }

   public static void main(String[] args){
      double PI = pi();
      System.out.println("Eine Naeherung fuer PI ist "+PI);
   }
}
```

Nach 100 Iterationen stellen wir fest:

```
C:\JavaCode>java BerechnePI
Eine Naeherung fuer PI ist 4.0
```

Es scheint sich ein Fehler eingeschlichen zu haben...

Fehlerhafte Berechnungen aufspüren III

Um den Fehler aufzuspüren, versuchen wir die Berechnungen schrittweise nachzuvollziehen. Überprüfen wir zunächst, ob das Vorzeichen und der Nenner für die Berechnung stimmen. Dazu fügen wir in die Schleife folgende Ausgabe ein:

```
...
for (int i=1; i<max_iterationen; i+=2){
   System.out.println("i:"+i+" vorzeichen:"+vorzeichen);
   PI += vorzeichen*(1/i);
   vorzeichen *= -1;
}
...
```

Als Ausgabe erhalten wir:

```
C:\JavaCode>java BerechnePI
i:1 vorzeichen:1
i:3 vorzeichen:-1
i:5 vorzeichen:1
i:7 vorzeichen:-1
...
```

Das Vorzeichen alterniert, ändert sich also in jeder Iteration. Das ist korrekt.

Fehlerhafte Berechnungen aufspüren IV

Die Ausgabe erweitern wir, um zu sehen, wie sich PI im Laufe der Berechnungen verändert:

```
...
System.out.println("i:"+i+" vorzeichen:"+vorzeichen+" PI:"+PI);
...
```

Wir erhalten:

```
C:\JavaCode>java BerechnePI
i:1 vorzeichen:1 PI:0.0
i:3 vorzeichen:-1 PI:1.0
i:5 vorzeichen:1 PI:1.0
i:7 vorzeichen:-1 PI:1.0
...
```

Vor dem ersten Schritt hat PI den Initialwert 0. Nach dem ersten Schritt ist PI=1. Soweit so gut. Allerdings ändert sich PI in den weiteren Iterationen nicht mehr. Hier tritt der Fehler zum ersten Mal auf.

Fehlerhafte Berechnungen aufspüren V

Werfen wir einen Blick auf die Zwischenergebnisse:

```
...
double zwischenergebnis = vorzeichen*(1/i);
System.out.println("i:"+i+" Zwischenergebnis:"+zwischenergebnis);
...
```

Jetzt sehen wir, an welcher Stelle etwas schief gelaufen ist:

```
C:\JavaCode>java BerechnePI
i:1 Zwischenergebnis:1.0
i:3 Zwischenergebnis:0.0
i:5 Zwischenergebnis:0.0
i:7 Zwischenergebnis:0.0
...
```

Der Fehler ist zum Greifen nah! Die Berechnung **vorzeichen*(1/i)** liefert in jedem Schritt, außer dem ersten, den Wert 0.0 zurück.

Das Problem kennen wir bereits!!!

Die Berechnung **1/i** liefert immer das Ergebnis **0**, da sowohl **1** als auch **i** vom Typ **int** sind. Der Compiler verwendet die ganzahlige Division, was bedeutet, dass alle Stellen nach dem Komma abgerundet werden. Dieses Problem lässt sich leicht lösen, indem wir die **1** in **1.d** ändern und damit aus dem implizit angenommenen **int** ein **double** machen. Eine andere Möglichkeit wäre das Casten von **i** zu einem **double**.

Fehlerhafte Berechnungen aufspüren VI

Ändern wir das Programm entsprechend:

```
public class BerechnePI{
   static int max_iterationen = 100;
   static public double pi(){
      double PI = 0;
      int vorzeichen = 1;
      for (int i=1; i<=max_iterationen*2; i+=2){
         PI += vorzeichen*(1/(double)i);
         vorzeichen *= -1;
      }
      return 4*PI;
   }

   public static void main(String[] args){
      double PI = pi();
      System.out.println("Eine Naeherung fuer PI ist "+PI);
   }
}
```

In beiden Fällen liefert das korrigierte Programm eine gute Näherung:

```
C:\JavaCode>java BerechnePI
Eine Naeherung fuer PI ist 3.1315929035585537
```

Ein etwas geübter Programmierer erahnt einen solchen Fehler bereits beim Lesen des Codes. Das strategisch günstige Ausgeben von Zwischenergebnissen und Variablenwerten ist jedoch auch ein probates Mittel zum Auffinden von schwierigeren Fehlern. Wird das Programm jedoch größer, ist ein Code von anderen Programmieren beteiligt und sind die Berechnungen unübersichtlich, dann helfen die von den meisten Programmierumgebungen bereitgestellten Debugger enorm.

Objektorientierte Programmierung mit Java

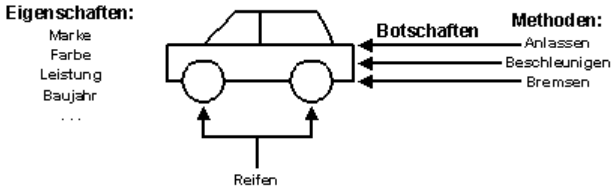
Inhalt:

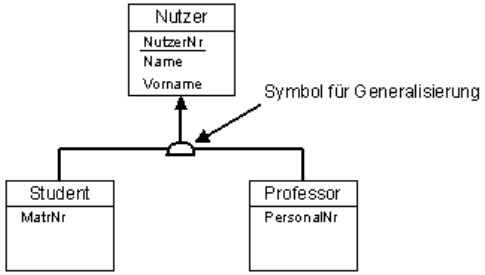
- Generalisierung, Spezialisierung
- Module, Klassen, Objekte
- Klassenhierarchien, Vererbung, abstrakte Klassen, Schnittstellen
- Entwicklung eines Fußballmanagers (EM 2008)

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007
Gries D., Gries P.: "*Multimedia Introduction to Programming Using Java*“, Springer-Verlag 2005
Abts D.: „*Grundkurs JAVA: Von den Grundlagen bis zu Datenbank- und Netzanwendungen*“, Vieweg-Verlag 2007

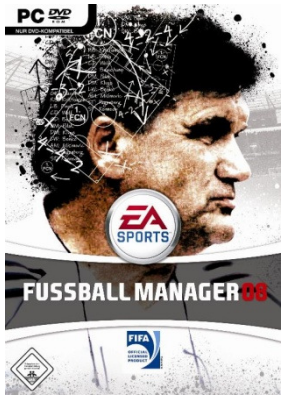
Einführung in die Objektorientierung (üblicherweise)

- Autos und Autoteile
- Professoren und Studenten
- ...





Wir machen das nicht! Die **Europomeisterschaft 2008** ist gerade vorbei und wir entwickeln zusammen einen einfachen Fußballmanager ...



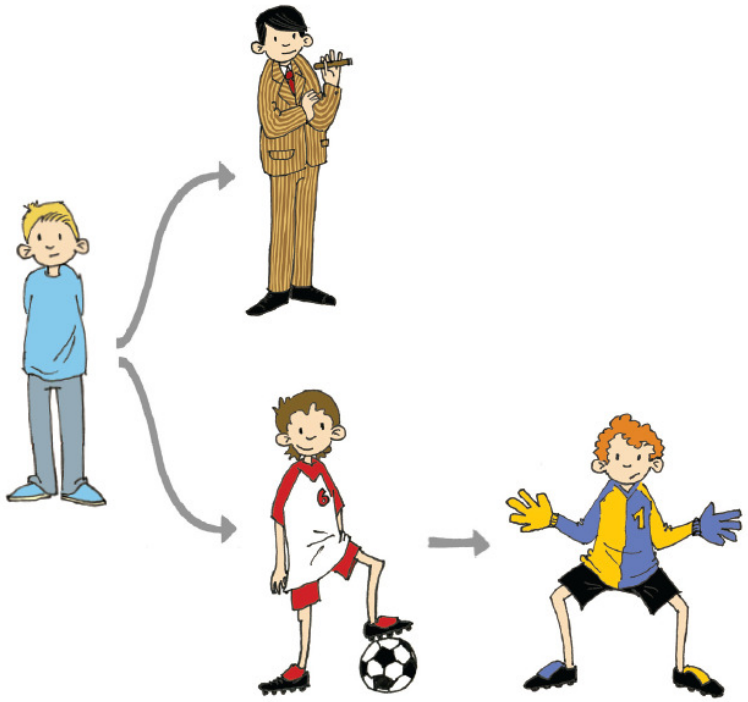
Vielleicht nicht so professionell, wie der von EA Sports, aber dafür können wir hinterher sagen, dass wir es selber gemacht haben ☺.

Generalisierung und Spezialisierung

Unter den beiden Begriffen Generalisierung und Spezialisierung verstehen wir zwei verschiedene Vorgehensweisen, Kategorien und Stammbäume von Dingen zu beschreiben. Wenn wir bei Dingen Gemeinsamkeiten beschreiben und danach kategorisieren, dann nennen wir das eine **Generalisierung**.

Mit der **Spezialisierung** beschreiben wir den umgekehrten Weg, aus einem „Ur-Ding“ können wir durch zahlreiche Veränderungen der Eigenschaften neue Dinge kreieren, die Eigenschaften übernehmen oder neue entwickeln.

Spezialisierung von links nach rechts.



Generalisierung von rechts nach links.

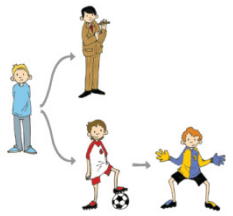
Gemeinsamkeiten von Spieler und Trainer

Wir haben **11 Spieler**, z.B. mit den Eigenschaften *Name*, *Alter*, *Stärke*, *Torschuss*, *Motivation* und *Tore*.
Neben den Spielern haben wir einen Trainer mit den Eigenschaften *Name*, *Alter* und *Erfahrung*.

Spieler
Name Alter Stärke Torschuss Motivation Tore

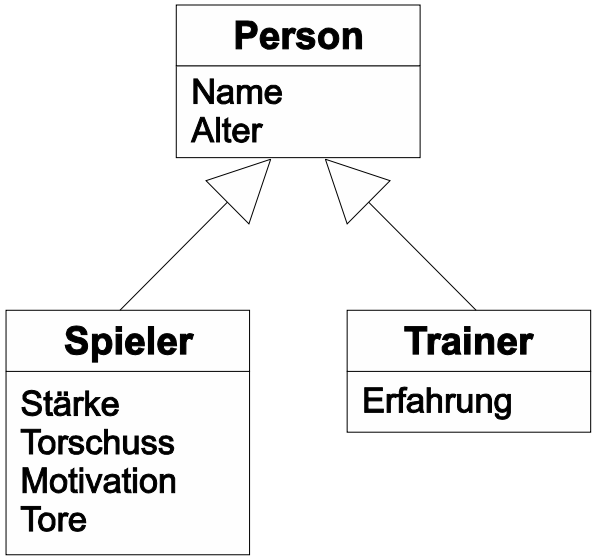
Trainer
Name Alter Erfahrung

Wir sehen schon, dass es Gemeinsamkeiten gibt.

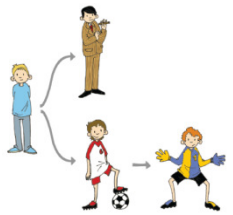


Klassen und Vererbung

Die Gemeinsamkeiten können wir zu einer eigenen Kategorie zusammenfassen und diese Eigenschaften an beide, **Spieler** und **Trainer**, vererben. Diese neue Kategorie nennen wir z.B. **Person**.
Spieler, **Trainer** und **Person** sind Kategorien oder Klassen.



Die Pfeile zeigen in die Richtung des Vorfahren. Wir haben also eine Repräsentation gefunden, die die Daten nicht mehr unnötig doppelt darstellt, wie es bei *Name* und *Alter* gewesen wäre, sondern sie übersichtlich nur einmal in der Klasse **Person** abgelegt. Des Weiteren wurde die Spezialisierung der **Person** zu den Klassen **Spieler** und **Trainer** durch zusätzliche Eigenschaften beschrieben.



Umsetzung in Java I

Um diese Darstellung in einem Javaprogramm zu implementieren, müssen wir drei Klassen **Person**, **Spieler** und **Trainer** anlegen.

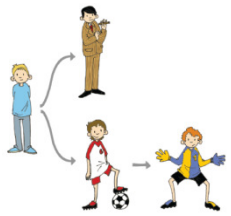
Beginnen wir mit der Klasse **Person**:

Das ist als „Bauplan“ zu verstehen.

```
public class Person{
   // Eigenschaften einer Person:
   public String name;
   public int alter;
}
```

Jetzt wollen wir die Klasse **Spieler** von **Person** ableiten und alle Eigenschaften, die die Klasse anbietet, übernehmen. In Java erweitern wir die Definition einer Klasse mit dem Befehl extends und den Namen der Klasse, von der wir ableiten wollen.

```
public class A extends B{
}
```



Umsetzung in Java II

Wir erweitern den Bauplan von Person.

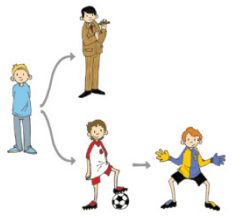
Jetzt können wir **Spieler** von **Person** ableiten:

```
public class Spieler extends Person{
   // Zusätzliche Eigenschaften eines Spielers:
   public int staerke;      // von 1 (schlecht) bis 10 (super)
   public int torschuss;    // von 1 (schlecht) bis 10 (super)
   public int motivation;   // von 1 (schlecht) bis 10 (super)
   public int tore;
}
```

Das war alles.

Jetzt haben wir zwei Klassen **Person** und **Spieler**. Alle Eigenschaften oder Attribute der Klasse **Person** sind nun auch in **Spieler** enthalten, darüber hinaus hat ein Spieler noch die Attribute *staerke*, *torschuss*, *motivation* und *tore*.

Mit diesen beiden Klassen haben wir eine einfache Vererbung realisiert.



Modifizierer public und private I

Nehmen wir an, dass zwei verschiedene Programmierer diese Klassen geschrieben haben und dass der Programmierer der Klasse **Person** für die Variable *alter* nur positive Zahlen zwischen 1 und 100 akzeptieren möchte. Er hat aber keinen Einfluss auf die Verwendung, da die Variable *alter* mit dem zusätzlichen Attribut public versehen wurde.

Das bedeutet, dass jeder, der diese Klasse verwenden möchte auf diese Attribute uneingeschränkt zugreifen kann!

Es gibt die Möglichkeit diese Variablen vor Zugriff zu schützen, indem das Attribut private verwendet wird.

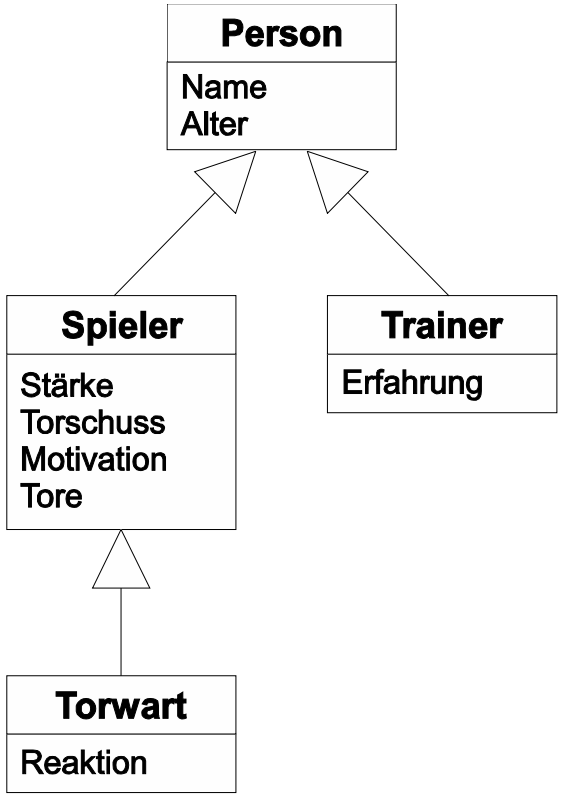
Modifizierer public und private II

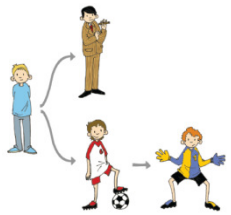
Jetzt kann diese Variable nicht mehr außerhalb der Klasse angesprochen werden. Um aber die Variablen verändern und lesen zu können, schreiben wir zwei Funktionen und vergeben ihnen das Attribut public:

```
public class Person{
   // Eigenschaften einer Person:
   private String name;
   private int alter;

   // Funktionen (get und set):
   public String getName(){
      return name;
   }

   public void setName(String n){
      name = n;
   }

   public int getAlter(){
      return alter;
   }

   public void setAlter(int a){
      alter = a;
   }
}
```

Torwart als Spezialisierung eines Spielers

Bei den Spielern gibt es noch einen Sonderling, den Torwart. Als zusätzliche Eigenschaft hat er *reaktion*, damit wird später entschieden, ob er die Torschüsse hält oder nicht. Sicherlich könnten wir an dieser Stelle die Spieler noch in Abwehr, Mittelfeld und Angriff unterscheiden, aber fürs erste soll die Spezialisierung der Klasse **Spieler** zum **Torwart** als Spezialfall eines Spielers genügen:

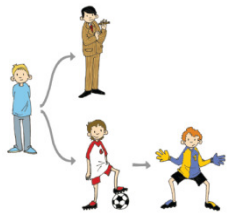




Umsetzung in Java

Hier die aktuelle Klasse **Trainer** mit den entsprechenden get-set-Funktionen:

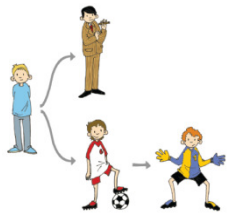
```
public class Trainer extends Person{
   // Zusätzliche Eigenschaften eines Trainers:
   private int erfahrung;

   // Funktionen (get und set):
   public int getErfahrung(){
      return erfahrung;
   }

   public void setErfahrung(int e){
      erfahrung = e;
   }
}
```



Objekte und Instanzen I

Bisher haben wir die Dinge klassifiziert (einen Bauplan erstellt) und die Gemeinsamkeiten in zusätzlichen Klassen beschrieben, aber noch keinen Spieler oder Trainer, der diese Eigenschaften und Funktionen besitzt erzeugt und untersucht.

Wenn von einer Klasse ein Exemplar erzeugt wird (die Klasse stellt sozusagen den Bauplan fest), dann nennt man das ein Objekt oder eine Instanz dieser Klasse.



Objekte und Instanzen II

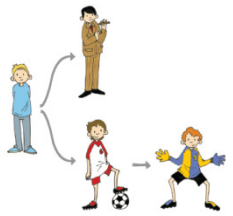
Wenn wir beispielsweise ein Programm schreiben wollen, dass mit Jürgen Klinsmann als Trainer arbeiten möchte, dann erzeugen wir eine neue Instanz der Klasse **Trainer** und geben ihm die Informationen name="Jürgen Klinsmann, alter=42 und erfahrung=7.

```
public class Test{
   public static void main(String[] args){
      Trainer trainer = new Trainer();
      trainer.setName("Jürgen Klinsmann");
      trainer.setAlter(42);
      trainer.setErfahrung(7);
   }
}
```

Wir erzeugen eine Instanz der Klasse Trainer. Ähnlich wie bei den primitiven Datentypen müssen wir Speicherplatz reservieren und machen das hier in der dritten Zeile.

Jetzt können wir mit dem Objekt trainer arbeiten. Sollten die Daten des Objekts geändert werden, so können wir das über die mit public versehenen Funktionen der Klassen **Trainer** und **Person** tun.
Wir sehen schon, dass die Funktion *setErfahrung* in der Klasse **Trainer** definiert wurde und verwendet werden kann.

Die **Person** Funktion *setName* wurde aber in definiert und kann trotzdem verwendet werden.



Konstruktoren I

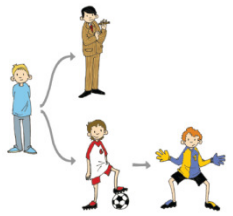
Im vorhergehenden Beispiel war es etwas umständlich, erst ein Objekt zu erzeugen und anschließend die Parameter über die set-Methoden zu setzen. Es geht bedeutend einfacher mit den Konstruktoren. Sie sind quasi die Funktionen, die bei der Reservierung des Speicherplatzes, bei der Erzeugung eines Objekts, ausgeführt werden.

Der Konstruktor entspricht der Syntax:

```
public <Klassenname>(Parameterliste){
}
```

Beim **Trainer-Beispiel** könnten wir zum Beispiel den folgenden Konstruktor angeben:

```
public class Trainer extends Person{
   // Zusätzliche Eigenschaften eines Trainers:
   private int erfahrung;

   // Konstruktoren
   public Trainer(String n, int a, int e){
      super(n, a);
      erfahrung       = e;
   }

   // Funktionen (get und set):
   ...
}
```



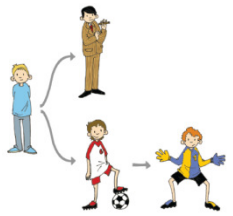
Konstruktoren II

So sah die Erzeugung ☺ von Trainer Jürgen Klinsmann vorhin aus:

```
public class Test{
   public static void main(String[] args){
      Trainer trainer = new Trainer();
      trainer.setName("Jürgen Klinsmann");
      trainer.setAlter(42);
      trainer.setErfahrung(7);
   }
}
```

Und so können wir es jetzt viel einfacher mit dem Konstruktur machen:

```
public class Test{
   public static void main(String[] args){
      Trainer trainer = new Trainer("Jürgen Klinsmann", 42, 7);
   }
}
```

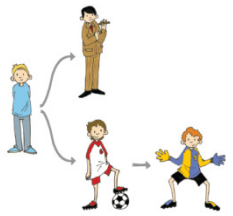


Konstruktoren III

Die Anweisung super mit den Parametern *name* und *alter* ruft den Konstruktor der Klasse auf, von der geerbt wird. In diesem Beispiel also den Konstruktor der Klasse **Person**.

Da aber *name* und *alter* in der Klasse **Person** gespeichert sind und wir dort ebenfalls mit einem K Konstruktor eine einfachere Initialisierung eines Objekts haben möchten, ändern wir die Klasse **Person**, wie folgt:

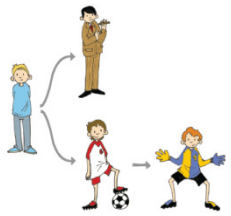
```
public class Person{
   // Eigenschaften einer Person:
   private String name;
   private int alter;

   // Konstruktoren
   public Person(String n, int a){
      name  = n;
      alter = a;
   }

   // Funktionen (get und set):
   ...
}
```



Die Klasse Spieler I

Die Klasse **Spieler** verwendet den Konstruktor der Klasse **Person**:

```
import java.util.Random;

public class Spieler extends Person{
   // Zusätzliche Eigenschaften eines Spielers:
   private int staerke;      // von 1 (schlecht) bis 10 (super)
   private int torschuss;    // von 1 (schlecht) bis 10 (super)
   private int motivation;   // von 1 (schlecht) bis 10 (super)
   private int tore;

   // Konstruktoren
   public Spieler(String n, int a, int s, int t, int m){
      super(n, a);
      staerke         = s;
      torschuss       = t;
      motivation      = m;
      tore            = 0;
   }

   ...
```



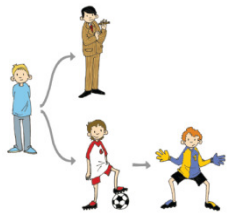
Die Klasse Spieler II

Der **Spieler** erhält zwei neue Methoden *addTor* und *schiesstAufTor*:

```
   ...

   // Funktionen (get und set):
   ...

   // Spielerfunktionen:

   // Der Spieler hat ein Tor geschossen
   public void addTor(){
      tore++;
   }

   // eine Zahl von 1-10 liefert die Qualität des Torschusses mit
   // einem kleinen Zufallswert +1 oder -1
   public int schiesstAufTor(){
      Random r = new Random();
      // Entfernungspauschale :)
      torschuss = Math.max(1, Math.min(10, torschuss - r.nextInt(3)));
      // +-1 ist hier die Varianz
      int ret = Math.max(1, Math.min(10, torschuss + r.nextInt(3)-1));
      return ret;
   }
}
```

Zufallszahlen haben wir bereits bei Conways Game of Life gesehen.

Die Klasse **Spieler** hat eine Funktion *schiesstAufTor*, falls dieser Spieler in der Partie eine Torchance erhält, dann wird eine zufällige Zahl im Bereich von 1-10 gewählt, die abhängig von der Torschuss-qualität des Spielers ist.

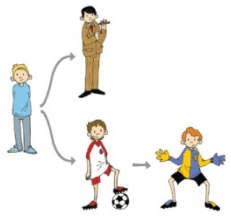


Die Klasse Torwart

Der Torwart erhält eine Funktion, die entscheidet, ob der abgegebene Torschuss pariert oder durchgelassen wird (ebenfalls mit einem zufälligen Ausgang):

```
import java.util.Random;
public class Torwart extends Spieler{
   // Zusätzliche Eigenschaften eines Torwarts:
   private int reaktion;

   // Konstruktoren
   public Torwart(String n, int a, int s, int t, int m, int r){
      super(n, a, s, t, m);
      reaktion = r;
   }

   // Funktionen (get und set):
   ...

   // Torwartfunktionen:

   // Als Parameter erhält der Torwart die Torschussstärke und nun muss
   // entschieden werden, ob der Torwart hält oder nicht
   public boolean haeltDenSchuss(int schuss){
      Random r = new Random();
      // +-1 ist hier die Varianz
      int ret = Math.max(1, Math.min(10, reaktion + r.nextInt(3)-1));
      if (ret>=schuss)
         return true;  // gehalten
      else
         return false; // TOR!!!
   }
}
```

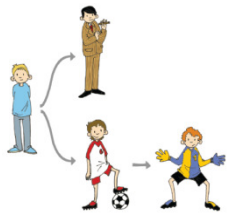
Hier werden 3 Konstruktoren aufgerufen! Towart, Spieler und Person.



Die Mannschaft I

Wir haben die Spieler und den Trainer modelliert, jetzt ist es an der Zeit eine Mannschaft zu beschreiben. Eine **Mannschaft** ist eine Klasse mit den Eigenschaften *name*, *Trainer*, *Torwart* und *Spieler*. Es gibt wieder einen Konstruktor und die get-set-Funktionen.

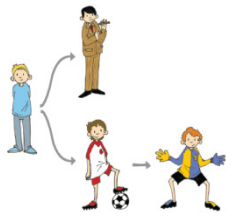
```
public class Mannschaft{
   // Eigenschaften einer Mannschaft:
   private String name;
   private Trainer trainer;
   private Torwart torwart;
   private Spieler[] kader;

   // Konstruktoren
   public Mannschaft(String n, Trainer t, Torwart tw, Spieler[] s){
      name            = n;
      trainer         = t;
      torwart         = tw;
      kader           = s;
   }

   // Funktionen (get und set):
   ...
```



Die Mannschaft II

Zusätzlich besitzt die Klasse **Mannschaft** die Funktionen *getStaerke* und *getMotivation* , die die durchschnittliche Stärke, bzw. Motivation der Mannschaft als Zahlenwert wiedergibt.

```
   ...
   // Mannschaftsfunktionen:

   // liefert die durchschnittliche Mannschaftsstaerke
   public int getStaerke(){
      int summ = 0;
      for (int i=0; i<10; i++)
         summ += kader[i].getStaerke();
      return summ/10;
   }

   // liefert die durchschnittliche Mannschaftsmotivation
   public int getMotivation(){
      int summ = 0;
      for (int i=0; i<10; i++)
         summ += kader[i].getMotivation();
      return summ/10;
   }
}
```



Interfaces I

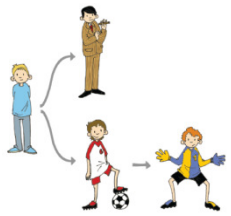
Die Klasse **Mannschaft** und alle dazugehörigen Klassen **Spieler**, **Torwart** und **Trainer** wurden definiert. Es ist an der Zeit, ein Freundschaftsspiel zweier Mannschaften zu realisieren. Auf die hier vorgestellte Weise könnten ganze Ligen und Turniere implementiert werden. Das wird Teil der Übungsaufgaben und der eigenen Motivation sein ☺.

Wir lernen dabei eine weitere wichtige Vererbungsvariante kennen. Zunächst beschreiben wir allgemein, wie wir uns ein Freundschaftsspiel vorstellen und implementieren es dann.

Jeder der ein Freundschaftsspiel, so wie wir es verstehen, implementieren möchte, kann sich an dieser Schnittstelle orientieren und bleibt mit unserem Kontext kompatibel. Es werden in diesem Interface (Schnittstelle) nur die Funktionen beschrieben, die jeder implementieren muss. Es gibt keine funktionsfähigen Programme, sondern nur die Funktionsköpfe:

Interface = Schnittstelle

```
public interface Freundschaftsspiel{
   String getHeimMannschaft();
   String getGastMannschaft();
   int getHeimPunkte();
   int getGastPunkte();

   String getErgebnisText();
}
```

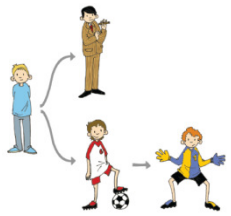


Die Klasse Fussballfreundschaftsspiel I

Anhand dieses Interfaces können wir speziell für den Fußball eine neue Klasse **FussballFreundschaftsspiel** entwerfen. Wir müssen alle vorgegebenen Funktionen eines Interfaces implementieren. Es können noch mehr dazu kommen, aber es dürfen keine fehlen.

Das Schlüsselwort implements signalisiert, dass wir ein Interface implementieren.

```
import java.util.Random;

public class Fussballfreundschaftsspiel implements Freundschaftsspiel{
  private String nameHeimMannschaft;
  private String nameGastMannschaft;
  private int punkteHeim;
  private int punkteGast;

  // Konstruktor
  public Fussballfreundschaftsspiel(){
     punkteHeim      = 0;
     punkteGast      = 0;
  }

  // Methoden des Interface, die implementiert werden müssen:
  public String getHeimMannschaft(){
     return nameHeimMannschaft;
  }

  public String getGastMannschaft(){
     return nameGastMannschaft;
  }

  ...
```



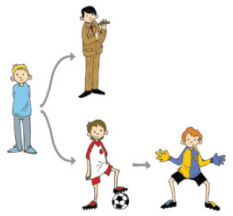
Die Klasse Fussballfreundschaftsspiel II

Die restlichen zwei Funktionen:

```
...
public int getHeimPunkte(){
   return punkteHeim;
}

public int getGastPunkte(){
   return punkteGast;
}
```

Nachdem die Funktionen *getHeimMannschaft*, *getGastMannschaft*, *getHeimPunkte* und *getGastPunkte* implementiert sind, folgt die Methode *starteSpiel*.

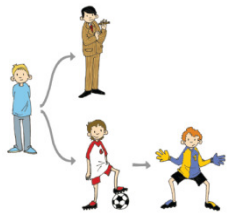
```
// Ein Fussballfreundschaftsspiel zwischen beiden Mannschaften wird
// gestartet.
public void starteSpiel(Mannschaft m1, Mannschaft m2){
   nameHeimMannschaft      = m1.getName();
   nameGastMannschaft      = m2.getName();
   punkteHeim              = 0;
   punkteGast              = 0;

   // jetzt starten wir das Spiel und erzeugen für die 90 Minuten
   // Spiel plus Nachspielzeit die verschiedenen Aktionen
   // (wahrscheinlichkeitsbedingt) für das Freundschaftsspiel
   Random r = new Random();
   ...
```



Die Klasse Fussballfreundschaftsspiel III

Weiter geht's mit der Methode *starteSpiel*:

```
...
boolean spiellaeuft  = true;
int spieldauer       = 90 + r.nextInt(5);
int zeit             = 1;
int naechsteAktion;

// solange das Spiel laeuft, koennen Torchancen entstehen...
while (spiellaeuft){
   naechsteAktion = r.nextInt(15)+1;

   // Ist das Spiel schon zu Ende?
   if ((zeit + naechsteAktion>spieldauer)||(zeit>spieldauer)){
      spiellaeuft = false;
      break;
   }

   // ******************************************************
   // Eine neue Aktion findet statt...
   zeit = zeit + naechsteAktion;
...
```

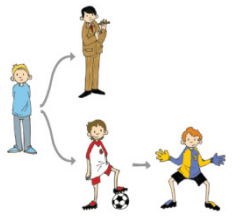


Die Klasse Fussballfreundschaftsspiel IV

Jetzt müssen wir entscheiden, wer den Torschuss abgibt, wie stark er schießt und ob der Torwart den Ball hält. Das berechnen wir nach folgender Wahrscheinlichkeitsverteilung.:

1) Wähle eine Mannschaft für eine Torchance aus, als Kriterien dienen Stärke und Motivation der Mannschaften sowie die Erfahrung des Trainers. Nach der Berechnung aller Einflüsse, hat die bessere Mannschaft eine größere Torchance.

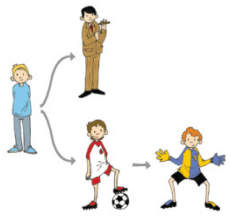
```
   ...
      // Einfluss der Motivation auf die Stärke:
      float staerke_1 = (m1.getStaerke()/2.0f) +
            ((m1.getStaerke()/2.0f) * (m1.getMotivation()/10.0f));
      float staerke_2 = (m2.getStaerke()/2.0f) +
            ((m2.getStaerke()/2.0f) * (m2.getMotivation()/10.0f));

      // Einfluss des Trainers auf die Stärke:
      int abweichung = r.nextInt(2);
      if (staerke_1 > m1.getTrainer().getErfahrung())
         abweichung = -abweichung;
      staerke_1 = Math.max(1, Math.min(10, staerke_1 + abweichung));

      abweichung = r.nextInt(2);
      if (staerke_2 > m2.getTrainer().getErfahrung())
         abweichung = -abweichung;
      staerke_2 = Math.max(1, Math.min(10, staerke_2 + abweichung));
...
```



Die Klasse Fussballfreundschaftsspiel V

2) Wähle zufällig einen Spieler aus dieser Mannschaft, berechne den Torschuss und gib dem Torwart der anderen Mannschaft die Möglichkeit, diesen Ball zu halten.

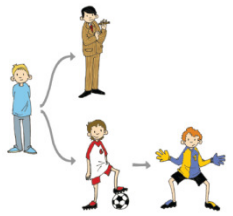
```
...
   int schuetze    = r.nextInt(10);

   if ((r.nextInt(Math.round(staerke_1+staerke_2))-staerke_1)<=0){
      Spieler s       = m1.getKader()[schuetze];
      Torwart t       = m2.getTorwart();
      int torschuss   = s.schiesstAufTor();
      // haelt er den Schuss?
      boolean tor     = !t.haeltDenSchuss(torschuss);

      System.out.println();
      System.out.println(zeit+".Minute: ");
      System.out.println("   Chance fuer "+m1.getName()+" ...");
      System.out.println("   "+s.getName()+" zieht ab");

      if (tor) {
         punkteHeim++;
         s.addTor();
         System.out.println("   TOR!!!    "+punkteHeim+":"+
                punkteGast+" "+s.getName()+"("+s.getTore()+")");
      } else {
         System.out.println("   "+m2.getTorwart().getName()
                           +" pariert glanzvoll.");
      }
    }
    else
...
```



Die Klasse Fussballfreundschaftsspiel VI

Teil 2:

```
    ...
        {
           Spieler s       = m2.getKader()[schuetze];
           Torwart t       = m1.getTorwart();
           int torschuss   = s.schiesstAufTor();
           boolean tor     = !t.haeltDenSchuss(torschuss);

           System.out.println();
           System.out.println(zeit+".Minute: ");
           System.out.println("   Chance fuer "+m2.getName()+" ...");
           System.out.println("   "+s.getName()+" zieht ab");

           if (tor) {
              punkteGast++;
              s.addTor();
              System.out.println("  TOR!!!    "+punkteHeim+":"+
                     punkteGast+" "+s.getName()+"("+s.getTore()+")");
           } else {
              System.out.println("   "+m1.getTorwart().getName()
                                 +" pariert glanzvoll.");
           }
        }
        // ************************************************************
     }
  }
```

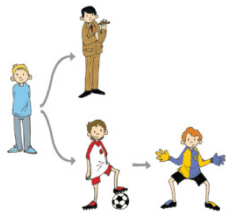


Die Klasse Fussballfreundschaftsspiel VII

Es fehlt für die vollständige Implementierung aller Funktionen des Interfaces noch die Methode *getErgebnisText:*

```
   ...
   public String getErgebnisText(){
      return  "Das Freundschaftsspiel endete \n\n"+nameHeimMannschaft
              +" - "+nameGastMannschaft+" "+punkteHeim+":"
              +punkteGast+".";
   }
}
```

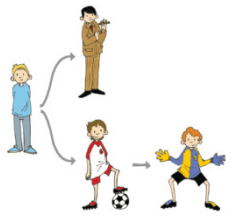
Jetzt haben wir alle notwendigen Funktionen implementiert und können schon im nächsten Schritt mit einem Spiel beginnen.



Freundschaftsspiel Deutschland-Brasilien der WM-Mannschaften von 2006 I

Nach der ganzen Theorie und den vielen Programmzeilen, können wir uns zurücklehnen und ein, bei der WM nicht stattgefundenes Spiel Deutschland gegen Brasilien, anschauen. Dazu müssen wir zunächst beide Mannschaften definieren und anschließend beide mit der Klasse Fußballfreundschaftsspiel eine Partie spielen lassen.

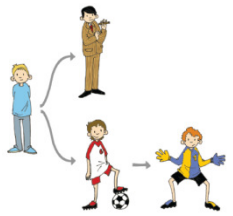
```
public class FussballTestKlasse{
   public static void main(String[] args){
      // ************************************************************
      // Mannschaft 1
      Trainer t1       = new Trainer("Juergen Klinsmann", 34, 9);
      Torwart tw1      = new Torwart("J. Lehmann", 36, 8, 1, 9, 7);

      Spieler[] sp1    = new Spieler[10];
      sp1[0]           = new Spieler("P. Lahm", 23, 9, 5, 9);
      sp1[1]           = new Spieler("C. Metzelder", 25, 8, 2, 7);
      sp1[2]           = new Spieler("P. Mertesacker", 22, 9, 2, 8);
      sp1[3]           = new Spieler("M. Ballack", 29, 7, 5, 8);
      sp1[4]           = new Spieler("T. Borowski", 26, 9, 8, 9);
      sp1[5]           = new Spieler("D. Odonkor", 22, 7, 5, 8);
      sp1[6]           = new Spieler("B. Schweinsteiger", 22, 2, 3, 2);
      sp1[7]           = new Spieler("L. Podolski", 21, 7, 8, 9);
      sp1[8]           = new Spieler("M. Klose", 28, 10, 9, 7);
      sp1[9]           = new Spieler("O. Neuville", 33, 8, 8, 7);
      // ************************************************************

      ...
```



Freundschaftsspiel Deutschland-Brasilien der WM-Mannschaften von 2006 II

weiter geht's:

```
...
// ************************************************************
// Mannschaft 2
Trainer t2       = new Trainer("Carlos Alberto Parreira", 50, 3);
Torwart tw2      = new Torwart("Dida", 25, 9, 1, 6, 8);

Spieler[] sp2    = new Spieler[10];
sp2[0]           = new Spieler("Cafu", 33, 8, 4, 6);
sp2[1]           = new Spieler("R. Carlos", 32, 9, 9, 2);
sp2[2]           = new Spieler("Lucio", 29, 10, 9, 9);
sp2[3]           = new Spieler("Ronaldinho", 25, 10, 9, 5);
sp2[4]           = new Spieler("Zé Roberto", 27, 7, 7, 4);
sp2[5]           = new Spieler("Kaká", 22, 10, 8, 10);
sp2[6]           = new Spieler("Juninho", 26, 7, 10, 3);
sp2[7]           = new Spieler("Adriano", 23, 8, 8, 4);
sp2[8]           = new Spieler("Robinho", 19, 9, 8, 9);
sp2[9]           = new Spieler("Ronaldo", 28, 4, 10, 2);
// ************************************************************

Mannschaft m1 = new Mannschaft("Deutschland WM 2006",t1,tw1,sp1);
Mannschaft m2 = new Mannschaft("Brasilien WM 2006",t2,tw2,sp2);
Fussballfreundschaftsspiel f1 = new Fussballfreundschaftsspiel();
...
```

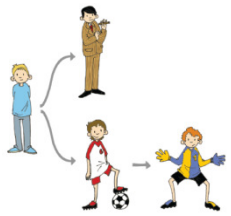


Freundschaftsspiel Deutschland-Brasilien der WM-Mannschaften von 2006 III

Wir wollen das Spiel starten und das Ergebnis ausgeben:

```
        ...
        System.out.println("-----------------------------------------");
        System.out.println("Start des Freundschaftspiels zwischen");
        System.out.println();
        System.out.println(m1.getName());
        System.out.println("   Trainer: "+m1.getTrainer().getName());
        System.out.println();
        System.out.println("   und");
        System.out.println();
        System.out.println(m2.getName());
        System.out.println("   Trainer: "+m2.getTrainer().getName());
        System.out.println("-----------------------------------------");

        f1.starteSpiel(m1, m2);

        System.out.println();
        System.out.println("-----------------------------------------");
        System.out.println(f1.getErgebnisText());
        System.out.println("-----------------------------------------");
    }
}
```

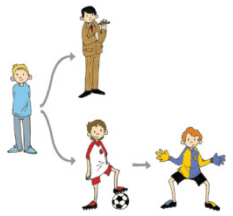
Das Spiel ist vorbereitet und die Akteure warten ungeduldig auf den Anpfiff ...



Abstrakte Klassen I

Im Vergleich zu einem Interface, bei dem es nur Funktionsköpfe gibt, die implementiert werden müssen, gibt es bei der abstrakten Klasse die Möglichkeit, Funktionen bereits bei der Definition der Klasse zu implementieren.

Eine abstrakte Klasse muss mindestens eine abstrakte Methode (also ohne Implementierung) besitzen. Demzufolge ist das Interface ein Spezialfall der abstrakten Klasse, denn ein Interface besteht nur aus abstrakten Methoden und Konstanten.



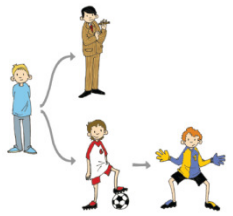
Abstrakte Klassen II

Schauen wir uns ein ganz kurzes Beispiel dazu an. Die Klasse **A** verwaltet die Variable *wert* und bietet bereits die Methode *getWert*. Die Methode *setWert* soll aber implementiert werden und deshalb wird sie mit dem Schlüsselwort abstract versehen:

```
public abstract class A{
   public int wert;

   public int getWert(){
      return wert;
   }

   public abstract void setWert(int w);
}
```

Die Klasse **B** erbt die Informationen der Klasse **A**, also *wert* und die Methode *getWert*, muss aber die Methode *setWert* implementieren. Durch das Schlüsselwort public ist es der Klasse **B** nach der Vererbung erlaubt, auf die Variable *wert* zuzugreifen:

```
public class B extends A{
   public void setWert(int w){
      wert = w;
   }
}
```



Abstrakte Klassen III

Jetzt nehmen wir noch eine Testklasse **Tester** dazu und testen mit ihr die gültige Funktionalität. Als erstes wollen wir versuchen eine Instanz der Klasse **A** zu erzeugen.

```
A a = new A();
```

Das schlägt mit der folgenden Ausgabe fehl:

```
C:\JavaCode>javac Tester.java
Tester.java:3: A is abstract; cannot be instantiated
                A a = new A();
                      ^
1 error
```

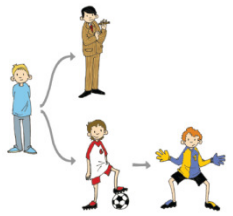
Es ist nicht erlaubt von einer abstrakten Klasse eine Instanz zu erzeugen!

Dann erzeugen wir eine Instanz der Klasse **B**:

```
B b = new B();
b.setWert(4);
System.out.println(""+b.getWert());
```

Und erhalten:

```
C:\JavaCode>javac Tester.java
C:\JavaCode>java Tester
4
```



Objektorientierte Sicht auf die ersten Konzepte

Inhalt:

- Referenzvariablen, Punktnotation, this
- Überladung von Methoden und Konstruktoren
- Garbage Collector
- Statische Datentypen, Wrapperklassen
- Die Klasse String

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007
Gries D., Gries P.: "*Multimedia Introduction to Programming Using Java*“, Springer-Verlag 2005
Abts D.: „*Grundkurs JAVA: Von den Grundlagen bis zu Datenbank- und Netzanwendungen*“, Vieweg-Verlag 2007

Referenzvariablen I

Zur Erinnerung, wir haben im vorherigen Abschnitt die Klasse **Person** implementiert. Hier noch einmal der Programmcode dieser Klasse:

```
public class Person{
   // Eigenschaften einer Person:
   private String name;
   private int alter;

   // Konstruktoren
   public Person(String n, int a){
      name  = n;
      alter = a;
   }

   // Funktionen (get und set):
   ...
}
```

Referenzvariablen II

Um zu verstehen, dass Referenzen Adressverweise auf einen reservierten Speicherplatz sind, schauen wir uns folgendes Beispiel an:

```
Person p1;
p1 = new Person("Hugo", 12);
Person p2;
p2 = p1;
if (p1 == p2)
   System.out.println("Die Referenzen sind gleich");
```

In der ersten Zeile wird p1 deklariert. p1 ist eine Referenzvariable und beinhaltet eine Adresse. Diese Adresse ist momentan nicht gegeben, sollten wir versuchen auf p1 zuzugreifen, würde Java einen Fehler beim Kompilieren mit der Begründung verursachen: „Variable p1 ist nicht initialisiert“. Die zweite Zeile stellt nun aber einen Speicherplatz bereit und erzeugt ein Objekt der Klasse **Person** und vergibt den Attributen gleich Werte. p1 zeigt nun auf diesen Speicherbereich.

Die dritte Zeile verhält sich äquivalent zur ersten. In Zeile vier weisen wir aber nun die Adresse von p1 der Variablen p2 zu. Beide Variablen zeigen nun auf den gleichen Speicherplatz, auf dasselbe Objekt. Sollten wir Veränderungen in p1 vornehmen, so treten diese Veränderungen ebenfalls bei p2 auf, denn es handelt sich um dasselbe Objekt!

Objektorientierte Sicht auf die ersten Konzepte

Zugriff auf Attribute und Methoden durch Punktnotation I

Wir haben diese Syntax bereits schon mehrfach verwendet, aber nun werden wir es konkretisieren. Existiert eine Referenz auf ein Objekt, so können wir mit einem Punkt nach der Referenz und dem Namen des entsprechenden Attributs (das nennen wir dann Instanzvariable) bzw. der entsprechenden Methode (analog Instanzmethode) auf diese zugreifen.

Referenz.Attribut
Referenz.Methode

Dazu schauen wir uns ein kleines Beispiel an und werden den Unterschied zwischen Variablen, die primitive Datentypen repräsentieren und Variablen, die Referenzen repräsentieren, erläutern.

```
int a    = 2;
Person p = new Person("Hans", 92);
// An dieser Stelle hat p.getName() den Rückgabewert "Hans"
komischeFunktion(a, p);
```

Wir werden nun a und p an eine Funktion übergeben. Für den primitiven Datentypen a gilt die Übergabe , "call by value". Das bedeutet, dass der Inhalt, also die in die Funktion übergeben wird. Anders sieht es bei dem Referenzdatentyp aus, hier wird die Referenz, also die Adresse übergeben, wir nennen das „call by reference".

Zugriff auf Attribute und Methoden durch Punktnotation II

Hier nochmal der Aufruf:

```
int a    = 2;
Person p = new Person("Hans", 92);
// An dieser Stelle hat p.getName() den Rückgabewert "Hans"
komischeFunktion(a, p);
```

und die „komische Funktion“:

```
public void komischeFunktion(int x, Person y){
   // Wir ändern die Werte der Eingabeparameter.
   x = 7;
   y.setName("Gerd");
}
```

Sollten wir nach Ausführung der Zeile *komischeFunktion(a, p);* wieder den Rückgabewert von p erfragen, so erhalten wir „Gerd“. Die Variable a bleibt indes unangetastet.

Referenzvariable this

Wir haben schon in einem vorhergehendenAbschnitt gesehen, wie sich die Instanzvariablen von „außen“ setzen lassen. Jedes Objekt kann aber auch mittels einer Referenzvariablen auf sich selbst reflektieren und seine Variablen und Funktionen ansprechen.

Dazu wurde die Klasse **Person** im folgenden Beispiel vereinfacht:

```
public class Person{
   private String name;
   public void setName(String name){
      this.name  = name;
   }
}
```

Wir können mit der Referenzvariablen this auf „unsere“ Instanzvariablen und -methoden zugreifen.

Prinzip des Überladens I

Oft ist es sinnvoll eine Funktion für verschiedene Eingabetypen zur Verfügung zu stellen. Damit wir nicht jedes Mal einen neuen Funktionsnamen wählen müssen, gibt es die Möglichkeit der Überladung in Java. Wir können einen Funktionsnamen mehrfach verwenden, falls sich die Signatur der Funktionen eindeutig unterscheiden lässt. Als Signatur definieren wir die Kombination aus *Rueckgabewert*, *Funktionsname* und *Eingabeparameter*, wobei nur der Typ der Eingabe und nicht die Bezeichnung entscheidend ist.

Schauen wir uns ein Beispiel an:

```
// Max-Funktion für Datentyp int
int max(int a, int b){
   if (a<b) return b;
   return a;
}

// Max-Funktion für Datentyp double
double max(double a, double b){
   ... // analog zu der ersten Funktion
}
```

Beim Aufruf der Funktion *max* wird die passende Signatur ermittelt und diese Funktion entsprechend verwendet. Wir können uns merken, trotz Namensgleichheit sind mehrere Funktionen mit verschiedenen Signaturen erlaubt.

Prinzip des Überladens II

Dieses Prinzip lässt sich auch auf Konstruktoren übertragen:

```
public class Person{
   private String name;
   private int alter;
   // Konstruktor 1
   public Person(){
      name       = "";   // z.B. als Defaultwert
      alter      = 1;    // z.B. als Defaultwert
   }
   // Konstruktor 2
   public Person(String name){
      this.name  = name;
      alter      = 1;    // z.B. als Defaultwert
   }
   // Konstruktor 3
   public Person(String name, int alter){
      this.name  = name;
      this.alter = alter;
   }
}
```

Nun aber ein Beispiel zu der neuen **Person-Klasse**:

```
Person p1 = new Person();
Person p2 = new Person("Herbert", 30);
```

Die erste Zeile verwendet den passenden Konstruktor (Konstruktor 1) mit der parameterlosen Signatur und die zweite verwendet den dritten Konstruktor.

Prinzip des Überladens III

Kleiner Hinweis: Genau dann, wenn kein Konstruktor angegeben sein sollte, denkt sich Java den Defaultkonstruktor hinzu, der keine Parameter erhält und keine Attribute setzt. Es kann trotzdem ein Objekt dieser Klasse erzeugt werden.

Der Copy-Konstruktor

Bei der Übergabe von Objekten, werden diese nicht kopiert, sondern lediglich die Referenz übergeben, das wissen wir bereits. Wir müssen also immer daran denken, eine lokale Kopie des Objekts vorzunehmen.

Elegant lässt sich das mit dem Copy-Konstruktor lösen:

```
public class CopyKlasse{
   public int a, b, c, d;

   public CopyKlasse(){
      a=0, b=0, c=0, d=0;
   }

   public CopyKlasse(CopyKlasse ck){
      this.a = ck.a;
      this.b = ck.b;
      this.c = ck.c;
      this.d = ck.d;
   }
}
```

Zuerst erzeugen wir eine Instanz der Klasse **CopyKlasse** und anschließend erstellen wir eine Kopie:

```
CopyKlasse a = new CopyKlasse();
CopyKlasse copyVonA = new CopyKlasse(a);
```

Garbage Collector

Bisher haben wir andauernd Speicher für unsere Variablen reserviert, aber wann wird dieser Speicher wieder frei gegeben? Java besitzt mit dem Garbage Collector eine, „Müllabfuhr“, die sich automatisch um die Freigabe des Speichers kümmert.

Das ist gegenüber anderen Programmiersprachen ein Komfort. Dieser Komfort ist aber nicht umsonst, Java entscheidet eigenständig den Garbage Collector zu starten. Es könnte also auch dann geschehen, wenn gerade eine performancekritische Funktion gestartet wird.

Statische Attribute und Methoden I

Im Erweiterten Klassenkonzept war die Nutzung von Attributen und Methoden an die Existenz von Objekten gebunden.

```
Random r  = new Random();
int zuffi = r.nextInt(7);
```

Um die Funktion *nextInt* der Klasse **Random** verwenden zu können, mussten wir als erstes ein Objekt der Klasse erzeugen.

Mit dem Schlüsselwort static können wir nun aus Instanzvariablen und -methoden, unabhängig verwendbare Klassenvariablen und -methoden umwandeln.

```
public class Tasse{
   public static int zaehler=0;
   public Tasse(){
      zaehler++;
   }
   public int getZaehler(){
      return zaehler;
   }
}
```

Statische Attribute und Methoden II

Testen wir unser Programm:

```
Tasse t1 = new Tasse();
Tasse t2 = new Tasse();
Tasse t3 = new Tasse();
int k = t1.getZaehler();
```

Die Variable k hat nach dem dreifachen Aufruf des Konstruktors den Wert 3. Unabhängig welche Instanz wir dazu befragen würden. Die statische Variable *zaehler* existiert nur einmal und wird von allen Instanzen geteilt.

Das gleiche gilt für Funktionen:

```
public class MeinMathe{
   public static int max(int a, int b){
      return a<b ? b : a;
   }
}
```

In Java gibt es die Möglichkeit ein **if**-Statement folgendermaßen auszudrücken:

<Bedingung> ? <Anweisung wenn Bedingung true> : <Anweisung wenn false>

Statische Attribute und Methoden III

In der Klasse **MeinMathe** könnten beispielsweise verschiedene Funktionen zusammengetragen werden.

Da nun die Funktion statisch gemacht wurde, können wir sie verwenden, ohne ein Objekt der Klasse **MeinMathe** zu erzeugen:

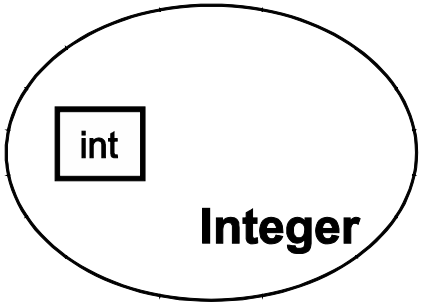
```
int d = MeinMathe.max(3, 22);
```

Sollten wir das static weg lassen, so müssen wir wieder ein Objekt erzeugen und können durch die Referenz an die Funktion gelangen:

```
MeinMathe m = new MeinMathe();
int d = m.max(3, 22);
```

Primitive Datentypen und ihre Wrapperklassen I

Zu jedem primitiven Datentyp gibt es eine entsprechende Wrapperklasse (=Hülle). In dieser Klasse werden unter anderem eine Reihe von Konvertierungsmethoden angeboten.



Ein paar Beispiele:

```
int a     = 4;
Integer i = new Integer(a);
int b     = i.intValue();

String s  = i.toString();
String l  = "7";
Integer k = new Integer(l);
```

Primitive Datentypen und ihre Wrapperklassen II

Die Wrapperklassen für die primitiven Datentypen heißen: **Boolean**, **Character**, **Byte**, **Short**, **Integer**, **Long**, **Float** und **Double**. Neben den Konvertierungsmethoden liefern die Wrapperklassen auch die Randwerte ihres entsprechenden Wertebereichs.

```
byte  b_min = Byte.MIN_VALUE;  // liefert den kleinstmöglichen byte (-128)
float f_max = Float.MAX_VALUE;
```

Die Klasse String I

Ein **String** repräsentiert eine Zeichenkette. Objekte vom Typ **String** sind nach der Initialisierung nicht mehr veränderbar.

```
String name1     = "Frodo";
String name2     = new String("Bilbo");
String name3     = "Beutlin";

// Verkettung von Zeichenketten
String zusammen1 = name3 + ", " + name2;
String zusammen2 = name3.concat(", ") + name1;
String zusammen3 = (name3.concat(", ")).concat(name1);
System.out.println("zusammen1: "+zusammen1); // Beutlin, Bilbo
System.out.println("zusammen2: "+zusammen2); // Beutlin, Frodo
System.out.println("zusammen3: "+zusammen3); // Beutlin, Frodo

// Laenge einer Zeichkette ermitteln
int laenge = name1.length();

// Teile einer Zeichenkette extrahieren
String teil = name1.substring(2, 5);  // Zeile 17
System.out.println("teil: "+teil);
```

Die erste Zeile zeigt uns eine angenehme Eigenschaft von Java. Wir müssen nicht jedes Mal den String-Konstruktor aufrufen, wenn eine Zeichenkette erzeugt werden soll.

Vergleich von Zeichenketten I

Wenn wir uns daran erinnern, dass wir es auch in diesem Beispiel mit Referenzvariablen zu tun haben, dann erklärt es sich einfach, dass Stringvergleiche einer Methode bedürfen.

```
String name1 = "Hugo";
String name2 = "Hugo";
if (name1 == name2)
   ...
```

Die beiden Variablen *name1* und *name2* sind Referenzvariablen. Sie repräsentieren jeweils eine Adresse im Speicher. Wenn wir nun versuchen diese Adressen zu vergleichen, erhalten wir selbstverständlich ein false.
Es sind ja zwei verschiedene Objekte. Was wir aber eigentlich mit einem Vergleich meinen ist die Überprüfung des Inhalts beider Strings.

```
String name1 = "Hugo";
String name2 = "Hugo";
if (name1.equals(name2))
   ...
```

Mit der von der Stringklasse angebotenen Funktion *equals* lassen sich Strings auf ihren Inhalt vergleichen. Nun liefert der Vergleich auch true.

So war es früher!!!

Vergleich von Zeichenketten II

Jetzt geht es allerdings doch! Grund dafür ist der Stringpool. Ein Beispiel für Integer:

```
public class ReferenzenTest{
    public static void main(String[] args){
        // Beispiel 1:
        Integer i = new Integer(1);
        Integer j = new Integer(1);

        // Fehlerhafter Referenzvergleich
        if (i==j)
            System.out.println("TRUE");
        else
            System.out.println("FALSE");

        // korrekter Referenzvergleich
        if (i.intValue()==j.intValue())
            System.out.println("TRUE");
        else
            System.out.println("FALSE");

        ...
```

Vergleich von Zeichenketten III

Wir vergleichen Zeichenketten in Beispiel 2:

```
        ...
        // Beispiel 2: Referenzvergleich verwendet Stringpool
        String a = "Hugo";
        String b = "Hugo";

        // (fehlerhafter) Referenzvergleich im Buch beschrieben
        if (a==b)
            System.out.println("TRUE");
        else
            System.out.println("FALSE");

        // korrekter Referenzvergleich
        if (a.equals(b))
            System.out.println("TRUE");
        else
            System.out.println("FALSE");
    }
}
```

und erhalten:

```
C:\JavaCode>java ReferenzenTest
FALSE
TRUE
TRUE
TRUE
```

Es geht also doch!

Objektorientierte Sicht auf die ersten Konzepte

Vergleich von Zeichenketten IV

Allerdings muss man aufpassen, wenn der String durch new erzeugt wird!

```
public class ReferenzenTest{
    public static void main(String[] args){
        String a = new String("Hugo");
        String b = new String("Hugo");

        if (a==b)
            System.out.println("TRUE");
        else
            System.out.println("FALSE");

        if (a.equals(b))
            System.out.println("TRUE");
        else
            System.out.println("FALSE");
    }
}
```

und erhalten:

```
C:\JavaCode>java ReferenzenTest
FALSE
TRUE
```

Dann klappt es nicht!

Praktische Beispiele

Inhalt:

- Kryptographie
  - Verschiebung (Caesar)
  - XOR-Codierung
- Verwendung von Zufallszahlen
- Dynamischer Datentyp Vector
- Projekt: Black Jack

Ratz D., et.al.: „*Grundkurs Programmieren in Java*“, 4.Auflage, Hanser-Verlag 2007
Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*“, Springer-Verlag 2007

Praktische Beispiele

Abstrakte Encoder-Klasse

Zwei Methoden werden bei der Verschlüsselung benötigt:

```
public abstract class Encoder {
    // verschlüsselt einen String
   public abstract String encode(String s);

   // entschlüsselt einen String
   public abstract String decode(String s);
}
```

Caesar-Codierung I

Durch Verschiebung (Schlüssel) der Buchstaben im Alphabet kann ein Text codiert und wieder decodiert werden:

```
public class CAESAR_Codierer extends Encoder{
  // geheimer Schlüssel
  private int key;

  // Definition des Alphabets
  public static final int ALPHABETSIZE = 26;
  public static final char[] alpha =
            {'A','B','C','D','E','F','G','H','I',
             'J','K','L','M', 'N','O','P','Q','R',
             'S','T','U','V','W','X','Y','Z'};

  // Übersetzungslisten für die Buchstaben
  protected char[] encrypt = new char[ALPHABETSIZE];
  protected char[] decrypt = new char[ALPHABETSIZE];

  public CAESAR_Codierer(int key) {
    this.key = key;
   computeNewAlphabet();
  }

  private void computeNewAlphabet(){
    for (int i=0; i<ALPHABETSIZE; i++)
      encrypt[i] = alpha[(i + key) % ALPHABETSIZE];

    for (int i=0; i<ALPHABETSIZE; i++)
      decrypt[encrypt[i] - 'A'] = alpha[i];
  }
```

Caesar-Codierung II

Jetzt lassen sich die Methoden encode und decode sehr einfach implementieren:

```
  // ************************************************************************
  // Encoder-Funktionen
  public String encode(String s) {
    char[] c = s.toCharArray();

    for (int i=0; i<c.length; i++)
      if (Character.isUpperCase(c[i]))
        c[i] = encrypt[c[i] - 'A'];

    return new String(c);
  }

  public String decode(String s){
    char[] c = s.toCharArray();

    for (int i=0; i<c.length; i++)
      if (Character.isUpperCase(c[i]))
        c[i] = decrypt[c[i] - 'A'];

    return new String(c);
  }
  // ************************************************************************
}
```

XOR-Verschlüsselung I

Jetzt lernen wir das XOR kennen und verwenden es gleich, um einen Text zu verschlüsseln:

B1	B2	B1 XOR B2
0	0	0
0	1	1
1	0	1
1	1	0

In Java wird das XOR durch den Operator „^“ repräsentiert.

```
boolean a, b, c;
a = true;
b = false;
c = a ^ b;
```

XOR-Verschlüsselung II

Alle Zeichen werden über die binäre Operation xor (exklusives Oder) verknüpft. Dabei werden alle Zeichen als binäre Zahlen aufgefasst:

```
public class XOR_Codierer extends Encoder{
   // hier wird der geheime Schlüssel abgelegt
   private int key;

   public XOR_Codierer(int k){
      key = k;
   }

   // verschlüsselt durch XOR-Operation mit key
   // die Zeichen der Zeichenkette s
   public String encode(String s) {
      char[] c = s.toCharArray();

      for (int i=0; i<c.length; i++)
         c[i] = (char)(c[i]^key);

      return new String(c);
   }

   // entschlüsselt mit Hilfe der Funktion
   // encode die Zeichenkette s
   public String decode(String s){
      return encode(s);
   }
}
```

XOR-Verschlüsselung III

Jetzt wollen wir den Encoder testen:

```
public class Demo{
   public static void demo(Encoder enc, String text) {
      String encoded = enc.encode(text);
      System.out.println("codiert  : " + encoded);
      String decoded = enc.decode(encoded);
      System.out.println("decodiert: " + decoded);

      if (text.equals(decoded))
         System.out.println("Verschluesselung erfolgreich!");
      else
         System.out.println("PROGRAMMFEHLER!");
   }

   public static void main(String[] args){
      int key     = 1;
      String text = "";
      try{
         key   = Integer.parseInt(args[0]);
         text  = args[1];
      } catch(Exception e){
         System.out.println("Fehler ist aufgetreten!”);
         System.out.println(“Bitte nochmal Demo <int> <String> aufrufen.");
      }

      Encoder enc = new XOR_Codierer(key);
      demo(enc, text);
   }
}
```

Praktische Beispiele

Zufallszahlen

Es gibt verschiedene Möglichkeiten Zufallszahlen zu verwenden. Oft benötigt man sie als Wahrscheinlichkeitsmaß im Intervall [0,1]. In anderen Fällen ist es wünschenswert aus einer Menge A mit n Elementen eines auszuwählen {1, 2, ..., n}.

Wir unterscheiden zunächst einmal den Datentyp der Zufallszahl. In jedem Fall verwenden wir die Klasse aus dem Package java.util.

```
import java.util.Random;
...
```

Um eine der Klassenmethoden verwenden zu können, erzeugen wir eine Instanz der Klasse **Random**:

```
...
Random randomGenerator = new Random();
...
```

Praktische Beispiele

Ganzzahlige Zufallszahlen vom Typ int und long

Das kleine Lottoprogramm (6 aus 49) dient als Beispiel für die Erzeugung der Funktion *nextInt(n)*. Es werden Zufallszahlen aus dem Bereich [0, 1, ..., n-1] gewählt. Wenn Zahlen aus dem Bereich long benötigt werden, so kann die Funktion *nextLong(n)* analog verwendet werden.

```
import java.util.*;

public class Lotto {
   public static void main(String[] args)
   {
      Random rg = new Random();
      int[] zuf = new int[6];

      System.out.print("Lottotipp (6 aus 49): ");
      int wert, i=0;

      aussen:
      while(i<6){
         wert = rg.nextInt(49) + 1; // +1 da nicht 0,...,48 sondern 1,...,49

         // schon vorhanden?
         for (int j=0; j < i; j++)
            if (zuf[j]==wert)
               continue aussen;

         zuf[i] = wert;
         i++;
         System.out.print(wert + " ");
      }
   }
}
```

Praktische Beispiele

Ganzzahlige Zufallszahlen vom Typ float und double

Für die Erzeugung einer Zufallszahl aus dem Intervall [0,1] gibt es eine kürzere Schreibweise. In der Klasse **Math** im Package java.lang gibt es eine statische Funktion *random*, die eine Instanz der Klasse **Random** erzeugt, die Funktion *nextDouble* aufruft und den erzeugten Wert zurückliefert.

Wir schreiben lediglich die folgende Zeile:

```
double zuffi = Math.random();
```

Bei der Initialisierung der Klasse **Random** gibt es zwei Varianten. Die erste mit dem parameterlosen Konstruktor initialisiert sich in Abhängigkeit zur Systemzeit und erzeugt bei jedem Start neue Zufallszahlen. Für Programme, bei denen beispielsweise zeitkritische Abschnitte getestet werden, die aber abhängig von der jeweiligen Zufallszahl sind oder Experimente, bei denen die gleichen Stichproben verwendet werden sollen, ist der Konstruktor mit einem long als Parameter gedacht.

Wir können beispielsweise einen long mit dem Wert 0 immer als Startwert nehmen und erhalten anschließend immer dieselben Zufallszahlen:

```
long initwert          = 0;
Random randomGenerator = new Random(initwert);
```

Praktische Beispiele

Das Kartenspiel Black Jack

Jetzt wollen wir ein kleines Kartenspiel realisieren.

Spieler

Kartenspiel

Karte

BlackJack

(Abbildung von Wikipedia)

Das Spiel Black Jack I

Für unsere einfache Implementierung benötigen wir die Klassen **Spieler**, **Karte**, **Kartenspiel** und **BlackJack**.
Bevor wir jedoch mit dem Programmieren beginnen, werden wir uns mit den Spielregeln vertraut machen:

Gespielt wird mit einem 52er-Blatt, also 2, 3, ..., 10, Bube, Dame, Koenig, Ass für Karo , Herz, Pik und Kreuz.

Wertigkeiten der Karten

1. Asse zählen nach belieben ein oder elf Punkte.
2. Zweier bis Zehner zählen entsprechend ihren Augen zwei bis zehn Punkte.
3. Bildkarten (Buben, Damen, Könige) zählen zehn Punkte.

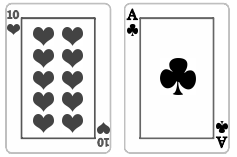
Praktische Beispiele

Das Spiel Black Jack II

Der „vereinfachte“ Spielverlauf

Wir spielen mit vereinfachten Regeln. Vor Beginn eines Spiels platziert der Spieler seinen Einsatz. Dann werden zwei Karten an den Dealer und zwei an den Spieler vergeben. Die Wertigkeiten der jeweils zwei Karten werden zusammengezählt. Hier kann sich schon entschieden haben, wer gewonnen hat:

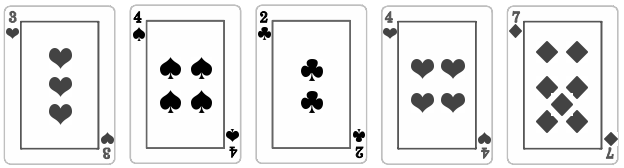
1. Hat der Dealer bereits 21 Punkte erreicht, verliert der Spieler automatisch.
2. Trifft 1 nicht zu und hat der Spieler 21 Punkte, so gewinnt er mit BlackJack.



Wertigkeit 21

Nun beginnt der Spieler solange neue Karten anzufordern, bis er meint genug Karten zu haben. Die Summe der Wertigkeiten darf dabei 21 nicht überschreiten, sonst hat er verloren und die Einsätze gehen an die Bank. Hat der Spieler genug Karten, beginnt der Dealer seinerseits Karten anzufordern. Auch hier gilt, überschreitet die Summe der Wertigkeiten die 21, so hat der Dealer verloren und das Geld geht an den Spieler. Nimmt der Dealer jedoch keine Karte mehr auf, gelten folgende Entscheidungen:

1. Hat der Dealer 5 Karten in der Hand, verliert der Spieler.
2. Ist die Summe der Wertigkeiten beim Dealer gegenüber dem Spieler größer oder gleich, gewinnt der Dealer.
3. Ansonsten gewinnt der Spieler.



Wertigkeit 20

Praktische Beispiele

Der dynamische Datentyp Vector I

Im Gegensatz zu einem Array, bei dem die Anzahl der Elemente bei der Initialisierung festgelegt wird, verhält sich der von Java angebotene Datentyp Vector dynamisch. Wenn wir also vor der Verwendung einer Liste die Anzahl der Elemente nicht kennen, können wir diesen Datentyp nehmen.

Ein kleines Beispiel dazu:

```
import java.util.Vector;
public class VectorTest{
   public static void main(String[] args){
      Vector v = new Vector();
      for (int i=0; i<4; i++) // füge nacheinander Elemente in den Vector ein
         v.addElement(new Integer(i));
      System.out.println("Vector size = "+v.size()); // Anzahl der Elemente im Vector v

      // Auslesen der aktuellen Inhalts
      for (int i=0; i<v.size(); i++){
         Integer intObjekt = (Integer)v.elementAt(i);
         int wert = intObjekt.intValue();
         System.out.println("Element "+i+" = "+wert);
      }
      System.out.println();
      v.insertElementAt(new Integer(9), 2); // wir geben ein neues Element hinzu
      v.removeElementAt(4); // und löschen ein Element

      for (int i=0; i<v.size(); i++) // Auslesen der aktuellen Inhalts
         System.out.println("Element "+i+" = "
                     +((Integer)v.elementAt(i)).intValue());
   }
}
```

Der dynamische Datentyp Vector II

Unser Beispiel liefert folgende Ausgabe:

```
C:\JavaCode>java VectorTest
Vector size = 4
Element 0 = 0
Element 1 = 1
Element 2 = 2
Element 3 = 3

Element 0 = 0
Element 1 = 1
Element 2 = 9
Element 3 = 2
```

Die Klasse Spieler I

Vom Spieler wollen wir noch erfahren können, welchen Wert seine „Hand`` besitzt. Schauen wir uns die Klasse **Spieler** einmal an:

```
import java.util.Vector;

public class Spieler {
   private String spielerName;
   private int geld;
   private Vector karten;

   // Konstruktor
   public Spieler(String n, int g) {
      spielerName     = n;
      geld            = g;
      karten          = new Vector();
   }

   // Funktionen (get und  set)
   ...

   public void clear() {
      karten.removeAllElements();
   }

   public void addKarte(Karte k) {
      karten.addElement(k);
   }
   ...
```

Die Klasse Spieler II

weiter gehts:

```
...
public int getAnzahlKarten() {
   return karten.size();
}

// Spieler-Methoden
public Karte getKarte(int p) {
   if ((p >= 0) && (p < karten.size()))
      return (Karte)karten.elementAt(p);
   else
      return null;
}
...
```

Die Klasse Spieler III

Die Methode *aktuelleBewertung*:

```
   ...
   public int aktuelleWertung() {
      int wert, anzahl;
      boolean istAss;

      wert      = 0;
      istAss    = false;
      anzahl    = getAnzahlKarten();
      Karte karte;
      int kartenWert;

      // wir durchlaufen unseren aktuellen Kartenstapel
      for (int i=0; i<anzahl; i++) {
         karte          = getKarte(i);          // karte ist Instanz der Klasse Karte
         kartenWert     = karte.getWert();      // Funktion getWert liefert den Wert der karte

         // Bewertung der Bilder
         if (kartenWert > 10) kartenWert = 10;
         if (kartenWert == 1) istAss = true;    // mindestens 1 Punkt für Ass

         wert += kartenWert;
      }

      // Ass-Wert selber bestimmen
      if (istAss && (wert + 10 <= 21))          // oder lieber 10 Punkte für Ass?
         wert = wert + 10;

      return wert;
   }
}
```

Praktische Beispiele

Die Klasse Karte I

Es ist sinnvoll auch für eine Spielkarte eine Klasse zu entwerfen, dass macht unser Programm transparent und wiederverwendbar. Eine Karte kann jede beliebige aus einem Pokerkartenspiel mit 52 Karten sein:

```
public class Karte {
   // Bewertung der Karten und Definition der Farben
   private final static int KARO=0, HERZ = 1, PIK   =2, KREUZ=3;
   private final static int BUBE=11, DAME=12, KOENIG=13, ASS =1;
   private final int farbe, wert;

   // Konstruktor
   public Karte(int f, int w) {
      farbe = f;
      wert      = w;
   }

   // Funktionen (get und set)   ...

   // Karten-Methoden
   public String farbe2String() {
      switch (farbe) {
         case KARO:
                return "Karo";
         case HERZ:
                return "Herz";
         case PIK:
                return "Pik";
         case KREUZ:
                return "Kreuz";
      }
      System.out.println("Farbe falsch! : "+farbe);
      return "-1";
   }
```

Die Klasse Karte II

weiter gehts:

```
   ...
   public String wert2String() {
      if ((wert>=2)&&(wert<=10))
         return ""+wert;

      switch (wert) {
         case 1:
                return "A";
         case 11:
                return "B";
         case 12:
                return "D";
         case 13:
                return "K";
        }
      return "-1";
   }

   public String Karte2String() {
      return farbe2String() + "-" + wert2String();
   }
}
```

Zusätzlich zu den get-set-Funktionen gibt es noch drei Ausgabemethoden, um die interne Repräsentation der Karten (0,...,12) in eine lesbare Form zu übersetzen (2,3,..,B,D,K,A).

Praktische Beispiele

Die Klasse Kartespiel I

Die Karten können wir zu der Klasse **Kartenspiel** zusammenfassen und Methoden zum Mischen und Herausgeben von Karten anbieten.

```
import java.util.*;

public class KartenSpiel {
   // 52er Kartenstapel (2-10,B,D,K,A für Karo,Herz,Pik und Kreuz)
   private Karte[] stapel;
   private int kartenImSpiel;

   // Konstruktor
   public KartenSpiel() {
      stapel = new Karte[52];
      int zaehler = 0;
      for (int f=0; f<4; f++ ) {
         for (int w=1; w<14; w++ ) {
            stapel[zaehler] = new Karte(f, w);
            zaehler++;
         }
      }
      mischen();
   }
   ...
```

Die Klasse Kartespiel I

weiter geht's:

```
   ...
   // KartenSpiel-Methoden
   public void mischen() {
     Karte temp;
     kartenImSpiel = 52;
     Random rg = new Random();
     for (int i=kartenImSpiel-1; i>=0; i--) {
        int zuff    = rg.nextInt(kartenImSpiel);
        if (zuff != i) {
            temp           = stapel[i];          // tausche
            stapel[i]      = stapel[zuff];
            stapel[zuff]   = temp;
        }
     }
   }

   public int kartenAnzahl() {
      return kartenImSpiel;
   }

   public Karte gibEineKarte() {
      if (kartenImSpiel == 0)
         mischen();
      kartenImSpiel--;
      return stapel[kartenImSpiel];
    }
}
```

In der Methode *mischen* kommen unsere Zufallszahlen ins Spiel. Wir gehen alle Karten des Spiels durch, ermitteln jeweils eine zufällige Position im Stapel und tauschen die beiden Karten an diesen Positionen.

Die Klasse BlackJack I

Jetzt wollen wir zur eigentlichen Spielklasse **BlackJack** kommen. Diese Klasse verwaltet Spieler und Dealer, die jeweiligen Geldbeträge und den gesamten Spielablauf:

```
import java.io.*;

public class BlackJack{
   private KartenSpiel kartenSpiel;
   private Spieler spieler, dealer;
   private int einsatz;
   private boolean spiellaeuft;

   // Konstruktor
   public BlackJack(String n){
      kartenSpiel = new KartenSpiel();
      spieler     = new Spieler(n, 500);
      dealer      = new Spieler("Dealer", 10000);
      einsatz     = 0;
      spiellaeuft = false;
   }

   // Funktionen (get und set)
   ...

   public boolean getSpielStatus(){
      return spiellaeuft;
   }
   ...
```

Die Klasse BlackJack II

weiter geht's :

```
...
// BlackJack-Methoden
public void neuesSpiel(){
   spieler.clear();
   dealer.clear();
   spieler.addKarte(kartenSpiel.gibEineKarte());
   dealer.addKarte(kartenSpiel.gibEineKarte());
   spieler.addKarte(kartenSpiel.gibEineKarte());
   dealer.addKarte(kartenSpiel.gibEineKarte());

   spiellaeuft = true;
}

public void neueKarte(){
   spieler.addKarte(kartenSpiel.gibEineKarte());
}

public void dealerIstDran(){
   while ((dealer.aktuelleWertung()<=16)&&
          (dealer.getAnzahlKarten()<5))
      dealer.addKarte(kartenSpiel.gibEineKarte());
}
...
```

Praktische Beispiele

Die Klasse BlackJack III

weiter geht's :

```
...
public boolean erhoeheEinsatz(){
   if (dealer.getGeld()-50>=0){
      dealer.setGeld(dealer.getGeld()-50);
      einsatz+=50;
   }
   else {
      System.out.println();
      System.out.println("WOW! DU HAST DIE BANK PLEITE GEMACHT!");
      System.out.println();
      System.exit(1);
   }
   if (spieler.getGeld()-50>=0){
      spieler.setGeld(spieler.getGeld()-50);
      einsatz+=50;
      return true;
   }
   return false;
}
...
```

Die Klasse BlackJack IV

Wir wollen noch drei statische Ausgabemethoden *hilfe*, *ausgabeKartenSpieler* und *kontoDaten*, die unabhängig von der Klasse **BlackJack** sind, definieren.

```
...
// statische Methoden
private static void hilfe(){
   System.out.println();
   System.out.println("Eingaben: ");
   System.out.println("   n = eine neue Karte");
   System.out.println("   d = fertig, Dealer ist dran");
   System.out.println("   + = Einsatz um 50$ erhoehen");
   System.out.println("   r = neue Runde");
   System.out.println("   x = Spiel beenden");
   System.out.println("   ? = Hilfe");
   System.out.println();
}
...
```

Die Klasse BlackJack V

Wir wollen noch drei statische Ausgabemethoden *hilfe*, *ausgabeKartenSpieler* und *kontoDaten*, die unabhängig von der Klasse **BlackJack** sind, definieren.

```
...
private static void ausgabeKartenSpieler(Spieler s, Spieler d){
   System.out.println();
   System.out.print("Du erhaelst: ");
   for (int i=0; i<s.getAnzahlKarten(); i++) {
      Karte karte    = s.getKarte(i);
      System.out.print(karte.karte2String()+" ");
   }
   System.out.println("(Wertung="+s.aktuelleWertung()+")");
   System.out.print("Der Dealer erhaelt: ");
   for (int i=0; i<d.getAnzahlKarten(); i++) {
      Karte karte    = d.getKarte(i);
      System.out.print(karte.karte2String()+" ");
   }
   System.out.println("(Wertung="+d.aktuelleWertung()+")");
   System.out.println();
}

private static void kontoDaten(Spieler s, Spieler d){
   System.out.println();
   System.out.println("$$$ "+s.getName()+": "+s.getGeld()+", Bank: "
                      +d.getGeld()+" $$$");
   System.out.println();
}
...
```

Die Klasse BlackJack VI

Für unser einfaches BlackJack-Spiel soll es genügen, den Spielbetrieb in die main-Methode einzufügen. Das macht die Methode zwar sehr lang, aber das Programm insgesamt relativ kurz. Nach einer Willkommenszeile wird die Eingabe des Spielernames verlangt und schon kann das Spiel mit einem Einsatz begonnen werden.

```
...
public static void main(String[] args){
   System.out.println("-----------------------------------------");
   System.out.println("-  WILLKOMMEN zu einem Spiel BlackJack!   -");
   System.out.println("-----------------------------------------");
   hilfe();

   InputStreamReader stdin = new InputStreamReader(System.in);
   BufferedReader console  = new BufferedReader(stdin);

   System.out.print("Geben Sie Ihren Namen an: ");
   String name = "";
   try {
       name = console.readLine();
   } catch(IOException ioex){
      System.out.println("Eingabefehler");
      System.exit(1);
   }

   System.out.println();
   System.out.println("Hallo "+name
               +", Dir stehen 500$ als Kapitel zur Verfuegung.");
   System.out.println("Mach Deinen Einsatz(+) und
                       beginne das Spiel(r).");
   System.out.println();
   ...
```

Die Klasse BlackJack VII

Weiter geht's mit der main:

```
...
// Nun starten wir eine Runde BlackJack
BlackJack blackjack = new BlackJack(name);

kontoDaten(blackjack.getSpieler(), blackjack.getDealer());
```

Der Spielbetrieb läuft in einer while-Schleife und kann durch die Eingabe 'x' beendet werden. Die zur Verfügung stehenden Eingaben sind: 'n' für eine neue Karte, 'd' für Spielerzug ist beendet und Dealer ist dran, '+' erhöht den Einsatz um 50$, 'r' eine neue Runde wird gestartet, '?' zeigt die zur Verfügung stehenden Eingaben und wie bereits erwähnt, beendet 'x' das Spiel.

Die Klasse BlackJack VIII

Weiter geht's mit der main:

```
boolean istFertig = false;
String input="";
while (!istFertig){
   try {
      input = console.readLine();
   } catch(IOException ioex){  System.out.println("Eingabefehler");  }
   if (input.equals("n")){
      // eine zusätzliche Karte bitte
      if (blackjack.getSpielStatus()) {
         blackjack.neueKarte();
         if (blackjack.getSpieler().aktuelleWertung()>21){
            System.out.println("Du hast verloren! Deine Wertung
                     liegt mit "+
                     blackjack.getSpieler().aktuelleWertung()+
                     " ueber 21.");
            blackjack.getDealer().setGeld(
                     blackjack.getDealer().getGeld()+
                     blackjack.einsatz);
            blackjack.einsatz       = 0;
            blackjack.spiellaeuft   = false;
            kontoDaten(blackjack.getSpieler(),
                      blackjack.getDealer());
         }
         else {
            ausgabeKartenSpieler(blackjack.getSpieler(),
                      blackjack.getDealer());
         }
      }
   }
```

Die Klasse BlackJack IX

Nachdem der Spieler genug Karten genommen hat, wird mit d der Zug abgeschlossen und der Dealer ist dran. Dabei wurde bereits geprüft, ob die Wertung der Hand des Spielers über 21 lag.

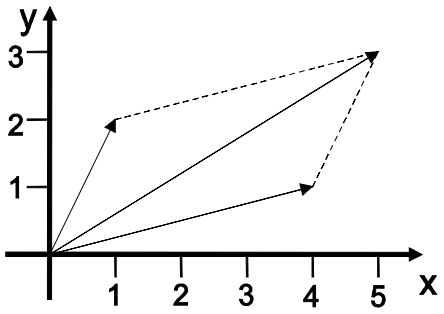
DEN REST DES BLACKJACK-PROGRAMMS FINDET IHR HIER:
http://www.java-uni.de/index.php?Seite=11

Praktische Beispiele

Lineare Algebra I

Es gibt viele nützliche Bibliotheken, die wir verwenden können. JAMA ist beispielsweise die meist verwendete Bibliothek für Methoden der Linearen Algebra. Hier ein Beispiel zur Vektoraddition:

```
import Jama.*;
public class JAMATest{
   public static void main(String[] args){
      double[][] vector1 = {{1},{2}};
      double[][] vector2 = {{4},{1}};

      Matrix v1 = new Matrix(vector1);
      Matrix v2 = new Matrix(vector2);

      Matrix x  = v1.plus(v2);

      System.out.println("Matrix x: ");
      x.print(1, 2);
   }
}
```



Praktische Beispiele

Lineare Algebra II

Nun wollen wir die Determinante einer Matrix berechnen.

```
import Jama.*;
public class JAMATest{
   public static void main(String[] args){
      double[][] array = {{-2,1},{0,4}};
      Matrix a = new Matrix(array);
      double d = a.det();

      System.out.println("Matrix a: ");
      a.print(1, 2);

      System.out.println(“Determinante: ” + d);
   }
}
```

det(a) = (-2)*4 – 1*0 = -8

Praktische Beispiele

Installation der JAMA-Bibliothek

Um JAMA zu installieren, gehen wir zunächst zu dem aufgeführten Link:
http://math.nist.gov/javanumerics/jama/

Wir speichern nun das Zip-Archiv vom Source (zip archive, 105Kb) z.B. in den Ordner "c:\Java\". Jetzt ist das Problem, dass die bereits erstellten .class-Dateien nicht zu unserem System passen. Desshalb müssen die sechs ".class" Dateien im Ordner *Jama* löschen und zusätzlich das Class-File im Ordner *util*. Jetzt ist es quasi clean.

Nun gehe in den Ordner "c:\Java" und gebe folgendes ein:

```
C:\Java>javac Jama/Matrix.java
  Note: Jama\Matrix.java uses unchecked or unsafe operations.
  Note: Recompile with -Xlint:unchecked for details.J
```

Jetzt wurde das Package erfolgreich compilert.

Informatikerwitz:
Es gibt 10 Gruppen von Menschen: diejenigen, die das Binärsystem verstehen, und die anderen.

Eine eigene Bibliothek bauen I

Die Erzeugung eines eigenen Packages unter Java ist sehr einfach. Wichtig ist das Zusammenspiel aus Klassennamen und Verzeichnisstruktur. Angenommen wir wollen eine Klasse **MeinMax** in einem Package meinMathe anbieten.

Dann legen wir ein Verzeichnis meinMathe an und speichern dort z.B. die folgende Klasse:

```
package meinMathe;

public class MeinMax{
   public static int maxi(int a, int b){
      if (a<b) return b;
      return a;
   }
}
```

Durch das Schlüsselwort package haben wir signalisiert, dass es sich um eine Klasse des Package meinMathe handelt. Unsere kleine Matheklasse bietet eine bescheidene *maxi*-Funktion.

Nachdem wir diese Klasse mit javac compiliert haben, können wir ausserhalb des Ordners eine neue Klasse schreiben, die dieses Package jetzt verwendet.

Eine eigene Bibliothek bauen II

Dabei ist darauf zu achten, dass der Ordner des neuen Packages entweder im gleichen Ordner wie die Klasse liegt, die das Package verwendet, oder dieser Ordner im PATH aufgelistet ist.

Hier unsere Testklasse:

```
import MeinMathe.MeinMax;

public class MatheTester{
   public static void main(String[] args){
      System.out.println("Ergebnis = "+MeinMax.maxi(3,9));
   }
}
```

Wir erhalten nach der Ausführung folgende Ausgabe:

```
C:\JavaCode>java MatheTester
Ergebnis = 9
```

Grafische Benutzeroberflächen

Inhalt:

- Fenstermanagement unter AWT
- Zeichenfunktionen
- Die Klasse Color
- Fensterereignisse

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007

Ein Fenster erzeugen I

Wir können schon mit wenigen Zeilen ein Fenster anzeigen, indem wir eine Instanz der Klasse **Frame** erzeugen und sie sichtbar machen. Bevor wir die Eigenschaft , „ist sichtbar“ mit *setVisible*(true) setzen, legen wir mit der Funktion *setSize*(breite, hoehe) die Fenstergröße fest.

```
import java.awt.Frame;
public class MeinErstesFenster {
  public static void main(String[] args) {
     // öffnet ein AWT-Fenster
     Frame f = new Frame("So einfach geht das?");
     f.setSize(300, 200);
     f.setVisible(true);
  }
}
```

Nach dem Start öffnet sich folgendes Fenster:

Ein Fenster erzeugen II

Nach der Erzeugung des Fensters ist die Ausgabeposition die linke obere Ecke des Bildschirms.

Das Fenster lässt sich momentan nicht ohne Weiteres schließen. Wie wir diese Sache in den Griff bekommen und das Programm nicht jedes mal mit Tastenkombination STRG+C (innerhalb der Konsole) beenden müssen, sehen wir später. Da das dort vorgestellte Konzept doch etwas mehr Zeit in Anspruch nimmt, experimentieren wir mit den neuen Fenstern noch ein wenig herum.

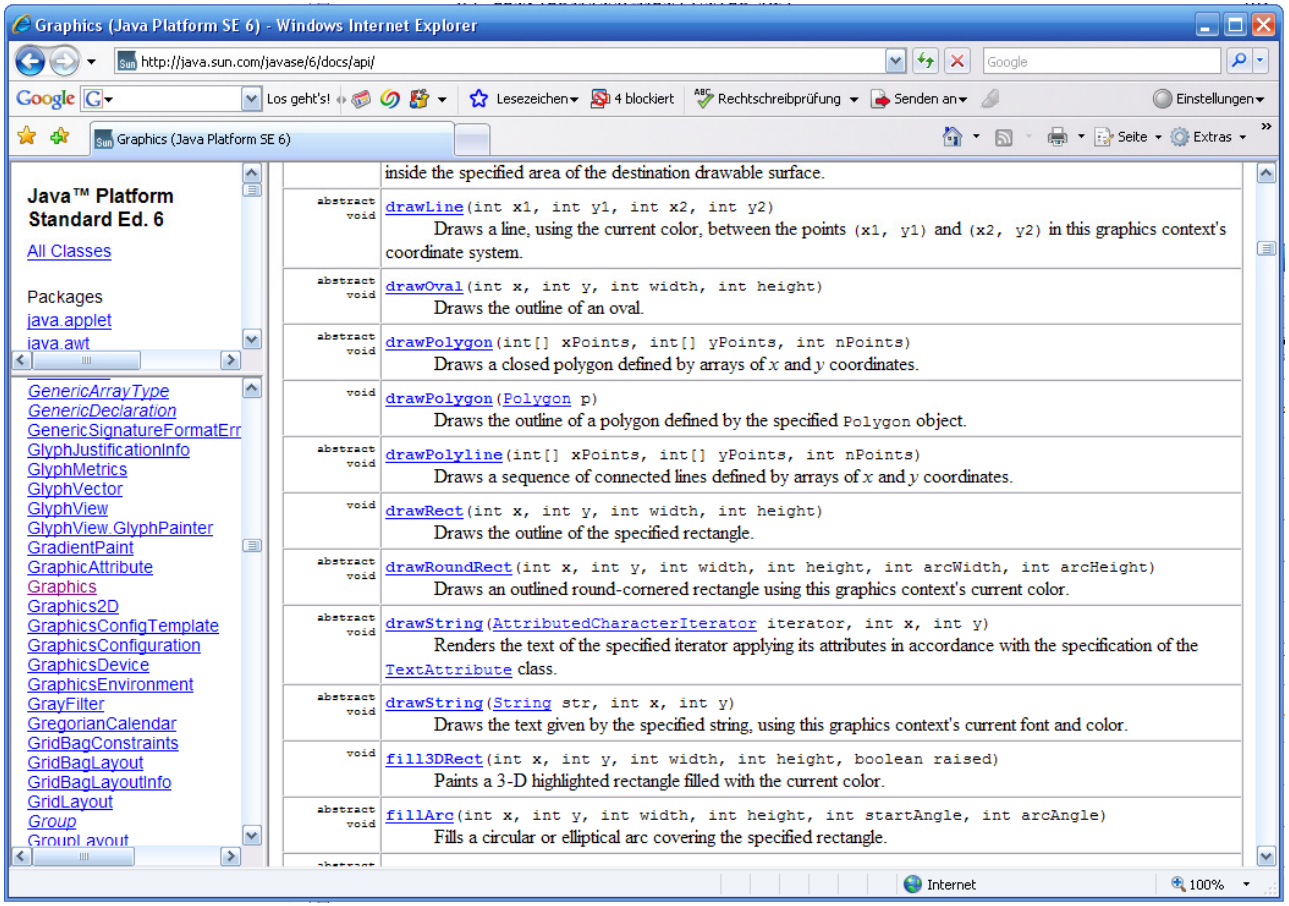
Wir wollen das Fenster zentrieren:

```
import java.awt.*;
public class FensterPositionieren extends Frame {
   public FensterPositionieren(int x, int y){
      setTitle("Ab in die Mitte!");
      setSize(x, y);
      Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
      setLocation((d.width-getSize().width)/2,  (d.height-getSize().height)/2);
      setVisible(true);
   }

   public static void main( String[] args ) {
      FensterPositionieren f = new FensterPositionieren(200, 100);
   }
}
```

Das Fenster wird jetzt mittig auf dem Monitor platziert.

Zeichenfunktionen innerhalb eines Fensters

AWT bietet eine Reihe von Zeichenfunktionen an. Um etwas in einem Fenster anzeigen zu können, müssen wir die Funktion *paint* der Klasse **Frame** überschreiben. Als Parameter sehen wir den Typ **Graphics**. Die Klasse **Graphics** beinhaltet unter anderem alle Zeichenfunktionen. Schauen wir dazu mal in die API:



Textausgaben

Ein Text wird innerhalb des Fensterbereichs ausgegeben. Zusätzlich zeigt dieses Beispiel, wie sich die Vorder- und Hintergrundfarben über die Funktionen *setBackground* und *setForground* manipulieren lassen.

Die Klasse **Color** bietet bereits vordefinierte Farben und es lassen sich neue Farben, bestehend aus den drei Farbkomponenten rot, blau und grün erzeugen.

```
import java.awt.*;
public class TextFenster extends Frame {
   public TextFenster(String titel) {
      setTitle(titel);
      setSize(500, 100);
      setBackground(Color.lightGray);
      setForeground(Color.blue);
      setVisible(true);
   }

   public void paint( Graphics g ){
      g.drawString("Hier steht ein kreativer Text.", 120, 60);
   }

   public static void main( String[] args ) {
      TextFenster t = new TextFenster("Text im Fenster");
   }
}
```

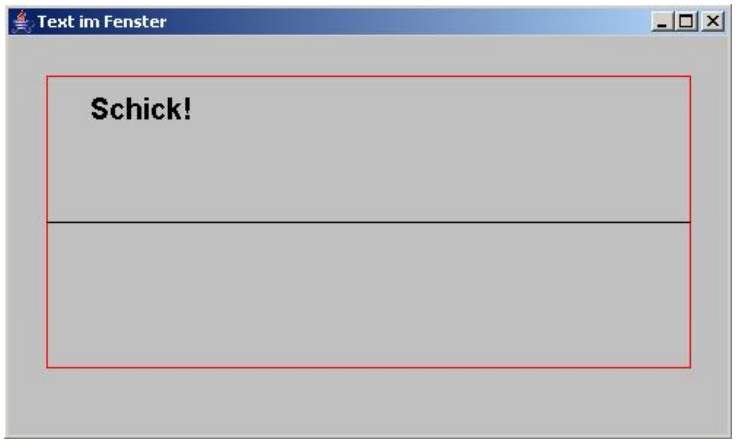
Zeichenelemente I

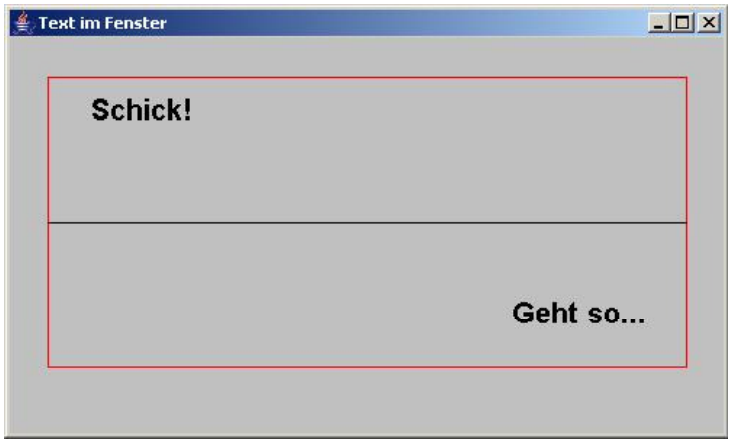
Exemplarisch zeigt dieses Beispiel die Verwendung der Zeichenfunktionen *drawRect* und *drawLine*. Auch hier haben wir eine zusätzliche Funktionalität eingebaut, die Wartefunktion:

```
import java.awt.*;
public class ZeichenElemente extends Frame {
  public ZeichenElemente(String titel) {
      setTitle(titel);
      setSize(500, 300);
      setBackground(Color.lightGray);
      setForeground(Color.red);
      setVisible(true);
  }

  public static void wartemal(long millis){
      try {
        Thread.sleep(millis);
      } catch (InterruptedException e){}
  }

  public void paint( Graphics g ){
      g.drawRect(30,50,440,200);
      g.setColor(Color.black);
      g.drawLine(30,150,470,150);
      g.setFont(new Font("SansSerif", Font.BOLD, 20));
      g.drawString("Schick!", 60, 80);
      wartemal(3000);
      g.drawString("Geht so...", 350, 220);
  }

  public static void main( String[] args ) {
      ZeichenElemente t = new ZeichenElemente("Linien im Fenster");
  }
}
```

Zeichenelemente II

Liefert folgende Ausgabe:





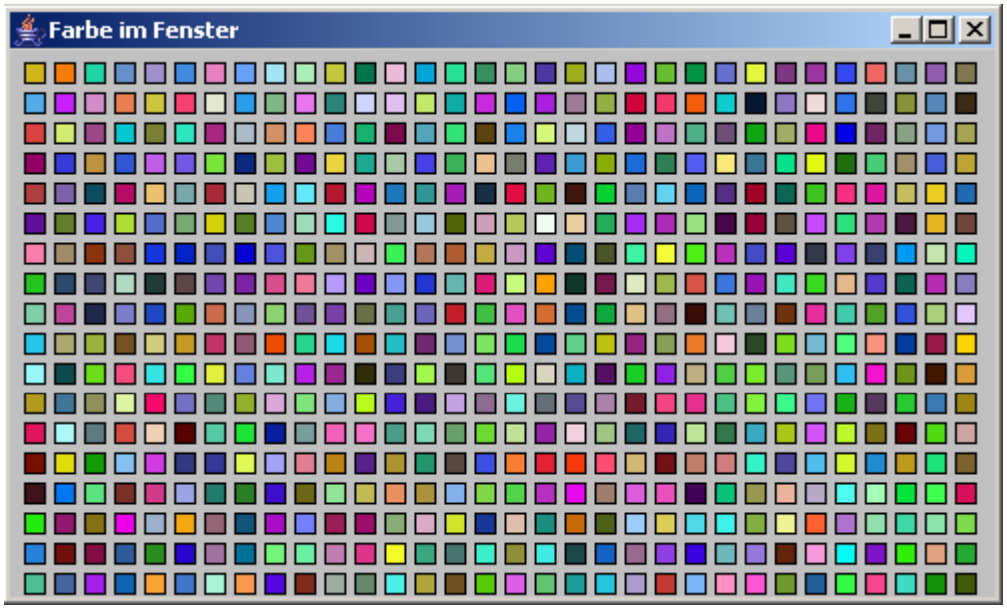
Die Klasse Color I

Für grafische Benutzeroberflächen sind Farben sehr wichtig. Um das RGB-Farbmodell zu verwenden, erzeugen wir viele farbige Rechtecke, deren 3 Farbkomponenten rot, grün und blau zufällig gesetzt werden. Dazu erzeugen wir ein **Color**-Objekt und setzen dessen Farbwerte.

```
import java.awt.*;
import java.util.Random;

public class FarbBeispiel extends Frame {
   public FarbBeispiel(String titel) {
      setTitle(titel);
      setSize(500, 300);
      setBackground(Color.lightGray);
      setForeground(Color.red);
      setVisible(true);
   }

   public void paint(Graphics g){
      Random r = new Random();
      for (int y=30; y<getHeight()-10; y += 15)
         for (int x=12; x<getWidth()-10; x += 15) {
            g.setColor(new Color(r.nextInt(256),
                                 r.nextInt(256),
                                 r.nextInt(256)));
            g.fillRect(x, y, 10, 10);
            g.setColor(Color.BLACK);
            g.drawRect(x - 1, y - 1, 10, 10);
         }
   }

   public static void main(String[] args) {
      FarbBeispiel t = new FarbBeispiel("Viel Farbe im Fenster");
   }
}
```

Die Klasse Color II

Wir erhalten kleine farbige Rechtecke und freuen uns auf mehr.



Bilder laden und anzeigen I

Mit der Methode *drawImage* können wir Bilder anzeigen lassen. In den Zeilen 14 bis 16 geschieht aber leider nicht das, was wir erwarten würden. Die Funktion *getImage* bereitet das Laden des Bildes nur vor. Der eigentliche Ladevorgang erfolgt erst beim Aufruf von *drawImage*. Das hat den Nachteil, dass bei einer Wiederverwendung der Methode *paint* jedes Mal das Bild neu geladen wird.

```
import java.awt.*;
import java.util.Random;

public class BildFenster extends Frame {
  public BildFenster(String titel) {
      setTitle(titel);
      setSize(423, 317);
      setBackground(Color.lightGray);
      setForeground(Color.red);
      setVisible(true);
  }

  public void paint(Graphics g){
      Image pic = Toolkit.getDefaultToolkit().getImage(
                                  "C:\\kreidefelsen.jpg");
      g.drawImage(pic, 0, 0, this);
  }

  public static void main( String[] args ) {
      BildFenster t = new BildFenster("Bild im Fenster");
  }
}
```

Jetzt könnten wir sogar schon eine Diashow der letzten Urlaubsbilder realisieren.

Bilder laden und anzeigen II

Kleine Ausgabe:

Um zu verhindern, dass das Bild neu geladen werden soll, können mit Hilfe der Klasse **Mediatracker** die Bilder vor der eigentlichen Anzeige in den Speicher laden.

Bilder laden und anzeigen III

Eine globale Variable img vom Typ **Image** wird angelegt und im Konstruktor mit dem **Mediatracker** verknüpft. Der **Mediatracker** liest die entsprechenden Bilder ein und speichert sie:

```
// wird dem Konstruktor hinzugefügt
img = getToolkit().getImage("c:\\kreidefelsen.jpg");
Mediatracker mt = new Mediatracker(this);
mt.addImage(img, 0);
try {
   mt.waitForAll();
} catch (InterruptedException e){}
```

Jetzt können wir in der *paint*-Methode mit dem Aufruf

```
g.drawImage(img, 0, 0, this);
```

das Bild aus dem **Mediatracker** laden und anzeigen.

Auf Fensterereignisse reagieren und sie behandeln I

Als Standardfensterklasse werden wir in den folgenden Abschnitten immer von dieser erben:

```
import java.awt.*;

public class MeinFenster extends Frame {
  public MeinFenster(String titel, int w, int h){
    this.setTitle(titel);
    this.setSize(w, h);

    // zentriere das Fenster
    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
    this.setLocation((d.width-this.getSize().width)/2,
                     (d.height-this.getSize().height)/2);
  }
}
```

Die Klasse **MeinFenster** erzeugt ein auf dem Bildschirm zentriertes Fenster und kann mit einem Konstruktor und den Attributen titel, breite und hoehe erzeugt werden.

Auf Fensterereignisse reagieren und sie behandeln II

Unser folgendes Beispiel erbt zunächst von der Klasse **MeinFenster** und implementiert anschließend das Interface **WindowListener**. In Zeile 4 verknüpfen wir unsere Anwendung mit dem Interface **WindowListener** und erreichen damit, dass bei Ereignissen, wie z.B. „schließe Fenster“, die entsprechenden implementierten Methoden aufgerufen werden.

```
import java.awt.*;
import java.awt.event.*;
public class FensterSchliesst extends MeinFenster implements WindowListener {
  public FensterSchliesst(String titel, int w, int h){
         super(titel, w, h);
         addWindowListener(this);  // wir registrieren hier den Ereignistyp für WindowEvents
         setVisible(true);
  }

  public void windowClosing( WindowEvent event ) {
    System.exit(0);
  }
  public void windowClosed( WindowEvent event )      {}
  public void windowDeiconified( WindowEvent event ) {}
  public void windowIconified( WindowEvent event )   {}
  public void windowActivated( WindowEvent event )   {}
  public void windowDeactivated( WindowEvent event ) {}
  public void windowOpened( WindowEvent event )      {}
  // ************************************************************

  public static void main( String[] args ) {
    FensterSchliesst f = new FensterSchliesst("Schliesse mich!", 200, 100);
  }
}
```

Auf Fensterereignisse reagieren und sie behandeln III

Leider haben wir mit der Implementierung des Interfaces **WindowListener** den Nachteil, dass wir alle Methoden implementieren müssen. Das Programm wird schnell unübersichtlich, wenn wir verschiedene Eventtypen abfangen wollen und für jedes Interface alle Methoden implementieren müssen.

Hilfe verspricht die Klasse **WindowAdapter**, die das Interface **WindowListener** bereits mit leeren Funktionskörpern implementiert hat. Wir können einfach von dieser Klasse erben und eine der Methoden überschreiben. Um die restlichen brauchen wir uns nicht zu kümmern.

```
import java.awt.*;
import java.awt.event.*;
public class FensterSchliesstSchick extends MeinFenster{
  public FensterSchliesstSchick(String titel, int w, int h){
      super(titel, w, h);
      // Wir verwenden eine Klasse, die nur die gewünschten Methoden
      // der Klasse WindowAdapter überschreibt.
      addWindowListener(new WindowClosingAdapter());
      setVisible(true);
  }

  public static void main( String[] args ) {
    FensterSchliesstSchick f = new FensterSchliesstSchick("Schliesse mich!", 200, 100);
  }
}

class WindowClosingAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
         System.exit(0);
  }
}
```

Auf Fensterereignisse reagieren und sie behandeln IV

Wir können die Klasse **WindowClosingAdapter** auch als innere Klasse deklarieren.

```
import java.awt.*;
import java.awt.event.*;

public class FensterSchliesstSchick2 extends MeinFenster{
  public FensterSchliesstSchick2(String titel, int w, int h){
         super(titel, w, h);
         addWindowListener(new WindowClosingAdapter());
         setVisible(true);
  }

  //*****************************************************************
  // innere Klasse
  private class WindowClosingAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
           System.exit(0);
    }
  }
  //*****************************************************************

  public static void main( String[] args ) {
    FensterSchliesstSchick2 f = new FensterSchliesstSchick2("Schliesse mich!", 200, 100);
  }
}
```

Auf Fensterereignisse reagieren und sie behandeln V

Eine noch kürzere Schreibweise könnten wir erreichen, indem wir die Klasse **WindowAdapter** nur lokal erzeugen und die Funktion überschreiben [Ullenboom 2006, siehe Java-Intensivkurs].

Wir nennen solche Klassen innere, anonyme Klassen.

```
import java.awt.event.*;
public class FensterSchliesstSchickKurz extends MeinFenster{
  public FensterSchliesstSchickKurz(String titel, int w, int h){
    super(titel, w, h);
    // Wir verwenden eine innere anonyme Klasse. Kurz und knapp.
    addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent e) {
           System.exit(0);
       }
    });
    setVisible(true);
  }

  public static void main( String[] args ) {
    FensterSchliesstSchickKurz f = new FensterSchliesstSchickKurz("Schliesse mich!", 200,
   100);
  }
}
```

Layoutmanager

Java bietet viele vordefinierte Layoutmanager an. Der einfachste Layoutmanager **FlowLayout** fügt die Elemente, je nach Größe, ähnlich einem Textfluss, links oben beginnend, in das Fenster ein.

Diese lassen sich z.B. innerhalb eines Frames mit der folgenden Zeile aktivieren:

```
setLayout(new FlowLayout());
```

Die Komponenten Label und Button I

Zwei Button und ein Label werden in die GUI eingebettet. Die Anordnung der Elemente innerhalb des Fensters kann von einem Layoutmanager übernommen werden. Für dieses Beispiel haben wir den Layoutmanager **FlowLayout** verwendet.

```
import java.awt.*;
import java.awt.event.*;
public class GUI_Button extends MeinFenster{
   private Button button1, button2;
   private Label label1;

   // Im Konstruktor erzeugen wir die GUI-Elemente
   public GUI_Button(String titel, int w, int h){
      super(titel, w, h);
      setSize(w, h);
      // Wir registrieren den WindowListener, um auf
      // WindowEvents reagieren zu können
      addWindowListener(new MeinWindowListener());
      // wir bauen einen ActionListener, der nur auf Knopfdruck
      // reagiert
      ActionListener aktion = new Knopfdruck();
      ...
```

Die Komponenten Label und Button II

weiter geht's:

```
      setLayout(new FlowLayout());

      button1 = new Button("Linker Knopf");
      add(button1);
      button1.addActionListener(aktion);
      button1.setActionCommand("b1");
      button2 = new Button("Rechter Knopf");
      add(button2);
      button2.addActionListener (aktion);
      button2.setActionCommand("b2");
      label1 = new Label("Ein Label");
      add(label1);
      setVisible(true);
   }

   // Innere Klassen für das Eventmanagement
   class MeinWindowListener extends WindowAdapter{
      public void windowClosing(WindowEvent event){
         System.exit(0);
      }
   }

   class Knopfdruck implements ActionListener{
      public void actionPerformed (ActionEvent e){
         label1.setText(e.getActionCommand());
      }
   }

   public static void main( String[] args ) {
      GUI_Button f = new GUI_Button("Schliesse mich!", 500, 500);
   }
}
```

Die Komponenten Label und Button III

Als Ausgabe erhalten wir:

Oder wir reagieren individuell:

```
class Knopfdruck implements ActionListener{
   public void actionPerformed (ActionEvent e){
      // wir behandeln die Ereignisse
      String cmd = e.getActionCommand();
      if (cmd.equals("b1"))
         Button1Clicked();
      if (cmd.equals("b2"))
         Button2Clicked();
   }
}
```

Die Komponenten TextField I

Wie wir Eingaben über ein TextField erhalten, zeigt das folgende Beispiel:

```
import java.awt.*;
import java.awt.event.*;

public class GUI_Button_TextField extends MeinFenster{
  private Button button1;
  private Label label1;
  private TextField textfield1;

  // Im Konstruktor erzeugen wir die GUI-Elemente
  public GUI_Button_TextField(String titel, int w, int h){
    super(titel, w, h);
    setSize(w, h);

    // Wir registrieren den WindowListener, um auf
    // WindowEvents reagieren zu können
    addWindowListener(new MeinWindowListener());

    // wir bauen einen ActionListener, der nur auf Knopfdruck
    // reagiert
    ActionListener aktion = new Knopfdruck();

    setLayout(new FlowLayout());

    textfield1 = new TextField("hier steht schon was", 25);
    add(textfield1);

    button1 = new Button("Knopf");
    add(button1);
    button1.addActionListener(aktion);
    button1.setActionCommand("b1");
    ...
```

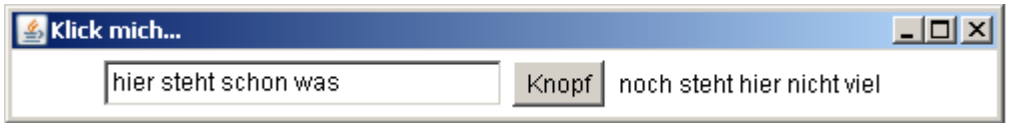
Die Komponenten TextField II

weiter geht's:

```
    ...
    label1 = new Label("noch steht hier nicht viel");
    add(label1);
    setVisible(true);
  }

  private void Button1Clicked(){
          String txt = textfield1.getText();
          label1.setText(txt);
  }

  // Innere Klassen für das Eventmanagement
  class MeinWindowListener extends WindowAdapter{
    public void windowClosing(WindowEvent event){
      System.exit(0);
    }
  }

  class Knopfdruck implements ActionListener{
    public void actionPerformed (ActionEvent e){
        // wir behandeln die Ereignisse
        String cmd = e.getActionCommand();
        if (cmd.equals("b1"))
           Button1Clicked();
    }
  }

  public static void main( String[] args ) {
    GUI_Button_TextField f = new GUI_Button_TextField("Klick mich...", 500, 500);
  }
}
```

Die Komponenten TextField III

Sollte der Button gedrückt werden, so wird der Textinhalt von *textfield1* in *label1* geschrieben.



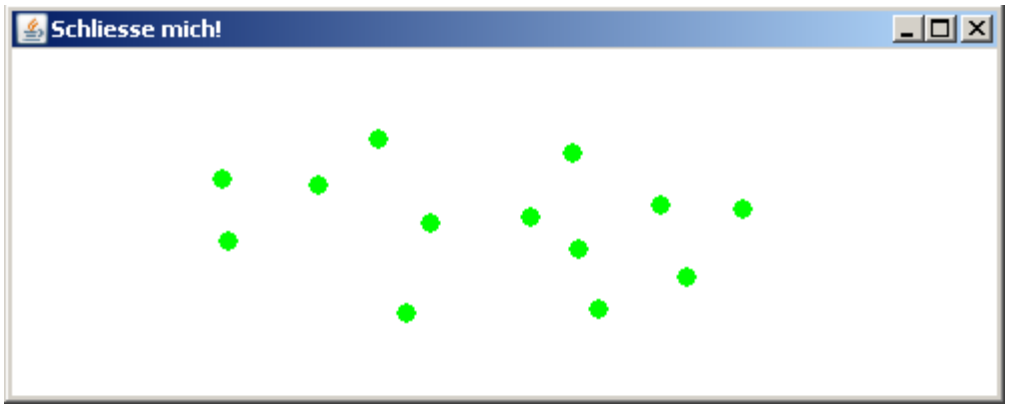
Auf Mausereignisse reagieren I

Wir haben mit **WindowListener** und **ActionListener** bereit zwei Interaktionsmöglichkeiten kennen gelernt. Es fehlt noch eine wichtige, die Reaktion auf Mausereignisse. Im folgenden Beispiel wollen wir für einen Mausklick innerhalb des Fensters einen grünen Punkt an die entsprechende Stelle zeichnen. Hier wird einfach das Interface **MouseListener** implementiert oder die leeren Methoden der Klasse **MouseAdapter** überschrieben.

```
import java.awt.*;
import java.awt.event.*;
public class MausKlick extends MeinFenster {
   public MausKlick(String titel, int w, int h){
      super(titel, w, h);
      setSize(w, h);
      // Wir verwenden eine innere anonyme Klasse. Kurz und knapp.
      addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent e) {
             System.exit(0);
          }
      });
      addMouseListener(new MouseAdapter() {
          public void mousePressed(MouseEvent e) {
             Graphics g = getGraphics();
             g.setColor(Color.green);
             g.fillOval(e.getX(),e.getY(),10,10);
          }
      });
      setVisible(true);
   }

   public static void main( String[] args ) {
      MausKlick f = new MausKlick("Schliesse mich!", 500, 200);
   }
}
```

Auf Mausereignisse reagieren II

Liefert nach mehrfachem Klicken mit der Maus, innerhalb des Fensters, folgende Ausgabe:



Appletprogrammierung

Inhalt:

- Einführung in HTML
- Konstruktion eines Applets
- AppletViewer
- Applikation zu Applet umbauen
- Flackernde Applets vermeiden

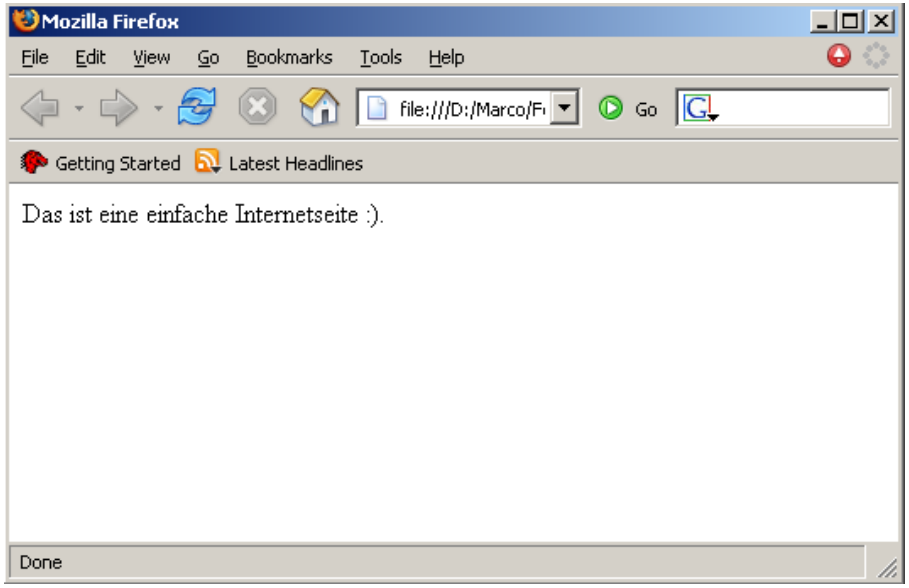
Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007
Gries D., Gries P.: "*Multimedia Introduction to Programming Using Java*“, Springer-Verlag 2005
Abts D.: „*Grundkurs JAVA: Von den Grundlagen bis zu Datenbank- und Netzanwendungen*“, Vieweg-Verlag 2007

Kurzeinführung in HTML

HTML (= HyperText Markup Language) ist eine Sprache, mit der sich Internetseiten erstellen lassen. Jeder Internetbrowser kann diese Sprache lesen und die gewünschten Inhalte darstellen. Mit folgenden Zeilen lässt sich eine sehr einfache HTML-Seite erzeugen.

Dazu gibt man die Codezeilen in einem beliebigen Editor ein und speichert sie mit der Endung „.html“.

```
<HTML>
   <BODY>
      Das ist eine einfache Internetseite :).
   </BODY>
</HTML>
```



Bauen eines kleinen Applets

Wir benötigen für ein Applet keinen Konstruktor. Wir verwenden lediglich die Basisklasse **Applet** und können einige Methoden überschreiben.

init()	Bei der Initialisierung übernimmt die *init*-Methode die gleiche Funktion wie es ein Konstruktor tun würde. Die Methode wird beim Start des Applets als erstes genau einmal aufgerufen. Es können Variablen initialisiert, Parameter von der HTML-Seite übernommen oder Bilder geladen werden.
start()	Die *start*-Methode wird aufgerufen, nachdem die Initialisierung abgeschlossen wurde. Sie kann sogar mehrmals aufgerufen werden.
stop()	Mit *stop* kann das Applet eingefroren werden, bis es mit *start* wieder erweckt wird. Die *stop*-Methode wird z.B. aufgerufen, wenn das HTML-Fenster, indem das Applet gestartet wurde, verlassen wird.
destroy()	Bei Beendigung des Applets, z.B. Schließen des Fensters, wird *destroy* aufgerufen und alle Speicherressourcen wieder freigegeben.

Verwendung des Appletviewers I

Nach der Installation von Java steht uns das Programm **AppletViewer** zur Verfügung. Wir können die HTML-Seite, in der ein oder mehrere Applets eingebettet sind, aufrufen. Es wird separat für jedes Applet ein Fenster geöffnet und das Applet darin gestartet. Nun lassen sich alle Funktionen quasi per Knopfdruck ausführen und testen.

So starten wir den Appletviewer:

```
appletviewer <html-seite.html>
```

Verwendung des Appletviewers II

Verwenden wir für das erste Applet folgenden Programmcode und testen das Applet mit dem Appletviewer:

```
import java.awt.*;
import java.applet.*;

public class AppletZyklus extends Applet {
   private int zaehler;
   private String txt;

   public void init(){
      zaehler=0;
      System.out.println("init");
   }

   public void start(){
      txt = "start: "+zaehler;
      System.out.println("start");
   }

   public void stop(){
      zaehler++;
      System.out.println("stop");
   }

   public void destroy(){
      System.out.println("destroy");
   }

   public void paint(Graphics g){
      g.drawString(txt, 20, 20);
   }
}
```

Verwendung des Appletviewers II

Folgende HTML-Seite erzeugt das Applet. Die beiden Parameter *width* und *height* legen die Größe des Darstellungsbereichs fest.

```
<HTML>
   <BODY>
       <APPLET width=200 height=50 code="AppletZyklus.class"></APPLET>
   </BODY>
</HTML>
```

Wir erhalten nach mehrmaligem Stoppen und wieder Starten folgende Ausgabe auf der Konsole:

```
C:\Applets>appletviewer AppletZyklus.html
init
start
stop
start
stop
start
stop
start
stop
destroy
```

Eine Applikation zum Applet umbauen I

In den meisten Fällen ist es sehr einfach, eine Applikation zu einem Applet umzufunktionieren. Wir erinnern uns an die Klasse **Farbbeispiel** :

```
import java.awt.*;
import java.util.Random;

public class FarbBeispiel extends Frame {
   public FarbBeispiel(String titel) {
      setTitle(titel);
      setSize(500, 300);
      setBackground(Color.lightGray);
      setForeground(Color.red);
      setVisible(true);
   }

   public void paint(Graphics g){
      Random r = new Random();
      for (int y=30; y<getHeight()-10; y += 15)
         for (int x=12; x<getWidth()-10; x += 15) {
            g.setColor(new Color(r.nextInt(256), r.nextInt(256), r.nextInt(256)));
            g.fillRect(x, y, 10, 10);
            g.setColor(Color.BLACK);
            g.drawRect(x - 1, y - 1, 10, 10);
         }
   }

   public static void main(String[] args) {
      FarbBeispiel t = new FarbBeispiel("Viel Farbe im Fenster");
   }
}
```

Eine Applikation zum Applet umbauen II

Schritt 1: **Konstruktor zu init**

Da ein Applet keinen Konstruktor benötigt, wird die Klasse innerhalb einer HTML-Seite erzeugt. Wir können an dieser Stelle die Parameter für *width*, *height* und *titel* übergeben und in *init* setzen.

```
<HTML>
   <BODY>
       <APPLET CODE = "FarbBeispielApplet.class" WIDTH=500 HEIGHT=300>
          <PARAM NAME="width"  VALUE=500>
          <PARAM NAME="height" VALUE=300>
          <PARAM NAME="titel"  VALUE="Farbe im Applet">
       </APPLET>
   </BODY>
</HTML>
```

```
import java.awt.*;
import java.applet.*;
import java.util.Random;

public class FarbBeispielApplet extends Applet {
   private int width;
   private int height;
   private String titel;

   public void init() {
      // Parameterübernahme
      width  = Integer.parseInt(getParameter("width"));
      height = Integer.parseInt(getParameter("height"));
      titel  = getParameter("titel");
      setSize(width, height);
      setBackground(Color.lightGray);
      setForeground(Color.red);
   }
   ...
```

Eine Applikation zum Applet umbauen III

Schritt 2: **paint-Methode anpassen**

Für unser Beispiel können wir den Inhalt der *paint*-Methode komplett übernehmen.

```
   ...
   public void paint(Graphics g){
      Random r = new Random();
      for (int y=30; y<getHeight()-10; y += 15)
         for (int x=12; x<getWidth()-10; x += 15) {
            g.setColor(new Color(r.nextInt(256),
                                 r.nextInt(256),
                                 r.nextInt(256)));
            g.fillRect(x, y, 10, 10);
            g.setColor(Color.BLACK);
            g.drawRect(x - 1, y - 1, 10, 10);
         }
   }
}
```

Das Applet ist fertig, wir können es mit dem Appletviewer starten und erhalten die gleiche Ausgabe.

TextField-Beispiel zum Applet umbauen I

Im ersten Applet gab es keine Ereignissbehandlung. Aber auch dafür wollen wir uns ein einfaches Beispiel anschauen. Das TextField-Beispiel aus Abschnitt . Nach der Konvertierung zu einem Applet sieht es wie folgt aus:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class GUI_Button_TextField_Applet extends Applet{
  private int width;
  private int height;
  private String titel;

  Button button1;
  Label label1;
  TextField textfield1;

  public void init() {
      // Parameterübergabe
      width  = Integer.parseInt(getParameter("width"));
      height = Integer.parseInt(getParameter("height"));
      titel  = getParameter("titel");
      setSize(width, height);

      // wir bauen einen ActionListener, der nur auf Knopfdruck
      // reagiert
      ActionListener aktion = new Knopfdruck();

      setLayout(new FlowLayout());

      textfield1 = new TextField("hier steht schon was", 25);
      add(textfield1);
      ...
```

TextField-Beispiel zum Applet umbauen II

weiter geht's:

```
      ...
      button1 = new Button("Knopf");
      add(button1);
      button1.addActionListener(aktion);
      button1.setActionCommand("b1");

      label1 = new Label("noch steht hier nicht viel");
      add(label1);
  }

  private void Button1Clicked(){
      String txt = textfield1.getText();
      label1.setText(txt);
  }

  // ************************************************************
  // Innere Klassen für das Eventmanagement
  class Knopfdruck implements ActionListener{
    public void actionPerformed (ActionEvent e){
        // wir behandeln die Ereignisse
        String cmd = e.getActionCommand();
        if (cmd.equals("b1"))
           Button1Clicked();
    }
  }
  // ************************************************************
}
```

Die Ereignissbehandlung verhält sich analog zu den bisher bekannten Applikationen.

Flackernde Applets vermeiden I

Bei den ersten eigenen Applets wird eine unangenehme Eigenschaft auftauchen. Sie flackern. Schauen wir uns dazu einmal folgendes Programm an:

```
<HTML>
   <BODY>
      <APPLET width=472 height=482 code="BildUndStrukturFlackern.class">
      </APPLET>
   </BODY>
</HTML>
```

Flackerndes Applet:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.URL;
import java.lang.Math;
import java.util.Random;

/* Bild als Hintergrund und mausempfindliche Linienstruktur im
   Vordergrund erzeugt unangenehmes flackern */
public class BildUndStrukturFlackern extends Applet {
   int width, height, mx, my, counter=0;
   final int N=100;
   Point[] points;
   Image img;
   Random r;

   ...
```

Flackernde Applets vermeiden II

Weiter geht's:

```
...
public void init() {
   width  = getSize().width;
   height = getSize().height;
   setBackground(Color.black);

   // Mauskoordinaten und MouseMotionListener
   addMouseMotionListener(new MouseMotionHelper());

   img = getImage(getDocumentBase(), "apfelmaennchen.jpg");
   r = new Random();
}

// ********************************************************
// Klassenmethoden
private void zeichneLinien(Graphics g){
   for (int i=1; i<N; i++) {
      // wähle zufällige Farbe
      g.setColor(new Color(r.nextInt(256),
                           r.nextInt(256),
                           r.nextInt(256)));

       // verbinde die Punkte
       g.drawLine(mx+(int)((r.nextFloat()-0.2)*(width /2)),
                  my+(int)((r.nextFloat()-0.2)*(height/2)),
                  mx+(int)((r.nextFloat()-0.2)*(width /2)),
                  my+(int)((r.nextFloat()-0.2)*(height/2)));
   }
}
...
```

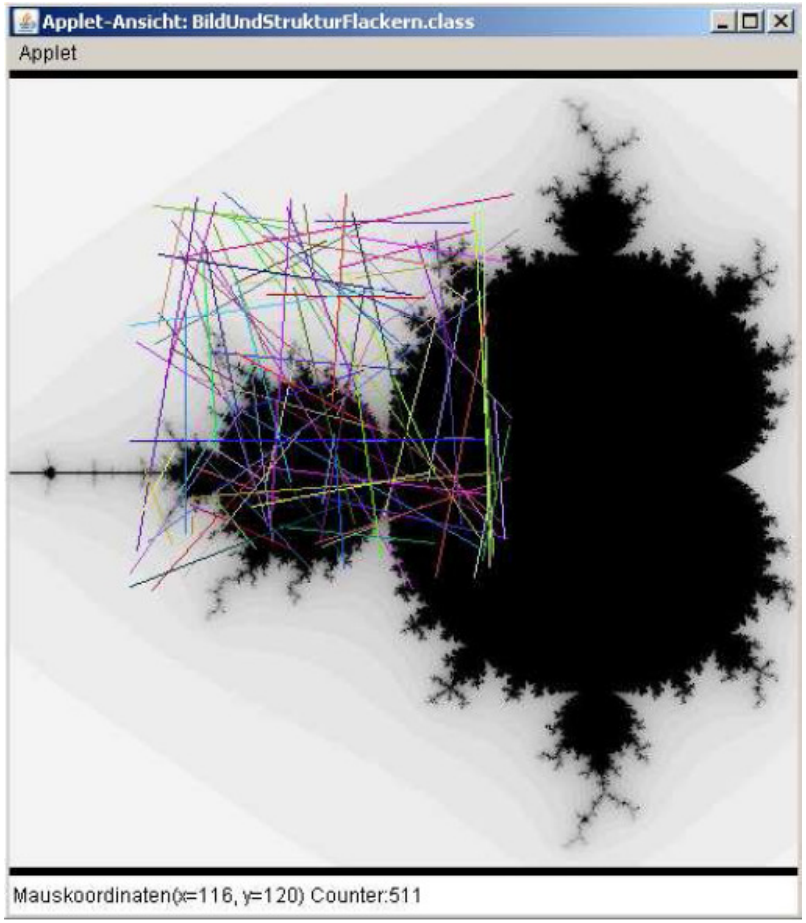
Flackernde Applets vermeiden III

Weiter geht's:

```
   ...
   private void zeichneBild(Graphics g){
      g.drawImage(img, 0, 5, this);
   }
   // ************************************************************

   // ************************************************************
   private class MouseMotionHelper extends MouseMotionAdapter{
      // Maus wird innerhalb der Appletflaeche bewegt
      public void mouseMoved(MouseEvent e) {
         mx = e.getX();
         my = e.getY();
         showStatus("Mauskoordinaten(x="+mx+", y="+my+")
                    Counter:"+counter);
         // ruft die paint-Methode auf
         repaint();
         e.consume();
      }
   }
   // ************************************************************

   public void paint(Graphics g) {
      zeichneBild(g);
      zeichneLinien(g);
      counter++;
   }
}
```

Flackernde Applets vermeiden IV

Bei der Bewegung der Maus über das Apfelmännchen werden zufällige Linien mit zufälligen Farben erzeugt. Die Darstellung ist relativ rechenintensiv und das Applet beginnt zu Flackern. Nebenbei erfahren wir in diesem Beispiel, wie wir dem Browser mit der Methode *showStatus* eine Ausgabe geben können.



Ghosttechnik I

Unsere erste Idee an dieser Stelle könnte sein, die Darstellung in der Art zu beschleunigen, dass wir zunächst auf einem unsichtbaren Bild arbeiten und dieses anschließend neu zeichnen.

Diese Technik könnten wir als Ghosttechnik bezeichnen, da wir auf einem unsichtbaren Bild arbeiten und es anschließend wie von Geisterhand anzeigen lassen.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.URL;
import java.lang.Math;
import java.util.Random;

/* Bild als Hintergrund und mausempfindliche Linienstruktur im
   Vordergrund erzeugt unangenehmes Flackern, trotz Ghostimage!
*/
public class BildUndStrukturFlackernGhost extends Applet {
   int width, height, mx, my, counter=0;
   final int N=100;
   Point[] points;
   Image img;

   // ++++++++++++++++++
   Image img_ghost;
   Graphics g_ghost;
   // ++++++++++++++++++

   Random r;
   ...
```

Ghosttechnik II

weiter geht's:

```
...
public void init() {
   width  = getSize().width;
   height = getSize().height;
   setBackground(Color.black);
   // Mauskoordinaten und MouseMotionListener
   addMouseMotionListener(new MouseMotionHelper());
   img = getImage(getDocumentBase(), "fractalkohl.jpg");

   // ++++++++++++++++++
   img_ghost = createImage(width, height);
   g_ghost   = img_ghost.getGraphics();
   // ++++++++++++++++++

   r = new Random();
}

// ************************************************************
// Klassenmethoden
private void zeichneLinien(Graphics g){
   for (int i=1; i<N; i++) {
     // wähle zufällige Farbe
     g.setColor(new Color(r.nextInt(256), r.nextInt(256), r.nextInt(256)));

     // verbinde N zufällig erzeugte Punkte
     g.drawLine(mx+(int)((r.nextFloat()-0.2)*(width /2)),
                my+(int)((r.nextFloat()-0.2)*(height/2)),
                mx+(int)((r.nextFloat()-0.2)*(width /2)),
                my+(int)((r.nextFloat()-0.2)*(height/2)));
   }
}
...
```

Ghosttechnik III

weiter geht's:

```
   ...
   private void zeichneBild(Graphics g){
      g.drawImage(img, 0, 0, this);
   }

   private class MouseMotionHelper extends MouseMotionAdapter{
      // Maus wird innerhalb der Appletflaeche bewegt
      public void mouseMoved(MouseEvent e) {
         mx = e.getX();
         my = e.getY();
         showStatus("Mauskoordinaten(x="+mx+", y="+my+")
                     Counter:"+counter);
         // ruft die paint-Methode auf
         repaint();
         e.consume();
      }
   }

   public void paint(Graphics g) {
      zeichneBild(g_ghost);
      zeichneLinien(g_ghost);
      // ++++++++++++++++++
      g.drawImage(img_ghost, 0, 0, this);
      // ++++++++++++++++++
      counter++;
   }
}
```

Wenn wir dieses Applet starten, müssen wir ebenfalls ein Flackern feststellen, es ist sogar ein wenig schlimmer geworden. Im Prinzip stellt diese Technik eine Verbesserung dar. In Kombination mit dem folgenden Tipp, lassen sich rechenintensive Grafikausgaben realisieren.

Überschreiben der paint-Methode

Der Grund für das permanente Flackern ist der Aufruf der *update*-Funktion. Die *update*-Funktion sorgt dafür, dass zunächst der Hintergrund gelöscht und anschließend die *paint*-Methode aufgerufen wird.

Da wir aber von der Appletklasse erben, können wir diese Methode einfach überschreiben!

```
...
public void update( Graphics g ) {
   zeichneBild(g_ghost);
   zeichneLinien(g_ghost);

   g.drawImage(img_ghost, 0, 0, this);
}

public void paint( Graphics g ) {
   update(g);
   counter++;
}
...
```

Das gestartete Applet zeigt nun kein Flackern mehr.

Beispiel mit mouseDragged I

Abschließend wollen wir uns noch ein Beispiel für die Verwendung der *mouseDragged()*-Methode anschauen. Mit der Maus kann ein kleines Objekt mit der gedrückten linken Maustaste verschoben werden. In diesem Fall wird der kleine Elch im schwedischen Sonnenuntergang platziert.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class MausBild extends Applet {
   int width, height, x, y, mx, my;
   Image img, img2, img_ghost;
   Graphics g_ghost;
   boolean isInBox = false;

   public void init() {
      width  = getSize().width;
      height = getSize().height;
      setBackground(Color.black);

      addMouseListener(new MouseHelper());
      addMouseMotionListener(new MouseMotionHelper());

      img  = getImage(getDocumentBase(), "schweden.jpg");
      img2 = getImage(getDocumentBase(), "elch.jpg");

      img_ghost = createImage(width, height);
      g_ghost   = img_ghost.getGraphics();

      // mittige Position für den Elch zu Beginn
      x = width/2  - 62;
      y = height/2 - 55;
   }
   ...
```

Beispiel mit mouseDragged II

weiter geht's:

```
...
private class MouseHelper extends MouseAdapter{
  public void mousePressed(MouseEvent e) {
     mx = e.getX();
     my = e.getY();
     // ist die Maus innerhalb des Elch-Bildes?
     if (x<mx && mx<x+124 && y<my && my<y+111)
        isInBox = true;
     e.consume();
  }

  public void mouseReleased(MouseEvent e) {
     isInBox = false;
     e.consume();
  }
}

private class MouseMotionHelper extends MouseMotionAdapter{
   public void mouseDragged(MouseEvent e) {
      if (isInBox) {
         int new_mx = e.getX();
         int new_my = e.getY();
         // Offset ermitteln
         x += new_mx - mx;            y += new_my - my;
         mx = new_mx;                 my = new_my;
         repaint();
         e.consume();
      }
   }
}
...
```

Beispiel mit mouseDragged III

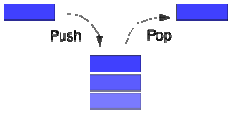
und fertig:

```
   ...
   public void update( Graphics g ) {
      g_ghost.drawImage(img, 0, 0,  this);
      g_ghost.drawImage(img2, x, y, this);
      g.drawImage(img_ghost, 0, 0,  this);
   }

   public void paint( Graphics g ) {
      update(g);
   }
}
```

Elementare Datenstrukturen

Inhalt:

- Stack
- Queue



Folieninhalte teilweise übernommen von PD Dr. Klaus Kriegel (Informatik B, SoSe 2000)

Stack I

Ein Stack (Stapel oder Keller) verwaltet eine geordnete Menge von Objekten, wobei ein Element angefügt oder das letzte hinzugefügte wieder weggenommen werden kann.

Definition des Binomialkoeffizienten:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}.$$

Rekursive Definitionsvorschrift:

$$\binom{n}{0} = 1 \qquad \binom{n}{1} = n \qquad \binom{n}{n} = 1$$

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$$

Effiziente Vorschrift:

$$\binom{n}{k} = \prod_{i=1}^{k} \frac{n+1-i}{i}$$

$$\binom{n}{n-k} = \binom{n}{k}$$

Übungsaufgabe

Rekursive Funktion in Java:

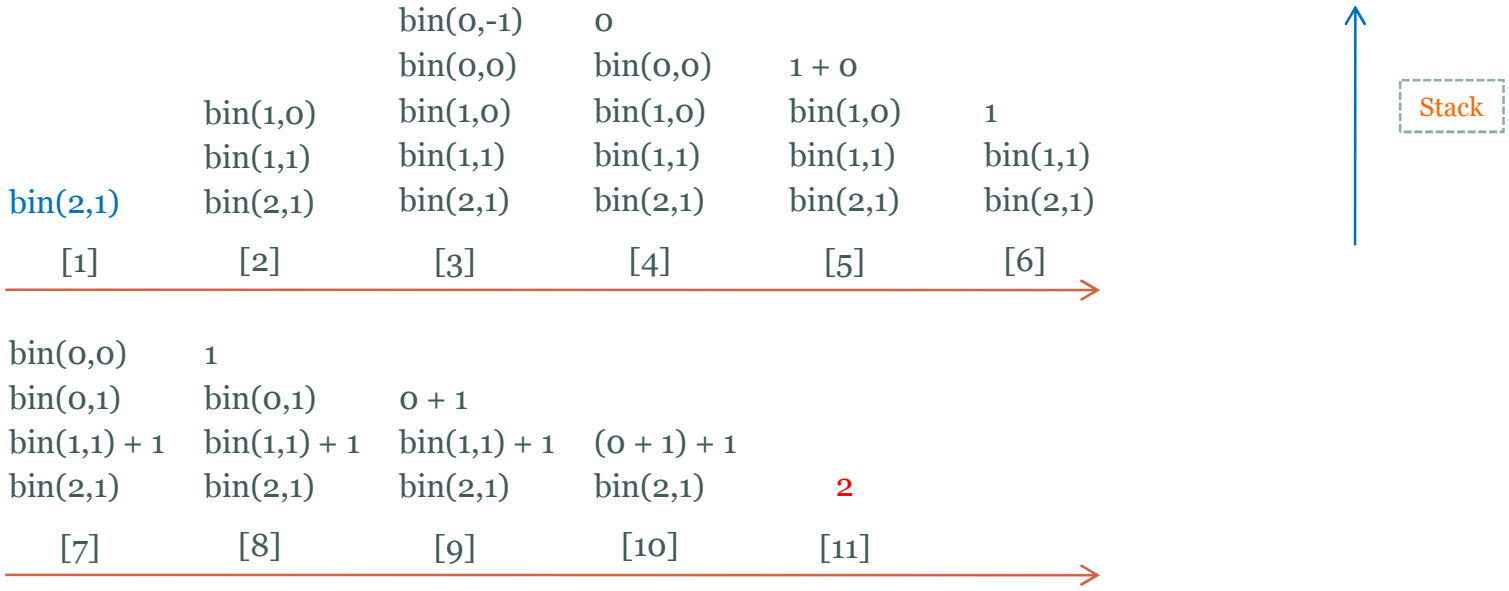
```
public static int bino(int n, int k) {
   if (n==0)
      if (k==0) return 1;
      else      return 0;

   return bino(n-1, k) + bino(n-1, k-1);
}
```

Stack II

Rekursive Funktion in Java:

```
public static int bino(int n, int k) {
   if (n==0)
      if (k==0) return 1;
      else      return 0;

   return bino(n-1, k) + bino(n-1, k-1);
}
```

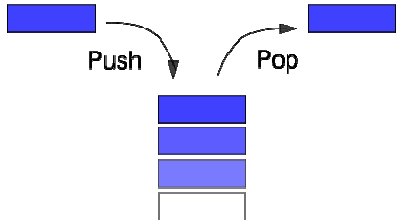
Abarbeitung der Funktion:



Stack III

Methoden eines Stacks:

push(Objekt)	fügt Objekt als oberstes Element dem Stack hinzu
Objekt pop()	gibt das oberste Element des Stacks zurück und entfernt es
boolean isEmpty()	prüft, ob der Stack leer ist
int size()	liefert die Anzahl der im Stack vorhandenen Elemente zurück



(Abbildung von Wikipedia)

Beispielanwendung:

Auswertung eines Rechenausdrucke in UPN (umgekehrt polnische Notation) oder Postfixnotation.

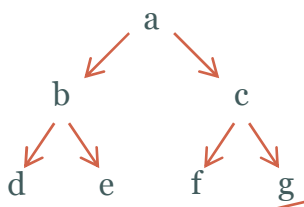
normale Schreibweise: (7*4+2) / (3*4+3)

UPN: 7 4 * 2 + 3 4 * 3 + /

```
public int evaluateUPN(String[] e) {
   for (int i=0; i<=n; i++){
      if (isNumber(e[i]))
         push(e[i]);
      else
         push(pop() e[i] pop())  // e[i] ist dabei der ermittelte Operator
   }
   return pop();
}
```

Prefix, Infix, Postfix

Verschiedene Notationsformen in binären Bäumen:

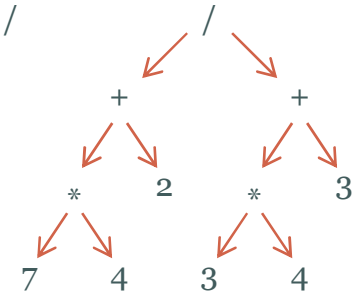


also erst den Knoten, dann links und anschließend rechts

Die Rekursionsformel für prefix lautet KLR. [a, b, d, e, c, f, g]
Die Rekursionsformel für infix lautet LKR. [d, b, e, a, f, c, g]
Die Rekursionsformel für postfix lautet LRK. [d, e, b, f, g, c, a]

Syntaxbaum für die Beispielformel

normale Schreibweise: (7*4+2) / (3*4+3)
UPN (postfix): 7 4 * 2 + 3 4 * 3 + /



Realisierung durch ein Array I

Wir speichern die Objekte in einem Array und merken uns, wie viele Elemente zur Zeit in dem Array sind. Der Arrayanfang ist das untere Ende und das Arrayende das aktuell letzte Elementindex des Stacks.

```
public class Stack {
   private int currentIndex;
   private int[] array;

   public Stack(){
      currentIndex      = 0;
      array             = new int[100];
   }

   public void push(int a){
      if (currentIndex<99) {
         currentIndex++;
         array[currentIndex] = a;
      }
   }

   public int pop() {
      if (currentIndex<=0)
         return -1;
      currentIndex--;

      return array[currentIndex+1];
   }
   ...
```

Realisierung durch ein Array II

Weiter geht's:

```
   ...
   public Boolean isEmpty(){
      return currentIndex == 0;
   }

   public int size(){
      return currentIndex;
   }

   public static void main(String[] args){
      Stack s = new Stack();
      System.out.println("size: "+s.size());
      s.push(7);
      s.push(4);
      System.out.println("size: "+s.size());
      System.out.println("pop "+s.pop());
      System.out.println("pop "+s.pop());
      System.out.println("pop "+s.pop());
      System.out.println("size: "+s.size());
   }
}
```

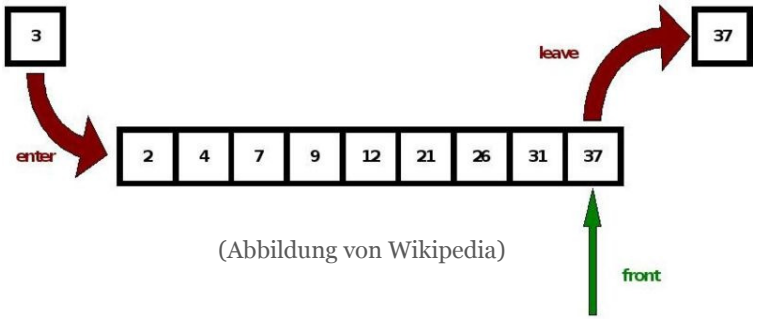
liefert:

```
size: 0
size: 2
pop 4
pop 7
pop -1
size: 0
```

Queue I

Eine Queue (Warteschlange) liefert im Gegensatz zum Stack das am längsten vorhandene Objekt zurück. Wie bei einer Warteschlange, muss gewartet werden, bis man an der Reihe ist.

Stack	LIFO (last in - first out)
Queue	FIFO (first in - first out)



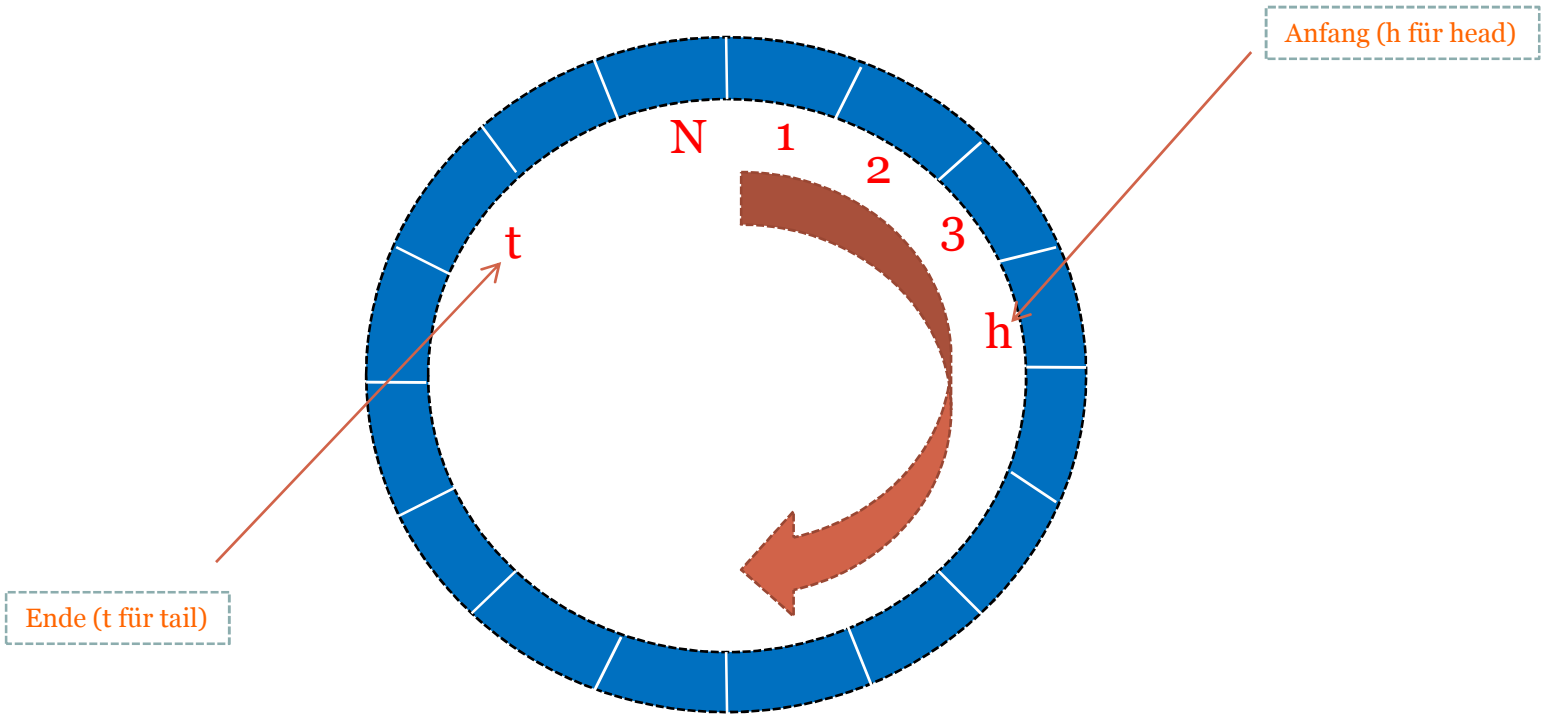
(Abbildung von Wikipedia)

Methoden einer Queue:

enqueue(Objekt)	fügt Objekt als hinterstes Element der Queue hinzu
Objekt dequeue()	gibt das vorderste Element der Queue zurück und entfernt es
boolean isEmpty()	prüft, ob die Queue leer ist
int size()	liefert die Anzahl der in der Queue vorhandenen Elemente zurück

Realisierung durch ein Array (zyklisch)

Wir stellen uns vor, dass wir die beiden Enden das Arrays zusammenkleben und somit einen Kreis erhalten.



Realisierung durch ein Array I

Wir speichern die Objekte in einem Array und merken uns, wie viele Elemente zur Zeit in dem Array sind. Im Unterschied zum Stack müssen wir sowohl den Anfangs- als auch das Endindex speichern. Damit die Daten nicht wie eine Raupe durch das Array wandern, werden wir es zyklisch verwalten.

```
public class Queue {
   private int h, t;
   private int n;                    // n = aktuelle Anzahl der Elemente
   private final int N = 100;        // maximale Anzahl zu speichernder Elemente

   private int[] queue;

   public Queue(){
      h     = 0;
      t     = 0;
      n     = 0;
      queue = new int[N];
   }

   public void enqueue(int a){
      if (n<99) {
         n++;
         if (t<N-1)
            t++;
         else
            t = 0;
         queue[t] = a;
      }
   }
   ...
```

Realisierung durch ein Array II

Weiter geht's:

```
...
public int dequeue() {
   if (n<=0)
      return -1;

      n--;
      if (h<N-1) {
         t++;
         return queue[t-1];
      } else {
         t = 0;
         return queue[N-1];
      }
}

public Boolean isEmpty(){
   return n == 0;
}

public int size(){
   return n;
}
...
```

Realisierung durch ein Array III

Weiter geht's:

```
   ...
   public static void main(String[] args){
      Queue q = new Queue();
      System.out.println("size: "+q.size());
      q.enqueue(7);
      q.enqueue(4);
      System.out.println("size: "+q.size());

      System.out.println("dequeue "+q.dequeue());
      System.out.println("dequeue "+q.dequeue());
      System.out.println("dequeue "+q.dequeue());

      System.out.println("size: "+q.size());
   }
}
```

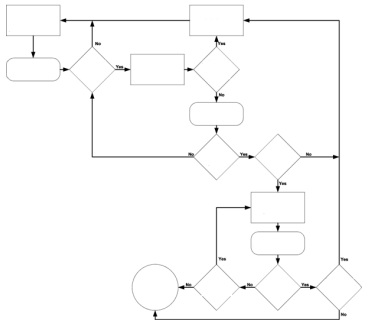
Liefert folgende Ausgabe:

```
size: 0
size: 2
dequeue 7
dequeue 4
dequeue -1
size: 0
```

Techniken der Programmentwicklung

Inhalt:

- Algorithmus
- Entwurfstechniken
- Einfache Algorithmen
- Primzahlen, InsertionSort, BubbleSort



Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007

Algorithmusbegriff

Ein Algorithmus (*algorithm*) (nach Al-Chwarizmis, arab. Mathematiker, 9. Jhdt.) ist ein Verfahren zur Lösung einer Klasse gleichartiger Probleme, bestehend aus Einzelschritten, mit folgenden Eigenschaften:

- Jeder Einzelschritt ist für die ausführende Instanz unmittelbar verständlich und ausführbar.
- Das Verfahren ist endlich beschreibbar.
- [Die Ausführung benötigt eine endliche Zeit („terminiert“).]

(kurz: „ein allgemeines, eindeutiges, endliches Verfahren“)

Algorithmusbegriff (konkreter)

sequentieller Algorithmus (*sequential algorithm*):
bei der Ausführung striktes Nacheinander der Einzelschritte

nichtsequentieller Algorithmus (*concurrent algorithm*):
Ausführungsreihenfolge der Einzelschritte bleibt teilweise offen

deterministischer Algorithmus (*deterministic algorithm*):
bei der Ausführung ist der jeweils nächste Schritt und sein Effekt eindeutig festgelegt (ist stets sequentiell)

nichtdeterministischer/stochastischer Algorithmus:
sonst

Statue Al-Chwarizmis,
TU Teheran
(Bild von Wikipedia)

Definition Algorithmus

Mit Algorithmen bezeichnen wir genau definierte Handlungsvorschriften zur Lösung eines Problems oder einer bestimmten Art von Problemen . Es geht darum, eine genau spezifizierte Abfolge von Anweisungen auszuführen, um ein gegebenes Problem zu lösen.

Entwurfstechniken

In diesem Abschnitt werden wir weniger auf die Effizienz einzelner Problemlösungen zu sprechen kommen, als vielmehr verschiedene Ansätze aufzeigen, mit denen Probleme gelöst werden können.

Es soll ein Einblick in die verschiedenen **Programmiertechniken** zum Entwurf von Programmen geben. Zunächst werden ein paar Begriffe und Techniken erläutert werden. Anschließend festigen wir diese mit kleinen algorithmischen Problemen und deren Lösungen.

Prinzip der Rekursion I

Rekursion bedeutet, dass eine Funktion sich selber wieder aufruft.

Als Rekursion (lat. recurrere „zurücklaufen“) bezeichnet man die Technik in Mathematik, Logik und Informatik, eine Funktion durch sich selbst zu definieren (rekursive Definition). Wenn man mehrere Funktionen durch wechselseitige Verwendung voneinander definiert, spricht man von wechselseitiger Rekursion. [Wikipedia]

Beispiel Fakultät

Die Fakultät für eine natürliche Zahl n ist definiert als: n! := n * (n-1) * (n-2) * ... * 2 * 1
nicht-rekursiv:

```
public static int fakultaet(int n){
   int erg = 1;
   for (int i=n; i>1; i--)
      erg *= i;
   return erg;
}
```

rekursiv:

```
public static int fakultaet(int i){
    if (i==1)
       return 1;
    else
       return i * fakultaet(i-1);
}
```

Prinzip der Rekursion II

In den meisten Fällen verkürzt sich die Notation durch eine rekursive Formulierung erheblich. Die Programme werden zwar kürzer und anschaulicher, aber der Speicher- und Zeitaufwand nimmt mit jedem Rekursionsschritt zu.

Bei der Abarbeitung einer rekursiven Funktion, werden in den meisten Fällen alle in der Funktion vorkommenden Parameter erneut im Speicher angelegt und mit diesen weitergearbeitet. Für zeitkritische Programme sollte bei der Implementierung aus Gründen der Effizienz auf Rekursion verzichtet werden .

Beispiel Fibonacci

Die Fibonacci-Folge ist ein häufig verwendetes Beispiel für rekursive Methoden. Sie beschreibt beispielsweise das Populationverhalten von Kaninchen. Zu Beginn gibt es ein Kaninchenpaar. Jedes neugeborene Paar wird nach zwei Monaten geschlechtsreif. Geschlechtsreife Paare werfen pro Monat ein weiteres Paar.

Um die ersten n Zahlen dieser Folge zu berechnen, schauen wir uns die folgende Definition an:

fib(0)=0 und fib(1)=1
fib(n)=fib(n-1)+fib(n-2) , für n>=2

Die Definition selbst ist schon **rekursiv**. Beginnend bei dem kleinsten Funktionswert lässt sich diese Folge, da der neue Funktionswert gerade die Summe der beiden Vorgänger ist, leicht aufschreiben:

0,1,1,2,3,5,8,13,21,34,55,89,... .

Brute Force

Brute Force bedeutet übersetzt soviel wie „*Methode der rohen Gewalt*“ . Damit ist gemeint, dass alle möglichen Kombination, die für eine Lösung in Frage kommen können, durchprobiert werden.

Moderne Schachprogramme basieren auf der Brute-Force-Technik. Alle möglichen Züge bis zu einer bestimmten Suchtiefe werden ausprobiert und die resultierenden Stellungen bewertet. Anschließend führen die berechneten Bewertungen dazu, aus den zukünftigen möglichen Stellungen den für die aktuelle Stellung besten Zug zu bestimmen.

Man könnte hier auf die Frage kommen:

Warum können Schachprogramme dann nicht einfach bis zum Partieende rechnen und immer gewinnen?

Das ist in der Tatsache begründet, dass das Schachspiel sehr komplex ist. Es gibt bei einer durchschnittlichen Zuganzahl von 50 Zügen, schätzungsweise 10^{120} unterschiedliche Schachstellungen. Im Vergleich dazu wird die Anzahl der Atome im Weltall auf 10^{80} geschätzt. Das ist der Grund, warum Schach aus theoretischer Sicht noch interessant ist.

Greedy

Greedy übersetzt bedeutet gierig, und genau so lässt sich diese Entwurfstechnik auch beschreiben. Für ein Problem, dass in Teilschritten gelöst werden kann, wählt man für jeden Teilschritt die Lösung aus, die den größtmöglichen Gewinn verspricht. Das hat aber zur Folge, dass der Algorithmus für bestimmte Problemstellungen nicht immer zwangsläufig die beste Lösung findet. Es gibt aber Klassen von Problemen, bei denen dieses Verfahren erfolgreich arbeitet.

Als praktisches Beispiel nehmen wir die Geldrückgabe an der Kasse. Kassierer verfahren meistens nach dem Greedy-Algorithmus. Gebe zunächst den Schein oder die Münze mit dem größtmöglichen Wert heraus, der kleiner oder gleich der Restsumme ist. Mit dem Restbetrag wird gleichermaßen verfahren.

Beispiel: **Rückgabe von 1 Euro und 68 Cent.**
(Die Lösung von links nach rechts)

Für dieses Beispiel liefert der Algorithmus immer die richtige Lösung. Hier sei angemerkt, dass es viele Problemlösungen der Graphentheorie gibt, die auf der Greedy-Strategie basieren.

Dynamische Programmierung, Memoisierung I

Bei der Dynamischen Programmierung wird die optimale Lösung aus optimalen Teillösungen zusammengesetzt. Teillösungen werden dabei in einer geeigneten Datenstruktur gespeichert, um kostspielige Rekursionen zu vermeiden. Rekursion kann kostspielig sein, wenn gleiche Teilprobleme mehrfach gelöst werden. Einmal berechnete Ergebnisse werden z.B. in Tabellen gespeichert und später gegebenenfalls darauf zugegriffen.

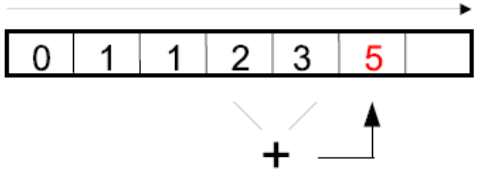
Memoisierung ist dem Konzept der Dynamischen Programmierung sehr ähnlich. Eine Datenstruktur wird beispielsweise in einen Rekursionsformel so eingearbeitet, dass auf bereits ermittelte Daten zurückgegriffen werden kann. Anhand der uns bereits gut bekannten Fibonacci-Zahlen wollen wir diese Verfahren untersuchen.

Fibonacci-Zahlen mit Dynamischer Programmierung

Die Erzeugung der Fibonacci-Zahlen lässt sich mit Dynamischer Programmierung wesentlich effizienter realisieren. Wir müssen uns eine geeignete Datenstruktur wählen. In diesem Fall ist eine Liste sehr hilfreich und wir könnten das Programm in etwa so formulieren:

```
n-elementige, leere Liste fibi erzeugen
fibi[0] = 0
fibi[1] = 1
for (i=2 to n)
   fibi[i] = fibi[i-1] + fibi[i-2]
```

Die Funktionswerte werden in einer Schleife ermittelt und können anschließend ausgegeben werden:



Fibonacci-Zahlen mit Memoisierung

Im folgenden Beispiel verwenden wir die Datenstruktur fibi im Rekursionsprozess, um bereits ermittelte Zwischenlösungen wiederzuverwenden:

```
m-elementige, leere Liste fibi erzeugen
fibi[0] = 0
fibi[1] = 1
```

Die Initialisierung muss einmal beim Programmstart ausgeführt werden und erzeugt eine m-elementige, leere Liste fibi. Die ersten zwei Elemente tragen wir ein.

Bei der Anfrage *fib(n)* mit einem n das kleiner als m ist, wird die Teilfolge der Fibonaccizahlen bis n in die Datenstruktur eingetragen.

```
fib(n) = if (fibi[n] enthält einen Wert)
            return fibi[n]
         else {
            fibi[n] = fib(n-1) + fib(n-2)
            return fibi[n]
         }
```

Entweder ist der Funktionswert bereits einmal berechnet worden und kann über die Liste fibi zurückgegeben werden, oder er wird rekursiv ermittelt und gespeichert.

Divide and Conquer

Das Divide-and-Conquer Verfahren (Teile und Herrsche) arbeitet rekursiv.

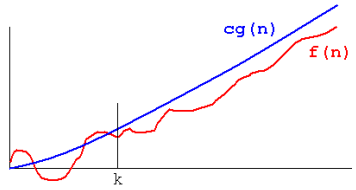
Ein Problem wird dabei, im Divide-Schritt, in zwei oder mehrere Teilprobleme aufgespalten. Das wird solange gemacht, bis die entstandenen Teilprobleme klein genug sind, um direkt gelöst zu werden. Die Lösungen werden dann in geeigneter Weise, im Conquer-Schritt, kombiniert und liefern am Ende eine Lösung für das Originalproblem.

Viele **effiziente Algorithmen** basieren auf dieser Technik . Später werden wir die Arbeitsweise von Divide-and-Conquer anhand des Sortieralgorithmus *QuickSort* unter die Lupe nehmen.

Analyse von Laufzeit und Speicherbedarf

Inhalt:

- Laufzeitanalyse
- O-Notation



Folieninhalte teilweise übernommen von PD Dr. Klaus Kriegel (Informatik B, SoSe 2007)

Analyse von Laufzeit und Speicherbedarf

Laufzeitanalyse I

Die Laufzeit T(n) eines Algorithmus ist die maximale Anzahl der Schritte bei Eingaben der Größe n. Allgemein bezeichnet man die Größe einer Eingabe mit n.

Wie man die Größe misst, muss konkret festgelegt werden, z.B. für Sortieralgorithmen wählt man sinnvollerweise die Anzahl der zu sortierenden Objekte.

Welche Schritte dabei gezählt werden (z.B. arithmetische Operationen, Vergleiche, Speicherzugriffe, Wertzuweisungen) hängt sehr stark von dem verwendeten **Modell** oder der verwendeten **Programmiersprache** ab. Sogar ein **Compiler** kann Einfluss auf die Anzahl der Schritte haben.

Oft unterscheiden sich die Laufzeiten des gleichen Algorithmus' unter Zugrundelegungung verschiedener Modelle um konstante Faktoren. Das Ziel der folgenden Betrachtungen besteht darin, den Einfluss solcher modell- und implementierungsabhängiger Faktoren auszublenden und damit einen davon unabhängigen Vergleich von Laufzeiten zu ermöglichen.

Laufzeitanalyse II

Einfaches Beispiel:

```
int dummFunction (int[] n) {
   int i = 2*12+2;                  // 2
   return i;
}
```

Die Funktion *dummFunction* benötigt in Bezug auf die Eingabegröße n folgende Arbeitsschritte:

dummFunction(n) = 2

Unabhängig von der Eingabe benötigt die Funktion konstant viele Arbeitsschritte. Wir sprechen dann von konstanter Laufzeit.

Laufzeitanalyse III

Einfaches Beispiel:

```
int sumFunction (int[] n) {
   int sum = 0;                // 1
   for (int i=0; i<n; i++)     // n-mal
      sum = sum + n[i];        //    1

   return sum;
}
```

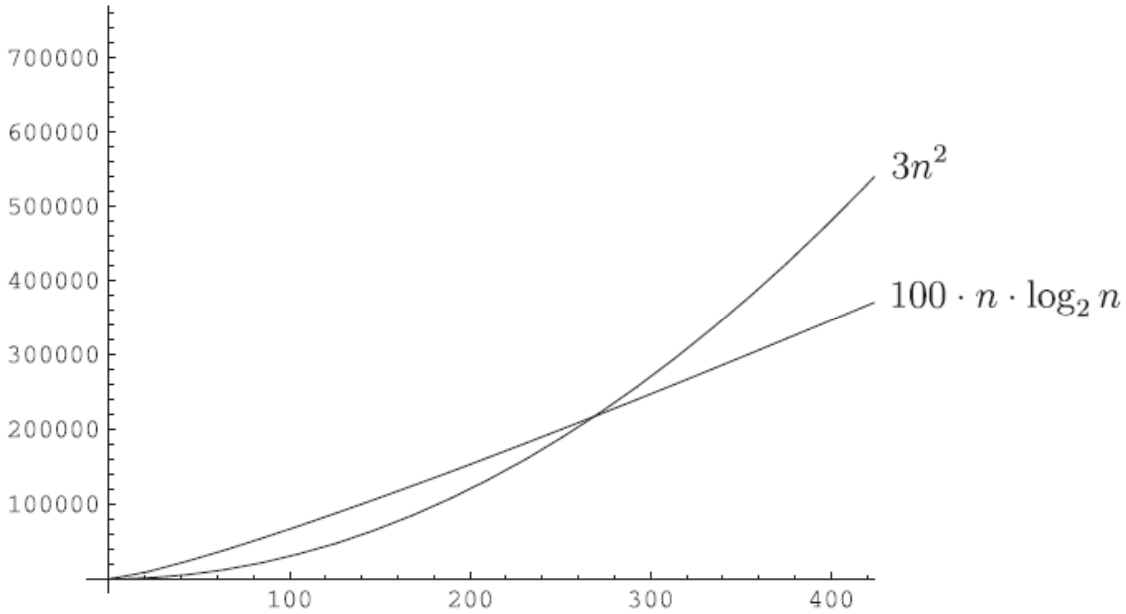
Die Funktion *sumFunction* benötigt in Bezug auf die Eingabegröße n folgende Arbeitsschritte:

sumFunction(n) = 1 + n * 1 = n+1

Für diese Fälle wollen wir konstante Werte, die keinen signifikanten Einfluss haben, weglassen. Für dieses Beispiel ergebe sich *sumFunction(n)=n*, dann sprechen wir von linearer Laufzeit.

Laufzeitanalyse IV

Vergleich von Laufzeiten bezieht sich auf das langfristige Verhalten.



Analyse von Laufzeit und Speicherbedarf

O-Notation I

Die Landau-Symbole werden verwendet, um Laufzeiten für Algorithmen anzugeben und vergleichen zu können:

$$\mathcal{O},\ \Omega,\ \Theta,\ o \textbf{ und } \omega$$

Asymptotische obere Schranke:

Definition: $g(n)$ ist *asymptotische obere Schranke* von $f(n)$, falls eine Konstante $c > 0$ und ein $n_0 \in \mathbb{N}$ existieren, so dass für alle $n \geq n_0$ gilt:

$$f(n) \leq c \cdot g(n)$$

Die Menge aller Funktionen $f(n)$, für die eine gegebenene Funktion $g(n)$ eine asymptotische obere Schranke ist, wird mit $\mathcal{O}(g(n))$ bezeichnet. Analog dazu kann man mit unteren Schranken und starken oberen bzw. unteren Schraken verfahren.

O-Notation II

Definition: Seien f und g Funktionen $\mathbb{N} \rightarrow \mathbb{R}^+$.

- Obere Schranke:

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists c > 0 \; \exists n_0 \; \forall n \geq n_0 \; f(n) \leq c \cdot g(n)\}$$

- Untere Schranke:

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0 \; \exists n_0 \; \forall n \geq n_0 \; c \cdot g(n) \leq f(n)\}$$

- Gleiches Wachstum:

$$\Theta(g(n)) = \mathcal{O}(g(n) \cap \Omega(g(n)))$$

- Starke obere Schranke:

$$o(g(n)) = \{f(n) \mid \forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; f(n) \leq c \cdot g(n)\}$$

- Starke untere Schranke:

$$\omega(g(n)) = \{f(n) \mid \forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \; c \cdot g(n) \leq f(n)\}$$

O-Notation III

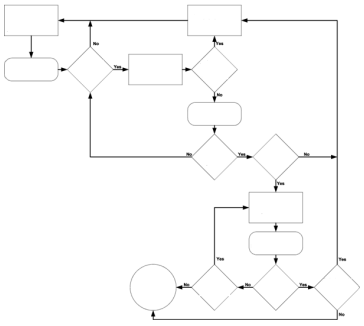
Obwohl wir mit den Landau-Symbolen Mengen angeben...

Achtung: Anstelle der korrekten Schreibwiese $f(n) \in O(g(n))$ (für $g(n)$ ist asymptotische obere Schranke von $f(n)$) wird auch häufig die Notation $f(n) = \mathcal{O}(g(n))$ verwendet.

Brute Force

Inhalt:

- Spieleprogrammierung
- MinMax-Prinzip
- AlphaBeta-Algorithmus
- TicTacToe mit unfehlbarem Gegner



Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007

Brute Force

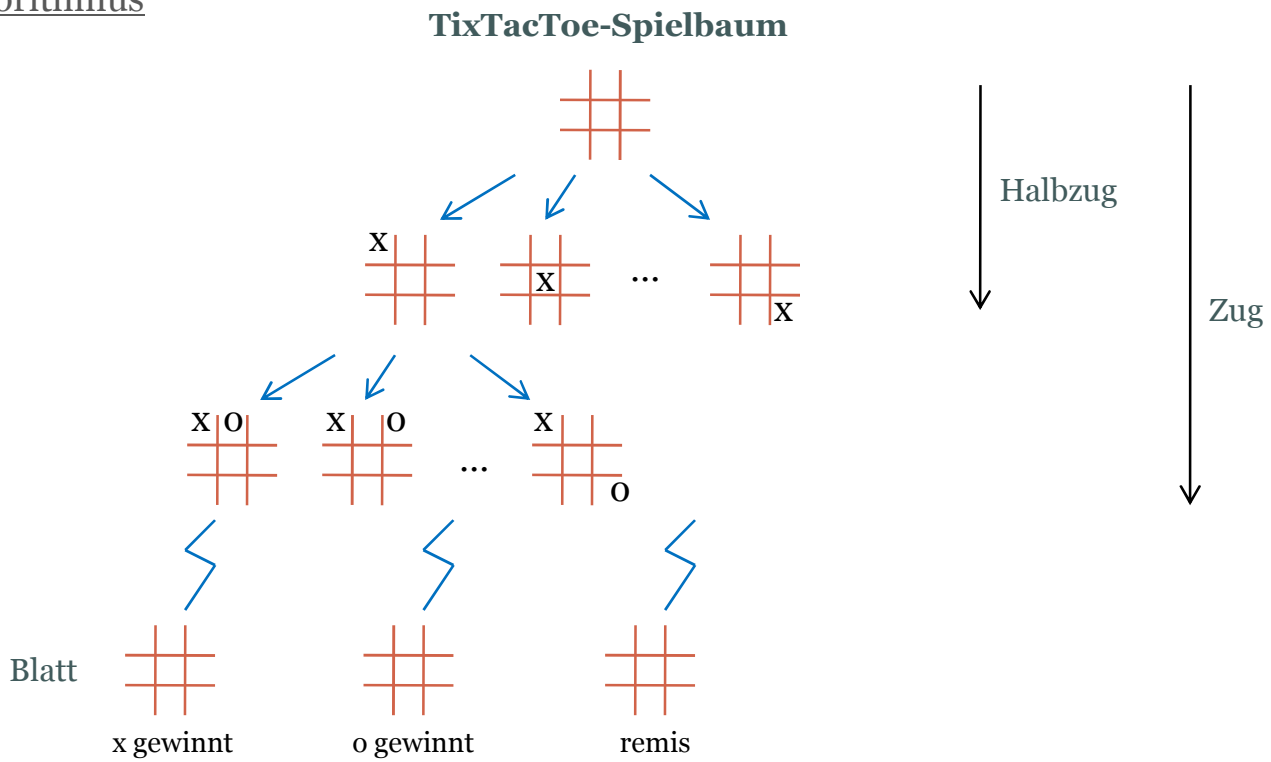
Spieleprogrammierung

Die meisten Spiele lassen sich als Suchprobleme verstehen. Dabei muss das mögliche Verhalten des Gegner in Betracht gezogen und entsprechend agiert werden.

Verhalten des Gegners ist ungewiss und meistens kontraproduktiv.

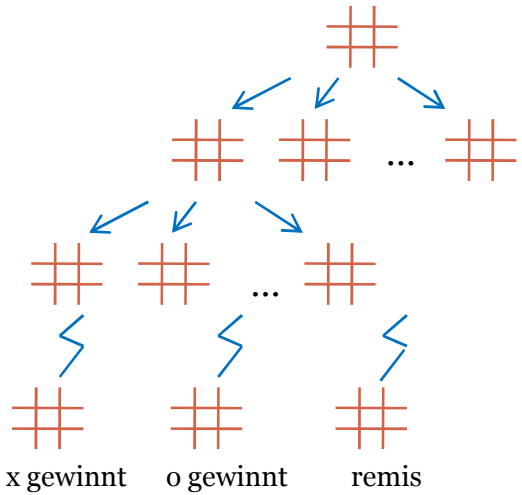
Idee: Gegner ist unfehlbar, spielt immer den bestmöglichen Zug (optimale Spielstrategie)

MinMax-Algorithmus

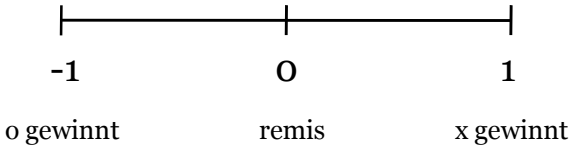


Brute Force

MinMax-Algorithmus



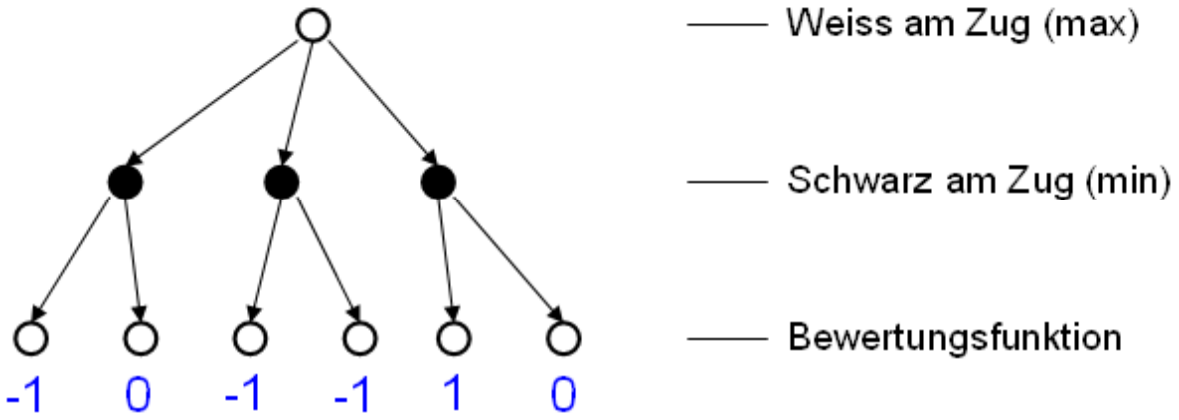
Bewertung für terminale Stellungen:



Brute Force

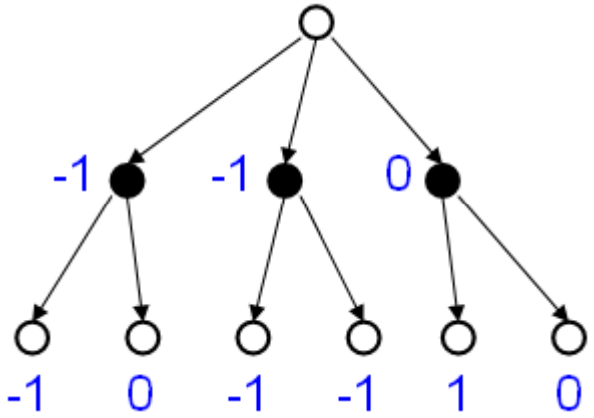
MinMax-Prinzip I

Das MinMax-Prinzip ist relativ einfach, schauen wir es uns anhand eines Beispiels einmal an. Angenommen wir haben zwei Spieler Schwarz und Weiss. Es liegt eine Stellung vor, in der Weiss drei mögliche Züge hat. Anschließend ist Schwarz an der Reihe und hat jeweils zwei mögliche Züge. Eine Bewertungsfunktion liefert uns für jede der ermittelten Stellungen einen der Werte {-1,0,1}. Dabei soll -1 für einen Sieg von Schwarz, 0 für ein Unentschieden und 1 für einen Sieg von Weiß stehen.



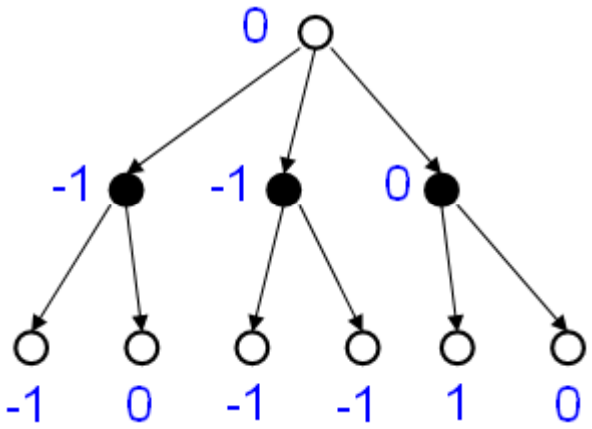
MinMax-Prinzip II

Der Spieler Schwarz gewinnt, wenn er zu einer Stellung mit der Bewertung -1 gelangt. Darum wird er sich immer für den kleineren Wert entscheiden, er minimiert seine Stellung. Schwarz wählt also immer den kleinsten Wert aus. Somit ergeben sich folgende Bewertungen an den mittleren Knoten.



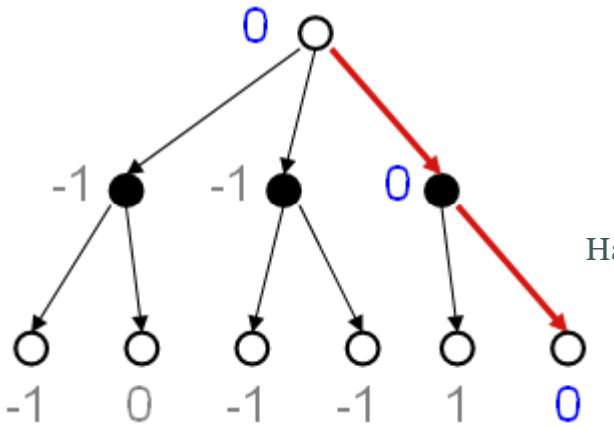
MinMax-Prinzip III

Weiss versucht seinerseits die Stellung zu maximieren und wählt immer den größten Wert aus. In diesem Fall entscheidet er sich für den Weg mit der Bewertung 0, der ihm ein Unentschieden sichert.



MinMax-Prinzip IV

Bei dem rekursiven Durchlaufen der Stellungen können wir uns nicht nur die Bewertungen speichern, sondern auch den Zug, der zu der jeweils besten Stellung führt. Am Ende der Berechnung können wir neben einer Bewertung sogar die beste Zugfolge, die sogenannte Hauptvariante, ausgeben.



Das im Wechsel stattfindende Maximieren und Minimieren gibt dem Algorithmus MinMax seinen Namen.

Brute Force

MinMax-mit unbegrenzter Suchtiefe

Wir können den MinMax-Algorithmus für den Fall formulieren, dass wir wie bei TicTacToe bis zu einer Stellung rechnen wollen und können, bei der eine Entscheidung über Sieg, Niederlage und Unentschieden getroffen und kein weiterer Zug möglich ist. Diese Stellungen nennen wir terminale Stellungen.

Stellungen in denen Weiss, also die Partei am Zug ist, die den Wert maximieren möchte, nennen wir MaxKnoten und analog dazu MinKnoten, diejenigen bei denen Schwarz am Zug ist.

sei n die aktuelle Spielstellung
S sei die Menge der Nachfolger von n

$$\text{minmax}(n) = \begin{cases} \text{Wert}(n), & \text{wenn } n \text{ terminale Stellung} \\ \max_{s \in S} \text{minmax}(s), & \text{wenn } n \text{ ist MaxKnoten} \\ \min_{s \in S} \text{minmax}(s), & \text{wenn } n \text{ ist MinKnoten} \end{cases}$$

Es gibt im Schach schätzungsweise $2{,}28 * 10^{46}$ legale Schachpositionen und im Durchschnitt ist eine Partie 50 Züge lang. Eine weiße oder schwarze Aktion wird als Halbzug definiert und zwei Halbzüge ergeben einen Zug. Das ergebe 10^{120} Partieverläufe, die wir untersuchen müssten. Diese Zahl ist unvorstellbar groß, nur zum Vergleich: Die Anzahl der Atome im Weltall wird auf 10^{80} geschätzt. Es ist uns also nicht möglich von Beginn des Spiels bis zu allen terminalen Stellungen zu rechnen.

Aber wie können wir Programme trotzdem dazu bringen, Schach zu spielen?

Brute Force

MinMax-mit begrenzter Suchtiefe

Die Idee besteht darin, nach einer bestimmten Suchtiefe abzubrechen und eine Bewertung dieser Stellung vorzunehmen. Diese Bewertung wird dann im Suchbaum zurückpropagiert .

Wie nun diese Bewertungsfunktion auszusehen hat, ist damit nicht gesagt. Sie sollte positive Werte für Weiss liefern, wobei größere Werte eine bessere Stellung versprechen und analog dazu negative Werte für favorisierte Stellungen von Schwarz. In der Schachprogrammierung wird meistens eine Funktion f verwendet, die n verschiedene Teilbewertungen fi einer Stellung S gewichtet aufaddiert und zurückgibt:

$$f(S) = \sum_{i=1}^{n} \alpha_i f_i(S)$$

Mit Hilfe der Bewertungsfunktion f, der wir ab jetzt den sprechenden Namen *evaluate* geben wollen, und einer Variable t, die die aktuelle Tiefe speichert, lässt sich nun der MinMax-Algorithmus wie folgt formulieren:

```
maxKnoten(Stellung X, Tiefe t) -> Integer
   if (t==0)
      return evaluate(X)
   else
      w := -"unendlich"
      for all Kinder X1,...,Xn von X
         v=minKnoten(Xi, t-1)
         if (v>w)
            w=v
      return w
```

Die Funktion *minKnoten* wird analog definiert.

Brute Force

MinMax-mit begrenzter Suchtiefe

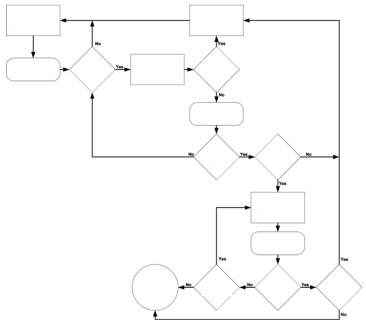
Jetzt folgt ein TicTacToe-Programm.

Den Quellcode findet Ihr hier:
http://www.java-uni.de/index.php?Seite=11

Sortieralgorithmen

Inhalt:

- InsertionSort
- BubbleSort
- QuickSort



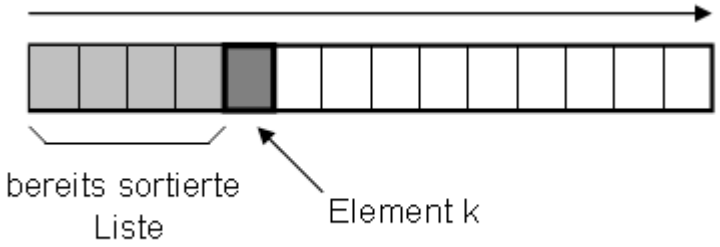
Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007

InsertionSort I

Das Problem unsortierte Daten in eine richtige Reihenfolge zu bringen eignet sich gut, um verschiedene Programmiertechniken und Laufzeitanalysen zu veranschaulichen.

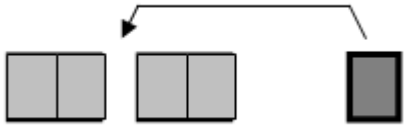
Eine sehr einfache Methode Daten zu sortieren, ist das „Einfügen in den sortierten Rest“. Wir sortieren eine Liste, indem wir durch alle Positionen der Liste gehen und das aktuelle Element an dieser Position in die Teilliste davor einsortieren.

Wenn wir das Element k einsortieren möchten, können wir davon ausgehen, dass die Teilliste vor diesem Element, die Positionen 1 bis k-1, bereits sortiert ist.



InsertionSort II

Um das Element k in die Liste einzufügen, prüfen wir alle Elemente der Teilliste, beginnend beim letzten Element, ob das Element k kleiner ist, als das gerade zu prüfenden Element j. Sollte das der Fall sein, so rückt das Element j auf die Position j+1.



Das wird solange gemacht, bis die richtige Position für das Element k gefunden wurde.

InsertionSort III

Als Beispielimplementierung schauen wir uns die Klasse **InsertionSort** an:

```
public class InsertionSort {
   private static void insert(int[] a, int pos){
      int value = a[pos];
      int j     = pos-1;

      // Alle Werte vom Ende zum Anfang der bereits sortierten Liste
      // werden solange ein Feld nach rechts verschoben, bis die
      // Position des Elements a[pos] gefunden ist. Dann wird das
      // Element an diese Stelle kopiert.
      while(j>=0 && a[j]>value){
         a[j+1] = a[j];
         j--;
      }
      a[j+1] = value;
   }

   public static void sortiere(int[] x) {
      // "Einfügen in den sortierten Rest"
      for (int i=1; i<x.length; i++)
         insert(x, i);
   }

   public static void main(String[] args) {
      int[] liste = {0,9,4,6,2,8,5,1,7,3};
      sortiere(liste);
      for (int i=0; i<liste.length; i++)
         System.out.print(liste[i]+" ");
   }
}
```

InsertionSort III

Nach der Ausführung ist die int-Liste sortiert und wird ausgegeben.

```
C:\JavaCode>java InsertionSort
0 1 2 3 4 5 6 7 8 9
```

Zur Laufzeit können wir sagen, dass die Komplexität des Algorithmus im average case mit $O(n^2)$ quadratisch ist.
Je besser die Daten vorsortiert sind, desto schneller arbeitet das Verfahren.

Im best case ist sie sogar linear, also O(n).

Sortieralgorithmen

BubbleSort I

Der BubbleSort-Algorithmus vergleicht der Reihe nach zwei benachbarte Elemente einer n-elementigen Liste x und vertauscht sie, falls sie nicht in der richtigen Reihenfolge vorliegen. Ist er am Ende der Liste angekommen wird der Vorgang wiederholt.

Der Algorithmus endet, wenn alle Elemente in der richtigen Reihenfolge vorliegen, im letzten Durchgang also keine Vertauschoperationen mehr stattgefunden haben. Dies geschieht nach maximal (n-1)*(n/2) Schritten.

Folgender Algorithmus würde immer die maximale Anzahl an Schritten ausführen, selbst wenn die Liste bereits sortiert ist:

```
for (i=1 to n-1)
   for (j=0 to n-i-1)
      if (x[j] > x[j+1])
         vertausche x[j] und x[j+1]
```

Aus theoretischer Sicht ist dieser Sortieralgorithmus sehr ineffizient. Die Komplexität ist quadratisch, also $O(n^2)$. Aus praktischen Gesichtspunkten muss hier aber gesagt werden, dass zu sortierende Listen, die bereits schon eine gewisse Vorsortierung besitzen und nicht allzu groß sind, mit BubbleSort in der Praxis relativ schnell zu sortieren sind.

Beispielsweise ist das in der Schachprogrammierung so. Es werden Listen von legalen Zügen generiert und anschließend sortiert. Als Sortierungskriterium werden meistens Funktionen verwendet, die auf Heuristiken, wie z.B. „Wie oft hat sich dieser Zug in der Vergangenheit als sehr gut erwiesen…“, basieren. Da die Zuglisten relativ kurz sind (im Schnitt unter 50) und durch intelligente Zuggeneratoren meistens schon gut vorsortiert wurden, eignet sich BubbleSort ausgezeichnet .

Sortieralgorithmen

BubbleSort II

Bei einer effizienten Implementierung bricht der Algorithmus bereits ab, wenn keine Tauschoperationen mehr durchgeführt wurden:

```
public class BubbleSort {
   public static void sortiere(int[] x) {
      boolean unsortiert=true;
      int temp;

      while (unsortiert){
         unsortiert = false;
         for (int i=0; i<x.length-1; i++)
            if (x[i] > x[i+1]) {
               temp       = x[i];
               x[i]       = x[i+1];
               x[i+1]     = temp;
               unsortiert = true;
            }
      }
   }

   public static void main(String[] args) {
      int[] liste = {0,9,4,6,2,8,5,1,7,3};
      sortiere(liste);
      for (int i=0; i<liste.length; i++)
         System.out.print(liste[i]+" ");
   }
}
```

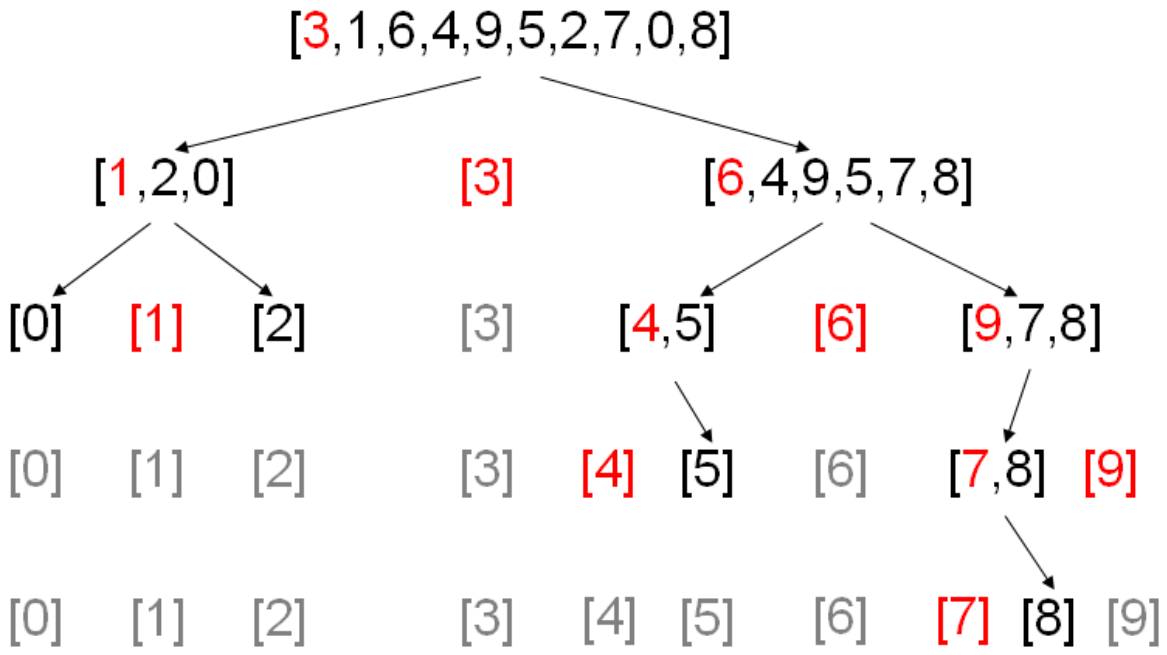
Die Liste wird sortiert ausgegeben:

```
C:\JavaCode>java BubbleSort
0 1 2 3 4 5 6 7 8 9
```

QuickSort I

Der QuickSort-Algorithmus ist ein typisches Beispiel für die Entwurfstechnik Divide-and-Conquer. Um eine Liste zu sortieren wird ein Pivotelement p ausgewählt und die Elemente der Liste in zwei neue Listen gespeichert, mit der Eigenschaft, dass in der ersten Liste die Elemente kleiner oder gleich p und in der zweiten Liste die Elemente größer p liegen. Mit den beiden Listen wird wieder genauso verfahren.

Machen wir uns das an einem Beispiel klar. Wir wollen in diesem Beispiel die Liste [3,1,6,4,9,5,2,7,0,8] sortieren. Dazu wählen wir ein Pivotelement, z.B. das erste Element in der Liste aus und teilen die Liste in zwei neue Listen auf. Diese beiden Listen werden wieder aufgespalten usw.



QuickSort II

Der Algorithmus fügt schließlich die einelementigen Listen nur noch zusammen, da deren Reihenfolge bereits richtig ist und liefert die sortierte Liste.

Es gibt verschiedene Möglichkeiten für die programmiertechnische Umsetzung:

```
import java.util.Random;

public class QuickSort {
   public static void sortiere(int x[]) {
      qSort(x, 0, x.length-1);
   }

   public static void qSort(int x[], int links, int rechts) {
      if (links < rechts) {
         int i = partition(x,links,rechts);
         qSort(x,links,i-1);
         qSort(x,i+1,rechts);
      }
   }
   ...
```

QuickSort III

weiter geht's:

```
   ...
   public static int partition(int x[], int links, int rechts) {
      int pivot, i, j, help;
      pivot = x[rechts];
      i     = links;
      j     = rechts-1;
      while(i<=j) {
         if (x[i] > pivot) {
            // tausche x[i] und x[j]
            help = x[i];
            x[i] = x[j];
            x[j] = help;
            j--;
         } else i++;
      }
      // tausche x[i] und x[rechts]
      help      = x[i];
      x[i]      = x[rechts];
      x[rechts] = help;

      return i;
   }

   public static void main(String[] args) {
      int[] liste = {0,9,4,6,2,8,5,1,7,3};
      sortiere(liste);
      for (int i=0; i<liste.length; i++)
         System.out.print(liste[i]+" ");
   }
}
```

QuickSort IV

Auch der QuickSort-Algorithmus kann die -Liste in die korrekte Reihenfolge bringen.

```
C:\JavaCode>java QuickSort
0 1 2 3 4 5 6 7 8 9
```

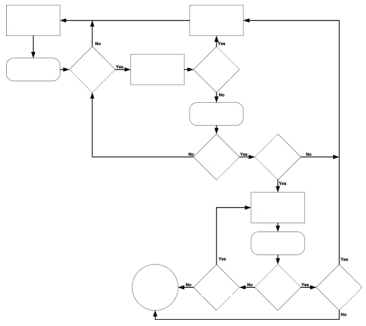
Für die Bestimmung der Laufzeit ist die Wahl des Pivotelements entscheidend. Im worst case wird immer genau das Element ausgesucht, welches die Liste so zerlegt, dass auf der einen Seite das Pivotelement vorliegt und auf der anderen Seite der Rest der Liste. Der Aufwand läge dann bei $O(n^2)$.

Im besten Fall kann das Pivotelement die beiden Listen in zwei gleichgroße Listen zerlegen. In diesem Fall hat der Berechnungsbaum eine logarithmische Tiefe und die Laufzeit des Algorithmus ist nach oben mit $O(n*\log(n))$ beschränkt. Das gilt auch für den average case.

Dynamische Programmierung

Inhalt:

- Sequenzanalyse
- Needleman-Wunsch



Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007

Dynamische Programmierung

Needleman-Wunsch-Algorithmus I

Vereinfachten Definition des Algorithmus. Gegeben sind zwei Zeichenketten x=(x1, x2,...,xn) und y=(y1, y2, ..., ym). Wir eine Datenstruktur zum Speichern der Zwischenlösungen. In diesem Fall verwenden wir eine nxm-Matrix E, die ganzzahlige Werte speichert. Wir interpretieren einen Eintrag in E(i,j) als besten Ähnlichkeitswert für die Teilsequenzen (x1, x2, ..., xi) und (y1,y2,...,yj).

Als Startwerte setzen wir:

$$E(0,0) = 0$$

$$E(i,0) = 1, \ \forall i = 0,1,\ldots,n$$
$$E(0,j) = 1, \ \forall j = 0,1,\ldots,m$$

Die Rekursionsformel sieht dann wie folgt aus:

$$E(i,j) = max \begin{cases} E(i-1,j-1) + bonus(x_i, y_j) \\ E(i-1,j) \\ E(i,j-1) \end{cases}$$

Die Bonusfunktion *bonus(a,b)* liefert eine 1, falls a und b gleiche Zeichen sind, andernfalls eine 0. So maximieren wir am Ende die größte Übereinstimmung.

Needleman-Wunsch-Algorithmus II

Hier ein Implementierungsbeispiel zu unserer Needleman-Wunsch-Variante:

```
public class NeedlemanWunsch{
   private int[][] E;
   private String n, m;

   public NeedlemanWunsch(String a, String b){
        n = a; m = b;
        E = new int[n.length()+1][m.length()+1];
        initialisiere();
   }

   public void initialisiere(){
      // Starteintrag wird auf 0 gesetzt
      E[0][0] = 0;

      // fülle die erste Zeile und erste Spalte mit 0-en
      for (int i=1; i<=n.length(); i++)
         E[i][0] = 0;
      for (int j=1; j<=m.length(); j++)
         E[0][j] = 0;
   }

   private int cost(char a, char b){
      if (a==b) return 1;
      else return 0;
   }
   ...
```

Needleman-Wunsch-Algorithmus III

weiter geht's:

```
   ...
   public int compare(){
      for (int i=1; i<=n.length(); i++)
         for (int j=1; j<=m.length(); j++)
            E[i][j] = Math.max(E[i-1][j-1]
                        + cost(n.charAt(i-1), m.charAt(j-1)),
                     Math.max(E[i-1][j], E[i][j-1]));
      return E[n.length()][m.length()];
   }

   public static void main(String[] args){
      String a = "Hallo Waldfee";
      String b = "Hola fee";
      NeedlemanWunsch NW = new NeedlemanWunsch(a, b);
      System.out.println("Ergebnis: "+NW.compare());
   }
}
```

Bei der Ausführung erhalten wir das Ergebnis 6, da die beiden Zeichenketten 6 gemeinsame Buchstaben in der gleichen Reihenfolge enthalten. Der Needleman-Wunsch Algorithmus ist auf vielfältige Weise erweiterbar, z.B. lassen sich mit kleinen Änderungen leicht Unterschiede in Textdateien finden.

Da wir jeden Eintrag der Matrix genau einmal berechnen und durchlaufen, haben wir eine Laufzeit von O(nm). Dieser Algorithmus ist sehr schnell implementiert, aber für die Bioinformatik gibt es inzwischen bessere Algorithmen (z.B. Smith-Waterman, Hirschberg).

Bildverarbeitung

Inhalt:

- RGB-Farbmodell
- Fraktal: Apfelmännchen
- Komplexe Zahlen
- BufferedImage
- Invertieren
- Farbraumkonvertierungen
- Binarisieren

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007
Burger W. ,Burge M.J.: "*Digitale Bildverarbeitung*", 2.Auflage, Springer-Verlag 2006

RGB-Farbmodell I

Das menschliche Auge kann hauptsächlich Licht von drei verschiedenen Wellenlängen wahrnehmen. Diese Wellenlängen werden vom Auge als die Farben Rot, Grün und Blau wahrgenommen. Die Mischung der Intensitäten dieser drei Wellenlängen ermöglicht es, viele verschiedene Farben zu sehen.

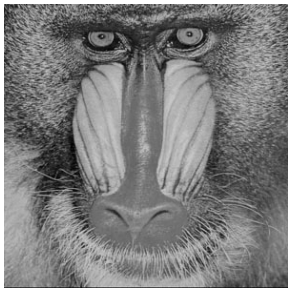
Der Bau von Monitoren wurde dadurch relativ einfach, da man, um eine bestimmte Farbe zu erzeugen, nur Licht mit drei verschiedenen Wellenlängen zu erzeugen braucht. Die exakte Mischung macht dann eine spezifische Farbe aus.

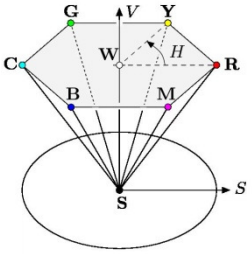
Dieses Farbmodell wird RGB-Modell genannt, da es auf den drei Grundfarben Rot, Grün und Blau basiert. **Schwarz** erhält man, wenn keine der drei Farbkanäle Licht liefert. **Weiß** durch die gleichzeitige, maximale Aktivierung aller drei Grundfarben. **Gelb** zum Beispiel wird durch eine Mischung aus Rot und Grün erzeugt. Man spricht hierbei von additiver Farbmischung, da die wahrgenommene Farbe immer heller wird, je mehr Grundfarben man hinzunimmt. Die Farbmischung eines Tuschkastens dagegen wird als subtraktiv bezeichnet. Vermischt man - wie Kinder es gerne machen - viele Farben miteinander, erhält man etwas sehr dunkles, meist ein hässliches Braun.

Farbräume

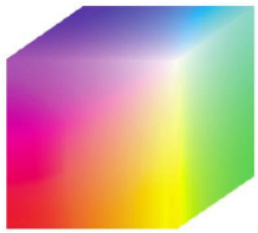
Es gibt weitere wichtige Farbräume, wie z.B.:

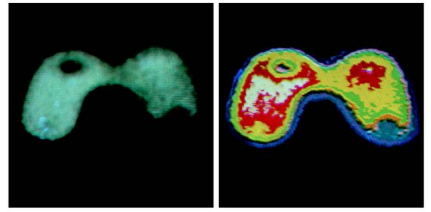
- Binäre Bilder
- Grauwertbilder
- HSI/HSV (Hue, Saturation, Value/Brightness/Intensity)
- CMY (cyan, magenta, yellow)
- CMYK
- Colorimetrische Farbräume (kalibriertes Farbsystem)
- Pseudofarben
- ...





(a) HSV





a b

FIGURE 6.20 (a) Monochrome image of the Picker Thyroid Phantom. (b) Result of density slicing into eight colors. (Courtesy of Dr. J. L. Blankenship, Instrumentation and Controls Division, Oak Ridge National Laboratory.)

RGB-Farbmodell II

Die Funktion *setColor* des **Graphics**-Objekts setzt die Farbe, die zum Zeichnen durch zukünftige Befehle benutzt werden soll. Der folgende Code erzeugt ein Fenster mit 5 farbigen Rechtecken.

```
import java.awt.*;

public class Bunt extends FensterSchliesstSchickKurz{
   public Bunt(String title1, int w, int h){
       super(title1, w, h);
   }
   public void paint(Graphics g){
       g.setColor(Color.RED);
       g.fillRect(1,1,95,100);

       g.setColor(new Color(255,0,0)); // Rot
       g.fillRect(101,1,95,100);

       g.setColor(new Color(0,255,0)); // Grün
       g.fillRect(201,1,95,100);

       g.setColor(new Color(0,0,255)); // Blau
       g.fillRect(301,1,95,100);

       g.setColor(new Color(255,255,0)); // Rot + Grün = ?
       g.fillRect(401,1,95,100);

       g.setColor(new Color(100,100,100)); // Rot + Grün + Blau = ?
       g.fillRect(501,1,95,100);
   }

   public static void main(String[] args){
       Bunt b = new Bunt ("Farbige Rechtecke", 500, 100);
   }
}
```

RGB-Farbmodell III

Jetzt sind wir in der Lage einen kontinuierlichen Farbverlauf zu erzeugen:

```
import java.awt.*;

public class Farbverlauf extends FensterSchliesstSchickKurz{
   static int fensterBreite = 600;
   static int fensterHoehe  = 300;

   public Farbverlauf(String title1, int w, int h){
       super(title1, w, h);
   }

   public void paint(Graphics g){
      for(double x=1; x<fensterBreite ; x++){
         for(double y=1; y<fensterHoehe  ; y++){
            int rot   = (int) Math.floor( 255*x/fensterBreite );
            int blau  = (int) Math.floor( 255*y/fensterHoehe  );
            g.setColor(new Color(rot,0,blau));
            g.fillRect(x,y,1,1);
         }
      }
   }
   public static void main(String[] args){
      Farbverlauf b = new Farbverlauf ("Farbverlauf", fensterBreite ,
                          fensterHoehe);
      b.setVisible(true);
   }
}
```

RGB-Farbmodell IV

Das führt zu folgender Ausgabe:

Apfelmännchen:

Machen wir das Ganze noch etwas interessanter und erzeugen geometrische Muster, die eine hohe Selbstähnlichkeit aufweisen. Das bedeutet, dass ein Muster aus mehreren verkleinerten Kopien seiner selbst besteht.

Dazu ist allerdings ein kleiner Ausflug in die Mathematik notwendig. Im Speziellen wird uns eine kleine Einführung in die Komplexen Zahlen helfen, Fraktale zu verstehen und realisieren zu können.

Komplexe Zahlen I:

Eine Komplexe Zahl besteht also aus einem Paar reeller Zahlen. Der Teil einer Komplexen Zahl der auf der x-Achse abgetragen wird, ist der Realteil, der Wert auf der y-Achse wird Imaginärteil genannt. Dieser wird durch ein 'i' markiert. Ein Beispiel für eine Komplexe Zahl ist: $1+1i$.

Mit Komplexen Zahlen können wir auch rechnen. Nehmen wir zum Beispiel die Addition

$$(-2+3i)+(1-2i),$$

so ist das Ergebnis mit einem Zwischenschritt einfach

$$(-2+3i)+(1-2i)=-2+1+(3-2)i=-1+1i.$$

Komplexe Zahlen II:

Etwas komplizierter ist die Multiplikation. Das Produkt $A*B$ der beiden Komplexen Zahlen $A=a_1+a_2i$ und $B=b_1+b_2i$ ist definiert als

$$a_1*b_1-a_2*b_2+(a_1*b_2+a_2*b_1)i.$$

Eine Besonderheit sei noch angemerkt, denn es gilt:

$$(0+1i)*(0+1i)=1i*1i=i^2=-1.$$

Komplexe Zahlen III:

Die Objektorientierung von Java bietet eine sehr einfache Möglichkeit das Rechnen mittels Komplexer Zahlen zu implementieren. Sehen wir uns ein Beispiel an:

```
class KomplexeZahl{
   double re;      // Speichert den Realteil
   double im;      // Speichert den Imaginärteil

   public KomplexeZahl(double r, double i){  // Konstruktor
      re = r;
      im = i;
   }

   public KomplexeZahl plus(KomplexeZahl k){
      return new KomplexeZahl(re + k.re,im + k.im);
   }

   public KomplexeZahl mal(KomplexeZahl k){
      return new KomplexeZahl(re*k.re - im*k.im, re*k.im + im*k.re);
   }

   public double norm(){  // Berechnet den Abstand der Zahl vom Ursprung
      return Math.sqrt(re*re + im*im);
   }

   public String text(){  // Gibt die Zahl als Text aus
      return re+" + "+im+" i";
   }
}
```

Komplexe Zahlen IV:

Die Rechnung mit Komplexen Zahlen erweist sich jetzt als genauso einfach, wie die Rechnung mit natürlichen Zahlen:

```
KomplexeZahl zahl1 = new KomplexeZahl(1, 1);
KomplexeZahl zahl2 = new KomplexeZahl(2, -2);
System.out.println("Ergebnis: "+(zahl1.plus(zahl2)).text());
```

Liefert:

```
C:\Java>java KomplexeZahl
Ergebnis: 3.0 + -1.0 i
```

Apfelmännchen I:

Wir können jetzt mit Komplexen Zahlen rechnen, aber was hat das mit Bildern zu tun? Unsere Motivation ist der Einstieg in die Fraktale.

Komplexe Zahlen eignen sich zur Berechnung von Bildern, da jede Zahl direkt einem Punkt auf einer 2-dimensionalen Fläche zugeordnet werden kann. Berechnen wir also für eine Fläche von Komplexen Zahlen jeweils für jede dieser Zahlen einen Funktionswert, so füllen wir Stück für Stück eine besagte Fläche mit Werten. Jetzt müssen wir jedem Funktionswert nur noch eine Farbe zuordnen und fertig ist das Meisterwerk.

Betrachten wir zunächst folgendes Programm und seine Ausgabe.

```
import java.awt.*;

public class Fraktal extends FensterSchliesstSchickKurz{
   static int resolution = 500;

   public Fraktal (String title1, int w, int h){
      super(title1, w, h);
   }

   public int berechnePunkt(double x, double y){
      KomplexeZahl c = new KomplexeZahl(x,y);
      KomplexeZahl z = new KomplexeZahl(0,0);
      for (double iter = 0;iter < 40; iter ++){
         z = (z.mal(z)).plus(c);
         if(z.norm() > 4) break;
      }
      return (int)Math.floor((255)*iter/40);
   }
   ...
```

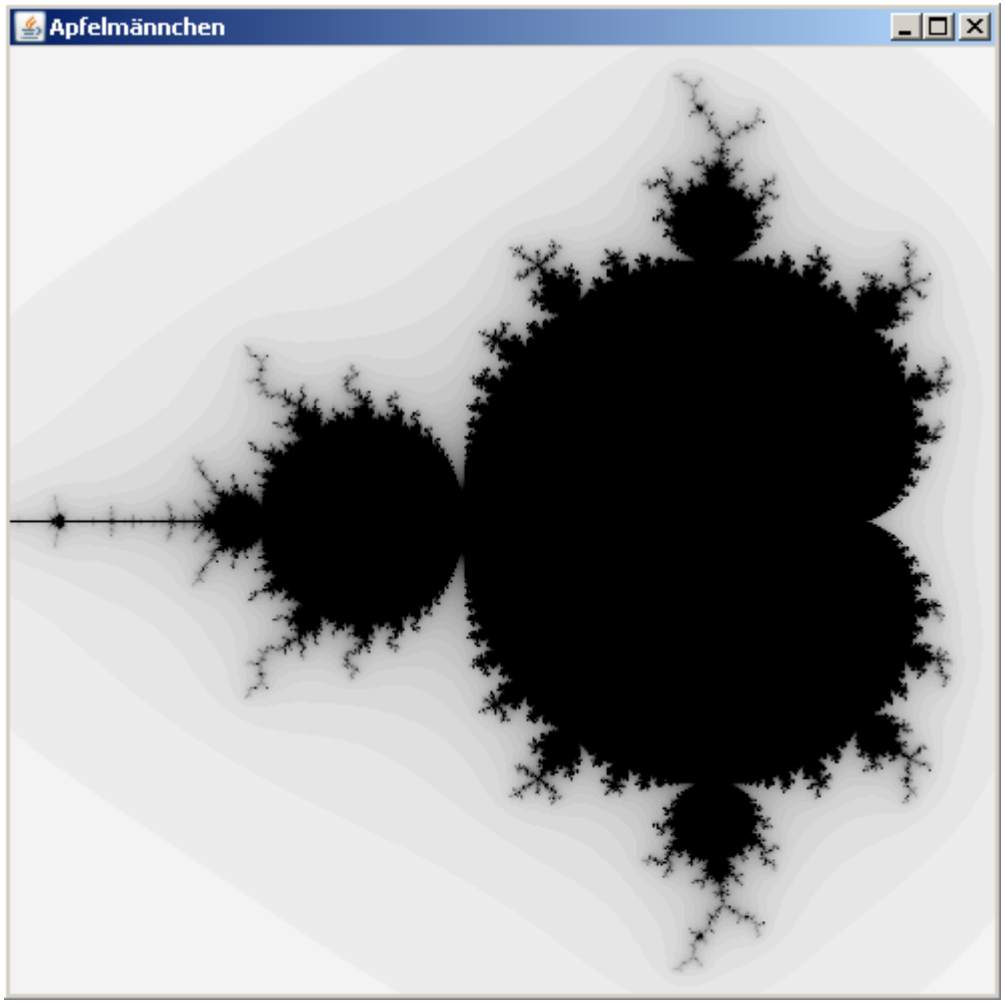
Apfelmännchen II:

weiter geht's:

```
   ...
   public void berechneBild(Graphics g){
      for (double x=0; x<aufloesung;x++){
         for (double y=0; y<aufloesung; y++){
            int Fxy = 255 - berechnePunkt(2.5*x/aufloesung - 1.9, 2.5*y/aufloesung - 1.3);
            g.setColor(new Color(Fxy, Fxy, Fxy));
            g.fillRect((int)x, (int)y, 1, 1);
         }
      }
   }

   public void paint(Graphics g){
      berechneBild(g);
   }

   public static void main(String[] args){
      Fraktal f = new Fraktal ("Apfelmännchen", aufloesung, aufloesung);
      f.setVisible(true);
   }
}
```

Apfelmännchen III:

Unser Programm erzeugt ein Fraktal mit dem berühmten Namen Apfelmännchen und das sieht wie folgt aus:



Apfelmännchen IV:

Unter Verwendung der Klasse BufferedImage wollen wir mehr Farbe ins Spiel bringen:

```
public class FraktalBuntFinal extends FensterSchliesstSchickKurz{
   static int aufloesung     = 100;
   static int fensterRandLRU = 5;   // Fensterrand links, rechts, unten
   static int fensterRandO   = 31;  // Fensterrand oben
   static int itermax        = 2000;// Maximale Anzahl von Iterationen
   static int schwellenwert  = 35;  // bis zum Erreichen des Schwellwerts.
   static int[][] farben     = {
              {   1,  255,255,255}, // Hohe Iterationszahlen sollen hell,
              { 300,   10, 10, 40}, // die etwas niedrigeren dunkel,
              { 500,  205, 60, 40}, // die "Spiralen" rot
              { 850,  120,140,255}, // und die "Arme" hellblau werden.
              {1000,   50, 30,255}, // Innen kommt ein dunkleres Blau,
              {1100,    0,255,  0}, // dann grelles Grün
              {1997,   20, 70, 20}, // und ein dunkleres Grün.
              {itermax, 0,  0,  0}};// Der Apfelmann wird schwarz.

   static double bildBreite  = 0.000003628;
   // Der Ausschnitt wird auf 3:4 verzerrt
   static double bildHoehe   = bildBreite*3.f/4.f;
   // Die Position in der Komplexen-Zahlen-Ebene
   static double [] bildPos  = {-0.743643135-(2*bildBreite/2),
                                0.131825963-(2*bildBreite*3.f/8.f)};

   BufferedImage bild;
   ...
```

Apfelmännchen V:

weiter geht's:

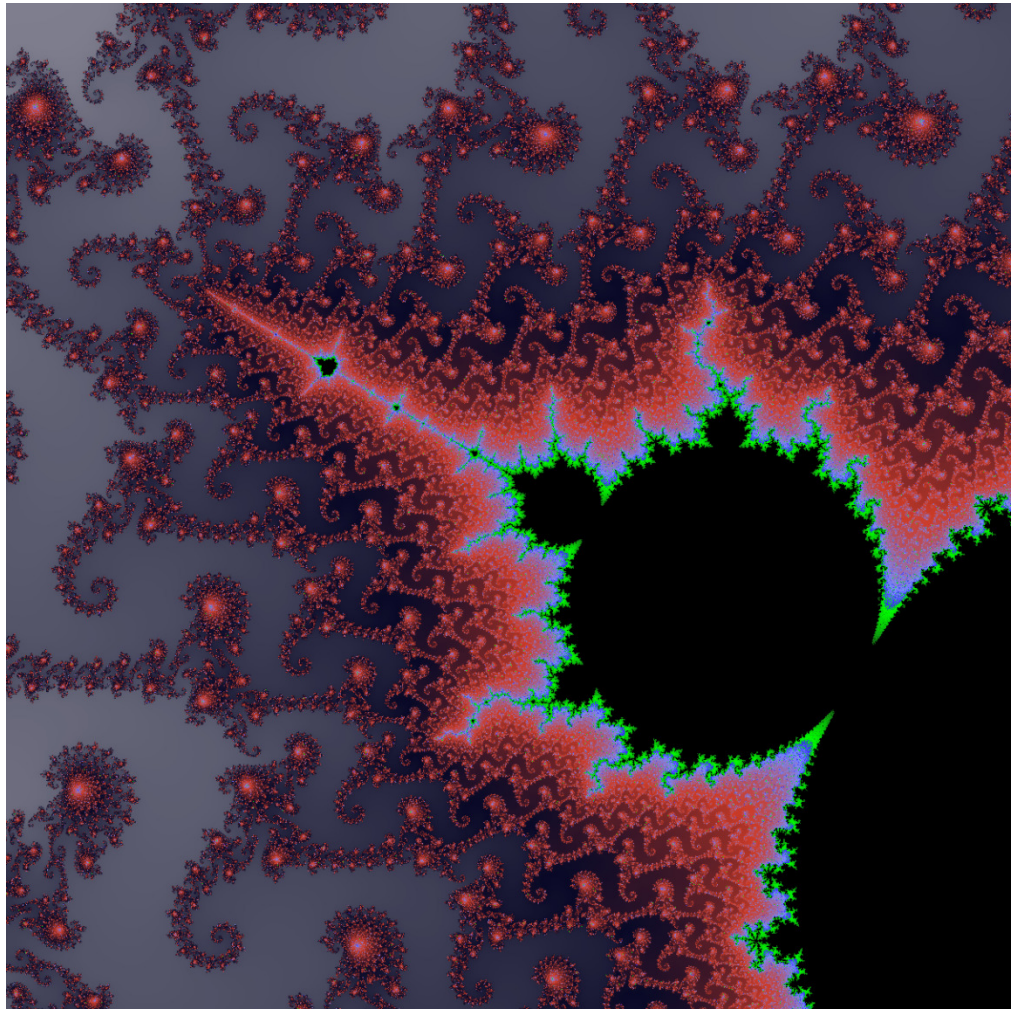
```
...
public FraktalBuntFinal (String title1, int w, int h){
   super(title1, w, h);
   bild = new BufferedImage(aufloesung, aufloesung,BufferedImage.TYPE_INT_RGB);
   Graphics gimg = bild.createGraphics();

   berechneBild(gimg);
      try {
         ImageIO.write(bild, "BMP", new File("ApfelmannBunt.bmp"));
      } catch(IOException e){
         System.out.println("Fehler beim Speichern!");
   }
}

public Color berechneFarbe(int iter){
   int F[] = new int[3];
   for (int i=1; i<farben.length-1; i++){
      if (iter < farben[i][0]){
         int iterationsInterval = farben[i-1][0]-farben[i][0];
         double gewichtetesMittel = (iter-farben[i][0])/(double)iterationsInterval;
         for (int f=0; f<3; f++){
            int farbInterval = farben[i-1][f+1]-farben[i][f+1];
            F[f] = (int)(gewichtetesMittel*farbInterval)
                        +farben[i][f+1];
         }
         return new Color(F[0], F[1], F[2]);
      }
   }
   return Color.BLACK;
}
...
```

Apfelmännchen VI:

```
   public int berechnePunkt(KomplexeZahl c){
      KomplexeZahl z = new KomplexeZahl(0,0);
      int iter = 0;
      for (; (iter <= itermax) && (z.norm() < schwellenwert); iter ++)
         z = (z.mal(z)).plus(c);
      return iter;
   }

   public void berechneBild(Graphics g){
      for (int x=0; x<aufloesung;x++){
         for (int y=0; y<aufloesung; y++){
            KomplexeZahl c = new KomplexeZahl(bildBreite*(double)(x)/aufloesung + bildPos[0],
                                              bildBreite*(double)(y)/aufloesung + bildPos[1]);
            g.setColor(berechneFarbe(berechnePunkt(c)));
            g.fillRect(x, y, 1, 1);
         }
      }
   }

   public void paint(Graphics g){
      g.drawImage(bild, fensterRandLRU, fensterRandO, this);
   }

   public void update(Graphics g){
      paint(g);
   }

   public static void main(String[] args){
      FraktalBuntFinal f = new FraktalBuntFinal("Apfelmännchen - Bunt", aufloesung+2*fensterRandLRU,
                               aufloesung+fensterRandLRU+fensterRandO);
      f.setVisible(true);
   }
}
```

Apfelmännchen VII:

Wir erhalten folgende Ausgabe:



Bilder invertieren I:

Als erstes wollen wir ein Bild invertieren. Dies bedeutet jeden Bildpixel mit seinem farblichen Gegenspieler einzufärben. Benutzen wir die gewohnte RGB-Darstellung, kann eine Invertierung leicht durchgeführt werden. Ist der Helligkeitswert einer der Grundfarben x, so ist sein invertierter Helligkeitswert 255-x. Aus 0 wird also 255, aus 255 wird 0. 127 wird zu 128, ein mittelhelles Grau bleibt demnach fast unverändert.

Die Sache wird ein wenig komplizierter, da die drei Farbkanäle gemeinsam in einem Integer gespeichert werden. Dabei geht man wie folgt vor:

> Sind r, g und b die Helligkeitswerte der drei Grundfarben, jeweils im Bereich von 0 bis 255, dann ergibt sich der gemeinsame RGB-Wert aus:
>
> rgb=256*256*r+256*g+b.

Das Schöne an dieser Art der Darstellung ist, dass alle drei Farben speicherplatzgünstig in einem einzigen int gespeichert und aus diesem wieder eindeutig rekonstruiert werden können. Ein int in Java verfügt über 8 Byte. Hier sind die ersten beiden Bytes für den Alpha-Wert (Transparenz) der Zahl reserviert, der in der reinen RGB-Darstellung keine Rolle spielt. Die restlichen 3 Byte kodieren jeweils einen Farbkanal.

Bilder invertieren II:

Um an den konkreten Wert nur eines Bytes aus dem int zu kommen, setzten wir die anderen auf 0 und verschieben das entsprechende Byte an die niederwertigste Stelle. Dazu benutzen wir das bitweise UND, dass die UND-Funktion elementweise auf zwei Bitworte anwendet. Die einzelnen Farbkanäle ließen sich dann durch

```
int rot   = (farbe & 256*256*255)/(256*256);
int gruen = (farbe & 256*255)/256;
int blau  = (farbe & 255);
```

extrahieren.

Bilder invertieren III:

Jetzt können wir eine Funktion zum Invertieren eines RGB-Bildes implementieren:

```
public BufferedImage invertiere(BufferedImage b){
   int x = b.getWidth();
   int y = b.getHeight();
   BufferedImage ib = new BufferedImage(x,y,BufferedImage.TYPE_INT_RGB);
   for (int i=0; i<x; i++){
      for (int k=0; k<y; k++){
         // 255 + 256*255 + 256*256*255
         int neu = 255 + 256*255 + 256*256*255 - b.getRGB(i,k);
         ib.setRGB(i,k,neu);
      }
   }
   return ib;
}
```

Bilder invertieren III:

Das könnte folgende Ausgabe liefern:

Farbbilder zu Grauwertbilder konvertieren I:

Das gleiche Prinzip können wir nutzen, um aus einem Farbbild ein Graubild zu erstellen. Laufen wir wieder über jeden Bildpunkt, berechnen einen durchschnittlichen Helligkeitswert aus den drei Kanälen und schreiben diesen in die drei Kanäle des Pixels im neuen Bild. Dazu ist es allerdings nötig, zuerst den Farbwert jedes einzelnen Kanals zu bestimmen.

Das lässt sich realisieren, indem wir die beiden Zeilen innerhalb der for-Schleife von *invertiere* durch diese ersetzen:

```
int alt      = b.getRGB(i,k);
int rot      = (alt & 256*256*255)/(256*256);
int gruen    = (alt & 256*255)/256;
int blau     = (alt & 255);
int grauwert = (int) Math.floor(0.299*rot + 0.587*gruen + 0.114*blau);
int neu      = 256*256*grauwert + 256*grauwert + grauwert;
ib.setRGB(i,k,neu);
```

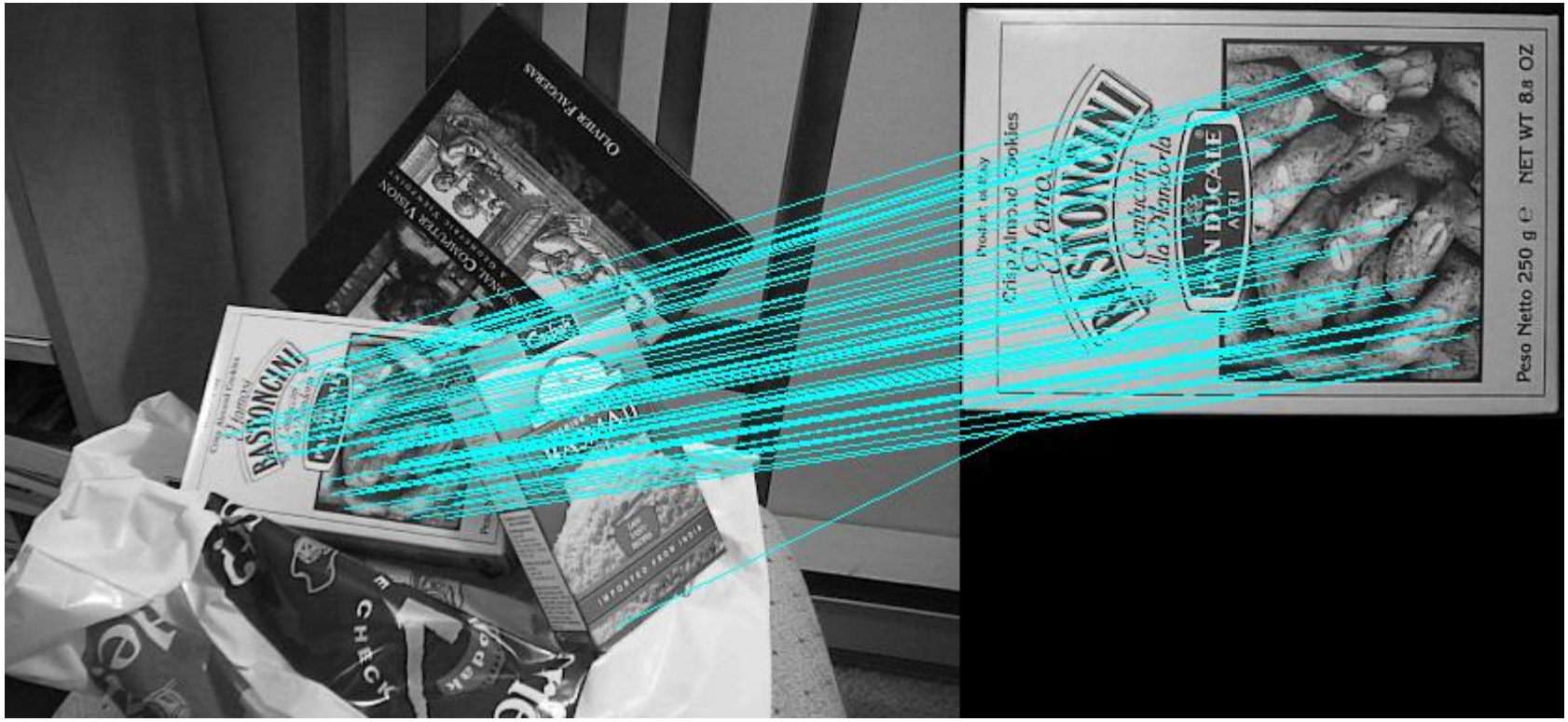
Eine Besonderheit ist hierbei, dass wir anstelle des einfachen arithmetischen Mittelwerts der Helligkeiten ein gewichteten Mittelwert berechnen. Dies ist durch die Biologie des menschlichen Auges motiviert. Grüne Farbanteile im Licht werden stärker wahrgenommen als rote und diese wiederum stärker als blaue.

Farbbilder zu Grauwertbilder konvertieren II:

Liefert für das vorherige Beispiel folgende Ausgabe:

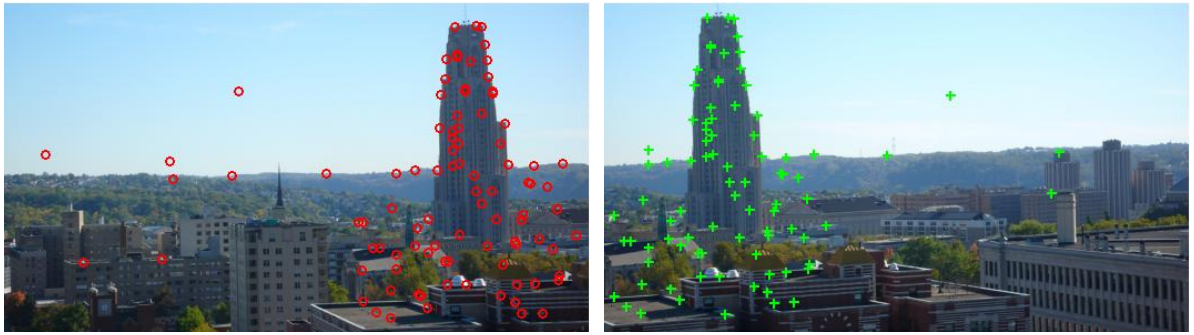
Farbbilder zu Grauwertbilder konvertieren III:

Eine Anwendung ist beispielsweise die Objektwiedererkennung. Die meisten Merkmalsdetektoren arbeiten auf Grauwertbildern:



Farbbilder zu Grauwertbilder konvertieren IV:

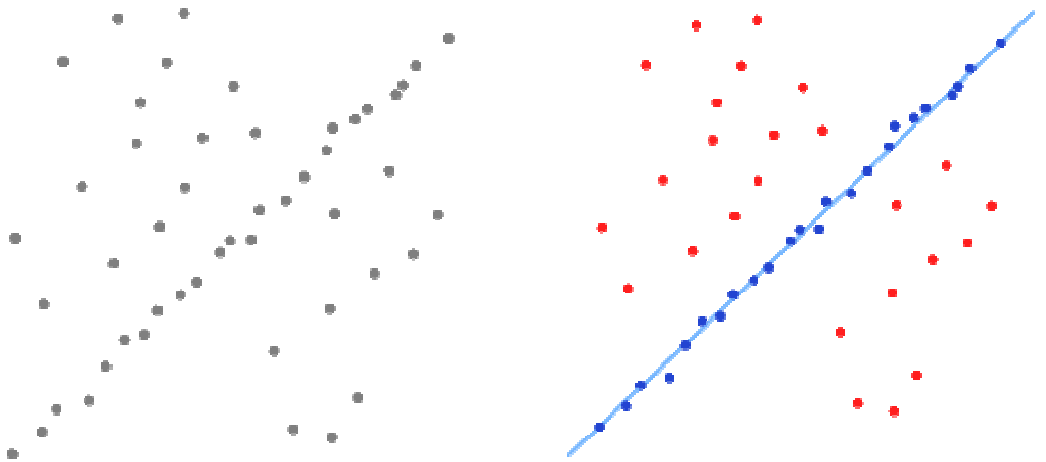
Verschiedene Aufnahmen



zusammenstitchen (Mosaiking):

Mosaiking I:

Ransac-Algorithmus:



1) Wähle zufällig so viele Punkte aus den Datenpunkten wie nötig sind, um die Parameter des Modells zu berechnen. Das geschieht in Erwartung, dass diese Menge frei von Ausreißern ist.

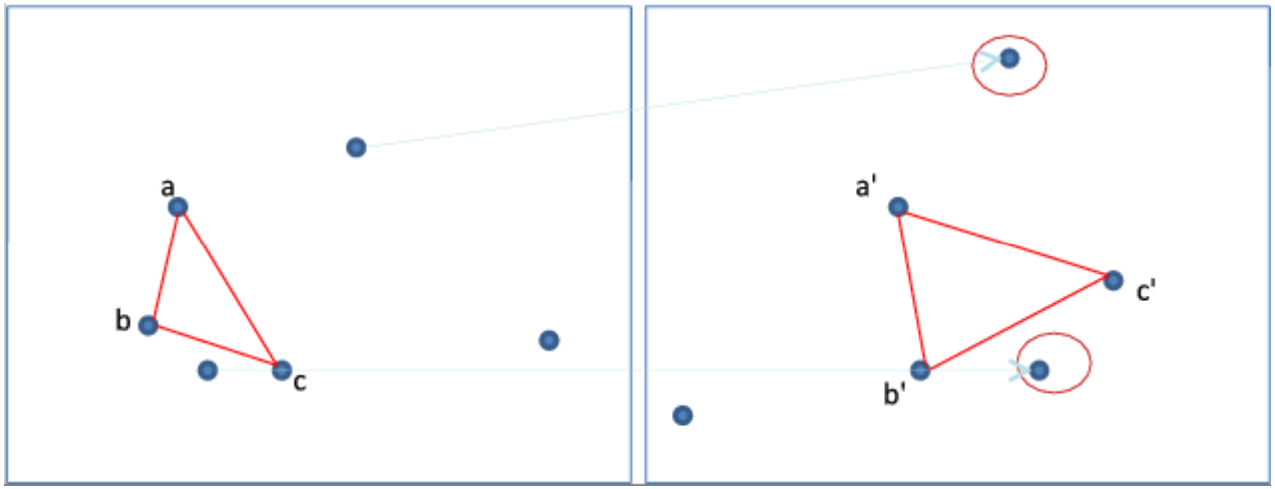
2) Ermittle mit den gewählten Punkten die Modellparameter.

3) Bestimme die Teilmenge der Messwerte, deren Abstand zur Modellkurve kleiner als ein bestimmter Grenzwert ist (diese Teilmenge wird Consensus set genannt). Alle Punkte, die einen größeren Abstand haben, werden als grobe Fehler angesehen. Enthält die Teilmenge eine gewisse Mindestanzahl an Werten, wurde vermutlich ein gutes Modell gefunden und der Consensus set wird gespeichert.

4) Wiederhole die Schritte 1–3 mehrmals.

Mosaiking II:

Beispiel:



Least Squares mit Heuristik versus RANSAC I:

Least Squares (Methode der kleinsten Quadrate) mit der Heuristik, dass der schlechteste Punkt aus der Menge entfernt wird, ist ein Standardverfahren, um derartige Probleme zu lösen . Für ein gegebenes Modell wird die Summe der quadratischen Abweichungen minimiert. Zunächst wird angenommen, dass alle Punkte zum vorliegenden Modell gehören und für jeden Punkt die quadratische Abweichung berechnet.

Ziel ist eine Gerade in der Form: f(x)=a*x+b. Für n Datenpunkte {(x1, y1),(x2, y2),...,(xn, yn)} kann die Gerade mit Hilfe des Verschiebungssatzes angegeben werden

$$a = \frac{\sum_{i=1}^{n}(y_i-\mu_y)(x_i-\mu_x)}{\sum_{i=1}^{n}(x_i-\mu_x)^2} \text{ und } b = \mu_y - a \cdot \mu_x$$

mit

$$\mu_x = \frac{1}{n}\sum_{i=1}^{n} x_i \text{ und } \mu_y = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

Least Squares mit Heuristik versus RANSAC II:

Zwei Punkte, die diese Gerade beschreiben sind p1=(0,b) und p2=(1,a*b). Jetzt lässt sich die Distanz für alle Punkte zu dieser Geraden berechnen, z.B. für einen Punkt p0=(x0, y0) mit

$$dist = \frac{|(x_2 - x_1)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}.$$

Sollten Punkte ausserhalb der erlaubten Toleranz t liegen, so wird der am entferntest liegende gelöscht und für die verbleibenden Punkte ein neues Geradenmodell berechnet. Das wird solange wiederholt, bis alle Punkte innerhalb einer Toleranz liegen.

Beispielcode findet sich hier:
http://www.java-uni.de/forum/viewtopic.php?f=9&t=227

Binarisierung I:

Den Grauwert kann man auch verwenden, um das Bild zu binarisieren. Dies bedeutet, man entscheidet für jeden Pixel, ob er entweder schwarz oder weiß sein soll, abhängig von der Intensität des Grauwerts.

```
int alt = b.getRGB(i,k);
int rot   = (alt & 256*256*255)/(256*256);
int gruen = (alt & 256*255)/256;
int blau  = (alt & 255);
int grauwert = (int) Math.floor(0.299*rot + 0.587*gruen + 0.114*blau);
if (grauwert > 125)
   ib.setRGB(i,k,256*256*255 + 256*255 + 255);
else
   ib.setRGB(i,k,0);
```

Unsere Binarisierungsmethode setzt Pixel mit einem Grauwert *grau* größer als 125 auf Weiss und alle anderen auf Schwarz. Da alle Bildpunkte mit dem gleichen Schwellwert binarisiert werden, nennt man dieses Verfahren auch globale Binarisierung.

Binarisierung II:

Unser Beispiel binarisiert:

Binarisierung III:

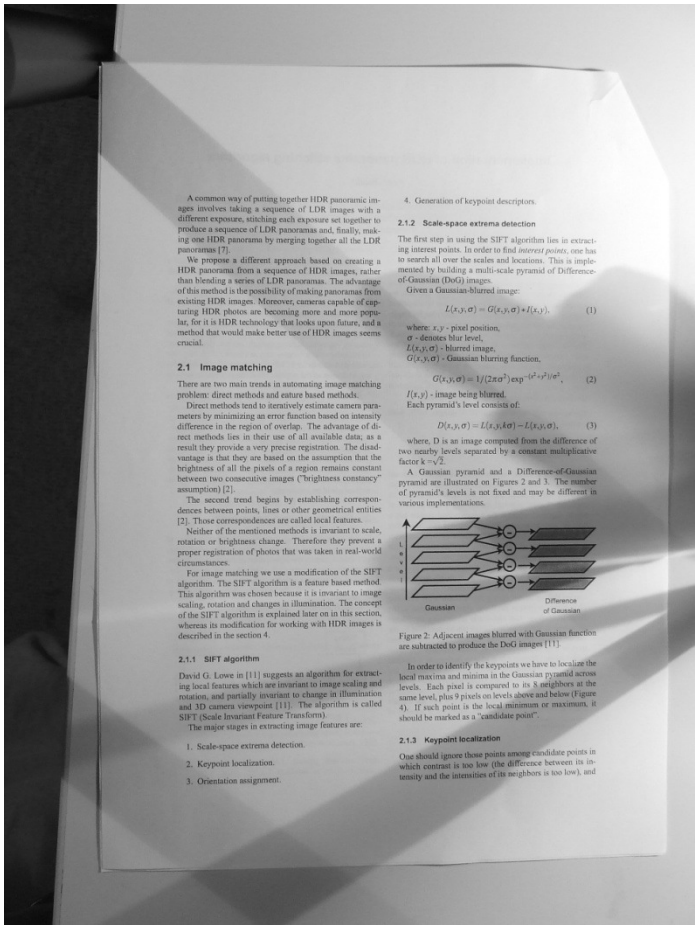
Bessere Verfahren ermitteln die Schwellwerte für kleinere Flächen, diese bezeichnet man als lokale Binarisierung.

mein aktuelles Verfahren ☺

Drei Standardverfahren (von links nach rechts: Kavallieratou, Niblack, YunLi)

Binarisierung IV:

Binarisierung von Text zur Eliminierung von Schatten und Identifikation von Text:



A common way of putting together HDR panoramic images involves taking a sequence of LDR images with a different exposure, stitching each exposure set together to produce a sequence of LDR panoramas and, finally, making one HDR panorama by merging together all the LDR panoramas [7].

We propose a different approach based on creating a HDR panorama from a sequence of HDR images, rather than blending a series of LDR panoramas. The advantage of this method is the possibility of making panoramas from existing HDR images. Moreover, cameras capable of capturing HDR photos are becoming more and more popular, for it is HDR technology that looks upon future, and a method that would make better use of HDR images seems crucial.

2.1 Image matching

There are two main trends in automating image matching problem: direct methods and enture based methods.

Direct methods tend to iteratively estimate camera parameters by minimizing an error function based on intensity difference in the region of overlap. The advantage of direct methods lies in their use of all available data; as a result they provide a very precise registration. The disadvantage is that they are based on the assumption that the brightness of all the pixels of a region remains constant between two consecutive images ("brightness constancy" assumption) [2].

The second trend begins by establishing correspondences between points, lines or other geometrical entities [2]. Those correspondences are called local features.

Neither of the mentioned methods is invariant to scale, rotation or brightness change. Therefore they prevent a proper registration of photos that was taken in real-world circumstances.

For image matching we use a modification of the SIFT algorithm. The SIFT algorithm is a feature based method. This algorithm was chosen because it is invariant to image scaling, rotation and changes in illumination. The concept of the SIFT algorithm is explained later on in this section, whereas its modification for working with HDR images is described in the section 4.

2.1.1 SIFT algorithm

David G. Lowe in [11] suggests an algorithm for extracting local features which are invariant to image scaling and rotation, and partially invariant to change in illumination and 3D camera viewpoint [11]. The algorithm is called SIFT (Scale Invariant Feature Transform).

The major stages in extracting image features are:

1. Scale-space extrema detection.
2. Keypoint localization.
3. Orientation assignment.
4. Generation of keypoint descriptors.

2.1.2 Scale-space extrema detection

The first step in using the SIFT algorithm lies in extracting interest points. In order to find *interest points*, one has to search all over the scales and locations. This is implemented by building a multi-scale pyramid of Difference-of-Gaussian (DoG) images.

Given a Gaussian-blurred image:

$$L(x,y,\sigma) = G(x,y,\sigma) * I(x,y), \qquad (1)$$

where: x, y - pixel position,
σ - denotes blur level,
$L(x,y,\sigma)$ - blurred image,
$G(x,y,\sigma)$ - Gaussian blurring function,

$$G(x,y,\sigma) = 1/(2\pi\sigma^2)\exp^{-(x^2+y^2)/\sigma^2}, \qquad (2)$$

$I(x,y)$ - image being blurred.
Each pyramid's level consists of:

$$D(x,y,\sigma) = L(x,y,k\sigma) - L(x,y,\sigma), \qquad (3)$$

where, D is an image computed from the difference of two nearby levels separated by a constant multiplicative factor $k = \sqrt{2}$.

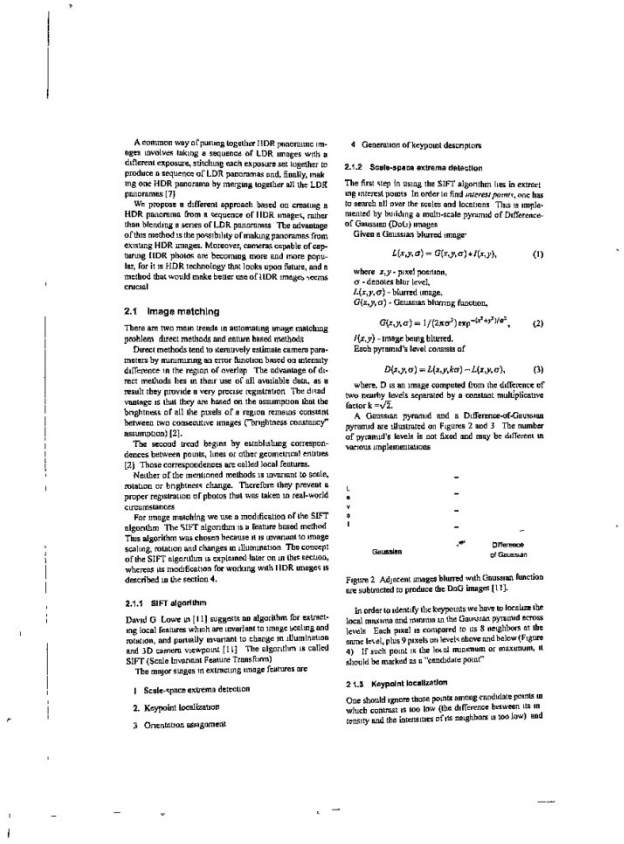
A Gaussian pyramid and a Difference-of-Gaussian pyramid are illustrated on Figures 2 and 3. The number of pyramid's levels is not fixed and may be different in various implementations.

Figure 2: Adjacent images blurred with Gaussian function are subtracted to produce the DoG images [11].

In order to identify the keypoints we have to localize the local maxima and minima in the Gaussian pyramid across levels. Each pixel is compared to its 8 neighbors at the same level, plus 9 pixels on levels above and below (Figure 4). If such point is the local minimum or maximum, it should be marked as a "candidate point".

2.1.3 Keypoint localization

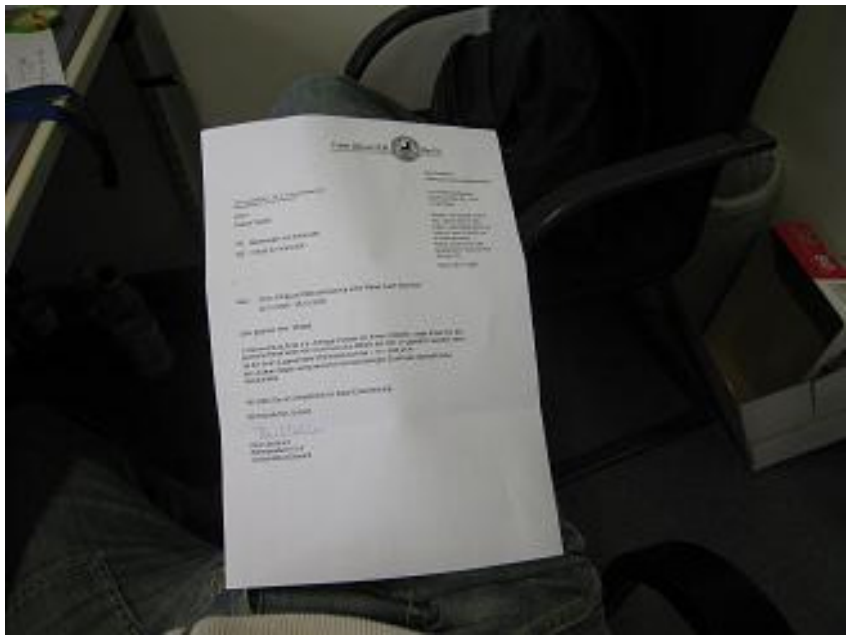
One should ignore those points among candidate points in which contrast is too low (the difference between its intensity and the intensities of its neighbors is too low), and



Bildverarbeitung

Binarisierung V:

Binarisierung von Text zur Eliminierung von Schatten und Identifikation von Text:



Programmiersprachen

Inhalt:

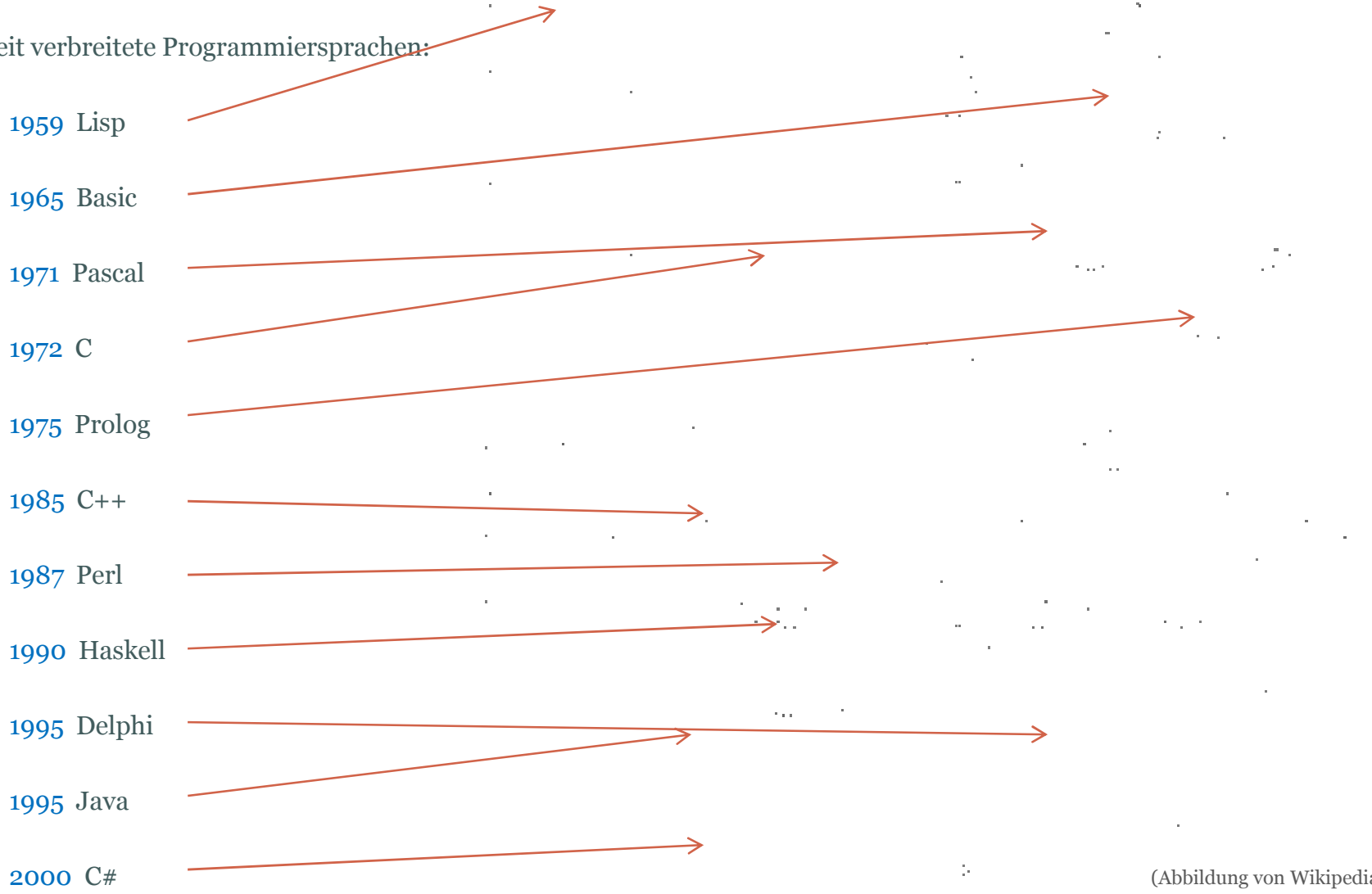
- Geschichte der Programmiersprachen
- Programmierparadigma
- Programmiersprachen im Studium
- Funktionale Programmierung mit Haskell
- Funktionale versus Imperative Programmierung
- Logische Programmierung mit Prolog
- Motivation zu Java

Folieninhalte teilweise übernommen von Prof. Heinz Schweppe (ALP II, SoSe 2008) und Prof. Löhr (ALP II, SoSe 2004)

Programmiersprachen

Geschichte der Programmiersprachen

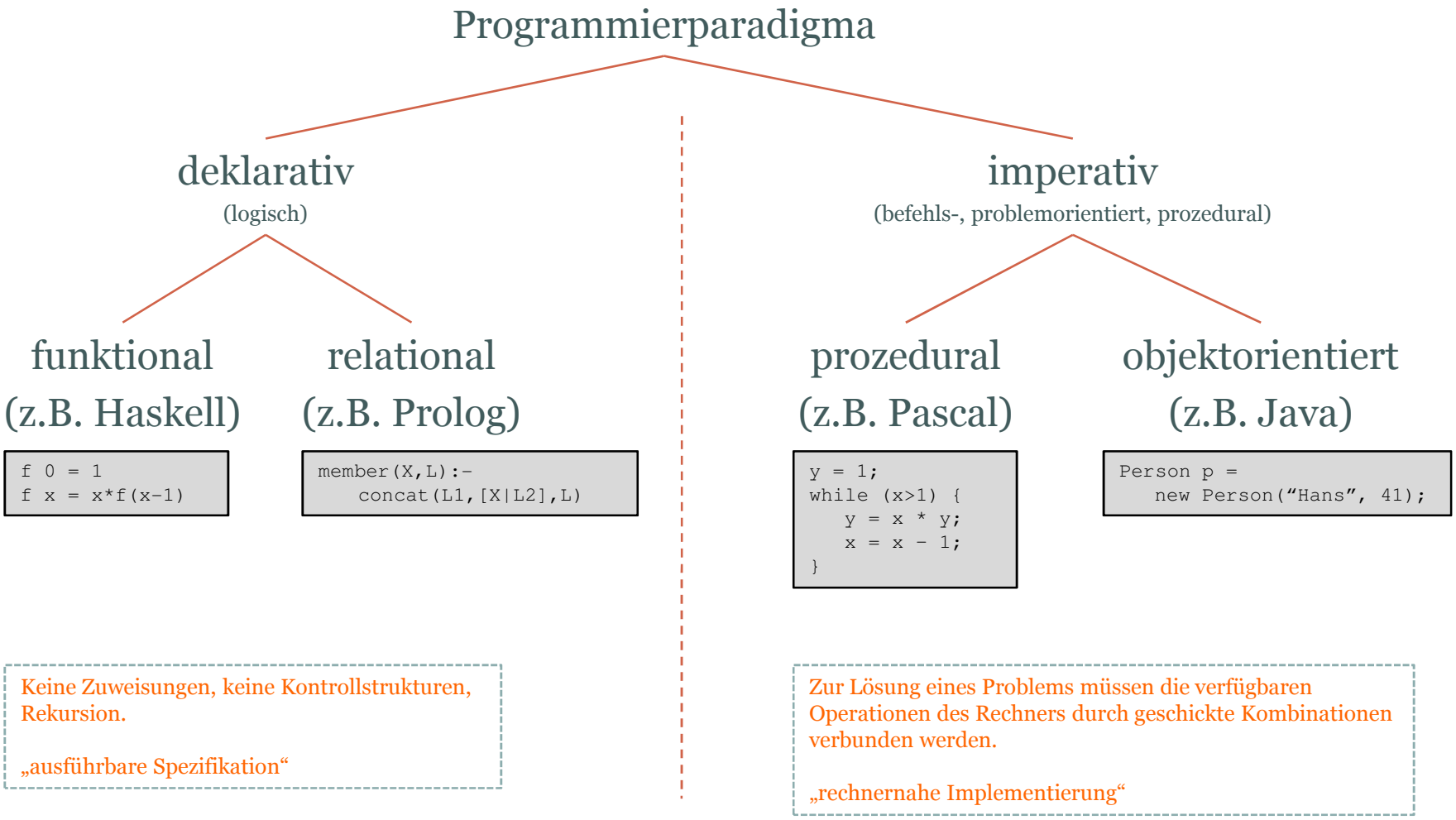
Weit verbreitete Programmiersprachen:



(Abbildung von Wikipedia)

Programmiersprachen

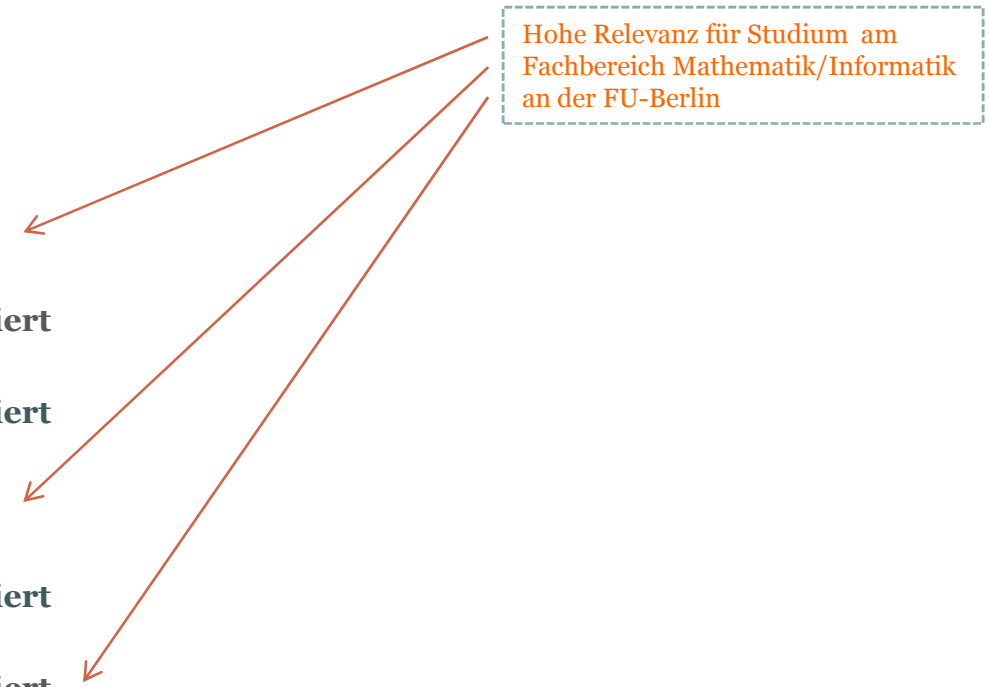
Programmierparadigma



Programmiersprachen

Klassifizierung von Programmiersprachen

1959	Lisp	**relational**
1965	Basic	**prozedural**
1971	Pascal	**prozedural**
1972	C	**prozedural**
1975	Prolog	**relational**
1985	C++	**objektorientiert**
1987	Perl	**objektorientiert**
1990	Haskell	**funktional**
1995	Delphi	**objektorientiert**
1995	Java	**objektorientiert**
2000	C#	**objektorientiert**



Funktionale Programmierung mit Haskell

Oft sehen Programme in Haskell sehr elegant aus und dienen der Spezifikation von höheren Programmiersprachen.
In Haskell wird die Unterscheidung von Datentypen geschult.

Als Beispiel sehen wir die member-Funktion

```
member :: Int -> [Int] -> Bool
member b   []  = False
member b (a:x) = (a==b) || member b x
```

und die Fakultäts-Funktion

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

Der QuickSort-Algorithmus läßt sich sehr kurz formulieren:

```
qSort[]     = []
qSort (a:x) = qSort(y|y<-x,y<=a) ++ [a] ++ qSort(y|y<-x,y>a)
```

Spezifikation einer Funktion

Mathematik: Spezifikation einer Funktion

undefiniert für x<0

$$f(x) = \begin{cases} 1 & \text{, für } x=0 \\ x * f(x-1) & \text{, für } x>0 \end{cases}$$

Informatik: funktionales Programm

```
f 0 = 1
f x = x*f(x-1)
```

Informatik: imperativer Algorithmus und imperatives Programm

setze y=1;
solange x>1, wiederhole:
    setze y auf x*y
    und x auf x-1

```
y = 1;
while (x>1) {
   y = x * y;
   x = x - 1;
}
```

Funktionale versus Imperative Programmierung I

Funktionale Programme:
Definition von Ausdrücken *(expressions)*

Berechnung:
Auswertung von Ausdrücken (liefern Wert)

Variablen:
Namen für Ausdrücke oder Werte

```
f 0 = 1
f x = x*f(x-1)
```

Imperative Programme:
Anweisungsfolgen *(Befehle, statements)*
a1, a2, a3, ...
imperare (lat.) = befehlen

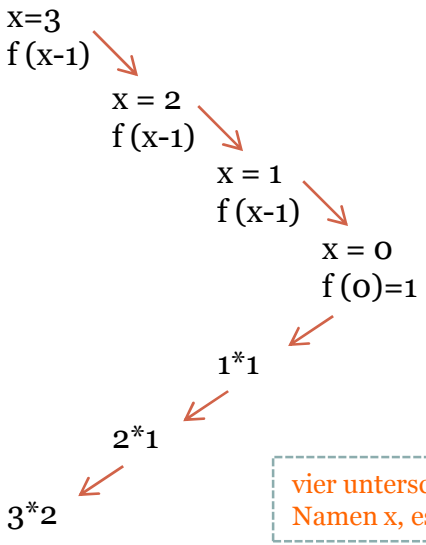
Berechnung/Programmausführung:
Ausführung der Anweisungen, bewirkt Effekte durch Änderung von Zuständen von Variablen (u.a.)

Variablen:
Namen von Behältern für Werte (d.h. Speicherzellen), enthalten einen Wert, der durch einen anderen Wert ersetzt werden kann.

```
y = 1;
while (x>1) {
   y = x * y;
   x = x - 1;
}
```

Funktionale versus Imperative Programmierung II

```
f 0 = 1
f x = x*f(x-1)
```



vier unterschiedliche Werte mit dem Namen x, es wird kein y benötigt

```
f 3 hat den Wert 6 (und f 2 = 2 )
```

```
y = 1;
while (x>1) {
   y = x * y;
   x = x - 1;
}
```

x=3

```
y = 1;
while (x>1) {
   y = x * y;
   x = x - 1;
}
```

X	1
Y	6

systematische Veränderung von zwei Speicherzellen

Logische Programmierung mit Prolog I

Prolog ist die „Sprache der KI“. Wir können Regeln definieren und automatisch schlussfolgern. Prolog ist eine sehr mächtige Sprache, da eine automatische, optimierte Suche hinter den Kulissen arbeitet. Es folgen ein paar Beispiele.

Ist X Element einer Liste L?

```
?member (c, [a,b,c]).
yes

?member (X, [a,b]).
X=a;
X=b;
```

Definition des member-Prädikats:

```
member(X, [X|T]).
member(X, [Y|T]) :- member (X, T).
```

Logische Programmierung mit Prolog II

Die Konkatenation von Listen lässt sich elegant formulieren

```
concat ([], L, L).
concat ([X|L1], L2, [X|L3]) :- concat(L1, L2, L3).
```

Mit Hilfe des concat-Prädikats, können wir member ausdrücken

```
member(X, L) :- concat(_, [X|_], L).
```

Wir können Elemente von Listen entfernen und neue Listen beschreiben

```
del(X, [X|Tail], Tail).
del(X, [Y|Tail], [Y|Tail2]) :- del(X, Tail, Tail2).
```

Mit del können wir sogar insert ausdrücken

```
insert(X, List, List_with_X) :- del(X, List_with_X, List).
```

Oder kurz mal alle Permutationen

```
perm([],[]).
perm([X|L],P) :- perm(L, L1), insert(X,L1,P).
```

Programmiersprachen

Motivation zu Java

- + kleiner Befehlsumfang
- + strikte Typsicherheit
- + plattformunabhängig (Windows, Linux, ...)
- + Applets
- + Einsatz in mobilen Geräten

- - Geschwindigkeit
- - Speicher wird nicht direkt angesprochen (JVM)

Spezifikation und Verifikation imperativer Programme

Inhalt:

- Zustand, Zustandraum, Effekt
- Hoare-Kalkül, partielle Korrektheit, Termination
- Pre- und Postcondition

$$\{P\}\ S\ \{Q\}$$

Gries D., Gries P.: "*Multimedia Introduction to Programming Using Java*", Springer-Verlag 2005

Semantik von Anweisungen

Die Menge der Variablen eines Programms bestimmt den Zustandsraum eines Programms.

Eine bestimmte Belegung der Variablen definiert einen Zustand.

Die Ausführung einer Anweisung bewirkt als Effekt einen geänderten Zustand , d.h. manche Variablen erhalten neue Werte.

Es gibt verschiedene Möglichkeiten für die Festlegung der Semantik einer Anweisung:

operationell, d.h. durch Beschreibung der ausgeführten Aktionen, meist umgangssprachlich,
z.B.: i++; // erhöht i um 1

axiomatisch, d.h. durch Aussagen/Prädikate über die Zustände vor bzw. nach der Ausführung,
z.B.: { P^{i+1} } i++; { P };

denotationell, d.h. durch Funktion : $Zustand^{i} \rightarrow Zustand_{W(i)+1}$,
z.B.: [[i++;]] : w -> w|

Hoare Kalkül I

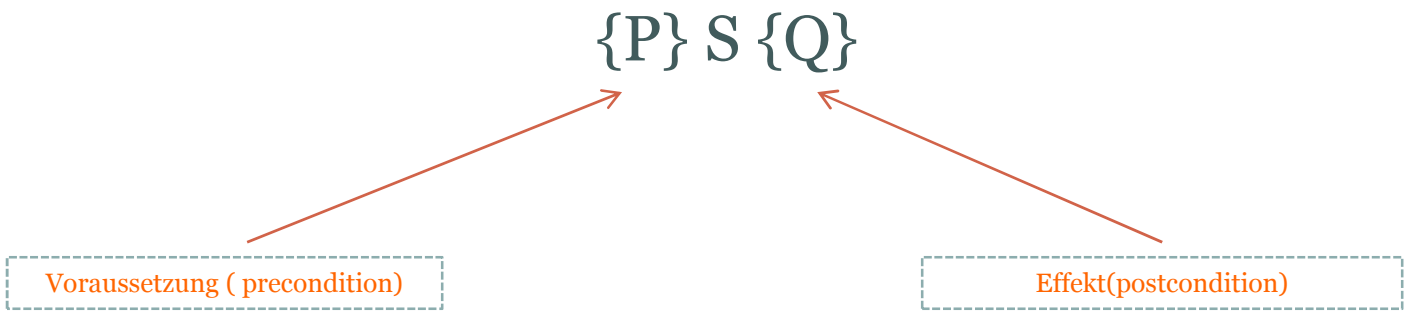
Charackterisierung des Programmzustands vor und nach der Ausführung der Anweisung S charackterisiert Effekt von S

$$\{P\}\ S\ \{Q\}$$

Vor Ausführung erfüllt Programmzustand Prädikat P, nach Ausführung Prädikat Q.

Dieses Tripel wird auch Hoare-Tripel nach Tony Hoare genannt, der diese Notation 1969 eingeführt hat.

Hoare Kalkül II



(P, Q) heißt Zusicherung (assertion, eigentlich Behauptung).

Wenn P(z) vor Ausführung von S, dann Q(z‘), mit z‘ ist Zustand nach Ausführung von S.

Beispiel:

```
{x=0}
x=x+1;
{x>0}
```

Pre- und Postcondition werden in geschweiften Klammern vor und nach den Answeisungen geschrieben. Hier werden fett geschriebene Klammern (rot) verwendet, da in Java ebenfalls geschweifte Klammern zur Syntax gehören.

Hoare Kalkül III

Wir verstehen das folgende Hoare-Tripel:

```
{x=0}
x=x+1;
{x>0}
```

so:

„Angenommen, die Zusicherung x=0 ist wahr und die Anweisung x=x+1; wurde ausgeführt, wenn die Ausführung dann terminiert, wird die Zusicherung x>0 wahr sein."

Ein Hoare-Tripel selbst kann wahr oder falsch sein:

`{x=0}x=x+1;{x>0}`	true
`{x=0}x=x-1;{x=0}`	false
`{true}if(x>y) swap x and y; {x<=y}`	true
`{false} x=1; {x=0}`	true

Hoare Kalkül IV

Weitere Beispiele:

```
{x>0} while(x>0) x=x+1;{x=0}
```

false

```
{true} x=1;{false}
```

false

```
{true} while(Math.random()!=.5) {false}
```

false

Im letzten Beispiel ist nicht einmal die Terminierung gerantiert.

Spezifikation mit dem Hoare Kalkül

Wir können das Hoare-Tripel verwenden, um unsere Programme zu spezifizieren.

Beispiel 1) Spezifikation einer Anweisung S: Gegeben sei ein y>0, speichere x^y in z.

```
{y>0} S {z=x^y}
```

Beispiel 2) Spezifikation einer Anweisung S: Kopiere das Maximum von x und y in z.

```
{true} S {z=max(x,y)}
```

Beispiel 3) Spezifikation eines Programms S: Sortiert das Array b aufsteigend.

```
{true} S {b ist aufsteigend geordnet}
```

Inferenz Regeln

Angenommen, das folgende Hoare-Tripel ist wahr:

```
{x>=0} S {y=x und z=x+1}
```

Daraus können wir folgern, dass S im Initalzustand x=0 und am Ende z=x+1 ist:

```
{x=0} S {y=x und z=x+1}
```

```
{x>=0} S {z=x+1}
```

Wir können die Precondition mit etwas ersetzen, das daraus folgt und ebenso die Postcondition mit etwas, das diese schlussfolgert (Implikation).

```
{x=0}
{x>=0}
S
{y=x und z=x+1}
{z=x+1}
```

stärkere Vorbedingung

schwächere Vorbedingung

Implikation

Die logische Implikation ist wie folgt definiert:

B1	B2	B1 => B2
0	0	1
0	1	1
1	0	0
1	1	1

Stark/Schwach

P ist stärker als Q und Q ist schwächer als P, wenn P=>Q immer wahr ist.

Eine Precondition-Regel verstärken:
Vorausgesetzt P=>Q, {Q} S {R} können wir schlussfolgern: {P} S {R}

Eine Postcondition-Regel abschwächen:
Vorausgesetzt R=>T, {Q} S {R} können wir schlussfolgern: {Q} S {T}

Axiomatische Definition von Anweisungen I

Einführung in die Notation:

```
[v\e] S
```

Bedeutung: Jedes Auftreten der Variable v wird durch e ersetzt.

Beispiel 1:

```
[v\v+1] (v>=w)
```

wird zu

```
(v+1>=w)
```

Beispiel 2:

```
[v\x+y] (x*v>=v)
```

wird zu

```
x*(x+y)>=(x+y)
```

Axiomatische Definition von Anweisungen II

Diese Notation läßt sich erweitern

```
[v,w\e,f] S
```

Bedeutung: Jedes Auftreten der Variable v wird durch e und w durch f ersetzt.

Beispiel 1:

```
[v,w\w,v] (v>=w)
```

wird zu

```
(w>=v)
```

Beispiel 2:

```
[x,n\x+b[n],n+1] (x ist Summe von b[0..n-1])
```

wird zu

```
x+b[n] ist Summe von b[0..n+1-1]
```

Leere Anweisungen (empty statement)

Der leere Anweisungsblock {} tut nichts. In anderen Sprachen bezeichnet man die leere Anweisung als skip.

```
{R} skip {R}
```

gilt für alle Bedingungen

Zuweisungen (assignment statement) I

Ist wie folgt definiert:

```
{e ist wohl definiert und [v\e] R} v=e; {R}
```

So kann man vorraussetzen, dass eine Zuweisung mit R true liefert.

Zuweisungen (assignment statement) II

Es folgen ein paar Beispiele:

```
{0=0} x=0; {x=0}
```

```
{x+1=0} x=x+1; {x=0}
```

```
{x+ n+1 = sum of 1..n} x=x + n+1; {x=sum of 1..n}
```

```
{x = sum of 1..n-1} n=n-1; {x=sum of 1..n}
```

Mehrere Zuweisungen (multiple assignment statement)

In Java ist dieses Konzept nicht erlaubt.

```
{e ist wohl definiert und [v,w\e,f] R} v,w=e,f; {R}
```

Sequenzregel

```
{Q} S1; S2 {R}
```

S1 und S2 sind Anweisungen, angenommen wir finden eine Bedingung P, die folgendes erfüllt:

```
{Q} S1 {P}
```

und

```
{P} S2 {R}
```

dann können wir daraus schließen, dass:

```
{Q} S1; S2 {R}
```

Beispiel für die Sequenzregel I

Es folgt Beispiel, in dem wir Schritt für Schritt die Bedinungen aufstellen wollen

```
t=x; x=y; y=t;
```

wir können eine einfache Postcondition angeben

```
R: x=X und y=Y
```

wobei X und Y Namen für virtuelle Konstanten sind. Fangen wir mit der letzten Anweisung an

```
{P1: x=X und t=Y}
y=t;
{R: x=X und y=Y}
```

Jetzt verwenden wir P1 als Postcondition für die zweite Anweisung und berechnen die entsprechende Precondition P2. Schließlich wird P2 als Postcondition für die erste Anweisung verwendet:

```
{Q: y=X und x=Y}
t=x;
{P2: y=X und t=Y}
x=y;
{P1: x=X und t=Y}
y=t;
{R: x=X und y=Y}
```

Beispiel für die Sequenzregel II

Jetzt können wir die Sequenzregel verwenden, um P1 und P2 zu eliminieren

```
{Q: y=X und x=Y}
t=x;
x=y;
y=t;
{R: x=X und y=Y}
```

Das Hoare-Tripel erzählt uns, dass die Anweisungsfolge die Werte von x und y tauscht.

Bedingungen (conditional statement)

Eine Bedingung wie z.B. if mit Pre- und Postcondition. Unter der Bedingung, dass Q und B => R, {Q und B} S {R} schlussfolgern wir

```
{Q} if (B) S {R}
```

Das gleiche für die if-else-Anweisung. Unter den Bedingung, dass

```
{Q und B} S1 {R}
```

und

```
{Q und !B} S2 {R}
```

schlussfolgern wir

```
{Q} if (B) S1 else S2 {R}
```

While-Schleife I

Wir definieren die while-Schleife

```
{Q} while (B) S {R}
```

Wir müssen dafür ein paar Voraussetzungen (Prämissen) entwickeln. Das führt uns zu dem Begriff Invariante.

```
{Q}
{P}
while (B)
   {P&&B} S {P}
{P&&!B}
{R}
```

Folgende Bedingungen müssen dafür erfüllt sein:

1. Q=>P
2. {P&&B} S {P}
3. P&&!B => R
4. Die Schleife terminiert

While-Schleife II

Wie kann man zeigen, dass eine Schleife terminiert?

Die einzige Möglichkeit, dass die Schleife nicht terminiert ist, dass die Bedingung (B) niemals falsch wird. Eine Schranke (bound function) t für eine While-Schleife ist ein Integerausdruck mit den folgenden beiden Eigenschaften:

1) In jeder Iteration wird t dekrementiert.

```
{P&&B}
tsave=t; S
{t<tsave}
```

2) Wenn eine andere Iteration ausgeführt wird, gilt t>0
   P&&B => t>0

Damit ist t eine obere Schranke für die Anzahl der Iterationen. Damit können wir die Terminierung einer While-Schleife formulieren!

Eine Schleife terminiert, wenn eine Schranken-Funktion t für diese Schleife existiert.

While-Schleife III

Im weiteren werden wir folgende Notation für eine While-Schleife wählen:

```
{Q}
// invariant:        P:
// bound function : t
while (B)
   S
{R}
```

Entwicklung einfacher Programme I

Beispiel: **Finden des Maximums**

```
{true} S {R: z=max(x, y)}
```

Da wir wissen, was mit„Maximum„ gemeint ist, können wir die Postcondition umschreiben zu

```
{R: (z=x && x>=y} || (z=y && y>=x)
```

Das führt uns zu if-else

```
if (x>=y) z=x;
else z=y;
```

Wir sehen an diesem Beispiel, dass die Postcondition zu Beginn wichtiger ist, als die Precondition.

Entwicklung einfacher Programme II

Beispiel: **Schleifen**

Wir erinnern uns an die Definition

```
{Q}
// invariant:        P:
// bound function : t
while (B)
   S
{R}
```

Wir müssen vier Fragen beantworten:

1) Unter welcher Voraussetzung startet die Schleife? (Q)
2) Wann terminiert die Schleife? (P&&!B=>R)
3) Wie machen wir Fortschritte? (Dekrementierung von t)
4) Welche Invariante müssen wählen? ({P&&B} S {P})

Finden der Invariante

Dazu schauen wir uns nochmal die vollständige Schreibweise an

```
{Q}
// invariant:        P:
// bound function : t
while (B)
   {P&&B} S {P}
{P&&!B}
{R}
```

Oft ist es hilfreich R zu verallgemeinern.

Beispiel 1:

Einführung in Threads

Gastdozent: Prof. Dr. Margaritha Esponda

Inhalt:

- Motivation und Konzepte
- Verwendung von Threads in Java

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007
Gries D., Gries P.: "*Multimedia Introduction to Programming Using Java*“, Springer-Verlag 2005
Abts D.: „*Grundkurs JAVA: Von den Grundlagen bis zu Datenbank- und Netzanwendungen*“, Vieweg-Verlag 2007

Einführung in Threads