2 Einfache imperative Algorithmen in Java

Am Beispiel von 2807 ´ 7 im Papyrus Rhind, 16. Jh. v. Chr.

- **Das Ägyptische Multiplizieren**

Auch bekannt als „Methode der russischen Bauern“

Gegeben zwei natürliche Zahlen a, b > 0

Halbiere b, verdopple a mit dem Ergebnis (a‘,b‘) solange, bis b‘ = 1

Summiere alle a‘, für die b‘ ungerade

Ergebnis: a x b

a=17 b=11

17	11
34	5
68	2
136	1

a x b = 136+34+17 = 187

Wenn ihr nur duplieren und halbieren könnet, so könnet ihr das übrige ohne das Eins mal Eins multipliciren.
Christian von Wolff, 1679- 1754
(Philosoph und Mathematiker, Univ. Halle)

8	9
16	4
32	2
64	1

Spezifikation als Funktion...

```
                a,  wenn b = 1
f (a,b) =       a + f(2 * a , (b-1)/2),        wenn b ungerade
                 f(2 * a ,  b/2), sonst
```

oder in Haskell

```
f:: Int -> Int -> Int
f a 1 = a
f a b = if (odd b) then a + f (2*a) (b `div` 2)
        else f (2*a) (b `div` 2)
```

und die Implementierung in Java

Eine einfache rekursive Methode in Java

```
static int f( int a, int b)
{
  if (b == 1) {System.out.println(a); return a;}
  if (b%2 == 1) return a + f (2*a, b/2);
     else return f (2*a, b/2);
}
```

Das komplette Programm (rekursiv)

```
// Aegyptische Multiplikation funktional
// HS, 3-99

public class Aegyptisch
{
  public static int f (int a, int b)
  {
        if (b==1) return  a;
        if ( b%2 == 1) return a +  f(2*a,b/2);
                       else return f(2*a,b/2);
  }

  public static  void main ( String [ ] args ) {
    if (args.length <= 1)
       {System.out.println
          "Nicht genuegend Argumente: 2 nat. Zahlen!"); return;}
        System.out.println ("Produkt: " +
          f(Integer.parseInt(args[0]), Integer.parseInt(args[1]))
        );

    }
}
```

kein großer Unterschied zu funktionaler Implementierung?

Return?

Erläuterung zum Programm

beliebter Anfängerfehler

- Anweisungen enden mit Semikolon **; immer!**
- Fallunterscheidung beginnt mit **`if`**, KEIN `then`
- Gleichheitsvergleich mit **==**
- Bedingung steht *immer* in Klammern: **(b==1)**
- Typ der Argumente Teil der formalen Parameter
- Werte explizit mit **`return`** zurückgeben: Fallunterscheidung ist Anweisung, kein Ausdruck!

aber Semikolon als Abschluß der Anweisung

Fallunterscheidung:

`if (``<Bedingung>`**`)`** `<Anweisung>` [**`else`** `<Anweisung>`]

Metasymbole für optionale Teile

Das Hauptprogramm

Der heisst in bester C-Tradition **`args`** (Konvention)

- Hauptprogramm heißt immer `main`
- hat immer eine Zeichenkette (Typ: `String []`) als formalen Parameter
- Die Zeichenkette kann leer sein oder enthält Kommandozeilenargumente
- ... die mit Methoden, die Zeichenketten interpretieren (z.B. ganze Zahlen erkennen („parsen“))
- Ausgabe mit `System.out.print` oder `System.out.println` (mit Zeilenwechsel)
- Mehrere AusgabeZeichenketten mit Verkettungsoperator + verbinden, Typmix (z.B. mit `Integer` o.ä.) möglich
- Jede Klasse kann eine Methode `main` enthalten; `main` der Klasse, die auf der Kommandozeileaufgerufen wird, ist das Hauptprogramm

`nicht ++ wie in Haskell`

Kommentare

Kommentare:

```
/* Dies ist ein
   zweizeiliger Kommentar */

// Ein C++-kompatibler Kommentar
// „//“ wirkt für den Rest der Zeile
// genau wie „--“ in Haskell

/** leitet auch einen mehrzeiligen
 * Kommentar ein. Das Dokumentationswerkzeug
 * javadoc generiert aus solchen Kommentaren
 * eine Dokumentation
 */
```

`* ist kein Kommentar-begren-zer, aber empfehlenswert`

Sorgfältige Kommentierung zeichnet den guten Programmierer aus!

Iteration statt Rekursion

Funktional?

```
f:: Int -> Int -> Int
f a b = f´ 0 a b
  where f´ akk a 1 = akk + a
        f´ akk a b =
           if (odd b) then   f´ (akk+x) (2*a) (b `div` 2)
           else              f´akk  (2*a) (b `div` 2)
```

Endrekursion = Iteration ?!

Zwischenergebnisse in Akkumulator speichern.

Heisser Kandidat für eine **Zustandsvariable!**

Rekursive Aufrufe, die ohne weiteres den Wert berechnen, also nicht hierher zurückkehren (lat: recurrere)

```
akk = 0;
solange (b > 1)  tue folgendes
  { wenn b ungerade akk=akk+a;
    verdopple a;
    halbiere b-1;}
  }
liefere den Wert von (akk +a) als Ergebnis
```

etwas ``verbose´´

...und in Java

```
public static int fIterativ(int a, int b)
     { int akk = 0;
       while (b > 1)
         { if (b%2 != 0) akk=akk+a;
           a=a*2;
           b=b/2;  // hier wird nur der
                   //ganzzahlige Anteil verwendet
         }
       return akk+a;
     }
```

Schleifenkörper

`while`-Schleife: wiederhole Folgeanweisung solange wie Bedingung gilt

Formale Parameter können **wie lokale Variablen** verwendet werden.

Copy-Semantik der formalen Parameter:

```
public int myMethod (Int x)
{ int xLokal = x;
  // Testlicher Code: x durch xLokal ersetzt
   ...
}
```

Diese Anweisung steht nicht im Quellcode, sondern wird vom Compiler erzeugt

wieso ist das wichtig?

.. und nochmal

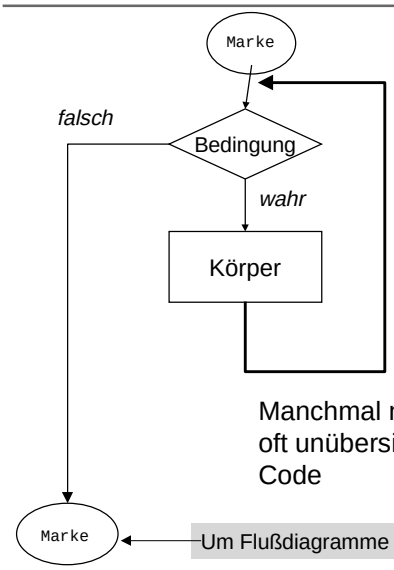
```
public static int fIterativ(int a, int b)
     { int akk = 0;
       while (b > 0)
         { if (b%2 == 0) b=b/2;
             else {b=(b-1)/2; akk=akk+a;}
           a=a*2;
         }
       return akk;
     }
```

leistet die Methode dasselbe?

- Sind die Transformationen `(b>0)` statt `(b>1)` usw. korrekt?
- Kann man das beweisen?
- Ist die Methode überhaupt korrekt implementiert?
- Multipliziert sie überhaupt richtig??

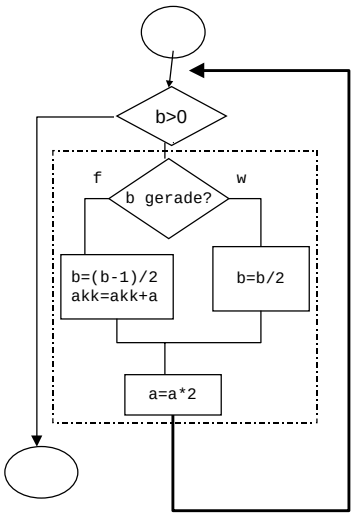
`Spezifikation??`

Schleifen




```
while <Bedingung>
   {Körper}
```

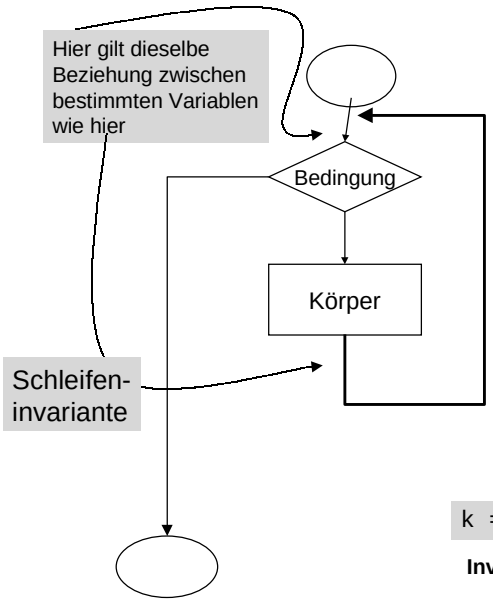
Ägytisches Multipl.



Manchmal nützlich, aber oft unübersichtlicher als Code

Um Flußdiagramme „zu verkleben“

Invarianten



Eine Beziehung zwischen Variablen, die sich in einer Anweisungfolge nicht ändert, heißt **Invariante** (bzgl. der Anweisungen)

wichtig für Korrektheitsbeweise

```
if (b%2 == 0) b=b/2;
 else {b=(b-1)/2;
       akk=akk+a;}
a=a*2;
```

`k == a*b + akk`

Invariante: `a*b+akk` ist konstant!

Korrektheit mittels Invarianten

```
public static int fIterativ(int a, int b)
      { int akk = 0, k = a*b +akk;
        while (b > 0)
          { if (b%2 == 0) b=b/2;
              else {b=(b-1)/2; akk=akk+a;}
            a=a*2;
          }
        return akk;
      }
```

k==a*b+akk

Hier gilt: k = a*b

k==a*b+akk

Hier gilt die Invariante (schon bewiesen)

k==a*b+akk && b=0

und hier wegen a*b==0 :
k ==akk
also: a*b == k == akk

Bisher gezeigt: Wenn Programm **`return`** erreicht, dann hat **`akk`** den Wert **`a*b`**. Reicht das?

Endlosschleifen

- Wenn die *Bedingung* einer Schleife sich nicht durch Anweisungen im Körper verändert, terminiert sie nicht (Endlosschleife)

Aber selbst wenn: entscheidend ist, dass Bedingung irgendwann falsch wird.

```
While (true);
```

Konstante **true** wird nie **false**

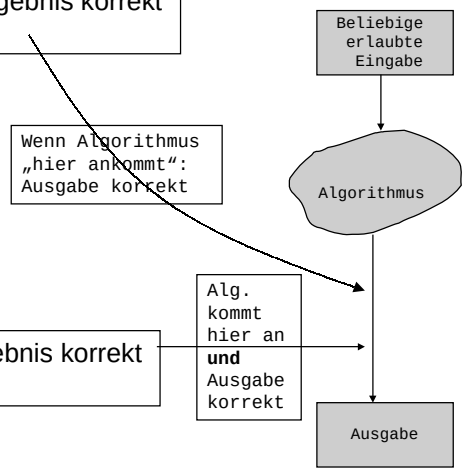
Eine leere Anweisung gibt es auch!

```
x = -1;
while (x!=0)
{ x = (x-1)/2;
System.out.println(x);
}
```

Immer noch einfach zu erkennen, aber allgemein sogar **unentscheidbar**, ob beliebige Schleife terminiert!

Partielle und totale Korrektheit

Algorithmus *partiell* korrekt, wenn Ergebnis korrekt
- vorausgesetzt er terminiert -



Wenn Algorithmus „hier ankommt“: Ausgabe korrekt

Alg. kommt hier an **und** Ausgabe korrekt

Algorithmus *total* korrekt, wenn Ergebnis korrekt *und* der Algorithmus *terminiert*.

Termination der Ägyptischen Multiplikation

Behauptung Der Algorithmus hält für alle nat. Zahlen als Eingabe.

Argument :

Fortgesetztes “ganzzahliges” Halbieren von b führt schließlich auf b <= 0.

Wieso ?
Zeige dazu

(1) Für alle geraden Zahlen b >= 2 (!) gilt b/ 2 < b.

Beweis: 1 < 2

==> 1*b < 2*b (da b > 0)

==> b/ 2 < b

(2) Für alle ungeraden Zahlen b >= 1 gilt (b- 1)/ 2 < b.

-1 < 0

==> b - 1 < b

==> (b - 1)/ 2 < b (da b - 1 >= 0)

In jedem Schleifen-durchlauf kommen wir der Bedingung: b<= 0 eins näher!

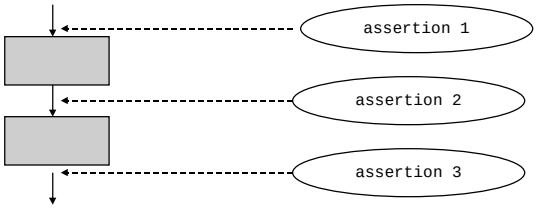
Streng monoton fallende Folge *natürlicher* Zahlen ist endlich,

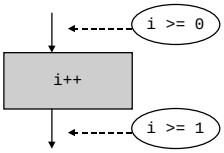
==> Terminierung!

Zusicherungen

Zusicherung (assertion): Aussage (wahr) über den Programmzustand;
Technisch: prädikatenlogische Formel,
die Beziehung zwischen Programmvariablen ausdrückt .

Idee eines Korrektheitsbeweises:
Zusicherungen zwischen je zwei Anweisungen des Programms einstreuen und schrittweise zeigen, daß Zusicherung nach Anweisung aus Zusicherung vor Anweisung folgt, wenn Anweisung ausgeführt wird.





Zusicherungen und Laufzeittests

Korrektheits*beweis* mit Zusicherungen erfordert logische Folgerung zwischen Zusicherungen auf Basis der Effekte von Anweisungen.

Aber Zusicherungen lassen sich auch zur Laufzeit überprüfen:

```
public static int fIterativ(int a, int b)
     { int akk = 0,        k = a*b +akk;
       assert(k == a*b + akk);
        while (b > 0)
         { if (b%2 == 0) b=b/2;
                else {b=(b-1)/2; akk=akk+a;}
           a=a*2;
              assert(k == a*b + akk);
         }
       assert(k == a*b + akk && y == 0);
       return akk;
     }
```

```
static void assert(boolean t)
{ if (!t)
  {System.out.println (...);
   ...
   System.exit(1);
  }
}
```

Wenn das Programm ohne Fehlermeldung terminiert, heißt das noch nicht, dass es korrekt ist! Warum?

Nochmal Schleifen...

Schleifen wie `while (B) {}` gehören zu den **Steuerkonstrukten** imperativer Programmiersprachen.

Steuerkonstrukte werden meist als **Kontrollstrukturen** (vom englischen **control structures**) bezeichnet.

normalerweise:
nach Befehl i
kommt der lexikalisch
folgende i+1

Wichtigste Aufgabe: **Reihenfolge der Befehlsausführung** verändern.
„Steuerung der Befehlsausführung“

Bisher:
Fallunterscheidung `if ... else ..`, `while-Schleife`

while rekursiv ?!

```
while (b) s;
```
ist äquivalent zu:

```
solange (b);
```

mit der Definition:

```
static void solange(boolean b);
{if b then {s; solange(b);}
}
```

z.B.
```
static void solange(b);
{if b then {x=(x-1)/2;
solange(b);}
}
```

Aufruf:
```
x = 20;
solange(x!=0);
```

Achtung: ist sichergestellt, dass `x=(x-1)/2`
x tatsächlich verändert?
Ist im allgemeinen **nicht** so!
Gilt aber sicher, wenn
`static int x`
ausserhalb `solange` definiert ist.

Dazu später mehr.

for-Schleife

```
for (init; boolexpression; incr)
   anweisung
folgeanweisung;
```

Anweisung kann natürlich Block {...} sein.

äquivalent zu:

mit einer Ausnahme....

```
init;
while (boolexpression)
  {anweisung
   incr;
  }
folgeanweisung;
```

for-Schleife

Beispiel:

```
int s;
for (i=0; i <= 100; i++)
   if (i%3==0) s=s+i*i;
System.out.println(s);
```

oder einfacher:

```
int s;
for (i=3; i <= 100; i=i+3)
     s=s+i*i;
System.out.println(s);
```

```
int s;
for (i=0; i <= 100; i++)
   if (i%3!=0) continue;
      else s=s+i*i;
System.out.println(s);
```

continue: beende den Schleifendurchgang und beginne den nächsten.
für `while`-Äquivalenz kein `continue` erlaubt

continue gehört zu den Steueranweisungen; auch in `while` - Schleife verwendbar, ist ein **Sprungbefehl**

Hier ist das Ende der Schleife

Die offizielle Syntax (fast)

forStatement:
for (ForInit$_{opt}$; Expression$_{opt}$; ForUpdate$_{opt}$)
Statement

Optionale Bestandteile

ForInit:
StatementExpressionList
LocalVariableDeclaration

Alternativen

ForUpdate:
StatementExpressionList

StatementExpressionList:
StatementExpression
StatementExpressionList , StatementExpression

Verlassen von Schleifen

`continue` verläßt den aktuellen Schleifendurchlauf.
Manchmal nützlich, die gesamte Schleife zu verlassen - das leistet `break` .

```
for (i=0; i <= 100000; i++)
     if (matrikelNr(i) = gesuchteNummer)
         {System.out.println(...); break;}
naechsteAnweisung();
```

Verlässt die Schleife unbedingt.

```
nextRun: for (i=0; i <= 1000 ; i++)
           for (j=0; j<=1000;  j++)
            {if ((j+i == ...) then continue nextRun;
             diesUndDas;
            }
naechsteAnweisung();
```


Sprungbefehle .....

.... sind oft - nicht immer - von Übel!
Spaghetti-Code ist schwer lesbar und fast immer falsch.

Deshalb *keine unbedingten Sprünge* der Art
Goto <label>
in Java erlaubt.

Und was ist mit **`break <label>`** ??

`break <label>` und `continue <label>` nur aus Schleifen heraus möglich!
Mit einer Ausnahme für `break` ....

`switch` , dazu gleich

break und geschachtelte Schleifen

```
public class NoPrimest
{
    public static void main(String[] a)
    {int last = 20;
    for( int i=2; i<=last; i++)
         { int j = 2;
           Test:
           while (j*j <=i)
           { if (i%j == 0)
                {System.out.println(i + " j: " +j);
                 break Test;
                }
             else j++;
           }
         }
    }
}
```

Da geht's weiter

```
Ergebnis:
4 j: 2
6 j: 2
8 j: 2
9 j: 3
10 j: 2
12 j: 2
14 j: 2
15 j: 3
16 j: 2
18 j: 2
20 j: 2
```

break label
verläßt Schleife und alle umfassenden Anweisungen bis zu (einschließlich!) der mit `label` benannten. Die muß die unterbrochene umfassen.

break; Performance von Schleifen

```
....
     Test: System.out.println ("Jetzt komme j: "+j);
     while (j*j <=i)
     { if (i%j == 0)
          {System.out.println(i + " j: " +j);
           break Test;
          }
       else j++;
     }
   }
...
```

Syntaxfehler: keine umfassende Anweisung mit breaklabel `Test`

Kann man die Berechnung der Bedingung vereinfachen?

Beachte: Keine Konstanten in Schleifen berechnen!
Schleifenbedingung einfach halten!

Programmierstil

```
int n = ...;
....
for (i=0; i <= n; i++)
  {.....;
   n = ...;
}
naechsteAnweisung();
```

Veränderung der Abbruchbedingung innerhalb der Schleife ist **sehr** schlechter Programmierstil!

```
for (double x=0.0; x >=1.0; x+=0.1)
   anweisung;
naechsteAnweisung();
```

Syntaktisch korrekt, aber fehleranfällig

Programmierstil

for nur für ganzzahlige Laufvariablen,

sonst besser **while (B) {...}**

oder **do {...} while (B);**

```
for (; b!=EOF ;)
  {b = System...read(..);
   if b == EOF break;}
```

schlecht!

```
b = System....read(...);
while (b!=EOF)
  {b = System...read(..);}
```

besser

```
do
 {b = System...read(..);}
  while (b!=EOF) ;
```

so soll es sein

Geschachtelte Schleifen

Eulersche Zahl:
e = 1 + 1/1! + 1/2! + 1/3! +1/4! + 1/5! + ...

```
public class Euler1
{
    public static void main(String[] a)
    { double sum = 1.0;
      for (int i=1; i<= 40; i++)
       {sum = sum + 1.0/fak(i);
        System.out.println (i + " " +sum);
       }
    }

    static int fak(int n)
    { int k = 1;
      for (int i=1; i<=n; i++) k=k*i;
      return k;
    }
}
```

Gleitkommazahl, doppelte Genauigkeit

```
1 2.0
2 2.5
3 2.6666666666666665
4 2.708333333333333
5 2.7166666666666663
6 2.7180555555555554
7 2.7182539682539684
8 2.71827876984127
9 2.7182815255731922
10 2.7182818011463845
11 2.718281826198493
12 2.7182818282861687
13 2.718281828803753
....
30 2.7182818
31 2.7182818
32 2.7182818
33 2.7182818
34 Inf
35 Inf
36 Inf ...
```

Oh !?

fak(i) ??

2	2	19	109641728
3	6	20	-2102132736
4	24	21	-1195114496
5	120	22	-522715136
6	720	23	862453760
7	5040	24	-775946240
8	40320	25	2076180480
9	362880	26	-1853882368
10	3628800	27	1484783616
11	39916800	28	-1375731712
12	479001600	29	-1241513984
13	1932053504	30	1409286144
14	1278945280	31	738197504
15	2004310016	32	-2147483648
16	2004189184	33	-2147483648
17	-288522240	34	0
18	-898433024	35	0
			...

Hier ist etwas falsch, oder?

Leichte Verbesserung:

```
public class Euler2
{
    public static void main(String[] a)
    { double sum = 1.0, term;
      for (int i=1; i<= 15; i++)
       {term = 1.0;
        for (int j; j<= i; j++) term = term*j;
         sum = sum + 1.0/term;
         System.out.println (i + " " +sum);
       }
    }
}
```

Wie oft wird `term=term*j` ausgeführt?

Inline-Berechung schneller als Methodenaufruf.
Darf nicht zu Lasten der Lesbarkeit des Programms gehen!!

Alte Idee: Zwischenergebnisse ausnutzen

```
public class Euler3
{
    public static void main(String[] a)
    { double sum = 1.0, term=1.0;
      for (int i=1; i<= 15; i++)
       {term = term / i;
        sum = sum +  term;
        System.out.println (i + " " + sum +
                            " " +(java.lang.Math.E - sum));
       }
}
```

Nur **eine** Schleife

```
1 2.0  0.7182818284590451
2 2.5  0.2182818284590451
3 2.6666666666666665  0.05161516179237857
4 2.708333333333333   0.009948495125712054
5 2.7166666666666663  0.0016151617923787498
....
15 2.718281828458995  5.0182080713057076E14
16 2.718281828459043  2.220446049250313E-15
17 2.7182818284590455  -4.440892098500626E-16
18 2.7182818284590455  -4.440892098500626E-16
19 2.7182818284590455  -4.440892098500626E-16
```

Wieviele Operationen? Korrekt?

Warum konstante Differenz?

Probleme mit if ...else ..; switch

Geschachtelte **if**-Anweisungen möglich, aber unschön:

```
if (i%10 == 1) if (i > 100) i++;
else if (i%10==2 || i%10 == 4) i--;
     else i = i*2;
```

Solche Fallunterunter-scheidungen sind Indiz für schlecht entworfenes Programm. Sie sind **fast immer falsch!** Vermutlich auch hier

```
Beispiel:
i==101 --> i==102
i==182 --> i==182
i==183 --> i==183
Wert ???
```

Wer solche Programme schreibt, darf sich nicht über Fehler wundern.

Immer noch unschön, aber konsistent zu Einrückung:

```
if (i%10 == 1) {if (i > 100) i++;}
else if (i%10==2 || i%10 == 4) i--;
     else i = i*2;
```

```
Beispiel:
i==101 --> i==102
i==182 --> i==181
i==183 --> i==366
```


switch

Besser mit **switch** (case-) Anweisung:

break ist hier wichtig, sonst andere Semantik!

```
switch (i%10) {
   case (1) : if (i>100) i++; break;
   case (2): case (4): i--; break;
   default : i=i*2;
}
```

Switch muss **int, char** oder **short, byte** liefern, ... Datentypen in Kürze.

Übersichtlicher Quellcode Voraussetzung für **korrekte Programme!**

switch : Achtung !

```
class Toomany {
        static void howMany(int k) {
                switch (k) {
                case 1:    System.out.print("one ");
                case 2:    System.out.print("too ");
                case 3:    System.out.println("many");
                }
        }

        public static void main(String[] args) {
                howMany(3);
                howMany(2);
                howMany(1);
        }
}
```

Ausgabe:

```
many
too many
one too many
```


Sprünge in Ausnahmesituationen

Ausnahmebehandlung (exception handling)

Bsp: Nochmal ägyptische Multiplikation

wenn a == 0?
oder b == 0?

```
public static int f (int a, int b)
{
      if (b==1) return  a;
      if ( b%2 == 1) return a +  f(2*a,b/2);
                     else return f(2*a,b/2);
}
```

```
e:\Inst\lehre\alp2\progs> java Aegyptisch 23 0
^C
e:\Inst\lehre\alp2\progs> java Aegyptisch 23 0
Exception in thread "main" java.lang.StackOverflowError
```

Ausnahmen

Ausnahmen sind außergewöhnliche Programmereignisse , z.B.:
- fehlerhafte Eingaben
- Laufzeitfehler, die vom System ausgelöst werden (z.B. Division durch 0, Stackoverflow)
- Fehler in der Anwendung

Ausnahmen sind im Gegensatz zu Programmfehlern (bugs) vom Programmierer im Programm berücksichtigt.

```
public static int f (int a, int b)
{     if (b<=0) {System.out.println(....); System.out.exit(1);}
      else
      { if (b==1) return  a;
        if ( b%2 == 1) return a +  f(2*a,b/2);
                       else return f(2*a,b/2);
      }
}
```

unbefriedigend, Fehlerbehandlung und Algorithmus verzahnt.

Ausnahmebehandlung

```
public static int f (int a, int b)
{
      if (b==1) return  a;
      try
      {  if ( b%2 == 1) return a +    f(2*a,b/2);
                      else return f(2*a,b/2);
      }
      catch (StackOverflowError e)
      {System.out.println(„Rekursion terminiert nicht“;
       System.exit(1);
      }
}
```

Könnte man etwas Sinnvolleres tun?

Einfachster Fall:
Ausnahmen werden vom Laufzeitsystem ausgelöst (**geworfen**).
Folge: der durch `try {...}` gesicherte Block wird verlassen und der `catch (<Ausnahmetyp> e)`-Bock ausgeführt.
Auslösen verschiedener Ausnahmen (unterschiedlichen Typs) in einem `try`-Block möglich, auch durch den Programmierer (`throw <exception>`) - später mehr dazu.

Ausnahmebehandlung bei Eingaben

```
public static int readInt()
{     int i = 0;
      BufferedReader inp = ....;
      String s = .....         ;

try {       s = inp.readLine();
            i = Integer.parseInt(s);
 }
 catch(NumberFormatException e)
 {       System.out.println(s + " was no int, assigning '0'");

 }
 catch(IOException e)
 {       System.out.println("readInt: Input error");
         System.exit(1);
 }
return i;
 }
```

IO-Exceptions **müssen** behandelt werden

Keine Ausnahmebehandlung bei arithm. Überlauf

Despite the fact that **overflow, underflow**, or loss of information may occur, evaluation of a multiplication operator * **never throws a run-time exception**. (aus: language-spec)

Also: Arithmetischer Überlauf muss „per Hand“ geprüft werden.

Wie macht man das?
`if (a*b > INT_MAX)..`

keine gute Idee!

Maximalwerte: `-2147483648 <=int 2147483647`

Zyklische Zahldarstellung: Zweierkomplement

Zyklische Behandlung des Überlaufs:

MAX_VALUE +1 = MIN_VALUE

```
e:\Inst\lehre\alp2\progs>java OverflowTest
MAX_VALUE (int) =          2147483647
MAX_VALUE+1  =             -2147483648
MAX_VALUE+1 als long:      -2147483648
MAX_VALUE+1-2 =            2147483646
Maximaler Longwert=        9223372036854775807
Maximaler Longwert +1 =    -9223372036854775808
Maximaler Longwert+1-2 =   9223372036854775806
```


Arithmetischer Überlauf: Pseudozufallszahlen

```
public static int randomInt( )
    {
        state = (A * state) % M;
        return state;
```

Hilft es hier, zu erkennen, dass es einen Überlauf gibt??

Ist das richtig??

beweisen!

```
public static int randomInt( )
    {
     int tmpState = A * ( state % Q ) - R * ( state / Q );
        if( tmpState >= 0 )
            state = tmpState;
        else
           { state = tmpState + M;
             System.out.println("kleiner 0: " + tmpState);
           }
        return state;
    }
```

Kurze Zusammenfassung (Java)

Variablen, Konstanten: typisierte Bezeichner für Speicherzellen.

Java-Syntax für Bezeichner : Groß /Kleinbuchstaben, $, _, Ziffern, erstes Zeichen darf keine Ziffer sein.
Java-Konvention f. Variablen: Kleinschreibung, Worte innerhalb Bezeichner mit großem Anfangsbuchstaben: Bsp. `lastElement`
Konstanten: `final <Typ> <Bezeichner> = <wert>`
Konvention: Nur Großbuchstaben

Anweisungen: Zuweisung <variable> = <ausdruck>
Fallunterscheidung, Schleifen, Sprunganweisungen, Mehrfachverzweigung (switch)

Java-Syntax: Semikolon **;** schließt Anweisung ab. Eine oder mehrere Anweisungen können durch Blockklammern zusammengefaßt werden {...}

Kurze Zusammenfassung (Java)

Methoden (in anderen Programmiersprachen [Funktions-]prozeduren):

- Deklarationen von lokalen Variablen und Folge von Anweisungen, die Effekte haben und /oder Werte berechnen; immer in {.. } eingeschlossen
- haben formale Parameter (oder keine - aber immer Klammern)
  `void foo( ){}` eins der kürzesten Programme
- besitzen Typ (des berechneten Wertes) oder sind void .
- sind Bestandteil von Klassen

beachte:

- Methoden könnén keine aktuellen Parameter sein *(map ???)*
- Funktionen als Werte? auch nicht möglich!

Java-Konvention: Methodenbezeichner mit kleinem Anfangsbuchstabem, Anweisungen innerhalb des Blocks „mit Geschmack“ einrücken.