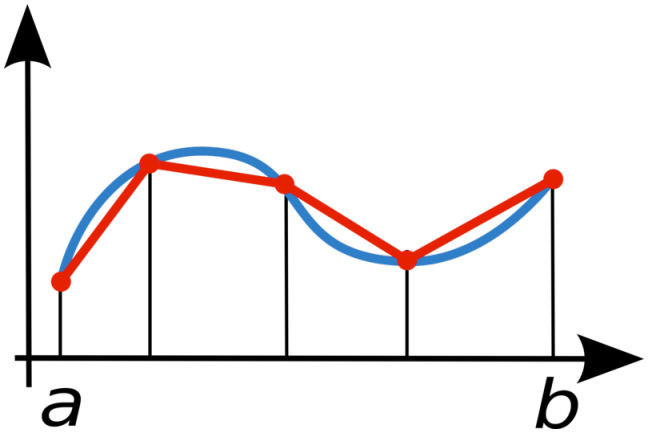
Da bin ich mit dem Abitur fertig und nun kommt Mathe zurück. Sch**sse!



Numerische Algorithmen

OOPM, Ralf Lämmel

Vergl.: Numerische Mathematik
abgeleitet von [Wikipedia, 1 Nov 2009]

Die **numerische Mathematik,** kurz **Numerik** genannt, beschäftigt sich mit der Konstruktion und Analyse von Algorithmen für kontinuierliche mathematische Probleme. Interesse an solchen Algorithmen besteht meist aus einem der folgenden Gründe:

1. Es gibt zu dem Problem keine explizite Lösungsdarstellung.
2. Die explizite Darstellung ist nicht für eine schnelle bzw. genaue Berechnung geeignet.

Unterschieden werden **direkte Verfahren** (z.B. *gaußsches Eliminationsverfahren*), die nach endlicher Zeit bei unendlicher Rechnergenauigkeit die exakte Lösung eines Problems liefern, und **Näherungsverfahren** (z.B. das *Newton-Verfahren*), die Approximationen liefern.

Berechnung der Summe 1, .., n
(**Degeneriertes** Beispiel für ein **direktes** Verfahren)

```
public static int sum(int n) {
  int result = 0;
  while (n>0)
    result += n--;
  return result;
}
```

Dekrementierung findet nach lesendem Zugriff statt.

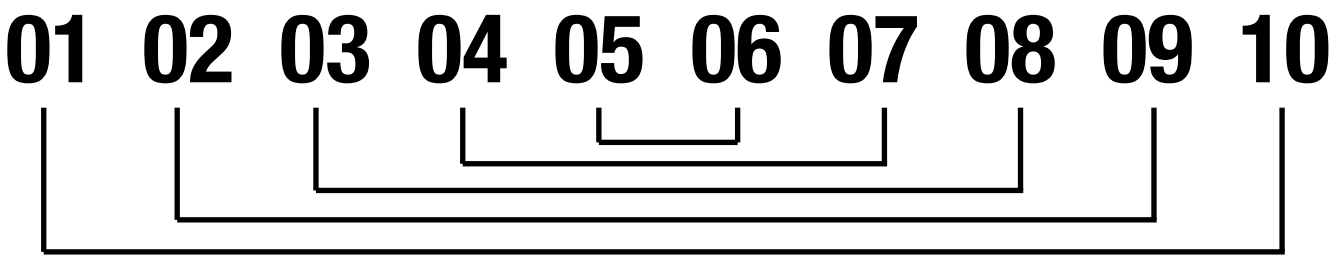
Vergl. Struktur der Lösung mit der Lösung der Fakultätsfunktion.

Berechnung der Summe 1, .., n

(Verwendung einer **expliziten** Lösungsdarstellung)

```
public static int sum(int n) {
  return n/2 * (n+1);
}
```

Inkorrekt: Etwa 1/2 ist 0 im Datentype „int“. Wir brauchen 0.5



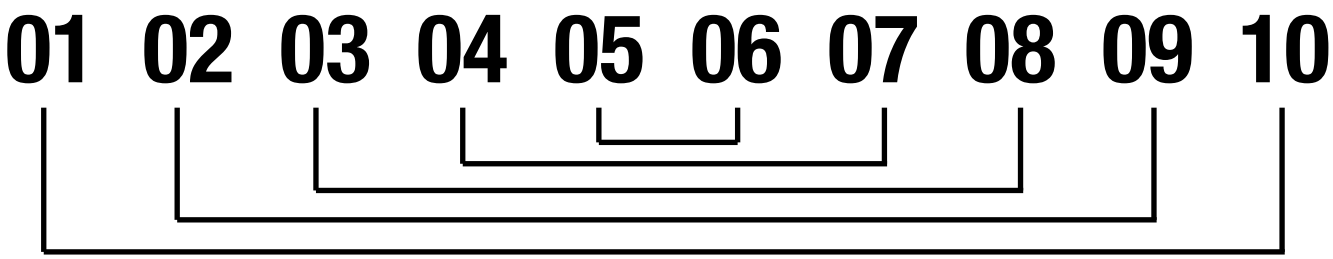
Vergl. Effizienz der iterativen und der expliziten Lösungsdarstellung.

Berechnung der Summe 1, .., n

(Verwendung einer **expliziten** Lösungsdarstellung)

```
public static float sum(int n) {
  return n/2.0f * (n+1);
}
```

Unschön: wir arbeiten nun „unnötigerweise“ im Datentyp float



Vergl. Effizienz der iterativen und der expliziten Lösungsdarstellung.

Potenzierung (engl. Exponentiation)

- Problem
  - Berechne x^n
- Naive Methode
  - $x^n = x * \ldots * x$

  n mal

```
public static int power(int x, int n) {
   int result = 1;
   for (int i=1; i<=n; i++)
      result *= x;
   return result;
}
```

- Wie geht das schneller?

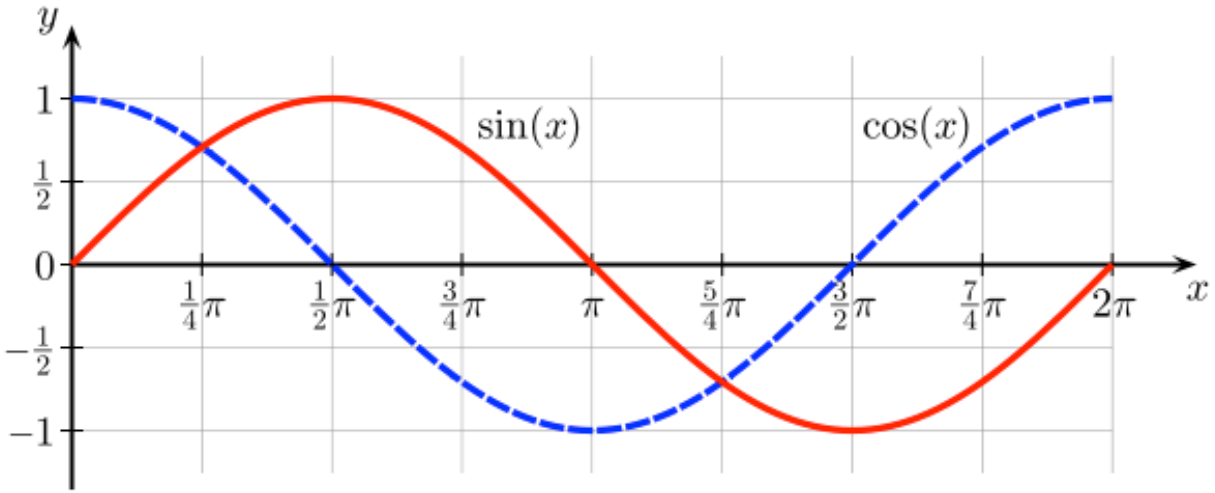
Schnelle Potenzierung

- Zu berechnen: x^n
- Fall 1: n ist gerade, etwa $n = 2k$
  - Berechne $x' = x^2$.
  - Gib x'^k zurück.
- Fall 2: n ist ungerade, etwa $n = 2k + 1$
  - Berechne $x' = x^2$.
  - Gib x * x'^k zurück.

Imperative und iterative Java-Implementation schneller Potenzierung

```
public static int power(int x, int n) {
    int k = n;
    int p = x;
    int y = 1;
    while (k>0)
        if (k % 2 == 0) {
            p = p * p;
            k = k / 2;
        }
        else {
            y = y * p;
            k = k - 1;
        }
    return y;
}
```

Die Sinusfunktion



Quelle: http://de.wikipedia.org/wiki/Sinus_und_Kosinus

Die Sinusfunktion

Rad	Deg	Sin	Cos	Tan	Csc	Sec	Cot		
.0000	**00**	.0000	1.0000	.0000	-----	1.0000	-----	**90**	**1.5707**
.0175	**01**	.0175	.9998	.0175	57.2987	1.0002	57.2900	**89**	**1.5533**
.0349	**02**	.0349	.9994	.0349	28.6537	1.0006	28.6363	**88**	**1.5359**
.0524	**03**	.0523	.9986	.0524	19.1073	1.0014	19.0811	**87**	**1.5184**
.0698	**04**	.0698	.9976	.0699	14.3356	1.0024	14.3007	**86**	**1.5010**
.0873	**05**	.0872	.9962	.0875	11.4737	1.0038	11.4301	**85**	**1.4835**
.1047	**06**	.1045	.9945	.1051	9.5668	1.0055	9.5144	**84**	**1.4661**
.1222	**07**	.1219	.9925	.1228	8.2055	1.0075	8.1443	**83**	**1.4486**
.1396	**08**	.1392	.9903	.1405	7.1853	1.0098	7.1154	**82**	**1.4312**
.1571	**09**	.1564	.9877	.1584	6.3925	1.0125	6.3138	**81**	**1.4137**

Quelle: http://math2.org/math/trig/tables.htm

Taylor-Reihe für Sinusfunktion

- Der Sinuswert entspricht dem folgenden Limit:

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} \mp \cdots$$

- Konvergiert die Reihe? Warum?
- Wie definiert man ein gutes Abbruchkriterium?

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} \mp \cdots$$

Naive Implementation

```
static double sin(double x) {
    double result = 0;
    int n = 0;
    double delta;
    do {
        delta = power(-1,n) * power(x,2*n + 1) / factorial(2*n + 1);
        result += delta;
        n++;
    } while (Math.abs(delta) > MAX_ERROR);
    return result;
}
```

Was ist eine do-while-Schleife?

Was ist ein guter Wert für MAX_ERROR?

Kann man das wesentlich effizienter machen?

Einschub: Schleifenkonstruktionen

- While-Schleife (*Vorabtesten* der Bedingung)
  - Syntax:
    - **while** (Bedingung) Anweisung;
    - **while** (Bedingung) { ... }
- Do-While-Schleife (Nachgeschaltetes Testen)
  - Syntaktischer Zucker:
    - **do** Statement; **while** (Bedingung);
  - Expansion:
    - Anweisung; **while** (Bedingung) Anweisung;

Einschub: “Gesittete Sprünge”

- **Break**
  - Nichttrivialer syntaktischer Zucker:
    - **while** (B) { A1; **break**; }
  - Expansion:
    - boolean x = false; **while** (B && !x) { A1; x = true; }

- **Continue**
  - Nichttrivialer syntaktischer Zucker:
    - **while** (B1) { A1; if (B2) **continue**; A2; }
  - Expansion:
    - **while** (B1) { A1; if (!B2) A2; }

Verwendung von Break

```
static double sin(double x) {
   double result = 0;
   int n = 0;
   while (true) {
      double delta =
     power(-1,n) * power(x,2*n + 1) / factorial(2*n + 1);
      result += delta;
      if (Math.abs(delta) < MAX_ERROR)
         break;
      n++;
   }
   return result;
}
```

Endlosschleife

Deklaration im Schleifenbereich

Frühzeitiger Abbruch

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} \mp \cdots$$

- Mathematische Einblicke:
  - $(-1)^n$ ist für das *alternierende* Vorzeichen.
  - Es gibt Summanden für alle ungerade Zahlen.
  - Die Glieder können auseinander ermittelt werden:
    - sin(x) = delta(x,0) - delta(x,1) + delta(x,2) - ...
      - delta(x,k) = $x^{(2k+1)}$ / (2k+1)!
    - delta(x,0) = x
    - delta(x,k+1) = delta(x,k) * x^2 / ((2k+3)*(2k+2))

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} \mp \cdots$$

```
static double sin(double x) {
   double x2 = x * x;
   double delta = x;
   double result = x;
   int i = 1;
   while (Math.abs(delta) > MAX_ERROR) {
      delta = -delta * x2 / ((i+1)*(i+2));
      result += delta;
      i += 2;
   }
   return result;
}
```

Schleifenvariable läuft über ungerade Zahlen.

Potenzierung (von x) wird vereinfacht zu einer Multiplikation.

Fakultät wird vereinfacht zu wenigen Additionen & Multiplikationen.

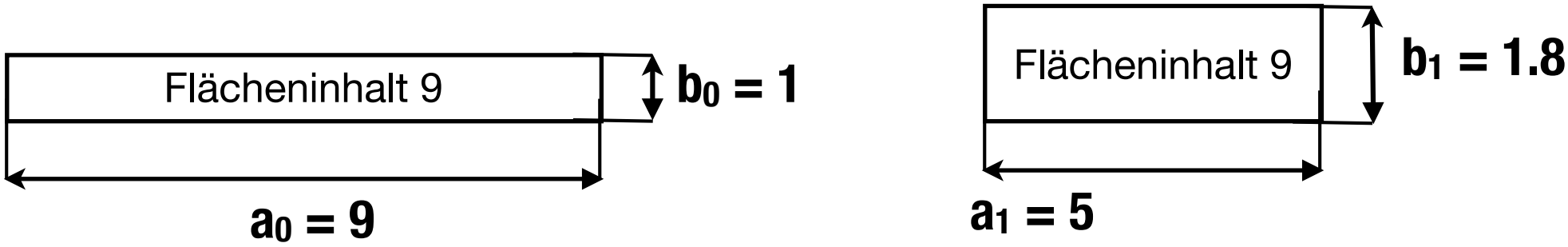
Maschinenepsilon: Differenz zwischen 1 und kleinster, darstellbarer Zahl die größer ist als 1 (z.B.:1.1920929E-7)

```
public static float macheps() {
  float r = 1.0f;
  do
    r /= 2.0f;
  while ((1.0f + (r / 2.0f)) != 1.0f);
  return r;
}
```

Das “f” in “2.0f” markiert eine float-Konstante.
Siehe auch “d” für double.
Ohne Markierung befinden wir uns in double.
Ein gutes “MAX_ERROR” im Beispiel wäre 1.0E-6.

Approximative Wurzelberechnung nach dem Heron-Verfahren

http://de.wikipedia.org/wiki/Heron-Verfahren



$$a_1 = (a_0 + b_0) / 2 = 5$$
$$b_1 = x / a_1 = 1.8$$

Gegeben: x

Gesucht: Wurzel aus x

Ansatz:

- Wurzel ist Seitenlänge eines Quadrates mit Flächeninhalt x.

- Beginne mit Rechteck mit dem Flächeninhalt x.

- Passe Seitenlängen an zur Annäherung an ein Quadrat.

Approximative Wurzelberechnung

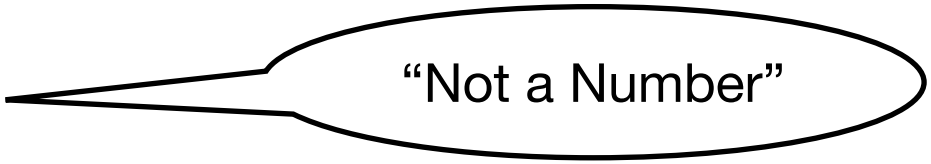
- Eingabe: *x*
- Ausgabe: *r* = (Näherung der) Wurzel von *x*
- Schritte

1. Nimm x als erste Näherung von *r* an.
2. Brich ab falls *r*r* nahe genug an *x*.
3. **Ersetze *r* durch (*x* / *r* + *r*) / 2.**
4. Gehe zurück nach 2.

Falls *r* grösser als die Wurzel ist, dann ist *x*/*r* kleiner als die Wurzel. Damit ist (*x* / *r* + *r*) / 2 eine bessere Abschätzung.

Approximative Wurzelberechnung

```
public static double sqrt(double x, double epsilon) {
  if (x < 0)
    return Double.NaN;
  double r = x;
  while (Math.abs(x - r * r) > epsilon)
    r = (x / r + r) / 2.0;
  return r;
}
```



Weitere Grundlagen zu den Zahlentypen

- Was gibt es überhaupt an Zahlentypen?
- Was ist die Genauigkeit von int, long, u.a.?
- Wie werden int/float/double/... dargestellt?
- Was gibt es an grundlegenden Operatoren?

Ganze Zahlen: Typen und Grenzen

Nur beiläufig erwähnt.
Nicht prüfungsrelevant.

Typ	Bits	Grenzen
byte	8	-128 ... 127
short	16	-32768 ... 32767
int	32	-2147483648 ... 2147483647
long	64	...

Maximaler Byte-Wert

```
public static byte MaxByte() {
  byte i = 0;
  byte r;
  do
    r = i;
  while (++i > 0);
  return r;
}
```

Maximaler Integer-Wert

```
public static int MaxInteger() {
  int i = 0;
  int r;
  do
    r = i;
  while (++i > 0);
  return r;
}
```

Darstellung ganzer Zahlen
1. Option: **Vorzeichendarstellung**

000 = + 0	**1**00 = - 0
001 = + 1	**1**01 = - 1
010 = + 2	**1**10 = - 2
011 = + 3	**1**11 = - 3

- Führendes Bit fungiert als Vorzeichen
- Nachteile
  - Null wird doppelt repräsentiert.
  - Schriftliche Addition ist nicht mehr anwendbar.

001 + 110 ≠ 111

Darstellung ganzer Zahlen
2. Option: **Zweierkomplement**

000 = + 0	**1**00 = - 4
001 = + 1	**1**01 = - 3
010 = + 2	**1**10 = - 2
011 = + 3	**1**11 = - 1

- Führendes Bit repräsentiert -2^n
  - *n* ist hier die Anzahl der Binärstellen ohne Vorzeichen
- Vorteile
  - Positive Zahlen werden gleich repräsentiert.
  - Zahl + Bitweises Komplement = -1
  - Einfache arithmetische Operationen

001 + 110 = 111

Darstellung reeller Zahlen

- **Anforderungen**
  - Abdeckung eines großen Intervalls reeller Zahlen
  - Stellengenauigkeit unabhängig von *Größenordnung*
- Konzept der Gleitkommazahlen
  - Vorzeichenbit *V*
  - Exponent *E* (etwa 8 Bits)
  - Mantisse *M* (etwa 23 Bits)
  - Normierungen (hier nicht besprochen)

Gleitkommazahlen: Typen und Grenzen

Nur beiläufig erwähnt.
Nicht prüfungsrelevant.

Typ	Größe	Bereich	Stellen
float	32	+/- $3.4 * 10^{38}$	6-7
double	64	+/- $1.8 * 10^{308}$	15

Notation für Gleitkommazahlen:

- Standard-Notation: eine Folge von Ziffern für den ganzzahligen Teil, gefolgt von einem Punkt, gefolgt von Ziffern fur den Nachkommaanteil. Das optionale Vorzeichen entspricht der Anwendung eines Operators.
  - 123450.0, -123450.0, 0.000012345
- Wissenschaftliche Notation: ... gefolgt von “e” und optionales Vorzeichen und Exponent zur Basis 10.
  - 1.2345e5, -1.2345e+5, 1.2345e-5
- Postfix “f” vs. “d” für float vs. double.

Operatoren für Zahlen und Wahrheitswerte

- Arithmetische Operatoren: +, -, *, /, % (+ und - auch unär)
- Binäre Vergleichsoperatoren: ==, !=, <=, >=
- Ternärer Operator: x > y ? x - y : y - x
- Boolesche Operatoren: !, &, |, ^
- Kurzschluss-Operatoren: &&, ||
- Prefix In/Dekrementierung: ++x, --x
- Postfix In/Dekrementierung: x++, x--
- Bitweise Operatoren: ~x, x&y, x|y, x>>n, x<<n

Komplizierte Semantik

Siehe auch Package **`idioms.operators`**.

Verwendung des ternären Operators

Absolutbetrag mit ternärem Operator

```
public static float abs(float x) {
   return x < 0 ? - x : x;
}
```

Absolutbetrag mit ‘if-then-else’

```
public static float abs(float x) {
   if (x < 0)
      return - x;
   else
      return x;
}
```

Der ternäre Operator trägt also ein ‘if-then-else’ auf der Ebene der Ausdrücke (anstatt der Anweisungen) bei.

Kurzschluss-Operatoren

	CODE	BEDEUTUNG
OHNE KURZSCHLUSS	`x & y` `x | y`	**Werte x und y aus. Verknüpfe die Teilergebnisse.**
MIT KURZSCHLUSS	`x && y` `x || y`	**Werte x aus. Werte y nur dann aus, wenn es denn Wahrheitswert noch beeinflussen kann.**

Wir nutzen diese Gesetze:
false ‘und’ *y* = false
true ‘oder’ *y* = true

Postix/prefix increment/decrement

	CODE	VARIABLENBELEGUNG
PREFIX	`int x = 1;` `int y = ++x;`	`x ➡ 2` `y ➡ 2`
POSTFIX	`int x = 1;` `int y = x++;`	`x ➡ 2` `y ➡ 1`

Ein Prefix-Operator wird ausgeführt bevor der umschliessende Ausdruck ausgewertet wird. Beim Post-Operator geschieht es danach.

Beachte: Viele mathematische Funktionen werden durch java.lang.Math angeboten.

Method Summary	
static double	**abs**(double a) Returns the absolute value of a double value.
static float	**abs**(float a) Returns the absolute value of a float value.
static int	**abs**(int a) Returns the absolute value of an int value.
static long	**abs**(long a) Returns the absolute value of a long value.
static double	**acos**(double a) Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*.
static double	**asin**(double a) Returns the arc sine of a value; the returned angle is in the range -*pi*/2 through *pi*/2.
static double	**atan**(double a) Returns the arc tangent of a value; the returned angle is in the range -*pi*/2 through *pi*/2.
static double	**atan2**(double y, double x) Returns the angle *theta* from the conversion of rectangular coordinates (x, y) to polar coordinates (r, *theta*).

■■■

- **Zusammenfassung**
  - Grundlegende Programmierfähigkeiten
    - Iterative Verfahren
    - Approximative Verfahren
  - Herausforderungen der numerischen Mathematik
    - Umgang mit diskreten, endlichen Float/Doubles
    - Abwägung von Genauigkeit, Zeit-/Speicheraufwand
- **Ausblick**
  - Nächstes Mal: Programmierung mit Feldern
  - Übernächstes Mal: Rekursive Funktionen