Übersicht

4 Funktionen (in Java)

- Einführung
- Funktionen in Java
- Rekursion

Übersicht

Einführung

- Wir können bereits *beliebige* Algorithmen implementieren.
  - Bekannte Grundkonzepte sind dazu ausreichend.
  - Können wir das *besser* machen?
- Abstraktionsprinzip
  - Jede Funktionalität sollte nur *einmal* implementiert sein
  - Hinarbeiten auf Abstraktion ⇒ Lesbarkeit, Wartbarkeit, . . .
    - *DRY*-Prinzip – "**D**on't **r**epeat **y**ourself !"
    - oder auch *DIE* – "**D**uplication **i**s **e**vil !"
    - oder einfach *code reuse*
  - Funktionen sind *ein* mögliches Mittel der Abstraktion
  - *Jetzt:* Algorithmen – *später* auch Daten(strukturen)
- Rekursion
  - „selbstbezogene" Definition
  - Alternative zur Iteration (Schleife)
  - Ersetzt Schleifen in funktionalen Programmiersprachen
  - Oft intuitiver als Schleife!

Übersicht

4 Funktionen (in Java)
- Einführung
- Funktionen in Java
- Rekursion

Beispiel: approximiere Quadratwurzel

- Nach Heron (Anwendung des Newton-Verfahrens)
- Für $x \geqslant 0$ berechne $s \approx \sqrt{x}$, so dass der Fehler $|s^2 - x| < \epsilon$

```
double x=...;           // input (expect x⩾0)
double EPS=x*1e-12;     // maximum error
double s=1.0;           // output: square root
double error;           // current error

do {
  double t=x/s;         // invariant: s·t=x
  s=(s+t)/2.0;          // we want: s≈t

  error=s*s-x;
  error=(error>=0.0 ? error : -error);

} while (error>=EPS);
```

Beispiel: approximiere Quadratwurzel

- Wir wollen $\sqrt{x}$ in anderen Algorithmen verwenden
  - Bisherige Möglichkeit: *copy&paste* — unschön!
  - Jetzt: definiere *Funktion* $\mathtt{sqrt} : \mathbb{R}_0^+ \to \mathbb{R}$ mit $\mathtt{sqrt}(x) = \sqrt{x}$
- In Java

```
public static double sqrt(double x) {
  double EPS=x*1e-12;
  ...
  return s;
}
```

- Neue Schlüsselwörter `public`, `static` und `return`
  - `public` und `static` kennen wir schon von `main(String[] args)` — *dazu später (OOP) mehr!*
  - `return` beendet Funktion und liefert Ergebnis zurück
- Implementierung von `sqrt` berechnet *Absolutbetrag* des Fehlers

Beispiel: Absolutbetrag

- Definiere $\mathrm{abs} : \mathbb{R} \to \mathbb{R}$ mit $\mathrm{abs}(x) = |x| = \begin{cases} x \geqslant 0: & x \\ x < 0: & -x \end{cases}$
- In Java

```
public static double abs(double x) {
  if (x>=0.0)
     return x;
  else
     return -x;
}
```

oder kürzer mit `?:`-Operator

```
public static double abs(double x) {
  return x>=0.0 ? x : -x;
}
```

Definition einer Funktion in Java

- Definition einer *Funktion* wird eingeleitet durch `static`
- Danach folgt die Signatur der Funktion

```
// f : T1 × T2 × ··· × Tn  →  T

public static T f(T1 arg1, T2 arg2, ..., Tn argn)
```

  - Bezeichner f = Name der Funktion
  - T Rückgabetyp
  - $T_1,\ldots,T_n$ Typen der *Argumente* `arg1`,...,`argn`
- *Signatur* kann auch nur den Typen Namen bezeichnen.
- Nach der Signatur folgt Block mit *Definition* der Funktion
  - `return` Anweisung verlässt die Funktion und liefert Ergebnis
  - „Sprung“ zurück in die aufrufende Funktion

Definition einer Funktion in Java: Sonderfälle

- Liste von Argumenten kann *leer* sein

```
public static int one() { return 1; }
```

  - z.B. Konstante 1-Funktion

- Funktionen *müssen keinen* Rückgabewert liefern
  - In anderen Programmiersprachen (z.B. Pascal): Prozeduren
  - In Java: spezieller “(Nicht-)Datentyp” `void`

```
public static void main(String[] args) {
  if (args.length<2)
    return; // NO return VALUE
  ...
}
```

i.d.R. eine Form von Zustandsänderung z.B. Ausgabe

- Auswertung der Funktion = Funktions*aufruf* (*function call*)
- Beispiel

```
public static void main(String[] args) {
  double x=Double.parseDouble(args[0]);

  x=abs(x);
  double y=sqrt(x);

  y=sqrt(abs(x));    // or rather like this
}
```

- **Syntax**

```
T y;                    // ausser fuer T=void
y=f(arg1,arg2,...,argn);
```

Beispiel: Quadratwurzel

```java
/** Approximate square root of x>=0;
    @param x input
    @return square root of x,
            result is undefined if x<0!
 */
public static double sqrt(double x) {
  double EPS=x*1e-12;
  double s=1.0;

  do {
    double t=x/s;
    s=(s+t)/2.0;

  } while (abs(s*s-x)>=EPS);

  return s;
}
```

Ein Java Programm

- Argument der `main()` Funktion: Feld von Zeichenketten `args`
  - Kommandozeilenargumente, z.B. `java Test 1 2 3`

```
public static main(String[] args) {
  double x=Double.parseDouble(args[0]);
  System.out.println(sqrt(abs(x)));
}
```

- Was genau passiert in der dritten Zeile?
  1. Aufruf von `abs(x)`: $\xi_1 = |x|$
  2. Aufruf von `sqrt`(ξ_1): $\xi_2 = \sqrt{\xi_1}$
     1. Aufruf von `abs(s*s-x)`: $\xi_{error} = |s^2 - x|$
  3. Aufruf von `System.out.println`(ξ_2)
- Aufrufe lassen sich visualisieren
  - Probeweise `System.err.println` einbauen
  - im Debugger (Verlauf und *call stack*)

Weitere Beispiele

```java
// Maximum max : ℤ×ℤ → ℤ with max(a,b) = max{a,b}
public static int max(int a,int b) {
  return a>=b ? a : b;
}

// Factorial n!
public static int factorial(int n) {
  int x=1;
  for (int i=1;i<=n;++i)
    x*=i;                         // No check for overflow!
  return x;
}

// Binomial coefficient $\binom{n}{i}$
public static int bincoeff(int n,int i) {
  // Naive! Even worse: no checks for overflow!
  return factorial(n)/factorial(i)/factorial(n-i);
}
```

Überladen einer Funktionsdefinition

- Funktionen unterscheiden sich durch ihre Signatur, d.h. durch
  - *Funktionsnamen* **und** / **oder**
  - *Typ* und *Argumentliste* (Anzahl, Reihenfolge und Typen)
- Beispiel: Die folgenden Funktionen *unterscheiden* sich!

```
// Maxmium max : ℤ × ℤ → ℤ
public static int max(int a,int b) {
  return a>=b ? a : b;
}

// Maxmium max : ℝ × ℝ → ℝ
public static double max(double a,double b) {
  return a>=b ? a : b;
}
```

- Java wählt *automatisch* die passende Variante

Beispiel: Maximum

- Aufruf der Funktion `max` für `int` und `double` Argumente

```
int    i1=max(1,2);              // max : ℤ × ℤ → ℤ
double d1=max(1.0,2.0);          // max : ℝ × ℝ → ℝ
double d2=max(1,2.0);            // max : ℝ × ℝ → ℝ
int    i2=max((int) d2,i1);// max : ℤ × ℤ → ℤ
```

- Anmerkungen zur Typumwandlung
  - `d2`: *Automatische* Umwandlung in „mächtigeren" Typ `double`
  - `i2`: *explizite* Umwandlung $\Rightarrow$ `int` Argumente

Polymorphe Funktionen

Definition (Polymorphe Funktion)

Eine Funktion, die verschiedene Datentypen als Argument(e) und ggf. Rückgabewert erlaubt, heißt *polymorph*.

- Die `max()` Funktion im Beispiel ist *polymorph* (vielgestaltig).
  - Der Funktionsname bleibt gleich.
  - Der *Typ* des Arguments (und in diesem Fall auch des Rückgabewerts) unterscheidet sich.
- Bei mehreren Argumenten *mindestens ein* Typ unterschiedlich
- Es kann sich auch die *Anzahl* der Argumente unterscheiden.
- Man spricht auch vom Überladen einer Funktion (*function overloading*).
- Beachte Typumwandlung!

Beispiel: Quadratwurzel mit Fehlerschranke als Argument

- Überlade Funktion zur Definition von *Standardargumenten*

```
/* Approximate square root of x>=0;
     @param x input
     @param reltol relative error tolerance
                   (default value: 1e-12)
     @return square root of x,
             result is undefined if x<0!
*/
public static double sqrt(double x,double reltol) {
  double EPS=x*reltol;
  double s=1.0;
  ...
  return s;
}

public static double sqrt(double x) {
  return sqrt(x,1e-12);
}
```

Auswertung von Argumenten

- Java übergibt grundsätzlich Kopien der Werte (*call-by-value*)
- Zuweisung innerhalb ändert Wert außerhalb der Funktion nicht

```
public static void foo(int x) { ++x; }
int y=0;
foo(y);
System.out.println(y); // no change: y==0!
```

- Für **primitive Datentypen** (z.B. `int`) *Kopie* des Werts
- Für **alle anderen** Datentypen (z.B. Felder) *Kopie* der Referenz

```
public static void bar(int[] x) { ++x[0]; }
int[] y={0,0};
bar(y);
System.out.println(y[0]); // y[0]=1 changed!
```

Auswertung von Argumenten in Java

- **Primitive** Datentypen (z.B. `foo(int x)`)

```
int y=2;
foo(y);
```

id	type	value
y	int	2

Argument x

id	type	value
x	int	2

- **Alle anderen** Datentypen (z.B. `bar(int[] x)`)

```
int[] y={1,2,3};
bar(y);
```

id	type	value
y	int[]	•

Argument x

id	type	value
x	int[]	•

1 | 2 | 3

Call-by-value / call-by-reference

- Call-by-value
  - Argumente werden als *Werte* an Funktion übergeben
  - Wie in mathematischer Schreibweise üblich z.B. $y = f(x)$
  - Argument als neue Variable, die mit *Kopie* des Wertes initialisiert wurde
  - Änderung der Argumente wirken sich nur innerhalb der Funktion aus!
  - Ausnahme: Es wird eine Referenz übergeben, z.B. ein Feld: Dann sind bleibende Änderungen referenzierter Daten möglich.
- Call-by-reference
  - Argumente sind *grundsätzlich* Referenzen auf Daten
  - z.B. auf Variablen der aufrufenden Funktion
  - Änderungen betreffen referenzierte Daten
  - Call-by-reference Semantik muss Referenzen auf alle Datentypen **und** Referenzen ermöglichen.
    Das ist in Java nicht möglich! (z.B. aber in C mit Zeigern)

Einschränkungen durch call-by-value in Java

- Java folgt *call-by-value* und erstellt dabei
  - *Kopien* von Werten (primitive Datentypen)
  - *Kopien* von Referenzen (andere)

- **Konsequenz:** – Folgendes funktioniert so nicht in Java!

```
// cannot "reset" an array in a function
public static void reset(int[] x) {
  x=new int[2]; x[0]=x[1]=0; // No effect outside!
}
// cannot have a function swap references
public static void swap(Object a,Object b) {
  Object tmp=b; a=b; b=tmp;  // No effect outside!
}
```

- „Problem“: Zuweisung an *Kopien* von Referenzen!
- In Java können referenzierte Daten manipuliert werden, nicht aber Referenzen selbst!

Kurze Zusammenfassung: Funktionen in Java

- Funktionen als Mittel zur Abstraktion
  - Jede Funktionalität ist einmal implementiert.
  - Strukturiere Programm in Funktionen (als „Unterprogramme“)
- Signatur: Argumentliste und Typ des Rückgabewerts
- Definition als Block
- Neue Schlüsselwörter `return`, `void`
  - auf `public static` werden wir *später* eingehen (OOP)
- Polymorphe Funktionen oder Überladen von Funktionen
  - Funktionen unterscheiden sich nicht im Namen aber in der Signatur
  - *Weiterer Abstraktionsschritt!*
  - Noch mehr Abstraktion: *generics* $\Rightarrow$ Typen als Parameter (später!)
- *Call-by-value* und *call-by-reference*

- Als nächstes: *Rekursion*

Rekursion

- Rekursion = Anwendung desselben Prinzips auf Teilprobleme
- Beispiel: Türme von Hanoi
  - *Wenn ich das kleinere (Teil-)Problem lösen kann, dann weiß ich, wie ich das eigentliche Problem lösen kann.*
  - Hier: *kleiner* = Turm der Höhe $n-1$
  - D.h. *ich kann jedes Problem kleiner machen!*
  - Dabei ist das „*kleinste* Problem“ einfach direkt zu lösen!
  - Hier: ziehe unterste Scheibe
- Oft auch *divide-and-conquer* („teile und herrsche“) Ansatz
  - z.B im Kapitel zu *Suchen & Sortieren*:
  - Binäre Suche, Quicksort, Mergesort
- Viele Anwendungen auch im nächsten Semester in der Vorlesung *Algorithmen und Datenstrukturen*

Definition (Rekursive Funktion)

Eine Funktion heißt *rekursiv*, wenn sie – direkt oder indirekt – durch sich selbst definiert ist.

- Beispiel: Fakultätsfunktion

$$n! \;=\; \underbrace{1\cdot 2\cdot\cdots(n-1)\cdot n}_{\text{iterative Definition}} \;=\; \underbrace{\begin{cases} n\leqslant 1: & 1\\ n>1: & n\cdot(n-1)! \end{cases}}_{\text{rekursive Definition}}$$

- Beispiel: odd,even : $\mathbb{N}_0 \to \{\mathtt{true},\mathtt{false}\}$ (Indirektion)

$$\mathtt{odd}(x)=\begin{cases} x=0: & \mathtt{false}\\ x>0: & \mathtt{even}(x-1)\end{cases},\quad \mathtt{even}(x)=\begin{cases} x=0: & \mathtt{true}\\ x>0: & \mathtt{odd}(x-1)\end{cases}$$

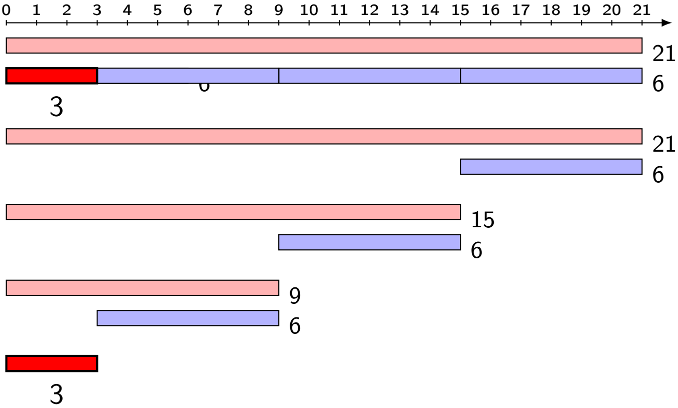
Beispiele in Java

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

public static int factorial(int n) {
  if (n<=1)
    return 1;
  else
    return n*factorial(n-1);
}

public static boolean odd(int x) {
  return x==0 ? false : even(x-1);
}
public static boolean even(int x) {
  return x==0 ? true : odd(x-1);
}
```

Beispiel: Euklidscher Algorithmus

- Der *größte gemeinsame Teiler* zweier ganzer Zahlen m und n ist die größte natürliche Zahl, durch die sowohl m als auch n ohne Rest teilbar sind.
  - $\mathrm{ggT}(m, n)$ oder auch $\mathrm{gcd}(m, n)$ (*greatest common divisor*)
  - $\mathrm{ggT}(0, 0)$ ist *undefiniert*
- Beispiel: $\mathrm{ggT}(8, 12) = 4$
  - Berechnung z.B. durch Primfaktorzerlegung
    $8 = 2^3, \quad 12 = 2^2 \cdot 3 \;\Rightarrow\; \mathrm{ggT}(8, 12) = 2^2 = 4$
  - Anwendung z.B. Kürzen von Brüchen $\frac{8}{12} = \frac{2 \cdot 4}{3 \cdot 4} = \frac{2}{3}$
- Der Euklidsche Algorithmus berechnet $\mathrm{ggT}(m, n)$
  - Beschrieben in Euklids *Elemente* (um 325 v. Chr.)
- Idee: Sei $m \leqslant n$, dann $\mathrm{ggT}(m, n) \;=\; \mathrm{ggT}(m, n - m)$
  - Rekursive Vorschrift zur Berechnung
  - Problem wird in jedem Rekursionsschritt kleiner (bis $m = 0$)

- Idee: Sei $m \leqslant n$, dann $\mathrm{ggT}(m,n) = \mathrm{ggT}(m,n-m)$
- Visualisierung: trage ganzzahlige Längen ab
  - Beispiel: $\mathrm{ggT}(6,21)=3$



- $\mathrm{ggT}(6,21)=\mathrm{ggT}(6,15)=\mathrm{ggT}(6,9)=\ldots=3$

Euklidscher Algorithmus

- Rechenregeln
  - $\text{ggT}(0,0)$ ist *undefiniert*
  - $\text{ggT}(0,n) = n$ — Endbedingung
  - $\text{ggT}(m,n) = \text{ggT}(n,m)$ — Vertauschen wenn $m > n$
  - $\text{ggT}(m,n) = \text{ggT}(m,n-m)$ — Anwenden wenn $m \leqslant n$
- ggT rekursiv in Java

```java
public static final int UNDEF=0;

// compute gcd(m,n) for m>=0 and n>=0
public static int gcd(int m,int n) {
  if (m==0 && n==0) return UNDEF;
  else if (m==0)    return n;
  else if (m>n)     return gcd(n,m);
  else              return gcd(m,n-m);
}
```

  - final definiert eine Konstante (Konvention: Großschreibung)
  - Jede Zahl ist durch 1 teilbar $\Rightarrow$ UNDEF=0 sinnvoll

- Zusätzliche Ausgabe

```
System.err.println("gcd("+m+","+n+")");
```

- Berechne `gcd(6,21)`
  - Aufruf in `System.out.println(gcd(6,21));`

```
public static int gcd(int m,int n) {
  System.err.println("gcd("+m+","+n+")");

  if (m==0 && n==0) return UNDEF;
  else if (m==0)    return n;
  else if (m>n)     return gcd(n,m);
  else              return gcd(m,n-m);
}
```

```
gcd(6,21)
gcd(6,15)
gcd(6,9)
gcd(6,3)
gcd(3,6)
gcd(3,3)
gcd(3,0)
gcd(0,3)
3
```

Iterative Darstellung des Euklidschen Algorithmus

- Rekursion lässt sich (hier einfach) durch Iteration ersetzen
  - Ähnlich wie bei der Fakultätsfunktion
  - Spezialfall: *Endrekursion*
- Rekursive Darstellung

```
public static int gcd(int m,int n) {
  if (m==0 && n==0) return UNDEF;
  else if (m==0)    return n;
  else if (m>n)     return gcd(n,m);
  else              return gcd(m,n-m);
}
```

- Iterative Darstellung

```
public static int gcd_iterative(int m,int n) {
  if (m==0 && n==0) return UNDEF;
  while (m!=0) {
    if (m>n)      { int i=m; m=n; n=i; }
    else          { n-=m; }
  }
  return n;
}
```

Iterationsschritte von gcd_iterative(m,n)

- Zusätzliche Ausgabe

```
System.err.println("m="+m+", n="+n);
```

- Berechne `gcd_iterative(6,21)`
  - Aufruf in `System.out.println(gcd_iterative(6,21));`

```
public
static int gcd_iterative(int m,int n) {
  if (m==0 && n==0) return UNDEF;
  while (m!=0) {
    System.err.println("m="+m+", n="+n);
    if (m>n)      { int i=m; m=n; n=i; }
    else          { n-=m; }
  }
  return n;
}
```

```
m=6, n=21
m=6, n=15
m=6, n=9
m=6, n=3
m=3, n=6
m=3, n=3
m=3, n=0
3
```

Anmerkungen zum ggT

- Effizientere Version des Euklidschen Algorithmus mit Regel

$$\mathrm{ggT}(m,n) = \mathrm{ggT}(m, n \bmod m)$$

- Das entspricht k Anwendungen der alten Regel $\mathrm{ggT}(m, n-m)$

$$\mathrm{ggT}(m,n) = \mathrm{ggT}(m, n - k \cdot m)$$

mit $k = \lfloor n/m \rfloor$, denn

$$n \bmod m = n - \lfloor n/m \rfloor \cdot m = n - k \cdot m \ .$$

- *„Soviel wie möglich auf einmal abziehen!“*

Anmerkungen zur Rekursion

- Rekursion oder Iteration? – *Welche Darstellung ist „besser"?*
  - Algorithmische Umsetzung und Lesbarkeit
  - Umsetzung des Java-Compilers und Effizienz/Einschränkungen
- Viele Algorithmen sind ihrer Natur nach rekursiv
  - Direkte Umsetzung als rekursiver Algorithmus,
    z.B. Rechenregeln für ggT $\Rightarrow$ rekursive Definition
  - Iterative Darstellung bedeutet „Mehrarbeit"
- Beschränkungen *v.a. in Java*(!)
  - Schleifen i.d.R. effizienter als Rekursion
    Das ist uns hier nicht wichtig!
  - Rekursionstiefe in der Praxis beschränkt! (in Java „*immer*")
- Nicht jede Rekursion lässt sich *so einfach* iterativ schreiben!
  - Nur *Endrekursion* z.B. ggT, *mehr dazu später*
  - Nicht z.B. Quicksort, Mergesort
  - Mehr dazu in *Algorithmen und Datenstrukturen*

Abschließendes Beispiel: Türme von Hanoi

```java
class Hanoi {

    // Solve Towers of Hanoi for n disks.
    public static void solve(int n) {
        move_towers(n,0,2,1);
    }
    // Move a single disk.
    public static void move_disk(int from,int to) {
        System.out.println("move disk "+from+" to "+to);
    }
    // Solve problem recursively.
    public static void move_towers(int n,int from,int to,int free) {
        if (n==1)
            move_disk(from,to);
        else {
            move_towers(n-1,from,free,to);
            move_disk(from,to);
            move_towers(n-1,free,to,from);
        }
    }

    public static void main(String args[]) {
        int n=Integer.parseInt(args[0]);
        solve(n);
    }
}
```

Abschließendes Beispiel: Türme von Hanoi

```
// Solve problem recursively.
public static void
move_towers(int n,int from,int to,int free) {
  if (n==1)
    move_disk(from,to);
  else {
    move_towers(n-1,from,free,to);
    move_disk(from,to);
    move_towers(n-1,free,to,from);
  }
}
```

- Berechnung der Lösung für n Scheiben:

  `move_towers(n,0,2,1);`

- Ausgabe: der Einzelschritte in `move_disk(from,to);`

Bemerkungen zur Rekursion

- Rekursion erlaubt oft intuitive Definition von Algorithmen
  - Viele Definitionen sind von Natur aus rekursiv.
  - *divide-and-conquer* Ansatz (z.B. in Quicksort)
- Rekursion kann Schleifen (Iteration) ersetzen
  - Beide Konzepte sind gleich mächtig.
  - Komplexere Beispiele folgen im Lauf der Vorlesung.
- Rekursion als ein Grundkonzept von *funktionalen* Programmiersprachen (z.B. Lisp, ML, Haskell,...)
  - Oft *keine* Schleifen!
  - Effiziente Umsetzung
- Rekursive Funktionen können auch in Java oft vorteilhaft sein!
  - *Rekursionstiefe* ist praktisch beschränkt, in Java auch für primitive Rekursion (`java.lang.StackOverflowError`)
  - Denn Funktionsaufruf benötigt Speicher für lokale Variablen und Rücksprungadresse.

Zusammenfassung

- Funktionen als Mittel zur Abstraktion
- Definition von Funktionen in Java
  - Signatur
  - Polymorphe Funktionen
  - *Call-by-value* und *call-by-reference*
- Rekursion
  - Anwendung desselben Prinzips auf Teilprobleme
  - „Rekursion = Iteration“
    - Beispiele: Fakultätsfunktion, Euklidscher Algorithmus ✓
    - Beispiel: Türme von Hanoi **?**
  - Rekursion → Iteration: benötigt *im allgemeinen* zusätzliche Datenstruktur (*stack*) — *später*

- Als nächstes: *Objektorientierte Programmierung*