

Kapitel 4: Methoden

- Definition
 - Methoden sind Anweisungsfolgen (Blöcke) mit Namen
- Überblick
 - Methoden ohne Parameter
 - Lokale Variablen: Lebensdauer und Sichtbarkeit
 - Methoden mit Parametern
 - Funktionen
 - Rekursive Methoden

Methoden: Einführung

- Motivation
 - Bisher: Ein Programm besteht aus einem großen Block (*main*).
 - Bessere Strukturierung wäre wünschenswert.
- Definition von Methoden
 - Methoden sind benannte Blöcke, d.h. Folgen von Anweisungen.
 - Aufruf einer Methode über ihren Namen (Bezeichner).
 - Methoden können Parameter haben; Verhalten hängt von Eingabe ab.
 - Methoden können ein Ergebnis zurückliefern (= Funktionen).
- Vorteile der Verwendung von Methoden
 - Strukturierung von Programmen, d.h. größere Übersichtlichkeit.
 - Wiederverwendung von mehrfach benutzten Programmteilen.
 - Definition benutzerspezifischer Operationen.

Methoden ohne Parameter

- Beispiel: Ausgabe einer Überschrift

```
class ArtikelListe {
    static void printHeader () {
        System.out.println ("Artikelliste");
        System.out.println ("=====");
    }

    public static void main (String[] arg) {
        ...
        printHeader ();
        ...
    }
}
```

Methodenkopf (Deklaration)
Methodenrumpf (Implementierung, Block)

ist auch eine Methode!
 Aufruf der obigen Methode

printHeader(): Methode hat keine Parameter
 void: Methode liefert keinen Wert zurück
 static: Methode wird statisch aufgerufen

Aufruf von Methoden

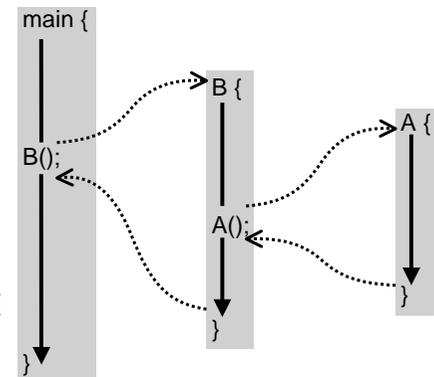
Beispiel

```
class MyProg {
    static void A () {
        ...
    }

    static void B () {
        ... A(); ...
    }

    public static void main (String[] arg) {
        ... B(); ...
    }
}
```

Ablauf



Lokale und globale Variablen

- Lokale Variablen
 - In Methoden (allg.: Blöcken) können Variablen lokal deklariert werden.
- Globale Variablen
 - Deklaration außerhalb von Methoden (d.h. auf Klassenebene).
 - Deklaration beginnt mit **static** (vorerst).

```
static void A() {
    int i, j;
    float x;
    ...
}
```

```
class MyProg {
    static int g;
    static float f;
    ...
    static void main(...) {...}
}
```

- Verwendung nur innerhalb der eigenen Methode möglich.
- Benutzung innerhalb aller Methoden möglich.

Lebensdauer von Variablen

- Reservierung des Speicherplatzes
 - globale Variablen werden statisch angelegt (einmal für Programmablauf).
 - lokale Variablen werden für jeden Methodenaufruf eigens angelegt.

	lokale Variablen	globale Variablen
angelegt	<i>dynamisch</i> : neu bei jedem Methodenaufruf	<i>statisch</i> : einmalig zu Programmbeginn
freigegeben	jeweils am Ende der Methode	am Programmende

- die Werte globaler Variablen bleiben also über Methodenaufrufe hinweg erhalten, die Werte lokaler Variablen gehen am Ende der zugehörigen Methode verloren.

Beispiel für Speicherbelegung

```
class MyProg {
    static int g;

    static void A () {
        int a;
        ... (3) ...
    }

    static void B () {
        int b;
        (2) ... A(); ... (4) ... A(); ... (5)
    }

    public static void main (String[] arg) {
        int m;
        (1) ... B(); ... (6)
    }
}
```

lokale Variablen (Kellerspeicher, Stapel, "Stack")

globale Variablen

Sichtbarkeitsbereich von Variablen

- Sichtbarkeitsbereich (Gültigkeitsbereich, *Scope*)
 - Programmstück, in dem auf eine Variable zugegriffen werden kann.
 - ab Deklaration bis zum Ende des Blocks, in dem Deklaration steht.
 - außerhalb dieses Blocks ist der Bezeichner nicht sichtbar.
 - Sichtbarkeitsbereich globaler Variablen wird unterbrochen durch Methoden mit gleichnamigen lokalen Variablen.
- Im Beispiel rechts:
 - lokales x verdeckt in P globales x.
 - Zugriff auf x in P betrifft lokales x.
 - globales x lebt noch, ist in P aber nicht sichtbar (*verschattet*).

Beispiel

```
class MyProg {
    static int x;
    static int y;

    static void P () {
        int x;
        while (...) {
            int z;
            ...
        }
        ...
    }
}
```

Lokalität von Bezeichnern

- Regeln
 - ein Name darf in einem Block nur einmal deklariert werden.
 - *Variablen* sind erst nach ihrer Deklaration sichtbar.
 - *Methoden* können auch schon vor Ihrer Deklaration verwendet werden.
- Vorteile lokaler Variablen
 - Übersichtlichkeit und Lesbarkeit: Deklaration und Benutzung nahe beieinander, in verschiedenen Methoden können Variablen mit gleichem Namen verwendet werden.
 - Sicherheit: andere Methoden können lokale Variablen nicht verändern.
 - Effizienz: Zugriff auf lokale Variablen ist meist schneller (Compiler!).

Funktionen

- Deklaration einer Funktion
 - Funktionen sind Methoden mit einem Rückgabewert.
 - Der Typ einer Funktion (Signatur) setzt sich aus den Typen der formalen Parameter sowie aus dem Typ des Rückgabewertes zusammen.
- Implementierung (Rumpf) und Verwendung
 - Implementierung: Rückgabe des Ergebnisses an den Aufrufer mit **return**.
 - Verwendung der Funktion: ein Funktionsaufruf ist ein Ausdruck.
z.B. `max(x,0) + y`
- Beispiel


```
static int max (int x, int y) {
    if (x > y) return x; else return y;
}
```

 - Signatur, `int × int → int`
 - Implementierung

Methoden mit Parametern

- Definition

Parameter (Argumente) sind Werte, die beim Aufruf an eine Methode übergeben werden.
- Beispiel Deklaration


```
static void printMax (int x, int y) {
    if (x > y) System.out.print (x);
    else System.out.print (y);
}
```

Verwendung (Aufruf)

```
printMax (100, 2*i);
```
- Formale Parameter
 - gehören zur Methodendeklaration, stehen im Methodenkopf.
 - sind lokale Variablen.
- Aktuelle Parameter
 - werden beim Aufruf angegeben.
 - sind Ausdrücke (Werte).
- Parameterübergabe
 - Werte aktuelle Parameter aus.
 - Weise Werte an entsprechende formale Parameter zu.
 - durch Zuweisung entsteht im formalen Parameter eine Kopie des aktuellen Parameters

Beispiel: Zweierlogarithmus

- Definition (Deklaration und Implementierung)


```
/** logarithmus dualis of x for x > 0 */
static int log2 (int x) { // assume: x > 0
    int res = 0;
    while (x > 1) {
        x = x / 2;
        res++;
    }
    return res;
}
```

Funktion `int → int`

x	res
17	0
8	1
4	2
2	3
1	4
- Verwendung


```
int ld17 = log2(17); // ld17 == 4
```

Prozeduren

- Definition
 - Methoden ohne Rückgabewert (Typ *void*) heißen *Prozeduren*.
 - *void*: leerer Typ, d.h. leerer Wertebereich, keine Operationen.
 - Verlassen einer Prozedur mit **return** (ohne Wert) möglich (vgl. **break**):
`static void P () { ... if (...) return; ... }`
- Seiteneffekte
 - Prozeduren liefern kein Ergebnis, sondern verändern Systemzustand.
 - *Seiteneffekte*: Wirkungen, die nicht der Signatur zu entnehmen sind.
 - Ausgabe von Ergebnissen auf die Standardausgabe (System.out).
 - Veränderung globaler Variablen → möglichst vermeiden!
 - Seiteneffekte tragen oft zur Unübersichtlichkeit von Programmen bei.
 - Funktionen liefern ein Ergebnis und sollten keine Seiteneffekte haben.

Beispiel: Euklidischer Algorithmus

```
/** Größter gemeinsamer Teiler */
static int ggt (int x, int y) {
    int rest = x % y;
    while (rest != 0) {
        x = y; y = rest; rest = x % y;
    }
    return y;
}
```

```
/** Kürzen eines Bruchs */
static void reduce (int numerator, int denominator) {
    int x = ggt (numerator, denominator);
    System.out.println (numerator/x + "/" + denominator/x);
}
```

Überladen von Methoden

- Konzept des Überladens
 - Verwendung desselben Bezeichners (Namens) bei verschiedenen Methoden.
 - Unterschiedliche Parameterlisten (Argumente).
 - Nur unterschiedlicher Ergebnistyp reicht nicht aus.
 - Beim Aufruf wird Methode mit passenden Parametern gewählt.
 - Wichtig: nur bei gleichartiger Semantik verwenden.
- Beispiel: Methode println

- void println ();	- void println (int);
- void println (boolean);	- void println (long);
- void println (char);	- void println (float);
- void println (String);	- void println (double);

System.out.println (5);	→ void println (int)
System.out.println (5.0);	→ void println (double)
System.out.println ('5');	→ void println (char)

Beispiel: Berechnung von Potenzen

<pre>/** berechne base ^ exp */ static long power (int base, int exp) { long result = 1; for (int i = 1; i <= exp; i++) { result = result * base; } return result; }</pre>	<pre>/** Potenziere base ^ exp effizienter */ static long power (int base, int exp) { long result = 1; while (exp > 0) { if (exp % 2 == 0) { base = base * base; exp = exp / 2; } else { // exp is odd result = result * base; exp = exp - 1; } } return result; }</pre>
---	---

Beobachtung:

$b^e = b^{2^e/2} = (b^2)^{e/2}$
 d.h. für gerade Exponenten e:
 quadriere Basis b und halbiere e.

Rekursive Methoden

- Begriff
 - Methoden können sich selbst aufrufen
 - direkt rekursiv: $m() \rightarrow m()$
 - indirekt rekursiv: $m() \rightarrow n() \rightarrow m()$

Beispiel zur Fakultät

```

factorial (4) =
factorial (3) * 4 =
(factorial (2) * 3) * 4 =
(((factorial (1) * 2) * 3) * 4) =
((((factorial (0) * 1) * 2) * 3) * 4) =
(((1 * 1) * 2) * 3) * 4 =
(((1 * 2) * 3) * 4) =
(2 * 3) * 4 =
6 * 4 =
24
    
```

- Beispiel: Fakultätsfunktion
 - Def: $n! = 1 * 2 * \dots * (n-1) * n$;
 - also: $n! = (n-1)! * n$; sei $0! = 1$

```

static long factorial (long n) {
    if (n == 0) return 1;
    else return factorial (n - 1) * n;
}
    
```

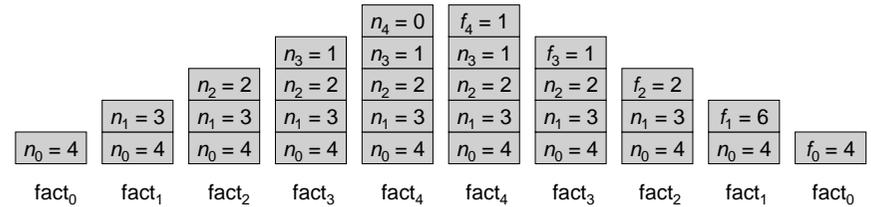
Frage:
es gibt nur eine lokale Variable n ,
wo die vielen n_i merken?

Aufrufkeller für lokale Variablen

Beispiel: factorial (4)

Wie oben:

- Parameter sind lokale Variablen.
- Für jeden Methodenaufruf werden eigene lokale Variablen angelegt.



Rekursion allgemein

- Allgemeines Schema
 - if (Problem klein genug)
 - nicht-rekursiver Zweig; // Terminierungsfall
 - else
 - rekursiver Zweig mit kleinerem Problem;
- Nachweis der Terminierung
 - Es gibt einen Terminierungszeitpunkt.
 - Das Problem wird bei jedem rekursiven Aufruf signifikant „kleiner“, d.h. der Abstand zum Terminierungsfall wird kleiner.
 - Bsp. Fakultät:
 - Terminierungsfall: $n == 0$
 - Verkleinerung: Aufruf mit $n - 1 < n$

Rekursion vs. Iteration

- Rekursive Lösungen
 - Rekursion ist mächtiges, allgemeines Programmierprinzip.
 - Jede rekursive Lösung lässt sich auch iterativ (d.h. mit Schleifen) lösen:
 - Beispiel: Factorial (n) == ProduktBis (n)
- Vergleich
 - Rekursive Formulierung ist oft eleganter.
 - Iterative Lösung ist oft effizienter aber komplizierter.
- Beispiel: effiziente Potenzierung $base \wedge exp$

```

static long power (int base, int exp) {
    if (exp == 0) return 1; // b^0 = 1
    else if (exp % 2 == 0) return power (base * base, exp / 2); // b^e = b^{2 * e/2}
    else return base * power (base, exp - 1); // b^e = b * b^{e-1}
}
    
```

Entrekursivierung: Beispiel Fakultät

- Unmittelbare rekursive Lösung


```
static long factorial (long n) {
    if (n == 0) return 1;
    else return n * factorial (n - 1);
}
```
- Problem
 - Nachklappern der Multiplikationen.
 - Idee der endständigen Rekursion:
 - führe Ergebnis rekursiv mit.
 - n zählt Schritte bis zum Ende.
 - Rekursion ohne Nachklappern!
 - alte Werte auf dem Stapel werden nicht mehr benötigt.
 - einfache Überführung in Iteration.
- Endständig-rekursive Lösung


```
static long factorial (long n) {
    return fact1 (n, 1);
}
static long fact1 (long n, long res) {
    if (n == 0) return res;
    else return fact1 (n - 1, res * n);
}
```
- Iterative Lösung


```
static long factorial (long n) {
    long result = 1;
    while (n > 0) {
        result = result * n;
        n = n - 1;
    }
    return result;
}
```

Beispiel: Fibonacci-Zahlen

- Definition


```
fib(0) = fib(1) = 1;
fib(n) = fib(n - 1) + fib(n - 2) für n > 1
```

Resultierende Folge:
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- Problem

Explosion der Aufrufe (= 2^n)
- Unmittelbare rekursive Lösung


```
static long fib (long n) {
    if (n == 0 || n == 1) return 1;
    else return fib(n - 1) + fib(n - 2);
}
```

Fibonacci-Zahlen iterativ

- Endständige Rekursion
 - übergebe Ergebnis in Parametern.
 - n zählt Schritte bis zum Ende.

```
static long fib (int n) {
    return fib1 (n, 1, 0);
}
static long fib1 (int n, long res, long p) {
    if (n == 0) return res;
    else return fib1 (n - 1, res + p, res);
}
```

Einfache Umwandlung in Schleife mit Zuweisung $(n, res, p) \leftarrow (n-1, res+p, res)$:
Jetzt nur mehr linear viele Aufrufe.
- Iterative Lösung


```
static long fib (int n) {
    long result = 1, previous = 0;
    while (n > 0) {
        long pprev = previous;
        previous = result;
        result = result + pprev;
        n--;
    }
    return result;
}
```

Vergleich zur Rekursion:
= nur lineare Anzahl Additionen in n .
+ keine sich stapelnden Variablen.