Methoden

Parameterlose Methoden

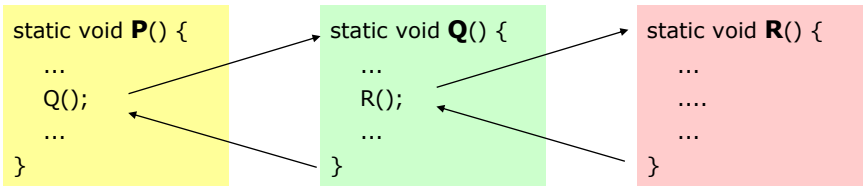
Beispiel: Ausgabe einer Überschrift

```
class Sample {

    static void printHeader() {          // Methodenkopf
      System.out. println("Liste");      // Methodenrumpf
      System.out. println("--------");
    }

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

Zweck von Methoden

- Wiederverwendung häufig benutzten Codes.
- Definition benutzer-spezifischer Operationen.
- Strukturierung des Programms.

Wie funktioniert ein Methodenaufruf?



Namenskonventionen für Methoden:

Namen sollten mit Verb und Kleinbuchstaben beginnen

Beispiele: *printHeader, findMaximum, traverseList, ...*

Parameter

Werte, die vom Rufer an die Methode übergeben werden

```
class Sample {
    static void printMax(int x, int y) {
        if (x > y)      System.out.print( x);
        else            System.out.print( y);
    }
    public static void main (String[] arg) {
        ...
        printMax( 100, i * 2 );
    }
}
```

formale Parameter
- im Methodenkopf (hier x, y)
- sind Variablen der Methode

aktuelle Parameter
- an der Aufrufstelle (hier 100, 2* i)
- können Ausdrücke sein

Parameterübergabe

Aktuelle Parameter werden den entsprechenden formalen Parametern zugewiesen.

x = 100; y = 2 * i;

Aktuelle Parameter müssen mit formalen Parametern zuweisungskompatibel sein.

Formale Parameter enthalten Kopien der aktuellen Parameter,

Funktionen

Methoden, die Ergebniswerte an den Rufer liefern.

```
class Sample {

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

    public static void main (String[] arg) {
        ...
        System.out.println( max( 100, i +j ) + 1);
    }
}
```

- Haben Funktionstyp (z. B. *int*) statt void (= kein Typ).
- Liefern Ergebnis mittels return Anweisung an den Rufer.
  (x muß zuweisungskompatibel mit int sein)
- Werden wie Operanden in einem Ausdruck benutzt.

Funktionen Methoden mit Rückgabewert
Prozeduren Methoden ohne Rückgabewert

Weiteres Beispiel

Ganzzahliger Zweierlogarithmus

```
class Sample {

    static int log2 (int x) {     // assert: x >= 0
        int res = 0;
        while (x > 1) {x = x / 2; res++;}
        return res;
    }

    public static void main (String[] arg) {
        int x = log2(17) ;        // x == 4
        .....
    }
}
```

Return in Prozeduren

```
class ReturnDemo {

   static void printLog2 (int x) {
      if (x < 0) return;              // kehrt zum Rufer zurück
      int res = 0;
      while (x > 1) {x = x / 2; res++;}
      System.out.println( res);
   }

   public static void main (String[] arg) {
      int x = In. readInt();
      printLog2( x);
       ...
      if (! In. done()) return;       // beendet das Programm
}
}
```

Funktionen	müssen mit return beendet werden.
Prozeduren	können mit return beendet werden.

Lokale und statische Variablen

```
class C {
   static int a, b;
   static void P() {
      int x, y;
      .......
   }
   ...
}
```

Statische Variablen
auf Klassenebene mit ***static*** deklariert;
auch in Methoden dieser Klasse sichtbar.

Lokale Variablen
in einer Methode deklariert
(lokal zu dieser Methode; nur dort sichtbar).

Reservieren und Freigeben von Speicherplatz

Statische Variablen
am Programmbeginn angelegt,
am Programmende wieder freigegeben.

Lokale Variablen
bei jedem Aufruf der Methode neu angelegt,
am Ende der Methode wieder freigegeben.

Beispiel: Summe einer Zahlenfolge

falsch!

```
class Wrong {

    static void add (int x) {
        int sum = 0;
        sum = sum + x;
    }

    public static void main( String[] arg) {
        add( 1); add( 2); add( 3);
        Out. println(" sum = " + sum);
    }
}
```

richtig!

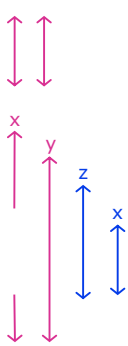
```
class Correct {

    static int sum = 0;

    static void add (int x) {
        sum = sum + x;
    }

    public static void main( String[] arg) {
        add( 1); add( 2); add( 3);
        Out. println(" sum = " + sum);
    }
}
```

- *sum* ist in *main* nicht sichtbar.
- *sum* wird bei jedem Aufruf von *add* neu angelegt (alter Wert geht verloren).

Sichtbarkeitsbereich von Namen

Programmstück, in dem auf diesen Namen zugegriffen werden kann
(auch *Gültigkeitsbereich* oder *Scope* des Namens genannt)

```
class Sample {

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



Regeln

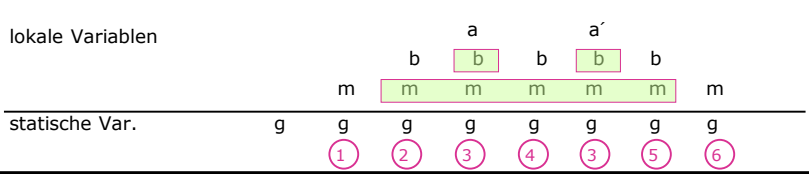
- Ein Name darf in einem Block nicht mehrmals deklariert werden (auch nicht in geschachtelten Anweisungsblöcken).
- Der Sichtbarkeitsbereich eines Namens beginnt bei seiner Deklaration und geht bis zum Ende seines Blocks.
- Lokale Namen verdecken Namen, die auf Klassenebene deklariert sind.
- Auf Klassenebene deklarierte Namen sind in allen Methoden der Klasse sichtbar.

Beispiel zu Sichtbarkeitsregeln

```
class Sample {
   static void P() {
      System.out. println( x);              // gibt 0 aus
   }
   static int x = 0;
   public static void main( String[] arg) {
      System.out. println( x);              // gibt 0 aus
      int x = 1;                            // verdeckt statisches x
      System.out. println( x);              // gibt 1 aus
      P();                                  // gibt 0 aus
      if (x > 0) {
         int x;                             // Fehler: x ist in main bereits deklariert
         int y;
      } else {
         int y;                             // ok, kein Konflikt mit y im then- Zweig
      }
      for (int i = 0; ...) {...}
      for (int i = 1; ...) {...}            // ok, kein Konflikt mit i aus letzter Schleife
   }
}
```

Lebensdauer von Variablen

```
class LifenessDemo {
  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
  }
}
```



Lokalität

Variablen möglichst lokal deklarieren, nicht als statische Variablen.

Vorteile

- Übersichtlichkeit
  Deklaration und Benutzung nahe beisammen.
- Sicherheit
  Lokale Variablen können nicht durch andere Methoden zerstört werden.
- Effizienz
  Zugriff auf lokale Variable ist oft schneller als auf statische Variable.

Überladen von Methoden

Methoden mit gleichem Namen aber verschiedenen Parameterlisten können in derselben Klasse deklariert werden.

```
static void write (int i) {...}
static void write (float f) {...}
static void write (int i, int width) {...}
```

Beim Aufruf wird diejenige Methode gewählt, die am besten zu den aktuellen Parametern paßt.

```
write( 100);          write (int i)
write( 3.14f);        write (float f)
write( 100, 5);       write (int i, int width)
short s = 17;
write( s);            write (int i);
```

Rekursion

Was heißt "rekursiv"

Eine Methode *m()* heißt rekursiv, wenn sie sich selbst aufruft

```
m() {  m();          }      direkt rekursiv
m() {  n()  { m();   } }    indirekt rekursiv
```

Beispiel: Berechnung der Fakultät (n!)

$$n! = \underbrace{1 * 2 * 3 * \ldots * (n-1)}_{(n-1)!} * n$$

rekursive Definition

$n! = (n-1)! * n$

$1! = 1$

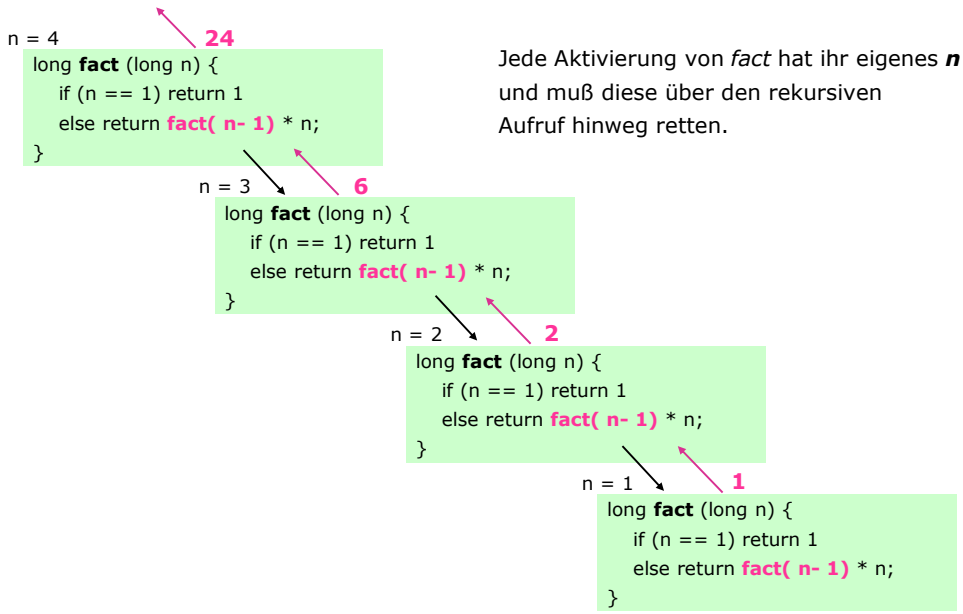
Rekursive Methode zur Berechnung der Fakultät

```
long fact (long n) {
    if (n == 1)
       return 1;
    else
    return fact( n- 1) * n;
}
```

allgemeines Muster

if (*Problem klein genug*)
  nichtrekursiver Zweig;
else
  rekursiver Zweig

Ablauf einer rekursiven Methode



Jede Aktivierung von *fact* hat ihr eigenes ***n*** und muß diese über den rekursiven Aufruf hinweg retten.

Rekursiv und iterativ

rekursiv

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

iterativ

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

Jeder rekursive Algorithmus kann auch iterativ programmiert werden

- rekursiv: meist kürzerer Quellcode
- iterativ: meist kürzere Laufzeit

Rekursion bei rekursiven Datenstrukturen nützlich (Bäume, Graphen, ...)