Softwareentwicklung 1 - gtec

Methoden

Parameterlose Methoden

Parameterlose Methoden für

- Wiederverwendung häufig benutzten Codes
- Definition benutzerspezifischer Operationen
- Strukturierung des Programms

Beispiel: Ausgabe einer Überschrift

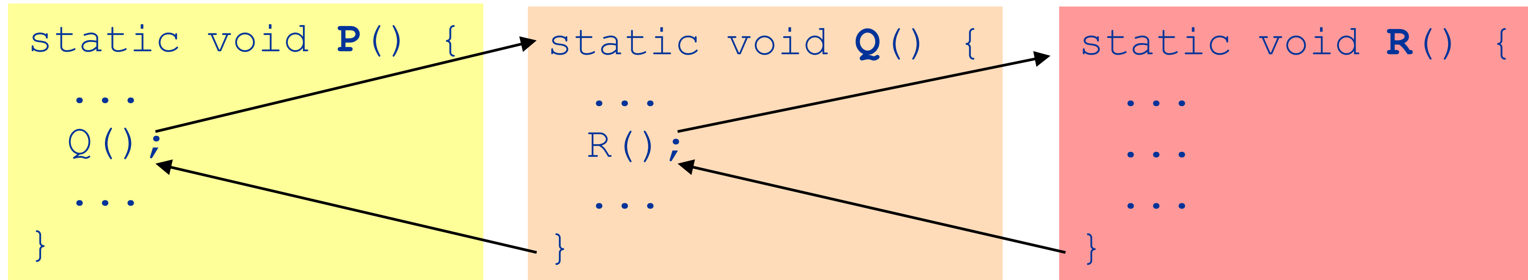
```
class Sample {

  static void printHeader() {       // Methodenkopf
    Out.println("Artikelliste");    // Methodenrumpf
    Out.println("------------");
  }

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

Wie funktioniert ein Methodenaufruf?

Bei Methodenaufruf erfolgt ein Sprung zum Anfang der Methode

Rumpf der Methode wird ausgeführt

Nach der Ausführung des Rumpfs erfolgt ein Rücksprung zum aufrufenden Programm



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) Out.print(x); else Out.print(y);
  }

  public static void main (String[] arg) {
    ...
    printMax(100, 2 * i);
  }
}
```

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 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) {
    ...
    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

Beispiel: Maximum von 2 Zahlen

Programm soll 2 Zahlen einlesen und bestimmen, welche die größere ist

```
public class Max3 {
    public static void main (String[] args) {
        int nr1 = readNr(1);
        int nr2 = readNr(2);
        int maxNr = max(nr1, nr2);
        putOutMax(maxNr);
    }

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

    static int readNr(int i) {
        Out.println("" + i + ". Zahl eingeben: ");
        int nr = In.readInt();
        return nr;
    }

    static void putOutMax(int max) {
        Out.println("Größte Zahl ist: " + max);
    }
}
```

Beispiel: Zweierlogarithmus

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
    ...
  }
}
```

Beispiele

Größter gemeinsamer Teiler nach Euklid

```
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 z, int n) {
  int x = ggt(z, n);
  Out.print(z/x); Out.print("/"); Out.print(n/x);
}
```

Beispiele

Prüfe, ob x eine Primzahl ist

```
static boolean isPrime (int x) {
   if (x ==1 || x == 2) return true;
   if (x % 2 == 0) return false;
   int i = 3;
   while (i * i <= x) {
      if (x % i == 0)
         return false;
      i = i + 2;
   }
   // i > √x  und x ist durch keine Zahl
   // kleiner als i teilbar
   return true;
}
```

Alternative

```
for ( int i = 3;
      i * i <= x;
      i += 2 )
   if (x % i == 0)
      return false;
```

Beispiele

Berechne val^{exp}

```
static long power (int val, int exp) {
  long res = 1;
  for (int i = 1; i <= exp; i++) res = res * val;
  return res;
}
```

Dasselbe effizienter

```
static long power (int val, int exp) {
  long res = 1;
  while (exp > 0) {
    if (exp % 2 == 0) {
      val = val * val; exp = exp / 2;   // x^2n = (x*x)^n
    } else {
      res = res * val; exp--;   // x^n+1 = x*x^n
    }
  }
  return res;
}
```

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++;}
    Out.println(res);
  }

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

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

Sichtbarkeit von Variablen

Sichtbarkeit

= Der Bereich eines Programms, in dem auf die Variable zugegriffen werden kann

***Lokale Variable*:**

Block (`{ ... }`), in dem sie deklariert wurde,
von ihrer Deklaration bis zum Ende des Blocks

Globale Variable (Klassenvariable, static) und Konstante:

gesamtes Programm (gesamte Klasse)

Parameter:

gesamter Rumpf der Methode

Beispiel zu Sichtbarkeitsregeln

```
class Sample {
  static void P() {
    Out.println(x);                  // gibt 0 aus
  }
  static int x = 0;
  public static void main(String[] arg) {
    Out.println(x);                  // gibt 0 aus
    int x = 1;                       // verdeckt statisches x
    Out.println(x);                  // gibt 1 aus
    P();
    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
  }
}
```

Beispiel static Variabl

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)

Lebensdauer von Variablen

Lebensdauer

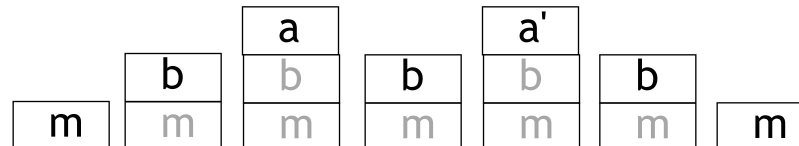
= Dauer der Existenz einer Variablen

- Lokale Variablen leben vom Eintritt in den Block, in dem sie deklariert wurden, bis zum Austritt aus dem Block
- Globale (**static**) Variablen leben vom Start des Programms bis zum Ende des Programms
- Parameter leben vom Anfang der Prozedur bis zum Ende der Prozedur

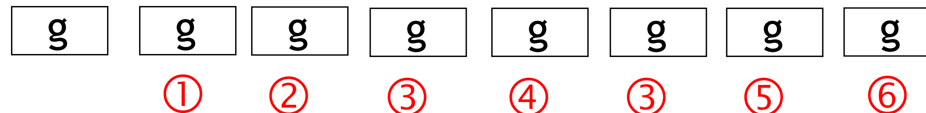
Lebensdauer von Variablen

```
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...
    }
}
```

lokale Variablen (Methodenkeller)



statische Var.



Lebensdauer / Sichtbarkeit bei Methoden

Parameter und lokale Variablen in Methoden existieren und sind sichtbar im Rumpf der Methode

```
public class Max3 {
    public static void main (String[] args) {
        int nr1 = readNr(1);
        int nr2 = readNr(2);
        int maxNr = max(nr1, nr2);
        putOutMax(maxNr);
    }
    static int max(int x, int y) {
        if (x > y) {
            return x;
        } else {
            return y;
        }
    }
    static int readNr(int i) {
        TextIO.put("" + i + ". Zahl eingeben: ");
        int nr = TextIO.getlnInt();
        return nr;
    }
    static void putOutMax(int max) {
        TextIO.putln("Größte Zahl ist: " + max);
    }
}
```

- `args`, `nr1`, `nr2` `maxNr` existieren und sind sichtbar in `main`
- `x`, `y` existieren und sind sichtbar in `max`
- `i` und `nr` existieren und sind sichtbar in `readNr`
- `max` existiert und ist sichtbar in `putOutMax`

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

Deklaration von Konstanten

Konstante sind Variable, die nicht geändert werden können

müssen am Anfang des Programms als static final (= im ganzen Programm bekannt) deklariert werden (siehe später)

müssen bei der Deklaration initialisiert werden

Schreibweise in Großbuchstaben

```
public class ConstantDeclaration {
  static final int FACTOR = 10;
              // Integer-Konstante factor mit Wert 10
  static final boolean NO = false;
              // Boolean-Konstante no mit Wert false

  public static void main (String[] arg) {
      …
  }
```

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);
```