Programmierkurs Java

UE 6 - Funktionen

Dr.-Ing. Dietrich Boles

Gliederung

- Prozeduren
- Funktionen
- Lokale Variablen
- Parameter
- Gültigkeitsbereich
- Eigenschaften
- Funktionsbibliotheken
- Prozedurale Zerlegung
- Vorteile
- Rekursion
- Zusammenfassung

Definition:

- Teil eines Programmes, das eine in sich abgeschlossene Aufgabe löst.
- „Unterprogramm“

Java-spezifisch: Prozeduren sind so genannte *Methoden*

zwei Aspekte:

- Prozedurdefinition: Vereinbarung eines neuen „Befehls“
- Prozeduraufruf: Ausführung des neuen Befehls

Prozeduren / Beispiel

```
public class Beispiel {

    public static void main(String[] args) {
        System.out.println("Start");
        tuWas();
        System.out.println("zwischendurch");
        tuWas();
        System.out.println("Ende");
    }

    static void tuWas() {
        int zahl = IO.readInt("Zahl:");
        System.out.println(zahl * zahl);
    }

}
```

Prozeduren / Definition

```
<Proc-Def>    ::= <Proc-Kopf> <Proc-Rumpf>
<Proc-Kopf>   ::= ["public"] "static" "void" <Proc-Name>
                  "(" [ <Param-Defs> ] ")"
<Proc-Name>   ::= <Bezeichner>
<Proc-Rumpf>  ::= <Block>
<Param-Defs>   später
```

Anmerkungen:

- keine Auswirkungen auf den Programmablauf
- führt neue Prozedur mit dem angegebenen Namen ein
- In einer Prozedur definierte Variablen sind in anderen Prozeduren nicht zugreifbar (→ Gültigkeitsbereich)!

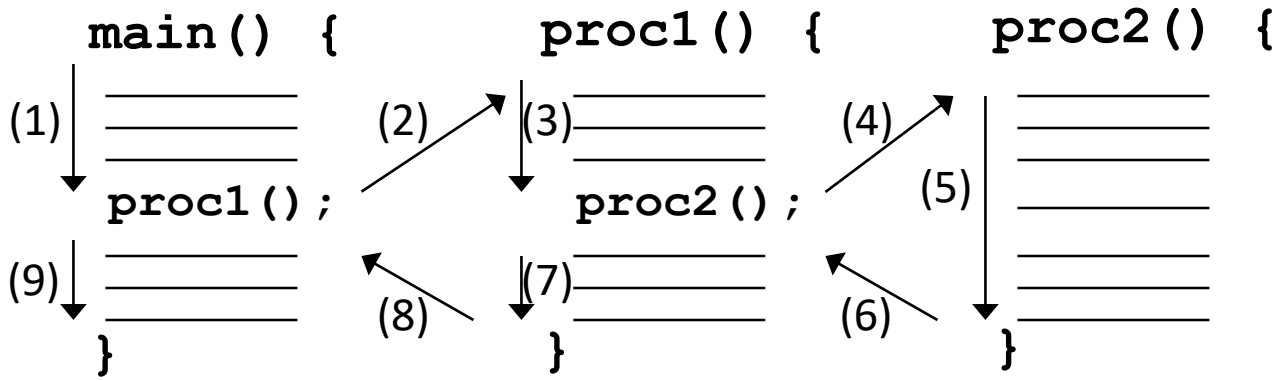
Ort im Programm:

- innerhalb einer Klasse
- vor/nach main-Prozedur
- keine Schachtelung von Prozedurdefinitionen möglich

```
<Proc-Aufruf> ::= <Proc-Name> "(" [ <Param-List> ] ")" ";"
<Param-List>   später
```

Nebenbedingungen / Semantik:

- Prozedur mit entsprechender Signatur (Name, Parameter, …) muss definiert sein
- Prozeduraufruf ist eine **Anweisung**
- beim Aufruf einer Prozedur werden die Anweisungen im Prozedurrumpf ausgeführt bis der Prozedurrumpf vollständig abgearbeitet ist oder bis eine return-Anweisung ausgeführt wird



Definition:

- Teil eines Programmes, das durch die Ausführung von Anweisungen einen Wert berechnet

Abgrenzung:

- "Prozeduren tun etwas"
- "Funktionen berechnen und liefern einen Wert"

Mathematik: $f : \text{Definitionsbereich} \rightarrow \text{Wertebereich}$

Funktionsname — Parameter — Funktionstyp

Drei Aspekte:

- Wertberechnung und -rückgabe
- Funktionsdefinition
- Funktionsaufruf

Funktionen / Definition (1)

```
<Funk-Def>    ::= <Funk-Kopf> <Funk-Rumpf>
<Funk-Kopf>   ::= ["public"] "static" <Typ> <Funk-Name>
                  "(" [ <Param-Defs> ] ")"
<Funk-Name>   ::= <Bezeichner>
<Funk-Rumpf>  ::= <Block>
```

Funktionstyp

Nebenbedingung:

- im Rumpf muss es **in jedem möglichen** Funktionsdurchlauf eine return-Anweisung geben, wobei der Typ des return-Ausdrucks konform zum Funktionstyp ist

Anmerkungen:

- keine Auswirkungen auf den Programmablauf
- führt neue Funktion mit dem angegebenen Namen und Funktionstyp ein

Ort im Programm:

- innerhalb einer Klasse
- vor/nach main-Prozedur
- keine Schachtelung von Funktionsdefinitionen möglich

Beispiel:

```
class FunkProbe {

  static int funkEins() {
      IO.println("in funkEins");
      return 1;
  }

  public static void main(String[] args) { ...  }

  static char funkZwei() {
      if ( IO.readInt() == 0 )
          return 'a';
      IO.println("in funkZwei");
      return 'b';
  }
}
```

Fehlerhafte Beispiele:

```
class FunkProbe2 {
  static int liefereWert() {
    if ( IO.readInt() == 0 ) return -2;
    // Fehler: im else-Fall wird kein return ausgeführt!
  }
  static double funkDrei() {
    if (IO.readInt() == 0)
      return 'a';        // ok: impliziter Typcast!
    IO.println("in funkDrei");
    return 'a' == 'b';   // Fehler: ungültiger Typ!
  }
  static boolean test() {
    return 2 == 0;
    int wert = 2; // Fehler: Anweisung wird nicht erreicht!
} }
```

Funktionen / Aufruf (1)

```
<Funk-Aufruf> ::= <Funk-Name> "(" [ <Param-List> ] ")"
```

Nebenbedingungen / Semantik:

- Funktion mit entsprechender Signatur muss definiert sein
- Funktionsaufruf ist ein **Ausdruck**
- beim Aufruf einer Funktion werden die Anweisungen im Funktionsrumpf ausgeführt, bis eine return-Anweisung ausgeführt wird
- nach der Berechnung des Wertes des return-Ausdrucks der return-Anweisung wird die Funktion verlassen und der Wert als Funktionswert zurückgeliefert

```
class FunkProbe {
    static int funk() {
        int zahl = IO.readInt("Zahl:");
        return 2*zahl;
    }
    public static void main(String[] args) {
        IO.println(funk() * 4 * funk());
}   }
```

```
<Funk-Anweisung> ::= <Funk-Aufruf> ";"
```

- **Semantik**:
  - Der von der Funktion berechnete und gelieferte Wert wird ignoriert

Beispiel:

```
class FunkProbe {
    static int funk() {
        int zahl = IO.readInt("Zahl:");
        return 2*zahl;
    }
    public static void main(String[] args) {
        funk();
        IO.println(funk() * 4 * funk());
    }
}
```

- Prozedur-/Funktionsdefinition:
  - Funktionsname: Anfangsbuchstabe klein; Anfangsbuchstaben neuer Wortbestandteile groß
  - Funktionsname: aussagekräftig !!!!!
  - Funktionskopf und `{` in eine Zeile
  - `}` unterhalb des `v` von `void`
  - Vor `()` kein Leerzeichen
  - Hinter `()` ein Leerzeichen
  - innere Anweisungen um 4 Spalten einrücken
  - zwei Funktionsdefinitionen durch Leerzeile trennen
- Prozedur-/Funktionsaufruf:
  - Vor `()` kein Leerzeichen

Lokale Variablen

- **Lokale Variable**: In einem Prozedur- oder Funktionsrumpf definierte Variable
- *Gültig* innerhalb des Rumpfes
- *Lebendig* während der Prozedur- bzw. Funktionsausführung

```
public static void main(String[] args) {
    int zahl = IO.readInt();
    proz();
    zahl = zahl * zahl;
    System.out.println(zahl);
}

static void proz() {
    int zahl = IO.readInt();
    zahl = 2 * zahl;
    System.out.println(zahl);
}
```

Parameter / Motivation (1)

- Parameter erhöhen die Flexibilität
- Beispiel:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} \quad \text{für} \quad 0 <= m <= n$$

$$\binom{6}{3} = \frac{6!}{3!(6-3)!}$$

- benötigt wird

```
static int fak6() {...}
static int fak3() {...}
```

- gewünscht:

```
static int fakn() {...}
```

wobei der Wert n erst zur Laufzeit angegeben werden muss

```
class ParameterMotivation {
   static int fak3() {
     int zahl = 3; int erg = 1;
     for(int zaehler=2; zaehler<=zahl; zaehler++)
       erg = erg * zaehler;
     return erg;
   }
   static int fak6() {
     int zahl = 6; int erg = 1;
     for(int zaehler=2; zaehler<=zahl; zaehler++)
       erg = erg * zaehler;
     return erg;
   }
   public static void main(String[] args) {
     int ueber_6_3 = fak6() / (fak3() * fak3());
 } }
```

Parameter / Motivation (3)

```
class ParameterMotivation {

  static int fak(int zahl) {
    int erg = 1;
    for(int zaehler=2; zaehler<=zahl; zaehler++)
      erg = erg * zaehler;
    return erg;
  }

  public static void main(String[] args) {
    int ueber_6_3 = fak(6) / (fak(3) * fak(3));
  }

}
```

Parameter / Motivation (4)

```
class ParameterMotivation {
  static int fak(int zahl) {
    int erg = 1;
    for(int zaehler=2; zaehler<=zahl; zaehler++)
      erg = erg * zaehler;
    return erg;
  }
  static int ueber(int n, int m) {
    if ((0 <= m) && (m <= n))
      return fak(n) / (fak(m) * fak(n-m));
    return -1; // Fehlerfall (spaeter Exceptions)
  }
  public static void main(String[] args) {
    int ueber_6_3 = ueber(6, 3);
    int ueber_7_4 = ueber(7, 4);
} }
```

```
<Param-Defs> ::= <Typ> <Param-Name>
                 { "," <Typ> <Param-Name> }
<Param-Name> ::= <Bezeichner>
```

Semantik:

- Einführung einer lokalen Variable für die Prozedur/Funktion
- Gültigkeitsbereich ist der Prozedur/Funktionsblock
- Bezeichnung: "formaler Parameter"

Beispiele:

```
static int summe(int op1, int op2) {
    return op1 + op2;
}
static void ausgabe(char zeichen, int anzahl) {
    for (int i = 0; i < anzahl; i++)
        IO.print(zeichen);
}
```

Parameter / Übergabe (1)

```
<Param-List> ::= <Ausdruck> { "," <Ausdruck> }
```

Semantik:

- Übergabe eines "Initialwertes" für den formalen Parameter
- Bezeichnung: "aktueller Parameter"

Bedingungen:

- Anzahl an formalen Parametern = Anzahl an aktuellen Parametern
- für alle Parameter in der entsprechenden Reihenfolge: Typ des aktuellen Parameters typkonform zum Typ des formalen Parameters

Schema:

```
static int summe(int op1, int op2) {
  // int op1 = "Wert des ersten aktuellen Parameters";
  // int op2 = "Wert des zweiten aktuellen Parameters";
  return op1 + op2;
}
public static void main(String[] args) {
  int s1 = summe(2, 3*4);
  int s2 = summe(s1, summe(5, 8));
}
```

Parameter / Übergabe (2)

```
class Beispiel {

  static int funk(int n1, int n2, int n3, int n4) {
    return n1 / n2 + n3 * n4;
  }

  public static void main(String[] args) {
    int n = 2;
    int zahl = funk(n++, n = n+2, --n, n);
    // entspricht hier: funk(2, 5, 4, 4)
    // Grund: Parameterauswertung von links nach rechts
  }
}
```

Gültigkeitsbereich / Überladen von Funktionen (1)

- **Gültigkeitsbereich ("Scope"):**
  - Gültigkeitsbereich = Orte im Programm, an denen die Funktion benutzt werden kann
  - der Gültigkeitsbereich eines Funktionsnamens ist die gesamte Klasse, in der die Funktion definiert wird
  - Funktionsnamen müssen innerhalb eines Blocks und aller inneren Blöcke eindeutig sein; Ausnahme: Überladen von Funktionen!
  - Funktionsnamen sind nur innerhalb ihres Gültigkeitsbereichs anwendbar

- **Überladen von Funktionen:**
  - zwei oder mehrere Funktionen können innerhalb eines Gültigkeitsbereichs denselben Namen besitzen, wenn
    - sie eine unterschiedliche Anzahl an Parametern besitzen oder
    - wenn sich die Parametertypen an entsprechender Stelle unterscheiden

- die Definition wird später noch erweitert

Gültigkeitsbereich / Überladen von Funktionen (2)

```
class Beispiel {
  static float div(float op1, float op2) {
    return op1 / op2;
  }
  static int div(int op1, int op2) {
    return op1 / op2;
  }
  static float div(int op1, int op2) // Fehler!

  public static void main(String[] args) {
    int z1    = div(2, 3);         // int-div
    float z2  = div(2.0F, 3.0F);   // float-div
    float z3  = div(2, 3);         // int-div
    float z4  = div(2.0F, 3);      // float-div
  }
}
```

Funktionen / Eigenschaften in Java

- Parameter sind spezielle funktionslokale Variablen
- Formale Parameter werden zur Laufzeit mit aktuellen Parameter(werten) initialisiert
- nur Werteparameter (call-by-value)
- die Typen der Parameter müssen bei der Funktionsdefinition festgelegt werden (keine flexiblen Parametertypen; später `Object`)
- als Parametertypen können (bisher nur) Standarddatentypen verwendet werden (später auch Klassentypen)
- der Funktionstyp muss bei der Definition angegeben werden (keine flexiblen Funktionstypen; später generische Funktionen)
- Funktionen können nur einen einfachen Wert liefern (insbesondere kein Kreuzprodukt von Werten; später Objekte)
- aktuelle Funktionsparameter werden von links nach rechts ausgewertet

- „API-Programmierer“ definieren (allgemeingültige) Funktionsbibliotheken
- Anwendungsprogrammierer nutzen die Bibliotheken
- Beispiel: Klasse IO

```
public class Math {
  static int fak(int n)
  static double power(double value, int exp)
  static int random(int min, int max)
  static double sqrt(double value)
  static double sin(double value)
  ...
}
```

- Seiteneffekt: Funktionsaufruf führt zu Änderungen außerhalb der entsprechenden Funktion
  - Globale Variablen
  - Ausgaben auf die Konsole
- Seiteneffekte können zu Fehlern und schwer zu durchschauenden Abläufen führen

  → möglichst vermeiden (insbesondere in Funktionsbibliotheken)

```
// kann nur von Konsolenprogrammen verwendet werden
static void betrag(int zahl) { // Seiteneffekt
    System.out.print(zahl < 0 ? -zahl : zahl);
}

// kann auch von GUI-Programmen oder Server-Programmen
// verwendet werden
static int betrag(int zahl) { // kein Seiteneffekt
    return zahl < 0 ? -zahl : zahl;
}
```

Funktionen / Prozedurale Zerlegung (1)

- Entwurfsverfahren: Schrittweise Verfeinerung / Prozedurale Zerlegung
- Ziel: Komplexitätsreduktion
- Prinzip:
  - Gesamtproblem zu komplex => Aufteilung in einfachere Teilprobleme
  - Lösen der Teilprobleme:
    - Teilproblem zu komplex => Aufteilung in (noch) einfachere Teilprobleme
    - ...
    - Zusammensetzung der Lösungen der Teilprobleme zur Lösung des (übergeordneten) Teilproblems
  - Zusammensetzung der Lösungen der Teilprobleme zur Lösung des Gesamtproblems
- Umsetzung: Lösung von Teilproblemen in Prozeduren/Funktionen

```
class Haus {

   public static void main(String[] args) {
      int eingabe = IO.readInt();
      zeichneHaus(eingabe);
   }

   static void zeichneHaus(int groesse) {
      zeichneDach(groesse);
      zeichneObergeschoss(groesse);
      zeichneUntergeschoss(groesse);
   }

   static void zeichneDach(int groesse) { ... }

   static void zeichneObergeschoss(int groesse) { ... }

   static void zeichneUntergeschoss(int groesse) { ... }
}
```

Vorteile:

- bessere Übersichtlichkeit von Programmen
- Platzeinsparung
- einfachere Fehlerbeseitigung
- Flexibilität (→ Parameter)
- Wiederverwendbarkeit (→ Funktionsbibliotheken)
- separate Lösung von Teilproblemen (→ prozedurale Zerlegung)

Rekursion:

Eine Funktion heißt **rekursiv**, wenn sie während ihrer Abarbeitung erneut aufgerufen wird.

Funktionsinkarnation:

konkreter Aufruf einer Funktion

Rekursionstiefe:

Anzahl der aktuellen Inkarnationen einer Funktion

Variableninkarnation:

konkrete Ausprägung (Speicherplatz) einer Variablen

Iterativer Algorithmus:

Algorithmus, der Wiederholungsanweisungen verwendet

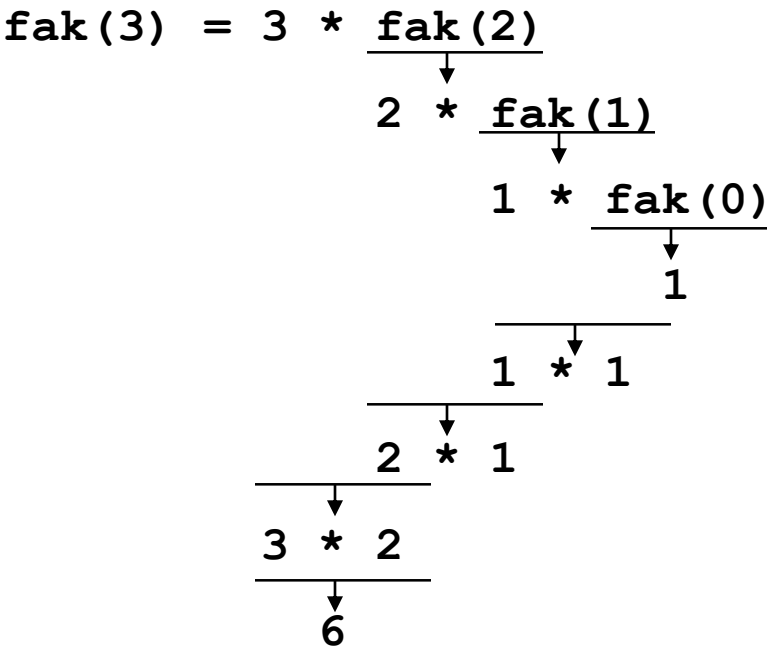
Rekursiver Algorithmus:

Algorithmus, der rekursive Funktionen verwendet

Berechnung der Fakultätsfunktion:

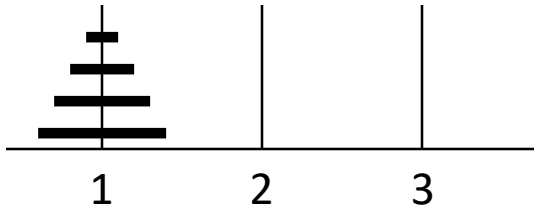
$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)! & \text{sonst} \end{cases}$$

```
static int fak(int n) {
  if (n <= 0) return 1;
  else return n * fak(n-1);
}
```



Türme von Hanoi:

Gegeben: 3 Pfosten mit n Scheiben
Ziel: Lege alle n Scheiben von 1 nach 3
Restriktion 1: immer nur eine Scheibe bewegen
Restriktion 2: niemals größere auf kleinere Scheibe



http://www.webgamesonline.com/towers-of-hanoi/

```
class Hanoi {
  public static void main(String[] args) {
    int hoehe = IO.readInt("Hoehe: ");
    verlegeTurm(hoehe, 1, 3, 2);
  }
  static void verlegeTurm(int hoehe, int von,
                                     int nach, int ueber) {
    if (hoehe > 0) {
      verlegeTurm(hoehe-1, von, ueber, nach);
      IO.println(von + "-" + nach);
      verlegeTurm(hoehe-1, ueber, nach, von);
} } }
```

Zusammenfassung

- Prozedur: Teil eines Programmes, das eine in sich abgeschlossene Aufgabe löst

- Funktion: Teil eines Programmes, das durch die Ausführung von Anweisungen einen Wert berechnet

- Parameter: funktionslokale Variable, deren Initialwert jeweils beim Aufruf der Funktion berechnet wird

- Rekursion: Definition einer Funktion durch sich selbst