Prof. Andreas Goerdt TU Chemnitz
M. Pippig, M. Riedel, U. Schönke Wintersemester 2003/2004

Algorithmen und Programmierung

7. Übung – Lösungsvorschläge

1. Aufgabe:

```
import Prog1Tools.IOTools;
public class Potenz {
  public static void main(String[] args) {
    int a, b, i, ergebnis;

    a = IOTools.readInteger("Bitte a eingeben: ");
    b = IOTools.readInteger("Bitte b eingeben: ");

    // Berechnung mit while
    ergebnis = 1; // Init. auch auf a moeglich, dann mit i=2 beginnen
    i = 1;
    while (i <= b){
      ergebnis = ergebnis * a;
      i = i + 1;
    }
    System.out.println(a + " hoch " + b + " mit while: " + ergebnis);

    // Berechnung mit do while
    ergebnis = 1;  // hier keine Init. auf a moeglich!
    i = 1;
    do{
      ergebnis = ergebnis * a;
      i = i + 1;
    } while (i <= b);
    System.out.println(a + " hoch " + b + " mit do-while: " + ergebnis);

    // Berechnung mit for
    ergebnis = 1;  // Init. auch auf a moeglich, dann mit i=2 beginnen
    for(i = 1; i <= b; i++){
      ergebnis = ergebnis * a;
    }
    System.out.println(a + " hoch " + b + " mit do-while: " + ergebnis);

    return;
  }
}
```

2. Aufgabe:

   1. Beispiel:

   Der Programmierer will offensichtlich ausgeben, ob der Wert von x kleiner 0 oder größer gleich 5 ist (Werte zwischen 0 und 4 werden nicht behandelt). Da der `else`-Zweig jedoch zur inneren `if`-Anweisung gehört – und nicht, wie uns die Einrückung weismachen will, zur äußeren –, liefert das Programmfragment bei $0 \leq x \leq 4$ die Ausgabe `x >= 5`. Möglichkeiten zur Korrektur:

```
if (x < 5)
    if (x < 0)
        System.out.println("x < 0");
    else ;
else
    System.out.println("x >= 5");

// oder:
if (x < 5){
    if (x < 0)
        System.out.println("x < 0");
}
else
    System.out.println("x >= 5");
```

   2. Beispiel:

   Der Programmierer möchte den Wert $\frac{1}{x}$ berechnen und vorher sicherstellen, dass x nicht gleich 0 ist. Die zweite Ausgabe-Anweisung gehört jedoch nicht mehr zur `if`-Anweisung, so dass im Falle $x = 0$ dennoch versucht wird, die Division auszuführen. Korrektur:

```
if (x > 0){
    System.out.println("ok! x > 0");
    System.out.println("1/x = " + (1/x));
}
```

   3. Beispiel:

   Der Programmierer möchte die Werte von x und y vertauschen, falls $y > x$ ist. In diesem Fall erhält zunächst x den Wert von y. Danach erhält y den Wert von x, welches aber bereits den Wert von y hat. Der ursprüngliche Wert von x ist verloren. Korrektur durch Einführung einer Hilfsvariable, die den Wert von x zwischenspeichert:

```
// Initialisierung von hilf (gleicher Typ wie x und y)
if (y > x) {
    // vertausche x und y
    hilf = x;
    x = y;
    y = hilf;
}
```

3. Aufgabe:

a) Bei der *Deklaration* werden durch Angabe des Komponententyps und der Dimensionsanzahl (Anzahl der leeren Klammerpaare) lediglich die Variablen angelegt, mit denen die Feldobjekte angesprochen werden können (Referenzen). Erst bei der *Erzeugung* wird der durch die Feldvariable referenzierte Speicherplatz bereitgestellt.

b) Es gibt folgende 3 Möglichkeiten:

1. Explizite Felderzeugung mittels new: z. B. `a = new double[5];`
2. Mittels Initialisierliste bei der Deklaration:
   z. B. `double[] a = {1.1, 2.2, 3.3, 4.4, 5.5};`
3. Kombination von 1. und 2.:
   z. B. `a = new double[] {1.1, 2.2, 3.3, 4.4, 5.5};`

c) Die einzelnen Elemente haben keinen Namen. Man greift über die indizierte Feld-Variable zu. Als Indizes sind 0 bis $n-1$ erlaubt (n: Größe des Feldes). Das letzte Element eines Feldes mit dem Namen a hat den Index `a.length-1`. Ein unzulässiger Index führt zu einer *ArrayIndexOutOfBoundsException*.

d)

```
byte      a, b;
byte[]    aReihe, bReihe, aZeile, bZeile;
byte[][]  aMatrix, bMatrix;
```

e)

```
int[][][][] Feld = new int[6][][][];
for (int d1 = 0; d1 < 6; d1++) {
    Feld[d1] = new int[10][][];
    for (int d2 = 0; d2 < 10; d2++) {
        Feld[d1][d2] = new int[8][];
    }
}

//oder (allgemeiner)
int[][][][] Feld = new int[6][][][];
for (int d1 = 0; d1 < Feld.length; d1++) {
    Feld[d1] = new int[10][][];
    for (int d2 = 0; d2 < Feld[d1].length; d2++) {
        Feld[d1][d2] = new int[8][];
    }
}
```

4. Aufgabe:

```
int[][] Dreieck = new int[100][];
for (int i = 0; i < Dreieck.length; i++){
    Dreieck[i] = new int[i+1];
}
```

5. Aufgabe:

   Ein Binärbruch, also eine gebrochene Binärzahl, ist (dezimal interpretiert) eine Summe aus Zweierpotenzen. Die Stellen vor dem Komma repräsentieren dabei die Zweierpotenzen mit positivem Exponenten, die Stellen nach dem Komma jene mit negativem Exponenten. Beispiel: Den Binärbruch

   $$101101,01101_2$$

   kann man auch schreiben als

   $$(1\cdot 2^5+0\cdot 2^4+1\cdot 2^3+1\cdot 2^2+0\cdot 2^1+1\cdot 2^0+0\cdot 2^{-1}+1\cdot 2^{-2}+1\cdot 2^{-3}+0\cdot 2^{-4}+1\cdot 2^{-5})_{10}$$

   Da Zweierpotenzen (auch mit negativem Exponenten) immer endlich sind, ist auch deren Summe endlich. Folglich ist jeder endliche Binärbruch auch als endlicher Dezimalbruch darstellbar.

   „Dreierbrüche“ und „Fünferbrüche“ definieren wir analog als Summe von Dreier- bzw. Fünferpotenzen.

   Dass nicht jeder endliche Dreierbruch als endlicher Dezimalbruch darstellbar ist, lässt sich einfach an einem Beispiel zeigen: Die Zahl

   $$0,1_3$$

   entspricht

   $$0\cdot 3^0+1\cdot 3^{-1}=\frac{1}{3}=0,3333333..._{10}=0,\overline{3}_{10}$$

   Dies ist ein unendlicher Dezimalbruch, der entsprechende Dreierbruch ist jedoch endlich.

   Andererseits gilt jedoch die Aussage, dass jeder endliche Fünferbruch auch als endlicher Dezimalbruch darstellbar ist. Das ist der Fall, da Fünfer-Potenzen mit negativem Exponenten immer als endlicher Dezimalbruch dargestellt werden können und dies somit auch für deren Summe gilt. Nachweis: $\frac{1}{5^1}=0,2$ ist ein endlicher Bruch. Bei der Addition und Multiplikation von endlichen Brüchen entstehen wieder endliche Brüche.

6. Aufgabe:

   $\lfloor \log_2(\log_2 a) \rfloor + 1$ gibt an, wie oft man aus einer Zahl a die Quadratwurzel ziehen kann, bis man einen Wert < 2 erhält. Beispiel:

$$\begin{array}{rcr} \sqrt{256} & = & 16 \\ \sqrt{16} & = & 4 \\ \sqrt{4} & = & 2 \\ \sqrt{2} & = & 1,41421 \end{array}$$

   Wir können also 4 mal die Wurzel aus 256 ziehen, bis wir einen Wert < 2 erhalten.

$$\lfloor \log_2(\log_2 256) \rfloor + 1 = \lfloor \log_2 8 \rfloor + 1 = 3 + 1 = 4$$

   Nachweis:

   Wir betrachten zunächst den doppelten Logarithmus einer Zahl der Form $2^{(2^x)}$ mit $x \in \mathbb{N}$:

$$\begin{array}{rcl} \log_2(\log_2 2^{(2^x)}) & = & \log_2(2^x) \\ & = & x \end{array}$$

   Ziehen wir aus $2^{(2^x)}$ die Quadratwurzel, erhalten wir

$$\sqrt{2^{(2^x)}} = 2^{(2^{x-1})}$$

   Ziehen wir aus $2^{(2^x)}$ x mal die Quadratwurzel, erhalten wir

$$\underbrace{\sqrt{\cdots\sqrt{\sqrt{2^{(2^x)}}}}}_{x\text{mal}} = 2$$

   Nun betrachten wir eine beliebige reelle Zahl z, $z \in \left[2^{(2^x)}, 2^{2^{x+1}}\right)$. Da die Logarithmus-Funktion monoton ist, gilt

$$\lfloor \log_2(\log_2 z) \rfloor = x$$

   Folglich gibt $\lfloor \log_2(\log_2 z) \rfloor + 1$ an, wie oft man aus einer Zahl z die Wurzel ziehen kann bis man einen Wert < 2 erhält.