Inhaltsverzeichnis

Bruchzahlen in Rechnern

Eine Bruchzahl, mit der man (von Hand oder mit einem Rechner) praktische Berechnungen durchführen will, hat **zwei wichtige Eigenschaften**: Eine bestimmte **Größe** und eine bestimmte **Genauigkeit**. Diese beiden Eigenschaften sind **unabhängig voneinander**, wie die folgenden Beispiele deutlich machen sollen:

Beispiele für Bruchzahlen mit unterschiedlichen **Größen** und **Genauigkeiten**:

```
1   3.456345645634564563456456345 * 10^+25 (ziemlich gross, ziemlich   genau)
2   3.4                           * 10^+25 (ziemlich gross, ziemlich ungenau)
3   3.456345645634564563456456345 * 10^-25 (ziemlich klein, ziemlich   genau)
4   3.4                           * 10^-25 (ziemlich klein, ziemlich ungenau)
```

In Java kann man Bruchzahlen auf **zwei** ganz verschiedene Weisen darstellen:

1. Als **primitive Werte** (der Gleitpunkt-Typen **float** und **double**)
2. Als **Objekte** (der Klasse java.math.**BigDecimal**).

Primitive Bruchzahlen brauchen relativ **wenig Speicherplatz** und **Rechenzeit**, ihre **Größe** und ihre **Genauigkeit** sind allerdings **begrenzt**. Bruchzahl-**Objekte** brauchen **mehr Speicherplatz** und **Rechenzeit** als primitive Werte, die **Größe** und **Genauigkeit** der dargestellten Bruchzahlen ist dafür aber (für praktische Zwecke) **unbegrenzt**.

Aufgabe: Welche **Genauigkeit** und welche **maximale Grösse** haben **float**-Werte in Java?

Aufgabe: Ebenso für **double**-Werte.

Aufgabe: Verschaffen Sie sich die Quelldatei der Klasse **BigDecimal** (die Datei BigDecimal.java) und stellen Sie fest, wie eine Bruchzahl durch ein **BigDecimal**-Objekt dargestellt wird.. ´Was folgt aus dieser Darstellung über die **Genauigkeit** und **maximale Grösse** von BigDecimal-Bruchzahlen?

Ungenauigkeit von Gleitpunktzahlen

Wenn man (z.B. in einem Java-Programm) **Gleitpunktzahlen** addiert, subtrahiert, multipliziert oder dividiert ist das Ergebnis häufig **nicht** mathematisch **exakt** sondern mit einem (kleinen) **Fehler behaftet**. Wenn man **Kettenrechnungen** durchführt (d.h. mit dem Ergebnis **einer** Rechnung eine **weitere** Rechnung durchführt und mit dem Ergebnis dieser Rechnung noch eine **weitere** etc.) kann aus den kleinen Fehlern ein immer **größerer Fehler** entstehen. Hier ein Beispielprogramm, bei dem das in besonders hohem Maße der Fall ist:

```
1 // Datei Gleit1.java
2 /* ------------------------------------------------------------------------
3 Dieses Programm soll bestimmte Gefahren demonstrieren, die beim Rechnen
4 mit Gleitpunktzahlen drohen (Rundungsfehler). Die Formeln
5 ((1 + r) * x) - (r * x * x) und x + (r * x * (1 - x)) sind beide
6 mathematisch aequivalent zur Formel x + r*x * r*x*x (und somit auch zu-
7 einander aequivalent). Ihr Quotient sollte deshalb immer gleich 1 sein.
8 Zwei Variablen x1 und x2 werden mit 0.1 initialisiert und dann wird
9 ihr Wert wiederholt durch die folgenden Zuweisungen veraendert:
```

```
10
11 x1 = ((1 + 3) * x1) - (3 * x1 * x1); // Formel1
12 x2 = x2 + (3 * x2 * (1 - x2));       // Formel2
13
14 Die Werte von x1 und x2 und ihr Quotient x1 / x2 werden nach jeder
15 solchen Zuweisung ausgegeben. Aufgrund von Rundungsfehlern ist der
16 Quotient nicht immer gleich 1.0, wie er eigentlich sein sollte.
17
18 Gleit1 rechnet mit float-Zahlen (Gleit2 mit double-Zahlen)
19 ------------------------------------------------------------------------ */
20 import de.tfh_berlin.einaus.EM01;
21 import de.tfh_berlin.einaus.AM01;
22
23 public class Gleit1 {
24    static public void main(String[] susi) throws java.io.IOException {
25       // Ein paar Konstanten:
26       final float SEHR_GROSS = 1000.000F; // Tausend
27       final float SEHR_KLEIN =    0.001F; // Ein Tausendstel
28
29       // Ein paar Variablen:
30             float x1         =    0.100F; // Fuer Formel 1
31             float x2         =    0.100F; // Fuer Formel 2
32             float q;                      // Quotient x1/x2
33       // ---------------------------------------------------------------
34       AM01.p("Gleit1: Wie oft (1..500)? ");
35       int wieOft = EM01.liesInt();
36
37       // Ueberschriften ausgeben:
38       AM01.pln();
39       AM01.pln("Schritt          Zahl x1          Zahl x2   Quotient x1/x2");
40
41       // Wiederholt x1, x2 und q berechnen und ausgeben:
42       for (int i=1; i<=wieOft; i++) {
43          x1 = ((1 + 3) * x1) - (3 * x1 * x1); // x1 nach Formel1 berech.
44          x2 = x2 + (3 * x2 * (1 - x2));       // x2 nach Formel2 berech.
45          q  = x1 / x2;
46
47          AM01.p(i , 7);
48          AM01.p(x1, 7, 9);
49          AM01.p(x2, 7, 9);
50          AM01.p(q,  7, 9);
51          AM01.pln();
52
53          // Falls der Quotient q sehr gross oder sehr klein ist, wird die
54          // Ausgabe angehalten, damit die Ausgbezeilen nicht ungelesen
55          // "vor den Ausgen des Benutzers vorbeiflimmern":
56          if (q < SEHR_KLEIN || SEHR_GROSS < q) {
57             AM01.p("Weiter mit Return-Taste: ");
58             String s = EM01.liesString();
59          }
60       } // for
61       // ---------------------------------------------------------------
62    } // main
63 } // class Gleit1
64 /* ------------------------------------------------------------------------
65 Auszuege aus einem Dialog mit dem Programm Gleit1:
66
67 Gleit1: Wie oft (1..500)? 500
68
69 Schritt          Zahl x1          Zahl x2   Quotient x1/x2
70       1      0.370000005      0.370000005      1.000000000
71       2      1.069299936      1.069299936      1.000000000
72       3      0.846992731      0.846992671      1.000000119
73       4      1.235780954      1.235780954      1.000000000
74       5      0.361660004      0.361660123      0.999999642
75       6      1.054246187      1.054246426      0.999999762
76       7      0.882679701      0.882679105      1.000000715
77       8      1.193348408      1.193349242      0.999999285
78     ...      ...              ...              ...
79     129      0.264641762      0.195566759      1.353204250
80     130      0.848461270      0.667527914      1.271049857
```

```
81     131      1.234185457      1.333331108      0.925640643
82     132      0.367100716      0.000008941  41059.519531250
83     ...      ...              ...              ...
84     248      0.050074577      1.065626502      0.046990741
85     249      0.192775920      0.855826497      0.225251168
86     250      0.659615993      1.225988984      0.538027644
87     251      1.333184242      0.394808948      3.376783133
88     252      0.000596523      1.111613512      0.000536628
89     ...      ...              ...              ...
90     483      1.320042610      1.283196092      1.028714657
91     484      0.052632809      0.193007708      0.272697955
92     485      0.202220604      0.660274863      0.306267321
93     486      0.686202884      1.333210707      0.514699519
94     487      1.332188368      0.000490427   2716.384521484
95
96 ------------------------------------------------------------------- */
```

Man beachte, dass der Schritt **131** allein betrachtet "ganz harmlos" aussieht und der Quotient **x1/x2** dort weniger als 10 % vom richtigen und exakten Wert **1.0** abweicht. Nach nur **einem** weiteren Schritt hat der Quotient dann aber den katastrophal falschen Wert **41059**.519531250! Später, nach den Schritten **250** und **251** ist der Quotient dann nur etwa um den Faktor **2** zu klein bzw. um den Faktor **3.3** zu gross, aber nach einem weiteren Schritt ist er um einen Faktor von ungefähr **2700** zu klein. Hätten Sie das erwartet?

Das Programm **Gleit1** ist natürlich so **konstruiert**, dass die Rundungsfehler besonders deutlich sichtbare Auswirkungen haben. Vor allem werden sehr lange **Kettenrechnungen** durchgeführt (z.B. solche mit 500 "Kettengliedern" oder Rechenschritten). Das kommt in praktischen Programmen eher **selten** vor. Aber die Zahlen **x1** = 1.234185457 und **x2** = 1.33333110 (siehe oben Schritt **131**) könnten sich ja auch auf irgendeine andere Weise ergeben (z.B. als Messergebnisse eines Sensors) und dann ist ein völlig falsches Ergebnis nur noch **ein** Schritt weit weg.

Im Programm **Gleit1** wird mit **float**-Zahlen gerechnet. Wenn man statt dessen **double**-Zahlen verwendet, treten nicht etwa weniger sondern etwa **gleich viele** "Katastrophen" auf und der erste Quotient über 1000 erscheint sogar schon 4 Schritte **früher** (siehe Beispielprogramm **Gleit2**).

Binäre Gleitpunktzahlen und dezimale Menschen

Gleitpunktzahlen der Typen **float** und **double** werden maschinenintern als eine Art **Binärbruch** dargestellt. Die meisten Menschen haben **Mühe**, mit Binärbrüchen umzugehen und ziehen **Dezimalbrüche** vor. Daraus ergeben sich 2 Probleme:

1. Es gibt **Bruchzahlen**, die man zwar als (endliche) **Dezimalbrüche** darstellen kann, aber nicht als (endliche) **Binärbrüche**, z.B. die Zahl **1/10** (ein Zehntel, als Dezimalbruch: **0.1**, als Binärbruch: **0.0001100110011001...** oder **0.0001<u>1001</u>** mit der Periode **<u>1001</u>**). Wenn ein Benutzer die Zahl **0.1** z.B. in ein Feld einer Excel-Tabelle **eingibt**, steht danach der etwas zu grosse, **fehlerbehaftete** Wert
0.1000000000000000055511151231257827021181583404541015625
in dem Feld. Viele **Benutzer** rechnen nicht mit dem kleinen Unterschied zwischen **0.1** und dieser Zahl und sind erstaunt oder verärgert, wenn sie **Auswirkungen** des Unterschieds wahrnehmen (z.B. in anderen Feldern der Excel-Tabelle).

2. Für den **Benutzer** werden Gleitpunktzahlen fast nie realistisch als **Binärbrüche**, sondern meistens (ein bisschen unrealistisch) als **Dezimalbrüche** angezeigt. **Die** Zahl, die der **Benutzer sieht** und **die** Zahl, mit der der **Rechner rechnet**, stimmen dann häufig **nicht** (genau) überein. Auch diese Diskrepanz kann beim Benutzer Erstaunen und Verärgerung auslösen.

Bei **wissenschaftlichen** Berechnungen genügt es häufig, wenn das Ergebnis **ungefähr** richtig ist und der Fehler eine **gewisse Schranke** grantiert nicht überschreitet (z.B. eine **relative Schranke** wie 3 % oder eine **absolute Schranke** wie 0.05 m). Bei **kommerziellen** Berechnungen, insbesonde-

re wenn es um **Geldbeträge** geht, erwartet man dagegen in vielen Fällen ein **mathematisch exaktes** Ergebenis oder zumindest eines, welches ein **dezimal rechnender** Mensch (oder Taschenrechner) nachvollziehen kann. Das folgende Beispielprogramm soll deutlich machen, warum man für **kommerzielle** Berechnungen in aller Regel **keine Gleitpunktzahlen** verwenden soll:

```
 1 // Datei Preise01.java
 2 /* ------------------------------------------------------------------------
 3 Ungenauigkeiten beim Berechnen von Geldbetraegen, Gewichten, ... etc.
 4 ("beim kommerziellen Rechnen") mit Gleitpunktzahlen.
 5 ------------------------------------------------------------------------ */
 6 import de.tfh_berlin.einaus.AM01;
 7
 8 class Preise01 {
 9    // ---------------------------------------------------------------------
10    static final String MINUSZEILE = "-----------------------------" +
11                                     "-----------------------------";
12    // ---------------------------------------------------------------------
13    static public void main(String[] susi) {
14
15       AM01.pln("Preise01: Jetzt geht es los!");
16       AM01.pln(MINUSZEILE);
17       // ------------------------------------------------------------------
18       // Parametersatz01 fuer PrintRechnung (ziemlich genau):
19       float preisProKg01    =  4.90F;
20       float gewicht01       = 22.400F;
21       // ------------------------------------------------------------------
22       // Parametersatz02 fuer PrintRechnung (etwas ungenau):
23       float gewichtsEinheit = 0.1F; // 100 g
24       float    preisEinheit = 0.1F; //  10 Cent
25
26       float preisProKg02    =   129 * preisEinheit    - 8;
27       float gewicht02       = 25824 * gewichtsEinheit - 2560;
28       // ------------------------------------------------------------------
29       // Parametersatz03 fuer PrintRechnung (sehr ungenau):
30       float preisProKg03    =  4.9049999F;
31       float gewicht03       = 22.4004999F;
32       // ------------------------------------------------------------------
33       // Drei unterschiedlich genaue (sonst gleiche) Rechnungen ausgeben:
34       printRechnung(preisProKg01, gewicht01);
35       printRechnung(preisProKg02, gewicht02);
36       printRechnung(preisProKg03, gewicht03);
37
38       AM01.pln("Preise01: Das war's erstmal!");
39    } // main
40    // ---------------------------------------------------------------------
41    static void printRechnung(float preisProKg, float gewicht) {
42       // Gibt eine Rechnung mit anzahl vielen Posten (Zeilen) aus:
43
44       int   anzahl         = 10;
45       float preisProStk    = preisProKg  * gewicht;
46       float preisProPosten = preisProStk * anzahl;
47       float summe          = 0.0F;
48
49       final int SB = 12; // Spalten-Breite (in Zeichen)
50
51       String t11 = AM01.pad("Preis/kg",  SB, AM01.RECHTSBUENDIG);
52       String t12 = AM01.pad("Gewicht",   SB, AM01.RECHTSBUENDIG);
53       String t13 = AM01.pad("Preis/Stk", SB, AM01.RECHTSBUENDIG);
54       String t14 = AM01.pad("Anzahl",    SB, AM01.RECHTSBUENDIG);
55       String t15 = AM01.pad("Preis",     SB, AM01.RECHTSBUENDIG);
56
57       String t21 = AM01.pad("in Euro",   SB, AM01.RECHTSBUENDIG);
58       String t22 = AM01.pad("in kg",     SB, AM01.RECHTSBUENDIG);
59       String t23 = AM01.pad("",          SB, AM01.RECHTSBUENDIG);
60
61       AM01.pln(t11 + t12 + t13 + t14 + t15);
62       AM01.pln(t21 + t22 + t21 + t23 + t21);
63       for (int i=0; i<anzahl; i++) {
64          AM01.p(AM01.pad(preisProKg,     5, 2), SB, AM01.RECHTSBUENDIG);
```

```
65          AM01.p(AM01.pad(gewicht,        6, 3), SB, AM01.RECHTSBUENDIG);
66          AM01.p(AM01.pad(preisProStk,    8, 2), SB, AM01.RECHTSBUENDIG);
67          AM01.p(AM01.pad(anzahl,            7), SB, AM01.RECHTSBUENDIG);
68          AM01.p(AM01.pad(preisProPosten, 9, 2), SB, AM01.RECHTSBUENDIG);
69          summe += preisProPosten;
70          AM01.pln();
71       }
72       AM01.p("Rechnungs-Summe:", 4*SB);
73       AM01.p(AM01.pad(summe, 9, 2), SB, AM01.RECHTSBUENDIG);
74       AM01.pln();
75
76       AM01.pln(MINUSZEILE);
77       AM01.pln("Der (ziemlich) genaue Preis/kg: " + preisProKg);
78       AM01.pln("Das (ziemlich) genaue Gewicht:  " + gewicht);
79       AM01.pln(MINUSZEILE);
80    } // printRechnung
81    // ------------------------------------------------------------------------
82 } // class Preise01
83 /* ------------------------------------------------------------------------
84
```

```
 85 Ausgabe des Programms Preise01:
 86
 87 Preise01: Jetzt geht es los!
 88 ------------------------------------------------------------
 89     Preis/kg     Gewicht   Preis/Stk      Anzahl       Preis
 90      in Euro       in kg     in Euro                  in Euro
 91         4.90      22.400      109.76          10     1097.60
 92         4.90      22.400      109.76          10     1097.60
 93         4.90      22.400      109.76          10     1097.60
 94         4.90      22.400      109.76          10     1097.60
 95         4.90      22.400      109.76          10     1097.60
 96         4.90      22.400      109.76          10     1097.60
 97         4.90      22.400      109.76          10     1097.60
 98         4.90      22.400      109.76          10     1097.60
 99         4.90      22.400      109.76          10     1097.60
100         4.90      22.400      109.76          10     1097.60
101 Rechnungs-Summe:                                    10976.00
102 ------------------------------------------------------------
103 Der (ziemlich) genaue Preis/kg: 4.9
104 Das (ziemlich) genaue Gewicht:  22.4
105 ------------------------------------------------------------
106     Preis/kg     Gewicht   Preis/Stk      Anzahl       Preis
107      in Euro       in kg     in Euro                  in Euro
108         4.90      22.400      109.76          10     1097.61
109         4.90      22.400      109.76          10     1097.61
110         4.90      22.400      109.76          10     1097.61
111         4.90      22.400      109.76          10     1097.61
112         4.90      22.400      109.76          10     1097.61
113         4.90      22.400      109.76          10     1097.61
114         4.90      22.400      109.76          10     1097.61
115         4.90      22.400      109.76          10     1097.61
116         4.90      22.400      109.76          10     1097.61
117         4.90      22.400      109.76          10     1097.61
118 Rechnungs-Summe:                                    10976.07
119 ------------------------------------------------------------
120 Der (ziemlich) genaue Preis/kg: 4.9000006
121 Das (ziemlich) genaue Gewicht:  22.400146
122 ------------------------------------------------------------
123     Preis/kg     Gewicht   Preis/Stk      Anzahl       Preis
124      in Euro       in kg     in Euro                  in Euro
125         4.90      22.400      109.87          10     1098.74
126         4.90      22.400      109.87          10     1098.74
127         4.90      22.400      109.87          10     1098.74
128         4.90      22.400      109.87          10     1098.74
129         4.90      22.400      109.87          10     1098.74
130         4.90      22.400      109.87          10     1098.74
131         4.90      22.400      109.87          10     1098.74
132         4.90      22.400      109.87          10     1098.74
133         4.90      22.400      109.87          10     1098.74
134         4.90      22.400      109.87          10     1098.74
135 Rechnungs-Summe:                                    10987.44
136 ------------------------------------------------------------
137 Der (ziemlich) genaue Preis/kg: 4.9049997
138 Das (ziemlich) genaue Gewicht:  22.4005
139 ------------------------------------------------------------
140 Preise01: Das war's erstmal!
141 ------------------------------------------------------------------------ */
```

Das Programm **Preise01** gibt drei ähnliche **Rechnungen** aus. Jede Rechnung betrifft **10 Posten** (Zeilen) á **10 Stück** einer Ware, die "nach Gewicht" verkauft wird. Damit die Rechnungen leicht überprüft werden können, sind die 10 Posten (Zeilen) in diesem Beispiel alle gleich.

Die erste Rechnung ist so wie sie sein sollte. Bei der zweiten Rechnung ist der Preis eines **Postens** um **einen Cent zu hoch** und die **Rechnungs-Summe** (in der letzten Zeile) je nach Rechenweg um **3 Cent zu niedrig** oder um **7 Cent zu hoch**. Die dritte Rechnung gibt dem Empfänger das Gefühl, dass ihm ein um **20 Cent zu hoher Preis/Stk** berechnet wurde und enthält weitere Fehler. Man beachte, dass die ersten beiden Spalten (**Preis/kg** und **Gewicht**) in allen drei Rechnungen genau **übereinstimmen**.

Kommerzielle Berechnungen mit BigDecimal-Objekten

Kommerzielle Berechnungen, die häufig mit **dezimalen** Taschenrechnern (oder von **dezimalen** Kopfrechnern) überprüft werden (insbesondere Berechnungen von Geldbeträgen), sollte man nicht mit **Gleitpunktzahlen** durchführen, sondern (in Java) mit Objekten des Typ **BigDecimal**. Hier ein kleines Beispielprogramm (Ähnlichkeiten mit dem vorigen Beispielprogramm **Preise01** sind kein Zufall):

```
 1 // Datei Preise02.java
 2 /* ------------------------------------------------------------------------
 3 Objekte des Typs BigDecimal repraesentieren Dezimalbrueche, bei denen man
 4 die Anzahl der Nachpunktstellen genau festlegen und die man auf viele
 5 verschiedene Weisen runden kann. Sie sind besonders fuer kommerzielle
 6 Berechnungen geeignet.
 7 ------------------------------------------------------------------------ */
 8 import java.math.BigDecimal;
 9 import de.tfh_berlin.einaus.AM01;
10
11 class Preise02 {
12    // ---------------------------------------------------------------------
13    static final String MINUSZEILE = "-----------------------------" +
14                                     "-----------------------------";
15    // ---------------------------------------------------------------------
16    static public void main(String[] susi) {
17       AM01.pln("Preise02: Jetzt geht es los!");
18       AM01.pln(MINUSZEILE);
19
20       // Parametersatz fuer PrintRechnung (genau):
21       BigDecimal preisProKg    = new BigDecimal( "4.90");
22       BigDecimal gewicht       = new BigDecimal("22.400");
23       // ------------------------------------------------------------------
24       printRechnung(preisProKg, gewicht);
25
26       AM01.pln("Preise02: Das war's erstmal!");
27    } // main
28    // ---------------------------------------------------------------------
29    static void printRechnung(BigDecimal preisProKg, BigDecimal gewicht) {
30       // Gibt eine Rechnung mit anzahl vielen Posten (Zeilen) aus:
31
32       BigDecimal anzahl         = new BigDecimal("10.00");
33       BigDecimal summe          = new BigDecimal( "0.00");
34
35       // Produkte berechnen:
36       BigDecimal preisProStk    = preisProKg .multiply(gewicht);
37       BigDecimal preisProPosten = preisProStk.multiply(anzahl);
38
39       // Die Produkte auf 2 Stellen runden:
40       preisProStk    = preisProStk   .setScale(2, BigDecimal.ROUND_HALF_UP);
41       preisProPosten = preisProPosten.setScale(2, BigDecimal.ROUND_HALF_UP);
42
43       final int SB = 12; // Spalten-Breite (in Zeichen)
44
45       String t11 = AM01.pad("Preis/kg",  SB, AM01.RECHTSBUENDIG);
46       String t12 = AM01.pad("Gewicht",   SB, AM01.RECHTSBUENDIG);
47       String t13 = AM01.pad("Preis/Stk", SB, AM01.RECHTSBUENDIG);
48       String t14 = AM01.pad("Anzahl",    SB, AM01.RECHTSBUENDIG);
49       String t15 = AM01.pad("Preis",     SB, AM01.RECHTSBUENDIG);
50
51       String t21 = AM01.pad("in Euro",   SB, AM01.RECHTSBUENDIG);
52       String t22 = AM01.pad("in kg",     SB, AM01.RECHTSBUENDIG);
53       String t23 = AM01.pad("",          SB, AM01.RECHTSBUENDIG);
54
55       AM01.pln(t11 + t12 + t13 + t14 + t15);
56       AM01.pln(t21 + t22 + t21 + t23 + t21);
57
58       for (int i=0; i<anzahl.intValue(); i++) {
59          AM01.p(preisProKg,     SB, AM01.RECHTSBUENDIG);
60          AM01.p(gewicht,        SB, AM01.RECHTSBUENDIG);
61          AM01.p(preisProStk,    SB, AM01.RECHTSBUENDIG);
```

```
62          AM01.p(anzahl,         SB, AM01.RECHTSBUENDIG);
63          AM01.p(preisProPosten, SB, AM01.RECHTSBUENDIG);
64          summe = summe.add(preisProPosten);
65          AM01.pln();
66       }
67       AM01.p("Rechnungs-Summe:", 4*SB);
68       AM01.p(summe, SB, AM01.RECHTSBUENDIG);
69       AM01.pln();
70
71       AM01.pln(MINUSZEILE);
72       AM01.pln("Der genaue Preis/kg: " + preisProKg);
73       AM01.pln("Das genaue Gewicht:  " + gewicht);
74       AM01.pln(MINUSZEILE);
75    } // printRechnung
76    // ------------------------------------------------------------------------
77 } // class Preise02
78 /* ------------------------------------------------------------------------
79 Ausgabe des Programms Preise02:
80
81 Preise02: Jetzt geht es los!
82 ------------------------------------------------------------
83     Preis/kg     Gewicht   Preis/Stk      Anzahl       Preis
84      in Euro       in kg     in Euro                 in Euro
85         4.90      22.400      109.76       10.00     1097.60
86         4.90      22.400      109.76       10.00     1097.60
87         4.90      22.400      109.76       10.00     1097.60
88         4.90      22.400      109.76       10.00     1097.60
89         4.90      22.400      109.76       10.00     1097.60
90         4.90      22.400      109.76       10.00     1097.60
91         4.90      22.400      109.76       10.00     1097.60
92         4.90      22.400      109.76       10.00     1097.60
93         4.90      22.400      109.76       10.00     1097.60
94         4.90      22.400      109.76       10.00     1097.60
95 Rechnungs-Summe:                                    10976.00
96 ------------------------------------------------------------
97 Der genaue Preis/kg: 4.90
98 Das genaue Gewicht:  22.400
99 ------------------------------------------------------------
100 Preise02: Das war's erstmal!
101 ------------------------------------------------------------------------ */
```

Die Objekte des Java-Typs **BigDecimal** repräsentieren **Dezimalbrüche mit** (praktisch) **unbegrenzter Größe und Genauigkeit**. Jedes **BigDecimal**-Objekt **bd** enthält **zwei** Ganzzahl-Attribute:

```
1  BigInteger uw; // Unskalierter Wert (kann positiv, gleich 0 oder negativ sein)
2  int        sf; // Skalenfaktor      (kann nur positiv oder gleich 0 sein!)
```

Das Objekt **bd** repräsentiert den Wert ("den Dezimalbruch") $\mathbf{uw / 10^{sf}}$.

Beispiel: Der Dezimalbruch **12.345** wird durch **uw** gleich **12345** und **sf** gleich **3** dargestellt.

Aufgabe: Wieviele **Stellen nach dem Dezimalpunkt** kann eine **BigDecimal**-Zahl ("die durch ein **BigDecimal**-Objekt repräsentierte Zahl") **höchstens** haben?

Aufgabe: Geben Sie Zahlen an (mindestens zwei), die man **nicht** als **Dezimalbruch** darstellen (und somit auch **nicht** durch ein **BigDecimal**-Objekt repräsentieren) kann. Geben Sie für jede der Zahlen **zwei** andere Darstellungen an (nicht als Dezimalbruch, sondern 1. als ... und 2. als ...).

BigDecimal-Objekte sind **nicht veränderbar** (ähnlich wie **String**-Objekte und Objekte der acht Wickelklassen **Boolean**, **Byte**, **Character**, **Short**, **Integer**, **Long**, **Float** und **Double**). Die Rechenmethoden **add**, **subtract**, **multiply** und **divide** eines BigDecimal-Objekts **bd** sind **Funktionen ohne Seiteneffekt**, d.h. sie liefern jeweils ein **neues** Objekt als Ergebnis und lassen das Objekt **bd** unverändert.

Die Methoden **add**, **subtract** und **multiply** eines **BigDecimal**-Objekts liefern **exakte** (mathematisch korrekte) Ergebnisse. Beim **Dividieren** ist das nicht so einfach möglich, weil das **Ergebnis** (dargestellt als Dezimalbruch) unendlich viele Nachpunktstellen haben kann (z.B. ist **1/3** gleich

0.33333... und **1/7** ist gleich **0.142857142857142857142...**). Beim **Dividieren** von **BigDecimal**-Objekten (mit der Methode **divide**) muss man deshalb festlegen, auf wieviele Nachpunktstellen genau das Ergebnis berechnet und wie es **gerundet** werden soll.

Aufgabe: Seien **bd1** und **bd2** zwei **BigDecimal**-Objekte mit den Skalenfaktoren **sf1** und **sf2**. Welchen **Skalenfaktor** haben die Objekte **bd1.add(bd2)**, **bd1.subtract(bd2)**, **bd1.multiply(bd2)**, **bd1.divide(bd2, BigDecimal.ROUND_UP)** und **bd1.divide(bd2, 17, BigDecimal.ROUND_UP)**? Lesen Sie dazu in der Dokumentation der Klasse **BigDecimal** und/oder probieren Sie es aus mit dem Beispielprogramm **BigDecimal01**.

Ein Methodenaufruf wie etwa

```
3  ... bd1.setScale(3) ...
```

liefert ein Objekt mit einem **genau gleichen Wert** wie **bd1** aber mit einem Skalenfaktor 3 ("mit 3 Nachpunktstellen"). Das geht nur dann, wenn **bd1** keine vierte oder spätere ("weiter rechts stehende") Nachpunktstelle hat, die **ungleich 0** ist. Mit dieser Operation kann man also auf keinen Fall "signifikante Nachpunktstellen" (von 0 verschiedene Nachpunktstellen) abschneiden. Wenn man es doch versucht, wird eine Ausnahme (**ArithmeticException**) geworfen.

Ein Methodenaufruf wie etwa

```
4  ... bd1.setScale(3, BigDecimal.ROUND_UP) ...
```

liefert ein Objekt mit einem **ungefähr** (oder **genau**) **gleichen Wert** wie **bd1**, aber mit einem Skalenfaktor 3 ("mit 3 Nachpunktstellen"). Falls nötig, wird der Wert von **bd1** dazu nach der Methode **ROUND_UP** (siehe weiter unten) auf 3 Nachkommastellen **gerundet**. Mit dieser Operation kann man also auch "signifikante Nachpunktstellen" abschneiden und sollte sie entsprechend vorsichtig anwenden.

Verschiedene Arten BigDecimal-Zahlen zu Runden

Bei kommerziellen Berechnungen (insbesondere wenn es um Geldbeträge geht) ist die genaue Art **wie gerundet wird**, häufig sehr **wichtig** und teilweise gesetzlich **vorgeschrieben**. Deutsche Finanzämter müssen z.B. ihre Forderungen an Steuerzahler auf ganze Euros **abrunden**. Die Klasse **BigDecimal** bietet **acht** verschiedene Arten zu runden an. Konkret enthält die Klasse dazu acht **int-Konstanten** namens

ROUND_UP,
ROUND_DOWN,
ROUND_CEILING,
ROUND_FLOOR,
ROUND_HALF_UP,
ROUND_HALF_DOWN,
ROUND_HALF_EVEN und
ROUND_UNNECESSARY,

die man als Parameter der Methoden **divide** und **setScale** angeben kann, um die entsprechende Rundungsart auszuwählen.

Die "Pseudo-Rundungsart" **ROUND_UNNECESSARY** sollte man nur angeben, wenn kein Runden nötig ist (wenn man also höchstens solche Nachpunktstellen abschneiden lässt, die gleich 0 sind). Falls man sich dabei irrt, wird eine Ausnahme (ArithmeticException) geworfen.

Kurze Erläuterungen der "richtigen" (nicht-pseudo) Rundungsarten
("N wird eher kleiner" ist eine Abkürzung für "N wird kleiner oder bleibt gleich"):

ROUND_UP: Hin zur 0. Positive Zahlen werden eher kleiner,
negative Zahlen eher grösser, der Betrag eher kleiner.

ROUND_DOWN: Weg von der 0. Positive Zahlen werden eher grösser,
negative Zahlen eher kleiner, der Betrag eher grösser.

ROUND_CEILING: Nach rechts ("in Richtung plus Unendlich"). Positive und negative Zahlen werden eher grösser, der Betrag von positiven Zahlen wird eher grösser, der Betrag von negativen Zahlen eher kleiner.

ROUND_FLOOR: Nach links ("in Richtung minus Unendlich"). Positive und negative Zahlen werden eher kleiner, der Betrag von positiven Zahlen wird eher kleiner, der Betrag von negativen Zahlen eher grösser.

ROUND_HALF_UP: Zum nächsten Nachbarn
(und hin zur 0, wenn beide Nachbarn gleich weit weg sind).

ROUND_HALF_DOWN: Zum nächsten Nachbarn
(und weg von der 0, wenn beide Nachbarn gleich weit weg sind).

ROUND_HALF_EVEN: Zum nächsten Nachbarn
(und zum geraden Nachbarn, wenn beide Nachbarn gleich weit weg sind).

Hinweis: In der online-Dokumentation der Java-Standardklassen von Sun (insbesondere in der Dokumentation der Klasse **BigDecimal**) werden die Worte **magnitude** und **absolute value** beide mit der Bedeutung **Betrag** verwendet (wie in "Der **Betrag** von -5 ist +5" und "Der **Betrag** von +5 ist auch +5" etc.). Das ist unglücklich, weil **magnitude** in vielen englischen Texten in einer anderen Bedeutung ("Größenordnung") verwendet wird.

Anmerkung: Dass die verschiedenen Rundungsarten hier durch **int-Konstanten** bezeichnet werden gehört **nicht** zu den Spitzenleistungen moderner Softwaretechnik.

Das Beispielprogramm **BigDecimal02** gibt die folgende Tabelle aus. Sie enthält Beispiele für die Wirkung der verschiedenen Rundungsarten "in allen möglichen Fällen". Die Zahl in der ersten Spalte hat **3 Nachpunktstellen** und wird jeweils auf **2 Nachpunktstellen** genau **gerundet**:

```
 1 BigDecimal02: So werden BigDecimal's gerundet:
 2 Zahl       CEILING   UP        FLOOR     DOWN      HALF_DOWN HALF_EVEN HALF_UP
 3 ------------------------------------------------------------------------------
 4 12.345     12.35     12.35     12.34     12.34     12.34     12.34     12.35
 5 12.346     12.35     12.35     12.34     12.34     12.35     12.35     12.35
 6 12.347     12.35     12.35     12.34     12.34     12.35     12.35     12.35
 7 12.348     12.35     12.35     12.34     12.34     12.35     12.35     12.35
 8 12.349     12.35     12.35     12.34     12.34     12.35     12.35     12.35
 9 12.350     12.35     12.35     12.35     12.35     12.35     12.35     12.35
10 12.351     12.36     12.36     12.35     12.35     12.35     12.35     12.35
11 12.352     12.36     12.36     12.35     12.35     12.35     12.35     12.35
12 12.353     12.36     12.36     12.35     12.35     12.35     12.35     12.35
13 12.354     12.36     12.36     12.35     12.35     12.35     12.35     12.35
14 12.355     12.36     12.36     12.35     12.35     12.35     12.36     12.36
15 12.356     12.36     12.36     12.35     12.35     12.36     12.36     12.36
16 12.357     12.36     12.36     12.35     12.35     12.36     12.36     12.36
17 12.358     12.36     12.36     12.35     12.35     12.36     12.36     12.36
18 12.359     12.36     12.36     12.35     12.35     12.36     12.36     12.36
19 12.360     12.36     12.36     12.36     12.36     12.36     12.36     12.36
20 12.361     12.37     12.37     12.36     12.36     12.36     12.36     12.36
21 12.362     12.37     12.37     12.36     12.36     12.36     12.36     12.36
22 12.363     12.37     12.37     12.36     12.36     12.36     12.36     12.36
23 12.364     12.37     12.37     12.36     12.36     12.36     12.36     12.36
24 ------------------------------------------------------------------------------
25 -2.345     -2.34     -2.35     -2.35     -2.34     -2.34     -2.34     -2.35
26 -2.346     -2.34     -2.35     -2.35     -2.34     -2.35     -2.35     -2.35
27 -2.347     -2.34     -2.35     -2.35     -2.34     -2.35     -2.35     -2.35
28 -2.348     -2.34     -2.35     -2.35     -2.34     -2.35     -2.35     -2.35
29 -2.349     -2.34     -2.35     -2.35     -2.34     -2.35     -2.35     -2.35
30 -2.350     -2.35     -2.35     -2.35     -2.35     -2.35     -2.35     -2.35
31 -2.351     -2.35     -2.36     -2.36     -2.35     -2.35     -2.35     -2.35
32 -2.352     -2.35     -2.36     -2.36     -2.35     -2.35     -2.35     -2.35
33 -2.353     -2.35     -2.36     -2.36     -2.35     -2.35     -2.35     -2.35
34 -2.354     -2.35     -2.36     -2.36     -2.35     -2.35     -2.35     -2.35
35 -2.355     -2.35     -2.36     -2.36     -2.35     -2.35     -2.36     -2.36
36 -2.356     -2.35     -2.36     -2.36     -2.35     -2.36     -2.36     -2.36
37 -2.357     -2.35     -2.36     -2.36     -2.35     -2.36     -2.36     -2.36
38 -2.358     -2.35     -2.36     -2.36     -2.35     -2.36     -2.36     -2.36
39 -2.359     -2.35     -2.36     -2.36     -2.35     -2.36     -2.36     -2.36
40 -2.360     -2.36     -2.36     -2.36     -2.36     -2.36     -2.36     -2.36
41 -2.361     -2.36     -2.37     -2.37     -2.36     -2.36     -2.36     -2.36
42 -2.362     -2.36     -2.37     -2.37     -2.36     -2.36     -2.36     -2.36
43 -2.363     -2.36     -2.37     -2.37     -2.36     -2.36     -2.36     -2.36
44 -2.364     -2.36     -2.37     -2.37     -2.36     -2.36     -2.36     -2.36
45 ------------------------------------------------------------------------------
46 BigDecimal02: Das war's erstmal!
```

Aufgabe: Erfinden Sie **weitere Rundungsarten** für **BigDecimal**-Objekte und schreiben Sie entsprechende Methoden.

Aufgabe: Schreiben Sie eine verbesserte Version des **Tabellenkalkulations-Programms** Excel, bei der der Benutzer wählen kann, ob Zahlen als primitive Werte des Typs **double** (wie in der verbreiteten Version von Excel) oder als Objekte der Klasse **BigDecimal** realisiert werden sollen. Als **Voreinstellung** sollten sie **BigDecimal** einbauen.