

Lehrveranstaltung MB1-PR1 (Programmieren 1) an der Beuth Hochschule für Technik Berlin

Datei 14WS_PR1_Aufgaben

Hinweise auf Fehler, Verbesserungsvorschläge oder Berichte über Erfahrungen mit diesen Übungen sind jederzeit willkommen, am liebsten per e-mail an die Adresse grude@beuth-hochschule.de.

Inhaltsverzeichnis

Stil-Regeln (für Java-Programmtexte)	2
Aufgabe 1: Grundlagen	4
Aufgabe 2: Rechnen mit Ganzzahlen	6
Aufgabe 3: Rechnen mit Bruchzahlen	8
Aufgabe 4: Dreieck	10
4.1. Definitionen und Erläuterungen	10
4.2. Die Tabelle mit Testbeispielen	10
4.3. Die Klasse Tripel	11
Aufgabe 5: In Worten	12
Aufgabe 6: Schleifen	14
6.1. Die 2 Funktionen zeile und spalte	14
6.2. Die 4 Funktionen schach01 bis schach04	14
6.3. Die Funktion schach05	15
6.4. Die Funktion gitter	16
Aufgabe 7: StringBuilder (Florian)	17
Aufgabe 8: Reihungen und Sammlungen (Veronika01)	18
Aufgabe 9: Reihungen und Sammlungen (Veronika02)	19
Aufgabe 10: Klasse Punkt3D	21
Aufgabe 11: Klasse Quader etc.	22
Aufgabe 12: Klammern prüfen	23
Aufgabe 13: Miez, eine noch ziemlich kleine Katze	25

Stil-Regeln (für Java-Programmtexte)

1. Namen von Klassen, Schnittstellen, Variablen und Methoden

1.1. Namen von *Klassen* und *Schnittstellen* beginnen mit einem *großen* Buchstaben (z.B. Dreiseit, Math, String). Namen von *Variablen* und *Methoden* beginnen mit einem *kleinen* Buchstaben (z.B. otto, summe, print, size).

1.2. Kamel-Notation: Besteht ein Name aus *mehreren* Worten, so beginnt das zweite und jedes weitere Wort mit einem *großen* Buchstaben (z.B. Klassennamen OutputStreamWriter, InWorten, Namen von Variablen und Methoden: charAt, deepToString, endErgebnis, machNochWas).

1.3. Namen von *unveränderlichen Variablen* (names of final fields and final local variables) bestehen nur aus Großbuchstaben, Ziffern und Unterstrichen (z.B. PI, E, MAX_VALUE, MWST, MWST2008). Namen von *Paketen* bestehen nur aus kleinen Buchstaben und Ziffern (z.B. java, lang, erben, erben3d).

2. Einrücktiefe und allgemeine Einrückregeln

2.1. Pro Stufe wird um *3 Zeichen* eingerückt (*nicht* mit Tab-Zeichen, sondern mit Blanks).

2.2. Die erste und letzte Zeile einer Klassenvereinbarung werden *nicht eingerückt*. Alle Zeilen zwischen der ersten und letzten werden um mindestens eine Stufe eingerückt, etwa so:

```

1 class Karola {                               // Erste Zeile
2     int otto = 17;                            //   1 Stufe  eingerueckt
3     void gibOttoAus() {                       //   1 Stufe  eingerueckt
4         System.out.println("Jetzt kommt otto:"); //   2 Stufen eingerueckt
5         System.out.println("   " + otto);     //   2 Stufen eingerueckt
6     }                                         //   1 Stufe  eingerueckt
7 }                                           // Letzte Zeile

```

2.3. Kommentarzeilen werden so eingerückt wie die Zeile *danach*, etwa so:

```

1 // Richtig eingerueckte Kommentarzeilen:
2 // ----- richtig
3 int summiere(int n1, int n2) {
4     // Liefert die Summe von n1 und n2
5     return n1 + n2;
6 }
7 // ----- richtig

1 // Falsch eingerueckte Kommentarzeilen:
2 // ----- falsch
3 int summiere(int n1, int n2) {
4     // Liefert die Summe von n1 und n2
5     return n1 + n2;
6 }
7 // ----- falsch

```

2.4. package- und import-Befehle werden *nicht eingerückt*.

3. Format von if-Anweisungen

3.1. Kurze if-Anweisungen (ohne else-Teil und mit nur *einer* einfachen Anweisung im Rumpf) können (und sollen) *ohne* geschweifte Klammern *in einer einzigen Zeile* notiert werden, etwa so:

```

1     if (a < b) break;
2     if (n%2==0) printf("n ist eine gerade Zahl");

```

3.2. In allen anderen Fällen wird der then-Rumpf und der else-Rumpf (falls vorhanden) in geschweifte Klammern eingefasst und um eine Stufe mehr eingerückt, als die erste Zeile der if-Anweisung:

```

3     if (a < b) {
4         printf("Nur eine Anweisung im then-Rumpf!");
5     }
6

```

```
7     if (a <= b+2) {
8         a = a+3;    // then-Rumpf, 1 Stufe eingerueckt
9         b = a-3;    // then-Rumpf, 1 Stufe eingerueckt
10    } else {        // Nur eine Trennzeile zwischen then- und else-Rumpf
11        a = a-3;    // else-Rumpf, 1 Stufe eingerueckt
12        b = b+3;    // else-Rumpf, 1 Stufe eingerueckt
13        printf("A"); // else-Rumpf, 1 Stufe eingerueckt
14    }
```

4. Format von Schleifen

4.1. Enthält der Rumpf einer Schleife nur *eine* kurze Anweisung, kann (und soll) die ganze Schleife ohne geschweifte Klammern auf *einer einzigen Zeile* notiert werden, etwa so:

```
15     while (i<tab.length) pln(tab[i]);
16     for (int n=1; n<=10; n++) if (tab[i]!=0) p(n + " ");
```

4.2. In allen anderen Fällen wird der Rumpf der Schleife in geschweifte Klammern eingefasst und um eine Stufe mehr eingerückt als die erste Zeile der Schleife, etwa so:

```
17     while (true) {
18         int n = EM.liesInt();
19         if (n==0) break;
20         pln(n);
21     }
22
23     for (int n=2; n<=20; n+=2) {
24         pln("Hier kommt noch eine gerade Zahl: " + n);
25     }
```

5. Format von switch-Anweisungen

5.1. Sind die Anweisungsfolgen für die einzelnen Fälle (engl. cases) sehr kurz, kann (und soll) jeder Fall auf *einer einzigen Zeile* notiert werden, etwa so:

```
26     switch(n) {
27         case 1:  erg="ein";      break;
28         case 2:  erg="zwei";     break;
29         default: erg="Fehler12"; break;
30     }
```

5.2. In allen anderen Fällen ist eine switch-Anweisung folgendermaßen zu formatieren:

```
31     switch(n) {
32         case 1:
33             pln("ein");
34             anz += 3;
35             break;
36         case 2: case 3: case 4: case 5:
37             pln("groesser als eins");
38             anz += 4;
39             break;
40         default:
41             pln("-----");
42             pln("Fehler12");
43             pln("-----");
44             anz += 9;
45             break;
46     }
```

6. Ausnahme-Regel

Von diesen Regeln darf man (in seltenen Fällen) abweichen, wenn der Programmtext dadurch leichter lesbar oder besser verständlich wird.

Aufgabe 1: Grundlagen

Team-Name: Namen der Team-Mitglieder:

1.1. Nennen Sie alle Tätigkeiten des *Programmierers* und des *Ausführers*. Es genügt, wenn Sie die entsprechenden Tätigkeitsworte im Infinitiv aufzählen (z.B. laufen, essen, schlafen).

1.2. Zerlegen Sie die Zahl 918 in ihre *Primfaktoren*.

1.3. Die erste Zeile der folgenden Tabelle enthält eine Ganzzahl dargestellt in fünf verschiedenen Zahlensystemen. Füllen Sie die leeren Kästchen der übrigen Zeilen aus (siehe dazu auch das Papier **ZahlenSysteme.pdf**):

2-er-System (binär)	7-er-System (septal)	8-er-System (oktal)	10-er-System (dezimal)	16-er-System (hexadezimal)
10001	23	21	17	11
10101				
	36			
		36		
			255	
				A1

1.4. Die erste Zeile der folgenden Tabelle enthält eine Bruchzahl dargestellt in zwei verschiedenen Zahlensystemen. Füllen Sie die leeren Kästchen aus. Berechnen Sie für die binäre Darstellung der dezimalen Bruchzahl 0.1 mindestens 10 Binärstellen nach dem Punkt.

2-er-System (binär)	10-er-System (dezimal)
11.11	3.75
0.1	
0.01	
0.001	
0.0001	
101.0101	
	2.5
	7.25
	8.75
	5.5625
	0.1

Die Teilaufgaben **1.5.** bis **1.10.** stehen auf der folgenden Seite!

1.5. Lesen Sie das Papier **DivUndModFuerGanzzahlen.pdf** durch, ehe Sie die folgende Tabelle ausfüllen:

dend	dor	div LL	div RR	div LR	div RL	mod LL	mod RR	mod LR	mod RL
+7	+2								
-7	-2								
+7	-2								
-7	+2								

1.6. Sei n irgendeine im 10-er-System dargestellte Ganzzahl (z.B. 123 oder -7385 oder 0 etc.). Welcher der folgenden Ausdrücke beschreibt die letzte Ziffer von n (d.h. die Ziffer an der Stelle 0, die mit dem Stellenwert 10^0)? Dabei bezeichnet der Schrägstrich / die *Ganzzahldivision* div_{RL} und das Prozentzeichen % die *Restoperation* mod_{RL} .

A: $n / 100$

B: $n - n / 10$

C: $n \% 10$

D: $n / 10$

1.7. Sei n irgendeine im 10-er-System dargestellte Ganzzahl.

Welche der folgenden Zuweisungen *entfernt* die letzte Ziffer aus n (d.h. die Ziffer an der Stelle 0, die mit dem Stellenwert 10^0)? Oder anhand von Beispielen erläutert: Welche der Zuweisungen ändert den Wert von n von 123 zu 12 oder von -758 zu -75 etc. ?

A: $n = n / 100$

B: $n = n - n / 10$

C: $n = n \% 10$

D: $n = n / 10$

1.8. Sei n irgendeine im 10-er-System dargestellte Ganzzahl.

Wie kann man die letzte Ziffer von n berechnen (d.h. die Ziffer an der Stelle 0, die mit dem Stellenwert 10^0)? Ebenso für die Ziffer mit dem Stellenwert 10^6 (die Ziffer an der Stelle 6) ?

1.9. Sei n irgendeine im 2-er-System dargestellte Ganzzahl (z.B. 101 oder -11010011 oder 0 etc.).

Wie kann man die letzte Ziffer von n berechnen (d.h. die Ziffer an der Stelle 0, die mit dem Stellenwert 2^0) ? Ebenso für die Ziffer mit dem Stellenwert 2^6 (die Ziffer an der Stelle 6) ?

1.10. Sei n irgendeine im 5-er-System dargestellte Ganzzahl (z.B. 142 oder -4301 oder 0 etc.).

Wie kann man die letzte Ziffer von n berechnen (d.h. die Ziffer an der Stelle 0, die mit dem Stellenwert 5^0) ? Ebenso für die Ziffer mit dem Stellenwert 5^6 (die Ziffer an der Stelle 6) ?

Aufgabe 2: Rechnen mit Ganzzahlen

Team-Name:	Namen der Team-Mitglieder:
------------	----------------------------

Schreiben Sie ein Java-Programm namens `GanzRech`, welches wiederholt von der Standardeingabe zwei Ganzzahlen in zwei `int`-Variable `g1` und `g2` einliest und die folgenden Zeilen ausgibt (die Eingaben des Benutzers sind fett hervorgehoben):

```
Bitte zwei Ganzzahlen g1 und g2 eingeben: 17 -5
g1      ist gleich 17
g2      ist gleich -5
g1 + g2 ist gleich 12
g1 - g2 ist gleich 22
g1 * g2 ist gleich -85
g1 / g2 ist gleich -3
g1 % g2 ist gleich 2
```

Erforschen Sie mit Hilfe dieses Programms, wie Ihr Java-Ausführer mit Ganzzahlen (vom Typ `int`) rechnet. Stellen Sie fest, wie groß der kleinste `int`-Wert und der größte `int`-Wert sind (es ist nicht verboten, in Büchern oder im Internet nachzusehen). Diese Werte werden im folgenden kurz `MIN` und `MAX` genannt.

Berechnen Sie mit Ihrem Programm `GanzRech` (**nicht** im Kopf oder mit einem Taschenrechner!) die Werte der folgenden Ausdrücke und tragen Sie sie in die Tabelle ein:

1. MIN		2. MAX	
3. MIN - 1		4. MAX + 1	
5. MIN - 2		6. MAX + 2	
7. MIN - 10		8. MAX + 10	
9. MIN - MIN		10. MAX + MAX	
11. MAX / -1		12. MIN / -1	
13. 10_000*10_000		14. 100_000*100_000	
15. 14 / 5		16. 14 % 5	
17. -14 / 5		18. -14 % 5	
19. 14 / -5		20. 14 % -5	
21. -14 / -5		22. -14 % -5	
23. 123 / 0			

24. Ungefähr wie viele positive `int`-Werte kennt Ihr Java-Ausführer?

Geben Sie diese Anzahl **auf 3 Ziffern genau** an

(z.B. *25.5 Millionen* oder *5.32 Trilliarden* oder so ähnlich).

25. Ungefähr wie viele `int`-Werte (positive und negative zusammen) kennt Ihr Java-Ausführer?

Geben Sie diese Anzahl **auf 2 Ziffern genau** an

(z.B. *58 Millionen* oder *7.8 Trillionen* oder so ähnlich).

26. Ist Ihnen aufgefallen, dass Ihr Programm `GanzRech` für die Rechenaufgabe 14. (siehe oben) ein falsches Ergebnis ausgegeben hat?

Füllen Sie in der folgenden Tabelle die drei leeren Kästchen (in der rechten Spalte) aus:

14. Rechenaufgabe (mit <code>int</code> -Werten)	100_000*100_000
Mathematisch korrektes Ergebnis KE (auf 2 Ziffern genau)	
Tatsächliches Ergebnis TE Ihres Programms <code>GanzRech</code> (auf 2 Ziffern genau)	
Formel die erklärt, wie der Java-Ausführer zum tatsächlichen (aber falschen) Ergebnis kommt. Ihre Formel darf die Variablen KE und TE und Ihre Lösung der vorigen Teilaufgabe (25.) enthalten.	

Aufgabe 3: Rechnen mit Bruchzahlen

Team-Name:	Namen der Team-Mitglieder:
------------	----------------------------

Schreiben Sie ein Java-Programm namens `BruchRech`, welches wiederholt von der Standardeingabe zwei Bruchzahlen in zwei `float`-Variablen `b1` und `b2` einliest und die folgenden Zeilen ausgibt (die Eingaben des Benutzers sind fett hervorgehoben):

```
Bitte zwei Bruchzahlen b1 und b2 eingeben: 10.5 -3.0
b1      ist gleich 10.5
b2      ist gleich -3.0
b1 + b2 ist gleich 7.5
b1 - b2 ist gleich 13.5
b1 * b2 ist gleich -31.5
b1 / b2 ist gleich -3.5
b1 % b2 ist gleich 1.5
```

Erforschen Sie mit Hilfe dieses Programms, wie Ihr Java-Ausführer mit Bruchzahlen (vom Typ `float`) rechnet. Stellen Sie fest, wie groß der größte `float`-Wert und der kleinste positive `float`-Wert (d.h. der kleinste `float`-Wert oberhalb von 0) ungefähr sind (oder schauen Sie irgendwo nach, z.B. im Buch "Java ist eine Sprache" oder im Internet). Diese Werte werden im folgenden mit `MIN` und `MAX` bezeichnet.

Berechnen Sie mit Ihrem Programm `BruchRech` (**nicht** im Kopf oder mit einem Taschenrechner!) die Werte der folgenden Ausdrücke und tragen Sie sie in die Tabelle ein.

Einige der Ausdrücke enthalten die Namen `ip` bzw. `in`. Sie sollen anstelle von `ip` irgendeine positive `float`-Zahl (eine Zahl größer als 0) angeben und anstelle von `in` irgendeine negative `float`-Zahl (eine Zahl kleiner als 0) angeben.

Untersuchen Sie, wie der Wert eines Ausdrucks (z.B. der Wert des Ausdrucks `ip / -0`, siehe Teilaufgabe 19) sich ändert, wenn Sie für `ip` verschiedene (positive `float`-) Zahlen einsetzen.

1) MAX		2) MIN	
3) MAX * 1.001		4) -MAX * 1.001	
4) 3.5e38 + 0		6) -3.5e38 + 0	
7) Infinity * ip		8) ip * -Infinity	
9) ip * 0		10) 0 * ip	
11) Infinity * 0		12) 0 * -Infinity	
13) Infinity / ip		14) -Infinity / ip	
15) ip / Infinity		16) ip / -Infinity	
17) Infinity / Infinity		18) -Infinity / -Infinity	
19) ip / -0		20) in / -0	
21) ip / +0		22) in / +0	
23) -0 / ip		24) -0 / in	
25) +0 / ip		26) +0 / in	
27) +0 / +0		28) +0 / -0	
29) 1.4e-45 + 0		30) -1.4e-45 + 0	
31) 0.8e-45 + 0		32) -0.8e-45 + 0	
33) 0.7e-45 + 0		34) -0.7e-45 + 0	

Ergänzen Sie die folgenden Formulierungen

(die Zahlen an den Zeilenanfängen wie 7), 9), 11) etc. beziehen sich auf die *vorige Tabelle*):

7) <i>Infinity</i> mal irgendwas positives ist gleich
9) Irgendwas positives mal 0 ist gleich
11) <i>Infinity</i> mal 0 ist gleich

Beantworten Sie sich selbst: Was müsste bei 11) herauskommen auf Grund von Regel 7)? Was müsste bei 11) herauskommen auf Grund von Regel 9)? Was kommt bei 11) tatsächlich heraus?

13) <i>Infinity</i> durch irgendwas positives ist gleich
15) Irgendwas positives durch <i>Infinity</i> ist gleich
17) <i>Infinity</i> durch <i>Infinity</i> ist gleich

Beantworten Sie sich selbst: Was müsste bei 17) herauskommen auf Grund von Regel 13)? Was müsste bei 17) herauskommen auf Grund von Regel 15)? Was kommt bei 17) tatsächlich heraus?

Drücken Sie Ihre Erkenntnisse aus den Zeilen 21), 25) und 27) der obigen Tabelle durch entsprechende Formulierungen aus (so ähnlich wie z.B. "Minus Unendlich durch irgendetwas negatives ist *Infinity*"):

21) ?
25) ?
27) ?

Beantworten Sie sich selbst: Was müsste bei 27) herauskommen auf Grund von Regel 21)? Was müsste bei 27) herauskommen auf Grund von Regel 25)? Was kommt bei 27) tatsächlich heraus?

Um (als Höhepunkt und Ziel dieser Aufgabe) die letzte Teilaufgabe lösen zu können, müssen Sie das abstrakte Prinzip hinter den obigen "Dreiergruppen" {7), 9), 11)} und {13), 15), 17)} und {21), 25), 27)} verstanden haben (es ist immer das selbe Prinzip). Man versteht dieses Prinzip leichter, wenn man sich die Fragen nach dem Text **Beantworten Sie sich selbst** wirklich selbst beantwortet (und mit seiner GruppenpartnerIn diskutiert). Wenden Sie sich an die BetreuerIn Ihrer Übungsgruppe, wenn Sie diese Fragen nicht verstehen (aber erst, wenn Sie mindestens 3 Minuten lang versucht haben, ohne Ihre BetreuerIn klarzukommen).

28) Erklären Sie allgemein, aber kurz und *genau*, wann der Ausführer das Ergebnis NaN ("not a number") liefert. Zählen Sie nicht viele Einzelfälle auf, sondern geben Sie eine einfache Regel an, die alle Fälle abstrakt aber einsichtig beschreibt. Eine unpräzise Lösung wie "Der Ausführer liefert NaN, wenn das Ergebnis einer Berechnung nicht definiert ist" ist auf jeden Fall falsch (und hat zu wenig mit den obigen "Dreiergruppen" {7), 9), 11)} und {13), 15), 17)} und {21), 25), 27)} und den darunter stehenden Fragen zu tun). Ihre Antwort sollte sich bequem in den folgenden Kasten schreiben lassen:

--

Aufgabe 4: Dreieck

4.1. Definitionen und Erläuterungen

Die Seiten a , b und c eines *Dreiecks* müssen alle echt *größer als 0* sein und müssen die folgenden drei *Dreiecksungleichungen* erfüllen:

$$a < b + c$$

$$b < c + a$$

$$c < a + b$$

Ein *Dreieck* ist ein "schwach verallgemeinertes Dreieck". Die Seiten a , b und c eines Dreiecks müssen *größer oder gleich 0* sein und die folgenden drei *Dreiecksungleichungen* erfüllen:

$$a \leq b + c$$

$$b \leq c + a$$

$$c \leq a + b$$

Allgemein gilt: Jedes Dreieck ist auch ein Dreieck. Aber es gibt Dreiecke, die keine Dreiecke sind (z.B. das Dreieck mit den Seiten 1, 2, 1 oder das Dreieck mit den Seiten 3, 3, 0 oder das punktförmige Dreieck mit den Seiten 0, 0, 0).

Wie ein Dreieck kann auch ein Dreieck folgende Eigenschaften haben bzw. nicht haben:

<i>gleichseitig</i>	alle drei Seiten sind gleich	z.B. 5, 5, 5 oder 0, 0, 0
<i>gleichschenkelig</i>	mindestens zwei Seiten sind gleich	z.B. 5, 3, 5 oder 2, 2, 0
<i>schief</i>	alle drei Seiten sind verschieden lang	z.B. 4, 5, 6 oder 7, 1, 4
<i>rechtwinklig</i>	es gilt $a^2 = b^2 + c^2$ oder $b^2 = c^2 + a^2$ oder $c^2 = a^2 + b^2$	z.B. 3, 4, 5 oder 8, 6, 10

Zwei wichtige Festlegungen:

- Ein Dreieck mit genau *einer* 0-Seite (z.B. 3, 3, 0 oder 7, 0, 7 etc.) soll als *rechtwinklig* gelten. Aber Vorsicht: Z.B. stellt das Zahlentripel 3, 4, 0 gar kein Dreieck dar und somit auch kein rechtwinkliges).
- Das *punktförmige* Dreieck (mit den Seiten 0, 0, 0) soll als *gleichseitig* und *gleichschenkelig*, aber *nicht* als rechtwinklig gelten (weil alle Winkel eines gleichseitigen Dreiecks 60° und nicht 90° groß sind).

Wichtiger Hinweis: Wenn drei Ganzzahlen *kein Dreieck* beschreiben (wie z.B. die Zahlen 2, 2, 5), dann können sie natürlich auch kein gleichseitiges oder gleichschenkeliges oder schiefes oder rechtwinkliges oder nach-frischem-Kaffee-duftendes Dreieck darstellen.

4.2. Die Tabelle mit Testbeispielen

Bevor Sie weiterlesen, sollten Sie die folgende Tabelle ausfüllen (mit `true` bzw. `false` oder mit T bzw. F) :

Seiten --->	0,0,0	1,2,3	3,5,4	4,3,5	2,2,0	2,0,2	5,2,2	2,0,3	0,3,2
dreieck?	true								
gleichseitig?	true								
gleichschenkelig?	true								
schief?	false								
rechtwinklig?	false								

Nicht weiter blättern, bevor Sie die Tabelle ausgefüllt haben! Zeigen Sie die Tabelle möglichst der BetreuerIn Ihrer Übungsgruppe, bevor Sie weitermachen. Falls Sie mit Ihrer BetreuerIn noch Fehler in der Tabelle finden, können Sie sich durch diese Vorgehensweise *viel unnötige Arbeit ersparen*.

4.3. Die Klasse Tripel

1. Kopieren Sie die Datei `Tripel.java`. in ein neues Verzeichnis namens `Aufgabe4`. Übergeben Sie die Datei dem Java-Ausführer und lassen Sie das Programm `Tripel` ausführen.

2. Kopieren Sie die Datei `TripelJut.java`. ebenfalls in das Verzeichnis `Aufgabe3`. Übergeben Sie die Datei dem Java-Ausführer und lassen Sie das Programm `TripelJut` ausführen. Das Programm `TripleJut` testet die Klasse `Tripel` (in der Datei `Triple.java`). Es wird jetzt Fehler feststellen und einen roten Balken zeigen.

3. Verbessern Sie die Datei `Tripel.java`. Suchen Sie dazu den Konstruktor (auch wenn Sie nicht wissen, was ein Konstruktor ist, können Sie ihn an seinen *Kommentaren* erkennen). Suchen Sie im Konstruktor die folgenden 5 Zeilen:

```
57     IST_DREISEIT      = false;
58     IST_GLEICHSEITIG  = false;
59     IST_GLEICHSCHEKLI = false;
60     IST_SCHIEF        = false;
61     IST_RECHTWINKLIG  = false;
```

Ersetzen Sie das erste `false` durch einen boolean Ausdruck, der genau dann den Wert `true` hat, wenn die Zahlen `A`, `B` und `C` ein Dreieck bilden.

Ersetzen Sie die übrigen vier `false`-Literals ganz entsprechend durch geeignete boolean Ausdrücke.

Testen Sie die verbesserte Klasse `Triple` erneut, indem Sie das Testprogramm `TripleJut` erneut ausführen lassen (sie brauchen es *nicht* erneut zu übergeben, d.h. zu compilieren).

Anforderung 1: Das Testprogramm `TripleJut` muss *einen grünen Balken zeigen* (sonst müssen Sie ihre boolean Ausdrücke noch verbessern).

Anforderung 2: Ihre boolean Ausdrücke dürfen *keine überflüssigen Teilausdrücke* enthalten.

Beispiele für boolean Ausdrücke *mit überflüssigen* Teilausdrücken:

```
(A != B) && true      // && true ist überflüssig
(A == B) || false    // || false ist überflüssig
A<B && B<C && A<C     // && A<C ist überflüssig
A>0 && B>0 && A+B>0   // && A+B>0 ist überflüssig
```

Lassen Sie die BetreuerIn Ihrer Übungsgruppe prüfen, ob Sie die **Anforderung 2** erfüllt haben.

Wenn Ihre Klasse `Triple` beide Anforderungen gleichzeitig erfüllt, haben Sie diese Aufgabe gelöst.

Anmerkung: Das Testprogramm `TripleJut` wurde mit dem Rahmenprogramm (engl. framework) `JUnit` (Version 3.8, nicht Version 4.0 oder 4.2 etc.) erstellt. Hier sollen Sie üben, ein solches Testprogramm bei der Entwicklung eines Programms zu *benutzen*. Später sollen Sie dann selbst solche Testprogramme *schreiben*.

Aufgabe 5: In Worten

Vereinbaren Sie eine Klasse namens `InWorten` und darin vier Methoden, die den folgenden Spezifikationen entsprechen. Achtung: *liefern* oder *zurückgeben* ist etwas ganz anderes als *ausgeben*!

```

1  static public String inWorten_1_9(int n) {
2      // Wenn n zwischen 1 und 9 liegt, wird das entsprechende deutsche,
3      // maennliche Zahlwort als Ergebnis geliefert, und sonst
4      // eine Fehlermeldung (die u.a. den Namen der Methode enthaelt).
5      ...
6  }
7
8  static public String inWorten_10_19(int n) {
9      // Wenn n zwischen 10 und 19 liegt, wird das entsprechende deutsche
10     // Zahlwort geliefert, und sonst eine Fehlermeldung (die u.a.den Namen
11     // der Methode enthaelt).
12     ...
13 }
14
15 static public String inWorten_20_90(int n) {
16     // Wenn n zwischen 2 und 9 liegt, wird das entsprechende deutsche
17     // Wort fuer die Zahl n * 10 geliefert, und sonst eine Fehlermeldung
18     //(die u.a.den Namen der Methode enthaelt).
19     ...
20 }
21
22 static public String inWorten_0_999(int n) {
23     // Wenn n zwischen 0 und 999 liegt, wird das entsprechende deutsche,
24     // maennliche Zahlwort geliefert, und sonst eine Fehlermeldung (die
25     // u.a.den Namen der Methode enthaelt).
26     ...
27 }

```

Als Beispiele folgen hier ein paar deutsche, männliche Zahlworte:

```

null
ein           // maennlich, wie in "ein Euro", nicht "eine DM"
zwei
drei
vier
fuenf        // ue statt ü !
...
sechzehn     // nicht "sechszehn" !
siebzehn
...
einundzwanzig
zweiundzwanzig
...
einhundert
einhundertundein // maennlich, wie in "einhundertundein Euro"
...
fuenfhundertundein // maennlich, wie in "fuenfhundertundein Dollar"
...
neunhundertundneunundneunzig // nicht: "neunhundertneunundneunzig"

```

Anforderung 1: Entwickeln Sie die vier Methoden *in der angegebenen Reihenfolge*.

Anforderung 2: In der letzten Methode (`inWorten_0_999`) sollen Sie die drei vorher entwickelten Methoden *aufrufen* (und *nicht* alles, was diese Methoden leisten, *noch mal* programmieren).

Anforderung 3: Lassen Sie immer, wenn Sie eine Methode fertig geschrieben haben, das JUnit-Testprogramm `InWortenJut` ausführen. Wenn das Testprogramm in Ihrer neuen Methode noch Fehler findet, sollten Sie die unbedingt verbessern, ehe Sie eine weitere Methode in Angriff nehmen. Sie haben die Aufgabe gelöst, wenn Sie alle vier Methoden programmiert haben und das Testprogramm `InWortenJut` einen grünen Balken zeigt.

Anforderung 4: Beim Lösen dieser Aufgabe sollen Sie vor allem `if`-Anweisungen und `switch`-Anweisungen verwenden, aber *keine Reihungen* (arrays) und *keine Sammlungen* (collections), auch wenn Sie mit diesen Konstrukten schon vertraut sind.

Tipp 1: Es folgt hier ein Vorschlag, wie man die letzte (und umfangreichste) Methode strukturieren sollte. Wenn Sie möchten, wird dieser Tipp in den Übungen näher erläutert.

```
1  static public String inWorten_0_999(int n) {
2      // Wenn n zwischen 0 und 999 liegt, wird das entsprechende deutsche,
3      // maennliche Zahlwort geliefert, und sonst eine Fehlermeldung.
4
5      // 1. Wenn n falsch ist, eine Fehlermeldung liefern (nicht ausgeben!)
6      ...
7
8      // 2. Den Sonderfall "n ist gleich 0" behandeln:
9      ...
10
11     // 3. Ein paar Variablen vereinbaren und
12     // die einzelnen Dezimalziffern von n berechnen:
13     final int e = n / 1 % 10; // Einerziffer
14     final int z = n / 10 % 10; // Zehnerziffer
15     final int h = n / 100 % 10; // Hunderterziffer
16     final int ze = n % 100; // Zehner- und Einerziffer zusammen
17     String erg = ""; // Fuer das Ergebnis dieser Funktion
18
19     // 4. Die Hunderterziffer h bearbeiten,
20     // mit "hundert" und evtl "und" dahinter
21     ...
22
23     // 5. Die Sonderfaelle "ze liegt zwischen 10 und 19" bearbeiten:
24     ...
25
26     // 6. Die Einerziffer e bearbeiten, evtl "und" anhaengen:
27     ...
28
29     // 7. Die Zehnerziffer z bearbeiten:
30     ...
31 } // inWorten_0_999
```

Tipp 2: Wenn Sie die Methode `inWorten_0_999` programmieren und an eine Stelle kommen, an der Sie das Ergebnis der Methode fertig berechnet haben, sollten Sie es sofort mit `return` zurück liefern (statt es noch lange in einer Variablen aufzubewahren und erst später zurück zu liefern). Dadurch werden Ihre Befehle einfacher und leichter verstehbar. Beispiel:

```
32 // Nicht so gut:
33 if (n == 0) erg = "null";
34 ...
35 ...
36 return erg;
37
38
39 // Einfacher:
40 if (n == 0) return "null";
```

Tipp 3: (Für Fortgeschrittene) Wenn Sie mit der Methode `inWorten_0_999` alle eintausend Worte für die Zahlen von 0 bis 999 erzeugen und (jedes Wort auf einer neuen Zeile) in eine Datei schreiben, muss diese Datei (unter Windows) genau **27047 Bytes** lang sein (unter Unix/Linux oder Mac OS: **26047 Bytes**). Um das zu prüfen, können Sie die Klasse `InWorten` mit einer `main`-Methode versehen, darin die Zahlworte mit dem Befehl `System.out.println` zur Standardausgabe ausgeben lassen und die Ausgabe wie folgt in eine Datei umleiten, z.B. in eine Datei namens `tmp`:

```
> java InWorten > tmp
```

Aufgabe 6: Schleifen

Schreiben Sie eine öffentliche Klasse namens `Schleifen`, die die folgenden 8 Funktionen enthält:

```

1 static public                                // Liefert
2 String zeile();                             // eine Zeile (A1..A8)
3 static public
4 String spalte();                             // eine Spalte (A1..H1)
5 static public
6 String schach01();                          // ein Schachbrettmuster (A1..H8)
7 static public
8 String schach02();                          // ein Schachbrettmuster (H1..A8)
9 static public
10 String schach03();                         // ein Schachbrettmuster (H8..A1)
11 static public
12 String schach04();                         // ein Schachbrettmuster (A8..H1)
13 static public
14 String schach05(String s1, String s2); // diverse Schachbrettmuster
15 static public
16 String gitter(int spa, int zeil);         // diverse Gittermuster

```

Diese Funktionen sollen in `StringBuilder`-Objekten bestimmte Zeichenketten zusammenbauen und dann (mit `return`) als `String`-Ergebnis liefern (*nicht* ausgeben!). Dadurch wird nicht ausgeschlossen, dass Sie z.B. in einer `main`-Methode einige der Funktionen aufrufen und ihre Ergebnisse ausgeben lassen, z.B. so: `println(schach03());`

Anforderung: Alle Funktionen müssen ihre Ergebnisse *mit Hilfe von Schleifen berechnen* und dürfen sie nicht schon als `String`-Literele oder auf ähnliche Weise enthalten.

6.1. Die 2 Funktionen `zeile` und `spalte`

Wenn man das Ergebnis der Funktion `zeile` zum Bildschirm ausgibt, soll es dort wie folgt aussehen:

```
A1 A2 A3 A4 A5 A6 A7 A8
```

Diese Zeile endet mit *zwei transparenten Zeichen*: Einem Blank ' ' und einem Zeilenwechsel-Zeichen '\n'. Daraus folgt, dass das Ergebnis der Funktion `zeile` ein `String` der Länge $(8 \cdot 3 + 1)$ gleich 25 sein muss.

Wenn man das Ergebnis der Funktion `spalte` zum Bildschirm ausgibt, soll es dort wie folgt aussehen:

```
A1
B1
C1
D1
E1
F1
G1
H1
```

Auch hier endet jede der 8 Zeilen mit einem Blank ' ' und einem Zeilenwechsel-Zeichen '\n'. Daraus folgt, dass das Ergebnis der Funktion `spalte` ein `String` der Länge $(8 \cdot 4)$ gleich 32 sein muss.

6.2. Die 4 Funktionen `schach01` bis `schach04`

Wenn man das Ergebnis der Funktion `schach01` zum Bildschirm ausgibt, soll es dort so aussehen:

```
A1 A2 A3 A4 A5 A6 A7 A8
B1 B2 B3 B4 B5 B6 B7 B8
C1 C2 C3 C4 C5 C6 C7 C8
D1 D2 D3 D4 D5 D6 D7 D8
E1 E2 E3 E4 E5 E6 E7 E8
F1 F2 F3 F4 F5 F6 F7 F8
G1 G2 G3 G4 G5 G6 G7 G8
H1 H2 H3 H4 H5 H6 H7 H8
```

Auch hier soll jede der 8 Zeilen mit einem Blank ' ' und einem Zeilenwechsel-Zeichen '\n' enden. Daraus folgt, dass das Ergebnis der Funktion `schach01` die Länge (8x25 gleich) 200 haben muss.

Die Funktionen `schach02`, `schach03` und `schach04` sollen ganz ähnliche Schachbrett-Muster liefern, aber jeweils um 90 Grad nach rechts gedrehte, etwa so:

```
H1 G1 F1 E1 D1 C1 B1 A1   H8 H7 H6 H5 H4 H3 H2 H1   A8 B8 C8 D8 E8 F8 G8 H8
H2 G2 F2 E2 D2 C2 B2 A2   G8 G7 G6 G5 G4 G3 G2 G1   A7 B7 C7 D7 E7 F7 G7 H7
H3 G3 F3 E3 D3 C3 B3 A3   F8 F7 F6 F5 F4 F3 F2 F1   A6 B6 C6 D6 E6 F6 G6 H6
H4 G4 F4 E4 D4 C4 B4 A4   E8 E7 E6 E5 E4 E3 E2 E1   A5 B5 C5 D5 E5 F5 G5 H5
H5 G5 F5 E5 D5 C5 B5 A5   D8 D7 D6 D5 D4 D3 D2 D1   A4 B4 C4 D4 E4 F4 G4 H4
H6 G6 F6 E6 D6 C6 B6 A6   C8 C7 C6 C5 C4 C3 C2 C1   A3 B3 C3 D3 E3 F3 G3 H3
H7 G7 F7 E7 D7 C7 B7 A7   B8 B7 B6 B5 B4 B3 B2 B1   A2 B2 C2 D2 E2 F2 G2 H2
H8 G8 F8 E8 D8 C8 B8 A8   A8 A7 A6 A5 A4 A3 A2 A1   A1 B1 C1 D1 E1 F1 G1 H1
```

6.3. Die Funktion `schach05`

Die Funktionen `schach01` bis `schach04` haben viele Gemeinsamkeiten und ein paar Unterschiede. Deshalb ist es nahe liegend, das Gemeinsame durch *eine einzige Funktion* auszudrücken und die Unterschiede durch *Parameter* darzustellen. Sie sollen jetzt eine solche Funktion

```
static public String schach05(String s1, String s2) { ... }
```

programmieren.

Wenn man die Funktion `schach05` mit geeigneten Parametern aufruft, soll sie die gleichen Ergebnisse liefern, wie die Funktionen `schach01` bis `schach04`, etwa so:

```
schach05("ABCDEFGH", "12345678") liefert das Gleiche wie schach01()
schach05("12345678", "HGFEDCBA") liefert das Gleiche wie schach02()
schach05("HGFEDCBA", "87654321") liefert das Gleiche wie schach03()
schach05("87654321", "ABCDEFGH") liefert das Gleiche wie schach04()
```

Die Funktion `schach05` soll aber auch kleinere oder größere Schachbrett-Muster liefern können, wenn man sie mit entsprechenden (kleineren oder größeren) Parametern aufruft. Hier ein paar Beispiele (platzsparend angeordnet):

```
schach05("ABCDEFGHIJKLMNO", "123"):
```

```
A1 A2 A3
B1 B2 B3
C1 C2 C3
D1 D2 D3
E1 E2 E3
F1 F2 F3
G1 G2 G3
H1 H2 H3
I1 I2 I3
J1 J2 J3
K1 K2 K3
L1 L2 L3
M1 M2 M3
N1 N2 N3
O1 O2 O3
```

```
schach05("123", "ABCDEFGHIJKLMNO"):
```

```
A1 B1 C1 D1 E1 F1 G1 H1 I1 J1 K1 L1 M1 N1 O1
A2 B2 C2 D2 E2 F2 G2 H2 I2 J2 K2 L2 M2 N2 O2
A3 B3 C3 D3 E3 F3 G3 H3 I3 J3 K3 L3 M3 N3 O3
```

```
schach05("AB", "123"):
```

```
A1 A2 A3
B1 B2 B3
```

```
schach05("ABC", "12"):
```

```
A1 A2
B1 B2
C1 C2
```

```
schach05("AB", "12"):
```

```
A1 A2
B1 B2
```

```
schach05("A", "1"):
```

```
A1
```

Auch hier endet jede Ergebniszeile mit einem Blank und einem Zeilenwechsel-Zeichen "\n", und so liefert z.B.

der Aufruf `schach05("AB", "123")` einen String der Länge (2x10 gleich) 20,

der Aufruf `schach05("ABC", "12")` einen String der Länge (3x7 gleich) 21,

der Aufruf `schach05("AB", "12")` einen String der Länge (2x7 gleich) 14,

der Aufruf `schach05("A", "1")` einen String der Länge (1x3 gleich) 3, etc.

Tipp: Die Funktion `schach05` hat zwei `String`-Parameter (`s1` und `s2`). Sie dürfen sich darauf verlassen, dass *einer* dieser beiden Strings nur *Ziffern* enthält und der *andere* nur (große) *Buchstaben*. Sie wissen aber nicht, ob `s1` Ziffern und `s2` Buchstaben enthält oder ob es umgekehrt ist. Im Ergebnis Ihrer Funktion sollen die Buchstaben aber immer *vor* den Ziffern stehen (z.B. A1, nicht 1A). Dieses Problem kann man besonders elegant mit zwei Methoden aus dem Modul `Math` lösen:

```

1 static int max(int n1, int n2);
2 static int min(int n1, int n2);

```

Diese Methoden liefern als Ergebnis das Maximum (bzw. das Minimum) ihrer beiden Parameter (z.B. ist `Math.max(7, 5)` gleich 7 und `Math.min(7, 5)` ist gleich 5). Angenommen, Sie haben zwei `char`-Variablen `c1` und `c2`, von denen eine einen Buchstaben und die andere eine Ziffer enthält (Sie wissen aber nicht, welche was enthält). Betrachten Sie die folgenden Variablen-Vereinbarungen:

```

1 char d1 = (char) Math.max(c1, c2);
2 char d2 = (char) Math.min(c1, c2);

```

Was für ein Zeichen steht jetzt in `d1`? Und in `d2`?

6.4. Die Funktion `gitter`

Auch diese Funktion soll `String`-Ergebnisse *liefern* (und *nichts ausgeben*). Wenn man ein solches `String`-Ergebnis zum Bildschirm ausgibt, soll es dort als ein Gittermuster erscheinen, z.B. so:

```

gitter(2, 3):   gitter(5, 2):   gitter(1, 1):   gitter(0, 2):
+--+
| | |
+--+
| | |
+--+
| | |
+--+

gitter(5, 0):   gitter(0, 0):   gitter(-3, -5):
+-----+
+
+

```

In einem Ergebnis der Funktion `gitter` soll am Ende jeder Zeile nur ein (transparentes) Zeilenwechsel-Zeichen `'\n'` stehen (kein zusätzliches Blank). Somit gilt:

Der Aufruf `gitter(5, 0)` liefert einen String der Länge 12 ("`+-----+\n`")
 Der Aufruf `gitter(0, 0)` liefert einen String der Länge 2 ("`+\n`").
 Der Aufruf `gitter(0, 2)` liefert einen String der Länge 10 ("`+\n| \n+ \n| \n+ \n`")
 Der Aufruf `gitter(1, 1)` liefert einen String der Länge 12
 Der Aufruf `gitter(5, 2)` liefert einen String der Länge (5x12 gleich) 60

Die Funktion `gitter` soll *negative Parameter* automatisch durch 0 ersetzen. Deshalb liefert ein Aufruf wie `gitter(-3, -5)` das gleiche Ergebnis wie `gitter(0, -3)` oder wie `gitter(0, 0)`.

Tipp 1 (für die Methode `gitter`): Mit dem folgenden Befehl kann man einen negativen Inhalt der Variablen `n` besonders elegant durch 0 ersetzen (und andere Werte unverändert lassen):

```

3 n = Math.max(n, 0);

```

Tipp 2 (für die Methode `gitter`): Statt ein Gitter mit Hilfe einer *geschachtelten Schleife* aus einzelnen Zeichen zusammenzubauen, kann man sich auch fragen: Aus was für *Zeilen* besteht ein Gitter? Dann kann man erst mal *eine* solche Zeile zusammenbauen (in einem `StringBuilder`-Objekt, mit einer *einfachen* Schleife). Um aus diesen *Bausteinen* ein Gitter zu erzeugen, braucht man nur noch eine weitere *einfache* Schleife, keine *geschachtelte*.

Tipp 3 (für alle Methoden): Rüsten Sie die Klasse `Schleifen` mit einer `main`-Methode aus, rufen Sie darin die Funktionen `zeile`, `spalte`, `schach01`, `schach02`, ... auf und geben Sie die Ergebnisse zum Bildschirm aus, z.B. so:

```

4 pln("->" + zeile() + "<-");
5 pln("->" + spalte() + "<-");
6 ...

```

Auf diese Weise können Sie sich einen ersten Eindruck davon verschaffen, ob Ihre Funktionen richtig funktionieren.

Anforderung (für alle Methoden): Jedes mal, wenn Sie eine der Methoden geschrieben haben, sollen Sie Ihre Klasse `Schleifen` mit dem (vorgegebenen) JUnit-Testprogramm `SchleifenJut` testen.

Aufgabe 7: StringBuilder (Florian)

Schreiben Sie eine Klasse namens `Florian`, die die folgenden beiden Methoden enthält:

```
1  static public String blanksRaus(String s) {
2      // Liefert einen String, der alle von BLANK verschiedenen Zeichen
3      // aus s (in der gleichen Reihenfolge wie in s) enthaelt.
4      // Beispiele:
5      // blanksRaus(" A B C ") ist gleich "ABC"
6      // blanksRaus("ABC")    ist gleich "ABC"
7      // blanksRaus("")       ist gleich ""
8      ...
9  } // blanksRaus

1 static public String blanksRein(String s) {
2     // Liefert einen String, der alle Zeichen aus s (in der gleichen
3     // Reihenfolge wie in s) enthaelt, aber zusaetzlich zwischen je
4     // zwei Zeichen aus s ein BLANK.
5     // Beispiele:
6     // blanksRein("ABC") ist gleich "A B C"
7     // blanksRein("A")  ist gleich "A "
8     // blanksRein("")   ist gleich ""
9     ...
10  } // blanksRein
```

Anforderung 1: Beim Lösen dieser Aufgabe sollen Sie üben, mit `StringBuilder`-Objekten umzugehen. Deshalb sollen Sie in der Funktion `blanksRaus` nicht die Funktionen

`s.replace` oder

`s.replaceAll`

aufrufen (weil damit die Lösung zu einfach wäre und Sie nicht genug lernen würden :-).

Anforderung 2: Vermeiden Sie beim Schreiben der Funktion `blanksRein` eine

Schleife mit einer `if`-Anweisung im Rumpf

(d.h. vermeiden Sie es, sich "mit einem `if`-Regenschirm unter eine Schleifen-Dusche zu stellen").

Hinweis: Beachten Sie, dass Objekte der Klasse `String` in Java *unveränderbar* sind. Verwenden Sie Objekte der Klasse `StringBuilder`, wenn Sie an einer Zeichenkette (häufig) etwas verändern wollen. Informieren Sie sich über die Klassen `String` und `StringBuilder`, indem Sie in der Online-Dokumentation nachschauen und/oder indem Sie im Buch "Java ist eine Sprache" die Abschnitte 10.1 und 10.3 über die Klassen `String` und `StringBuilder` durchlesen. Oder schauen Sie sich das Beispielprogramm `String02` gründlich an.

Testen Sie Ihre Klasse `Florian` mit dem vorgegebenen Programm `FlorianJut`. Wenn eine Ausführung des Testprogramms länger als eine Minute dauert, haben Sie zu ineffizient programmiert und sollten Ihre Lösung noch mal verbessern. Finden Sie in einem solchen Fall genau heraus, warum Ihre Lösung ineffizient ist (z.B. mit Hilfe der BetreuerIn Ihrer Übungsgruppe).

Aufgabe 8: Reihungen und Sammlungen (Veronika01)

Ergänzen Sie das folgende "Skelett" des Moduls (der Klasse) Veronika01 zu einer vollständigen Klasse, indem Sie die mit // MUSS ERSETZT WERDEN gekennzeichneten Zeilen durch sinnvolle Zeilen ersetzen. Sie dürfen im Modul Veronika01 auch zusätzliche "Hilfsmethoden" vereinbaren, wenn Ihnen das sinnvoll erscheint.

```
1 public class Veronika01 {
2     // -----
3     static public int[] liesIntReihung() {
4         // Liest solange Ganzzahlen von der Standardeingabe ein, bis der
5         // Benutzer eine 0 eingibt. Liefert die eingegebenen Zahlen (ohne
6         // die abschliessende 0) als eine Reihung von int (int[]).
7
8         return null; // MUSS ERSETZT WERDEN!
9
10    } // liesIntReihung
11    // -----
12    static public void gibAusIntReihung(int[] ir) {
13        // Gibt ir in lesbarer Form zur Standardausgabe aus (auch dann, wenn
14        // ir eine leere Reihung ist!).
15
16        // MUSS ERSETZT WERDEN!
17
18    } // gibAusIntReihung
19    // -----
20    static public boolean liegenFalschRum01(int n1, int n2) {
21        // Liefert true genau dann wenn n1 groesser ist als n2.
22
23        return false; // MUSS ERSETZT WERDEN!
24
25    } // liegenFalschRum01
26    // -----
27    static public void sortiere01(int[] ir) {
28        // Sortiert die Komponenten von ir in aufsteigender Folge:
29        // (Algorithmus: Wiederholtes Vertauschen benachbarter Komponenten,
30        // bubble sort)
31
32        // MUSS ERSETZT WERDEN!
33
34    } // sortiere01
35    // -----
36    // Eine Methode mit einem kurzen Namen:
37    static public void pln(Object ob) {System.out.println(ob);}
38    // -----
39 } // class Veronika01
```

Diese Zeilen 1 bis 39 finden Sie auch in der vorgegebenen Datei Veronika01.java (Sie brauchen sie also nicht abzutippen).

In der Prozedur sortiere01 sollen Sie ("häufig wiederholt") zwei nebeneinander liegende Komponenten der Reihung ir vergleichen und wenn sie falsch rum liegen, vertauschen, etwa so:

```
40     if (liegenFalschRum01(ir[i], ir[i+1])) {
41         ... // Vertausche ir[i] und ir[i+1]
42     }
```

Anmerkung: Methoden zu testen, die Daten *einlesen* (z.B. von der Tastatur) oder *ausgeben* (z.B. zum Bildschirm) wirft ganz neue und spezielle Probleme auf.

Das Testprogramm Veronika01Jut testet nur die Methoden liegenFalschRum01 und sortiere01 (und nur einen einzigen speziellen Aufruf der Methode gibAusIntReihung). Wenden Sie das Testprogramm an (bis es einen grünen Balken zeigt). Vereinbaren Sie dann zusätzlich in der Klasse Veronika01 eine main-Methode, die die Methoden liesIntReihung und gibAusReihung gründlich testet (indem sie die zu testenden Methoden mehrmals mit verschiedenen Parametern aufruft).

Aufgabe 9: Reihungen und Sammlungen (Veronika02)

Schreiben Sie einen Modul namens Veronika02, der alle Methoden aus dem Modul Veronika01 (siehe vorige Aufgabe) und zusätzlich die folgenden 5 Methoden enthält:

```
1 // -----
2 static public int summe(int[] ir) {
3     // Liefert die Summe aller Komponenten von ir.
4
5     return -17; // MUSS ERSETZT WERDEN!
6
7 } // summe
8 // -----
9 static public int max(int[] ir) {
10    // Liefert die groesste Ganzzahl aus ir (und Integer.MIN_VALUE falls
11    // ir eine leere Reihung ist.
12
13    return -17; // MUSS ERSETZT WERDEN!
14
15 } // max
16 // -----
17 static public boolean enthaeltDoppelte(int[] ir) {
18    // Liefert true, wenn mindestens eine Ganzzahl mehr als einmal in ir
19    // vorkommt, und sonst false.
20
21    return true; // MUSS ERSETZT WERDEN!
22
23 } // enthaeltDoppelte
24 // -----
25 static public boolean istPrim(int n) {
26    // Liefert true genau dann wenn der Betrag von n eine Primzahl ist.
27
28    return true; // MUSS ERSETZT WERDEN!
29
30 } // istPrim
31 // -----
32 static public int maxPrim(int[] ir) {
33    // Liefert 0 als Ergebnis, wenn r keine Primzahl enthaelt. Liefert
34    // sonst die groesste Primzahl, die in r enthalten ist. Negative
35    // Zahlen werden wie die ensprechenden positiven Zahlen behandelt,
36    // d.h. -7 gilt als Primzahl, ist aber kleiner als die Primzahl +3.
37
38    return -17; // MUSS ERSETZT WERDEN!
39
40 } // maxPrim
41 // -----
42 static public boolean liegenFalschRum02(int n1, int n2) {
43    // Liefert true genau dann wenn
44    // n1 und n2 gerade sind und n1 groesser als n2 ist oder
45    // n1 und n2 ungerade sind und n2 groesser als n1 ist oder
46    // n1 ungerade und n2 gerade ist.
47
48    boolean n1IstGerade = n1%2 == 0;
49    boolean n2IstGerade = n2%2 == 0;
50
51    return false; // MUSS ERSETZT WERDEN!
52
53 } // liegenFalschRum02
54 // -----
55 static public void sortiere02(int[] ir) {
56    // Sortiert die Reihung ir auf folgende Weise:
57    // Zuerst kommen alle geraden Zahlen in aufsteigender Folge, und
58    // danach kommen alle ungeraden Zahlen in absteigender Folge
59    // (Algorithmus: Wiederholtes Vertauschen benachbarter Komponenten,
60    // bubble sort)
61
62    // MUSS ERGAENZT WERDEN!
63
64 } // sortiere02
```

Diese Zeilen 1 bis 64 finden Sie auch in der vorgegebenen Datei `Veronika02.java` (Sie brauchen sie also nicht abzutippen).

Wichtige Anforderung: Die Prozedur `sortiere02` soll eine **Kopie** der Prozedur `sortiere01` (aus der vorigen Aufgabe) sein, in der Sie nur anstelle der Funktion `liegenFalschRum01` die neue Funktion namens `liegenFalschRum02` aufrufen.

Testen Sie Ihre Lösung `Veronika02` mit dem Testprogramm `Veronika02Jut`.

Aufgabe 10: Klasse Punkt3D

Betrachten Sie die Vereinbarung der Klasse `E01Punkt` (im Buch S. 287). Vereinbaren Sie dann eine Klasse `Punkt3D` als Erweiterung von `E01Punkt` und schreiben Sie ein JUnit-Testprogramm namens `Punkt3D_Jut`, mit dem man die Klasse `Punkt3D` testen kann. Wie man JUnit-Programme schreibt wird z.B. in dem Papier [JUnitEinfuehrung.pdf](#) beschrieben (lernen Sie dieses Papier auswendig, oder lesen Sie es wenigstens durch :-). Beachten Sie beim Lösen dieser Aufgabe die folgenden Einzelheiten:

1. Ändern Sie die Datei `E01Punkt.java` so, dass die Klasse `E01Punkt` öffentlich (engl. `public`) ist und nicht mehr zum *namenlosen Paket* gehört, sondern zum Paket namens `erben`. Compilieren Sie die Datei danach noch einmal. Ohne diese (oder ähnliche) Änderungen könnte die Klasse `Punkt3D` die Klasse `E01Punkt` nicht beerben (oder: erweitern). Weitergehende Änderungen an der Klasse `E01Punkt` (z.B. die *privaten* Attribute `x` und `y` *öffentlich* machen) sind **nicht** erlaubt.

2. Die Klasse `Punkt3D` soll ebenfalls öffentlich sein, aber zum Paket `erben3d` gehören.

3. Die Klasse `Punkt3DJut` soll nicht zum Paket `erben3d` gehören, sondern zum namenlosen Paket.

Hinweis: Da die Klassen `Punkt3D` und `Punkt3D_Jut` zu verschiedenen Paketen gehören, dürfen ihre Quelldateien `Punkt3D.java` und `Punkt3D_Jut.java` nicht im selben Verzeichnis stehen.

4. Jedes Objekt der Klasse `E01Punkt` enthält zwei Attribute (engl. `fields`) `x` und `y`. Ein Objekt der Klasse `Punkt3D` soll ein zusätzliches Attribut namens `z` enthalten.

5. Die Klasse `E01Punkt` hat einen Konstruktor mit *zwei* `double`-Parametern. Ganz entsprechend soll die Klasse `Punkt3D` einen Konstruktor mit *drei* `double`-Parametern enthalten.

6. Jedes `E01Punkt`-Objekt enthält eine Methode namens `text`, die die Koordinaten des `E01Punktes` als `String` liefert (z.B. so: `"(2.5, 3.0)"`). Jedes `Punkt3D`-Objekt soll eine entsprechende Methode namens `text` enthalten (die `Strings` wie z.B. `"(2.5, 3.0, 1.8)"` liefert). Benutzen Sie zur Berechnung des richtigen Textes die Methode `super.text()` (d.h. die Methode `text()` aus der Oberklasse `E01Punkt`) und zusätzliche Befehle.

7. Jedes `E01Punkt`-Objekt enthält eine Methode namens `urAbstand`, die den Abstand des Punktes vom Ursprung (d.h. vom Punkt `(0.0, 0.0)`) als Ergebnis liefert. Entsprechend soll jedes `Punkt3D`-Objekt eine Methode namens `urAbstand` enthalten, die den Abstand des Punktes vom Ursprung (d.h. vom Punkt `(0.0, 0.0, 0.0)`) als Ergebnis liefert. Benutzen Sie zur Berechnung dieses Abstands die Methode `super.urAbstand()` (d.h. die Methode `urAbstand()` aus der Oberklasse `E01Punkt`) und zusätzliche Befehle.

8. Jedes `E01Punkt`-Objekt enthält eine Methode namens `urSpiegeln`, die den Punkt am Ursprung spiegelt. Entsprechend soll jedes `Punkt3D`-Objekt eine Methode namens `urSpiegeln` enthalten, die das Entsprechende leistet. Benutzen Sie für das Spiegeln eines `Punkt3D`-Objekts die Methode `super.urSpiegeln()` (d.h. die Methode `urSpiegeln()` aus der Oberklasse `E01Punkt`) und zusätzliche Befehle.

9. In der Klasse `Punkt3D` soll *keine* neue `toString`-Methode vereinbart werden (die geerbte `toString`-Methode funktioniert auch für `Punkt3D`-Objekte. Können Sie erklären, warum?).

10. In der Klasse `Punkt3D` sollen zwei parameterlose Objektfunktionen namens `getFlaeche` und `getVolumen` (Rückgabebetyp: `double`) vereinbart werden, die immer `0.0` als Ergebnis liefern.

11. Ihre Testklasse `Punkt3D_Jut` soll die Methoden `urAbstand`, `urSpiegeln`, `text`, `getFlaeche` und `getVolumen` von mindestens 3 verschiedenen `Punkt3D`-Objekten testen.

12. Es gibt in JUnit eine `assertEquals`-Methode mit 3 `double`-Parametern. Der dritte Parameter stellt eine *absolute Differenz* dar, die bei gleichen `double`-Zahlen noch geduldet werden kann. In der Praxis will man aber meistens eine *relative Differenz* dulden (z.B. eine Differenz von 3 Prozent oder von 0.05 Promille etc.). Ihre Tests sollen Abweichungen von *1.5 Prozent* dulden.

Die Datei `StringBuilderJut.java` enthält ein Beispiel für eine mit JUnit erstellte Testklasse, an der Sie sich (beim Schreiben des Testprogramms `Punkt3D_Jut`) orientieren können.

Aufgabe 11: Klasse Quader etc.

Betrachten Sie die Klasse `Punkt3D` (Ihre Lösung zur vorigen Aufgabe) und programmieren Sie dann folgende Klassen:

- Eine Klasse namens `Quader` als Erweiterung der Klasse `Punkt3D`.
- Eine Klasse namens `Wuerfel` als Erweiterung der Klasse `Quader`.
- Eine Klasse namens `Kugel` als Erweiterung der Klasse `Punkt3D`.

Schreiben Sie außerdem ein Testprogramm namens `QuWuKuJut`, welches die Klassen `Quader`, `Wuerfel` und `Kugel` testet (ganz ähnlich, wie Ihr Testprogramm `Punkt3D_Jut` aus der vorigen Aufgabe die Klasse `Punkt3D` testet). Der Clou dieses Testprogramms: Sie können (und sollen) alle Testobjekte (der Typen `Quader`, `Wuerfel` und `Kugel`) in *einer einzigen Reihung* speichern. Und auch sonst sollten Sie möglichst oft *Reihungen* verwenden statt einzelne Variablen.

Die Klassen `Quader`, `Wuerfel` und `Kugel` sollen sich "ganz entsprechend verhalten", wie die Klassen `E01Rechteck`, `E01Quadrat` und `E01Kreis` (deren Vereinbarungen man im Abschnitt 12.3 des Buches "Java ist eine Sprache" bzw. bei den Beispielprogrammen findet), ihre Objekte sollen aber *dreidimensionale* Körper darstellen. Bevor Sie anfangen zu programmieren sollten Sie folgende Fragen klären (evtl. mit Hilfe Ihrer BetreuerIn):

1. Was entspricht dem *Umfang* und der *Fläche* einer zweidimensionalen Figur (z.B. eines Rechtecks) bei einer dreidimensionalen Figur (z.B. bei einem Quader)?
2. Wie sehen die Strings aus, die von den `toString`-Methoden der E01-Klassen (`E01Punkt`, `E01Rechteck`, `E01Quadrat`, ...) als Ergebnis geliefert werden? Wie sollten die Ergebnisse der `toString`-Methoden Ihrer neuen Klassen also aussehen?
3. Wie viele Attribute werden in der Klasse `E01Quadrat` vereinbart? Wie viele Attribute sollten Sie in Ihrer Klasse `Wuerfel` vereinbaren?
4. Wie viele Methoden werden in der Klasse `E01Quadrat` überschrieben? Wie viele Methoden sollten Sie in Ihrer Klasse `Wuerfel` überschreiben?
5. Von welchem Typ ist der Ausdruck $4/3$ und welchen Wert hat er (genau)? Von welchem Typ ist der Ausdruck $4.0/3.0$ und welchen Wert hat er (ungefähr)?

Beachten Sie beim Schreiben des Testprogramms `QuWuKuJut` folgende Punkte:

6. Lassen von jeder zu testenden Klasse mindestens 3 Testobjekte erzeugen (insgesamt also mind. 9 Testobjekte). Speichern Sie all diese Testobjekte in *einer* Reihung. Initialisieren Sie diese Reihung innerhalb einer parameterlosen Prozedur namens `setUp`. Finden Sie heraus, wann das `JUnit`-Rahmenprogramm diese `setUp`-Methode aufruft (siehe die Datei `StringBuilderJut.java`).
7. Testen Sie *alle* Methoden, die in den Testobjekten enthalten sind (auch geerbte Methoden, die schon mal getestet wurden).
8. Vereinbaren Sie für jede zu testende Methode (z.B. für die Methode `toString` und für die Methode `urAbstand` etc.) eine Reihung, die alle benötigten Soll-Daten enthält.

Beachten Sie schließlich folgende Einzelheiten:

9. Die Klassen `Quader`, `Wuerfel` und `Kugel` sollen (genau wie die Klasse `Punkt3D`) zum Paket `erben3d` gehören.
10. Das Testprogramm `QuWuKuJut` soll *nicht* zum Paket `erben3d` gehören, sondern zum namenlosen Paket.
11. Im Testprogramm `QuWuKuJut` dürfen Sie die Klassen `Wuerfel`, `Quader` und `Kugel` importieren (d.h. für ihre vollen Namen Abkürzungen vereinbaren).

Aufgabe 12: Klammern prüfen

Ein Text kann verschiedene Arten von Klammern enthalten, z.B. runde (. . .), eckige [. . .] und geschweifte { . . . }. Für solche Klammern gibt es "universelle Rechtschreibregeln", die weitgehend unabhängig davon sind, ob es sich bei dem Text um eine Bachelor-Arbeit an der Beuth Hochschule, den Quelltext eines Java-Programms oder einen Kriminalroman handelt. Es folgen hier ein paar Beispiele für Texte, die *falsch gepaarte* Klammern enthalten. Darin sollen die Auslassungen ". . ." Textstücke darstellen, die *keine* Klammern enthalten:

	1	2	3	4	5	6	7	8	9	10	11
Beispiel 1:	(. . .]								
Beispiel 2:	(. . .)	. . .]						
Beispiel 3:	(. . .	[. . .]						
Beispiel 4:)	. . .	(
Beispiel 5:]	. . .]	. . .	}	. . .	{	. . .	[. . .	[
Beispiel 6:	(. . .	[. . .)	. . .]				
Beispiel 7:	(. . .									
Beispiel 8:	(. . .	[. . .]	. . .					
Beispiel 9:	{	. . .	{	. . .	[. . .					

Bei den Beispielen 4 bis 6 ist nur die *Reihenfolge* der Klammern falsch, nicht ihre *Anzahl*. Es genügt also *nicht*, beim Prüfen eines Textes die Klammern zu *zählen*.

Zur Einarbeitung 1: Lesen Sie jedes Beispiel zeichenweise (von links nach rechts) durch und stellen Sie fest: Ab welcher Stelle sind Sie sicher, dass der Text einen Klammerfehler enthält? Sie können "die Stellen" durch die Zahlen zwischen 1 und 11 bezeichnen, die über den Beispiel-Texten stehen.

Zur Einarbeitung 2: Welche der Beispiel-Texte könnte man dadurch korrigieren, dass man bestimmte Zeichen hinten anhängt? Welche Zeichen muss man bei diesen Beispielen anhängen? Welche Beispiele kann man *nicht durch Anhängen* von Zeichen korrigieren?

Zur Einarbeitung 3: Was hat die Korrektheit der Klammern mit der *Zeilenstruktur* eines Textes zu tun (d.h. damit, ob der gesamte Text auf *einer* Zeile steht oder auf *mehrere* Zeilen verteilt wurde?).

Zur Einarbeitung 4: Was hat die Klammerstruktur eines Textes mit einem Stapel (oder Keller, engl. stack) zu tun? *Wann* sollte man (beim zeichenweise Lesen und Prüfen eines Textes) etwas auf den Stapel legen? *Wann* sollte man etwas vom Stapel entfernen? *Was* sollte man auf den Stapel legen: Öffnende *und* schließende Klammern? Nur *öffnende* Klammern? Nur *schließende* Klammern?

Aufgabenstellung: Sie sollen ein Programm namens KlammernPruefenA schreiben, mit dem man einen beliebigen Text daraufhin prüfen kann, ob alle runden, eckigen und geschweiften Klammern richtig gepaart sind.

Die Prüfung soll abgebrochen werden, sobald *ein* (erster) Fehler erkannt wurde. Das Ergebnis einer Prüfung soll eine Meldung sein, die eine der folgenden 3 Formen hat:

Form 1: Alle Klammern sind ok!

Form 2: Klammer falsch:], Zeile 5, Spalte 13

Form 3: Am Ende fehlen Klammern:]})}]

Um Meldungen der Form 2 erzeugen zu können, muss man beim Einlesen des Textes die Zeilen- und Spalten-Nummern "mitzählen". Beginnen Sie die Nummerierung der Zeilen und Spalten jeweils mit 1 (nicht mit 0).

Beispiel-01: Falls eine Datei mit dem Zeichen ') ' beginnt, soll folgende Meldung erscheinen:

Klammer falsch:), Zeile 1, Spalte 1

Den Pfadnamen der zu prüfenden Datei soll man Ihrem Programm beim Aufruf als Kommandozeilen-Parameter übergeben können.

Beispiel-02: Ein Aufruf des Programms KlammernPrüfenA:

```
> java KlammernPruefenA D:\ordner27\Klammern1.ein
```

Daraufhin soll die Datei D:\ordner27\Klammern1.ein geprüft werden.

Tipp: Beginnen Sie Ihr Programm `KlammernPruefenA` mit folgenden import-Befehlen:

```
1 import java.io.FileReader;
2 import java.io.BufferedReader;
3 import java.util.Stack;
4 import java.util.EmptyStackException;
5
6 public class KlammernPruefenA {
7     ...
```

Mit einem Objekt der Klasse `FileReader` kann man Daten aus einer Datei lesen, allerdings (etwas vereinfacht gesagt) nur *zeichenweise*. Ein `BufferedReader`-Objekt enthält eine Methode namens `readLine`, mit der man Daten *zeilenweise* lesen kann.

Das Programm `KlammernPruefenA` testen

Wenden Sie Ihr Programm `KlammernPruefenA` (um es zu testen) mindestens auf die vorgegebenen Dateien `Klammern1.ein`, `Klammern2.ein` und `Klammern3.ein`. Diese Dateien enthalten (als letzte Zeile) die Meldung, die Ihr `KlammernPruefenA`-Programm ausgeben sollte. Sie dürfen auch das vorgegebene Skript `tstA.cmd` benutzen.

Freiwillige Zusatzaufgabe

Wenn Sie mit dem Programm `KlammernPruefenA` seine eigene Quelldatei `KlammernPruefenA.java` prüfen, werden sehr wahrscheinlich *nicht* die Meldung `Alle Klammern sind ok!`

bekommen. Warum nicht?

Können Sie die Quelldatei `KlammernPruefenA.java` so ändern, dass die `ok`-Meldung erscheint?

Interessanter und anspruchsvoller ist es, ein Programm `KlammernPruefenB` zu schreiben, welches "Klammerfehler" in `String`-Literalen und in Kommentaren nicht als Fehler zählt, sondern einfach "überliest".

Aufgabe 13: Miez, eine noch ziemlich kleine Katze

Ein Miez-Objekt ist im wesentlichen ein *Fenster* (technisch: ein `JFrame`-Objekt) mit einem *Knopf* (einem `JCheckBox`-Objekt) und einem *Etikett* (einem `JLabel`-Objekt) darin. In einem solchen Miez-Fenster werden *Mausereignisse* der Art `MouseClicked` "beobachtet", d.h. abgefangen und zur Standardausgabe ausgegeben. Außerdem sollen die Koordinaten solcher `MouseClicked`-Ereignisse ("Wo im Fenster hat der Benutzer mit der Maus geklickt?") in das Etikett geschrieben werden.

Der Clou: Erst wenn der `JCheckBox`-Knopf AN-geklickt wird, soll beim `JFrame`-Fenster ein `MouseListener`-Objekte angemeldet werden (und dadurch die Beobachtung von `MouseClicked`-Ereignisse beginnen). Wenn der Knopf später AUS-geklickt wird, soll das `MouseListener`-Objekt wieder abgemeldet werden (und die Beobachtung von `MouseClicked`-Ereignisse wieder aufhören).

Die Aufgabe Schritt für Schritt:

Erweitern Sie die Klasse `javax.swing.JFrame` zu einer Klasse namens **Miez**, die folgende Objektelement enthält:

1. Ein `JCheckBox`-Objekt namens **knopf01** (Beschriftung "Clicked")
2. Ein `JLabel`-Objekt namens **etikett01** (anfängs "-----")
3. Ein `Box`-Objekt namens **kasten01** (eine horizontale Box)
4. Ein `ActionListener`-Objekt namens **behandlerClickedK** ("K" wie "Knopf") mit der üblichen Methode `actionPerformed` darin.
5. Ein `MouseListener`-Objekt namens **behandlerClickedM** ("M" wie "Maus") mit eine Methode `MouseClicked` darin.

Die Klasse **Miez** soll einen **Konstruktor** mit einem `String`-Parameter **titel** haben, der folgendes leistet:

1. Er ruft `super(titel)` auf.
2. Er fügt den **knopf01** und das **etikett01** in den **kasten01** ein (mit der Methode `kasten01.add`)
3. Er fügt den **kasten01** in das aktuelle **Miez**-Objekt ein (mit der Methode `this.add`)
4. Er meldet den **behandlerClickedK** beim **knopf01** an (mit der Methode `knopf01.addActionListener`)
5. Er legt fuer das aktuelle **Miez**-Objekt eine vernuenftige Groesse fest (z.B. 500 mal 300 Pixel, mit der Methode `this.setBounds`) und macht es sichtbar (mit der Methode `this.setVisible`).

Die Methode `public void mouseClicked(MouseEvent me)`; im Behandlerobjekt **behandlerClickedM** soll das Ereignis **me** zur Standardausgabe ausgeben. Ausserdem soll sie die x- und y-Koordinate des Ereignisses **me** ermitteln und (in lesbarer Form) in das **etikett01** schreiben (mit der Methode `etikett01.setText`)

Die Methode `public void actionPerformed(ActionEvent ae)`; im Behandlerobjekt **behandlerClickedK** soll das Ereignis **ae** zu Standardausgabe ausgeben. Ausserdem soll sie folgendes leisten:

Wenn der **knopf01** "an ist" (`knopf01.isSelected()`) soll das Behandlerobjekt **behandlerClickedM** bei der `ContentPane` des aktuellen **Miez**-Objekts **angemeldet** werden (mit der Methode `getContentPane().addMouseListener`).

Sonst soll das angemeldete Behandlerobjekt **abgemeldet** werden (mit der Methode `getContentPane().removeMouseListener`).

Zusatz 1 (Miez01): Ganz entsprechend wie `MouseClicked`-Ereignisse sollen auch `MouseDragged`-Ereignisse (an- und abschaltbar) beobachtet werden.

Zusatz 2 (Miez02): Ganz entsprechend sollen auch `MouseWheelMoved`-Ereignisse (an- und abschaltbar) beobachtet werden.

Zusatz 3 (Katze): Ganz entsprechend sollen Mausereignisse *aller acht Arten* (`MouseClicked`, `MouseEntered`, `MouseExited`, `MousePressed`, `MouseReleased`, `MouseDragged`, `MouseMoved` und `MouseWheelMoved` (einzeln an- und abschaltbar) beobachtet werden. Alle Größen, die acht Mal vorkommen (für jede Art von Ereignis einmal) sollten in Reihungen organisiert werden.