Programmierkurs Java

Dr.-Ing. Dietrich Boles

Aufgaben zu UE 6 - Funktionen

(Stand 12.11.2019)

Aufgabe 1:

Die Ulam-Zahlenfolge beginnt mit einer beliebigen natürlichen Zahl *a*. Ist *a* gerade, so erhält man die nächste Zahl *b* durch Halbierung; ist *a* ungerade, so erhält man *b* dadurch, dass man *a* verdreifacht und 1 hinzuaddiert. Die Folge endet, wenn ein Glied den Wert 1 hat.

Es stellt sich die Frage, ob die Ulam-Folge für alle natürlichen Zahlen als Startwert endlich ist. Man vermutet, dass dies tatsächlich so ist, da bis heute kein Startwert bekannt ist, für den die Folge nicht endlich ist. Ein Beweis ist den Mathematikern aber bisher noch nicht gelungen. Die Ulam-Folge gehört deshalb zu den so genannten "offenen Problemen".

Definieren Sie zunächst eine Funktion *ulamWert*, die als Parameter einen int-Wert *n* übergeben bekommt und als Ergebnis gemäß der oben angegebenen Gesetzmäßigkeit den nächsten Wert der Ulam-Folge nach *n* liefert.

Schreiben Sie dann ein Java-Programm, dass zunächst den Benutzer auffordert, einen int-Wert *zahl* einzugeben. Anschließend soll die komplette Ulam-Folge zu *zahl* auf den Bildschirm ausgeben werden. Nutzen Sie dabei die von Ihnen definierte Funktion *ulamWert*.

Beispiel:

```
Eingabe: 3          Ausgabe: 3 10 5 16 8 4 2 1
```

Aufgabe 2:

Implementieren Sie folgende Funktionen, und zwar jeweils zweimal: einmal als iterative und einmal als rekursive Lösung (bis auf `getBetrag`) :

- `int getBetrag(int zahl)`; liefert den Betrag der übergebenen Zahl; Bsp.: `getBetrag(-3) -> 3`
- `int getAnzahlZiffern(int zahl);` liefert die Anzahl an Ziffern der übergebenen Zahl;

  Bsp.: getAnzahlZiffern(2542) -> 4 bzw. getAnzahlZiffern(-342) -> 3
- `int getZiffernWert(int zahl, int stelle)`; liefert den Wert der Ziffer der übergebenen Zahl an der Stelle `stelle`; die Stellen werden dabei von rechts nach links angegeben und beginnen bei 0; Sie können davon ausgehen, dass gilt: 0 <= stelle < getAnzahlZiffern(zahl);

Bsp.: `getZiffernWert(27381, 3) -> 7` bzw. `getZiffernWert(-27381, 0) -> 1`

- `int ersetzeZiffer(int zahl, int stelle, int wert)`; ersetzt die Ziffer der übergebenen Zahl an der Stelle `stelle` durch den übergebenen `wert` und liefert die neue Zahl; die Stellen werden dabei von rechts nach links angegeben und beginnen bei 0; Sie können davon ausgehen, dass gilt: 0 <= stelle < getAnzahlZiffern(zahl) und 0 <= wert <= 9;

  Bsp.: `ersetzeZiffer(24135, 3, 7) -> 27135` bzw. `ersetzeZiffer(-12345, 0, 6) -> -12346`

Schreiben Sie ein kleines Programm, mit dem Sie die Funktionen testen können.

Aufgabe 3:

Implementieren Sie in Java folgende Prozeduren/Funktionen! Achten Sie auf Randfälle und nicht korrekte Parameterübergaben! Überprüfen Sie aber zunächst für jede Funktion, ob sie überhaupt mit den (bisher bekannten) Konzepten in Java implementiert werden kann und wenn nicht, begründen Sie, wieso nicht!

(1) eine Funktion, die testet, ob eine als Parameter übergebene natürliche Zahl eine Fibonacci-Zahl ist oder nicht (Beispiel: f(8) == true),

(2) eine Prozedur, die die Werte zweier als Parameter übergebener double-Variablen vertauscht (Beispiel: double a=2.0; double b=5.9; f(a, b); Ergebnis: a==5.9; b==2.0;),

(3) eine Funktion, die einen übergebenen double-Wert rundet und als int-Wert zurückliefert (Beispiel: f(-2.6) == 3)

(4) eine Funktion, die die nächst kleinere Primzahl einer als Parameter übergebenen natürlichen Zahl (größer als 2) liefert (Beispiel: f(11) == 7),

(5) eine Funktion, die als Parameter einen int-Wert übergeben bekommt und die überprüft, ob die Ziffer 7 in dem int-Wert vorkommt (Beispiel: f(-2578) == true),

(6) eine Funktion, die als ersten Parameter eine Funktion g:char->int und als zweiten Parameter einen char-Wert *zeichen* übergeben bekommt und die als Ergebnis den Wert g(zeichen) liefert (Beispiel: public static int pos(char zeichen) {return zeichen - ‘a‘ ;} und f(pos, ‘b‘) == 2),

(7) eine Funktion, die als Parameter einen int-Wert n übergeben bekommt und die als Ergebnis die Summe der Zahlen zwischen 0 und n zurückliefert; ist der Wert des übergebenen Parameters jedoch kleiner als 0, soll die Funktion den Wert false liefern (Beispiel: f(4) = 10, f(-2) == false)

(8) eine Funktion, die als ersten Parameter einen int-Wert n und daraufhin n float-Parameter übergeben bekommt, deren Summe geliefert werden soll (Beispiel: f(3, 1.1f, 2.2f, 3.3f) == 6.6f).

(9) eine Funktion, die als ersten Parameter einen float-Wert *x* (zwischen 0 und 1000) und als zweiten Parameter einen positiven int-Wert *n* (zwischen 1 und 5) übergeben bekommt. Die Funktion soll den Wert x auf n Nachkommastellen runden (Beispiel: f(2.2576F, 3) == 2.258F)

(10) eine Funktion, die die Summe zweier Uhrzeiten als Ergebnis liefert; Uhrzeiten werden dabei als float-Werte realisiert, wobei die Vorkommastellen die

Stunden und die Nachkommastellen die Minuten darstellen (Beispiel: f(22.13f, 3.48f) == 2.01f)

(11) eine Funktion, die die Summe und die Differenz zweier als Parameter übergebener int-Werte zurückliefert (Beispiel: f(4, 3) = (7, 1))

Schreiben Sie ein Programm, das einem Benutzer eine Auswahl zur Ausführung der implementierbaren Funktionen anbietet, anschließend jeweils passende Werte für die aktuellen Parameter einliest, die ausgewählte Funktion aufruft und ein Ergebnis auf dem Bildschirm ausgibt. Achten Sie darauf, dass die Funktionen nur mit zulässigen Werten aufgerufen werden.

Aufgabe 4:

Schauen Sie sich folgendes Programm an:

```
public class Fragezeichen {
public static void main(String[] args) {
int x = IO.readInt();
int m1 = 2147483647;
int m2 = -2147483648;
int s = 0;
int zahl = 0;
for (int i=0; i<x; i++) {
zahl = IO.readInt("Z: ");
m1 = m(m1, zahl);
m2 = m2 > zahl ? m2 : zahl;
s = s + zahl;
}
if (x > 0) {
IO.println("K: " + m1);
IO.println("G: " + m2);
IO.println("M: " + ((double)s / (double)x));
}
}
public static int m(int zahl1, int zahl2) {
return zahl1 < zahl2 ? zahl1 : zahl2;
}
}
```

Was tut dieses Programm? Wandeln Sie das Programm um in ein "schönes" Programm:

- Nutzen Sie Leerzeichen, Zeilenumbruchzeilen, etc. zur übersichtlicheren Formatierung!
- Wählen Sie aussagekräftige Bezeichner!
- Definieren Sie Funktionen für eine bessere Übersichtlichkeit!
- Integrieren Sie Kommentare, wo notwendig!
- Wählen Sie aussagekräftige Hinweise an den Benutzer!
- Schränken Sie den Gültigkeitsbereich von Variablen so eng wie möglich ein!

Aufgabe 5:

Schreiben Sie ein Programm, das die Ziffern einer eingegebenen positiven Zahl in aufsteigender Reihenfolge sortiert und diese ausgibt. 0en fallen weg. Sie dürfen dabei keine Strings, Arrays oder Rekursion benutzen!

Beispiel:

```
Eingabe: 1423        Ausgabe: 1234
Eingabe: 1442624     Ausgabe: 1224446
Eingabe: 100132      Ausgabe: 1123
```

Aufgabe 6:

Schreiben Sie ein Java-Programm, das einen int-Wert zahl mit 0 < zahl < 10000 einliest, ihre Quersumme berechnet und die durchgeführte Berechnung sowie den Wert der Quersumme wie nachfolgend beispielhaft dargestellt ausgibt:

```
Ganze Zahl zwischen 1 und 9999 eingeben: 2546
Die Quersumme ergibt sich zu: 2 + 5 + 4 + 6 = 17
```

Schreiben und benutzen Sie dabei eine Funktion `int liefereZiffer(int zahl, int stelle)` , die von einer übergebenen Zahl den Ziffernwert an der Stelle "`stelle`" (von hinten) liefert; Beispiel: `liefereZiffer(2548, 3)` soll den Wert "5" liefern.

Aufgabe 7:

Zwei verschiedene natürliche Zahlen a und b heißen befreundet, wenn die Summe der (von a verschiedenen) Teiler von a gleich b ist und die Summe der (von b verschiedenen) Teiler von b gleich a ist.

Ein Beispiel für ein solches befreundetes Zahlenpaar ist (a,b) = (220,284), denn a = 220 hat die Teiler 1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110 und es gilt

1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284 = b.

Weiterhin hat b = 284 die Teiler 1, 2, 4, 71, 142 und es gilt

1 + 2 + 4 + 71 + 142 = 220 = a.

Schreiben Sie ein Java-Programm, das in einer Schleife jeweils zwei Zahlen einliest und entscheidet, ob diese miteinander befreundet sind. Das Programm soll abbrechen, falls zwei nicht-befreundete Zahlen eingegeben wurden. Implementieren und verwenden Sie dabei eine `int`-Funktion `teilersumme`, die von der ihr übergebenen Zahl die Teilersumme (ohne die Zahl selbst) zurückliefert.

Der Programmablauf könnte in etwa wie folgt aussehen:

```
Erste Zahl: 220
Zweite Zahl: 284
Die beiden Zahlen sind miteinander befreundet!
Erste Zahl: 10744
Zweite Zahl: 10856
Die beiden Zahlen sind miteinander befreundet!
Erste Zahl: 23
Zweite Zahl: 22
Die beiden Zahlen sind nicht miteinander befreundet!
```

Aufgabe 8:

Eine Natürliche Zahl heißt potent, wenn sie sich als Summe von Potenzen (>= 1) ihrer Ziffern darstellen lässt. Beispielsweise ist 24 wegen $2^3 + 4^2 = 24$ potent. Schreiben Sie ein Java-Programm, das alle zweistelligen potenten Zahlen ausgibt. Implementieren und verwenden Sie dabei eine Funktion `int potenz(int zahl, int n)`, die die *n*-te Potenz von *zahl* berechnet und zurückliefert sowie eine Funktion `boolean potent(int zahl)`, die genau dann true liefert, wenn *zahl* potent ist.

Aufgabe 9:

Kehrt man bei einer Natürlichen Zahl die Reihenfolge der Ziffern um, so erhält man ihre Spiegelzahl. Eine Zahl heißt *Palindrom*, wenn sie mit ihrer Spiegelzahl übereinstimmt (Bsp.: 15851).

(1) Schreiben Sie ein Java-Programm, das Zahlen einliest und jeweils überprüft, ob es sich bei der Zahl um ein Palindrom handelt. Definieren Sie dazu eine geeignete Funktion! Implementieren Sie eine iterative und eine rekursive Lösung!

(2) Folgendermaßen lassen sich Palindrome erzeugen: Man addiert zu einer gegebenen Natürlichen Zahl ihre Spiegelzahl, zur Summe wieder deren Spiegelzahl usw.; dies wird solange wiederholt, bis ein Palindrom entstanden ist (Beispiel: 178 + 871 + 9401 + 05401 = 15851). Achtung: Bei bestimmten Zahlen bricht der Algorithmus (wahrscheinlich) niemals ab; Bsp. 196.

Schreiben Sie ein Java-Programm, das nach der Eingabe einer Natürlichen Zahl versucht, daraus gemäß dem beschriebenen Algorithmus ein Palindrom zu erzeugen.

Übrigens, interessante Nicht-Zahlen-Palindrome sind (siehe http://de.wikipedia.org/wiki/Liste_deutscher_Palindrome):

- Nie, Amalia, lad 'nen Dalai-Lama ein
- Nie reibt im Regen Neger mit Bier ein
- Ein Neger mit Frust surft im Regen nie
- Emma rede, sei lieb, nett und nun lese die Novelle von Eid, Eseln und Nutten bei Liese, der Amme.

Aufgabe 10:

Das Spiel „27“ ist ein Zahlenspiel für 2 Spieler. Bei dem Spiel geht es darum, dass zwei Spieler abwechselnd entweder die Zahl 1 oder die Zahl 2 auswählen, deren Wert von einer Spielzahl abgezogen wird, deren Anfangswert 27 beträgt. Wer als erster die 0 oder eine kleinere Zahl erreicht, hat gewonnen. Es beginnt Spieler A.

Aufgabe: Schreiben Sie ein Java-Programm, bei dem zwei Menschen gegeneinander das Spiel „27“ spielen können. Im Folgenden wird ein möglicher Spielablauf skizziert (Benutzereingaben in <>):

```
Spielzahl = 27
Spieler A, 1 oder 2 wählen: <2>
Spielzahl = 25
Spieler B, 1 oder 2 wählen: <1>
Spielzahl = 24
Spieler A, 1 oder 2 wählen:
. . .
Spielzahl = 4
Spieler A, 1 oder 2 wählen: <2>
Spielzahl = 2
Spieler B, 1 oder 2 wählen: <2>
Spieler B hat gewonnen!
```

Alternative: Zum Spiel „27“ gibt es eine Gewinnformel sprich Gewinnstrategie. Wenn Spieler B diese kennt und entsprechend spielt, gewinnt er immer. Versuchen Sie diese Gewinnformel herzuleiten. Wenn Sie dies schaffen, können Sie das zu entwickelnde Programm auch derart abändern, dass Spieler A ein Mensch ist und Spieler B der Computer ist, der diese Strategie anwendet und daher immer gewinnt. Ein möglicher Spielablauf sähe dann so aus (Benutzereingaben in <>):

```
Spielzahl = 27
Spieler A, 1 oder 2 wählen: <2>
Spielzahl = 25
Computer wählt 1
Spielzahl = 24
Spieler A, 1 oder 2 wählen:
. . .
Spielzahl = 3
Spieler A, 1 oder 2 wählen: <2>
Spielzahl = 1
Computer wählt 1
Spieler B hat gewonnen!
```

Hinweis: Nutzen Sie Funktionen, um Ihr Programm zu strukturieren. Implementieren Sie bspw. eine Funktion `static int eingeben(char spieler, int aktZahl)`, der als Parameter der aktuelle Spieler (‚A‘ oder ‚B‘) und die aktuelle Zahl übergeben werden und die dann durch Frage an den Benutzer bzw. bei der Alternative Anwenden der Gewinnstrategie die nächste Zahl ermittelt und zurückliefert.

Aufgabe 11:

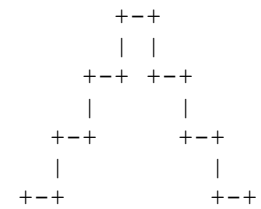
Bald ist Weihnachten und Sie möchten sich selbst mal eine Freude machen, indem Sie einen ASCII-Weihnachtsbaum auf den Bildschirm Ihres Computers malen.

Aufgabe: Schreiben Sie ein Java-Programm, das den Benutzer zunächst auffordert, eine Zahl größer gleich 2 einzugeben und das anschließend einen entsprechend hohen Weihnachtsbaum der folgenden Form auf den Bildschirm ausgibt:

Höhe = 2:

```
   +-+
   | |
 +-+ +-+
```

Höhe = 4:



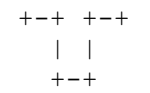
Hinweis: Implementieren Sie geeignete Funktionen, um Ihr Programm übersichtlich zu gestalten.

Aufgabe 12:

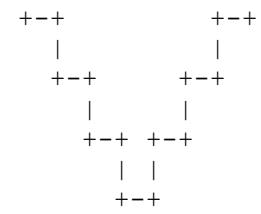
Sie sind Fußballfan und Sie möchten sich selbst mal eine Freude machen, indem Sie ein ASCII-Fußballstadion auf den Bildschirm Ihres Computers malen.

Aufgabe: Schreiben Sie ein Java-Programm, das den Benutzer zunächst auffordert, eine Zahl größer gleich 1 einzugeben und das anschließend ein Fußballstadion mit entsprechend hohen Tribünen in der folgenden Form auf den Bildschirm ausgibt:

Höhe = 1:



Höhe = 3:



Hinweis: Implementieren Sie geeignete Funktionen, um Ihr Programm übersichtlich zu gestalten.

Aufgabe 13:

Bei dieser Aufgabe geht es darum, zu berechnen, ob sich in einem rechtwinkligen Koordinatensystem zwei Kreise schneiden oder nicht.

Ein Benutzer des Programms soll zunächst insgesamt 6 double-Werte eingeben können:

- x-Koordinate des ersten Kreises
- Y-Koordinate des ersten Kreises
- Radius des ersten Kreises (positiver Wert!)
- x-Koordinate des zweiten Kreises

- y-Koordinate des zweiten Kreises
- Radius des zweiten Kreises (positiver Wert!)

Anschließend soll das Programm berechnen, ob sich die beiden Kreise schneiden oder nicht. Im ersten Fall soll die Ausgabe "Kreise schneiden sich!", im zweiten Fall die Ausgabe "Kreise schneiden sich nicht!" erfolgen.

Beispielablauf (Benutzereingaben in <>):

```
x-Koordinate Kreis 1 eingeben: <1.0>
Y-Koordinate Kreis 1 eingeben: <1.0>
Radius Kreis 1 eingeben: <1.0>
x-Koordinate Kreis 2 eingeben: <2.0>
Y-Koordinate Kreis 2 eingeben: <2.0>
Radius Kreis 2 eingeben: <1.0>
Kreise schneiden sich!
```

Zum Berechnen der Wurzel eines double-Wertes stellt Java übrigens folgende Funktion zur Verfügung: `public static double Math.sqrt(double zahl).` Die können Sie nutzen: `double wurzel = Math.sqrt(47.11);`

Aufgabe 14:

Schreiben Sie ein Java-Programm, das den Benutzer zunächst auffordert, einen positiven int-Wert `groesse` einzugeben und das anschließend ein entsprechend großes „eingepacktes Bonbon“ der folgenden Form auf den Bildschirm ausgibt:

groesse = 2:

```
\  /
 \/
 /\
/  \
----
|  |
|  |
----
\  /
 \/
 /\
/  \
```

groesse = 3:

```
\    /
 \  /
  \/
  /\
 /  \
/    \
------
|    |
|    |
|    |
------
\    /
 \  /
  \/
```

Hinweis: Gehen Sie nach dem Prinzip der „prozeduralen Zerlegung“ vor. Definieren Sie bspw. eine Funktion `int groesseAbfragen()` die einen gültigen int-Eingabewert (> 0) abfragt und liefert. Definieren Sie dann bspw. zwei weitere Prozeduren `void verpackungsknotenZeichnen(int groesse)` und `void bonbonZeichnen(int groesse)`. Erstere kann dabei zweimal genutzt werden: zum Zeichnen des oberen und zum Zeichnen des unteren Knoten des Verpackungspapiers. Die zweite Prozedur zeichnet das eigentliche Bonbon, d.h. den mittleren Teil.

Aufgabe 15:

Sie sollen das bekannte Spiel „Schnick-Schnack-Schnuck“ so implementieren, dass es der Computer gegen einen Menschen spielen kann.

Regeln: Bei Schnick-Schnack-Schnuck werden mehrere Spielrunden absolviert. In jeder Spielrunde wählen die beiden Spieler gleichzeitig eines der folgenden Symbole: Brunnen, Schere, Stein oder Papier. Dabei gilt:

- Brunnen gewinnt gegen Schere
- Brunnen gewinnt gegen Stein
- Brunnen verliert gegen Papier
- Schere verliert gegen Stein
- Schere gewinnt gegen Papier
- Stein verliert gegen Papier

Der Sieger erhält jeweils einen Punkt. Wählen die beiden Spieler dasselbe Symbol, ist das ein Unentschieden und keiner der beiden Spieler erhält einen Punkt.

Ein Spiel endet, wenn ein Spieler insgesamt 10 Punkte erreicht. Dieser Spieler ist Sieger.

Ablauf: In jeder Spielrunde generiert der Computer per Zufall eines der vier Symbole. Anschließend fordert er den menschlichen Spieler zur Eingabe eines Symbols auf. Er ermittelt das Ergebnis und gibt es auf den Bildschirm aus.

Beispiel für einen Programmablauf (Benutzereingaben in <>):

```
Spielrunde 1:
Symbol eingeben (Brunnen, Schere, Stein, Papier): -> Schere
Computer: Brunnen; Mensch: Schere -> Computer gewinnt
Spielstand: Computer = 1; Mensch = 0
…
Spielrunde 21:
Symbol eingeben (Brunnen, Schere, Stein, Papier): -> Stein
Computer: Brunnen; Mensch: Stein -> Computer gewinnt
Spielstand: Computer = 10; Mensch = 5

Spielende: Computer hat gewonnen
```

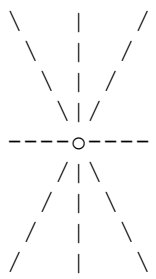
Hinweis: Implementieren Sie geeignete Funktionen, um Ihr Programm übersichtlich zu gestalten.

Aufgabe 16:

Bald ist Winter, und hoffentlich wird bald der erste Schnee fallen. Sie sollen diesem Tag vorgreifen und schon jetzt Schneeflocken auf den Bildschirm zaubern.

Aufgabe: Schreiben Sie ein Java-Programm, das den Benutzer zunächst auffordert, eine Zahl größer gleich 0 einzugeben und das anschließend eine entsprechend große Schneeflocke der folgenden Form auf den Bildschirm ausgibt:

Größe = 4:



Hinweis: Implementieren Sie geeignete Funktionen, um Ihr Programm übersichtlich zu gestalten.

Aufgabe 17:

Schreiben Sie ein Java-Programm, das die Anzahl an gleichen Ziffern zweier eingelesener Zahlen berechnet.

Zunächst sollen vom Benutzer zwei positive int-Werte eingelesen werden. Anschließend soll die Anzahl an gleichen Ziffern in den beiden Zahlen berechnet werden. Jede Ziffer darf dabei nur einmal berücksichtigt werden! Zum Schluss soll die Anzahl an gleichen Ziffern auf den Bildschirm ausgegeben werden.

Beispiele**:**

```
Zahl: 12367
Zahl: 4005
Anzahl an gleichen Ziffern: 0

Zahl: 123
Zahl: 222444
Anzahl an gleichen Ziffern: 1

Zahl: 5656
Zahl: 6755
Anzahl an gleichen Ziffern: 3
```

Hinweis: Implementieren und nutzen Sie eine Funktion

```
int getAnzahlGleicheZiffern(int zahl1, int zahl2).
```

Aufgabe 18:

Bei dieser Aufgabe sollen Sie einen Mathetrainer zum Üben des Multiplizierens und Dividierens implementieren.

Der Mathetrainer generiert dabei jeweils zufällig zwei int-Werte zwischen 0 und 9 sowie zufällig entweder den Multiplikations- oder den Divisionsoperator. Er präsentiert die Aufgabe und fordert den Benutzer auf, das Ergebnis einzugeben. Gibt dieser das falsche Ergebnis ein, teilt ihm der Mathetrainer das korrekte Ergebnis mit. Gibt der Benutzer das korrekte Ergebnis ein, wird er gelobt. In beiden Fällen wird anschließend die Gesamtanzahl der bisherigen korrekten Antworten ausgegeben. Das Ganze wird wiederholt, bis der Benutzer 10 korrekte Antworten gegeben hat.

Beispiel für einen Programmablauf (in <> stehen Benutzereingaben):

```
Start des Mathetrainers
8 / 1 = <8>
Richtig!
Korrekte Antworten: 1
9 / 2 = <4>
Richtig!
Korrekte Antworten: 2
6 * 1 = <4>
Leider falsch! Korrektes Ergebnis ist 6
Korrekte Antworten: 2
0 / 8 = <0>
Richtig!
Korrekte Antworten: 3

...

3 * 2 = <6>
Richtig!
Korrekte Antworten: 9
0 * 5 = <0>
Richtig!
Korrekte Antworten: 10
Ende des Mathetrainers
```

Hinweis: Implementieren und verwenden Sie dabei geeignete Funktionen, um Ihr Programm übersichtlich zu gestalten, z.B.:

- `static boolean frageBearbeiten()` zum Bearbeiten einer einzelnen Frage
- `static int generiereZahl()` zum Generieren einer Zahl zwischen 0 und 9
- `static char generiereOperator()` zum Generieren einer der beiden Operatoren

Aufgabe 19:

Beim Spiel *Mäxchen* würfelt ein Spieler zwei Würfel. Würfelt er eine 1 und eine 2, ein so genanntes *Mäxchen*, bekommt er 1000 Punkte. Würfelt er einen Pasch, d.h. zwei gleiche Zahlen, erhält er das Hundertfache der entsprechenden Zahl (also bspw. 400

bei zwei 4en) als Punkte, ansonsten ist die Punktzahl 10 * höhere Augenzahl + niedrigere Augenzahl. Der Wurf 3, 5 hat also bspw. den Wert 53.

Implementieren Sie eine Funktion `maexchen`, die zwei gewürfelte Zahlen als Parameter übergeben bekommt und die erzielten Punkte als Funktionswert liefert.

Schreiben Sie dann ein Programm, in dem die Funktion `maexchen` mit jeweils zwei zufällig erzeugten Zahlen zwischen 1 und 6 so oft aufgerufen wird, bis 100.000 Punkte erreicht sind. Anschließend soll der prozentuale Anteil der Mäxchen-Würfe als `double`-Wert ausgegeben werden.

Aufgabe 20:

Bei dieser Aufgabe geht es darum, ein Programm zu entwickeln, das es erlaubt, die Reaktionszeit der Nutzer zu messen. Der Programmablauf sieht folgendermaßen aus:

- Zunächst wird „Achtung: Start“ ausgegeben.
- Dann werden zwischen 5 und 10 Runden absolviert. Die genaue Zahl der Runden wird per Zufall bestimmt. (`getRandomNumber(<max>)`)
- In jeder Runde wird folgendes gemacht:
  - Es wird zunächst zwischen 2 und 5 Sekunden gewartet. (`Util.wait(<sec>)`). Die genaue Wartezeit wird per Zufall bestimmt.
  - Dann bestimmt das Programm per Zufall einen Kleinbuchstabe (‚a‘ – ‚z‘) und gibt ihn auf dem Bildschirm aus. (Alternative: Zahl)
  - Der Benutzer muss jetzt versuchen, so schnell wie möglich den Buchstaben einzugeben (inkl. <Enter>). Die Zeit zwischen Ein- und Ausgabe wird gemessen (`Util.getMilliseconds()`)
  - Falls der ausgegebene Kleinbuchstabe und das eingegebene Zeichen gleich sind, wird die Zeit zwischen Ausgabe des Buchstabens und Eingabe des Nutzers verarbeitet (die Reaktionszeit). Ansonsten merkt sich das Programm einen Fehler.
- Sind alle Runden absolviert, gibt das Programm „Geschafft: Ende“ aus. Weiterhin werden ausgegeben:
  - Anzahl an Fehlversuchen
  - Mittelwert der Reaktionszeiten
  - Langsamster Versuch
  - Schnellster Versuch

Beispiel für einen Programmablauf (Benutzereingaben stehen in Klammern (<>)):

```
Achtung: Start!
b
<b>
u
<u>
v
<d>
m
```

```
<m>
a
<a>
Geschafft: Ende!
Fehlversuche: 1 von 5
Reaktionszeit-Mittelwert: 2.5748 Sekunden
Langsamster Versuch: 5.813 Sekunden
Schnellster Versuch: 1.218 Sekunden
```

Folgende Funktionen sind dabei vorgegeben:

```
// liefert eine Zahl, die die aktuelle Zeit in Millisek repräsentiert
static long getMilliSeconds() {
  return new java.util.Date().getTime();
}

// liefert eine Zufallszahl zwischen 0 und max (einschließlich)
static int getRandomNumber(int max) {
  return new java.util.Random().nextInt(max + 1);
}

// hält das Programm seconds Sekunden an
static void wait(int seconds) {
  try {
    Thread.sleep(seconds * 1000);
  } catch (Exception exc) {
  }
}

// liefert den größt-möglichen long-Wert
static long getMaxLongNumber() {
  return Long.MAX_VALUE;
}
```

Aufgabe 21:

Bald ist Weihnachten und Sie möchten sich selbst mal eine Freude machen, indem Sie einen ASCII-Weihnachtsbaum auf den Bildschirm Ihres Computers malen.

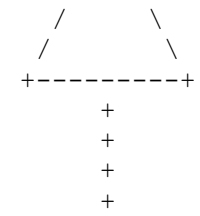
Aufgabe: Schreiben Sie ein Java-Programm, das den Benutzer zunächst auffordert, eine Zahl („Höhe“) größer gleich 1 einzugeben und das anschließend einen entsprechend hohen Weihnachtsbaum der folgenden Form auf den Bildschirm ausgibt:

Höhe = 1:

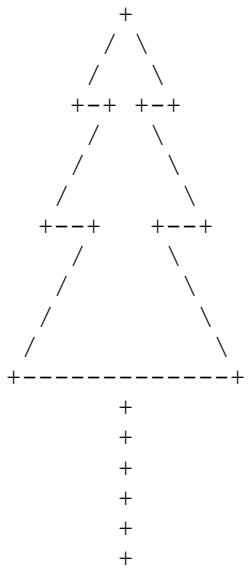
```
   +
  / \
 /   \
+-----+
   +
   +
```

Höhe = 2:

```
   +
  / \
 /   \
+-+ +-+
 /   \
```



Höhe = 3:



Hinweis: Implementieren Sie geeignete Funktionen, um Ihr Programm übersichtlich zu gestalten.

Aufgabe 22:

„Glückspiel“ ist ein Spiel für zwei Spieler. Es wird über mehrere Spielrunden gespielt bis ein Spieler 5 Punkte erreicht hat. Dieser Spieler hat dann gewonnen. Eine Spielrunde läuft wie folgt ab:

- Der Computer generiert eine geheime Zufallszahl zwischen 0 und 9.
- Die Spieler geben nacheinander eine Zahl ein. Dabei darf der zweite Spieler nicht dieselbe Zahl eingeben wie der erste Spieler.
- Für jeden Spieler wird die Differenz zwischen seiner Zahl und der generierten Zufallszahl berechnet. Der Spieler mit der betragsmäßig niedrigeren Differenz erhält einen Punkt. Ist die Differenz bei beiden Spielern gleich, erhält keiner der Spieler einen Punkt.

Damit es fair zugeht, wechselt in jeder Spielrunde der Spieler, der als erster seine Zahl eingibt. In der ersten Spielrunde beginnt Spieler 1, in der zweiten Spieler 2, usw.

Implementieren Sie das Spiel „Glückspiel“ derart, dass es zwei menschliche Spieler gegeneinander spielen können. Behandeln Sie fehlerhafte Benutzereingaben adäquat. Orientieren Sie sich an dem folgenden beispielhaften Programmablauf (Eingaben stehen in <>):

```
Ich habe eine Zufallszahl zwischen 0 und 9 generiert!
Spieler 1, deine Zahl: <5>
Spieler 2, deine Zahl: <3>
Zufallszahl war die 2
```

```
Spieler 2 gewinnt die Runde
Spielstand: Spieler 1 hat 0 Punkte; Spieler 2 hat 1 Punkte.

Ich habe eine Zufallszahl zwischen 0 und 9 generiert!
Spieler 2, deine Zahl: <2>
Spieler 1, deine Zahl: <7>
Zufallszahl war die 7
Spieler 1 gewinnt die Runde
Spielstand: Spieler 1 hat 1 Punkte; Spieler 2 hat 1 Punkte.

Ich habe eine Zufallszahl zwischen 0 und 9 generiert!
Spieler 1, deine Zahl: <5>
Spieler 2, deine Zahl: <6>
Zufallszahl war die 5
Spieler 1 gewinnt die Runde
Spielstand: Spieler 1 hat 2 Punkte; Spieler 2 hat 1 Punkte.

...
Ich habe eine Zufallszahl zwischen 0 und 9 generiert!
Spieler 1, deine Zahl: <4>
Spieler 2, deine Zahl: <5>
Zufallszahl war die 7
Spieler 2 gewinnt die Runde
Spielstand: Spieler 1 hat 2 Punkte; Spieler 2 hat 5 Punkte.

Spieler 2 hat gewonnen
```

Hinweis: Implementieren Sie geeignete Funktionen, um Ihr Programm übersichtlich zu gestalten.

Aufgabe 23:

Ein Marathonlauf hat eine Länge von 42,195 km. Für einen Marathonläufer, der eine bestimmte Zielzeit anvisiert, ist es wichtig zu wissen, mit welchen Zeiten er die einzelnen Kilometer laufen muss, um mindestens seine Zielzeit zu erreichen. Daran kann er einschätzen, ob er zu schnell oder zu langsam ist. Er will dabei natürlich möglichst gleichmäßig schnell laufen. Helfen Sie ihm beim Berechnen der einzelnen Kilometerzeiten.

Aufgabe: Implementieren Sie ein Programm, das den Benutzer zunächst nach den anvisierten Stunden und anschließend nach den anvisierten Minuten beim anstehenden Marathon fragt. Anschließend soll das Programm berechnen und ausgeben, bei welchen Zeiten (Stunden:Minuten:Sekunden) er die einzelnen Kilometer passieren muss, um bei einem möglichst gleichmäßigen Tempo möglichst exakt aber mindestens seine Zielzeit zu erreichen. Weiterhin soll das Programm die entsprechende Halbmarathon- und Marathonzeit berechnen und ausgeben.

Orientieren Sie sich, was den Programmablauf als auch was die Ein- und Ausgaben angeht, an folgendem Beispiel (Benutzereingaben stehen in <>):

```
Anvisierte Stunden: <3>
Anvisierte Minuten: <30>
1 KM = 00:04:58
2 KM = 00:09:57
3 KM = 00:14:55
...
13 KM = 01:04:41
14 KM = 01:09:40
15 KM = 01:14:39
...
24 KM = 01:59:26
```

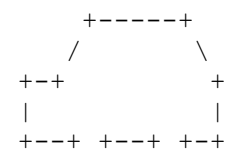
```
25 KM = 02:04:25
26 KM = 02:09:23
27 KM = 02:14:22
...
37 KM = 03:04:08
38 KM = 03:09:07
39 KM = 03:14:05
40 KM = 03:19:04
41 KM = 03:24:03
42 KM = 03:29:01
Halbmarathon = 01:44:59
Marathon = 03:29:59
```

Hinweis: Implementieren Sie gegebenenfalls geeignete Funktionen, um Ihr Programm übersichtlich zu gestalten.

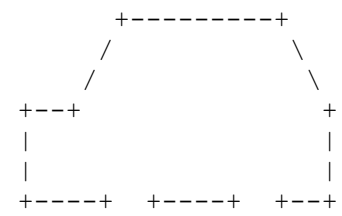
Aufgabe 24:

Schreiben Sie ein Java-Programm, das den Benutzer zunächst zur Eingabe eines positiven `int`-Wertes `n` auffordert und anschließend ein „Auto“ in der im Folgenden skizzierten Form auf den Bildschirm ausgibt. Überlegen Sie anhand der Beispiele selbst, wo und in welcher Form sich der Wert von `n` auf die Größe und Gestalt des Autos auswirkt.

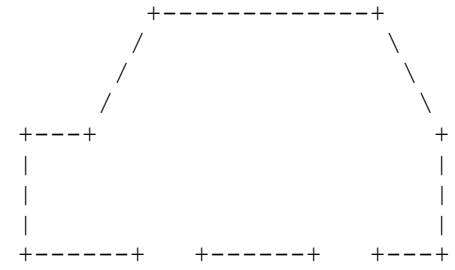
Beispiel (n = 1):



Beispiel (n = 2):



Beispiel (n = 3):



Hinweis: Implementieren Sie gegebenenfalls geeignete Funktionen, um Ihr Programm übersichtlich zu gestalten.

Aufgabe 25:

Die sogenannte binäre Exponentiation (auch *Square & Multiply* genannt) ist eine effiziente Methode zur Berechnung von natürlichen Potenzen, also Ausdrücken der Form x^k mit einer reellen Zahl x und einer natürlichen Zahl k (Quelle: Wikipedia)

Der Algorithmus hat folgende Form:

- Umwandlung von k in die zugehörige Binärdarstellung.
- In der Binärdarstellung ersetzen jeder 0 durch Q und jeder 1 durch QM.
- Nun wird Q als Anweisung zum Quadrieren und M als Anweisung zum Multiplizieren aufgefasst.
- Somit bildet die resultierende Zeichenkette von links nach rechts gelesen eine Vorschrift zur Berechnung von x^k. Man beginne mit 1, quadriere für jedes gelesene Q das bisherige Zwischenergebnis und multipliziere es für jedes gelesene M mit x.

Beispiel: Sei k = 23. Die Binärdarstellung von 23 lautet 10111. Daraus ergibt sich nach den Ersetzungen QMQQMQMQM. Beginnen wir nun mit 1. Als erstes wird quadriert und mit x multipliziert. Es ergibt sich als Zwischenergebnis x. Der anschließende Rechenvorgang sieht dann noch folgendermaßen aus: „quadriere, quadriere, multipliziere mit x, quadriere, multipliziere mit x, quadriere, multipliziere mit x“.

Sukzessive geschrieben sieht das Ganze folgendermaßen aus:

$$\underset{}{1} \overset{Q}{\rightarrow} 1 \overset{M}{\rightarrow} x \overset{Q}{\rightarrow} x^2 \overset{Q}{\rightarrow} x^4 \overset{M}{\rightarrow} x^5 \overset{Q}{\rightarrow} x^{10} \overset{M}{\rightarrow} x^{11} \overset{Q}{\rightarrow} x^{22} \overset{M}{\rightarrow} x^{23}$$

Man sieht am Beispiel, dass man sich mit Hilfe der binären Exponentiation einige Rechenschritte sparen kann. Anstatt von 22 Multiplikationen werden nur noch 9 benötigt, indem man fünfmal quadriert und viermal mit x multipliziert.

Aufgabe: Implementieren Sie eine Funktion `binExp`, die als Parameter einen `double`-Wert `x` und einen `int`-Wert `k` (Sie können davon ausgehen, dass dieser positiv ist) übergeben bekommt. Die Methode soll mit Hilfe des Algorithmus der binären Exponentiation den Wert von x^k berechnen und als Funktionswert liefern.

Hinweis: Zur Umrechnung von Dezimal- in Binärzahlen bedienen Sie sich bitte des bekannten Modulo-Algorithmus: Die Dezimalzahl wird fortwährend bis 0 durch 2 dividiert. Die entsprechende Binärzahl ergibt sich durch Notation der errechneten Reste von unten nach oben.

Beispiel: 41 im Dezimalsystem ist gleich 101001 im Binärsystem:

$$
\begin{array}{rcrcrl}
41 & :2 & = & 20 & \text{Rest} & \mathbf{1} \uparrow \\
20 & :2 & = & 10 & \text{Rest} & \mathbf{0} \\
10 & :2 & = & 5 & \text{Rest} & \mathbf{0} \\
5 & :2 & = & 2 & \text{Rest} & \mathbf{1} \\
2 & :2 & = & 1 & \text{Rest} & \mathbf{0} \\
1 & :2 & = & 0 & \text{Rest} & \mathbf{1}
\end{array}
$$

Aufgabe 27:

Implementieren Sie in Java ein Programm, das solange einzelne Zeichen vom Terminal einliest, bis ein `#`-Zeichen eingegeben wird, und anschließend die eingegebenen Zeichen in umgekehrter Reihenfolge wieder auf den Bildschirm ausgibt (Achtung: keine Arrays oder Strings verwenden!).

Beispiel:

```
Eingabe: a
         b
         c
         #
Ausgabe: cba
```

Aufgabe 28

Schreiben Sie ein Programm, das zunächst eine positive Zahl vom Terminal einliest. Anschließend soll die duale Repräsentation der Zahl auf dem Bildschirm ausgegeben werden. Entwickeln Sie eine rekursive Lösung, d.h. Sie dürfen keine Schleifen-Anweisungen (`while`, `for`, `do`, ...) benutzen.

Beispiel:

```
Eingabe: 23
Ausgabe: 10111
```

Aufgabe 29:

Schreiben Sie ein Java-Programm, dass solange Zahlen von der Tastatur einliest, bis der Nutzer eine 0 eingibt, und das diese Zahlen in umgekehrter Reihenfolge wieder auf den Bildschirm ausgibt. Eine Speicherung der Zahlen in einem Array oder einer ähnlichen Datenstruktur ist nicht erlaubt! Nutzen Sie Rekursion!

Aufgabe 30:

Implementieren Sie in Java eine **rekursive** Funktion

```
static double potenz(double zahl, int pot)
```

die die `pot`-te Potenz von `zahl` berechnet. Der Wert von `pot` kann auch negativ sein!. Schreiben Sie ein kleines Programm, das die Funktion testet.

Beispiele:

potenz(2.0, 3) == 8

potenz(2.0, -3) == 0.125

Aufgabe 31:

Folgendes Programm berechnet iterativ die Quersumme einer eingegebenen positiven Zahl:

```
public class Quersumme {

  public static int quersumme(int zahl) {
    int ergebnis = 0;
    while (zahl != 0) {
      ergebnis = ergebnis + zahl % 10;
      zahl /= 10;
    }
    return ergebnis;
  }

  public static void main(String[] args) {
    int eingabe = IO.readInt("Zahl (>=0): ");
    IO.println(quersumme(eingabe));
  }

}
```

Formen Sie das Programm um in ein gleichwertiges rekursives Programm (`while`, `for`, `do` sind nicht erlaubt!)!

Aufgabe 32:

Folgendes Programm berechnet iterativ die Spiegelzahl einer eingegebenen Zahl:

```
public class Spiegelzahl {

  public static int reverse(int zahl) {
    int ergebnis = 0;
    while (zahl != 0) {
      ergebnis = ergebnis * 10 + zahl % 10;
      zahl /= 10;
    }
    return ergebnis;
  }

  public static void main(String[] args) {
    int eingabe = IO.readInt("Zahl (>=0): ");
    IO.println(reverse(eingabe));
  }

}
```

Formen Sie das Programm um in ein gleichwertiges rekursives Programm (`while`, `for`, `do` sind nicht erlaubt!)!

Aufgabe 33:

Beim folgenden Java-Programm handelt es sich um einen rekursiven Primzahltest:

```
public class PrimzahltestRekursiv {
  public static boolean test(int teiler, int zahl) {
    if (zahl % teiler == 0) {
      return false;
    } else if (teiler > zahl/teiler) {
      return true;
    } else {
      return test(teiler + 2, zahl);
    }
  }

  public static void main(String[] args) {
    int eingabe = IO.readInt("Zahl (> 3): ");

    if ((eingabe % 2 != 0) && test(3, eingabe)) {
      IO.println(eingabe + " ist Primzahl");
    } else {
      IO.println(eingabe + " ist keine Primzahl");
    }
  }
}
```

Formen Sie das Programm um in ein gleichwertiges iteratives Java-Programm!

Aufgabe 34:

Das aus der Mathematik bekannte Pascalsche Dreieck hat folgende Gestalt

```
0                          1
1                      1       1
2                  1       2       1
3              1       3       3       1
4          1       4       6       4       1
5      1       5      10      10       5       1
6      .   .   .   .   .   .   .   .   .   .
```

(1) Ermitteln Sie das **rekursive** Bildungsgesetz für das Pascalsche Dreieck und schreiben Sie eine rekursive Java-Funktion `int pascal(int zeile, int spalte)`, die den Wert des Pascalschen Dreieck in der Zeile „`zeile`“ und der Spalte „`spalte`“ rekursiv berechnet (`while`, `for`, `do` sind nicht erlaubt!)!

(2) Schreiben Sie ein Java-Programm, das das Pascalsche Dreieck auf den Bildschirm ausgibt (die ersten 10 Zeilen).

Aufgabe 35:

Entwickeln Sie ein iteratives Java-Programm zur Lösung des Problems "Türme von Hanoi", d.h. der Einsatz von Rekursion ist nicht erlaubt.

Aufgabe 36:

Implementieren Sie eine Funktion mit folgender Signatur

```
static boolean zifferEnthalten(int zahl, int ziffer)
```

Als erster Parameter wird eine Zahl (beliebiger int-Wert) übergeben, als zweiter Parameter eine Ziffer (int-Wert zwischen 0 und 9). Die Funktion soll überprüfen, ob die Ziffer in der Zahl vorkommt. In diesem Fall (und nur dann) soll die Funktion den Wert `true` liefert. Ansonsten soll sie den Wert `false` liefern.

Beispiele:

```
zifferEnthalten(4567, 6) -> true
```

```
zifferEnthalten(-3356, 8) -> false
```

Aufgabe (a): Implementieren Sie die Funktion iterativ, d.h. durch Nutzung von Schleifen.

Aufgabe (b): Implementieren Sie die Funktion rekursiv, d.h. ohne Nutzung von Schleifen.

Aufgabe 37:

Schreiben Sie ein Java-Programm, das den Benutzer zunächst auffordert, eine Zahl größer gleich 2 einzugeben und das, sobald der Benutzer eine entsprechende Zahl eingegeben hat, einen entsprechend großen Pfeil der folgenden Form auf den Bildschirm ausgibt:

Zahl = 2:

```
 \
  \
  /
 /
```

Zahl = 4:

```
  \
   \
    \
     \
     /
    /
   /
  /
 /
```

Teilaufgabe (a): Sie sollen Wiederholungsanweisungen benutzen!

Teilaufgabe (b): Sie dürfen keine Wiederholungsanweisungen benutzen, sondern sollen stattdessen Rekursion einsetzen.

Aufgabe 38:

Die Zahl „Vier“ ist eine bedeutsame Zahl (4 Himmelsrichtungen, 4 Jahreszeiten, …). Und es ist möglich, ausgehend von der Zahl vier jede andere natürliche Zahl durch geeignete Operationen zu erzeugen. Folgende Operationen sind dafür geeignet:

- Man fügt am Ende die Ziffer 4 hinzu
- Man fügt am Ende die Ziffer 0 hinzu
- Man teilt durch 2 (wenn die Zahl gerade ist).

Schreiben Sie ein Java-Programm, bei dem der Benutzer eine natürliche Zahl eingibt und das dann die entsprechenden Zwischenzahlen auf dem Weg zu Erzeugung der Zahl gemäß den obigen Regeln ausgibt.

Beispiel: Eingabe = 2524

```
4
2
1
10
5
50
504
252
2524
```

Tipp: Gehen Sie den umgekehrten Weg ausgehend von der zu erzeugenden Zahl hin zur Zahl vier.

Aufgabe 39:

Schreiben Sie ein Programm, bei dem der Nutzer zunächst aufgefordert wird, eine positive Zahl einzugeben. Das Programm soll anschließend entsprechend viele * auf den Bildschirm ausgeben. Achtung: Sie dürfen keine Schleifen verwenden!

Beispiel:

```
Eingabe = 8
Ausgabe: ********
```

Aufgabe 40:

Schreiben Sie ein Java-Programm, das die Anzahl der Überträge beim Addieren zweier Zahlen berechnet.

Algorithmus: Zunächst sollen vom Benutzer zwei nicht negative Zahlen eingelesen werden. Anschließend soll die Anzahl an Überträgen beim ziffernweisen Addieren der beiden Zahlen von rechts nach links berechnet werden. Zum Schluss soll die Anzahl an Überträgen auf den Bildschirm ausgegeben werden. Achtung: Sie dürfen keine Schleifen verwenden!

Beispiel:

```
Zahl: 123
Zahl: 594
Anzahl an Uebertraegen: 1
```

Aufgabe 41:

Denken Sie sich eine Zahl zwischen 1 und 100 (bzw. eine beliebige Zahl N) aus. Der Computer soll sie in möglichst wenigen Schritten erraten. Implementieren Sie dazu das Halbierungsverfahren. D.h. der Computer fragt zunächst ab, ob sich die Zahl zwischen 1 und 50 befindet. Ist das der Fall, fragt er als nächstes, ob sie sich zwischen 1 und 25 befindet. Ist dies nicht der Fall, muss sie sich ja zwischen 26 und 50 befinden und der Computer fragt daher, ob sie sich zwischen 26 und 38 befindet, usw.

Teilaufgabe (a): Entwickeln Sie eine iterative (Einsatz von Schleifen) Lösung.

Teilaufgabe (b): Entwickeln Sie eine rekursive (ohne Schleifen) Lösung.

Beispiel für einen Programmablauf, bei dem Sie sich die Zahl 37 ausgedacht haben (Benutzereingaben in Klammern <>):

```
Liegt die Zahl zwischen 1 und 50? (j/n)<j>
Liegt die Zahl zwischen 1 und 25? (j/n)<n>
Liegt die Zahl zwischen 26 und 38? (j/n)<j>
Liegt die Zahl zwischen 26 und 32? (j/n)<n>
Liegt die Zahl zwischen 33 und 35? (j/n)<n>
Liegt die Zahl zwischen 36 und 37? (j/n)<j>
Liegt die Zahl zwischen 36 und 36? (j/n)<n>
Ihre Zahl ist die 37
```

Aufgabe 42:

Schreiben Sie ein Java-Programm, das den Benutzer zunächst auffordert, einen positiven int-Wert `groesse` einzugeben und das anschließend entsprechend große „Blöcke“ auf den Bildschirm zeichnet, und zwar `groesse` mal.

groesse = 2:

```
+--+
|  |
|  |
+--+
|  |
|  |
+--+
```

groesse = 3:

```
+---+
|   |
|   |
|   |
+---+
|   |
|   |
|   |
+---+
|   |
|   |
|   |
+---+
```

Aufgabe 43:

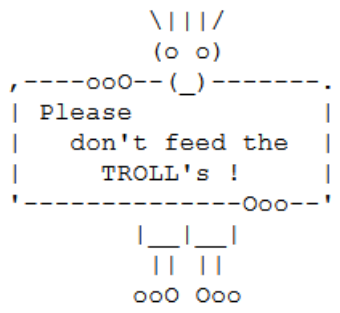
Schreiben Sie ein Java-Programm, das den Benutzer zunächst zur Eingabe eines positiven `int`-Wertes `n` auffordert und anschließend jeweils „Figuren“ in der im Folgenden skizzierten Form auf den Bildschirm ausgibt. Überlegen Sie anhand der Beispiele selbst, wo und in welcher Form sich der Wert von `n` auf die Größe und Gestalt der Figuren auswirkt.

```
Beispiel (n = 1):

+
 \
  +
 /
+

Beispiel (n = 2):

+
 \
  +
 /
+
 \
  \
   +
  /
 /
+
```

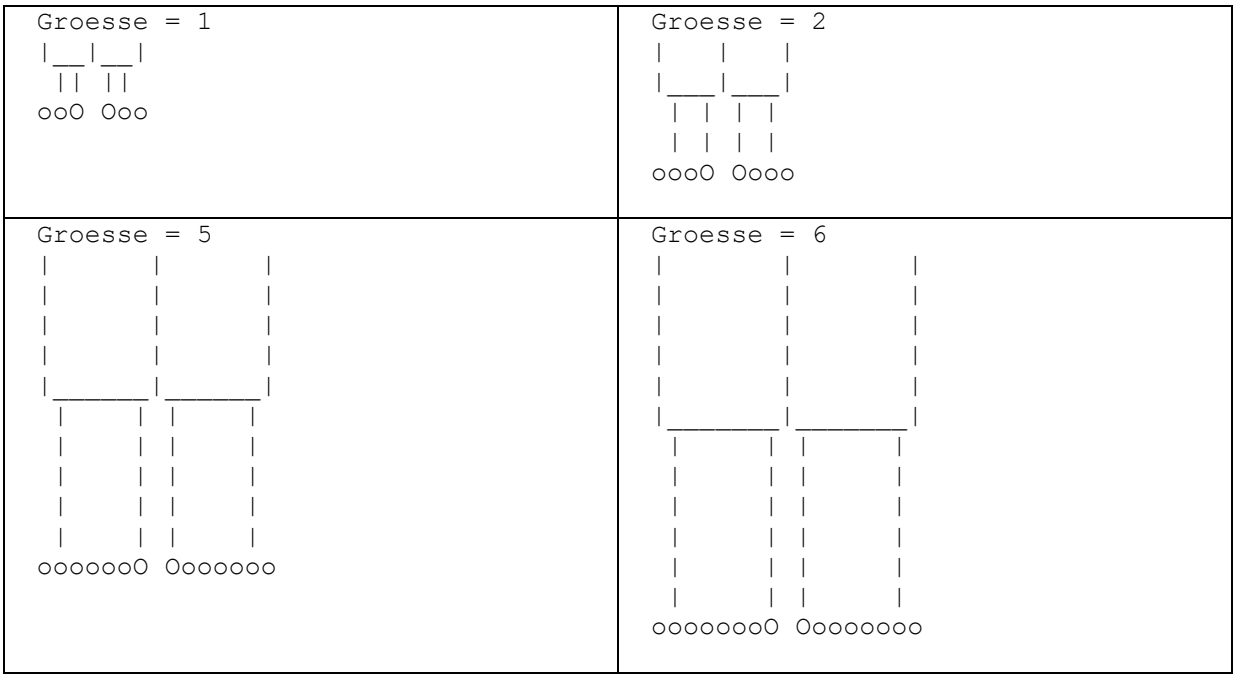
```
Beispiel (n = 3):

+
 \
  +
 /
+
 \
  \
   +
  /
 /
+
 \
  \
   \
    +
   /
  /
 /
+
```

Aufgabe 44:

Sicher kennen Sie die folgende ASCII-Dont-Feed-the-Trolls-Zeichnung:



Für diese Aufgabe von Bedeutung ist nur der untere Teil. Schreiben Sie ein Java-Programm, das vom Benutzer einen int-Wert `groesse` abfragt, der größer als 0 ist. Anschließend sollen die Hose, die Beine und die Zehen des Trolls in der entsprechenden Größe auf den Bildschirm gezeichnet werden. Arrays dürfen in dieser Aufgabe nicht benutzt werden!



Aufgabe 45:

Bleistift-Tennis (siehe http://de.wikipedia.org/wiki/Tennis_(Bleistiftspiel)) ist ein strategisches Papier-und-Bleistift-Spiel für zwei Spieler S_1 und S_2. Das Spielfeld besteht aus vier Feldern mit den Bezeichnungen -2, -1, 1 und 2 und einer Mittellinie mit der Bezeichnung 0. Die negativen Felder gehören zu Spieler S_1 und die positiven zu Spieler S_2. Zu Beginn befindet sich der Ball auf der Mittellinie. Jeder Spieler besitzt Punkte, Spieler S_1 P_1 Punkte, Spieler S_2 P_2 Punkte. Anfangs besitzen beide Spieler N Punkte (konkret N = 50).

Ein Match besteht aus mehreren Spielrunden. In jeder Spielrunde wählt jeder Spieler S_i eine Zahl zwischen 0 und P_i. Die Punktzahl beider Spieler wird jeweils um ihre gewählte Zahl reduziert. Ist die gewählte Zahl beider Spieler identisch, verbleibt der Ball an seiner aktuellen Position; andernfalls gewinnt der Spieler mit der größeren gewählten Zahl die Runde. Befindet sich der Ball aktuell in der Spielhälfte des Gewinners, so wird er auf das betragsmäßig kleinste Feld des Verlierers platziert. Befindet sich der Ball bereits in der Spielhälfte des Verlierers, so wird der Ball um ein Feld in Richtung des Verlierers bewegt. Ziel des Spiels ist es, den Ball über die Grundlinie des anderen Spielers herauszuschlagen.

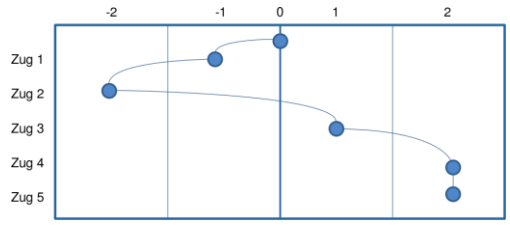
Für den Spielausgang wird festgelegt:

- Das Spiel wird beendet, wenn der Ball entweder über eine der Grundlinien geschlagen wurde, oder wenn kein Spieler mehr Punkte besitzt.
- In beiden Fällen wird das Spiel für den Spieler als verloren gewertet, auf dessen Seite der Ball liegt.
- Überlegen Sie selbst, ob es prinzipiell auch ein Unentschieden geben kann.

Der Reiz des Spiels besteht darin, dass die Wahl eines hohen Zuges zwar den Ball auf die Seite des Gegners bringt, aber gleichzeitig weniger Punkte als beim Gegner für die kommenden Züge verbleiben.

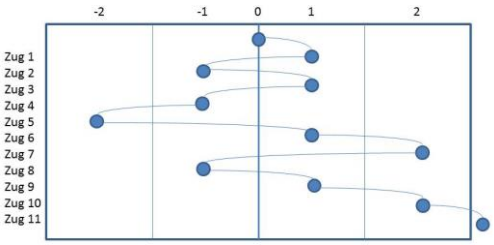
Beispiel 1: Im ersten Beispiel gewinnt Spieler 1, nachdem beide Spieler keine Punkte mehr haben (Ball noch im Feld).

t	Spieler 1 Zug $S_{1,t}$	Spieler 2 Zug $S_{2,t}$	Spieler 1 Status $Z_{1,t}$	Spieler 2 Status $Z_{2,t}$	Ballort B_t	Kommentar
0			50	50	0	Start
1	5	10	45	40	-1	
2	5	10	40	30	-2	
3	15	10	25	20	1	
4	15	10	10	10	2	
5	10	10	0	0	2	Spieler 1 gewinnt



Beispiel 2: Im zweiten Beispiel gewinnt Spieler 1, indem er mit seinen letzten Punkten den Ball über die Grundlinie hinaus schlagen kann.

t	Spieler 1 Zug $S_{1,t}$	Spieler 2 Zug $S_{2,t}$	Spieler 1 Status $Z_{1,t}$	Spieler 2 Status $Z_{2,t}$	Ballort B_t	Kommentar
0			50	50	0	Start
1	11	3	39	47	1	
2	1	10	38	37	-1	
3	15	11	23	26	1	
4	1	9	22	17	-1	
5	3	6	19	11	-2	
6	11	3	8	8	1	
7	4	3	4	5	2	
8	1	5	3	0	-1	
9	2	0	1	0	1	
10	1	0	1	0	2	
11	0	0	0	0	3	Spieler 1 gewinnt



Aufgabe: Schreiben Sie ein imperatives Java-Programm, das es zwei menschlichen Spielern erlaubt, auf einer Computer-Konsole gegeneinander Bleistift-Tennis zu spielen. Orientieren Sie sich, was die Ein- und Ausgaben des Programms angeht, an folgendem Beispielablauf; bezogen auf obiges Beispiel 1 (Benutzereingaben in <>):

```
Ball im Feld 0
Punktzahl Spieler 1 = 50
Punktzahl Spieler 2 = 50

Spieler 1! Zahl (>= 0; <= 50): <5>
Spieler 2! Zahl (>= 0; <= 50): <10>
Ball im Feld -1
Punktzahl Spieler 1 = 45
Punktzahl Spieler 2 = 40

Spieler 1! Zahl (>= 0; <= 45): <5>
Spieler 2! Zahl (>= 0; <= 40): <10>
Ball im Feld -2
Punktzahl Spieler 1 = 40
Punktzahl Spieler 2 = 30

Spieler 1! Zahl (>= 0; <= 40): <15>
Spieler 2! Zahl (>= 0; <= 30): <10>
Ball im Feld 1
Punktzahl Spieler 1 = 25
Punktzahl Spieler 2 = 20

Spieler 1! Zahl (>= 0; <= 25): <15>
Spieler 2! Zahl (>= 0; <= 20): <10>
Ball im Feld 2
```

```
Punktzahl Spieler 1 = 10
Punktzahl Spieler 2 = 10

Spieler 1! Zahl (>= 0; <= 10): <10>
Spieler 2! Zahl (>= 0; <= 10): <10>
Ball im Feld 2
Punktzahl Spieler 1 = 0
Punktzahl Spieler 2 = 0

Sieger ist Spieler 1
```

Ignorieren Sie einfach die Tatsache, dass Spieler 2 hierbei natürlich im Vorteil ist, da er die Eingabe von Spieler 1 sehen kann. Nehmen Sie an, die Spieler müssen vor dem Spielzug die gewählte Zahl gleichzeitig geheim auf einem Zettel notieren und dann diese Zahl auch eingeben.

Aufgabe 46:

Der *Barcode 2/5 Industrial* wird zur Identifizierung von Objekten verwendet. Mit dem Barcode lassen sich beliebige nicht-negative ganze Zahlen codieren.

Nachfolgend wird am Beispiel der Zahl 1984 erläutert, wie die Codierung erfolgt.

1. Schritt:

Die Zahl wird ziffernweise in eine Folge von Nullen und Einsen überführt (pro Ziffer 5-stellig). Dabei wird die folgende Tabelle benutzt:

Ziffer	1.Stelle	2.Stelle	3.Stelle	4.Stelle	5.Stelle
0	0	0	1	1	0
1	1	0	0	0	1
2	0	1	0	0	1
3	1	1	0	0	0
4	0	0	1	0	1
5	1	0	1	0	0
6	0	1	1	0	0
7	0	0	0	1	1
8	1	0	0	1	0
9	0	1	0	1	0

Die Zahl 1984 wird in die Folge 10001010101001000101 überführt.

2. Schritt:

Zur Berechnung einer Prüfziffer werden die Ziffern der gegebenen Zahl von links nach rechts abwechselnd mit dem Faktor 3 oder mit dem Faktor 1 multipliziert. Begonnen wird links mit dem Faktor 3. Die entstandenen Produkte werden addiert. Die Prüfziffer ist die kleinste nichtnegative ganze Zahl, die zu dieser Summe zu addieren ist, um ein Vielfaches von 10 zu erhalten.

Für 1984 ergibt sich die Summe 1*3 + 9*1 + 8*3 + 4*1 = 40 und daher die Prüfziffer 0. Die Prüfziffer wird mit Hilfe der Tabelle in eine Folge von Nullen und Einsen

überführt und an die bisherige Folge angehängt. Für 1984 erhält man nun die Folge 10001010101001000101001 10.

3. Schritt:

Am Anfang und am Ende der Folge werden ein Start- und ein Stoppzeichen ergänzt. Das Startzeichen wird durch 110, das Stoppzeichen durch 101 dargestellt. Für 1984 ergibt sich nun die vollständige Folge 11010001010101001000101001 10101.

Bei der Ausgabe des Barcodes wird für jede 1 ein breiter und für jede 0 ein schmaler schwarzer Strich gedruckt.

Aufgabe:

Implementieren Sie eine Methode

```
static String berechneBarcode(int zahl)
```

die eine nicht-negative ganze Zahl übergeben bekommt, den Barcode 2/5 Industrial gemäß dem oben angegebenen Algorithmus berechnet und diesen als String liefert. Implementieren und nutzen Sie weiterhin geeignete Hilfsmethoden für die Umsetzung der einzelnen Teilschritte.

(Quelle: Abiturprüfung Grundfach Informatik Thüringen 2003, http://www.erasmus-reinhold-gymnasium.de/inf/abi/inf_gf_2003.pdf)

Aufgabe 47:

Die International Bank Account Number, kurz IBAN genannt, ist eine international standardisierte Nummer, welche jedes Girokonto in einem der an diesem System teilnehmenden Länder eindeutig bezeichnet und definiert. Ab dem Jahr 2014 ersetzt die IBAN in der EU die bestehenden Bankleitzahlen und Kontonummern bei Überweisungen.

Aufgabe: Implementieren Sie in Java eine Funktion

```
static String get_DE_IBAN(int blz, long kontonummer)
```

der eine gültige 8-stellige deutsche Bankleitzahl (blz) und eine gültige 1- bis 10-stellige deutsche Kontonummer (kontonummer) übergeben wird und die die entsprechende IBAN berechnet und als String zurückliefert.

Algorithmus zur Berechnung der IBAN in Deutschland:

Die IBAN beginnt immer mit dem Länderkennzeichen (DE für Deutschland) und der zweistelligen Prüfsumme für die gesamte IBAN, die aufgrund einer genau festgelegten Formel berechnet wird (s.u.). Es folgen die 8 Stellen lange Bankleitzahl und die 10-stellige Kontonummer (hat die Kontonummer keine 10 Stellen, werden die fehlenden Stellen von vorn mit Nullen aufgefüllt).

Die Berechnung der zweistelligen Prüfsumme erfolgt in mehreren Schritten. Zunächst wird eine **24**-stellige (**Achtung**!) Zahl erzeugt. Diese setzt sich in Deutschland wie folgt zusammen: Die 8-stellige Bankleitzahl gefolgt von der 10-stelligen Kontonummer, gefolgt von der Ziffernfolge 1314, gefolgt von 00. Diese 24-stellige Zahl wird anschließend Modulo 97 genommen. Das heißt, es wird der Rest berechnet, der sich bei der Teilung der 24-stelligen Zahl durch 97 ergibt. Das Ergebnis wird von der Zahl 98 subtrahiert. Ist diese Zahl kleiner als 10, so wird der

Zahl eine Null vorangestellt, sodass sich immer ein zweistelliger Wert für die Prüfsumme ergibt.

Beispiel 1:

```
Bankleitzahl:          70090100
Kontonummer:           1234567890
24-stellig:            700901001234567890131400
24-stellig Modulo 97:  90
Prüfsumme:             08 (98 - 90, ergänzt um führende Null)
IBAN:                  DE08700901001234567890
```

Beispiel 2:

```
Bankleitzahl:          29050000
Kontonummer:           234564
24-stellig:            290500000000234564131400
24-stellig Modulo 97:  86
Prüfsumme:             12 (98 - 86)
IBAN:                  DE12290500000000234564
```

Aufgabe 48:

Gegeben seien zwei natürliche Zahlen z1 und z2 zwischen 10 und 19. Dann lässt sich das Produkt dieser beiden Zahlen mit folgendem Algorithmus berechnen (Beispiel z1 = 13 und z2 = 17):

(1) zu z1 addiert man die letzte Ziffer von z2 (13 + 7 = 20)
(2) an das Ergebnis von (1) fügt man die 0 an (200)
(3) zu dem Ergebnis von (2) addiert man das Produkt der letzten Ziffern der beiden Zahlen (200 + 3*7 = 221)
(4) das Ergebnis von (3) ist das Produkt

Implementieren Sie eine int-Funktion *mult*, die das Produkt zweier als Parameter übergebener int-Werte gemäß dem oben beschriebenen Algorithmus berechnet und als Funktionswert liefert. In der Funktion können Sie davon ausgehen, dass als aktuelle Parameter korrekte Werte übergeben werden.

Randbedingung: Der einzige Datentyp, der zur Lösung dieser Aufgabe benutzt werden darf, ist `int`.

Aufgabe 49:

Implementieren Sie eine int-Funktion *getLetzteZiffern*, der ein int-Wert *z* und ein zweiter int-Wert *n* als Parameter übergeben werden und die als Rückgabewert die Zahl liefert, die sich aus den letzten *n* Stellen der Zahl *z* berechnet. Sie können davon ausgehen, das gilt: *z* > 0 und *n* > 0 und *n* <= AnzahlZiffern(*z*).

Beispiele:

```
getLetzteZiffern(12345, 3)   → 345
getLetzteZiffern(35724, 2)   → 24
getLetzteZiffern(7, 1)       → 7
```

Aufgabe 50:

Die Zahl 1089 ist eine besondere Zahl:

- Nimm irgendeine Ausgangszahl mit 3 unterschiedlichen Ziffern (abc).
- Ermittle die Spiegelzahl der Zahl durch das Umdrehen der Ziffern (cba)
- Ermittle die Differenz zwischen der Spiegelzahl und der Zahl (|abc – cba| = def)
- Addiere die Differenz und deren Spiegelzahl (def + fed).
- Das Ergebnis ist immer 1089.

Beispiel:

Ausgangszahl:	160
Spiegelzahl:	061
Differenz:	099
Spiegelzahl der Differenz:	990
Summe:	1089

Sie trauen dieser Aussage aber nicht und möchten sie durch Ausprobieren überprüfen.

Aufgabe (a):

Implementieren Sie eine Funktion

```
static boolean is1089(int ausgangszahl)
```

die ausgehend von der als Parameter übergeben Zahl den obigen Algorithmus anwendet und überprüft, ob das Ergebnis tatsächlich 1089 ist. Genau dann liefert die Funktion true.

Aufgabe (b):

Schreiben Sie ein Programm, das durch Aufrufe der Funktion *is1089* für alle Ausgangszahlen zwischen 0 und 999 die obige Aussage überprüft. Für die Ausgangszahlen x, für die die Aussage nicht gilt, soll auf die Konsole ausgegeben werden: „x ist keine 1089er Zahl“. Beispielsweise ist 101 keine 1089er Zahl (sie hat jedoch auch nicht 3 unterschiedliche Ziffern).

Aufgabe 51:

Das **Steinspiel** ist ein Spiel für *m* Spieler, die nacheinander an der Reihe sind und nicht aussetzen dürfen. Spieler 1 beginnt. Anfangs gibt es einen Haufen mit *n* Steinen. Ein Spielzug besteht darin, eine bestimmte Menge an Steinen vom Steinhaufen zu entfernen. Der Spieler, der den letzten Stein entfernt, gewinnt. Beim ersten Spielzug des ersten Spielers muss mindestens 1 Stein genommen werden, es

dürfen aber nicht alle Steine genommen werden. Bei allen weiteren Spielzügen muss mindestens 1 Stein genommen werden, aber es dürfen maximal doppelt so viele Steine genommen werden, wie im Spielzug des Spielers, der unmittelbar vorher an der Reihe war. Natürlich können/dürfen auch nicht mehr Steine genommen werden, wie noch auf dem Haufen liegen.

Implementieren Sie ein Java-Programm, mit dem menschliche Spieler gegeneinander an einer Konsole das Steinspiel spielen können. Anfangs sollen positive Zahlen *m* und *n* für die Anzahl an Spielern und die Anzahl an initialen Steinen von der Konsole eingelesen werden. Das Programm soll Benutzereingaben überprüfen und bei fehlerhaften Eingaben bzw. Spielzügen den Benutzer erneut zur Eingabe auffordern. Orientieren Sie sich, auch was die Ausgaben angeht, an folgendem Beispielablauf meiner Musterlösung (Benutzereingaben in Klammern <>):

```
Anzahl an Spielern (> 0): <3>
Anzahl an Steinen (> 0): <50>
Anzahl Steine auf Haufen = 50
Spieler 1 ist am Zug!
Anzahl zu entfernende Steine: <2>
Anzahl Steine auf Haufen = 48
Spieler 2 ist am Zug!
Anzahl zu entfernende Steine: <5>
Fehler! Anzahl zu entfernende Steine: <4>
Anzahl Steine auf Haufen = 44
Spieler 3 ist am Zug!
Anzahl zu entfernende Steine: <1>
Anzahl Steine auf Haufen = 43
Spieler 1 ist am Zug!
Anzahl zu entfernende Steine: <3>
Fehler! Anzahl zu entfernende Steine: <2>
Anzahl Steine auf Haufen = 41
Spieler 2 ist am Zug!
Anzahl zu entfernende Steine: <4>
Anzahl Steine auf Haufen = 37
Spieler 3 ist am Zug!
Anzahl zu entfernende Steine: <8>
Anzahl Steine auf Haufen = 29
Spieler 1 ist am Zug!
Anzahl zu entfernende Steine: <16>
Anzahl Steine auf Haufen = 13
Spieler 2 ist am Zug!
Anzahl zu entfernende Steine: <13>
Spielende! Spieler 2 hat gewonnen!
```

Aufgabe 52:

Implementieren Sie eine Funktion **`minimum`**, die drei int-Werte als Parameter übergeben bekommt, den kleinsten der drei Werte ermittelt und diesen als Funktionswert zurückliefert.

Implementieren Sie weiterhin ein kleines Testprogramm, das drei beliebige int-Werte von der Konsole einliest, dann die Funktion `minimum` mit den drei eingelesen Werten als Parameter aufruft und den durch die Funktion ermittelten kleinsten Wert auf die Konsole ausgibt.

Aufgabe 53:

Groker ist ein Spiel für zwei Spieler. Jeder Spieler hat einen Vorrat von unbegrenzt vielen Chips und einen Gewinnhaufen, der zu Beginn leer ist. Das Spiel wird in mehreren Runden gespielt. In jeder Runde setzen die Spieler jeweils für den Gegenspieler verdeckt eine beliebige Anzahl Chips – ihren Einsatz –, mindestens aber einen Chip. Aufgedeckt wird dann gleichzeitig. Unterscheidet sich die Zahl der gesetzten Chips um höchstens 10, dürfen beide Spieler ihren Einsatz komplett dem eigenen Gewinnhaufen hinzufügen. Ist die Differenz größer, darf lediglich derjenige mit dem kleineren Einsatz die Chips seinem Gewinnhaufen hinzufügen. Derjenige Spieler hat gewonnen, der in seinem Gewinnhaufen als erster mindestens 100 Chips mehr als sein Gegenspieler hat (aus Bundeswettbewerb Informatik).

Implementieren Sie ein Java-Programm, mit dem zwei menschliche Spieler gegeneinander an einer Konsole das Spiel *Groker* spielen können. Das Programm soll Benutzereingaben überprüfen und bei fehlerhaften Eingaben den Benutzer erneut zur Eingabe auffordern. Orientieren Sie sich, auch was die Ausgaben angeht, an folgendem Beispielablauf meiner Musterlösung (Benutzereingaben in Klammern <>):

```
Spieler 1! Bitte Anzahl eingeben (>0): <66>
Spieler 2! Bitte Anzahl eingeben (>0): <61>
Gewinnhaufen Spieler 1: 66
Gewinnhaufen Spieler 2: 61
Spieler 1! Bitte Anzahl eingeben (>0): <55>
Spieler 2! Bitte Anzahl eingeben (>0): <44>
Gewinnhaufen Spieler 1: 66
Gewinnhaufen Spieler 2: 105
Spieler 1! Bitte Anzahl eingeben (>0): <80>
Spieler 2! Bitte Anzahl eingeben (>0): <65>
Gewinnhaufen Spieler 1: 66
Gewinnhaufen Spieler 2: 170
Spielende! Spieler 2 hat gewonnen!
```

Aufgabe 54:

Zwei natürliche Zahlen a und b heißen befreundet, wenn die Teilersumme von a gleich b und die Teilersumme von b gleich a ist. Als Teilersumme einer Zahl x wird dabei bezeichnet die Summe der (von x verschiedenen) Teiler von x.

Ein Beispiel für ein solches befreundetes Zahlenpaar ist (a,b) = (220,284), denn a = 220 hat die Teiler 1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110 und es gilt

1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284 = b.

Weiterhin hat b = 284 die Teiler 1, 2, 4, 71, 142 und es gilt

1 + 2 + 4 + 71 + 142 = 220 = a.

Teilaufgabe 1: Implementieren Sie zunächst eine Funktion **`static int teilersumme(int zahl)`**, die von der ihr übergebenen Zahl die Teilersumme berechnet und zurückliefert. Die Funktion soll Seiteneffekt-frei sein, d.h. insbesondere keine Ausgaben auf die Konsole produzieren.

Teilaufgabe 2: Implementieren sie dann eine ebenfalls Seiteneffekt-freie Funktion **`static boolean befreundet(int zahl1, int zahl2)`**, die überprüft und liefert, ob die zwei als Parameter übergebenen Zahlen befreundet sind. Bei der Implementierung soll die in Teilaufgabe 1 definierte Funktion aufgerufen werden.

Teilaufgabe 3: Schreiben Sie ein Java-Programm, das in einer Schleife jeweils zwei natürliche Zahlen (> 0) einliest und entscheidet, ob diese miteinander befreundet sind. Dabei soll die in Teilaufgabe 2 definierte Funktion aufgerufen werden. Das Programm soll abbrechen, falls zwei nicht-befreundete Zahlen eingegeben wurden.

Der Programmablauf könnte in etwa wie folgt aussehen:

```
Erste Zahl: 220
Zweite Zahl: 284
Die beiden Zahlen sind miteinander befreundet!
Erste Zahl: 10744
Zweite Zahl: 10856
Die beiden Zahlen sind miteinander befreundet!
Erste Zahl: 23
Zweite Zahl: 22
Die beiden Zahlen sind nicht miteinander befreundet!
```

Aufgabe 55:

Der größte gemeinsame Teiler (ggT) zweier positiver ganzer Zahlen ist die größte Zahl, durch die sich die beiden Zahlen ohne Rest teilen lassen. Definieren und implementieren Sie eine Seiteneffekt-freie Funktion, die zwei positive int-Werte als Parameter übergeben bekommt und den ggT der beiden Werte berechnet und als Funktionswert liefert. Definieren Sie die Funktion dabei auf zwei verschiedene Art und Weisen.

Algorithmus (1):

Sei a die erste und b die zweite Zahl. Solange b ungleich 0 ist, wiederhole: a wird zum alten Wert von b und b wird zum Rest der ganzzahligen Division der alten Werte von a und b. Das Ergebnis ist dann der Wert von a.

Beispiel:

`a = 24, b = 18`

`a = 18, b =  6`

`a =  6, b =  0  → ggT = 6`

Algorithmus (2):

Implementieren Sie eine rekursive Funktion, die folgende mathematische Definition implementiert:

`ggT(m, m) = m`

`ggT(m, n) = ggT(m-n, n) falls m > n`

`ggT(m, n) = ggT(n, m)`

Beispiel:

`ggT(24,18) = ggT(6,18) = ggT(18,6) = ggT(12,6) = ggT(6,6) = 6`

Aufgabe 56:

Der größte gemeinsame Teiler (ggT) zweier positiver ganzer Zahlen ist die größte Zahl, durch die sich die beiden Zahlen ohne Rest teilen lassen. Definieren und

implementieren Sie eine Seiteneffekt-freie Funktion, die zwei positive int-Werte als Parameter übergeben bekommt und den ggT der beiden Werte berechnet und als Funktionswert liefert. Definieren Sie die Funktion dabei auf zwei verschiedene Art und Weisen.

Algorithmus (1):

Sei a die erste und b die zweite Zahl. Solange b ungleich 0 ist, wiederhole: Falls a größer als b ist, wird für den nächsten Schleifendurchlauf b von a abgezogen, ansonsten wird a von b abgezogen. Das Ergebnis ist dann der Wert von a.

Beispiel:

```
a = 24, b = 18
a =  6, b = 18
a =  6, b = 12
a =  6, b =  6
a =  6, b =  0  → ggT = 6
```

Algorithmus (2):

Definieren Sie eine rekursive Funktion, die folgende mathematische Definition implementiert:

```
ggT(m, 0) = m
ggT(m, m) = m
ggT(m, n) = ggT(n, m%n)
```

Beispiel:

```
ggT(18,24) = ggT(24,18) = ggT(18,6) = ggT(6,0) = 6
```

Aufgabe 57:

Trinken bzw. programmieren wir zwischendurch doch mal ein Gläschen Sekt. Schreiben Sie dazu ein Programm, das den Benutzer zunächst auf der Konsole zur Eingabe eines **`int`**-Wertes **`hoehe`** größer als 2 auffordert und abhängig von dem Wert von **`hoehe`** anschließend ein ASCII-Glas der folgenden Form auf der Konsole ausgibt. Setzen Sie bitte das Prinzip der prozeduralen Zerlegung ein und vermeiden Sie globale Variablen.

Beispiel: hoehe = 3:

Beispiel: hoehe = 4:

Beispiel: hoehe = 5:

Aufgabe 58:

Das Gewichtheben ist eine schwerathletische Sportart, bei der eine Langhantel durch Reißen oder Stoßen zur Hochstrecke gebracht wird, das heißt mit ausgestreckten Armen über den Kopf gestemmt wird. In dieser Aufgabe geht es um das Zeichnen derartiger Langhanteln. Schreiben Sie dazu ein Programm, das den Benutzer zunächst auf der Konsole zur Eingabe eines **`int`**-Wertes **`groesse`** (>= 3 und

ungerade) auffordert und abhängig von dem Wert von **`groesse`** anschließend eine ASCII-Langhantel der folgenden Form auf der Konsole ausgibt.

Beispiel: groesse = 3:

```
 |   |
||---||
 |   |
```

Beispiel: groesse = 5:

```
  |     |
 ||     ||
|||-----|||
 ||     ||
  |     |
```

Beispiel: groesse = 7:

```
   |       |
  ||       ||
 |||       |||
||||-------||||
 |||       |||
  ||       ||
   |       |
```

Aufgabe 59:

Eine natürliche Zahl n wird vollkommene Zahl (oder auch perfekte Zahl) genannt, wenn sie gleich der Summe aller ihrer positiven Teiler außer sich selbst ist.

Beispiel 28: Die positiven Teiler von 28 sind 1, 2, 4, 7, 14, 28 und es gilt: 1+2+4+7+14=28. Weitere vollkommene Zahlen sind 6 und 496.

Teilaufgabe 1:

Implementieren Sie zunächst eine Funktion **`static int teilersumme(int zahl)`**, die von der ihr übergebenen Zahl die Teilersumme, d.h. die Summe aller ihrer positiven Teiler, berechnet und zurückliefert. Die Funktion soll Seiteneffekt-frei sein, d.h. insbesondere keine Ausgaben auf die Konsole produzieren.

Teilaufgabe 2:

Implementieren sie dann eine ebenfalls Seiteneffekt-freie Funktion **`static boolean vollkommeneZahl(int zahl)`**, die überprüft und liefert, ob die als Parameter übergebene Zahle vollkommen ist. Nutzen Sie dabei die in Teilaufgabe 1 definierte Funktion.

Teilaufgabe 3:

Schreiben Sie ein Java-Programm, das in einer Schleife jeweils eine natürliche Zahl (> 0) einliest und entscheidet, ob diese vollkommen ist. Dabei soll die in Teilaufgabe 2 definierte Funktion aufgerufen werden. Das Programm soll abbrechen, falls eine nicht-vollkommene Zahl eingegeben wurde.

Der Programmablauf könnte in etwa wie folgt aussehen (Eingaben in <>):

```
Zahl (> 0): <28>
Die Zahl 28 ist vollkommen!
Zahl (> 0): <496>
Die Zahl 496 ist vollkommen!
Zahl (> 0): <234>
Die Zahl 234 ist nicht vollkommen!
```

Aufgabe 60:

Um festzustellen, ob eine natürliche Zahl *n* (> 0) „fröhlich" ist, addiert man die Quadrate ihrer Ziffern. Mit der Summe verfährt man wieder genauso und so weiter. Trifft man dabei irgendwann auf die 1, dann war die ursprüngliche Startzahl „fröhlich". Trifft man dagegen auf eine 4, so war sie „traurig". Der Algorithmus endet übrigens immer bei 1 oder 4.

Die 7 ist bspw. eine fröhliche Zahl, denn

$7^2 = 49$
$4^2 + 9^2 = 16 + 81 = 97$
$9^2 + 7^2 = 81 + 49 = 130$
$1^2 + 3^2 + 0^2 = 1 + 9 + 0 = 10$
$1^2 + 0^2 = 1$

Teilaufgabe 1:

Implementieren Sie zunächst eine Funktion **`static boolean istFroehlich(int zahl)`**, die überprüft, ob die als Parameter übergebene Zahl fröhlich ist. Die Funktion soll Seiteneffekt-frei sein, d.h. insbesondere keine Ausgaben auf die Konsole produzieren.

Teilaufgabe 2:

Schreiben Sie ein Java-Programm, das in einer Schleife jeweils eine natürliche Zahl (> 0) einliest und entscheidet und auf die Konsole ausgibt, ob diese fröhlich ist. Dabei

soll die in Teilaufgabe 1 definierte Funktion aufgerufen werden. Das Programm soll abbrechen, falls eine traurige Zahl eingegeben wurde.

Der Programmablauf könnte in etwa wie folgt aussehen (Eingaben in <>):

```
Zahl eingeben (> 0): <7>
Die Zahl 7 ist froehlich
Zahl eingeben (> 0): <19>
Die Zahl 19 ist froehlich
Zahl eingeben (> 0): <37>
Die Zahl 37 ist traurig
```

Aufgabe 61:

Eine natürliche Zahl heißt Armstrong-Zahl, wenn die Summe ihrer Ziffern, jeweils potenziert mit der Stellenanzahl der Zahl, wieder die Zahl selbst ergibt.

Ein Beispiel für eine solche Zahl ist die fünfstellige Zahl 54748:

$54748 = 5^5 + 4^5 + 7^5 + 4^5 + 8^5 = 3125 + 1024 + 16807 + 1024 + 32768 = 54748$

Teilaufgabe 1:

Implementieren Sie zunächst eine Funktion

```
static boolean isArmstrongNumber(int number)
```

Die Funktion soll überprüfen, ob die als Parameter übergebene Zahl eine Armstrong-Zahl ist und entsprechend *true* oder *false* zurückliefern.

Achtung:

- Die Funktion soll Seiteneffekt-frei sein, d.h. insbesondere keine Ausgaben auf die Konsole produzieren.
- Sie dürfen keine Funktionen der JDK-Klassenbibliothek benutzen. Definieren Sie für Teilberechnungen ggfls. selbst entsprechende Hilfsfunktionen.
- Sie können davon ausgehen, dass der als Parameter übergebene Wert größere als 0 ist. Um mögliche Zahlenbereichüberläufe müssen Sie sich nicht kümmern.

Teilaufgabe 2:

Schreiben Sie ein Java-Programm, das durch Aufruf der Funktion *isArmstrongNumber* aus Teilaufgabe 1 alle Armstrong-Zahlen zwischen 1 und 100000 berechnet und auf die Konsole ausgibt.

Aufgabe 62:

Schreiben Sie dazu ein Programm, das den Benutzer zunächst auf der Konsole zur Eingabe eines positiven **`int`**-Wertes **`groesse`** auffordert und abhängig von dem Wert von **`groesse`** eine Zeichnung der folgenden Form auf der Konsole ausgibt.

Beispiel: groesse = 3:

```
+++
+++
+++
---
+++
+++
+++
```

Beispiel: groesse = 5:

```
+++++
+++++
+++++
+++++
+++++
-----
+++++
+++++
+++++
+++++
+++++
```