

## Übungen zu Kapitel 7

### 7.1 Switch:

In einer ausführbaren Klasse soll innerhalb der `main()`-Methode in einer `int`-Variablen `test` eine Zahl zwischen 0 und 9 gespeichert werden. Mittels `switch`-Mehrfachverzweigung soll der Name der eingegebenen Zahl *in Worten* ausgegeben werden – d.h. wenn `test` den Wert 8 trägt, soll "Der Wert beträgt acht!" ausgegeben werden. Durch den `default`-Zweig sollen alle anderen Werte behandelt werden.

Was passiert, wenn Sie `break`-Anweisungen vergessen (oder testweise entfernen) ?

Verändern oder ergänzen Sie das Programm:

- Wenn `test` den Wert 0, 1, 2 oder 3 trägt, dann geben Sie aus "Zahl im Bereich Null bis Drei".
- Wenn `test` den Wert 4, 5 oder 7 trägt, dann geben Sie aus "Zahl im Bereich Vier bis Sieben".
- Wenn `test` den Wert 8 oder 9 trägt, dann geben Sie aus "Zahl im Bereich Acht bis Neuen".

Wieder soll ein `default`-Zweig vorhanden sein.

### 7.2 Harmonische Reihe:

Berechnen Sie in der `main()`-Methode die *harmonische Reihe*, d.h. die Summe  $\sum (1/n)$  - und zwar für  $n = 1$  bis 10000. Benutzen Sie `for`-Schleifen.

Führen Sie die Berechnung zuerst mit `float`-Variablen und Werten durch. Berechnen Sie den Wert der Reihe `sum` zweimal (in zwei separaten `for`-Schleifen):

a) von "unten nach oben":  $1/1 + 1/2 + 1/3 + \dots + 1/9999 + 1/10000$

b) von "oben nach unten":  $1/10000 + 1/9999 + 1/9998 + \dots + 1/2 + 1/1$

Geben Sie die Ergebnisse aus und vergleichen Sie die Werte!

Wiederholen Sie die Berechnungen mit `double`-Variablen! Was stellen Sie fest in Bezug auf die Genauigkeit der Resultate ?

Tip: Achten Sie in der Schleife darauf, dass Sie bei Berechnung der Brüche (Divisionen) wirklich jeweils Ergebnisse von Typ `float` bzw `double` erhalten!

### 7.3 PI-Näherung:

Implementieren Sie einen Algorithmus zur (groben) statistischen Näherung der Kreiszahl Pi (Monte-Carlo-Verfahren). Es soll mit `double`-Genauigkeit gerechnet werden. Folgende drei Schritte werden  $n$ -mal wiederholt ( $n$  muss eine relativ große Zahl sein, zB 1000000):

1. Mittels des Methodenaufrufs `Math.random()` (Paket `java.lang`) werden `double`-Zufallszahlen  $x$  und  $y$  zwischen 0 und 1 gezogen. Die Anweisungen lauten:  $x = \text{Math.random}()$  und  $y = \text{Math.random}()$ . Ein aufeinander folgendes Paar solcher Zahlen wird als Koordinaten eines Punktes in der  $(xy)$ -Ebene interpretiert.
2. Man berechnet jeweils den `double`-Wert  $\text{dist} = x*x + y*y$ , d.h. das Abstandsquadrat des Punktes vom Ursprung.
3. Wenn  $\text{dist} \leq 1.0$ , so liegt der Punkt innerhalb oder auf einem Viertelkreis mit Radius 1. In diesem Fall wird ein `int`-Zähler `count` um eins hochgezählt.

Nach  $n$ -maliger Wiederholung dieser Schritte berechnet man eine Näherung für Pi zu  $Pi \approx 4 * \text{count} / n$ .

Geben Sie das Ergebnis aus und vergleichen Sie es mit dem genaueren Wert, der als Konstante `Math.PI` zur Verfügung steht.

Die Idee des Algorithmus: Die Fläche eines Viertelkreises mit Radius 1 beträgt  $\pi/4$ . Die Fläche eines den Viertelkreis umgebenden Quadrats mit Kantenlänge 1 beträgt 1. Das Verhältnis der Flächen Viertelkreis/Quadrat beträgt  $\pi/4$ . Die Verteilung der "Treffer" in Viertelkreis und Quadrat verhält sich für große Zahlen wie die Flächen.

Anmerkung: Der Aufruf `Math.random()` ist problemlos möglich, da die Klasse `Math` zum automatisch importierten Paket `java.lang` gehört, das in jedem Java-Programm zur Verfügung steht.

#### 7.4 Eine Nacht in Monte Carlo:

Mit der Anweisung `k=(int)(Math.random()*37)` kann man sich eine ganzzahlige Integer-Zufallszahl `k` im Bereich `[0....36]` verschaffen. Man interpretiere `k` als das Ergebnis eines Roulette-Wurfs und simuliere folgendes Spiel:

Sie begeben sich mit 10000 GE Kapital in das Spielkasino und setzen 10 GE auf Pair. Sie erhalten nach dem nächsten Spiel 20 GE ausgezahlt, falls `k` gerade und ungleich Null ist. Der Gewinn ist stets der doppelte Einsatz im Erfolgsfall. Falls `k` Null ist, bleibt Ihr Einsatz stehen. Falls `k` ungerade ist, verlieren Sie Ihren Einsatz.

Sie machen `n` Spiele mit folgender Strategie (Schleife!):

- Wenn Sie noch im Besitz von mindestens 10000 GE sind und das letzte Spiel gewonnen haben, setzen Sie 10 GE.
- Wenn Sie noch im Besitz von mindestens 10000 GE sind aber das letzte Spiel verloren haben, verdoppeln Sie Ihren Einsatz.
- Wenn Sie nur noch weniger als 10000 GE haben, setzen Sie das Doppelte Ihres Gesamtverlustes.
- Falls Ihr Restkapital für den vorgesehenen Einsatz nicht mehr ausreicht, setzen Sie Ihr gesamtes Restkapital.
- Falls Sie den erlaubten Höchsteinsatz von 1000 GE überschreiten würden, setzen Sie 1000 GE.

Geben Sie für jedes Spiel folgendes aus: Die Nummer des Spiels, die geworfene Zahl, das Kapital vorher, den Einsatz, das Kapital nach dem Wurf.

#### 7.5 Vollkommene Zahlen:

Das Problem stammt aus dem Bereich der Mathematik und zeigt den Gebrauch verschachtelter Schleifen: Ermitteln Sie alle *vollkommenen Zahlen* zwischen 1 und 1000.

Eine natürliche Zahl `n` heißt vollkommen, wenn die Summe aller ihrer Teiler (1 und `n` eingeschlossen) gleich  $2*n$  ist. Ein Beispiel: Die Zahl 6 ist vollkommen, denn  $1 + 2 + 3 + 6 = 2 * 6$ .

Für jede Zahl zwischen 1 und 100 müssen ihre Teiler bestimmt und aufaddiert werden - es ist zu prüfen, ob dabei das Doppelte der untersuchten Zahl herauskommt. Mittels Modulo-Operator lässt sich testen, ob eine Division glatt (mit Teilerrest Null) aufgeht, d.h. ob eine Ganzzahl Teiler einer anderen ist.

Die äußere Schleife läuft dabei über alle zu untersuchenden Zahlen, die innere Schleife testet alle möglichen Teiler und summiert die ganzzahligen Teiler auf.

#### 7.6 Leere Anweisung:

Reproduzieren Sie folgenden verbreiteten Anfängerfehler in einem ausführbaren Programm:

```
int k = 10;
if( k > 10 ); // Semikolon??
    System.out.println( "Größer als 10!" );
for( int i = 0; i <= k; i++); { // Semikolon??
    System.out.println( "Quadrat von " + i + " = " + i*i );
}
```

Hintergrund: Der Compiler interpretiert die Semikolons als *leere* Anweisungen, die die `if`- bzw. `for`-Anweisung abschließen. Eine leere Anweisung ist eine syntaktisch *vollwertige* Anweisung, die jedoch nichts tut. Somit würden die darauf folgenden Anweisungen nicht mehr zur Verzweigung bzw. Schleife gehören und unbedingt bzw. nur einmal ausgeführt werden.

Einfache Anweisungen werden in Java stets direkt mit einem Semikolon abgeschlossen. Dagegen enden alle Verzweigungen und abweisende Schleifen erst mit dem zugehörigen abzuarbeitenden Anweisungsblock.