

Lösungen: Einführung in die Programmierung, 22.07.2019

Ausgangsdatei: 12_EidP_Klausur_22.07.2019.pdf

Die Lösungen decken alle Aufgaben 1 bis 8 einschließlich aller Teilaufgaben ab. Öffentliche Java-Klassen gehören beim Übersetzen in jeweils eigene Dateien. Die auf dem Deckblatt erwähnten Bonuspunkte bilden keine zusätzliche Aufgabe.

Aufgabe 1: Multiple Choice, Schleifen und Zahlensysteme

a) Zeichenketten und Paketsichtbarkeit

Anweisung	Liefert „154 ist nicht 15+4“?
<code>System.out.print(15+4+" ist nicht "+15+4);</code>	Falsch: „19 ist nicht 154“. Zuerst werden 15 und 4 addiert, spätere Zahlen werden an den String angehängt.
<code>System.out.print("" + 15 + 4 + " ist nicht 15+4");</code>	Richtig: Der führende leere String löst sofort die Zeichenkettenverkettung aus.
<code>System.out.print(15+4+" ist nicht 15+4");</code>	Falsch: „19 ist nicht 15+4“.
<code>System.out.print(15+(4+" ist nicht 15+4"));</code>	Richtig: Der Klammerausdruck ist bereits ein String, an den die 15 vorne angefügt wird.

Aussage zum Objektattribut mit Paketsichtbarkeit	Bewertung und Begründung
Wird in UML unterstrichen.	Falsch. Unterstreichungen kennzeichnen ein Klassenattribut.
Wird in UML mit ~ gekennzeichnet.	Richtig. ~ steht für Paketsichtbarkeit.
Wird in Java durch static gekennzeichnet.	Falsch. static würde ein Klassenattribut bezeichnen.
Wird in Java durch package gekennzeichnet.	Falsch. Paketsichtbarkeit entsteht durch Weglassen von public, protected und private.

b) Endwerte der Variablen

Anweisungsfolge	x	y	Begründung
for (x=0; x<10; x+=2) y--;	10	-5	Die fünf Durchläufe benutzen x = 0, 2, 4, 6, 8. Danach wird x noch auf 10 erhöht.
for (x=0; x<100; x+=2) y++;	100	50	Es gibt 50 Durchläufe für x = 0 bis 98 in Zweierschritten.
x=40; while(x>0) { x/=2; y+=2; }	0	12	x nimmt nach den sechs Divisionen 20, 10, 5, 2, 1 und 0 an.
x=30; y=11; do { x--; } while(x-- > y++);	16	18	Der Rumpf verringert x einmal; der anschließende Test verringert x erneut und erhöht y, auch beim letzten, falschen Test.

Spur der letzten Schleife:

Durchlauf	x nach dem Rumpf	Vergleich mit alten Werten	x nach Test	y nach Test
1	29	29 > 11: true	28	12
2	27	27 > 12: true	26	13
3	25	25 > 13: true	24	14
4	23	23 > 14: true	22	15
5	21	21 > 15: true	20	16
6	19	19 > 16: true	18	17
7	17	17 > 17: false	16	18

c) Hexadezimalsumme

$1F_{16} = 31$ und $AA_{16} = 170$; die Summe ist 201. Stellenweise gilt $F + A = 15 + 10 = 25 = 1 \cdot 16 + 9$. Es bleibt 9, mit Übertrag 1 zur nächsten Stelle. Dort ergeben $1 + A + 1 = 12 = C$.

```

  1F
+ AA
----
  C9

```

$C9_{16} = 1100\ 1001_2$

Aufgabe 2: Strings nach gerader und ungerader Länge

```
import java.util.Scanner;

public class GeradeWorte {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        String[] worte = new String[50];
        for (int i = 0; i < worte.length; i++) {
            worte[i] = eingabe.next();
        }
        for (int i = 0; i < worte.length; i++) {
            if (worte[i].length() % 2 == 0) {
                System.out.println(worte[i]);
            }
        }
        for (int i = 0; i < worte.length; i++) {
            if (worte[i].length() % 2 != 0) {
                System.out.println(worte[i]);
            }
        }
    }
}
```

Erst werden alle 50 Strings gespeichert. Die erste Ausgabeschleife gibt die geraden, die zweite die ungeraden Längen aus. Innerhalb jeder Gruppe bleibt die Eingabereihenfolge erhalten. Jede Ausgabe verwendet `println` und steht somit in einer eigenen Zeile.

Die Eingabe wird hier passend zum Klassennamen als 50 durch Leerraum getrennte Wörter verstanden. Soll jeder String eine vollständige Eingabezeile einschließlich möglicher Leerzeichen sein, wird ausschließlich `next()` durch `nextLine()` ersetzt; die restliche Lösung bleibt unverändert. `length()` wird wie ausdrücklich vorgegeben als Maß verwendet.

Aufgabe 3: Dreierfaktoren

a) Maximalen Exponenten bestimmen

```
public static int faktoren3(int n) {
    assert n != 0;
    int t = 0;
    while (n % 3 == 0) {
        n /= 3;
        t++;
    }
    return t;
}
```

Jede erfolgreiche Division entfernt genau einen Faktor 3. Sobald ein Rest auftritt, gibt es keinen weiteren solchen Faktor. Für 99 verläuft die Division $99 \rightarrow 33 \rightarrow 11$; deshalb ist $t = 2$. Für eine nicht durch 3 teilbare Zahl ist $t = 0$, denn $3^0 = 1$ teilt jede ganze Zahl. Die Lösung funktioniert auch für negative n , etwa $-99 \rightarrow -33 \rightarrow -11$. $n = 0$ muss ausgeschlossen sein, weil 0 durch jede Potenz von 3 teilbar ist und es keinen maximalen Exponenten gäbe.

b) Maximum unter den geraden Exponenten

```
public static int maximum(int n) {
    assert n > 0;
    int maximum = 0;
    for (long zahl = 1; zahl <= n; zahl++) {
        int t = faktoren3((int) zahl);
        if (t % 2 == 0 && t > maximum) {
            maximum = t;
        }
    }
    return maximum;
}
```

Es werden alle faktoren3-Werte für 1 bis n geprüft, und nur gerade Ergebnisse können das Maximum ersetzen. Der Startwert 0 ist gültig, weil $\text{faktoren3}(1) = 0$ vorkommt. Der long-Schleifenzähler verhindert, dass $\text{zahl}++$ bei $n = \text{Integer.MAX_VALUE}$ überläuft; jeder tatsächlich übergebene Wert liegt trotzdem im int-Bereich.

Beispiele: $\text{maximum}(8) = 0$, $\text{maximum}(9) = 2$, $\text{maximum}(80) = 2$ und $\text{maximum}(81) = 4$. Eine gleichwertige schnellere Lösung nutzt, dass ein gerader Exponent $2k$ genau dann vorkommt, wenn $9^k \leq n$:

```
public static int maximumSchnell(int n) {
    assert n > 0;
    int t = 0;
    while (n >= 9) {
        n /= 9;
        t += 2;
    }
    return t;
}
```

Diese zweite Methode erläutert eine Alternative; für die verlangte Lösung genügt die erste Methode `maximum`.

Aufgabe 4: Arrays

a) Teilsummen im vorhandenen Array

```
public static void teilsummen(int[] feld) {
    assert feld != null;
    for (int i = 1; i < feld.length; i++) {
        feld[i] += feld[i - 1];
    }
}
```

Vor jedem Durchlauf enthält `feld[i - 1]` bereits die Summe aller ursprünglichen Werte bis `i - 1`. Durch Addition des noch unveränderten Wertes `feld[i]` entsteht die Summe bis `i`. Die Laufrichtung von links nach rechts ist dafür entscheidend. Ein neues Array wird nicht erzeugt. Das Beispiel wird zu `{1, 8, 21, 16, 18, 45, 56, 60, 111}`. Leere und einelementige Arrays benötigen keine Änderung. Wie in der Aufgabe wird angenommen, dass die gewünschten Summen in `int` darstellbar sind.

b) Gittermuster

```
public static char[][] gitter(int n) {
    assert n > 0;
    char[][] feld = new char[2 * n + 1][2 * n + 1];
    for (int z = 0; z < feld.length; z++) {
        for (int s = 0; s < feld[z].length; s++) {
            if (z % 2 == 1 || s % 2 == 1) {
                feld[z][s] = '*';
            } else {
                feld[z][s] = ' ';
            }
        }
    }
    return feld;
}
```

„Jede zweite“ beginnt im dargestellten Beispiel mit der zweiten Zeile beziehungsweise Spalte, also mit dem nullbasierten Index 1. Ein Stern steht deshalb dort, wo mindestens einer der Indizes ungerade ist. Die übrigen Positionen erhalten ausdrücklich ein Leerzeichen. Für `n = 1` und `n = 2` ergeben sich:

```
*
***
*

* *
*****
* *
*****
* *
```

Die Dimension $2n + 1$ muss wie üblich in den `int`-Indexbereich und in den verfügbaren Speicher passen.

Aufgabe 5: Fahrzeugklassen

EBus

```
public class EBus extends Fahrzeug {
    private static int anzahl = 0;
    private boolean erster;

    public EBus(int anzPersonen) {
        super(anzPersonen, null);
        erster = (++anzahl == 1);
    }

    @Override
    public double berechneVerbrauchPro100Km() {
        return 0.0;
    }

    @Override
    protected double kmProLiter() {
        return 0.0;
    }

    @Override
    public String toString() {
        return (erster ? "Ein EBus " : "Noch ein EBus ")
            + super.toString();
    }
}
```

Die Klasse verwendet genau zwei Attribute: einen gemeinsamen Zähler und die pro Objekt gespeicherte Kennzeichnung des ersten Busses. Es genügt nicht, in `toString` nur den aktuellen Zähler abzufragen, da bei der Ausgabe bereits alle drei Busse existieren. Die Treibstoffart ist die null-Referenz; die Stringverkettung der Oberklasse stellt sie als "null" dar.

`kmProLiter` muss überschrieben werden. Die geerbte Berechnung $100/0.0$ würde bei double den Wert Infinity liefern, nicht die verlangte 0.0. Die Rückgabe 0.0 setzt die spezielle Ausgabekonvention der Aufgabe um; für einen Elektrobus ist die Größe Kilometer pro Liter ohnehin kein sinnvoller physikalischer Vergleich.

Pkw

```
public class Pkw extends Fahrzeug {
    public Pkw(int anzPersonen) {
        super(anzPersonen, "Benzin");
    }

    @Override
    public double berechneVerbrauchPro100Km() {
        return 5.0;
    }

    @Override
    public String toString() {
        return "Der Pkw " + super.toString();
    }
}
```

Die Personenzahl stammt aus dem Konstruktor. Weitere Attribute sind nicht nötig; 100/5.0 liefert 20.0 Kilometer pro Liter.

Lkw

```
public class Lkw extends Fahrzeug {
    private static int anzahl = 0;
    private int nummer;

    public Lkw() {
        super(2, "Diesel");
        nummer = ++anzahl;
    }

    @Override
    public double berechneVerbrauchPro100Km() {
        return 25.0;
    }

    @Override
    public String toString() {
        return "Lkw " + nummer + " " + super.toString();
    }
}
```

Auch hier gibt es genau zwei Attribute. Die individuelle Nummer bleibt erhalten, während der gemeinsame Zähler steigt. 100/25.0 ergibt 4.0. Mit der vorgegebenen main-Methode entsteht exakt:

```
Lkw 1 tankt Diesel für 2 Personen, 4.0
Lkw 2 tankt Diesel für 2 Personen, 4.0
Ein EBus tankt null für 41 Personen, 0.0
Noch ein EBus tankt null für 39 Personen, 0.0
Noch ein EBus tankt null für 41 Personen, 0.0
Der Pkw tankt Benzin für 5 Personen, 20.0
```

Aufgabe 6: Fehlersuche

Laut Diagramm erweitert B die konkrete Klasse A und implementiert I. C erweitert B und erbt damit dessen Implementierung von I. A kennt nur `methode1`; B ergänzt `methode2` und `methodel`; I deklariert ausschließlich `methodel`.

Zeile	Fehler?	Begründung
<code>einA.methode1();</code>	Nein	In A deklarierte Methode.
<code>einA.methodel();</code>	X	Der statische Typ A besitzt diese Methode nicht.
<code>einB.methode1();</code>	Nein	Von A geerbt.
<code>einB.methode2();</code>	Nein	In B deklariert.
<code>einB.methodel();</code>	Nein	B implementiert diese Interface-Methode.
<code>einI.methode2();</code>	X	Das Interface I deklariert <code>methode2</code> nicht.
<code>einI.methodel();</code>	Nein	Die Methode ist in I deklariert.
<code>B einB;</code>	Nein	Zulässige Variablendeklaration.
<code>einB = new A();</code>	X	Ein A-Objekt ist kein B-Objekt; die Zuweisung von Oberklasse zu Unterklasse ist unzulässig.
<code>einB = new C();</code>	Nein	C ist eine Unterklasse von B.
<code>I einI;</code>	Nein	Interface-Referenzen dürfen deklariert werden.
<code>einI = new A();</code>	X	A implementiert I nicht.
<code>einI = new C();</code>	Nein	C ist über B ebenfalls ein I.
<code>einI = new I();</code>	X	Interfaces können nicht unmittelbar instanziiert werden.

Alle übrigen Methodenköpfe, Klassendeklarationen und Klammern sind korrekt. Die Zeilen werden einzeln auf Übersetzbarkeit beurteilt.

Aufgabe 7: MediaPlayer und Medium

Abstrakte Klasse Medium

```
public abstract class Medium {  
    private String name;  
  
    public Medium(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public abstract int berechneSpeicherverbrauch();  
}
```

Der Name gehört zum Objekt. Der Speicherverbrauch wird durch Audio beziehungsweise Video bestimmt; daher bleibt die Methode in Medium abstrakt. Die Unterklassen sind nicht Teil der verlangten Implementierung.

Klasse MediaPlayer

```
public class MediaPlayer {
    private int freierSpeicher;
    private Medium[] medien = new Medium[1000];
    private int anzahlMedien = 0;

    public MediaPlayer(int speicherkapazität) {
        freierSpeicher = speicherkapazität;
    }

    public int getFreierSpeicher() {
        return freierSpeicher;
    }

    public int getAnzahlMedien() {
        return anzahlMedien;
    }

    public int addMedium(Medium medium) {
        if (medium == null || anzahlMedien == medien.length) {
            return -1;
        }
        int verbrauch = medium.berechneSpeicherverbrauch();
        if (verbrauch > freierSpeicher) {
            return -1;
        }
        for (int i = 0; i < medien.length; i++) {
            if (medien[i] == null) {
                medien[i] = medium;
                anzahlMedien++;
                freierSpeicher -= verbrauch;
                return i;
            }
        }
        return -1;
    }

    public Medium removeMedium(String name) {
        for (int i = 0; i < medien.length; i++) {
            if (medien[i] != null) {
                String gespeicherterName = medien[i].getName();
                boolean gleich = name == null
                    ? gespeicherterName == null
                    : name.equals(gespeicherterName);
                if (gleich) {
                    Medium entfernt = medien[i];
                    medien[i] = null;
                    anzahlMedien--;
                    freierSpeicher +=
                        entfernt.berechneSpeicherverbrauch();
                    return entfernt;
                }
            }
        }
        return null;
    }
}
```

Das Array und der Zähler sind zusätzliche Attribute ausschließlich zur Verwaltung der Assoziation. Es werden höchstens 1000 Medien aufgenommen. Eine freie Position allein reicht nicht: Auch der

freie Speicher muss den Bedarf decken; genau passender Speicher ist zulässig.

Die Suche nach dem zu löschenden Namen läuft aufsteigend und endet beim ersten Treffer, also beim kleinsten Index. String-Inhalte werden mit equals verglichen; == würde nur Referenzidentität prüfen. Die zusätzliche Nullbehandlung erlaubt auch null als Namen. Der freigegebene Speicher wird zurückgebucht, der Zähler vermindert und die entfernte Referenz zurückgegeben. Lücken sind erlaubt, da die Aufgabe kein Zusammenrücken fordert; addMedium nutzt die erste freie Lücke wieder.

Vorausgesetzt werden eine nichtnegative Kapazität, nichtnegative Speicherverbräuche und ein während der Speicherung unveränderter Verbrauch eines Mediums. Diese übliche Modellannahme ist nötig, um den beim Entfernen wieder gutzuschreibenden Verbrauch allein aus den vorgegebenen Methoden zu bestimmen.

Aufgabe 8: Speicherabbild

Benenne das A-Objekt mit O1 und das B-Objekt mit O2. Der Aufruf `a.gleich(b)` läuft auf O1; sein Parameter `a` verweist auf O2. Direkt vor `return` gilt:

Speicherbereich	Inhalt
Klassenattribut <code>A.c</code>	3
O1, Typ A	<code>A.m = 3; A.n = 7</code>
O2, Typ B	<code>A.m = 7; A.n = 3; B.m = 10</code>
main: <code>a; b</code>	<code>a → O1; b → O2</code>
main: <code>erg</code>	Noch nicht initialisiert; der Methodenaufruf ist noch nicht beendet.
main: <code>args</code>	Referenz auf das übergebene String-Array, dessen Inhalt nicht angegeben ist.
<code>A.gleich: this</code>	Referenz auf O1
<code>A.gleich: Parameter a</code>	Referenz auf O2
<code>A.gleich: erg</code>	false

`A(3, 7)` setzt beim ersten Objekt `m = 3` und `n = 7`. `B(3, 7)` ruft `super(n, m)`, also `A(7, 3)`, auf; zusätzlich setzt B sein eigenes `m` auf `3 + 7 = 10`. Nur die beiden A-Konstruktoren erhöhen `c`, sodass `1 → 2 → 3` entsteht.

In `gleich` sind die unqualifizierten `m` und `n` die Felder des Empfängerobjekts O1. Die Getter des Parameters lesen dagegen den A-Anteil von O2. Daher wird `(3 == 7) & (7 == 3)` berechnet, also false. Der boolesche Operator `&` wertet beide Seiten aus; hier ändert dies das Ergebnis nicht. Nach der Rückkehr wird auch `main.erg` zu false, und das Programm gibt false aus.