

Lösungen: Einführung in die Programmierung, 19.02.2019

Ausgangsdatei: 11_EidP_Klausur_19.02.2019.pdf

Alle Aufgaben 1 bis 8 und ihre Teilaufgaben werden behandelt. Öffentliche Java-Klassen werden beim Übersetzen in eigene, entsprechend benannte Dateien geschrieben. Die Bonuspunkte auf dem Deckblatt enthalten keine weitere Aufgabenstellung.

Aufgabe 1: Wissen und Zahlendarstellungen

a) Multiple Choice

Anweisung	Liefert Hello154? Begründung
<code>System.out.print("Hello"+15+4);</code>	Richtig. Die Verkettung läuft von links nach rechts: "Hello15" + 4.
<code>System.out.print("Hello"+150+4);</code>	Falsch. Ausgabe: Hello1504.
<code>System.out.print("Hello"+(15+4));</code>	Falsch. Die Klammer erzwingt die Addition; Ausgabe: Hello19.
<code>System.out.print(("Hello"+15)+4);</code>	Richtig. Ausgabe: Hello154.

Aussage zum Klassenattribut mit Paketsichtbarkeit	Bewertung und Begründung
Wird in UML unterstrichen.	Richtig. Unterstreichug bedeutet Klassenattribut.
Wird in UML mit ~ gekennzeichnet.	Richtig. ~ bezeichnet die Paketsichtbarkeit.
Wird in Java durch static gekennzeichnet.	Richtig. static kennzeichnet die Zugehörigkeit zur Klasse.
Wird in Java durch package gekennzeichnet.	Falsch. Ein Sichtbarkeitsmodifizierer wird weggelassen, etwa <code>static int anzahl;</code>

Aussage über Klasse A	Bewertung und Begründung
Wenn A eine abstrakte Methode enthält, muss A abstrakt sein.	Richtig. Eine nicht vollständig implementierte Klasse darf nicht konkret sein.
Wenn A abstrakt ist, muss A eine abstrakte Methode enthalten.	Falsch. <code>abstract</code> kann auch eine vollständig implementierte Klasse nicht instanzierbar machen.
Wenn A ein Interface implementiert, muss A alle Methoden implementieren.	Falsch. A kann abstrakt sein und abstrakte Verpflichtungen offenlassen.
Wenn A nicht abstrakt ist und ein Interface implementiert, muss A alle Methoden implementieren.	Im üblichen Klausursinn richtig: Für jede abstrakte Interface-Methode muss eine passende öffentliche Implementierung verfügbar sein. Sie darf auch geerbt sein.

Gemeint ist keine Pflicht zur erneuten Methodendefinition in A: Geerbte Implementierungen und geeignete default-Methoden genügen. Statische und private Interface-Methoden sind von dieser Pflicht ausgenommen.

b) 1022 in Binär- und Hexadezimaldarstellung

$1022 = 1024 - 2 = 2^{10} - 2$. Daher sind die Bits für die Gewichte 512 bis 2 gesetzt und das Einerbit ist 0:

$$\begin{aligned} 1022_{10} &= 1111111110_2 \\ &= 0011\ 1111\ 1110_2 \\ &= 3FE_{16} \end{aligned}$$

Kontrolle: $3 \cdot 256 + 15 \cdot 16 + 14 = 768 + 240 + 14 = 1022$.

c) -17 im 8-Bit-Zweierkomplement

$$\begin{aligned} +17: & \quad 00010001 \\ \text{Bits invertieren:} & \quad 11101110 \\ 1 \text{ addieren:} & \quad 11101111 \end{aligned}$$

Die gesuchte Darstellung ist 11101111. Kontrolle mit negativem höchstwertigem Bit: $-128 + 64 + 32 + 8 + 4 + 2 + 1 = -17$. Alternativ ergibt $256 - 17 = 239$ genau das Bitmuster 11101111.

Aufgabe 2: Summe und arithmetischer Mittelwert

```
import java.util.Scanner;

public class Mittelwert {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        long summe = 0;
        long anzahl = 0;
        int zahl = eingabe.nextInt();
        while (zahl != -1) {
            summe += zahl;
            anzahl++;
            zahl = eingabe.nextInt();
        }
        System.out.println("Summe: " + summe);
        if (anzahl == 0) {
            System.out.println("Mittelwert: nicht definiert");
        } else {
            double mittelwert = (double) summe / anzahl;
            System.out.println("Mittelwert: " + mittelwert);
        }
    }
}
```

-1 wird vor dem Addieren geprüft und zählt deshalb weder zur Summe noch zur Anzahl. Die Umwandlung nach double vor der Division verhindert das Abschneiden von Nachkommastellen. Werden etwa 2, 3 und -1 eingegeben, sind Summe = 5 und Mittelwert = 2.5. Bei sofortigem -1 ist die Summe 0, aber ein Mittelwert über keine Zahlen nicht definiert. long als Akkumulator verringert das Überlaufisiko gegenüber int; die Eingaben bleiben wie verlangt int.

Aufgabe 3: Häufigster Wert und häufigste Ziffer

a) Häufigster Arraywert

```
public static int berechneHaeufigstenWert(int[] feld) {
    assert feld != null;
    int[] anzahl = new int[10];
    for (int i = 0; i < feld.length; i++) {
        anzahl[feld[i]]++;
    }
    int besterWert = 0;
    for (int wert = 1; wert < anzahl.length; wert++) {
        if (anzahl[wert] > anzahl[besterWert]) {
            besterWert = wert;
        }
    }
    return besterWert;
}
```

Für jeden zulässigen Wert von 0 bis 9 speichert `anzahl` dessen Häufigkeit. Die aufsteigende Suche ersetzt den bisherigen Gewinner nur bei strikt größerer Häufigkeit. Bei Gleichstand bleibt dadurch die kleinere Zahl erhalten. Im Beispiel werden 3 und 7 jeweils dreimal gezählt, also lautet das Ergebnis 3.

Die Aufgabenstellung schließt ein leeres Feld nicht aus. Obiger Code liefert dann 0, weil alle zehn Häufigkeiten 0 sind und 0 die kleinste zulässige Zahl ist. Falls ausdrücklich ein im Feld vorkommender Wert verlangt wird, wäre das Ergebnis für ein leeres Feld nicht definiert; dann müsste zusätzlich `feld.length > 0` vorausgesetzt werden.

b) Häufigste Dezimalziffer

```
public static int berechneHaeufigsteZiffer(int n) {
    assert n > 0;
    int anzahlZiffern = 0;
    for (int rest = n; rest > 0; rest /= 10) {
        anzahlZiffern++;
    }
    int[] ziffern = new int[anzahlZiffern];
    for (int i = 0; i < ziffern.length; i++) {
        ziffern[i] = n % 10;
        n /= 10;
    }
    return berechneHaeufigstenWert(ziffern);
}
```

Modulo 10 liefert jeweils die letzte Ziffer, ganzzahlige Division durch 10 entfernt sie. Die Ziffern werden zwar rückwärts abgelegt, doch für ihre Häufigkeiten spielt die Reihenfolge keine Rolle. Für 2031771 kommen 1 und 7 je zweimal vor; die Methode aus a) entscheidet zugunsten von 1. Innere Nullen werden durch den Restoperator ebenfalls gezählt.

Aufgabe 4: Arrays

a) Reihenfolge ohne neues Array umkehren

```
public static void umkehr(int[] feld) {
    assert feld != null;
    for (int links = 0; links < feld.length / 2; links++) {
        int rechts = feld.length - 1 - links;
        int temp = feld[links];
        feld[links] = feld[rechts];
        feld[rechts] = temp;
    }
}
```

Jeweils das erste und letzte, dann das zweite und vorletzte Element werden getauscht. temp bewahrt den ersten Wert auf, bevor er überschrieben wird. Es entsteht kein neues Array. Bei ungerader Länge bleibt das mittlere Element stehen. Das Beispiel ergibt {51, 4, 11, 27, 2, -5, 13, -7, 1}; leere und einelementige Arrays bleiben korrekt unverändert.

b) Treppenumuster

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

In Zeile z liegen die n Sterne in den Spalten z bis z + n - 1. Die übrigen Zellen werden ausdrücklich mit Leerzeichen gefüllt, weil der Standardwert eines char-Arrays das Nullzeichen und kein Leerzeichen ist. Für n = 5 lautet das Muster:

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

Wie bei solchen Arrayaufgaben üblich muss die gewünschte Größe im verfügbaren Speicher und im int-Indexbereich liegen.

Aufgabe 5: Berufsklassen

Anwalt

```
public class Anwalt extends Person {
    public Anwalt(String name) {
        super(name);
    }

    @Override
    public int getGehalt() {
        return 20000;
    }

    @Override
    public String getBeruf() {
        return "Anwalt";
    }
}
```

Der Name wird im geerbten Person-Anteil initialisiert. Beruf und Gehalt sind für die geforderte Anwaltsklasse konstant.

Politiker

```
public class Politiker extends Person {
    private int gehalt;
    private char geschlecht;

    public Politiker(String name, int gehalt, char geschlecht) {
        super(name);
        this.gehalt = gehalt;
        this.geschlecht = geschlecht;
    }

    @Override
    public int getGehalt() {
        return gehalt;
    }

    @Override
    public String getBeruf() {
        return geschlecht == 'w' ? "Politikerin" : "Politiker";
    }
}
```

Das Gehalt wird je Objekt gespeichert. Für die in der Vorlage verwendeten Werte 'm' und 'w' wird die passende Berufsbezeichnung zurückgegeben.

Manager

```
public class Manager extends Person {
    private static int anzahlManager = 0;
    private int nummer;

    public Manager() {
        super("Franz der " + (++anzahlManager) + ".");
        this.nummer = anzahlManager;
    }

    @Override
    public int getGehalt() {
        return nummer * 10000;
    }

    @Override
    public String getBeruf() {
        return "Manager";
    }
}
```

Der Klassenzähler vergibt fortlaufende Nummern; die Objektvariable nummer hält die eigene Nummer dauerhaft fest. Dadurch verdient der erste Manager weiterhin 10000 Euro, auch nachdem der zweite erstellt wurde. Das geerbte Person.toString ruft getBeruf und getGehalt dynamisch auf. Keine Unterklasse überschreibt toString. Es entsteht exakt:

```
Andreas ist Anwalt und verdient 20000 Euro
Helmut ist Politiker und verdient 10000 Euro
Heidi ist Politikerin und verdient 5000 Euro
Franz der 1. ist Manager und verdient 10000 Euro
Franz der 2. ist Manager und verdient 20000 Euro
```

Aufgabe 6: Fehlersuche

Das Diagramm zeigt: A ist abstrakt, B erweitert A und implementiert I. C implementiert I, ist aber keine Unterklasse von A. A stellt `methode1` und `methode2` bereit; nur `methode1` ist abstrakt. I deklariert ausschließlich `methodel`.

Zeile	Fehler?	Begründung
<code>einA.methode1();</code>	Nein	Die Methode ist in A deklariert. Ein virtueller Aufruf einer abstrakt deklarierten Methode ist zulässig.
<code>einA.methode2();</code>	Nein	Die konkrete Methode ist in A vorhanden.
<code>einA.methodel();</code>	X	Der statische Typ A kennt <code>methodel</code> nicht und implementiert I nicht.
<code>einB.methode1();</code>	Nein	B implementiert die abstrakte Methode aus A.
<code>einB.methode2();</code>	Nein	B erbt die Methode von A.
<code>einB.methodel();</code>	Nein	B implementiert I und diese Methode.
<code>einI.methode1();</code>	X	<code>methode1</code> gehört nicht zum statischen Typ I.
<code>einI.methodel();</code>	Nein	Diese Methode ist in I deklariert.
<code>A einA;</code>	Nein	Eine Variable mit abstraktem Typ ist zulässig.
<code>einA = new A();</code>	X	A ist abstrakt und kann nicht unmittelbar instanziiert werden.
<code>einA = new B();</code>	Nein	Eine B-Referenz lässt sich nach A erweitern.
<code>einA = new C();</code>	X	C ist keine Unterklasse von A.
<code>I einI;</code>	Nein	Eine Interface-Referenzvariable ist zulässig.
<code>einI = new B();</code>	Nein	B implementiert I.
<code>einI = new C();</code>	Nein	C implementiert I.
<code>einI = new I();</code>	X	Ein Interface kann nicht unmittelbar instanziiert werden.

Alle übrigen Klassendeklarationen, Methodenköpfe und Klammern sind korrekt. Die fehlerhaften Zeilen werden unabhängig voneinander bewertet; ein Gesamtprogramm mit diesen Fehlern wäre selbstverständlich nicht übersetzbar.

Aufgabe 7: Gebäude und Raum

Klasse Gebäude

```
public class Gebäude {
    private String adresse;
    private float gesamtfläche;
    private Raum[] räume = new Raum[10];
    private int anzahlRäume = 0;

    public Gebäude(String adresse, float gesamtfläche) {
        this.adresse = adresse;
        this.gesamtfläche = gesamtfläche;
    }

    public float getGesamtfläche() {
        return gesamtfläche;
    }

    public int getAnzahlRäume() {
        return anzahlRäume;
    }

    public boolean addRaum(Raum raum) {
        if (raum == null || anzahlRäume == räume.length) {
            return false;
        }
        if (berechneRaumfläche() + raum.berechneFläche()
            > gesamtfläche) {
            return false;
        }
        räume[anzahlRäume] = raum;
        anzahlRäume++;
        return true;
    }

    public float berechneRaumfläche() {
        float summe = 0.0f;
        for (int i = 0; i < anzahlRäume; i++) {
            summe += räume[i].berechneFläche();
        }
        return summe;
    }

    public float berechneWohnfläche() {
        float summe = 0.0f;
        for (int i = 0; i < anzahlRäume; i++) {
            if (räume[i] instanceof Wohnraum) {
                summe += räume[i].berechneFläche();
            }
        }
        return summe;
    }
}
```

Array und Zähler sind die einzigen ergänzten Attribute und verwalten die Assoziation mit höchstens zehn Räumen. Ein Raum wird nur eingefügt, wenn sowohl ein Platz als auch ausreichend Fläche vorhanden ist. Exaktes Ausschöpfen der Fläche ist erlaubt, deshalb lautet der Ablehnungsvergleich `>` und nicht `>=`. `berechneWohnfläche` addiert über `instanceof` nur

Wohnräume; Garagen werden ausgeschlossen. Wohnraum und Garage werden entsprechend dem UML als vorhandene Unterklassen vorausgesetzt.

Abstrakte Klasse Raum

```
public abstract class Raum {
    private String bezeichnung;
    private float länge;
    private float breite;

    public Raum(String bezeichnung, float länge, float breite) {
        this.bezeichnung = bezeichnung;
        this.länge = länge;
        this.breite = breite;
    }

    public float berechneFläche() {
        return länge * breite;
    }
}
```

Die Klasse ist laut UML abstrakt, obwohl ihre einzige normale Methode konkret ist. Die Fläche eines rechteckigen Raums ist Länge · Breite. Es werden gültige, nichtnegative Abmessungen vorausgesetzt; zusätzliche Getter oder Hilfsmethoden werden nicht ergänzt. Das UML verlangt für die Flächen ausdrücklich float, daher verwendet die Lösung diesen Typ.

Aufgabe 8: Speicherabbild

Bezeichne das erste B-Objekt mit O1 und das zweite mit O2. Direkt vor return in `b1.gleich(b2)` gilt:

Speicherbereich	Inhalt
Klassenattribut A.c	5
O1, Typ B	A.m = 3; A.n = 7; B.m = 10
O2, Typ B	A.m = 1; A.n = 4; B.m = 5
main: b1; b2	b1 → O1; b2 → O2
main: erg	Noch nicht initialisiert, weil gleich noch nicht zurückgekehrt ist.
main: args	Referenz auf das übergebene String-Array; Inhalt nicht angegeben.
B.gleich: this	Referenz auf O1
B.gleich: b	Referenz auf O2
B.gleich: erg	false

Pro B-Objekt wird c zweimal erhöht: einmal im A-Konstruktor und einmal im B-Konstruktor. Aus dem Startwert 1 wird damit 5. Die beiden gleichnamigen m-Felder sind voneinander unabhängig. Die Getter lesen die privaten Felder des A-Anteils; der direkte Zugriff m beziehungsweise b.m innerhalb B liest dagegen B.m.

Der Vergleich lautet deshalb $(3 + 7 + 10) == (1 + 4 + 5)$, also $20 == 10$ und damit false. Die Getteraufrufe sind am Zeitpunkt 1 bereits beendet; nur main und gleich sind noch aktiv. Nach der Rückkehr erhält main.erg den Wert false und das Programm gibt false aus.