

Lösungen: Einführung in die Programmierung, 18.07.2018

Ausgangsdatei: 10_EidP_Klausur_18.07.2018.pdf

Vollständige Lösungen der Aufgaben 1 bis 8. Öffentliche Java-Klassen sind in getrennten Dateien zu speichern. Die Bonuspunkte auf dem Deckblatt sind keine zusätzliche Aufgabenstellung. Eine versionsabhängige Aussage zu Interfaces wird ausdrücklich eingeordnet.

Aufgabe 1: Multiple Choice und Wissen

a) Anweisungsfolgen und Aussagen

Anweisungsfolge	Bewertung und Begründung
<code>double x; float y = x; x = y;</code>	Falsch. x ist als lokale Variable nicht initialisiert; außerdem ist double nach float ohne explizite Typumwandlung nicht zulässig.
<code>long x; float y = x; x = y;</code>	Falsch. x ist nicht initialisiert. long nach float wäre zwar erlaubt, aber die spätere Zuweisung float nach long benötigt eine Typumwandlung.
<code>long x = true;</code>	Falsch. boolean ist nicht in einen numerischen Typ konvertierbar.
<code>int x=5; long y=5; boolean z=(x==y);</code>	Richtig. Zum Vergleich wird x nach long erweitert; das Ergebnis ist true.

Aussage zur Übersetzung	Bewertung und Begründung
Java-Quellprogramme werden vom Java-Compiler in Maschinencode übersetzt.	Falsch im üblichen Java-Modell. javac erzeugt Bytecode für die JVM.
Java-Quellprogramme werden durch einen Interpreter ausgeführt.	Falsch bei wörtlicher Auslegung: Der Quelltext wird zunächst übersetzt; die JVM interpretiert beziehungsweise kompiliert den Bytecode.
Ein Java-Quellprogramm besteht immer aus einer Datei.	Falsch. Es darf aus mehreren Quelldateien bestehen.
Java-Bytecode wird von einer virtuellen Maschine ausgeführt.	Richtig. Das ist die Aufgabe der JVM.

Die Unterscheidung zwischen Quelltext und Bytecode ist für die zweite Aussage entscheidend. Auch wenn eine JVM einen Interpreter enthält, interpretiert sie nicht unmittelbar den Java-Quelltext.

Ausdruck	Ergebnis und Begründung
<code>0xffff + 1 == 0xffff1</code>	false. Links steht 4096 = 0x1000, rechts 65521.
<code>0xffffffff + 1 == 0</code>	true. Ohne L ist 0xffffffff der int-Wert -1; -1 + 1 = 0.
<code>0x0fffffff + 1 < 0</code>	false. Das Ergebnis 0x10000000 = 268435456 ist positiv.

Ausdruck	Ergebnis und Begründung
<code>0xffffffffL + 1 < 0</code>	false. Mit L ist der erste Wert 4294967295; die Summe 4294967296 passt positiv in long.
Aussage über Interface A	Bewertung und Begründung
A muss einen Konstruktor besitzen.	Falsch. Interfaces haben keine Konstruktoren.
Von A können keine Objekte erzeugt werden.	Richtig im direkten Sinn: <code>new A()</code> ist unzulässig. Eine implementierende Klasse kann instanziiert werden.
Von A dürfen keine Unterklassen gebildet werden.	Falsch im gemeinten Sinn von Untertypen. Unterinterfaces können A erweitern; Klassen können A implementieren.
A darf keine privaten Methoden enthalten.	Versionsabhängig: für Java 8 richtig, seit Java 9 falsch. Private Hilfsmethoden mit Rumpf sind seit Java 9 erlaubt.

Die Klausur nennt keine Java-Version. Deshalb hat die letzte Aussage ohne zusätzliche Versionsannahme keine eindeutige Antwort. Bei einer Lehrveranstaltung auf Grundlage von Java 8 wäre dort „richtig“ anzukreuzen. Private abstrakte Interface-Methoden sind auch seit Java 9 nicht erlaubt.

b) Zweierkomplement für int

Ein Java-int hat 32 Bit und den Wertebereich -2^{31} bis $2^{31} - 1$, also -2147483648 bis 2147483647. Das höchstwertige Bit besitzt das Gewicht -2^{31} , die übrigen Bits haben positive Zweierpotenzgewichte. Für eine negative Zahl -a erhält man das Bitmuster durch Invertieren der 32 Bits von a und anschließendes Addieren von 1.

```
+5: 00000000 00000000 00000000 00000101
-5: 11111111 11111111 11111111 11111011
```

Beim zweiten Muster ergeben invertierte Bits zunächst ...11111010; plus 1 ergibt ...11111011. Es existiert nur eine Darstellung der Null. int-Arithmetik verwirft bei Überlauf die über 32 Bit hinausgehenden Bits.

Aufgabe 2: Fußballergebnisse und Punktetabelle

```
import java.util.Scanner;

public class Spielplan {
    public static int punkte(int eigeneTore, int fremdeTore) {
        if (eigeneTore > fremdeTore) {
            return 3;
        }
        return eigeneTore == fremdeTore ? 1 : 0;
    }

    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        int punkteDeutschland, punkteItalien, punkteNiederlande;
        do {
            punkteDeutschland = 0;
            punkteItalien = 0;
            punkteNiederlande = 0;
            System.out.println("(1) Italien - Niederlande");
            int italien = eingabe.nextInt();
            int niederlande = eingabe.nextInt();
            punkteItalien += punkte(italien, niederlande);
            punkteNiederlande += punkte(niederlande, italien);

            System.out.println("(2) Niederlande - Deutschland");
            niederlande = eingabe.nextInt();
            int deutschland = eingabe.nextInt();
            punkteNiederlande += punkte(niederlande, deutschland);
            punkteDeutschland += punkte(deutschland, niederlande);

            System.out.println("(3) Deutschland - Italien");
            deutschland = eingabe.nextInt();
            italien = eingabe.nextInt();
            punkteDeutschland += punkte(deutschland, italien);
            punkteItalien += punkte(italien, deutschland);
        } while (punkteItalien == punkteNiederlande
            || punkteItalien == punkteDeutschland
            || punkteNiederlande == punkteDeutschland);
        System.out.println("Mannschaft Punkte");
        System.out.println("Italien " + punkteItalien);
        System.out.println("Niederlande " + punkteNiederlande);
        System.out.println("Deutschland " + punkteDeutschland);
    }
}
```

Jeder Durchlauf liest alle drei Spiele neu ein und setzt zuvor sämtliche Punkte zurück. Die Hilfsmethode `punkte` liefert aus Sicht eines Teams 3 für einen Sieg, 1 für ein Unentschieden und sonst 0. Sie wird pro Spiel für beide Mannschaften aufgerufen. Sobald irgendeines der drei Mannschaftspaare punktgleich ist, wiederholt die `do-while`-Schleife die gesamte Eingabe. Ausgegeben wird die schließlich eindeutige Tabelle; ihre Reihenfolge muss laut Aufgabenstellung nicht sortiert sein. Es werden gültige, nichtnegative Torzahlen vorausgesetzt.

Aufgabe 3: Summe, Dezimaldarstellung und Quersumme

a) Arrayelemente addieren

```
public static int berechneSumme(int[] feld) {
    int summe = 0;
    for (int i = 0; i < feld.length; i++) {
        summe += feld[i];
    }
    return summe;
}
```

Die Schleife nimmt jedes Element genau einmal in die Summe auf. Für {5, 1, 5, 4} entsteht 15; das leere Array ergibt 0. Voraussetzung ist `feld != null`. Bei mathematischen Summen außerhalb des `int`-Bereichs würde Java überlaufen; die Aufgabenlösung setzt einen darstellbaren Summenwert voraus.

b) Ziffern der Dezimaldarstellung

```
public static int[] dezimalDarstellung(int n) {
    // Voraussetzung: n >= 0
    int[] ziffern = new int[laengeDezimalDarstellung(n)];
    for (int i = ziffern.length - 1; i >= 0; i--) {
        ziffern[i] = n % 10;
        n /= 10;
    }
    return ziffern;
}
```

Die vorgegebene Methode `laengeDezimalDarstellung` liefert die Arraylänge. Der Rest modulo 10 ist die letzte Ziffer; die ganzzahlige Division durch 10 entfernt sie. Deshalb wird von rechts nach links gefüllt. 5145 ergibt {5, 1, 4, 5}. Auch 0 wird korrekt als Array {0} der Länge 1 dargestellt.

c) Quersumme

```
public static int quersumme(int n) {
    return berechneSumme(dezimalDarstellung(n));
}
```

Zuerst entstehen die einzelnen Ziffern, anschließend werden diese addiert. Für 5145 ergibt sich $5 + 1 + 4 + 5 = 15$; für 0 ergibt sich 0. Die erlaubten eigenen Methoden werden wiederverwendet.

Aufgabe 4: Bildarrays

a) Horizontale Symmetrie

```
public static boolean istHorizontalSymmetrisch(byte[][] bild) {
    int spalten = bild[0].length;
    for (int z = 0; z < bild.length; z++) {
        for (int s = 0; s < spalten / 2; s++) {
            if (bild[z][s] != bild[z][spalten - 1 - s]) {
                return false;
            }
        }
    }
    return true;
}
```

Innerhalb jeder Zeile werden spiegelbildliche Positionen verglichen. Die mittlere Spalte bei ungerader Breite braucht keinen Vergleich. Bereits eine Abweichung widerlegt die Symmetrie.

Hinweis zur Aufgabenformulierung: Java-byte ist vorzeichenbehaftet und reicht von -128 bis 127. Graustufen von 128 bis 255 werden deshalb als entsprechende 8-Bit-Muster in negativen byte-Werten gespeichert. Die Prüfung auf Gleichheit und die anschließende Drehung funktionieren davon unabhängig; ein unsigned Grauwert wäre beispielsweise mit `bild[z][s] & 0xff` lesbar.

b) Drehung um 90 Grad nach links

```
public static byte[][] drehe90GradLinks(byte[][] bild) {
    int zeilen = bild.length;
    int spalten = bild[0].length;
    byte[][] gedreht = new byte[spalten][zeilen];
    for (int z = 0; z < zeilen; z++) {
        for (int s = 0; s < spalten; s++) {
            gedreht[spalten - 1 - s][z] = bild[z][s];
        }
    }
    return gedreht;
}
```

Die neue Zeilennummer ist die umgekehrte alte Spaltennummer, die neue Spaltennummer ist die alte Zeilennummer. So wird aus dem 3×4-Beispiel:

```
4  8 12
3  7 11
2  6 10
1  5  9
```

Aufgabe 5: Tiere und Freund-Interface

a) Klasse Tiger

```
public class Tiger extends Tier implements Freund {  
    public Tiger(String name, Freund freund) {  
        super(name, freund);  
    }  
  
    @Override  
    public String getName() {  
        return name;  
    }  
  
    @Override  
    public String gibLaut() {  
        return "Grrr";  
    }  
}
```

Tiger muss Freund implementieren, damit ein Tiger als freund einer Ziege gespeichert werden kann. Die Klasse ergänzt den benötigten Konstruktor, die abstrakte Tier-Methode gibLaut sowie die Interface-Methode getName. Auf das geerbte protected-Feld name kann Tiger zugreifen.

b) Objekte erzeugen

```
Tiger t2 = new Tiger("Rolf", null);  
Ziege t1 = new Ziege("Shaun", t2);
```

Rolf wird zuerst erzeugt, damit seine Referenz anschließend an die Ziege übergeben werden kann. t1.freund verweist auf dasselbe Tiger-Objekt wie t2; t2.freund ist null.

c) Gemeinsame Textdarstellung in Tier

```
@Override  
public String toString() {  
    return name + ": " + gibLaut() + "!";  
}
```

Der Aufruf gibLaut wird dynamisch an Ziege beziehungsweise Tiger gebunden. Das Ausrufezeichen wird nur in toString ergänzt; Tiger.gibLaut liefert deshalb "Grrr" ohne Ausrufezeichen. Die Ausgabe ist:

```
Shaun: Mäh!  
Rolf: Grrr!
```

Aufgabe 6: Lager, Behälter und Karton

a) bis d) Assoziation und Lager-Methoden

Das folgende Fragment zeigt die geforderten Attribute und fett gedruckten Methoden innerhalb der Klasse Lager. Der im UML vorhandene, nicht fett gedruckte Konstruktor Lager(String bezeichnung) wird vorausgesetzt und weist seinem Attribut den übergebenen Namen zu; er ist laut Aufgabe nicht erneut zu programmieren.

```
private String bezeichnung;
private Behälter[] behälter = new Behälter[500];
private int anzahlBehälter = 0;

public String getBezeichnung() {
    return bezeichnung;
}

public int getAnzahlBehälter() {
    return anzahlBehälter;
}

public boolean addBehälter(Behälter b) {
    if (b == null || anzahlBehälter == behälter.length) {
        return false;
    }
    behälter[anzahlBehälter] = b;
    anzahlBehälter++;
    return true;
}

public double berechneGesamtvolumen() {
    double volumen = 0.0;
    for (int i = 0; i < anzahlBehälter; i++) {
        volumen += behälter[i].berechneVolumen();
    }
    return volumen;
}
```

Das Array bildet die UML-Obergrenze 500 ab, der Zähler begrenzt den belegten Bereich. Die Summe benutzt den dynamisch passenden Volumenaufruf, unabhängig davon, ob ein Element ein Karton oder eine Dose ist. Wie bei der üblichen Umsetzung dieser einseitig navigierbaren Assoziation wird vorausgesetzt, dass übergebene Behälter nicht bereits einem anderen Lager zugeordnet sind; eine lagerübergreifende Eigentumsprüfung ist in den angegebenen Methoden nicht vorgesehen.

e) Abstrakte Basisklasse und automatische Nummern

```
public abstract class Behälter {
    private String aufschrift;
    private int nummer;
    private static int letzteNummer = 0;

    public Behälter(String aufschrift) {
        this.aufschrift = aufschrift;
        this.nummer = ++letzteNummer;
    }

    public abstract double berechneVolumen();

    // Die nicht fett gedruckten Getter sind hier wie gefordert
    // ausgespart; sie gehören zur vorgegebenen Klasse.
}
```

Alle Behälter-Unterklassen teilen denselben static-Zähler. Der erste Konstruktoraufruf vergibt 1, der nächste 2 usw. Die individuelle nummer ist hingegen ein Objektattribut und verändert sich bei späteren Erzeugungen nicht. berechneVolumen bleibt abstrakt, da das Volumen von der konkreten Form abhängt.

f) und g) Karton-Konstruktor und Volumen

```
public class Karton extends Behälter {
    private double breite;
    private double hoehe;
    private double tiefe;

    public Karton(String aufschrift, double breite,
                  double hoehe, double tiefe) {
        super(aufschrift);
        this.breite = breite;
        this.hoehe = hoehe;
        this.tiefe = tiefe;
    }

    @Override
    public double berechneVolumen() {
        return breite * hoehe * tiefe;
    }

    // Die nicht fett gedruckten Getter werden nicht verlangt.
}
```

super initialisiert Aufschrift und Nummer. Ein quaderförmiger Karton besitzt das Volumen Breite · Höhe · Tiefe, beispielsweise $2 \cdot 3 \cdot 4 = 24$. Sinnvolle, nichtnegative Abmessungen werden vorausgesetzt. Die Klasse Dose muss laut Aufgabenstellung nicht programmiert werden.

Aufgabe 7: Speicherabbild

Der Zeitpunkt liegt innerhalb von A.summe, das von B.summe über super aufgerufen wurde. Benenne die Objekte O1 und O2:

Speicherbereich	Inhalt am Zeitpunkt 1
A.c	3
O1, Typ A	A.m = 1; A.n = 2
O2, Typ B	A.m = 4; A.n = 3; B.m = 7
main: args	null gemäß der Vorlage
main: a; b	a → O1; b → O2
main: n	Noch nicht initialisiert: die rechte Seite der Zuweisung läuft noch.
B.summe: this; i; erg	this → O2; i = 10; erg = 7
A.summe: this; i; erg	this → O2; i = 10; erg = 17

Jeder A-Konstruktor erhöht c einmal, also $1 \rightarrow 2 \rightarrow 3$. B(3, 4) ruft super(4, 3) auf, sodass die beiden Felder im A-Anteil vertauscht initialisiert werden. Das separate B.m enthält $3 + 4 = 7$.

In B.summe wird erg zunächst 7. Während der Auswertung von $\text{erg} += \text{super.summe}(i)$ ist das Ergebnis des Unteraufrufs noch nicht zurückgegeben; erg bleibt deshalb am markierten Zeitpunkt 7. A.summe berechnet $4 + 3 + 10 = 17$. Nach der Rückkehr ergibt B.summe $7 + 17 = 24$; anschließend erhält main.n diesen Wert und die spätere Ausgabe lautet 24.

Aufgabe 8: Programmausgabe

A: 11
A: 21
A: double
B: 11
A: 32
B: double
A: 44
C: 100140
B: double
C: 1001

Beim ersten Objekt ist $z = 1$ und $A.x = 11$. `print(20)` benutzt die `int`-Überladung und gibt $1 + 20 = 21$ aus; `print(2.5)` benutzt die `double`-Überladung.

Beim B-Objekt führt der implizite B-Konstruktor den A-Konstruktor aus. Damit ist $z = 2$. Das als A deklarierte `einB` greift bei `einB.x` auf $A.x = 11$ zu. B überschreibt nur `print(double)`: `print(30)` bleibt `A.print(int)` und ergibt 32, `print(2.5)` schreibt „B: double“.

Beim C-Objekt erhöhen A- und C-Konstruktor z auf insgesamt 4; `C.x` wird 1001. `C.print(40)` ruft zuerst die geerbte `int`-Methode über `super` auf und gibt $4 + 40 = 44$ aus. Danach werden bei `"C: " + x + i` die Werte von links nach rechts als Text verkettet: 1001 und 40 ergeben "100140", keine Addition. Die `double`-Methode stammt aus B. Der letzte Zugriff liest `C.x = 1001`.