

Lösungen zur Klausur vom 28.07.2023

Quelle: 20_EidP_Klausur_28.07.2023 (MI und INF).pdf. Einführung in die Programmierung, INF/MI, Version 1.1. Alle fünf Aufgaben und sämtliche Teilaufgaben sind bearbeitet. Vereinfachungen bzw. Fehler in der Aufgabenstellung werden an der jeweiligen Stelle kenntlich gemacht.

Aufgabe 1 - Wissensfragen

Die Nummern der Antworten beziehen sich jeweils auf ihre Reihenfolge im Aufgabenblatt.

1a) Java und Bytecode

Die im vereinfachten Lehrmodell erwarteten zwei Kreuze stehen bei Aussage 2 und Aussage 4.

Nr.	Bewertung und Begründung
1	Falsch. Java-Quellcode wird in Bytecode übersetzt; dafür ist kein eigener Quellcode-Compiler pro Zielprozessortyp nötig.
2	Richtig. Der Compiler erzeugt plattformunabhängigen Bytecode. Für dessen Ausführung wird eine passende JVM benötigt.
3	Falsch. Ein Interpreter führt Bytecode aus; er rekonstruiert keinen Java-Quellcode.
4	Im vereinfachten Modell richtig: reine Interpretation verursacht gegenüber direkter nativer Ausführung zusätzlichen Aufwand. Als allgemeine Aussage über Java-Programme ist „langsamer“ jedoch zu pauschal; JIT-Kompilierung kann häufig ausgeführten Code in Maschinencode übersetzen.
5	Falsch. Die JVM dient der Ausführung des Bytecodes; der Java-Compiler muss nicht mittels einer virtuellen Maschine die Zielplattform simulieren.
6	Falsch. Der Java-Compiler erzeugt Bytecode, keinen neuen Interpreter für die Zielplattform.

Streng formuliert ist nur Aussage 2 uneingeschränkt korrekt. Aussage 4 ist die offenbar beabsichtigte zweite Antwort, wenn man das in der Frage verwendete Interpretationsmodell zugrunde legt.

1b) Sichtbarkeit

Richtig sind Aussage 4 und Aussage 5.

Nr.	Bewertung	Grund
1	Falsch	protected ist zusätzlich im gleichen Package und unter bestimmten Bedingungen in Unterklassen zugänglich.
2	Falsch	Eine Unterklasse außerhalb des Packages kann auf geerbte protected-Member zugreifen; dabei gelten besondere Regeln für den qualifizierenden Empfänger.

Nr.	Bewertung	Grund
3	Falsch	public-Member sind grundsätzlich auch außerhalb des Packages zugänglich, wenn der deklarierende Typ zugänglich ist.
4	Richtig	public-Member sind für Unterklassen im gleichen Package sichtbar.
5	Richtig	Ohne Sichtbarkeitsmodifikator gilt Package-Sichtbarkeit.
6	Falsch	Eine Unterklasse außerhalb des Packages erhält keinen Zugriff auf package-private-Member allein durch die Vererbung.

1c) Syntax und Semantik

Die beabsichtigten Antworten sind Aussage 2 und Aussage 5. Die Berechnung $A - B$ ist ein Semantikfehler relativ zur verlangten Addition: Der Ausdruck ist gültiges Java, berechnet aber das falsche Ergebnis. Die zwei Rückgabetypen void und int sind ein Syntaxfehler. Im Blatt steht hierzu irrtümlich „Zeile 4“; tatsächlich stehen die beiden Wörter in Zeile 3.

Eine korrigierte Methode lautet:

```
public int addieren(int A, int B) {
    int ergebnis = A + B;
    return ergebnis;
}
```

Eine normale Hilfsklasse benötigt keine main-Methode. Großbuchstaben in Variablenamen sind erlaubt, auch wenn sie hier nicht der üblichen Konvention entsprechen. TaschenAddierer ist ein gültiger Klassenname und ergebnis ein gültiger Variablenname.

1d) Arrays

Alle vier Aussagen sind falsch.

Nr.	Warum falsch?
1	Arrays sind geordnet nach Index, aber nicht automatisch nach Werten sortiert. Der erste Index ist 0, nicht 1.
2	Bei Länge n reichen die gültigen Indizes von 0 bis n - 1.
3	++ erhöht einen numerischen Wert; es vergrößert kein Array. Die Arraylänge ist nach der Erzeugung fest.
4	Java erlaubt auch mehr als zwei Dimensionen, realisiert als Arrays von Arrays.

1e) Operatoren

Nr.	Aussage in Kurzform	Bewertung	Erklärung
1	= vergleicht	Falsch	= weist zu; == vergleicht auf Gleichheit.
2	?= prüft Ungleichheit	Falsch	Einen solchen Operator gibt es nicht; Ungleichheit heißt !=.
3	Einzelner senkrechter Strich: Oder	Richtig	Der einzelne senkrechte Strich ist bei boolean das nicht kurzschließende logische Oder.
4	Einzelner senkrechter Strich: beide Operanden	Richtig	Auch bei true links wird der rechte Operand ausgewertet.
5	^ ist logisches Und	Falsch	^ ist bei boolean exklusives Oder.
6	^ wertet beide Operanden aus	Richtig	Beide Werte werden zur Bestimmung des exklusiven Oder benötigt.
7	&& ist exklusives Oder	Falsch	&& ist kurzschließendes logisches Und.
8	&& wertet beide Operanden immer aus	Falsch	Bei false links wird der rechte Operand übersprungen.

Damit lautet die Folge: falsch, falsch, richtig, richtig, falsch, richtig, falsch, falsch. Die Aussagen zur Auswertung beziehen sich auf normal abgeschlossene Auswertungen; ein vorzeitiger Abbruch durch eine Ausnahme ist davon zu unterscheiden.

1f) Überladen

Compilerfehler verursachen die Paare 01 und 06 sowie 03 und 05. Beide Methoden eines solchen Paares haben denselben Namen und dieselben Parametertypen in derselben Reihenfolge. Nur der Rückgabotyp unterscheidet sich; das genügt in Java nicht zum Überladen.

Die übrigen angebotenen Paare unterscheiden sich in mindestens einem Parametertyp und sind als Deklarationen zulässig. Die Namen der Parameter sind für diese Entscheidung ebenfalls unerheblich.

Aufgabe 2 - Kleinste Eingabewerte

Die Methoden können in eine gemeinsame Klasse eingefügt werden. Wie im Blatt gefordert sind keine umgebende Klasse und keine import-Anweisung nötig. Scanner wird mit seinem vollständigen Namen verwendet. Es werden gültige int-Eingaben bis einschließlich eines nichtnegativen Abschlusswerts vorausgesetzt. Rechenergebnisse sollen im int-Bereich liegen; andernfalls würde die Java-int-Arithmetik überlaufen.

a) Ein Maximum gegebenenfalls ersetzen

```
void ersetzeMaximum(int[] feld, int wert) {
    if (feld == null || feld.length == 0) return;
    int maxIndex = 0;
    for (int i = 1; i < feld.length; i++) {
        if (feld[i] > feld[maxIndex]) maxIndex = i;
    }
    if (wert < feld[maxIndex]) feld[maxIndex] = wert;
}
```

maxIndex bezeichnet nach der Schleife einen Index mit maximalem Wert. Nur dieser eine Platz wird ersetzt und nur bei echt kleinerem wert. Bei mehrfach vorkommendem Maximum genügt der zuerst gefundene Index. Aus {-5, -7, -3} wird bei wert = -6 das Feld {-5, -7, -6}. Ist wert mindestens so groß wie das Maximum, bleibt alles unverändert.

b) Summe der zwei kleinsten Zahlen

```
int berechneSummeVon2Min() {
    java.util.Scanner eingabe = new java.util.Scanner(System.in);
    int[] kleinste = new int[2];
    int wert = eingabe.nextInt();
    while (wert < 0) {
        ersetzeMaximum(kleinste, wert);
        wert = eingabe.nextInt();
    }
    return kleinste[0] + kleinste[1];
}
```

Beide Arrayplätze beginnen bei 0. Da alle berücksichtigten Eingaben negativ sind, ersetzt jede der ersten beiden Zahlen einen noch vorhandenen Nullplatz. Danach hält das Array stets die zwei kleinsten bisher gelesenen Zahlen: Eine neue Zahl kommt nur hinein, wenn sie kleiner als der größere der beiden gespeicherten Werte ist.

Die Nullplätze sind für die Summe neutral. Daher erhält man ohne negative Zahl 0, bei genau einer negativen Zahl diese Zahl selbst und sonst die Summe der zwei kleinsten Zahlen. Wiederholte Zahlen zählen als getrennte Eingaben. Für die Beispielsequenz werden -7 und -6 behalten, also -13. Der abschließende Wert ab 0 wird nicht verarbeitet. Der Scanner wird hier nicht geschlossen, damit System.in verfügbar bleibt.

c) Produkt der k kleinsten Zahlen

```
int berechneProduktVonMin(int k) {
    java.util.Scanner eingabe = new java.util.Scanner(System.in);
    int[] kleinste = new int[k];
    int anzahl = 0;
    int wert = eingabe.nextInt();
    while (wert < 0) {
        if (anzahl < k) {
            kleinste[anzahl++] = wert;
        } else {
            ersetzeMaximum(kleinste, wert);
        }
        wert = eingabe.nextInt();
    }
    int produkt = 1;
    for (int i = 0; i < anzahl; i++) {
        produkt *= kleinste[i];
    }
    return produkt;
}
```

Solange weniger als k Werte vorliegen, werden sie vollständig gespeichert. Danach enthält das Array stets die k kleinsten bisher gesehenen Eingaben: Ein kleinerer neuer Wert verdrängt genau das bisherige Maximum dieser Auswahl. `anzahl` bleibt ab dann k.

Nur belegte Arrayplätze werden multipliziert. Bei weniger als k Eingaben zählen alle vorhandenen Werte; ohne Eingabe bleibt das leere Produkt 1. Für das Beispiel mit $k = 3$ werden -7, -6 und -5 behalten; das Produkt ist -210. Die Reihenfolge im Auswahlarray ist unerheblich.

Aufgabe 3 - Vektoren und Matrizen

Die Arrays bilden rechteckige, vorhandene Matrizen ab. Für ein Produkt $A \cdot B$ muss die Spaltenanzahl von A gleich der Zeilenanzahl von B sein. Bei den Zeilen- und Spaltenzugriffen werden gültige Indizes vorausgesetzt. Die vier statischen Methoden gehören in dieselbe Klasse.

a) Skalarprodukt

```
public static int skalarProdukt(int[] v, int[] w) {
    int summe = 0;
    for (int i = 0; i < v.length; i++) {
        summe += v[i] * w[i];
    }
    return summe;
}
```

Die Methode addiert die Produkte entsprechender Komponenten. Das Beispiel auf dem Aufgabenblatt enthält einen Rechenfehler: Für $v = (2, 4, 1)$ und $w = (-3, 0, 3)$ gilt $2 \cdot (-3) + 4 \cdot 0 + 1 \cdot 3 = -6 + 0 + 3 = -3$, nicht 1.

b) Zeile zurückgeben

```
public static int[] getZeile(int[][] A, int i) {
    return A[i];
}
```

Ein zweidimensionales Array ist in Java ein Array von Zeilenreferenzen. Daher liefert $A[i]$ bereits die gesuchte Zeile. Hier wird die vorhandene Zeile zurückgegeben, keine Kopie; spätere Änderungen an dieser Referenz würden auch die Matrix verändern.

c) Spalte zurückgeben

```
public static int[] getSpalte(int[][] B, int j) {
    int[] spalte = new int[B.length];
    for (int i = 0; i < B.length; i++) {
        spalte[i] = B[i][j];
    }
    return spalte;
}
```

Eine Spalte liegt nicht als eigenes Teilarray vor. Deshalb wird sie aus dem Element mit Spaltenindex j jeder Zeile zusammengesetzt. Die Länge der zurückgegebenen Spalte ist die Zeilenanzahl von B.

d) Matrixprodukt

```
public static int[][] matrixProdukt(int[][] A, int[][] B) {
    int zeilen = A.length;
    int spalten = B[0].length;
    int[][] C = new int[zeilen][spalten];
    for (int i = 0; i < zeilen; i++) {
        for (int j = 0; j < spalten; j++) {
            C[i][j] = skalarProdukt(getZeile(A, i), getSpalte(B, j));
        }
    }
    return C;
}
```

Für diese übliche Matrixdarstellung wird mindestens eine Zeile von B vorausgesetzt; sonst ist dessen Spaltenanzahl aus einem `int[][]` nicht ablesbar. Ist A eine $r \times k$ -Matrix und B eine $k \times s$ -Matrix, hat C die Größe $r \times s$. Jedes `C[i][j]` ist das Skalarprodukt von Zeile i aus A und Spalte j aus B.

Beispiel: $\{\{1, 2\}, \{3, 4\}\}$ multipliziert mit $\{\{5, 6\}, \{7, 8\}\}$ ergibt $\{\{19, 22\}, \{43, 50\}\}$. Etwa ist $C[0][0] = 1 \cdot 5 + 2 \cdot 7 = 19$. Wie beim Skalarprodukt wird angenommen, dass die `int`-Ergebnisse nicht überlaufen.

Aufgabe 4 - UML-Modell für Veröffentlichungen

a) Aussagen über Vererbung und Assoziationen

Nr.	Aussage	Bewertung und Begründung
1	Jeder Herausgeber ist Nutzer.	Richtig. Editor erweitert Reviewer; Reviewer erweitert UserAcct.
2	Jeder Autor hat mindestens ein Paper.	Falsch. Am Paper-Ende der Autorenasoziation steht 0..20. Ein Autor darf noch kein Paper besitzen.
3	Paper-Konstruktoren müssen mindestens einen Autor zuordnen.	Richtig. Am Author-Ende steht 1..20; ein gültiges fertiges Paper braucht daher mindestens einen Autor.
4	Author-Konstruktoren müssen ein Paper zuordnen.	Falsch. Die für einen Autor erlaubte Paper-Anzahl beginnt bei 0.
5	Jeder Gutachter ist Editor.	Falsch. Die Vererbungsrichtung ist umgekehrt: Jeder Editor ist Reviewer, nicht jeder Reviewer Editor.

Die richtige Folge lautet richtig, falsch, richtig, falsch, falsch. Die Multiplizität am gegenüberliegenden Ende gibt an, wie viele Objekte der betreffenden Zielklasse zugeordnet sein dürfen.

b) UserAcct

Der kursiv geschriebene UML-Klassenname kennzeichnet UserAcct als abstrakte Klasse. Sie wird daher mit `abstract` deklariert, obwohl die gezeigten Methoden bereits vollständig implementiert werden können.

```
public abstract class UserAcct {
    private static int naechsteId = 0;
    private final int userId;
    private String username;
    private String emailaddress;

    public UserAcct(String username, String emailaddress) {
        userId = naechsteId++;
        this.username = username;
        this.emailaddress = emailaddress;
    }

    public int getId() { return userId; }
    public String getUsername() { return username; }
    public String getEmailaddress() { return emailaddress; }

    public void setEmailaddress(String adress) {
        emailaddress = adress;
    }
}
```

`naechsteld` ist `static` und deshalb allen Instanzen einschließlich Unterklassen gemeinsam. Die erste Instanz erhält 0, die nächste 1 usw. `final` verhindert eine nachträgliche Änderung der einmal zugewiesenen `userId`; `private` verhindert direkten fremden Zugriff. Ein Setter für `userId` wird nicht

angeboten.

Wie in einer elementaren sequenziellen Klausurlösung üblich werden keine nebenläufigen Konstruktoraufrufe und kein Umlauf des int-Zählers vorausgesetzt. Eine unbegrenzte Anzahl eindeutiger Kennungen ist mit dem im UML vorgegebenen endlichen Typ int grundsätzlich unmöglich.

c) Editor und seine Paper-Assoziation

Editor erbt von Reviewer. Der Konstruktor von Reviewer muss die beiden Strings an UserAcct weiterreichen. Die folgende Ergänzung gehört deshalb in Reviewer; dessen übrige Implementierung wird in dieser Aufgabe nicht verlangt:

```
public Reviewer(String username, String emailaddress) {
    super(username, emailaddress);
}
```

Für Editor wird ein eigenes Array für die Rolle als Herausgeber verwendet. Es ist von eventuell in Reviewer vorhandenen Papers in der Gutachterrolle zu unterscheiden. Pro Editor sind 0 bis 20 Papers erlaubt, pro Paper höchstens ein Editor. Die unten gezeigten Ergänzungen in Paper sorgen für die Gegenreferenz.

```
public class Editor extends Reviewer {
    private Paper[] papers = new Paper[20];
    private int anzahl;

    public Editor(String username, String emailaddress) {
        super(username, emailaddress);
    }

    public void assignPaper(int paperId, Reviewer rev) {
    }

    public void decidePaper(int paperId) {
    }

    public boolean addPaper(Paper paper) {
        if (paper == null || anzahl == papers.length
            || paper.getEditor() != null) {
            return false;
        }
        papers[anzahl++] = paper;
        paper.setEditorIntern(this);
        return true;
    }

    public boolean removePaper(Paper paper) {
        for (int i = 0; i < anzahl; i++) {
            if (papers[i] == paper) {
                for (int j = i; j < anzahl - 1; j++) {
                    papers[j] = papers[j + 1];
                }
                papers[--anzahl] = null;
                paper.setEditorIntern(null);
                return true;
            }
        }
        return false;
    }
}
```

```
    public Paper[] getPapers() {  
        Paper[] kopie = new Paper[anzahl];  
        for (int i = 0; i < anzahl; i++) kopie[i] = papers[i];  
        return kopie;  
    }  
}
```

Die folgenden Member ergänzen die bestehende Klasse Paper; sie bilden keine zusätzliche gleichnamige Klasse:

```
private Editor editor;  
  
public Editor getEditor() {  
    return editor;  
}  
  
// Nur von der Assoziationsverwaltung in Editor aufzurufen.  
void setEditorIntern(Editor editor) {  
    this.editor = editor;  
}
```

Beim Hinzufügen prüft Editor die Kapazität und ob das Paper bereits einen Herausgeber besitzt. Beim Entfernen wird auch die Gegenreferenz gelöscht. Soll ein Paper den Herausgeber wechseln, entfernt man es zunächst beim bisherigen Editor und fügt es anschließend beim neuen hinzu. Die Kopie in getPapers verhindert, dass Aufrufer durch direktes Verändern des Verwaltungsarrays die Grenzen umgehen.

Die Körper von assignPaper und decidePaper bleiben wie ausdrücklich verlangt leer. Der Code für Editor setzt die übrigen UML-Klassen voraus; insbesondere ist der gezeigte Reviewer-Konstruktor Teil der Vererbungskette.

d) SubjectArea mit höchstens fünf Unterthemen

```
public class SubjectArea {
    private String name;
    private SubjectArea[] subAreas = new SubjectArea[5];
    private int anzahl;

    public SubjectArea(String name) {
        this.name = name;
    }

    public String getName() { return name; }

    public SubjectArea[] getSubAreas() {
        SubjectArea[] kopie = new SubjectArea[anzahl];
        for (int i = 0; i < anzahl; i++) kopie[i] = subAreas[i];
        return kopie;
    }

    public boolean addSubArea(SubjectArea area) {
        if (area == null || anzahl == subAreas.length) return false;
        for (int i = 0; i < anzahl; i++) {
            if (subAreas[i] == area) return false;
        }
        subAreas[anzahl++] = area;
        return true;
    }

    public boolean removeSubArea(SubjectArea area) {
        for (int i = 0; i < anzahl; i++) {
            if (subAreas[i] == area) {
                for (int j = i; j < anzahl - 1; j++) {
                    subAreas[j] = subAreas[j + 1];
                }
                subAreas[--anzahl] = null;
                return true;
            }
        }
        return false;
    }
}
```

Dies ist eine Selbstassoziation: Eine SubjectArea kann andere SubjectArea-Objekte referenzieren. Das Array beschränkt die Zahl auf fünf, anzahl beschreibt die belegten Plätze. Anfangs sind keine Unterthemen vorhanden, was zur Untergrenze 0 passt. Die Methoden verhindern doppelte identische Referenzen und geben nur die tatsächlich assoziierten Unterthemen zurück. Zusätzliche Regeln über Zyklen oder die Zahl der Oberthemen stehen nicht im UML-Diagramm und werden deshalb hier nicht vorausgesetzt.

Aufgabe 5 - Ausgabe und Speicherabbild

a) Konsolenausgabe

20.0
2.0

Beim ersten Wurf ist mittleresV stets 5,0. Vier Sekunden ergeben $5,0 \cdot 4 = 20,0$.

Beim zweiten Wurf wird die angefragte Dauer 5 wegen $\text{maxZeit} = 4$ auf 4 reduziert. Der dynamische Objekttyp WurfMitWiderstand bestimmt die aufgerufene Überschreibung: $4,0 / (2,0 \cdot 4) = 0,5$. Die Weite ist daher $0,5 \cdot 4 = 2,0$. Weil double-Werte ausgegeben werden, erscheinen die Dezimalpunkte und die Nachkommastelle wie oben.

b) Speicherabbild am markierten Zeitpunkt

O1 bezeichnet das Wurf-Objekt und O2 das WurfMitWiderstand-Objekt. Der markierte Zeitpunkt wird beim zweiten Aufruf von weite erreicht, nachdem in WurfMitWiderstand.mittleresV der lokale Parameter v auf 0,5 gesetzt wurde.

Statischer Bereich	Wert
Wurf.anzahl	2

Das statische Feld beginnt automatisch bei 0. `id = anzahl++` weist zunächst den alten Wert als ID zu und erhöht danach den gemeinsamen Zähler. Es gibt keine zweite unabhängige `anzahl`-Variable in der Unterklasse.

Heapobjekt	Felder
O1, Typ Wurf	geschwindigkeit = 5.0; maxZeit = 5; id = 0
O2, Typ WurfMitWiderstand	geschwindigkeit = 4.0; maxZeit = 4; id = 1; widerstand = 2.0
ARGS, Typ String[]	Programmargumente, vom Aufruf abhängig

Aktiver Stackframe	Variable	Wert
main	args	→ ARGS
main	wurf1	→ O1
main	wurf2	→ O2, deklarierter Typ Wurf
main	erg1	20.0
Wurf.weite	this	→ O2
Wurf.weite	sekunden	4
Wurf.weite	wert	0.0
WurfMitWiderstand.mittleresV	this	→ O2
WurfMitWiderstand.mittleresV	sekunden	4
WurfMitWiderstand.mittleresV	v	0.5

Die Zuweisung an `wert` in `weite` ist noch nicht abgeschlossen: Zuerst muss der Aufruf `mittleresV` zurückkehren, erst danach wird mit `sekunden` multipliziert und zugewiesen. Daher steht `wert` noch auf 0,0. Ebenso ist die zweite Zuweisung an `erg1` in `main` noch nicht abgeschlossen; dort steht weiterhin das Ergebnis 20,0 des ersten Wurfs.

Die Änderung des Parameters `v` verändert nicht das Objektfeld `geschwindigkeit`: Primitive Parameter werden als Wertkopien übergeben. Das Objekt `O2` behält deshalb `geschwindigkeit = 4,0`. Nach den folgenden Rückgaben wird `wert` zu 2,0 und anschließend `erg1` zu 2,0.