

Lösungen zur Klausur vom 21.02.2022

Quelle: 17_EidP_Klausur_A_21.02.2022.pdf. Einführung in die Programmierung, Klausur A. Alle fünf Aufgaben sind bearbeitet. Zu Aufgabe 5 enthält das Original einen Compilerfehler; dieser wird erläutert und die vermutlich beabsichtigte Ausführung nach einer ausdrücklich angegebenen Korrektur gelöst.

Aufgabe 1 - Fließkommazahlen

Eine binäre Fließkommazahl besteht aus Vorzeichen, Exponent und Signifikand. Für normalisierte Zahlen wird die führende 1 des Signifikanden nicht eigens gespeichert. Der Exponent wird mit einem festen Versatz (Bias) abgelegt.

Für -47,5 gilt zunächst $47,5 = 32 + 8 + 4 + 2 + 1 + 1/2 = 101111,1$ (binär). Die normalisierte Darstellung ist daher:

$$-47,5 = (-1)^1 \cdot 1,011111 \text{ (binär)} \cdot 2^5$$

Das Vorzeichenbit ist 1, der tatsächliche Exponent 5 und der hinter der führenden 1 gespeicherte Bruchteil 011111 gefolgt von Nullen.

Bei Java float (32 Bit) gibt es 1 Vorzeichenbit, 8 Exponentenbits mit Bias 127 und 23 gespeicherte Bruchbits. Der Exponent wird als $5 + 127 = 132 = 1000100$ gespeichert:

Vorzeichen	Exponent	Bruchteil
1	10000100	01111100000000000000000

Bei double sind es 1 + 11 + 52 Bits und Bias 1023; der gespeicherte Exponent ist dann 1028.

-47,5 ist in beiden Formaten exakt darstellbar, weil seine binäre Darstellung endlich ist und in die verfügbaren Bits passt.

Ein endlicher Dezimalbruch muss nicht binär endlich sein. So besitzt $0,1 = 1/10$ im gekürzten Nenner einen Faktor 5. Ein endlicher Binärbruch hat dagegen einen Nenner, der eine Zweierpotenz ist. Deshalb hat 0,1 die periodische Binärdarstellung 0,0001100110011... und muss für float oder double gerundet werden. Zusätzlich begrenzt die feste Anzahl von Signifikandbits die Genauigkeit auch bei sehr großen Zahlen.

Aufgabe 2 - Teilwörter löschen

Vorausgesetzt sind nicht-null-Strings und ein nicht-null-Array mit nicht-null-Einträgen. Bei den ersten drei Methoden wird das leere Wort als Teilwort an jedem gültigen Grenzindex 0 bis `w.length()` behandelt. Für Teil iv sind nichtleere Löschwörter erforderlich: Das leere Wort könnte durch Löschen niemals vollständig aus einem String verschwinden. Diese unmögliche Eingabe wird hier durch null signalisiert.

```
public class TeilwortLöschen {
    public boolean istTeilwortAnIndex(String w, String v, int i) {
        if (i < 0 || i > w.length()
            || v.length() > w.length() - i) {
            return false;
        }
        for (int j = 0; j < v.length(); j++) {
            if (w.charAt(i + j) != v.charAt(j)) return false;
        }
        return true;
    }

    public int findeTeilwort(String w, String v) {
        for (int i = 0; i <= w.length() - v.length(); i++) {
            if (istTeilwortAnIndex(w, v, i)) return i;
        }
        return -1;
    }

    public String löscheTeilwort(String w, String v) {
        int start = findeTeilwort(w, v);
        if (start == -1) return null;
        String ergebnis = "";
        for (int i = 0; i < w.length(); i++) {
            if (i < start || i >= start + v.length()) {
                ergebnis = ergebnis + w.charAt(i);
            }
        }
        return ergebnis;
    }

    public String löscheAlleTeilworte(String w, String[] vArray) {
        for (int i = 0; i < vArray.length; i++) {
            if (vArray[i].length() == 0) return null;
        }
        boolean geaendert;
        do {
            geaendert = false;
            for (int i = 0; i < vArray.length; i++) {
                String neu = löscheTeilwort(w, vArray[i]);
                if (neu != null) {
                    w = neu;
                    geaendert = true;
                    break;
                }
            }
        } while (geaendert);
        return w;
    }
}
```

i) Teilwort am gegebenen Index

Zuerst wird geprüft, ob der gesamte Vergleich innerhalb des Strings liegt. Anschließend muss für jeden Index j des Suchwortes v gelten: $w[i + j] = v[j]$. Bei "abcdef", "bcde", 1 trifft dies zu; für "cef" nicht.

ii) Erstes Vorkommen

Die möglichen Anfangsindizes werden aufsteigend durchlaufen. Deshalb ist der erste Treffer zugleich der kleinste gültige Index. Bleibt die Suche ohne Treffer, ergibt sich -1. Für "abcdeabf" und "ab" ist das Ergebnis 0, obwohl ein zweiter Treffer bei 5 existiert.

iii) Ein Vorkommen löschen

Die Methode übernimmt alle Zeichen außerhalb des Intervalls von `start` bis `start + v.length() - 1`. Sie löscht damit genau das erste gefundene Vorkommen. Aus "abcdef" wird beim Löschen von "cdef" der String "ab". Ohne Vorkommen wird wie gefordert null zurückgegeben.

iv) Alle löschbaren Teilwörter entfernen

Nach jeder Löschung beginnt die Suche erneut beim ersten Löschwort. Das ist wichtig, weil durch das Zusammenrücken von Zeichen ein neues Vorkommen entstehen kann. Für "ababcf" mit {"abc", "bf"} entstehen nacheinander "abf" und "a".

Jede erfolgreiche Löschung verkürzt den String mindestens um ein Zeichen. Daher endet der Algorithmus. Im letzten Durchlauf wurde jedes Löschwort ohne Treffer geprüft, sodass keines mehr enthalten ist. Unterschiedliche Löschreihenfolgen können unterschiedliche Endwörter liefern; die Aufgabenstellung erlaubt dies.

Aufgabe 3 - Blockmuster

Für die Feldgrößen wird $n, m \geq 0$ vorausgesetzt; in Teil ii zusätzlich $s, t > 0$. Die Methoden sind ohne Attribute und ohne Bibliotheksmethoden implementiert.

i) Blöcke der Größe 2×2

```
public int[][] erzeugeFeldEinfach(int n, int m) {
    int[][] feld = new int[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            feld[i][j] = (i / 2 + j / 2) % 2;
        }
    }
    return feld;
}
```

Die Ganzzahldivision $i / 2$ bzw. $j / 2$ bestimmt die Blockzeile und Blockspalte. Ist deren Summe gerade, wird 0 gespeichert, andernfalls 1. Zwei benachbarte Blöcke haben dadurch verschiedene Werte. Die allgemeine Methode aus ii) wird nicht aufgerufen.

Für $n = 5, m = 6$ ergibt sich das Muster des Blatts:

```
0 0 1 1 0 0
0 0 1 1 0 0
1 1 0 0 1 1
1 1 0 0 1 1
0 0 1 1 0 0
```

ii) Blöcke der Größe $s \times t$

```
public int[][] erzeugeFeld(int n, int m, int s, int t) {
    int[][] feld = new int[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            feld[i][j] = ((i / s) % 2 + (j / t) % 2) % 2;
        }
    }
    return feld;
}
```

i / s bestimmt die Blockzeile und j / t die Blockspalte. Die getrennte Reduktion modulo 2 vermeidet einen möglichen Überlauf bei der Addition sehr großer Blockindizes. Am rechten und unteren Rand dürfen Blöcke abgeschnitten sein. Für $s = t = 2$ entsteht genau die Lösung von i).

Aufgabe 4 - Gebäude und Putzplan

Die Bezeichner entsprechen dem UML-Diagramm; insbesondere heißt die Methode `alleFensterGeputzt` mit kleinem `a`. Die Großschreibung im erläuternden Text ist inkonsistent. Jeder öffentliche Typ wird in einer eigenen gleichnamigen Datei gespeichert.

Für alle hier angegebenen Konstruktoren werden gültige Argumente vorausgesetzt: nicht-null-Referenzen, nichtnegative Flächen/Kostensätze und die jeweils genannte Anzahl verschiedener Objekte. Diese Vorbedingungen vermeiden Bibliotheksklassen für Ausnahmen.

1) Assoziationsattribute von Gebaeude

Das Gebäude benötigt ein Fenster-Array mit 1 bis 100 tatsächlich vorhandenen Fenstern sowie genau eine Putzplan-Referenz. Diese Attribute und die Methode aus Teil 4 stehen im gemeinsamen Gebaeude-Code am Ende. Das Array wird exakt auf die tatsächliche Fensteranzahl dimensioniert.

2) Putzplan und getEintrag

```
public class Putzplan {
    private Eintrag[] eintraege;

    // Ergänzung: anfang enthält 0 bis 100 vorhandene Einträge.
    public Putzplan(Eintrag[] anfang) {
        eintraege = new Eintrag[anfang.length];
        for (int i = 0; i < anfang.length; i++) {
            eintraege[i] = anfang[i];
        }
    }

    public Eintrag getEintrag(int i) {
        if (i < 0 || i >= eintraege.length) return null;
        return eintraege[i];
    }
}
```

Gefordert sind das Attribut und `getEintrag`; der gezeigte Konstruktor ist eine sinnvolle Ergänzung zum Aufbau eines gültigen Plans. Für einen Index ohne Eintrag liefert der Getter `null`. So lassen sich die maximal 100 möglichen Positionen ohne weitere Verwaltungsfunktion durchsuchen.

3) Eintrag, Fenster und Reinigungsarten

```
public abstract class Eintrag {
    private String name;
    private String firma;

    public Eintrag(String name, String firma) {
        this.name = name;
        this.firma = firma;
    }

    public String getName() { return name; }
    public String getFirma() { return firma; }
    public abstract int getKosten();
}

public class Fenster {
    private int flaeche;
```

```
public Fenster(int flaeche) {
    this.flaeche = flaeche;
}

public int getFlaeche() { return flaeche; }
}

public class Bodenreinigung extends Eintrag {
    public Bodenreinigung(String name, String firma) {
        super(name, firma);
    }

    public int getKosten() { return 1570; }
}

public class Fensterreinigung extends Eintrag {
    private int kostenProQuadratMeter;
    private Fenster[] fenster = new Fenster[100];
    private int anzahl;

    public Fensterreinigung(String name, String firma,
        int kostenProQuadratMeter, Fenster erstesFenster) {
        super(name, firma);
        this.kostenProQuadratMeter = kostenProQuadratMeter;
        fenster[0] = erstesFenster;
        anzahl = 1;
    }

    public boolean enthaelt(Fenster f) {
        for (int i = 0; i < anzahl; i++) {
            if (fenster[i] == f) return true;
        }
        return false;
    }

    public boolean fuegeHinzu(Fenster f) {
        if (f == null || anzahl == fenster.length || enthaelt(f)) {
            return false;
        }
        fenster[anzahl++] = f;
        return true;
    }

    public boolean entferne(Fenster f) {
        if (anzahl <= 1) return false;
        for (int i = 0; i < anzahl; i++) {
            if (fenster[i] == f) {
                for (int j = i; j < anzahl - 1; j++) {
                    fenster[j] = fenster[j + 1];
                }
                fenster[--anzahl] = null;
                return true;
            }
        }
        return false;
    }

    public int getAnzahlFenster() { return anzahl; }

    public Fenster getFenster(int i) {
        if (i < 0 || i >= anzahl) return null;
    }
}
```

```

        return fenster[i];
    }

    public int getKosten() {
        int summe = 0;
        for (int i = 0; i < anzahl; i++) {
            summe += fenster[i].getFlaeche() * kostenProQuadratMeter;
        }
        return summe;
    }
}

```

Eintrag ist abstract, weil getKosten je nach Reinigungsart anders bestimmt wird. Fensterreinigung benötigt mindestens ein und höchstens 100 Fenster. Das erste wird bereits im Konstruktor übergeben; die Entfernung des letzten Fensters wird abgewiesen. enthaelt verwendet Objektidentität, denn zwei verschiedene Fenster mit gleicher Fläche sind nicht dasselbe Fenster.

Die Fensterreinigung berechnet die Summe aus Fensterfläche · Kostensatz. Beispielsweise kosten Fenster mit Flächen 2 und 3 bei Kostensatz 100 insgesamt 500. Eine Bodenreinigung kostet unabhängig davon immer 1570. Der geforderte Rückgabety int setzt voraus, dass die Kostenrechnung in dessen Wertebereich bleibt.

4) Sind alle Fenster im Putzplan enthalten?

```

public class Gebaeude {
    private Fenster[] fenster;
    private Putzplan putzplan;

    // Ergänzung: 1 bis 100 Fenster und genau ein Putzplan.
    public Gebaeude(Fenster[] vorhandeneFenster, Putzplan plan) {
        fenster = new Fenster[vorhandeneFenster.length];
        for (int i = 0; i < fenster.length; i++) {
            fenster[i] = vorhandeneFenster[i];
        }
        putzplan = plan;
    }

    public boolean alleFensterGeputzt() {
        for (int i = 0; i < fenster.length; i++) {
            boolean gefunden = false;
            for (int j = 0; j < 100 && !gefunden; j++) {
                Eintrag e = putzplan.getEintrag(j);
                if (e instanceof Fensterreinigung) {
                    Fensterreinigung r = (Fensterreinigung) e;
                    if (r.enthaelt(fenster[i])) gefunden = true;
                }
            }
            if (!gefunden) return false;
        }
        return true;
    }
}

```

Für jedes Gebäudefenster wird ein passender Fensterreinigungs-Eintrag gesucht. Erst nach erfolgreichem instanceof-Test wird der Typ umgewandelt. Bodenreinigungen und nicht belegte Einträge zählen nicht. Sobald ein Fenster in keinem Eintrag vorkommt, ist false richtig; true gibt es nur, wenn alle Fenster gefunden wurden. Laut Aufgabenstellung genügt das Vorkommen im Plan als Kriterium für „geputzt“.

Aufgabe 5 - Speicherabbild und Fehler im Original

Wörtlich gegebener Code

Das Original kompiliert nicht: `b1` ist als `A` deklariert, aber `A` besitzt nur `tuWas(int)`. Der Aufruf `b1.tuWas(b1.m, b2.m)` mit zwei Argumenten ist daher bei der statischen Methodenprüfung nicht zulässig. Dass `b1` zur Laufzeit ein `B`-Objekt referenzieren würde, macht zusätzliche Überladungen nicht über eine `A`-Referenz verfügbar. Folglich erreicht das unveränderte Programm den markierten Zeitpunkt nicht und hat dort kein Speicherabbild.

Klar bezeichnete Korrektur für die beabsichtigte Auswertung

Für die folgende Auswertung wird ausschließlich der Methodenempfänger gecastet; die Feldzugriffe bleiben wie im Original:

```
erg = ((B) b1).tuWas(b1.m, b2.m);
```

Diese Korrektur ist wichtig: Würde man stattdessen schon die Deklaration in `B` `b1` ändern, änderte sich zusätzlich die Bedeutung von `b1.m` und damit das Ergebnis des Speicherbildes.

Ablauf nach dieser Korrektur

Das statische Feld `A.a` beginnt bei 2. Jeder `B(1)`-Konstruktor ruft `A(2)` auf und erhöht dabei `a` um 2; anschließend erhöht `B` es um 1. Nach zwei Objekten gilt `A.a = 8`. Beide besitzen zunächst `A.m = 2`, `B.m = 3` und `B.n = 0`.

`b1.m` verwendet wegen des deklarierten Typs `A` das Feld `A.m = 2`. `b2.m` verwendet `B.m = 3`. `B.tuWas(int,int)` wird daher mit lokalen Parametern `a = 2` und `b = 3` aufgerufen; die lokale Variable `erg` erhält 3.

`super.tuWas(a + 1)` ruft ausdrücklich `A.tuWas(3)` auf. Dort wird `O1.A.m = 3` gesetzt. Der interne Aufruf `tuWas(2)` hat jedoch kein `super` und ist dynamisch: Er ruft `B.tuWas(2)` auf. Diese Methode setzt `O1.B.m = 2 + 2 = 4` und kehrt zurück. Es gibt keine weitere Rekursion.

Statischer Speicher und Heap am Zeitpunkt 1

Bereich/Objekt	Inhalt
Statisches Feld <code>A.a</code>	8
<code>O1</code> , dynamischer Typ <code>B</code>	<code>A.m = 3</code> ; <code>B.m = 4</code> ; <code>B.n = 0</code>
<code>O2</code> , dynamischer Typ <code>B</code>	<code>A.m = 2</code> ; <code>B.m = 3</code> ; <code>B.n = 0</code>
<code>ARGS</code> , Typ <code>String[]</code>	Programmargumente; Inhalt nicht vorgegeben

Aufrufstapel am Zeitpunkt 1

Stackframe	Variable	Wert
main	args	→ ARGS
main	b1	→ O1, deklarierter Typ A
main	b2	→ O2, deklarierter Typ B
main	erg	noch nicht zugewiesen
B.tuWas(int,int)	this	→ O1
B.tuWas(int,int)	a	2
B.tuWas(int,int)	b	3
B.tuWas(int,int)	erg	3

Die Aufrufe A.tuWas(3) und B.tuWas(2) sind bereits abgeschlossen. Die lokalen Parameter namens a sind vom statischen Feld A.a unabhängig. Nach dem folgenden return erhält main.erg den Wert 3.

Zum Vergleich: Bei der alternativen Reparatur B b1 = new B(1) wären die Argumente 3 und 3; danach hätte O1 die Felder A.m = 4 und B.m = 5. Das ist eine andere, ausdrücklich geänderte Programmfassung.