Praktikum Grundlagen der Programmierung

Prof. Dr. Harald Räcke, R. Palenta, A. Reuss, S. Schulze Frielinghaus WS 2016/17

Übungsblatt 6 Abgabefrist: 05.12.2016 (vor 05:00 Uhr)

Σ Punkte: 20 | Σ Bonuspunkte: 0

Aufgabe 6.1 (P) Scoping

Wir wollen uns in dieser Aufgabe mit dem Sichtbarkeitsbereich von Variablen (*Scope*) befassen. Gegeben ist dazu die Klasse `Scoping` aus der Datei `Scoping.java`. Geben Sie für jedes Vorkommen einer Variable an, wo diese deklariert wurde. Geben Sie außerdem jeweils an, wo ihr Scope beginnt und wo er endet.

Hinweis: Sie können den Debugger Ihrer IDE verwenden, um sich globale und lokale Variablen sowie deren Werte während des Programmablaufs anzeigen zu lassen.

Lösungsvorschlag 6.1

`Scoping`:

- Zeile 2: hier deklariert, global sichtbar; *verschattet* in Zeilen 8 bis 10, 19 bis 23 und 34 bis 42
- Zeilen 3 + 33: deklariert in Zeile 3, global sichtbar; *verschattet* in Zeilen 7 bis 10, 12 bis 14, 20 bis 23 und 35 bis 42

`main`:

- Zeilen 34 + 35 (**jeweils**!): hier deklariert, lokal in `main`, sichtbar ab hier bis Ende von `main`
- Zeilen 36 + 37 + 41 + 38 (`a` auf rechter Seite) + 39 (linke Seite): deklariert in Zeile 34, Sichtbarkeit s. o.
- Zeilen 38 links + 39 rechts (`b` jeweils): deklariert in Zeile 35, Sichtbarkeit s. o.
- Zeile 40: hier deklariert, lokal in `main`, sichtbar ab hier bis Ende von `main`
- Zeile 42: deklariert in Zeile 40, Sichtbarkeit s. o.

`fred`:

- Zeile 5: deklariert in Zeile 3, global sichtbar

`georg`:

- Zeile 7: deklariert in Zeile 7, lokal in `georg`, dort von Anfang bis Ende sichtbar
- Zeilen 8 + 9: deklariert in Zeile 8, lokal in `georg`, dort ab hier bis zum Ende sichtbar

`hans`:

- Zeile 12: deklariert in Zeile 12, lokal in `hans`, dort von Anfang bis Ende sichtbar
- Zeile 13(linkes und rechtes `b`) + 14: deklariert in Zeile 12, lokal in `hans`, Sichtbarkeit s. o.

`ian`:

- Zeile 16: deklariert in Zeile 16, lokal in `ian`, dort von Anfang bis Ende sichtbar
- Zeile 17: deklariert in Zeile 17, lokal in `ian`, dort ab hier bis zum Ende sichtbar
- Zeile 19(alle vier `a`'s): deklariert in Zeile 19, sichtbar ab dort im Block der For-Schleife bis Zeile 23
- Zeile 20: deklariert in Zeile 20, sichtbar ab dort im Block der For-Schleife bis Zeile 23
- Zeile 21(`a`): wie Zeile 19
- Zeile 21(`pin`): wie Zeile 16
- Zeile 22(erstes `b` und zweites `b`): wie Zeile 20
- Zeile 22(`a`): wie Zeile 19
- Zeile 24(`a`): deklariert in Zeile 2, global sichtbar (außer verschattet)
- Zeile 24(`b`): deklariert in Zeile 3, global sichtbar (außer verschattet)
- Zeile 24(`c`): deklariert in Zeile 17, lokal in `ian`, dort ab hier bis zum Ende sichtbar

`josef`:

- Zeile 28 + 29: wie bei `hans`: alles lokal, deklariert in Zeile 28

Aufgabe 6.2 (P) Rekursion 1.0

Wir wollen uns in dieser Aufgabe mit dem Wesen(tlichen) der rekursiven Ausführung von Prozeduren befassen. Dazu wählen wir eine simple Aufgabe, die sich mittels einer For-Schleife leicht lösen lässt: die Ausgabe aller Zahlen von 1 bis n, wobei n vom Benutzer eingegeben wird. D. h. n steht zu Beginn noch nicht fest. Ein Programmgerüst ist mittels `Rekursion.java` gegeben.

In einer For-Schleife kann n z. B. einfach als Obergrenze gewählt werden, ähnlich in einer While-Schleife. Um die Aufgabe rekursiv zu lösen, benötigen wir eine Abbruchbedingung für die rekursiven Aufrufe.

- Versuchen Sie, das Verhalten der Prozedur `AusgabeMitSchleife` mittels einer rekursiven Umsetzung zu erreichen. Welche Probleme ergeben sich dabei? Wie könnte man diese lösen?
- Um zu veranschaulichen, welche Instanzen einer rekursiven Prozedur gerade aktiv sind, lassen Sie eine Ausgabe vor und nach dem rekursiven Aufruf machen. Lassen Sie dabei auch die *Rekursionstiefe* darstellen, indem die eigentliche Ausgabe um die entsprechende Anzahl Leerzeichen eingerückt wird. Die Ausgabe sollte so aussehen wie beim Aufruf der Prozedur `AusgabeTiefe`.

Lösungsvorschlag 6.2

siehe `RekursionSol.java`

- siehe die Prozedur `AusgabeMitRekursion`. Diese verwendet eine Hilfsfunktion, um korrekt die Bedingung für den Abbruch umzusetzen. D. h. es werden sowohl der Parameter n als auch die *Laufvariable* i verwaltet und beim rekursiven Aufruf übergeben. Somit kann der Effekt eines Schleifendurchlaufs simuliert werden.
- siehe `AusgabeTiefeRekursiv` (und `AMT`).

Aufgabe 6.3 (P) Wurzel berechnen

Das Heron-Verfahren (auch „babylonisches Wurzelziehen“ genannt) ist ein alter iterativer Algorithmus zur Bestimmung einer rationalen Näherung der Quadratwurzel $\sqrt{a}$ einer reellen Zahl a. Die Berechnungsvorschrift lautet:

$$x_{n+1} \quad = \quad \frac{1}{2} \cdot (x_n + \frac{a}{x_n})$$

Hierbei bezeichnet a die Zahl, deren Quadratwurzel bestimmt werden soll. Der Startwert x_0 ist eine beliebige positive Zahl.

Schreiben Sie ein *rekursives* Java-Programm `Wurzel.java`, das für eine einzulesende Zahl a die Quadratwurzel $\sqrt{a}$ näherungsweise nach dem Heron-Verfahren berechnet.

Das Verfahren soll abgebrochen und die Näherung ausgegeben werden, sobald das Quadrat der Näherung von a um weniger als einen Betrag ϵ abweicht, wobei ϵ vom Benutzer (zu Beginn) festgelegt wird.

Hinweis: Für die Implementierung könnte eine Hilfsfunktion hilfreich sein.

Lösungsvorschlag 6.3

siehe `Wurzel.java`

Aufgabe 6.4 (P) π

Gegeben sei folgende Rekursionsgleichung zur Annäherung von π:

$$f_n = \begin{cases} \frac{4}{2n+1} + f_{n-1} & n \text{ gerade} \\ \frac{-4}{2n+1} + f_{n-1} & n \text{ ungerade} \end{cases}$$

wobei $f_0 = 4$.

Schreiben Sie ein Programm, das nach Eingabe des Index n die Annäherung f_n für π *rekursiv* berechnet und ausgibt.

Lösungsvorschlag 6.4

siehe `Pi.java`

Aufgabe 6.5 (H) Rekursion, die (lat.): siehe Rekursion [13 Punkte]

Verwenden Sie für diese Aufgabe nur die erlaubten Java-Methoden (siehe Anhang)!

Lösen Sie die folgenden Teilaufgaben mittels rekursiven Methoden. Es dürfen keine Schleifen vorkommen.

Eine ganze Zahl $n \in \mathbb{Z}$ ist gerade, wenn $n = \pm 2k$ und ungerade, wenn $n = \pm 2k + 1$ für ein beliebiges $k \in \mathbb{N}_0$ gilt.

Schreiben Sie eine Klasse `Toolbox` und implementieren darin die Methode

- `public static int evenSum(int n)`, welche die Summe aller ganzen geraden Zahlen von 0 bis einschließlich `n` berechnet, falls `n` ≥ 0, oder von `n` bis 0 berechnet, falls `n` < 0 ist. Als arithmetische Operatoren sind nur die Additions- bzw. Subtraktionsoperatoren `+` und `-` erlaubt (die binären sowie unären Varianten jeweils). D. h. insbesondere der Multiplikationsoperator `*` sowie `*=` sind nicht erlaubt. Gehen Sie davon aus, dass `n` $\in [-10^4, +10^4]$.

  Beispiel: `evenSum(-8)` liefert `-20` als Ergebnis.

- `public static int multiplication(int x, int y)`, welche zwei `Integer` `x` und `y` multipliziert und das Ergebnis zurückliefert. Als arithmetische Operatoren sind nur die Additions- bzw. Subtraktionsoperatoren `+` und `-` erlaubt (die binären sowie unären Varianten jeweils). D. h. insbesondere der Multiplikationsoperator `*` sowie `*=` sind nicht erlaubt. Gehen Sie davon aus, dass `x`, `y` $\in [-10^4, +10^4]$.

- `public static void reverse(int[] m)`, welche ein `int`-Array `m` als ersten Parameter erwartet und die Einträge in dem selbigen Array wie folgt vertauscht: Sei `m` ein Array der Form $[a_1, a_2, \ldots, a_n]$, dann soll nach dem Aufruf `reverse(m)` das Array `m` die Form $[a_n, \ldots, a_2, a_1]$ haben. D. h. die (Hilfs-)Methode verändert das Array `m` und erstellt keine neuen Arrays (auch keine temporären).

- `public static int numberOfOddIntegers(int[] m)`, welche ein `int`-Array `m` als ersten Parameter erwartet und die Anzahl an ungeraden Integern aus dem Array `m` zurückgibt. Das Array `m` darf nicht verändert werden.

  Beispiel: Sei `int[] m = new int[]{4, 7, 42, 5, 1, -5, 0, -4, -3}`, dann liefert `numberOfOddIntegers(m)` als Ergebnis `5`.

- `public static int[] filterOdd(int[] m)`, welche ein `int`-Array `m` als ersten Parameter erwartet und ein `int`-Array zurückliefert. Das Ergebnis-Array enthält genau alle ungeraden Integer aus dem Array `m`. Des Weiteren spiegelt das Ergebnis-Array die Ordnung der Elemente aus dem Array `m` wider. D. h. hat das Array `m` an der Stelle i und j mit $i < j$ ungerade Integer x und y, dann finden sich die Integer x und y in dem Ergebnis-Array an Stellen l und k wieder, sodass $l < k$ gilt. Das Array `m` darf nicht verändert werden.

  Beispiel: Sei `int[] m = new int[]{4, 7, 42, 5, 1, -5, 0, -4, -3}`, dann liefert `filterOdd(m)` als Ergebnis ein Array, welches äquivalent zu folgendem ist: `new int[]{7, 5, 1, -5, -3}`.

Hinweis: Sie dürfen zusätzliche (Hilfs-)Methoden definieren, aber keine Member-Variablen. Alle Variablen, die nicht innerhalb einer Methode deklariert sind (also weder Parameter noch lokale Variablen sind), sind Member-Variablen.

Schreiben Sie Ihre Lösung in die Datei '`Toolbox.java`'.

Lösungsvorschlag 6.5

siehe `ToolboxSolution.java`

Korrekturbemerkung: Der Operator `&` ist ein bitweiser Operator und somit kein arithmetischer Operator und somit erlaubt.

evenSum:	– für positive Eingaben: 1 Punkt
	– für negative Eingaben: 1 Punkt
multiplication:	– falls ein Parameter 0 ist: 1 Punkt
	– für positive Eingaben: 1 Punkt
	– für beliebige Eingaben: 1 Punkt
reverse:	– für das leere Array: 1 Punkt
	– für nicht leere Arrays: 1 Punkt
numberOfOddIntegers:	– für das leere Array: 1 Punkt
	– für ein Array mit nur positiven Integern: 1 Punkt
	– für ein Array mit positiven und negativen Integern: 1 Punkt
filterOdd:	– für das leere Array: 1 Punkt
	– für ein Array mit nur positiven Integern: 1 Punkt
	– für ein Array mit positiven und negativen Integern: 1 Punkt

Aufgabe 6.6 (H) Matrizen-Multiplikation optimieren [7 Punkte]

Sei A eine $n \times m$ Matrix (wobei n die Anzahl an Zeilen und m die Anzahl an Spalten definiert) und B eine $m \times k$ Matrix. Dann ist das Ergebnis der Matrixmultiplikation AB eine $n \times k$ Matrix. Für die Matrixmultiplikation benötigen wir nmk viele Multiplikationen.

Möchten wir mehr als zwei Matrizen multiplizieren, können wir das dank der Assoziativität der Matrixmultiplikation auf verschiedene Weisen tun. Beispiel: Sei A eine 10×30 Matrix, B eine 30×5 Matrix und C eine 5×60 Matrix. Wir sind an der Matrix ABC interessiert. Diese kann auf zwei verschiedene Arten berechnet werden:

$$(AB)C \tag{1}$$
$$A(BC) \tag{2}$$

Bei Variante (1) benötigen wir 4500 viele Multiplikationen. Bei Variante (2) benötigen wir 27000 Multiplikationen.

Wir möchten im Folgenden ein Programm schreiben, welches für eine endliche Kette von Matrixmultiplikationen $A_1 A_2 \cdots A_k$ die minimal benötigte Anzahl an Multiplikationen ausgibt.

Sei A_j, $j \in [1, k]$ eine $n \times m$ Matrix, dann sind die Funktionen d_1 und d_2 wie folgt definiert:

$$d_1(j) \coloneqq n \qquad \text{und} \qquad d_2(j) \coloneqq m$$

Die minimal benötigte Anzahl an Multiplikationen ist gleich $f(1, k)$, wenn

$$f(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\{\, f(i,x) + f(x+1,j) + (d_1(i) \cdot d_2(x) \cdot d_2(j)) \mid i \leq x < j \,\} & \text{if } i < j \end{cases}$$

für beliebige $i, j \in \mathbb{N}$ gilt. Intuitiv bedeutet das, dass die Funktion $f(i, j)$ eine minimale Anzahl an Multiplikationen für die Matrizenmultiplikation $A_i A_{i+1} \cdots A_j$ berechnet.

Im Folgenden soll die Funktion f in Java implementiert werden. Eine $n \times m$ Matrix repräsentieren wir in Java als ein Integer-Array der Länge 2, wobei das erste Element als Eintrag die Zahl n hat und das zweite Element die Zahl m. Beispiel: `new int[]{4, 3}` repräsentiert eine 4×3 Matrix.

Eine Kette von Matrixmultiplikationen stellen wir als ein Array dar, wobei jeder Eintrag im Array eine Matrix repräsentiert. Sprich selbst wieder ein Array ist. Beispiel: Die Matrixmultiplikation ABC mit den oben definierten Matrizen A, B und C kann wie folgt repräsentiert werden: `new int[][]{{10, 30}, {30, 5}, {5, 60}}`. Der erste Eintrag repräsentiert die Matrix A, der zweite die Matrix B und der dritte die Matrix C.

Gegeben ist folgendes Grundgerüst:

```
public class MatrixMultOptimization {

    public static int f(int[][] mm) {
        return f(mm, 0, mm.length - 1);
    }

    public static int f(int[][] mm, int i, int j) {
        // TODO
    }

}
```

Ein Aufruf der Methode `public static int f(int[][] mm)` soll für ein Matrizenmultiplikations-Problem `mm` die minimale Anzahl an Multiplikationen berechnen. Beispiel:
Sei `int[][] mm = new int[][]{{10, 30}, {30, 5}, {5, 60}}`, dann soll `f(mm)` den Wert `4500` liefern.

Implementieren Sie dafür die Methode `public static int f(int[][] mm, int i, int j)`.

Schreiben Sie Ihre Lösung in die Datei '`MatrixMultOptimization.java`'.

Lösungsvorschlag 6.6

siehe `MatrixMultOptimizationSolution.java`

Korrekturbemerkung:

- Basisfall $i = j$: 1 Punkt
- Minimum einer Menge korrekt berechnet?: 2 Punkte
- Für jeden Test 1 Punkt:

- {{10, 30}, {30, 5}, {5, 60}}
- {{10, 50}, {50, 10}, {10, 50}}
- {{50, 10}, {10, 20}, {20, 5}}
- {{2, 10}, {10, 5}, {5, 100}, {100, 2}, {2, 20}}

Erlaubte Java-Methoden

MiniJava:

```
public static int readInt()
public static int readInt(String s)
public static int read()
public static int read(String s)
public static String readString()
public static String readString(String s)
public static void write(String output)
public static void write(int output)
public static int drawCard()
public static int dice()
```

returns an integer from the interval $[2, 11]$

returns an integer from the interval $[1, 6]$

String:

```
public char charAt(int index)
```

Returns the char value at the specified index. An index ranges from `0` to `length() - 1`. The first char value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

Example: `String s = "Hello Students"; char c = s.charAt(7);` saves the character `'t'` in the variable `c`.

```
public boolean isEmpty()
```

Returns `true` if, and only if, `length()` is `0`.

```
public int length()
```

Returns the length of this string. The length is equal to the number of Unicode code units in the string.

Example: `String s = "Hello Students"; int l = s.length();` saves the value `14` in the variable `l`.

```
public boolean equals(Object obj)
```

Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

Example: `String s = "Hello Students"; boolean v = s.equals("Hello Students");` here `v` is evaluated to `true`.

System:

`System.out.print(x)` prints the object `x` to the console

`System.out.println(x)` prints the object `x` to the console and terminates the line

Selbstgeschriebene Methoden, die selber nur erlaubte Methoden verwenden, sind erlaubt, sofern dies nicht explizit in der Aufgabenstellung verboten wurde. Methoden, die im gegebenen Programmgerüst definiert wurden, dürfen verwendet werden.