Projektblatt P5 (21 P)

Abgabe: Freitag 20. Oktober 2023, 12:00h

Entpacken Sie zunächst die Archiv-Datei `vorgaben-p5.zip`, in der sich die Rahmendateien für die zu lösenden Aufgaben befinden. Ergänzen Sie alle `.java` Dateien zunächst durch einen Kommentar, der Ihren Namen beinhaltet. Ergänzen Sie die Dateien dann durch Ihre Lösungen gemäß der Aufgabenstellung unten. Der abgeänderte bzw. hinzugefügte Java-Sourcecode in den Rahmendateien sollte syntaktisch richtig und vollständig formatiert sein. Alle Dateien mit der Endung `.java` sollten am Ende fehlerfrei übersetzt werden können.
Verpacken Sie die Dateien für Ihre Abgabe in einem ZIP-Archiv mit dem Namen `IhrNachname.IhrVorname.P05.zip`, die Sie auf Ilias hochladen.
Führen Sie dazu in dem Verzeichnis, in dem Sie die Dateien bearbeitet haben, folgenden Befehl auf der Kommandozeile aus:

```
zip  IhrNachname.IhrVorname.P05.zip *.java
```

Aufgabe 1: Rekursion verstehen — 5 P

Im Folgenden sind zwei Klassenmethoden gegeben, eine rekursive Methode fr und eine nicht-rekursive Methode f, die nur dazu da ist, fr für einen beliebigen ganzzahligen, positiven Wert z aufzurufen:

```
1     public static int fr(int a, int b){
2         if (a > b) {
3             return 0;
4         } else if (a == b) {
5             return b;
6         } else {
7             return a + b + fr(a+1,b-1);
8         }
9     }
10
11    public static int f(int z){
12        return fr(1,z);
13    }
```

Sie finden die Methoden auch in der Datei `Rekursion1.java`, zusammen mit einer `main`-Methode zum Testen.

(a) Geben Sie die vollständige Aufrufhierarchien für die Methode `fr` an, wenn diese von der Methode f aufgerufen wird:

- wenn f mit dem Wert $z = 4$ aufgerufen wird
- wenn f mit dem Wert $z = 7$ aufgerufen wird

Geben Sie dazu nacheinander für jeden (rekursiven) Aufruf der Methode die Parameterwerte an, mit denen die Methode aufgerufen wird, sowie das Ergebnis, welches zurückgegeben wird.

(b) Was berechnet die Methode f für einen positiven ganzzahligen Wert z? Beschreiben Sie die durch f gegebene Funktion entweder in einem Satz oder geben Sie eine passende Formel an.

(c) Was würde passieren, wenn man den rekursiven Aufruf in der Funktion fr durch eine der beiden folgenden Aufrufe ersetzen würde? Begründen Sie Ihre Antwort jeweils kurz.

- `fr(++a, --b)`
- `fr(a++, b--)`

Für Ihre Antworten ergänzen Sie den Blockkommentar am Ende der Datei `Rekursion1.java`.

Aufgabe 2: Rekursive Klassenmethoden 6 P

Implementieren Sie in der Datei `Rekursion2.java` die im Folgenden beschriebenen Klassenmethoden und testen Sie die Methoden dann mit Hilfe des Codes in der `main`-Methode der Klasse `Rekursion2`. Die Klassenmethoden sollen dabei jeweils rekursiv arbeiten.

(a) Eine einfache (wenn auch nicht sehr sinnvolle) Methode, die Wurzel einer Zahl $x >= 1$ zu approximieren, besteht darin, solange eine Zahl z aus dem Intervall $[1, x]$ zu wählen, bis das Quadrat dieser Zahl nah genug an x liegt.

Implementieren Sie hierzu in der Klassenmethode `wurzelRek` folgenden rekursiven Algorithmus. Die Methode besitzt vier Parameter: die Zahl x, einen Wert `epsilon` sowie die Grenzen a, b eines Intervalls $[a, b]$

(1) Definieren Sie zunächst eine Variable für den zu testenden Wert z und initialisieren Sie diesen mit dem Wert in der Mitte des Intervalls $[a, b]$

(2) Wenn der Absolutbetrag der Differenz zwischen z^2 und x kleiner als `epsilon` ist, geben Sie den Wert von z als Ergebnis zurück.

(3) Anderenfalls prüfen Sie ob der Wert von z^2 kleiner oder größer als x ist.

- Im ersten Fall muss z größer gewählt werden. Geben Sie daher dann das Ergebnis des rekursiven Aufrufs der Methode `wurzelRek` zurück, bei dem die Intervallgrenzen von der Mitte bis zu Ende des aktuellen Intervalls reichen.
- Im zweiten Fall muss z kleiner gewählt werden. Geben Sie daher dann das Ergebnis des rekursiven Aufrufs der Methode `wurzelRek` zurück, bei dem die Intervallgrenzen vom Beginn bis zur Mitte des aktuellen Intervalls reichen.

Für den Wert `2.0` sollte die Ausgabe dann wie folgt aussehen:

`sqrt(2.0)` ≈ `1.4142135605216026`

Sie dürfen hier die Methode `Math.abs` verwenden, aber nicht die Methode `Math.pow`.

(b) Eine n-stellige ganze Zahl $z = d_n d_{n-1} \dots d_2 d_1$ kann wie folgt aus den Ziffern berechnet werden:

$$((\dots(((d_n * 10) + d_{n-1}) * 10) + \dots + d_2) * 10) + d1$$

Beispiel: Für $n = 3$ und $d = 142$ gilt:

$$142 = (((1 * 10) + 4) * 10) + 2$$

Die Klassenmethode `reverseNumber` soll diese Eigenschaft nutzen, um die Umkehrzahl zu einer Zahl z zu berechnen, also die Zahl, die aus den gleichen Ziffern wie die Zahl z besteht, nur in umgekehrter Reihenfolge. Nullen am Ende einer Zahl gehen dabei "verloren", da führende Nullen nicht angezeigt werden.

Beispiele:

- Die Umkehrzahl zu 1357842 ist 2487531
- Die Umkehrzahl zu 300 ist 3

Die Methode `reverseNumber` hat hierzu zwei Parameter: Der Wert von z entspricht zu Beginn der umzukehrenden Zahl und muss in jedem rekursiven Aufruf um die letzte Stelle reduziert werden. Der Parameter r ist zu Beginn 0 und soll am Ende die Umkehrzahl enthalten. Dazu muss der Wert in jedem rekursiven Schritt mit 10 multipliziert und die letzte Stelle der Zahl z addiert werden. Implementieren Sie die Methode wie im Folgenden beschrieben:

- Wenn die Zahl z den Wert 0 hat, ist das Rekursionsende erreicht. Geben Sie in diesem Fall den Wert von r als Ergebnis zurück.
- In allen anderen Fällen soll das Ergebnis des rekursiven Aufrufs der Methode zurückgegeben werden. Wählen Sie die Parameterwerte beim Aufruf so, wie zuvor beschrieben.

Die Ausgabe für die Zahl $z = 123910$ sollte dann wie folgt aussehen:

```
123910 -> 19321
```

Aufgabe 3: Konstruktoren, Verkettete Listen 10 P

In dieser Aufgabe sollen Sie Methoden zur Verwaltung von zwei verschiedenen Listenklassen schreiben.
Hierzu ist in den Dateien `FlussDaten.java`, `Fluss.java`, `FlussListe.java`, `StringListe.java` und `TestListen.java` Folgendes vorgegeben:

- Die Datei Klasse `FlussDaten.java` enthält die Klasse `FlussDaten`, die Folgendes beinhaltet
  - eine Klassenvariable mit den Daten zu allen Flüssen
  - eine Klassenmethode `toInt`, die einen `String` in einen ganzzahligen Wert umwandelt (sofern der String eine solche repräsentiert)
  - eine Klassenmethode, die ein Feld mit Instanzen generiert und zurückgibt, welche alle in dem Feld `flussDaten` repräsentierten Flüsse enthält
- Die Datei `Fluss.java` enthält die Klasse `Fluss`, die Sie vom letzten Projektblatt kennen. Nur die `toString`-Methode, die einen String zurückgibt, welcher den Fluss beschreibt, ist verändert worden: Die Instanzmethode hat jetzt keinen Parameter mehr und der String, der als Ergebnis zurückgegeben wird, umfasst jetzt auch das Quellgebiet und die Mündung.
- Die Datei `FlussListe.java` beinhaltet die Klassen `FlussListenelement` und `FlussListe`, mit deren Hilfe Listen von Instanzen der Klasse `Fluss` generiert werden können, die jeweils in Listenelementen gekapselt werden. In der Klasse `FlussListe` ist hierbei Folgendes vorgegeben:
  - Der Konstruktor initialisiert eine leere Liste. Hierzu werden zwei Pseudolistenelemente `head` und `z` erzeugt, wobei `head` auf `z` zeigt und `z` auf sich selbst.
  - Die Methode `printListe` gibt alle Listenelemente in der Konsole aus.
  - Mit der Methode `insert` kann ein neuer Fluss am Kopf der Liste eingefügt werden.
  - Zwei weitere Methoden sind noch leer und müssen von Ihnen später implementiert werden.

- Die Datei `StringListe.java` beinhaltet die Klassen `StringListenelement` und `StringListe`, mit deren Hilfe Listen von `String` Objekten generiert werden können, welche jeweils in Listenelementen gekapselt werden. In der Klasse `StringListe` ist hierbei Folgendes vorgegeben:
  - Der Konstruktor initialisiert eine leere Liste (siehe Klasse `FlussListe`)
  - Die Methode `printListe` gibt alle Listenelemente in der Konsole aus.
  - Zwei weitere Methoden sind noch leer und müssen von Ihnen später impelmentiert werden.
- Die Datei `TestListen.java` beinhaltet die Klasse `TestListen`, die Sie später zum Testen Ihrer Implementierung verwenden sollen. In der `main`-Methode dieser Klasse wird zunächst ein Feld mit allen Flüssen erstellt. Diese werden dann in eine Liste (`fListe`) eingefügt.

Ergänzen Sie die Klassen `FlussListe`, `StringListe` und `TestListen` nun wie folgt:

(a) Implementieren Sie zunächst die Methode `filterNachMuendung` in der Klasse `FlussListe`. Diese Methode soll eine bereits in der Methode vorgegebene neue Instanz der Klasse `FlussListe` mit allen Flüssen füllen, deren Mündung dem als Parameter übergebenen Stringwert entspricht. Durchlaufen Sie hierzu die aktuelle Liste (auf der die Methode aufgerufen wird), vergleichen Sie die Mündungen in den einzelnen Listenelementen mit dem Wert des Methodenparameters und speichern Sie die Flüsse mit dieser Mündung in der neuen Liste. Verwenden Sie zum Vergleich von zwei Werten der Klasse `String` die Methode `equals`. Beispiel: Für zwei Strings `s1` und `s2` liefert der Aufruf `s1.equals(s2)` genau dann den Wert `true`, wenn `s1` und `s2` identische Werte haben.

(b) Implementieren Sie nun die Methode `contains` in der Klasse `StringListe`: Die Methode soll überprüfen, ob ein als Parameter übergebener String in der Liste (gekapselt in einem Listenelement) vorhanden ist. Ist dies der Fall, soll die Methode den Wert `true` zurückgeben, ansonsten den Wert `false`. Verwenden Sie auch hier für den Vergleich von String-Werten die Methode `equals`.

(c) Implementieren Sie die Methode `insertNoDuplicates` in der Klasse `StringListe`. Die Methode soll einen als Parameter übergebenen String am Ende der aktuellen Liste einfügen, sofern dieser noch nicht in der Liste gespeichert ist. Überprüfen Sie daher zunächst mit Hilfe der Methode `contains`, ob dies der Fall ist. Wenn nicht, definieren Sie sich eine Hilfsvariable vom Typ `StringListenelement`, die Sie, beginnend beim Kopf der Liste, solange auf das nächste Listenelement weitersetzen, bis das nachfolgende Listenelement das Ende der Liste markiert (also das Hilfselement auf das letzte, "echte" Listenelement zeigt). Erzeugen Sie dann ein neues Listenelement mit dem als Parameter übergebenen String und fügen Sie es nach dem Element ein, auf das die Hilfsvariable zeigt.

(d) Implementieren Sie die Methode `getMuendungen` in der Klasse `FlussListe`. Die Methode soll die vorgegebene Liste `mListe` mit allen Mündungen füllen, die in mindestens einem der Flüsse aus der aktuellen Liste vorkommen. Durchlaufen Sie daher die aktuelle Liste mit Hilfe einer Hilfsvariablen vom Typ `FlussListenelement` und nutzen Sie die Methode `insertNoDuplicates`, um die String-Werte für die Mündungen in der Liste `mListe` zu speichern.

(e) Ergänzen Sie die `main`-Methode der Klasse `TestListen` wie folgt:

- Initialisieren Sie die vorgegebene Variable `mListe` mit Hilfe der Methode `getMuendungen` mit allen in der Liste `fListe` vorkommenden Mündungen. Geben Sie die Liste (d.h. alle Elemente) in der Konsole aus.
- Anschließend soll in einer Schleife, die solange ausgeführt wird, bis der Wert der Variable `muendung` nicht mehr in der Liste `mListe` enthalten ist, Folgendes gemacht werden:
  - Mit Hilfe der Methode `filterNachMuendung` sollen aus der Liste `fListe` alle Flüsse mit dieser Mündung ausgefiltert werden. Der Inhalt der Liste soll dann in der Konsole ausgegeben werden.
  - Anschließend soll ein neuer Wert für die Variable `muendung` eingelesen werden.

Wenn Sie die Klasse `TestListen` ausführen, sollte die Ausgabe (beispielhaft für die Mündung *Nordsee*) wie folgt aussehen:

```
1. Kwango
2. Pazifischer Ozean
3. Golf von Carpentaria

... (Einträge 4 - 121 gelöscht)

122. Mittelmeer

Mündung: Nordsee
1. Elbe mit Moldau (1245 km)
    Quellgebiet: Riesengebirge, Böhmerwald
    Mündung:     Nordsee

2. Rhein (1233 km)
    Quellgebiet: Schweizer Alpen
    Mündung:     Nordsee
```