P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

http://proglang.informatik.uni-freiburg.de/teaching/java/2017/

list-operations

Interaktive Operationen auf einer Liste

Woche 04 Aufgabe 1/4

Herausgabe: 2017-05-15
Abgabe: 2017-05-26

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project `list-operations`
Package `listoperations`
Klassen

`Main`
`public static void main(String[])`

Dieses Programm erlaubt das interaktive Manipulieren von Wörterlisten. Es liest auf `stdin` zeilenweise Kommandos ein, die Listenoperationen darstellen. Die Kommandos sollen dann auf einer Liste von `String`s ausgeführt werden, soweit dies möglich ist.

Die Kommandos sollen zeilenweise eingelesen werden und entsprechen folgendem Format:

- `append` w: Fügt das Wort w zum Ende der Liste hinzu.
- ∗ `pop`: Entfernt das letzte Wort von der Liste.
- ∗ `set` i w: Überschreibt das Wort an Index i mit w.
- `remove-first` w: Entfernt das erste Vorkommen von w in der Liste.
- `remove-all` w: Entfernt alle Vorkommen von w in der Liste.
- `print`: Gibt „`::`" auf `stdout` aus, gefolgt von den Wörter der Liste. Jedem Wort soll ein Leerzeichen („ ") vorangestellt werden.
- `sort`: Sortiert die Liste.
- `reverse`: Dreht die Reihenfolge der Elemente in der Liste um.

Gültige Wörter bestehen aus den Zeichen `a-z`, `A-Z` und `0-9`.

Bei der Eingabe einer ungültigen Zeile, die also keinem der Kommandos entspricht, soll die Fehlermeldung

```
INVALID COMMAND
```

ausgegeben werden. (Danach sollen weitere Kommandos eingegeben werden können)

Falls ein Kommando auf der Liste nicht ausführbar ist, soll die Fehlermeldung

`DOES NOT COMPUTE`

ausgegeben werden. Nur Kommandos die oben mit einem Stern (*) gekennzeichnet sind, können unter Umständen diese Fehlermeldung verursachen.

Beispieleingabe 01:

```
append Hallo
append Welt
print
set 0 Hi
pop
print
append World
append Welt
append World
append Welt
print
remove-first World
print
remove-all Welt
print
reverse
print
sort
print
```

Erwartete Ausgabe 01:

```
:: Hallo Welt
:: Hi
:: Hi World Welt World Welt
:: Hi Welt World Welt
:: Hi World
:: World Hi
:: Hi World
```

Beispieleingabe 02:

```
append Tschau
print
append !
```

Erwartete Ausgabe 02:

```
:: Tschau
INVALID COMMAND
```

Beispieleingabe 03:

```
append ATTACK
print
set 2000 BOOM
```

Erwartete Ausgabe 03:

```
:: ATTACK
DOES NOT COMPUTE
```

Hinweise

- Hier gibt es eine Einführung zum Interface `java.util.List`:

  `http://docs.oracle.com/javase/tutorial/collections/interfaces/list.html`.

  Ansonsten kann auch die API Dokumentation hilfreich sein.

- `Scanner` erlaubt auch Eingaben nach einfachen Mustern mit sog. Regulären Ausdrücken zu durchsuchen. Wie immer sind die entsprechenden Methoden sind in der `Scanner`-API dokumentiert. Hier ist der Java-Syntax für reguläre Ausdrücke beschrieben:

  `https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html`

  Bei Oracle gibt es auch ein ausführliches Tutorial zu Regulären Ausdrücken in Java.

  `https://docs.oracle.com/javase/tutorial/essential/regex/`

  Die für diese Aufgabe interessantesten Kapitel sind
  - String Literals,
  - Character Classes,
  - Predefined Character Classes und
  - Quantifiers

Programmieren in Java

http://proglang.informatik.uni-freiburg.de/teaching/java/2017/

anagrams

Anagramme erkennen

Woche 04 Aufgabe 2/4

Herausgabe: 2017-05-15
Abgabe: 2017-05-26

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project `anagrams`
Package `anagrams`
Klassen

Main
`public static void main(String[])`

Ein Wort ist ein Anagramm von einem anderen Wort, wenn es aus den gleichen Zeichen in möglicherweise unterschiedlicher Reihenfolge besteht. Zum Beispiel die Anagramme von CAT sind CAT, ACT, TAC, TCA, ATC, CTA. Die Aufgabe besteht darin über `stdin` zwei Worte einzulesen und zu überprüfen ob sie Anagramme sind. Groß- und Kleinschreibung soll nicht berücksichtigt werden. Über `stdout` ist `"Anagrams"` auszugeben falls beide Worte Anagramme sind und `"Not anagrams"` wenn dies nicht der Fall ist.

Sie können davon ausgehen, dass immer genau zwei Worte eingeben werden. Worte sind hier Strings, die aus den Zeihen `A-Z` und `a-z` bestehen.

Beispieleingabe 01:

```
cAt tCa
```

Erwartete Ausgabe 01:

```
Anagrams
```

Beispieleingabe 02:

```
feger regen
```

Erwartete Ausgabe 02:

```
Not anagrams
```

Hinweise: Hier können die API von `java.lang.String` und die Funktionen in `java.util.Arrays` nützlich sein.

Programmieren in Java

http://proglang.informatik.uni-freiburg.de/teaching/java/2017/

phone-book

Ein Telephonbuch

Woche 04 Aufgabe 3/4

Herausgabe: 2017-05-15
Abgabe: 2017-05-26

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project	`phone-book`
Package	`phonebook`
Klassen	

Main
`public static void main(String[])`

Das Programm ließt zuerst eine bestimmte Menge an Telefonbucheinträgen ein, bestehend aus Name und Telefonnummer. Anschließend kann der Benutzer Telefonnummern durch Angabe von Namen durchsuchen.

Das Eingabeformat für Telefonbucheinträge ist:

- Zunächst wird die Anzahl der Einträge n als Integer gelesen.
- Dann folgen n Eingaben des Formats `<Name> <Nummer>`
- `<Name>` hat das Format `<Vorname> <Nachname>` oder `<Vorname>`. `<Vorname>` und `<Nachname>` sind nicht-leere Wörter, die nur aus Buchstaben bestehen.
- `<Nummer>` ist eine nicht-leeres Wort, dass aus den Ziffern `0-9` besteht.

Entspricht die Eingabe des Telefonbuchs nicht diesem Format, soll das Programm mit der Fehlermeldung

```
Error: cannot parse phone book.
```

(auf `stdout`) abbrechen.

Im Telefonbuch gilt immer die letzte eingegebene Nummer für einen Namen. Wenn während der Eingabe des Telefonbuchs ein Name mehrfach vorkommt, soll bei den weiteren Vorkommen beim Einlesen die Meldung

```
Warning: overwriting entry for: <Name>
```

In der Meldung soll `<Name>` durch den entsprechenden Namen ersetzt werden.

Nachdem das Telefonbuch eingelesen ist, ließt das Programm zeilenweise Namen ein und gibt die entsprechenden Telefonnummern im Format

```
<Name> = <Nummer>
```

aus. Zeilen, die nur aus *Whitespace* bestehen, sollen ignoriert werden. Befindet sich ein Name nicht im Telefonbuch, soll die Meldung

```
Not found: <Name>
```

erscheinen. Bei Zeilen, die keine korrekten Namen enthalten, soll die Meldung

```
Not a name: <Eingabe>
```

erscheinen.

Beispieleingabe 01:

```
4
uncle sam 99912222
tom
  11122222
harry
12299933
harry 12299934
uncle sam
uncle tom
t$m
harry
```

Erwartete Ausgabe 01:

```
Warning: overwriting entry for: harry
uncle sam = 99912222
Not found: uncle tom
Not a name: t$m
harry = 12299934
```

Beispieleingabe 02:

```
1
number of mike is 0123
```

Erwartete Ausgabe 02:

```
Error: cannot parse phone book.
```

Hinweise Hier können wieder Reguläre Ausdrücke nützlich sein (vlg. `list-operations`)

P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

http://proglang.informatik.uni-freiburg.de/teaching/java/2017/

unique-chars

Anzahl verschiedener Zeichen

Woche 04 Aufgabe 4/4

Herausgabe: 2017-05-15
Abgabe: 2017-05-26

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project `unique-chars`
Package `uniquechars`
Klassen

Main

`public static int uniqueChars(Map<String, Integer> cache, String input)`
 	
`public static List<Integer> allUniqueChars(`
`  Map<String, Integer> cache,`
`  List<String> input)`
 	
`public static List<Integer> allUniqueChars(List<String> input)`

Die Funktion `uniqueChars` berechnet, wie viele *unterschiedliche* Zeichen im String `input` vorkommen. Dabei benutzt sie einen *Cache* in Form einer `String`-zu-`Integer` Map. Das heißt, wenn das Ergebnis für einen bestimmten String schon in `cache` steht, soll dieses Ergebnis übernommen und nicht neu berechnet werden. (Zeichen die sich nur in der Groß-Kleinschreibung unterscheiden sind auch unterschiedlich).

Die Funktion `allUniqueChars` berechnet jeweils die Anzahl der unterschiedlichen Zeichen in den Strings der Liste `input` und gibt diese in einer Liste von `Integern` zurück. Das heißt, die Ergebnisliste hat die gleiche Länge wie `input` und an der Stelle i des Ergebnisses steht die Anzahl der unterschiedlichen Zeichen des Strings von `input.get(`i`)`.

Die Funktion `allUniqueChars` hat zwei überladenen Versionen. Die erste Version nimmt eine Map `cache` als Argument, die für alle Berechnungen verwendet werden soll, analog zum `cache` in `uniqueChars`. Die zweite Version soll intern einen frischen Cache erzeugen und dann diesen für die Berechnungen verwenden.

Diese Aufgabe soll gelöst werden, ohne unnötig Code zu duplizieren. Diese Anforderung fließt in die Bewertung der Code-Qualität ein.

Beispielaufruf 01:

```
Map<String, Integer> m = new HashMap<>();
uniqueChars(m, "Hhelloo");
```

ergibt `5`

Beispielaufruf 02:

```
Map<String, Integer> m = new HashMap<>();
m.put("hhelloo", 5);
uniqueChars(m, "hhelloo");
```

ergibt `5`

Beispielaufruf 03:

```
List<String> l = new ArrayList<String>();
l.add("hhelloo");
l.add("wwoorrld");
Map<String, Integer> m = new HashMap<String, Integer>();
allUniqueChars(m, l);
```

ergibt `[4, 5]` (als `List<Integer>`)

Hinweise

- Links zu Collections, Lists und Maps:
  - `https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html`
  - `https://docs.oracle.com/javase/tutorial/collections/interfaces/list.html`
  - `https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html`
- Der Typ einer Collection wie `List<Integer>` trägt den Typ der enthaltenen Elemente in spitzen Klammern. Soll der Inhalt von primitiven Typ sein, wie `int`, muss man diesen allerdings „ausschreiben" (`Integer`). (Wen die Details dahinter interessieren: `https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html`)
- Tipp zum Testen: Collections lassen sich mit ihrer `toString`-Methode gut darstellen und ausdrucken.