Vorlesung „Java I“ SS 04 9. Übung (21. 6. 2004)

http://www.coli.uni-sb.de/cl/courses/java-04/

---

1 Web-Crawler, Teil 2 (3 Punkte)

In dieser Aufgabe geht es darum, den HTML-Parser aus der letzten Übung zu einem richtigen Web-Crawler auszubauen. Der wesentliche fehlende Bestandteil ist ein Objekt, das die Menge der noch zu besuchenden URLs verwaltet. Da jede URL nur einmal besucht werden soll, muss das Objekt sich alle schon besuchten URLs merken und alle Versuche, eine URL mehr als einmal hinzuzufügen, einfach ignorieren.

Implementiere dazu eine generische Klasse `OneTimeQueue<E>`, die die folgenden Methoden hat:

- `void add(E x)`: fügt das Element `x` hinten an die Queue an, aber nur, wenn `x` noch nie zur Queue hinzugefügt wurde;
- `int size()`: gibt die Anzahl der Elemente zurück, die im Moment in der Queue sind;
- `E removeFirst()`: entfernt das erste Element aus der Queue und gibt es zurück.

Verwende ein Objekt der Klasse `OneTimeQueue<URL>`, um die zu besuchenden URLs in Deinem Crawler zu verwalten. Das Hauptprogramm soll zwei Kommandozeilen-Argumente akzeptieren. Das erste gibt die URL an, bei der der Crawler starten soll; das zweite gibt einen String an, der in einer URL enthalten sein muss, damit sie vom Crawler besucht wird. (Damit kann man verhindern, dass das ganze Netz durchsucht wird.)

Beispiel (Eingaben sind *kursiv* gedruckt):
`koller@cicero:~$` *java Crawler http://www.coli.uni-sb.de/cl/courses/lego-02 http://www.coli.uni-sb.de/cl/courses/lego-02*

```
Visiting: http://www.coli.uni-sb.de/cl/courses/lego-02
Visiting: http://www.coli.uni-sb.de/cl/courses/lego-02/planning.phtml
Visiting: http://www.coli.uni-sb.de/cl/courses/lego-02/grading.phtml
...
```

2 Suchen und Sortieren (3+1 Punkte)

Implementiere ein Programm, das eine deutsche Wortliste lexikographisch sortiert und ein effizientes Nachschlagen von Wörtern in der Liste erlaubt.

Unter der lexikographischen Ordnung versteht man eine Ordnungsrelation auf Strings, die wie folgt definiert ist. Um zwei Strings s und t zu vergleichen, geht man von links nach rechts durch die beiden Strings, bis ein String zu Ende ist oder man eine Position i gefunden hat, an der s und t verschiedene Zeichen enthalten. Endet einer der Strings vor dem anderen, ohne dass man verschiedene Zeichen gefunden hat, so ist dieser String kleiner. Findet man ein verschiedenes Zeichen, so ist der String mit dem kleineren Zeichen kleiner. Andernfalls sind die beiden Strings gleich.

Die lexikographische Ordnung ist zwar in der Methode `compareTo` der Klasse `String` implementiert, aber diese Methode vergleicht die beiden Zeichen einfach numerisch. Wir wollen aber für die Sortierung das Zeichen `ä` zwischen `a` und `b` einsortieren, das Zeichen `ö` zwischen `o` und `p`, usw. Das Zeichen `ß` kommt hinter `z`. Wir ignorieren Groß- und Kleinschreibung (siehe `toLowerCase()` in Klasse `Character`).

Schreibe eine Klasse, die das Interface `Comparator<String>` implementiert und diese deutsche lexikographische Ordnung berechnet. Verwende Deine Klasse, um die Wörter in der Datei `ngerman` auf der Webseite in der korrekten Reihenfolge zu sortieren und auszugeben (siehe Methode `sort` aus der Klasse `Collections` in `java.util`).

Implementiere zweitens eine effiziente Methode, die nachschaut, ob ein Wort in der Wortliste ist. Verwende dazu die Methode `binarySearch` aus der Klasse `Collections`.

Für den Bonuspunkt sollst Du die binäre Suche auf Effizienzgewinne überprüfen. Schreibe eine naive Methode zum Nachschlagen eines Eintrags in der Wortliste, die einfach von vorne nach hinten die Liste durchsucht (lineare Suche). Schreibe dann ein Hauptprogramm, das die Wortliste aus der sortierten Datei einliest und eine binäre sowie eine lineare Suche für einige Wörter durchführt. Ab einigen hundert nachzuschlagenden Wörtern aus der Mitte der Liste solltest Du einen Laufzeitunterschied merken.

3 Tic-Tac-Toe, Teil 2 (0+3 Punkte)

In der 5. Übung ging es darum, einige Klassen für ein Tic-Tac-Toe-Spiel zu schreiben. Wir fügen jetzt zu diesen Klassen ein Interface hinzu, mit dem zwei Spieler über das Netzwerk miteinander spielen können.

Jeder Spieler verwendet einen *Client*, der sich über ein Netzwerk mit einem *Server* verbindet. Der Server verwaltet den aktuellen Zustand des Spiels (in einem `IBoard`- und einem `ILogic`-Objekt); der Client ist für das Einlesen von Zügen und die Ausgabe des Spielbretts zuständig. In dieser Aufgabe geht es nur um den Client; den Server stellt Steffen zur Verfügung.

Die Netzwerkverbindung findet über einen Socket statt (siehe Klasse `Socket` in der Package `java.net`). Der geeignete Konstruktor von `Socket` bekommt den Namen des Computers, auf dem der Server läuft, sowie eine Portnummer als Argumente und baut dann eine Verbindung auf. Man kann sich dann Input- und Output-Streams für die Verbindung holen.

Client und Server tauschen Java-Objekte aus, indem sie die Klassen `ObjectInputStream` und `ObjectOutputStream` verwenden (siehe deren Dokumentation). Die einzelnen Nachrichten-Klassen stellt Steffen in einem Jar-File in der Package `games.messages` zur Verfügung. Das Protokoll für die Kommunikation sieht wie folgt aus:

(a) Sobald sich zwei Clients mit dem Server verbunden haben, verschickt der Server an jeden Client ein Objekt der Klasse `Start`. Man kann die Methoden `getWidth()` und `getHeight()` dieses Objekts verwenden, um herauszufinden, wie groß das Spielbrett ist, und dann entsprechend ein Brett-Objekt anlegen. Die `getColor()`-Methode gibt an, welche Farbe der Client spielt.

(b) Sobald jemand das Spiel gewonnen hat, verschickt der Server ein `GameOver`-Objekt, dessen `getColor()`-Methode angibt, wer gewonnen hat.

(c) Sobald jemand einen Zug ausführt, verschickt der Server an alle Clients ein `Turn`-Objekt, das den Zug (mit Farbe) angibt.

(d) Wenn der Server erwartet, dass ihm jemand einen Zug schickt, versendet er an jeden Client ein `Prompt`-Objekt. Der Client für diese Farbe muss dann einen Zug einlesen und an den Server schicken. Alle anderen Clients sollen die Nachricht ignorieren.

(e) Um einen Zug auszuführen, schickt ein Client eine `Turn`-Nachricht an den Server. Dieser antwortet mit einem `Result`-Objekt, dessen `getValid()`-Methode angibt, ob der Zug in der aktuellen Position gültig war. Der Server erwartet, dass der Client so lange Züge schickt, bis einer gültig ist, und führt diesen Zug dann aus (und verschickt seinerseits `Turn`-Objekte, usw.).

4 Konstruktion mit Typ-Parametern (0+1 Punkte)

Was haben sich die Designer von Java dabei gedacht, als sie innerhalb einer generischen Klasse mit Typ-Parameter `E` Ausdrücke der Form `new E()` verboten haben?

Abgabe bis 28. 6. 2004, 9 Uhr

(java-uebungen@coli.uni-sb.de)