Übungsaufgaben EIDI 1

Version 1.0.0

Christian Femers

6. Februar 2019

Geschätzte Zeit:
60–120 min mit IDE, 100–160 min auf Papier; je zusätzlich 80–180 min für Selectionsort

1 Synchrone Rekursion

Wir wollen eine *einfach verkettete Liste* für die Verwaltung von 2-Tupeln bzw. Objekt-paaren erstellen. Um sie für verschiedene Zwecke einsetzen zu können, soll sie parametriert werden, also Generics nutzen. Außerdem soll die Liste auch in nebenläufigen Programmen verwendet werden können und die Nutzung durch verschiedene Threads unterstützen, indem auf unnötige Synchronisation verzichtet wird. Leider haben bösartige Pinguine den Java-Compiler sabotiert, sodass dieser keine Schleifen mehr kompilieren kann, Sie müssen also ohne auskommen. Lösen Sie daher diese Aufgabe **rekursiv**; Sie dürfen **keinerlei Schleifen** verwenden.

Um bestmögliche Parallelität zu erreichen, wollen wir für die Synchronisation `java.util.concurrent.locks.ReentrantReadWriteLock` verwenden. Sie können die folgenden Methoden verwenden:

- `new ReentrantReadWriteLock()` erzeugt ein neues `ReentrantReadWriteLock`-Objekt.
- Mit `readLock().lock()` kann der ausführende Thread versuchen, das Lese-Lock zu akquirieren. Falls das Schreib-Lock *von einem anderen Thread* besetzt ist, blockiert der Aufruf die Ausführung, bis das Schreib-Lock wieder frei ist. Der Thread kann dieses Lock auch mehrfach akquirieren.
- `readLock().unlock()` gibt das Lese-Lock wieder frei. Ein Thread muss `unlock()` genau so oft aufrufen, wie er `lock()` aufgerufen hat, um das Lock freizugeben. Besitzt ein Thread das Lock nicht (mehr), wird eine `IllegalMonitorStateException` geworfen.
- Mit `writeLock().lock()` kann der ausführende Thread versuchen, das Schreib-Lock zu akquirieren. Falls das Schreib-Lock **oder** das Lese-Lock noch *von einem anderen Thread* besetzt ist, blockiert der Aufruf die Ausführung, bis das Schreib-Lock wieder frei ist. Der Thread kann dieses Lock auch mehrfach akquirieren.
- `writeLock().unlock()` verhält sich analog zu `readLock().unlock()`

Um einen möglichst großen Teil der Implementierung nach außen hin zu verstecken, sollen so wenig Methoden wie möglich öffentlich, d.h. `public` gemacht werden. Machen Sie daher nur Methoden öffentlich, wenn sie die Aufgabenstellung explizit dazu auffordert.

1.1 Die Klasse ListElement

Implementieren Sie zuerst die Klasse `ListElement`. Diese soll Element unserer verketteten Liste repräsentieren. Dafür benötigt Sie ein `ListElement`-Attribut, das auf das nächste Element zeigt,

sowie zwei Attribute vom Typ der zwei Typ-Parameter `S` und `T`, die den in dem Element gespeicherten Inhalt repräsentieren. Stellen Sie für die Elementinhalte jeweils öffentliche Getter und Setter bereit. Achten Sie darauf, dass hier *race-conditions* auftreten können; behandeln das Problem für jedes Objekt mit einem eigenen `ReentrantReadWriteLock`. Sorgen Sie dafür, das auf keines der Attribute unkontrolliert zugegriffen werden kann.

Um von der Liste aus Schreibzugriffe auf den Elementen während umfassenderen Operationen verhindern zu können, muss die Liste mit den Locks der Elemente interagieren können. Stellen Sie daher *nicht-öffentliche* Getter bereit, mit denen die Liste auf die `ReentrantReadWriteLock`s zugreifen kann. Die Methoden für das Lesen und Modifizieren des nächsten Listenelements sollen ebenfalls *nicht-öffentlich* sein.
Stellen Sie einen (oder mehrere) nicht-öffentlichen Konstruktor zur Verfügung, sodass nur die Liste neue `ListElement`-Objekte erzeugen kann und überschreiben und implementieren Sie die Methode `java.lang.Object.equals(Object)` gemäß dem `equals()`-Kontrakt[1, S. 207]:

> **Der `equals()`-Kontrakt**
>
> Vielfach benötigt man eine Prüfung auf inhaltliche Gleichheit. Dazu muss in eigenen Klassen die Methode `equals(Object)` passend überschrieben und dabei deren Kontrakt eingehalten werden. In der JLS [34] ist dazu folgende Signatur festgelegt:
>
> ```
> public boolean equals(Object obj)
> ```
>
> Diese Methode muss eine Äquivalenzrelation mit folgenden Eigenschaften realisieren:
>
> - **Null-Akzeptanz** – Für jede Referenz `x` ungleich `null` liefert `x.equals(null)` den Wert `false`.
> - **Reflexivität** – Für jede Referenz `x`, die nicht `null` ist, muss `x.equals(x)` den Wert `true` liefern.
> - **Symmetrie** – Für alle Referenzen `x` und `y` darf `x.equals(y)` nur den Wert `true` ergeben, wenn `y.equals(x)` dies auch tut.
> - **Transitivität** – Für alle Referenzen `x`, `y` und `z` gilt: Sofern `x.equals(y)` und `y.equals(z)` den Wert `true` ergeben, dann muss dies auch `x.equals(z)` tun.
> - **Konsistenz** – Für alle Referenzen `x` und `y`, die nicht `null` sind, müssen mehrmalige Aufrufe von `x.equals(y)` konsistent den Wert `true` bzw. `false` liefern.

Berücksichtigen Sie hierbei nur die Attribute, die für die "inhaltliche Gleichheit" wichtig sind.

Schreiben Sie außerdem die Methoden

- `public final void updateVal1(UnaryOperator<S> operation)` und
- `public final void updateVal2(UnaryOperator<T> operation)`.

`UnaryOperator` ist ein parametriertes, funktionales Interface mit der abstrakten Methode `T apply(T t)`, `T` ist hier der Typ-Parameter. Die `apply`-Methode von `operation` soll dabei als eine atomare bzw. synchronisierte Operation betrachtet werden. Der Operator bekommt dabei den aktuellen Wert, macht damit irgendetwas und die Rückgabe wird der neue Wert.

1.2 Die Klasse ConcurrentList

Erstellen Sie die Klasse `ConcurrentList` und implementieren Sie folgendes Interface. Nutzen Sie dafür die zuvor erstelle Klasse `ListElement`. Die Klasse soll lediglich über zwei Attribute verfügen, das Lock und das erste Listenelement. Achten Sie darauf, dass innerhalb Ihrer Implementierung weder *race-conditions* noch *dead-locks* auftreten, selbst wenn ein Nutzer die Liste

– absichtlich oder unabsichtlich – unsachgemäß benutzt. Sie müssen also alle Methoden, bei denen Parameter übergeben werden, gegen möglicherweise geworfene Exceptions absichern.

```
1  public interface List<S,T> {
2    int size();
3
4    void add(S value1, T value2);
5
6    int indexOf(ListElement<S, T> e);
7
8    ListElement<S, T> get(int index);
9
10   ListElement<S, T> remove(int index);
11
12   void forEach(Consumer<ListElement<S, T>> action);
13
14   void reverse();
15
16   void doSelectionSort(Comparator<ListElement<S,T>> comp);
17 }
```

Implementieren Sie das folgende Verhalten:

1. `size()` gibt die Anzahl der aktuell gespeicherten Elemente zurück.
2. `add(S, T)` fügt an das Ende der Liste ein neues Element mit den übergebenen Werten hinzu, `null` ist erlaubt.
3. `indexOf(ListElement<S, T>)` gibt den Index von dem `ListElement` zurück oder -1, wenn es nicht enthalten oder `null` ist. Nutzen Sie hierfür die vorher implementierte equals-Methode.
4. `get(int)` gibt das `ListElement` an dem gegebenen Index zurück.
5. `remove(int)` gibt das `ListElement` an dem gegebenen Index zurück und entfernt es aus der Liste.
6. `forEach(Consumer<ListElement<S, T>>)` führt die accept-Methode des Consumers nacheinander für alle Elemente der Liste aus.
7. `reverse()` kehrt die Reihenfolge der Elemente in der Liste um.
8. `doSelectionSort()` sortiert die Liste mit Selectionsort und benutzt dafür den übergebenen Comparator. Mit `comp.compare(o1, o2)` können zwei `ListElement`-Objekte verglichen werden. Ist der Rückgabewert positiv, gilt $o1 > o2$, ist er negativ, gilt $o1 < o2$; andernfalls $o1 \simeq o2$. Sortieren Sie die Liste aufsteigend. Achten Sie darauf, dass sich die Liste in jedem Fall in einem validen Zustand befindet. *[Achtung: sehr anspruchsvoll]*

Verwenden Sie so wenig Synchronisation wie möglich, um nicht die Parallelität zu behindern; aber genug, damit keine *race-conditions* oder *dead-locks* auftreten können. So sollen z.B. mehrere `get(int)`- und `size()`-Aufrufe gleichzeitig möglich sein. Sie können für die Liste selbst wieder ein `ReentrantReadWriteLock` verwenden und beliebig viele private Hilfsmethoden schreiben. Denken Sie daran, dass Sie die Aufgaben rekursiv lösen müssen. Stellen Sie einen öffentlichen, parameterlosen Konstruktor zur Verfügung. Werfen Sie bei der Übergabe von ungültigen Parametern passende Exceptions.

Implementieren Sie für `ConcurrentList` ebenfalls die Methode equals() gemäß dem Kontrakt.

Literatur

[1] M. Inden, *Der Weg zum Java-Profi, Konzepte und Techniken für die professionelle Java-Entwicklung*, 3., aktualisierte und überarbeitete Auflage. Heidelberg: dpunkt.verlag, 2015, xxv, 1391 Seiten, Literaturverzeichnis: Seiten 1365-1368, ISBN: 9783864902031.