Übungsaufgaben EIDI 1

Version 1.0.0

Christian Femers

8. April 2019

ACHTUNG: LÖSUNGEN

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.

Musterlösung:

```
1  package ueb_blatt_1.src;
2
3  import java.util.Objects;
4  import java.util.concurrent.locks.ReentrantReadWriteLock;
5  import java.util.function.UnaryOperator;
6
7  public final class ListElement<S, T> {
8
9      private final ReentrantReadWriteLock val1Lock;
10     private final ReentrantReadWriteLock val2Lock;
11
12     private ListElement<S, T> next;
13
14     private S val1;
15     private T val2;
16
17     ListElement(S val1, T val2) {
18         this(null, val1, val2);
19     }
```

```
20
21      ListElement(ListElement<S, T> next, S val1, T val2) {
22          this.val1Lock = new ReentrantReadWriteLock();
23          this.val2Lock = new ReentrantReadWriteLock();
24          this.next = next;
25          this.val1 = val1;
26          this.val2 = val2;
27      }
28
29      final ListElement<S, T> getNext() {
30          return next;
31      }
32
33      final void setNext(ListElement<S, T> next) {
34          this.next = next;
35      }
36
37      final ReentrantReadWriteLock getVal1Lock() {
38          return val1Lock;
39      }
40
41      final ReentrantReadWriteLock getVal2Lock() {
42          return val2Lock;
43      }
44
45      public final S getVal1() {
46          val1Lock.readLock().lock();
47          S temp = val1;
48          val1Lock.readLock().unlock();
49          return temp;
50      }
51
52      public final T getVal2() {
53          val2Lock.readLock().lock();
54          T temp = val2;
55          val2Lock.readLock().unlock();
56          return temp;
57      }
58
59      public final void setVal1(S val1) {
60          val1Lock.writeLock().lock();
61          this.val1 = val1;
62          val1Lock.writeLock().unlock();
63      }
64
65      public final void setVal2(T val2) {
66          val2Lock.writeLock().lock();
67          this.val2 = val2;
68          val2Lock.writeLock().unlock();
69      }
70
```

```
71      public final void updateVal1(UnaryOperator<S> operation) {
72          val1Lock.writeLock().lock();
73          try {
74              this.val1 = Objects.requireNonNull(operation, "Operator must not
                    ↪  be null").apply(val1);
75          } finally {
76              val1Lock.writeLock().unlock();
77          }
78      }
79
80      public final void updateVal2(UnaryOperator<T> operation) {
81          val2Lock.writeLock().lock();
82          try {
83              this.val2 = Objects.requireNonNull(operation, "Operator must not
                    ↪  be null").apply(val2);
84          } finally {
85              val2Lock.writeLock().unlock();
86          }
87      }
88
89      @Override
90      public String toString() {
91          return "(" + val1 + ", " + val2 + ")";
92      }
93
94      @Override
95      public boolean equals(Object obj) {
96          /*
97           * equals zu synchronisieren ist begrenzt sinnvoll, gemacht habe ich
    ↪  es, um
98           * wenigstens Cache-Kohärenz zu erzwingen; zu Fehlern führt es
    ↪  jedenfalls nicht,
99           * man kann es nur nicht wirklich sinnvoll einsetzten. Hier bräuchte
    ↪  es eine
100          * equals(Object o, Consumer<Boolean> callback) Methode.
101          */
102         if (this == obj)
103             return true;
104         if (!(obj instanceof ListElement))
105             return false;
106         ListElement<?, ?> other = (ListElement<?, ?>) obj;
107         this.val1Lock.readLock().lock();
108         this.val2Lock.readLock().lock();
109         other.val1Lock.readLock().lock();
110         other.val2Lock.readLock().lock();
111         try {
112             return Objects.equals(val1, other.val1) && Objects.equals(val2,
                    ↪  other.val2);
113         } finally {
114             other.val2Lock.readLock().unlock();
115             other.val1Lock.readLock().unlock();
```

```
116             this.val2Lock.readLock().unlock();
117             this.val1Lock.readLock().unlock();
118         }
119     }
120
121     @Override
122     public int hashCode() {
123         /*
124          * Die einzige legale Möglichkeit, hashCode() gemäß Kontrakt für
    ↪  komplett
125          * veränderliche Objekte zu implementieren
126          */
127         return 0;
128     }
129 }
```

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.

Musterlösung:

```
1  package ueb_blatt_1.src;
2
3  import java.util.Comparator;
4  import java.util.Objects;
5  import java.util.concurrent.locks.ReentrantReadWriteLock;
6  import java.util.function.Consumer;
7
8  public final class ConcurrentList<S, T> implements List<S, T> {
9
10     private final ReentrantReadWriteLock lock;
11
12     private ListElement<S, T> first;
13
14     public ConcurrentList() {
15         lock = new ReentrantReadWriteLock();
16     }
17
18     @Override
19     public void add(S value1, T value2) {
20         lock.writeLock().lock();
21         try {
22             if (first == null)
23                 first = new ListElement<>(value1, value2);
24             else
25                 addRecursive(first, value1, value2);
26         } finally {
27             lock.writeLock().unlock();
```

```
28          }
29      }
30
31      private void addRecursive(ListElement<S, T> e, S value1, T value2) {
32          if (e.getNext() == null)
33              e.setNext(new ListElement<>(value1, value2));
34          else
35              addRecursive(e.getNext(), value1, value2);
36      }
37
38      @Override
39      public ListElement<S, T> get(int index) {
40          if (index < 0)
41              throw new IndexOutOfBoundsException(index);
42          lock.readLock().lock();
43          try {
44              return getRecursive(first, index);
45          } finally {
46              lock.readLock().unlock();
47          }
48      }
49
50      private ListElement<S, T> getRecursive(ListElement<S, T> e, int index) {
51          if (e == null)
52              throw new IndexOutOfBoundsException();
53          if (index == 0)
54              return e;
55          return getRecursive(e.getNext(), index - 1);
56      }
57
58      @Override
59      public ListElement<S, T> remove(int index) {
60          if (index < 0)
61              throw new IndexOutOfBoundsException(index);
62          lock.writeLock().lock();
63          ListElement<S, T> res = null;
64          try {
65              if (index == 0) {
66                  if (first == null)
67                      throw new IndexOutOfBoundsException(index);
68                  res = first;
69                  first = first.getNext();
70              } else
71                  res = removeRecursive(first, index);
72          } finally {
73              lock.writeLock().unlock();
74          }
75          return res;
76      }
77
```

```
78      private ListElement<S, T> removeRecursive(ListElement<S, T> e, int index)
            ↪  {
79          ListElement<S, T> next = e.getNext();
80          if (next == null)
81              throw new IndexOutOfBoundsException();
82          if (index == 1) {
83              e.setNext(next.getNext());
84              return next;
85          }
86          return removeRecursive(e.getNext(), index - 1);
87      }
88
89      @Override
90      public int size() {
91          lock.readLock().lock();
92          try {
93              return sizeRecursive(first);
94          } finally {
95              lock.readLock().unlock();
96          }
97      }
98
99      private int sizeRecursive(ListElement<S, T> e) {
100         if (e == null)
101             return 0;
102         return 1 + sizeRecursive(e.getNext());
103     }
104
105     @Override
106     public int indexOf(ListElement<S, T> e) {
107         if (e == null)
108             return -1;
109         lock.readLock().lock();
110         try {
111             return indexOf(e, first, 0);
112         } finally {
113             lock.readLock().unlock();
114         }
115     }
116
117     private int indexOf(ListElement<S, T> e, ListElement<S, T> current, int
            ↪  index) {
118         if (current == null)
119             return -1;
120         if (current.equals(e))
121             return index;
122         return indexOf(e, current.getNext(), index + 1);
123     }
124
125     @Override
126     public void reverse() {
```

```
127         lock.writeLock().lock();
128         try {
129             if (first == null || first.getNext() == null)
130                 return;
131             ListElement<S, T> oldFirst = first;
132             first = reverseRecursive(first);
133             oldFirst.setNext(null);
134         } finally {
135             lock.writeLock().unlock();
136         }
137     }
138 
139     private ListElement<S, T> reverseRecursive(ListElement<S, T> e) {
140         ListElement<S, T> next = e.getNext();
141         if (next == null)
142             return e;
143         ListElement<S, T> res = reverseRecursive(next);
144         next.setNext(e);
145         return res;
146     }
147 
148     @Override
149     public void doSelectionSort(Comparator<ListElement<S, T>> comp) {
150         Objects.requireNonNull(comp, "Comparator must not be null");
151         lock.writeLock().lock();
152         try {
153             if (first == null || first.getNext() == null)
154                 return;
155             doSelectionSortRecursive(first, 0, comp);
156         } finally {
157             lock.writeLock().unlock();
158         }
159     }
160 
161     private void doSelectionSortRecursive(ListElement<S, T> e, int index,
        ↪  Comparator<ListElement<S, T>> comp) {
162         if (e == null)
163             return;
164         e.getVal1Lock().readLock().lock();
165         e.getVal2Lock().readLock().lock();
166         ListElement<S, T> max = null, newE = null;
167         try {
168             doSelectionSortRecursive(e.getNext(), index + 1, comp);
169             max = maximumUpTo(first, index, comp);
170             newE = get(index);
171             swap(max, newE);
172         } finally {
173             if (max != null) {
174                 max.getVal2Lock().readLock().unlock();
175                 max.getVal1Lock().readLock().unlock();
176             } else {
```

```
177                 if (newE == null)
178                     newE = get(index);
179                 newE.getVal2Lock().readLock().unlock();
180                 newE.getVal1Lock().readLock().unlock();
181             }
182         }
183     }
184 
185     private ListElement<S, T> maximumUpTo(ListElement<S, T> current, int end,
        ↪  Comparator<ListElement<S, T>> comp) {
186         if (end == 0)
187             return current;
188         ListElement<S, T> max = maximumUpTo(current.getNext(), end - 1,
            ↪  comp);
189         if (comp.compare(current, max) > 0)
190             return current;
191         return max;
192     }
193 
194     private void swap(ListElement<S, T> e1, ListElement<S, T> e2) {
195         if (e1 == e2)
196             return;
197         swap(first, e1, e2, null);
198     }
199 
200     private void swap(ListElement<S, T> current, ListElement<S, T> e1,
        ↪  ListElement<S, T> e2,
201             ListElement<S, T> previous) {
202         if (current == null)
203             return;
204         ListElement<S, T> next = current.getNext();
205         boolean e1IsCurrent = current == e1;
206         boolean e2IsCurrent = current == e2;
207         if (e1IsCurrent || e2IsCurrent) {
208             ListElement<S, T> other = e1IsCurrent ? e2 : e1;
209             if (next == other) {
210                 current.setNext(other.getNext());
211                 other.setNext(current);
212                 if (previous == null)
213                     first = other;
214                 else
215                     previous.setNext(other);
216                 return;
217             }
218             // we can only do that because e1 always comes before e2,
                ↪  removing the if allows
219             // for any order but is slower
220             if (!e2IsCurrent)
221                 swap(next, e1, e2, current);
222             if (previous == null)
223                 first = other;
```

```
            else
                previous.setNext(other);
            other.setNext(next);
        } else
            swap(next, e1, e2, current);
    }

    @Override
    public void forEach(Consumer<ListElement<S, T>> action) {
        Objects.requireNonNull(action, "Consumer must not be null");
        lock.readLock().lock();
        try {
            forEachRecursive(first, action);
        } finally {
            lock.readLock().unlock();
        }
    }

    private void forEachRecursive(ListElement<S, T> current,
        ↪  Consumer<ListElement<S, T>> action) {
        if (current == null)
            return;
        action.accept(current);
        forEachRecursive(current.getNext(), action);
    }

    @Override
    public boolean equals(Object obj) {
        /*
         * Ähnlich wie bei ListElement.
         */
        if (obj == this)
            return true;
        if (!(obj instanceof ConcurrentList))
            return false;
        ConcurrentList<?, ?> other = (ConcurrentList<?, ?>) obj;
        this.lock.readLock().lock();
        other.lock.readLock().lock();
        try {
            return equalsRecursive(first, other.first);
        } finally {
            other.lock.readLock().unlock();
            this.lock.readLock().unlock();
        }
    }

    private boolean equalsRecursive(ListElement<?, ?> e1, ListElement<?, ?>
        ↪  e2) {
        if (e1 == e2)
            return true;
        if (e1 == null || e2 == null)
```

```
273             return false;
274         if (!e1.equals(e2))
275             return false;
276         return equalsRecursive(e1.getNext(), e2.getNext());
277     }
278 
279     @Override
280     public String toString() {
281         if (first == null)
282             return "[]";
283         return "[" + toStringRecursive(first);
284     }
285 
286     private String toStringRecursive(ListElement<S, T> e) {
287         ListElement<S, T> next = e.getNext();
288         if (next == null)
289             return e.toString() + "]";
290         return e.toString() + ", " + toStringRecursive(next);
291     }
292 
293     @Override
294     public int hashCode() {
295         /*
296          * Die einzige legale Möglichkeit, hashCode() gemäß Kontrakt für
    ↪  komplett
297          * veränderliche Objekte zu implementieren
298          */
299         return 0;
300     }
301 }
```

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.