Name:

Matrikelnummer:

Bearbeitungszeit: 90 min.

Gegeben sind im Paket `threads.gegeben` die Klassen `ReadLock` und `WriteLock`, sowie die Klasse `Synchronizer` mit einem parameterlosen Konstruktor und den Methoden

```
public ReadLock acquireRead(),
public void releaseRead(ReadLock lock),
public WriteLock acquireWrite(),
public void releaseWrite(WriteLock lock).
```

Die Klasse `Synchronizer` stellt einen Synchronisationsmechanismus für Threads zur Verfügung. Ein Thread kann mit der Methode `acquireRead()` ein `ReadLock` bzw. mit der Methode `acquireWrite()` ein `WriteLock` anfordern, und diese mittels `releaseRead()` bzw. `releaseWrite()` wieder freigeben. Die Methoden `acquireRead` und `acquireWrite()` warten, bis ein entsprechendes Lock verfügbar ist. Die Klasse `Synchronizer` gewährleistet, dass gleichzeitig entweder eine beliebige Anzahl von ReadLocks vergeben ist oder genau ein `WriteLock`. Die Anforderung eines WriteLocks hat Vorrang vor späteren ReadLocks, d.h., dass nach einem `acquireWrite()` alle nachfolgenden `acquireRead()` warten müssen, bis das `acquireWrite()` ein `WriteLock` erhalten hat, und dieses auch wieder frei gegeben wurde. Nach der Erzeugung eines `Synchronizer` sind noch keine Locks vergeben.

Aufgabe a) [3 Punkte]

Schreiben Sie die `public` Klasse `threads.StringList` mit einem parameterlosen Konstruktor und den Methoden

```
public void add(String s),
public String get(int i),
```

die eine Liste von Strings verwaltet. Mit `add(s)` wird der String `s` am Ende der Liste hinzugefügt, mit `get(i)` wird der String an der Indexstelle `i` zurückgegeben (der erste Listenplatz hat den Index 0). Existiert die Indexstelle `i` nicht, soll `null` zurückgegeben werden.

Die Methoden `add(s)` und `get(i)` sollen so synchronisiert sein, dass mehrere Threads Elemente der Liste mittels `get(i)` lesen können, nur jeweils ein Thread mittels `add(s)` die Liste verändern kann, und die Liste nicht gleichzeit gelesen und verändert werden kann.

Aufgabe b) [2 Punkte]

Schreiben Sie die `public` Klasse `threads.Ausgabe` als Unterklasse der Java-Klasse `Thread` mit dem Konstruktor

```
public Ausgabe(String name, int numRepetitions, StringList list).
```

Nach Starten eines Threads `Ausgabe` soll dieser der Reihe nach die Listenelemente von `list` an den Indexstellen `i=0,...,numRepetitions-1` mittels `get(i)` lesen und mittels `System.out.println()` ausgeben. Nach jedem Lesevorgang soll der Thread den String `name+" "+i` mittels `add()` in `list` schreiben.

Aufgabe c) [2 Punkte]

Schreiben Sie die `public` Klasse `threads.Test` mit der `static` Methode

```
public static void main(String[] args),
```

die eine `StringList` und 10 Threads `Ausgabe` mit diese `StringList` und `numRepetitions=100` erzeugt. Die Namen der Threads sollen `"Thread0",...,"Thread9"` sein. Dann sollen diese Threads gestartet werden, und auf ihre Beendigung gewartet werden. Nach der Beendigung aller Threads soll `"Threads beendet"` mittels `System.out.println()` ausgegeben werden.

Aufgabe d) [3 Punkte]

Schreiben Sie die `public` Unterklasse `ExtendedSynchronizer` von `Synchronizer` mit der zusätzlichen Methode

```
public WriteLock extendToWrite(ReadLock lock),
```

die es einem Thread, der schon ein `ReadLock` besitzt, erlaubt, dieses in ein `WriteLock` umzuwandeln, ohne das `ReadLock` zuvor frei geben zu müssen. Beim Aufruf von `extendToWrite()` soll das `ReadLock` automatisch frei gegeben werden, bevor das `WriteLock` vergeben wird. **Zwischen der Freigabe des ReadLocks und der Vergabe des WriteLocks dürfen aber keine anderen WriteLocks vergeben werden.** Von den aktiven ReadLocks kann höchstens eines in ein `WriteLock` umgewandelt werden. Wird `extendToWrite()` für ein zweites `ReadLock` aufgerufen, während ein anderes `extendToWrite()` noch aktiv ist, wird zwar das zweite `ReadLock` frei gegeben, aber kein `WriteLock` vergeben, sondern `null` von der Methode zurückgegeben.

Die Bedingung, dass entweder mehrere ReadLocks oder höchstens ein `WriteLock` vergeben sein dürfen, bleibt aufrecht. Ein Aufruf `extendToWrite()` muss also warten, bis ein `WriteLock` verfügbar wird.

Hinweis: Es könnte `acquireWrite()` so überschrieben werden, dass es wartet, bis kein `extendToWrite()` aktiv ist, und ein eventuell erhaltenes `WriteLock` sofort wieder frei gibt, falls während des Wartens auf das `WriteLock` ein `extendToWrite()` aktiv wurde.