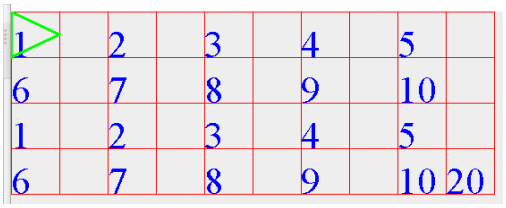
Bearbeitungszeitraum: eine Woche

a) Diese Aufgabe schliesst direkt an das vorhergehende Aufgabenblatt an. Es soll mit demselben Rahmenwerk gelöst werden.

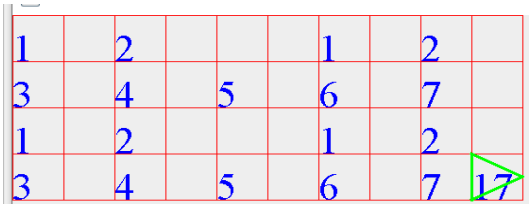
1 Aufgabe: Körner essen

1.1 Beschreibung der Aufgabe

vorher



nachher



In der obigen Abbildung sieht man das Territorium für diese Aufgabe. Auf der linken oberen Kachel befinden sich beim Start drei Sucher gleichzeitig. Die drei Dreiecke der Sucher liegen in der Abbildung übereinander und sehen aus wie ein Dreieck. Diese Sucher bewegen sich von links nach rechts. Wenn sie das Ende einer Zeile erreicht haben, machen sie ganz links in der nächsten Zeile weiter, bis sie ganz unten rechts angekommen sind. Bei jeder Kachel prüfen sie, ob die Körnerzahl größer oder gleich der Anzahl der Esser (also der Sucher) ist. Wenn das der Fall ist, dann essen sie ein Korn, d.h. `zahlKoerner` wird um eins verringert. Ansonsten wir die Körnerzahl nicht geändert. In der obigen Abbildung links ist das Territorium vor Start der Sucher abgebildet. Rechts sehen wir das Territorium, nachdem die Sucher ihre Arbeit beendet haben. Man sieht, dass in den Kacheln (0,0) und (0,2) vorher und nachher die gleiche Körnerzahl steht, da die Körnerzahl auf beiden Kacheln kleiner als die Anzahl der Sucher ist. Bei (0,4) ist dagegen nachher kein Korn mehr vorhanden, da die Körnerzahl vorher genau gleich der Zahl der Sucher war und die drei Sucher daher jeweils die Körnerzahl um eins verringert haben.

Zum Schluss werden die Körnerzahlen des Territoriums ausgegeben:

```
1   0   2   0   0   0   1   0   2   0
3   0   4   0   5   0   6   0   7   0
1   0   2   0   0   0   1   0   2   0
3   0   4   0   5   0   6   0   7   17
```

Am Ende befinden sich alle drei Sucher in der unteren rechten Ecke.

Aber es gibt dabei folgende Komplikationen:

- Was, wenn einer von den Suchern zuerst bei Kachel (0,4) eintrifft und dort bereits die Körnerzahl von 3 auf 2 erniedrigt hat, bevor der nächste Sucher eintrifft? Dann würde entgegen der Aufgabenstellung der zweite und auch der dritte Sucher die Körnerzahl nicht weiter erniedrigen. Wie können wir auch den folgenden Suchern die anfängliche Körnerzahl zeigen?
- Mehrere Threads können gleichzeitig auf das Attribut `zahlenKoerner[i][j]` der Klasse Territorium zugreifen. Dies ist problematisch, da diese Zugriffe mal lesend und mal schreibend sein können (Register Problem – siehe Vorlesung).

1.2 Verwendete Klassen / Interfaces von java.util.concurrent

1.2.1 Threadpool / ExecutorService

Siehe voriges Aufgabenblatt.

1.2.2 CyclicBarrier

Das Prinzip einer `CyclicBarrier` haben wir in der Vorlesung besprochen. Das `CyclicBarrier` Objekt aus dem `java.util.concurrent` Paket wird im Konstruktor auf eine bestimmte Anzahl von Threads eingestellt: z.B. drei. Wenn nur ein oder zwei Threads die `await`() Methode von `CyclicBarrier` aufrufen, bleiben diese Threads in der `await`() Methode hängen. Erst wenn der dritte Thread die `await`() Methode aufruft, werden alle drei Threads gleichzeitig aus der `CyclicBarrier` befreit und können hinter `await`() weiterarbeiten. Eine `CyclicBarrier` kann im Gegensatz zu einem `CountDownLatch` beliebig oft verwendet werden.

1.2.3 AtomicInteger

Wie in der Vorlesung besprochen muss der gleichzeitige Zugriff von lesenden und schreibenden Threads auf Attribute mit primitivem Datentyp schon allein aus technischen Gründen (Register Problem, 64bit Problem) mit `synchronized` Methoden erfolgen. `synchronized` verringert allerdings die Parallelität und damit die Effizienz des Programms. Zusätzlich ergibt sich durch gegenseitigen Ausschluss die Gefahr von Verklemmungen. Daher gibt es im `java.util.concurrent` Paket zu jedem primitiven Datentyp eine `Atomic` Klasse, z.B. für `int` Variablen gibt es die `AtomicInteger` Klasse. Ein Objekt dieser Klasse enthält genau einen Integer Wert, den man dem Konstruktor mitgeben kann. Das Objekt verfügt über Methoden, die ohne weiteres gleichzeitig von mehreren Threads betreten werden dürfen, z.B.

die Methode `dekrementAndGet`(). Diese Methode ermöglicht es, dass mehrere Threads gleichzeitig den `AtomicInteger` um eins erniedrigen können. Der Get Anteil ist in dieser Aufgabe unwichtig.

1.3 Durchführung

1.3.1 Die Klasse KoernerEsser

Sie sollen die Klassen des Pakets `sucher` verwenden. Dort befindet sich eine Klasse mit dem Namen `KoernerEsser`. Diese Klasse ist eine Unterklasse von `SucherImpl` und daher auch eine Unterklasse von `Thread` und `Runnable`. Ein `KoernerEsser` Objekt kann also einem `ExecutorService` zur Ausführung übergeben werden.

Die Klasse `KoernerEsser` ist unvollständig. Die Stellen, die fehlen, sind in der Klasse als Kommentare beschrieben. Ihre Aufgabe besteht daher darin, die Kommentare in Code zu verwandeln. Das, was programmiert werden soll, ist in den Kommentaren beschrieben.

1.3.2 Methoden, die vervollständigt werden müssen

- `main`
- `bearbeiteKachel`
- Konstruktor von `KoernerEsser`

1.3.3 Die Methode bearbeiteKachel

Was implementiert werden soll ist im Kommentar beschrieben.

- Mit welcher Methode von Territorium kann man die Zahl der Körner um eins verringern?
- Studieren Sie den Code der Methode. Warum müsste man diese Methode synchronized machen und warum kann man sich das trotzdem sparen?
- Wie kann man die `CyclicBarrier` verwenden, damit jeder Esser die ursprüngliche Zahl der Körner sieht?

1.3.4 Die Methode main

Was implementiert werden soll ist im Kommentar beschrieben.

Beantworten Sie folgende Frage:

- Warum muss man den Threadpool in unserer Aufgabe mit genau drei Threads bestücken? Tipp: hat mit der `CyclicBarrier` zu tun.

1.3.5 Der Konstruktor

- Erweitern Sie die Parameterliste des Konstruktors um eine `CyclicBarrier` wie im Kommentar beschrieben.

b) In der Vorlesung wurde die Wirkungsweise eines Resourcen Managers besprochen, der nach dem Prinzip „Anfordern von Betriebsmitteln mit Bedarfsanalyse“ arbeitet. In Felix gibt es ein Programm, das einen solchen Resourcen Manager simuliert. Mit Hilfe der Methode `istSicher()` der Klasse `ResourcenManager` kann geprüft werden, ob ein bestimmter Belegungszustand von Betriebsmitteln durch `Beleger` (Threads) gegen Verklemmung sicher ist. Das Programm arbeitet folgendermaßen:

- `Beleger` merken sich, wie viele Betriebsmittel sie belegen. Dort wird gleichzeitig die maximal mögliche Forderung von Betriebsmitteln durch den `Beleger` festgehalten. Eine Belegung des gesamten Systems wird also durch eine Liste der noch nicht fertigen `Beleger` (Threads) repräsentiert. Dafür gibt es eine Extra Klasse `Belegung`.
- Um festzustellen, ob eine Belegung sicher ist (Methode `istSicher()` der Klasse `Belegung`), wird ein möglicher Ablauf der Beleger (Threads) gesucht, der alle Beleger fertig werden lässt. Dazu wird bei jedem Beleger einer Belegung geprüft, ob er fertig werden kann. Ein Beleger kann fertig werden, wenn folgendes gilt: es sind noch genügend Betriebsmittel da, um seine Restforderung erfüllen zu können. Wenn kein Beleger fertig werden kann gibt die Methode `istSicher()` `false` zurück. Wenn einer fertig werden kann aber noch nicht alle fertig sind, wird ein neues `Belegung` Objekt erzeugt, in das alle Beleger hineingeschrieben werden, die bisher noch nicht fertig wurden. Das neue `Belegung` Objekt wird als Kind (`kindbelegung`) an das aktuelle `Belegung` Objekt angehängt.
- Dann wird auf dieser neuen `Belegung` wieder `istSicher()` aufgerufen (rekursiver Aufruf). Wenn diese Methode `false` zurückgibt, heißt das, dass im schlimmsten Fall kein Beleger der neuen Belegung fertig werden kann. Dann gibt es keine Ablaufreihenfolge, die ein Fertigwerden der Beleger garantiert: der ursprüngliche Belegungszustand ist nicht sicher. Liefert die `istSicher()` Methode der Kindbelegung aber `true`, so ist wegen der rekursiven Aufrufe von `istSicher()` ein Ablauf bis zum Ende mit Sicherheit möglich. Es kann `true` an die aufrufende Methode zurückgegeben werden.

I) Das Programm ist fast fertig. Sie müssen nur noch den Teil zwischen den Zeilen
```
// fertig programmieren
// Ende fertig programmieren
```
fertig programmieren. Testen Sie das Programm mit dem Beispiel aus der Vorlesung. Wenn der Beleger A die aktuelle Belegung 3 hat, ist die Belegung sicher. Wenn der Beleger A die aktuelle Belegung 4 hat, ist die Belegung unsicher. Entsprechendes wird bei Ablauf des Programms auf dem Bildschirm ausgegeben. Bei sicheren Belegungen wird auch die Reihenfolge ausgegeben, in der die Beleger A, B und C fertig werden. Die Belegungen für A, B und C werden in der main Methode festgelegt.