

Aufgabe 12.1 (P) Threads

Betrachten Sie folgendes Programm. Beschreiben Sie, welches Problem bei der Ausführung auftreten kann. Wie heißt diese Art von Problem? Wie kann das Problem für dieses Beispiel gelöst werden?

```
1  class Producer implements Runnable {
2      @Override public void run () {
3          //...
4          synchronized (Threads.lock1) {
5              synchronized (Threads.lock2) {
6                  //...
7              }
8          }
9      }
10 }
11
12 class Consumer implements Runnable {
13     @Override public void run () {
14         //...
15         synchronized (Threads.lock2) {
16             synchronized (Threads.lock1) {
17                 //...
18             }
19         }
20     }
21 }
22
23 public class Threads {
24     static Object lock1 = new Object();
25     static Object lock2 = new Object();
26
27     public static void main (String[] args) throws InterruptedException {
28         Thread t1 = new Thread(new Producer());
29         t1.start();
30         Thread t2 = new Thread(new Consumer());
31         t2.start();
32
33         t1.join();
34         t2.join();
35     }
36 }
```

Lösungsvorschlag 12.1

Es kann ein *Deadlock* auftreten, da die beiden Mutexe lock1 und lock2 in unterschiedlichen Reihenfolgen angefordert werden. So kann es sein, dass jeweils ein Thread eines der beiden Mutexe blockiert und auf den anderen wartet. Das Problem kann hier dadurch gelöst werden, die Mutexe jeweils in der gleichen Reihenfolge anzufordern.

Aufgabe 12.2 (P) Rennbedingungen

Betrachten (und testen) Sie das folgende Programm:

```
1  class Counter extends Thread {
2      private long target[]; //used as reference on value
3      private long amount; //how many times shall we count
4
5      Counter(long target[], long amount) {
6          this.target = target;
7          this.amount = amount;
8      }
9
10     public void run() {
11         for(long c = 0; c < amount; c++) {
12             long tmp = target[0];
13             tmp = tmp + 1;
14             target[0] = tmp;
15         }
16     }
17 }
18
19 public class Race {
20     public static void main(String args[]) throws InterruptedException {
21         long value[] = {0};
22         long inc = 100000000; //must be big enough!
23
24         Counter c1 = new Counter(value, inc);
25         Counter c2 = new Counter(value, inc); //same value!
26
27         /*
28         //alternative version:
29         c1.start();
30         c1.join();
31
32         c2.start();
33         c2.join();
34         */
35
36         c1.start();
37         c2.start();
38
39         c1.join();
40         c2.join();
```

```

41
42     System.out.println("counter: expected " + 2*inc + " got " +
    ↪     value[0]);
43 }
44 }

```

1. Was ist eine *race condition*?
2. Was bedeutet es, wenn ein Thread sich in einem *kritischen Abschnitt* befindet? Wozu werden kritische Abschnitte benötigt? Erläutern Sie auch, wie man kritische Abschnitte in Java definieren kann.
3. Was versteht man unter *shared data*? Was hat das Array **target** damit zu tun?
4. Warum liefert die alternative Version des Codes das erwartete Ergebnis, die normale Version aber nicht?
5. Benutzen Sie ein *Lock*, um eine Klasse **SecureCounter** zu erstellen, bei der immer das erwartete Ergebnis herauskommt.
6. Diskutieren Sie den Unterschied zwischen der Benutzung des Java-Schlüsselwortes **synchronized** und einem Lock.
7. Warum ist der **SecureCounter** deutlich langsamer als der originale?

Lösungsvorschlag 12.2

1. Race-Conditions sind Situationen, in denen das Ergebnis paralleler Ausführung entscheidend von der Ausführungsreihenfolge abhängt.
2. Ein kritischer Abschnitt ist ein Codebereich, der aufgrund von Race-Conditions nicht von mehreren Threads parallel ausgeführt werden darf. In einem kritischen Abschnitt befindet sich also jeweils nur ein Thread gleichzeitig. Kritische Abschnitte spielen beim Zugriff mehrerer Threads auf geteilte Daten eine wichtige Rolle. In Java lassen sich kritische Abschnitte durch das Schlüsselwort **synchronized** definieren. Das Schlüsselwort erwartet ein Objekt als Parameter, das als Mutex fungiert (Alternativ: nimmt einer Methode vorangestellt das aktuelle Objekt als Mutex), und einen Block von Anweisungen, innerhalb dessen sich der Thread im kritischen Abschnitt befindet.
3. Von verschiedenen Threads zugreifbare Daten (z. B. *target*)
4. Hm....
5. Siehe Lösung.
6. Siehe Kommentar in der Lösung.
7. Allgemein: Synchronisation ist teuer (d. h. kostet Zeit).

Aufgabe 12.3 (P) Parallelverordnung

In dieser Aufgabe geht es darum, alle Anordnungen (*Permutationen*) des Inhaltes eines Feldes von *int*-Zahlen parallel zu berechnen. Für ein Feld der Länge n sollen n Threads erzeugt werden, die jeweils Elemente im Feld vertauschen und die resultierenden Anordnungen ausgeben. Jedem der Threads soll dazu eine andere Teilmenge von Anordnungen zugewiesen werden, sodass die Threads jeweils unterschiedliche Ausgaben erzeugen und zusammen alle möglichen Anordnungen abdecken. Alle Threads sollen gleich viele Anordnungen ausgeben und es sollen keine Anordnungen doppelt ausgegeben werden.

1. Beschreiben Sie Ihre Herangehensweise für eine Implementierung.
2. Vervollständigen Sie das gegebene Programmgerüst durch Ihre Implementierung.

Hinweis: Sie dürfen davon ausgehen, dass alle Zahlen im Feld verschiedene Werte haben.

Hinweis: Das Feld darf für die verschiedenen Threads kopiert werden. Verwenden Sie hierzu `System.arraycopy(from, fromStartIndex, to, toStartIndex, length)`. Der dargestellte Aufruf kopiert `length` viele Elemente eines Feldes `from` ab Index `fromStartIndex` in ein Feld `to` ab Index `toStartIndex`.

Hinweis: Sie können ein Feld `numbers` mittels `System.out.println(Arrays.toString(numbers));` ausgeben.

```
1  import java.util.Arrays;
2
3  public class Parallelverordnung implements Runnable {
4      private int[] numbers;
5
6      public Parallelverordnung (int[] numbers) {
7          this.numbers = numbers;
8      }
9
10
11     public static void main (String[] args) {
12         //readArray() muss nicht von Ihnen implementiert werden!
13         int[] numbers = readArray();
14
15     }
16 } // Ende der Klasse Parallelverordnung
```

Lösungsvorschlag 12.3

1. Da es genau n Threads für n Elemente gibt, können wir genau ein Element pro Thread fixieren (z.B. das erste Element). Wir tauschen also für den i -ten Thread das i -te Element an die erste Position und lassen den Thread alle $i - 1$ vielen Positionen ab Position 1 permutieren und ausgeben. Die Threads dürfen natürlich nicht alle in der selben Suppe rühren, wir kopieren das Feld also für jeden Thread. Da jeder

Thread die gleiche Menge an Elementen permutiert, erzeugt jeder Thread die gleiche Menge an Ausgaben. Da jeder Thread ein anderes Element fixiert an der ersten Position hat, werden keine Ausgaben doppelt erzeugt. Da jedes Element bei einem Thread an der ersten Position verweilt, werden alle Permutationen ausgegeben.

Es bleibt noch zu erklären, wie ein Thread vorgeht, um seine beweglichen Elemente zu permutieren. Wir erkennen hierbei, dass es sich um ein rekursives Problem handelt. Nehmen wir an, wir müssen einen Bereich ab Position i des Feldes permutieren (zu Beginn gilt $i = 1$). Wir tauschen jeweils eines der Elemente im Bereich ab Position i des Feldes an Position i des Feldes und permutieren dann alle Elemente im Bereich ab $i + 1$. Anschließend tauschen wir das Element zurück und tauschen das nächste Element an Position i und so weiter. Wir geben das Feld als neue Permutation aus, es keinen rekursiven Aufruf gibt, wir also ab Position $length - 1$ zu permutieren haben.

Die Lösung befindet sich in der Datei `ParallelverordnungSolution.java`.

Aufgabe 12.4 (P) Map

Ein *Map* wendet eine Funktion auf jedes Element in einem Array an und speichert die Resultate in einem neuen Array. Die Ordnung der Elemente aus dem ursprünglichen Array spiegelt sich in den Elementen des resultierenden Arrays wieder. Beispiel: Seien die Funktion $f: \mathbb{Z} \rightarrow \mathbb{N}$ und ein Array $[i_1, \dots, i_k]$ mit Elementen aus \mathbb{Z} gegeben. Das resultierende Map ist dann gegeben als $[f(i_1), \dots, f(i_k)]$, wobei die Elemente alle aus \mathbb{N} kommen.

Im Folgenden wollen wir solch ein Map implementieren. Da wir davon ausgehen, dass unsere Funktion f sehr teuer ist, wollen wir das Map *parallel* implementieren. Dafür zerlegen wir ein Array der Länge k in $n \in \mathbb{N}$ viele Bereiche. Dabei gilt, dass $k \% n$ viele Bereiche die Größe $\lfloor k/n \rfloor + 1$ haben und alle übrigen Bereiche die Größe $\lfloor k/n \rfloor$. Jeder Bereich soll von einem Thread bearbeitet werden.

Für die Implementierung gehen wir wie folgt vor. Wir definieren ein Interface `Fun<T, R>`, welches die Schnittstelle zu einer (unären) Funktion $f: T \rightarrow R$ beschreibt:

```
1 public interface Fun<T, R> {  
2     public R apply(T x);  
3 }
```

Schreiben Sie eine öffentliche Klasse `Map`, welche die Methode

```
1 public static <T, R> void map(Fun<T, R> f, T[] a, R[] b, int n)  
2     throws InterruptedException
```

bereitstellt. Die Methode `map` erstellt n viele Threads. Jedem Thread wird ein unterschiedlicher Bereich aus dem Array `a` zugeteilt, auf dem ein jeweiliger Thread die Funktion `f` auf die Elemente anwendet. Wenn die Methode zurückkehrt muss folgendes gelten: $f(a[i]) = b[i]$ für alle validen Indizes i . Des Weiteren müssen dann alle in der Methode `map` gestarteten Threads beendet sein. Überlegen Sie sich, mit welchen nicht

sinnvoll behandelbaren bzw. ungültigen Parametern die Methode `map` aufgerufen werden kann und fangen Sie diese Fälle über eine `IllegalArgumentException` ab. D.h. die Methode `map` muss für alle möglichen Parameter das korrekte Ergebnis liefern oder eine `IllegalArgumentException` werfen. Überlegen Sie sich, wie Sie die Resultate von den einzelnen Threads in der Methode `map` akkumulieren können.

Schreiben Sie eine Klasse `IntToString` welche das Interface `Fun<Integer, String>` implementiert. Die Methode `apply` soll einen Integer entgegennehmen und die jeweilige Stringrepräsentation des Integers zurückgeben. Testen Sie Ihre parallele Map-Implementierung mit der Funktion.

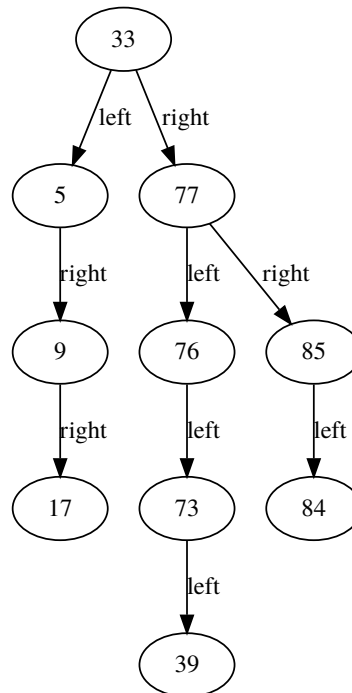
Hinweise:

- Diese Fälle müssen mittels einer Exception behandelt werden:
`f == null || a == null || b == null || a.length > b.length || n < 0`
- Der Fall `a.length < b.length` darf auch ignoriert werden.
- Der Fall `n > a.length` darf auch mittels einer Exception gelöst werden.

Die Hausaufgabenabgabe erfolgt über Moodle. Bitte geben Sie Ihren Code als UTF8-kodierte (ohne BOM) Textdatei(en) mit der Dateiendung `.java` ab. Geben Sie **keine** Projektdateien Ihrer Entwicklungsumgebung ab. Geben Sie **keinen** kompilierten Code ab (`.class`-Dateien). Wenn Sie Ihre Dateien als Archiv abgeben möchten, verwenden Sie bitte ausschließlich `.zip`-Dateien. Sie dürfen Packages verwenden. Achten Sie darauf, dass Ihr Code kompiliert. Bitte vermerken Sie aus Datenschutzgründen nicht Ihren Namen oder Ihre Matrikelnummer im Code. Hausaufgaben, die sich nicht im vorgegebenen Format befinden, werden nur mit Punktabzug oder gar nicht bewertet.

Aufgabe 12.5 (H) Suchtbäume

[7 Punkte]



In dieser Aufgabe geht es darum, binäre Suchbäume für generisch vergleichbare Objekte zu implementieren, die Zugriffoperationen aus verschiedenen Threads erlauben. Sie finden eine Vorlage für die Implementierung in der Klasse `Suchtbaum`. Es sollen folgende Methoden implementiert werden:

1. `public void insert(T element)`: Fügt ein Element in den Baum ein; ist ein gleich großes Element schon im Baum vorhanden, so soll eine `RuntimeException` erzeugt werden.
2. `public boolean contains(T element)`: Testet, ob ein bestimmtes Element im Baum vorhanden ist
3. `public void remove(T element)`: Entfernt ein Element aus dem Baum; ist das Element nicht im Baum vorhanden, so soll eine `RuntimeException` erzeugt werden.
4. `public String toString()`: Wandelt den Baum in das Graphviz-Dot-Format um

Alle Operationen erhalten natürlich die übliche Suchbauminvariante. Die `remove`-Methode sucht zunächst den Knoten k , der das Element enthält, welches entfernt werden muss. Nun unterscheidet sie drei Fälle, die hier als Hilfestellung skizziert sind:

1. Hat k keine Kinder, so wird k einfach entfernt.
2. Hat k genau ein Kind, so wird der Knoten k durch den Kindknoten ersetzt.
3. Hat k zwei Kinder, entfernen wir das größte Element e_g des linken Teilbaums; dabei tritt nur Fall 1 oder 2 auf, da das größte Element keinen rechten Nachfolger haben kann. Wir tauschen anschließend das Element, welches wir entfernen möchten, mit e_g aus.

Die `toString`-Methode soll den Baum in das Graphviz-Dot-Format¹ überführen. Graphen im Dot-Format lassen sich sehr leicht visualisieren, z.B. unter <https://dreampuf.github.io/GraphvizOnline/>. Der oben gezeigte Baum sieht im Dot-Format wie folgt aus:

```
1 digraph G {
2   33;
3   33 -> 5 [label=left];
4   5;
5   5 -> 9 [label=right];
6   9;
7   9 -> 17 [label=right];
8   17;
9   33 -> 77 [label=right];
10  77;
11  77 -> 76 [label=left];
12  76;
13  76 -> 73 [label=left];
14  73;
15  73 -> 39 [label=left];
16  39;
17  77 -> 85 [label=right];
18  85;
19  85 -> 84 [label=left];
20  84;
21 }
```

Der Baum soll es erlauben, parallel von mehreren Threads zugegriffen zu werden. Es soll dabei eine beliebige Anzahl von parallelen Operationen stattfinden können, die lediglich Daten aus dem Baum lesen (dies betrifft die Methoden `contains()` und `toString()`). Möchte ein Thread dagegen den Baum ändern, so muss er zunächst warten, bis alle lesen- und schreibenden Zugriffe abgeschlossen sind. Wir nehmen an, dass es weitaus mehr lesende als schreibende Threads gibt. Weitere lesende Zugriffe dürfen daher nun nicht mehr beginnen, weil der schreibende Thread sonst unter Umständen sehr lange warten müsste (vgl. *Verhungern von Threads*), bis er an der Reihe ist. Sind alle konkurrierenden Operationen abgeschlossen, kann der schreibende Thread seine Operation ausführen. Anschließend werden lesende oder weitere schreibende Operationen wieder möglich.

¹<https://www.graphviz.org/>

Achten Sie bei Ihrer Implementierung auf Exceptions. Es muss insbesondere **in jedem Fall** sichergestellt sein, dass ein Thread, der andere Threads blockiert, diese Blockade bei Verlassen der Klasse wieder auflöst. Lesen Sie sich in diesem Kontext die Java-Dokumentation zum Schlüsselwort **finally** durch².

Gehen Sie bei Ihrer Implementierung wie folgt vor:

1. Implementieren Sie die Methoden des Graphen ohne Threading; implementieren Sie die **remove**-Methode zuletzt, da diese komplizierter als die anderen Methoden ist. Nutzen Sie die **toString**-Methode zum Debuggen.
2. Erweitern Sie die Implementierung um die Möglichkeit von parallelen Zugriffen.
3. Prüfen Sie, ob Ihre Implementierung Blockaden immer auch wieder auflöst, auch im Fall von Exceptions.
4. Prüfen Sie, ob Ihre Implementierung sicherstellt, dass keine weiteren Lese-Operationen gestartet werden, sobald eine Schreiboperation auf den Zugriff auf die Datenstruktur wartet.
5. Schreiben Sie ein Hauptprogramm zum Testen Ihrer Implementierung!

Hinweis: Sie dürfen die gegebenen Methoden – wo nötig – um **throws**-Deklarationen erweitern.

Hinweis: Tests für den Baum ohne Threads finden Sie in der Klasse **SuchtbaumTest**.

Lösungsvorschlag 12.5

Punkteverteilung:

1. **insert()**: 1 Punkte
2. **contains()**: 0.5 Punkte
3. **remove()**: 1.5 Punkte
4. **toString()**: 0.5 Punkte
5. Threading (Synchronisation im Suchtbaum und Tests): 3.5 Punkte

Korrekturhinweise:

- Alles aus der Java-Standardbibliothek ist erlaubt.
- Keine Starvation-Verhinderung: 1.5 Punkte Abzug
- Synchronisation in den Tests: 1 Punkt Abzug
- Nutzung von **ReentrantReadWriteLock** im *Non-fair mode (default)* beachtet Starvation nicht und ist daher falsch \rightsquigarrow 1.5 Punkte Abzug

²<https://docs.oracle.com/javase/tutorial/essential/exceptions/finally.html>

- Nutzung von `ReentrantReadWriteLock` im *Fair mode* ist ebenfalls falsch, aber sehr subtil (wird von den Tests nicht abgedeckt). Folgende Sequenz von Ereignissen wird falsch gehandhabt:

1. Writer W1 erreicht den Baum und beginnt mit dem Schreiben.
2. Reader R erreicht den Baum und muss warten.
3. Writer W2 erreicht den Baum und muss warten.
4. W1 schließt den Schreibvorgang ab. Nach *fairem* Scheduling kommt nun R dran und darf lesen. Nach der Angabe haben Writer jedoch absolute Priorität und daher kommt stattdessen W2 an die Reihen.






Es gibt 1 Punkt Abzug. Leute, die es statt mit dem Java-API selbst versucht haben und einen ähnlichen Bug haben, erhalten 0.5 Punkte Abzug.

Aufgabe 12.6 (H) Paralleluine

[8 Punkte]

Die Pinguine sind zurück. Wir wollen in dieser Aufgabe eine Pinguinkolonie simulieren. Die Kolonie besteht aus einem zweidimensionalen Spielfeld, das graphisch angezeigt wird. Jeder Pinguin wird dabei durch einen Thread repräsentiert. Wir nennen sie daher auch Paralleluine. Ein einzelnes Feld im Spielfeld ist entweder mit Eis oder mit Wasser bedeckt (unveränderlich) und beherbergt entweder einen Paralleluin oder nicht. Die folgenden Abschnitte enthalten eine Beschreibung der Ablaufregeln sowie eine Beschreibung der Vorgaben, die bei der Umsetzung einzuhalten sind. Ein Programmgerüst ist gegeben und muss verwendet werden.

Je nach Alter, Geschlecht und Brütstatus lassen Paralleluine sich in fünf Kategorien einteilen (siehe Tabelle).

Bezeichnung	Symbol	Konstante (Zeichnen)	Geschlecht	Alter	brütend?
Frauin		FRAUIN	W	>25	nein
Mannuin		MANNUIN	M	>25	nein
Brütuin		SCHWANGUIN	M/W	>25	ja
Kleuin		KLEINUIN	M	≤25	nein
Kleuinin		KLEINUININ	W	≤25	nein

Die Klassen `Main` und `GUI` dürfen nicht geändert werden. Die Klasse `Main` enthält das Hauptprogramm. Die Klasse `GUI` bietet folgende Methoden:

- `protected void generateAntarctic(int[] [] landscape, Penguin[] [] placed, boolean standard)`:
Stellt den Anfangszustand her. In `landscape` werden die Konstanten zum Zeichnen, in `placed` die Pinguin-Objekte eingetragen. Der Parameter `standard` gibt an, ob der für die Spielfeldgröße vorgesehene Standard verwendet werden soll oder ob die Pinguine zufällig ausgewählt werden sollen. Die beiden Arrays müssen in ihrer Größe übereinstimmen.

- `protected static void setForeground(int[] [] where, int x, int y, int fg)`:
Setzt im übergebenen Array an der Stelle [x][y] den Vordergrund auf den übergebenen Wert. Zusätzlich zu den Werten für Pinguine aus der Tabelle gibt es die Konstante NIXUIN, die ein unbesetztes Feld spezifiziert.
- `public static boolean ocean(int[] [] where, int x, int y)`:
Informiert den Aufrufer darüber, ob an dieser Stelle Wasser (`true`) oder Eis ist.

In den Klassen `Penguin` und `Colony` ist die Implementierung zu vervollständigen, so dass die nachfolgend beschriebenen Regeln umgesetzt werden. Ein Paralleluin kann folgende Aktionen ausführen:

1. Warten: Nach jeder anderen Aktion wartet ein Paralleluin zufällige 200 bis 500 Millisekunden.
2. Bewegen: Ist das Feld direkt links (bzw. rechts bzw. oben bzw. unten, jeweils 25% Wahrscheinlichkeit, es auszuwählen) neben ihm frei, kann der Paralleluin sich dort hin bewegen. Diese Bewegung muss allerdings synchronisiert werden, um Race-Conditions und Deadlocks zu vermeiden. Details hierzu sind weiter unten aufgeführt.
3. Brüten: Befindet sich ein nicht brütender Mannuin direkt links neben einer nicht brütenden Frauin, und befinden sich beide auf dem Eis, so legt die Frauin mit 5% Wahrscheinlichkeit fest, dass sie statt einer Bewegung entweder selber einen Brutvorgang startet oder (50% Wahrscheinlichkeit) den benachbarten Mannuin zu einem Brutvorgang veranlasst. Dieser ist verpflichtet, die Entscheidung zu akzeptieren. Das Brüten endet (frühestens) nach 9 Sekunden. Während gebrütet wird, darf keine andere Aktion ausgeführt werden.
4. Schlüpfen: Ein neuer Pinguin des Alters 0 (mit Wahrscheinlichkeit 50% eine Kleinuin) wird auf dem Ausgangsfeld hinterlassen, wenn ein Brütuin sein Feld verlässt.
5. Altern: Bei jeder Bewegung erhöht sich das Pinguinalter um 1.
6. Verschwinden: Hat ein Paralleluin den Rand des Spielfelds erreicht, kann er sich über diesen Rand hinwegbewegen und das Spielfeld endgültig verlassen. Der Thread soll dann enden. Die Bewegung zählt wie eine Bewegung innerhalb des Spielfelds auf ein angrenzendes Feld, d. h. in einer Ecke ist die Auswahlwahrscheinlichkeit 50%, sonst 25%.

In der Klasse `Colony` gibt es folgende Attribute und Methoden:

- `public Colony(int width, int height, boolean standard)`:
Der Konstruktor bekommt die Spielfeldgröße sowie die Entscheidung, ob der Standard als Anfangszustand gewählt werden soll, übergeben.
- `private final int[] [] landscape`:
Das Array wird als Parameter zum Zeichnen mittels `GUI.draw` verwendet.
- `private final Penguin[] [] placed`:
Das Array enthält die Pinguine an ihren aktuellen Positionen.

- `public final Object[] [] squareLocks:`
Das Array enthält für jedes Feld ein Objekt, das als Lock verwendet werden kann, um die Aktionen der Pinguine synchronisiert auszuführen. Es müssen sowohl Race-Conditions als auch Deadlocks verhindert werden. Bedenken Sie auch, dass ein Mannin sich nicht bewegen (können) darf, wenn er zum Brüten herangezogen wird.
- `public final Object drawLock:`
Kann als Lock verwendet werden, um das Zeichnen mit der Methode `GUI.draw` zu synchronisieren.
- `public void move(Penguin peng, int x, int y, int xNew, int yNew):`
Diese Methode soll den Paralleluin vom Feld `[x][y]` weg auf das Feld `[xNew][yNew]` bewegen. Sie wird aufgerufen, nachdem bereits mittels Synchronisation sichergestellt ist, dass die Bewegung erfolgen darf. Die Methode soll außerdem immer wieder `GUI.draw` aufrufen: Hat sich der übergebene Paralleluin seit dem letzten Aufruf von `GUI.draw` bewegt, soll dies bemerkt und vor der Aktion neu gezeichnet werden.

In der Klasse `Penguin` gibt es folgende Attribute und Methoden:

- `public Penguin(boolean female, int x, int y, int age, Colony col):`
Der Konstruktor bekommt das Geschlecht, die Anfangskoordinaten, das Alter sowie die Kolonie, zu der der Paralleluin gehört, übergeben.
- `public void run():`
Enthält den Code, den der Thread für diesen Penguin ausführen soll.
- `public int getFg():`
Soll die passende Konstante zum Zeichnen (siehe Tabelle) zurückgeben.

Alle vorgegebenen Klassen, Methoden, Attribute etc. sollen (wo nötig) **ergänzt** werden. Geben Sie die durchgeführten Aktionen auch auf der Konsole aus.

Lösungsvorschlag 12.6

Korrekturhinweise:

- Eine Methode `GUI.ocean` enthält einen unnötigen vierten Parameter. Nach diversen Rückfragen im Forum wurde eine Version der Methode ohne diesen Parameter hinzugefügt.

Punkteverteilung:

- 2 Punkte: Bewegung mit Synchronisation (keine Deadlocks oder Race-Conditions)
- 2 Punkte: Brüten mit Synchronisation (keine Totsperrern oder Rennbedingungen) und breeding-Attribut
- 1 Punkt: Schlüpfen und Altern
- 1 Punkt: Anzeigen (`getFg()`, `draw` aufrufen)
- 1 Punkt: `draw` nicht bei jeder Bewegung aufrufen
- 1 Punkt: Kein unnötiges oder falsches Warten (busy wait)
- Fehlende Ausgabe der Veränderungen auf der Konsole kann Unklarheiten des Ablaufs zur Folge haben und deshalb zu Punktabzug führen.

Aufgabe 12.7 (H) Bandwurm

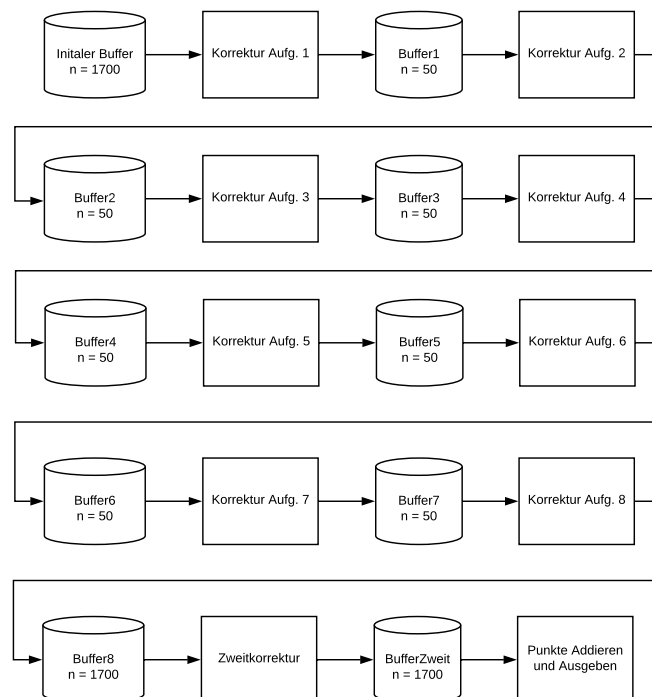
[5 Punkte]

Um eine möglichst gerechte und schnelle Korrektur der anstehenden Korrekturen zu gewährleisten, werden wir dies hier simulieren. Dafür haben wir bereits die Klasse **Klausur** fertig implementiert.

Ihre Aufgabe ist es, die Klassen **Tutor** und **Uebungsleitung** zu erstellen, wobei jeder Tutor für entweder die Korrektur einer Aufgabe oder das Zusammenzählen der Punkte und die Ausgabe des Ergebnisses (auch als Eintragen der Punkte bekannt) zuständig ist. Um eine möglichst schnelle Korrektur zu garantieren, werden die Aufgaben natürlich zur gleichen Zeit korrigiert, daher wird jeder Tutor aus einem Thread bestehen, der für eine Aufgabe zuständig ist.

Bei der Arbeit unterstützt wird der Tutor durch die Klasse **Korrekturschema**, die für die Antwort auf eine Aufgabe die korrekte Punktzahl zurückgibt, sowie für eine Gesamtpunktzahl die korrekte Note.

Ihre Aufgabe ist es zudem, die Klasse **Klausurkorrektur** zu erstellen. Diese ist für die generelle Organisation zuständig. Der Ablauf der Korrektur ist wie folgt:



Die Klasse enthält somit insgesamt 7 Puffer der Größe 50 sowie 3 Puffer der Größe 1700. Der Zugriff auf diese Puffer soll mit Hilfe von Semaphoren synchronisiert werden, wobei der initiale Puffer im Konstruktor der Klasse mit genau so vielen Klausuren gefüllt wird. Insgesamt gibt es 60 Tutoren – sprich Sie erstellen 60 Threads –, wobei die Aufteilung der Tutoren auf die einzelnen Aufgaben Ihnen überlassen ist, es muss nur für jede Aufgabe und das Addieren der Punkte mindestens einen Tutor geben.

Die Tutoren müssen nur ihre Aufgaben korrigieren, indem sie in den Puffer links von sich schauen, ob es da eine zu bearbeitende Klausur gibt. Falls dies der Fall ist, bearbeiten sie diese, indem sie sie aus dem Puffer entnehmen, ihre Aufgabe korrigieren und in den

nächsten Puffer schreiben, falls dort noch Platz ist. Sollte der nächste Puffer bereits komplett gefüllt sein, so wartet der Tutor, bis es einen freien Platz im Puffer gibt. **Hinweis: Falls Sie unsicher sind, wie Sie dies mit Semaphoren lösen können, so schauen Sie nochmals in die Vorlesungsfolien.**

Die **Uebungsleitung** besteht aus zwei Threads. Sie ist für die Zweitkorrektur zuständig. Die Zweitkorrektur übernimmt in 9 von 10 Fällen die Punktzahl der Erstkorrektur und erhöht oder verringert ansonsten die Punktzahl einer Aufgabe um einen Punkt. Die Punkte können dabei niemals negativ werden oder die maximale Punktzahl einer Aufgabe übersteigen, siehe **Korrekturschema**.

Überlegen Sie sich, wann ein Tutor fertig ist mit seiner Korrektur und wann die Korrektur insgesamt komplett geschafft ist; erst wenn dies der Fall ist, soll eine Ausgabe der Form **"Korrektur der Info 1 Klausur beendet :)"** erfolgen und der Mainthread sich beenden.

Aufgabe 12.8 (H) Paralleles Sortieren

[5 Bonuspunkte]

In dieser Aufgabe werden wir den Ihnen bekannten Sortieralgorithmus *Mergesort* parallelisieren, um eine bessere Laufzeit zu erreichen.

Wir erweitern dafür den Algorithmus in der folgenden Art und Weise:

1. Die Klasse `ParallelMergeSort` verfügt über eine statische Variable `int numberOfThreads = 8`; die bestimmt, wie viele Threads erstellt werden sollen.
2. Jedes Mal, wenn wir das zu sortierende Array aufteilen und rekursiv sortieren, prüfen wir zuerst, ob bereits alle Threads erstellt wurden. Ist dies nicht der Fall, erstellen wir einen neuen Thread, der dafür zuständig ist, den entsprechenden Teil des Arrays zu sortieren.

Hinweis: Es kann passieren, dass mehrere Threads zugleich versuchen, neue Threads zu erstellen; stellen Sie sicher, dass niemals zu viele Threads erstellt werden.

3. Sind bereits alle Threads erstellt, werden die entsprechenden Teile des Arrays mit passenden Methoden der Klasse `NormalMergeSort` sortiert. Sie dürfen hierzu – wo nötig – die Zugriffsbeschränkung von Methoden dieser Klasse ändern.

Wir liefern die Klassen `NormalMergeSort` und `MergeSortTest`. Die Testklasse sortiert ein großes Array mit der *single-threaded* Variante sowie mit verschiedenen Werten für `numberOfThreads` und gibt im Anschluss die Laufzeit zurück. Die Klasse prüft zudem, ob die Arrays korrekt sortiert werden.

Sie müssen die folgende Klasse implementieren, sodass sie den oben beschriebenen *parallelen Mergesort* ausführt.

```
1 public class ParallelMergeSort {
2     public static int numberOfThreads = 8;
3
4     public static void mergeSort(int[] arr) {
5         // TODO
6     }
7 }
```

In dieser Aufgabe wollen wir den Parser und Code-Generator um Objektorientierung erweitern, sodass wir anschließend die Sprache *MiniJava mit Funktionen, Objekten und Arrays*, kurz MiniJava++, übersetzen und interpretieren können. Der Interpreter wird dazu nicht verändert. Wie üblich können Sie Ihre Lösung auf der Musterlösung des letzten Blattes oder Ihren eigenen Lösungen aufbauen. Es sollen dazu Programme der folgenden Form geparkt und übersetzt werden können:

```
1  class Bar {
2      int a;
3
4      Bar() {
5          a = 10;
6      }
7
8      int x() {
9          return 22;
10     }
11 }
12
13 class Foo extends Bar {
14     int a;
15     int b;
16
17     Foo() {
18         super.Bar();
19         a = 4;
20         b = 2;
21     }
22
23     int x() {
24         return 99;
25     }
26
27     int y() {
28         return this.x() + 2;
29     }
30 }
31
32 int main() {
33     Foo foo1, foo2;
34     Bar bar1, bar2;
35     foo1 = new Foo();
36     foo2 = foo1;
37     bar1 = new Bar();
38     bar2 = foo1;
39     return foo1.x() + foo2.x() + bar1.x() + bar2.x();
40 }
```

Bevor wir die Grammatik erweitern, sollten einige Anmerkungen beachtet werden:

- Alle Methoden sind **public**, alle Attribute **private**. Die Schlüsselwörter kommen in der Sprache nicht vor.
- Funktionen dürfen weiterhin außerhalb von Klassen vorkommen, wie z.B. die obige **main**-Funktion.
- Methodenaufrufe erfolgen ausschließlich in der Form `obj.name(...)`, wobei es die Schlüsselwörter **this** und **super** gibt, um auf Methoden der aktuellen Klasse bzw. der Superklasse zuzugreifen. Funktionsaufrufe gibt es weiterhin; sie rufen Funktionen außerhalb von Klassen auf.
- Attribute können, da sie ja privat sind, nur innerhalb der Klasse zugegriffen werden. Ein Zugriff auf ein Attribut erfolgt wie auf eine lokale Variable, also ohne vorangestelltes **this** oder **super**. Verschattungen sind ausgeschlossen und ein Fehler, der Programmierer muss darauf achten, jeden Namen nur ein einziges Mal pro Kontext zu verwenden.
- Hat eine Klasse eine Basisklasse, so muss der Basiskonstruktor **immer explizit** und immer im ersten Statement³ des Unterklassenkonstruktors aufgerufen werden. Der Konstruktor der Superklasse **Class** wird über `super.Class(...)` aufgerufen. Das Einhalten dieser Regeln muss nicht extra geprüft werden als Teil der Fehlerbehandlung.
- Jede Klasse **muss** genau einen Konstruktor haben. Konstruktoren werden nicht automatisch generiert.
- Methodenaufrufe werden dynamisch gebunden, wie dies von Java bekannt ist; wir implementieren also Überschreibung wie in Java. Beachten Sie, dass Aufrufe an **this** ebenfalls dynamisch gebunden werden, Aufrufe an **super** dagegen nicht.
- Es gibt nach wie vor keinerlei Überladung in MiniJava++. Ein bestimmter Methodenname darf pro Vererbungshierarchie daher nur genau dann mehrfach vorkommen, wenn Methoden überschrieben werden sollen.

Wir erweitern nun die Grammatik von MiniJava+ (vgl. Blatt 11) zu *MiniJava++* entsprechend Abbildung 1 (beachten Sie, dass Operatoren zum Sparen von Platz weggelassen wurden; es gibt hier keine Änderungen).

Objekte bestehen in MiniJava++ aus zwei Elementen:

- Die Werte der Attribute: Jedes Objekt speichert die Werte seiner Attribute. Jedes Attribut belegt genau eine Heap-Zelle der Größe 32 Bit. Ein Attribut kann eine **int**-Zahl oder auch ein anderes Objekt oder ein Array sein.
- Eine Referenz auf die sog. virtuelle Tabelle: Zu jeder Klasse gehört eine virtuelle Tabelle. Diese virtuelle Tabelle (virtual table, *vtable*) speichert die Anfangsadressen der Methoden der jeweiligen Klasse und ihrer Oberklassen (ohne Konstruktoren). Im obigen Beispiel hat die Klasse **Foo** eine virtuelle Tabelle der Größe 2, die Klasse **Bar** dagegen hat eine Tabelle der Größe 1.

³Es dürfen Deklarationen vor dem Aufruf des Oberklassenkonstruktors vorkommen.


```

<program>    ::= (<function> | <class>)*

<class>      ::= class <name> { <field>* <constructor> <function>* }
               | class <name> extends <name> {
                 <field>* <constructor> <function>* }

<field>      ::= <type> <name> ;

<constructor> ::= <name> ( <params> ) { <decl>* <stmt>* }

<function>   ::= <type> <name> ( <params> ) { <decl>* <stmt>* }

<params>     ::=  $\epsilon$  | (<type> <name>)(, <type> <name>)*

<decl>       ::= <type> <name> (, <name> )* ;

<type>       ::= <name> | <name> []

<stmt>       ::= ;
               | { <stmt>* }
               | <name> = <expr>;
               | <name> [ <expr> ] = <expr>;
               | <name> = read();
               | write( <expr> );
               | if ( <cond> ) <stmt>
               | if ( <cond> ) <stmt> else <stmt>
               | while( <cond> ) <stmt>
               | return <expr>;
               | <expr>;

<expr>       ::= <number>
               | <name>
               | new <name> [ <expr> ]
               | <expr> [ <expr> ]
               | <name> ( ( $\epsilon$  | <expr>(, <expr>)* ) )
               | <name> . <name> ( ( $\epsilon$  | <expr>(, <expr>)* ) )
               | new <name> ( ( $\epsilon$  | <expr>(, <expr>)* ) )
               | length ( <expr> )
               | ( <expr> )
               | <unop> <expr>
               | <expr> <binop> <expr>

<cond>       ::= true | false
               | ( <cond> )
               | <expr> <comp> <expr>
               | <bunop> ( <cond> )
               | <cond> <bbinop> <cond>

```

Abbildung 1: Grammatik von MiniJava++

Generell gilt, dass eine Objektreferenz des Typs einer Unterklasse auch eine gültige Referenz vom Typ der Oberklasse ist. Dies wird erreicht, indem man die Referenz auf den *vtable* an Offset 0 des Objekts legt. Anschließend folgen die Attribute der obersten Oberklasse, gefolgt von denen der Unterklassen. In gleicher Weise verfährt man mit den Adressen der Methoden im *vtable*; hier ist allerdings zusätzlich zu beachten, dass Methoden, die bereits in einer Oberklasse vorkommen, keinen weiteren Eintrag im Bereich der Unterklasse im *vtable* erhalten. Ein Objekt des Typs `Foo` sieht wie in Abbildung 2 gezeigt aus.

Offset	Objekt	Offset	vtable
3	<code>Foo.b</code>	1	Ref. auf <code>y</code>
2	<code>Foo.a</code>	0	Ref. auf <code>x</code>
1	<code>Bar.a</code>		
0	Ref. auf <i>vtable</i>		

Abbildung 2: Aussehen eines Beispielobjekts

Das dynamische Binden von Methodenaufrufen wird durch passende Einträge im *vtable* realisiert. Wird ein neues Objekt über `new` erzeugt, so laufen folgende Schritte ab:

1. Das Objekt und ein passend großer *vtable* werden allokiert, sowie die Referenz des *vtables* an die richtige Stelle im Objekt geschrieben. Die hierzu verwendete Funktion für eine Klasse `Class` nennen wir `$alloc_Class`. Die Funktion liefert die Heap-Referenz auf das neue Objekt zurück.
2. Der Konstruktor der Klasse wird aufgerufen. Der Konstruktor führt wiederum einige feste Schritte aus:
 - (a) Hat die Klasse eine Oberklasse, so wird zunächst das erste Statement des Konstruktors ausgeführt. Dieses muss zwingend der Aufruf des Konstruktors der Oberklasse sein (vgl. oben). Die virtuelle Tabelle ist jetzt auf die Methoden der Oberklasse initialisiert.
 - (b) Die virtuelle Tabelle wird mit den Adressen der Methoden der Unterklasse initialisiert. Auf diese Weise werden die Verweise auf überschriebene Methoden derart umgebogen, dass sie sodann auf die Methoden der Unterklasse zeigen.
 - (c) Der durch den Nutzer gegebene Rest des Konstruktors wird ausgeführt.

Gehen Sie bei Ihrer Implementierung wie folgt vor:

1. Erweitern Sie zunächst den Code-Generator um die Klassen `Class`, `Constructor`, `ObjectInitializer`, `MethodCall` und `ExpressionStatement`. Letzteres benötigen Sie für den Aufruf des Konstruktors der Basisklasse. Sie müssen außerdem das Enum `Type` durch eine oder mehrere Klassen ersetzen, damit hier nun auch Namen von Klassen als Typen vorkommen können.
2. Erweitern Sie den `FormatVisitor`, sodass er mit allen neuen Klassen umgehen kann.
3. Erweitern Sie den Parser, sodass dieser MiniJava++ parsen kann.
4. Erweitern Sie die Code-Generierung zunächst derart, dass Klassen ohne Vererbung möglich werden.

5. Fügen Sie schließlich Ihrer Implementierung die Möglichkeit von Vererbung hinzu.

Hinweis: Beachten Sie, dass die Klasse `CompilerTest` um Tests für diese Aufgabe erweitert wurde.

Lösungsvorschlag 12.9

Punkteverteilung:

1. Hierarchieerweiterung: 0.5 Punkte
2. Erweiterung des Formatierers: 0.5 Punkt
3. Parsererweiterung: 0.5 Punkte
4. Klassen ohne Vererbung: 1.5 Punkte
5. Vererbung: 2 Punkte

Korrekturhinweise:

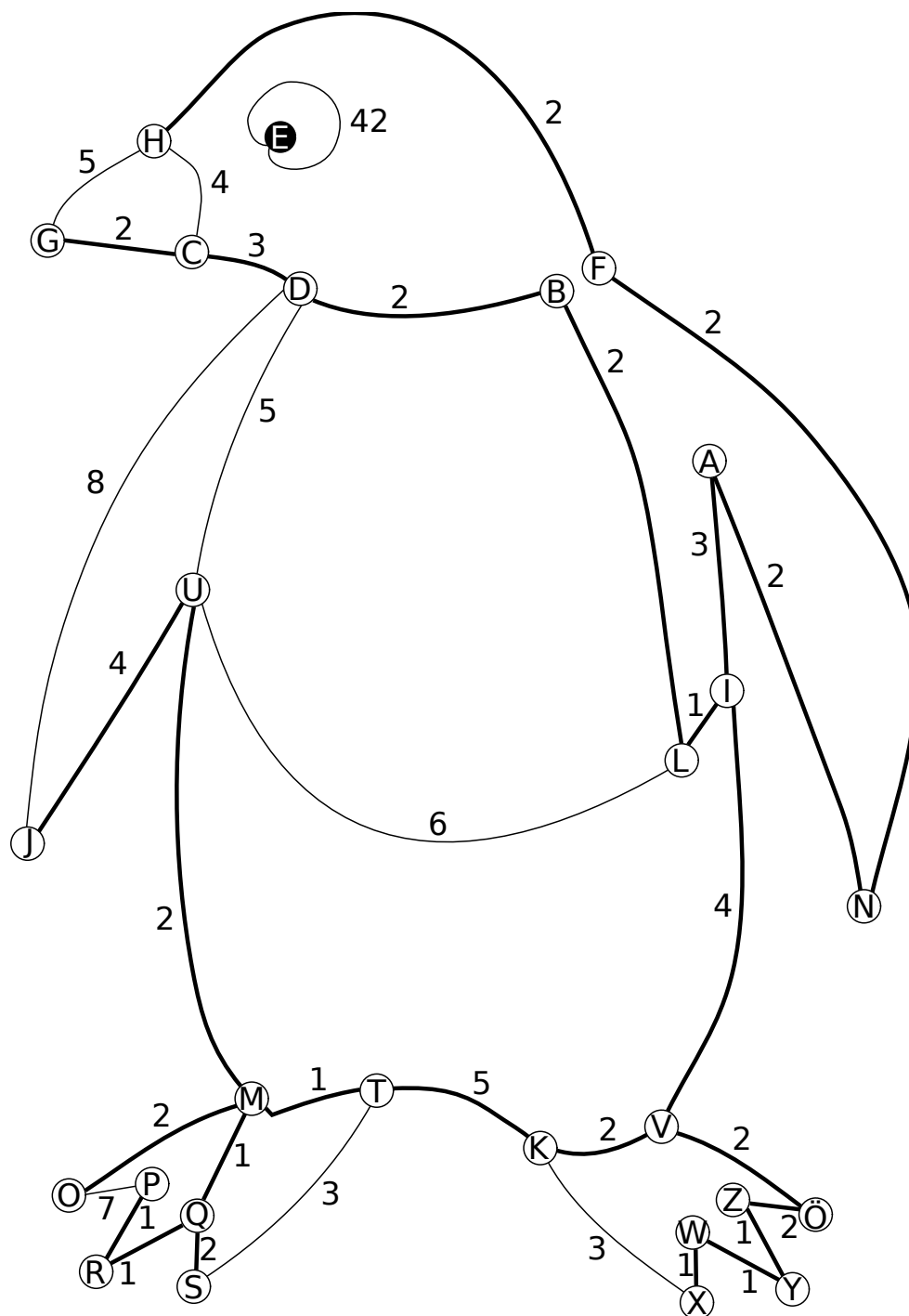
- Aufgabe ist sehr anspruchsvoll \rightsquigarrow Leistung würdigen, nicht wild Punkte abziehen

Aufgabe 12.10 (H) Ende gut, alles Pinguin gut!

[2 Bonuspunkte]

Uns ist aufgefallen, dass wir bisher im Praktikum noch keine machbare Aufgabe gestellt haben. In dieser Aufgabe geht es nun darum, einen Pinguin möglichst natürlich und farbenfroh auszumalen. Gehen Sie bei Ihrer Malung wie folgt vor:

1. Beginnen Sie mit dem Kopf des Pinguins.
2. Setzen Sie Ihre Arbeit mit dem Rumpf des Tierchens fort.
3. Runden Sie Ihr Werk durch die Gestaltung der Extremitäten ab.



Lösungsvorschlag 12.10

Die Übungsleitung war von dieser Aufgabe leider überfordert und konnte auch sonst in der Informatik-Fakultät keine passende Kompetenz finden.