Florian Tischler
Mat. Nr.: 1315066
Gruppe: 6
Gruppenmitglieder: Tischler Florian, Mathias Hölzl
gelöste Aufgaben: 3/3
resultierende Punkte: 100%

Aufgabe 1

Wird der EventListener bereits im Konstruktor registriert, wird implizit die **this** Referenz veröffentlicht da Instanzen innerer (nicht statischer) Klassen immer eine Referenz auf die umgebende Klasseninstanz hält. Die **this** Referenz sollte aber deswegen niemals im Konstruktor veröffentlicht werden, da man durch diese auf ein Objekt zugreifen kann welches noch nicht vollständig initialisiert wurde (was meistens zu Fehlverhalten führt).

Immutable Klassen haben den Vorteil, dass ihr innerer Zustand nicht von außen geändert werden kann und damit keine Racebedingungen etc. auftreten können. Deshalb sind solche Klassen immer threadsafe.

Anmerkungen zur Implementierung

Mit den Parametern am Anfang der main() Methode kann die Threadzahl und die Ausführungszeit angepasst werden.

Aufgabe 2

Das Problem bei der gegebenen Implementierung ist, dass eine Bedingung geprüft wird und anhand von diesem Ergebnis handelt, ohne jedoch auszuschließen, dass zwischen dem Prüfen der Bedingung der daraus folgenden Handlung der Wert der Bedingung gleichbleibt. Somit kann es vorkommen, dass der Wert für i zur Zeit des Checks ein gültiger Wert wäre aber bis der Wert dann übernommen wird er nicht mehr gültig ist. Um dieses Verhalten zu provozieren wurde zwischen der Bedingung und dem Setzen ein sleep() eingebaut. Am Ende befindet sich noch ein abschließender Test ob die Invariante noch gültig ist. Somit wird überprüft ob ein fehlerhaftes Verhalten ausgelöst wurde (setUpper() analog).

```
public void setLower(int i) {
    // Warning -- unsafe check-then-act
    if (i > upper.get()) {
        throw new IllegalArgumentException("can't set upper to " + i + " <
lower");
    }

    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    lower.set(i);

    if(lower.get() > upper.get()) {
        throw new IllegalStateException("invariant does not hold");
    }
}
```

Um das die Klasse threadsafe zu gestalten wurden einfach beide Methoden als synchronized gekennzeichnet.

Aufgabe 3

Die Messungen befinden sich in der Datei Task3_log.txt. Es fällt auf, dass die synchronized Variante (dabei wird immer die komplette Map gesperrt) viel langsamer ist als die Concurrent Variante. Das hängt wahrscheinlich damit zusammen, dass die ConcurrentMap immer den kleinstmöglichen Teil sperrt (z.B.: muss bei einer HashMap immer nur das betrachtete Element der Hash-Liste gesperrt werden). Auch die Umverteilung der Zugriffe hat eine direkte Auswirkung auf die Performance da Lesezugriffe viel billiger als Einfüge bzw. Löschvorgänge. Die Verteilung (5, 90, 5) war daher mit Abstand am performantesten. Was ich mir nicht ganz erklären kann ist warum die Performance (Zugriffe pro Sekunde) mit der Anzahl an Zugriffen steigt (siehe Log). Ich tippe darauf, dass es einigen „konstanten“ Overhead gibt der bei kleinen Mapgrößen mehr ins Gewicht fällt als bei großen.