Übung 7: ConBench – Multithreaded Benchmarks

In dieser Übung entwickeln Sie ein Tool um multithreaded Benchmarks auszuführen. Ein Benchmark ist eine Performance Messung. In unserem Falle messen wir die Laufzeit eines Stück Programm Codes.

Eine einfache Benchmark Klasse sieht wie folgt aus:

```
@Benchmark(100000)
public static final class RWLock {
    final ReadWriteLock lock = new ReentrantReadWriteLock();
    final Lock writeLock = lock.writeLock();
    final Lock readLock = lock.readLock();

    @Threads({ 1, 1, 1, 1 })
    public void write(int nTimes, int nThreads) {
        for (int i = 0; i < nTimes; i++) {
            writeLock.lock();
            try { burnCycles(1); }
            finally { writeLock.unlock(); }
        }
    }

    @Threads({ 1, 2, 4, 8 })
    public void read(int nTimes, int nThreads) {
        for (int i = 0; i < nTimes; i++) {
            readLock.lock();
            try { burnCycles(1); }
            finally { readLock.unlock(); }
        }
    }
}
```

Die Klasse ist annotiert mit @Benchmark. Die Annotation nimmt als Argument einen int-Parameter, dessen Bedeutung gleich erklärt wird.

Die Klasse enthält zwei Methoden, die mit @Threads annotiert sind. Jede dieser Methoden nimmt zwei int-Parameter `nTimes` und `nThreads`.
Der Parameter `nThreads` sagt aus, mit wie vielen Threads die Methode parallel ausgeführt wird. Er kann verwendet werden um die zu Messende Aufgabe über die arbeitenden Threads zu verteilen.
Der Parameter `nTimes` bestimmt, wie häufig der zu messende Code innerhalb der Methode ausgeführt werden soll. Entsprechend führt jede Methode den zu untersuchende Code innerhalb einer for-Schleife `nTimes` mal aus. Zur Laufzeit werden die Methoden mit dem int-Wert der @Benchmark-Annotation aufgerufen[1].

Die for-Schleife innerhalb einer Benchmark-Methode ist Vorschrift - jede Benchmark-Methode muss den zu untersuchenden Code in einer for-Schleife `nTimes` mal ausführen. Das Framework misst die Laufzeit der Methoden und teilt dann diese durch `nTimes` um die Laufzeit einer Ausführung des zu untersuchenden Codes auszugeben.

Alle @Threads annotierten Methoden einer @Benchmark Klasse werden gleichzeitig ausgeführt. Die @Threads Annotation nimmt als Argument ein int[]. Jedes Array Element beschreibt die Anzahl Threads womit die annotierte Methode *parallel* ausgeführt werden soll. Die Idee ist, dass die Methode mit unterschiedlicher Last ausgeführt und dabei die Laufzeit gemessen wird. Das Framework garantiert, dass alle @Threads int[] einer Benchmark Klasse die gleiche Länge haben.

[1] Der zu untersuchende Code muss viele Male ausgeführt werden, damit der HotSpot Compiler den Java Bytecode auf nativen Maschinen Code übersetzt hat. Der ambitionierte Student ist eingeladen, den Parameter aus @Benchmark zu entfernen und zur Laufzeit-Zeit einen geeigneten Wert zu ermitteln bevor die Zeitmessung beginnt.

Für obiges Beispiel werden vier Messungen gemacht:

1. Messung: write mit 1 Thread und read mit 1 Thread -> Total 2 Threads
2. Messung: write mit 1 Thread und read mit 2 Threads -> Total 3 Threads
3. Messung: write mit 1 Thread und read mit 4 Threads -> Total 5 Threads
4. Messung: write mit 1 Thread und read mit 8 Threads -> Total 9 Threads

Resultat einer Ausführung
Die Referenzimplementierung des Frameworks liefert für obigen Benchmark folgende Ausgabe:

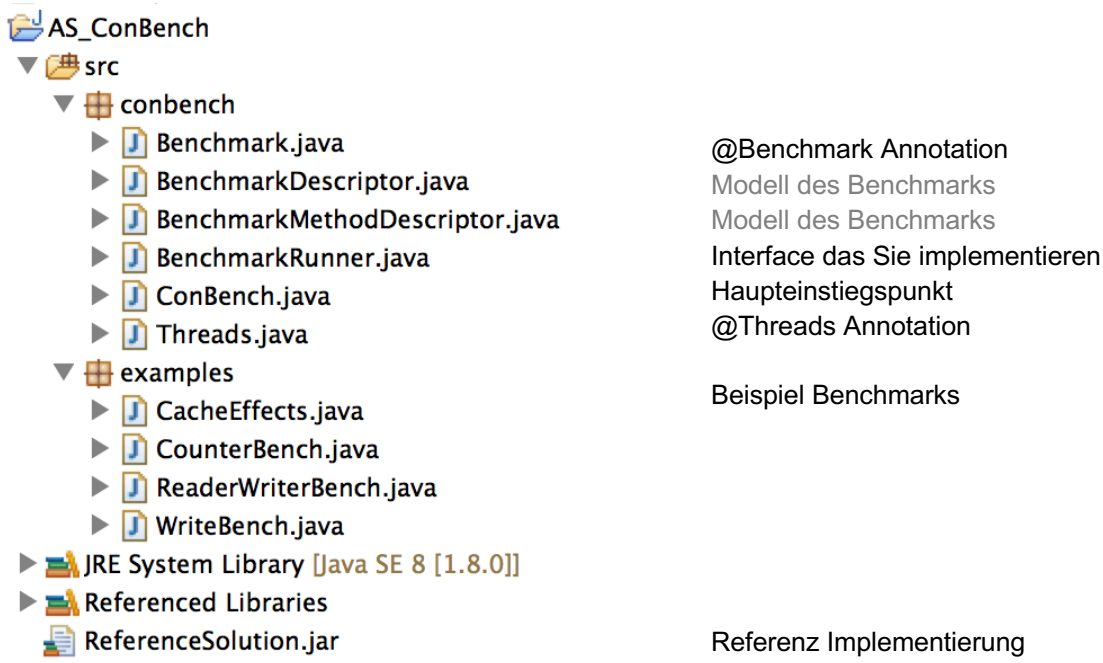
```
RWLock: Warming up ...
 Starting benchmark ...
 - Run[0] write(1), read(1), Duration: 8490ns
 - Run[1] write(1), read(2), Duration: 7985ns
 - Run[2] write(1), read(4), Duration: 9791ns
 - Run[3] write(1), read(8), Duration: 12.458µs
```

Bedeutung einer Resultat-Zeile:

Run[0]	Durchführung mit erstem Element des @Threads(int[]) Arguments
read(1), write(1)	Ausgeführte Methoden mit (#Threads)
Duration: 8490ns	Ausführungszeit einer Ausführung (Totalzeit / nTimes)

Aufgaben:

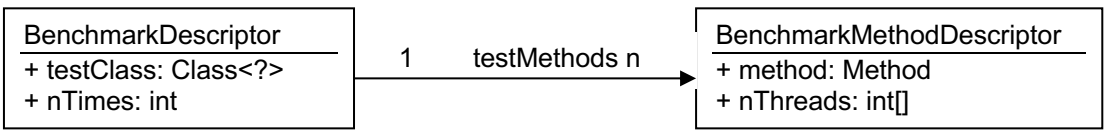
Auf dem AD finden sie das 07_AS_ConBench.zip Archiv. Importieren Sie das Projekt in Ihren Eclipse Workspace. Das Projekt hat folgende Struktur:



Die Klasse ConBench ist der Haupteinstiegspunkt. Die main Methode interpretiert die String Argumente als Klassennamen. Diese müssen vollqualifiziert übergeben werden. Alternativ können die Testklassen direkt in der main Methode angegeben werden:

```
Class<?>[] benchClasses = { CounterBench.class, WriteBench.class };
```

Die Klassen werden dann vom Framework analysiert und es wird ein Modell erzeugt, das die Benchmark Struktur abbildet. Das Model sieht so aus (Konstruktoren wurden weggelassen):



Für jede @Benchmark annotierte Klasse wird ein BenchmarkDescriptor Objekt erzeugt, das auf mehrere BenchmarkMethodDescriptor Objekte verweist, eines für jede @Threads annotierte Methode innerhalb der Klasse. Als Beispiel für einen Benchmark mit mehreren Methoden betrachten Sie die Klasse ReaderWriterBench. ConBench stellt sicher, dass alle nThreads Arrays eines BenchmarkDescriptors dieselbe Länge haben.

Ihre Aufgabe ist es nun das Interface BenchmarkRunner zu implementieren:

```
public interface BenchmarkRunner {
    void runBenchmark(BenchmarkDescriptor desc);
}
```

Die Idee ist, dass die Methode runBenchmark den durch das Argument beschriebene Benchmark ausführt und die Messresultate auf die Konsole ausgibt. Beachten Sie, dass die Benchmark Threads alle

zum gleichen Zeitpunkt loslaufen sollten und dass sie schon gestartet sind bevor Sie die Startzeit messen.

Damit Sie überhaupt etwas ausführen können, benötigen sie ein wenig Java Reflection:

```
// Neue Instanz erzeugen
Object testObject = testClass.newInstance()

// Ausführen der Methode method auf dem Objekt testObject mit dem Argument nTimes
method.invoke(testObject, nTimes, nThreads)
```

1. Implementieren sie das Interface BenchmarkRunner und geben Sie eine Instanz Ihres Runners von der Methode `ConBench.createRunner()` zurück. Bevor Sie die Ausführungszeit messen, sollten Sie die JVM aufwärmen, indem Sie den Code einige Male ausführen bevor Sie mit der Messung beginnen. Detailliertere Informationen zum Thema Microbenchmarks finden Sie unter [1]. Ausser JVM vorwärmen, müssen Sie nichts unternehmen[2].
2. Führen sie die bestehenden Benchmarks aus. Wie erklären Sie die unterschiedlichen Zeiten des CacheEffects Benchmarks?
3. (Optional) Erstellen Sie eigene Benchmarks. Vergleichen Sie z.B. die Performance des von Ihnen in der letzten Übung implementierten ImmutableTreeSets mit einem in ein Synchronized Wrapper verpacktes j.u.TreeSet. Bei welchen Zugriffsmustern ist das ImmutableTreeSet überlegen?

[1] https://wikis.oracle.com/display/HotSpotInternals/MicroBenchmarks

Abgabe: 15./16. April 2019

[2] Sie sollten sich aber bewusst sein, dass die Messungen ziemlich ungenau sein können.