www.javaspektrum.de

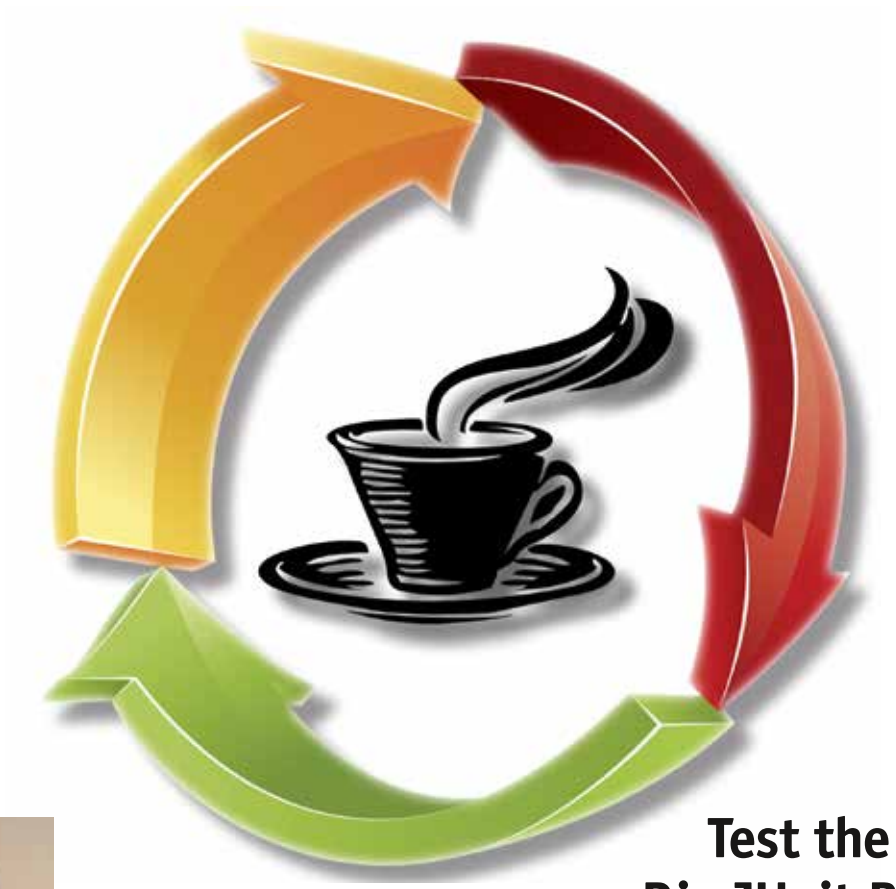
Ausgabe 4/2020
D € 9,90 · A € 10,90 · CHF 18,20

4 194156 109901 04

JavaSPEKTRUM

Magazin für professionelle Entwicklung und digitale Transformation

Das moderne Java-Ökosystem – JVMs, SDKs und mehr



Test the Test – Die JUnit Platform

Java EE ist tot … es lebe Spring (Boot)

Interview

Professor Ulrich Kelber, Bundesdatenschutzbeauftragter, über Datenschutz als das Kernelement der Digitalisierung

Fachthemen

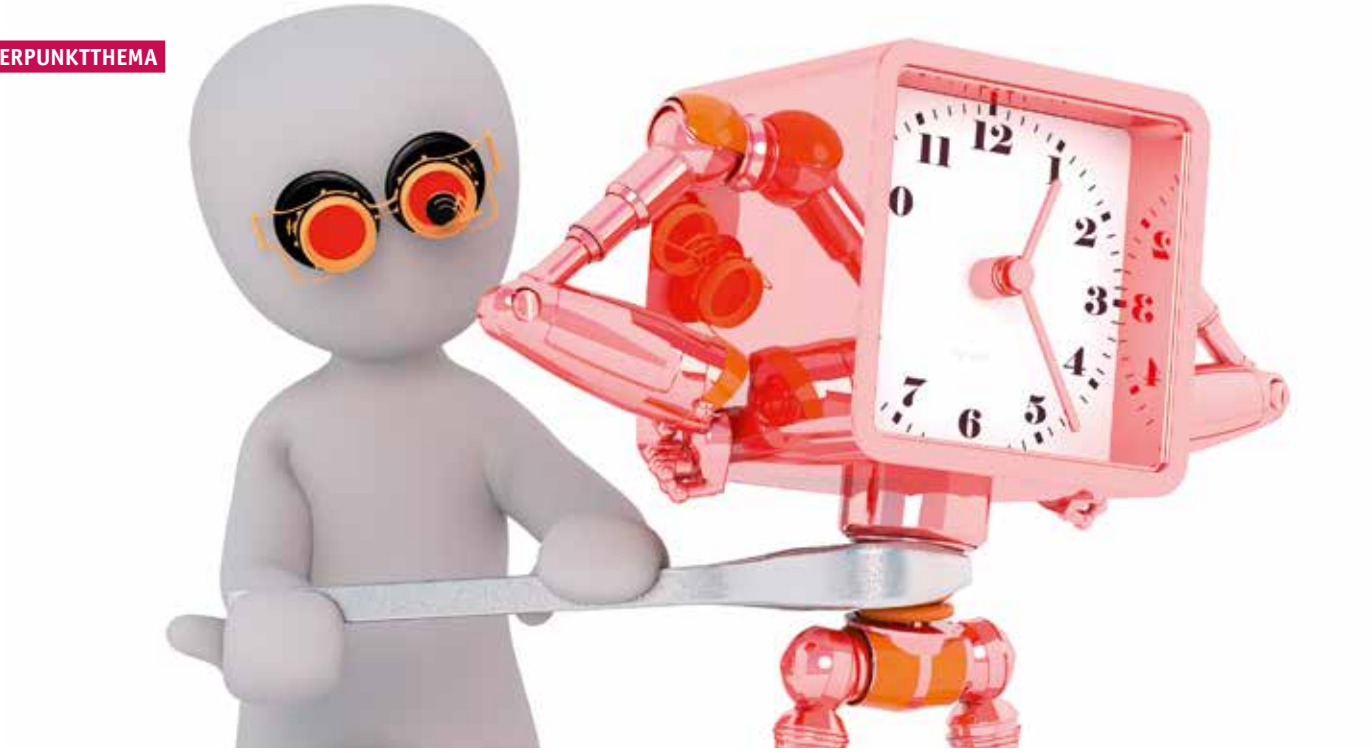
Testen von und mit Künstlicher Intelligenz

Fehlerkultur als Basis für erfolgreiche agile Unternehmen

zum Inhalt

Das moderne Java-Ökosystem – JVMs, SDKs und mehr

G30325



Laufzeit messen

Performanzanalysen in Java – Teil 1: Java Microbenchmarks

Christian Heitzmann

Bei der Performanzbeurteilung eigener Java-Programme und -Methoden spielt präzise Zeitmessung eine zentrale Rolle. Teil 1 dieser zweiteiligen Artikelserie demonstriert die Schwierigkeiten klassischer Stoppuhrmessungen und gibt eine Einführung in den Java Microbenchmark Harness (JMH), mit dem sich wesentlich fundiertere Messresultate erzielen lassen.

Im Rahmen dieses Artikels möchten wir zu Demonstrationszwecken vergleichen, auf welche Art sich eine `List<Integer>` am schnellsten erzeugen und mit 10 000 zufälligen Zahlenwerten füllen lässt. Zur Auswahl stehen:

- eine uninitialisierte `ArrayList`,
- eine vorinitialisierte `ArrayList`; sie reserviert bereits im Vorfeld den Platz für die 10 000 zu erwartenden Elemente,
- eine `LinkedList`.

Listing 1 zeigt die Klasse `ListCreateAndFill`, deren drei `public`-Methoden wir in ihrem Laufzeitverhalten untersuchen möchten.

```
import java.util.*;
import java.util.concurrent.*;

public final class ListCreateAndFill {
  private static final int ELEMENT_COUNT = 10_000;

  public List<Integer> createAndFillDefaultArrayList() {
    List<Integer> defaultArrayList = new ArrayList<>();
    addRandomNumbers(defaultArrayList);
    return defaultArrayList;
  }
  public List<Integer> createAndFillInitialCapacityArrayList() {
    List<Integer> initialCapacityArrayList
          = new ArrayList<>(ELEMENT_COUNT);
    addRandomNumbers(initialCapacityArrayList);
    return initialCapacityArrayList;
  }
  public List<Integer> createAndFillLinkedList() {
    List<Integer> linkedList = new LinkedList<>();
    addRandomNumbers(linkedList);
    return linkedList;
  }
  private List<Integer> addRandomNumbers(List<Integer> list) {
    for (int i = 1; i < ELEMENT_COUNT; i++) {
      int randomNumber = ThreadLocalRandom.current().nextInt();
      list.add(Integer.valueOf(randomNumber));
    }
    return list;
  }
}
```

Listing 1: Klasse ListCreateAndFill

Einzelne Stoppuhrmessungen mit currentTimeMillis und nanoTime

Die einfachste und naheliegendste Art, die Laufzeit einer Methode zu messen, besteht wohl darin, ihre Ausführung mit zwei Aufrufen

currentTimeMillis()	1. Lauf	2. Lauf	3. Lauf
Default ArrayList	2 ms	2 ms	2 ms
Initial Capacity ArrayList	1 ms	1 ms	1 ms
LinkedList	1 ms	1 ms	2 ms

Tabelle 1: Stoppuhrmessungen mit currentTimeMillis

Christian Heitzmann ist Gründer und Geschäftsführer der SimplexaCode AG in Luzern, die sich auf Softwareentwicklung, -schulung und -beratung mit Schwerpunkt technischer Implementierungsthemen und Java spezialisiert hat. Er entwickelt seit über 20 Jahren Software und hat während vieler Jahre Algorithmen und Mathematik unterrichtet.
E-Mail: christian.heitzmann@simplexacode.ch

```
public final class WallClockMeasurements {
  private static final double NANOS_PER_MILLI = 1_000_000.;
  private ListCreateAndFill listCreateAndFill
        = new ListCreateAndFill();

  public static void main(String[] args) {
    WallClockMeasurements wallClockMeasurements
       = new WallClockMeasurements();
    wallClockMeasurements.measureElapsedTimesUsingCurrentTimeMillis();
    /* [...] Other calls omitted. */
  }
  private void measureElapsedTimesUsingCurrentTimeMillis() {
    System.out.println("currentTimeMillis()");
    System.out.println("------------------");
    {
      long startTimeMillis = System.currentTimeMillis();
      listCreateAndFill.createAndFillDefaultArrayList();
      long endTimeMillis = System.currentTimeMillis();
      long elapsedTimeMillis = endTimeMillis - startTimeMillis;
      System.out.format("Default ArrayList:          %d ms%n",
                  Long.valueOf(elapsedTimeMillis));
    }
    {
      long startTimeMillis = System.currentTimeMillis();
      listCreateAndFill.createAndFillInitialCapacityArrayList();
      long endTimeMillis = System.currentTimeMillis();
      long elapsedTimeMillis = endTimeMillis - startTimeMillis;
      System.out.format("Initial Capacity ArrayList: %d ms%n",
                  Long.valueOf(elapsedTimeMillis));
    }
    {
      long startTimeMillis = System.currentTimeMillis();
      listCreateAndFill.createAndFillLinkedList();
      long endTimeMillis = System.currentTimeMillis();
      long elapsedTimeMillis = endTimeMillis - startTimeMillis;
      System.out.format("LinkedList:                 %d ms%n",
                  Long.valueOf(elapsedTimeMillis));
    }
    System.out.println();
  }
}
```

Listing 2: Stoppuhrmessungen mit currentTimeMillis

von `System.currentTimeMillis()` zu ummanteln und von den beiden so ermittelten Zeitpunkten *vor* und *nach* dem Methodenaufruf die Differenz zu berechnen und auszugeben. Die Klasse `WallClockMeasurements` in Listing 2 zeigt das Gerüst einer solchen Messkonstellation und nimmt die Messungen jeweils für alle drei Methoden vor, sodass wir die Resultate anschließend direkt auf einen Blick vergleichen können.

```
private void measureElapsedTimesUsingNanos() {
  System.out.println("nanos()");
  System.out.println("-------");
  {
    long startTimeNanos = System.nanoTime();
    listCreateAndFill.createAndFillDefaultArrayList();
    long endTimeNanos = System.nanoTime();
    double elapsedTimeMillis
       = (endTimeNanos - startTimeNanos) / NANOS_PER_MILLI;
    System.out.format("Default ArrayList:          %f ms%n",
                Double.valueOf(elapsedTimeMillis));
  }
  /* [...] Other time measurements omitted. */
}
```

Listing 3: Stoppuhrmessungen mit nanoTime

Auf meinem Rechner (iMac Retina 5 K mit 4 GHz Intel Core i7 und Java 8) sehen die Programmausgaben bei dreimaliger Ausführung wie in Tabelle 1 aus.

Die Ergebnisse sind wenig aussagekräftig. Sie schwanken von Durchlauf zu Durchlauf, wie man am Beispiel von *LinkedList* gut sieht, und Millisekunden sind für die Messung von so kurzen Methoden-Ausführungszeiten nicht ausreichend. Die Bezeichnung „Millis“ gibt übrigens nur die *Einheit* des zurückgegebenen Zeitstempels an; die eigentliche *Genauigkeit* (engl. *granularity*) oder *Auflösung* (engl. *resolution*) kann je nach Betriebssystem noch gröber ausfallen [JAPIcurrentTimeMillis].

Zugriff auf eine hochauflösende Uhr erhält man in Java mit `System.nanoTime()` [JAPInanoTime]. In dessen API heißt es: *„Returns the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds.“* Und weiter: *„This method provides nanosecond precision, but not necessarily nanosecond resolution [...] - no guarantees are made except that the resolution is at least as good as that of currentTimeMillis().“*

Fügen wir eine private Methode `measureElapsedTimesUsingNanos` hinzu und rufen sie im Anschluss an die erste Methode `measureElapsedTimesUsingCurrentTimeMillis` auf. Der neue Teil des Programmcodes ist in Listing 3 zu finden. (Aus Platzgründen werden ab jetzt nicht mehr alle Aufrufe abgedruckt. Den vollständigen Quellcode finden Sie unter [Code].) Auf meiner Maschine generiert er die Ausgaben aus Tabelle 2 bei dreimaliger Ausführung.

Die Messwerte von `nanos()` machen einen numerisch hochpräzisen Eindruck, der aber dadurch getrübt wird, dass er sich zwischen den Durchläufen teilweise recht stark unterscheidet. So zeigen sich für *LinkedList* zum Beispiel Schwankungen im Bereich von 0,74 ms bis 0,91 ms.

Es kommt aber noch schlimmer: Kommentieren wir den Aufruf von `measureElapsedTimesUsingCurrentTimeMillis` aus, verzichten also auf seinen Aufruf vorab, und lassen wir nur die Zeiten von `nanos()` messen und ausgeben, siehe Tabelle 3.

currentTimeMillis() | nanos()	1. Lauf	2. Lauf	3. Lauf
Default ArrayList	3 ms | 0.630342 ms	3 ms | 0.608932 ms	3 ms | 0.621924 ms
Initial Capacity ArrayList	0 ms | 0.610858 ms	1 ms | 0.562408 ms	1 ms | 0.537961 ms
LinkedList	1 ms | 0.743514 ms	1 ms | 0.908039 ms	1 ms | 0.758326 ms

Tabelle 2: Stoppuhrmessungen mit currentTimeMillis und nanoTime

nanos()	1. Lauf	2. Lauf	3. Lauf
Default ArrayList	2.538454 ms	2.489637 ms	2.762718 ms
Initial Capacity ArrayList	0.764182 ms	0.742832 ms	1.095337 ms
LinkedList	0.963779 ms	0.976926 ms	1.621343 ms

Tabelle 3: Stoppuhrmessungen (nur) mit nanoTime

```
private void measureAverageTimesInLoopsUsingNanos() {
  System.out.println("Loops");
  System.out.println("-----");
  final int LOOP_COUNT = 100_000;
  {
    long startTimeNanos = System.nanoTime();
    for (int i = 1; i <= LOOP_COUNT; i++) {
      listCreateAndFill.createAndFillDefaultArrayList();
    }
    long endTimeNanos = System.nanoTime();
    double averageTimeMillis = (endTimeNanos - startTimeNanos)
        / NANOS_PER_MILLI / LOOP_COUNT;
    System.out.format("Default ArrayList:         %f ms%n",
                  Double.valueOf(averageTimeMillis));
  }
  /* [...] Other time measurements omitted. */
}
```

Listing 4: Stoppuhrmessungen in Schleifen

Die Ausführungszeit für die *Default ArrayList* ist nun etwa viermal länger als zuvor. Wie kann so etwas passieren? Heutige Java-Compiler sind hochoptimiert und verwenden *Just-in-Time-Kompilierung (JIT)* [WikiJIT]. Das bedeutet, dass Teile des Java-Bytecodes bei Bedarf zur Laufzeit direkt in Maschinencode kompiliert werden. Die Entscheidung, ob und wann kompiliert wird, hängt davon ab, wie häufig gewisse Stellen eines Quellcodes ausgeführt werden. Für selten ausgeführte Stellen lohnt sich die (relativ teure) Kompilierung nicht; für oft ausgeführte Stellen hingegen schon.

Wird die Methode `measureElapsedTimesUsingCurrentTimeMillis` *vor* der Methode `measureElapsedTimesUsingCurrentTimeMillis` ausgeführt, so reicht dies offenbar schon aus, um die JIT-Kompilierung insbesondere für den *Default-ArrayList*-Teil anzustoßen. Die zweite Messrunde (mit `nanos()`) kann dann bereits vom schnelleren Maschinencode profitieren. Fällt die erste Messrunde (mit `currentTimeMillis()`) weg, so kann der Teil `nanos()` nicht vom Maschinencode profitieren, sondern muss im Gegenteil sogar noch „auf seine Kosten" den Kompiliervorgang abwarten.

Stoppuhrmessungen in Schleifen

Das *„Aufwärmen"* (engl. *warmup*) spielt also eine zentrale Rolle, wenn es darum geht, stabile Laufzeiten zu messen. Listing 4 zeigt die neue Methode `measureAverageTimesInLoopsUsingNanos`, welche die Füllmethoden nun jeweils 100 000-mal aufruft, davon die *Gesamt*zeit misst, und anschließend den Durchschnittswert *pro Methodenaufruf* berechnet und ausgibt. Auf meinem Rechner ergeben sich die Resultate aus Tabelle 4.

```
private void measureAverageTimesInNestedLoopsUsingNanos() {
  System.out.println("Nested Loops");
  System.out.println("------------");
  final int OUTER_LOOP_COUNT = 10;
  final int INNER_LOOP_COUNT = 100_000;
  {
    for (int i = 1; i <= OUTER_LOOP_COUNT; i++) {
      long startTimeNanos = System.nanoTime();
      for (int j = 1; j <= INNER_LOOP_COUNT; j++) {
        listCreateAndFill.createAndFillDefaultArrayList();
      }
      long endTimeNanos = System.nanoTime();
      double elapsedTimeMillis = (endTimeNanos - startTimeNanos)
          / NANOS_PER_MILLI / INNER_LOOP_COUNT;
      System.out.format("Default ArrayList:         %f ms%n",
                    Double.valueOf(elapsedTimeMillis));
    }
  }
}
```

Listing 5: Stoppuhrmessungen in verschachtelten Schleifen

Wir sind nun ganz grob einen Faktor 10 „besser" als die Resultate aus den ersten Messungen. Allerdings erkennen wir auch hier nicht, welchen Einfluss eine allfällige JIT-Kompilierung auf die Laufzeiten hat. Die Unterschiede zwischen den drei Testreihen sind auch hier deutlich zu sehen. Es liegt der Verdacht nahe, dass das jeweilige Aufstarten der *Java Virtual Machine* (JVM) einen erheblichen Einfluss auf die Laufzeiten hat.

Betrachten wir also mit der Methode `measureAverageTimesInNestedLoopsUsingNanos` in Listing 5, wie sich die gemessenen Laufzeiten mit der Zeit entwickeln. Dazu ummanteln wir die Schleife, welche die 100 000 Aufrufe vornimmt, mit einer Schleife, die das Ganze 10-mal wiederholt.

Die Ausgaben unterscheiden sich nun zwischen den drei Ausführungen nicht mehr groß, von daher reicht es aus, wenn ich von den Resultaten meiner Maschine nur noch eine paar wenige Werte abdrucke, siehe Tabelle 5.

Teilweise sind die jeweiligen Aufwärmphasen gut zu erkennen, insbesondere bei *Default ArrayList* (von 0,061 ms auf 0,057 ms) und *LinkedList* (von 0,069 ms auf 0,063 ms). Die *Initial Capacity ArrayList* hingegen scheint sich auf einem im Vergleich zum Anfang leicht erhöhten Wert einzupendeln (von 0,048 ms auf 0,050 ms).

Mit diesen Resultaten bewegen wir uns erstmals auf halbwegs sicherem Terrain, um zumindest qualitativ beantworten zu können, dass das Erzeugen und Befüllen einer mit der Größe vorinitialisierten `ArrayList` schneller geht als dasjenige einer uninitialisierten `ArrayList` oder gar einer `LinkedList`.

Loops	1. Lauf	2. Lauf	3. Lauf
Default ArrayList	0.064710 ms	0.058706 ms	0.058787 ms
Initial Capacity ArrayList	0.047972 ms	0.048044 ms	0.047930 ms
LinkedList	0.069275 ms	0.070304 ms	0.061753 ms

Tabelle 4: Stoppuhrmessungen in Schleifen

Nested Loops	1. Wert	2. Wert	3. Wert		9. Wert	10. Wert
Default ArrayList	0.060611 ms	0.057353 ms	0.057718 ms	[...]	0.057649 ms	0.057501 ms
Initial Capacity ArrayList	0.048143 ms	0.048088 ms	0.050514 ms	[...]	0.050477 ms	0.050439 ms
LinkedList	0.069067 ms	0.069195 ms	0.063010 ms	[...]	0.063426 ms	0.063816 ms

Tabelle 5: Stoppuhrmessungen in verschachtelten Schleifen

Zwischenfazit: Stoppuhrmessungen

Wir haben bis jetzt gesehen: Stoppuhrmessungen sind sehr diffizil. Sie benötigen eine Menge Hintergrundverständnis und recht aufwendige Implementierungen für halbwegs korrekte, mehr oder weniger aussagekräftige Resultate. Die Anzahl der Schleifendurchläufe für eine gute Balance zwischen Gesamtlaufzeit und Messgenauigkeit muss durch Ausprobieren herausgefunden werden. Insgesamt steht der benötigte Aufwand also in keinem Verhältnis zum Nutzen. Nicht zuletzt suggerieren die hochpräzisen numerischen Resultate Genauigkeiten, die es in den jeweiligen Szenarien gar nicht gibt.

Java Microbenchmark Harness (JMH)

Der *Java Microbenchmark Harness (JMH)* wurde der breiten Java-Öffentlichkeit wahrscheinlich erst mit der Einführung von Java 12 bekannt. Dort erhielt mit dem *JDK Enhancement Proposal JEP 230* die sogenannte *Microbenchmark Suite* Einzug in das JDK.

Um gleich an dieser Stelle mit einem weitverbreiteten Irrtum (der sich bis in die Fachliteratur hinein erstreckt) aufzuräumen:

```
import java.util.*;
import java.util.concurrent.*;
import org.openjdk.jmh.annotations.*;

public class ListCreateAndFillBenchmark {

  private static final int ELEMENT_COUNT = 10_000;

  @Benchmark
  public List<Integer> createAndFillDefaultArrayList() {
    List<Integer> defaultArrayList = new ArrayList<>();
    addRandomNumbers(defaultArrayList);
    return defaultArrayList;
  }
  @Benchmark
  public List<Integer> createAndFillInitialCapacityArrayList() {
    List<Integer> initialCapacityArrayList
          = new ArrayList<>(ELEMENT_COUNT);
    addRandomNumbers(initialCapacityArrayList);
    return initialCapacityArrayList;
  }
  @Benchmark
  public List<Integer> createAndFillLinkedList() {
    List<Integer> linkedList = new LinkedList<>();
    addRandomNumbers(linkedList);
    return linkedList;
  }
  private List<Integer> addRandomNumbers(List<Integer> list) {
    for (int i = 1; i < ELEMENT_COUNT; i++) {
      int randomNumber = ThreadLocalRandom.current().nextInt();
      list.add(Integer.valueOf(randomNumber));
    }
    return list;
  }
}
```

Listing 6: Klasse ListCreateAndFillBenchmark

Mit JEP 230 beziehungsweise Java 12 ist weder der JMH noch irgendein anderes „Performanz-Mess-Tool" Teil des JDKs geworden [JEP230]. JMH war, ist und bleibt weiterhin ein externes Projekt, zu finden unter [JMH]. Mit JEP 230 beziehungsweise Java 12 wurden hingegen nur Benchmarks für die Implementierungen der *JVM* und des *JDK APIs selbst* ins JDK aufgenommen. So kann die Performanz der JVM und der JDK APIs mithilfe des (externen) JMH gemessen werden. Wer also vorhat, einzelne Teile der JVM oder des APIs zu optimieren, der findet im JDK fertige Benchmarks vor, mit denen er die Performanz vor und nach seiner Optimierung vergleichen kann (analog einem mit (Unit-)Tests abgedeckten Code, der mit einem *externen* Testframework wie zum Beispiel *JUnit* getestet werden kann). Dieses Projekt existierte allerdings schon seit Java 8 [JMHJDKBench]. Der einzige Unterschied ist, dass es früher noch ausgelagert war, und seit Java 12 nicht mehr extern, sondern fester Bestandteil des JDKs ist. Für den allergrößten Teil der normalsterblichen Java-Entwickler dürfte das jedoch uninteressant sein, weswegen ich auch nicht nachvollziehen kann, wieso die „Microbenchmark Suite" bei der Entwickler-Community mit der Veröffentlichung von Java 12 so groß angeworben wurde.

Wie die Bezeichnung „Harness" ausdrückt, handelt sich beim JMH um ein „Geschirr" oder „Gurtzeug", welches um die zu testenden Klassen und Methoden gelegt wird. Wie wir gleich sehen werden, reicht es dazu meist schon aus, die zu untersuchenden Methoden und Klassen mit der Annotation `@Benchmark` zu versehen. In einem Maven-Build-Prozess werden dann automatisch die entsprechenden Benchmark-Klassen generiert, die dann die eigentlichen Messungen vornehmen. Im Prinzip geschieht das, was wir im vorherigen Abschnitt in mühevoller Handarbeit in der Klasse `WallClockMeasurements` selbst zusammengebaut haben, nur dass die generierten Codes der JMH-Annotationsprozessoren deutlich umfangreicher, flexibler und technisch ausgereifter sind.

Messungen mit dem JMH

Um die Methoden unserer (später noch minimal anzupassenden) Klasse `ListCreateAndFill` mit dem JMH messen zu können, erzeugen wir in *Eclipse* zuerst ein neues *Maven Project*. Im *Archetype Catalog* suchen wir dann nach `org.openjdk.jmh.jmh-java-benchmark-archetype` und geben für unser Projekt eine beliebige *Group Id* und *Artifact Id* an. Daraufhin wird ein neues Eclipse-Projekt erzeugt, in dem wir unsere zu testende Klasse in den Ordner `src/main` einfügen können. Unter [JMH] ist erklärt, wie man ein solches Maven-Projekt auf Kommandozeile, für eine andere IDE oder gar eine andere (JVM-) Programmiersprache generiert.

Listing 6 zeigt die Klasse `ListCreateAndFillBenchmark`, die sich von der Vorgängerversion einzig und allein am Namen und den drei Annotationen `@Benchmark` am Kopf der drei `public`-Methoden unterscheidet. Außerdem darf sie nicht mehr `final` sein. Fügen wir diese Klasse in das soeben erzeugte Maven-Projekt ein und führen anschließend ein *Maven clean install* aus, so sehen wir, wie im Ordner `target` diverse Sourcen, Properties und JAR-Dateien erzeugt wer-

```
@Warmup(iterations = 3)
@Measurement(iterations = 2)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
```

Listing 7: Annotationen für Anpassungen des JMH

den. Es ist sehr eindrücklich, einmal einen Blick in die generierten Quellcode-Dateien zu werfen.

Der eigentliche Benchmark ist ein selbstständiges Java-Programm, dessen `main`-Methode sich in der Klasse `org.openjdk.jmh.Main` in der JAR-Datei `benchmarks.jar` befindet. Ein gesamter Durchlauf benötigt um die 25 Minuten. Im Folgenden ist ein kleiner Ausschnitt aus der sehr langen Programmausgabe meiner Maschine zu sehen:

```
# Run progress: 66.67% complete, ETA 00:08:21
# Fork: 1 of 5
# Warmup Iteration   1: 12947.563 ops/s
# Warmup Iteration   2: 14917.584 ops/s
# Warmup Iteration   3: 15859.657 ops/s
# Warmup Iteration   4: 15859.592 ops/s
# Warmup Iteration   5: 15819.332 ops/s
Iteration   1: 15857.831 ops/s
Iteration   2: 15868.819 ops/s
Iteration   3: 15853.244 ops/s
Iteration   4: 15852.005 ops/s
Iteration   5: 15858.302 ops/s
```

Ein paar Beobachtungen sind bemerkenswert:

- Den eigentlichen (sehr zeitintensiven und damit genauen) Messungen geht jeweils eine Warmup-Phase im gleichen Zeitumfang voraus. Man kann schön erkennen, wie sich die Ausführungsgeschwindigkeit während der Aufwärmphase erhöht und anschließend erstaunlich stabil und konstant bleibt.
- Jeder komplette Testdurchlauf wird 5-mal vorgenommen. Aus diesen 5 Durchläufen wird anschließend ein Durchschnittswert gebildet.
- Die Zusammenfassung der Messresultate erfolgt mit statistischen Zusatzdaten wie dem Wertebereich, der Standardabweichung und Angaben zum Fehler (aus Platzgründen hier nicht abgedruckt).
- Die Ausgabe des *Durchsatzes* (engl. *throughput*) kann je nach Leseart praktischer sein als die Angabe der Laufzeit im Milli-, Mikro- oder Nanosekundenbereich. Der Durchsatz („Operationen pro Sekunde“) ist der Kehrwert der (durchschnittlichen) Ausführungszeit („Sekunden pro Operation“).

Anpassungen des JMH

Selbstverständlich lassen sich die Benchmarks mittels Annotationen im Detail anpassen. Ein guter Einstieg findet sich in der API-Dokumentation des JMH [JMHAPI], dort dann im Paket `org.openjdk.jmh.annotations`. Listing 7 zeigt exemplarisch ein paar Annotationen, die sowohl an einer zu messenden Methode als auch an der Klasse angebracht werden können, wobei in letztem Fall die Annotationen für alle Methoden dieser Klasse gelten:

- `@Warmup(iterations = 3)`: Es finden nur 3 (statt 5) Warmup-Iterationen statt.
- `@Measurement(iterations = 2)`: Dito, bezogen auf die eigentlichen Mess-Iterationen.
- `@BenchmarkMode(Mode.AverageTime)`: Die Programmausgabe erfolgt in *Zeit pro Operation* (statt *Operationen pro Zeit*).
- `@OutputTimeUnit(TimeUnit.MILLISECONDS)`: Die Zeiteinheit (sowohl für *Throughput* als auch für *AverageTime*) wird auf Millisekunden festgelegt. So haben wir den idealen Vergleich mit unseren anfänglichen Stoppuhrmessungen.

Die Resultate auf meiner Maschine sehen vielversprechend aus:

```
# Run progress: 66.67% complete, ETA 00:04:11
# Fork: 1 of 5
# Warmup Iteration   1: 0.077 ms/op
# Warmup Iteration   2: 0.067 ms/op
# Warmup Iteration   3: 0.063 ms/op
Iteration   1: 0.063 ms/op
Iteration   2: 0.063 ms/op
```

Fazit

Mit dem *Java Microbenchmark Harness (JMH)* lassen sich mit wenig Aufwand Laufzeitmessungen mit erstaunlicher Präzision, Stabilität und (mathematischem) Informationsgehalt vornehmen. Sie adaptieren sich selbst und sind „selbstgestrickten“ Messroutinen klar vorzuziehen, zumal die Einrichtung des JMH als Maven-Archetype sehr einfach vonstattengeht.

Mit einem soliden Messwerkzeug in der Tasche werden wir im zweiten Teil dieser Artikelserie einigen „Java-Mythen“ auf den Zahn fühlen und objektiv messen, ob und was an ihnen dran ist. Unabhängig davon gilt nach wie vor: Die Messung von Laufzeiten und die Interpretation jener Ergebnisse sind stets zwei verschiedene Paar Schuhe. Nicht triviale Performanz-Optimierungen sollen nur in ganz klaren Fällen und nur mit größtem Bedacht vorgenommen werden.

Literatur und Links

[Code] Quellcode zum Herunterladen, https://link.simplexacode.ch/zh2n

[Hun14] M. Hunger, Java Microbenchmark Harness (JMH), in: JavaSPEKTRUM, 3/2014

[Ind19] M. Inden, Java – Die Neuerungen in Version 9 bis 12, dpunkt.verlag, 2019

[JAPIcurrentTimeMillis] Java Platform, Standard Edition & Java Development Kit Version 14 API Specification, Class System, Method currentTimeMillis, https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/System.html#currentTimeMillis()

[JAPInanoTime] Java Platform, Standard Edition & Java Development Kit Version 14 API Specification, Class System, Method nanoTime, https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/System.html#nanoTime()

[JEP230] OpenJDK, JEP 230: Microbenchmark Suite, https://openjdk.java.net/jeps/230

[JMH] OpenJDK, Code Tools: jmh, https://openjdk.java.net/projects/code-tools/jmh/

[JMHAPI] JMH Core 1.23 API, https://javadoc.io/doc/org.openjdk.jmh/jmh-core/1.23/index.html

[JMHJDKBench] OpenJDK, Code Tools: JMH JDK Microbenchmarks, https://openjdk.java.net/projects/code-tools/jmh-jdk-microbenchmarks/

[WikiJIT] https://de.wikipedia.org/wiki/Just-in-time-Kompilierung