www.javaspektrum.de

Ausgabe 5/2020
D € 9,90 · A € 10,90 · CHF 18,20

4 194156 109901 05

JavaSPEKTRUM

Magazin für professionelle Entwicklung und digitale Transformation

Kubernetes –
Smarte Orchestrierung von Docker und Co.

Mit Java in das Kubernetes-Cluster – ein Wegweiser

Kubernetes und Edge-Computing vor Ort nutzen

Interview

Björn Goerke, CTO der Car.Software-Organisation über die Digitalisierung bei Volkswagen und wie der Konzern Google und Tesla die Stirn bietet

Fachthemen

Mehrwerte aus Datenträgern mit Massive Online Analysis

Perfomanzanalysen in Java – Teil 2: Praxisbeispiele

zum Inhalt

Sonderbeilage: IT Karriere Magazin – Jobs in der Softwarebranche

Kubernetes – Smarte Orchestrierung von Docker und Co.

G30325

Java-Mythen auf den Zahn gefühlt

Performanzanalysen in Java – Teil 2: Praxisbeispiele

Christian Heitzmann

Im ersten Teil dieser zweiteiligen Artikelserie wurde der Java Microbenchmark Harness (JMH) eingeführt, der selbst programmierten Stoppuhrmessungen in jeglicher Hinsicht überlegen ist. Mit diesem Werkzeug in der Tasche wollen wir im vorliegenden zweiten Teil Alltagsbeispiele aus der Programmierpraxis unter die Lupe nehmen und untersuchen, ob das Bauchgefühl in Sachen Performanz wirklich mit der Realität übereinstimmt.

Wer schon länger in Java programmiert, wird sich mit Sicherheit schon die eine oder andere Performanzfrage gestellt, gelesen oder mit Kollegen diskutiert haben. Heute im Jahre 2020 sind die (erfahrungsgemäß meist sinnlosen) Diskussionen auf „Bit-, Byte- und CPU-Takt-Ebene“ zum Glück in den Hintergrund geraten, ganz anders als noch vor 10 bis 15 Jahren, als man sogar noch Java gegenüber dem ach so performanten C++ rechtfertigen musste.

Bei derartigen Diskussionen spiel(t)en meines Erachtens Ideologien oder Fehlinformationen meist eine größere Rolle als objektive (Mess-)Daten. Wer meine regelmäßigen Artikel aus JavaSPEKTRUM kennt, weiß, dass ich als Autor der objektiven Nachvollziehbarkeit immer oberste Priorität beimesse – sei es durch vollständige, reproduzierbare Code-Beispiele oder durch die Angabe anerkannter Referenzen.

Vier Praxisbeispiele

Ich habe vier überschaubare Beispiele mit „Kaffeezimmer-Diskussionspotenzial“ ausgewählt und möchte Sie, lieber Leser, vorab bitten, sich ein paar Minuten Zeit zu nehmen und für jedes Beispiel zu überlegen und zu notieren, welche Variante wohl die „schnellste“ sein wird und um welchen Faktor sie sich von den anderen aufgezeigten Varianten unterscheidet (jeweils in einer Single-Thread-Umgebung):

- *Zufallszahlen:* Welcher Zufallszahlengenerator liefert langfristig die schnellste Rate an zufälligen Integer-Werten? `java.util.Random`, `java.util.concurrent.ThreadLocalRandom` oder `java.security.SecureRandom` mit `NativePRNG`-Algorithmus? Gibt es einen Geschwindigkeitsunterschied zwischen Aufrufen von `nextInt` in einer Schleife und der streambasierten Variante des jeweiligen Generators?
- *String-Konkatenationen:* Es sollen 10 000 kurze Strings zu einem einzigen String verkettet werden. Zur Auswahl stehen die Konkatenation mit dem Operator `+`, die Methode `String#concat(String)` sowie ein `StringBuilder` und ein `StringBuffer`, jeweils mit und ohne Angabe einer *initialen Kapazität*.
- *Hash-Codes:* Ist eine handgeschriebene Implementierung von `hashCode()` der seit Java 7 bestehenden Convenience-Methode `Objects.hash(Object...)` überlegen? Lohnt es sich, den Hash-Code für unveränderbare (immutable) Objekte in einem *Cache* zu speichern?
- *Operationen als Parameter:* Einer Berechnungsmethode soll die Angabe der von ihr vorzunehmenden Operation via Parameter angegeben werden. Zur Auswahl stehen die Übergabe eines *Enums* (z. B. `performOperation(OperationEnum.ADDITION, ...)`), die Übergabe des Operationsnamens als *String* (z. B. `performOperation("SUBTRACTION", ...)`) und die Übergabe der Operation als *Klasse*, de-

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

ren Operationsmethode anschließend via Reflection aufgerufen wird (`performOperation(MultiplicationOperation.class)`).

Zufallszahlen

Im JavaSPEKTRUM-Artikel [Hei19] hatte ich die Funktionsweisen und Unterschiede der Zufallszahlengenerierung einmal im Detail erörtert. Für sogenannte *Pseudozufallszahlen* darf und soll man `java.util.Random` komplett durch `java.util.concurrent.ThreadLocalRandom` ersetzen. Für *echte Zufallszahlen* mit sicherheitskritischen Anforderungen bedarf es `java.security.SecureRandom` mit einer Auswahl verschiedener Algorithmen.

Wir wollen zuerst die Performanz von `Random` mit jener von `ThreadLocalRandom` vergleichen. Listing 1 zeigt den ersten Teil der

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

@State(Scope.Benchmark)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class RandomNumberBenchmark {

  private static final int RANDOM_NUMBER_COUNT = 1_000_000;
  private Random random = new Random();
  private SecureRandom secureRandom;

  public RandomNumberBenchmark() {
    try {
      secureRandom = SecureRandom.getInstance("NativePRNG");
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    }
  }

  /* [...] Main method omitted. */

  @Benchmark
  public long randomWithForLoop() {
    long sum = 0;
    for (int i = 1; i <= RANDOM_NUMBER_COUNT; i++) {
      sum += random.nextInt();
    }
    return sum;
  }
  @Benchmark
  public long threadLocalRandomWithForLoop() {
    long sum = 0;
    for (int i = 1; i <= RANDOM_NUMBER_COUNT; i++) {
      sum += ThreadLocalRandom.current().nextInt();
    }
    return sum;
  }
```

Listing 1: RandomNumberBenchmark – Random vs. ThreadLocalRandom

Benchmarks. Die eigentlichen Messungen finden in den mit der Annotation `@Benchmark` gekennzeichneten Methoden `randomWithForLoop` und `threadLocalRandomWithForLoop` statt. Es werden in einer Schleife jeweils 1 000 000 Zufallszahlen generiert. Aus diesen wird wiederum die Summe gebildet und zurückgegeben. Das ist wichtig, damit aus Sicht des (Just-in-Time-)Compilers die Werte auch wirklich verwendet werden; ansonsten besteht die Möglichkeit, dass die Aufrufe an den Zufallszahlengenerator „wegoptimiert" werden, was unserem Benchmark dann wenig zuträglich wäre.

Die Laufzeitunterschiede sind zugunsten von `ThreadLocalRandom` deutlich. Auf meiner Maschine (für diese wie für alle anderen Benchmarks ein iMac Retina 5K mit 4 GHz Intel Core i7 und Java 8) liegt dazwischen ein Faktor größer als 7:

```
Benchmark                       Mode  Cnt    Score   Error  Units
randomWithForLoop               avgt   25    8.891 ± 0.006  ms/op
threadLocalRandomWithForLoop    avgt   25    1.206 ± 0.001  ms/op
```

Ich persönlich hätte einen derart großen Unterschied nicht erwartet. Die Berechnung innerhalb von `Random` ist im Kern ein banaler *linearer Kongruenzgenerator*, der auf eine einfache Formel zurückfällt. In [BlogJDKsSourceCode] hatte ich vor einiger Zeit einmal beschrieben, wie man den Quellcode des JDKs einsehen kann. Darin sieht man, dass in `Random` die Seeds stets in einem `AtomicLong` gespeichert und verarbeitet werden. So erkauft sich `Random` Threadsicherheit auf Kosten der Laufzeit. Im JavaSPEKTRUM-Artikel [Hei20-a] habe ich die atomaren Datentypen – nebst anderen Locking-Mechanismen – ausführlich erklärt. `ThreadLocalRandom` erreicht Threadsicherheit auf eine andere, wesentlich effizientere Art als das mittlerweile 25 Jahre alte `Random`. Meine Aussage in [Hei19], künftig nur noch `ThreadLocalRandom` statt `Random` einzusetzen, hat sich also auch in Sachen Laufzeit voll bestätigt.

Seit der Einführung von Streams in Java 8 bietet `ThreadLocalRandom` auch streambasierte Methoden zur Verarbeitung großer Mengen an Zufallszahlen an. Die Berechnung der Summe aller 1 000 000 Zufallszahlen lässt sich in Java sehr elegant wie in Listing 2 gezeigt formulieren.

```
@Benchmark
public long threadLocalRandomWithStream() {
  return ThreadLocalRandom.current()
      .ints(RANDOM_NUMBER_COUNT).sum();
}
```

Listing 2: RandomNumberBenchmark – ThreadLocalRandom als Stream

Doch wie sieht es mit der Laufzeit aus?

```
Benchmark                       Mode  Cnt    Score   Error  Units
threadLocalRandomWithStream     avgt   25    2.443 ± 0.002  ms/op
```

Hier liegt ein Faktor 2 zuungunsten der Stream-Variante vor. Auch das hätte ich so in dieser Form nicht erwartet, wird Streams – notabene sogar in offizieller Zertifizierungsliteratur [SieBatRob9] – doch nachgesagt, dass das JDK in ihnen diverse Optimierungen vornehmen kann. Das soll allerdings weder für noch gegen einen generellen Einsatz von Streams sprechen. Sie haben ihre Berechtigung, wenn sie die Lesbarkeit *deutlich* erhöhen (für obiges Beispiel würde ich das bejahen), gleichzeitig kann ich mich dem Autor Joshua Bloch nur anschließen, wenn er von einem voreiligen Einsatz von Streams ohne nennenswerten Zugewinn deutlich abrät [Blo45].

Schauen wir uns abschließend die Performanz von `SecureRandom` an. Listing 3 zeigt sowohl die Schleifen- als auch die Stream-Variante.

```
@Benchmark
public long secureRandomWithForLoop() {
  long sum = 0;
  for (int i = 1; i <= RANDOM_NUMBER_COUNT; i++) {
    sum += secureRandom.nextInt();
  }
  return sum;
}
@Benchmark
public long secureRandomWithStream() {
  return secureRandom.ints(RANDOM_NUMBER_COUNT).sum();
}
```

Listing 3: RandomNumberBenchmark – SecureRandom

„Echte" Zufallszahlen, die aus einer Entropiequelle des Betriebssystems bezogen werden, haben ihren Preis – ein stolzer Faktor von über 450 liegt zwischen `ThreadLocalRandom` und `SecureRandom`:

```
Benchmark                    Mode  Cnt    Score   Error  Units
secureRandomWithForLoop      avgt   25  546.999 ± 1.445 ms/op
secureRandomWithStream       avgt   25  550.321 ± 3.384 ms/op
```

Die Diskussion darüber lässt sich aber kurz halten. Werden in sicherheitskritischen Anwendungen echte Zufallszahlen statt generierter Pseudozufallszahlen benötigt, dürfen an dieser Anforderung keine Abstriche gemacht werden – Performanz hin oder her. Umgekehrt bietet sich der Verzicht auf sichere Zufallszahlengeneratoren an, wenn sie nur deswegen eingesetzt werden, weil sie „cool" sind. `ThreadLocalRandom` ist mehr als gut genug für sämtliche nicht sicherheitskritischen und wissenschaftlichen Anwendungen.

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

@State(Scope.Benchmark)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class StringBenchmark {

  private static final int SUB_STRING_COUNT = 10_000;
  private List<String> subStringList;

  public StringBenchmark() {
    subStringList = ThreadLocalRandom.current()
        .ints(SUB_STRING_COUNT, 0, Integer.MAX_VALUE)
        .mapToObj(Integer::toString)
        .collect(Collectors.toList());
  }

  /* [...] Main method omitted. */

  @Benchmark
  public String stringPlus() {
    String fullString = "";
    for (String currentSubString : subStringList) {
      fullString += currentSubString;
    }
    return fullString;
  }
  @Benchmark
  public String stringConcat() {
    String fullString = "";
    for (String currentSubString : subStringList) {
      fullString = fullString.concat(currentSubString);
    }
    return fullString;
  }
```

Listing 4: StringBenchmark – + vs. concat

String-Konkatenationen

Widmen wir uns einem etwas alltäglicheren Java-Thema: dem Zusammenfügen von Strings. Im ersten Schritt wollen wir herausfinden, ob ein nennenswerter Laufzeitunterschied zwischen dem `+`-Operator und der `String#concat(String)`-Methode besteht. Im Code in Listing 4 werden zuerst 10 000 kurze, zufällige Teilstringsequenzen vorbereitet und in der Liste `subStringList` gespeichert. Die Generierung der Teilstrings, die aus zufälligen Integer-Zahlen (als Strings) bestehen und damit je eine Länge zwischen 1 und 11 besitzen, soll ja nicht mitgemessen werden. Die beiden eigentlichen Benchmarks in Form der Methoden `stringPlus` und `stringConcat` sind selbsterklärend.

Im Messresultat sind keine nennenswerten Unterschiede zu erkennen:

```
Benchmark                    Mode  Cnt    Score   Error  Units
stringPlus                   avgt   25   81.977 ± 0.145 ms/op
stringConcat                 avgt   25   81.585 ± 0.099 ms/op
```

Das war einerseits zu erwarten. Andererseits schreibt die Java Language Specification [JLS15181] zum String Concatenation Operator `+`: *„To increase the performance of repeated string concatenation, a Java compiler may use the `StringBuffer` class or a similar technique to reduce the number of intermediate String objects that are created by evaluation of an expression."* Zumindest unter meiner Umgebung war das nicht der Fall. Dass `concat` ebenfalls eine solche Optimierung anbietet, ist ausgeschlossen, wie der Blick in den Quellcode jener Methode zeigt. Verwendet man also den Operator `+` anstelle der Methode `concat`, so besteht zumindest noch theoretisch die Möglichkeit, dass ein Java-Compiler derartige Stellen optimiert. Und leserlicher ist es noch dazu.

Dass es auch schneller geht, sollte allen Java-Entwicklern bekannt sein. Listing 5 verwendet nun einen `StringBuilder`, einmal ohne und einmal mit Angabe einer initialen Kapazität (hier 10 x 10 000).

```
@Benchmark
public String defaultStringBuilder() {
  StringBuilder stringBuilder = new StringBuilder();
  for (String currentSubString : subStringList) {
    stringBuilder.append(currentSubString);
  }
  return stringBuilder.toString();
}
@Benchmark
public String initialCapacityStringBuilder() {
  StringBuilder stringBuilder
     = new StringBuilder(10 * SUB_STRING_COUNT);
  for (String currentSubString : subStringList) {
    stringBuilder.append(currentSubString);
  }
  return stringBuilder.toString();
}
```

Listing 5: StringBenchmark – StringBuilder

Dass die Ausführung schneller ist, überrascht wohl nicht. Dass der Geschwindigkeitszuwachs hingegen bei einem Faktor über 450 beziehungsweise sogar über 500 liegt, ist durchaus unerwartet und beeindruckend:

```
Benchmark                    Mode  Cnt   Score   Error  Units
defaultStringBuilder         avgt   25   0.180 ± 0.001  ms/op
initialCapacityStringBuilder avgt   25   0.159 ± 0.003  ms/op
```

Die Klasse `StringBuilder` wurde mit Java 5 als Ersatz für die schon immer existierende Klasse `StringBuffer` eingeführt und nach meiner

```
@Benchmark
public String defaultStringBuffer() {
  StringBuffer stringBuffer = new StringBuffer();
  for (String currentSubString : subStringList) {
    stringBuffer.append(currentSubString);
  }
  return stringBuffer.toString();
}
@Benchmark
public String initialCapacityStringBuffer() {
  StringBuffer stringBuffer
     = new StringBuffer(10 * SUB_STRING_COUNT);
  for (String currentSubString : subStringList) {
    stringBuffer.append(currentSubString);
  }
  return stringBuffer.toString();
}
```

Listing 6: StringBenchmark – StringBuffer

Wahrnehmung seit jeher auch konsequent eingesetzt. Zur offiziellen Begründung heißt es in ihrer API-Dokumentation [JAPIStringBuilder]: *„This class is designed for use as a drop-in replacement for* `StringBuffer` *in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to* `StringBuffer` *as it will be faster under most implementations. “*

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

public final class ValueClass {

  private boolean booleanField;
  private byte byteField;
  private double doubleField;
  private int cachedHashCode;

  private ValueClass(boolean booleanField,
                byte byteField,
                double doubleField) {
    this.booleanField = booleanField;
    this.byteField = byteField;
    this.doubleField = doubleField;
//    cachedHashCode = directHashCode();
  }

  public static ValueClass newRandomInstance() {
    return new ValueClass(ThreadLocalRandom.current().
       nextBoolean(),
                  (byte) ThreadLocalRandom.current().
                  nextInt(),
                  ThreadLocalRandom.current().nextDouble());
  }

  /* [...] toString and equals method omitted. */

  @Override
  public int hashCode() {
    return directHashCode();
//    return objectsHash();
//    return cachedHashCode;
  }
  private int directHashCode() {
    int result = 17;
    result = 37 * result + Boolean.hashCode(booleanField);
    result = 37 * result + Byte.hashCode(byteField);
    result = 37 * result + Double.hashCode(doubleField);
    return result;
  }
  private int objectsHash() {
    return Objects.hash(Boolean.valueOf(booleanField),
                  Byte.valueOf(byteField),
                  Double.valueOf(doubleField));
  }
}
```

Listing 7: ValueClass

Jetzt liegt natürlich die Frage auf der Hand, ob diese Begründung wirklich den Tatsachen entspricht, und wenn ja, wie groß die Unterschiede sind. Listing 6 mit seinen darin enthaltenen zwei Benchmarks gibt Aufschluss darüber.

Die Unterschiede zum `StringBuilder` sind – wider Erwarten – marginal:

```
Benchmark                    Mode  Cnt   Score   Error  Units
defaultStringBuffer          avgt   25   0.187 ± 0.002  ms/op
initialCapacityStringBuffer  avgt   25   0.167 ± 0.002  ms/op
```

Hash-Codes

Eine korrekte Implementierung der Methoden `equals` und `hashCode` gehört zu den essenziellen Teilen eines funktional richtigen und performanten Java-Programms. In der Praxis habe ich hingegen selten wirklich sattelfeste Implementierungen gesehen. Das ist mit Sicherheit dem abstrakten, mathematisch anmutenden und zugegeben nicht ganz einfachen Regelwerk zu verdanken [JAPIObject, Blo10, Blo11, BlogEqualsAndHashCode], mit dem sich der „normale Business-Developer“ nicht unbedingt auseinandersetzen will. Oft begnügt man sich stattdessen – unhinterfragt – mit generiertem Code aus der IDE, Bibliotheken wie *Google Guava* oder lässt sie komplett aus. Wahrscheinlich war genau das der Grund, wieso mit Java 7 die Convenience-Klasse `java.util.Objects` eingeführt wurde, die unter anderem die gebrauchsfertigen statischen Methoden `(deep)equals(Object, Object)` und `hash(Object...)` anbietet [JAPIObjects].

Im vorliegenden Beispiel möchten wir vergleichen, ob eine gute handgeschriebene `hashCode`-Methode jener Standardimplementierung von `Objects.hash` überlegen ist. Dazu habe ich in Listing 7 eine „Werte-Klasse“ namens `ValueClass` erstellt, die ein `boolean`-, ein `byte`- und ein `double`-Attribut enthält. Die eigentliche Implementierung von `hashCode` lässt sich mit Ein- und Auskommentieren der entsprechenden Aufrufe auswählen.

In Listing 8 werden zuerst 100 000 Instanzen mit zufälligen Werten via der Fabrikmethode `ValueClass.newRandomInstance()` erzeugt. Im eigentlichen Benchmark werden all diese Werte einer in der Größe vorinitialisierten `HashSet` hinzugefügt.

Die handgeschriebene `hashCode`-Methode ist knapp 3-mal schneller:

```
Benchmark                                  Mode  Cnt  Score   Error  Units
addElementsToHashSet | directHashCode()    avgt   25  1.985 ± 0.049  ms/op
addElementsToHashSet | objectsHash()       avgt   25  5.638 ± 0.172  ms/op
```

Die Begründung ist wohl darin zu finden, dass zum einen durch den Vararg-Parameter von `Objects.hash(Object...)` jeweils im Hintergrund ein „Parameter-Array“ erzeugt werden muss, zum anderen jeder primitive Datentyp (wie hier `boolean`, `byte` oder `double`) in seine entsprechende Wrapper-Klasse (hier `Boolean`, `Byte` und `Double`) verpackt werden muss. So bequem die `hash`-Methode auch sein mag, für Heavy-Use-Anwendungen sehe ich persönlich von ihr ab. Für sporadische Anwendungen oder einfach nur aus Gründen der Formalität (damit der Vertrag mit `equals` eingehalten ist) ist sie aber eine durchaus valable Alternative.

Zuletzt wollen wir anschauen, ob sich das Cachen des Hash-Codes für unveränderbare (immutable) Objekte lohnt. Dazu darf in `ValueClass` die Zeile `cachedHashCode = directHashCode();` im Konstruktor nicht mehr auskommentiert sein, und die Methode `hashCode` muss nun die Zeile `return cachedHashCode;` aktiviert haben. Der Benchmark liefert daraufhin die Antwort:

```
Benchmark                                Mode  Cnt  Score   Error  Units
addElementsToHashSet | cachedHashCode    avgt   25  1.823 ± 0.035  ms/op
```

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

@State(Scope.Benchmark)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class HashCodeBenchmark {

  private static final int ELEMENT_COUNT = 100_000;
  private List<ValueClass> randomElementList
     = new ArrayList<>(ELEMENT_COUNT);
  private Set<ValueClass> randomElementSet
     = new HashSet<>(ELEMENT_COUNT);

  public HashCodeBenchmark() {
    for (int i = 0; i < ELEMENT_COUNT; i++) {
      randomElementList.add(ValueClass.newRandomInstance());
    }
  }

  /* [...] Main method omitted. */

  @Benchmark
  public void addElementsToHashSet() {
    randomElementSet.addAll(randomElementList);
  }
}
```

Listing 8: HashCodeBenchmark

Mit einem nennenswerten Geschwindigkeitsvorteil ist es nicht weit her. Dies mag allerdings auch der Tatsache geschuldet sein, dass es sich bei diesen wenigen, primitiven Datentypen sowieso um eine hocheffiziente Hash-Code-Berechnung handelt. Werden die einzubeziehenden Attribute komplexer und die Berechnungsmethode in der Folge darauf aufwendiger, dann kann sich durchaus ein messbares Resultat zeigen. Der guten Ordnung halber sei aber auch hier an die Warnung von Joshua Bloch erinnert, dass „Spielereien" wie *Lazy Initialization* und dergleichen meist nur zu schlechter lesbarem Code mit wenig Performanzgewinn führen, dafür aber das Risiko für Programmfehler, vor allem in Multi-Thread-Umgebungen, erheblich steigt [Blo83].

Operationen als Parameter

Das letzte Alltagsbeispiel ist nicht weniger relevant, wird dem Leser aber als Aufgabe zum Selber-Ausprobieren überlassen. Die kompletten Code-Beispiele finden Sie unter [Code]. Im `OperationBenchmark`, der aus Platzgründen weder abgedruckt noch detailliert erörtert werden kann, geht es darum, wie einer Berechnungsmethode die vorzunehmende Operation angegeben werden kann. Zur Auswahl stehen hardcodierte Enums wie `OperationEnum.ADDITION`, Strings wie `"SUBTRACTION"` und die Übergabe einer Klasse als `class`-Objekt (z. B. `MultiplicationOperation.class`), deren Methode dann via Reflection aufgerufen wird.

Was auf den ersten Blick wie ein weit hergeholtes, theoretisches Szenario aussieht, hat heutzutage absolute Praxisrelevanz. Nicht wenige (selbstgestrickte) Frameworks und Persistenzlösungen verzichten auf die typsicheren Enums zugunsten typunsicherer Strings, weil sich letztere einfacher persistieren oder serialisieren lassen. Auch das Hantieren mit `class`-Objekten ist (zumindest hinter den Kulissen) bei Dependency-Injection-Frameworks gang und gäbe. Dass man sich aber mit derart „weichcodierter" Java-Programmierung von Typsicherheit und Nachvollziehbarkeit zusehends entfernt, liegt auf der Hand. Welchen Einfluss sie auf die Performanz hat, dürfen Sie im letzten Beispiel selber testen. Fühlen Sie sich frei, Ihre Beobachtungen oder Fragen mit mir zu teilen.

Fazit

Performanzanalysen auf dieser Detailstufe sind immer eine Gratwanderung. Im vorliegenden zweiten Teil dieser Artikelserie wurde demonstriert, wie einfach und hochpräzise sich der im ersten Teil eingeführte JMH in der Praxis einsetzen lässt. Selbstgebastelte Stoppuhrmessungen dürften damit der Vergangenheit angehören. Weiter hat sich gezeigt, dass Bauchschätzungen in Sachen Performanz nicht immer richtig sind. Selbst ich als Autor hatte das eine oder andere „Aha-Erlebnis", wenn ich mit meinen ursprünglichen Vermutungen daneben lag.

Im Zentrum stand immer die Frage nach der hier zu vermittelnden Kernbotschaft. Und diese ist klar, bestätigt sich doch die Grundprämisse, dass es sich in Java nicht lohnt, aufgrund irgendwelcher Performanzvorteile den Code zu „verbiegen" oder die Sprache zu „vergewaltigen". Ein gesunder Einsatz der richtigen Sprachmittel für den richtigen Einsatzzweck (zum Beispiel ein `StringBuilder` statt `+`-Operatoren), ohne auf Kosten der Les- und Wartbarkeit zu optimieren, ist mit Sicherheit die nachhaltigste Lösung. Vor diesem Hintergrund beantworten sich dann auch die exemplarisch aufgezeigten Fragen wie „Schleifen oder Streams?", „Hash-Code selbst schreiben oder delegieren?" oder „Cachen oder nicht?" von selbst.

Und ja, so viel sei der Lösung der letzten Aufgabe vorweggenommen: Es ist durchaus sinnvoll, den voreiligen Einsatz von Dependency-Injection-Frameworks hier und da auch einmal zu hinterfragen ;-)

Literatur und Links

[Blo10/11/45/83] J. Bloch, Effective Java, 3rd Edition, Items 10/11/45/83, Addison-Wesley, 2018

[BlogEqualsAndHashCode] C. Heitzmann, A New and Consistent Way of Implementing equals and hashCode | Part 1, SimplexaCode Software Blog, https://link.simplexacode.ch/zxxx

[BlogJDKsSourceCode] C. Heitzmann, Using JDK's Source Code to Clarify Implementation Specifics, SimplexaCode Software Blog, https://link.simplexacode.ch/jjrk

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

[Hei19] C. Heitzmann, Theorie und Praxis von Zufallszahlengeneratoren, in: JavaSPEKTRUM, 2/2019

[Hei20-a] C. Heitzmann, Locking in Java, in: JavaSPEKTRUM, 2/2020

[Hei20-b] C. Heitzmann, Performanzanalysen in Java – Teil 1: Java Microbenchmarks, in: JavaSPEKTRUM, 4/2020

[JAPIObject] Java Platform, Standard Edition 8 API Specification, Class Object, https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

[JAPIObjects] Java Platform, Standard Edition 8 API Specification, Class Objects, https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html

[JAPIStringBuilder] Java Platform, Standard Edition 8 API Specification, Class StringBuilder, https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html

[JLS15181] Java Language Specification, Section 15.18.1: String Concatenation Operator +, https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.18.1

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

[SieBatRob9] K. Sierra, B. Bates, E. Robson, OCP Java SE 8 Programmer II Exam Guide, Chapter 9: Streams, Oracle Press, 2018