Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

InIT Institut für angewandte Informationstechnologie

Bachelorarbeit Informatik

Best-Practices für performante Java-Programmierung [Software Engineering]

Autoren	Marco De Tomasi Markus Rutz
Hauptbetreuung	Dr. Mark Cieliebak Prof. Dr. Markus Thaler

Datum	07.06.2016

Erklärung betreffend das selbständige Verfassen einer Bachelorarbeit an der School of Engineering

Mit der Abgabe dieser Bachelorarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Bachelorarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:	Unterschriften:
…………………………………	………………………………………………………...
	………………………………………………………
	………………………………………………………

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Bachelorarbeiten zu Beginn der Dokumentation nach dem Titelblatt mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Zusammenfassung

Qualitativ hochwertige Software muss vereinfacht ausgedrückt korrekt, effizient bzw. performant und wartungsfreundlich sein. Wann ist Software performant? Das Ziel dieser Bachelorarbeit ist, eine Sammlung von Best-Practices für die Entwicklung von effizienter Software zu finden. Dazu werden Code-Fragmente mit unterschiedlichen Implementierungsansätzen für das Lösen einer einzelnen Aufgabe gegenübergestellt.

Die Testumgebung für das Messen der Laufzeiten einzelner Codeabschnitte wurde aus einer früheren Diplomarbeit übernommen und angepasst. Um genaue Messresultate zu erhalten, wurde das Benchmarking-Framework JMH (Oracle) verwendet. Die Resultate der Testreihen wurden als einzelne Dateien abgespeichert und in einem weiteren Schritt in die Datenbank übernommen. Nach dem Importieren der Resultate in die Datenbank, konnten die Resultate als Balkendiagrammen dargestellt werden. Dies ermöglichte eine visuelle Auswertung der Laufzeiten.

Verschiedene Code-Fragmente wurden mit Hilfe der Testumgebung untersucht. Wie zeiteffizient ist beispielsweise das Löschen von mehrfach vorhandenen Werten aus Arrays oder Listen mit Hilfe von Streams? Bringt in diesem Fall die Nutzung von Streams einen Vorteil gegenüber dem Einsatz eines klassischen Programmieransatzes, also dem Umwandeln der beiden vorliegenden Datenstrukturen in ein Set? Wie sollte ein Stack implementiert werden, so dass dieser bei allen Operationen möglichst schnell ist?

Zum Beispiel führt das Löschen von mehrfach vorhandenen Werten in Listen mit Hilfe von Streams zu Performance-Einbussen. Mit der Konvertierung des Arrays zu einem Set wird eine tiefere Performance erreicht als mit Einsatz eines sequenziellen Streams. Bei der Verwendung eines Parallel-Streams ist die gemessene Laufzeit bei Arrays circa zwei und bei Listen circa drei Mal langsamer als die Verwendung von Sets. Ein als ArrayDeque implementierter Stack ist in allen relevanten Operationen schneller als ein Stack, der als LinkedList umgesetzt ist.

Abstract

Qualitative high-value software has to be in simple terms correct, efficient respectively performant and easy to maintain. The aim of this bachelor thesis is to find a collection of best practices for the development of efficient software. For this purpose, code fragments with different implementation approaches are confronted with each other for solving a single task.

The test environment for the runtime measurement of particular code snippets was taken from a previous thesis and has then been adapted. In order to obtain precise measurement results, the benchmarking framework JMH (Oracle) was used. The results of the performed test series were stored as individual files and in a further step imported into the database. Afterwards these results could be queried from the database to achieve the creation of bar charts. This allowed a visual analysis of the different runtimes.

Different code fragments were analyzed with the aid of the test environment. How time efficient is e.g. removing recurring values from an array or from a list by dint of a stream? Does in this case using a stream provide an advantage over using a classical programming approach, as converting the present data structures into a set? How should a stack be implemented so that all operations perform as fast as possible?

For example, the elimination of recurring values from lists with the aid of streams leads to performance penalties. By converting an array into a set, a lower performance will be achieved than by using a sequential stream. If a parallel stream is used, the measured runtime will be, in case of an array, about twice as slow as using sets. In case of a list, the parallel stream is about three times as slow as the usage of sets. A stack that is implemented as an ArrayDeque class performs faster on all relevant operations than a stack that is based on a linked list.

Danksagung

Wir möchten allen Personen danken, die uns bei unserer Arbeit unterstützt haben:

- Dr. Mark Cieliebak und Prof. Dr. Markus Thaler, die uns die Wahl dieses interessanten Themas ermöglicht haben. Wir danken ihnen für ihre freundliche und konstruktive Betreuung und Unterstützung über die gesamte Bearbeitungszeit der Bachelorarbeit.
- Stefania De Tomasi, die uns sprachliche Korrekturvorschläge gegeben hat.
- Unseren Freunden und Kommilitonen, die uns vor allem neben der Arbeit unterstützt haben.
- Unseren Familien, die uns das Studium überhaupt ermöglicht haben und uns dabei stets unterstützt haben.

Vielen Dank!

Marco De Tomasi, Markus Rutz

Inhaltsverzeichnis

1. Einleitung 8
1.1 Motivation 8
1.2 Problemstellung 9
2. Grundlagen 10
2.1 Development-Cycle einer Java-Applikation 10
2.2 Java Virtual Machine 10
2.3 Strategien des Java-Compilers zur Optimierung von Bytecode 13
2.4 Strategien des JIT-Compilers zur Optimierung von Bytecode 19
3. Setup 23
3.1 Benchmarks 23
3.2 Hard- und Software für Performance-Messungen 27
3.3 Details zur Testumgebung 27
3.4 Konzept für die Messung mit JMH 28
3.5 Wahl des Testdurchlaufs 30
4. Vergleiche verschiedener Code-Fragmente 32
4.1 Konzept: Multiplikation zweier Werte 34
4.2 Instanzvariablen im Vergleich zu lokalen Variablen 36
4.3 Kopieren eines Arrays 39
4.4 Initialisierung einer ArrayList 40
4.5 Performance von Integer-Arrays gegenüber Integer-ArrayLists 42
4.6 Autoboxing und Casting 44
4.7 Comparable-Interface oder Comperator 45
4.8 Unterschiedliche Definition der “Schleifen-Obergrenzen-Variable“ sowie der Zählvariable 48
4.9 Filtern von Werten im Array und in ArrayList 50
4.10 Function-Composition und Methoden-Verkettung 53
4.11 Fakultät berechnen 54
4.12 Mathematische Multiplikation und Division sowie Multiplikation oder Division mit Bitshift 58
4.13 Objekt zurücksetzen oder ein neues Objekt verwenden 60
4.14 Objekte kopieren 62
4.15 Polymorphismus 64
4.16 Mehrfach vorhandene Werte aus ArrayLists oder Arrays entfernen 66
4.17 Listen oder Arrays sortieren 68
4.18 Aufruf einer statischen Methode im Gegensatz zu einer nicht-statischen Methode 71

4.19 Die Verkettung von Strings 72
4.20 Das Vergleichen von Strings 74
4.21 Die Trennung von Strings 76
4.22 Summe aus den Werten eines Arrays oder einer Liste bilden 79
4.23 Switch-Case-Anweisung gegenüber If-Else-Kette 82
4.24 Die Switch-Case-Anweisung und die Datentypen der Testfälle 83
4.25 Synchronisationsarten im Vergleich 87
4.26 Operationen auf den Klassen ArrayList und LinkedList 90
4.27 Operationen auf den Klassen HashMap, LinkedHashMap und TreeMap 96
4.28 Operationen auf den Klassen HashSet, LinkedHashSet und TreeSet 105
4.29 Implementierung eines Stacks mit ArrayDeque und LinkedList 109
5. Diskussion und Ausblick 111
5.1 Zusammenfassung der Resultate 111
5.2 Weitere mögliche Messungen 112
6. Verzeichnisse 113
7. Anhang 120

1. Einleitung

1.1 Motivation

Als Software-Entwicklerin oder Entwickler stellt man sich manchmal die Frage, ob durch geschickte Programmierung bei der späteren Ausführung des Programms Zeit eingespart werden kann. Eine einfache Codezeile wie die Addition zweier Zahlen und die Ausgabe des Resultats entspricht in der Programmiersprache Java vier CPU-Instruktionen (Rauh 2015). Ein moderner Prozessor wie zum Beispiel das Modell Core i7-6700 Skylake 3.4 GHz von Intel führt diese vier Instruktionen in knapp einer Nanosekunde aus, angenommen es wird nur ein Thread des Prozessors verwendet (Pavlov 2016). Meistens sind deswegen nur Codefragmente interessant, die mehrfach aufgerufen werden. Diese Aufrufe können dabei bewusst von der Software-Entwicklerin bzw. dem Software-Entwickler selbst ausgelöst werden. Wird hingegen ein Codefragment auf einem sehr stark genutzten Server bereitgestellt und dabei mehrfach durchlaufen, erzielt dies einen ähnlichen Effekt. Das sehr häufige Durchlaufen (Beispiel: 1 Millionen Mal) eines Codeabschnittes mit einer etwas längeren Laufzeit (Beispiel: 7.2 Millisekunden) führt unter Umständen zu ziemlich langen Laufzeiten des Programmcodes. Im Fall, dass dieser Codeabschnitt sequenziell ausgeführt wird, beträgt die Laufzeit einer Million Aufrufe dieses Codeabschnittes ganze zwei Stunden.

Wird davon ausgegangen, dass ein bestimmtes Codefragment genau N Mal ausgeführt wird, kann sich ein Unterschied in der Laufzeit dieses Codefragments auch N-fach auf die Laufzeit der N Aufrufe dieses Codefragments auswirken? Ob die Ausführungszeit genau das N-fache beträgt, ist direkt davon abhängig, welche Optimierungen durch den Compiler vor oder auch während der Ausführung des Programms vorgenommen werden. Werden beim Kompilieren keinerlei Optimierungen vorgenommen, ist davon auszugehen, dass die N-fache Ausführung des Codefragments auch die N-fache Laufzeit des Codefragments benötigen wird. Bei den meisten Compilern werden jedoch immer Optimierungen vorgenommen, um den Code so gut wie möglich zu optimieren und damit die Laufzeit zu senken. Somit darf nicht davon ausgegangen werden, dass die N-fache Ausführung des Codefragmentes auch die N-fache Laufzeit des Codefragmentes beträgt. Compiler optimieren den Code teilweise sehr stark.

Die Motivation besteht nun darin, unterschiedliche Codefragmente, die im Grunde das Gleiche bewirken, bei einer sehr hohen Anzahl an Wiederholungen in Bezug auf die Laufzeit zu untersuchen. Zu jedem Zeitpunkt, an dem eine Messung stattfindet, sollen alle Optimierungen des Compilers bereits durchgeführt worden sein.

1.2 Problemstellung

Auch bei der Entwicklung von Java-Applikationen sollte darauf geachtet werden, dass Codefragmente, die sehr oft ausgeführt werden, möglichst effizient programmiert sind. Im Rahmen dieser Bachelorarbeit werden alle Performance-Messungen in der Programmiersprache Java implementiert und durchgeführt.

Die erarbeitete Testumgebung aus der Bachelorarbeit "Best-Practices zur Performance-Optimierung bei der Software-Entwicklung in Java" von Manuel Simbürger und Roman Thöni (Simbürger/Thöni 2015) wurde als Grundlage verwendet und an einigen Stellen verbessert. Zusätzlich wurden die bereits implementierten und durchgeführten Tests und deren Resultate kritisch hinterfragt. Danach wurden sie teilweise etwas abgeändert oder teilweise auch komplett neu aufgesetzt.

Mit folgenden zwei Arten von Änderungen kann die Performance in Java-Code gesteigert und somit die Laufzeit verkürzt werden:

1. Verwendung eines effektiveren Codes, der schnellere Laufzeiten ermöglicht.
    a. Sollte bezüglich Performance eine for-Schleife, eine foreach-Schleife, ein Iterator oder ein ListIterator verwendet werden?
    b. Bieten Java Streams im direkten Vergleich mit häufiger eingesetzten Code-Fragmenten aus Java 7 (oder früheren Java-Versionen) eine bessere oder eine schlechtere Performance?
    c. Welche weiteren Implementierungsansätze mit unterschiedlichen Laufzeiten existieren?

2. Verwendung von Code, der vom Just-In-Time-Compiler oder vom Java-Compiler möglichst gut optimiert werden kann.
    a. Unter welchen Voraussetzungen optimiert der JIT-Compiler Schleifen?
    b. Ist es aus Performance-Sicht sinnvoll, Variablen mit dem Keyword final zu deklarieren?
    c. Welche zusätzlichen Optimierungen führt der JIT-Compiler durch?

Die vorliegende Bachelorarbeit beschäftigt sich mit diesen und ähnlichen Fragen und zeigt die daraus resultierenden Best Practices auf. Diese sollen Software-Entwicklerinnen und Entwicklern eine Hilfestellung bieten, um bei Entscheidungen über verschieden implementierte Codefragmente, die die gleiche Funktionalität umsetzen, das effizienteste Codefragment zu eruieren und danach dieses einsetzen zu können.

2. Grundlagen

2.1 Development-Cycle einer Java-Applikation

In diesem Abschnitt wird der Development-Cycle einer Java-Applikation aufgezeigt (Molloy 2006).

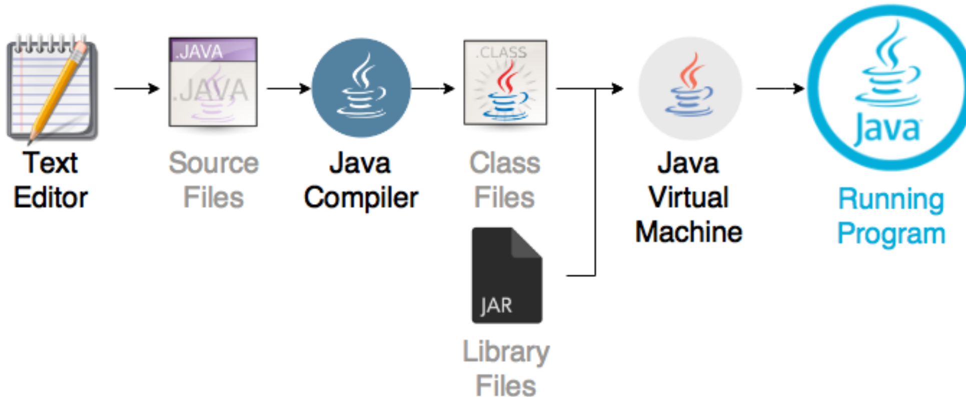

Abbildung 1: Java Development-Cycle (Eigene Abbildung)

Der Development-Cycle einer Java-Applikation beginnt wie bei praktisch jedem Programm im Text Editor, zum Beispiel in demjenigen einer IDE. Die Source-Dateien müssen dabei mit der Endung “.java“ abgelegt werden. Nach dem Programmieren der Source-Dateien können diese mit dem Java-Compiler zu Java-Bytecode übersetzt werden. Bei diesem Schritt werden durch den Java-Compiler bereits Optimierungen vorgenommen. Es werden zusätzlich auch die benötigten Java-Libraries eingebunden oder im Classpath deren Pfad ergänzt. Java-Bytecode sind Dateien mit der Endung “.class“ und sind nicht direkt, sondern indirekt über die Java Virtual Machine (JVM) ausführbar. Die Java Virtual Machine liest dabei den Bytecode ins Memory und übersetzt ihn während der Laufzeit in die lokale Maschinensprache. Die Hauptaufgabe der JVM ist folglich das Fungieren als Interpreter. Darauf wird im folgenden Kapitel später noch genauer eingegangen.

2.2 Java Virtual Machine

In diesem Abschnitt wird die Java Virtual Machine und deren Funktionalität beleuchtet (Oracle 2016 B). Beschreibungen bezüglich des Just-In-Time-Compilers beziehen sich auf die Hotspot-Variante des JIT-Compilers aus der Version Java Development Kit 8 (JDK8) von Oracle (Oracle 2016 E).

Die Java Virtual Machine (JVM) ist Voraussetzung für das Hardware- sowie Betriebssystem-unabhängige Ausführen von Java-Anwendungen. Sie ist ein sehr zentraler Bestandteil des Java Runtime Environments (JRE). Die JVM setzt direkt auf dem Host-Betriebssystem auf und ist selbst ein abstrakter Rechner, der über ein Instruction Set verfügt und verschiedene Speicherbereiche während der Laufzeit manipuliert. Der übersetzte Java-Bytecode besteht aus genau diesen JVM-Instructions, einer Symbol Table sowie ergänzenden Informationen. Die isolierte Ausführung von Bytecode in einer JVM soll Benutzer unter anderem

von Schadcode schützen. Die virtuelle Maschine ermöglicht ausserdem, dass der kompilierte Bytecode sehr kompakt bleibt.

Von der Java Virtual Machine können mehrere Instanzen gestartet werden. Jede zusätzliche Instanz wird in Form eines neuen Prozesses erstellt. Pro Instanz wird ein eigener JVM-Stack erzeugt. Dieser hält lokale Variablen und partielle Resultate. Bei Methoden-Aufrufen wird der Stack ebenfalls benötigt, beispielsweise für die Ablage von Rücksprungadressen. Der JVM-Heap existiert hingegen für alle vorhandenen JVM-Instanzen nur einmal. Auf dem Heap befinden sich pro Klasse Strukturen wie der Laufzeit-Konstanten-Pool, Variablen, Methoden-Daten, Code für die Methoden sowie Konstruktoren.

Bytecode-Interpreter

In diesem Abschnitt wird auf den Bytecode-Interpreter eingegangen (Wikibooks 2016).

Die Hauptaufgabe der JVM ist das Interpretieren des Java-Bytecodes. Die JVM, sprich auch der Interpreter, wissen dabei aber nichts von der Java-Programmiersprache, sondern kennen nur das Datei-Format ".class", das Bytecode enthält (Oracle 2016 B). Bei der Interpretation soll dieser Bytecode für die konkrete Kombination von Hardware und Betriebssystem in nativen Code umgesetzt und ausgeführt werden. Die alte Java-Architektur arbeitete ausschliesslich als Interpreter. Die permanente Umsetzung von Bytecode zu nativem Code hat jedoch einen entscheidenden Nachteil: Dieser Prozess benötigt viel CPU-Rechenleistung und ist sehr zeitaufwändig.

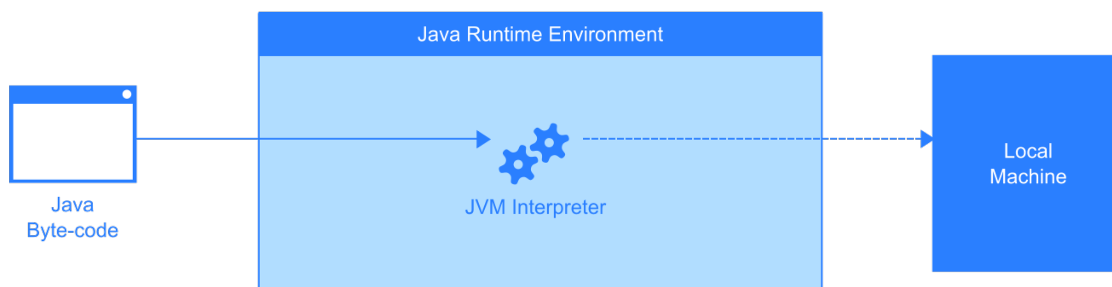

Abbildung 2: Bytecode-Interpreter (Wikibooks 2016)

Just-In-Time-Compiler

In diesem Abschnitt wird auf den Just-In-Time-Compiler eingegangen (Wikibooks 2016).

Ab Version 1.2 von JRE wird eine robustere und effizientere JVM verwendet. Anstatt den Bytecode zu interpretieren, wird der Bytecode in einen nativ ausführbaren Code für das jeweilige System umgesetzt. Dieser Prozess wird als Just-In-Time-Compilation bezeichnet und vom Just-In-Time-Compiler (JIT-Compiler) durchgeführt. Der JIT-Compiler führt diese Kompilierung jedoch nur beim ersten Ausführen des Bytecodes aus. Danach wird die zuvor kompilierte Version des Bytecodes verwendet (Bypass Compilation). Sobald Änderungen am Bytecode entstehen, wird der entsprechende Code neu kompiliert. Der JIT-Compiler spart

gegenüber dem Interpreter eine Menge Zeit und benötigt deutlich weniger CPU-Rechenleistung. Beim erstmaligen Ausführen wird natürlich gegenüber darauffolgenden Aufrufen eine kleine Verzögerung auftreten.

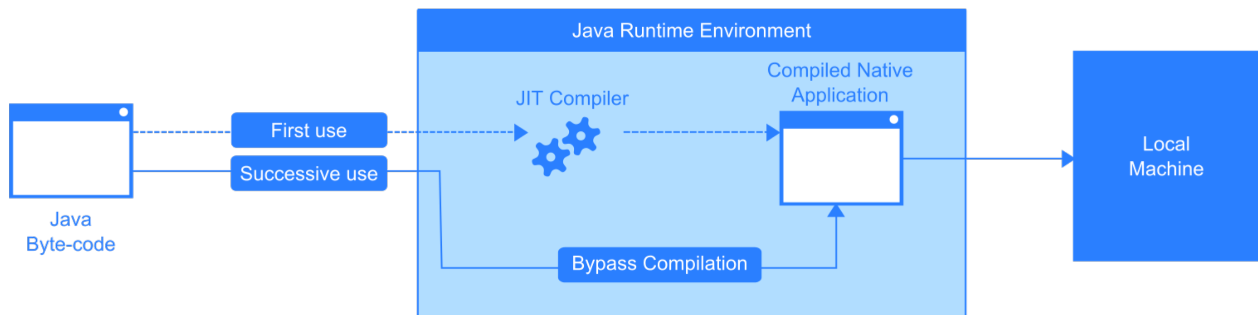

Abbildung 3: Kompilierung von Java-Bytecode innerhalb der JVM (Wikibooks 2016)

Neben einer Performance-Verbesserung durch die Bypass Compilation optimiert der JIT-Compiler auch mit verschiedenen anderen Strategien während der Laufzeit den Bytecode. Auf diese Optimierungsstrategien sowie auf diejenigen Optimierungsstrategien, die bereits vom Java-Compiler vorgenommen werden, wird im folgenden Kapitel näher eingegangen.

2.3 Strategien des Java-Compilers zur Optimierung von Bytecode

Der Java-Compiler javac wendet bei der Übersetzung von Java-Code zu Bytecode verschiedene Optimierungen an. Diese Optimierungen werden also vor der eigentlichen Laufzeit des Java-Programms vorgenommen und sind deswegen auch direkt im Bytecode ersichtlich. Einige Optimierungen, die der Java-Compiler anwenden kann, werden hier aufgelistet:

Abkürzung	Name
JAC.O1	Constant Folding
JAC.O2	Dead Code Elimination
JAC.O3	Boolesche Ausdrücke
JAC.O4	String-Verkettung
JAC.O5	Ersetzen einer final-Variable durch ein Literal

Tabelle 1: Optimierungsstrategien des Java-Compilers

Diese Optimierungen sind vom Brian Gordon Blog (Gordon 2014) entnommen worden. Gewisse Optimierungen wurden mit eigenen Beispielen ergänzt. Alle aufgelisteten Optimierungen werden im Folgenden genauer beschrieben.

JAC.O1 – Constant Folding

Angenommen wird, dass sich in einer Klasse die folgende statische Klassenvariable befindet:

```
private static int SECONDS_IN_30_DAYS = 60*60*24*30;
```

Abbildung 4: Resultat einer Multiplikation wird einer statischen Variable zugewiesen (Gordon 2016)

In der Klassenvariable SECONDS_IN_30_DAYS wird die Anzahl Sekunden von 30 Tagen als Multiplikation von Integer-Werten abgespeichert. Dies ist besser lesbar als wenn die Zahl 2’592’000 angegeben wird. Im folgenden Bytecode-Ausschnitt ist ersichtlich, dass bei der Übersetzung zu Bytecode javac 60*60*24*30 berechnet und die Zahl 2’592’000 in den Constant-Pool der “.class”-Datei schreibt:

```
0: ldc          #5                  // int 2592000
2: putstatic    #3                  // Field SECONDS_IN_30_DAYS:I
5: return
```

Abbildung 5: Bytecode zur Initialisierung einer statischen Variable (Gordon 2016)

Dieser Code-Ausschnitt befindet sich im Static-Initializer der “.class”-Datei. Wenn dieser Code später durchlaufen wird, lädt der Java-Interpreter mithilfe der Instruktion ldc direkt die Konstante aus dem Constant-

Pool in den Stack, ohne dabei eine Multiplikation von Integer-Werten durchzuführen. Dies bedeutet wiederum, dass zur Laufzeit keine Multiplikation stattfindet.

Wie sieht es nun aus, wenn die statische Variable eine boolean-Variable ist? Wertet javac in diesem Fall den booleschen Ausdruck aus? Folgende Codezeile ist gegeben:

```
private static boolean troo = true && (false || false || (true && true));
```

Abbildung 6: Einer Variable wird eine konstante logische Aussage zugewiesen (Gordon 2016)

Der resultierende Bytecode ist der Folgende:

```
0: iconst_1
1: putstatic     #3                  // Field troo:Z
4: return
```

Abbildung 7: Bytecode mit der Initialisierung der statischen, booleschen Variable (Gordon 2016)

In diesem Fall wird eine noch bessere Optimierung gemacht, weil es eine spezielle Bytecode-Instruktion gibt, die den Integer-Wert 1 lädt. Diese Bytecode-Instruktion heisst iconst_1. Dies führt dazu, dass gar nicht auf den Constant-Pool zugegriffen werden muss.

JAC.O2 – Dead Code Elimination

Wenn Code unerreichbar ist, wird dieser nach der Übersetzung von Code zu Bytecode eliminiert. Siehe folgendes Beispiel:

```
public static void main(String[] args) {
    if(false) {
        System.out.println("The universe is broken.");
    }
}
```

Abbildung 8: Unerreichbarer Java-Code (Gordon 2016)

Der Bytecode, der aus dem obigen Java-Code entsteht, sieht so aus:

```
public static void main(java.lang.String[]);
  Code:
     0: return
```

Abbildung 9: Bytecode des unerreichbaren Java-Codes (Gordon 2016)

An folgendem weiteren Beispiel kann eindrücklich aufgezeigt werden, dass der Java-Compiler Code gänzlich wegoptimieren kann. In einer Klasse existiert eine private-Methode, die in der Klasse jedoch nirgends aufgerufen wird. Diese Java-Klasse könnte in etwa so aussehen:

```
public class Test {
    public static void main(String[] args) {

    }

    private int calc() {
        int a = 3, b = 5;
        return a + b;
    }
}
```

Abbildung 10: Eine nicht verwendete private-Methode (Eigene Abbildung)

Der erstellte Bytecode von javac sieht so aus:

```
Compiled from "Test.java"
public class Test {
  public Test();
    Code:
       0: aload_0
       1: invokespecial #8                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: return
}
```

Abbildung 11: Die ".class"-Datei ohne die private-Methode (Eigene Abbildung)

JAC.O3 – Boolesche Ausdrücke

Gemäss der De Morganschen Gesetzen haben die folgenden beiden If-Verzweigungen die gleiche logische Bedeutung:

```
if(!a || !b) {
    doSomething();
}
```

Abbildung 12: Erste Variante der If-Verzweigung (Gordon 2016)

```
if(!(a && b)) {
    doSomething();
}
```

Abbildung 13: Zweite Variante der If-Verzweigung (Gordon 2016)

In den nächsten Grafiken ist ersichtlich, dass javac für beide Codestücke den gleichen Bytecode erstellt:

```
19: iload_2
20: ifeq          27
23: iload_3
24: ifne          33
30: invokevirtual #6
```

Abbildung 14: Bytecode der ersten If-Verzweigung (Gordon 2016)

```
19: iload_2
20: ifeq          27
23: iload_3
24: ifne          33
30: invokevirtual #6
```

Abbildung 15: Bytecode der zweiten If-Verzweigung (Gordon 2016)

JAC.O4 – String-Verkettung

Viele Leute erzählen, dass die String-Verkettung mit + ein Performance-Killer ist. Sie werden wahrscheinlich die Nutzung des StringBuilders empfehlen. Ist dies wirklich Best Practice?

In der Realität ist es so, dass javac jede String-Verkettung mit + in die Anweisung StringBuilder.append umsetzt. In den folgenden Grafiken sind der Java-Code und der jeweilige Bytecode dargestellt:

```
return str1 + " : " + str2;
```

Abbildung 16: String-Verkettung in einem Statement (Gordon 2016)

```
11: new           #3                  // class java/lang/StringBuilder
14: dup
15: invokespecial #4                  // Method java/lang/StringBuilder."<init>":()V
18: aload_1
19: invokevirtual #5                  // Method java/lang/StringBuilder.append:(Ljava/lan
22: ldc           #6                  // String  :
24: invokevirtual #5                  // Method java/lang/StringBuilder.append:(Ljava/lan
27: aload_2
28: invokevirtual #5                  // Method java/lang/StringBuilder.append:(Ljava/lan
31: invokevirtual #7                  // Method java/lang/StringBuilder.toString:()Ljava/
```

Abbildung 17: Bytecode der String-Verkettung in einem Statement (Gordon 2016)

Diese Umsetzung ist nicht perfekt, da für jedes Statement ein neues StringBuilder-Objekt erzeugt wird. In der folgenden Abbildung 19 sieht man zwei new-Instruktionen (Zeile 8 und 35).

```
String cat = str1 + " : " + str2;
return cat + " 123";
```

Abbildung 18: Java-Code mit zwei String-Verkettungszeilen (Gordon 2016)

```
 8: new           #2                  // class java/lang/StringBuilder
11: dup
12: invokespecial #3                  // Method java/lang/StringBuilder."<init>":()V
15: aload_1
16: invokevirtual #4                  // Method java/lang/StringBuilder.append:(Ljava/lar
19: ldc           #5                  // String  :
21: invokevirtual #4                  // Method java/lang/StringBuilder.append:(Ljava/lar
24: aload_2
25: invokevirtual #4                  // Method java/lang/StringBuilder.append:(Ljava/lar
28: invokevirtual #6                  // Method java/lang/StringBuilder.toString:()Ljava/
31: astore_3
35: new           #2                  // class java/lang/StringBuilder
38: dup
39: invokespecial #3                  // Method java/lang/StringBuilder."<init>":()V
42: aload_3
43: invokevirtual #4                  // Method java/lang/StringBuilder.append:(Ljava/lar
46: ldc           #8                  // String  123
48: invokevirtual #4                  // Method java/lang/StringBuilder.append:(Ljava/lar
51: invokevirtual #6                  // Method java/lang/StringBuilder.toString:()Ljava/
```

Abbildung 19: Bytecode mit zwei Aufrufen des StringBuilder-Konstruktors (Gordon 2016)

Gerade bei String-Verkettungen mit + innerhalb einer Schleife erzeugt dieser Umstand einen unnötigen Overhead. In diesem Fall ist es also Best Practice, vor der Schleife einen StringBuilder zu initialisieren und bei jeder Iteration der Schleife die String-Verkettung mit StringBuilder.append vorzunehmen.

Im nächsten Fall werden String-Konstanten in einem Statement verkettet.

```
return "We the People of the United States, in Order to form a more perfect "
     + "Union, establish Justice, insure domestic Tranquility, provide for the "
     + "common defence, promote the general Welfare, and secure the Blessings "
     + "of Liberty to ourselves and our Posterity, do ordain and establish this "
     + "Constitution for the United States of America.";
```

Abbildung 20: Verkettung von String-Konstanten in einem Statement (Gordon 2016)

In diesem Fall verkettet javac die Strings direkt und speichert einen einzigen String in den Constant-Pool. Der Bytecode-Interpreter lädt nämlich mit ldc den String aus dem Constant-Pool in den Stack.

```
0: ldc           #2                  // String We the People of the United States, in Ord
```

Abbildung 21: Ein String in den Stack laden (Gordon 2016)

JAC.O5 – Ersetzen einer final-Variable durch ein Literal

Konstante Klassenvariablen sollten in Java final deklariert werden. Dies hilft der Entwicklerin und dem Entwickler die mehrfache Zuweisung eines Wertes an diese Variablen zu unterbinden. Zudem optimiert Javac den Einsatz der final-Variable, indem auf den entsprechenden Wert im Code ein Inlining angewandt wird. Angenommen folgender Java-Code ist vorhanden:

```java
private static int SECONDS_IN_30_DAYS = 60*60*24*30;

public static void main(String[] args) {
    System.out.println(SECONDS_IN_30_DAYS);
}
```

Abbildung 22: Der Wert einer statischen Variable auf der Konsole drucken (Gordon 2016)

Der entsprechende Bytecode sieht folgendermassen aus:

```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStr
     3: getstatic     #3                  // Field SECONDS_IN_30_DAYS:I
     6: invokevirtual #4                  // Method java/io/PrintStream.println:(I)V
     9: return

static {};
  Code:
     0: ldc           #5                  // int 2592000
     2: putstatic     #3                  // Field SECONDS_IN_30_DAYS:I
     5: return
```

Abbildung 23: Static Variable wird im static-Block initialisiert. Wert wird mit getstatic geholt. (Gordon 2016)

Wenn die Variable SECONDS_IN_30_DAYS final ist, wird im Bytecode auf die Initialisierung im static-Block verzichtet. In der main-Methode wird statt der Instruktion getstatic die Instruktion ldc eingesetzt, um im Code direkt ein Integer-Literal einzusetzen. In den Abbildungen 24 und 25 sind der Java-Code und der Bytecode aufgeführt.

```java
private static final int SECONDS_IN_30_DAYS = 60*60*24*30;

public static void main(String[] args) {
    System.out.println(SECONDS_IN_30_DAYS);
}
```

Abbildung 24: SECONDS_IN_30_DAYS ist hier static final (Gordon 2016)

```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStr
     3: ldc           #3                  // int 2592000
     5: invokevirtual #4                  // Method java/io/PrintStream.println:(I)V
     8: return
```

Abbildung 25: Inlining einer static final Variable durch den Einsatz der Instruktion ldc (Gordon 2016)

2.4 Strategien des JIT-Compilers zur Optimierung von Bytecode

Die Java Virtual Machine führt während der Laufzeit eines Programms Statistiken. Diese Statistiken beziehen sich unter anderem auf die Häufigkeit der Ausführung eines bestimmten Codestückes. Wenn das bestimmte Codestück häufig ausgeführt wird, deutet dies darauf hin, dass das Codestück eine beträchtliche Relevanz bezüglich der Performance des Programms haben kann. Für die Java Virtual Machine ist dies ein guter Grund, den JIT-Compiler anzustossen. Wenn der JIT-Compiler ein Bytecode kompilieren muss, führt dieser dabei gewisse Optimierungen durch. Welche Optimierungen der JIT-Compiler durchführt, ist von der Kompilierungsstufe abhängig, in der sich eine Methode befindet (Oaks 2014, 102).

Der JIT-Compiler führt zwei Arten von Optimierungen durch (Oaks 2014, 88). Die eine Art ist die Standard-Optimierung. Dabei wird die ganze Methode optimiert und vor der nächsten Ausführung der neue Code aus dem Speicher geladen. Bei der zweiten Art, der On-Stack-Replacement-Optimierung, wird ein Teil des Codes einer Methode optimiert und während der Ausführung dieser Methode der entsprechende Stack-Bereich ersetzt. Einige Optimierungen, die der JIT-Compiler anwenden kann, werden hier aufgelistet:

Abkürzung	Name
JIT.O1	Constant Folding
JIT.O2	Dead Code Elimination
JIT.O3	Code Inlining
JIT.O4	Schleifen
JIT.O5	Copy Propagation

Tabelle 2: Optimierungsstrategien des Just-In-Time-Compilers

JIT.O1 – Constant Folding

Falls das Resultat einer Berechnung vorhersehbar und nicht von einer Zustandsvariable abhängig ist, wird es mit hoher Wahrscheinlichkeit vom JIT-Compiler optimiert. Ob der JIT-Compiler eine Optimierung durchführt, ist von der konkreten Berechnung abhängig. Um dies zu veranschaulichen, wird dieses Beispiel analysiert:

```
1  private double x = Math.PI;
2
3  @Benchmark
4  public double testWrong()
5  {
6      return Math.sin( Math.PI );
7  }
8
9  @Benchmark
10 public double testRight()
11 {
12     return Math.sin( x );
13 }
```

Abbildung 26: Input-Werte sollen, wenn möglich, von Instanzvariablen gelesen werden (Vorontsov 2014).

Es gibt zwei Benchmarks: Im ersten Benchmark testWrong wird der Sinus-Wert von der Konstante Pi der Klasse Math berechnet und zurückgegeben. Im zweiten Benchmark wird der Sinus-Wert einer Instanzvariable berechnet und zurückgegeben. In diesem Fall entspricht die Instanzvariable dem Wert der Konstante Pi der Klasse Math.

Obwohl der JIT-Compiler die Methode Math.sin bei testRight() sowie bei testWrong() nicht wegoptimiert, die Berechnung immer wieder durchführt und somit auch die Laufzeit richtig ermittelt, ist der Benchmark-Stil der ersten Methode schlecht. Wenn beispielsweise Math.sin durch Math.log ersetzt wird, gibt es in testWrong eine Optimierung und die Methode gibt irgendwann nur noch eine Konstante zurück (Vorontsov 2014). Da es sehr schwierig ist zu wissen, welche Methoden der JIT-Compiler nicht optimiert, ist es am besten, wenn Input-Werte von Instanzvariablen gelesen werden.

JIT.O2 – Dead Code Elimination

Der JIT-Compiler kann, wie auch der Java-Compiler, "toten" Code eliminieren. Der Java-Compiler kann dies nur sehr limitiert. Der JIT-Compiler deckt hingegen mehr Fälle ab. Wenn beispielsweise eine Methode keinen Wert zurückgibt (void-Methode) und innerhalb der Methode nur lokale Variablen mutiert werden, kann dieser Code vom JIT-Compiler gänzlich wegoptimiert werden. Besitzt eine Methode hingegen einen Rückgabewert, wird dieser im Grunde meistens auch durch den aufrufenden Code verwendet. Sollte der Rückgabewert aber nicht verwendet werden, optimiert der JIT-Compiler diesen Aufruf auch weg (Vorontsov 2014).

JIT.O3 – Code Inlining

Abhängig von den konkreten Umständen wendet der JIT-Compiler auf den Code einer Methode Inlining an. Dies bedeutet, dass der Methodenaufruf durch den effektiven Code der Methode ersetzt wird. Damit wird die Latenz des Methodenaufrufs vermieden. Diese Optimierung macht insbesondere bei kleinen Methoden Sinn, da der Overhead für den Methodenaufruf in diesem Fall sehr gross ist. Klassische Beispiele für kleine Methoden sind Setter- und Getter-Methoden, bei denen eine Instanzvariable auf einen beistimmten Wert gesetzt wird oder der Wert einer Instanzvariablen abgefragt wird. Angenommen, folgende Klasse ist vorhanden:

```
public class Point {
    private int x, y;

    public void getX() { return x; }
    public void setX(int i)  { x = i; }
}
```

Abbildung 27: Die Klasse Point (Oaks 2014, 96)

Diese Klasse Point wird nun im nachfolgenden Codestück instanziiert. Zusätzlich wird mit der Getter-Methode der Wert abgefragt, mit 2 multipliziert und mit der Setter-Methode wieder der Instanzvariable zugewiesen:

```
Point p = getPoint();
p.setX(p.getX() * 2);
```

Abbildung 28: Instanziierung von Point und Einsatz von Getter- und Setter-Methoden (Oaks 2014, 96)

Der Code, den der JIT-Compiler in Anbetracht des Code-Inlinings kompiliert, ist der Folgende:

```
Point p = getPoint();
p.x = p.x * 2;
```

Abbildung 29: Optimierung des JIT-Compilers als Java-Code dargestellt (Oaks 2014, 96)

Der JIT-Compiler entscheidet auf Basis verschiedener Kriterien, ob auf den Code einer Methode ein Inlining angewendet werden soll. Wird eine Methode sehr viele Male aufgerufen und der Bytecode ist kleiner als 325 Bytes, führt dies dazu, dass der JIT-Compiler auf den entsprechenden Code ein Inlining anwendet. Wenn die Methode wenig aufgerufen wird, aber der Bytecode kleiner als 35 Bytes gross ist, wird auf den Code dieser Methode trotzdem ein Inlining angewandt (Oaks 2014, 96). Es kann aber auch weitere Fälle geben, die nicht dokumentiert sind, bei denen der JIT-Compiler auf bestimmten Code ebenfalls Inlining anwenden kann.

JIT.O4 – Schleifen

In Benchmarks sind Schleifen heikle Konstrukte, weil der JIT-Compiler diese mit hoher Wahrscheinlichkeit optimieren kann. Damit eine Schleife nicht optimiert werden kann, müssen in der Schleife ein oder mehrere Statements vorhanden sein, die der JIT-Compiler nicht umgehen oder nicht durch kostengünstigere Operationen ersetzen kann.

JIT.O5 – Copy Propagation

Die Optimierung "Copy Propagation" findet statt, wenn der JIT-Compiler eine unnötige Variable durch eine andere ersetzt. Als Beispiel wird folgende Ausgangslage betrachtet:

```
public  void newMethod() {
     y = b.value;
     ...do stuff...
     z = y;
     sum = y + z;
}
```

Abbildung 30: Von y nach z kopieren

In diesem Fall ist ersichtlich, dass eine Summe aus y und z gebildet wird und in der vorherigen Zeile y der Variable z zugewiesen wird. Die Variablen y und z sind beide lokale Variablen. Da in diesen zwei Codezeilen klar ersichtlich ist, dass y und z gleich sein müssen, ist bei der Berechnung der Summe die Variable z eine unnötige Variable. Sie wird deswegen mittels Copy Propagation wegoptimiert. Der daraus entstehende Code sieht so aus:

```
public  void newMethod() {
    y = b.value;
    ...do stuff...
    y = y;
    sum = y + y;
}
```

Abbildung 31: Die Variable z wurde hier durch y ersetzt.

Der JIT-Compiler führt jeweils nicht nur eine Optimierung durch. Nach der Copy Propagation würde der JIT-Compiler im Code der Abbildung 31 in einem weiteren Optimierungs-Schritt beispielsweise die Zeile y=y löschen, weil sie nichts bewirkt (siehe JIT.O2).

3. Setup

3.1 Benchmarks

Wichtige Aspekte beim Durchführen eines Benchmarks

Beim Durchführen von Messungen in der Java-Laufzeitumgebung (JRE) müssen unter anderem folgende Aspekte in Betracht gezogen werden:

- Wie wird mit den durchgeführten Optimierungen des Java-Compilers umgegangen?
- Wie wird mit den durchgeführten Optimierungen des Just-In-Time-Compilers umgegangen?
- Welche Testumgebung wird verwendet?
- Verwendete Hard- und Software sowie deren Konfiguration (siehe Kapitel 3.2)

Wichtig ist zudem, dass bei Benchmarks generell gilt: Je geringer die Laufzeit, desto weniger zuverlässig ist die Aussagekraft der Messung.

- **Milli**benchmarks are not really hard
- **Micro**benchmarks are challenging, but OK
- **Nano**benchmarks are the damned beasts!
- **Pico**benchmarks...

(Shipilëv 2013, 19)

Umgang mit Optimierungen eines Compilers im Allgemeinen

Optimierungen an sich müssen nicht verhindert werden, da sie in der produktiven Umgebung auch durchgeführt werden. Im Gegenteil: Diese führen meist zu einer kürzeren Laufzeit und dienen somit der Effizienzsteigerung. Als Folge davon muss beim Auswerten von Benchmark-Resultaten jedoch auch immer damit gerechnet werden, dass Code vom Compiler optimiert worden ist. Zwingend ist also eine sorgfältige Interpretation der Benchmark-Resultate unter Berücksichtigung der vorgenommenen Optimierungen.

Umgang mit Optimierungen des Java-Compilers

Der Java-Compiler nimmt bei der Übersetzung von Source-Code zu Bytecode immer die bekannten Optimierungen (siehe Kapitel 2.3) vor. Die Möglichkeit, diese Optimierungen zu deaktivieren, ist nicht vorgesehen und besteht somit nicht (Oracle 2016 C).

Umgang mit Optimierungen des Just-In-Time-Compilers

Es besteht hingegen die Möglichkeit, den JIT-Compiler zu deaktivieren. Dies würde das spätere Auswerten der Messergebnisse eventuell vereinfachen. Ist der JIT-Compiler deaktiviert, liest der Interpreter nämlich Bytecode per Bytecode ein und führt diesen direkt aus, ohne dabei selber noch zusätzliche Optimierungen

(siehe Kapitel 2.4) vorzunehmen. Optimierungen würden in diesem Szenario ausschliesslich vom Java-Compiler beim Übersetzen von Source-Code zu Bytecode (im Schritt zuvor) vorgenommen. Da die Messungen jedoch innerhalb einer Messumgebung, die möglichst ähnlich wie eine produktive Umgebung aufgesetzt ist, stattfinden sollen, scheidet die Möglichkeit aus, den JIT-Compiler zu deaktivieren.

Deshalb ist es nötig, die Benchmark-Resultate unter Berücksichtigung der vom Compiler vorgenommenen Optimierungen zu analysieren.

Um bei der Laufzeit-Ermittlung von sehr oft durchlaufenem Code Messwerte zu erhalten, die bereits ab dem ersten Messwert eine gewisse Konstanz aufweisen, sollte vor dem Durchführen jeder Messungen derselbe Code bereits einige tausend Male ausgeführt worden sein. Dies stellt sicher, dass der Hotspot-Compiler den entsprechenden Code bereits vollständig kompiliert und optimal profiliert hat. Intelligente Optimierungs-entscheidungen wurden vom Hotspot-Compiler bereits getroffen und umgesetzt (OpenJDKWiki 2012).

Eigene Implementierung der Messungen

Die Laufzeit von Java-Code kann mit den üblichen Hilfsmitteln, die von der Java-Laufzeitumgebung (JRE) zur Verfügung gestellt werden, ermittelt werden (Oracle 2016 A). Dies ist einerseits die Methode System.currentTimeMillis(), welche die gemessene Zeit in Bezug auf den 1.1.1970 00:00 (UTC) Uhr darstellt. Andererseits ist es die Methode System.nanoTime(), welche die Systemzeit vom genauesten zur Verfügung stehenden System-Zeitgeber in Nanosekunden zurückgibt.

Die Methode System.currentTimeMillis() gibt möglicherweise ungenaue Werte zurück. Die Ungenauigkeit kann auftreten, sobald die Messdauer weniger als 20 Millisekunden beträgt und deswegen ist dieser Ansatz nur bedingt für Performance-Messungen zu empfehlen (Gräsel 2012).

Die Genauigkeit der Methode System.nanoTime() ist hingegen systemabhängig. Der Bezugszeitpunkt ist unbekannt. Deswegen kann diese Methode nur zur Berechnung von verstrichener Zeit genutzt werden. Da der genaueste zur Verfügung stehende System-Zeitgeber verwendet wird, ist es jedoch die beste derzeit zur Verfügung stehende Möglichkeit.

Eine Umsetzung der Implementierung unter Verwendung der üblichen Java-Hilfsmittel würde folgender-massen aussehen:

```
public void performanceTest() {
    final long timeStart = System.nanoTime();
    // ------------------------------------------------
    // --- [Code, dessen Laufzeit gemessen werden soll] ---
    // ------------------------------------------------
    final long timeEnd = System.nanoTime();

    long timePassedInNs = timeEnd - timeStart;
    System.out.println("Laufzeit des Tests: " + timePassedInNs + " Nanosek.");
}
```

Verwendung eines Frameworks für die Messungen

Neben der Möglichkeit der Verwendung einer eigenen Implementierung, stehen im Internet verschiedene Frameworks zur Messung der Laufzeiten von Java-Code zur Verfügung. Eines davon ist JMH (Java Microbenchmarking Harness). Es hebt sich von der Konkurrenz vor allem in dem Punkt ab, dass es von den gleichen Entwicklern von Oracle stammt, die auch den JIT-Compiler implementiert haben (Vorontsov 2014). Das JMH-Framework umgeht die gängigen Fallstricke, die bei der HotSpot-VM und bei der OpenJDK-VM auftreten können (Shipilëv 2013, 16). JMH ist nützlich für alle Arten von Microbenchmarks: Von solchen im Nanosekunden-Bereich bis hin zu solchen im Sekunden-Bereich (Vorontsov 2014).

Um dem JMH-Framework mitzuteilen, welcher Code ausgeführt werden soll, wird jedes Codefragment, dessen Laufzeit ermittelt werden soll, in einer Methode mit der @Benchmark-Annotation eingebettet. Diese Methode wird dann vom JMH-Framework nach entsprechender Konfiguration selbstständig aufgerufen.

Beim Einsatz des JMH-Frameworks sollten einige wichtige Punkte beachtet werden:

- Um Dead-Code-Elimination (Optimierungen JAC.O2 und JIT.O2, siehe Kapitel 2.3 sowie 2.4) zu verhindern, darf die auszuführende JMH-Benchmark-Methode nicht void sein und muss immer das Resultat der Berechnungen aus der Benchmark-Methode zurückgeben. Besteht das Resultat aus mehreren Rückgabewerten, sollten diese, beispielsweise in einem Objekt zusammengefasst, ebenfalls zurückgegeben werden. JMH kümmert sich um den Rest (Vorontsov 2014).
- Sofern das Resultat einer Berechnung vorhersehbar ist, ist es sehr gut möglich, dass der JIT-Compiler diesen Code mit Hilfe von Constant Folding (Optimierungen JAC.O1 und JIT.O1, siehe Kapitel 2.3 sowie 2.4) wegoptimiert. Eine Strategie, die Constant Folding verhindern soll, ist die Verwendung von unvorhersehbaren State-Objekten, um Berechnungen unvorhersehbar “zu machen”. Als State-Objekte können beispielsweise Instanzvariablen verwendet werden. Die konsequente Verwendung von State-Objekten in Kombination mit ein permanenten Rückgabe der Resultate verhindern das Constant Folding (Vorontsov 2014).
- Es sollten, wenn möglich, keine Loops verwendet werden. Der JIT-Compiler ist sehr intelligent implementiert und kann diese Loops sehr oft auf die eine oder andere Weise optimieren (Vorontsov 2014).

Konfiguration des JMH-Frameworks

Als Konfigurationsmöglichkeiten stehen unter anderen die Optionen “warmupIterations“, “warmupBatchSize“, “measurementIterations“, “measurementBatchSize“, “operationsPerInvocation“ und “timeUnit“ zur Verfügung. Ein Messdurchgang einer mit @Benchmark annotierten Methode besteht aus einer spezifizierten Anzahl “measurementIterations“, also Messiterationen. Eine Messiteration ist dabei jedoch nicht zwingend als

einzelner Aufruf der Methode zu verstehen. Mit “measurementBatchSize“ kann nämlich zusätzlich spezifiziert werden, wie oft die Methode innerhalb einer Iteration aufgerufen wird.

Direkt vor den Messdurchgängen werden sogenannte Warmup-Durchgänge durchlaufen. Die Bezeichnungen “Iterations“ und “BatchSize“ sind dabei gleich zu verstehen wie bei den Messungen selbst. Wird eine genügend grosse “warmupBatchSize“ gewählt, kann mit Warmup-Durchgängen sichergestellt werden, dass der JIT-Compiler später, beim eigentlichen Messen der Laufzeiten, alle Optimierungen schon durchgeführt hat und das Vornehmen dieser Optimierungen die Laufzeiten der eigentlichen Messungen nicht beeinflusst.

Mit “operationsPerInvocation“ kann JMH mitgeteilt werden, dass der Benchmark mehr als eine Operation per Invokation durchführt, sodass JMH die gemessene Laufzeit entsprechend anpasst. Dabei wird die ermittelte Laufzeit nach dem Benchmark durch “operationsPerInvocation“ geteilt und als neue Laufzeit festgelegt. Da die Optionen “warmupBatchSize“ sowie “measurementBatchSize“ JMH veranlassen, pro Iteration mehrere Aufrufe der mit @Benchmark annotierten Methode durchzuführen, kann durch Setzen von “operationsPerInvocation“ auf den gleichen Wert wie “measurementIterations“ ein Mitteln des Messwertes erzielt werden. Die Option “timeUnit“ spezifiziert das Ausgabeformat der Messzeiten. Mögliche Werte sind beispielsweise Nanosekunden, Mikrosekunden oder Millisekunden.

Folgende Werte wurden für alle Messungen verwendet:

JMH-Property	**Im Code verwendete Variable**	**Wert**
warmupBatchSize measurementBatchSize operationsPerInvocation	References.BENCHMARK.REPETITIONS	500’000
warmupIterations	References.BENCHMARK.WARMUP_ITERATIONS	20
measurementIterations	References.BENCHMARK.MEASUREMENT_ITERATIONS	20
timeUnit	-	TimeUnit. NANOSECONDS

Tabelle 3: Verwendete Werte beim Durchführen der Messungen mit dem JMH-Framework

3.2 Hard- und Software für Performance-Messungen

Für die Performance-Messungen wurden zwei PCs mit Linux Debian 8 (64-Bit) aufgesetzt. Das Java Development Kit 8 (JDK8) von Oracle (Oracle 2016 E) wurde zudem auf beiden PCs installiert. Zusätzlich wurde auf dem einen PC eine MySQL-Datenbank eingerichtet. Mit iptables wurden Regeln definiert, dass die Firewall nur eingehende SQL-Verbindungen vom selben PC (Host-PC der SQL-Datenbank) sowie von der MAC-Adresse des zweiten Test-PCs erlaubt. Dies garantiert, in Kombination mit einem Datenbank-Passwort, eine hohe Sicherheit der persisiterten Messdaten.

Um einen gleichmässigen Betrieb der beiden PCs zu garantieren, wurde im BIOS jeweils das Power-Management ausgeschaltet sowie im Betriebssystem die Clock-Frequenz der CPU auf den fixen Wert gesetzt (Shipilëv 2013, 26).

Es wurde darauf geachtet, dass während den Messungen möglichst wenige Prozesse liefen. Auch auf das Verwenden der grafischen Oberfläche wurde verzichtet.

3.3 Details zur Testumgebung

Die Testumgebung besteht aus folgenden drei Java-Maven-Projekten (siehe auch Datenträger, Kapitel 7):

Projekt	*Erläuterungen*
java-performance-tests	Dieses Projekt enthält die JMH-Library sowie die im Rahmen dieser Arbeit entwickelten und überarbeiteten Benchmarks (Tests), die im Kapitel 4 entsprechend dokumentiert wurden. Zur Sicherstellung der Funktionalität gewisser Benchmarks, wurden in diesem Maven-Projekt einige J-Unit-Tests implementiert.
java-result-parser	Dieses Projekt enthält einen Parser, der die von JMH generierten Resultat-Dateien einliest und in die zuvor konfigurierte SQL-Datenbank schreibt.
java-create-charts	Dieses Projekt enthält die JFreeChart-Libary und dient der Erstellung von Diagrammen. Vergleichbare Resultate werden aus der Datenbank ausgelesen und als Diagramm-Bilddateien abgespeichert.

Tabelle 4: Details zur Testumgebung

3.4 Konzept für die Messung mit JMH

Herausforderungen: Die meisten Operationen, die getestet werden, dauern nur wenige Nanosekunden oder einen Bruchteil davon. In diesem Bereich können möglicherweise Messungenauigkeiten auftreten. Zudem sind die Messungen mit Sicherheit ungenauer als beispielsweise bei Operationen, deren Laufzeit im Millisekunden-Bereich liegt.

Anmerkung: Einfachheitshalber wurden bei den folgenden Code-Fragmenten Annotations verwendet, um die Konfiguration von JMH sinnesgemäss abzubilden. Im "*java-performance-tests*"-Maven-Projekt (siehe auch Datenträger, Kapitel 7) wurden diese Einstellungen jedoch in der Methode *doBenchmarkForClass(...)* in der Klasse *App* vorgenommen. Dies ermöglicht die zentrale Konfiguration der Benchmark-Methoden. Ein Konkretisieren einzelner Methoden ist jedoch nach wie vor mittels Annotations möglich.

Konzept am Beispiel des Benchmarks loopPlus() der Klasse StringConcatenationTest

Stand des Benchmarks (sinngemäss nach der Diplomarbeit von Manuel Simbürger und Roman Thöni)

```
@Benchmark
@Warmup(iterations=10)
@Measurement(iterations=10)
@OperationsPerInvocation(value=1)
@BenchmarkMode(value=Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public String loopPlus() {
    String string = "";
    int dummy = 0;
    for(long i = 0; i < References.BENCHMARK.REPETITIONS; i++) {
        string = "";
        for(String s : strings) {
            string = string + s;
        }
        dummy += string.length();
    }
    return dummy;
}
```

Vorteile

(+) Jeder Aufruf des Benchmarks an sich dauert länger, da mit einer for-Schleife im Methoden-Körper dafür gesorgt wird, dass die zu testende Operation N-fach (N = References.BENCHMARK.REPETITIONS = 1'000'000) ausgeführt wird. Durch die mehrfache Ausführung derselben Operation wird das Eliminieren von Messungenauigkeiten angestrebt.

Nachteile

(-) Da der JIT-Compiler an Schleifen, die vielfach ausgeführt werden, unter Umständen Optimierungen vornimmt (OpenJDK 2016), ist es nicht empfehlenswert, die zu testende Operation mit Hilfe einer Schleife N-fach zu durchlaufen (N=1'000'000).

(-) Die Laufzeit des ganzen for-Loops wird gemessen, nicht die Laufzeit der String-Verkettung an sich. Mit der Verwendung der Annotation @OperationsPerInvocation(value=References.BENCHMARK.REPETITIONS) könnte dies jedoch verbessert werden.

(-) Die for-Schleife bringt zusätzliche Komplexität mit sich. Das JMH-Framework sollte genutzt werden, um die Benchmarks pro Iteration mehrfach durchzuführen.

(-) Die Rückgabe der Werte im Millisekunden-Bereich sind für einige der neu erstellten Messungen zu grob.

Vorschlag zur Verbesserung des Benchmarks

```
@Benchmark
@Warmup(iterations=20, batchSize=References.BENCHMARK.REPETITIONS)
@Measurement(iterations=20, batchSize=References.BENCHMARK.REPETITIONS)
@OperationsPerInvocation(value=References.BENCHMARK.REPETITIONS)
@BenchmarkMode(value=Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public String loopPlus() {
    String string = "";
    for(String s : strings) {
        string = string + s;
    }
    return string;
}
```

Vorteile

(+) Pro Warmup- und Measurement-Iteration wird jeweils N Mal der Benchmark durchlaufen (N = References.BENCHMARK.REPETITIONS = 500’000). N wird mit dem Attribut “batchSize“ bei den Annotations @Warmup und @Measurement festgelegt. Eine for-Schleife ist somit nicht mehr nötig.

(+) Die Anzahl Iterationen für die Warmup- sowie die Measurement-Phase wurden verdoppelt (iterations=20), die Anzahl der Aufrufe pro Iteration halbiert (batchSize=500'000). Da JMH bei der Ausgabe die Messwerte pro Iteration aufzeigt, ist mit dieser Änderung eine genauere Interpretation der Laufzeiten möglich. Dies ist z.B. für das Beobachten des JIT-Compilers bei den ersten Durchläufen der Warmup-Phase interessant.

(+) Zu Beginn jeder Iteration wird die Zeit in Nanosekunden festgehalten. Nach dem N-fachen Durchführen (N=500'000) der zu messenden Operation, wird wiederum die Zeit in Nanosekunden festgehalten und die Differenz zur Startzeit ermittelt.

(+) Die Annotation @OperationsPerInvocation dividiert die Laufzeit einer Iteration mit N durchgeführten Operationen (N=500'000) durch den mitgegebenen Wert (value=500'000). Somit wird für jede Iteration die durchschnittliche Laufzeit der Benchmark-Methode als Messergebnis verwendet. Diese ist (auch für Operationen im tiefen Nanosekundenbereich) präzise.

Nachteile

Keine Nachteile bekannt.

3.5 Wahl des Testdurchlaufs

Während der Bachelorarbeit wurden viele Testdurchläufe ausgeführt. Die Laufzeiten der Benchmarks fielen auf den beiden Test-PCs unterschiedlich aus, aber die Ranglisten der Benchmarks an sich fiel in den allermeisten Fällen gleich aus. Deswegen wurde die Entscheidung getroffen, den Test-PC zu verwenden, der technisch auf dem neueren Stand ist. Die Entscheidung fiel auf den PC mit dem Computernamen CLT-DSK-T-1544. Dieser verfügte zum Zeitpunkt, an dem die Testdurchläufe stattfanden, über ein Intel Core i7-4770 Prozessor mit 3.4 GHz sowie über 8 GB DRAM (Dynamic Random Access Memory).

Nachdem die letzten Benchmarks verfasst wurden, fanden fünf weitere Testdurchläufe statt. Die Messungen dieser fünf Testdurchläufe unterscheiden sich in bestimmten Fällen wenig. In anderen Fällen weisen die Messungen einen hohen Unterschied auf. Nachdem diese Tatsache festgestellt wurde, entstand die Frage, wie die Testergebnisse dokumentiert werden sollten. Die Option, den Durchschnitt über alle Testdurchläufe zu bilden, wurde ausgeschlossen. Der Grund dafür ist, dass dabei stabile und instabile Messungen miteinander vermischt worden wären. Ausserdem ist die Systemlast nicht bei jedem Testdurchlauf genau gleich, obwohl bei der Ausführung der Testdurchläufe darauf geachtet wurde, dass neben der Testausführung so wenig Prozesse wie möglich liefen.

Aus diesem Grund wurde entschieden, den zuverlässigsten Testdurchlauf im vierten Kapitel als Resultate zu dokumentieren. Doch welcher Testdurchlauf hat die zuverlässigsten Resultate generiert? Theoretisch gibt es viele Kriterien, die angewendet werden könnten. In der Tat stellen sich jedoch nur wenige der Kritierien als sinnvoll heraus. Da das JMH-Framework für jeden Benchmark eine Messabweichung, in dem Fall Fehler genannt, berechnet, kann diese gebraucht werden, um die Stabilität und damit die Zuverlässigkeit der Resultate abzuschätzen. Das Vertrauensintervall einer Messung ist immer 99.9%. Die Messabweichung, auch als Fehler genannt, entspricht der Hälfte des Vertrauensintervalls.

JMH-Ausgabe am Beispiel des Benchmarks “measureArrayCopy“

Result "measureArrayCopy":

287528.827 ±(99.9%) 40903.087 ns/op [Average]

(min, avg, max) = (235432.551, 287528.827, 332379.158), stdev = 47104.055

CI (99.9%): [246625.740, 328431.914] (assumes normal distribution)

Legende zur JMH-Ausgabe

- Messabweichung (rot)
- Vertrauensintervall (grün)

Wie in der nächsten Tabelle ersichtlich ist, wurde auf Basis zweier Kriterien der optimale Testdurchlauf gewählt: Die Summe aller Messabweichungen und die Anzahl aller minimal erreichten Messabweichungen. Bei der Summe aller Messabweichungen geht es darum, eine möglichst kleine Zahl zu erreichen. Bei der Anzahl minimaler Messabweichungen ist das Ziel, eine möglichst hohe Zahl zu erreichen. Schlussendlich wurde **der dritte Testdurchlauf** als bester Testdurchlauf gekürt, weil er die zweitbeste Punktzahl und die beste Summe aller Messabweichungen besitzt. Der fünfte Testdurchlauf hat zwar die beste Punktzahl erreicht, wurde aber auf Grund der höchsten Summe (schlechtestes Resultat) aller Messabweichungen nicht gewählt.

Aus Platzgründen wird im Folgenden nicht die ganze Tabelle dargestellt, sondern der Anfang und das Ende der Tabelle. Die komplette Tabelle befindet sich in der beigelegten CD im Anhang dieser Bachelorarbeit.

Benchmarks und ihre Fehler

Benchmarks	Fehler - Serie 1	Fehler - Serie 2	Fehler - Serie 3	Fehler - Serie 4	Fehler - Serie 5	Min. Fehler				
ArrayCopyOperationsTest.measureArrayCopy	40903,100	40749,300	39110,200	40001,100	41260,700	0	0	1	0	0
ArrayCopyOperationsTest.measureClone	47495,100	46312,000	46052,200	45185,000	46770,700	0	0	0	1	0
ArrayCopyOperationsTest.measureCopyOf	40372,200	38262,800	38408,200	39217,900	40264,800	0	1	0	0	0
ArrayCopyOperationsTest.measureCopyRange	1899,910	1521,430	1611,550	1477,950	1387,410	0	0	0	0	1
ArrayCopyOperationsTest.measureLoopCopy	1557,610	1383,100	1144,910	1393,470	1068,720	0	0	0	0	1
ArrayListsInitializationTest.addToListWith100000InitialCap	2540,690	1215,920	792112,000	1400,690	797094,000	0	1	0	0	0
ArrayListsInitializationTest.addToListWith10000InitialCap	2194,490	2206,110	1803,380	2325,560	1750,700	0	0	0	0	1
ArrayListsInitializationTest.addToListWith10InitialCap	2566,960	2204,910	1993,600	1969,420	1968,040	0	0	0	0	1
ArrayListsInitializationTest.addToListWith300000InitialCap	1492,810	1492,520	1296,840	1539,950	1008,100	0	0	0	0	1
ArrayListsInitializationTest.addToListWithNoInitialCap	4531,030	1479,060	1804,650	1994,390	1947,500	0	1	0	0	0
ArrayVsArrayListTest.basicRuntime	3390,760	3170,720	3178,960	3826,020	3929,820	0	1	0	0	0
ArrayVsArrayListTest.fillArrayListWithObjects	24719,800	32107,000	29254,900	25511,000	17169,500	0	0	0	0	1
ArrayVsArrayListTest.fillArrayWithObjects	15999,300	9065,490	10323,600	12358,500	12148,500	0	1	0	0	0
AutoboxingVsCastingTest.autoboxingPrimitAndWrapperInt	0,026	0,001	0,001	0,001	0,001	0	1	1	1	1
AutoboxingVsCastingTest.autoboxingPrimitAndWrapperInteger	0,013	0,020	0,012	0,015	0,016	0	0	1	0	0
AutoboxingVsCastingTest.autoboxingPrimitOnly	0,186	0,001	0,001	0,001	0,011	0	1	1	1	0

Abbildung 32: Anfang der Tabelle “Benchmarks und ihre Fehler”

SwitchStatementsTest.switchEnum	0,010	0,014	0,181	0,179	0,025	1	0	0	0	0
SwitchStatementsTest.switchInt	1014,000	0,098	0,085	0,026	0,057	0	0	0	1	0
SwitchStatementsTest.switchShort	0,011	0,146	0,112	0,112	0,147	1	0	0	0	0
SwitchStatementsTest.switchString	7630,000	0,046	0,067	0,108	0,034	0	0	0	0	1
SynchronizedAccessTest.useAtomicTypes	0,146	0,507	0,133	0,287	0,290	0	0	1	0	0
SynchronizedAccessTest.useReentrantLock	0,299	0,187	0,196	0,144	0,187	0	0	0	1	0
SynchronizedAccessTest.useSemaphore	0,426	0,161	0,202	0,211	0,125	0	0	0	0	1
SynchronizedAccessTest.useSynchronizedBlock	0,183	0,184	0,153	0,192	0,175	0	0	1	0	0
Anzahl Punkte:						42	63	72	69	74
Summe Fehler:	41693246,898	38857261,121	37027561,187	39277371,614	45110448,003					

Abbildung 33: Ende der Tabelle “Benchmarks und ihre Fehler”

4. Vergleiche verschiedener Code-Fragmente

Kapitel	Seite	Name	Testklassen
4.1	34	Konzept: Multiplikation zweier Werte	MultiplyOpVsAdditionInLoopOpTest
4.2	36	Instanzvariablen im Vergleich zu lokalen Variablen	ModuloInstanceVsLocalVariablePlusEqualsTest Modulo3InstanceVsLocalVariablePlusEqualsTest
4.3	39	Kopieren eines Arrays	ArrayCopyOperationsTest
4.4	40	Initialisierung einer ArrayList	ArrayListsInitializationTest
4.5	42	Performance von Integer-Arrays gegenüber Integer-ArrayLists	ArrayVsArrayListTest
4.6	44	Autoboxing und Casting	AutoboxingVsCastingTest
4.7	45	Comparable-Interface oder Comperator	ComparableVsComparatorTest
4.8	49	Unterschiedliche Definition der Schleifen-Obergrenzen-Variable sowie der Zählervariable	CounterAndLoopMaxTypeTest
4.9	50	Filtern von Werten im Array und in ArrayList	FilterValuesTest
4.10	53	Function-Composition und Methoden-Verkettung	FunctionVsMethodCompositionTest
4.11	54	Fakultät berechnen	IterativeVsRecursive200FactorialTest IterativeVsRecursive500FactorialTest IterativeVsRecursive1000FactorialTest IterativeVsRecursive3000FactorialTest IterativeVsRecursive8000FactorialTest
4.12	58	Mathematische Multiplikation und Division sowie Multiplikation und Division mit Bitshift	MathVsBitOperationsTest
4.13	60	Objekte zurücksetzen oder ein neues Objekt verwenden	ObjectCloningAndCleaningTest
4.14	62	Objekte kopieren	ObjectCloningAndCleaningTest
4.15	64	Polymorphismus	PolymorphismTest
4.16	66	Mehrfach vorhandene Werte aus ArrayLists oder Arrays entfernen	RemoveDuplicateValuesInArrayTest
4.17	68	Listen oder Arrays sortieren	SortArraysAndArrayListsTest
4.18	71	Aufruf einer statischen Methode im Gegensatz zu einer nicht-statischen Methode	StaticVsNonStaticMethodsTest
4.19	72	Die Verkettung von Strings	StringConcatenationTest
4.20	74	Das Vergleichen von Strings	StringSplittingAndComparingTest
4.21	75	Die Trennung von Strings	StringSplittingAndComparingTest
4.22	79	Summe aus den Werte eines Arrays oder einer Liste bilden	SumArraysAndArrayListsTest
4.23	82	Switch-Case-Anweisung gegenüber If-Else-Kette	SwitchCaseAndIfElseChainTest
4.24	83	Die Switch-Case-Anweisung und die Datentypen der Testfälle	SwitchStatementsTest
4.25	87	Synchronisationsarten im Vergleich	SynchronizedAccessTest

4.26	90	Operationen auf den Klassen ArrayList und LinkedList	ListArrayListTest ListLinkedListTest
4.27	96	Operationen auf den Klassen HashMap, LinkedHashMap und TreeMap	MapsHashMapTest MapsLinkedHashMapTest MapsTreeMapTest
4.28	105	Operationen auf den Klassen HashSet, LinkedHashSet und TreeSet	SetsHashSetTest SetsLinkedHashSetTest SetsTreeSetTest
4.29	109	Implementierung eines Stacks mit ArrayDeque und LinkedList	StackAsDequeList StackAsLinkedListTest

Tabelle 5: Beschreibung der Tests mit Kapitelnummern

4.1 Konzept: Multiplikation zweier Werte

Thema/Fragestellung

Die Multiplikation zweier Werte x und y kann in einer Programmiersprache auf mindestens zwei Arten erreicht werden. Die direkte Verwendung des mathematischen Multiplikation-Operators *, also x * y oder y * x, ist klar die intuitivste Variante. Eine zweite Variante ist das x-fache Durchlaufen einer Schleife. Im Schleifenkörper wird dann jeweils bei jeder Iteration das Resultat um y erhöht. Dies entspricht ebenfalls der Multiplikation der beiden Werte x und y. Alternativ kann die Schleife natürlich auch y-fach durchlaufen werden und das Resultat jeweils um x erhöht werden. In diesem Experiment wird zusätzlich untersucht, ob die Deklaration des Resultats als lokale Variable oder als Instanzvarixable einen Einfluss auf die Geschwindigkeit hat.

Vermutung/Hypothese

Die Multiplikation mit dem mathematischen Multiplikations-Operator in Kombination mit einer lokalen Variable im Schleifenkörper ist wahrscheinlich die effizienteste Variante. Auch die Verwendung einer Instanzvariable dürfte die Laufzeit kaum wirklich verschlechtern. Die Verwendung einer Schleife hat vermutlich in beiden Fällen (mit lokaler Variable oder Instanzvariable) eine längere Laufzeit.

Programmcode

```
private int result;

@Benchmark
public long multiplyOp() {
	int multiplier = Short.MAX_VALUE;
	result = multiplier;

	result *= multiplier;
	return result;
}

@Benchmark
public long additionOpInLoop() {
	int multiplier = Short.MAX_VALUE;
	result = multiplier;

	for (int i = 1; i < multiplier; i++) {
		result += multiplier;
	}

	return result;   (1)
}

@Benchmark
public long multiplyOpLocalVar() {
	int multiplier = Short.MAX_VALUE;
	int res = multiplier;

	res *= multiplier;
	return res;
}
```

```
@Benchmark
public long additionOpInLoopLocalVar() {
	int multiplier = Short.MAX_VALUE;
	int res = multiplier;

	for (int i = 1; i < multiplier; i++) {
		res += multiplier;
	}
	return res;
}
```

Konzept

1 Um Dead-Code-Elimination (Optimierungen JAC.O2 und JIT.O2, siehe Kapitel 2.3 sowie 2.4) zu verhindern, wird bei jeder Benchmark-Methode das berechnete Resultat zurückgegeben. Das JMH-Framework sorgt dann bei diesen Methoden dafür, dass die Rückgabewerte konsumiert werden.

Diagramme und Beobachtungen

Abbildung 34: Laufzeiten der Benchmarks aus der Klasse MultiplyOpVsAdditionInLoopOpTest

Auf diesem Diagramm ist zu erkennen, dass die Multiplikation bei Verwendung der mathematischen Multiplikation-Operation jeweils sehr zeiteffizient ist. Die Multiplikation mittels mehrfacher mathematischer Addition durch die Verwendung einer Schleife, ist bei Verwendung einer lokalen Variable erstaunlicherweise ebenso performant. Bei der Verwendung einer Instanzvariable fällt die Laufzeit jedoch deutlich länger aus.

Auswertung/Empfehlung

Am Ausgang dieses Experimentes ist eindrücklich erkennbar, dass es extrem ineffizient sein kann, Instanzvariablen mehrfach zu mutieren. Wird bei der Nachbildung der Multiplikation mit einer Schleife und mehreren Additionen eine lokale Variable als Variable zum Persisieren des Resultats verwendet, ist dieser Ansatz hingegen gleich schnell wie die Multiplikation mit der “normalen“ Multiplikations-Operation. Dies ist auf eine intelligente Optimierung seitens JIT-Compilers zurückzuführen. Aus Gründen der Lesbarkeit und Einfachheit wird dennoch der Einsatz der “normalen“ Multiplikations-Operation empfohlen.

4.2 Instanzvariablen im Vergleich zu lokalen Variablen

Thema/Fragestellung

In der Programmierung kommt es oft vor, dass eine Variable beim Durchlaufen einer Schleife ständig verändert wird. Handelt es sich nun bei der Variable um eine Instanzvariable, stellt sich in Java folgende Frage: Lohnt es sich, eine temporäre, lokale Variable zu initialisieren, diese an Stelle der Instanzvariable ständig zu verändern und die lokale Variable (nach Durchlaufen aller Schleifendurchgänge) wieder der Instanzvariable zuzuweisen? Oder ist es gar gleich zeiteffizient, die Instanzvariable direkt zu verändern? Genau auf diese Frage wird in folgenden zwei Experimenten eingegangen. Dabei wird in der Schleife eine Modulo-Operation durchgeführt und eine Variable aufsummiert. Es wurden zwei Testszenarien entworfen: Während in einem Testszenario eine bestimmte Zahl Modulo 3 berechnet wird, wird im anderen Testszenario die gleiche Zahl Modulo (i+1) berechnet. Die Variable i steht hier für die Zählvariable der Schleife.

Vermutung/Hypothese

Es wird vermutet, dass es performanter ist, eine lokale Variable zu initialisieren, diese danach kontinuierlich zu verändern und das Resultat am Schluss der Instanzvariable zuzuweisen als die Instanzvariable direkt kontinuierlich zu verändern.

Programmcode

```
public class ModuloInstanceVsLocalVariablePlusEqualsTest {

       private int sumUpInstanceVariable;
       private int reps = References.BENCHMARK.REPETITIONS;

       @Benchmark
       public int instanceVarPlusEqual() {
              sumUpInstanceVariable = 0;
              for (int i = 0; i < reps; i++) {
                     sumUpInstanceVariable += reps % (i + 1);   (1)
              }

              return sumUpInstanceVariable;
       }

       @Benchmark
       public int localVarPlusEqual() {
              int sumUpLocalVariable = 0;
              for (int i = 0; i < reps; i++) {
                     sumUpLocalVariable += reps % (i + 1);
              }
              sumUpInstanceVariable = sumUpLocalVariable;

              return sumUpInstanceVariable;
       }
}

public class Modulo3InstanceVsLocalVariablePlusEqualsTest {
```

```
	private int modOperandRightSide = 3;
	private int reps = References.BENCHMARK.REPETITIONS;
	private int sumUpInstanceVariable;

	@Benchmark
	public int localVarMod3PlusEqual() {
		int sumUpLocalVariable = 0;
		for (int i = 0; i < reps; i++) {
			sumUpLocalVariable += reps % modOperandRightSide;
		}
		sumUpInstanceVariable = sumUpLocalVariable;

		return sumUpInstanceVariable;
	}

	@Benchmark
	public int instanceVarMod3PlusEqual() {
		sumUpInstanceVariable = 0;
		for (int i = 0; i < reps; i++) {
			sumUpInstanceVariable += reps % modOperandRightSide;
		}

		return sumUpInstanceVariable;
	}
}
```

Überlegungen

1 Mit Anwendung der Modulo-Operation auf die sich ständig ändernde Schleifenzählvariable i wird sichergestellt, dass bei jeder Iteration eine Berechnung durchgeführt wird. Diese kann weder vom Java-, noch vom JIT-Compiler komplett wegoptimiert werden. Die Schleifenzählvariable wird nämlich bei jeder Iteration inkrementiert und ist somit für den Compiler nicht vorhersehbar (Optimierung JIT.O1, siehe Kapitel 2.4).

Diagramm und Beobachtung

Wichtig: Alle folgenden Diagramme sind skaliert und zeigen einen speziell gewählten Ausschnitt. Der Ausschnitt ist so gewählt, dass die linke Kante des Diagramms beim Wert x und die rechte Kante des Diagramms beim Wert y angesetzt ist. Die Werte x und y berechnen sich folgendermassen:

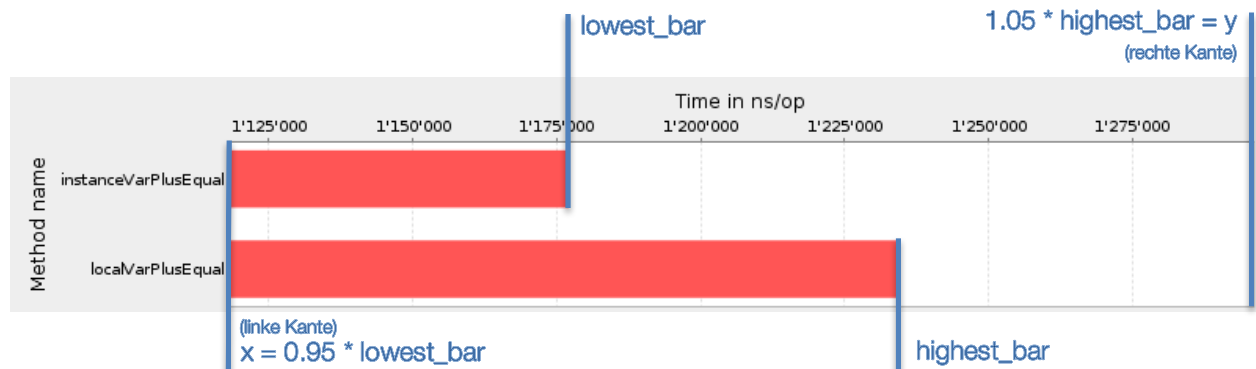

Abbildung 35: Laufzeiten der Benchmarks aus der Klasse ModuloInstanceVsLocalVariablePlusEqualsTest

Wenn Modulo (i+1) berechnet wird, ist also das Verändern der Instanzvariable schneller als das Verändern einer lokalen Variable.

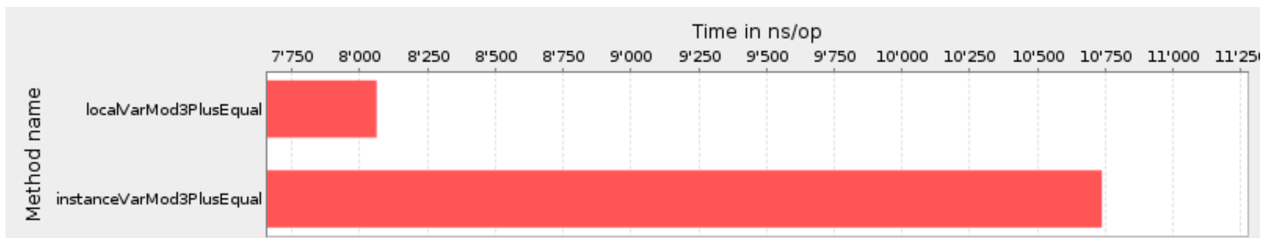

Abbildung 36: Laufzeiten der Benchmarks aus der Klasse Modulo3InstanceVsLocalVariablePlusEqualsTest

Wenn nun Modulo 3 berechnet wird, ist es genau umgekehrt. Das Verändern der lokalen Variable ist schneller als das Verändern der Instanzvariable. Hierbei muss beachtet werden, dass insgesamt pro Schleifendurchlauf ein Instanzvariablen-Lesezugriff (konkret: auf die Variable modOperandRightSide) mehr gemacht wird als bei der Berechnung von Modulo (i+1). Hingegen wird die Zählvariable i der Schleife sowie die Addition mit 1 hier nicht verwendet.

Auswertung/Empfehlung

Aus diesem Experiment folgt, dass es in Java nicht allgemein eindeutig ist, ob es besser ist, das Zwischenresultat in einer lokalen Variable oder in einer Instanzvariable abzuspeichern. Dies ist vom Gesamtkontext abhängig.

4.3 Kopieren eines Arrays

Thema/Fragestellung

Für das Kopieren eines Arrays stellt Java verschiedene Möglichkeiten zur Verfügung. Einerseits kann auf dem Array selbst die Operation clone() aufgerufen werden. Andererseits stellt die Klasse Arrays die zwei Methoden copyOf(...) und copyOfRange(...) zur Verfügung, um den Inhalt eines Arrays zu kopieren. Bei genauerer Betrachtung der Implementierung dieser zwei Methoden wird jedoch klar, dass diese Methoden schlussendlich beide System.arracopy(...) aufrufen (GrepCode™ 2015 A). Eine weitere Möglichkeit besteht darin, ein Array mit Hilfe einer for-Schleife zu kopieren. In den durchgeführten Benchmarks wurden alle erwähnten Möglichkeiten auf ein int-Array mit 500’000 zufällig gewählten Werten angewandt und miteinander verglichen.

Vermutung/Hypothese

Vermutlich wird das Klonen des Arrays am meisten Zeit in Anspruch nehmen, da clone(...)-Operationen in Java grundsätzlich sehr viel Zeit beanspruchen. Das manuelle Kopieren des Arrays mit Hilfe einer for-Schleife wird wahrscheinlich auch eher viel Zeit in Anspruch nehmen. Die übrigen Methoden sollten relativ performant sein.

Programmcode

```
private int[] array1 = new int[References.BENCHMARK.REPETITIONS];

@Setup(Level.Trial)
public void setUpArray() {
	for (int i = 0; i < References.BENCHMARK.REPETITIONS; i++) {
		array1[i] = (int) (Math.random() * References.BENCHMARK.RANGE);
	}
}

@Benchmark
public int[] measureClone() {
	return array1.clone();
}

@Benchmark
public int[] measureCopyOf() {
	return Arrays.copyOf(array1, array1.length);
}

@Benchmark
public int[] measureArrayCopy() {
	int[] copy = new int[array1.length];
	System.arraycopy(array1, 0, copy, 0, array1.length);
	return copy;
}

@Benchmark
public int[] measureCopyRange() {
	return Arrays.copyOfRange(array1, 0, array1.length);
}
```

```
@Benchmark
public int[] measureLoopCopy() {
        int[] copy = new int[array1.length];
        for (int i = 0; i < array1.length; i++) {
                copy[i] = array1[i];
        }
        return copy;
}
```

Diagramm und Beobachtung

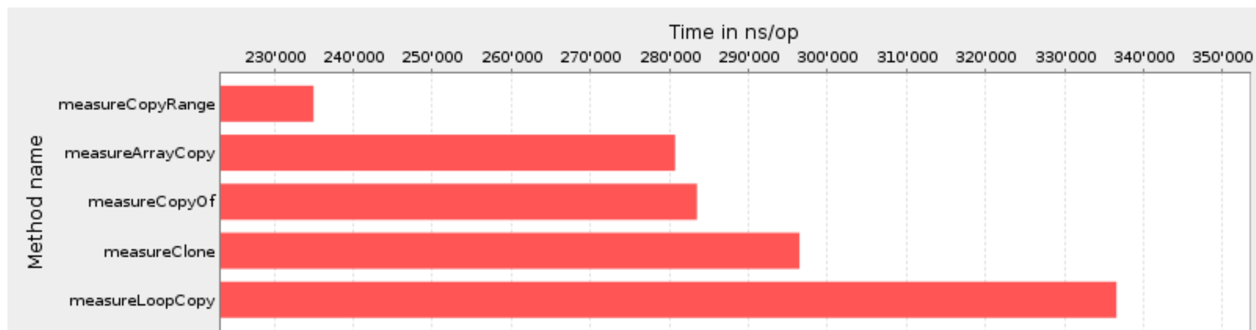

Abbildung 37: Laufzeiten der Benchmarks aus der Klasse ArrayCopyOperationsTest

Deutlich am schnellsten ist das Kopieren des Arrays mit Hilfe der Methode copyOfRange(...) der Klasse Arrays. Langsamer, aber in etwa gleich schnell, sind die Varianten System.arraycopy(...) und Arrays.copyOf(...). Das Klonen des Arrays ist entgegen den Erwartungen nicht die langsamste Methode. Das Kopieren des Arrays mit einer for-Schleife dauert nämlich ca. 40 Mikrosekunden länger.

Auswertung/Empfehlung

Für das häufige Kopieren von grossen Arrays wird empfohlen, die Methode Arrays.copyOfRange(...) zu verwenden. Dabei muss die zu kopierende Range so gesetzt werden, dass alle Elemente des Ursprungarrays berücksichtigt werden.

4.4 Initialisierung einer ArrayList

Thema/Fragestellung

Wird eine ArrayList initialisiert, kann die Software-Entwicklerin oder der Software-Entwickler zwischen dem parameterlosen Konstruktor und dem Konstruktor mit dem einen Parameter initialCapacity wählen. Doch welcher Konstruktor soll gewählt werden? Wie stark beeinflusst die Wahl die Performance beim späteren Hinzufügen von neuen Elementen? Fällt der Entscheid auf den Konstruktor mit Parameter: Welcher Wert wird für die Variable initialCapacity gewählt?

Bei genauerer Betrachtung der Implementierung der Klasse ArrayList wird ersichtlich, wie diese Liste intern aufgebaut ist und wie sie verwaltet wird. Wichtig zu wissen ist, dass die Klasse ArrayList für die Persistierung der Daten ein einfaches Object-Array mit dem Namen elementData verwendet (GrepCode™ 2015 B).

Wichtig ist auch, dass beim Aufruf des leeren Konstruktors die ArrayList in Java 8 nur lazy initialisiert wird. Hierbei wird dem Object-Array elementData ein leeres Objekt-Array zugewiesen. Erst beim Hinzufügen von Elementen wird elementData auf mindestens zehn Elemente vergrössert (GrepCode™ 2015 B). In Java 7 wurde im parameterlosen Konstruktor noch direkt der zweite Konstruktor mit dem Parameter initialCapacity aufgerufen. Die initialCapacity hatte dabei den Wert 10 (GrepCode™ 2011).

In Java 8 wie auch in Java 7 wird beim Aufruf des Konstruktors mit dem Parameter initialCapacity ein Object-Array mit der Grösse initialCapacity erzeugt und dieses dann der Variable elementData zugewiesen. Beim Hinzufügen von neuen Elementen mit der add(...)-Methode wird sichergestellt, dass das Object-Array elementData immer genügend Platz für ein neues zusätzliches Element bereitstellt.

Vermutung/Hypothese

Mit den gewonnenen Kenntnissen aus der Implementierung der ArrayList ist anzunehmen, dass es am effizientesten ist, eine ArrayList mit der (bereits im Voraus bekannten) angestrebten maximalen Grösse zu initialisieren. Muss nämlich bei jedem Hinzufügen eines neuen Elements das interne Object-Array elementData vergrössert werden, benötigt dies Zeit.

Programmcode

```
private List<Integer> list;
private int numberOfAssignements = 99999;

@Benchmark
public List<Integer> addToListWithNoInitialCap() {
	list = new ArrayList<Integer>();
	for (int i = 0; i < numberOfAssignements; i++) {
		list.add(i);
	}
	return list;
}

@Benchmark
public List<Integer> addToListWith10InitialCap() {
	list = new ArrayList<Integer>(10);
	for (int i = 0; i < numberOfAssignements; i++) {
		list.add(i);
	}
	return list;
}
// [...]

@Benchmark
public List<Integer> addToListWith300000InitialCap() {
	list = new ArrayList<Integer>(300000);
	for (int i = 0; i < numberOfAssignements; i++) {
		list.add(i);
	}
	return list;
}
```

Diagramm und Beobachtung

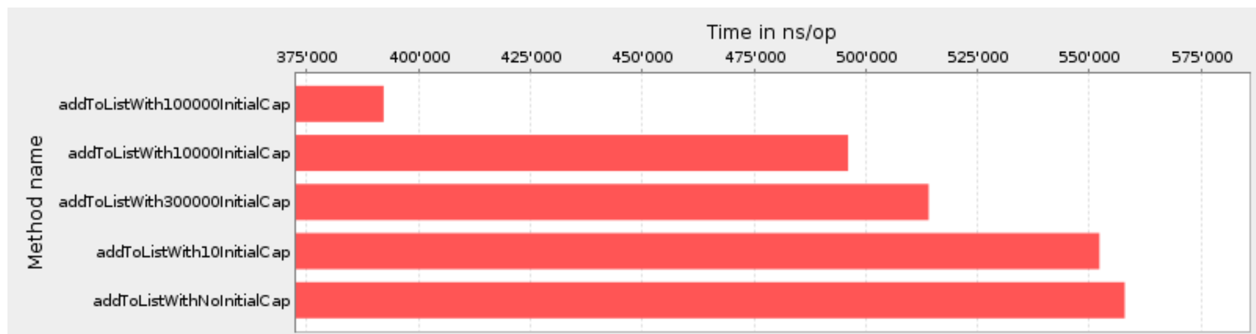

Abbildung 38: Laufzeiten der Benchmarks aus der Klasse ArrayListsInitializationTest

Wie bereits vermutet, ist das Initialisieren mit der passenden Initialkapazität am effizientesten. Wird die ArrayList mit einer zu kleinen Kapazität instanziiert, wird durch das häufige Vergrössern Zeit eingebüsst. Wird die ArrayList hingegen mit einer zu grossen Kapazität instanziiert, wird ebenfalls Zeit und vor allem auch Speicher eingebüsst.

In jedem der fünf Benchmarks werden einer ArrayList 99'999 Elemente hinzugefügt. Bei dem Benchmark addToListWith300000InitialCap() wird für 300'000 Elemente, sprich für 200'001 Elemente, zu viel Platz reserviert. Bei dem Benchmark addToListWithNoInitialCap() wird hingegen für 0 Elemente, sprich für 99'999 Elemente zu wenig, Platz geschafft. Obwohl bei der ersten Variante praktisch das Dreifache an benötigtem Platz reserviert wird, ist diese erste Variante zeiteffizienter als die zweite, bei der zu wenig Platz reserviert wird.

Auswertung/Empfehlung

Das Instanzieren einer ArrayList mit einer Initialkapazität, die der maximal angestrebten Grösse der ArrayList entspricht, ist sicherlich die effizienteste Variante. Meist ist es jedoch so, dass die Entwicklerin oder der Entwickler die maximale Grösse, die eine ArrayList je erreichen wird, nicht kennt. Wie bereits aufgezeigt, ist es aus Performance-Sicht sehr empfehlenswert, dass die Entwicklerin oder der Entwickler sich über die ungefähre maximale Grösse der ArrayList Gedanken macht.

4.5 Performance von Integer-Arrays gegenüber Integer-ArrayLists

Thema/Fragestellung

Es wird sowohl ein Array als auch eine ArrayList mit Integer-Wrapper-Werten abgefüllt. Wie gross ist der zusätzliche Aufwand, der durch die Verwendung einer ArrayList entsteht?

Vermutung/Hypothese

Der zusätzliche Aufwand besteht lediglich darin, eine Instanz einer ArrayList zu erstellen. Deswegen wird der Unterschied vermutlich minimal ausfallen.

Programmcode

```
private Integer[] arrayWithObjects;
private List<Integer> arrayListWithObjects;
private static final int REPETITIONS = References.BENCHMARK.REPETITIONS;

private void setupLists() {
	arrayWithObjects = new Integer[REPETITIONS];
	arrayListWithObjects = new ArrayList<Integer>(REPETITIONS);
}

@Benchmark
public List<Integer> basicRuntime() {
	setupLists();
	return arrayListWithObjects;
}

@Benchmark
public Integer[] fillArrayWithObjects() {
	setupLists();
	for(int counter = 0; counter < REPETITIONS; counter++)
	{
		arrayWithObjects[counter] = REPETITIONS % (counter+1);
	}
	return arrayWithObjects;
}

@Benchmark
public List<Integer> fillArrayListWithObjects() {
	setupLists();
	for(int counter = 0; counter < REPETITIONS; counter++)
	{
		arrayListWithObjects.add(REPETITIONS % (counter+1));
	}
	return arrayListWithObjects;
}
```

Diagramm und Beobachtung

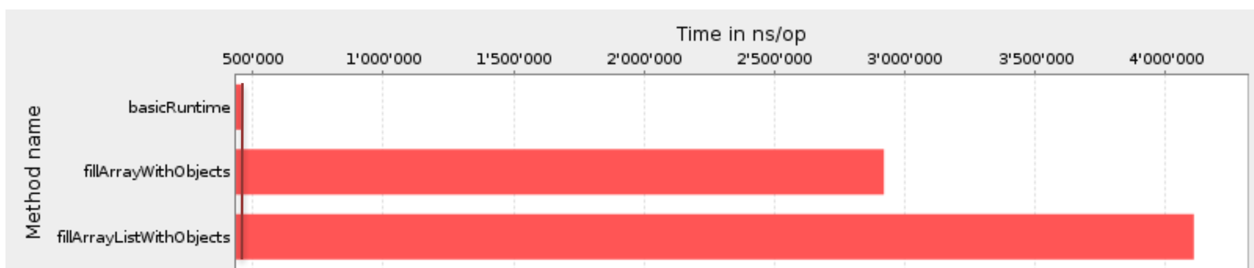

Abbildung 39: Laufzeiten der Benchmarks aus der Klasse ArrayVsArrayListTest

Hinweis: Der Benchmark basicRuntime zeigt die “Grund“-Laufzeit auf, die ca. 0.5 Millisekunden beträgt.

Die “Grund“-Laufzeit. Das Hinzufügen der 500’000 Integer-Werten zu einer ArrayList dauert mehr als 1 Millisekunde länger als das Zuweisen von 500’000 Integer-Werten an einem Integer-Array.

Auswertung/Empfehlung

In Bezug auf Performance wird empfohlen, wenn immer möglich, Arrays anstelle von ArrayLists zu verwenden.

4.6 Autoboxing und Casting

Thema/Fragestellung

Bei diesem Test werden mehrere Benchmarks mit Autoboxing und/oder Casting durchgeführt. Da es nicht möglich ist, die Benchmarks in sinnvolle Kategorien aufzuteilen und einander gegenüberzustellen, wird auf das Formulieren einer Hypothese, das Notieren der Beobachtung sowie auf das Empfehlen einer Variante verzichtet.

Programmcode

```
private int intVal = 420815;
private Integer integerVal = 518024;

// basic runtime
@Benchmark
public int autoboxingPrimitOnly() {
	return intVal + intVal;
}

@Benchmark
public int autoboxingPrimitAndWrapperInt() {
	return intVal + integerVal;
}

@Benchmark
public int castingPrimitAndWrapperInt() {
	return intVal + (int) integerVal;
}

@Benchmark
public Integer autoboxingPrimitAndWrapperInteger() {
	return intVal + integerVal;
}

@Benchmark
public Integer castingPrimitAndWrapperInteger() {
	return (Integer) intVal + integerVal;
}

@Benchmark
public int autoboxingWrapperAndPrimitInt() {
	return integerVal + intVal;
}

@Benchmark
public int castingWrapperAndPrimitInt() {
	return (int) integerVal + intVal;
}

@Benchmark
public Integer autoboxingWrapperAndPrimitInteger() {
	return integerVal + intVal;
}

@Benchmark
public Integer castingWrapperAndPrimitInteger() {
	return integerVal + (Integer) intVal;
}
```

```
        @Benchmark
        public int autoboxingWrapperAndWrapperInt() {
                return integerVal + integerVal;
        }

        @Benchmark
        public int castingWrapperAndWrapperInt1() {
                return (int) integerVal + (int) integerVal;
        }

        @Benchmark
        public int castingWrapperAndWrapperInt2() {
                return (int) (integerVal + integerVal);
        }

        @Benchmark
        public Integer autoboxingWrapperOnly() {
                return integerVal + integerVal;
        }
```

Diagramm

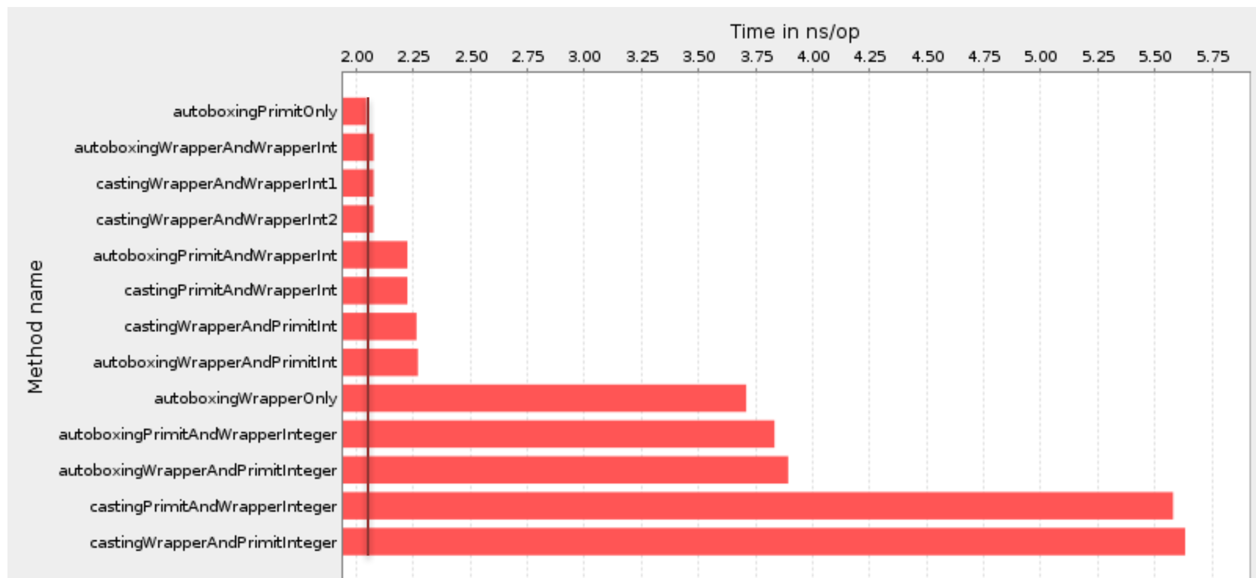

Abbildung 40: Laufzeiten der Benchmarks aus der Klasse AutoboxingVsCastingTest

Hinweis: autoboxingPrimitOnly zeigt hier gleichzeitig auch die "Grund"-Laufzeit aller Benchmarks auf.

4.7 Comparable-Interface oder Comperator

Thema/Fragestellung

Wie performant eine Liste sortiert werden kann, ist grundsätzlich eine Frage des verwendeten Algorithmus. Diese wird im Rahmen dieser Arbeit jedoch nicht beantwortet, da sich Sortieralgorithmen schon per se in ihrer Laufzeit unterscheiden (Rowell 2016).

Möchten Entwickler in Java nun eine Liste mit Objekten nach einem gewissen Kriterium sortieren, gibt es grundsätzlich zwei Varianten, dies zu bewerkstelligen. Entweder implementiert die betreffende Klasse das generische Comparable-Interface oder es wird eine generische Comparator-Instanz erzeugt. Der Vorteil des Comparators ist, dass die zu sortierende Klasse an sich nicht verändert werden muss.

Vermutung/Hypothese

Aufgrund der praktisch identischen Implementierung der beiden Varianten der Arrays.sort(...)-Methoden (Grepcode™ 2015 A) kann keine zuverlässige Aussage über mögliche Laufzeitunterschiede gemacht werden.

Programmcode

```
private Fruit[] fruits;
private Fruit[] fruitsBackup = new Fruit[References.BENCHMARK.LISTREPETITIONS];

@Setup(Level.Trial)
public void setupTests() {

     for(int i = 0; i < References.BENCHMARK.LISTREPETITIONS; i++) {

          Fruit fruitToAdd = null;

          int randNbrFruit = ThreadLocalRandom.current().nextInt(0, 3 + 1);
          int randNbrQuantity = ThreadLocalRandom.current().nextInt(70, 100 + 1);

          switch (randNbrFruit) {
          case 0:
           fruitToAdd = new Fruit("Pineapple", "Pineapple description", randNbrQuantity);
           break;
          case 1:
           fruitToAdd = new Fruit("Apple", "Apple description", randNbrQuantity);
           break;
          case 2:
           fruitToAdd = new Fruit("Orange", "Orange description", randNbrQuantity);
           break;
          case 3:
           fruitToAdd = new Fruit("Banana", "Banana description", randNbrQuantity);
           break;
          }
          fruitsBackup[i] = fruitToAdd;
     }
}

//basic runtime
@Benchmark
public Fruit[] basicRuntime() {
   fruits = Arrays.copyOf(fruitsBackup, fruitsBackup.length);
   return fruits;
}

@Benchmark
public Fruit[] comparableSort() {
   fruits = Arrays.copyOf(fruitsBackup, fruitsBackup.length);
   Arrays.sort(fruits);
   return fruits;
}

@Benchmark
public Fruit[] comparatorSort() {
   fruits = Arrays.copyOf(fruitsBackup, fruitsBackup.length);
   Arrays.sort(fruits, ComparableVsComparatorTest.FruitNameComparator);
   return fruits;
}
```

```java
/**
 * This class implements a Fruit.
 * @author mkyong
 * @src http://www.mkyong.com/java/java-object-sorting-example-comparable-and-comparator/
 */
public class Fruit implements Comparable<Fruit>{
   private String fruitName;
   private String fruitDesc;
   private int quantity;

   public Fruit(String fruitName, String fruitDesc, int quantity) {
           super();
           this.fruitName = fruitName;
           this.fruitDesc = fruitDesc;
           this.quantity = quantity;
   }

   //[...]

   @Override
   public int compareTo(Fruit compareFruit) {
           int compareQuantity = ((Fruit) compareFruit).getQuantity();
           //ascending order
           return this.quantity - compareQuantity;
   }

   //[...]
}

public static final Comparator<Fruit> FruitNameComparator = new Comparator<Fruit>() {
   public int compare(Fruit fruit1, Fruit fruit2) {
           int fruitQuantity1 = fruit1.getQuantity();
           int fruitQuantity2 = fruit2.getQuantity();
           //ascending order
           return fruitQuantity1 - fruitQuantity2;
   }
};
```

Diagramm und Beobachtung

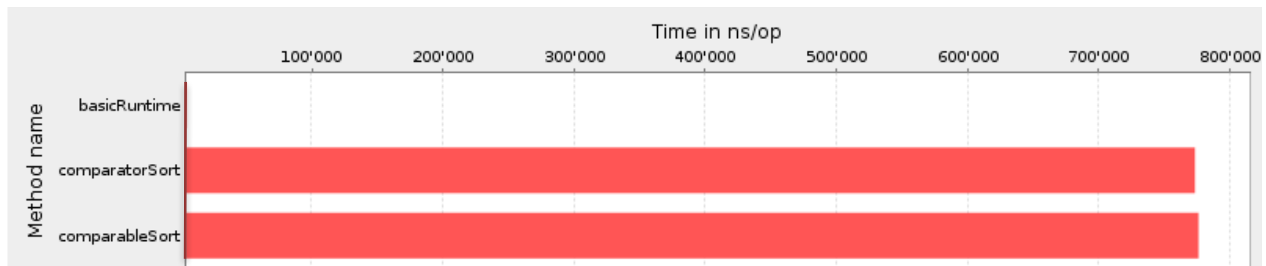

Abbildung 41: Laufzeiten der Benchmarks aus der Klasse ComparableVsComparatorTest

Der Unterschied zwischen dem Comparable-Interface und dem Comparator ist praktisch nicht sichtbar und beträgt bei einer Liste mit 10'000 Elementen nur wenige Nanosekunden. Die basicRuntime ist nahezu 0.

Auswertung/Empfehlung

Grunsätzlich können beide Varianten genutzt werden. Es hängt stark von der Anwendung ab, ob das Comparable-Interface implementiert oder der Comparator verwendet werden soll. Der Comparator hat den Vorteil, dass er auch verwendet werden kann, ohne dabei die zu sortierende Klasse abzuändern.

4.8 Unterschiedliche Definition der “Schleifen-Obergrenzen-Variable“ sowie der Zählvariable

Thema/Fragestellung

Wird im Schleifenkopf eine Bedingung deklariert, enthält die Bedingung meist eine bestimmte Zahl als obere Grenze. Doch welchen Einfluss haben verschiedene Keywords auf diese “Schleifen-Obergrenzen-Variable“? Wie wirken sich die Keywords auf die Performance aus? Ist [kein Keyword], static, static final, … oder gar eine hartcodierte Zahl im Schleifenkopf die effizienteste Variante?

Welche Performance erzielt eine lokale Variable als Zählvariable gegenüber einer Instanzvariable als Zählvariable?

Vermutung/Hypothese

Es ist nachgewiesenermassen nicht eindeutig bestimmbar, ob beim Verändern einer Variable innerhalb einer Schleife zur Verwendung von Instanzvariablen oder lokalen Variablen geraten werden soll, um tiefere Laufzeiten zu erreichen (siehe Kapitel 4.2). Ob derselbe Umstand für Zählvariablen jedoch gleichermassen gilt, muss mithilfe dieses Experiments ermittelt werden.

Programmcode

```
private static final int REPETITIONS = References.BENCHMARK.REPETITIONS;
private int repetitions2 = References.BENCHMARK.REPETITIONS;
private static int repetitions3 = References.BENCHMARK.REPETITIONS;

private int i = 0;


@Benchmark
public int loopMaxStaticFinal() {
	int counterLocalVariable = 0;

	for (int i = 0; i < REPETITIONS; i++) {
		counterLocalVariable += REPETITIONS % (i + 1);
	}

	return counterLocalVariable;
}


@Benchmark
public int loopMaxInstVar() {
	int counterLocalVariable = 0;

	for (int i = 0; i < repetitions2; i++) {
		counterLocalVariable += REPETITIONS % (i + 1);
	}

	return counterLocalVariable;
}
```

```
@Benchmark
public int loopMaxStatic() {
        int counterLocalVariable = 0;
        for (int i = 0; i < repetitions3; i++) {
                counterLocalVariable += REPETITIONS % (i + 1);
        }

        return counterLocalVariable;
}

@Benchmark
public int loopMaxMagicNumber() {
        int counterLocalVariable = 0;
        for (int i = 0; i < 500000; i++) // !!! check if magic number equals
                                          // REPETITIONS !!!
        {
                counterLocalVariable += REPETITIONS % (i + 1);
        }

        return counterLocalVariable;
}

@Benchmark
public int loopMaxMagicNumberCounterInst() {
        int counterLocalVariable = 0;
        for (i = 0; i < 500000; i++)      // !!! check if magic number equals
                                          // REPETITIONS !!!
        {
                counterLocalVariable += REPETITIONS % (i + 1);
        }

        return counterLocalVariable;
}

@Benchmark
public int loopMaxStaticFinalCounterInst() {
        int counterLocalVariable = 0;
        for (i = 0; i < REPETITIONS; i++) {
                counterLocalVariable += REPETITIONS % (i + 1);
        }

        return counterLocalVariable;
}

@Benchmark
public int loopMaxInstVarCounterInst() {
        int counterLocalVariable = 0;
        for (i = 0; i < repetitions2; i++) {
                counterLocalVariable += REPETITIONS % (i + 1);
        }

        return counterLocalVariable;
}

@Benchmark
public int loopMaxStaticCounterInst() {
        int counterLocalVariable = 0;
        for (i = 0; i < repetitions3; i++) {
                counterLocalVariable += REPETITIONS % (i + 1);
        }

        return counterLocalVariable;
}
```

Diagramm und Beobachtung

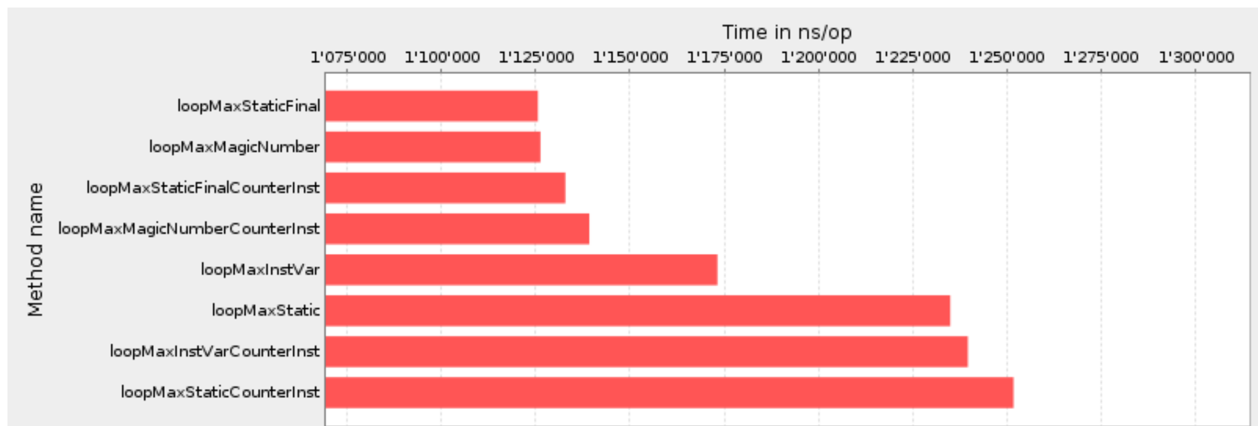

Abbildung 42: Laufzeiten der Benchmarks aus der Klasse CounterAndLoopMaxTypeTest

Die Kombination einer lokalen Zählvariable und einer Schleifen-Obergrenzen-Variable mit den Keywords static final erreicht die beste Performance, sprich die kürzeste Zeit. Die schlechteste Performance wird von der "Schleifen-Obergrenzen-Variable" mit dem Keyword static in Kombination mit einer Instanzvariable als Zählvariable erzielt, nämlich von loopMaxStaticCounterInst().

Auswertung/Empfehlung

Es wird empfohlen, aus den vorgestellten Varianten, die Kombination aus einer "Schleifen-Obergrenzen-Variable" mit den Keywords static final und einer lokalen Zählvariable zu verwenden. Anstelle der static final "Schleifen-Obergrenzen-Variable" kann auch eine Magic-Number verwendet werden.

4.9 Filtern von Werten im Array und in ArrayList

Thema/Fragestellung

Um auf einem int-Array oder einer Integer-ArrayList einen bestimmten Filter anzuwenden, kann das Array bzw. die ArrayList beispielsweise mit einer for-Schleife durchlaufen werden und dabei mit einer if-Bedingung die relevanten Daten gefiltert werden.

Ab Java 8 können für diese Aufgabe auch Streams eingesetzt werden. Diese erlauben das Filtern mit Hilfe einer dedizierten filter(...)-Methode. Mit der Methode parallel() kann zusätzlich ein paralleles Abwickeln des Filter-Vorgangs erreicht werden.

Vermutung/Hypothese

Sofern die parallel()-Methode effizient umsetzt, was sie verspricht, ist das Filtern des Arrays und der ArrayList mit diesem Ansatz am schnellsten. Die beiden anderen Ansätze könnten in etwa gleich schnell sein.

Programmcode

```
private static final int REPS = References.BENCHMARK.REPETITIONS /
  References.BENCHMARK.DIVISOR_REPETITIONS;

private int[] array = new int[REPS];
private int[] dummy;
private List<Integer> list;
private List<Integer> dummyList;

@Setup(Level.Trial)
public void setUpArray() {
	for (int i=0; i < REPS; i++) {
		array[i] = (int) (Math.random() * References.BENCHMARK.RANGE);
	}
	list = new ArrayList<Integer>();
	for (int i=0; i < REPS; i++) {
		list.add(array[i]);
	}
}

@Benchmark
public int[] filterValuesArrayInClassicWay() {
	int repetitions = REPS;
	dummy = new int[repetitions / References.BENCHMARK.MOD_VALUE];
	int j = 0;

	for (int i=0; i < repetitions; i++) {
		if (i % References.BENCHMARK.MOD_VALUE == 0) {
			dummy[j] = array[i];
			j++;
		}
	}
	return dummy;
}

@Benchmark
public List<Integer> filterValuesListInClassicWay() {
	int repetitions = REPS;
	dummyList = new ArrayList<Integer>();

	for (int i=0; i < repetitions; i++) {
		if (i % References.BENCHMARK.MOD_VALUE == 0) {
			dummyList.add(list.get(i));
		}
	}
	return dummyList;
}

@Benchmark
public int[] filterValuesArrayWithStream() {
	return IntStream
			.range(0, array.length)
			.filter(i -> i % References.BENCHMARK.MOD_VALUE == 0)
			.map(i -> array[i])
			.toArray();
}

@Benchmark
public int[] filterValuesArrayWithParallelStream() {
	return IntStream
			.range(0, array.length)
			.parallel()
			.filter(i -> i % References.BENCHMARK.MOD_VALUE == 0)
			.map(i -> array[i])
```

```
                        .toArray();
    }

    @Benchmark
    public List<Integer> filterValuesListWithStream() {
        return IntStream
                        .range(0, list.size())
                        .filter(i -> i % References.BENCHMARK.MOD_VALUE == 0)
                        .mapToObj(i -> list.get(i))
                        .collect(Collectors.toList());
    }

    @Benchmark
    public List<Integer> filterValuesListWithParallelStream() {
        return IntStream
                        .range(0, list.size())
                        .parallel()
                        .filter(i -> i % References.BENCHMARK.MOD_VALUE == 0)
                        .mapToObj(i -> list.get(i))
                        .collect(Collectors.toList());
    }
```

Diagramm und Beobachtung

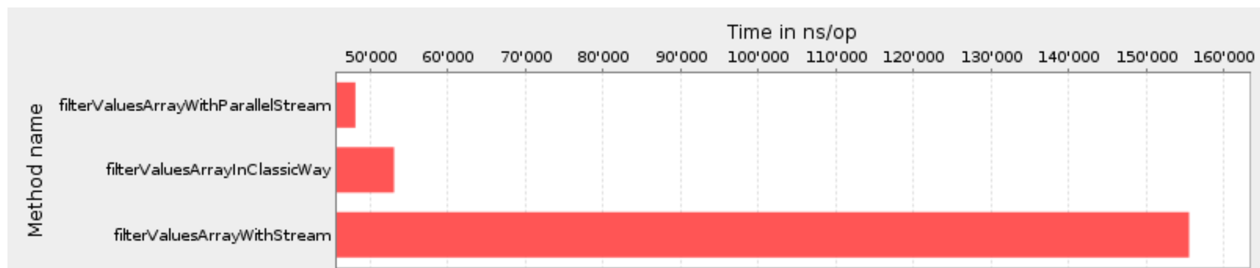

Abbildung 43: Laufzeiten der Array-Benchmarks aus der Klasse FilterValuesTest

Das Filtern des Arrays mit 100'000 int-Werten dauert mit der parallelen Stream-Verarbeitung am kürzesten. Auch das Filtern mit Hilfe der klassischen for-Schleife dauert nur etwa fünf Mikrosekunden mehr. Das Filtern des Arrays mit einem “normalen” Stream dauert knapp drei Mal so lange wie das Filtern mit paralleler Stream-Verarbeitung.

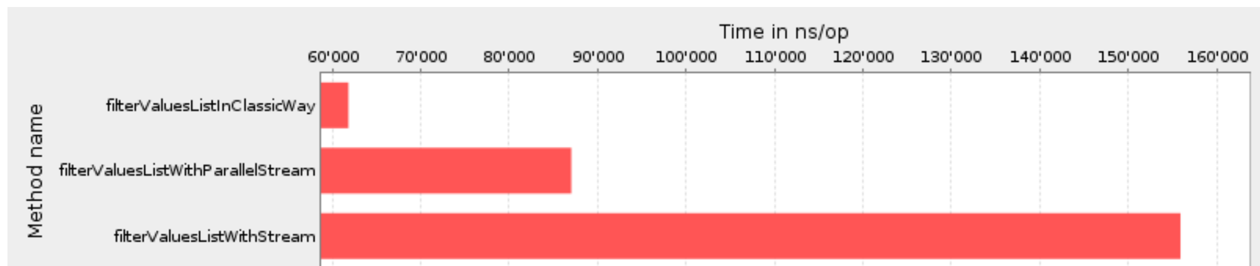

Abbildung 44: Laufzeiten der ArrayList-Benchmarks aus der Klasse FilterValuesTest

Bei der ArrayList mit 100'000 Integer-Werten ist hingegen das Filtern mit der klassischen for-Schleife am schnellsten. Mit mehr als 20 Mikrosekunden Abstand folgt das Filtern mit einem parallel geschalteten Stream. Die Verarbeitung mit dem “normalen” Stream dauert mehr als doppelt so lange wie das Filtern mit der klassischen for-Schleife.

Auswertung/Empfehlung

Es gibt einen Unterschied zwischen Arrays und ArrayLists. Für das Filtern von grossen Arrays sollten Streams mit der parallel()-Anweisung verwendet werden. Es ist jedoch zu beachten, dass mit sinkender Array-Grösse der Overhead für die Stream-Parallelisierung im Verhältnis zu der eigentlichen Filter-Operation unverhältnismässig gross werden kann und somit das Filtern mit parallelen Streams ineffizient wird. Für das Filtern von grossen ArrayLists sollte hingegen die for-Schleife gegenüber Streams bevorzugt werden.

4.10 Function-Composition und Methoden-Verkettung

Thema/Fragestellung

Ab Java 8 besteht die Möglichkeit, eine Function zu definieren, die ein Argument entgegennimmt und ein Resultat produziert. Eine Function kann beispielsweise eine Repräsentation einer mathematischen Funktion sein. Eine mathematische Funktion kann in Java aber auch mit Hilfe einer simplen Methode modelliert werden. Wie effizient sind diese verschiedenen Ansätze?

Vermutung/Hypothese

Da für die Function jeweils auf ein Objekt zugegriffen werden muss, ist der konventionelle Methoden-Verschachtelungsansatz vermutlich schneller als die Verwendung von Functions.

Programmcode

```
private int initValue = 3;

/*--- Lambda Section (Java 8) : Definitions ---*/
private Function<Integer, Integer> timesThree = a -> a * 3;
private Function<Integer, Integer> thirdPower = a -> a * a * a;

/*--- Lambda Section (Java 8) : Benchmarks ---*/
@Benchmark
public int functionCompositionWithLambdas() {
      return thirdPower.compose(timesThree).apply(initValue); // returns 729
}

@Benchmark
public int functionAndThenWithLambdas() {
      return timesThree.andThen(thirdPower).apply(initValue); // returns 729
}

/*--- Non-Lambda Section : Benchmarks ---*/
@Benchmark
public int methodChaining() {
      return thirdPower(timesThree(initValue)); // returns 729
}

/*--- Non-Lambda Section : Definitions ---*/
public int timesThree(int a) {
```

```
            return a * 3;
    }

    public int thirdPower(int a) {
            return a * a * a;
    }
```

Diagramm und Beobachtung

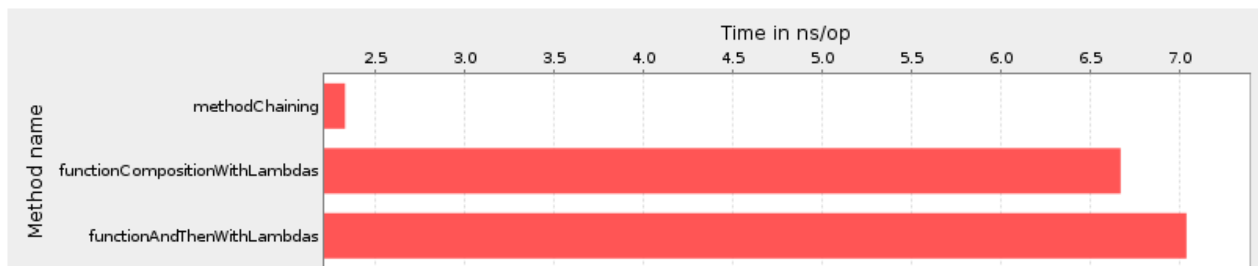

Abbildung 45: Laufzeiten der Benchmarks aus der Klasse FunctionVsMethodCompositionTest

Aus der Grafik ist ersichtlich, dass das Verwenden einer Function tatsächlich langsamer ist als das konventionelle Verketten von Methoden.

Auswertung/Empfehlung

Wird eine Function häufig eingesetzt, wird eine Performance-Einbusse eingegangen. Diese dürfte jedoch wenig ins Gewicht fallen, da sie im Regelfall tief ausfällt (mehrere Nanosekunden). Wie hoch die Einbusse bei teureren Operationen ausfallen würde, müsste jedoch in einem weiteren Schritt abgeklärt werden.

4.11 Fakultät berechnen

Thema/Fragestellung

Wie effizient ist die Fakultätberechnung von 200, 500, 1000, 3000 und 5000 mit …

- Iteration?
- Rekursion?
- Trampolin?

Das Trampolin optimiert eine endrekursive Methode, indem diese iterativ ausgeführt wird (Moertel 2013).

Vermutung/Hypothese

Die Berechnung mittels Rekursion wird vermutlich sehr ineffizient sein. Der Java Bytecode-Compiler sowie der JIT-Compiler können keine Tail-Call-Optimization vornehmen und somit die einfache Rekursion auch nicht optimieren (Ullenboom 2013).

Da der erste Benchmark bereits 200! berechnet, wird sich der Overhead für das Erstellen eines Trampolins bereits ab diesem Benchmark lohnen und vermutlich wird diese Variante gegenüber den anderen sehr effizient sein.

Programmcode

```
private BigInteger factorial200 = new BigInteger(String.valueOf(200));

//[...]

@Benchmark
public BigInteger factorial200RecursiveTrampoline() {
	return facRecTramp(factorial200, BigInteger.ONE).execute();
}

@Benchmark
public BigInteger factorial200Recursive() {
	return facRec(factorial200);
}

@Benchmark
public BigInteger factorial200Recursive2() {
	return facRec(factorial200, BigInteger.ONE);
}

@Benchmark
public BigInteger factorial200Iterative() {
	return facIter(factorial200);
}

//[...]


public BigInteger facRec(final BigInteger n) {
	if (n.longValue() <= 1) {
		return BigInteger.ONE;
	} else {
		return n.multiply(facRec(n.subtract(BigInteger.ONE)));
	}
}

public BigInteger facRec( final BigInteger n,
                          final BigInteger product) {
	if (n.longValue() <= 1) {
		return product;
	} else {
		return facRec(n.subtract(BigInteger.ONE), product.multiply(n));
	}
}

public Trampoline<BigInteger> facRecTramp( final BigInteger n,
                                           final BigInteger product) {
	if (n.longValue() <= 1) {
		return new Trampoline<BigInteger>() {
			@Override
			public BigInteger get() {
				return product;
			}
		};
	} else {
		return new Trampoline<BigInteger>() {
```

```
                        @Override
                        public Trampoline<BigInteger> run() {
                              return facRecTramp(n.subtract(BigInteger.ONE),
                                          product.multiply(n));
                        }
                  };
            }
      }

      private BigInteger facIter(BigInteger loopTopBound) {
            BigInteger result = BigInteger.ONE;

            for (BigInteger i = loopTopBound; i.compareTo(BigInteger.ZERO) > 0;
                              i = i.subtract(BigInteger.ONE)) {
                  result = result.multiply(i);
            }

            return result;
      }

      /**
       * This inner class is a custom implementation of a Trampoline
       *
       * @author pkukielka
       * @source https://gist.github.com/pkukielka/2842475
       * @param <T>
       */
      private class Trampoline<T> {
            public T get() {
                  return null;
            }

            public Trampoline<T> run() {
                  return null;
            }

            public T execute() {
                  Trampoline<T> trampoline = this;

                  while (trampoline.get() == null) {
                        trampoline = trampoline.run();
                  }

                  return trampoline.get();
            }
      }
```

Diagramm und Beobachtung

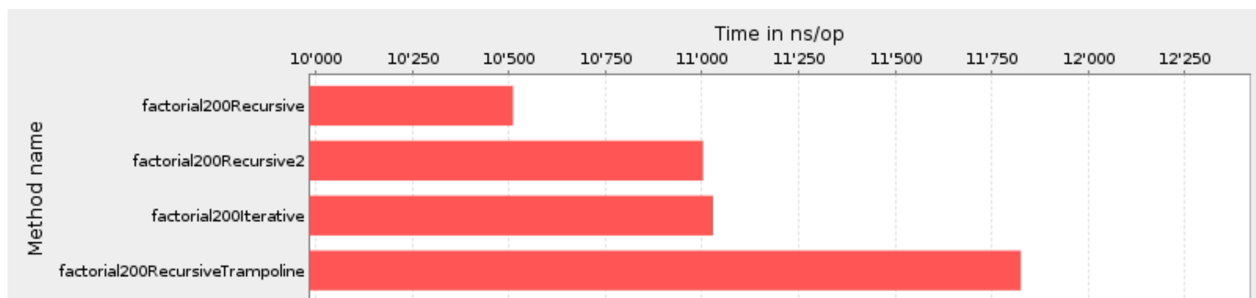

Abbildung 46: Laufzeiten der Benchmarks aus der Klasse IterativeVsRecursive200FactorialTest

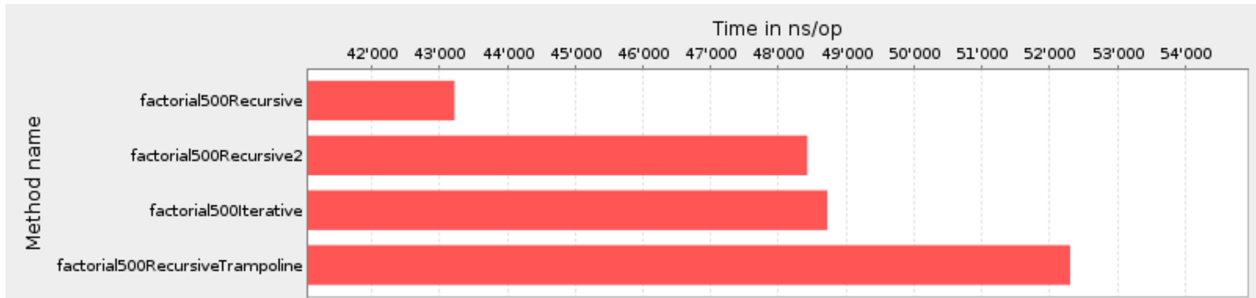

Abbildung 47: Laufzeiten der Benchmarks aus der Klasse IterativeVsRecursive500FactorialTest

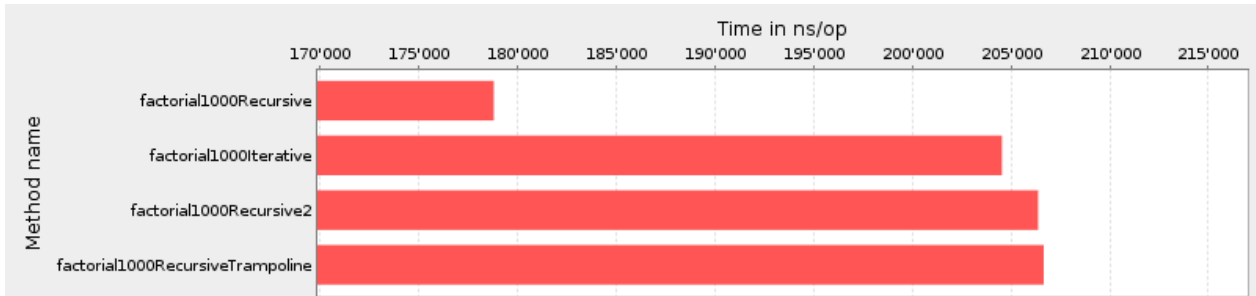

Abbildung 48: Laufzeiten der Benchmarks aus der Klasse IterativeVsRecursive1000FactorialTest

Erstaunlicherweise ist der Ansatz mit dem Trampolin allen anderen Programmieransätzen, mindestens bis zur Berechnung von 1000!, unterlegen. Bei 1000! ist die Variante "Iterative" schneller als "Recursive2".

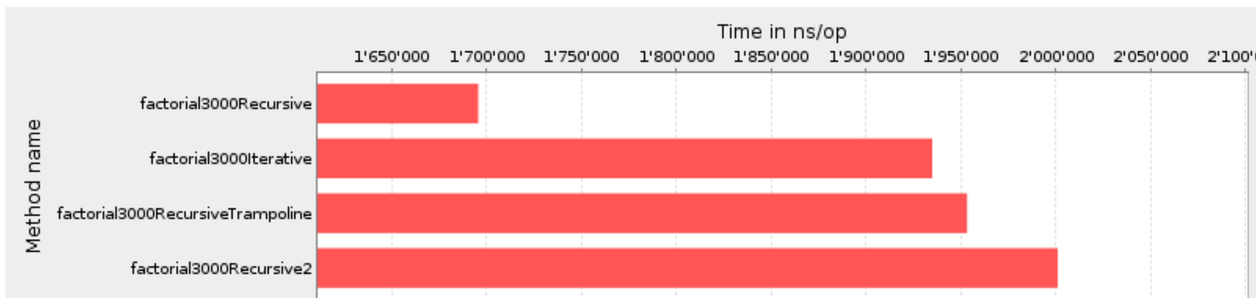

Abbildung 49: Laufzeiten der Benchmarks aus der Klasse IterativeVsRecursive3000FactorialTest

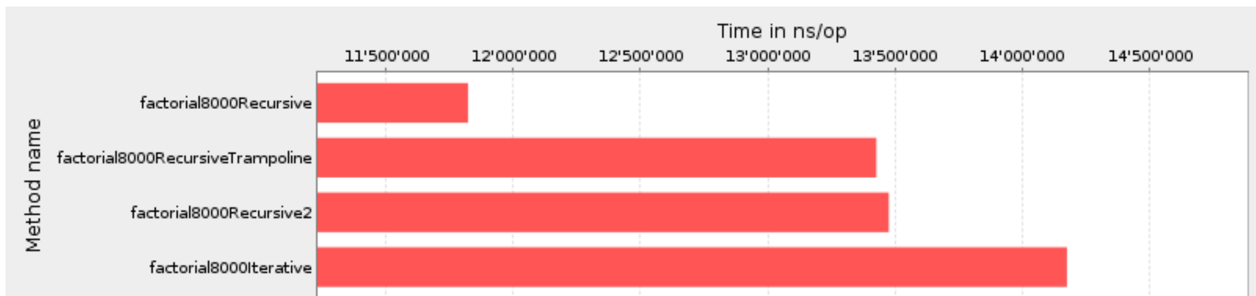

Abbildung 50: Laufzeiten der Benchmarks aus der Klasse IterativeVsRecursive8000FactorialTest

Der erste rekursive Ansatz ist bei allen fünf durchgeführten Faktultät-Berechnungen der schnellste Ansatz. Bei der Berechnung von 3000! erreicht der Ansatz mit dem Trampolin erstmals eine höhere Performance als die Variante "Recursive2". Bei der Berechnung von 8000! ist der Ansatz mit Trampolin sogar schneller als der iterative Ansatz. Eine Performance im Bereich der "Recursive"-Variante wird nicht erreicht.

Auswertung/Empfehlung

Es dauert doch einige Zeit, bis der Ansatz mit Trampolin seine Stärken gegenüber dem rekursiven Ansatz "Recursive2" oder auch dem iterativen Ansatz ausspielen kann. Deswegen müssen mögliche Performance-gewinne mittels Trampolin für jeden konkreten Anwendungsfall genauer angeschaut werden. Die Variante "Recursive2" ist in allen Fällen langsamer als die Variante "Recursive". Dies zeigt nochmals eindrücklich auf, dass der JIT-Compiler keine Tail-Call-Optimierung vornimmt (Ullenboom 2013). Die Variante "Recursive2" ruft nämlich eine endrekursive Methode auf, während die Variante "Recursive" dies nicht tut.

4.12 Mathematische Multiplikation und Division sowie Multiplikation oder Division mit Bitshift

Thema/Fragestellung

Wie effizient implementiert Java die mathematische Multiplikation und Division? Ist das Multiplizieren oder Dividieren mit Bitshift-Operationen effizienter?

Vermutung/Hypothese

Die Multiplikation und Division mit dem mathematischen Ansatz sind vermutlich gleich schnell wie mit dem Bitshift-Ansatz. Mit hoher Wahrscheinlichkeit setzt die Java Virtual Machine die mathematischen Operationen zu genau diesen Bitshift-Operationen um.

Programmcode

```
private int[] numbers = { 37831, 324266, 73923412, 45332747,
                          42545325, 74765326, 6836778, 84239849 };

private long dummy = 0;

@Benchmark
public long divisionMath() {
        dummy = 0;
        for (int num : numbers) {
                dummy += (num / 2);
        }
        return dummy;
}

@Benchmark
public long divisionShift() {
        dummy = 0;
        for (int num : numbers) {
                dummy += (num >> 1);
        }
        return dummy;
}
```

```
@Benchmark
public long multiplyMath() {
        dummy = 0;
        for (int num : numbers) {
                dummy += (num * 2);
        }
        return dummy;
}

@Benchmark
public long multiplyShift() {
        dummy = 0;
        for (int num : numbers) {
                dummy += (num << 1);
        }
        return dummy;
}
```

Diagramm und Beobachtung

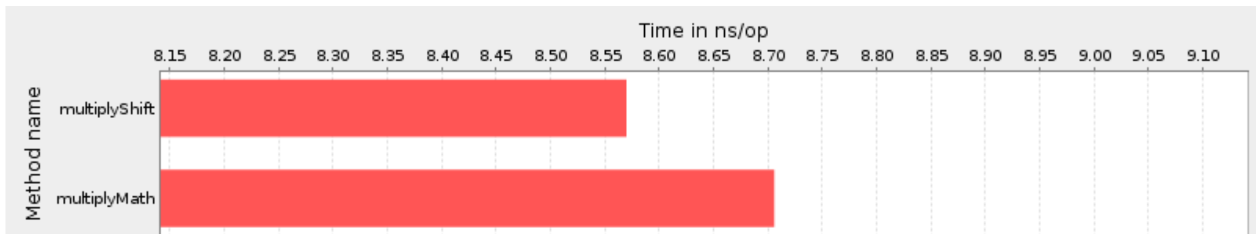

Abbildung 51: Laufzeiten der Multiplikation-Benchmarks aus der Klasse MathVsBitOperationsTest

Die Multiplikation mit dem mathematischen Ansatz beansprucht in etwa gleich viel Zeit wie die Bitshift-Multiplikation.

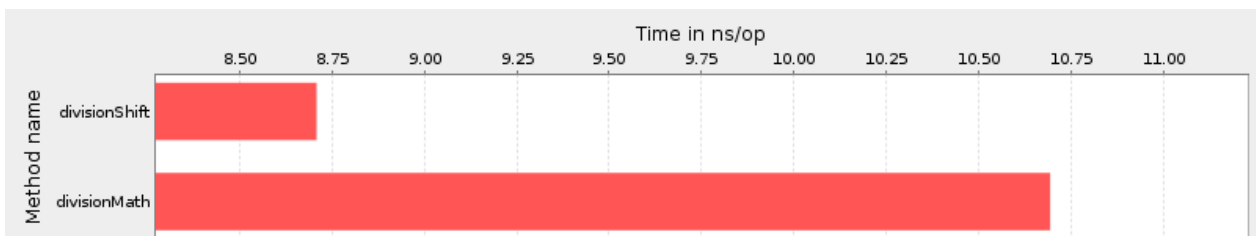

Abbildung 52: Laufzeiten der Division-Benchmarks aus der Klasse MathVsBitOperationsTest

Die Division mit dem Bitshift-Ansatz ist ein wenig schneller.

Auswertung/Empfehlung

Sofern nicht ein Höchstmass an Performance erreicht werden muss, sollte zum mathematischen Ansatz für das Multiplizieren und Dividieren von Zahlen gegriffen werden. Abgesehen davon, dass durch die Bitshift-Operation extrem wenig Zeit gespart wird, ist die Bitshift-Implementierung einigen Entwicklerinnen und Entwicklern auch nicht geläufig.

4.13 Objekt zurücksetzen oder ein neues Objekt verwenden

Thema/Fragestellung

Nach dem Zurücksetzen eines Objektes müssen alle Attribute des Objekts Default-Werte besitzen. Das Ziel des Programmierers ist es also, die Objektvariable neu verwenden zu können. Grundsätzlich gibt es zwei Arten, wie man dies im Java erreichen kann. Entweder werden mit Setter-Methoden alle Objektattribute auf ihre Default-Werte gesetzt oder es wird ein neues Objekt instanziiert und der Objektvariable zugewiesen.

Vermutung/Hypothese

Da beim Zurücksetzen der Objektattribute insgesamt mehr Speicherzugriffe getätigt werden müssen, könnte die Instanziierung eines neuen Objekts aus Sicht der Performance günstiger sein als das Zurücksetzen der Objektattribute. Allerdings kann auch die Instanziierung eines Objekts ziemlich teuer sein.

Programmcode

```
private Blackhole b = new Blackhole();
private Person p1 = new Person();
private Person p2 = new Person();


//basic runtime
@Benchmark
public Person basicRuntime() {
       p1 = new Person("Barack", "Obama", "Mr. President",
              "Pennsylvania Ave NW", 1600, "Washington", "DC", 20500);

       consumePersonDetails(p1);
       consumePersonDetails(p2);

       return p1;
}

@Benchmark
public Person resetPersonSetters(){
       p1 = new Person("Barack", "Obama", "Mr. President",
              "Pennsylvania Ave NW", 1600, "Washington", "DC", 20500);

       consumePersonDetails(p1);

       p1.setAddress("");
       p1.setFirstName("");
       p1.setPlace("");
       p1.setState("");
       p1.setStreetName("");
       p1.setStreetNumber(0);
       p1.setSurname("");
       p1.setZip(0);

       consumePersonDetails(p1);

       return p1;
}
```

```
@Benchmark
public Person resetPersonReInitialization(){
	p1 = new Person("Barack", "Obama", "Mr. President",
		"Pennsylvania Ave NW", 1600, "Washington", "DC", 20500);

	consumePersonDetails(p1);

	p1 = new Person();

	consumePersonDetails(p1);

	return p1;
}

/**
 * This method assures, that attributes that
 * have been set for a person are consumed.
 * @param person
 */
private void consumePersonDetails(Person person) {
	b.consume(person.getStreetNumber());
	b.consume(person.getZip());
	b.consume(person.getAddress());
	b.consume(person.getFirstName());
	b.consume(person.getPlace());
	b.consume(person.getState());
	b.consume(person.getStreetName());
	b.consume(person.getSurname());
}

private class Person implements Cloneable {
	private String firstName;
	private String surname;
	private String address;
	private String streetName;
	private int streetNumber;
	private String place;
	private String state;
	private int zip;

	//[...]

}
```

Diagramm und Beobachtung

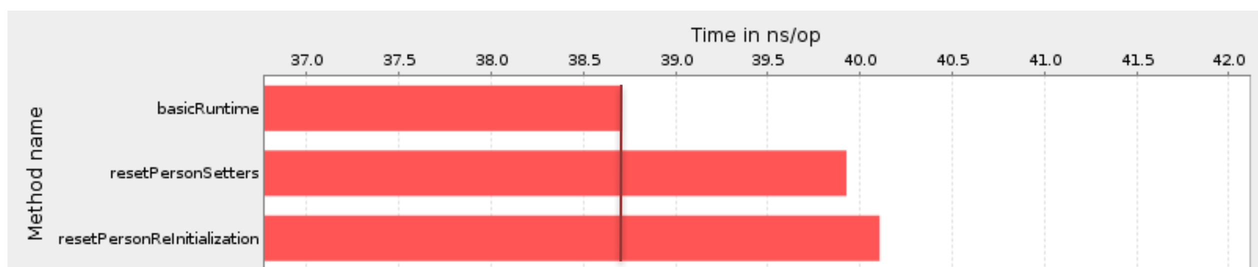

Abbildung 53: Laufzeiten der Cleaning-Benchmarks aus der Klasse ObjectCloningAndCleaningTest

Die basicRuntime, also die “Grund“-Laufzeit dieser Benchmarks fällt sehr hoch aus. Ein neues Person-Objekt zu instanziieren, ist deswegen praktisch gleich schnell wie das Zurücksetzen aller Objektattribute.

Auswertung/Empfehlung

Aus Sicht der Performance ist es egal, ob ein neues Objekt instanziiert und der Objektvariable zugewiesen wird oder alle Attribute des Objektes mit Setter-Aufrufen auf die Default-Werte zurückgesetzt werden. Allerdings könnte die Lage bei Objekten mit weniger oder mehr Attributen anders aussehen.

4.14 Objekte kopieren

Thema/Fragestellung

Ein Objekt ist gegeben und eine “Deep Copy” dieses Objektes soll erstellt werden. Bei einer “Deep Copy” muss das Resultat ein Objekt sein, dessen Zustand exakt dem Zustand des Ursprungsobjektes entspricht. Bei diesem Experiment wird auf drei verschiedene Arten eine “Deep Copy” desselben Objekts erstellt und dabei die Performance gemessen:

Bei der Variante clonePersonGettersAndSetters() werden zuerst alle Attribut-Werte des Ursprungobjektes mit Hilfe von Getter-Methoden ausgelesen. Danach wird ein neues Objekt instanziiert. Die ausgelesenen Attribut-Werte des Ursprungobjektes werden dann mit Hilfe von Setter-Methoden den richtigen Attributen des neu erstellten Objekts zugewiesen. Fertig ist die “Deep Copy” des Objektes.

Bei der Variante clonePersonGettersAndConstructor() werden ebenfalls zuerst alle Attribut-Werte des Ursprungobjektes mit Getter-Methoden ausgelesen. Beim Instanziieren des Zielobjektes wird hingegen der entsprechende Konstruktor verwendet und alle Attributwerte des Ursprungobjektes als Konstruktorargumente übergeben. Fertig ist die “Deep Copy” des Objektes.

Bei der Variante clonePerson() wird die Kopie des Objekts mit der Methode clone() der Klasse Object erstellt.

Vermutung/Hypothese

Das Kopieren mit der Methode clone() ist vermutlich am langsamsten. Es ist schwierig, einen Vergleich zwischen den beiden anderen Varianten zu ziehen. Es ist nämlich nicht offensichtlich, wie der JIT-Compiler diese beiden Varianten in Maschinencode umsetzt.

Programmcode

```
// [siehe auch Programmcode Kapitel 4.12]
// [...]

@Benchmark
public Person clonePerson() {
        try {
                p1 = new Person("Barack", "Obama", "Mr. President",
                        "Pennsylvania Ave NW", 1600, "Washington", "DC", 20500);
```

```
                p2 = (Person) p1.clone();

                consumePersonDetails(p1);
                consumePersonDetails(p2);

        } catch (CloneNotSupportedException e) {
                e.printStackTrace();
        }

        return p1;
}


@Benchmark
public Person clonePersonGettersAndConstructor() {
        p1 = new Person("Barack", "Obama", "Mr. President",
                "Pennsylvania Ave NW", 1600, "Washington", "DC", 20500);
        p2 = new Person(p1.getFirstName(), p1.getSurname(),
                      p1.getAddress(), p1.getStreetName(), p1.getStreetNumber(),
                      p1.getPlace(), p1.getState(), p1.getZip());

        consumePersonDetails(p1);
        consumePersonDetails(p2);

        return p1;
}


@Benchmark
public Person clonePersonGettersAndSetters() {
        p1 = new Person("Barack", "Obama", "Mr. President",
                "Pennsylvania Ave NW", 1600, "Washington", "DC", 20500);

        p2 = new Person();
        p2.setAddress(p1.getAddress());
        p2.setFirstName(p1.getFirstName());
        p2.setPlace(p1.getPlace());
        p2.setState(p1.getState());
        p2.setStreetName(p1.getStreetName());
        p2.setStreetNumber(p1.getStreetNumber());
        p2.setSurname(p1.getSurname());
        p2.setZip(p1.getZip());

        consumePersonDetails(p1);
        consumePersonDetails(p2);

        return p1;
}


//[...]


private class Person implements Cloneable {

        //[...]

        @Override
        public Object clone() throws CloneNotSupportedException {
            return super.clone();
        }

        //[...]
}
```

Diagramm und Beobachtung

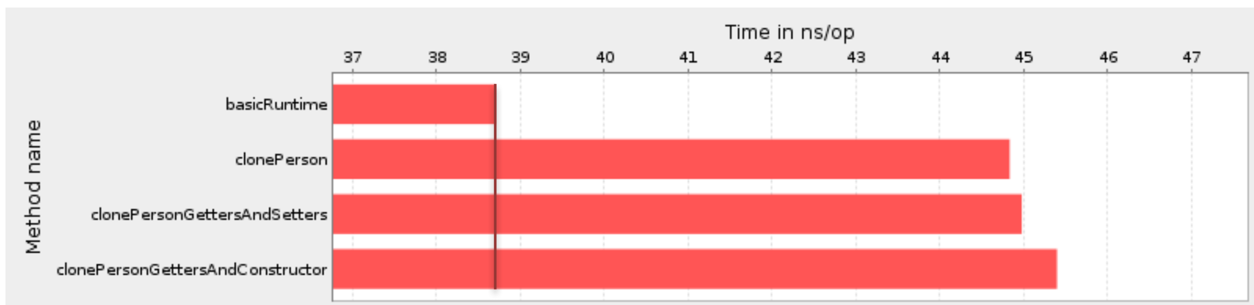

Abbildung 54: Laufzeiten der Benchmarks aus der Klasse ObjectCloningAndCleaningTest

Mit der clone()-Methode zu kopieren, ist die schnellste Variante.

Die Methoden clonePersonGettersAndSetters() und clonePersonGettersAndConstructor() sind ein wenig langsamer, auch wenn die Unterschiede nur winzig sind. Die basicRuntime beträgt knapp 39 Nanosekunden.

Auswertung/Empfehlung

Da die clone()-Methode wenig schneller ist als die zwei anderen Methoden, wird diese Kopiervariante empfohlen. Entwicklerinnen und Entwickler sollten sich aber darüber im Klaren sein, dass die Laufzeit aller dieser Methoden in Relation zu den zu kopierenden Attributen eines Objektes variiert.

4.15 Polymorphismus

Thema/Fragestellung

In der objektorientierten Programmierung haben Objekte die Fähigkeit, von anderen Klassen zu erben. Eine Klasse hat in Java maximal einen einzigen direkten Vorfahren. Durch Vererbung ist es beispielsweise möglich, dass ein Objekt eine Methode der Oberklasse aufrufen kann. In der Subklasse kann eine Methode der Oberklasse auch überschrieben werden. Deshalb muss die Java Virtual Machine zur Laufzeit feststellen können, aus welcher Klasse der Vererbungshierarchie die auszuführende Methode stammt. Wenn eine Methode in der Vererbungshierarchie mehrfach mit der gleichen Signatur anzutreffen ist und immer eine eigene Implementierung hat, ist sie polymorph.

Bei diesem Experiment geht es darum, herauszufinden, wie hoch der Performance-Unterschied ist, wenn in einer vierstufigen Vererbungshierarchie die Methode der untersten Klasse eine Methode der Oberklasse mit dem Keyword super aufruft, damit diese eine simple Addition durchführen kann. Die Oberklasse kann wiederum die Methode ihrer Oberklasse aufrufen und dies

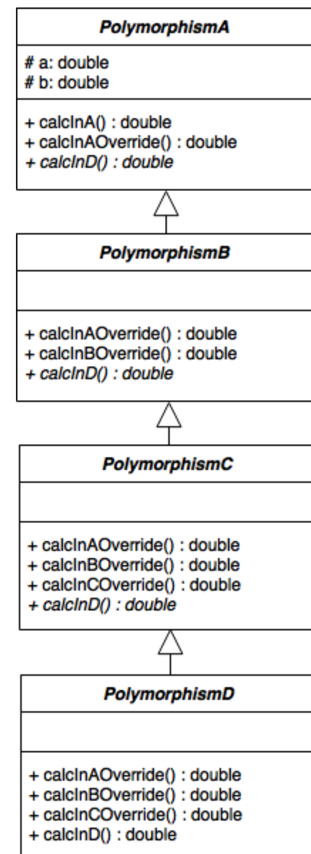

Abbildung 55: Vierstufige Vererbungshierarchie (Eigene Abbildung)

kann so weitergehen. Die Frage ist nun beispielsweise, ob die Addition in der Klasse D schneller ist als wenn die Addition in der Vererbungshierarchie von der Klasse D aus weiterdelegiert wird. Eine weitere Frage ist, wie schnell die Addition ist, wenn die Methode aus der Oberklasse A aufgerufen wird.

Vermutung/Hypothese

Die Vermutung ist, dass die Addition am schnellsten ist, wenn sie direkt in der untersten Klasse der Vererbungshierarchie ausgeführt wird.

Programmcode

```
// [zum Verständnis: siehe Klassendiagramm, Abbildung 55]
private PolymorphismD polyD = new PolymorphismD();

@Benchmark
@CompilerControl(Mode.DONT_INLINE)
public double calcInD() {
    return polyD.calcInD();
}

@Benchmark
@CompilerControl(Mode.DONT_INLINE)
public double calcInCOverride() {
    return polyD.calcInCOverride();
}

@Benchmark
@CompilerControl(Mode.DONT_INLINE)
public double calcInBOverride() {
    return polyD.calcInBOverride();
}

@Benchmark
@CompilerControl(Mode.DONT_INLINE)
public double calcInAOverride() {
    return polyD.calcInAOverride();
}

@Benchmark
@CompilerControl(Mode.DONT_INLINE)
public double calcInA() {
    return polyD.calcInA();
}
```

Diagramm und Beobachtung

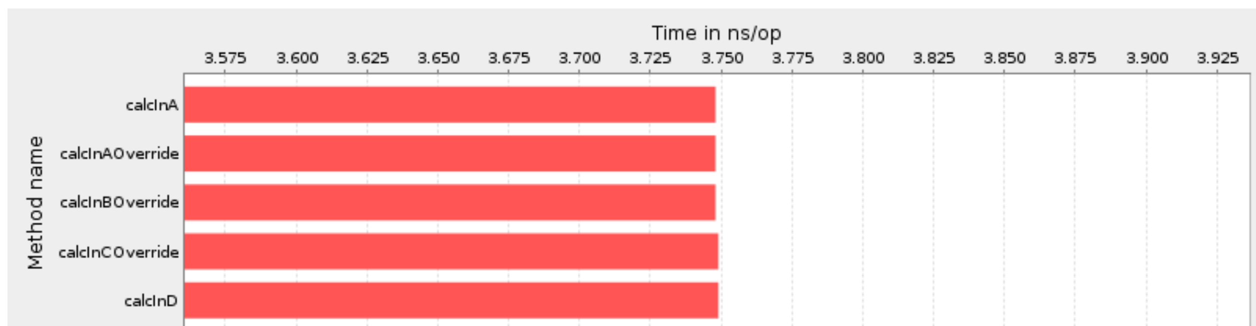

Abbildung 56: Laufzeiten der Benchmarks aus der Klasse PolymorphismTest

Es sind keine Unterschiede feststellbar.

Auswertung/Empfehlung

Das Resultat dieses Experiments ist sehr überraschend, da die Vermutung nicht mit dem Resultat übereinstimmt. Da bei einer vierstufigen Vererbungshierarchie keine Laufzeitunterschiede feststellbar sind, ist anzunehmen, dass Polymorphismus an sich wenig Zeit benötigt.

4.16 Mehrfach vorhandene Werte aus ArrayLists oder Arrays entfernen

Thema/Fragestellung

In Java gibt es mehrere Möglichkeiten, einen mehrfach vorhandenen Wert aus einer ArrayList oder einem Array zu entfernen. Dies kann durch klassische Schleifen realisiert werden. Eine weitere Möglichkeit ist die Umwandlung einer ArrayList zu einem HashSet und die anschliessende Zurückwandlung in eine ArrayList. Ab Java 8 gibt es zusätzlich die Stream API, die den Programmierhorizont erweitert. Aus einer Integer-ArrayList oder einem int-Array wird bei dieser Variante ein Stream erzeugt. Die Methode distinct() wird auf dem Stream aufgerufen. Sie stellt sicher, dass Werte in diesem Stream nur noch einmal vorkommen. Aus dem Stream kann schlussendlich wieder ein Array oder eine Liste erstellt werden. Bei der Stream-Verarbeitung kann zwischen einem sequenziellen und einem parallelen Stream gewählt werden. Die Programmiererin oder den Programmierer interessiert nun: Welche dieser Varianten erbringt die höchste Performance?

Vermutung/Hypothese

Die Varianten, die es vor Java 8 schon gab, sind aus Sicht der Performance wahrscheinlich am effizientesten, da der JIT-Compiler bei diesen die besten Optimierungen durchführen kann.

Programmcode

```
private static final int REPS = References.BENCHMARK.REPETITIONS /
                                References.BENCHMARK.DIVISOR_REPETITIONS;

private int[] array1 = new int[REPS];


private List<Integer> list1;

@Setup(Level.Trial)
public void setUpArray() {
        for (int i = 0; i < REPS; i++) {
                array1[i] = (int) (Math.random() * References.BENCHMARK.RANGE);
        }
```

```
		list1 = new ArrayList<Integer>();
		for (int i = 0; i < REPS; i++) {
			list1.add(array1[i]);
		}
	}

	@Benchmark
	public int[] distinctArrayElementsInClassicWay() {
		Set<Integer> dummySet = new LinkedHashSet<Integer>();
		for (Integer num : array1) {
			dummySet.add(num);
		}

		int[] intResultArray = new int[dummySet.size()];
		int i = 0;
		for (Integer currentInt : dummySet) {

			intResultArray[i] = currentInt;
			i++;
		}
		return intResultArray;
	}

	@Benchmark
	public List<Integer> distinctListElementsInClassicWay() {
		return new ArrayList<>(new LinkedHashSet<>(list1));
	}

	@Benchmark
	public int[] distinctArrayElementsStream() {
		return Arrays.stream(array1).distinct().toArray();

	}

	@Benchmark
	public List<Integer> distinctListElementsStream() {
		return list1.stream().distinct().collect(Collectors.toList());
	}

	@Benchmark
	public int[] distinctArrayElementsParallelStream() {
		return Arrays.stream(array1).parallel().distinct().toArray();
	}

	@Benchmark
	public List<Integer> distinctListElementsParallelStream() {
		return list1.parallelStream().distinct().collect(Collectors.toList());
	}
```

Diagramm und Beobachtung

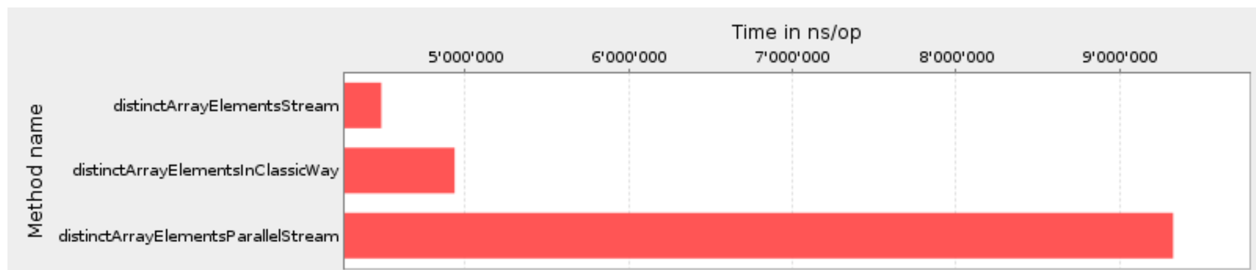

Abbildung 57: Laufzeiten der Benchmarks aus der Klasse RemoveDuplicateValuesInArrayTest

Das Löschen von mehrfach vorhandenen Werten im Array ist mit einem Stream am schnellsten. Die Variante mit der Schleife ist wenig langsamer. Die "ParallelStream"-Variante ist hingegen klar am langsamsten.

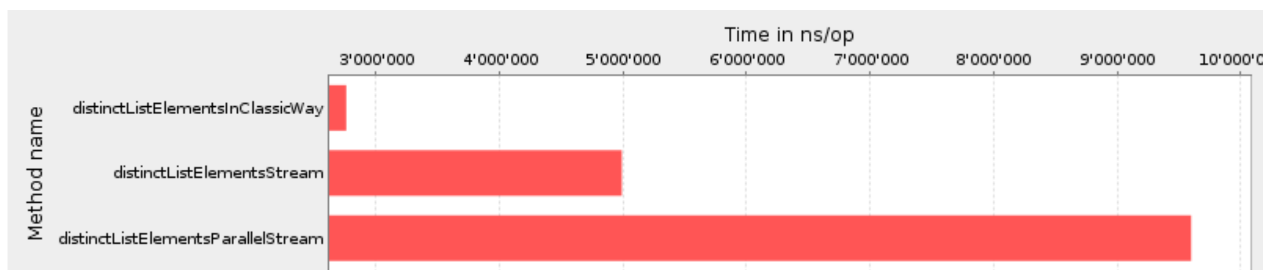

Abbildung 58: Laufzeiten der Benchmarks aus der Klasse RemoveDuplicateValuesInListTest

Beim klassischen Ansatz zum Entfernen von mehrfach vorhandenen Werten aus einer ArrayList muss im Gegensatz zum Array keine Schleife durchlaufen werden. Bei dieser Variante wird nämlich ein Trick angewandt. Der Trick besteht darin, dass eine Liste zuerst in ein Set umgewandelt und danach wieder zurück in eine Liste konvertiert werden kann. Hierbei entfallen die mehrfach vorhandenen Werte in der Liste. Dieser Trick ermöglicht eine schnellere Laufzeit als die Verwendung eines Streams.

Auswertung/Empfehlung

Wenn aus einer Liste mehrfach vorhandene Werte entfernt werden sollen, geht dies mit klassischen Hilfsmitteln, die bereits vor Java 8 existierten, am einfachsten und effizientesten. Da bei Arrays die klassische Variante etwas umständlich ist, wird die Stream-Variante empfohlen, die zudem auch schneller ist. Der parallele Stream ist hingegen deutlich langsamer und deswegen in beiden Fällen nicht empfehlenswert.

4.17 Listen oder Arrays sortieren

Thema/Fragestellung

Um ArrayLists oder Arrays in Java zu sortieren, können verschiedene Programmierkonstrukte eingesetzt werden. Einerseits bieten die Klassen Arrays und Collections die Methoden sort() oder parallelSort() an. Andererseits kann ab Java 8 die Stream API gebraucht werden, um ArrayLists oder Arrays zu sortieren. In

diesem Benchmark wird eine Integer-ArrayList sowie ein int-Array sortiert. Die Sortierung kann dabei sequentiell oder parallel ausgeführt werden. Dieses Experiment soll Entwicklerinnen und Entwicklern aufzeigen, welche Variante sie in Bezug auf Geschwindigkeit verwenden sollten.

Vermutung/Hypothese

Die Stream API ist wahrscheinlich langsamer als die herkömmliche Java API.

Programmcode

```
private static final int REPS = References.BENCHMARK.REPETITIONS
                                / References.BENCHMARK.DIVISOR_REPETITIONS;
private int[] array = new int[REPS];
private List<Integer> list;
private int[] dummy;
private List<Integer> dummyList;

@Setup(Level.Trial)
public void setUpArray() {
        for (int i = 0; i < REPS; i++) {
                array[i] = (int) (Math.random() * References.BENCHMARK.RANGE);
        }
        list = new ArrayList<Integer>();
        for (int i = 0; i < REPS; i++) {
                list.add(array[i]);
        }
}

@Benchmark
public int[] sortArrayInClassicWay() {
        dummy = Arrays.copyOf(array, array.length);
        Arrays.sort(dummy);
        return dummy;
}

@Benchmark
public List<Integer> sortListInClassicWay() {
        dummyList = new ArrayList<>(list);
        Collections.sort(dummyList);
        return dummyList;
}

@Benchmark
public int[] parallelSortArrayInClassicWay() {
        dummy = Arrays.copyOf(array, array.length);
        Arrays.parallelSort(dummy);
        return dummy;
}

@Benchmark
public List<Integer> parallelSortListInClassicWay() {
        Integer[] arr = new Integer[list.size()];
        arr = list.toArray(arr);
        Arrays.parallelSort(arr);
        return Arrays.asList(arr);
}

@Benchmark
public int[] sortArrayWithStream() {
        return Arrays
                        .stream(array)
```

```
                        .sorted()
                        .toArray();
    }

    @Benchmark
    public List<Integer> sortListWithStream() {
        return list
                        .stream()
                        .sorted()
                        .collect(Collectors.toList());
    }

    @Benchmark
    public int[] sortArrayWithParallelStream() {
        return Arrays
                        .stream(array)
                        .parallel()
                        .sorted()
                        .toArray();
    }

    @Benchmark
    public List<Integer> sortListWithParallelStream() {
        return list
                        .parallelStream()
                        .sorted()
                        .collect(Collectors.toList());
    }
```

Diagramm und Beobachtung

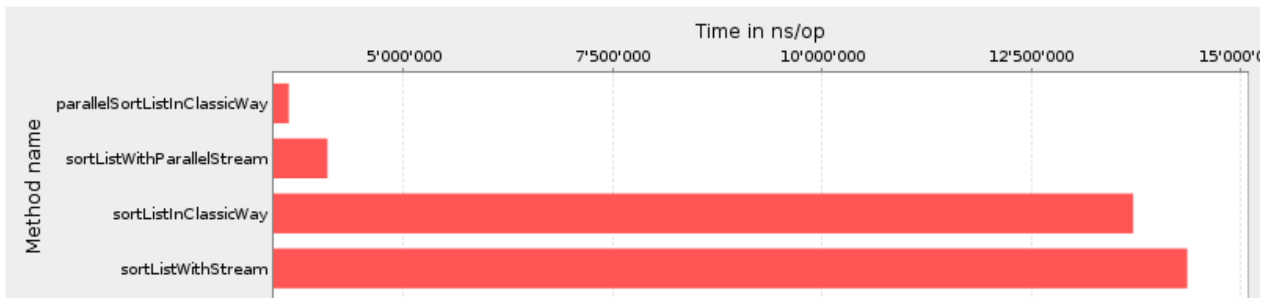

Abbildung 59: Laufzeiten der ArrayList-Benchmarks aus der Klasse SortArraysAndArrayListsTest

Der Benchmark parallelSortListInClassicWay() ist eindeutig der schnellste Benchmark. Die Sortierung der Liste mit einem parallelen Stream dauert ein bisschen länger und die restlichen Benchmarks dauern viel länger. Dies ist aber verständlich, da sortListInClassicWay() und sortListWithStream() nicht parallel sortieren.

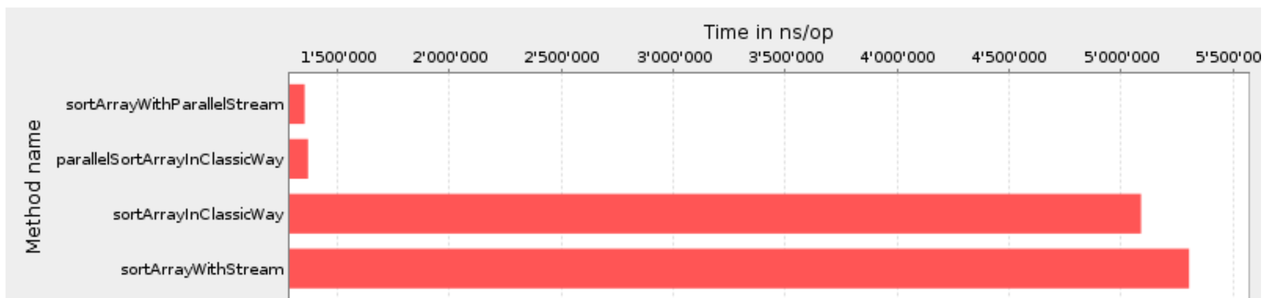

Abbildung 60: Laufzeiten der Array-Benchmarks aus der Klasse SortArraysAndArrayListsTest

Bei der Sortierung eines Arrays ist die Methode sortArrayWithParallelStream() ein bisschen schneller als die Methode parallelSortArrayInClassicWay(). Bei der sequenziellen Sortierung ist die klassische Sortierung schneller als die Sortierung mit der Stream API.

Auswertung/Empfehlung

Während bei Kapitel 4.16 ersichtlich ist, dass das Parallelisieren beim Entfernen von Duplikaten in ArrayLists oder Arrays nicht lohnenswert ist, sieht es bei diesem Experiment schon anders aus. Es lohnt sich, ein Array oder eine ArrayList parallel zu sortieren. Bei der ArrayList ist es zweifellos performanter, die ArrayList in ein Integer-Array umzuwandeln und Arrays.parallelSort() aufzurufen anstatt parallel().sorted() der Stream API anzuwenden. Beim Array sieht es anders aus. Das Sortieren eines parallelen Streams ist ein bisschen schneller als Arrays.parallelSort(). Wenn ein Array sequenziell sortiert werden muss, ist Arrays.sort() schneller als die Sortierung eines Streams.

4.18 Aufruf einer statischen Methode im Gegensatz zu einer nicht-statischen Methode

Thema/Fragestellung

Die Methode einer Klasse kann static oder nicht static sein. Dies ist vom Klassendesign abhängig. Eine Methode kann als static deklariert werden, wenn die Methode etwas durchführt, das unabhängig vom Objektzustand ist. Bei diesem Experiment geht es darum, zu messen, wie schnell der statische Aufruf einer Methode im Gegensatz zu einer nicht-statischen Methode ist. In beiden Fällen führt die aufgerufene Methode eine kleine mathematische Operation aus. Da der JIT-Compiler Code-Inlining vornimmt (Optimierung JIT.O3, siehe Kapitel 2.4), wurde der Mode DONT_INLINE der CompilerControl-Annotation gesetzt. Dies verhindert das Code Inlining dieser Methode und misst so auch die Zeit, die für den Methoden-Aufruf benötigt wird.

Vermutung/Hypothese

Die statische Methode ist wahrscheinlich schneller als die nicht-statische Methode, da nicht auf das Objekt an sich zugegriffen werden muss.

Programmcode

```
private StaticVsNonStaticMethods svnsm;

@Setup(Level.Iteration)
public void setNewInstance() {
	svnsm = new StaticVsNonStaticMethods();
}
```

```
@Benchmark
@CompilerControl(Mode.DONT_INLINE)
public double measureNonStaticMethod() {
       return svnsm.doNonStaticCalculation();
}

@Benchmark
@CompilerControl(Mode.DONT_INLINE)
public double measureStaticMethod() {
       return StaticVsNonStaticMethods.doStaticCalculation();
}
```

Diagramm und Beobachtung

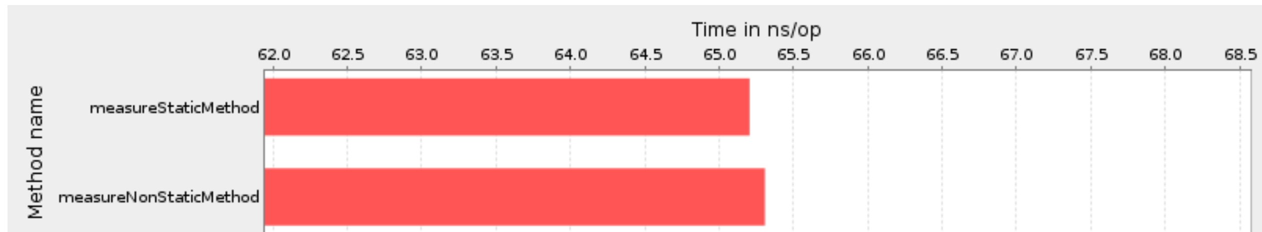

Abbildung 61: Laufzeiten der Benchmarks aus der Klasse StaticVsNonStaticMethodsTest

Der Benchmark measureStaticMethod() ist um einen Bruchteil einer Nanosekunde schneller als der Benchmark measureNonStaticMethod().

Auswertung/Empfehlung

Die statische Methode ist nur sehr wenig schneller als die nicht-statische Methode. Der Einsatz von statischen Methoden bringt also, bezüglich der Laufzeit, gegenüber den nicht-statischen Methoden, nicht wirklich einen Vorteil. Die Unabhängigkeit einer statischen Methode vom Objektzustand ihrer Klasse kann hingegen für die Entwicklerin oder den Entwickler praktisch sein und in Anbetracht dieses Aspektes wird die Verwendung von statischen Methoden als Utility-Methoden trotzdem empfohlen.

4.19 Die Verkettung von Strings

Thema/Fragestellung

Strings können in Java mit dem Plus-Operator verkettet werden. Dieser Programmieransatz hat jedoch den Ruf, bei mehreren Verkettungen nur eine tiefe Performance zu erzielen. Eine bekannte Klasse, die diese Aufgabe effizienter erledigt, ist die Klasse StringBuilder. Es gibt auch weitere Klassen wie StringBuffer und ab Java 8 die Klasse StringJoiner. Es stellt sich nun die Frage, mit welchem Verfahren die schnellste Performance erreicht wird.

Vermutung/Hypothese

Die Verkettungen mit dem Plus-Operator und dem Plus-Gleich-Operator sind mit hoher Wahrscheinlichkeit die langsamsten aller Varianten.

Programmcode

```
private String string = "Lorem ipsum dolor sit amet, ... ";
// Original-String im Code ist um einiges länger ...

private String[] strings = string.split(" ");

@Benchmark
public String loopPlus() {
	String string = "";
	for (String s : strings) {
		string = string + s;
	}
	return string;
}

@Benchmark
public String loopPlusEqual() {
	String string = "";
	for (String s : strings) {
		string += s;
	}
	return string;
}

@Benchmark
public String loopStringBuffer() {
	StringBuffer stringBuffer = new StringBuffer();
	for (String s : strings) {
		stringBuffer.append(s);
	}
	return stringBuffer.toString();
}

@Benchmark
public String loopStringBuilder() {
	StringBuilder stringBuilder = new StringBuilder();
	for (String s : strings) {
		stringBuilder.append(s);
	}
	return stringBuilder.toString();
}

@Benchmark
public String loopStringJoiner() {
	StringJoiner stringJoiner = new StringJoiner("");
	for (String s : strings) {
		stringJoiner.add(s);
	}
	return stringJoiner.toString();
}
```

Diagramm und Beobachtung

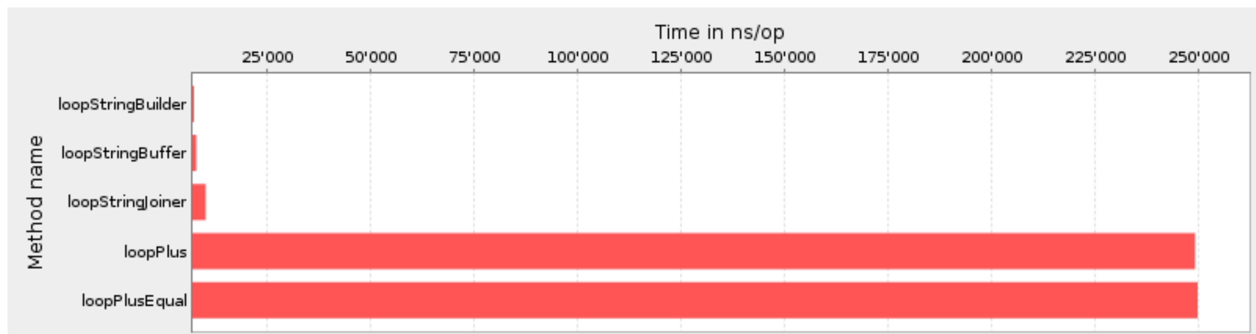

Abbildung 62: Laufzeiten der Benchmarks aus der Klasse StringConcatenationTest

Der Benchmark loopStringBuilder() ist am schnellsten. Die Performance von loopStringBuffer() liegt aber nicht weit entfernt von loopStringBuilder(). Die Verkettung mit dem StringJoiner ist etwas langsamer als die beiden schnellsten Varianten. Auffällig ist aber, dass loopPlus() und loopPlusEqual() extrem viel langsamer sind.

Auswertung/Empfehlung

Um Strings zu verketten sollte entweder der StringBuilder oder der StringBuffer verwendet werden. In einigen Fällen macht der Einsatz von StringJoiner auch Sinn. Beispielsweise wenn bei der Verkettung von Strings weitere Zeichen zwischen den Strings eingefügt werden müssen. Der StringJoiner eignet sich auch, wenn ein String als Präfix oder Suffix einem String angehängt werden muss (Oracle 2016 A).

Wie bereits in Kapitel 2.3 (Optimierung JAC.O4) erklärt, ersetzt der Bytecode-Compiler den Plus-Operator durch ein StringBuilder-Objekt. Es wird aber nicht bloss ein einziges StringBuilder-Objekt gebraucht, sondern bei jedem Statement, bei dem Strings verkettet werden, wird ein neues StringBuilder-Objekt instanziert. Deswegen wird von der Verwendung des Plus- bzw. des Plus-Gleich-Operators abgeraten.

4.20 Das Vergleichen von Strings

Thema/Fragestellung

Java bietet diverse Möglichkeiten, Strings miteinander zu vergleichen. Bei diesem Experiment werden zwei Strings auf Gleichheit überprüft, indem zwei verschiedene Verfahren gewählt werden. Einmal wird die Methode equals() von der String-Klasse verwendet. Beim zweiten Mal wird die Methode equals() der Collator-Klasse eingesetzt. Welches Verfahren ist performanter?

Vermutung/Hypothese

Die Hypothese ist, dass equals() von String schneller ist als equals() von Collator, weil die Collator-Klasse Strings genauer analysiert und vergleicht.

Programmcode

```
private String string = "Lorem ipsum dolor sit amet, ... ";
// Original-String im Code ist um einiges länger ...
private String differentString = "Lorem ipsum dolor sit amet, ... ";
// Ist ebenfalls um einiges länger und weicht von "string" minimal ab ...

// [...]
private Collator collator;
// [...]

private Blackhole b = new Blackhole();

@Setup(Level.Trial)
public void setupTest() {
        // [...]
        collator = Collator.getInstance();
}

// [...]

// -- compare two strings -- //
@Benchmark
public boolean compareStringsByStringClass() {
        return string.equals(differentString);
}

@Benchmark
public boolean compareStringsByCollatorClass() {
        return collator.equals(string, differentString);
}

// [...]
```

Diagramm und Beobachtung

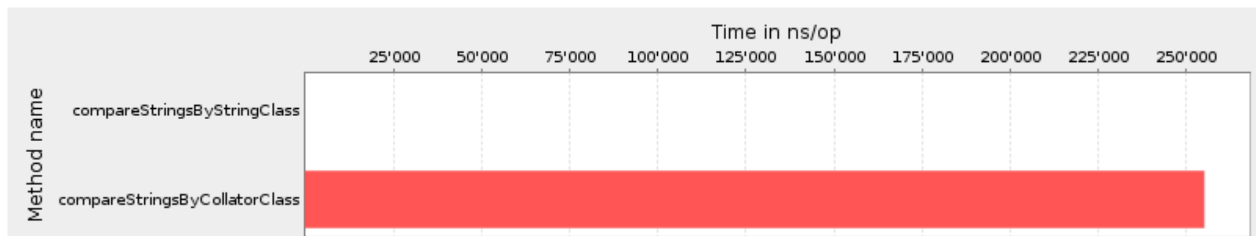

Abbildung 63: String-Vergleich-Benchmarks aus der Klasse StringSplittingAndComparingTest

Die Prüfung auf Gleichheit zweier Strings ist mit der String-Klasse viel schneller als mit der Collator-Klasse.

Auswertung/Empfehlung

Wenn ein einfacher String-Vergleich durchgeführt werden muss, um zu wissen, ob zwei Strings die gleichen Zeichen in der gleichen Reihenfolge enthalten, kann equals() der String-Klasse eingesetzt werden (Oracle 2016 A). Müssen zwei Strings aus einer länderspezifischen Locale verglichen werden, bietet die Collator-Klasse jedoch eine bessere API. Denn Collator beachtet die Regeln der entsprechenden Locale und prüft auf dieser Weise die Strings genauer als equals() der String-Klasse (Oracle 2016 A).

4.21 Die Trennung von Strings

Thema/Fragestellung

Ein String kann in einzelne Zeichen getrennt werden. Um dies zu realisieren, gibt es viele Möglichkeiten. Die String-Klasse bietet die Methode split(...) an, die ein String-Array zurückgibt. Alternativ kann in einer Schleife mit der Methode charAt(...) der String-Klasse jedes einzelne Zeichen ausgelesen werden. Eine weitere Möglichkeit ist, ein regulärer Ausdruck einzusetzen, um mit der Pattern-Klasse ein String zu trennen. Um dieselbe Aufgabe zu erledigen, bietet die String-Klasse die Methode toCharArray(). Eine weitere Alternative ist der BreakIterator. Welche dieser Möglichkeiten erzielt die schnellste Ausführung?

Ein String kann auch in Wörtern getrennt werden. Diesbezüglich gibt es folgende Wege, die Entwicklerinnen bzw. Entwickler einschlagen können: Sie können die Methode split(...) der String-Klasse anwenden, einen regulären Ausdruck mit der Pattern-Klasse konstruieren und den String damit trennen, einen Tokenizer einsetzen, einen BreakIterator gebrauchen oder einen Scanner verwenden. Welches Verfahren die höchste Performance aufzeigt, wird sich zeigen.

Vermutung/Hypothese

Welche Variante die Schnellste ist, um ein String in einzelne Zeichen zu trennen, ist schwierig vorherzusagen. Bei der Trennung in Wörtern ist vermutlich der Tokenizer die schnellste Art, wie ein String in Wörter getrennt werden kann.

Programmcode

```
// [siehe auch Programmcode Kapitel 4.18]

private String string = "Lorem ipsum dolor sit amet, ... ";
// Original-String im Code ist um einiges länger ...
private String differentString = "Lorem ipsum dolor sit amet, ... ";
// Ist ebenfalls um einiges länger und weicht von “string” minimal ab ...

private BreakIterator biChar;
private BreakIterator biLine;
//[...]
private Pattern pattern;
private StringTokenizer st;
private Scanner scanner;

private Blackhole b = new Blackhole();

@Setup(Level.Trial)
public void setupTest() {
       biChar = BreakIterator.getCharacterInstance();
       biLine = BreakIterator.getLineInstance();
       //[...]
}

// --- split by characters --- //
```

```java
	@Benchmark
	public void splitIntoCharsByStringClassSplit() {
		String[] splittedText = string.split(""); // split on every character
		consumeEachInArray(splittedText);
	}

	@Benchmark
	public void splitIntoCharsByStringClassCharAt() {
		consumeEachWithCharAt(string);
	}

	@Benchmark
	public void splitIntoCharsByCreatingCharArray() {
		char[] charArray = string.toCharArray();
		consumeEachInArray(charArray);
	}

	@Benchmark
	public void splitIntoCharsByRegexPattern() {
		pattern = Pattern.compile(""); // split on every character
		String[] splittedText = pattern.split(string);
		consumeEachInArray(splittedText);
	}

	@Benchmark
	public void splitIntoCharsByBreakIterator() {
		biChar.setText(string);
		consumeEachForward(biChar, string);
	}

	// --- split by words --- //
	@Benchmark
	public void splitIntoWordsByStringClass() {
		String[] splittedText = string.split(" ");
		consumeEachInArray(splittedText);
	}

	@Benchmark
	public void splitIntoWordsByRegexPattern() {
		pattern = Pattern.compile(" ");
		String[] splittedText = pattern.split(string);
		consumeEachInArray(splittedText);
	}

	@Benchmark
	public void splitIntoWordsByStringTokenizer() {
		consumeWithST(" ");
	}

	@Benchmark
	public void splitIntoWordsByBreakIterator() {
		biLine.setText(string);
		consumeEachForward(biLine, string);
	}

	@Benchmark
	public void splitIntoWordsByScanner() {
		scanner = new Scanner(string);
		scanner.useDelimiter(" ");
		consumeWithScanner();
	}

	//[...]

	// -- consume methods -- //
	private void consumeWithScanner() {
```

```
            while (scanner.hasNext()) {
                    String part = scanner.next();
                    b.consume(part);
            }
    }

    private void consumeWithST(String delim) {
            st = new StringTokenizer(string, delim);
            while (st.hasMoreTokens()) {
                    String nextToken = st.nextToken();
                    b.consume(nextToken);
            }
    }

    private void consumeEachWithCharAt(String s) {
            for (int i = 0; i < s.length(); i++) {
                    b.consume(s.charAt(i));
            }
    }

    private void consumeEachInArray(char[] charArray) {
            for (int i = 0; i < charArray.length; i++) {
                    b.consume(charArray[i]);
            }
    }

    private void consumeEachInArray(Object[] objArray) {
            for (int i = 0; i < objArray.length; i++) {
                    b.consume(objArray[i]);
            }
    }

    private void consumeEachForward(BreakIterator boundary, String source) {
            int start = boundary.first();
            for (int end = boundary.next(); end != BreakIterator.DONE;
                                    start = end, end = boundary.next()) {
                    b.consume(source.substring(start, end));
            }
    }
```

Diagramm und Beobachtung

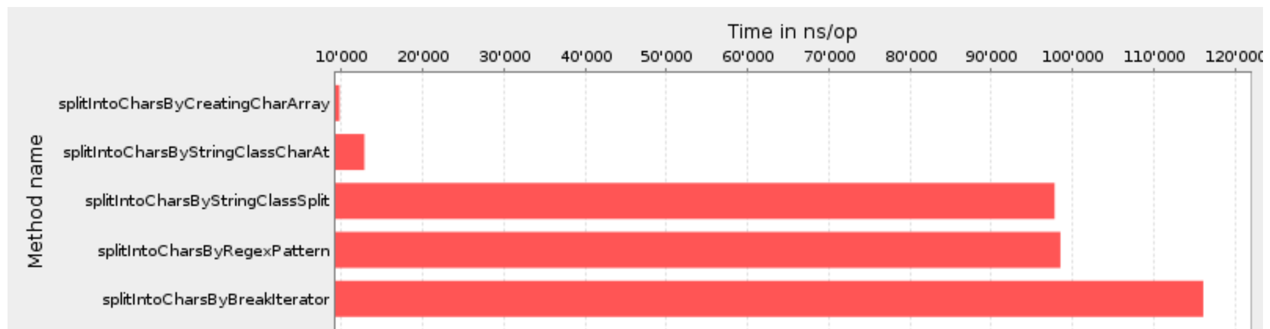

Abbildung 64: Text-In-Zeichen-Trennung-Benchmarks aus der Klasse StringSplittingAndComparingTest

Der Benchmark splitIntoCharsByCreatingCharArray() ist der Schnellste. Ein bisschen langsamer ist splitIntoCharsByStringClassCharAt(). Alle anderen Benchmarks schneiden bedeutend schlechter ab.

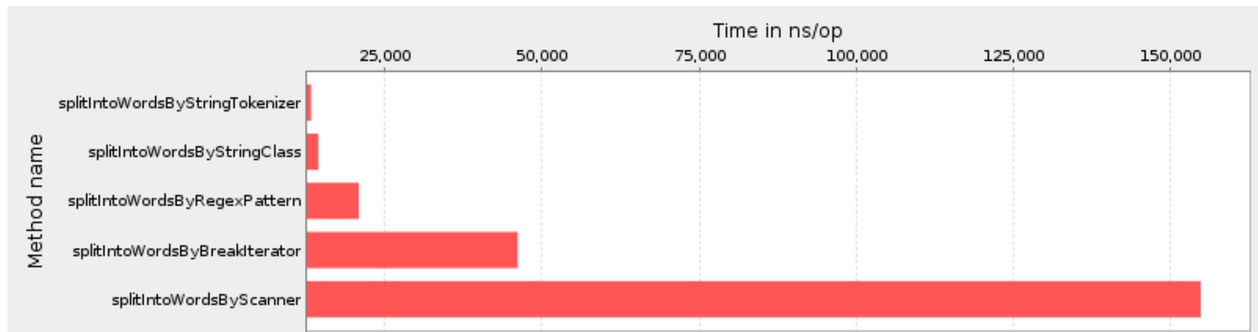

Abbildung 65: Text-In-Wörter-Trennung-Benchmarks aus der Klasse StringSplittingAndComparingTest

Der Benchmark splitIntoWordsByStringTokenizer() schneidet am besten ab. An zweiter Stelle kommt der Benchmark splitIntoWordsByStringClass(). Der Benchmark, der ein Pattern-Objekt nutzt, schneidet schlechter ab als splitIntoWordsByStringClass() und die restlichen Benchmarks schneiden sehr schlecht ab.

Auswertung/Empfehlung

Die Methoden toCharArray() oder charAt(...) in Kombination mit einer Schleife sind performante Verfahren, wie Zeichen aus einem String gewonnen werden können. Sie sollten deshalb bevorzugt werden. Die anderen Verfahren sollten vermieden werden und nur bei Aufgaben eingesetzt werden, bei denen ein String nach einem schwierigeren Kriterium getrennt werden soll.

Um Wörtern zu trennen, sollte ein StringTokenizer bevorzugt werden. Die Methode split(...) der String-Klasse bietet eine gute Performance. Die anderen Trennmöglichkeiten sollten vermieden werden, da sie eine beträchtlich schlechtere Performance bieten.

4.22 Summe aus den Werten eines Arrays oder einer Liste bilden

Thema/Fragestellung

Ein int-Array oder eine ArrayList aus Integern ist gegeben und die Summe aus allen Zahlen muss gebildet werden. Dies kann auf klassischer Weise mit einer Schleife geschehen. Dabei wird eine Variable aufsummiert. Alternativ kann ab Java 8 die Stream API verwendet werden, um einen Stream zu erzeugen und mit sum() als letzte Operation der Stream-Verarbeitung die Summe zu bilden. Statt sum() einzusetzen, kann auch reduce() auf den Stream angewendet werden. Die Methode reduce(...) muss ein Startwert, der beim Aufsummieren 0 entspricht, kennen und eine “Akkumulator-Methode” bekommen. Mit Streams kann auch parallel aufsummiert werden.

Vermutung/Hypothese

Die Hypothese ist, dass beim Array und bei der ArrayList das Aufsummieren mit einer klassischen for-Schleife am schnellsten ist. Die Verarbeitung mit parallelen Streams könnte jedoch ebenfalls sehr performant sein.

Programmcode

```
private static final int REPS =  References.BENCHMARK.REPETITIONS /
                                 References.BENCHMARK.DIVISOR_REPETITIONS;
private int[] array = new int[REPS];
private List<Integer> list;

@Setup(Level.Trial)
public void setUpArray() {
        for (int i = 0; i < REPS; i++) {
                array[i] = (int) (Math.random() * References.BENCHMARK.RANGE);
        }
        list = new ArrayList<Integer>();
        for (int i = 0; i < REPS; i++) {
                list.add(array[i]);
        }
}

@Benchmark
public int sumArrayInClassicWay() {
        int sum = 0;
        for (int i = 0; i < array.length; i++) {
                sum += array[i];
        }
        return sum;
}

@Benchmark
public int sumListInClassicWay() {
        int sum = 0;
        for (int i = 0; i < list.size(); i++) {
                sum += list.get(i);
        }
        return sum;
}

@Benchmark
public int sumArrayWithStream() {
        return Arrays.stream(array).sum();
}

@Benchmark
public int sumListWithStream() {
        return list.stream().mapToInt(i -> i).sum();
}

@Benchmark
public int sumArrayWithParallelStream() {
        return Arrays.stream(array).parallel().sum();
}

@Benchmark
public int sumListWithParallelStream() {
        return list.parallelStream().mapToInt(i -> i).sum();
}

@Benchmark
public int sumArrayWithStream2() {
        return Arrays.stream(array).reduce(
                        References.BENCHMARK.CONST_START_VAL_SUM, Integer::sum);
}

@Benchmark
public int sumListWithStream2() {
        return list.stream().reduce(References.BENCHMARK.CONST_START_VAL_SUM,
```

```
                                Integer::sum);
        }

        @Benchmark
        public int sumArrayWithParallelStream2() {
                return Arrays.stream(array).parallel().reduce(
                        References.BENCHMARK.CONST_START_VAL_SUM, Integer::sum);
        }

        @Benchmark
        public int sumListWithParallelStream2() {
                return list.parallelStream().reduce(
                        References.BENCHMARK.CONST_START_VAL_SUM, Integer::sum);
        }
```

Diagramm und Beobachtung

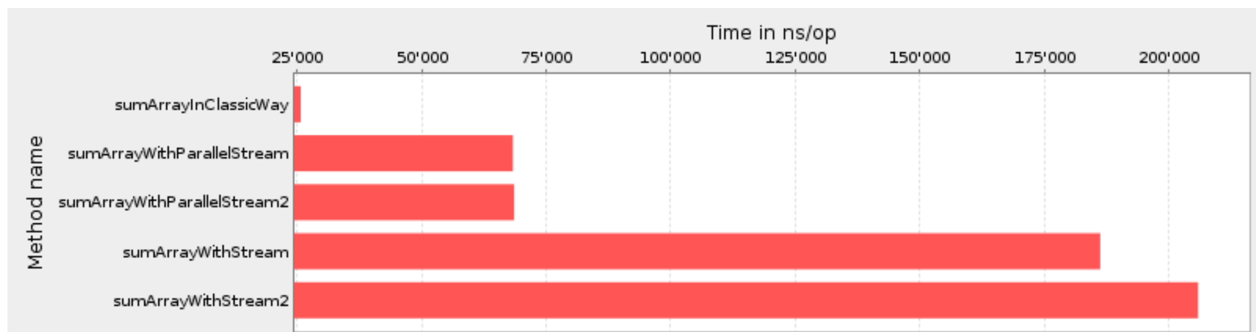

Abbildung 66: Laufzeiten der Array-Benchmarks aus der Klasse SumArraysAndArrayListsTest

Der Benchmark sumArrayInClassicWay(), der eine for-Schleife nutzt, um die Summe zu bilden, ist am schnellsten. Der Benchmark sumArrayWithParallelStream() ist schneller als sumArrayWithParallelStream2(), der reduce() einsetzt, um die Summe zu bilden. Der letztere Benchmark setzt sum() ein, um aus einem IntStream die Summe zu berechnen. Die Benchmarks, die "normale" Streams verwenden, sind deutlich langsamer.

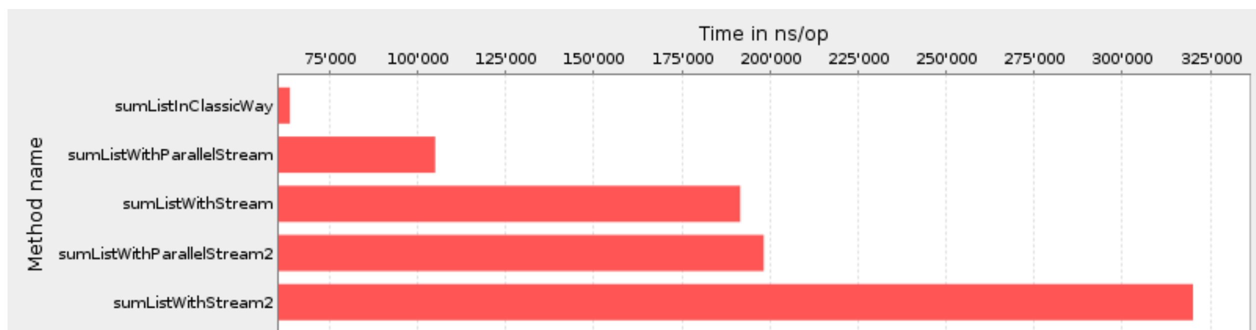

Abbildung 67: Laufzeiten der ArrayList-Benchmarks aus der Klasse SumArraysAndArrayListsTest

Bei der ArrayList ist die klassische Summenbildung mit einer for-Schleife ebenfalls der schnellste Benchmark. Alle Stream-Benchmarks, die mit sum() die Summe bilden, sind schneller als diejenigen, welche die Summe mit reduce() bilden.

Auswertung/Empfehlung

Die Summenbildung mit der for-Schleife ist sowohl bei der ArrayList wie auch beim Array am performantesten. Wenn bei der Summenbildung mit Arrays unbedingt ein Stream nötig sein sollte, so sollte bei modernen Rechnern mit Multicore-Prozessoren ein paralleler Stream eingesetzt werden. Dies gilt auch, wenn aus einer ArrayList die Summe gebildet werden soll. Bei einer Liste in Kombination mit einem parallelen Stream ist sum() effizienter als reduce().

4.23 Switch-Case-Anweisung gegenüber If-Else-Kette

Thema/Fragestellung

Da im Internet oft erwähnt wird, dass eine Switch-Case-Anweisung schneller als eine If-Else-Kette ist (Stack Overflow 2011), wird getestet, ob dies der Realität entspricht oder nur ein Mythos ist. Damit in diesem Experiment eine höhere Laufzeit der Benchmarks erreicht wird und damit eine bessere Aussagekräftigkeit erzielt wird, wird mit einer for-Schleife über die If-Else-Kette oder die Switch-Case-Anweisung iteriert. Dabei wird der Code jedes Cases bzw. jeder Bedingung ausgeführt.

Vermutung/Hypothese

Die Switch-Case-Anweisung ist vermutlich schneller als die If-Else-Kette.

Programmcode

```
private long dummy = 0L;
private int number = 1;

@Benchmark
public long switchCases() {
	dummy = 0;
	for (number = 1; number < 21; number++) {
		switch (number) {
		case 1:
			dummy += number;
			break;
		case 2:
			dummy += number;
			break;

		//[...]

		case 20:
			dummy += number;
			break;
		}
	}
	return dummy;
}
```

```
@Benchmark
public long ifElseChain() {
        dummy = 0;
        for (number = 1; number < 21; number++) {
                if (number == 1) {
                        dummy += number;
                } else if (number == 2) {
                        dummy += number;
                }
                //[...]
                } else if (number == 20) {
                        dummy += number;
                }
        }
        return dummy;
}
```

Diagramm und Beobachtung

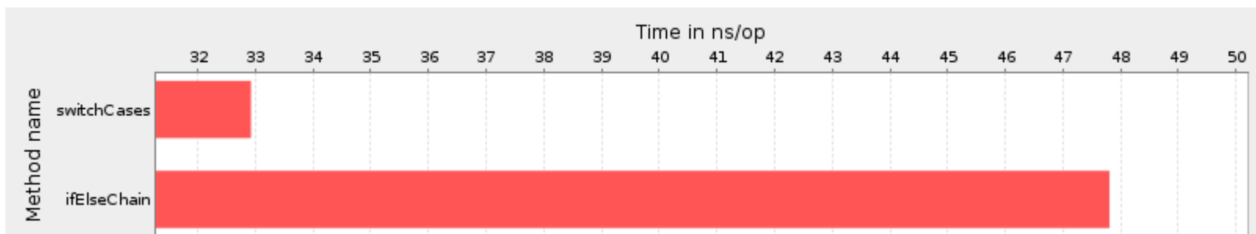

Abbildung 68: Laufzeiten der Benchmarks aus der Klasse SwitchCaseAndIfElseChainTest

Die Switch-Case-Anweisung ist schneller als die If-Else-Kette. Der Unterschied beträgt etwa 15 Nanosekunden. Da die Messzeit der Switch-Case-Anweisung circa 33 Nanosekunden und die Messzeit der If-Else-Kette circa 48 Nanosekunden betragen, ist der relative Unterschied nicht unbedeutend.

Auswertung/Empfehlung

Wenn mehrere Bedingungen geprüft werden müssen und nur eine zutreffen darf, ist eine Switch-Case-Anweisung einer If-Else-Kette zu bevorzugen. Dies liegt daran, dass eine Switch-Case-Anweisung in einen effizienteren Maschinencode umgesetzt werden kann.

4.24 Die Switch-Case-Anweisung und die Datentypen der Testfälle

Thema/Fragestellung

Bei der Switch-Case-Anweisung kann bei der switch-Anweisung eine Variable angegeben werden, die ein Char, ein Enum, ein Byte, ein Short, ein Int oder ein String sein kann (Oracle 2016 F). Die Frage ist nun, ob die Performance vom Datentyp der auszuwertenden Variable abhängig ist. Bei Switch-Case-Anweisungen

mit Zahlen werden die Testwerte 1 bis 26 gewählt. Bei Switch-Case-Anweisungen mit Enums, Chars und Strings werden die Buchstaben A bis Z gewählt (bei Chars und Strings sind es Kleinbuchstaben).

Vermutung/Hypothese

Die Switch-Case-Anweisung mit einem Byte oder Char in der Switch-Instruktion ist vermutlich am schnellsten, da diese Datentypen am wenigsten Speicher benötigen.

Programmcode

```
private int valueToFeedBlackhole = 0;

private String[] strings = { "a", "b", "c", /* ... */, "x", "y", "z" };
private static enum LETTERS { A, B, C, /* ... */, X, Y, Z }
private LETTERS[] lettersAsEnums = { LETTERS.A, LETTERS.B, LETTERS.C,
                                     /* ... */, LETTERS.Y, LETTERS.Z };

private int[] ints = { 1, 2, 3, /* ... */, 24, 25, 26 };
private byte[] bytes = { 1, 2, 3, /* ... */, 24, 25, 26 };
private short[] shorts = { 1, 2, 3, /* ... */, 24, 25, 26 };
private char[] lettersAsChars = { 'a', 'b', 'c', /* ... */, 'x', 'y', 'z' };

@Benchmark
public int switchString() {
	valueToFeedBlackhole = 0;

	for (int j = 0; j < 26; j++) {
		switch (strings[j]) {
		case "a":
			valueToFeedBlackhole += 1;
			break;
		case "b":
			valueToFeedBlackhole += 2;
			break;

		//[...]

		case "y":
			valueToFeedBlackhole += 25;
			break;
		case "z":
			valueToFeedBlackhole += 26;
			break;
		default:
			valueToFeedBlackhole += 0;
			break;
		}
	}
	return valueToFeedBlackhole;
}


@Benchmark
public int switchEnum() {
	valueToFeedBlackhole = 0;

	for (int j = 0; j < 26; j++) {
		switch (lettersAsEnums[j]) {
		case A:
			valueToFeedBlackhole += 1;
			break;
```

```
				//[...]

				case Z:
					valueToFeedBlackhole += 26;
					break;
				default:
					valueToFeedBlackhole += 0;
					break;
				}
			}
			return valueToFeedBlackhole;
	}

	@Benchmark
	public int switchInt() {
			valueToFeedBlackhole = 0;

			for (int j = 0; j < 26; j++) {
				switch (ints[j]) {
				case 1:
					valueToFeedBlackhole += 1;
					break;
				case 2:
					valueToFeedBlackhole += 2;
					break;
				//[...]
				case 25:
					valueToFeedBlackhole += 25;
					break;
				case 26:
					valueToFeedBlackhole += 26;
					break;
				default:
					valueToFeedBlackhole += 0;
					break;
				}
			}
			return valueToFeedBlackhole;
	}

	@Benchmark
	public int switchByte() {
			valueToFeedBlackhole = 0;

			for (int j = 0; j < 26; j++) {
				switch (bytes[j]) {
				case 1:
					valueToFeedBlackhole += 1;
					break;
				//[...]
				case 26:
					valueToFeedBlackhole += 26;
					break;
				default:
					valueToFeedBlackhole += 0;
					break;
				}
			}
			return valueToFeedBlackhole;
	}


	@Benchmark
	public int switchShort() {
			valueToFeedBlackhole = 0;
```

```
            for (int j = 0; j < 26; j++) {
                switch (shorts[j]) {
                case 1:
                    valueToFeedBlackhole += 1;
                    break;
                //[...]
                case 26:
                    valueToFeedBlackhole += 26;
                    break;
                default:
                    valueToFeedBlackhole += 0;
                    break;
                }
            }
            return valueToFeedBlackhole;
        }

        @Benchmark
        public int switchChar() {
            valueToFeedBlackhole = 0;

            for (int j = 0; j < 26; j++) {
                switch (lettersAsChars[j]) {
                case 'a':
                    valueToFeedBlackhole += 1;
                    break;
                //[...]
                case 'z':
                    valueToFeedBlackhole += 26;
                    break;
                default:
                    valueToFeedBlackhole += 0;
                    break;
                }
            }
            return valueToFeedBlackhole;
        }
```

Diagramm und Beobachtung

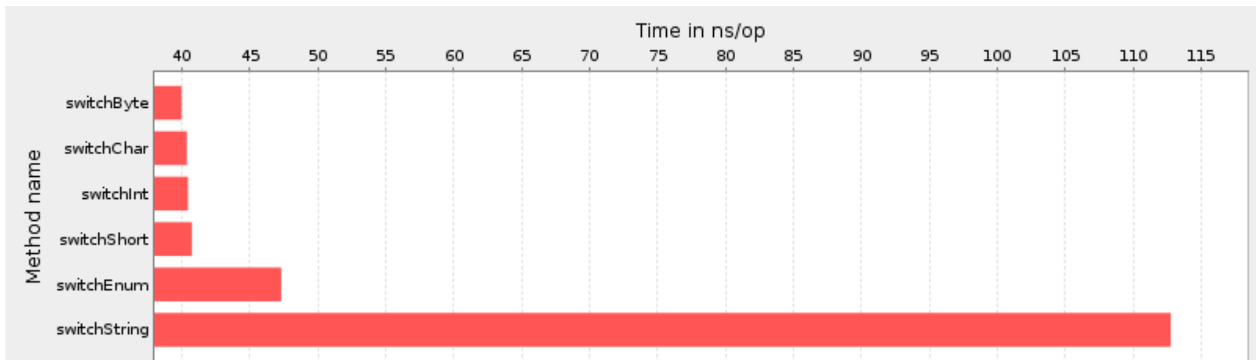

Abbildung 69: Laufzeiten der Benchmarks aus der Klasse SwitchStatementsTest

Bei dieser Grafik fällt auf, dass switchByte(), switchChar(), switchInt() und switchShort() alle in etwa gleich schnell sind. Der Benchmark switchString() ist der langsamste von allen gezeigten Benchmarks.

Auswertung/Empfehlung

Obwohl seit Java 7 in der Switch-Anweisung ein String erlaubt ist, sollte auf diese Möglichkeit verzichtet werden, falls die Performance eine wichtige Anforderung ist, die an die Applikation gestellt wird. Wenn der String bei den Cases nur aus einem Zeichen besteht, sollten die String-Literale durch Char-Literale ersetzt werden.

4.25 Synchronisationsarten im Vergleich

Thema/Fragestellung

Bei diesem Experiment werden verschiedenen Implementierungen geprüft, die die Synchronisation von Variablen über mehrere Threads erlauben. Konkret werden folgende Synchronisationsarten verglichen: ReentrantLock, Semaphore, synchronized-Block und Atomic-Typ. In jedem Benchmark werden mehrere Variablen gesetzt und danach abgefragt. Die abgefragten Werte werden so konsumiert, dass der JIT-Compiler diese Anweisung nicht wegoptimieren kann. Welche Synchronisationsart ist am effektivsten?

Vermutung/Hypothese

Es ist sehr schwierig einzuschätzen, welche Synchronisationsart am effizientesten ist.

Programmcode

```
private Semaphore sem;
private ReentrantLock reentrantLock;

private AtomicBoolean atomicBool;
private AtomicInteger atomicInteger;
private AtomicLong atomicLong;

private Boolean booleanVal;
private Integer integerVal;
private Long longVal;

Blackhole b = new Blackhole();

@Setup(Level.Trial)
public void setUp() {
       resetVariables();

       sem = new Semaphore(1);
       reentrantLock = new ReentrantLock();
}

@Benchmark
public void useAtomicTypes() {

       atomicBool.set(true);
       atomicInteger.set(10);
       atomicLong.set(10L);
```

```
        b.consume(atomicBool.get());
        b.consume(atomicInteger.get());
        b.consume(atomicLong.get());

        resetVariables();
    }

    @Benchmark
    public void useSemaphore() throws InterruptedException {

        sem.acquire();
        booleanVal = true;
        b.consume(booleanVal);
        sem.release();

        sem.acquire();
        integerVal = 10;
        b.consume(integerVal);
        sem.release();

        sem.acquire();
        longVal = 10L;
        b.consume(longVal);
        sem.release();

        resetVariables();
    }

    @Benchmark
    public void useReentrantLock() {

        reentrantLock.lock();
        booleanVal = true;
        b.consume(booleanVal);
        reentrantLock.unlock();

        reentrantLock.lock();
        integerVal = 10;
        b.consume(integerVal);
        reentrantLock.unlock();

        reentrantLock.lock();
        longVal = 10L;
        b.consume(longVal);
        reentrantLock.unlock();

        resetVariables();
    }

    @Benchmark
    public void useSynchronizedBlock() {

        synchronized (booleanVal) {
            booleanVal = true;
            b.consume(booleanVal);
        }

        synchronized (integerVal) {
            integerVal = 10;
            b.consume(integerVal);
        }

        synchronized (longVal) {
            longVal = 10L;
            b.consume(longVal);
        }
```

```
            resetVariables();

    }

    private void resetVariables() {
            atomicBool = new AtomicBoolean();
            atomicInteger = new AtomicInteger();
            atomicLong = new AtomicLong();

            booleanVal = false;
            integerVal = 0;
            longVal = 0L;
    }
```

Diagramm und Beobachtung

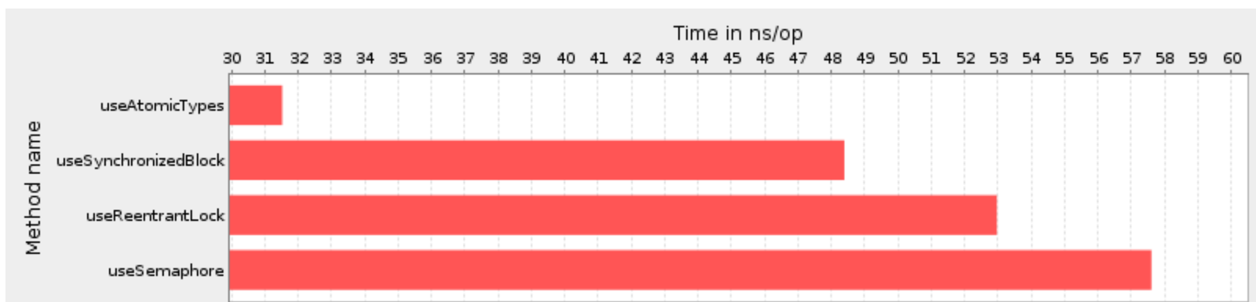

Abbildung 70: Laufzeiten der Benchmarks aus der Klasse SynchronizedAccessTest

Die Synchronisation mit einer Semaphore ist am teuersten. Die Nutzung von atomaren Typen ist die schnellste Synchronisationsart im Vergleich zu den anderen.

Auswertung/Empfehlung

Wenn der Zugriff auf eine einzelne Variable synchronisiert werden muss, ist der Einsatz von Atomic-Typen für eine hohe Performance die beste Wahl. Die anderen Programmierkonstrukte sollten bei der Synchronisation von mehreren und komplizierteren Instruktionen gewählt werden.

4.26 Operationen auf den Klassen ArrayList und LinkedList

Thema/Fragestellung

In Java bietet das List-Interface die Möglichkeit, eine Liste als Datenstruktur umzusetzen. Zwei bekannte Implementierungen, die es in Java schon gibt, sind ArrayList und LinkedList. Welche der beiden Implementierungen ist am schnellsten?

Vermutung/Hypothese

Die ArrayList ist sehr wahrscheinlich schneller als die LinkedList, weil die ArrayList in einen effizienteren Maschinencode umgesetzt wird.

Programmcode

Anmerkung: Der folgende Test wurde zwei Mal implementiert. Es wurde einerseits die Klasse ListArrayListTest, andererseits die Klasse ListLinkedListTest umgesetzt. Es wird nachfolgend jedoch nur die Klasse ListArrayListTest aufgeführt. Die Klasse ListLinkedListTest unterscheidet sich nämlich nur in dem Punkt, dass bei der Initialisierung (mit rotem Rahmen gekennzeichnet) an Stelle einer ArrayList eine LinkedList initialisiert wird. Die Klasse ListOperations ermöglicht das Verhindern von Code-Duplizierung.

```
@State(Scope.Thread)
public class ListsArrayListTest {

	private ListOperations listOps = new ListOperations(new ArrayList<>(),
	                                                     new ArrayList<>());
	private List<Integer> list;
	private static final int REPS = References.BENCHMARK.LISTREPETITIONS / 100;

	public ListsArrayListTest() {

		list = new ArrayList<>();
		for (int i = 0; i < REPS; i++) {
			list.add(i);
		}
	}

	// basic runtime
	@Benchmark
	public void basicRuntime() {
		listOps.setUpLists();
	}

	@Benchmark
	public List<Integer> addToList() {
		listOps.setUpLists();

		for (int i = 0; i < REPS; i++) {
			listOps.addTo(i);
		}

		return listOps.getAddList();
	}
```

```
@Benchmark
public List<Integer> addAtBeginningList() {
        listOps.setUpLists();

        int dummy = 42;
        for (int i = 0; i < REPS; i++) {
                listOps.addAtIndex(0, dummy);
        }

        return listOps.getAddList();
}

@Benchmark
public List<Integer> addAtMiddleList() {
        listOps.setUpLists();

        int insert = 42;
        int size = 0;
        for (int i = 0; i < REPS; i++) {
                size = listOps.addAtIndex(((int) (size / 2)), insert);
        }

        return listOps.getAddList();
}

@Benchmark
public List<Integer> addAtEndList() {
        listOps.setUpLists();

        int insert = 42;
        int size = 1;
        for (int i = 0; i < REPS; i++) {
                size = listOps.addAtIndex(size - 1, insert);
        }
        return listOps.getAddList();
}

@Benchmark
public List<Integer> addAllList() {
        listOps.setUpLists();

        for (int i = 0; i < list.size(); i++) {
                listOps.addAll(list);
        }

        return listOps.getAddList();
}

@Benchmark
public List<Integer> addAllAtBeginningList() {
        listOps.setUpLists();

        for (int i = 0; i < list.size(); i++) {
                listOps.addAllAtIndex(0, list);
        }

        return listOps.getAddList();
}

@Benchmark
public List<Integer> addAllAtMiddleList() {
        listOps.setUpLists();

        for (int i = 0; i < list.size(); i++) {
                listOps.addAllAtIndex((int) (list.size() * i / 2), list);
        }
```

```
                return listOps.getAddList();
        }

        @Benchmark
        public List<Integer> addAllAtEndList() {
                listOps.setUpLists();

                int size = 0;
                for (int i = 0; i < list.size(); i++) {
                        if (i > 0) {
                                size = list.size() * i - 1;
                        }

                        listOps.addAllAtIndex(size, list);
                }
                return listOps.getAddList();
        }

        @Benchmark
        public long forEachLoopList() {
                listOps.setUpLists();
                return listOps.forEachLoop();
        }

        @Benchmark
        public long forLoopList() {
                listOps.setUpLists();
                return listOps.forLoop();
        }

        @Benchmark
        public long functionalLoopList() {
                listOps.setUpLists();
                return listOps.functionalLoop();
        }

        @Benchmark
        public long functionalParallelLoopList() {
                listOps.setUpLists();
                return listOps.functionalParallelLoop();
        }

        @Benchmark
        public long iterateList() {
                listOps.setUpLists();
                return listOps.iterate();
        }

        @Benchmark
        public long iterateWithListIterator() {
                listOps.setUpLists();
                return listOps.iterateWithListIterator();
        }

        @Benchmark
        public long whileLoopList() {
                listOps.setUpLists();
                return listOps.whileLoop();
        }
}

public class ListOperations {

        private List<Integer> addList;
        private List<Integer> loopList;
        public ListOperations(List<Integer> list1, List<Integer> list2) {
```

```
            addList = list1;
            loopList = list2;

            setUpLists();
    }

    public final void setUpLists() {

            addList.clear();
            loopList.clear();

            for (int i = 0; i < References.BENCHMARK.LISTREPETITIONS; i++) {
                    loopList.add(i);
            }
    }

    public boolean addTo(int dummy) {
            return addList.add(dummy);
    }

    public int addAtIndex(int index, int dummy) {
            addList.add(index, dummy);
            return addList.size();
    }

    public boolean addAll(List<Integer> dummyList) {
            return addList.addAll(dummyList);
    }

    public boolean addAllAtIndex(int index, List<Integer> dummyList) {
            return addList.addAll(index, dummyList);
    }

    public long forEachLoop() {
            long dummy = 0;
            for (Integer list1 : loopList) {
                    dummy += list1;
            }
            return dummy;
    }

    public long forLoop() {
            long dummy = 0;
            for (int i = 0; i < loopList.size(); i++) {
                    dummy += loopList.get(i);
            }
            return dummy;
    }

    public long functionalLoop() {
            long dummy = 0;
            dummy = loopList.stream().map((list1) -> list1)
                            .reduce((int) dummy, Integer::sum);
            return dummy;
    }

    public long functionalParallelLoop() {
            long dummy = 0;
            dummy = loopList.parallelStream().map((list1) -> list1)
                            .reduce((int) dummy, Integer::sum);
            return dummy;
    }

    public long iterate() {
            long dummy = 0;
            Iterator<Integer> iterator = loopList.iterator();
```

```
            while (iterator.hasNext()) {
                dummy += iterator.next();
            }

            return dummy;
        }

        public long whileLoop() {
            long dummy = 0;
            int i = 0;

            while (i < loopList.size()) {
                Integer element = loopList.get(i);
                dummy += element;
                i++;
            }

            return dummy;
        }

        public long iterateWithListIterator() {
            long dummy = 0;
            ListIterator<Integer> listIterator = loopList.listIterator();

            while (listIterator.hasNext()) {
                dummy += listIterator.next();
            }

            return dummy;
        }

        public List<Integer> getAddList() {
            return addList;
        }
}
```

Diagramm und Beobachtung

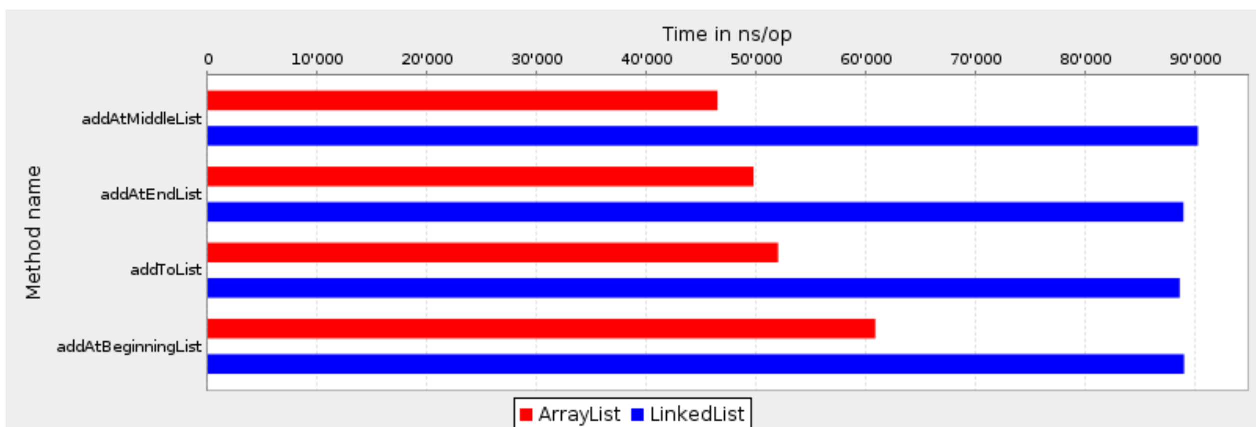

Abbildung 71: Add-Benchmarks aus den Klassen ListArrayListTest und ListLinkedListTest

Wenn ein einzelnes Element in einer Liste eingefügt wird, erreicht die ArrayList bei allen durchgeführten Benchmarks mit Abstand die höhere Performance als die LinkedList.

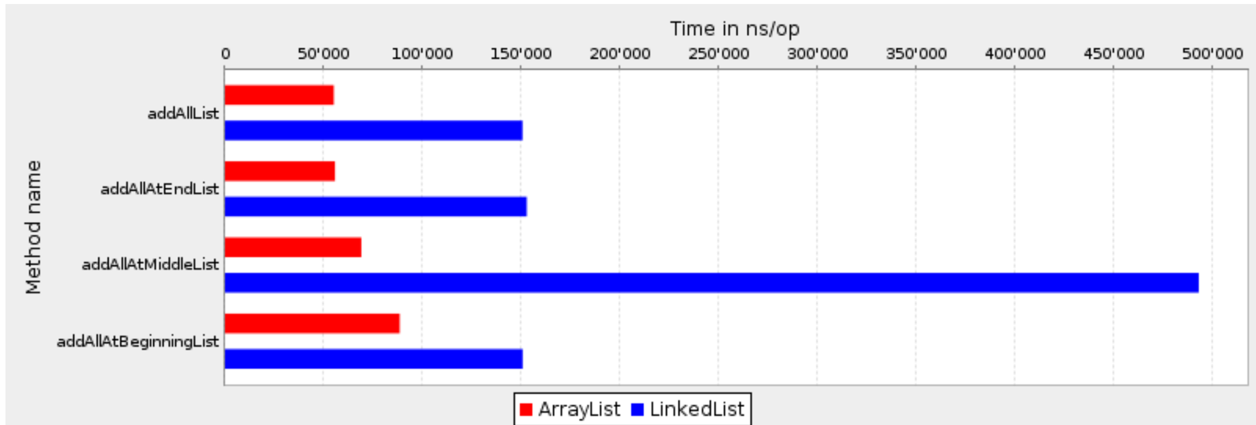

Abbildung 72: AddAll-Benchmarks aus den Klassen ListArrayListTest und ListLinkedListTest

Die ArrayList erreicht auch in diesem Fall eine höhere Performance als die LinkedList. Hier wird eine Sammlung von Elementen in der Liste eingefügt. Was bei dieser Grafik auffällt, ist, dass das Einfügen einer Sammlung von Elementen in der Mitte einer LinkedList besonders langsam ist.

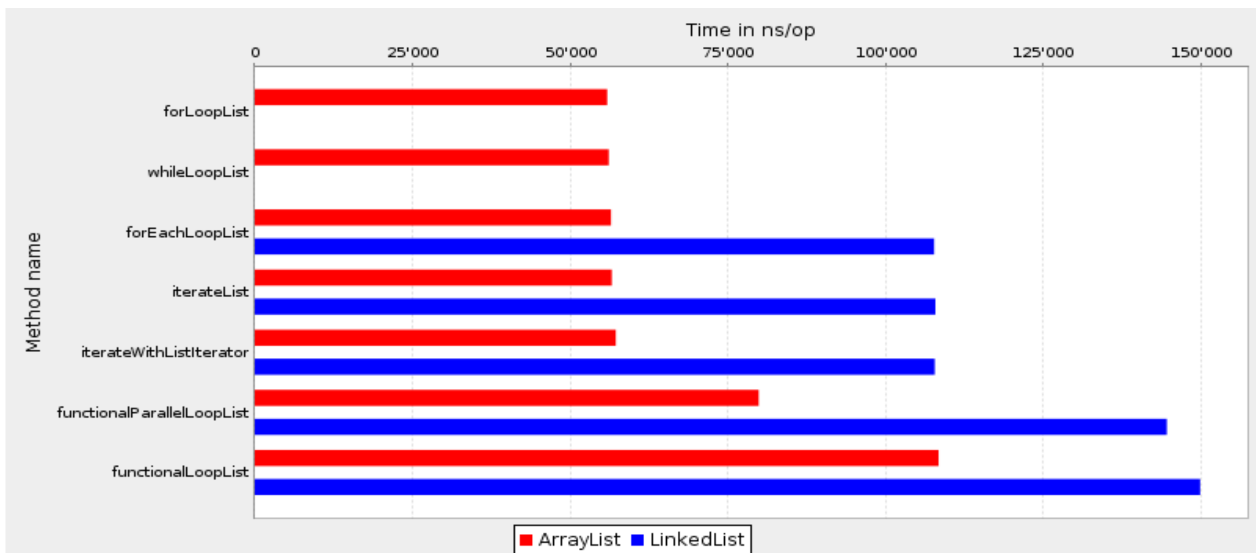

Abbildung 73: Iterations-Benchmarks aus den Klassen ListArrayListTest und ListLinkedListTest

Bei diesen Benchmarks wird verglichen, wie schnell eine ArrayList im Gegensatz zu einer LinkedList ist, wenn auf verschiedene Arten über die Liste iteriert wird und daraus die Summe gebildet wird. Bei dieser Grafik ist zu beachten, dass die Laufzeiten der LinkedList bei forLoopList() und whileLoopList() nicht dargestellt werden, weil diese in dieser Skala nicht hineinpassen. Diese werden erst auf der nächsten Grafik gezeigt. Auf der aktuellen Grafik ist ebenfalls zu sehen, dass die ArrayList eindeutig schneller ist als die LinkedList.

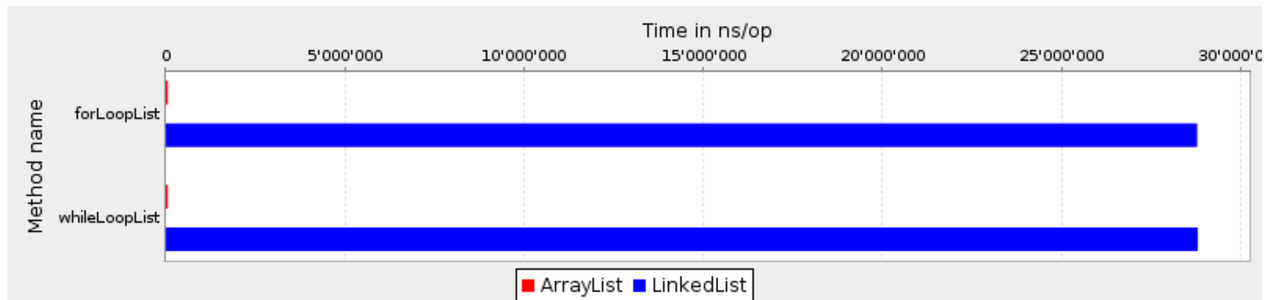

Abbildung 74: Loop-Benchmarks aus den Klassen ListArrayListTest und ListLinkedListTest

Auf dieser Grafik wird der markante Performanceunterschied zwischen den beiden List-Implementierungen bei forLoopList() und whileLoopList() gezeigt. Die ArrayList ist bei beiden dieser Benchmarks mit grossem Abstand die schnellere Liste.

Auswertung/Empfehlung

Wenn einer Liste häufig neue Elemente hinzugefügt werden müssen, ist die ArrayList deutlich schneller und damit die bessere Wahl als die LinkedList. Eine Besonderheit der LinkedList ist, dass das Einfügen eines einzelnen Elementes in der Mitte der Liste besonders langsam ist. Wenn in einer for- oder while-Schleife alle Elemente einer Liste abgefragt werden müssen, ist die ArrayList ebenfalls schneller als die LinkedList. Werden Iteratoren oder funktionale Konstrukte verwendet, ist der Unterschied hingegen weniger ausgeprägt. Über alle Ergebnisse betrachtet, ist die ArrayList performanter als die LinkedList.

4.27 Operationen auf den Klassen HashMap, LinkedHashMap und TreeMap

Thema/Fragestellung

Das Map-Interface in Java bietet die Möglichkeit, Schlüssel-Wert-Paare in einer Sammlung zu speichern. Es werden folgende Map-Implementierungen miteinander verglichen: HashMap, LinkedHashMap und TreeMap. Die HashMap ist eine Implementierung der Map als Hashtabelle. Die LinkedHashMap ist auch eine Hashtabelle, die jedoch als verkette Liste implementiert ist. Damit ist die Iterationsreihenfolge der LinkedHashMap vorhersehbar (Oracle 2016 A). Eine TreeMap ist eine Map, die als Rot-Schwarz-Baum implementiert ist (Oracle 2016 A). Für diverse Operationen werden nachfolgend Map-implementierungen miteinander verglichen und ihre Performance analysiert. Die Schlüssel und Werte aller getesteten Maps sind Integer. Alle Maps werden mit dem Default-Konstruktor instanziiert und danach mit 10’000 Schlüssel-Wert-Paaren befüllt.

Vermutung/Hypothese

Jede Map-Implementierung ist von bestimmten Charakteristika geprägt. Aus diesem Grund ist es unwahrscheinlich, dass ein Map-Typ bei allen Operationen schneller ist als die anderen Map-Typen. Da die HashMap die populärste Map ist, könnte sie bei vielen Operationen die beste Performance erzielen.

Programmcode

Anmerkung: Der folgende Test wurde drei Mal implementiert. Es wurden die Klassen MapsHashMapTest, MapsLinkedHashMapTest und MapsTreeMapTest umgesetzt. Es wird nachfolgend jedoch nur die Klasse MapsHashMapTest aufgeführt. Die Klassen MapsLinkedHashMapTest und MapsTreeMapTest unterscheiden sich nämlich nur in dem Punkt, dass bei der Initialisierung (mit rotem Rahmen gekennzeichnet) an Stelle einer HashMap eine LinkedHashMap beziehungsweise eine TreeMap initialisiert wird. Die Klasse MapOperations ermöglicht das Verhindern von Code-Duplizierung.

```
@State(Scope.Thread)
public class MapsHashMapTest {

	private MapOperations mapOps = new MapOperations(new HashMap<>(),
	                                                  new HashMap<>());

	@Setup(Level.Iteration)
	public void setUpLists() {
		mapOps.initializeMaps();
	}

	@Benchmark
	public void basicRuntime() {
		mapOps.initializeMaps();
	}

	@Benchmark
	public boolean containsKeyStart() {
		mapOps.initializeMaps();

		Integer key = mapOps.getKeys()[0];
		return mapOps.containsKey(key);
	}

	@Benchmark
	public boolean containsKeyMiddle() {
		mapOps.initializeMaps();

		Integer key = mapOps.getKeys()[1];
		return mapOps.containsKey(key);
	}

	@Benchmark
	public boolean containsKeyEnd() {
		mapOps.initializeMaps();

		Integer key = mapOps.getKeys()[2];
		return mapOps.containsKey(key);
	}

	@Benchmark
	public boolean containsValueStart() {
		mapOps.initializeMaps();
```

```
            Integer value = mapOps.getValues()[0];
            return mapOps.containsValue(value);
    }


    @Benchmark
    public boolean containsValueMiddle() {
            mapOps.initializeMaps();

            Integer value = mapOps.getValues()[1];
            return mapOps.containsValue(value);
    }

    @Benchmark
    public boolean containsValueEnd() {
            mapOps.initializeMaps();

            Integer value = mapOps.getValues()[2];
            return mapOps.containsValue(value);
    }

    @Benchmark
    public long getStart() {
            mapOps.initializeMaps();

            Integer key = mapOps.getKeys()[0];
            return mapOps.get(key);
    }

    @Benchmark
    public long getMiddle() {
            mapOps.initializeMaps();

            Integer key = mapOps.getKeys()[1];
            return mapOps.get(key);
    }

    @Benchmark
    public long getEnd() {
            mapOps.initializeMaps();

            Integer key = mapOps.getKeys()[2];
            return mapOps.get(key);
    }

    @Benchmark
    public Map<Integer, Integer> put() {
            mapOps.initializeMaps();

            for (int i = 0; i < References.BENCHMARK.MAPREPETITIONS; i++) {
                    mapOps.put(i, References.BENCHMARK.MAPREPETITIONS - i);
            }
            return mapOps.getEmptyMap();
    }

    @Benchmark
    public Map<Integer, Integer> putIfAbsentAllAbsent() {
            mapOps.initializeMaps();

            for (int i = 0; i < References.BENCHMARK.MAPREPETITIONS; i++) {
                    mapOps.putIfAbsentEmptyMap(i,
                            References.BENCHMARK.MAPREPETITIONS - i);
            }
            return mapOps.getEmptyMap();
    }
```

```
@Benchmark
public Map<Integer, Integer> putIfAbsentAllExisting() {
	mapOps.initializeMaps();

	for (int i = 0; i < References.BENCHMARK.MAPREPETITIONS; i++) {
		mapOps.putIfAbsentFullMap(i,
				References.BENCHMARK.MAPREPETITIONS - i);
	}
	return mapOps.getFullMap();
}

@Benchmark
public Map<Integer, Integer> removeElementEnd() {
	mapOps.initializeMaps();

	Integer key = mapOps.getKeys()[2];
	mapOps.remove(key);
	return mapOps.getFullMap();
}

@Benchmark
public Map<Integer, Integer> removeElementStart() {
	mapOps.initializeMaps();

	Integer key = mapOps.getKeys()[0];
	mapOps.remove(key);
	return mapOps.getFullMap();
}

@Benchmark
public Map<Integer, Integer> clearLoop() {
	mapOps.initializeMaps();

	for (int i = 0; i < References.BENCHMARK.MAPREPETITIONS; i++) {
		mapOps.remove(i);
	}
	return mapOps.getFullMap();
}

@Benchmark
public Map<Integer, Integer> replaceAll() {
	mapOps.initializeMaps();

	int size = mapOps.getFullMap().size();
	for (int i = 0; i < size; i++) {
		mapOps.replace(i, i + 1);
	}
	return mapOps.getFullMap();
}

@Benchmark
public Map<Integer, Integer> replaceFirst() {
	mapOps.initializeMaps();
	int size = mapOps.getFullMap().size();
	Integer key = mapOps.getKeys()[0];
	for (int i = 0; i < size; i++) {
		mapOps.replace(key, i + 1);
	}
	return mapOps.getFullMap();
}


@Benchmark
public Map<Integer, Integer> replaceLast() {
	mapOps.initializeMaps();
	int size = mapOps.getFullMap().size();
```

```
            Integer key = mapOps.getKeys()[2];
            for (int i = 0; i < size; i++) {
                  mapOps.replace(key, i + 1);
            }
            return mapOps.getFullMap();
      }

      @Benchmark
      public Map<Integer, Integer> swapAllValues() {
            mapOps.initializeMaps();

            int size = mapOps.getFullMap().size();
            for (int i = 0; i < size; i++) {
                  mapOps.replace(i, 0, i + 1);
            }
            return mapOps.getFullMap();
      }

      @Benchmark
      public Map<Integer, Integer> replaceAllBiFunction() {
            mapOps.initializeMaps();
            BiFunction<Integer, Integer, Integer> biFunction = (num1, num2) ->
                                                                  (num2 - num1);
            mapOps.replaceAll(biFunction);
            return mapOps.getFullMap();
      }
}

public class MapOperations {
      private Map<Integer, Integer> emptyMap;
      private Map<Integer, Integer> fullMap;
      private Integer[] keys = new Integer[3];
      private Integer[] values = new Integer[3];
      private int reps = References.BENCHMARK.LISTREPETITIONS;

      public MapOperations(Map<Integer, Integer> empty,
                           Map<Integer, Integer> full) {
            emptyMap = empty;
            fullMap = full;

            initializeMaps();
      }

      public final void initializeMaps() {
            emptyMap.clear();
            fullMap.clear();

            int insert = 0;
            int key = 0;

            for (int i = 0; i < reps; i++) {
                  insert = (insert + 97961) % reps; // 97961 is prime
                  key = (key + 13001 % reps); // 13001 is prime
                  fullMap.put(key, insert);

                  // Save first, middle and last value and key
                  if (i == 0) {
                        keys[0] = key;
                        values[0] = insert;
                  } else if (i == (int) (reps / 2)) {
                        keys[1] = key;
                        values[1] = insert;
                  } else if (i == reps - 1) {
                        keys[2] = key;
                        values[2] = insert;
                  }
```

```
			}
		}

		/**
		 * @return first middle and last value of the full map
		 */
		public final Integer[] getValues() { return values; }

		/**
		 * @return first middle and last key of the full map
		 */
		public final Integer[] getKeys() { return keys; }

		public boolean containsKey(Integer key) {
			return fullMap.containsKey(key);
		}

		public boolean containsValue(Integer value) {
			return fullMap.containsValue(value);
		}

		public Integer get(Integer key) {
			return fullMap.get(key);
		}

		public void put(Integer key, Integer value) {
			emptyMap.put(key, value);
		}

		public void putIfAbsentEmptyMap(Integer key, Integer value) {
			emptyMap.putIfAbsent(key, value);
		}

		public void putIfAbsentFullMap(Integer key, Integer value) {
			fullMap.putIfAbsent(key, value);
		}

		public void remove(Integer key) {
			fullMap.remove(key);
		}

		public void clear() {
			fullMap.clear();
		}

		public void replace(Integer key, Integer value) {
			fullMap.replace(key, value);
		}

		public void replace(Integer key, Integer oldValue, Integer newValue) {
			fullMap.replace(key, fullMap.get(key), newValue);
		}

		public void replaceAll(BiFunction<Integer, Integer, Integer> function) {
			fullMap.replaceAll(function);
		}

		public Map<Integer, Integer> getEmptyMap() {
			return emptyMap;
		}

		public Map<Integer, Integer> getFullMap() {
			return fullMap;
		}
}
```

Diagramm und Beobachtung

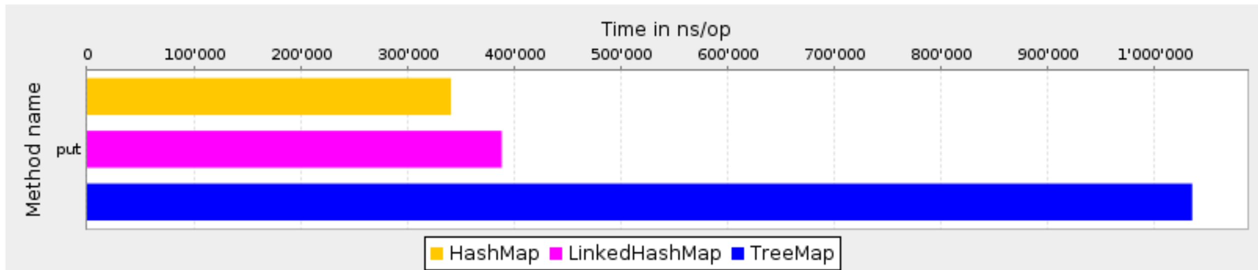

Abbildung 75: Put-Benchmarks aus den Maps-Benchmarkklassen

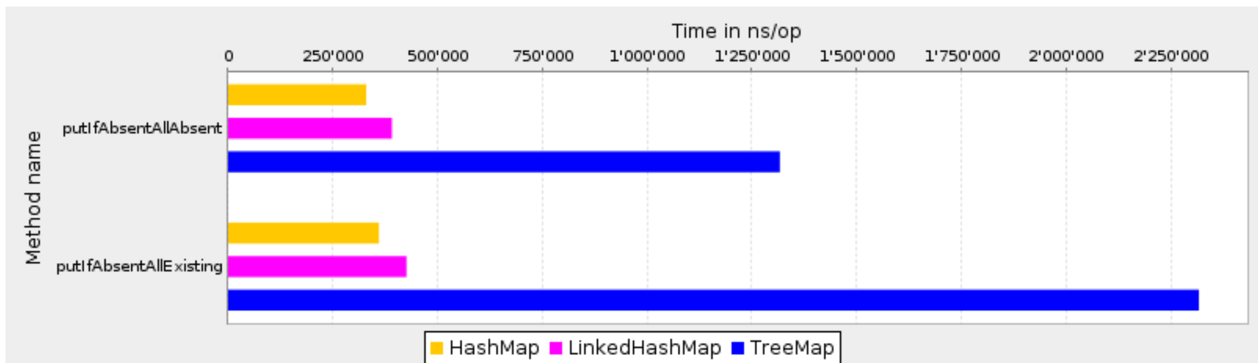

Abbildung 76: Put-Benchmarks aus den Maps-Benchmarkklassen

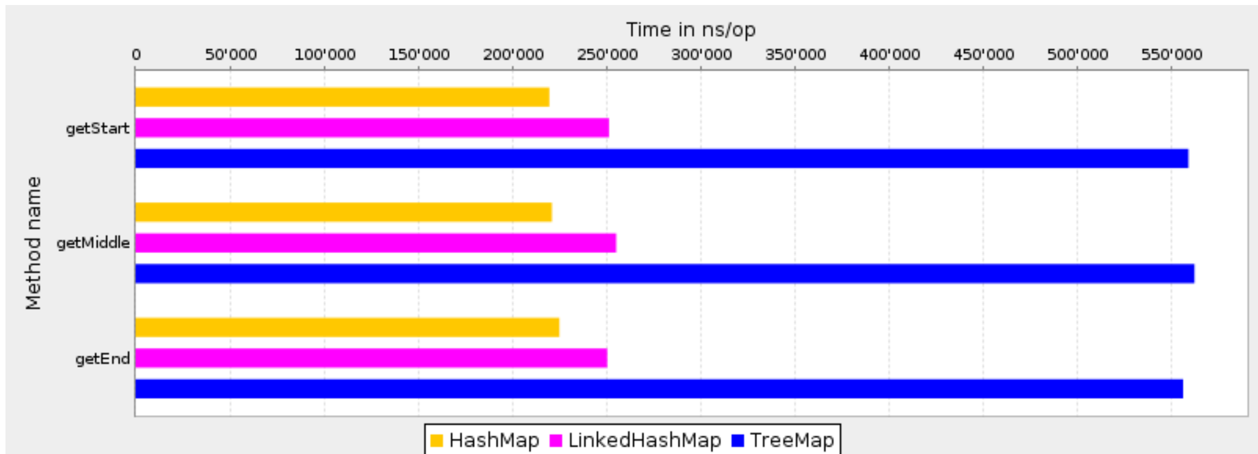

Abbildung 77: Get-Benchmarks aus den Maps-Benchmarkklassen

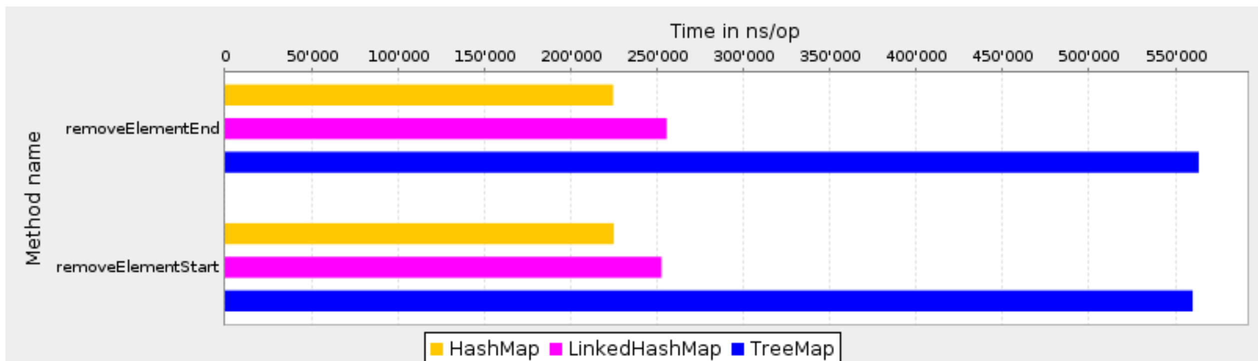

Abbildung 78: Remove-Benchmarks aus den Maps-Benchmarkklassen

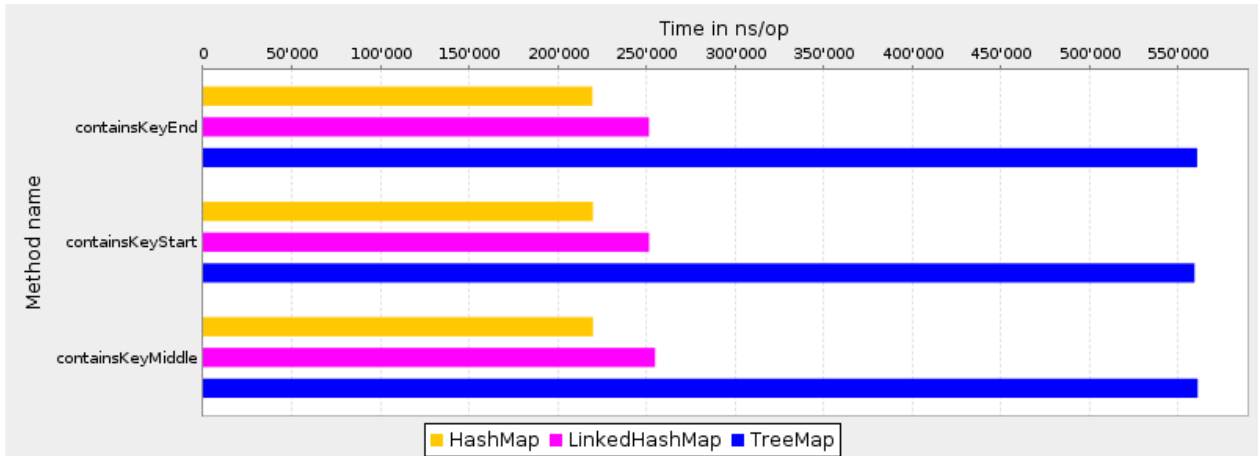

Abbildung 79: ContainsKey-Benchmarks aus den Maps-Benchmarkklassen

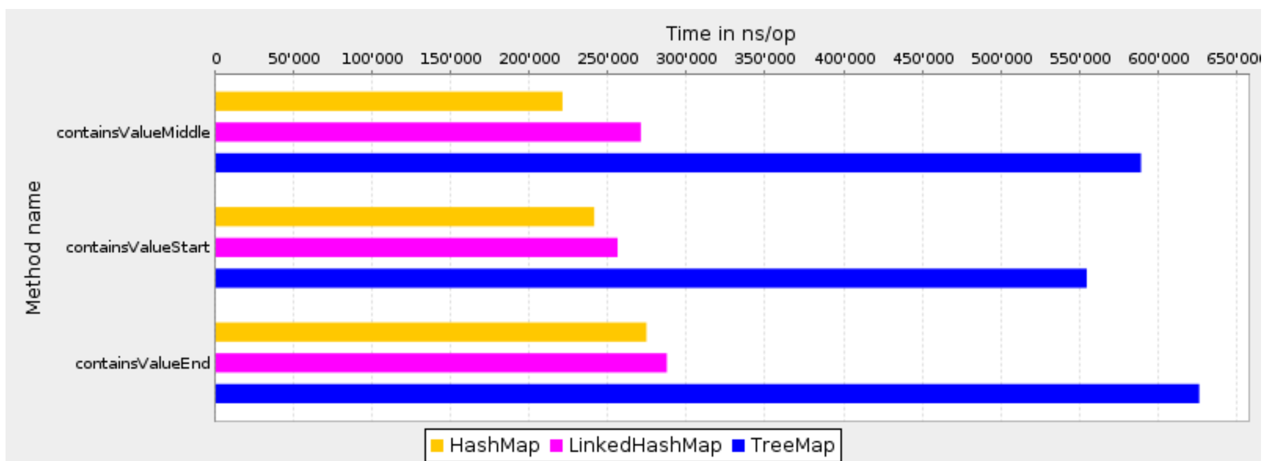

Abbildung 80: ContainsValue-Benchmarks aus den Maps-Benchmarkklassen

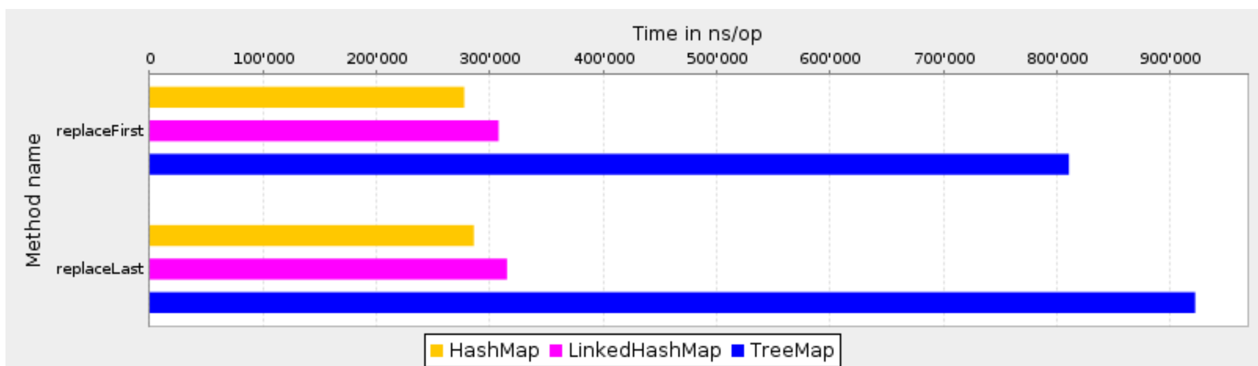

Abbildung 81: Replace-Benchmarks aus den Maps-Benchmarkklassen

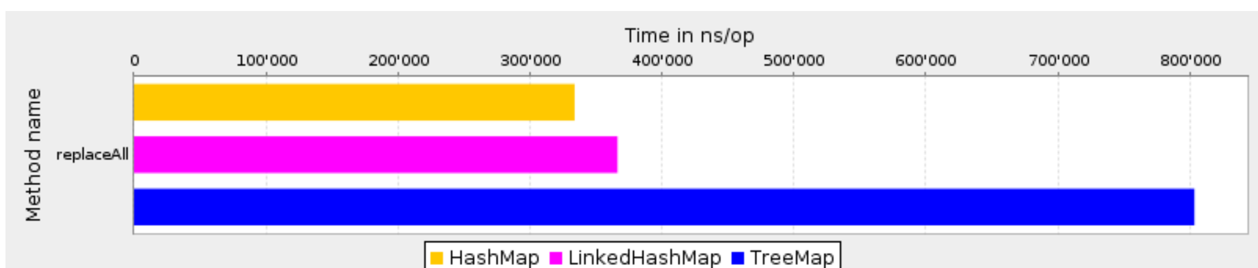

Abbildung 82: ReplaceAll-Benchmarks aus den Maps-Benchmarkklassen

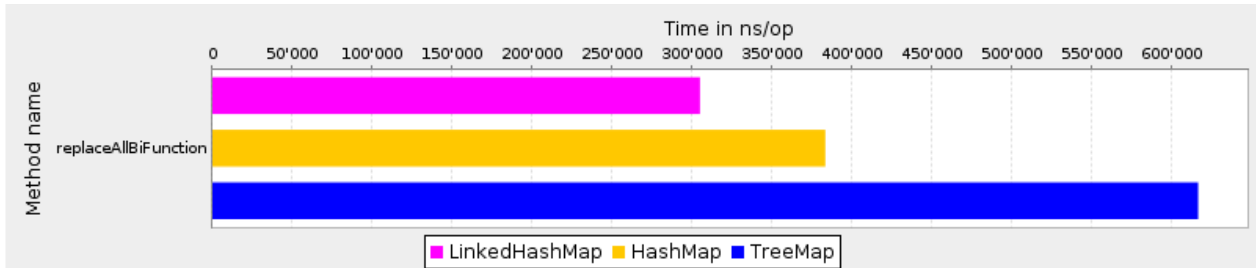

Abbildung 83: Replace-Benchmarks aus den Maps-Benchmarkklassen

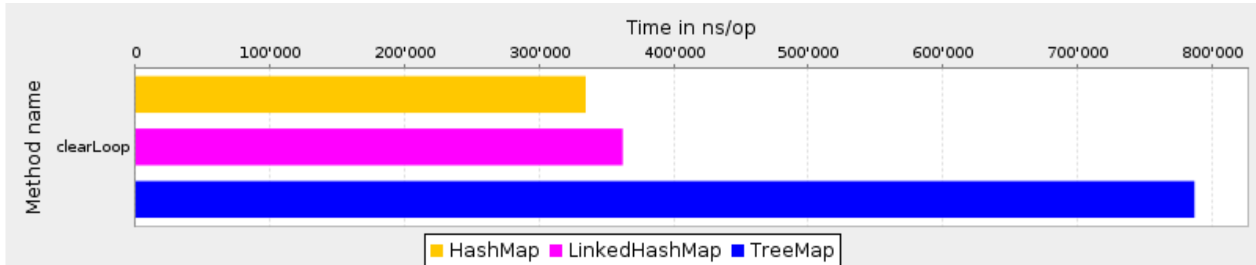

Abbildung 84: Clear-Benchmarks aus den Maps-Benchmarkklassen

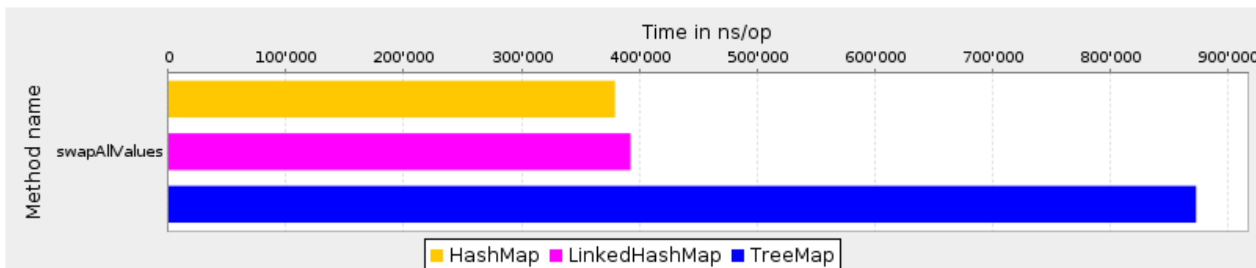

Abbildung 85: SwapAll-Benchmarks aus den Maps-Benchmarkklassen

Bei allen aufgeführten Vergleichen ist die HashMap am schnellsten, ausser bei dem Benchmark “replaceAllBiFunction“. Beim Test “swapAllValues“ ist die LinkedHashMap jedoch sehr wenig langsamer als die HashMap. Die TreeMap hat im Vergleich zu den anderen Maps immer eine deutlich höhere Laufzeit.

Auswertung/Empfehlung

Die dargestellten Tests zeigen, dass die HashMap schneller als die LinkedHashMap und die TreeMap ist, wenn es darum geht, nach einem Schlüssel oder einem Wert zu suchen, Werte zu ersetzen, Werte abzufragen oder neue Werte zu setzen, bestimmte Schlüssel-Wert-Paare zu löschen oder die ganze Map zu leeren. Deswegen wird die HashMap empfohlen. Die letzte Grafik ist ein Spezialfall, bei dem die Hashmap und die LinkedHashMap in etwa gleich schnell sind. Die TreeMap ist bei jeder dargestellten Operation eindeutig die langsamste Map-Implementierung. Es wird deswegen von der Verwendung der TreeMap abgeraten.

4.28 Operationen auf den Klassen HashSet, LinkedHashSet und TreeSet

Thema/Fragestellung

Das Set-Interface bietet verschiedene Implementierungen an, die verwendet werden können. Darunter sind beliebte Implementierungen wie das HashSet, das LinkedHashSet und das TreeSet. Doch welches Set bietet die beste Performance?

Vermutung/Hypothese

Da das HashSet die wahrscheinlich bekannteste Implementierung ist, könnte es gut sein, dass diese eine hohe Performance erreicht. Die beiden anderen Implementierungen sind eventuell langsamer.

Programmcode

Anmerkung: Der folgende Test wurde drei Mal implementiert. Es wurden die Klassen SetsHashSetTest, SetsLinkedHashSetTest und SetsTreeSetTest umgesetzt. Es wird nachfolgend jedoch nur die Klasse SetsHashSetTest aufgeführt. Die Klassen SetsLinkedHashSetTest und SetsTreeSetTest unterscheiden sich nämlich nur in dem Punkt, dass bei der Initialisierung (mit rotem Rahmen gekennzeichnet) an Stelle eines HashSets ein LinkedHashSet beziehungsweise ein TreeSet initialisiert wird. Die Klasse SetOperations ermöglicht das Verhindern von Code-Duplizierung.

```
@State(Scope.Thread)
public class SetsHashSetTest {
	private SetOperations setOps = new SetOperations(new HashSet<>(),
	                                                 new HashSet<>());

	private List<Integer> collection;
	private List<Integer> list;
	private Blackhole b = new Blackhole();

	public SetsHashSetTest() {
		collection = new ArrayList<Integer>();
		for (int i = 0; i < (References.BENCHMARK.SETREPETITIONS / 100); i++){
			for (int j = i * 100; j < (i + 1) * 100; j++) {
				collection.add(j);
			}
		}

		list = new LinkedList<>();

		list.clear();
		for (int i = 2000; i < 4000; i++) {
			list.add(i);
		}
	}

	// basic runtime
	@Benchmark
	public void basicRuntime() {
		setOps.initializeSets();
	}
```

```
	@Benchmark
	public void addToEmptySet() {
		setOps.initializeSets();

		for (int i = 0; i < References.BENCHMARK.SETREPETITIONS; i++) {
			boolean retVal = setOps.add(i);
			b.consume(retVal);
		}
	}

	@Benchmark
	public void addAllToEmptySet() {
		setOps.initializeSets();

		boolean retVal = setOps.addAll(collection);
		b.consume(retVal);
	}

	@Benchmark
	public void containsInFullSet() {
		setOps.initializeSets();

		for (Integer i : setOps.getEmptySet()) {
			boolean retVal = setOps.contains(i);
			b.consume(retVal);
		}
	}

	@Benchmark
	public void containsAllInFullSet() {
		setOps.initializeSets();

		boolean retVal = setOps.containsAll(collection);
		b.consume(retVal);
	}

	@Benchmark
	public void removeFromFullSet() {
		setOps.initializeSets();

		Iterator<Integer> it = setOps.getFullSet().iterator();
		while (it.hasNext()) {
			Integer currentInteger = it.next();
			b.consume(currentInteger);
			setOps.remove(it);
		}
	}

	@Benchmark
	public void removeAllFromFullSet() {
		setOps.initializeSets();

		boolean retVal = setOps.removeAll(collection);
		b.consume(retVal);
	}

	@Benchmark
	public void clearFullSet() {
		setOps.initializeSets();

		setOps.clear();
		b.consume(setOps);
	}

	@Benchmark
	public void retainAllInFullSet() {
```

```
                setOps.initializeSets();

                setOps.retainAll(list);
                b.consume(setOps);
        }
}
```

Diagramm und Beobachtung

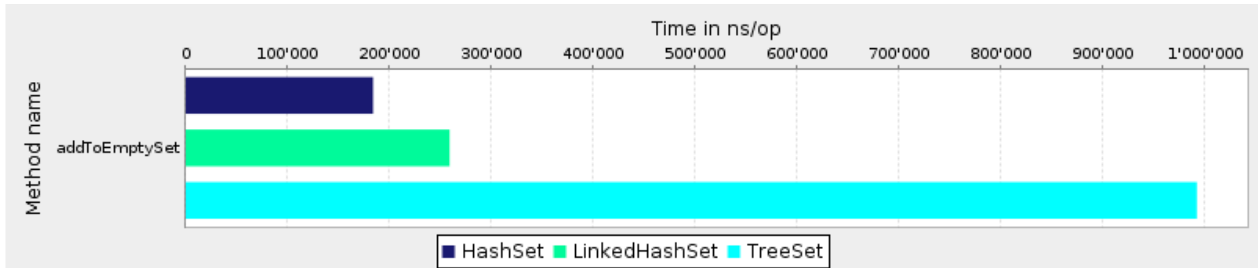

Abbildung 86: Add-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest

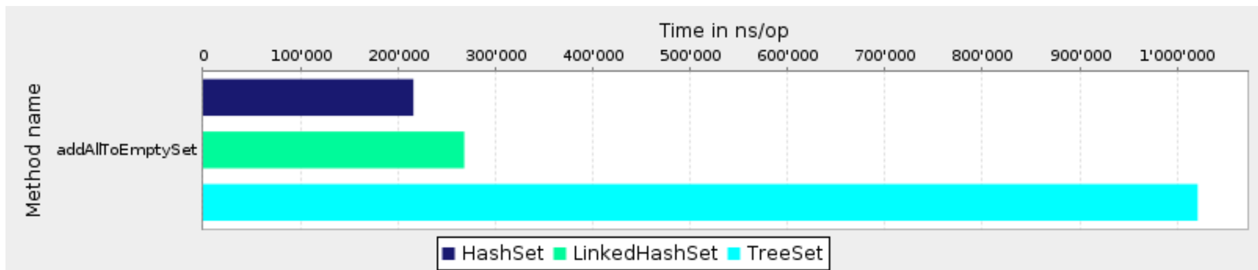

Abbildung 87: AddAll-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest

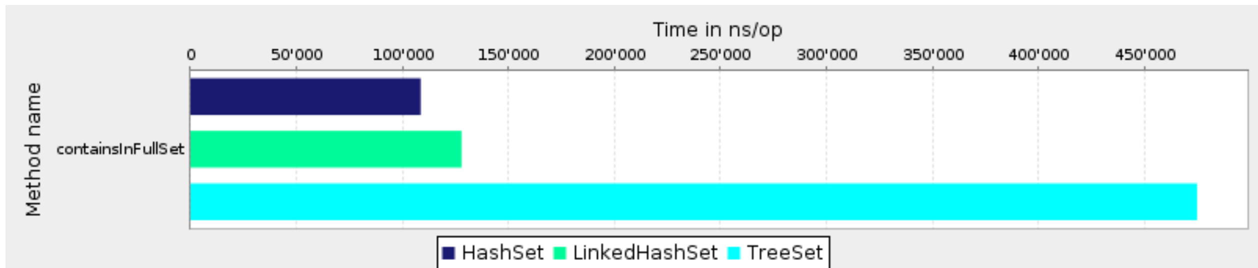

Abbildung 88: Contains-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest

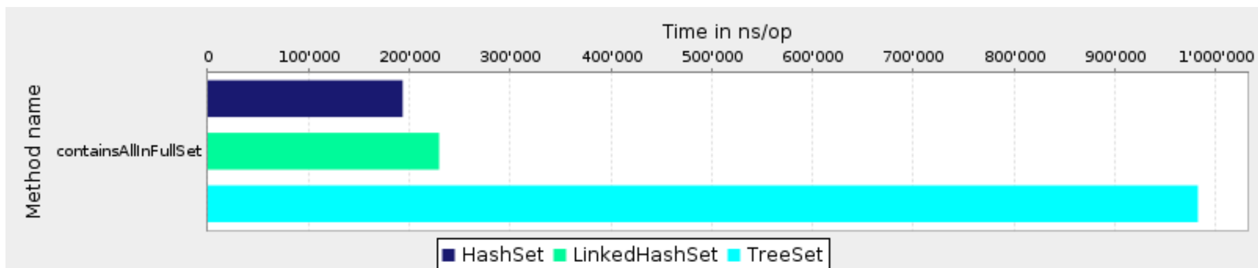

Abbildung 89: ContainsAll-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest

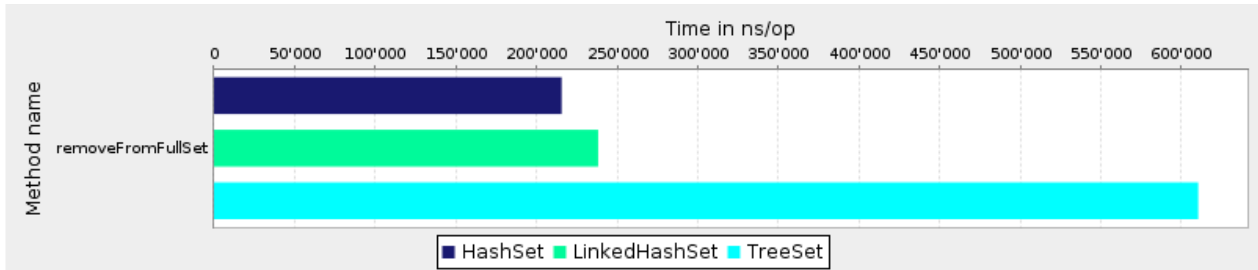

Abbildung 90: Remove-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest

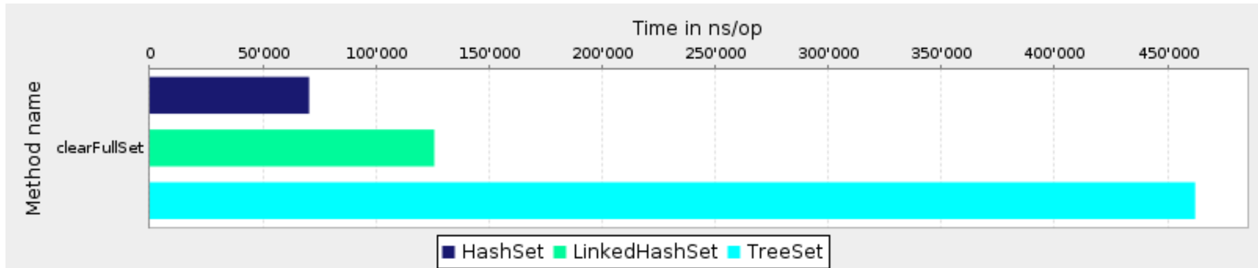

Abbildung 91: Clear-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest

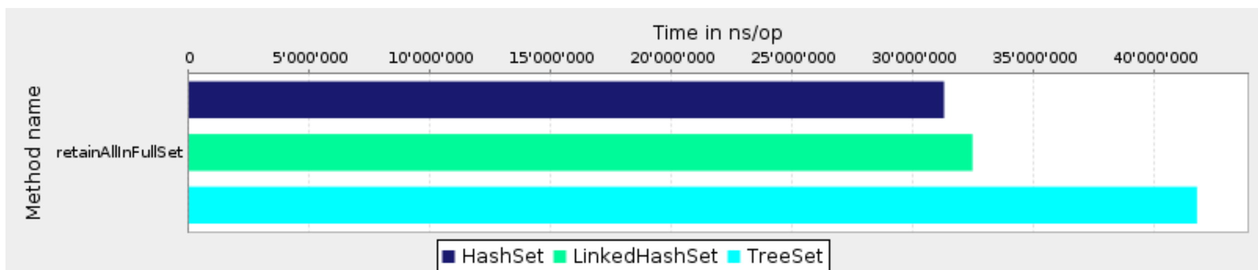

Abbildung 92: RetainAll-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest

Auf den oben aufgeführten Diagrammen sind die Operationen add(...), addAll(...), clear(...), contains(...), containsAll(...), remove() und retainAll(...) aufgeführt. Es ist gut erkennbar, dass die TreeSet-Implementierung jeweils mit Abstand die ineffizienteste Variante ist. Das HashSet und das LinkedHashSet sind beide effizient, wobei das HashSet insgesamt etwas effizienter ist als das LinkedHashSet.

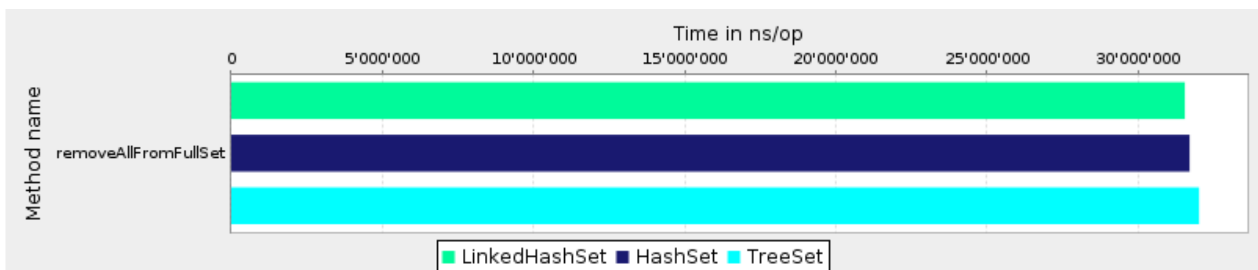

Abbildung 93: RemoveAll-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest

Bei der Methode removeAll(...) sind die Unterschiede weniger deutlich zu erkennen. Auffallend ist auch, dass hier das LinkedHashSet etwas besser abschneidet als das HashSet.

Auswertung/Empfehlung

Da das HashSet in sieben von acht Tests die kürzeste Laufzeit erreichte, wird empfohlen, wenn möglich, ein HashSet zu verwenden. Vor allem von der Verwendung des TreeSets wird aufgrund der auffallend längeren Laufzeit abgeraten.

4.29 Implementierung eines Stacks mit ArrayDeque und LinkedList

Thema/Fragestellung

Es wird empfohlen, an Stelle der Klasse Stack eine Implementierung des Deque-Interfaces zu verwenden (Oracle 2016 A). Für diese Benchmarks wurde die Implementierung ArrayDeque gewählt. Sie ist nicht synchronisiert, unterstützt das gleichzeitige Mutieren durch mehrere Threads also nicht. Zusätzlich wurde eine gängige Implementierung eines Stacks mit einer LinkedList umgesetzt. Doch welche Implementierung ist, bezogen auf die Performance, zu empfehlen?

Vermutung/Hypothese

Da die Klasse ArrayDeque für den Einsatz als Stack vorgesehen ist, wird diese vermutlich eine hohe Performance erreichen. Die gängige Implementierung des Stacks mit einer LinkedList als Datenstruktur erreicht aber wahrscheinlich eine ähnlich hohe Performance.

Programmcode

```
private Deque<String> stack;
private Blackhole b = new Blackhole();

//basic runtime
@Benchmark
public Object push() {
	initFillAndConsumeStack();
	return stack;
}

@Benchmark
public Object pop() {
	initFillAndConsumeStack();
	while (!stack.isEmpty())
	{
		b.consume(stack.pop());
	}
	return stack;
}

@Benchmark
public Object removeAll() {
	initFillAndConsumeStack();
	stack.removeAll(stack);
	return stack;
}
```

```java
private void initFillAndConsumeStack() {
        stack = new ArrayDeque<String>();
        for(int i = 0; i < References.BENCHMARK.REPETITIONS; i++) {
                stack.push(i + "-th call");
        }
        b.consume(stack);
}
```

Diagramm und Beobachtung

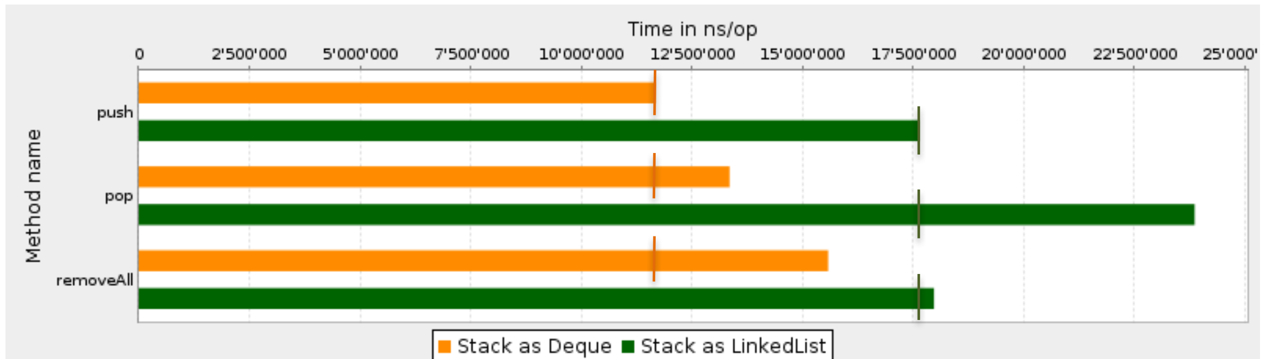

Abbildung 94: Benchmarks aus den Klassen StackAsDequeList und StackAsLinkedListTest

Hinweis: Der Benchmark push zeigt gleichzeitig auch die “Grund“-Laufzeit auf, die für die jeweilige Datenstruktur bei allen Benchmarks gilt.

Alle Methoden des Stacks, der als ArrayDeque implementiert wurde, sind performanter als diejenigen des Stacks, der als LinkedList implementiert wurde.

Auswertung/Empfehlung

Da die getesteten Methoden denjenigen Methoden entsprechen, die auf einem Stack wohl am häufigsten aufgerufen werden, kann als Fazit ausgesagt werden, dass die Stack-Implementierung mit einem ArrayDeque-Objekt wegen ihren Performance-Vorteilen der Stack-Implementierung mit einer LinkedList vorgezogen werden sollte.

5. Diskussion und Ausblick

5.1 Zusammenfassung der Resultate

In der Einleitung dieser Arbeit wird die Frage gestellt, ob bezüglich der Performance eine for-Schleife, eine foreach-Schleife, ein Iterator oder ein ListIterator gewählt werden soll. Ein Vergleich zwischen diesen und anderen Konstrukten, wie der while-Schleife und den äquivalenten, funktionalen Programmierkonstrukten, wurde im Zusammenhang mit den Listen durchgeführt. Die beiden Listen, die bei diesen Experimenten gebraucht wurden, sind die ArrayList und die LinkedList. Bei allen Iterationsvarianten ist die ArrayList schneller als die LinkedList.

Bei der ArrayList sind die for-Schleife, foreach-Schleife, while-Schleife, der Iterator und der ListIterator etwa gleich schnell, während die funktionalen Programmierkonstrukte langsamer sind. Dabei ist das Durchlaufen der Liste mit einem Parallel-Stream schneller als mit einem sequenziellen Stream. Bei der LinkedList sind die foreach-Schleife, der Iterator und der ListIterator die drei schnellsten Iterationsvarianten. Bezüglich der Performance kommen die funktionalen Programmierkonstrukte als Nächstes. Der Parallel-Stream ist auch bei der LinkedList schneller als der sequenzielle Stream, jedoch ist der Unterschied kleiner als bei der ArrayList. Die for- und while-Schleife sollten nie im Zusammenhang mit einer LinkedList verwendet werden, weil sie knapp 300 Mal langsamer sind als die foreach-Schleife, der Iterator und der ListIterator.

Eine weitere Frage, die zu Beginn der Bachelorarbeit gestellt wurde, ist, wie Java-Streams im Vergleich zu Code-Fragmenten aus Java 7 stehen. In dieser Arbeit wurde gezeigt, dass Java-Streams in einigen Fällen mit der Konkurrenz gut mithalten können und in anderen Fällen deutlich schlechter abschneiden als die anderen Code-Konstrukte. Bei der Anwendung der Schleifen auf Listen wurde gezeigt, dass die Streams nicht die beste Performance anbieten, aber sich im Gegensatz zu den anderen Programmierkonstrukten nicht zeitineffizient verhalten. Wenn es aber darum geht, mehrfach vorhandene Werte aus Listen zu löschen, sind Streams eine schlechte Wahl. Bei der Anwendung derselben Operation auf Arrays, entsteht ein anderes Resultat. Der sequenzielle Stream ist da deutlich schneller als das übliche Code-Fragment aus Java 7. Bei der Sortierung von Arrays oder Listen sind die Code-Fragmente aus Java 7 ein bisschen schneller als die Code-Fragmente mit Streams aus Java 8. Die Laufzeiten liegen aber nicht weit auseinander. Hingegen bei der Summenbildung aus Arrays oder Listen schneiden die Streams im Gegensatz zur gewöhnlichen for-Schleife markant schlechter ab.

Im Rahmen dieser Bachelorarbeit konnten weitere Experimente durchgeführt werden, die zu interessanten Ergebnissen geführt haben. Beispielsweise ist die Trennung von Strings in Wörtern mit dem StringTokenizer sehr performant, während der Scanner eine mit Abstand schlechtere Performance aufzeigt. Ein weiteres spannendes Ergebnis ist, dass der Performance-Unterschied zwischen dem "gewöhnlichen" mathematischen Operator und dem Shift-Operator bei der Multiplikation geringer ist als bei der Division.

5.2 Weitere mögliche Messungen

Es gibt weitere Experimente und Messungen, die in Java durchgeführt werden können und in dieser Bachelorarbeit nicht zum Zug gekommen sind. Die Streams wurden vor allem in Bezug auf Arrays und Listen getestet. Es sind aber weitere Experimente mit Streams denkbar. Es können beispielsweise Datenstrukturen wie Maps oder Sets verwendet werden. Es ist auch denkbar, weitere Experimente aufzubauen, bei denen verschiedene Streamtypen getestet und verglichen werden. In dieser Bachelorarbeit wurde nur der IntStream eingesetzt. Allenfalls können mit diesen Streams auch noch zusätzliche Lambda-Ausdrücke kombiniert und daraus weitere sinnvolle Experimente entstehen.

In dieser Bachelorarbeit wurden gewisse Themen aufgegriffen, die weiter ausgebaut werden können. Es wurden zum Beispiel Experimente mit Strings durchgeführt, um die Performance diverser String-Operationen zu ermitteln: Das Trennen von Strings in Wörter, das Trennen von Strings in Zeichen, die String-Verkettung und die Prüfung von Strings auf Gleichheit. Weitere Experimente, wie die Suche von Teilstrings innerhalb von bestimmten Strings oder die Ermittlung der Zeichenposition innerhalb eines Strings, sind denkbar.

6. Verzeichnisse

Literaturverzeichnis

Simbürger, Manuel / Thöni, Roman (2015): *Best-Practices zur Performance-Optimierung bei der Software-Entwicklung in Java.* (= Institut für angewandte Informationstechnologie). Zürich: Zürcher Hochschule für Angewandte Wissenschaften.

Oaks, Scott (2014): Java Performance. The Definitive Guide. Sebastopol: O'Reilly Media.

Gordon, Brian (2014): Optimizations performed by javac. URL: https://briangordon.github.io/2014/01/javac-optimizations.html [Stand: 6.6.2016]

Vorontsov, Mikhail (2014): Introduction to JMH. URL: http://java-performance.info/jmh/ [Stand: 6.6.2016]

Shipilëv, Aleksey (2013): *Java Microbenchmark Harness (the lesser of two evils).* URL: http://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf [Stand: 6.6.2016]

OpenJDK (2016): JMHSample_11_Loops.java. URL: http://hg.openjdk.java.net/code-tools/jmh/file/bcec9a03787f/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_11_Loops.java [Stand: 6.6.2016]

OpenJDKWiki (2012): So You Want To Write a Micro-Benchmark. URL: https://wiki.openjdk.java.net/display/HotSpot/MicroBenchmarks [Stand: 6.6.2016]

Oracle (2016 A): Java™ Platform, Standard Edition 8 API Specification. URL: https://docs.oracle.com/javase/8/docs/api/ [Stand: 6.6.2016]

Oracle (2016 B): Java Virtual Machine Specification. URL: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html [Stand: 6.6.2016]

Oracle (2016 C): javac - Java programming language compiler. URL: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html [Stand: 6.6.2016]

Oracle (2016 D): Understanding Just-In-Time Compilation and Optimization. URL: http://docs.oracle.com/cd/E15289_01/doc.40/e15058/underst_jit.htm [Stand: 6.6.2016]

Oracle (2016 E): Java SE Development Kit 8 Downloads. URL: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html [Stand: 6.6.2016]

Oracle (2016 F): The switch Statement. URL: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html [Stand: 6.6.2016]

Moertel, Tom (2013): Tricks of the trade: Recursion to Iteration, Part 4: The Trampoline. URL: http://blog.moertel.com/posts/2013-06-12-recursion-to-iteration-4-trampolines.html [Stand: 6.6.2016]

Ullenboom, Christian (2013): Die Java 8 JVM wird keine Tail Call Recursion Optimierung bekommen. URL: http://www.tutego.de/blog/javainsel/2013/06/die-java-8-jvm-wird-keine-tail-call-recursion-optimierung-bekommen/ [Stand: 6.6.2016]

Rauh, Stephan (2015): A Java Programmer's Guide to Byte Code. URL: http://www.beyondjava.net/blog/java-programmers-guide-java-byte-code/ [Stand: 6.6.2016]

Rowell, Eric (2016): Know Thy Complexities! URL: http://bigocheatsheet.com [Stand: 6.6.2016]

Molloy, Derek (2006): *The Java LifeCycle.* URL: http://www.eeng.dcu.ie/~ee553/ee402notes/html/ch04s02.html#javalifecycle [Stand: 6.6.2016]

Gräsel, Alexander (2012): Java: Ungenauigkeit / Messfehler in System.currentTimeMillis() und new Date(). URL: http://blog.axxg.de/java-ungenauigkeit-zeitmessung/ [Stand: 6.6.2016]

Pavlov, Igor (2016): 7-Zip LZMA Benchmark. URL: http://www.7-cpu.com [Stand: 6.6.2016]

Stack Overflow (2011): Why switch is faster than if. URL: http://stackoverflow.com/questions/6705955/why-switch-is-faster-than-if [Stand: 6.6.2016]

Grepcode™ (2015 A): Java 8 - java.util.Arrays. URL: http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/util/Arrays.java [Stand: 6.6.2016]

Grepcode™ (2015 B): Java 8 - java.util.ArrayList. URL: http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/util/ArrayList.java [Stand: 6.6.2016]

Grepcode™ (2011): Java 7 - java.util.ArrayList. URL: http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/7-b147/java/util/ArrayList.java [Stand: 6.6.2016]

Wikibooks (2016): *Java Programming/Print Version: Java Runtime Environment (JRE).* URL: https://en.wikibooks.org/wiki/Java_Programming/Print_version#Java_Runtime_Environment_.28JRE.29 [Stand: 6.6.2016]

Wikipedia (2016 A): *Integrierte Entwicklungsumgebung.* URL: https://de.wikipedia.org/wiki/Integrierte_Entwicklungsumgebung [Stand: 6.6.2016]

Wikipedia (2016 B): *Java Virtual Machine (JVM).* URL: https://de.wikipedia.org/wiki/Java_Virtual_Machine [Stand: 6.6.2016]

Abbildungsverzeichnis

Abbildung 1: Java Development-Cycle (Eigene Abbildung) 10
Abbildung 2: Bytecode-Interpreter (Wikibooks 2016) 11
Abbildung 3: Kompilierung von Java-Bytecode innerhalb der JVM (Wikibooks 2016) 12
Abbildung 4: Resultat einer Multiplikation wird einer statischen Variable zugewiesen (Gordon 2016) 13
Abbildung 5: Bytecode zur Initialisierung einer statischen Variable (Gordon 2016) 13
Abbildung 6: Einer Variable wird eine konstante logische Aussage zugewiesen (Gordon 2016) 14
Abbildung 7: Bytecode mit der Initialisierung der statischen, booleschen Variable (Gordon 2016) 14
Abbildung 8: Unerreichbarer Java-Code (Gordon 2016) 14
Abbildung 9: Bytecode des unerreichbaren Java-Codes (Gordon 2016) 14
Abbildung 10: Eine nicht verwendete private-Methode (Eigene Abbildung) 15
Abbildung 11: Die ".class"-Datei ohne die private-Methode (Eigene Abbildung) 15
Abbildung 12: Erste Variante der If-Verzweigung (Gordon 2016) 15
Abbildung 13: Zweite Variante der If-Verzweigung (Gordon 2016) 15
Abbildung 14: Bytecode der ersten If-Verzweigung (Gordon 2016) 15
Abbildung 15: Bytecode der zweiten If-Verzweigung (Gordon 2016) 16
Abbildung 16: String-Verkettung in einem Statement (Gordon 2016) 16
Abbildung 17: Bytecode der String-Verkettung in einem Statement (Gordon 2016) 16
Abbildung 18: Java-Code mit zwei String-Verkettungszeilen (Gordon 2016) 16
Abbildung 19: Bytecode mit zwei Aufrufen des StringBuilder-Konstruktors (Gordon 2016) 17
Abbildung 20: Verkettung von String-Konstanten in einem Statement (Gordon 2016) 17
Abbildung 21: Ein String in den Stack laden (Gordon 2016) 17
Abbildung 22: Der Wert einer statischen Variable auf der Konsole drucken (Gordon 2016) 18
Abbildung 23: Static Variable wird im static-Block initialisiert. Wert wird mit getstatic geholt. (Gordon 2016) 18
Abbildung 24: SECONDS_IN_30_DAYS ist hier static final (Gordon 2016) 18
Abbildung 25: Inlining einer static final Variable durch den Einsatz der Instruktion ldc (Gordon 2016) 18
Abbildung 26: Input-Werte sollen, wenn möglich, von Instanzvariablen gelesen werden (Vorontsov 2014). 19
Abbildung 27: Die Klasse Point (Oaks 2014, 96) 20
Abbildung 28: Instanziierung von Point und Einsatz von Getter- und Setter-Methoden (Oaks 2014, 96) 21
Abbildung 29: Optimierung des JIT-Compilers als Java-Code dargestellt (Oaks 2014, 96) 21
Abbildung 30: Von y nach z kopieren 21
Abbildung 31: Die Variable z wurde hier durch y ersetzt. 22
Abbildung 32: Anfang der Tabelle "Benchmarks und ihre Fehler" 31
Abbildung 33: Ende der Tabelle "Benchmarks und ihre Fehler" 31

Abbildung 34: Laufzeiten der Benchmarks aus der Klasse MultiplyOpVsAdditionInLoopOpTest 35
Abbildung 35: Laufzeiten der Benchmarks aus der Klasse ModuloInstanceVsLocalVariablePlusEqualsTest 37
Abbildung 36: Laufzeiten der Benchmarks aus der Klasse Modulo3InstanceVsLocalVariablePlusEqualsTest 38
Abbildung 37: Laufzeiten der Benchmarks aus der Klasse ArrayCopyOperationsTest 40
Abbildung 38: Laufzeiten der Benchmarks aus der Klasse ArrayListsInitializationTest 42
Abbildung 39: Laufzeiten der Benchmarks aus der Klasse ArrayVsArrayListTest 43
Abbildung 40: Laufzeiten der Benchmarks aus der Klasse AutoboxingVsCastingTest 45
Abbildung 41: Laufzeiten der Benchmarks aus der Klasse ComparableVsComparatorTest 47
Abbildung 42: Laufzeiten der Benchmarks aus der Klasse CounterAndLoopMaxTypeTest 50
Abbildung 43: Laufzeiten der Array-Benchmarks aus der Klasse FilterValuesTest 52
Abbildung 44: Laufzeiten der ArrayList-Benchmarks aus der Klasse FilterValuesTest 52
Abbildung 45: Laufzeiten der Benchmarks aus der Klasse FunctionVsMethodCompositionTest 54
Abbildung 46: Laufzeiten der Benchmarks aus der Klasse IterativeVsRecursive200FactorialTest 56
Abbildung 47: Laufzeiten der Benchmarks aus der Klasse IterativeVsRecursive500FactorialTest 57
Abbildung 48: Laufzeiten der Benchmarks aus der Klasse IterativeVsRecursive1000FactorialTest 57
Abbildung 49: Laufzeiten der Benchmarks aus der Klasse IterativeVsRecursive3000FactorialTest 57
Abbildung 50: Laufzeiten der Benchmarks aus der Klasse IterativeVsRecursive8000FactorialTest 57
Abbildung 51: Laufzeiten der Multiplikation-Benchmarks aus der Klasse MathVsBitOperationsTest 59
Abbildung 52: Laufzeiten der Division-Benchmarks aus der Klasse MathVsBitOperationsTest 59
Abbildung 53: Laufzeiten der Cleaning-Benchmarks aus der Klasse ObjectCloningAndCleaningTest 61
Abbildung 54: Laufzeiten der Benchmarks aus der Klasse ObjectCloningAndCleaningTest 64
Abbildung 55: Vierstufige Vererbungshierarchie (Eigene Abbildung) 64
Abbildung 56: Laufzeiten der Benchmarks aus der Klasse PolymorphismTest 65
Abbildung 57: Laufzeiten der Benchmarks aus der Klasse RemoveDuplicateValuesInArrayTest 68
Abbildung 58: Laufzeiten der Benchmarks aus der Klasse RemoveDuplicateValuesInListTest 68
Abbildung 59: Laufzeiten der ArrayList-Benchmarks aus der Klasse SortArraysAndArrayListsTest 70
Abbildung 60: Laufzeiten der Array-Benchmarks aus der Klasse SortArraysAndArrayListsTest 70
Abbildung 61: Laufzeiten der Benchmarks aus der Klasse StaticVsNonStaticMethodsTest 72
Abbildung 62: Laufzeiten der Benchmarks aus der Klasse StringConcatenationTest 74
Abbildung 63: String-Vergleich-Benchmarks aus der Klasse StringSplittingAndComparingTest 75
Abbildung 64: Text-In-Zeichen-Trennung-Benchmarks aus der Klasse StringSplittingAndComparingTest 78
Abbildung 65: Text-In-Wörter-Trennung-Benchmarks aus der Klasse StringSplittingAndComparingTest 79
Abbildung 66: Laufzeiten der Array-Benchmarks aus der Klasse SumArraysAndArrayListsTest 81
Abbildung 67: Laufzeiten der ArrayList-Benchmarks aus der Klasse SumArraysAndArrayListsTest 81
Abbildung 68: Laufzeiten der Benchmarks aus der Klasse SwitchCaseAndIfElseChainTest 83
Abbildung 69: Laufzeiten der Benchmarks aus der Klasse SwitchStatementsTest 86
Abbildung 70: Laufzeiten der Benchmarks aus der Klasse SynchronizedAccessTest 89
Abbildung 71: Add-Benchmarks aus den Klassen ListArrayListTest und ListLinkedListTest 94
Abbildung 72: AddAll-Benchmarks aus den Klassen ListArrayListTest und ListLinkedListTest 95
Abbildung 73: Iterations-Benchmarks aus den Klassen ListArrayListTest und ListLinkedListTest 95
Abbildung 74: Loop-Benchmarks aus den Klassen ListArrayListTest und ListLinkedListTest 96

Abbildung 75: Put-Benchmarks aus den Maps-Benchmarkklassen 102
Abbildung 76: Put-Benchmarks aus den Maps-Benchmarkklassen 102
Abbildung 77: Get-Benchmarks aus den Maps-Benchmarkklassen 102
Abbildung 78: Remove-Benchmarks aus den Maps-Benchmarkklassen 102
Abbildung 79: ContainsKey-Benchmarks aus den Maps-Benchmarkklassen 103
Abbildung 80: ContainsValue-Benchmarks aus den Maps-Benchmarkklassen 103
Abbildung 81: Replace-Benchmarks aus den Maps-Benchmarkklassen 103
Abbildung 82: ReplaceAll-Benchmarks aus den Maps-Benchmarkklassen 103
Abbildung 83: Replace-Benchmarks aus den Maps-Benchmarkklassen 104
Abbildung 84: Clear-Benchmarks aus den Maps-Benchmarkklassen 104
Abbildung 85: SwapAll-Benchmarks aus den Maps-Benchmarkklassen 104
Abbildung 86: Add-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest 107
Abbildung 87: AddAll-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest 107
Abbildung 88: Contains-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest 107
Abbildung 89: ContainsAll-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest 107
Abbildung 90: Remove-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest 108
Abbildung 91: Clear-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest 108
Abbildung 92: RetainAll-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest 108
Abbildung 93: RemoveAll-Benchmarks aus den Klassen Sets(Linked)HashSetTest und SetsTreeSetTest 108
Abbildung 94: Benchmarks aus den Klassen StackAsDequeList und StackAsLinkedListTest 110

Tabellenverzeichnis

Tabelle 1: Optimierungsstrategien des Java-Compilers 13
Tabelle 2: Optimierungsstrategien des Just-In-Time-Compilers 19
Tabelle 3: Verwendete Werte beim Durchführen der Messungen mit dem JMH-Framework 26
Tabelle 4: Details zur Testumgebung 27
Tabelle 5: Beschreibung der Tests mit Kapitelnummern 33

Glossar

Begriff	*Erklärung*
Annotation	Eine Annotation ist ein Sprachelement der Programmiersprache Java, um Metadaten im Sourcecode einzubetten. Dies hilft unter anderem, ein Programm zu steuern.
Benchmark	Ein Benchmark ist eine Java-Methode, die mit der @Benchmark-Annotation annotiert wurde. JMH berücksichtigt beim Ausführen der Performance-Messungen alle Methoden mit der @Benchmark-Annotation und misst deren Laufzeiten.
Bytecode	Nach dem Programmieren von Java-Programmcode kann dieser mit dem Java-Compiler zu Java-Bytecode übersetzt werden. Dieser wird dann von der Java Virtual Machine (JVM) zur Laufzeit in die lokale Maschinensprache übersetzt und ausgeführt.
Classpath	Ist der Pfad, der auf Klassen verweist, die beim Kompilieren und Assemblieren eines Java-Programms benötigt werden.
Compiler	Der Compiler ist ein Programm, das Sourcecode aus einer bestimmten Programmiersprache in einer unabhängigen Zwischensprache (wie Java-Bytecode) oder in einer maschinennahen Sprache übersetzt.
Constant-Pool	Jede Bytecode-Klasse enthält ein Pool, in dem Konstanten stehen. Eine Bytecode-Instruktion kann auf einen bestimmten Index des Pools referenzieren, wenn sie auf diese Konstante zugreifen möchte.
Development-Cycle	Der Development-Cycle beschreibt den Ablauf, sprich die benötigten Schritte, um vom Sourcecode zum lauffähigen Programm zu gelangen.
Framework	Ein Framework stellt ein Rahmen zur Verfügung, das Software-Entwicklerinnen und Software-Entwicklern erleichtert, eine bestimmte Aufgabe umsetzen zu können.
IDE	Ein Integrated Development Environment (IDE) ist eine Sammlung von Anwendungsprogrammen mit denen die Aufgaben der Software-Entwicklung möglichst ohne Medienbrüche bearbeitet werden können (Wikipedia, 2016 A).
Instruction Set	Das Instruction Set ist die Menge aller Maschinenbefehle, die ein Prozessor kennt und ausführen kann.
Interpreter	Der Interpreter ist ein Programm, das Sourcecode einliest, analysiert und direkt ausführt.
Invokation	Eine Invokation ist ein einzelner Aufruf eines Benchmarks innerhalb einer Iteration.
Iteration	Eine Iteration stellt ein Durchlauf eines Benchmarks dar (siehe auch Warmup- und Measurement-Iteration). Eine Iteration besteht aus einer oder mehreren Invokationen.
JMH	Das Framework Java Microbenchmarking Harness (JMH) dient zur Messung der Laufzeiten verschiedener Java-Code-Abschnitten.

JRE	Das Java Runtime Environment (JRE) ist eine speziell konzipierte Laufzeit-umgebung, die das Ziel hat, jede Java-Applikation, unabhängig von Betriebs-system, Platt-form oder Architektur des Computers, ausführen zu können (Wikibooks 2016). Die JRE setzt direkt auf dem Betriebsystem des jeweiligen Rechners auf.
JVM	Die Java Virtual Machine (JVM) ist eine virtuelle Maschine und ist der Teil des Java-Runtime Environments (JRE) für Java-Programme, der für die Ausführung des Java-Bytecodes verantwortlich ist (Wikipedia, 2016 B).
Measurement-Iteration	Measurement-Iterationen dienen der effektiven Messung. Die Benchmarks werden dabei meistens mehrfach ausgeführt und dabei wird die Zeit gemessen und von JMH festgehalten.
Memory	Memory ist ein Synonym für den Arbeitsspeicher eines Rechners. Im Arbeitsspeicher werden Daten abgelegt, auf die der Rechner schnell zugreifen kann.
Messabweichung	Entspricht einer Abweichung, die von der durchschnittlichen Messzeit eines Benchmarks ausgeht.
Symbol Table	Eine Symbol Table ist eine Datenstruktur, die wichtige Informationen über Bezeichner enthält. Diese Informationen können sich unter anderem auf die Deklaration oder Definition des Bezeichners beziehen. Eine Symbol Table wird oft von Compilern oder Interpretern erstellt.
Vertrauensintervall	Das Vertrauensintervall einer Messung ist der Bereich, in den eine bestimmte Messzeit mit einer gegebenen Wahrscheinlichkeit fallen wird. Wenn das Vertrauensintervall 99.9% entspricht, befindet sich eine gemessene Laufzeit zur Wahrscheinlichkeit von 99.9% in diesem Intervall.
Virtual Machine (VM)	Eine VM (Virtual Machine) ist eine Nachbildung eines Rechners und läuft dabei auf einem (meist realen) Host-Rechner. Im konkreten Fall von Java laufen alle Programme innerhalb der Java Virtual Machine (JVM). Dies erlaubt die hohe Portabilität.
Warmup-Iteration	Warmup-Iterationen helfen dem JMH-Framework, einen stabilen Zustand für die späteren Measurement-Iterationen herzustellen. Das Ziel dabei ist primär, den JIT-Compiler anzusprechen. Dieser nimmt nämlich nach einer gewissen Zeit verschiedene Optimierungen vor, die optimaler Weise während der Warmup-Iterationen stattfinden sollte.

7. Anhang

- **Offizielle Aufgabenstellung**
- **Datenträger**

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Best-Practices für performante Java-Programmierung
[Software Engineering]
BA16_ciel_1

BetreuerInnen: Mark Cieliebak, ciel
Markus Thaler, tham
Fachgebiete: Software (SOW)

Studiengang: IT
Zuordnung: Institut für angewandte Informationstechnologie (InIT)

Gruppengrösse: 2

Kurzbeschreibung:
Qualitativ hochwertige Software ist ein entscheidender Faktor für den nachhaltigen Erfolg von ICT-Produkten. Vereinfacht ausgedrückt gilt: *Gute SW = korrekt + effizient + wartungsfreundlich*.

In dieser Arbeit entwickeln Sie eine **Sammlung von Best-Practices, wie man möglichst effizienten Java-Code** schreibt. Dabei fokussieren wir auf fundamentale Fragen , die in der täglichen Arbeit eines Java-Entwicklers häufig vorkommen. Zum Beispiel: Wann ist ein Array schneller als eine ArrayList? Welchen Einfluss hat Autoboxing auf die Laufzeit? Wie viel Zeit kostet es, ein neues Objekt zu erzeugen? Viele Entwickler haben "ein Bauchgefühl" zu diesen Fragen, aber konkrete Daten dazu gibt es erstaunlicherweise wenig bis gar nicht.

Die Best-Practices sollen auf objektiven und nachvollziehbaren Daten und Messungen beruhen, die in strukturierten Laufzeit-Experimenten erhoben wurden.

Grundlage: In einer Bachelorarbeit wurde bereits ein System (Hardware und Software) aufgebaut, mit dem man auf Knopfdruck vergleichende Effizienz-Messungen machen kann: man programmiert z.B. mehrere Methoden, die jeweils ein Array komplett durchlaufen, und das System führt diese Methoden automatisch aus, ermittelt die Laufzeit der einzelnen Methoden und erzeugt eine entsprechende Grafik. Dieses System können Sie für Ihre Experimente verwenden.

Aufgabe Im Rahmen der Arbeit nehmen Sie das bestehende Framework in Betrieb (Hardware kann gestellt werden). Sie implementieren konkrete Software-Fragmente und vergleichen deren Performance. Ihre Ergebnisse stellen Sie übersichtlich dar.

Publikation Ein Ziel der Arbeit ist, dass Sie Ihre Ergebnisse und Best Practices in einer Publikation veröffentlichen, z.B. im Java-Magazin oder einer Software-Engineering Konferenz.

Voraussetzungen:
- Gute Java-Kenntnisse
- Struturiertes experimentelles Arbeiten

Die Arbeit ist vereinbart mit:
Marco De Tomasi (detomma1)
Markus Rutz (rutzma01)

Datenträger

Der beigefügte Datenträger am Ende des Buches beinhaltet im Hauptverzeichnis sechs Ordner:

- "java-create-charts/"
- "java-performance-tests/"
- "java-result-parser"
- "libs/"
- "sql/"
- „tmp/"

In den ersten drei Ordner befindet sich die gesamte Testumgebung als Source-Code (siehe auch Kapitel 3.3). Es sind 3 Java-Maven-Projekte. Im Ordner "libs/" sind dieselben drei Bestandteile der Testumgebung als ausführbare jar-Files zu finden. Im Ordner "sql/" befindet sich der SQL-Code für das Erstellen oder Widerherstellen der Datenbank. Im Hauptverzeichnis findet sich zudem der Ordner „tmp/". In diesem Ordner sind die Ergebnisse aller ausgewählten Testdurchläufe (siehe auch Kapitel 3.5) zu finden.

Im Hauptverzeichnis ist, zusätzlich zu den sechs Ordnern, der Bericht als PDF-Datei mit dem Dateinamen

- "BA16_Best_Practices_fuer_performante_Java_Programmierung_De_Tomasi_Rutz.pdf"

zu finden.