Java aktuell

iJUG Verbund
www.ijug.eu

Java 21
Das neue LTS-Release im Detail und Hürden beim Upgrade

Kotlin
Unit-Tests mit wenigen Zeilen Code

Work Happier
Spaß an der Arbeit und Glücklichsein als Erfolgsfaktoren

Ausgabe 01/2024
Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwickler
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR
4 191978 304903 01

JavaLand

on demand

JavaLand 2023 verpasst?

Jetzt On-demand-Ticket buchen und Vortragsaufzeichnungen anschauen!

Alle On-demand-Angebote im Ticketshop

Präsentiert von: iJUG Verbund · Heise Medien · DOAG

Veranstalter: JavaLand

Liebe Leserinnen und Leser,

mit dieser Ausgabe heißen wir das im September erschienene langersehnte, aktuelle Long-Term-Support-Release Java 21 offiziell willkommen. Falk Sippach geht mit uns ab Seite 12 die neue Version durch und nimmt alle Neuheiten und Details genauestens unter die Lupe. Im Anschluss präsentiert Nicolai Parlog einige Hürden, denen man beim Upgrade auf Java 21 eventuell begegnen könnte und gibt Hilfestellungen, diese zu überwinden.

Gerrit Grunwald stellt ab Seite 32 das Projekt CRaC (Coordinated Restore at Checkpoint) vor, das es sich zum Ziel gesetzt hat, die Startup-Zeit der JVM auf anderem Wege als mithilfe von Native Images zu lösen. Er gibt einen kurzen Überblick und zeigt, was mit CRaC möglich ist. Die meisten Entwickler sind nicht unbedingt Fans davon, Tests zu schreiben. Wie dies mithilfe weniger Zeilen Kotlin-Code einfacher von der Hand gehen kann, zeigt uns Martin Dilger in seinem Beitrag. Im Anschluss widmet sich Bennet Schulz der Anwendungssicherheit mithilfe des Security-Scanning-Tools Trivy. Dieses eignet sich sowohl zur Prüfung von Sicherheitslücken und Abhängigkeiten als auch zur Erstellung von SBOMs ("Software Bills of Materials").

Einen Kommentar zum Thema KI gibt Marco Schulz ab Seite 54. Er bewertet den derzeitigen Stand der künstlichen Intelligenz und ihre zukünftige Entwicklung. Zum Abschluss dieser Ausgabe zeigt uns Sabine Wojcieszak, weshalb allgemeines Wohlbefinden und Spaß an der Arbeit wichtige Erfolgsfaktoren für das Business sind. Dazu teilt sie einige Tipps und Tricks, wie wir uns selbst in eine positive Stimmung versetzen können.

Wir wünschen euch viel Spaß beim Lesen!

Lisa Damerow
Redaktionsleitung Java aktuell

INHALT

Das neue LTS-Release Java 21 im Detail

Mögliche Herausforderungen beim Upgrade auf Java 21

Höhere Anwendungssicherheit mit dem Security-Scanning-Tool Trivy

Wie Glücklichsein und Spaß unsere tägliche Arbeit erleichtern können

Andreas Badelt, stellv. Leiter der DOAG Cloud Native Community

JAVA TAGEBUCH

28. August 2023

Testcontainers Desktop

AtomicJar hat – quasi als Nebenprodukt aus dem Testcontainers-Cloud-Projekt – *Testcontainers Desktop* entwickelt und jetzt zum freien Download zur Verfügung gestellt. Allerdings handelt es sich im Gegensatz zu den sprachspezifischen *Testcontainer Libraries* wohl nicht um ein Open-Source-Projekt – was immer das für die zukünftige Strategie heißen könnte *[1]*.

2. September 2023

Spring Data: Aus N+1 mach 1 (hoffentlich)

Für alle N+1-Geplagten: Mit dem neuen *Milestone Build 3.2.0-M2* bietet *Spring Data JDBC* jetzt *Single Query Loading*, um „beliebige Aggregate mit einem einzelnen Select-Statement zu laden". Allerdings ist das „beliebig" bislang noch nicht wörtlich zu nehmen, es gibt noch starke Einschränkungen – das soll sich aber in weiteren Releases ändern. Zudem muss die Datenbank *Window Functions* unterstützen. Trotzdem ist Jens Schauder (Spring-Data-Team und der Kopf hinter dem Feature) optimistisch: „Dies ist der Anfang vom Ende des N+1-Problems". Eine genauere Beschreibung, was dahintersteckt, gibt's hier *[2]*.

16. September 2023

Der JCP wird 25

Relativ still ist der Java Community Process in sein Jubiläumsjahr gestartet. 25 Jahre alt ist er jetzt. Er hat seit Jahren, insbesondere natürlich durch die Übergabe von Java EE an die Eclipse Foundation, an Bedeutung und „Prominenz" verloren, ist aber immer noch das Herz (oder zumindest ein großer Teil davon) des Java-Ökosystems. Zumindest einen kurzen Blog-Eintrag vom halbjährlichen „Face-to-Face"-Treffen in New York gibt es *[3]*.

19. September 2023

Das JDK 21 ist da!

Das JDK 21 ist erschienen, mit der schon in vergangenen Ausgaben angerissenen Fülle an neuen Features. Ich kann sie hier nicht alle (erneut) aufzählen, es sind allein 15 JEPs, auch wenn einige noch Previews oder Inkubator-Features sind. Mal sehen, wie schnell die beiden JEPs, die die *Virtual Threads* komplettieren sollen, dann in Folge-Releases den Preview-Status verlassen: *Structured Concurrency* und *Scoped Values*. Mit letzteren (ein „immutable" Ersatz für *ThreadLocals*) konnte ich schon ein bisschen hantieren, sie wirkten bereits ziemlich „rund". Wer noch einen Überblick benötigt, kann zum Beispiel Falks Blog-Beitrag lesen *[4]*.

Babylon – neue Idee für's JDK

Ein neuer Projektvorschlag für das OpenJDK, unter keinem geringeren Namen als *Babylon*, scheint in der OpenJDK-Community auf großes Interesse zu stoßen. Ziel von *Project Babylon* ist es, Java für andere Programmiermodelle zu öffnen; beispielhaft werden SQL, differenzierbare Programmierung, Machine-Learning-Modelle und insbesondere GPUs genannt (letzteres vermutlich ja auch wegen des Einsatzes für ML).

Im Wesentlichen soll eine verbesserte und umfassendere Möglichkeit für reflektive Programmierung in beziehungsweise mit Java geschaffen werden, unter dem Titel *Code Reflection*, sodass der Code – sowohl während des Kompilierens als auch zur Laufzeit –, unter anderem mittels verbesserter und neuer APIs, beliebig transformiert und an andere Programmier- oder Architektur-Modelle angepasst werden kann. Falls diese Kurzfassung keinen Sinn ergibt – hier ist eine längere und bessere Schilderung der Pläne (enthält auch den Link zum ursprünglichen Vorschlag) *[5]*.

Jakarta EE-Umfrage 2023

Die Auswertung der diesjährigen Jakarta-EE-Umfrage – durchgeführt von Mitte März bis Ende Mai – ist da. Die Beteiligung war im Vergleich zum letzten Jahr um mehr als 50 % gestiegen, nicht zuletzt aufgrund einer deutlichen Zunahme der Rückmeldungen aus China mit einem Anteil von 27 %. „Traditionell" kommen bei dieser und ähnlichen Umfragen besonders viele Rückmeldungen aus (West-)Europa, gefolgt von den USA oder Asia/Pacific in wechselnder Reihenfolge. Diesmal ist der Anteil von Asia/Pacific sprunghaft auf 41 % gestiegen, deutlich vor Europa. Das liegt aber vermutlich neben einer größer und international aktiver werdenden IT-Industrie in diesen Ländern (siehe auch das vermehrte Mitwirken gerade chinesischer Unternehmen in den Eclipse-Projekten) auch an verstärktem Marketing in dieser Region.

Bei den Wünschen der Teilnehmenden hat sich seit dem letzten Jahr wenig geändert, Verbesserungen beim Support für Kubernetes und für Microservices bleiben die beiden Top-Prioritäten. Nur der Ruf nach schnellerem Support durch die Anbieter von Jakarta-EE- und Cloud-Anbietern wurde durch zwei andere Themen auf Platz fünf verdrängt: Java-SE-Innovationen schneller aufzugreifen sowie bessere Serverless-Unterstützung anzubieten. Wobei ersteres wohl stark durch den Feature-reiche Java SE 21 Release getrieben ist, allen voran *Virtual Threads*.

Interessant ist vielleicht noch, dass sich zwar der geplante Anteil monolithischer Applikationen für den Cloud-Einsatz in den Umfragen seit 2021 bei zirka 17 % eingependelt hat, aber dafür jetzt der Anteil hybrider Ansätze deutlich gestiegen ist, und die reinen Microservice-Lösungen überholt hat. Das passt dann doch halbwegs zu den Meldungen einiger Projekte in letzter Zeit („zurück zum Monolithen"), einhergehend mit der überraschenden Erkenntnis, dass auch wenn ich einen Hammer in der Hand halte, sich nicht alle Probleme direkt in Nägel verwandeln.

Das neue EE 10 hat binnen eines guten halben Jahres bereits einen Marktanteil von 17 % erreicht. Verglichen mit der „Konkurrenz" bei den Java-Cloud-Native-Frameworks hat Jakarta insgesamt einen gleichbleibenden Anteil von 53 % gegenüber 2022, während der von MicroProfile weiter zurückgeht (jetzt 26 %) und Spring verloren gegangene Anteile wieder zurückgewinnen konnte (66 %, Mehrfachnennungen waren möglich); wobei der Begriff „Konkurrenz", wie schon öfter erwähnt, ja nicht wirklich passt, weil insbesondere Jakarta und MicroProfile stark kooperieren, und auch in Spring viele Jakarta-APIs stecken.

25. September 2023

CargoTracker auf EE 10

Für alle, die eine aktuelle Demo-Anwendung für Jakarta EE und seine Features suchen: Der *CargoTracker* ist jetzt auf die neueste Version EE 10 gehoben worden (die Implementierung basiert auf Java SE 11 und dem Payara-Server). EE 9/9.1 wurde dabei direkt übersprungen. Das entsprechende Release-Tag zum direkten Download des Quellcodes ist v3.0 *[6]*.

26. September 2023

JavaLand 2024: Neuer Ort, neuer Termin

Länder zu verschieben ist ein heikles Thema. Da die JavaLand aber immer nur für wenige Tage im Jahr existiert, sollte es funktionieren, auch ohne jemandem wehzutun (außer vielleicht ein paar eingefleischten Achterbahnfans). Nach immer komplizierteren Verhandlungen mit dem Phantasialand (neben einer deutlichen Preiserhöhung wäre auch die traditionelle Nutzung der Fahrgeschäfte am ersten Konferenz-Abend nicht mehr möglich gewesen) haben sich das Konferenzleitungs-Team und der iJUG-Vorstand dazu entschlossen, einen neuen Ort für die Veranstaltung zu suchen.

Die Wahl fiel relativ schnell auf den Nürburgring (bei der Größe ist es ja nicht mehr so einfach, etwas Adäquates zu finden). Ausgerechnet die zehnte Auflage der JavaLand findet also nicht mehr am angestammten Ort und noch dazu an einem ungewohnten Termin statt: vom 9. bis 11. April 2024 (in NRW, Bayern und den meisten anderen Bundesländern ist das kurz nach den Osterferien).

Am Nürburgring gibt es zwar keine Karussells, aber neben „im Kreis fahren" hat die Location noch eine ganze Menge anderes zu bieten. Der neue Ort weckt sicherlich auch die kreativen Kräfte der Community, sodass wir keine langweilige Veranstaltung befürchten müssen.

28. September 2023

Temurin: JDK 21 Builds verzögern sich

Die Eclipse Temurin Builds des JDK 21 verspäten sich noch um mindestens einige Tage, so hat es das Projekt angekündigt. „Wir haben erst wenige Tage vor dem OpenJDK 21 Release davon erfahren, dass die Java 21 TCK-Tests eine neue Lizenzvereinbarung [Anmerkung: mit Oracle] erfordern", heißt es im Adoptium-Blog *[7]*. Jetzt müssen erst mal die Juristen ran und die neue Vereinbarung prüfen, dann kann der Release zertifiziert werden. Andere Builds (Amazon Corretto, Liberica etc.) sind schon in den ersten Tagen nach dem OpenJDK-Release erschienen, Oracles eigenes JDK sowieso. Hatten sie einfach schnellere Anwälte?

Warum das so kurz vor Toresschluss kam, kann vielleicht die Rechtsabteilung bei Oracle beantworten. Allerdings fehlt mir auch die Fantasie dafür, mir das als einen geplanten Schachzug auszumalen, der die Konkurrenz für einige Tage(?) ausbremsen sollte (mal sehen, wie lange es dauert). Es haben bestimmt schon viele auf Java 21 gewartet, aber ob sich dadurch jetzt Schlangen beim „Konkurrenz-Download" bilden – ich weiß nicht so recht.

Update vom 9. Oktober 2023: Laut demselben Blog-Eintrag haben sie jetzt das TCK erhalten und die Builds für die unterschiedlichen Plattformen werden dann in den nächsten Tagen sukzessive veröffentlicht.

29. September 2023

Temurin Comparison Builds

Ein positiveres Thema: Adoptium hat weiter an sicheren und reproduzierbaren Builds gearbeitet und kürzlich für JDK 17 und 21 *Temurin Comparison Builds* eingeführt: verifizierte (und verifizierbare) identische Builds zu den jeweiligen Versionen, die sicherstellen, dass sowohl Sourcecode als auch der gesamte Build-Prozess keinerlei Diskrepanzen (beziehungsweise Regressionen) aufweisen *[8]*.

30. September 2023

Jakarta EE: Versions-API

Jakarta EE soll ein strukturiertes Versions-API erhalten. Insbesondere vor dem Hintergrund zunehmender Abwärts-Inkompatibilitäten sollen damit abhängiger Software genaue Informationen zur Verfügung gestellt werden, auf welcher Plattform-/Profil-Version sie laufen und, welche Einzelspezifikationen mit welchen Versionen unterstützt werden. Die konkrete Ausgestaltung (System Properties etc.) muss aber noch diskutiert werden *[9]*.

Randnotiz: Inzwischen sind übrigens sieben verschiedene Implementierungen für die volle Jakarta-EE-10-Plattform zertifiziert, sechs für das *Web Profile* und fünf für das „minimalistische" *Core Profile* *[10]*.

6. Oktober 2023

JDK 22

Nach dem Release ist vor dem Release, zumal es eine Weile dauert, bis dieses Tagebuch gedruckt vorliegt. Im März kommt Java 22 raus, im Vergleich zum neuen LTS-Release 21 wird es dann wieder etwas ruhiger zugehen.

Neben vielen anderen kleinen Änderungen war ein „breaking

change" schon vor Monaten angekündigt worden: *JLine* ist der neue *Default Console Provider* (nicht mehr „opt-in"). Damit einhergehend wird `System.console()` dann die *Console* (und nicht „null") liefern, wenn die JVM nicht mit einem tatsächlichen Terminal verbunden ist (wie es beispielsweise bei *JShell* der Fall ist). Sollten Applikationen genau dieses Verhalten als Test nutzen, um ihre Umgebung zu „erkunden", würde das in Zukunft scheitern – dazu gibt es nun die Methode `Console.isTerminal()`.

Ansonsten haben zwar kürzlich die JEPs 454 bis 460 den Status *Candidate* erhalten *[11]*, aber nur das Foreign Function & Memory API, das seit Java 19 schon mehrere Preview-Runden hinter sich hat, und das Sprach-Feature *Unnamed Variables and Patterns* (erste Preview in 21) sollen bislang in 22 als stabil aufgenommen werden – ersteres mit kleinen Verbesserungen zur letzten Preview, letzteres ohne jede Änderung.

11. Oktober 2023

Jakarta EE Working Group-Wahlen

Die jährlichen Repräsentanten-Wahlen für die Jakarta EE Working Group sind abgeschlossen. Bei den Firmenvertretern ändert sich nicht allzu viel, lediglich Jun Qian von Primeton Information Technologies ersetzt Scott Stark von Red Hat im Spezifikationskomitee. Darüber hinaus wurden aber jetzt zwei bekannte Gesichter als Committer-Repräsentanten für vakante Positionen gefunden: Werner Keil (Spezifikationskomitee) und Otavio Santana (Marketing-&-Brand-Komitee).

12. Oktober 2023

MicroProfile 6.1

MicroProfile 6.1 ist von der Eclipse Working Group abgesegnet worden. Der iJUG und Red Hat haben zwar dagegen gestimmt, die anderen 10 Mitglieder waren aber dafür. Die Gründe für den iJUG waren im Wesentlichen länger bekannte Sicherheitslücken (CVEs) einzelner Spezifikationen (speziell Config), für die Fixes existieren, die aber in den Releases nicht enthalten sind. iJUG-Repräsentant Jan Westerkamp vertritt hier die (recht nachvollziehbare) Meinung, dass alle lösbaren CVEs im nächsten Release gepatcht werden *müssen* (mindestens mal die in Spezifikationen, die sowieso angefasst werden). Das Thema bekommt für ihn zu wenig Priorität – wobei die Gesetzgebung hier den Projekten in Zukunft sowieso „nachhelfen" wird (Stichwort *Cyber Resilience Act*). Daneben gab es auch Probleme mit dem TCK der *Telemetry*-Spezifikation, das falsche Abhängigkeiten zu Java Concurrency (*Web Profile*) enthält; die Test sind zwar prinzipiell optional, aber dummerweise auch noch per Default aktiviert.

Bei Red Hats Einwänden ging es wohl im Wesentlichen um das Metrics-Projekt, das in MP 6.1 erst mal weiter enthalten ist (Upgrade von 5.0 auf 5.1) und demnächst direkt von *MP Telemetry Metrics* abgelöst werden soll (basierend auf OpenTelemetry); RedHat hat MP Metrics aber bereits in seinen Produkten durch Micrometer ersetzt und muss seinen Kunden dann einen zweiten „breaking change" hin zu Telemetry erklären – der Nachteil, wenn man vorprescht.

18. Oktober 2023

Jakarta EE 11 nimmt Formen an

Der nächste Jakarta EE Release (11) ist noch recht grob für das 1. Halbjahr 2024 anvisiert, aber es wurden inzwischen Meilenstein-Releases festgelegt: M1 ist für den 5. Dezember geplant. Bis dahin müssen alle Spezifikationen, die in EE 11 enthalten sein wollen, Doku und APIs liefern, die Implementierung und das TCK können in einem späteren Meilenstein geliefert werden. Als einzige komplett neue Spezifikation soll *Jakarta Data* aufgenommen werden, MVC (3.0) und NoSQL (1.0) bleiben nach Abstimmung unabhängige Spezifikationen außerhalb der Plattform.

Viele der Neuerungen basieren auf den Features von Java SE 21 – *API Source* und *Binary Level* werden entsprechend auf SE 21 gehoben und die TCKs laufen dann natürlich darauf. Was die Kunden dann zu Java-SE-Upgrades zwingen wird, bringt auf der anderen Seite viele neue Features für die SW-Entwicklung, die sich vielfach in den Spezifikationen wiederfinden (zum Beispiel ein neues „virtual" Attribut für die *ManagedExecutorDefinition*-Annotation). Und der Sprung ist direkt von SE 11 auf 21, das LTS-Release SE 17 wird übersprungen.

Ein paar Dinge werden aus Jakarta rausfliegen, etwa *ManagedBeans* (endgültig ersetzt durch CDI), alles, was den abgekündigten *SE Security Manager* betrifft, und eine Reihe in EE 10 optionaler Spezifikationen (wie der ganze „SOAP und XML-Kram").

Referenzen:

[1] *https://www.atomicjar.com/2023/08/announcing-testcontainers-desktop-free-application/*
[2] *https://spring.io/blog/2023/08/31/this-is-the-beginning-of-the-end-of-the-n-1-problem-introducing-single-query*
[3] *https://blogs.eclipse.org/post/ivar-grimstad/jcp-25-year-anniversary*
[4] *https://www.heise.de/blog/Java-21-ist-eines-der-spannendsten-Releases-der-letzten-Jahre-9309203.html*
[5] *https://vived.substack.com/p/project-babylon-chance-for-linq-and*
[6] *https://github.com/eclipse-ee4j/cargotracker/releases/tag/v3.0*
[7] *https://adoptium.net/de/blog/2023/09/temurin21-delay/*
[8] *https://adoptium.net/de/blog/2023/09/Reproducible-Comparison-Builds/*
[9] *https://github.com/jakartaee/platform/issues/762*
[10] *https://jakarta.ee/compatibility/certification/10/*
[11] *https://openjdk.org/jeps/0*

Andreas Badelt
stellv. Leiter der DOAG Cloud Native Community
andreas.badelt@doag.org

Andreas Badelt ist seit 2001 ehrenamtlich im DOAG e.V. aktiv und hat dort inzwischen seine Heimat in der Cloud Native Community gefunden, wobei ihn das Java-Ökosystem bis heute fasziniert. Beruflich hat er von Ende des vorigen Jahrtausends an als Entwickler und später auch Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet. Seit 2016 ist er als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

MARKUS' ECLIPSE™-Corner

Heute müssen wir mal über Wunsch und Wirklichkeit sprechen.

Open Source, und damit auch das Java-Universum, funktioniert leider nicht so, wie wir alle denken. Auch wenn es sich nominal gut anhört, was Andreas im Java-Tagebuch zusammengefasst hat (an dieser Stelle mal wirklich einen ausgiebigen Applaus dafür, dass er dies seit vielen Jahren ehrenamtlich für uns alle macht), die Wahrheit sieht halt doch ein bisschen anders aus und muss differenzierter betrachtet werden, als es im absichtlich subjektiv-neutral formulierten Tagebuch möglich ist. Ja, bei JavaSE gibt Oracle (und fast nur Oracle) tatsächlich richtig Gas (die Beiträge der anderen OpenJDK-Mitglieder sind wirklich sehr lobenswert und technisch beeindruckend, in der Summe aber leider marginal), sodass mit JDK 23 sehr wahrscheinlich ein Stand erreicht sein wird, der Python, Rust, Go und anderen prominenten Konkurrenten möglicherweise (hoffentlich? endlich?) das Fürchten lehren wird. Wer sich etwas intensiver mit den einzelnen Änderungen beschäftigt (nicht nur mit den „großen" JEPs), sieht, dass „unter der Haube" massive Umbauten der JRE-interna im Gange sind, von denen wir alle einen spürbaren Nutzen haben werden. Ich bin sicher, dass der Trend weg von Java hin zu den obengenannten Sprachen damit deutlich gebremst wird (und ich hoffe, dass er sich in Teilen sogar umkehrt).

Dass Jakarta EE 11 nominell auf Java 21 aufsetzen soll, ist großartig, da der gesamte Sprach- und API-Umfang (und die unter der Haube werkelnden Optimierungen wie beispielsweise das NIO-Offloading) somit ohne großen Aufwand allen selbst-zertifizierten Implementierungen zur Verfügung steht. (Ja, jeder Hersteller darf einfach von sich behaupten, kompatibel zu Jakarta EE zu sein; eine Prüfungskommission gibt es nicht. Man darf Teile der TCK-Tests einfach abschalten, niemand zwingt einen, wirklich jedes Detail fehlerfrei zu beherrschen – schon gar nicht das eigentliche Highlight, das Java-SE-Bootstrap-API – das auch prompt kein einziges, zertifiziertes Produkt unterstützt).

Damit sind wir (oder ich) wieder beim Meckern (oder, wie es Jacob Beautemps ausdrückt: beim „großen aber"). Nachdem Oracle 2017 nach langem Nichtstun (Ed Burns nannte es mal „the long silence") endlich eingesehen hatte, Java EE lieber an die Eclipse Foundation zu übergeben (und somit gerne die Pflegekosten, aber eben nötigenfalls auch Bestimmungsgewalt mit den Konkurrenten Red Hat und IBM zu teilen), statt die API-Sammlung einen langsamen, aber sicheren Tod sterben zu lassen, kam die Eclipse Foundation (ein hauptsächlich durch vorgenannte Protagonisten finanzierter Unternehmensclub) in den Besitz ebenjener Altlast. Diese Stiftung, deren maßgeblicher Zweck es bis dahin augenscheinlich war (und es meiner Meinung nach auch hauptsächlich weiterhin ist), die Wartungskosten für bisherige Closed-Source-Produkte ihrer Mitglieder auf die Schultern vieler Beitragszahler zu verteilen, bezahlt jedoch trotz Einnahmen von etwa 10 Mio. € jährlich *[1]* keinen einzigen Entwickler. Sie investiert lediglich in den Betrieb einer technischen Infrastruktur sowie in Lobbyismus und Marketing mit dem Zweck, weitere Mitglieder zu gewinnen, die dann wiederum Mitgliedsgebühr bezahlen, oder die sich bereit erklären, Entwickler (zumindest in Teilzeit) zur Pflege der Altlast abzustellen. Dadurch wird klar, dass die Verheißungen, die Andreas im Java-Tagebuch prophezeit, nur dann eintreten werden, wenn Dritte sie durchführen – und somit auch finanzieren.

Dem iJUG e. V. als oberstem Interessensverband der Java-Programmierer im deutschsprachigen Raum war bereits 2017 völlig klar, dass Entwickler herangeschafft werden müssen, um Jakarta EE zu einem Erfolgsmodell zu machen. Jan Westerkamp und ich hatten uns daher im Juli 2018 informell mit Mike Milinkovic im Biergarten in der Stuttgarter Innenstadt getroffen, um auszuhandeln, worüber wir direkt danach auf der ordentlichen Mitgliederversammlung des iJUG abstimmten: Der iJUG ging eine Partnerschaft mit der Eclipse Foundation *[2]* in der Hoffnung ein, Java EE zu retten und zum Vorteil der deutschsprachigen Java-Nutzer zu lenken. Seither schlugen alle unsere Versuche aber leider weitgehend fehl, obwohl wir auf allen Kanälen (Java aktuell, Newsletter, Foren, Konferenzen, JUG-Besuche etc.) das Thema befeuerten. Weder konnten wir Entwickler oder Unternehmen davon überzeugen, sich nachhaltig an Jakarta EE zu beteiligen, noch konnten wir nennenswerten Nachwuchs über unser Stipendienprogramm gewinnen – weder als Mentoren noch als Stipendiaten. Wir stehen also bei Jakarta EE 11 dort, wo wir mit Jakarta EE 8 begonnen hatten: Oracle, Red Hat und IBM leisten (geplant) immer weniger, eine Kompensation durch neue Open-Source-Kontributoren findet kaum statt. Im Gegensatz zu beispielsweise Adoptium, wo die Community sich wirklich aktiv einbringt, haben wir bei Jakarta EE also wenig erreicht. Im Gegenteil, wie Andreas berichtet, hat die Mehrheit der Unternehmen ja dafür gestimmt, die berechtigten Sicherheitsbedenken des iJUG einfach zu ignorieren. Helmut Kohl nannte seinerzeit diese Strategie „aussitzen": Irgendwann wird es iJUG-Botschafter Jan Westerkamp leid sein, gegen Windmühlen zu kämpfen, und dann wird „dieser iJUG" hoffentlich Ruhe geben. Dass bislang jeder Termin, den die Jakarta EE Working Group bislang veröffentlicht hat, um viele Monate verschoben wurde, ist ja nun längst kein Geheimnis mehr. Was im Tagebuch als Terminplan steht ist somit also nur: Marketing der Eclipse Foundation. Die Realität wird sein – und da gehe ich gerne eine Wette mit euch ein –, dass es viel später, viel weniger und viel schlechter sein wird. Glaubt ihr nicht? Wartet ab oder vergleicht doch selbst einmal die Pläne der vergangenen Releases (EE 9, 9.1 und 10) mit der zurückliegenden Realität.

Auch wenn es regelmäßig eine Community-Befragung zur Wichtigkeit von neuen Features gibt: Jakarta EE ist eben kein Wunschkonzert. Irgendjemand muss die Arbeit letztendlich erledigen und die wenigsten scheinen dies ehrenamtlich machen zu wollen. SO sieht die Realität aus. Leider. Die Frage, die ich mir in Vertretung des iJUG stelle, ist: Was können wir also noch tun, um unsere Leser zu animieren, sich (persönlich, als JUG oder als Unternehmen) an Jakarta EE zu beteiligen? Schreibt mir gerne mal eure Vorschläge. Ich bin gespannt!

P. S.: Ein Nachtrag noch zum Thema China, das Andreas im Tagebuch angerissen hat. Lange Zeit hat die Bundesrepublik Deutschland das Land China als Partner und Chance betrachtet, lange Zeit war die Beteiligung Chinas an der strategischen Technologie Java/Jakarta eher gering. Vor einiger Zeit hat China begonnen, seine Machtinteressen weltweit auszuspielen *[3]*, seither betrachtet die Bundesrepublik China als „systemischen Rivalen" *[4]* (und somit als Risiko). Und nun also „engagiert" sich China – „zufällig" – bei Jakarta EE (zur Klarstellung: China hat sich de facto Stimmen gekauft, nicht etwa Entwickler finanziert). Ob die Anführungszeichen um das Wort „zufällig" korrekt gesetzt sind, wird die Zukunft zeigen.

Referenzen:

[1] *https://www.eclipse.org/org/foundation/reports/annual_report.php* – Finanzbericht der Eclipse Foundation

[2] *https://www.ijug.eu/de/presse/#tab-24733-0* – Pressemeldung des iJUG e. V. zum Beitritt in die Eclipse Foundation

[3] h*ttps://www.bundesregierung.de/breg-de/service/gesetzesvorhaben/china-strategie-2202212* – Pressemeldung zur China-Strategie der Bundesregierung

[4] *https://www.auswaertiges-amt.de/blob/2608578/810fdade376b1467f20bdb697b2acd58/china-strategie-data.pdf* – China-Strategie der Bundesregierung

Markus Karg

markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. JAX-RS hat der Sprecher der Java User Group Goldstadt von Anfang an mitgestaltet, zunächst als freier Contributor, seit JAX-RS 2.0 als Mitglied der Expert Groups JSR 339 und JSR 370.

Java 21 – Die nächste Version mit LTS-Unterstützung

Falk Sippach, embarc Software Consulting GmbH

Nur zwei Jahre nach dem OpenJDK 17, dem letzten Release mit Long Term Support, wurde im September 2023 bereits Version 21 veröffentlicht. Und diesmal sind eine ganze Menge von Funktionen auf den Release Train aufgesprungen (siehe Abbildung 1). Mit dabei sind auch bahnbrechende Features, die nach mehrjähriger Arbeit finalisiert wurden. Und es gibt ein paar Überraschungen, die die tägliche Entwicklerarbeit erleichtern werden. Darum werfen wir nun einen Blick auf die Neuerungen und wagen am Ende auch einen Ausblick in die Zukunft.


1
VA

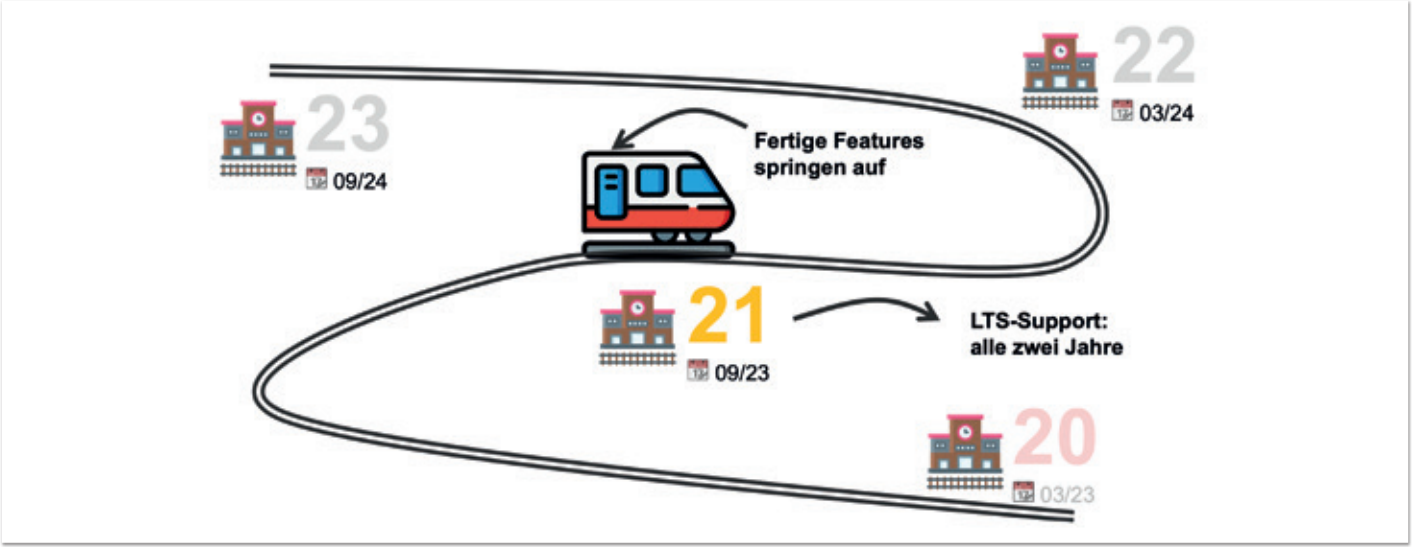


Abbildung 1: Halbjährlicher Release-Train (© Falk Sippach)

Für einen ersten Überblick ist wie immer die Projektseite des OpenJDK *[1]* ein guter Startpunkt. Dort sind diesmal immerhin 15 JEPs (JDK Enhancement Proposals) gelistet, so viele wie lange nicht:

- 430: String Templates (Preview)
- 431: Sequenced Collections
- 439: Generational ZGC
- 440: Record Patterns
- 441: Pattern Matching for switch
- 442: Foreign Function & Memory API (Third Preview)
- 443: Unnamed Patterns and Variables (Preview)
- 444: Virtual Threads
- 445: Unnamed Classes and Instance Main Methods (Preview)
- 446: Scoped Values (Preview)
- 448: Vector API (Sixth Incubator)
- 449: Deprecate the Windows 32-bit x86 Port for Removal
- 451: Prepare to Disallow the Dynamic Loading of Agents
- 452: Key Encapsulation Mechanism API
- 453: Structured Concurrency (Preview)

Einige der Themen sind schon einige Zeit dabei, wie zum Beispiel das *Pattern Matching for switch*, das gemeinsam mit den *Record Patterns* nun finalisiert wurde. Somit kann man *Pattern Matching* jetzt auch endlich produktiv einsetzen. In Zukunft werden weitere Pattern-Typen hinzukommen. Den Anfang machen diesmal die *Unnamed Patterns*, die gemeinsam mit den *Unnamed Variables* als Preview erschienen sind. Ebenfalls abgeschlossen, und damit ab sofort produktiv einsetzbar, sind die *Virtual Threads* aus *Project Loom*. In ihrem Umfeld sind auch neue Previews zur *Structured Concurrency* und *Scoped Values* erschienen, für die die Macher des JDK bis zur Finalisierung noch weiteres Feedback einsammeln möchten. Auch schon sehr lange und weiterhin als Previews, beziehungsweise Inkubator, dabei sind das *Foreign Function & Memory API* sowie das *Vector API*. Ersteres wird aller Voraussicht nach mit Java 22 finalisiert werden. Mit dem *Vector API* wird der Grundstock für effizientere Berechnungen basierend auf modernen Prozessorarchitekturen gelegt. Das wird uns auch noch in den nächsten Releases begleiten und ist aus Sicht der normalen Java-Entwickler nicht so relevant. Viel interessanter sind hingegen die neu eingeführten Previews zu den *String Templates* und den *Unnamed Classes & Instance Main Methods*. Mit den *Sequenced Collections* gibt es zudem wieder eine interessante Änderung an der Klassenbibliothek.

Übrigens, der Funktionsumfang des OpenJDK 21 stand bereits ein Vierteljahr vor dem eigentlichen Veröffentlichungstermin fest. Der

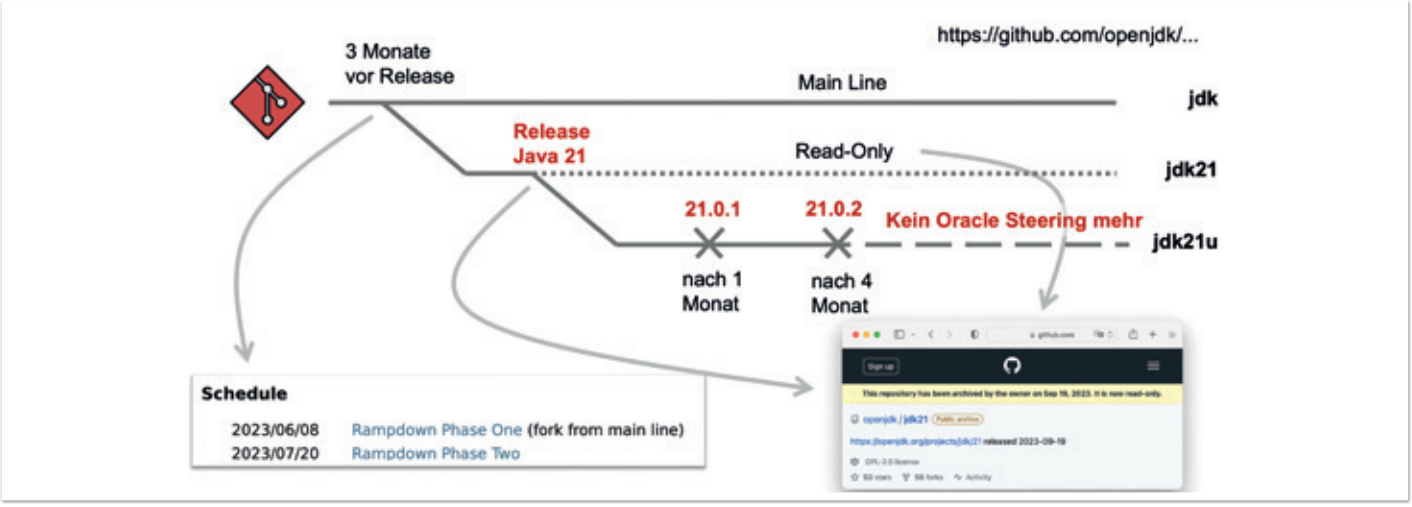


Abbildung 2: JDK-Release-Prozess (© Falk Sippach)

Release-Prozess ist im JEP 3 beschrieben *(siehe Abbildung 2)*. Demnach wird immer drei Monate vorher in der sogenannten *Rampdown Phase One* ein neues Repo („jdk21") von der *Main Line* abgezweigt (Fork), das zum Release-Termin auf read-only gesetzt wird. Gleichzeitig wird dann wiederum ein weiteres Repo abgespalten („jdk21u"), das noch ein halbes Jahr von Oracle für Updates genutzt wird. Nach einem und nach vier Monaten erscheinen die zwei offiziellen Patch-Versionen 21.0.1 und 21.0.2. Mit der Veröffentlichung der nächsten OpenJDK-Version, sechs Monate nach dem aktuellen Major-Release, wird dieses Update-Repo dann von Oracle nicht mehr weiter gepflegt. Gegebenenfalls springt dann einer der anderen OpenJDK-Distributions-Anbieter ein, verwaltet den Update-Branch weiter und pflegt Bug- und Security-Fixes von den verschiedenen Zulieferern ein. Davon können dann alle Anbieter die weiteren Patch-Releases bauen. Bei den letzten LTS-Versionen hat sich Red Hat dieser Sache angenommen.

Virtuelle Threads revolutionieren nebenläufige Programmierung

In Java gibt es bereits eine lange Geschichte zu *Concurrency* *(siehe Abbildung 3)*. Schon seit Java 1.3 werden native Threads eingesetzt, die direkt auf Plattform-Threads im Betriebssystem abgebildet werden. Diese sind in der Erzeugung teuer und benötigen vergleichsweise viel Speicherplatz. Das skaliert nur bis zu einer bestimmten Anzahl gleichzeitiger Threads (je nach Hardware nur einige 1.000), was wiederum ein Problem für moderne Unternehmensanwendungen ist. Denn diese sind heute häufig als Webanwendungen oder als verteilte Systeme implementiert, wobei jeder Request an einen Plattform-Thread gebunden wird. Dieser wird gegebenenfalls eine längere Zeit blockiert, weil auf externe Ressourcen (zum Beispiel auf eine Datenbank) oder langlaufende Berechnungen gewartet werden muss. Zu viele Anfragen führen dann schnell zu einer Überlastung zum Beispiel in Form eines *OutOfMemory*-Errors. Als Ausweg bleibt nur die horizontale Skalierung der Serveranwendung und das Vorschalten eines *Load Balancers*. Tatsächlich langweilen sich die CPUs der einzelnen Instanzen, weil die meiste Zeit gewartet werden muss. Könnten mehr Threads gleichzeitig laufen, ließe sich die CPU effizienter auslasten.

Virtuelle Threads versprechen Abhilfe. Aufgrund ihrer Leichtgewichtigkeit können Millionen von ihnen erzeugt werden. Aber Millionen gleichzeitiger Threads sind nicht das Ziel des *Project Loom*. Vielmehr geht es darum, mit dem klassischen *Thread API* einen vereinfachten („naiven") Umgang bei einer großen Menge von zweitweise durch Netzwerkanfragen blockierten Aufgaben zu ermöglichen. Der Quellcode ist dabei leicht les- sowie gut wartbar und lässt sich wie sequenzieller Code einfach debuggen. Blockiert ein virtueller Thread, wird er geparkt und der nächste aufgerufen. Da die *Virtual Threads* wenig Speicher verbrauchen und auch ihre Erzeugung sehr günstig ist, kann einfach jede Aufgabe (zum Beispiel *Web-Request*, aber auch nebenläufige Tasks) in einem neuen Thread gestartet werden. Das Pooling von Threads kann komplett entfallen. Auch auf alternative, schlecht les- und wartbare Ansätze wie die reaktive Programmierung oder die Callback-Hölle durch asynchrone Verarbeitung kann somit verzichtet werden.

Übrigens laufen *Virtual Threads* nicht schneller oder effizienter, denn unter der Haube verwenden sie letztlich auch wieder die Plattform-Threads. Es gibt einen Pool von *Carrier Threads*, denen virtuelle Threads vorübergehend zugewiesen werden. Sobald der virtuelle Thread auf eine blockierende Operation stößt, wird er vom *Carrier Thread* genommen, der dann einen anderen virtuellen Thread (einen neuen oder einen zuvor blockierten) übernehmen kann *(siehe Abbildung 4)*.

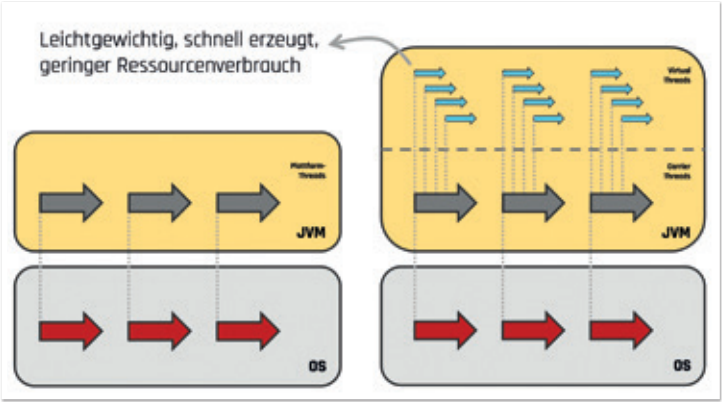


Abbildung 4: Plattform- vs. Virtual Threads

Die Virtual Threads wurden transparent in die bestehende Klassenhierarchie des JDK integriert. Für den Entwickler verhalten sie sich

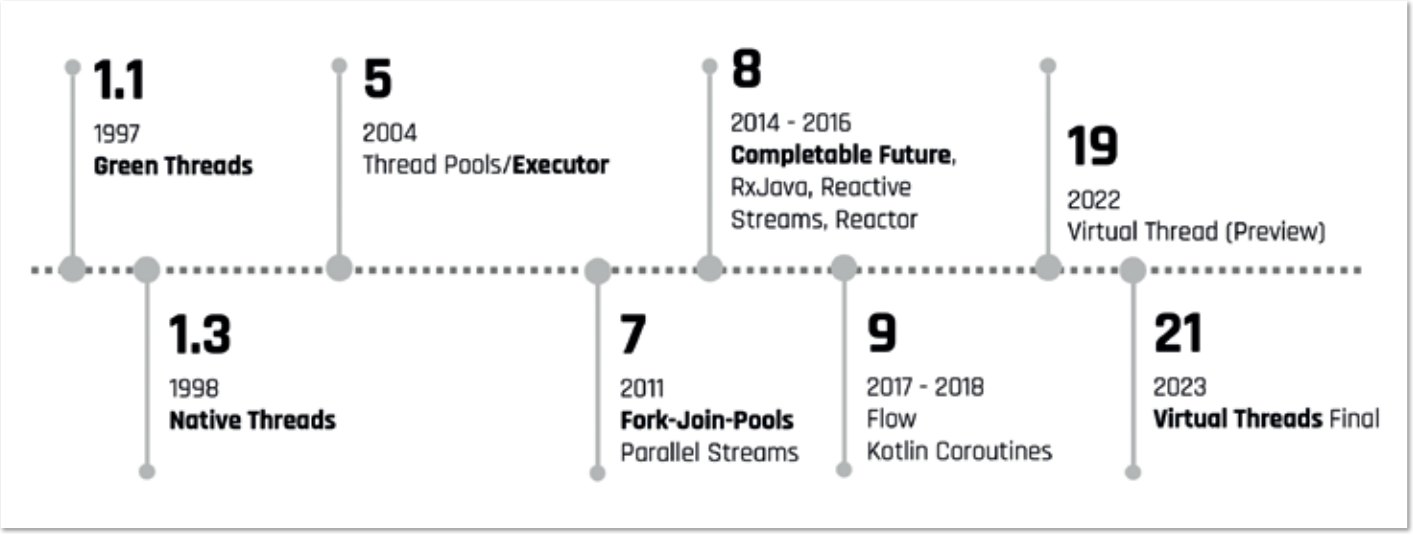


Abbildung 3: Geschichte von Threads in Java

nahezu wie die klassischen Plattform-Threads (mit einigen Ausnahmen, zum Beispiel in `synchronized`-Blöcken können sie Plattform-Threads blockieren). Die neuen Factory-Methoden erstellen entweder eine Instanz der Klasse `java.lang.Thread` (Plattform-Thread) oder von `java.lang.VirtualThread` *(siehe Listing 1)*. Sie erzeugen jeweils einen *ThreadBuilder*, der sich noch parametrieren lässt und über den man sich um die Konfiguration der Threads kümmern kann. Durch diese Abstraktion ist es einfach, auf die virtuelle Variante zu wechseln. Bestehende Anwendungen können auch allein durch den Wechsel auf Java 21 profitieren, sofern die verwendeten Frameworks (etwa *Spring Boot* oder *Quarkus*) bereits virtuelle Threads unterstützen. Entwickler müssen dann nicht selbst mit virtuellen Threads umgehen, sondern können das Framework konfigurieren, sodass beispielsweise für Web-Requests virtuelle Threads zum Einsatz kommen.

```
Thread.Builder builder = Thread.ofVirtual();
// Thread.Builder threadBuilder = Thread.ofPlatform();

builder.start(() -> {
  // im Thread auszuführender Code
  System.out.println(  Thread.currentThread().isVirtual());
});
```

Listing 1: Plattform- und virtuelle Threads erzeugen

Blockierende Operationen halten den ausführenden Thread nun also nicht mehr unnötig auf. Mit nur einem kleinen Pool von *Carrier Threads* ist eine viel höhere Zahl von Requests parallel verarbeitbar. Ob der eigene Code in einem virtuellen Thread läuft, lässt sich mit `Thread.currentThread().isVirtual()` herausfinden.

An den Virtual Threads arbeitet das Project-Loom-Team schon lange: Es hat sie im OpenJDK 19 erstmals als Preview eingeführt. Mit der Finalisierung in Java 21 sind *Virtual Threads* nun eine der spektakulärsten Neuerungen der letzten Jahre. Sie stehen in einer Reihe mit *Generics* (Java 5), *Lambdas/Stream API* (Java 8) und dem *Platform Module System* (Java 9). Es braucht aber noch ein paar Jahre, bis auch die Framework- und Serverhersteller ihre Produkte auf Virtual Threads umgestellt haben und Anwendern die Wahl lassen.

Erneut dabei: Structured Concurrency

Virtual Threads können auch sehr gut mit einem neuen API verwendet werden: *Structured Concurrency*. Mit JEP 453 ist es die dritte Wiedervorlage eines Previews. Beim Bearbeiten mehrerer paralleler Teilaufgaben erlaubt *Structured Concurrency* die Implementierung auf eine besonders les- und wartbare Art und Weise. Bisher wurden hierfür *Parallel Streams* oder der *ExecutorService* eingesetzt. Letzterer ist sehr mächtig, macht aber selbst einfache Umsetzungen kompliziert und fehleranfällig. Es ist beispielsweise schwer zu erkennen, wenn eine der Teilaufgaben einen Fehler produziert und dann alle anderen Teilaufgaben sofort sauber abgebrochen werden sollen. Läuft ein Task sehr lange, bekommt man erst spät mit, wenn andere Aufgaben Fehler produziert haben. Auch das Debuggen ist nicht einfach, da in den *Thread-Dumps* die Tasks nicht den jeweiligen Threads aus dem Pool zuzuordnen sind.

Bei der *Structured Concurrency* *(siehe Listing 2)* ersetzen Entwickler den *ExecutorService* durch einen `StructuredTaskScope`. Dabei sind verschiedene Strategien (`ShutdownOnFailure` in *Listing 2*) auswählbar. `scope.join()` blockiert, bis alle Tasks erfolgreich erledigt sind. Schlägt einer fehl oder wird abgebrochen, beendet das Programm auch die beiden anderen. `throwIfFailed()` gibt den Fehler weiter und überspringt das Ausgeben der Ergebnisse. Neben `ShutdownOnFailure` (alle Tasks liefern im Erfolgsfall ein Ergebnis zurück) wird noch `ShutdownOnSuccess` mitgeliefert, der alle noch laufenden Tasks abbricht, sobald der erste fertig ist. Diese Strategie eignet sich gut, wenn mehrere Prozesse parallel gestartet werden sollen und die schnellste Antwort gewinnen soll. Hat man darüberhinausgehende Anforderungen, kann man eigene Implementierungen von `StructuredTaskScope` umsetzen.

```
try (var scope = new StructuredTaskScope.ShutdownOn-
Failure()) {
  Future<String> task1 = scope.fork(() -> { … }
  Future<String> task2 = scope.fork(() -> { … }
  Future<String> task3 = scope.fork(() -> { … }

  scope.join();
  scope.throwIfFailed();

  System.out.println(task1.get());
  System.out.println(task2.get());
  System.out.println(task3.get());
}
```

Listing 2: Structured Concurrency

Der neue Ansatz bringt einige Vorteile: Zum einen bilden Tasks und Subtasks im Code eine abgeschlossene zusammengehörige Einheit. Es können sowohl Plattform als auch virtuelle Threads zum Einsatz kommen. Bei Fehlern bricht der Code noch laufende Subtasks ab. Zudem sind bei Fehlersituationen die Informationen besser, weil die Aufrufhierarchie sowohl in der Codestruktur als auch im *Stacktrace* der *Exception* sichtbar ist. Wegen des Preview-Status sind aber beim Kompilieren und Ausführen bestimmte Schalter zu aktivieren *(siehe Listing 3)*.

```
$ javac --enable-preview StructuredConcurrencyMain.java
$ java --enable-preview StructuredConcurrencyMain
```

Listing 3: Kompilieren und Ausführen von Preview-Funktionen

Scoped Values: Alternative zu ThreadLocals

Im Umfeld der *Virtual Threads* wurde in Java 20 eine Alternative zu den *ThreadLocal*-Variablen vorgestellt: *Scoped Values*. Sie befinden sich weiterhin im Preview-Status und erlauben es, einen temporären Wert für eine begrenzte Zeit zu speichern, wobei nur der Thread, für den der Wert abgelegt wurde, ihn wieder lesen kann.

Scoped Values legt man als öffentliche statische Felder an, sodass sie aus beliebigen, tiefer im Aufruf-Stack befindlichen, Methoden aufrufbar sind. Die Entwickler sparen sich so das Durchschleifen der Informationen über Methodenparameter. Verwenden mehrere Threads dasselbe `ScopedValue`-Feld, wird es je nach ausgeführtem Thread unterschiedliche Werte enthalten. Wer mit `ThreadLocals` vertraut ist, kennt das Konzept.

```
public class ScopedValues {

  public final static ScopedValue<SomeUser> CURRENT_USER = ScopedValue.newInstance();

  public static void main(String[] args) throws MalformedURLException, URISyntaxException {
     SomeController someController = new SomeController(new SomeService(new SomeRepository()));
 someController.someControllerAction(HttpRequest.newBuilder().uri(new URL("http://example.com").toURI()).build());
  }

  static class SomeController {
     final SomeService someService;

     SomeController(SomeService someService) {
        this.someService = someService;
     }

     public void someControllerAction(HttpRequest request) {
        SomeUser user = authenticate(request);
        ScopedValue.where(CURRENT_USER, user)
           .run(() -> someService.processService());
        someService.processService();
     }

     [..]
  }

  static class SomeService {
     final SomeRepository someRepository;

     SomeService(SomeRepository someRepository) {
        this.someRepository = someRepository;
     }

     void processService() {
        System.out.println(CURRENT_USER.orElseThrow(() -> new RuntimeException("no valid user")));
     }
  }
  [..]
}
```

Listing 4: Scoped Value

Das Beispiel in *Listing 4* extrahiert aus einem Web-Request Informationen zum angemeldeten Benutzer. Auf diese Informationen muss in der weiteren Aufrufkette – im Service oder im Repository – zugegriffen werden. Mit `ScopedValue.where()` wird das User-Objekt gesetzt und dann ein *Runnable* übergeben, für dessen Aufrufdauer der *Scoped Value* gültig sein soll. Alternativ lässt sich mit der `call()`-Methode eine Instanz von *Callable* übergeben, um auch Rückgabewerte auszuwerten. Der Versuch, den Service außerhalb des `ScopedValue`-Kontextes auszuführen, führt zu einer *Exception*, weil bei diesem Aufruf kein User-Objekt hinterlegt ist.

Das Auslesen von `ScopedValue` erfolgt über den `get()`-Aufruf. Fehlt ein Wert, kann man mit einem *Fallback* (`orElse()`) oder dem Werfen einer *Exception* (`orElseThrow()`) reagieren. Auch hier sind beim Kompilieren und Ausführen bestimmte Schalter zu aktivieren.

Die Klasse `ScopedValue` ist *immutable* und bietet daher keine `set()`-Methode an. Der Code ist besser les- und wartbar, weil es keine Zustandsänderungen am bestehenden Objekt geben kann. Muss für einen bestimmten Codeabschnitt (Aufruf einer weiteren Methode in der Kette) ein anderer Wert sichtbar sein, kann ein Rebinding des Wertes erfolgen, etwa indem man ihn auf null setzt *(siehe Listing 5)*.

```
ScopedValue.where(CURRENT_USER, null)
     .run(() -> someRepository.getSomeData());
```

Listing 5: Scoped Value Rebinding

Sobald der begrenzte Codeabschnitt beendet ist, wird der ursprüngliche Wert wieder sichtbar.

Scoped Values funktionieren auch mit *Structured Concurrency*. Die Sichtbarkeit wird an die über `StructuredTaskScope` erzeugten Kindprozesse vererbt. Somit können alle per `fork()` abgezweigten Kind-Threads ebenfalls auf die im *Scoped Value* befindlichen User-Informationen zugreifen. *Scoped Values* stehen wie `ThreadLocals` sowohl für Plattform- als auch für Virtual Threads zur Verfügung. Die Vorteile von *Scoped Values* sind das automatische Aufräumen der Inhalte, sobald der *Runnable-/Callable*-Prozess beendet ist (somit keine *Memory Leaks*) und die Unveränderlichkeit der *Scoped Values*, was die Verständlichkeit und Lesbarkeit erhöht. Die bei *Structured Concurrency* erzeugten Kindprozesse haben damit auch Zugriff auf den einen, unveränderbaren Wert, die Informationen müssen nicht wie bei `InheritedThreadLocals` kopiert werden (was zu höherem Speicherverbrauch führen kann).

Pattern Matching ist jetzt final

Das *Pattern Matching* wird bereits seit einigen Jahren im Rahmen von Project Amber *[2]* Stück für Stück eingeführt. Dazu waren diverse Änderungen in der Sprache selbst notwendig. Los ging es mit den *Switch Expressions* bereits im JDK 12. Ab Version 14 folgten die *Type Patterns* (*Pattern Matching for instanceof*) und *Records*. *Sealed Classes* wurden im JDK 15 eingeführt. Und mit 17 kam erstmals Pattern *Matching for switch* als ein Preview hinzu. Dieses Feature wurde mit Java 21 finalisiert und damit kann die erste Ausbaustufe des *Pattern Matching* nun produktiv eingesetzt werden. Ebenfalls finalisiert wurden

die im OpenJDK 19 eingeführten *Deconstruction (Record) Patterns*. In Zukunft werden weitere Mustertypen erwartet, wie zum Beispiel *Array-, Map-, POJO-* oder *Factory-Method-Patterns*. In Java 21 wurde mit dem JEP 443 *Unnamed Patterns and Variables* das Platzhalter-Muster eingeführt, wenn auch zunächst noch als Preview. Bei den *Record Patterns* gab es nochmal eine entscheidende Änderung. In Java 20 konnten auch in der `foreach`-Schleife die Elemente einer Collection direkt dekonstruiert werden `(for(Point(x, y)) : points {})`. Das wurde zunächst wieder entfernt, soll aber in späteren Releases in einem eigenen JEP erneut auftauchen.

Beim *Pattern Matching* geht es darum, bestehende Strukturen mit Mustern abzugleichen, um komplizierte Fallunterscheidungen effizient und wartbar implementieren zu können. Ein Pattern ist dabei eine Kombination aus einem Prädikat, das auf die Zielstruktur passt, und einer Menge von Variablen innerhalb dieses Musters. Diesen Variablen werden bei passenden Treffern die entsprechenden Inhalte zugewiesen und damit extrahiert. Die Intention des *Pattern Matching* ist die Destrukturierung von Datenobjekten, also das Aufspalten in die Bestandteile und das Zuweisen in einzelne Variablen zur weiteren Bearbeitung.

Das *Pattern Matching* kommt aus der funktionalen Programmierung und löst Fallunterscheidungen auf eine andere Art, als es in der objektorientierten Programmierung üblich ist. Brian Goetz beschreibt in seinem Artikel zu datenorientierter Programmierung *[3]* die Vorteile und Hintergründe. In Java funktioniert das über den um neue Funktionen erweiterten Switch. Gegenüber dem klassischen Switch-Statement mit den konstanten Patterns (primitive Ganzzahlen, Strings oder Enums) kann in den `case`-Zweigen nun auch auf beliebige Typ-Deklarationen (Type Patterns) oder auf Daten-Tuple (Records) geprüft werden. Bei den *Record Patterns* werden die Properties des Objekts dann für die weitere Bearbeitung direkt extrahiert und in Variablen gespeichert. Zudem gibt es eine neue, kompakte und vor allem weniger fehleranfällige Syntax für die Switch-Anweisung (Arrow-Notation). Der Switch kann als Expression außerdem direkt das Ergebnis zurückliefern.

Durch die parallele Einführung der *Sealed Classes* und der Records lassen sich in Java nun einfach algebraische Datentypen erstellen, die wiederum in den Case-Zweigen der Switch-Expression zur Fallunterscheidung herangezogen werden können. *Listing 6* zeigt die Implementierung einer verketteten Liste. Sie besteht aus zwei Teilen: Einem Listenelement (Element), das den Wert enthält und auf das nächste Element zeigt. Das letzte Element mit einem Wert zeigt immer auf eine leere Liste (Empty).

Bei diesem datenorientierten Ansatz sind die Operationen, die auf der Listenstruktur arbeiten, als separate statische Methoden definiert und können dort mithilfe der Fallunterentscheidungen die verschiedenen Optionen behandeln. In der `contains`-Operation wird in einer Switch-Expression die übergebene Liste untersucht. Ist es eine leere Liste, dann gibt sie direkt `false` zurück (erster `case`-Zweig). Andernfalls, wenn es ein Element ist, wird geprüft, ob der aktuelle Wert dem übergebenen Parameter `value` entspricht (zweiter `case`-Zweig). Dafür kommt die `when`-Clause (früher *Guarded Patterns* genannt) zum Einsatz. Sollten die Werte nicht gleich sein, wird im dritten `case`-Zweig die `contains`-Methode rekursiv aufgerufen und das nächste Element der Liste übergeben.

Die `case`-Zweige 2 und 3 verwenden *Record Patterns*, wobei die Properties der zu matchenden Record-Instanz direkt den Variablen `v` und `tail` zugewiesen werden. Dadurch spart man sich das nachträgliche Aufrufen der `getter`-Methoden und kann direkt mit den passenden Werten weiterarbeiten. Im `case`-Zweig 1 wird ein Type Pattern (*Pattern Matching for instanceof*) verwendet, also nur eine Typprüfung gemacht. Aber auch hier erfolgt implizit ein Cast und die Zuweisung zu einer Variablen.

```
public sealed interface LinkedList<T>
  permits LinkedList.Element, LinkedList.Empty {

  record Element<T>(T value, LinkedList<T> next)
     implements LinkedList<T> {}

  final class Empty<T> implements LinkedList<T> {}

  static <T> LinkedList<T> of(T... values) {
     if (values.length == 0){
        return new LinkedList.Empty<>();
     }

     LinkedList<T> current = new LinkedList.Empty<>();

     for (int i = values.length - 1; i >= 0; i--) {
        current =
           new LinkedList.Element<>(values[i], current);
     }
     return current;
  }

  static <T> boolean contains(
     T value, LinkedList<T> list) {

     return switch (list) {
        case Empty empty -> false;
        case Element<T>(T v, var tail)
           when Objects.equals(v, value) -> true;
        case Element<T>(T v, var tail) ->
           contains(value, tail);
     };
  }

  public static void main(String[] args) {
     LinkedList<Integer> list = of(1, 2, 3);
     System.out.println(contains(5, list)); // false
     System.out.println(contains(1, list)); // true
  }
}
```

Listing 6: Pattern Matching mit algebraischen Datentypen

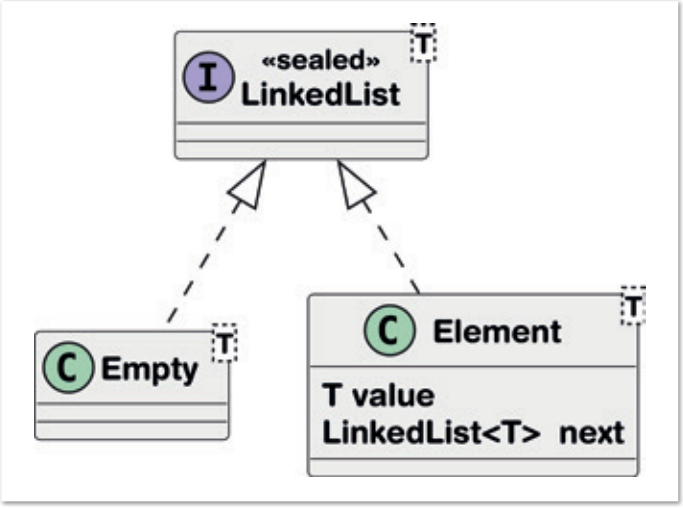


Abbildung 4: Klassendiagramm des algebraischen Datentyps

In einer *Switch Expression* wird immer eine sogenannte *Exhaustiveness*-Prüfung durchgeführt. Wenn der Compiler nicht sicherstellen kann, dass alle möglichen Fälle abgedeckt sind, muss es einen `default`-Zweig geben. Das entfällt hier allerdings, weil mit *Sealed Classes* gearbeitet wird. Dabei wird die Typhierarchie, bestehend aus einem *Interface (LinkedList)* und genau zwei zugelassenen Subklassen (Empty und Element) versiegelt *(siehe Abbildung 4)*. Dadurch werden im Switch alle Fälle abgedeckt und der `default` kann hier weggelassen werden. Vorteil dieses Vorgehens ist, bei Erweiterung der Klassenhierarchie um eine neue Subklasse, schlägt der Compiler sofort bei allen Switch-Expressions fehl, bis dort auch der fehlende Zweig hinzugefügt wird. In den alten Switch-Statements wurden fehlende Fälle einfach ignoriert, was zu diffusen Fehlern führen konnte.

Die moderne Syntax der Switch-Expression ist zudem besser lesbar und auch aus anderen Gründen weniger fehleranfällig als das altbekannte Switch-Statement (zum Beispiel kein Fallthrough mehr, klar definierte Scopes von Variablen, …). Außerdem kann auch explizit auf `null`-Werte geprüft werden, anstatt dass es zu einer impliziten `NullpointerException` kommt.

Neben den in Java 21 finalisierten Pattern-Matching-Bausteinen, sind neu die *Unnamed Patterns and Variables* als Preview (JEP 443) hinzugekommen. Es kommt häufig vor, dass Pattern-Ausdrücke, Variablen oder Methoden-/Lambda-Parameter explizit benannt werden müssen, obwohl diese Variablen gar nicht verwendet werden. Das endet häufig in Warnungen in der Entwicklungsumgebung oder den Lint-Tools. Diese Warnungen können unterdrückt werden und mit mehr oder weniger sinnvollen Namen wie „unused" oder „ignored" lässt sich die Intention der nicht genutzten Variable ausdrücken. Durch die neue Funktion lassen sich nicht relevante Variablenzuweisung jetzt einfach durch einen Unterstrich ersetzen. In *Listing 7* wird im Zweig 1 der Wert auf die Empty-Instanz nicht benötigt. Und auch in den Zweigen 2 und 3 wird jeweils nur eine der beiden Record-Properties verwendet. Die anderen Werte kann man einfach mit dem „_" ignorieren. Es dürfen sogar mehrere Variablen in einer Zeile/Anweisung mit dem einen Unterstrich ignoriert werden.

```
static <T> boolean contains(T value,
  LinkedList<T> list) {

  return switch (list) {
     case Empty _ -> false;
     case Element<T>(T v, _)
        when Objects.equals(v, value) -> true;
     case Element<T>(_, var tail) ->
        contains(value, tail);
  };
}
```

Listing 7: Unnamed Patterns

Früher war der Unterstrich ein legaler Bezeichner. Ab Java 9 darf der einzelne Unterstrich aber nicht mehr verwendet werden. Variablen dürfen zwar weiter mit einem Unterstrich beginnen, auch zwei Unterstriche („__") sind möglich. Man kann also Quellcode von vor Java 9 mit einem Unterstrich hin zu zwei Unterstrichen migrieren.

Die gleiche Funktionalität bieten die *Unnamed Variables* auch bei *Exceptions* in `catch`-Blöcken oder bei Lambda-Parametern *(siehe Listing 8)*.

```
try {
  var result = 5 / 0;
} catch(ArithmeticException _) {
  System.out.println("Division nicht möglich");
}

System.out.println(new ArrayList<>(List.of(1, 2, 3))
      .stream()
      .map(_ -> 42)
      .toList()); // [42, 42, 42]
```

Listing 8: Unnamed Variables

String Templates bringen String Interpolation

Bereits in Java 12 sollte mit den *Raw String Literals* eine neue Art bei der Erzeugung von Zeichenketten eingeführt werden. Da ging es um mehrzeilige Strings und das Ignorieren von Escape-Sequenzen. Der JEP 326 wurde allerdings aus diversen Gründen (verwirrende Syntax, anderes Verhalten als normale Strings in Bezug auf Mehrzeiligkeit, …) zurückgezogen. Aber bereits im OpenJDK 15 wurde durch Text-Blocks die einfache Definition von mehrzeiligen Strings erlaubt. Schon damals wurden erste Stimmen laut, dass Java aber auch einen String-Interpolations-Mechanismus benötigt, so wie das viele andere Sprachen bereits seit längerem anbieten.

Um Zeichenketten zur Laufzeit aus statischen Texten, dem Inhalt von Variablen und berechneten Werten zusammenzufügen, gibt es verschiedene Möglichkeiten (String-Verkettung, `StringBuilder/-Buffer`, `String.format()`, `MessageFormat`). Die verschiedenen Ansätze haben aber ganz unterschiedliche Herausforderungen. Durch den unveränderlichen Charakter (*Immutability*) von String-Instanzen kann die Performance bei String-Verkettungen ein Problem sein, außerdem leidet nahezu bei allen Varianten die Lesbarkeit und es kann leicht zu Fehlern kommen (beispielsweise beim Zusammenbauen von Datenbank-Abfragen).

Dank des JEP 430 lassen sich nun Text-Templates definieren. Die durch \{..} gekennzeichneten Platzhalter dürfen Variablen oder auch beliebige Java-Ausdrücke (Berechnungen, Methodenaufrufe, …) enthalten. Innerhalb eines Platzhalters sind Zeilenumbrüche erlaubt (um zum Beispiel Code-Kommentare einzufügen) und es dürfen innen wiederum Anführungsstriche verwendet werden. Die String-Templates lassen sich auch mit mehrzeiligen Strings kombinieren, um beispielsweise SQL-Abfragen oder XML/HTML- beziehungsweise JSON-Dokumente zu erzeugen.

Die Verarbeitung der Templates erfolgt über einen sogenannten *Template Processor*. Dieser Prozessor kann ein beliebiges Objekt erstellen. Im einfachsten Fall einen String, genauso gut könnte er aber auch ein *PreparedStatement* (JDBC) oder eine Liste fachlicher Entitäten erzeugen. Für letztere Fälle gibt es im Moment keine fertigen Implementierungen im JDK, das lässt sich aber ziemlich einfach selbst schreiben. Und in Zukunft werden Frameworks und Bibliotheken solche Funktionen mitbringen. Wichtig ist, dass bei der Implementierung von solchen Template-Prozessoren gewisse Sicherheitsrisiken (zum Beispiel SQL-Injection) behandelt werden, um so potenzielle Angriffe von außen abwehren zu können.

Im JDK enthalten sind drei Template-Processor-Implementierungen, deren Instanzen sich einfach als Variablen verwenden lassen.

Im Falle einer einfachen String Interpolation wird zum Beispiel der *Processor STR* verwendet *(siehe Listing 9)*. Die Syntax scheint auf den ersten Blick gewöhnungsbedürftig. Das Template wird der Prozessor-Instanz hinter einem Punkt (ähnlich einem Instanz-Methoden-Aufruf) übergeben. Diese Schreibweise ist der Abwärtskompatibilität geschuldet, damit bestehender Code sich weiter korrekt verhält. Die Zeichenfolge `\{` ist übrigens keine gültige Escape-Sequenz, dementsprechend kann es im bestehenden Java-Quellcode keine Strings geben, die `\{` enthalten. Der Compiler hätte in Versionen vor 21 einen Fehler geworfen. Dadurch ist sichergestellt, dass kein Alt-Code fälschlich als String-Template erkannt wird. In anderen Programmiersprachen mag die Interpolation einfacher gelöst sein, die Syntax in Java bietet aber flexiblere und vielseitigere Möglichkeiten gerade bei der Entwicklung von eigenen Prozessoren.

```
User user = new User("dieter@develop.de", new Role("admin"));

String info =
  STR."'\{user.email()}' hat Rolle '\{user.role().name()}'";

// 'dieter@develop.de' hat Rolle 'admin'
System.out.println(info);
```

Listing 9: Einfache String-Interpolation

Listing 10 zeigt ein Template mit einem mehrzeiligen String und mehreren Zeilen beim Platzhalter `title`, der zudem noch mit einem Präfix verkettet wird.

```
String title = "My Web Page";
String text = "Hello, world";
String html = STR."""
  <html>
     <head>
        <title>\{ // Comment
           "My company: " + title }
        </title>
     </head>
     <body>
        <p>\{text}</p>
     </body>
  </html>""";
```

Listing 10: Interpolation in mehrzeiligen Strings

Mit *FMT* gibt es einen weiteren *Template Processor*, der die aus `String.format()` bekannten Formatierungsregeln auswertet. Während *STR* automatisch in jeder Java-Klasse importiert wird, muss *FMT* explizit über den Import `import static java.util.FormatProcessor.FMT` aktiviert werden *(siehe Listing 11)*.

```
import static java.util.FormatProcessor.FMT;

double zahl = 2.0;
String formatted = FMT."Zahl: %7.2f\{zahl}";
System.out.println(formatted); // Zahl:    2.00
```

Listing 11: Formatierung der Ersetzung mit dem Format Processor

Um direkt ein String-Template-Objekt zu erzeugen, muss der dritte vom JDK bereitgestellte Prozessor *RAW* verwendet werden. Dadurch lassen sich der Aufbau des String-Templates analysieren und mit der Methode `fragments()` die statischen Textbausteine sowie mit `values()` die Referenzen auf die einzufügenden Ausdrücke anschauen. Mit `process(Processor)` kann man dann einen beliebigen Prozessor übergeben und die Interpolation anstoßen *(siehe Listing 12)*.
Sowohl *STR* als auch *FMT* liefern jeweils einen String zurück: String → String. Der große Vorteil im Vergleich zu anderen Programmiersprachen ist, dass ein Prozessor auch einen anderen Datentyp zurückgeben kann. So kann aus einem mehrzeiligen String beispielsweise direkt ein `JSONObject` erzeugt werden. Dazu muss von `StringTemplate.Processor` abgeleitet werden, zum Beispiel über die statische Hilfsmethode `Processor.of()`. Für die Erzeugung des `JSONObject` wird das String-Template zunächst interpoliert (die Platzhalter werden ersetzt) und das Ergebnis nach JSON konvertiert *(siehe Listing 13)*.

Dieses einfache Beispiel zeigt noch nicht die ganze Mächtigkeit. Anstatt direkt die parameterlose Methode `template.interpolate()` aufzurufen, kann man zunächst auf den Platzhaltern-Variablen `(template.values())` Transformationen durchführen, um zum Beispiel potenziell gefährliche Benutzereingaben zu *escapen*. Anschließend übergibt man der `interpolate`-Methode die Textbausteine `(fragments())` sowie die angepassten Werte `(newValues)` und kann aus dem Ergebnis wieder das `JSONObject` erzeugen *(siehe Listing 14)*.

Nach diesem Prinzip können wir beliebige String-Interpolationen durchführen, zum Beispiel mit einem Locale-Objekt Texte internationalisieren, Log-Messages zusammenbauen oder SQL-Statements in ein *JDBCPreparedStatement* umwandeln. Wann immer es eine textbasierte Vorlage gibt, die transformiert, validiert oder gesäubert werden muss, bieten die String-Templates in Java jetzt eine einfache, ins JDK eingebaute *Template Engine* ohne Abhängigkeiten zu Drittanbietern.

Unnamed Classes and Instance Main Methods

Für den versierten Java-Entwickler scheint diese Neuerung zunächst wenig relevant. Mit dem Preview des JEP 445 ist es jetzt möglich, Java-Main-Anwendungen viel schlanker und ohne überflüssigen Boilerplate-Code zu definieren. Einerseits lassen sich `main`-Methoden viel kompakter schreiben (ohne `public`, `static`, `String[] args`, ...). Außerdem müssen sie auch nicht mal mehr in eine Klassen-Definition eingebettet sein. Gerade für Programmieranfänger, die in Java mit einem einfachen Hello-World-Programm starten, stellte die bisherige Vorgehensweise eine übermäßige Hürde dar. Sie müssen bereits zu Beginn gleich mehrere, in dem Moment nicht relevante Konzepte (Klassen, statische Methoden, String-Arrays etc.) verstehen.

Aber auch der erfahrene Java-Entwickler profitiert. Denn dank des im OpenJDK 11 eingeführten JEP 330 (*Launch Single-File Source-Code Programs*) lassen Java-Anwendungen direkt ausführen (ohne vorherige Kompilierung). Jetzt sinkt der Aufwand weiter, da man in der **.java*-Datei nur noch eine schlanke `main`-Methode benötigt und sogar auf eine Klassendefinition verzichten kann. Ein einfaches Beispiel inklusive des Kommandozeilenbefehls zeigt *Listing 15*.

```
import static java.lang.StringTemplate.RAW;

template = RAW."Today is day
  \{LocalDate.now().getDayOfYear()} of year
  \{LocalDate.now().getYear()}.";

System.out.println(template.fragments()); System.out.println(template.values());

// Today is day 244 of year 2023.
String result = template.process(STR); System.out.println(result);
```

Listing 12: RAW-String-Template

```
// new Template Processor
var JSON = StringTemplate.Processor.of(
  (StringTemplate template) ->
     new JSONObject(template.interpolate()));

User user = new User("dieter@develop.de",
  new Role("admin"));

JSONObject json = JSON."""
  {
     "user": "\{ user.email() }",
     "roles": "\{user.role().name()}"
  }""";

// {"roles":"admin","user":"dieter@develop.de"} System.out.println(json);

}
```

Listing 13: Einfacher Custom Template Processor

```
Processor<JSONObject, JSONException> JSON =
  template -> {
        String quote = "\"";
        List<Object> newValues = new ArrayList<>();
        for (Object value : template.values()) {
           // Ersetzungen durchführen
           // ...
           newValues.add(value);
        }

        var result = StringTemplate.
           interpolate(template.fragments(), newValues);
        return new JSONObject(result);
  };
```

Listing 14: Fortgeschrittener Custom Template Processor

Die `main()`-Methode liegt hierbei in einer unbenannten Klasse, also einem ähnlichen Konzept wie die anonyme innere Klasse oder das unbenannte Package beziehungsweise Modul. Durch den fehlenden Namen kann aber nicht von außen auf die unbenannte Klasse zugegriffen werden. Die unbenannte Klasse darf aber noch weitere Attribute/Felder und Methoden enthalten *(siehe Listing 16)*. Dadurch fühlt sich Java fast schon wie eine Skriptsprache an und es werden ganz neue Anwendungsfälle ermöglicht. Wir können nun Shell-Skripte schreiben und profitieren trotzdem von dem mächtigen Funktionsumfang und der Typsicherheit Javas.

Es kann nun theoretisch mehrere `main`-Methoden in der *Unnamed Class* geben, die jeweils für sich allein eine gültige Start-Prozedur wären. Das Launch-Protokoll entscheidet, welche der `main`-Methoden den Vorrang bei der Ausführung hat. *Listing 17* zeigt ein Beispiel. Hier würde die erste Methode „gewinnen", weil sie statisch ist. Lässt

```
// > java --source 21 --enable-preview  Main.java
void main() {
  System.out.println("Hello, World!");
}
```

Listing 15: Instance Main Method

```
// > java --source 21 --enable-preview  Main.java

final String greeting = "Hello";

void main() {
  System.out.println(greet("World"));
}

String greet(String name) {
  return STR."\{greeting}, \{name}!";
}
```

Listing 16: Zusätzliche Felder und Methoden mit der Instance Main Methode

```
protected static void main() {
     System.out.println("protected static void main()");
}

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

Listing 17: Aufrufreihenfolge von main-Methoden in unbenannten Klassen

man das `static` weg, wird die zweite Methode aufgerufen, weil sie sichtbarer ist (`public` vs. `protected`).

Sequenced Collections mit einheitlichen Zugriffen

In Javas Collection-Framework existierten bislang keine Schnittstellen, die einheitliche Operationen für Sequenzen von Elementen in einer wohldefinierten Reihenfolge anbieten. Je nach Datenstruktur gab es sehr unterschiedliche, teilweise umständliche sowie teils nicht performante Wege, um beispielsweise auf das erste oder letzte Element zuzugreifen. Beispiele für solche Collection-Typen sind `List`, `Deque`, `SortedSet` oder auch die Implementierung `LinkedHashSet`. Der gemeinsame Super-Typ von `List` und `Deque` ist die Interface Collection, die gar keine Ordnung der Elemente vorgibt und dementsprechend keine entsprechenden Zugriffsmethoden zur Verfügung stellt.

Mit dem JEP 431 (*Sequenced Collections*) wurde nun eine neue Familie von Interfaces eingeführt, die den Zugriff auf Elemente in Collections mit wohldefinierter Reihenfolge vereinheitlichen. Für die Collections sind das die Interfaces *SequencedCollection* und *SequencedSet* *(siehe Abbildung 5)*.

Diese enthalten die neuen Methoden `addFirst(Object)`, `addLast(Object)`, `getFirst()`, `getLast()`, `removeFirst()`, `removeLast()` und `reversed()`. Letztere bietet eine umgekehrt geordnete Sicht auf die ursprüngliche Collection und ermöglicht somit die Iteration über die Elemente auch vom anderen Ende. Da es eine Sicht ist, werden Änderungen an der ursprünglichen Collection auch in der View sichtbar (und umgekehrt).

`SequencedSet` leitet von `SequencedCollection` ab, redefiniert aber die `reversed()`-Methode neu und gibt ein `SequencedSet` statt einer `SequencedCollection` zurück. *Listing 18* zeigt ein paar Beispiele.

```
List<Integer> list =
  new ArrayList<>(List.of(1, 2, 3, 4, 5));
List<Integer> reversed = list.reversed();
System.out.println(reversed); // 5, 4, 3, 2, 1

list.addFirst(0);
list.addLast(6);

System.out.println(list.getFirst()); // 0
System.out.println(list.getLast()); // 6
```

Listing 18: Sequenced Collection Beispiele

Alle neuen Methoden bringen Default-Implementierungen mit, werfen aber gegebenenfalls eine `UnsupportedOperationException` beziehungsweise interpretieren die Logik je nach Charakteristik des Subtyps anders. Beispielsweise kann bei einem `SortedSet` explizit kein erstes oder letztes Element eingefügt werden, da sich die Reihenfolge aus der Sortierung aller Elemente zueinander ergibt. Somit müssen `addFirst(Object)` und `addLast(Object)` jeweils eine *Exception* werfen. Trotz dieser Unschönheit lässt sich das `SortedSet` aber nun immerhin mit der neuen, einheitlichen Operation auslesen beziehungsweise über die `reversed`-Sicht umdrehen. Es bräuchte in dem Fall noch eine Unterscheidung zwischen `Immtuable`- und `MutableSequencedCollection`. Aber das ist grundsätzliches Problem des Java-Collections-Framework.

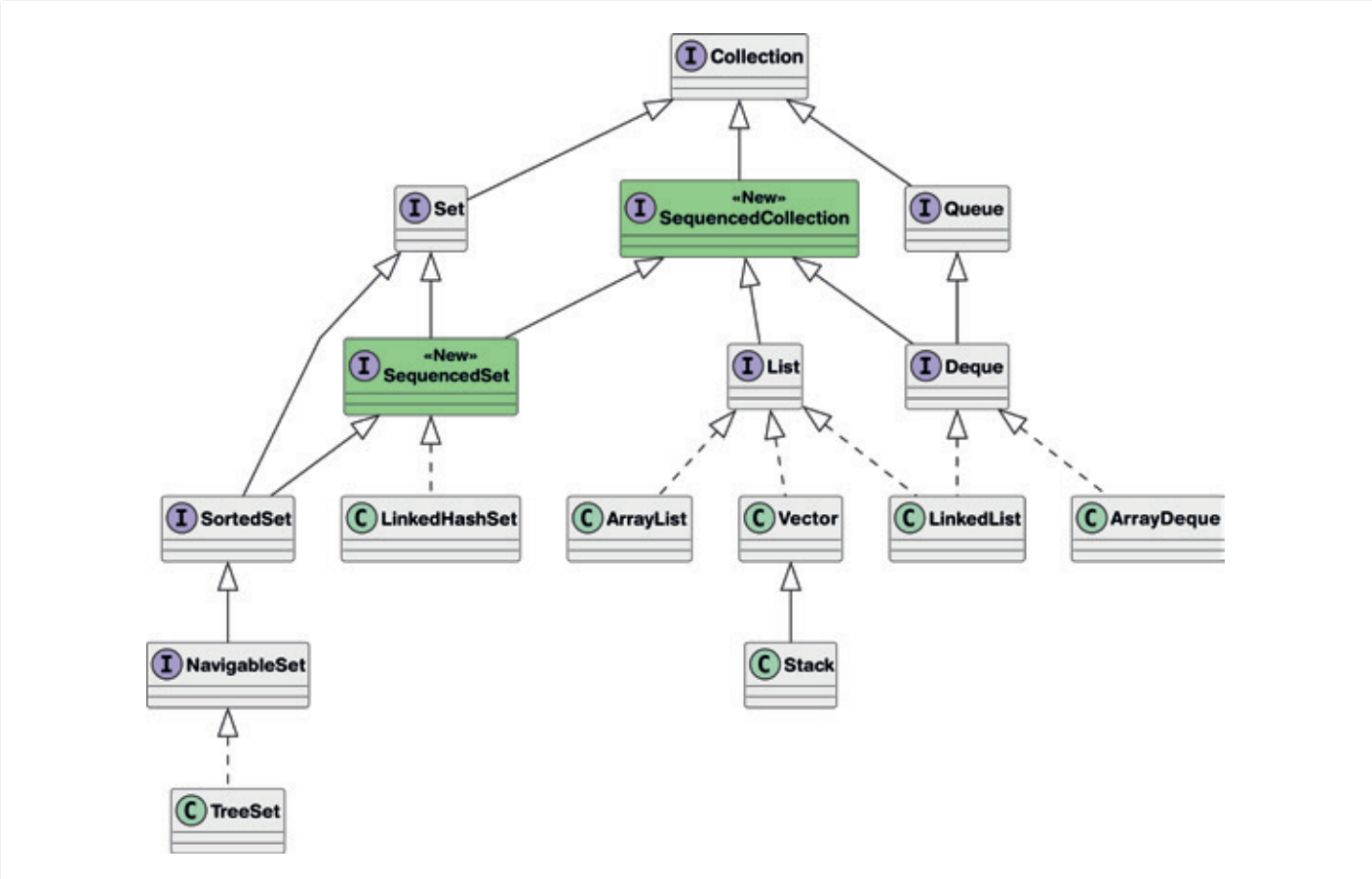


Abbildung 5: Auszug aus dem Collection-Framework mit den Interfaces und wichtigen Implementierungen

Auch für die Schlüsselwertepaar-Container gibt es ein neues Interface, das Methoden für Maps definiert, die Elemente in einer wohldefinierten Reihenfolge enthalten.

Hierbei kann der erste oder letzte Entry (Key-Value-Paar) gelesen (`firstEntry()` und `lastEntry()`), gelöscht (`pollFirstEntry()` und `pollLastEntry()`) und neue Entries an erster oder letzter Stelle eingefügt werden (`putFirst(Object, Object)`, `putLast(Object, Object)`). Außerdem kann mit `reversed()` die Reihenfolge gedreht werden und es wird je ein `SequencedSet` entweder von den Entries, den Keys oder den Values zurückgegeben. *Listing 19* zeigt auch hierzu einige Varianten.

```
SequencedMap<Integer, String > map =
  new LinkedHashMap<>(Map.of(1, "a", 2, "b"));
SequencedMap<Integer, String> reversedMap =
  map.reversed();
System.out.println(map); // {2=b, 1=a}
System.out.println(reversedMap); // {1=a, 2=b}

map.putFirst(0, "o");
map.putLast(3, "c");

System.out.println(map.pollFirstEntry()); // 0=o
System.out.println(reversedMap); // {3=c, 2=b, 1=a}

// [3, 2, 1]
System.out.println(map.sequencedKeySet().reversed());
```

Listing 19: Sequenced Map Beispiele

Das Collections-Framework ist schon alt, wird aber immer noch intensiv genutzt. Daher ist es schön, dass mit den `SequencedCollections` einheitliche, konsistente Operationen für Datenstrukturen mit einer wohldefinierten Reihenfolge hinzugekommen sind und die Nutzung damit für bestimmte Anwendungsfälle vereinfachen und vor allem auch lesbarer machen.

Was sonst noch passiert ist?

Das *Vector API* ist ein Dauerläufer und taucht seit Java 16 regelmäßig in den Releases auf, diesmal als 6. Inkubator (Vorstufe von Preview). Es geht dabei um die Unterstützung der modernen Möglichkeiten von SIMD-Rechnerarchitekturen mit Vektorprozessoren. *Single Instruction Multiple Data* (SIMD) lässt viele Prozessoren gleichzeitig unterschiedliche Daten verarbeiten. Durch die Parallelisierung auf Hardware-Ebene verringert sich beim SIMD-Prinzip der Aufwand für rechenintensive Schleifen.

Auch schon seit einigen Versionen dabei und diesmal erneut ein Preview, ist das *Foreign Function & Memory API*. Wer schon sehr lange in der Java-Welt unterwegs ist, wird auch das *Java Native Interface* (JNI) kennen. Damit kann man nativen C-Code aus Java heraus aufrufen. Der Ansatz ist aber relativ aufwändig und fragil. Das *Foreign Linker API* bietet einen statisch typisierten, rein Java-basierten Zugriff auf nativen Code. Zusammen mit dem *Foreign-Memory Access API* kann diese Schnittstelle den bisher fehleranfälligen und langsamen Prozess der Anbindung einer nativen Bibliothek beträchtlich vereinfachen. Mit letzterer bekommen Java-Anwendungen die Möglichkeit, außerhalb der Heaps zusätzlichen Speicher zu allokieren. Ziel der neuen APIs ist es, den Implementierungsaufwand um 90 % zu reduzieren und die Leistung um Faktor 4 bis 5 zu beschleunigen. Beide

APIs sind seit dem JDK 14 beziehungsweise 16 im JDK zunächst einzeln und ab 18 innerhalb eines Incubator-JEPs gemeinsam enthalten. Mit dem JEP 434 wurden nochmal einige Änderungen am API gemacht. Für Java 22 soll es finalisiert werden.

Bei den *Garbage Collectors* hat sich auch etwas getan. Der vor wenigen Jahren im OpenJDK 15 eingeführte ZGC (*Scalable Low-Latency Garbage Collector*) gehört zu einer neuen Generation. Das Ziel ist, große Datenmengen (Terabyte von RAM) mit möglichst kurzen GC-Pausen (< 10 ms) aufzuräumen und so die Anwendung nahezu immer antwortbereit zu halten. Bisher machte der ZGC keine Unterscheidung zwischen frischen und schon länger existierenden Objekten. Die Idee ist, dass Objekte, die bereits einen oder mehrere GC-Läufe überlebt haben, auch in Zukunft wahrscheinlich noch lange weiterleben werden. Diese werden dann in einen extra Bereich (*Old Generation*) verschoben und müssen so bei den Standard-Läufen nicht mehr verarbeitet werden. Das kann einen Schub bei der Performance geben.

Es gibt noch einige weitere, für Entwickler aber nicht so relevante JEPs. Für die vielen Details lohnt ein Blick auf die Release Notes *[4]*. Änderungen am JDK (Java Klassenbibliothek) lassen sich zudem sehr schön über den Java Almanac *[5]* nachvollziehen. In dieser Übersicht finden sich zum Beispiel auch alle Neuerungen zu den String-Templates und den *Sequenced Collections*. Außerdem gab es kleine Erweiterungen in der String-Klasse, zum Beispiel wurde eine `indexOf(String str, int beginIndex, int endIndex)`-Methode eingeführt, die in einem bestimmten Bereich nach einem Teilstring sucht. Bei `StringBuffer` und `StringBuilder` wurden jeweils zwei ähnliche Methoden eingeführt, die ein Zeichen oder eine Zeichenkette wiederholt an das bestehende Objekt anhängen: `repeat(CharSequence, int)`. Die Klasse *Character* wurde wiederum um eine Vielzahl von Methoden erweitert, die prüfen, ob ein Unicode-Zeichen, ein Emoji oder eine Variante davon darstellt. Java geht also auch hier mit dem Zeitgeist.

Fazit

Die regelmäßigen Releases bringen weiter halbjährlich neue Funktionen, die das Leben der Java-Entwickler verbessern und nicht langweilig werden lassen. Einige der Features werden in sogenannten Inkubator-Projekten bereits seit vielen Jahren im Hintergrund vorbereitet. Im Project Loom wurde schon lange an den *Virtual Threads* gearbeitet, die nun nach nur wenigen Preview-Iterationen bereits finalisiert wurden. Sie vereinfachen die Entwicklung nebenläufiger Anwendungen. Mit der *Structured Concurrency* und den *Scoped Values* sind da aber noch weitere wichtige Bausteine in Arbeit. Auch beim *Pattern Matching* wurde ein wichtiger Meilenstein erreicht. Mit der Finalisierung der *Record Patterns* und dem *Pattern Matching for switch* lässt sich diese neue Art der Programmierung jetzt auch produktiv einsetzen. Weitere Pattern Typen sind in der Planung, die *Unnamed Patterns* sogar als ein erster Preview bereits erschienen. Die String-Templates, *Sequenced Collections* und die *Unnamed Classes and Instance Main Methods* sind weitere interessante Features, die dieses umfangreiche Java-Release abrunden.

Neben diesen schon anwendbaren Funktionen warten viele sehnsüchtig auf die Reformen am Typsystem. Java hat aktuell ein zweigeteiltes Typsystem mit den primitiven und Referenztypen (Klassen). Die primitiven Datentypen wurden ursprünglich aus Performanceoptimierungsgründen eingeführt, haben aber im Handling entscheidende Nachteile. Referenztypen sind auch nicht immer die beste Wahl, insbesondere was die Effizienz und den Speicherverbrauch angeht. Es braucht etwas dazwischen, was sich so schlank und performant wie primitive Datentypen verhält, aber auch die Vorzüge von selbst zu erstellenden Referenztypen in Form von Klassen kennt. Schon bald könnten daher aus dem Inkubator-Projekt *Valhalla Value Types* (haben keine Identität) und *Universal Generics* (List<int>) ins JDK überführt werden und Javas Typsystem so viel flexibler machen. Ob das schon für Java 22 passieren wird, lässt sich im Moment noch nicht sagen. Aber eigentlich wäre es ein guter Zeitpunkt, um diese Features dann auf dem Weg zum nächsten LTS-Release (Java 25) in zwei Jahren zu stabilisieren.

Weitere JEPs oder JEP-Drafts, die im OpenJDK 22 auftauchen könnten:

- *Statements before super* (JEP 447): Zum Validieren von Eingabeparametern in Konstruktoren vor der Delegation an `super()` oder `this()`
- JEP 455: `Primitive Types in Patterns instanceof and switch`: Neues Type-Pattern, um auch auf primitive Datentypen (int, long, ...) prüfen zu können
- JEP draft: `Ahead of Time Compilation`: schnellerer Start und eine schnellere Ausführung durch Erweiterung der Java Virtual Machine (Laden Java-Anwendungen und -Bibliotheken, die in nativen Code kompiliert wurden)

Es passiert also weiterhin einiges in der Java-Welt. Wir dürfen gespannt sein und können weiterhin optimistisch in die Zukunft blicken. Denn das Java-Ökosystem ist auch nach fast 30 Jahren lebendiger denn je und weiterhin sehr innovativ.

Referenzen:

[1] *https://openjdk.java.net/projects/jdk/21/*
[2] *https://openjdk.java.net/projects/amber/*
[3] *https://www.infoq.com/articles/data-oriented-programming-java/*
[4] *https://jdk.java.net/21/release-notes*
[5] *https://javaalmanac.io/jdk/21/apidiff/17/*

Falk Sippach
falk@jug-da.de
https://twitter.com/sippsack

Falk Sippach ist bei der embarc Software Consulting GmbH als Softwarearchitekt, Berater und Trainer stets auf der Suche nach dem Funken Leidenschaft, den er bei seinen Teilnehmern, Kunden und Kollegen entfachen kann. Bereits seit über 15 Jahren unterstützt er in meist agilen Softwareentwicklungsprojekten im Java-Umfeld. Als aktiver Bestandteil der Community (Mitorganisator der JUG Darmstadt) teilt er zudem sein Wissen gern in Artikeln, Blog-Beiträgen sowie bei Vorträgen auf Konferenzen oder User Group Treffen und unterstützt bei der Organisation diverser Fachveranstaltungen. Falk twittert unter @sippsack.

Hürden beim Upgrade auf Java 21

Nicolai Parlog, Oracle

Java 21 ist ein großartiger Release mit Pattern Matching, Virtual Threads, Sequenced Collections, Key Encapsulation, Generational ZGC und vielem mehr. Insbesondere, wenn man von Version 17 kommt, ist das eine ganze Menge. All das kann man natürlich nur verwenden, wenn das Update auf 21 auch gelingt und während sich dabei keine einzelne große Hürde in den Weg stellt, gibt es doch einige kleinere Herausforderungen, die es zu überwinden gilt. Ob Verbesserungen, Bug Fixes oder Deprecations: Es gibt jede Menge Detailänderungen zwischen 17 und 21, die das eine oder andere Projekt betreffen werden.

In diesem Artikel gehen wir die ganze Liste durch, mit einer kurzen Beschreibung der jeweiligen Hürde, wie man sie überwindet und (wichtig!) einem Link zu mehr Details, den ich jedem und jeder ans Herz legen kann, wenn ein Problem tatsächlich auftritt. Wir schauen uns zunächst Änderungen auf API-Ebene an, dann die aktuellen Deprecations, auf die man reagieren muss, danach folgen die Änderungen der Laufzeitumgebung und zum Schluss Kommandozeilenwerkzeuge.

API-Änderungen

Sequenced Collections

Die bedeutendste Änderung im Bereich Collections ist die Einführung der neuen Interfaces `SequencedCollection`, `SequencedSet` und `SequencedMap`, die in die bestehende Vererbungshierarchie eingefügt wurden. So erbt nun beispielsweise `List` von `SequencedCollection` und `SortedSet` erbt von `SequencedSet`. Die Interfaces kommen mit Methoden zum Lesen, Hinzufügen und Entfernen von Elementen am Anfang und Ende einer Collection, so wie zum Invertieren *(siehe Listing 1)*.

```
void addFirst(E);
void addLast(E);
E getFirst();
E getLast();
E removeFirst();
E removeLast();

SequencedCollection<E> reversed();
```

Listing 1: Methoden auf SequencedCollection

Dank `default`-Methoden ist diese Neuerung weitestgehend rückwärtskompatibel. Zu Problemen kommt es nur, wenn bestehende Implementierungen eines Interfaces Methoden mit den neuen Signaturen haben, die anderweitig inkompatibel sind, zum Beispiel weil sie einen anderen Rückgabetyp haben *(siehe Listing 2)*.

```
public class StringList
      implements List<String> {

      /* [...] */

      public Optional<String> getFirst() {
            return size() == 0
                        ? Optional.empty()
                        : Optional.of(get(0));
      }
}
```

Listing 2: Beispiel einer List-Implementierung, die eine inkompatible `getFirst`-Methode enthält

Code wie dieser führt zu Compile- beziehungsweise Laufzeitfehlern. Das Problem lässt sich lösen, indem man im eigenen Code die Methoden kompatibel macht (syntaktisch *und* semantisch) und in Dependencies auf ein Update wartet.

Double/Float zu String

Die Methoden `Double::toString` und `Float::toString` versprechen, die String-Repräsentation mit der geringsten Anzahl an Ziffern zu finden, die die Zahl eindeutig von ihrem Nachbarn unterscheidet. Das funktionierte bis Java 18 nicht immer einwandfrei – Java 19 hat diesen Bug behoben *(siehe Listing 3)*.

Das kann dazu führen, dass sich Ausgaben oder Log-Dateien ändern oder gar Tests fehlschlagen.

```
// bis Java 18:
jshell> Double.toString(1e23)
$1 ==> "9.999999999999999E22"
// seit Java 19:
jshell> Double.toString(1e23)
$1 ==> "1.0E23"
```

Listing 3: String-Formatierung von 1e23 in verschiedenen Java-Versionen

Identität in IdentityHashMap

`IdentityHashMap` ist eine besondere `Map`, die zum Vergleichen nicht `equals` sondern == benutzt (und deswegen auch nicht `Object::hashCode`, sondern `Sytem::identityHashCode` für die Schlüssel). Die Methoden `remove(key, value)` und `replace(key, value, newValue)` haben bis einschließlich Java 19 das Argument `value` allerdings irrtümlicherweise mit `equals` mit dem Wert in der `Map` verglichen. Dieser Fehler wurde in Java 20 behoben, was dazu führen kann, dass Maps weniger Elemente entfernen beziehungsweise ersetzen als bisher *(siehe Listing 4)*.

Netzwerkschnittstellennamen auf Windows

Instanzen der Java-Klasse `NetworkInterface` repräsentieren Netzwerkschnittstellen, wozu unter anderem deren Name gehört. Diesen kann man mit `getName()` abfragen oder mittels `NetworkInterface.getByName(String)` nutzen, um das Gerät mit dem spezifizierten Namen zu finden.

Bisher hat Java den Netzwerkschnittstellen auf Windows synthetische Namen zugewiesen, etwa „net0" oder „eth0". Ab Java 21 ist dies nicht mehr der Fall und die Namen entsprechen denen, die das Betriebssystem verwendet, zum Beispiel „ethernet_0" oder „ethernet_32768". Das bedeutet, dass besonders Aufrufe von `NetworkInterface.getByName(String)` ihr Verhalten ändern und daher überprüft werden sollten.

URL-Validierung

Die Klasse `URL` hat einige Schwachstellen, insbesondere dass Aufrufe von `equals` dazu führen, dass Hostnamen per DNS zu IP-Adressen aufgelöst werden. Wer sie dennoch verwendet (statt `URI`), sollte wissen, dass sich in Java 20 die Validierung der URL-Syntax von Aufrufen wie `URL::openConnection` und `URLConnection::connect` in den URL-Konstruktor verschoben hat.

Fehler, die bisher beim Verbindungsaufbau oder, falls die URL dazu nicht verwendet wurde, gar nicht aufgetreten sind, können jetzt also bei der Konstruktion von URL auftreten. Falls Projekte mehr Zeit brauchen, um sich auf diese Änderung einzustellen, können sie mit der Systemeigenschaft `jdk.net.url.delayParsing` einstellen, dass die Validierung wieder erst beim Verbindungsaufbau stattfindet.

JNDI-Provider sind bei den URLs ebenfalls strikter geworden. Dieses Verhalten kann mit den folgenden Systemeigenschaften konfiguriert werden:

- für „ldap:" URLs: `com.sun.jndi.ldapURLParsing`
- für „dns:" URLs: `com.sun.jndi.dnsURLParsing`
- für „rmi:" URLs: `com.sun.jndi.rmiURLParsing`

```
record User(String name) { }

public static void main(String args[]) {
     var users = new IdentityHashMap<String, User>();
     String key = "abc";

     // füge eine (key, user)-Kombination ein
     users.put(key, new User("Jane Doe"));
     // versuche eine GLEICHE aber nicht
     // IDENTISCHE Kombination zu entfernen
     boolean removed = users.remove(key, new User("Jane Doe"));

     // `removed == true` bis einschließlich Java 19:
     // `removed == false` ab Java 20
}
```

Listing 4: Entfernen von gleichen, aber nicht identischen Key/Value-Paaren aus einer IdentityHashMap in verschiedenen Java-Versionen

Dabei sind jeweils folgende Werte möglich:

- „legacy": keine Validierung (bisheriges Verhalten)
- „compat": eingeschränkte Validierung, die möglichst kompatibel ist (Standardwert)
- „strict": strikte Validierung

Deprecations

Java wird fortlaufend durch Bug Fixes, Verfeinerungen und neue Features verbessert. Eine weniger offensichtliche aber mindestens ebenso wichtige Form der Weiterentwicklung ist das Entfernen von Features, die ihren Zweck nicht (mehr) erfüllen und die Weiterentwicklung von Java behindern. Für Entwicklerinnen und Entwickler bedeutet das unter Umständen, dass Code, der diese Features verwendet, angepasst werden muss.

Dynamic Agents

Agenten sind spezielle Komponenten, die durch Nutzung der `java.lang.instrument` oder JVM-TI-APIs den Byte-Code einer laufenden Anwendung umschreiben, beispielsweise um ihn für Performance-Messungen zu instrumentieren. Dies ist offensichtlich ein sehr invasiver Prozess und sollte nur durch explizite Erlaubnis der Betreiberinnen oder Betreiber einer Anwendung erlaubt sein.

Bei statischen Agenten – das sind solche, die zusammen mit der Anwendung gestartet werden – ist dies bereits der Fall, da sie eine Reihe von Kommandozeilenoptionen benötigen. Wird ein Agent erst nach dem Start einer Anwendung angehängt, nennt man ihn „dynamisch" und das benötigt keine Option beim Starten der Anwendung – es ist standardmäßig erlaubt.

Das ändert sich auch in Java 21 nicht, wird es aber in Zukunft. Dann wird das Anhängen dynamischer Agenten an eine Anwendung nur möglich sein, wenn sie mit der Option `-XX:+EnableDynamicAgentLoading` gestartet wurde.

Um die eigene Verwendung von Agenten zu analysieren, kann man Anwendungen heute schon mit `-XX:-EnableDynamicAgentLoading` starten. Dadurch werden dynamische Agenten verboten, sodass sie statt zu starten, Fehler erzeugen und sich so bemerkbar machen.

Finalization

Bei Finalization handelt es sich um einen alten Mechanismus zum Freigeben von externen Ressourcen, wie zum Beispiel File Handles. Er basiert auf `finalize()`-Methoden und einer komplexen Maschinerie in Garbage Collectors. Finalization hat eine Reihe von Schwächen, weshalb im Laufe der Jahre mit Try-with-Resources und der Cleaner-API zwei Alternativen geschaffen wurden.

Finalization wird nun langsam entfernt. Aktuell bedeutet das, dass `finalize()` deprecated ist (`forRemoval=true`). Der Mechanismus ist allerdings noch im Einsatz, kann aber mit `--finalization=disabled` deaktiviert werden. In Zukunft soll Finalization standardmäßig deaktiviert sein und mit `--finalization=enabled` aktiviert werden müssen, bevor das irgendwann auch nicht mehr möglich ist, weil der gesamte Mechanismus entfernt wurde.

Um herauszufinden, ob die eigene Anwendung auf Finalization angewiesen ist, kann man sie auf Java 21 einmal mit und einmal ohne `--finalization=disabled` laufen lassen und mit einem Monitoring Tool vergleichen, ob es Unterschiede in der Verwaltung externer Ressourcen gibt.

Konstruktoren von Primitive Wrappern

Wrapper-Klassen von primitiven Typen wie `Integer`, `Long`, `Float`, `Double` usw. werden im Rahmen von Project Valhalla tiefgehende Änderungen erfahren. Der Kern dessen ist, dass sie in Zukunft keine Identität mehr haben sollen. Das bedeutet, zwei Integer-Instanzen mit Wert 42 werden ebenso ununterscheidbar sein wie zwei `int`s mit dem Wert 42.

Dazu müssen identitätssensitive Operationen entfernt werden, was bei den Wrapper-Klassen die Konstruktoren sind. Denn während zwei Integer-Instanzen mit dem Wert 42 nicht unterscheidbar sein wollen, verspricht die Java Language Specification, dass zwei Konstruktoraufrufe zwei unterschiedliche Instanzen liefern müssen.

```
// deprecated
var answer = new Interger(42);
// Ersatz
var answer = Integer.valueOf(42);
```

Listing 5: Erstellung von Integer-Instanzen

Deswegen sind die Konstruktoren `deprecated` (`forRemoval=true`) – der Ersatz sind die `valueOf`-Methoden *(siehe Listing 5)*.

Security Manager

Die Klasse SecurityManager und die damit verbundenen Mechanismen von Checks und Permissions sollen die Sicherheit von Java-Anwendungen verbessern. In der Praxis erreicht der Security Manager dieses Ziel kaum – zum einen, weil extrem wenige Anwendungen ihn verwenden; zum anderen, weil es nicht einfach ist, ihn korrekt zu konfigurieren. Auf der anderen Seite ist er sehr weit in der Laufzeitumgebung verzweigt und benötigt fortlaufende Wartung und Erweiterung, wenn neue Features eingeführt werden. Diese Zeit könnte stattdessen an anderer Stelle in Javas Sicherheit investiert werden.

In der Summe ist der Beitrag des Security Managers zur Sicherheit in Java also negativ, weshalb er aktuell entfernt wird. In Java 21 kann er weiterhin gesetzt werden, sofern dies beim Start mit `java.security.manager=allow` erlaubt wurde. Der Typ SecurityManager und Methoden, die ihn verwenden, sind `deprecated (forRemoval=true)`.

Biased Locking

Biased Locking beschleunigt konsekutive Übernahmen eines Object Monitors durch den gleichen Thread. Der Monitor wird insbesondere benötigt, um `synchronized`-Methoden zu betreten, deren Prävalenz durch die Einführung nicht-synchronisierter Collections wie `HashMap` und `ArrayList` in Java 1.2 oder `ConcurrentHashMap` in 1.5 allerdings drastisch zurückgegangen ist. Da Biased Locking ein komplexer und weit verzweigter Mechanismus ist, erschwert er die Wartung von HotSpot. Die Kosten-Nutzen-Rechnung ist also negativ, weswegen Baised Locking schrittweise entfernt wird.

Beim Update auf Java 21 bedeutet das, dass allerlei VM-Optionen obsolet werden:

- `UseBiasedLocking`
- `BiasedLockingStartupDelay`
- `BiasedLockingBulkRebiasThreshold`
- `BiasedLockingBulkRevokeThreshold`
- `BiasedLockingDecayTime`
- `UseOptoBiasInlining`

Sie zu nutzen führt momentan zu einer Warnung und sie sollten aus Launch-Skripten entfernt werden, denn sobald die Optionen aus dem JDK entfernt werden, führt ihre Nutzung zu einem Fehler beim Start.

Laufzeitumgebung

HTTP-Timeouts

Der in Java 11 eingeführte HTTP/2-Client hatte bisher ein sehr großzügiges Timeout von 20 Minuten(!) für inaktive Verbindungen. In Java 20 wurde dieser Wert auf 30 Sekunden heruntergesetzt und außerdem mit den Systemeigenschaften `jdk.httpclient.keepalivetimeout` (für alle HTTP-Verbindungen mit dem Client) und `jdk.httpclient.keepalivetimeout.h2` (nur für HTTP/2-Verbindungen) konfigurierbar gemacht.

UTF-8 überall!

Javas Datei-APIs wie `Files::readAllLines` haben typischerweise zwei Varianten: Eine mit und eine ohne `Charset`-Parameter, mit dem man das Encoding einer Textdatei angeben kann. Übergibt man keines, versuchte Java bisher ein zur Umgebung passendes Encoding auszutüfteln und ist dabei auf Unix-basierten Betriebssystemen typischerweise bei UTF-8 und auf Windows zum Beispiel bei Windows-1252 gelandet.

In der Praxis bedeutet das, dass sich die Applikation je nach Betriebssystem, auf dem sie läuft, anders verhält. Das passt nicht gut zu einer Sprache, deren Slogan „Write once, run anywhere" ist. In diesem Sinne verwendet Java seit Version 18 grundsätzlich UTF-8 als Standard-Encoding.

Diese Änderung betrifft nur Projekte, die keine der folgenden Bedingungen erfüllen:

- `Charset` wird immer spezifiziert
- die Systemeigenschaft `file.encoding` ist auf „UTF-8" gesetzt
- läuft nur auf Unix-basierten Betriebssystemen

Betroffene Projekte sollten, wenn möglich, das passende Charset übergeben oder auf UTF-8-kodierte Dateien umsteigen. Kurzfristig lässt sich das alte Verhalten wiederherstellen, indem man die Systemeigenschaft `file.encoding` auf COMPAT setzt.

Eine gute Möglichkeit, sich vor einem Upgrade auf Java 18+ mit dieser Änderung vertraut zu machen, ist `file.encoding` auf UTF-8 zu setzen.

Standard-Charset

Im Rahmen der Umstellung auf UTF-8 als Standard wurde auch der Alias „default" für US-ASCII abgeschafft, sodass Aufrufe von `Charset.forName("default")` jetzt eine `UnsupportedOperationException` werfen. Die meisten dieser Aufrufe sollten entweder durch `Charset.forName("US-ASCII")` (also den gleichen Wert wie zuvor „default") oder `Charset.defaultCharset()` (also UTF-8, wenn nicht anders konfiguriert) ersetzt werden.

CLDR Version 42

Das JDK aktualisiert regelmäßig die Unicode-Version und das Upgrade auf 42 hat einige Änderungen nach sich gezogen, die sehr überraschend sein können. Denn Teil von Unicode ist auch die Formatierung von Daten, Zeiten, Einheiten usw. und in vielen davon wurde das normale Leerzeichen (U+0020) durch ein No-Break-

```
var midFormat = DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM);
var now = LocalTime.now().format(midFormat);
System.out.println(now);
```

Listing 6: Lokalisierte DateTime-Formattierung

Space (U+00A0) oder Narrow-No-Break Space (U+202F) ersetzt.

Der Code in *Listing 6* demonstriert dies.

Bis einschließlich Java 19 war die Ausgabe „6:14:18 PM"; seit Java 20 ist sie „6:14:18 PM" mit einem schmalen Leerzeichen vor „PM" (falls es das durch den Druckprozess bis aufs Papier geschafft hat 😊).

Auch an anderen Stellen in der Formatierung gab es kleine Änderungen. Diese betreffen ebenfalls weitestgehend die Präsentationsschicht, aber Code, der formatierte Daten/Zeiten parsen soll oder diese formatiert und mit erwarteten Ausgaben vergleicht (zum Beispiel im Rahmen von Tests), kann durchaus Fehler werfen. Diese sind dadurch besonders tückisch, dass sich " "," " und " " nicht in jedem Editor gut unterscheiden lassen.

Falls sich betroffener Code nicht aktualisieren lässt, kann mit dem JVM-Argument `java.locale.providers=COMPAT` das ursprüngliche Verhalten wiederhergestellt werden. Das schränkt aber Locale-bezogene Funktionen ein und ist zudem nicht permanent – es sollte also als temporäre Notlösung betrachtet werden.

XSL-Transformations

Bei der Transformation von XSLT-Stylesheets kann es neuerdings zu folgender Exception kommen:

```
com.sun.org.apache.xalan.internal.xsltc.compiler.util.
InternalError:
Internal XSLTC error: a method in the translet exceeds the
  Java Virtual Machine limitation on the length of a method
  of 64 kilobytes. This is usually caused by templates in a
  stylesheet that are very large. Try restructuring your
  stylesheet to use smaller templates.
```

Die Beschränkung auf 64KB ist neu und kann nicht hochgesetzt werden – um sie einzuhalten, kann man das XSLT-Template aufteilen. Alternativ könnte ein Third-Party-Transformer gewählt werden, der diese Beschränkung nicht hat.

Alte Klassen/

Ein Bug im Java Compiler hat Klassennamen zugelassen, die mit einem Forward Slash (/) enden, obwohl die JVM-Spezifikation dies verbietet. Der Compiler-Bug wurde in Java 1.5 behoben, aber um einen Übergang zu erlauben, hat die Laufzeitumgebung solche Klassen weiter geladen – bis Java 21: Ab jetzt setzt die Runtime die Spezifikation um und wirft in diesem Fall einen `ClassFormatError`.

Wer Klassen mit „`/`" im Namen hat, muss diese umbenennen und neu kompilieren. Sollte der Source Code solcher Klassen abhandengekommen sein, bleibt nur Bytecode-Manipulation der *.class-Dateien.

Metal 🤘

Wer Desktop-Anwendungen für macOS entwickelt, kennt sicherlich die Rendering-Pipeline Metal, die Apple OpenGL ersetzt. Sie wird von allen Apple-Geräten, die auf mindestens macOS 10.14 (Release: September 2018) laufen, unterstützt.

Seit Java 19 verwendet auch Java standardmäßig Metal. Anwendungen, für die das ein Problem ist, können die Systemeigenschaft `sun.java2d.opengl` auf `true` setzen, um weiterhin OpenGL zu verwenden.

Obsolete Optionen für G1

Im G1 Garbage Collector wurden Remembered Sets entfernt und Concurrent Refinement Threads stark überarbeitet, weshalb folgende VM-Optionen nicht mehr benötigt werden:

- `G1RSetRegionEntries`
- `G1RSetSparseRegionEntries`
- `G1UseAdaptiveConcRefinement`
- `G1ConcRefinementGreenZone`
- `G1ConcRefinementYellowZone`
- `G1ConcRefinementRedZone`
- `G1ConcRefinementThresholdStep`
- `G1ConcRefinementServiceIntervalMillis`

Aktuell führt ihre Verwendung zu einer Warnung, bald wird es ein Fehler sein.

Kommandozeilenwerkzeuge

Bei den Kommandozeilenwerkzeugen `javac`, `jar`, etc. hat es einige kleine Änderungen gegeben:

- Kompilierung mit `lint:serial` erzeugt neuerdings eine Warnung, falls ein serialisierbarer Typ (Instanz-)Felder hat, die weder serialisierbar noch transient sind.
- Das `jar` -Tool erzeugt keinen Index mehr und die Verwendung der Option `--generate-index` führt zu einer Warnung. Sollte eine JAR einen Index enthalten, ignoriert die Laufzeitumgebung ihn.
- Die `jlink`-Option `--compress` erwartet nicht mehr abstrakte Werte 0, 1, 2, sondern die konkreteren zip0 bis zip9 (Standardwert ist zip6), die direkt den ZIP-Kompressionsstufen entsprechen.
- Aufrufe von `jpackage --app-image` prüfen nun mehr Vorbedingungen (wie beispielsweise die Anwesenheit von `.jpackage.xml`), was dazu führen kann, dass inkorrekte Befehle nun einen Fehler werfen, statt ein fehlerbehaftetes Paket zu erzeugen.

Kleines Update-How-to

Das sind also all die Hürden, die einem beim Update begegnen können. Beziehungsweise die meisten – falls etwas anderes auftritt, enthalten die Release Notes der jeweiligen Versionen, beispielsweise die zu Java 21 *[1]*, diese und noch viel mehr Details und sollten bei Upgradehürden immer der erste Anlaufpunkt sein.

Falls sich bei einem Schritt von 17 zu 21 Probleme ergeben, kann es sehr hilfreich sein, die genaue Version zu finden, in der das Problem das erste Mal auftritt – zum Beispiel um die Release Notes danach zu durchsuchen. Um das herauszufinden, kann man in kleinen Schritten von 17 zu 18 zu 19 usw. updaten (alte JDK Builds finden sich im Archiv *[2]*). Aber nur in der Entwicklung und im Build, nicht in Produktion! Versionen 18, 19 und 20 erhalten keine Security-Updates mehr und können bekannte und veröffentlichte Sicherheitslücken enthalten.

Beim Update eines Projekts geht es aber nicht nur um den eigenen Code, sondern auch um Dependencies und die können über die glei-

chen Hürden stolpern. Deshalb ist der beste erste Schritt immer, Dependencies und Tools zu aktualisieren, bevor man die Java-Version hochzieht. Tatsächlich ist das Java-Update danach oft ziemlich einfach. Dann kann man endlich all die tollen neuen Features einsetzen – viel Spaß dabei!

Referenzen:

[1] *https://www.oracle.com/java/technologies/javase/21-relnote-issues.html*

[2] *https://jdk.java.net/archive/*

Nicolai Parlog

Oracle

Nicolai (aka nipafx) ist ein Java-Enthusiast mit Fokus auf Sprachfeatures und APIs, der leidenschaftlich gerne lernt und lehrt. Das macht er in Blog-Posts, Newslettern und Büchern; in Tweets, Videos und Streams; in Demo Repositories und auf Konferenzen – mehr dazu auf nipafx.dev. Er ist Java Developer Advocate bei Oracle und Organisator von Accento.

CloudLand

WWW.CLOUDLAND.ORG

Eventpartner: Heise Medien

DAS CLOUD NATIVE FESTIVAL

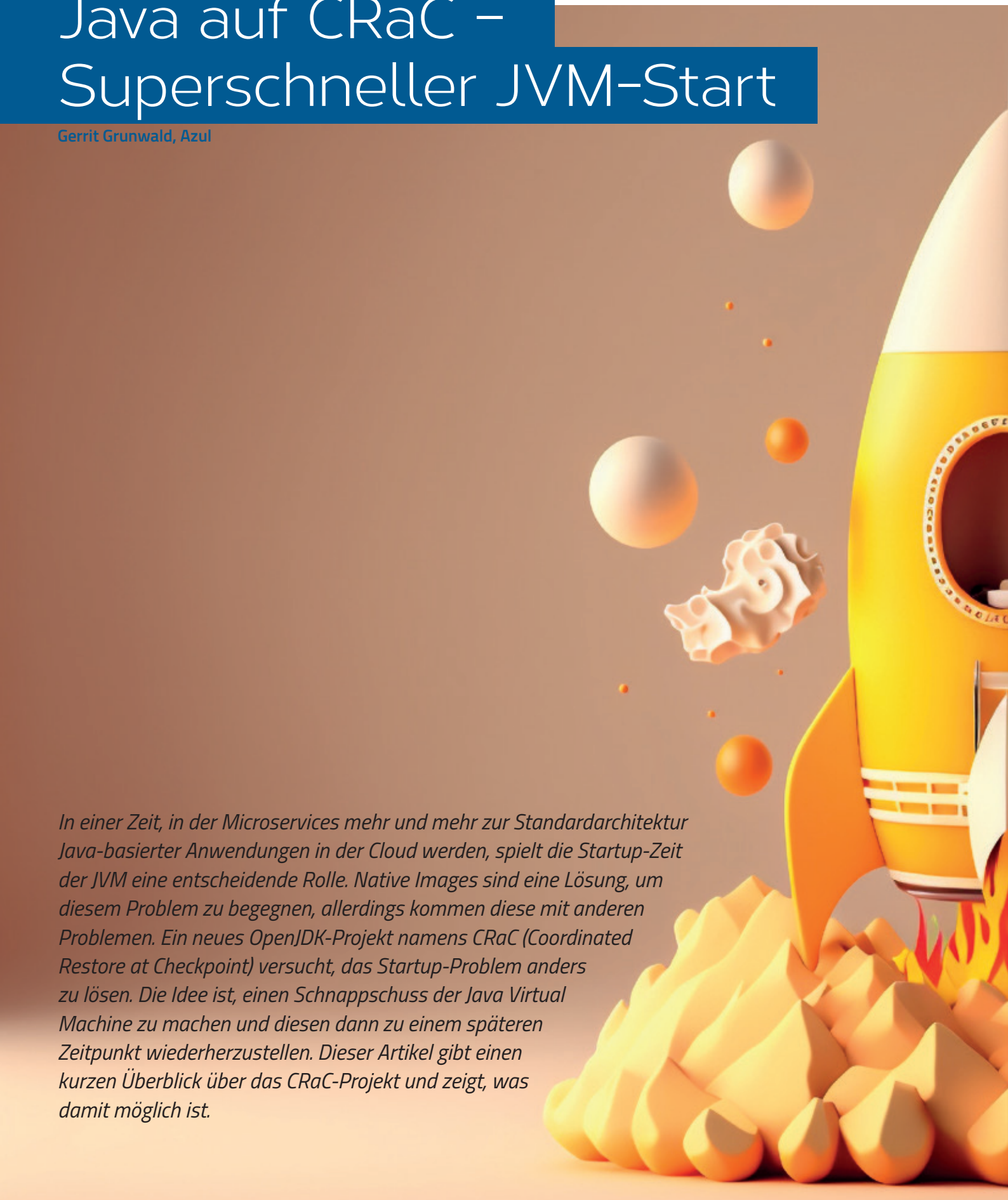
DAS EVENT DER DEUTSCHSPRACHIGEN CLOUD NATIVE COMMUNITY

18. – 21. JUNI

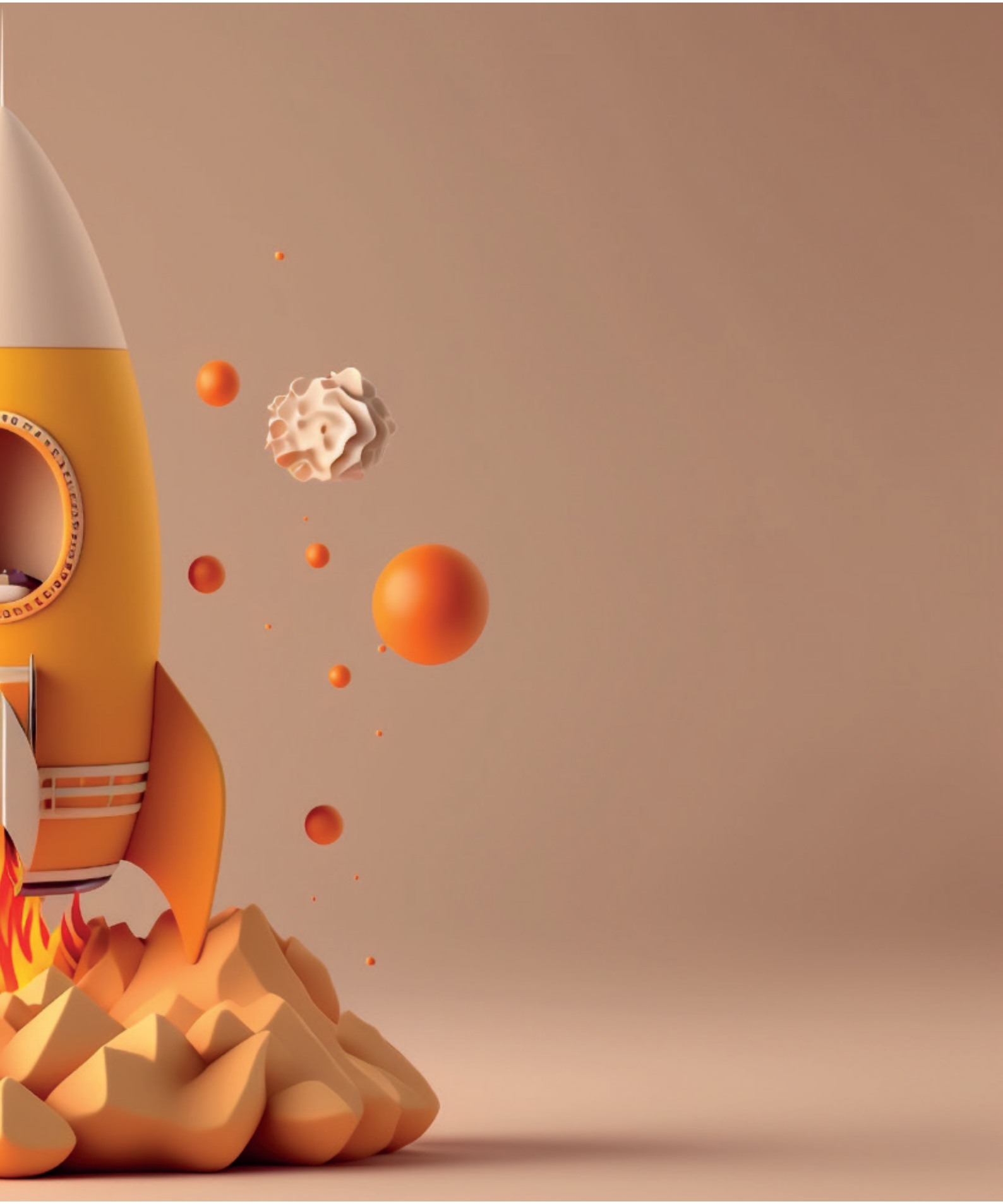
#CLOUDLAND2024

Java auf CRaC – Superschneller JVM-Start

Gerrit Grunwald, Azul



In einer Zeit, in der Microservices mehr und mehr zur Standardarchitektur Java-basierter Anwendungen in der Cloud werden, spielt die Startup-Zeit der JVM eine entscheidende Rolle. Native Images sind eine Lösung, um diesem Problem zu begegnen, allerdings kommen diese mit anderen Problemen. Ein neues OpenJDK-Projekt namens CRaC (Coordinated Restore at Checkpoint) versucht, das Startup-Problem anders zu lösen. Die Idee ist, einen Schnappschuss der Java Virtual Machine zu machen und diesen dann zu einem späteren Zeitpunkt wiederherzustellen. Dieser Artikel gibt einen kurzen Überblick über das CRaC-Projekt und zeigt, was damit möglich ist.



Die *Java Virtual Machine* (JVM) ist ein faszinierendes Stück Technik, das sich seit nunmehr 27 Jahren erfolgreich in der Industrie in zahlreichen Anwendungen bewiesen hat. In der Zeit, als die JVM entwickelt wurde, war so etwas wie die Cloud noch undenkbar, Anwendungen wurden auf einem Server (meist sogar mit einem einzelnen Prozessor) betrieben und liefen in der Regel relativ lange (Tage bis Monate).

Im Gegensatz dazu sind in modernen Cloud-Architekturen Anwendungen nicht mehr monolithisch aufgebaut, sondern bestehen aus einer Vielzahl sogenannter Microservices. Eine Anwendung wird also in viele Teile „zerlegt" und für jeden Teil ein eigener Microservice entwickelt. Diese Microservices laufen unabhängig voneinander und haben teilweise sehr kurze Laufzeiten. Das Ganze wird in skalierbaren Cloud-Diensten betrieben, in denen die Microservices je nach Bedarf in vielen Containern gestartet und beendet werden.

Sind solche Microservices in Java programmiert, bedeutet das, dass auch die JVM bei jedem Start „hochgefahren" werden muss. Um zu verstehen, warum das problematisch sein kann, müssen wir uns anschauen, wie die JVM funktioniert (zumindest zum Teil).

Java Virtual Machine

In einem vereinfachten Schaubild soll verdeutlicht werden, was in der JVM abläuft, wenn unser Java-Code ausgeführt wird *(siehe Abbildung 1)*. Der Java-Code wird von einer `.java`-Text-Datei durch den javac-Compiler in ein plattformübergreifende `.class`-Datei kompiliert.

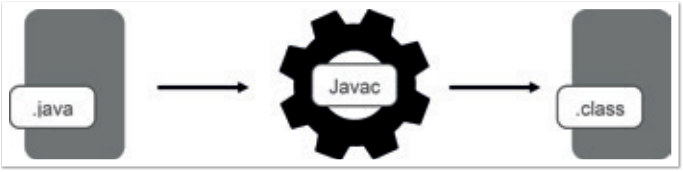


Abbildung 1

kein Bindestrich? class-Datei

Diese `.class` Datei enthält den sogenannten Bytecode, der von einer JVM ausgeführt werden kann. Dazu wird nun die `.class`-Datei von dem *ClassLoader* in den Speicher der JVM geladen *(siehe Abbildung 2)*. Dies ist eine sehr vereinfachte Sichtweise und ist in der Realität wesentlich komplexer, reicht allerdings für das Grundverständnis aus.

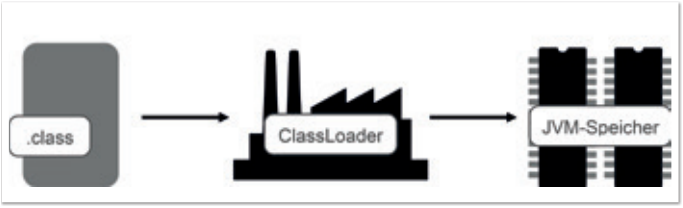


Abbildung 2

Vom JVM-Speicher gelangt der Code dann in die *Execution Engine (siehe Abbildung 3)*. Hier wird es nun interessant. Die *Execution Engine* besteht aus mehreren Teilen:

- *Interpreter*
- *C1 JIT Compiler (client)*
- *C2 JIT Compiler (server)*
- *Profiler*
- *Garbage Collector*

Für diesen Artikel wollen wir uns lediglich mit den linken drei Bereichen befassen, dem Interpreter und den beiden *JIT-(Just-in-Time-) Compilern*.

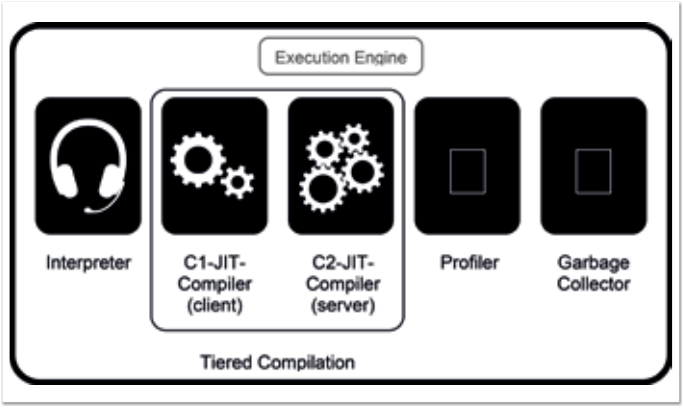


Abbildung 3

Execution Cycle

Der *Bytecode* gelangt zunächst aus dem JVM-Speicher in den *Interpreter*, in dem er Zeile für Zeile in den Maschinencode der jeweiligen Prozessorarchitektur übersetzt wird. Die Ausführgeschwindigkeit von Code, der interpretiert wird, ist allgemein hin langsamer als die von kompiliertem Code – Warum also der Interpreter?

Interpretierung macht es möglich, sehr schnell zu starten, da der Code nicht erst kompiliert werden muss, sondern zeilenweise interpretiert und ausgeführt wird. Sobald aber viel Code ausgeführt werden muss, und dieser zum Beispiel Schleifen und ähnliches enthält, wird das Ganze dann doch langsam, da der gleiche Code wieder und wieder interpretiert und ausgeführt werden muss.

Die Idee ist also, nun auf die Laufzeit zu schauen, wie oft bestimmte Methoden ausgeführt werden und anhand der Anzahl der Aufrufe zu entscheiden, ob Code weiterhin interpretiert oder kompiliert werden soll. Dabei werden Methodenaufrufe sowie Schleifendurchläufe gezählt. Erreicht der Zähler 1.000 im interpretierten Mode, wird der „hot" Code an den C1-Compiler (früher auch Client-Compiler) „weitergereicht", der den Code dann zu Maschinencode kompiliert.

Dabei hat der C1-Compiler den Vorteil, dass er Code sehr schnell – allerdings ohne viele Optimierungen anzuwenden – in Maschinencode kompilieren kann. Aus diesem Grund wurde er früher auch überwiegend für die clientseitige Ausführung von Programmen, sprich Desktop-Applikationen, verwendet. Auf dem Client ist die Startzeit von Applikationen ein wesentlicher Faktor für gute Usability.

Der kompilierte Code wird nun durch den Profiler „beobachtet" und es wird auch hier weiterhin verfolgt, wie oft dieser Code ausgeführt wird. Lag der Threshold beim Interpreter noch bei 1.000, so liegt er nun bei 5.000. Wird also der bereits kompilierte Code mehr als 5.000-mal ausgeführt (Methodenaufrufe oder Schleifendurchläufe), so wird der bereits kompilierte Code dem C2-Compiler zugeführt.

Der C2-Compiler verwendet die bereits gesammelten Daten vom Profiler über den C1-Code, um diesen hochgradig zu optimieren. Aus diesem Grund wurde der C2-Compiler in früheren JDKs auch überwiegend auf der Serverseite eingesetzt, da es hier nicht so sehr auf die Startzeiten der Applikation ankam, sondern eher auf die Performance des kompilierten Codes.

Seit JDK 8 sind nun aber beide Compiler immer gemeinsam im Einsatz in der JVM, was auch als *Tiered Compilation* bezeichnet wird. Die Java Virtual Machine entscheidet also zur Laufzeit, welcher Code kompiliert und optimiert wird, und kann sich somit perfekt auf die Bedürfnisse der jeweiligen Anwendung anpassen.

Abbildung 4 zeigt noch einmal den Ablauf auf einen Blick. Im Schaubild ist nun noch ein weiterer Schritt im Ablauf zu sehen, die sogenannte Deoptimisation (oder Deoptimierung).

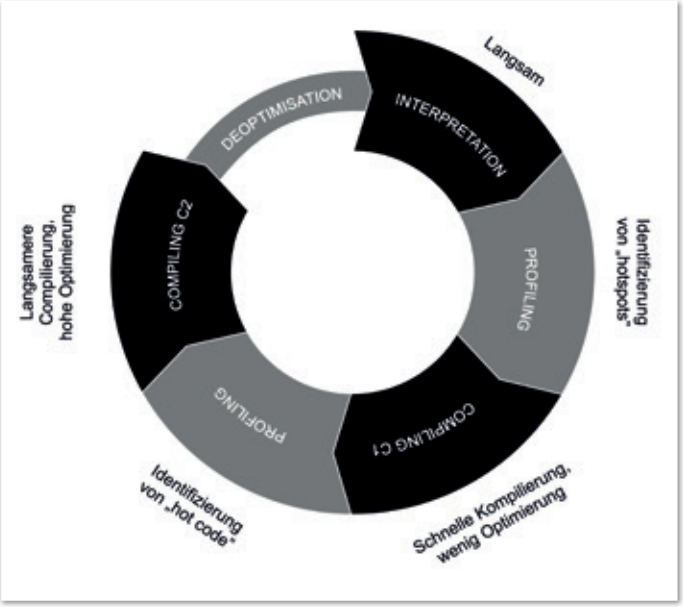


Abbildung 4

Deoptimierung

Die Deoptimierung ist wesentlich an der Performance der JVM beteiligt, sie ermöglicht es, bereits kompilierten und optimierten Code „wegzuschmeißen", erneut zu interpretieren und zu kompilieren.

Es stellt sich sofort die Frage, warum denn bereits kompilierter und optimierter Code noch einmal interpretiert und kompiliert werden müsse?

Das liegt an den sogenannten spekulativen Optimierungen, die der C2-Compiler durchführt. Dabei wird Code zur Laufzeit analysiert und durch Spekulationen optimiert, was zu erheblichen Geschwindigkeitssteigerungen führen kann.

Ein Beispiel für spekulative Optimierung ist die *Branch-Analysis*. Wenn wir uns den Code in *Listing 1* anschauen, sehen wir eine Methode, in der eine `if-else`-Abfrage vorkommt.

Nehmen wir nun an, dass der Code im `if`-Block niemals ausgeführt wird, da der an die Methode übermittelte Wert immer kleiner oder

```
int computeMagnitude(int value) {
      int bias;
      if (value > 9) {
            bias = compute(value);
      } else {
            bias = 1;
      }
      return Math.log10(bias + 99);
}
```

Listing 1: Methode mit if-else-Verzweigung vor spekulativer Optimierung

gleich 9 ist, so kann der C2-Compiler den Code wie folgt optimieren: Zunächst wird für den Aufruf von `bias = compute(value)` ein Aufruf an `uncommonTrap()` eingeführt. Weiterhin kann man den Ausdruck `return Math.log10(bias + 99)` für `bias = 1` zu folgendem Ausdruck `return 2` optimieren, da der Logarithmus von 100 zur Basis 10 = 2 ergibt.

Durch diese Optimierungen sieht unsere Methode nun aus wie in *Listing 2* gezeigt.

```
int computeMagnitude(int value) {
      if (value > 9) {
            uncommonTrap();
      }
      return 2;
}
```

Listing 2: Methode nach spekulativer Optimierung

Es ist unschwer zu erkennen, dass die Ausführung der optimierten Methode wesentlich schneller ist als die Ausführung der originalen Methode.

Zur Deoptimierung kommt es nun, wenn plötzlich doch ein Wert an die Methode übergeben wird, der größer als 9 ist. In diesem Fall wird der bereits kompilierte und optimierte Code verworfen und die originale Methode wieder interpretiert und optimiert, nun sowohl für den `if`- als auch den `else`-Block.

JVM-Startup

Wenn wir uns nun dem Start der Java Virtual Machine zuwenden, so lässt sich dieser in mehrere Bereiche unterteilen:

- Start der JVM selbst
- Start der Applikation
- Aufwärmen der Applikation

Abbildung 5 zeigt, was man unter den einzelnen Bereichen versteht. Das bedeutet, die reine JVM startet sehr schnell (wenige Millisekunden), das Starten der Anwendung dauert dagegen schon länger, da alle Klassen geladen, alle Ressourcen initialisiert und die Anwendungslogik ausgeführt werden müssen. Der Zeitraum bis hierhin wird im allgemeinen als *JVM-Startup* bezeichnet, oder auch als die Zeit bis zur ersten Antwort unserer Anwendung.

Das Antworten unserer Anwendung bedeutet jedoch nicht gleichzeitig, dass unsere Anwendung vollkommen kompiliert und opti-

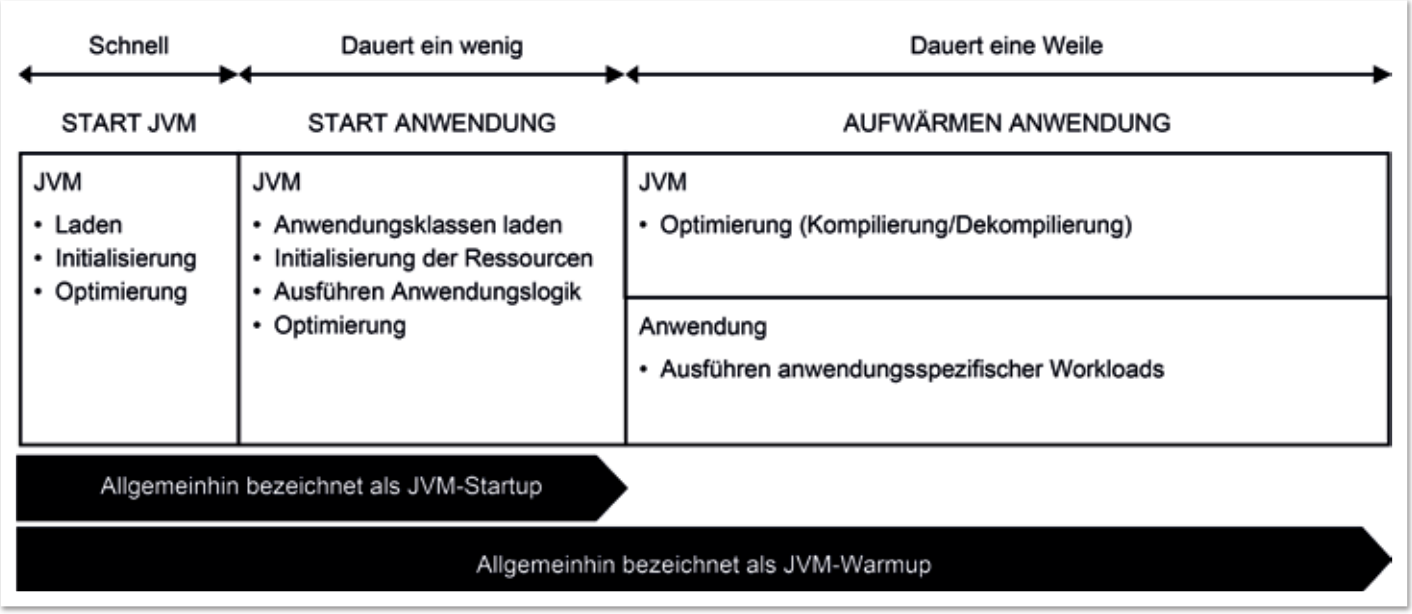


Abbildung 5

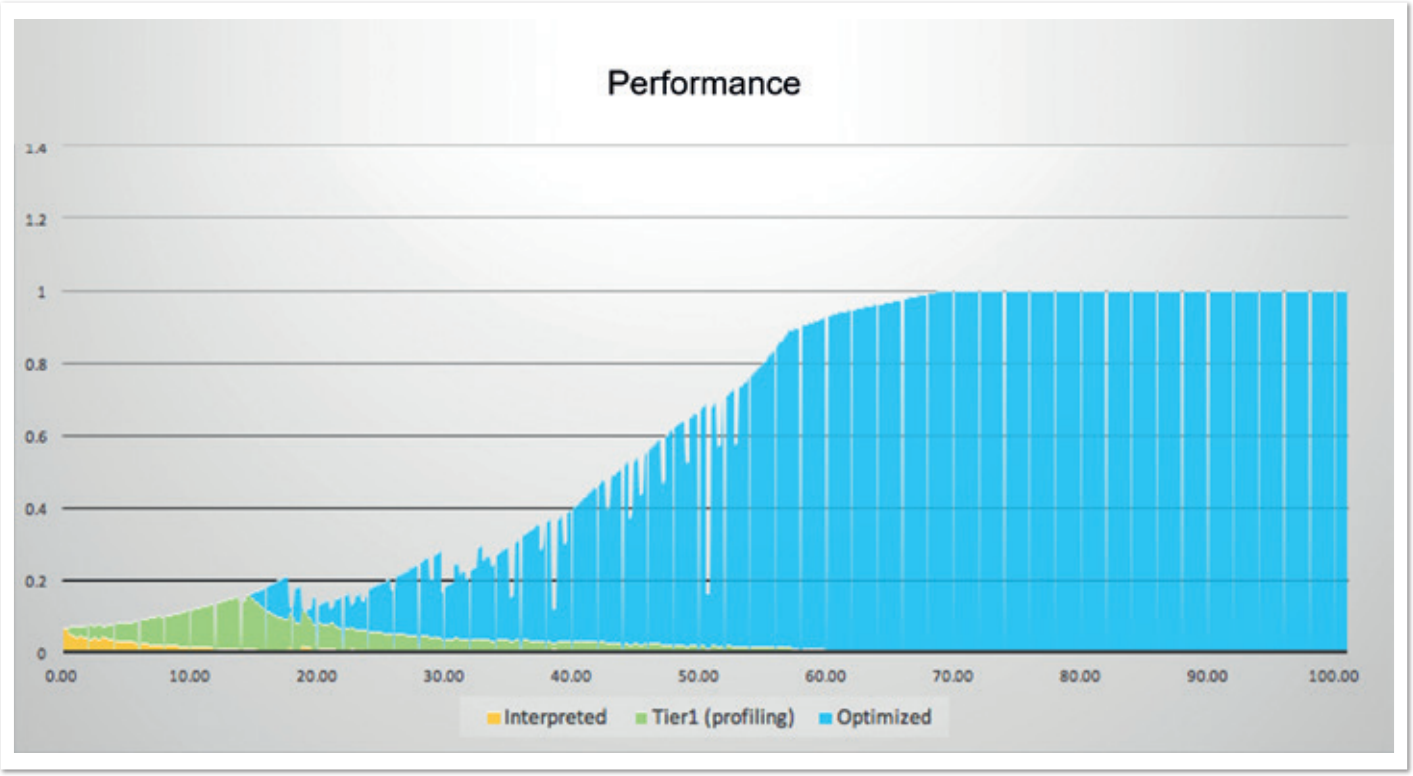


Abbildung 6

miert ist. Dieser Vorgang wird auch als *Application-Warmup* bezeichnet und bedeutet, dass möglichst viel Code unserer Anwendung „angefasst" und von den Compilern kompiliert und optimiert wird. Erst dann läuft unsere Anwendung mit voller Performance. Dieser gesamte Zeitraum wird dann als *JVM-Warmup* bezeichnet.

Das Problem mit dem Startup der JVM in einer von Microservices bestimmten Umgebung ist somit darin begründet, dass zunächst der Code geladen werden muss, dann eine Zeit laufen muss, um optimiert werden zu können, und dann erst mit voller Performance ausgeführt werden kann. Und das jedes Mal, wenn man die Anwendung startet, da die JVM kein Gedächtnis hat und somit nichts von ihren vorherigen Optimierungen weiß. Skaliert man also nun einen Microservice mit einer kurzen Laufzeit, so muss jeder Microservice immer wieder aufs Neue interpretiert, kompiliert und optimiert werden und das kostet Zeit.

Abbildung 6 zeigt noch einmal die Anteile von Interpreter (gelb), C1- (grün) und C2-Compiler (blau) in Bezug auf Laufzeit und Performance.

CRaC

Das OpenJDK-Projekt namens CRaC *(Coordinated Restore at Checkpoint)* soll genau hier Abhilfe schaffen. Es wurde von der Firma Azul entwickelt und dem OpenJDK-Projekt zugeführt *[1]*.

Die Idee ist relativ simpel: Man startet seine Anwendung und lässt sie eine Zeit laufen, wobei man am besten möglichst viele Bereiche der Anwendung aufruft, um sicherzustellen, dass so viel Code wie möglich kompiliert und optimiert wird. Dann erstellt man einen sogenannten *Checkpoint*, von dem man zu einem späteren Zeitpunkt die Anwendung wieder startet.

Das Prinzip vom Checkpoint-Restore ist nicht neu und die meisten kennen und nutzen es vermutlich täglich. Wenn man einen Laptop hat und das Display herunterklappt, während Anwendungen laufen, so empfinden wir es mittlerweile als normal, dass beim Aufklappen des Displays die Anwendungen genau dort weiterlaufen, wo wir sie verlassen haben, als wir den Laptop zugeklappt haben.

Hier passiert Folgendes: Das Betriebssystem registriert das Zuklappen des Laptops und speichert den aktuellen Zustand des Systems inklusive aller laufenden Anwendungen in einem Checkpoint. Nach dem Aufklappen des Laptops „findet" das Betriebssystem diesen Checkpoint (in Form von Dateien auf der Festplatte), lädt diesen und stellt aus den Dateien den Zustand vor dem Zuklappen wieder her.

CRaC versucht nun dasselbe auf JVM-Level. Dabei gibt es jedoch einige Dinge zu beachten. Die JVM ist ja eine Anwendung, die auf einem Betriebssystem, zum Beispiel Linux, läuft. Wir benötigen also ein Tool, das in der Lage ist, Checkpoints von Anwendungen zu erstellen. Unter Linux gibt es mehrere Tools, die diese Funktionalität anbieten, das wohl populärste von ihnen heißt CRIU *(Checkpoint Restore In Userspace)* und ist schon seit 2013 Teil des Linux-Kernels.

CRIU selbst ist erstellt worden, um Anwendungen oder Container „einzufrieren", einen Checkpoint zu erstellen und diesen zu einem späteren Zeitpunkt wiederherzustellen. Das Tool ist teilweise auch integriert oder wird verwendet von Applikationen wie OpenVZ, LXC/LXD, Docker, Podman und anderen.

Probleme

Leider gibt es unter Windows und MacOS kein solches Tool, was den Einsatz von CRaC zurzeit nur auf Linux-basierten Systemen ermöglicht. Jetzt muss man aber auch sagen, dass die Mehrzahl der Server auf Linux laufen und das Problem mit Windows und MacOS nicht wirklich eine Hürde für den Einsatz von CRaC darstellt.

Man sollte nun annehmen, da ja die JVM auch „nur" eine Anwendung unter Linux ist, dass man einen Checkpoint mit CRIU davon erstellen kann und diesen später wiederherstellen kann. Des Weiteren sei erwähnt, dass man zurzeit mit CRaC nur Anwendungen checkpointen kann, die kein GUI einsetzen, sprich keine Desktopanwendungen. Das könnte sich eventuell in der Zukunft noch ändern, ist aber momentan nicht möglich.

Im Prinzip *wäre* es möglich, das Problem liegt dabei allerdings nicht bei der JVM, sondern bei der Anwendung, die in der JVM läuft. Die Anwendung verwendet Ressourcen wie beispielsweise Dateien, in die sie schreibt oder aus denen sie liest, oder Datenbankverbindungen, die verwendet werden. Beim Erstellen des Checkpoints würden diese Verbindungen getrennt und wenn man dann die JVM inklusive der Anwendung aus eben diesem Checkpoint wiederherstellen möchte, würde die Anwendung davon ausgehen, dass diese Ressourcen immer noch zur Verfügung stehen. Eben genau wie zu dem Zeitpunkt, als der Checkpoint erstellt wurde. Es würde also zu einem Absturz der Anwendung kommen, da die Ressourcen nicht mehr zur Verfügung stehen. Genau dort setzt CRaC an, wie der Name schon besagt, handelt es sich um einen koordinierten Restore nach einem Checkpoint.

Um das Problem mit den Ressourcen zu lösen, erfordert CRaC, dass man an einigen Stellen im Sourcecode ein Interface vom Typ `Resource` implementiert, das genau zwei Methoden hat: `beforeCheckpoint()` und `afterRestore()`.

Mithilfe dieses Interfaces ist es nun möglich, vor Erstellung eines Checkpoints Ressourcen koordiniert zu schließen (in der `beforeCheckpoint()`-Methode) und diese nach einem Restore wiederherzustellen (in der `afterRestore()`-Methode). Die Implementierung des Resource Interface ist nur dort nötig, wo Ressourcen, wie zum Beispiel Dateien oder jegliche Art von Socket-Verbindungen, verwendet werden (etwa Datenbankverbindungen).

Damit die JVM weiß, wo die Dateien eines Checkpoints gespeichert werden sollen, muss man der JVM beim Starten der Anwendung den entsprechenden Pfad mitgeben. Das geschieht mit dem Kommandozeilenparameter `-XX:CRaCCheckpointTo=PATH`, wobei `PATH` den Pfad bezeichnet, in dem später die Dateien des Checkpoints gesichert werden. Der Start einer Anwendung sähe dann beispielsweise wie folgt aus: `java -XX:CRaCCheckpointTo=/Users/hansolo/checkpoints -jar myapp.jar`.

Hier werden dann die Dateien des Checkpoints in dem Home-Ordner des Users „hansolo" und dort im Ordner „checkpoints" abgelegt. Möchte man nun diesen Checkpoint wiederherstellen, so verwendet man den Kommandozeilenparameter `-XX:CRaCRestoreFrom=PATH`, im Falle unseres Beispiels also `java -XX:CRaCRestoreFrom=/Users/hansolo/checkpoints`.

Bei diesem Aufruf wird die JVM prüfen, ob in dem entsprechenden Verzeichnis Dateien eines Checkpoints liegen und diesen zurück in den Speicher laden.

Das Erstellen eines Checkpoints kann man entweder über die Kommandozeile mit dem Kommando `jcmd` oder über Code ausführen.

Bei der Verwendung von `jcmd` gibt es zwei Möglichkeiten: Entweder man verwendet den Namen des Anwendungs-jar oder die Prozess-ID (PID) des Anwendungsprozesses beim Aufruf des Kommandos `JDK.checkpoint`:

- `jcmd myapp.jar JDK.checkpoint`
- `jcmd PID JDK.checkpoint`

Der Aufruf des Checkpoint-Kommandos führt dazu, dass die JVM alle Dateien, die das Resource Interface implementiert haben und die im sogenannten *Global Context* registriert sind, in der umgekehrten Reihenfolge aufruft, in der sie registriert wurden. Man muss also im Code nicht nur das Resource Interface implementieren, sondern danach jede Implementierung im Global Context registrieren. Dies lässt sich über das Kommando `Core.getGlobalContext().register(RESOURCE)` im Code durchführen.

```
public class Main implements Resource {

      public Main() {
            System.out.println("Start without CRaC");

            Core.getGlobalContext().register(Main.this);

            init();

            printRandomGirlNames(5);
            printRandomBoyNames(5);

            System.out.println("Time to first response: " + ((System.nanoTime() - startTime) / MILLISECOND_IN_NS) + "ms");
      }

      private void init() {
            allNames = loadNames();
      }

      @Override public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {}

      @Override public void afterRestore(Context<? extends Resource> context) throws Exception {}
```

Listing 3: NamenService-Main-Methode (normaler Start)

Um es so einfach wie möglich zu gestalten, sehen wir uns eine Demo an, die einen `NamenService` darstellen soll *(siehe Listing 3)*. Der Service lädt beim Start 258.000 Vornamen aus einer JSON-Datei in den Speicher und liefert danach 5 zufällig ausgewählte Namen für Mädchen und 5 für Jungen. Natürlich würde man das in der Realität so nicht machen, aber für die Demo erfüllt es den Zweck.

Wie man sehen kann, implementieren wir hier das Resource Interface (auch wenn es hier eigentlich nicht nötig wäre) und wenn wir uns den Ablauf anschauen, so wird zunächst die `Resource` im `GlobalContext` registriert und dann wird die `init()`-Methode aufgerufen, in die alle Namen geladen werden. Danach werden aus den Namen die 5 zufällig gewählten Mädchen- und Jungen-Namen ausgegeben.

Beim normalen Start werden also jedes Mal zunächst die Namen aus der JSON-Datei geladen, bevor die zufällige Auswahl der Namen beginnt.

Wenn wir nun einen Checkpoint erstellen, nachdem die Anwendung gestartet wurde, und dann die Anwendung aus dem Checkpoint zurück in den Speicher laden, so wird die `afterRestore()`-Methode ausgeführt, die wir in *Listing 4* sehen.

Wie man sehen kann, werden beim Aufruf der `afterRestore()`-Methode die Namen nicht mehr geladen, da sie bereits vor dem Checkpoint in den Speicher geladen worden sind und somit zu diesem Zeitpunkt schon zur Verfügung stehen. Das heißt, wir können sofort die zufällig gewählten Namen ausgeben. Der Restore aus dem Checkpoint erspart uns also die Ladezeit der JSON-Datei. Wenn wir uns beide Starts nebeneinander anschauen, dann sehen wir, dass uns der Restore aus dem Checkpoint in diesem Fall zirka 1,3 Sekunden gespart hat *(siehe Abbildung 7)*.

Der komplette Code für das Beispiel kann auch heruntergeladen werden *[2]*.

Das Laden der JSON-Datei kann auch als Platzhalter für sämtliche Aufgaben stehen, die nach einem Programmstart ausgeführt wer-

```
public class Main implements Resource {

  public Main() {}

  private void init() {}

  @Override public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {}

  @Override public void afterRestore(Context<? extends Resource> context) throws Exception {
     System.out.println("Start using CRaC");

     startTime = System.nanoTime();

     printRandomGirlNames(5);
     printRandomBoyNames(5);

     System.out.println("Time to first response: " + ((System.nanoTime() - startTime) / MILLISECOND_IN_NS) + "ms");

  }
```

Listing 4: NamenService afterRestore()-Methode (Start aus Checkpoint)

Abbildung 7

den müssen, bevor eine Anwendung vollständig optimiert ist (Application-Warmup).

Durch den Einsatz von CRaC lässt sich also die Zeit für den Application-Warmup einsparen. Im Prinzip hängt es nur davon ab, zu welchem Zeitpunkt ich meinen Checkpoint erstelle. In diesem Falle gilt: je später, desto besser.

Checkpoint ja, aber wann?

Den richtigen Zeitpunkt für einen Checkpoint zu finden ist nicht einfach und hängt sehr von der Anwendung ab. Als Faustregel gilt, je mehr Code „angefasst" beziehungsweise kompiliert und optimiert wird, desto mehr Zeit kann ich später beim Restore sparen.

Eine Möglichkeit ist, alle vorhandenen Tests durchlaufen zu lassen, da diese im Idealfall möglichst alle Methoden aufrufen. Bei manchen Anwendungen ist es allerdings nötig, die Anwendung mit realistischen Daten „aufzuwärmen", um eine ideale Performance zu erzielen. In diesem Fall muss man dafür sorgen, dass man ebensolche Daten zur Verfügung hat. Wer wirklich gar keine Ahnung hat, wie man den besten Zeitpunkt für einen Checkpoint findet, der kann die Anwendung mit dem Parameter `-XX:+PrintCompilation` versehen. Danach lässt man die Anwendung laufen (am besten unter Verwendung eines typischen Workloads) und man wird auf der Konsole eine Ausgabe sämtlicher Kompilierungen sehen. Direkt nach dem Start ist die Anzahl hoch und es scheint gar nicht aufzuhören, zu scrollen, doch nach einiger Zeit wird man feststellen, dass die Anzahl der Kompilierungen abnimmt (da mehr und mehr Code bereits kompiliert und optimiert wurde). Der Zeitpunkt, an dem es nur noch zu wenigen Kompilierungen kommt, ist geeignet, um einen Checkpoint zu erstellen.

Typische Verwendung von CRaC

Typischerweise bietet sich die Verwendung von CRaC für alle Anwendungen an, die häufig neu gestartet werden müssen. Das sind im Regelfall Microservices, die in Containern laufen. Es ist jedoch auch möglich, monolithische Anwendungen mit CRaC beim Start zu beschleunigen.

Auch für Mac und Windows gibt es eine Lösung. Wie zuvor erwähnt, steht CRaC nur unter Linux zur Verfügung. Es gibt allerdings eine Bibliothek *[3]*, deren Sourcecode zur Verfügung steht und die das API im JDK kapselt. Programmiert man nun gegen dieses API anstatt des JDK-API, so kann man auch unter Windows und Mac gegen CRaC programmieren, ohne dass es zu Fehlermeldungen in der Entwicklungsumgebung kommt. Dazu fügt man die Bibliothek einfach zu den Abhängigkeiten des Projekts hinzu.

Gradle

```
Implementation 'org.crac:crac:0.1.3'
```

Maven

```
<dependency>
  <groupId>org.crac</groupId>
  <artificactId>crac</artifactId>
  <version>0.1.3</version>
</dependency>
```

Frameworks

Wirklich interessant wird der Einsatz von CRaC, wenn Frameworks wie Spring (Boot), Quarkus oder Micronaut CRaC unterstützen. Die gute Nachricht ist, dass Micronaut bereits sehr guten Support bietet, Quarkus rudimentären Support und Spring mit der nächsten Version 6.1 Ende 2023 vollen Support für CRaC bieten wird. Der Vorteil ist, dass man dann oft nur eine Annotation an die jeweilige Klasse anfügt und das Framework das Handling der Ressourcen übernimmt.

Performance

Jetzt haben wir viel über CRaC gelesen, doch wie gut ist es denn nun wirklich? Dazu haben wir ein paar Tests gemacht, die sich auch alle unter *[4]* finden. Bei diesen Tests wurde eine Anwendung unter verschiedenen Frameworks gestartet, dann ein Checkpoint erstellt und

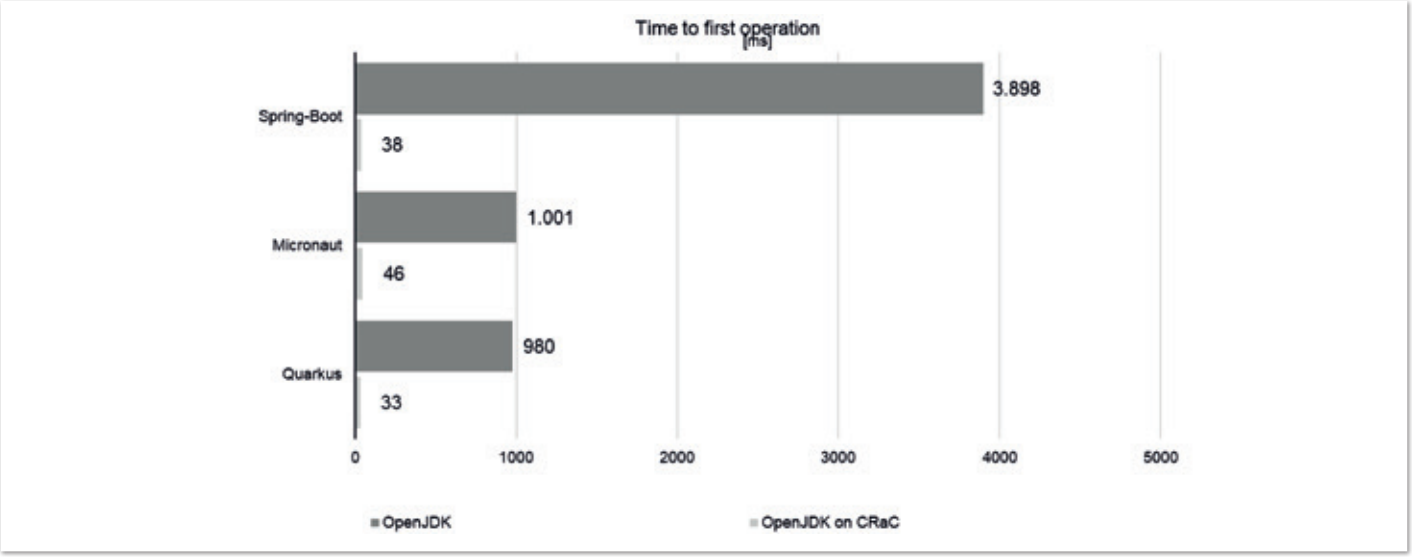


Abbildung 8

die Anwendung aus dem Checkpoint wiederhergestellt. Dabei wurde jeweils die Zeit bis zur ersten Antwort der Anwendung gemessen.

Wie man sehen kann *(siehe Abbildung 8)*, sind die Startzeiten beim Restore erheblich kürzer als die regulären Startzeiten. Interessanterweise sind die Zeiten beim Restore auch sehr ähnlich, wohingegen die regulären Startzeiten doch stark voneinander abweichen können. Das liegt daran, dass bei einem Restore lediglich die Anwendungen aus den Checkpoint-Dateien zurück in den Speicher geholt werden und so hängt die (Restore-)Startzeit dann nur noch von der Geschwindigkeit der Festplatte ab. Natürlich müssen nach dem Restore noch Ressourcen reinitialisiert werden, doch auch dies ist im Regelfall im Bereich von Millisekunden erledigt.

Fazit

CRaC ist ein Weg, eine Anwendung anzuhalten, ihren Zustand auf der Festplatte zu speichern und die Anwendung aus diesem Checkpoint später wiederherzustellen. Dabei sind zwar Änderungen im Code erforderlich, diese sind jedoch überschaubar und relativ minimalistisch. Es werden weiterhin alle Fähigkeiten des JDK unterstützt, zum Beispiel Reflection, Off-Heap-Memory und so weiter.

Durch die Verwendung von CRaC kann man kürzere Startzeiten im Bereich nativer Anwendungen erreichen – bei vollständiger Erhaltung der JVM und ihrer Fähigkeiten. Nach einem Restore aus einem Checkpoint läuft die Anwendung normal weiter und kann auch noch weiter je nach Bedarf zur Laufzeit optimiert werden.

Zurzeit steht lediglich eine kostenlose Implementierung von CRaC in der OpenJDK-Distribution Zulu der Firma Azul zur Verfügung. Diese Version erhält auch vollständigen Support von Azul und ist keine Beta-Version. Sie kann heruntergeladen *[5]* und kostenlos in der Produktion eingesetzt werden.

Wenn also die Startup-Zeiten der JVM für den ein oder anderen ein Problem darstellt, dann probiert doch einfach mal CRaC aus und gebt uns Bescheid, falls ihr Problem damit habt und welcher Art diese sind, damit wir sie möglichst schnell beheben können. Unser Ziel ist, CRaC offiziell im OpenJDK zu etablieren, sodass jede Distribution davon profitieren kann.

Referenzen:

[1] *https://openjdk.org/projects/crac*
[2] *https://github.com/HanSolo/crac8*
[3] *https://github.com/CRaC/org.crac*
[4] *https://github.com/CRaC*
[5] *https://azul.com*

Gerrit Grunwald
Azul

Gerrit Grunwald ist ein Software-Ingenieur, der sich schon seit 40 Jahren für das Programmieren begeistert. Er ist ein echter Anhänger von Open Source und hat sowohl an populären Projekten wie JFXtras.org als auch an seinen eigenen Projekten (TilesFX, Medusa, Enzo, SteelSeries Swing, SteelSeries Canvas, JDKMon) mitgewirkt.

Gerrit ist ein aktives Mitglied der Java-Community, wo er die Java User Group Münster (Deutschland) gegründet hat und leitet, er ist ein JavaOne Rockstar und Java Champion. Zudem spricht er auf internationalen Konferenzen und User Groups und schreibt für verschiedene Magazine.

Einfache Unit-Tests mit 7 Zeilen Kotlin-Code

Martin Dilger, NebulIT GmbH

99 % aller Entwickler lieben es, produktiv zu sein. Das Schreiben von Tests gehört meistens aber nicht unbedingt in diese Kategorie. Warum? Boilerplate-Code, Copy & Paste, redundante Test-Setups – drei Dinge, die womöglich jeder Entwickler hasst. Dieser Artikel zeigt einen einfachen Ansatz mit Kotlin, mit dem ein Großteil des Setup-Codes in Unit-Tests einfach wegfällt.

Guter Code muss einfach sein. Du liest den Code und weißt, ohne wirklich nachzudenken direkt, was der Code tun sollte. Ich vergleiche guten Code gerne mit einem traditionellen Tweet (auch wenn Tweets mittlerweile länger sind und gar nicht mehr Tweets heißen). Um deine Nachricht in 140 Zeichen unterzubringen, lässt du alles Unnötige weg, kürzt das, was noch da ist und formatierst den Rest in lesbare Häppchen.

Im Code entspricht das ein paar sprechenden Interfaces, einer modularen Struktur und einigen Klassen in „Häppchen"-Größe. Meistens hört der Zauber hier aber auf.

Die Unit-Tests, die die Funktionalität des Codes sicherstellen und als lebende Dokumentation dienen sollten, werden im Gegensatz zum Code bestenfalls stiefmütterlich behandelt. Ich sehe erstaunlich oft guten Code in Projekten. Selten trifft diese Aussage allerdings auf die zugehörigen Tests zu. Weil es schnell gehen muss, wird der Setup-Code für einen neuen Test allzu oft aus anderen ähnlichen Tests kopiert. Mit Mühe und Not werden kleine *Refactorings* gemacht und Teile davon in statische Factory-Methoden ausgelagert.

Dass dadurch unsichtbare Abhängigkeiten zwischen den Tests entstehen, bemerkt man oft erst, wenn es zu spät ist. Die Krönung des Ganzen ist dann die Auslagerung der Testfactories in die „Test-Library", damit alle Services von dieser Arbeit profitieren. Hinzu kommt, dass Tests im Großen und Ganzen immer noch nach der Implementierung geschrieben werden, wenn grundsätzlich alles fertig ist. Gefühlt ist der Zeitaufwand dann nicht so groß, weil man dann für Änderungen nicht ständig Test und Implementierung gleichzeitig anpassen muss. Mit TDD schreibt man zwar nachweislich besseren Code, aber es ist anstrengend.

Kurzum, so schön produktiv das Schreiben von Code heutzutage ist, so nervig und zeitaufwendig kann das Schreiben von Tests dazu sein, aber…

Es geht auch anders

Mit ein paar kleinen Tweaks und Kniffen lässt sich das Schreiben von Unit-Tests massiv vereinfachen.

```
@Entity
class Address {
      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      lateinit var uuid: UUID
      var street: String? = null
      var houseNumber: String? = null
      var zip: String? = null
      var city: String? = null
}
@Entity
class Person {
      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      lateinit var uuid: UUID
      var name:String? = null
      var birthday: LocalDate? = null
      var email: String? = null
      @ManyToOne(cascade = [CascadeType.ALL])
      var address: Address? = null
}

interface PersonRepository : JpaRepository<Person, UUID>
```

Listing 1: Entities und Repositories

Betrachten wir einen einfachen Unit-Test, der die Persistierung einer *Hibernate Entity* mit *Spring Data* in einer In-Memory-Datenbank testet. Ich gehe hier absichtlich nicht auf das komplette Setup ein. Es handelt sich um ein Standard Spring-Boot-Projekt mit *Spring Data JPA*. Das GitHub Repository ist unter *[1]* zu finden.

In *Listing 1* definieren wir zunächst die notwendigen *Hibernate Entities* und ein einfaches *Spring Data JPA Repository*.

An diesem Code ist absolut nichts Interessantes. Der Setup-Code für einen einfachen JPA-Test ist in wenigen Zeilen gemacht *(siehe Listing 2)* und sogar noch langweiliger. Der einfache Test ist nach Ausführung grün und stellt lediglich sicher, dass die Spring-Konfiguration korrekt ist.

```
@DataJpaTest
class PersonRepositoryTest {

      @Autowired
      private lateinit var personRepository: PersonRepository

      @Test
      fun `spring context config`() {
            assertNotNull(personRepository)
      }

}
```

Listing 2: Einfaches Test-Setup

Wie gehen wir nun vor, wenn wir die Persistierung unserer *Entity* tatsächlich testen möchten? Die erste Notwendigkeit besteht darin, die Entity-Klassen zu instanziieren und mit validen Testdaten zu befüllen *(siehe Listing 3)*.

Der Setup-Code ist viermal so lang wie die eigentliche Logik. Dadurch ist der Test unnötig schwer zu lesen. Dieser Code wiederholt sich im schlimmsten Fall in mehreren Tests. Geht es besser?

Für den Test selbst ist irrelevant, welche Daten in den Feldern stehen, solange sie valide sind. Warum also nicht den nächsten logischen Schritt gehen und die Daten einfach direkt mit Zufallsdaten befüllen?

Zunächst spürt man diesen kleinen Zweifel im Hinterkopf. *Das kann doch keine gute Idee sein!*

Bei genauer Betrachtung bringt das jedoch mehrere Vorteile:

1. Es ist unmöglich, unsichtbare Abhängigkeiten auf Basis von Testdaten zu erzeugen, da alle Testdaten für jeden Test zufällig neu generiert werden.
2. Das Befüllen unseres Testobjektes lässt sich automatisieren (Details folgen gleich).
3. Tests fokussieren sich auf das Wesentliche und sind so kurz wie möglich.
4. Eine deutliche Zeitersparnis beim Schreiben von Tests, da ein Großteil des Boilerplate-Codes einfach wegfällt.

Wie würde eine Hilfsfunktion idealerweise aussehen, die unsere *Entities* mit Zufallswerten füllt?

```
@Test
fun `persists person correctly`() {

      val person = Person().apply {
            this.name = "Max Muster"
            this.email = "max@muster.de"
            this.birthday = LocalDate.now().minusYears(41)
            this.address = Address().apply {
                  this.street = "Teststraße"
                  this.houseNumber = "41a"
                  this.zip = "80805"
                  this.city = "München"
            }
      }

      var personEntity = personRepository.save(person)
      assertTrue(personRepository.existsById(personEntity.uuid))
}
```

Listing 3: Initialisierung mit Boilerplate-Code

```
val person = randomData<Person>()
```

Einer der Gründe, warum ich so gerne mit Kotlin arbeite, ist die Einfachheit der Sprache. Hat man einmal verstanden, wie die Sprache funktioniert, braucht man wenig Code, um ans Ziel zu kommen.

In *Listing 4* ist ein erster Entwurf unserer Hilfsfunktion zu sehen:

```
inline fun <reified T> randomData(): T {
      val parameters = EasyRandomParameters()
            .collectionSizeRange(1, 4)
            .randomize(UUID::class.java) {
                  UUID.randomUUID()
            }
      val generator = EasyRandom(parameters)
      return generator.nextObject(T::class.java)
}
```

Listing 4: Zufallswerte mit RandomData

Hier geschieht tatsächlich jede Menge. Zunächst nutzen wir eine *Inline-Funktion*. In Kombination mit der `reified`-Deklaration des generischen Typ `T` umgehen wir so die *TypeErasure* zur Laufzeit. Der Kotlin-Compiler kopiert die Funktionsdeklaration direkt an die aufrufende Stelle und ersetzt hierbei den generischen Typ `T` durch den echten Typen. Dadurch ist kein Casting zur Laufzeit mehr notwendig.

Innerhalb der Funktion nutzen wir *EasyRandom [2]*, um eine Instanz vom Typ `T` zu erzeugen. Wie einzelne Felder befüllt werden, lässt sich über *EasyRandomParameters* steuern. Beispielsweise geben wir *EasyRandom* einen Hinweis, wie UUIDs zu befüllen sind und legen außerdem fest, dass Listen mit einer Größe von mindestens 2 und maximal 4 Elementen erstellt werden. Damit vermeiden wir viel zu große Listen mit 50 und mehr Elementen.

Damit lassen sich bereits nahezu beliebige Objekte mit Zufallswerten befüllen. Wie das aussehen kann, wird in *Abbildung 1* veranschaulicht.

Wir sehen aber auch, dass die Werte zwar formal korrekt, aber semantisch falsch befüllt werden. Woher sollte *EasyRandom* auch wissen, dass es sich beim Feld „*email*" um eine E-Mail-Adresse handelt.

Korrekte Werte mit BeanValidation

Dieses Problem löst *EasyRandom* recht elegant mit einer Integration des *BeanValidation*-API, das über eine zusätzliche Maven-Abhängigkeit direkt angezogen werden kann *(siehe Listing 5)*.

Abbildung 1: Befüllte Person-Instanz, aber mit invaliden Werten?

```
<dependency>
     <groupId>org.jeasy</groupId>
     <artifactId>easy-random-bean-validation</artifactId>
     <version>5.0.0</version>
     <scope>test</scope>
</dependency>
<dependency>
     <groupId>javax.validation</groupId>
     <artifactId>validation-api</artifactId>
     <version>2.0.1.Final</version>
</dependency>
```

Listing 5: BeanValidation Dependency

Hierbei ist zu beachten, dass sich *EasyRandom* selbst im *Maintenance Mode* befindet und nicht mehr aktiv weiterentwickelt wird. Um mit der aktuellen Version der *jakarta.validation-api* zu arbeiten, müsste das Projekt *geforkt* werden.

Mit diesen zusätzlichen Abhängigkeiten können wir unsere Entity-Klassen einfach annotieren *(siehe Listing 6)*. Wir erhalten damit automatisch semantisch korrekte Werte in unseren Testdaten.

```
@Entity
class Address {
     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     lateinit var uuid: UUID
     var street: String? = null
     var houseNumber: String? = null
     @Pattern(regexp = "[0-9]{5}")
     var zip: String? = null
     var city: String? = null
}

@Entity
class Person {
     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     lateinit var uuid: UUID
     var name:String? = null
     @Past
     var birthday: LocalDate? = null
     @Email
     var email: String? = null
     @ManyToOne(cascade = [CascadeType.ALL])
     var address: Address? = null
}
```

Listing 6: Semantisch korrekte Werte durch Annotieren der Entity-Klassen

Sind die Entity-Klassen nicht sowieso schon mit BeanValidation-Annotationen annotiert, fühlt es sich zunächst natürlich komisch an, die Klassen nur für die Tests zu annotieren. Ob dies akzeptabel ist, muss von Fall zu Fall entschieden werden. Es gibt im Zweifel auch andere Möglichkeiten, diese Funktionalität zu realisieren.

Die befüllte Instanz mit validen Werten ist in *Abbildung 2* dargestellt.

Customizing der Attribute

Mit diesem einfachen Setup haben wir jetzt schon die Möglichkeit, mit einer einzigen Zeile Code beliebige Testdaten zu generieren. Was uns aber fehlt, ist die Möglichkeit, Attribute gezielt für einzelne Test-Cases zu befüllen. Natürlich könnte man versuchen, mit *Reflections* zu arbeiten, Kotlin bietet uns aber über *Extension Functions* eine deutlich elegantere Methode mit minimalem Aufwand.

Indem wir die Signatur der Funktion um eine *Extension Function mit Receiver* erweitern, haben wir die Möglichkeit, Funktionen direkt auf dem dynamischen Typ `T` zu definieren und innerhalb der `randomData`-Funktion aufzurufen:

```
inline fun <reified T> randomData(block: T.()->Unit={}): T
```

Die Schreibweise ist zugegebenermaßen zunächst gewöhnungsbedürftig.

Wir definieren hier eine *Extension Function* auf dem Typ `T`, die als Lambda-Parameter in die `randomData`-Funktion übergeben wird. Die *Extension Function* erweitert zur Laufzeit den Typ T, aber nur innerhalb der `randomData`-Funktion.

In der übergebenen Lambda wiederum haben wir über `this` Zugriff auf die Instanz vom Typ `T` zur Laufzeit und können direkt darauf arbeiten.

```
val email = "test@test.de"
val person = randomData<Person> {
     this.email = email
}
```

Listing 7: Extension Function mit Receiver in Aktion

Abbildung 2: Befüllte Person-Instanz

```
inline fun <reified T> randomData(block: T.()->Unit = {}): T {
     val parameters = EasyRandomParameters()
           .collectionSizeRange(1, 4)
           .randomize(UUID::class.java) { UUID.randomUUID() }

     val generator = EasyRandom(parameters)
     //Aufruf der Extension Function über die Scope Function “also”
     return generator.nextObject(T::class.java).also(block)
}
```

Listing 8: Vollständige Deklaration der randomData-Funktion

```
@Test
fun `persists person correctly`() {
      val email = "test@test.de"
      val person = randomData<Person> {
            this.email = email
      }
      var personEntity = personRepository.save(person)
      assertEquals(personEntity.email, email)
}
```

Listing 9: Viel besser – ein lesbarer Test

Das Konzept wird sofort klar, wenn wir den Aufruf im Test betrachten *(siehe Listing 7)*. Zunächst wird die Person-Instanz mit Zufallswerten befüllt und anschließend haben wir die Möglichkeit, einzelne Felder über die übergebene Lambda zu befüllen.

Die Lambda wird standardmäßig als leere Lambda initialisiert, was es uns erlauben würde, die Lambda-Definition im Aufruf auch einfach wegzulassen. Damit ist `randomData<Person>()` auch weiterhin ein valider Aufruf unserer Funktion.

Da der Compiler durch die Inline-Deklaration für unser Beispiel bereits weiß, dass der Typ T nur eine Person sein kann, können wir typsicher auf die einzelnen Felder und Methoden der Person-Instanz zugreifen.

Natürlich müssen wir die übergebene Lambda noch auf die erzeugte Instanz anwenden, was recht elegant über die *Scope Function* `also` realisiert werden kann *(siehe Listing 8)*.

Eine weniger elegante, aber natürlich genauso korrekte, Möglichkeit wäre einfach `person.block()` vor dem Return-Statement aufzurufen.

Der Name `block` hat in diesem Kontext keine besondere Bedeutung und kann beliebig definiert werden. Erfahrungsgemäß erhöht es aber die Lesbarkeit des Codes ungemein, wenn diese Art Parameter immer gleich benannt wird.

Das Ergebnis unserer Arbeit sehen wir in *Listing 9*. Der Code enthält keinerlei Boilerplate-Code mehr und es ist sehr leicht, ihn auf einen Blick zu lesen.

Fazit

Dieser Artikel gibt ein konkretes Beispiel, wie die Lesbarkeit von Unit-Tests durch wenige Zeilen Kotlin-Code verbessert werden kann. Aber noch wichtiger als die Lesbarkeit ist die Tatsache, dass ein simples Werkzeug die Arbeit aller Entwickler im Team *einfacher* machen kann. Damit erhöht sich automatisch die *Velocity* des Teams und auch die Bereitschaft, gute Tests zu schreiben.

Natürlich lässt sich diese Funktionalität auch mit Java-Bordmitteln realisieren, wenn auch vielleicht nicht ganz so elegant.

Selbst wenn aufgrund technischer Vorgaben Kotlin im Projekt *noch* nicht im produktiven Code eingesetzt werden kann, spricht typischerweise nichts dagegen, Kotlin erstmal *nur* für die Tests einzusetzen.

Die Argumente in diesem Artikel können als Grundlage für eine Diskussion dienen.

Quellen

[1] *https://github.com/dilgerma/kotlin-random-tests*

[2] *https://github.com/j-easy/easy-random*

Martin Dilger

info@nebulit.de

Martin Dilger ist Geschäftsführer der NebulIT GmbH und seit 15 Jahren passionierter Softwareentwickler und Architekt. Mit einem starken Fokus auf Produktivität und Automatisierung arbeitet Martin Dilger hauptsächlich in den Bereichen Microservices, verteilte eventbasierte Architekturen mit Apache Kafka und bietet Schulungen zum Thema Kafka, Kotlin und Git.

Application Security mit Trivy

Bennet Schulz, codecentric AG

Neulich bin ich über ein Cheat Sheet der OWASP zu Docker-Security [1] gestolpert. Viele Punkte, wie beispielsweise das explizite Verwenden eines Nutzers oder auch das Updaten der Images, sind mittlerweile den meisten bekannt. Weniger bekannt beziehungsweise weniger stark verbreitet sind meiner Erfahrung nach noch die statischen Analyse-Tools zum Überprüfen von Sicherheitslücken in Docker-Images. Aus diesem Grund widme ich mich in diesem Artikel der ganzheitlichen Application Security mit dem Tool Trivy [2].

Trivy ist ein Security-Scanning-Tool, das folgende *targets* überprüfen kann:

- Container-Images
- Filesystems
- Git Repositories (remote)
- Virtual-Machine-Images
- Kubernetes
- AWS

Und dort nach Folgendem suchen kann:

- OS packages and software dependencies in use (SBOM)
- Bekannte Sicherheitslücken (CVEs)
- Falsche Infrastructure-as-Code(IaC)-Konfigurationen
- Sensible Informationen und Secrets
- Softwarelizenzen

Trivy unterstützt eine Palette von *targets*, die es auf Sicherheitslücken überprüfen kann. Dazu gibt es einige Integrationen beispielsweise für *Kubernetes Operators*, ein *VisualStudio Code Plug-in* oder auch eine *GitHub Action*. In diesem Artikel beschränken wir uns auf die Verwendung von Trivy in *GitHub Actions* und auf das Scannen von *Git Repositories*, *Pull Requests* sowie in *Continuous Integration Pipelines* und allem, was dazu gehört. Auf die Überprüfung von sensiblen Informationen und Secrets sowie auf Softwarelizenzen verzichten wir an dieser Stelle, da GitHub dies mittlerweile out of the box kann.

Im Folgenden werden wir uns die bekannte *Spring PetClinic* nehmen und die Trivy *GitHub Action* dort einbauen. Dazu fügen wir den in *Listing 1* gezeigten *GitHub Action workflow job* hinzu.

```yaml
name: build
on:
  push:
     branches:
        - main
  pull_request:
jobs:
  build:
     name: Build
     runs-on: ubuntu-20.04
     steps:
        - name: Checkout code
          uses: actions/checkout@v3

        - name: Run Trivy vulnerability scanner in repo mode
          uses: aquasecurity/trivy-action@master
          with:
             scan-type: 'fs'
             ignore-unfixed: true
             format: 'sarif'
             output: 'trivy-results.sarif'
             severity: 'CRITICAL'

        - name: Upload Trivy scan results to GitHub Security tab
          uses: github/codeql-action/upload-sarif@v2
          with:
             sarif_file: 'trivy-results.sarif'
```

Listing 1: GitHub workflow zum Prüfen der Maven- und Gradle-build-Dateien

Dieser Job namens `build` wird bei einem neuen *Pull Request* beziehungsweise bei einem `push` auf den `main branch` getriggert. Anschließend checkt die *Action* den Code zuerst aus, um ihn anschließend auf Sicherheitslücken zu überprüfen und abschließend in GitHub unter dem Reiter *Security/Code Scanning* hochzuladen.

Um zu testen, wie die *GitHub Action* sich bei einer Sicherheitslücke verhält, habe ich nun einen *Pull Request* erstellt und darin eine betroffene *log4j dependency* eingebaut. Das Resultat ist in *Abbildung 1* zu sehen.

Durch den zuvor erstellten *Pull Request* werden neun neue Sicherheitslücken hinzugefügt. Darunter auch der bekannte Bug, der selbst beim Bundesamt für Sicherheit in der Informationstechnik (kurz: BSI) einen Eintrag bekommen hat und der zu zahlreichen Beiträgen in diversen Medien führte.

So weit, so gut. Genau um solche Sicherheitslücken gar nicht erst einzubauen, haben wir diese *GitHub Action* hinzugefügt. Das Einzige, was mich persönlich an dieser Stelle noch etwas stört, ist, dass die *GitHub Action* in der Pull-Request-Ansicht keinerlei Hinweis darauf gibt, ob durch diesen *Pull Request* Sicherheitslücken hinzukommen *(siehe Abbildung 2)*. Erwartet hätte ich, dass in genau dieser Ansicht ein Hinweis auftaucht, der mir signalisiert, dass ich gerade dabei bin, Sicherheitslücken einzubauen, sodass ich diese als Ersteller des *Pull Request* gar nicht erst übersehen kann.

Dass der *Pull Request Build* komplett fehlschlägt, wäre möglicherweise etwas zu viel des Guten, da es durchaus einmal sein könnte, dass man durch Versionsupdates oder neue Features einmal eine kleine, potenzielle Sicherheitslücke einbaut. Häufig habe ich das bei der Verwendung großer *Libraries* gesehen, von denen man dann allerdings nur einen Bruchteil nutzt, den betroffenen Teil, der die Sicherheitslücke enthält, allerdings nicht. An dieser Stelle wäre ein konfigurierbarer *CVE Count* ganz hilfreich – so wie man es von dem *OWASP Dependency Check* kennt. Dadurch könnte man den *GitHub Action Build* so konfigurieren, dass er bei einem *CVE Count* von beispielsweise 10 fehlschlägt.

Überprüfen von Docker Images

Neben dem Überprüfen von *Maven* und *Gradle Dependencies* kann Trivy auch Docker-Container auf Sicherheitslücken überprüfen *[3]*. Dazu bedarf es lediglich einer Änderung der Parameter für die *GitHub Action (siehe Listing 2)*.

Anstatt wie zuvor `scan-type` und andere Parameter anzugeben, geben wir nun eine Referenz auf das Docker-Container-Image an.

Durch diesen *GitHub Action Workflow* wird von nun an das resultierende Docker-Image bei jedem *Pull Request* gebaut und auf Sicherheitslücken überprüft. Die Ergebnisse werden ebenfalls unter *Security/Code Scanning* angezeigt *(siehe Abbildung 3)*.

Dieses Mal ist die Liste um ein Vielfaches länger und die „Flughöhe", auf die wir uns hier begeben, ist eine deutlich geringere *(siehe Abbildung 4)*. Anders als zuvor benötigen wir bei dieser Überprüfung deutlich länger. Das liegt einerseits daran, dass wir das durch diesen *Pull Request* entstehende Docker-Image erst noch einmal bauen müssen, um zu sehen, ob wir durch diesen *Pull Request* zusätzliche

Sicherheitslücken einbauen würden. Dazu dauert die eigentliche Überprüfung auch deutlich länger, da die Menge an zu überprüfenden Abhängigkeiten deutlich größer ist.

Generierung von SBOMs (Software Bills of Materials)

Möchte man nun nicht nur für einen neu erstellen *Pull Request* wissen, ob dort Sicherheitslücken enthalten sind, sondern auch für bereits ausgelieferte Software-Versionen, dann schaffen SBOMs Abhilfe *[4]*.

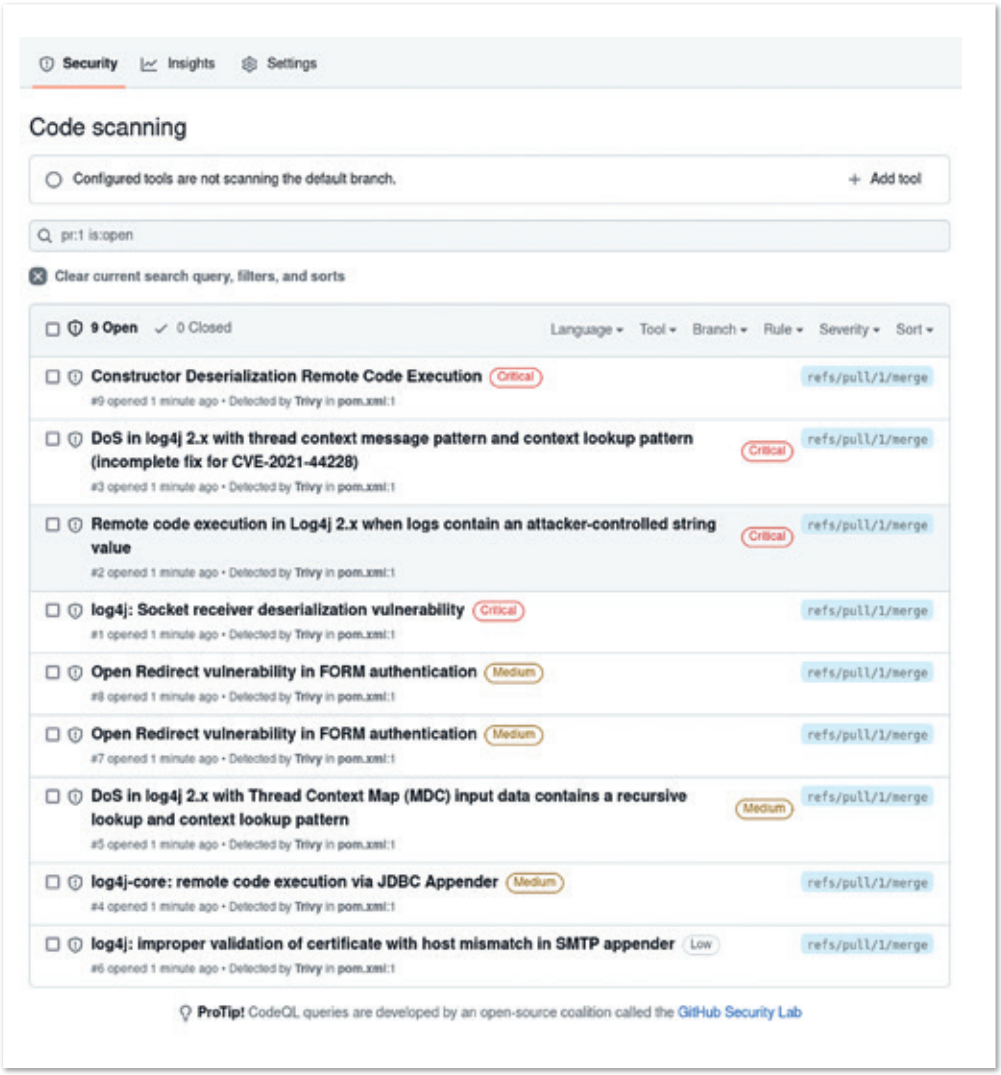


Abbildung 1: Pull Request mit eingebauter log4j dependency

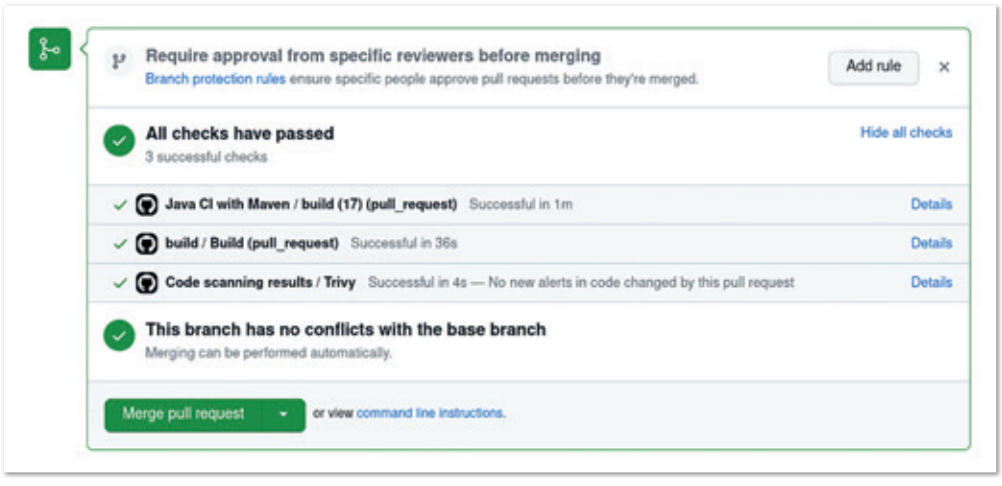


Abbildung 2: Fehlender Hinweis, dass Sicherheitslücken hinzugekommen sind

Ohnehin sollte über die Verwendung von SBOMs nachgedacht werden, denn seit der *Executive Order* der US-Regierung im Mai 2022 sind *Software Bills of Materials* in den USA Pflicht für jeden, der Software für die Regierung entwickelt. Der Grund dafür sind verschiedene Vorfälle, beispielsweise der von SolarWind, bei dem Hacker eine *Backdoor* in die Netzwerk-Monitoring-Plattform von SolarWinds eingeschleust haben. Oder auch den für Java-Entwickler bekannteren Vorfall mit *Apache Log4j*, der unter dem Namen „Log4-Shell" prominent wurde und bei dem Angreifer in der Lage waren, fast beliebigen Code auf den betroffenen Systemen auszuführen. Damit soll in Zukunft Schluss sein.

Abhilfe dafür sollen SBOMs liefern. Sie sind vergleichbar mit Stücklisten in der Automobilindustrie. Dies ist eine Liste, die zum Beispiel detailliert aufzeigt, welche Airbags welches Herstellers an welcher Stelle im Auto verbaut sind. Dadurch ist es Automobilherstellern möglich, nachzuvollziehen, welche Autos beispielsweise von einer fehlerhaften Charge an Airbags betroffen sind und zurückgerufen werden müssen, um sie zu reparieren. Ohne diese Stückliste gäbe es keinerlei Möglichkeit, für jedes ausgelieferte Auto zu kontrollieren, ob es von der fehlerhaften Charge an Airbags betroffen ist oder nicht.

Im Falle von Sicherheitslücken in Softwareversionen verhält es sich genauso. Im konkreten Fall von *Log4j* war es so, dass vielen zunächst gar nicht klar war, ob sie die betroffene Log4j-Version überhaupt verwenden. Ebenso war es für seit Jahren laufende Software gar nicht so einfach möglich, zu beantworten, ob die Version von der Sicherheitslücke betroffen ist oder nicht. Das Resultat dieses Vorfalls war ein hektisches, manuelles Suchen und Analysieren von Abhängigkeiten etlicher sich in Verwendung befindlicher Softwareversionen.

SBOMs hätten die Problematik gelöst. Dadurch wären Softwareent-

```yaml
name: container-image-scan
on:
  push:
     branches:
        - main
  pull_request:
jobs:
  build:
     name: Build
     runs-on: ubuntu-20.04
     steps:
        - name: Checkout code
          uses: actions/checkout@v3

        - name: Build an image from Dockerfile
          run: |
             docker build -t docker.io/my-organization/my-app:${{ github.sha }} .

        - name: Run Trivy vulnerability scanner
          uses: aquasecurity/trivy-action@master
          with:
             image-ref: 'docker.io/my-organization/my-app:${{ github.sha }}'
             format: 'sarif'
             output: 'trivy-results.sarif'

        - name: Upload Trivy scan results to GitHub Security tab
          uses: github/codeql-action/upload-sarif@v2
          with:
             sarif_file: 'trivy-results.sarif'
```

Listing 2: GitHub Workflow zum Prüfen von Docker-Images

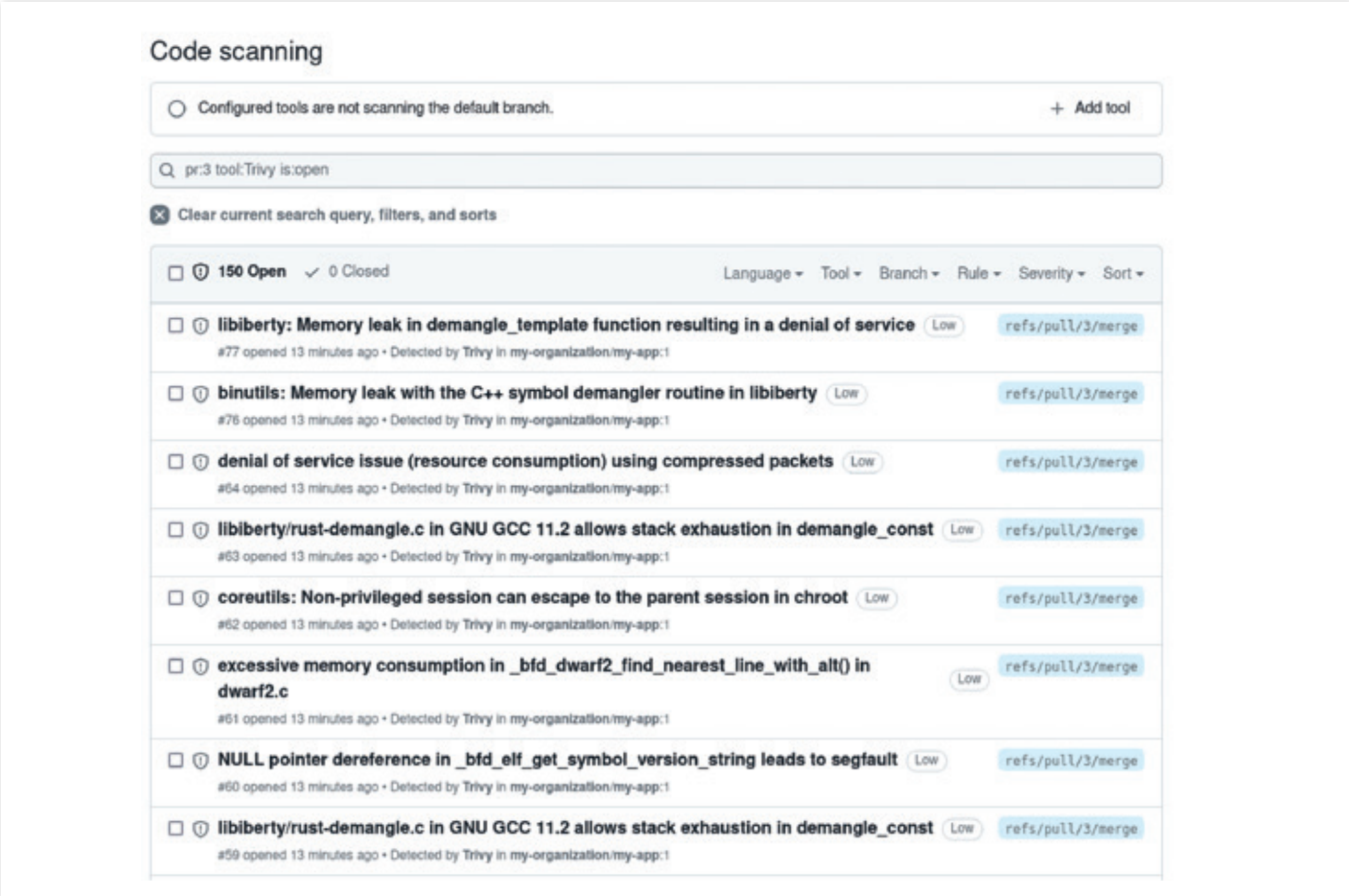


Abbildung 3: Ergebnis eines Pull Request beim Bau eines Docker-Image

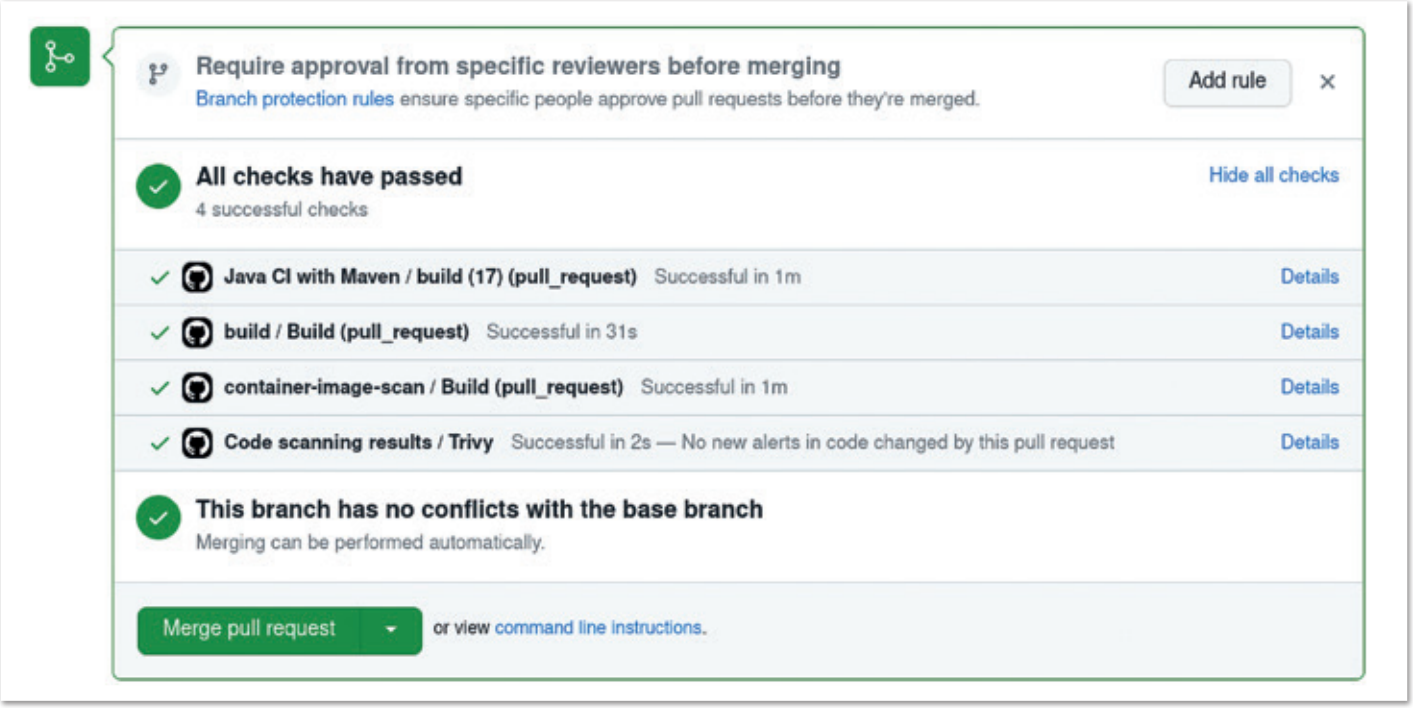


Abbildung 4: Check-Übersicht

wickler zu jedem Zeitpunkt in der Lage gewesen, zu sagen, ob sie von der Log4j-Sicherheitslücke betroffen sind oder nicht. Dabei listet eine SBOM nicht nur auf, was in die Software alles „eingebaut" wurde, sondern zusätzlich auch, von wem eine *Dependency* verwendet wurde. Als Extra listet eine SBOM auch die jeweilig verwendeten Softwarelizenzen auf, was aus rechtlicher Sicht ebenfalls relevant sein kann.

Um nun SBOMs in seine Anwendung zu integrieren, kann ebenfalls auf Trivy zurückgegriffen werden. Wie auch zuvor bedarf es dafür eines neuen *GitHub Workflows* *(siehe Listing 3)*.

Zusätzlich dazu muss noch der *Dependency Graph* in GitHub *enabled* werden. Das geschieht über Settings -> Code Security and Analysis -> Dependency Graph -> enable.

Erst dann ist die *Trivy GitHub Action* in der Lage, die SBOM-Datei hochzuladen.

Zusätzlich dazu ist es notwendig, Lese- und Schreibrechte für den *Workflow* zu vergeben. Das geschieht über Settings -> Actions -> General -> Workflow Permissions *(siehe Abbildung 5)*.

```yaml
name: Pull Request
on:
  push:
     branches:
        - main

permissions:
  contents: write

jobs:
  build:
     name: Checks
     runs-on: ubuntu-20.04
     steps:
        - name: Checkout code
          uses: actions/checkout@v3

        - name: Run Trivy in GitHub SBOM mode and submit results to Dependency Graph
          uses: aquasecurity/trivy-action@master
          with:
             scan-type: 'fs'
             format: 'github'
             output: 'dependency-results.sbom.json'
             image-ref: '.'
             github-pat: ${{ secrets.GITHUB_TOKEN }}
```

Listing 3: Erstellen einer SBOM

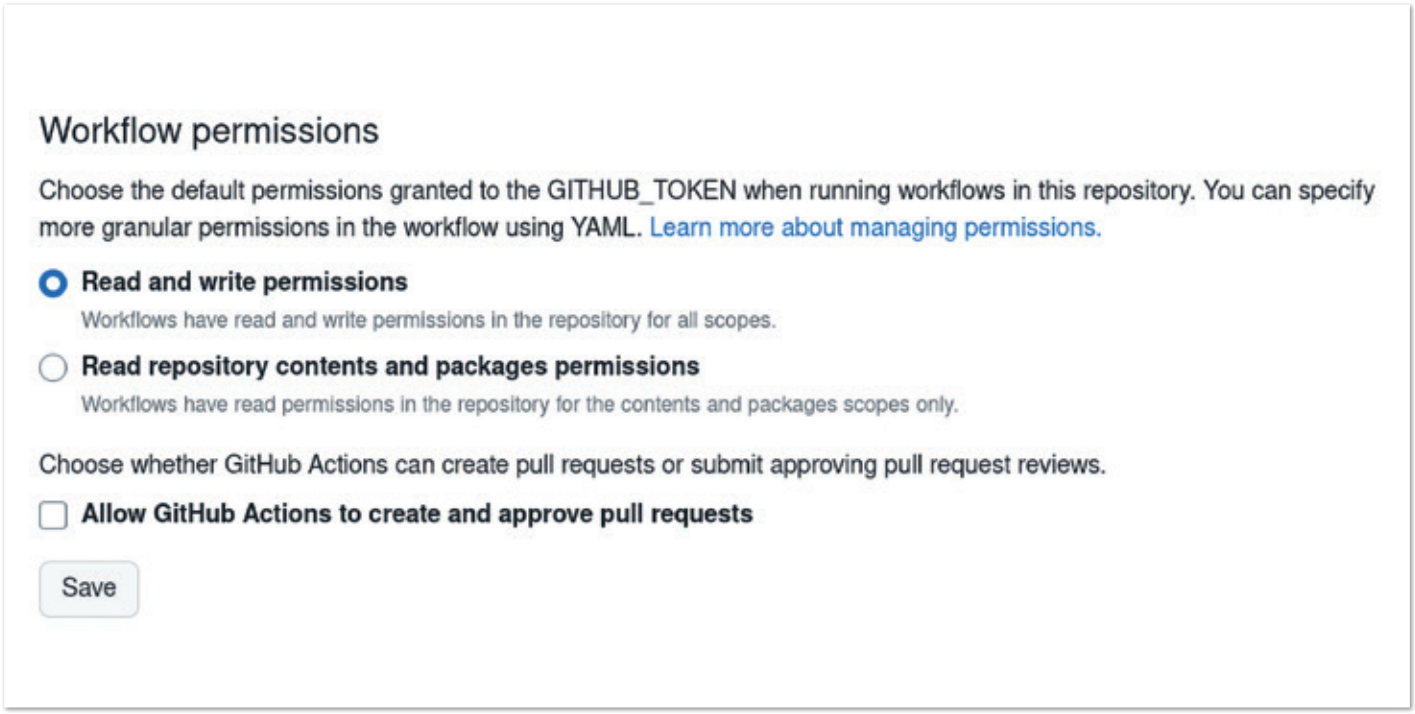


Abbildung 5: Vergabe der Lese- und Schreibrechte für den Workflow

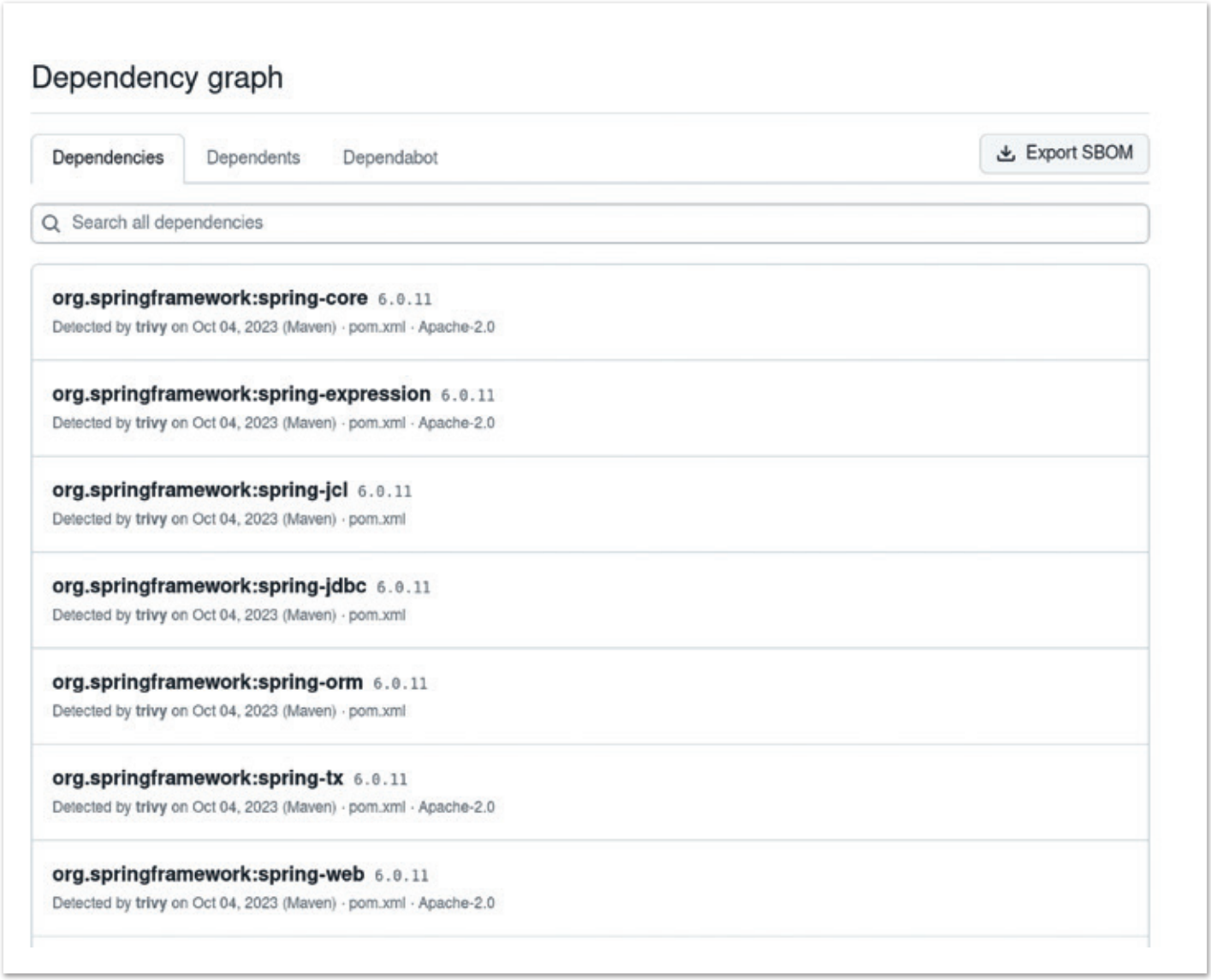


Abbildung 6: Software Bill of Materials im Dependency Graph

Nun ist der GitHub Workflow in der Lage, eine Software Bill of Materials zu erstellen. Zu finden ist sie unter Insights -> Dependency Graph und sieht wie in *Abbildung 6* gezeigt aus.

Fazit

In diesem Artikel haben wir eine Art Schweizer Taschenmesser für Application Security kennengelernt. Neben dem Überprüfen von Abhängigkeiten in den build-Dateien von Maven und Gradle, ist es mit Trivy ebenfalls möglich, Docker-Container-Images auf Sicherheitslücken zu überprüfen und auch SBOMs zu erzeugen.

Alle diese Überprüfungen sind mit relativ wenig Aufwand eingebaut und geben uns in kurzer Zeit weitere wichtige Einblicke in die Sicherheit unserer Software. Was das Thema SBOMs angeht, bin ich gespannt, was die Zukunft bringt. Derzeit fehlt es noch an einer out-of-the-box-Integration zum Archivieren der SBOMs zu einem bestimmten Software-Release, ebenso wie einem handlichen Tool zum Durchsuchen alter Software-Releases nach bekannten Sicherheitslücken. Da GitHub die Integration von SBOMs erst seit Ende März dieses Jahres hinzugefügt hat, bin ich guter Dinge, dass hier noch das eine oder andere Feature hinzukommen wird.

Quellen

[1] *https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html*

[2] *https://github.com/knqyf263/trivy*

[3] *https://anchore.com/blog/docker-security-best-practices-a-complete-guide/*

[4] *https://github.blog/2023-03-28-introducing-self-service-sboms/*

Bennet Schulz

bennet.schulz@codecentric.de

Bennet ist Software Engineer und seit etwa einem Jahr mit zwischenzeitlichen Ausflügen über die Produktentwicklung bei einem deutschen und einem US-amerikanischen Softwarehersteller wieder zurück bei codecentric. Seinen Schwerpunkt hat er im Java-Ökosystem. In seiner Freizeit arbeitet er an der Feature-Toggles-Library namens togglz und ist Teil des Programmkomitees der JavaLand-Konferenz sowie des Java Forum Nord.

Die dunkle Seite der künstlichen Intelligenz

Marco Schulz

Als Techniker bin ich recht schnell von allen möglichen Dingen zu begeistern, die irgendwie blinken und piepsen, ganz gleich, wie unnütz diese auch sein mögen. Elektronikspielereien ziehen mich an, wie das Licht Motten. Seit einer Weile ist eine neue Generation Spielwaren für die breite Masse verfügbar: Anwendungen der Künstlichen Intelligenz, genauer gesagt künstliche neuronale Netze. Die frei verfügbaren Anwendungen leisten bereits Beachtliches und es ist erst der Anfang dessen, was noch möglich sein wird. Vielen Menschen ist die Tragweite KI-basierter Anwendungen noch gar nicht bewusst geworden. Das ist auch nicht verwunderlich, denn das, was gerade im Sektor KI geschieht, wird unser Leben nachhaltig verändern. Wir können also zu Recht sagen, dass wir in einer Zeit leben, die gerade Geschichte schreibt. Ob die kommenden Veränderungen etwas Gutes werden, oder sie sich als eine Dystopie entpuppen, wird an uns liegen.


1011 3100 0100 0410 0413 0130
167 012 356 542 0101
1011 0100 3100
164 889 386 542 0101
31010 10010 13010 010 01013 10010 10010 040

Als ich im Studium vor sehr vielen Jahren als Vertiefungsrichtung Künstliche Intelligenz gewählt hatte, war die Zeit noch von sogenannten „Expertensystemen" geprägt. Diese regelbasierten Systeme waren für ihre Domäne hochspezialisiert und wurden für entsprechende Experten ausgelegt. Das System sollte die Experten bei der Entscheidungsfindung unterstützen. Mittlerweile haben wir auch die notwendige Hardware, um viel allgemeinere Systeme zu schaffen. Wenn wir Anwendungen wie ChatGPT betrachten, basieren diese auf neuronalen Netzen, was eine sehr hohe Flexibilität in der Verwendung erlaubt. Der Nachteil ist allerdings, dass wir als Entwickler kaum noch nachvollziehen können, welche Ausgabe ein neuronales Netz für eine beliebige Eingabe erzeugt. Ein Umstand, der die meisten Programmierer, die ich kenne, eher eine ablehnende Haltung einnehmen lässt, da diese so nicht mehr Herr über den Algorithmus sind, sondern nur noch nach dem Prinzip Versuch und Irrtum agieren können.

Dennoch ist die Leistungsfähigkeit neuronaler Netze verblüffend. Vorbei scheint nun die Zeit, in der man sich über unbeholfene, automatisierte, Software-gestützte Übersetzungen lustig machen kann. Aus eigener Erfahrung weiß ich noch, wie mühselig es war, den Google Translator aus dem Deutschen einen vernünftigen Satz ins Spanische übersetzen zu lassen. Damit das Ergebnis brauchbar war, konnte man sich über die Option Englisch – Spanisch behelfen. Alternativ, wenn man nur ein rudimentäres Englisch für den Urlaubsgebrauch spricht, konnte man noch sehr einfache deutsche Sätze formulieren, die dann wenigsten inhaltlich korrekt waren. Die Zeitersparnis für automatisiert übersetzte Texte ist erheblich, obwohl man diese Korrektur lesen muss und gegebenenfalls ein paar Formulierungen angepasst werden müssen.

So sehr ich es schätze, mit solchen starken Werkzeugen arbeiten zu können, müssen wir uns aber auch im Klaren sein, dass es auch eine Schattenseite gibt. Denn je mehr wir unserer täglichen Aufgaben über KI-gestützte Tools erledigen, umso mehr verlieren wir die Fähigkeit, diese Aufgaben künftig weiterhin manuell bearbeiten zu können. Für Programmierer bedeutet dies, dass sie im Laufe der Zeit über KI-gestützte IDEs ihre Ausdrucksfähigkeit im Quellcode verlieren. Das ist natürlich kein Prozess, der über Nacht stattfindet, sondern sich schleichend einstellt. Aber sobald diese Abhängigkeit einmal geschaffen ist, stellt sich die Frage, ob die verfügbaren, liebgewonnenen Werkzeuge weiterhin kostenfrei bleiben oder, ob für bestehende Abonnements möglicherweise drastische Preiserhöhungen stattfinden. Denn es sollte uns schon klar sein, das kommerziell genutzte Werkzeuge, die unsere Produktivität erheblich verbessern, üblicherweise nicht zum Schnäppchenpreis verfügbar sind.

Ich denke auch, dass das Internet, wie wir es bisher gewohnt sind, sich in Zukunft sehr stark verändern wird. Viele der kostenlosen Angebote, die bisher durch Werbung finanziert sind, werden mittelfristig verschwinden. Schauen wir uns dazu einmal als Beispiel den Dienst „Stack Overflow" an – in Entwicklerkreisen eine sehr beliebte Wissensplattform. Wenn wir nun künftig für die Recherche zu Fragestellungen der Programmierung ChatGPT oder andere neuronale Netze nutzen, sinken für Stack Overflow die Besucherzahlen kontinuierlich. Die Wissensbasis wiederum, die ChatGPT nutzt, basiert auf Daten von öffentlichen Foren wie Stack Overflow. Somit wird auf absehbare Zeit Stack Overflow versuchen, seine Dienste für KIs unzugänglich zu machen. Es könnte sicher auch eine Einigung mit Ausgleichszahlungen zu Stande kommen, sodass die wegfallenden Werbeeinnahmen kompensiert werden. Denn als Techniker muss uns nicht ausschweifend dargelegt werden, dass für ein Angebot wie Stack Overflow erhebliche Kosten für den Betrieb und die Entwicklung anfallen. Es bleibt dann abzuwarten, wie die Nutzer das Angebot künftig annehmen werden. Denn wenn auf Stack Overflow keine neuen Daten zu Problemstellungen hinzukommen, wird die Wissensbasis für KI-Systeme auch uninteressant. Daher vermute ich, dass bis zirka 2030 vor allem hochwertige Inhalte im Internet kostenpflichtig werden.

Wenn wir die Prognose des mittelfristigen Trends über den Bedarf von Programmierern betrachten, kommen wir zu der Frage, ob es künftig eine gute Empfehlung sein wird, Informatik zu studieren oder eine Ausbildung als Programmierer anzutreten. Ich sehe hier tatsächlich eine positive Zukunft und würde jedem, der eine Ausbildung als Berufung versteht und nicht als Notwendigkeit ansieht, seinen Lebensunterhalt zu bestreiten, in seinem Vorhaben bekräftigen. Meiner Ansicht nach werden wir weiterhin viele innovative Köpfe benötigen. Lediglich jene, die sich anstatt sich mit Grundlagen und Konzepten zu beschäftigen, lieber mal schnell ein aktuelles Framework erlernen wollen, um aufkommende Hypes des Marktes mitzunehmen, werden sicher nur noch geringen Erfolg in Zukunft erzielen. Diese Beobachtungen habe ich aber auch bereits vor der breiten Verfügbarkeit von KI-Systemen machen können. Deshalb bin ich der festen Überzeugung, dass sich langfristig Qualität immer durchsetzen wird.

Dass man sich stets Themen möglichst kritisch und aufmerksam nähern sollte, betrachte ich als eine Tugend. Dennoch muss ich sagen, dass so manche Ängste im Umgang mit KI recht unbegründet sind. Sie haben ja schon einige meiner möglichen Zukunftsvisionen in diesem Artikel kennengelernt. Aussagen wiederum, dass die KI einmal unsere Welt übernehmen wird, indem sie unbedarfte Nutzer subtil beeinflusst, um diese zu Handlungen zu motivieren, halte ich für einen Zeitraum bis 2030 eher für reine Fantasie und mit dem aktuellen Erkenntnisstand unbegründet. Viel realistischer sehe ich das Problem, dass findige Marketingleute das Internet mit minderwertigen, ungeprüften, nicht redigierten, KI-generierten Artikeln übersähen, um ihr SEO-Ranking aufzupeppen und diese wiederum als neue Wissensbasis der neuronalen Netze die Qualität künftiger KI-generierter Texte erheblich reduziert.

Die bisher frei zugänglichen KI-Systeme haben gegenüber dem Menschen einen entscheidenden Unterschied. Ihnen fehlt die Motivation, etwas aus eigenem Antrieb zu tun. Erst durch eine extrinsische Anfrage durch den Nutzer beginnt die KI, eine Fragestellung zu bearbeiten. Interessant wird es dann, wenn eine KI sich aus eigenem Antrieb heraus selbstgewählten Fragestellungen widmet und diese auch eigenständig recherchiert. In diesem Fall ist die Wahrscheinlichkeit sehr hoch, dass die KI sehr schnell ein Bewusstsein entwickeln wird. Läuft eine solche KI dann noch auf einem Hochleistungsquantencomputer, haben wir nicht genügend Reaktionszeit, um gefährliche Entwicklungen zu erkennen und einzugreifen. Daher sollten wir uns durchaus das von Dürrenmatt geschaffene Drama „Die Physiker " in unserem Bewusstsein halten. Denn die Geister, die ich einmal rief, werde ich möglicherweise nicht so schnell wieder los.

Grundsätzlich muss ich zugeben, dass mich das Thema KI weiterhin fasziniert und ich auf die künftige Entwicklung sehr gespannt bin. Dennoch finde ich es wichtig, auch vor der dunklen Seite der Künstlichen Intelligenz den Blick nicht zu versperren und dazu einen sachlichen Diskurs zu beginnen, um möglichst schadenfrei das vorhandene Potenzial dieser Technologie auszuschöpfen.

Referenzen

- Podcast Künstliche Intelligenz (KI) und unser Weg zum HomoDeus: *https://www.inspirierendanders.com/alle-folgen-und-g%C3%A4ste-1/episode/314a474c/164-iaf-kunstliche-intelligenz-ki-und-unser-weg-zum-homodeus-wann-sind-wir-gott*
- Podcast Künstliche Intelligenz: *https://elmar-dott.com/podcast/kuenstliche-intelligenz/*
- Artikel Schreckgespenst künstliche Intelligenz: *https://elmar-dott.com/articles/schreckgespenst-kunstliche-intelligenz/*

Marco Schulz

Marco Schulz, im Internet auch unter seinem Pseudonym Elmar Dott bekannt, realisiert seit über knapp 20 Jahren als freier Berater in internationalen Projekten große Web-Applikationen. Seine Schwerpunkte sind DevOps, Konfigurationsmanagement, Software-Architekturen & Release Management. Als Trainer teilt er sein Wissen in Schulungen und spricht auch regelmäßig auf Konferenzen über aktuelle Themen. Seine Homepage finden Sie unter: https://elmar-dott.com

Work Happier: der menschliche Weg, um Performance und Erfolg zu steigern!

Sabine Wojcieszak, getNext IT

In unserem tiefsten Inneren wissen wir es alle: Arbeit geht uns leichter, schneller und besser von der Hand, wenn wir Spaß haben und grundlegend glücklich sind. Doch immer noch werden Faktoren wie Spaß und Happiness nicht als echte Business-Vorteile gesehen; im Gegenteil, wird es hektisch und stressig, dann werden die Aspekte, die Spaß machen und uns glücklich stimmen, gerne „geopfert". Warum? Um produktiver zu sein, den nächsten Meilenstein zu erreichen und so erfolgreich zu sein. Warum dieser „Work Harder"-Ansatz aber kontraproduktiv für uns Menschen und damit für das Business ist und welche Alternativen wir nutzen können, zeigt dieser Artikel auf.

In Zeiten von Fachkräftemangel und einer gleichzeitig größeren Komplexität ist die Verbesserung der Produktivität ein zentrales Thema, um erfolgreich(er) zu sein. Unternehmen, Führungskräfte, Teams und die einzelnen Mitarbeitenden sind von dieser Herausforderung gleichermaßen betroffen. So erleben Zeitmanagement-Methoden eine Renaissance. Agile Methoden werden zum Mainstream. KI-Lösungen sind der erhoffte heilige Gral für eine verbesserte Produktivität und mehr Erfolg. Und wenn auch diese Bemühungen nicht zum erwünschten Ziel führen, dann rufen wir die „Work Harder"-Parole aus – schließlich sind wir ja alle Profis!

Das Ergebnis von „Work Harder" ist schnell, dass wir unsere Zeit micromanagen, keine Zeitpuffer mehr einplanen und noch mehr Aufgaben in denselben Zeitrahmen reinpressen. Auch die agilen Methoden werden, entgegen der grundlegenden Idee, dann dem Micromanagement unterworfen. Zeit für Spaß bei der Arbeit? „Nein, schließlich müssen wir was schaffen und erfolgreich sein!" Glücklich macht das allerdings kaum! Doch die Wissenschaft sagt uns schon lange, dass die Lösung des Problems nicht „Work Harder" ist; vielmehr hilft uns der „Work Happier"-Ansatz, um diese Herausforderungen „artgerecht" anzugehen. Schließlich sind wir alle „nur" Menschen!

Happiness und Spaß bei der Arbeit als Erfolgsfaktoren

In seinem Buch „The Happiness Advantage" beschreibt Shawn Achor *[1]* diverse Forschungsergebnisse sowohl aus der Positiven Psychologie als auch aus der Neurowissenschaft, die immer wieder zeigen, dass wir Menschen am besten performen, wenn wir glücklich sind und Spaß haben. Eine positive Stimmung, das Glücklichsein – Happiness – und Spaß bei der Arbeit versetzen uns und unser Gehirn in die richtige Stimmung *[2]*, um

- kreativ zu sein und neue Ideen zu entwickeln,
- offen zu sein, neue Ideen und Lösungen anzunehmen,
- zu kollaborieren und offen zu kommunizieren,
- Alternativen und Chancen leichter zu erkennen,
- zu lernen und zu experimentieren und
- mit Fehlschlägen und Kritik besser umgehen zu können.

Ein kurzer Blick auf die obenstehende Liste zeigt sofort: Das sind genau die Skills und das Mindset, das wir für unsere modernen Herausforderungen benötigen!

Die Kausalkette, die sich in vielen Köpfen manifestiert hat, sieht folgendermaßen aus:

Wir müssen härter arbeiten,
um Erfolg zu haben,
und dann sind wir glücklich.

Doch dass Erfolg nicht automatisch glücklich macht, ist inzwischen kein Geheimnis mehr. Besonders in unserer sich schnell verändernden Zeit ist der alte Denkansatz problematisch. Mit den ganzen Veränderungen kommen neue Ziele, die es zu erreichen gilt. Ist ein Ziel erreicht, nehmen wir uns kaum die Zeit, dieses zu feiern und zu genießen. Teilweise nehmen wir nicht einmal wahr, dass wir gerade erfolgreich gewesen sind. Die „Schneller-weiter-höher Mentalität" führt dazu, dass wir unseren eigenen Erfolg immer wieder relativieren. Wie soll sich also da das Glücklichsein überhaupt automatisch einstellen können?

Die positive Psychologie dreht diesen Ansatz komplett um:

Wir müssen glücklich/in einer positiven Stimmung sein, um „härter" = engagierter/ausdauernder/sinnvoller zu arbeiten, und dann haben wir Erfolg.

Hier wird also das Glücklichsein an den Anfang der Kette gestellt, wogegen im alten Ansatz das Glücklichsein sich automatisch einstellen soll, wenn wir Erfolg haben. Gleichzeitig sagt der neue Ansatz implizit auch, dass das Glücklichsein nicht unbedingt von allein einstellt, sondern wir daran arbeiten können und müssen, da es eine wichtige Voraussetzung für den Erfolg ist. Dieser neue Ansatz nimmt uns in die Verantwortung, uns um das Glücklichsein zu kümmern.

Was genau sind Happiness und Spaß bei der Arbeit?

Auch wenn Happiness und Spaß bei der Arbeit nicht ein und dasselbe sind, so gehören sie doch eng zusammen und bedingen sich gegenseitig.

Happiness – das Glücklichsein – ist ein andauernder Zustand positiver Empfindungen. Im Allgemeinen verbinden wir damit Emotionen wie:

Glück, Freude, Zufriedenheit, Fröhlichkeit, Glückseligkeit

Martin Seligmann, ein US-amerikanischer Psychologe auf dem Gebiet der Positiven Psychologie schreibt dem Begriff Happiness drei Attribute zu:

Freude/Vergnügen, Engagement und Bedeutung

Barbara Frederickson, die derzeit führende Forscherin zum Thema Positive Emotionen hat eine sehr stark erweiterte Liste an positiven Emotionen, die sie mit dem Begriff Happiness verbindet:

Heiterkeit, Freude, Dankbarkeit, Gelassenheit, Interesse, Hoffnung, Stolz, Liebe, Inspiration, Ehrfurcht

Klar wird, dass es keine ganz einheitliche Definition von Happiness gibt; dennoch sehen wir aber viele Aspekte in den verschiedenen Ansätzen, die sich auch sehr gut auch den Bereich der Arbeit übertragen lassen. Eine Arbeit,

- die interessant ist,
- die ein inspirierendes Umfeld bietet,
- die uns stolz macht,
- in der wir eine Bedeutung sehen und wir als Person eine Bedeutung für andere darstellen,
- in der wir uns engagieren können/wollen und
- wo wir Freude und Vergnügen empfinden können, kann dazu beitragen, dass wir glücklich sind.

Das Glücklichsein ist eng mit einer positiven Grundeinstellung verbunden. Allerdings heißen Happiness und eine positive Einstellung nicht, dass negative Aspekte nicht wahrgenommen, schöngeredet oder gar ignoriert werden. Auch wer grundlegend glücklich ist, erlebt die volle Bandbreite an Emotionen; jedoch ziehen die negativen Emotionen unsere Stimmung und auch Leistungsfähigkeit nicht dauerhaft runter. Glückliche Menschen sind resilienter.

Im Gegensatz zu Happiness als grundlegender, andauernder Zustand, ist der Begriff Spaß als *kurzfristiger Moment der Freude* defi-

Abbildung 1: Die Komponenten für den „echten Spaß" nach Cathrine Price (© Sabine Wojcieszak)

niert. Auch wenn es hier um kurzfristige Momente geht, haben auch sie eine längerfristige Wirkung: sie machen gute Laune, sie regen die Produktion unserer positiven Botenstoffe an und wir erinnern uns gerne an diese Momente. Ergebnis ist auch hier, das Arbeiten sehr viel leichter von der Hand gehen, unsere Performance steigt und die Bindung zu unseren Teammitgliedern stärker wird. Empfinden wir Spaß, gehen wir gerne die nötigen Schritte (Engagement) – auch wenn sie über das Übliche hinausgehen.

Die bekannte Wissenschaftsjournalistin Cathrine Price beschreibt die drei wichtigen Komponenten für „echten Spaß" *(siehe Abbildung 1)*: Spielerisches Herangehen (Playfulness), Flow und Verbundenheit (Connection)

Schritte für mehr Happiness

Lange Zeit wurde selbst von der Wissenschaft angenommen, dass das Glücklichsein eine angeborene Eigenschaft ist. Doch inzwischen ist das widerlegt; vielmehr hat jeder Mensch die Chance, glücklich zu sein – wir müssen es nur wollen und daran arbeiten. Mit einfachen Techniken und kleinen Schritten können wir lernen, glücklich zu sein. Da diese Schritte Verhaltensänderungen mit sich bringen, müssen wir auch wollen. Als dritter Faktor kommt noch das „Dürfen" ins Spiel: Lässt unser Umfeld es überhaupt zu, dass wir glücklich sind/sein wollen? So kommen auch auf dem Weg zum Glücklichsein die Bausteine des Können-Wollen-Dürfen-Modells *[4]* zum Tragen *(siehe Abbildung 2)*.

Um das Können einfacher zu gestalten, folgen hier sechs kleine, leicht verständliche Schritte:

1. Suche Dinge, über die du dich freuen kannst!

Die Welt ist nicht nur schlecht, negativ oder böse! Selbst in Zeiten von Krieg, Wirtschaftskrise oder Krankheit gibt es viele kleine Highlights, über die wir uns jeden Tag freuen können. Wir können auch *Mudita* praktizieren. Mudita kommt aus dem Buddhistischen und bedeutet Mitfreude: wir freuen uns über etwas Gutes, das andere erlebt haben! Auch wenn wir anderen helfen oder anderen eine Freude machen, ist es etwas, über das wir uns später selbst freuen können. In diesem Schritt richten wir aktiv unseren Blick weg von den negativen Dingen hin zum Positiven. Wenn du einen Schritt weitergehen willst, kannst du dir jeden Tag ein paar Minuten Zeit nehmen und aufschreiben (visualisieren), was du am Tag Gutes erlebt hast. Alternativ funktioniert das auch als *Dankbarkeitsliste*. Eine gute Hilfe ist an dieser Stelle das *Journaling*.

2. Verbrenne negative Gedanken!

Wenn ein bestimmter negativer Gedanke dich immer wieder quält, obwohl die Situation schon lange vorbei ist oder es genügend Ge-

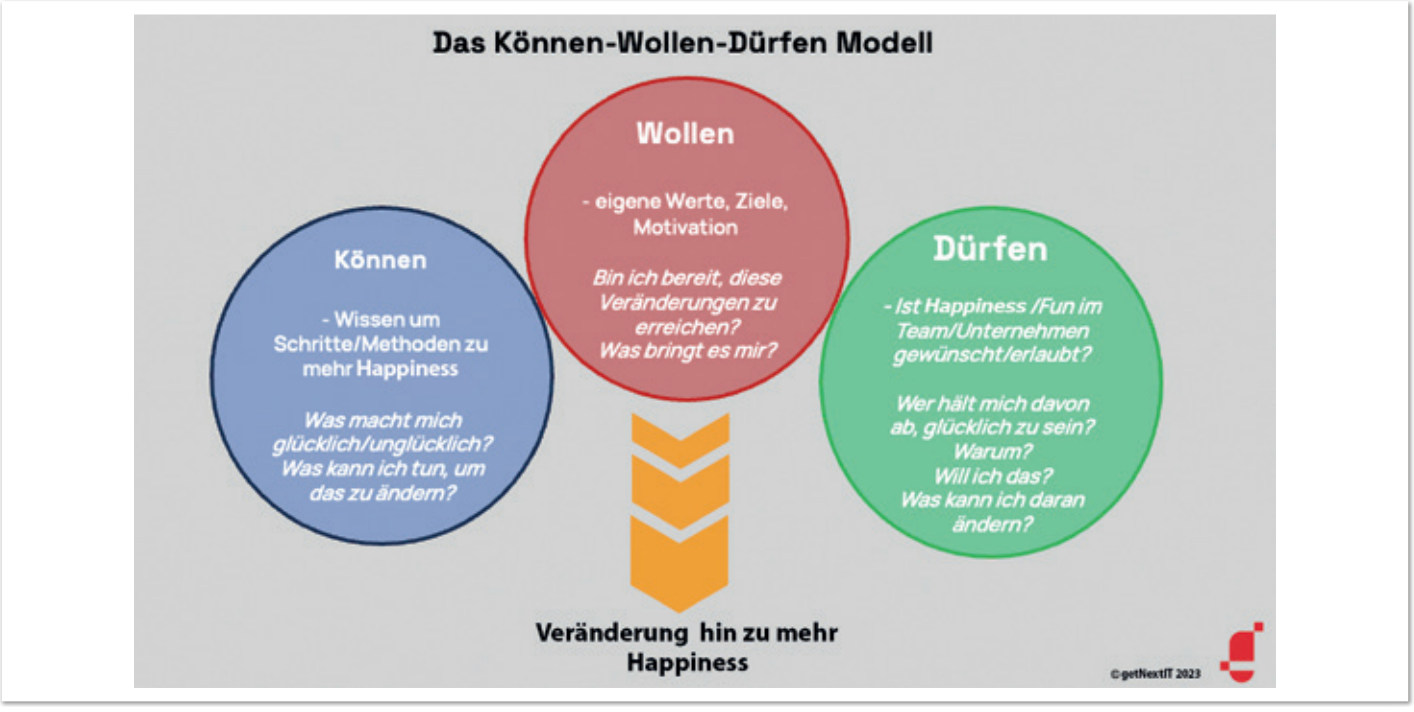


Abbildung 2: Das Können-Wollen-Dürfen-Modell mit Fragestellungen (© Sabine Wojcieszak)

genbeweise gibt, kann es dir helfen, visuell diesen Gedanken zu vernichten. Schreibe den Gedanken auf einen Zettel und verbrenne den Zettel! Spielkram? Nein, die Macht der Visualisierung auch für solche Situation ist längst wissenschaftlich bewiesen – und nicht nur für Kinder! Und sollte der Gedanke doch wieder auftauchen, hilft die Visualisierung deinem Gehirn, loszulassen!

3. Lächle!

Lächeln hat eine magische Wirkung – auf andere und auf uns selbst! Lächeln wir andere an, so werden sie in den meisten Fällen zurücklächeln – und wir bekommen eine positive Rückmeldung, die Stimmung ist besser! Allerdings setzt unser Lächeln auch bei uns selbst positive Prozesse in Gange – selbst wenn das Lächeln (zunächst) gefakt ist. Lächeln wir (ehrlich oder gefakt), fängt unser Gehirn an, Endorphine und Dopamin zu produzieren – Botenstoffe für unser Glücksempfinden. Gleichzeitig wird der Cortisol-Spiegel gesenkt, was zu mehr Entspannung führt!

Und wenn das Lächeln schon so gut wirkt, ist das herzhafte Lachen ebenso wichtig und hilfreich. Jennifer Aaker und Naomi Bagdonas geben an der Stanford Business School einen Kurs „Power of Humor in Leadership", in dem es darum geht, wie wir als Führungskraft Humor einsetzen können, um öfter zu lachen und Spaß zu haben. Dieser Kurs ist an der Universität gleichwertig zum Kurs „Finanzbuchhaltung". In ihrem Buch „Humor, Seriously" *[5]* berichten sie darüber, wieso Humor so eine Superpower ist und wie wir alle sie erlernen können. *Aber Achtung: Niemals machen wir Späße auf Kosten anderer oder über andere! – Leider ist das noch eine weit verbreitete Kultur in Unternehmen!*

4. Mach deinen Frieden mit dir selbst!

Du bist gut genug! Vergleiche dich nicht mit anderen. Deinen Erfolg kannst du nur an dir messen. Lerne deine Stärken kennen. Und vielleicht wirst du entdecken, dass andere bei dir etwas als Stärke sehen, was du selbst für dich als Schwäche wahrnimmst. Stärken und Schwächen sind nichts Absolutes, sondern müssen immer bezogen auf den Kontext beziehungsweise die Perspektive betrachtet werden. Es geht nicht um Perfektionismus, sondern um Authentizität!

5. Tanze!

Ja, richtig! Das Leben ist der richtige Platz zum Tanzen, denn auch Tanzen versetzt unser Gehirn in Glücksstimmung – sogar die selbsterklärten Nicht-Tanzenden! Musik hat schon eine ausgesprochen intensive Wirkung auf unsere Stimmung; in Kombination mit Bewegung (Musik + Bewegung = Tanzen) ist das für unser Gehirn der absolute Glücksrausch!

Wenn du jetzt denkst: „Ich kann doch aber gar nicht tanzen!", dann solltest du dir den Punkt 4 noch einmal durchlesen! Also erstelle dir eine Playlist mit deiner persönlichen *Gute-Laune Musik* und dann Musik an und Tanzen!

6. Choreografiere dein Leben!

Nimm dein Leben selbst in die Hand und lass dich nicht von anderen bestimmen. Werde dir klar darüber, was dich glücklich macht und was nicht! Suche und frage aktiv nach diesen Aspekten zu deinem Glück! Trenne dich von Dingen, die dich unglücklich machen und setze Grenzen! Gestalte deine Rolle – auch im Job – so, dass du dich darin wiederfindest. Choreografie verbinden wir gedanklich häufig mit Tanzen und Kreativität. Genau wie das Tanzen ist unser Leben ständig in Bewegung und wir benötigen Kreativität, um leichtfüßig all die neuen Herausforderungen zu bewältigen. Wie wichtig es ist, das eigene Leben zu choreografieren, beschreibt Valorie Kondos Fields, ehemalige Head Coach für das Gymnastik-Team an der UCLA in ihrem Buch „Life is short, don't wait to dance" *[6]*. Und zu dieser Choreografie gehört auch das „*Falling up*", die Einstellung, dass jeder Fehlschlag uns wachsen lässt – wenn wir denn die Chance nutzen und den Fehlschlag nicht als Niederlage, sondern als Lerngewinn annehmen.

Zugegeben, Schritt 6 ist der anspruchsvollste. Aber auch hier gilt: Es geht nicht um perfekt, es geht vielmehr um Schritt für Schritt. Mit jeder Entscheidung, die du für dich triffts, triffst du eine Entscheidung für das Glücklichsein.

Diese sechs Schritte für das Glücklichsein beschreiben sehr persönliche Aspekte. Dennoch sind diese Aspekte wichtig für die Performance und den Erfolg auf der Arbeit. Je glücklicher ich als Individuum bin, desto besser bin ich gerüstet für eine höhere Performance und mehr Erfolg. Damit beeinflusse ich auch den Erfolg und die Performance meines Teams. Viel wichtiger ist aber, dass ich durch meine Entscheidung, glücklicher zu sein, automatische Einfluss auf andere nehme. Meine positiven Reaktionen, mein Lächeln – die guten Vibes – wirken sich positiv auf andere aus, ohne dass sie sich bewusst dazu entschieden haben, glücklicher zu werden. Psychologen gehen davon aus, dass eine Person, die sich entschieden hat, glücklich zu sein, damit einen Impact auf rund 1.000 Leute (bis hin zum 3. Vernetzungsgrad) haben. Dies wird auch als der Butterfly-Effekt bezeichnet. Ganz besonders für Führungskräfte ist das ein wichtiger Hebel, denn sie haben hier die Chance, die Performance und den Erfolg im Team über die eigene, positive Haltung aktiv zu verbessern.

Mehr Spaß bei der Arbeit – aber wie?

Wenn es um den echten Spaß bei der Arbeit geht, dann ist die Welt der Agilität eine hilfreiche Herangehensweise. Mit agilen Methoden und Ansätzen können wir das Arbeitsumfeld so gestalten, dass wir echten Spaß empfinden können *(siehe Abbildung 3)*.

Playfulness ist in der agilen Welt fest verankert. Experimente sind eine spielerische Herangehensweise an ein Problem. Visualisierungsmethoden wie Team Canvases, Fun-Retrospektiven oder Planning Poker enthalten dieser spielerischen Ansätze.

Flow ist derzeit ein viel diskutiertes Thema. Arbeit fertigbekommen und all die Impedimente oder unnötigen Arbeitsschritte erkennen und beseitigen, sodass wir keinen *Waste* produzieren, ist sowohl im agilen Kontext als auch im Lean-Ansatz das erklärte Ziel. Nur so können wir unseren Kunden schnell einen Wert (Value) liefern und erfolgreich sein.

Connection bezieht sich im Arbeitskontext auf zwei Dinge: Zum einen ist es die Verbindung zu den Aufgaben und den übergeordneten Zielen. „Was ist mein Betrag dazu? Welchen Wert liefere ich?", sind hier wichtige Fragen. Diese werden in den agilen Methoden dadurch adressiert, dass wir das Big Picture kennen, Feedback bekommen und echte Wertschätzung erfahren. Zum anderen bezieht sich die Connection aber auch auf die zwischenmenschlichen Verbindungen. Auch hier beinhalten agilen Methoden viele Aspekte, die diese Verbindungen stärken sollen. Kollaboration, kleinen Teams, Teammeetings, Teamentscheidungen oder gegebenenfalls ein Scrum Master oder Agile Coach, der sich auch um die zwischenmenschlichen Aspekte kümmert, sind gute Ansätze hierbei.

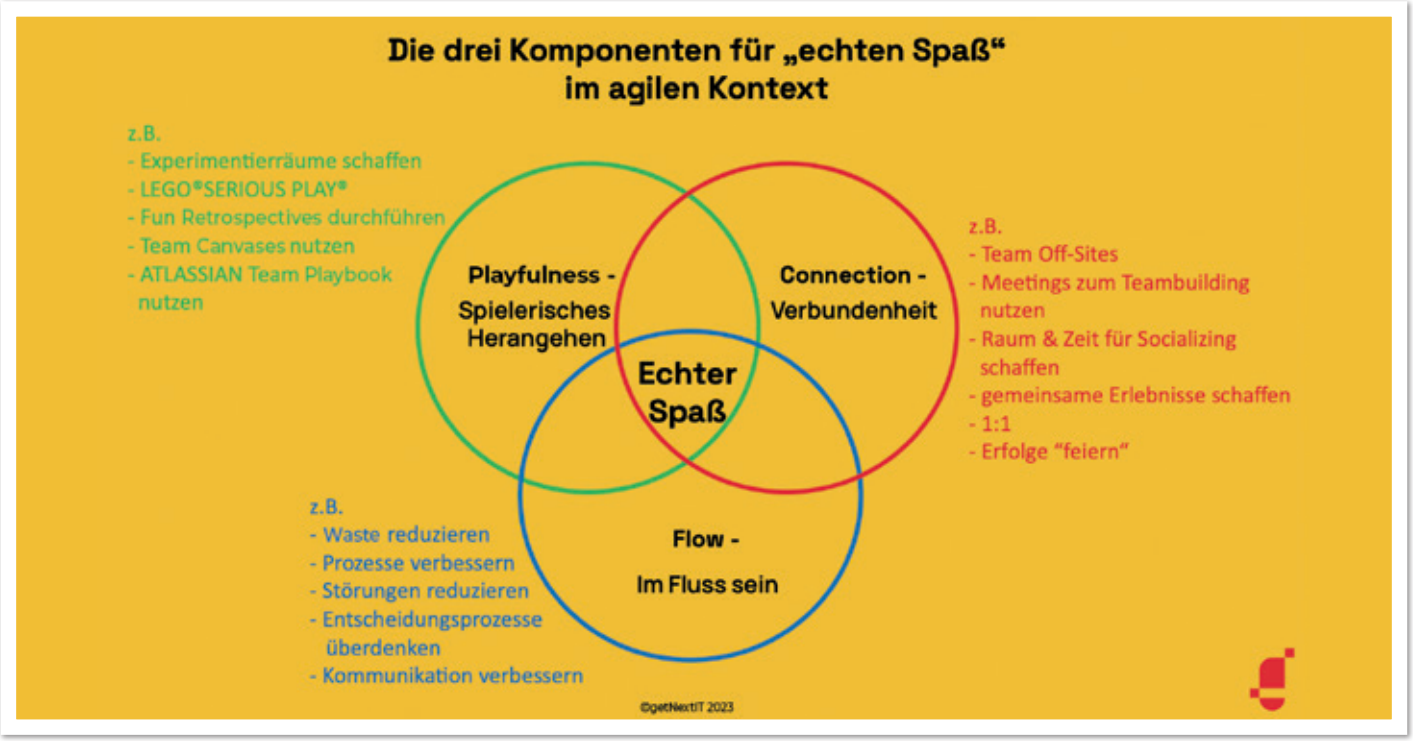


Abbildung 3: Der „echte Spaß“ im agilen Arbeitskontext (© Sabine Wojcieszak)

Wie viel jede einzelne Person bereit ist, an Zeit und Aufmerksamkeit für andere zu geben wird als soziales Investment bezeichnet. Dieses soziale Investment stärkt nicht nur die Connection-Ebene; es macht vielmehr auch glücklich. Anderen einen Gefallen tun, anderen helfen, sich Zeit nehmen – all das ist soziales Investment.

Wir sind in der agilen Welt also bestens dafür vorbereitet, echten Spaß bei der Arbeit zu schaffen. Allerdings geht es einmal mehr darum, zu verstehen, warum die Dinge so gemacht werden sollen. Wer einfach nur „Agile by the book“ macht, ohne den Wert und den Sinn dahinter zu verstehen, wird auch nicht an den richtigen Schrauben drehen, um den Spaß aufs Spielfeld zu holen.

Happiness und Spaß – echte Business-Vorteile

Wir leben in Zeiten voller Herausforderungen – im Privaten wie im Business. Veränderung ist die neue Norm; Lernen gehört heute zur täglichen Arbeit mit dazu. Daher brauchen wir Mitarbeitende, die offen sind für Neues, die lernen wollen, die resilient sind und kreative Lösungen finden können und wollen. Wenn wir also wissen, welche Skills und welches Mindset wir brauchen und uns gleichzeitig die Wissenschaft aufzeigt, was dazu beiträgt, dass Menschen genau in diesen Zustand der Leistungsfähigkeit kommen, dann tun wir gut daran, Happiness und Spaß nicht mehr als Spielerei abzutun. Sie bewirken genau das in den Menschen, was notwendig ist, um eine gute Performance zu zeigen und erfolgreich zu sein. Sowohl jede einzelne Person als auch Unternehmen sind verantwortlich dafür, an diesen Punkten zu arbeiten. Um mit den Worten von Shawn Achor zu schließen: „Es ist Zeit dafür, dass wir Happiness nicht mehr nur als Stimmung sehen, sondern als Arbeitsethik betrachten!“

Quellen

[1] Achor, Shawn (2010): *The Happiness Advantage: The seven principles that fuel success and performance at work.* Virgin Books, UK.

[2] Frederickson, B. L. (2001): *The role of positive emotions in positive psychology: The broaden-and-build theory of positive emotions.* American Psychologist, Washington, D.C.

[3] Price, Cathrine (2021): *The power of fun: How to feel alive again.* The Dial Press New York.

[4] Haufe-Evolve-Think-Tank: *https://www.haufe-akademie.de/blog/themen/personalentwicklung/selbstlernende-organisation-ein-unternehmen-auf-den-weg/*

[5] Aaker, J. and Bagdonas, N. (2020): *Humor, seriously. Why humor is a superpower at work and in life.* Penguin Random House, UK

[6] Kondos Field, Valorie u. Cooper, St. (2018): *Life is short, don´t wait to dance.* Hachette Book Group, New York

Sabine Wojcieszak
getNext IT
sabine@getnext-it.com

Sabine Wojcieszak ist die Enthusiastic-Success-Enablerin bei getNextIT in Kiel. Sie coacht Unternehmen, Führungskräfte, Teams und Individuen auf dem Weg zu einer besseren Zusammenarbeit – für den gemeinsamen Erfolg. Sie ist davon überzeugt, dass Erfolg nur mit und durch die Menschen im System erreicht werden kann. Happiness und Spaß sind unerlässlich für sie.

Sabine ist regelmäßig als Sprecherin auf internationalen Konferenzen anzutreffen und schreibt Blogs und Artikel Blogs zu diesen Themen. Als Speaker-Coach unterstützt sie Sprecher:innen dabei, mehr Sichtbarkeit und Impact zu erreichen.

JavaLand

www.javaland.eu

AM NÜRBURGRING

09. – 11.04. 2024

JETZT TICKETS

Präsentiert von: iJUG Verbund · Heise Medien · DOAG

NEUE LOCATION!

SICHERN!

Veranstalter: JavaLand

Mitglieder des iJUG

01 BED-Con e.V.
02 Clojure User Group Düsseldorf
03 DOAG e.V.
04 EuregJUG Maas-Rhine
05 JUG Augsburg
06 JUG Berlin-Brandenburg
07 JUG Bremen
08 JUG Bielefeld
09 JUG Bonn
10 JUG Darmstadt
11 JUG Deutschland e.V.
12 JUG Dortmund
13 JUG Düsseldorf rheinjug
14 JUG Erlangen-Nürnberg
15 JUG Freiburg
16 JUG Goldstadt
17 JUG Görlitz
18 JUG Hannover
19 JUG Hessen
20 JUG HH
21 JUG Ingolstadt e.V.
22 JUG Kaiserslautern
23 JUG Karlsruhe
24 JUG Köln
25 Kotlin User Group Düsseldorf
26 JUG Mainz
27 JUG Mannheim
28 JUG München
29 JUG Münster
30 JUG Oberland
31 JUG Ostfalen
32 JUG Paderborn
33 JUG Saxony
34 JUG Stuttgart e.V.
35 JUG Switzerland
36 JSUG
37 Lightweight JUG München
38 SUG Deutschland e.V.
39 JUG Thüringen
40 JUG Saarland
41 JUG Duisburg

iJUG Verbund
www.ijug.eu

Impressum

Java aktuell wird vom Interessenverbund der Java User Groups e.V. (iJUG) (Tempelhofer Weg 64, 12347 Berlin, *www.ijug.eu*) herausgegeben. Es ist das User-Magazin rund um die Programmiersprache Java im Raum Deutschland, Österreich und Schweiz. Es ist unabhängig von Oracle und vertritt weder direkt noch indirekt deren wirtschaftliche Interessen. Vielmehr vertritt es die Interessen der Anwender an den Themen rund um die Java-Produkte, fördert den Wissensaustausch zwischen den Lesern und informiert über neue Produkte und Technologien.

Java aktuell wird verlegt von der DOAG Dienstleistungen GmbH, Tempelhofer Weg 64, 12347 Berlin, Deutschland, gesetzlich vertreten durch den Geschäftsführer Fried Saacke, deren Unternehmensgegenstand Vereinsmanagement, Veranstaltungsorganisation und Publishing ist.

Die DOAG Deutsche ORACLE-Anwendergruppe e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Björn Bröhl. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. informiert kompetent über alle Oracle-Themen, setzt sich für die Interessen der Mitglieder ein und führen einen konstruktiv-kritischen Dialog mit Oracle.

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
ViSdP: Fried Saacke
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

Redaktionsbeirat:
Andreas Badelt, Marcus Fihlon, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, Bennet Schulz

Titel, Gestaltung und Satz:
Alexander Kermas,
DOAG Dienstleistungen GmbH

Bildnachweis:
Titel: Bild © Maxim
https://stock.adobe.com
S. 12 + 13: Bild © Cienpies Design
https://stock.adobe.com
S. 25: Bild © moodboard
https://stock.adobe.com
S. 31: Bild © Designed by Freepik
https://freepik.com
S. 32 + 33: Bild © Designed by svstudioart
https://freepik.com
S. 41: Bild © Designed by storyset
https://freepik.com
S. 46 + 47: Bild © Designed by storyset
https://freepik.com
S. 54 + 55: Bild © Dlgilife
https://stock.adobe.com
S. 58 + 59: Bild © Nuthawut
https://stock.adobe.com
S. 68: Bild © theromb
https://stock.adobe.com

Anzeigen:
DOAG Dienstleistungen GmbH
Kontakt: sponsoring@doag.org

Mediadaten und Preise:
www.doag.org/go/mediadaten

Druck:
WIRmachenDRUCK GmbH
www.wir-machen-druck.de

Inserentenverzeichnis

APEX connect

by DOAG

22. - 24.04.2024

VAN DER VALK AIRPORTHOTEL DÜSSELDORF

apex.doag.org