# Java aktuell

Ausgabe 04/2023

**Java 20**
Was gibt's Neues?

**JVM-Sprachen**
Kotlin und Clojure

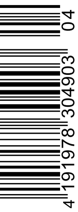
**Java**
Datenorientierte Programmierung, OpenAPI Style Validator

## JVM-Sprachen

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



www.cloudland.org

# CloudLand 2023

## DAS FESTIVAL DER DEUTSCHSPRACHIGEN CLOUD NATIVE COMMUNITY

• Microservices & DDD • CI/CD & Automatisierung
• Container & Cloud-Technologien • DevOps & Methodik

## 20. - 23. JUNI

im Phantasialand in Brühl

Tag 1 / 20. Juni
CloudCamp

Tag 2 / 21. Juni
Summer Night

Tag 3 / 22. Juni
Gaming Night

Tag 4 / 23. Juni
Customer Stories

#CloudLand2023

Eventpartner:

# Liebe Leserinnen und Leser,

willkommen zur Ausgabe 4/2023 der Java aktuell mit dem Schwerpunktthema "JVM-Sprachen". In dieser Ausgabe könnt ihr euch über zahlreiche Fachartikel freuen.

Los geht es mit Neuigkeiten rund um Java und die Community. Das Java-Tagebuch und die Eclipse Corner bringen euch wieder auf aktuellen Stand rund um die wichtigsten Ereignisse. Im Anschluss stellt Falk Sippach das im März erschienene OpenJDK 20 vor. Er beleuchtet aktuelle Neuerungen und wagt auch einen Blick in die Zukunft, was uns im nächsten Long-Term-Support-Release 21, das bereits im September erscheint, voraussichtlich erwartet. Einen weiteren Exkurs bietet Marc Hoffmann, der uns das Open-Data-Projekt Java-Almanac näher vorstellt. Was mit einem Notizzettel begann, hat sich mittlerweile zu einer Enzyklopädie vergangener und zukünftiger Java-Versionen entwickelt.

Ab Seite 26 wirft Fredrik Winkler mit uns gemeinsam einen Blick über den Tellerrand und portiert eine funktionale Java-Anwendung beispielhaft nach Kotlin und Clojure. Dabei arbeitet er Unterschiede heraus, um uns so die beiden Sprachen schmackhaft zu machen. Im Anschluss teilt Lars Adler seine Erfahrungen im Umzug von Java-Code nach Kotlin. Dabei geht er auf Herausforderungen ein und gibt Tipps und Tricks, die die Migration erleichtern.

Merlin Bögershausen zeigt ab Seite 40, wie man mit einem datenorientierten Ansatz eine einfache Rechnungserstellung implementieren kann. Dabei wendet er eine Vielzahl von Sprachfeatures aus dem Project Amber an. Im Anschluss daran stellt Claudio Altamura den OpenAPI Style Validator näher vor, der dabei unterstützen kann, API-Richtlinien umzusetzen. Der Validator hilft, unvollständige Beschreibungen sowie inkonsistente Nutzung von Namenskonventionen aufzudecken.

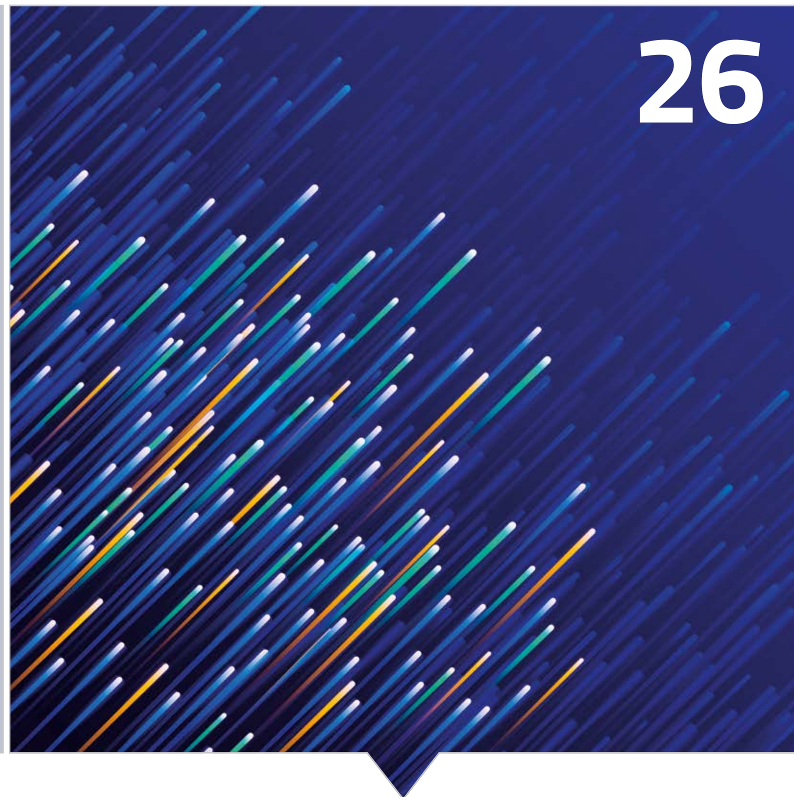
Abgerundet wird diese Ausgabe durch Joschua Töpfers Beitrag zum Remote Mob Programming. Ein Widerspruch in sich? Überhaupt nicht! Findet heraus, wie ihr diesen Zusammenarbeitsansatz trotz Remote Work umsetzen könnt.

Wir wünschen euch viel Spaß beim Lesen!
Eure

**Lisa Damerow**
Redaktionsleitung Java aktuell

# Inhalt

Java 20: Was ist neu?




Was erwartet einen Java-Developer in Kotlin und Clojure?


**3** Editorial

**6** Java-Tagebuch
*Andreas Badelt*

**9** Markus' Eclipse Corner
*Markus Karg*

**10** Was gibt es Neues in Java 20?
*Falk Sippach*

**20** Vom privaten Notizzettel zum Open-Data-Projekt javaalmanac.io
*Marc Hoffmann*

**26** Ein Blick über den Java-Tellerrand: Kotlin und Clojure
*Fredrik Winkler*

*Migration von Java nach Kotlin: Herausforderungen und Best Practices*

*Remote Mob Programming – ein Widerspruch in sich?*


**34** The Great Migration: Erfahrungen aus dem Umzug von Java nach Kotlin
*Lars Adler*

**40** Java can do that too! – Data Oriented Programming
*Merlin Bögershausen*

**48** Validierung von OpenAPI-Spezifikationen mit dem OpenAPI Style Validator
*Claudio Altamura*

**54** Remote Mob Programming – zuhause, aber nicht allein
*Joshua Töpfer*

**58** Impressum/Inserenten

## 27. Januar 2023

### Oracle stellt Java-Preisliste um

Oracle hat seine Preisliste für Java angepasst: Die neue „Java SE Universal Subscription" wird nach Größe des Unternehmens bezahlt – bis 999 Mitarbeiterinnen und Mitarbeiter kostet diese 15 US-Dollar im Monat, entsprechend etwas weniger in Euro – pro Person versteht sich! Wobei man die Definition im Kleingedruckten genau lesen muss *[1]*. Dazu gehören *alle*, die die „internal business operations" des Unternehmens unterstützen; unabhängig davon, ob sie Java nutzen oder nicht, und unabhängig von ihrem Anstellungsverhältnis – also beispielsweise auch befristet Angestellte oder Externe (Outsourcing, Beratung, andere Dienstleister…). Ab einer Größe von 40.000 sind es dann „nur" noch gute 5 Dollar pro Person und Monat. Die Konkurrenz frohlockt, und Eclipse-Chef Mike Milinkovic wirbt für die kostenlosen Temurin-OpenJDK-Versionen des Adoptium-Projekts (im Grunde identisch zu denen von Oracle – alles basiert ja direkt auf dem OpenJDK). Wobei zu Oracles Ehrenrettung gesagt werden muss, dass in der Subscription natürlich etwas mehr enthalten ist als JDK-Downloads – neben Support gehört zum Beispiel auch die Enterprise-Edition von Oracles GraalVM dazu.

Allerdings nicht ganz unwichtig: Die Lizenzbedingungen werden beim Download akzeptiert! Mit anderen Worten: Wenn nur ein einziger Mitarbeiter im Unternehmen ein Oracle JDK herunterlädt und nutzt, werden prinzipiell Lizenzgebühren basierend auf der Gesamtpersonenzahl fällig, und nicht mehr nur für diesen einen Mitarbeiter! Manche Unternehmen haben deswegen wohl schon die Downloads gesperrt *[2]*.

## 3. März 2023

### Adoptium: Aktualisierter Programmplan 2023

Auch Open-Source-Projekte können tolle Foliensätze produzieren. Meine ich jetzt gar nicht mal ironisch. Der „Adoptium Working Group Program Plan 2023" enthält viele Details zu den Strategien und Zielen für das laufende Jahr. Vieles davon ist natürlich für die Java-Entwicklerinnen und -Entwickler nicht so relevant. Aber die Folie mit den Prioritäten für Plattform-Builds, passenderweise mit „Consider Stopping" betitelt, vielleicht schon. Zumindest für diejenigen, die für eher exotische Plattformen entwickeln. Um die vorhandenen Kapazitäten besser zu nutzen, sollen nämlich Plattformen mit sehr geringen Download-Zahlen nicht mehr beziehungsweise nur noch für die Long-Term-Support-Versionen berücksichtigt werden. Wer sich betroffen wähnt, schaut hier nach *[3]* (Folien 27f.).

Dabei fällt mir auf, dass Java 8 bereits jetzt nicht für „Silicon"-Chips unterstützt wird – bis gerade hatte ich es nicht vermisst… komplett in die Röhre schauen aber alle „Besitzer" eines x86-32 Windows (schon jetzt nur eine „Prio 2"-Plattform), wenn Java 21 rauskommt. Aber gut, auch mit Java 17 lassen sich im Hardware-Museum tolle Sachen zeigen. Im OpenJDK-Projekt gibt es seit Kurzem den dazu passenden Entwurf für einen JEP, um diesen Port als „deprecated" zu markieren und in einem zukünftigen Release ganz einzustellen. Der JEP ist noch ohne Zuordnung, das Einstellen wird sicher erst nach 21 passieren. Das Hauptproblem sind virtuelle Threads; die aktuelle Implementierung für 32-Bit-Windows ist eigentlich ein „Fake", in dem virtuelle Threads doch wieder 1:1 auf Betriebssystem-Kernel-Threads gemappt werden (das widerspricht zwar dem Prinzip der virtuellen Threads und skaliert natürlich nicht, aber dann funktionieren sie zumindest grundsätzlich). Die Arbeit, das zu ändern, steckt man da lieber in zukunftsorientierte Themen.

## 10. März 2023

### MicroProfile & Jakarta EE – es ist kompliziert

Die Planungen für MicroProfile 7.0 sind längst gestartet, Ziel ist ein Release-Datum im Sommer (voraussichtlich Juni). JWT 3 ist ein Kandidat. Außerdem OpenTelemetry Metrics und dazu dann das Entfernen des „alten" MP Metrics aus der Plattform-Spezifikation („Umbrella"); ob das rechtzeitig zum Sommer fertig wird, ist aber noch unklar. Ein Großteil der Diskussionen in der Working Group richtet sich jedoch auf das Verhältnis zu Jakarta EE – technisch und organisatorisch. Nicht zuletzt aufgrund der etwas emotionaleren Diskussion kurz vor Fertigstellung von MicroProfile 6.0, welches EE-Release denn etwa für Kompatibilitäts-Zertifizierungen zugrunde zu legen sei. Vier Vorschläge sind auf den Tisch gekommen.

Erstens: Ein MicroProfile-Release definiert eine minimale Jakarta-EE-Version (auf Plattform-Ebene, nicht in den Einzel-Specs), Implementierungen können auch höhere Versionen unterstützen. Was aber unter anderem bedeutet, dass Jakarta auch in Zukunft keine MP-Spezifikationen referenzieren darf (wie es für Jakarta EE Security und MP JWT gerade diskutiert wird) – nur ein Transfer oder ein Fork der MP-Spezifikation wären erlaubt. Außerdem bedeutet es in Bezug auf Pflege und Testen signifikanten (Mehr-) Aufwand.

Zweitens: Der aktuelle MicroProfile „Umbrella" wird zu MicroProfile Full, basierend auf einer spezifischen Jakarta-EE-Version, flankiert von einem MicroProfile Lite ohne jede Abhängigkeit von Jakarta. Dieser Vorschlag ist aber bereits als zu kompliziert verworfen worden.

Drittens: Neue MicroProfile-Releases werden (auch zeitlich) auf Jakarta EE und Java SE LTS Releases abgestimmt, mit dem dadurch erforderlichen Koordinationsaufwand. Laut Jan Westerkamp vom iJUG, Befürworter dieser Variante, wäre man hier unter anderem auch am nächsten am „Semantic Versioning" dran, da sich „Breaking Changes" in Java SE LTS oder Jakarta-EE-Versionen in MP dann ebenfalls in einem Major Release manifestieren würden.

Viertens: MicroProfile hört als eigenständige Working Group auf zu existieren, überträgt seine Spezifikationen und wird zum Jakarta EE MicroProfile (analog dem Core, Full oder WebProfile von EE). Das ist aber schon rein organisatorisch eine größere Aufgabe und daher zumindest für Release 7 ebenfalls vom Tisch.

Optionen 1 und 3 sollen weiter diskutiert und demnächst zur Ab-

stimmung gebracht werden. Stay tuned! Wer mehr Details dazu möchte, kann das ganz gut hier nachvollziehen: *[4]*.

## 15. März 2023

### Verhältnis von MP JWT und Jakarta Security wird diskutiert

Zum konkreten Thema MP JWT und Jakarta EE Security haben die Projekte heute bilaterale Verhandlungen aufgenommen. Auch hier sind wieder eine Reihe von Vorschlägen entwickelt worden, von simpel (JWT Spec wandert zu Jakarta) bis eher kompliziert (neue Stand-alone-Spezifikation in MP, die beide Projekte „bedient"), die in den nächsten Wochen weiter diskutiert werden *[5]*.

### Quarkus 3 Alpha Release(s)

Quarkus 3.0 ist im Anmarsch, wie schon im vorletzten Tagebuch berichtet. Die erste Alpha-Version wurde bereits im November freigegeben. Inzwischen sind wir bei Alpha 6, mit der Quarkus unter anderem volle Unterstützung für Jakarta EE 10 bietet, inklusive des zunächst noch ausgelassenen JPA (mit Alpha 5 kam der Sprung auf Hibernate ORM 6, und damit auch Unterstützung für die neue JPA-Version). Aus Sicht der Nutzer ist damit Jakarta EE 9 komplett übersprungen worden. Es ging ja im Wesentlichen um neue Package-Namen, nicht um neue Funktionalität. Nach Aussage des Quarkus-Teams hat sich der Zwischenschritt aber als sehr nützlich erwiesen. Hier nochmal die wichtigsten neuen Features: Im Tandem mit EE 10 natürlich Support für das neue MicroProfile 6. Außerdem ist HTTP/3 dabei, ein neuer gRPC-Server, erster Support für virtuelle Threads und „Structured Concurrency", und für Performance-Spezialisten „io_uring" (Kernel-gestützter asynchroner I/O für Speichergeräte).

### Temurin-Java-Downloads verdoppelt

Die Zahl der Temurin-Java-Downloads aus dem Eclipse-Adoptium-Projekt hat sich laut InfoWorld über die vergangenen 12 Monate mehr als verdoppelt. Daran kann Oracles neue Preisliste noch nicht „schuld" sein (zumindest nicht wesentlich), aber anscheinend hat sich die Entwicklung zuletzt noch einmal signifikant beschleunigt. Die aktuellen Zahlen lassen sich ja direkt dem Dashboard *[6]* beziehungsweise dem darunterliegenden API entnehmen. Auch Anbieter wie Azul, die ihre Binaries über den Adoptium-Marktplatz zum Download anbieten, aber darüber hinaus auch Support anbieten (mit anderen Lizenzmodellen), berichten laut InfoWorld über mehr Interesse.

## 16. März 2023

### Adoptium: Mitglieder-Repräsentanten gewählt

Für die Adoptium Working Group bei Eclipse wurden die Mitglieder-Repräsentanten neu gewählt. Allzu viel Auswahl gab es nicht. Die „strategischen" Mitglieder sind automatisch vertreten, also ging es noch um drei Rollen: Bei den „Enterprise"-Mitgliedern ist mit Bloomberg und IBM (aufgestellt und gewählt wurde Murali K Veeravalli von IBM) die Auswahl der Firmen genauso klein wie bei den „normalen" Mitgliedern mit dem iJUG und der Open Elements GmbH. Letztere sind in Zukunft im Grunde beide vertreten, da Hendrik Ebbers für Open Elements aufgestellt wurde, aber gleichzeitig ja auch im iJUG aktiv ist. Hoffen wir, dass keine Interessenkonflikte aufkommen. Der dritte im Bunde ist weiterhin Stewart Addison (Infrastructure Lead im Adoptium-Projekt), der die „Committer" vertritt.

### Neues Eclipse-DIE-Release

Die Version 2023-03 der Eclipse IDE ist erschienen, ein simultanes Release von insgesamt 65(!) Projekten, von Eclipse Acceleo (Model zu Code Generator) bis Eclipse XWT (deklarative UIs). Für Java-Entwickler gibt es unter anderem eine Reihe von Verbesserungen bei der Codevervollständigung, rekursives Starten von Unittests in Subpackages und die direkte Integration mehrerer Views, insbesondere die Bytecode View, die sonst als Plug-ins nachinstalliert werden mussten.

Was die Codevervollständigung beziehungsweise das Language Server Protocol (LSP) angeht: Analog zum LSP4J (Binding für Java) arbeitet das Jakarta-Projekt aktuell an einem LSP4Jakarta *[7]*.

## 21. März 2021

### Java SE 20

Pünktlich zur JavaLand ist Java SE 20 erschienen (genau auf dieses Event hat das OpenJDK-Projekt sicher von Anfang an hingearbeitet). Ich wiederhole jetzt aber nicht noch mal alle neuen Features – stehen ja eh hier: *[8]* und im Artikel von Falk ab **Seite 10**. Außerdem sind es ausnahmslos Inkubator- und Preview-Features. Spannend wird es dann im Sommer, welche davon in Java 21 wirklich den Stempel „produktionsreif" erhalten.

Zu Java 21 gibt es noch nicht viel Neues, außer dass „Sequenced Collections" (JEP 431) als erstes Feature eingeplant wurden. Die Liste *[9]* wird sich sicher bald weiter füllen.

## 23. März 2023

### JavaLand mit Besucherrekord

Zwei kurze Tage nur, dann ist sie schon wieder vorbei, die JavaLand-Konferenz. Ok, vier, wenn man die JavaLand4Kids, die selbstorganisierende Unkonferenz am Montag sowie den Schulungs-Donnerstag dazuzählt. Was ziemlich eindeutig ist: Es gab mit fast zweieinhalbtausend Teilnehmerinnen und Teilnehmern einen neuen Besucherrekord. 2019 wurde zum ersten Mal die 2.000er-Marke durchbrochen. Die erste „Präsenz"-JavaLand nach zwei Jahren Zwangspause lief letztes Jahr mit 1.200 Java-Fans noch etwas verhalten an. Doch inzwischen sind die Unsicherheiten auch in den Firmen wohl verschwunden – und alle hungern nach Events, in denen nicht nur inhaltliche Druckbetankung geboten wird, sondern das „Networking" (wie hieß das eigentlich früher) ganz großgeschrieben wird. Nur die internationale Beteiligung hat etwas nachgelassen, die JavaLand ist ein bisschen „deutscher" geworden. Doch das könnte sich auch schon nächstes Jahr wieder ändern.

Bemerkenswert war auf jeden Fall die Community-Keynote, die sich dem Themenkomplex Mobbing, Bossing und Sexismus am Arbeitsplatz gewidmet hat, wobei aus der Community im Vorfeld eigene Geschichten dazu eingereicht werden konnten, die dann anonymisiert auf die Bühne gebracht wurden. Soweit ich weiß,

wird diese Keynote auch demnächst veröffentlicht (für alle, nicht nur wie die meisten Sessions nur für „Ticketinhaber“).

Aber auch „live“ wird für die Daheimgebliebenen ja inzwischen einiges geboten: Der Live-Stream, der alle Sessions aus dem Wintergarten überträgt, wurde immerhin bis zu 173-mal parallel genutzt. Was ja mehr als 173 Personen sein könnten. Falls es noch niemand macht, wäre das ja mal eine Idee als Entschädigung für Mitarbeiterinnen und Mitarbeiter, die nicht zur JavaLand fahren konnten: gemeinsames Rudelgucken des Wintergarten-Streams. Dazu Pommes mit Jatumba-Soße.

## 27. März 2021

### Planungen für Jakarta EE 11

Was gibt’s denn sonst Neues von Jakarta, auf dem Weg zu EE 11? Aktuell noch nicht viele Festlegungen, bis auf die Eckdaten: Das Release ist für Q1/2024 geplant, und die Mindest-Java-SE-Version wird das dann aktuelle LTE-Release 21 sein! In den nächsten Tagen wird aber das im vorletzten Tagebuch erwähnte Google Doc mit der öffentlichen Ideensammlung quasi geschlossen; dann sollen aus den einzelnen Themen „Issues“ im GitHub-Projekt werden, um dort die Diskussionen zu Ende zu führen und anschließend die Umsetzung zu starten. Was schon vorher deutlich wurde: „CDI Centric“ wird ein großes Thema sein. Alles andere ist dann vermutlich bei Erscheinen dieses Tagebuchs schon in deutlich mehr Details im GitHub-Projekt *[10]* zu sehen.

### Referenzen

[1] *https://www.oracle.com/us/corporate/pricing/price-lists/java-se-subscription-pricelist-5028356.pdf*
[2] *https://docs.google.com/document/d/1tDum7Bm6LtFXQbcUfxw6DTuVkhjuaXmqu1fpaKaTIto/edit*
[3] *https://www.eclipse.org/lists/adoptium-wg/pdfVlgB1oL2Tm.pdf*
[4] *https://docs.google.com/document/d/1FFYi8IcTdyBBXu-gw_ZCH-6mqPvdrL9QPohxPCN_eRJQ*
[5] *https://www.eclipse.org/lists/microprofile-wg/msg01892.html*
[6] *https://dash.adoptium.net*
[7] *https://github.com/eclipse/lsp4jakarta*
[8] *https://openjdk.org/projects/jdk/20/*
[9] *https://openjdk.org/projects/jdk/21/*
[10] *https://github.com/jakartaee*

**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

Schockschwerenot! Fast hätte es in dieser Ausgabe keine Eclipse-Corner gegeben! Schuld daran war eine verloren gegangene E-Mail: Normalerweise sendet mir Chefredakteurin Lisa Damerow meinen Abgabetermin rechtzeitig zu, weil ich zu schludrig bin, mir alle Termine bereits am Jahresbeginn aufzuschreiben. Diesmal ging die E-Mail leider verschütt, und im Stress des Tagesgeschäfts dachte ich schlicht nicht mehr an die Eclipse Corner. Hätte mich die Redaktion über ein Medium kontaktiert, das inhärent Lesebestätigungen verwendet, wie beispielsweise Matrix (der iJUG e. V. betreibt ja sogar einen eigenen Matrix-Server, wie ich selbst übrigens auch), hätte man sehr viel früher bemerkt, dass mir der Termin nicht bekannt ist. Hat man aber eben nicht.

Macht ja nichts, dank geopfertem Wochenende gibt es trotzdem eine Eclipse Corner! In dieser Ausgabe geht es (endlich) auch mal um ein anderes Java-Projekt der Eclipse Foundation als immer nur über Jakarta EE. Dieses Mal also meckere ich über die Eclipse IDE, Namensgeber eben jener Stiftung, über die ich an dieser Stelle regelmäßig meckere. Die Eclipse Foundation wurde ja bekanntlich von IBM ins Leben gerufen, um die Wartung von „Rational XDE" (übrigens mein erster Kontakt zu Eclipse dank einer kostenlosen Demo-CD) nicht allein aus der eigenen Tasche bezahlen zu müssen, sondern Konkurrent Borland mit an den Kosten zu beteiligen, deren Produkt „Together" seit 2006 ja analog zu IBMs XDE auf einer stabilen IDE, nämlich Eclipse, basierte (und nicht mehr auf dem zähen JBuilder alias Delphi-for-Java).

Als das nicht so richtig geklappt hat, hat IBMs Eclipse-Finanzierungswille deutlich nachgelassen, Erich Gamma als ehemaliger Eclipse-Leiter hat IBM verlassen und sich lieber Microsofts Visual Studio Code zugewendet und – gefühlt seit diesem Tag – macht Eclipse bei mir und all meinen Kollegen nur noch Mucken. Darum habe ich mich jetzt nach mehr als 20 Jahren von Eclipse verabschiedet und fast als letzter in meinem Unternehmen VS Code zugewendet. Ja, schon gut, cool bleiben – das ist nur eine sogenannte „alternative Wahrheit", wie sie in Politik und Gesellschaft seit Donald Trumps Präsidentschaft allenthalten salonfähig ist. Nichtsdestotrotz steckt darin ein Fünkchen Wahrheit, und das sieht so aus, dass ich wirklich seit Beginn dieses Jahres (schweren Herzens) auf die Eclipse IDE verzichtet habe, weil sie immer unbrauchbarer (schwergewichtiger, langsamer, buggy) wurde und mich im Alltagsgeschäft aufgehalten und geärgert hat. Ja, stimmt, VS Code hält mich auch auf und ärgert mich auch, weil auch VS Code ständig neue dumme Ideen hat (aktuell: Parameternamen einfach irgendwo mittenrein in den Methodennamen quetschen anstatt zwischen vorausgehender Klammer und nachfolgendem Wert). Doch VS Code startet binnen weniger Sekunden neu, und mit den paar „Quirks" habe ich leben gelernt. Das Eclipse-Team hingegen wollte jahrelang meine Bug Reports nicht bearbeiten, hat keine Lösungen geliefert, kann nach einer halben Dekade immer noch nicht respektive immer noch nicht sauber mit Git-Worktrees und Maven-Multi-Modul-Projekten umgehen und hat auf meine Anfragen bis zuletzt stets nur arrogant geantwortet. Sorry to say: Sad, but true. Hätte man mit mir, als bekanntem und passioniertem Open-Source-Entwickler, etwas mehr ehrlich und vertrauensvoll kommuniziert, wäre Eclipse vielleicht immer noch die beliebteste IDE im Java-Universum. Hat man aber eben nicht.

Auch meinem „Lieblingsprojekt" droht bald das gleiche Schicksal: Scheitern durch mangelnde Kommunikation. Jakarta EE 10 ist ja seit einiger Zeit raus, aber wie ihr am Drama um meinen Artikel über jenes Release schon erahnt habt, lief da nicht alles rund, und die 10 ist nicht wirklich ein großer Wurf. Nun will man in aller Eile (um endlich auf die versprochene Jahres-Kadenz zu kommen) die 11er raushauen, weiß dabei nur ein paar Eckdaten (etwa, dass wir alle gerne auf das kommende Java SE 21 wechseln würden und dass JAX-RS proprietäres DI blöd ist und wir alle lieber CDI wollen), aber sonst eben: nichts. Sicherlich haben die wenigsten von euch kommuniziert bekommen, dass dringend Input für die 11er benötigt wird – denn die Hersteller wissen nicht, was sie noch so alles an Jakarta EE ändern müssten, damit ihr von Spring, Quarkus, Helidon etc. endlich auf Jakarta EE als freien, offenen, hersteller- und produktneutralen Standard wechselt. Um die 11er zu einem Erfolg zu machen, hätte man die richtigen Leute fragen müssen: euch. Zwar steckt die EF und auch die Jakarta EE WG sehr viel Geld in Marketing (was ja auch eine Form der Kommunikation ist) und müht sich mit „public calls" leidlich um Offenheit, Transparenz und Feedback – doch dabei kommt nichts raus. Warum auch immer. Sei es, dass man euch nicht kommuniziert, dass Input benötigt wird oder wo dieser bis wann in welcher Form abzuliefern ist, sei es, dass ihr diesen Schuss einfach nicht gehört habt oder hören wollt und weiterhin glaubt, Jan Westerkamp und ich als iJUG-Botschafter in der Jakarta EE WG würden euch die Wünsche schon an der Nasenspitze ansehen. Jedenfalls ist bislang wenig von euren Wünschen bei der Jakarta EE WG angekommen. So tappen die Hersteller mal wieder im Dunkeln und mauscheln mehr oder minder alleine vor sich hin – das war ja aber nicht das Ziel, als der iJUG in die Jakarta EE WG eingetreten ist. Hätte man die Anforderung, Wünsche aus der Community zu erhalten, rechtzeitig und in der richtigen Form kommuniziert, hättet ihr sicherlich bereits alle eure Wünsche an die EF gemeldet – davon kann ich doch sicherlich ausgehen, oder? Hat man aber eben nicht.

**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.

# Was gibt es Neues in Java 20?

Falk Sippach, embarc Software Consulting GmbH

*Im März 2023 wurde das OpenJDK 20 veröffentlicht. Bereits im September wird mit Version 21 schon das nächste Long-Term-Support-Release (LTS) erwartet. In Java 20 sind daher noch einige Themen in Arbeit, die dann im LTS-Release voraussichtlich finalisiert werden. Darum werfen wir in diesem Artikel zunächst einen Blick auf die aktuellen Neuerungen und wagen am Ende auch einen Blick in die Zukunft.*

Für einen ersten Überblick ist wie immer die Projektseite des OpenJDK [1] ein guter Startpunkt. Dort sind diesmal nur 6 JEPs (JDK Enhancement Proposals) gelistet:

- JEP 429: Scoped Values (Incubator)
- JEP 432: Record Patterns (Second Preview)
- JEP 433: Pattern Matching for switch (Fourth Preview)
- JEP 434: Foreign Function & Memory API (Second Preview)
- JEP 436: Virtual Threads (Second Preview)
- JEP 437: Structured Concurrency (Second Incubator)

Einige der Themen sind schon lange dabei, wie zum Beispiel das Pattern Matching for switch mit der vierten Preview oder auch das Foreign Function & Memory API, das bereits vor einigen Jahren zunächst als Inkubator und mittlerweile als zweite Preview erschienen ist. Die Hoffnung ist groß, dass ein Großteil der JEPs dann mit dem LTS-Release von Java 21 einen Abschluss finden wird. Noch relativ neu dabei sind die Themen rund um die Virtual Threads. Doch an der Entwicklung zu dieser kleinen Revolution der konkurrierenden Programmierung wird im Hintergrund auch schon sehr viele Jahre gearbeitet. Bei den Virtual Threads selbst gab es nur minimale Änderungen. Man möchte mit der zweiten Preview einfach mehr Feedback von den Nutzern sammeln. Wenn das positiv verläuft, könnten sie vielleicht auch schon mit dem LTS-Release im Herbst finalisiert werden. Die anderen Themen in dem Bereich (Scoped Values und Structured Concurrency) befinden sich jedoch erst in der Inkubator-Phase und werden vermutlich noch einige Releases bis zur Fertigstellung benötigen.

## Virtual Threads

Starten wir mit einer der spektakulärsten Neuerungen der letzten Jahre. Ein halbes Jahr zuvor kamen im OpenJDK 19 die virtuellen Threads hinzu. Sie stehen von der Wichtigkeit in einer Reihe mit Generics (Java 5), Lambdas/Stream API (Java 8) sowie dem Plattform-Modul-System (Java 9) und könnten möglicherweise schon im nächsten LTS-Release im September 2023 finalisiert werden. Denn in Java 20 gab es keine nennenswerten Änderungen mehr an den Virtual Threads selbst. Vielmehr wurden sie als JEP 436 zur weiteren Sammlung von Feedback in einer zweiten Preview-Phase wiedervorgelegt. Da dieses Feature aber noch so neu ist, lohnt ein genauerer Blick darauf.

Im Wiki des Project Loom werden die Virtual Threads folgendermaßen angekündigt:

> *JVM features and APIs for supporting easy-to-use, high-throughput, lightweight concurrency and new programming models.*

Bereits seit den 90er Jahren bietet Java Möglichkeiten, Anwendungscode zu parallelisieren, also zum Beispiel Berechnungen auf mehrere, gleichzeitig laufende Threads zu verteilen. Die sind allerdings oft ein Flaschenhals. Denn bei Webanwendungen wird beispielsweise für jeden eingehenden Request ein Thread benötigt. Bisher entsprach ein Java-Thread immer einem Betriebssystem-Thread. Diese sind sehr ressourcenhungrig und damit in der Anzahl stark begrenzt. Versucht man nur ein paar Tausend Threads gleichzeitig zu starten, endet das auf normaler Hardware schnell in einem OutOfMemory-Fehler und die Anwendung antwortet nicht mehr wie gewünscht. Arbeiten sehr viele Nutzer gleichzeitig mit einem Websystem, kommt es bereits bei einer noch überschaubaren Anzahl von Usern an die Grenzen. Insbesondere wenn die Bearbeitung einzelner Threads länger dauert, da auf blockierende Aufrufe oder externe Dienste wie Datenbanken gewartet werden muss.

Reaktive Programmierung kann bei diesen Problemen nützlich sein, bringt aber neue Herausforderungen bezüglich Les-, Wart- und Debugbarkeit mit sich. Außerdem müssen alle Glieder in der Kette für die reaktive Programmierung vorbereitet sein, zum Beispiel benötigt es für die Datenbankzugriffe die entsprechenden Treiber.

Virtuelle Threads erlauben es nun, die konkurrierende Verarbeitung von parallel ausgeführten Aufgaben auch bei einer sehr großen Anzahl von Threads zu implementieren und dabei sogar leicht les- und gut wartbaren Code zu schreiben. Dieser lässt sich zudem wie jeder sequenzielle Quellcode mit herkömmlichen Mitteln debuggen. Die virtuellen verhalten sich dabei wie ganz normale Threads, werden jedoch nicht 1:1 auf das Betriebssystem abgebildet. Stattdessen gibt es einen Pool von Träger-Threads (Carrier Threads), denen dann virtuelle Threads vorübergehend zugewiesen werden. Sobald der virtuelle Thread auf eine blockierende Operation stößt, wird er vom Träger-Thread genommen, der dann einen anderen virtuellen Thread (einen neuen oder einen zuvor blockierten) übernehmen kann.

Die virtuellen Threads wurden transparent in die bestehende Klassenhierarchie des JDK integriert. Die neuen Factory-Methoden erstellen entweder eine Instanz der Klasse *java.lang.Thread* (Plattform-Threads) beziehungsweise von *java.lang.VirtualThread* (siehe Listing 1). Diese Factory-Methoden erzeugen jeweils einen ThreadBuilder, der sich noch parametrisieren lässt. Durch diese Abstraktion kann sehr einfach auf die virtuelle Variante gewechselt werden. Bestehende Anwendungen können allein durch den Wechsel auf Java 20 oder höher profitieren, sofern die verwendeten Frameworks (zum Beispiel Spring) bereits virtuelle Threads unterstützen. Solange sich die Virtual Threads aber noch im Preview-Modus befinden, sollte man mit dem Einsatz in produktiver Software vorsichtig sein und muss für erste Prototypen sowohl beim Kompilieren als auch beim Starten der JVM das Flag `--enable-preview` setzen.

```
Thread.ofPlatform().daemon().start(() -> {
  try {
      Thread.sleep(Duration.ofSeconds(1));
  } catch (InterruptedException e) {
      throw new RuntimeException(e);
  }
  System.out.println("Plattform Thread");
});

Thread.ofVirtual().start(() -> {
  try {
      Thread.sleep(Duration.ofSeconds(1));
  } catch (InterruptedException e) {
      throw new RuntimeException(e);
  }
  System.out.println("Virtueller Thread");
});
```

*Listing 1*

Blockierende Operationen halten den ausführenden Thread nun also nicht mehr unnötig auf. So kann mit nur einem kleinen Pool von

Carrier Threads eine viel höhere Anzahl von Requests parallel verarbeitet werden.

Herausfinden, ob der eigene Code in einem virtuellen Thread läuft, kann man mit dem Aufruf von `Thread.currentThread().isVirtual()`.

## Structured Concurrency

Virtuelle Threads sind die Voraussetzung für ein weiteres, spannendes Feature, das ebenfalls bereits im OpenJDK 19 Einzug gehalten hatte und nun zum zweiten Mal mit von der Partie ist: Structured Concurrency. Auch da gibt es mit dem JEP 437 eine Wiedervorlage ohne größere Änderungen. Allerdings handelt es sich hierbei um eine den Previews vorausgehende Inkubator-Version. Da müssen wir also noch etwas warten, bis dieses Feature finalisiert wird.

Bei der Bearbeitung von mehreren parallelen Teilaufgaben erlaubt Structured Concurrency die Implementierung auf eine besonders les- und wartbare Art und Weise. Bisher wurde für parallel zu verarbeitende Teilaufgaben (zum Beispiel Zugriff aufs Filesystem, Lesen aus der Datenbank und Aufrufe von Fremdsystemen) typischerweise ein *ExecutorService* eingesetzt. In *Listing 2* sehen wir ein Beispiel mit drei Tasks.

```
private static final ExecutorService executor = Executors.newCachedThreadPool();
Future<String> task1 = executor.submit(() -> { … }
Future<String> task2 = executor.submit(() -> { … }
Future<String> task3 = executor.submit(() -> { … }

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

*Listing 2*

Die Anwendung blockiert, bis die Ergebnisse aller drei Tasks fertig sind und gibt dann die Ergebnisse aus. Doch was passiert, wenn bei einem der Tasks ein Fehler auftritt? Wie kann man dann die anderen Tasks abbrechen? Wie kann man allgemein die Tasks abbrechen, wenn die gesamte Berechnung nicht mehr benötigt wird? Sollte Task 1 sehr lange laufen, erfährt man erst sehr spät, ob Task 2 oder 3 Fehler produziert haben. Natürlich lassen sich für diese Probleme Lösungen entwickeln. Die sind allerdings komplex und häufig schlecht lesbar. Auch das Debuggen wäre nicht einfach, da in den Thread-Dumps die Tasks nicht den jeweiligen Threads aus dem Pool zugeordnet werden können.

Bei der Structured Concurrency ersetzen wir den ExecutorService durch einen StucturedTaskScope *(siehe Listing 3)*, bei dem man verschiedene Strategien (hier ShutdownOnFailure) auswählen kann. Durch dieses API wird der Code nicht nur lesbarer, es lässt sich auch leichter mit potenziellen Fehlersituationen umgehen.

Mit `scope.join()` wird gewartet, bis alle Tasks erfolgreich erledigt sind. Schlägt einer fehl oder wird abgebrochen, werden auch die anderen beiden beendet. Mit `throwIfFailed()` kann der Fehler außerdem weitergegeben werden und das Ausgeben der Ergebnisse wird übersprungen.

Dieser neue Ansatz bringt einige Vorteile. Zum einen bilden Task und Sub-Tasks im Code eine abgeschlossene, zusammengehörige Einheit. Die Threads kommen zudem nicht aus einem Thread-Pool mit schwergewichtigen Plattform-Threads. Stattdessen wird jede Unteraufgabe in einem neuen virtuellen Thread ausgeführt. Bei Fehlern werden außerdem noch laufende Sub-Tasks abgebrochen. Des Weiteren sind bei Fehlersituationen die Informationen besser, weil die Aufrufhierarchie sowohl in der Code-Struktur als auch im Stacktrace der Exception sichtbar ist.

Aufgrund des Inkubator-Status benötigt die Structured Concurrency beim Kompilieren und Ausführen ebenfalls einige zusätzliche Informationen *(siehe Listing 4)*. Nicht vergessen, es könnte in den nächsten Java-Releases noch einige, auch sehr grundlegende Änderungen geben.

```
$ javac --enable-preview -source 19 --add-modules jdk.incubator.concurrent StructuredConcurrency.java
$ java --enable-preview --add-modules jdk.incubator.concurrent StructuredConcurrency
```

*Listing 4*

## Scoped Values

Erstmals im OpenJDK 20 dabei sind die Scoped Values (JEP 429). Sie sind eine moderne Alternative zu Thread-Local-Variablen und können insbesondere mit Virtual Threads kombiniert werden. Sie erlauben das Speichern eines temporären Wertes für eine begrenzte Zeit, wobei nur der Thread, der den Wert geschrieben hat, ihn auch wieder lesen kann.

Scoped Values werden in der Regel als öffentliche statische Felder angelegt, sodass sie von beliebigen, auch tiefer im Aufruf-Stack

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
  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 3*

befindlichen Methoden erreichbar sind, ohne dass wir sie als Methoden-Parameter durchschleifen müssen. Verwenden mehrere Threads dasselbe ScopedValue-Feld, kann/wird dieses je nach ausgeführtem Thread unterschiedliche Werte enthalten. Wenn man mit Thread-Locals vertraut ist, dann sollte dieses Konzept bekannt sein.

Betrachten wir ein einfaches Beispiel *(siehe Listing 5)*. Aus einem Web-Request extrahieren wir Informationen zum angemeldeten Benutzer. Auf diese Informationen müssen wir in der weiteren Aufrufkette (im Service, im Repository …) zugreifen. Eine Option (hier nicht abgebildet) wäre wie gesagt das „Durchschleifen" des User-Objekts als zusätzlichen Parameter in den aufzurufenden Methoden. Durch Scoped Values kann diese redundante und unübersichtliche zusätzliche Parameterauflistung aber vermieden werden. Das User-Objekt wird in eine statische Variable gespeichert und ist somit im weiteren Verlauf jederzeit abrufbar. Konkret wird mit `ScopedValue.where()` das User-Objekt gesetzt und dann der `run`-Methode ein Runnable übergeben, für dessen Aufrufdauer der Scoped Value gültig ist. Alternativ kann mit der `call`-Methode eine Instanz von Callable übergeben werden, um auch Rückgabewerte auszuwerten. Der Versuch, den Service außerhalb des Scoped-Value-Kontextes auszuführen (nächste Zeile), führt zu einer Exception, weil bei diesem Aufruf dann kein User-Objekt hinterlegt ist.

Das Auslesen des Scoped Value erfolgt über den Aufruf von `get()`. Bei Abwesenheit eines Wertes kann man mit einem Fallback (`orElse()`) oder dem Werfen einer Exception (*`orElseThrow()`*) reagieren.

Auch hier müssen beim Kompilieren und Ausführen gewisse Schalter aktiviert werden *(analog zu Listing 4)*.

Die Klasse ScopedValue ist immutable und bietet dementsprechend keine `set`-Methode an. Dadurch wird der Code besser les- und vorhersagbar, weil es keine Zustandsänderungen an bestehenden Objekten geben kann. Muss für einen bestimmten Code-Abschnitt (Aufruf einer weiteren Methode in der Kette) ein anderer Wert sichtbar sein, kann ein Rebinding des Wertes ausgeführt werden (beispielsweise auf null setzen wie in *Listing 6*). Sobald der begrenzte Code-Abschnitt beendet ist, wird der ursprüngliche Wert wieder sichtbar.

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

*Listing 6*

```
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 5*

Scoped Values arbeiten auch mit der Structured Concurrency zusammen. Die Sichtbarkeit wird an die über einen StructuredTaskScope erzeugten Kind-Prozesse vererbt. Somit können alle per `fork()` abgezweigten Kind-Threads ebenfalls auf die im Scoped Value befindlichen User-Informationen zugreifen.

Die Scoped Values stehen genau wie die Thread-Locals sowohl für Plattform- als auch für virtuelle Threads zur Verfügung. Die Vorteile sind:

- Automatisches Aufräumen der Inhalte, sobald der Runnable-/Callable-Prozess beendet ist, somit entstehen keine Memory Leaks.
- Immutablility der Scoped Values erhöht die Verständlichkeit beim Lesen.
- Bei der Structured Concurrency erzeugte Kind-Prozesse haben auch Zugriff auf den einen, unveränderbaren Wert. Die Informationen werden nicht kopiert (wie bei InheritedThreadLocals), wodurch der Speicherverbrauch niedriger ist.

## Pattern Matching

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 Preview hinzu. Dies ist jetzt bereits zum vierten Mal mit von der Partie. Seit OpenJDK 19 gibt es zudem die sogenannten Deconstruction (Record) Patterns. In Zukunft werden vermutlich weitere Mustertypen folgen, wie zum Beispiel für Arrays, Maps, POJOs oder Factory-Methoden sowie das Platzhalter-Muster für zu ignorierende Ausdrücke.

Beim Pattern Matching geht es darum, bestehende Strukturen mit Mustern abzugleichen, um komplizierte Fallunterscheidungen effizient und wartbar behandeln 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 wir es in der objektorientierten Programmierung normalerweise tun würden. Brian Goetz beschreibt in seinem Artikel zu datenorientierter Programmierung *[3]* die Vorteile und Hintergründe. In Java funktioniert das über den um neue Funktionalität erweiterten *switch*.

Für die Abrechnung von Telefongesprächen könnte die Unterscheidung der verschiedenen Tarife wie in *Listing 7* erfolgen. Das Beispiel ist fiktiv und teilweise etwas künstlich aufgebläht, um verschiedene Optionen zeigen zu können. Gegenüber dem klassischen Switch-Statement mit konstanten Patterns (primitive Ganzzahlen, Strings, Enums) lässt sich in den `case`-Zweigen nun auch auf beliebige Typ-Deklarationen (Type Patterns) oder Daten-Tuple (Records) prüfen, deren Inhalte dann für die weitere Bearbeitung direkt extrahiert und in Variablen gespeichert werden. Wenn der „tarif" vom Typ `PrivatTarif` oder `ProfiTarif` ist, kann so jeweils der enthaltene Wert („rabatt" oder „preis") direkt für weitere Berechnungen verwendet werden.

Die moderne Syntax dieser Switch-Expression ist besser lesbar und weniger fehleranfällig als das altbekannte Statement (zum Beispiel kein Fall-Through mehr, klar definierte Scopes von Variablen usw.) Außerdem kann jetzt explizit auf `null`-Werte geprüft werden. Ein `default`-Zweig für die nicht abgedeckten Fälle ist weiterhin erlaubt. Der Compiler fordert den `default`-Teil jetzt sogar explizit ein, wenn nicht alle möglichen Fallunterscheidungen behandelt werden. Für diese sogenannte Exhaustiveness-Prüfung benötigt es algebraische Datentypen. Neben den schon lange bekannten Enums stehen mit Sealed Classes (Summentypen) und Records (Produkttypen) hier nun weitere Optionen zur Verfügung. Die daraus resultierende Typsicherheit zur Compile-Zeit ist ein wichtiger Baustein zur Vermeidung von Programmierfehlern aufgrund von redundantem Quellcode.

Im Laufe der verschiedenen Preview-Phasen hat man immer wieder an der neuen Syntax gefeilt. Im OpenJDK 19 wurde beispielsweise von den Guarded Patterns (mit „&&") auf die When-Clauses umgestellt, wie sie auch in *Listing 7* verwendet werden. Dadurch können Type Patterns bei einem Treffer sehr kompakt um boolesche Ausdrücke für die weitere detaillierte Fallunterscheidung ergänzt werden. *Listing 8* zeigt dafür einen konkreten Anwendungsfall (große und kleine Dreiecke).

Hier wird eine Sealed-Classes-Hierarchie verwendet, das heißt, es gibt nur die beiden Subklassen „Rechteck" und „Dreieck". Würde in einem `case`-Zweig noch das Rechteck behandelt, dürfte man den `default`-Zweig sogar weglassen. Der Compiler kann durch die Exhaustiveness-Prüfung sicherstellen, dass alle möglichen Subklas-

```
Zeitpunkt zeitpunkt = [..]
int minuten = [..]

gesamtGebuer += switch (tarif) {
     case PrivatTarif(double rabatt) -> berechnePrivatAnruf(rabatt, zeitpunkt, minuten);
     case BusinessTarif bt when bt.isVipKunde() -> berechneVIPAnruf(zeitpunkt, minuten);
     case BusinessTarif bt -> berechneBusinessAnruf(bt, zeitpunkt, minuten);
     case ProfiTarif(int preis) -> minuten * pt.preis();
     case null -> 60;
};
```

*Listing 7*

```
sealed class Shape permits Rectangle, Triangle {
}

final class Rectangle extends Shape {
}

final class Triangle extends Shape {
  int calculateArea() {
     return 1000;
  }
}

Shape s = new Triangle();

switch (s) {
  case Triangle t when t.calculateArea() > 100 -> System.out.println("Large triangle");
  case Triangle t -> System.out.println("Small triangle");
  default -> System.out.println("Not a triangle");
}
```

Listing 8

sen von Shape innerhalb der Switch-Anweisung behandelt werden. Das bedeutet für die Zukunft: Fügt man eine neue Subklasse (zum Beispiel Circle) hinzu, wird der Compiler sofort bei allen Switch-Anweisungen fehlschlagen, die den neuen Typ noch nicht auswerten.

Zum letzten Preview hat sich beim Pattern Matching for switch nicht mehr viel getan und für das OpenJDK 21 ist mit JEP 441 die Finalisierung angekündigt [4]. Bis dahin muss diese Funktion allerdings sowohl beim Kompilieren als auch Starten mit `--enable-preview` explizit eingeschaltet werden.

In der aktuellen Version gab es nur zwei kleinere Änderungen. Das eine betrifft die Art der Exception, die in dem theoretisch möglichen Fall geworfen wird, wenn die Vollständigkeitsprüfung erst zur Laufzeit fehlschlägt. Das könnte passieren, wenn eine Sealed-Class-Hierarchie erweitert wird, aber nur diese neuen Klassen kompiliert werden. Bereits bestehender Bytecode mit Switches, der dann eigentlich nicht mehr kompilieren dürfte, muss nicht unbedingt neu erstellt werden (etwa beim Einsatz von Bibliotheken/Frameworks). In diesem Fall würde die Typprüfung dann erst zur Laufzeit mit einer `MatchException` (statt dem zuvor verwendeten und hier irreführenden `IncompatibleClassChangeError`) fehlschlagen.

Die zweite Änderung betrifft die Typ-Inferenz (automatische Erkennung von Typen) bei Records mit generischen Komponenten. Bisher musste man den Typ in spitzen Klammern mitschreiben, jetzt kann der Compiler auf den entsprechenden Typ aus dem Kontext schlussfolgern und wir können diese Redundanz vermeiden.

## Record Patterns

In Listing 7 haben wir bereits die Verwendung von Records Patterns gesehen. Letztlich ist es auch nur eine Prüfung auf den Typ, wobei zusätzlich noch die Inhalte (Felder des Records) direkt extrahiert werden. Diese Variablen sind dann in dem nachfolgenden Block (`case`-Zweig, `if`-Block …) gültig und können beispielsweise in den When-Clauses oder für weitere Berechnung direkt verarbeitet werden. Das wird auch Deconstruction Pattern genannt und so auch in der Definition zu Pattern Matching in der Scala-Dokumentation explizit erwähnt [5]:

*Pattern matching is a mechanism for checking a value against a pattern. A successful match can also deconstruct a value into its constituent parts. It is a more powerful version of the switch statement in Java, and it can likewise be used in place of a series of if/else statements.*

Die erste Fassung für Record Patterns sollte ursprünglich schon für das JDK 17 eingeplant werden, damals noch in der Kombination mit Array Patterns. Da gab es jedoch zunächst noch Ungereimtheiten und so hat man diese beiden Pattern-Typen getrennt. Array Patterns sowie Ideen für andere Pattern-Typen stehen noch auf der To-do-Liste der Amber-Entwickler. Da werden allerdings vermutlich erst nach dem nächsten LTS-Release weitere JEPs folgen.

## Was sind eigentlich Records?

Records sind, vereinfacht gesagt, eine spezielle Art der Klassen-Definition, wobei uns Entwicklern das Schreiben von unnötigem Quellcode-Ballast (Boilerplate) durch den Compiler abgenommen wird. Dadurch lassen sich sehr schlank benannte Tuple-Typen erstellen und einfach per kanonischen Konstruktor instanziieren. Beim Record Pattern gehen wir nun den umgekehrten Weg und dekonstruieren/zerlegen eine Datenstruktur in ihre Einzelteile. Das könnte man auch sehr einfach mit dem Type Pattern und instanceof erreichen. Allerdings müssen die Entwickler dort zusätzlich durch explizite Aufrufe der Accessor-Methoden die relevanten Informationen nach dem Matchen des Musters manuell auslesen. Das Record Pattern stellt hingegen eine deklarative Beschreibung des Musters dar, wobei die Extraktion unter der Haube passiert und direkt mit den relevanten, bereits zugewiesenen Informationen weitergearbeitet werden kann. Das spart Codezeilen und macht den Sourcecode konsistenter, besser les- und damit auch wartbarer.

In Listing 9 werden die zwei Datenklassen Point und Rectangle definiert. Beim Prüfen auf den Typ werden die Properties des Records den Variablen zugewiesen (ähnlich einem Konstruktor-Aufruf).

Das funktioniert auch mit geschachtelten (nested) Records. Zum Beispiel suchen wir deklarativ nach dem Muster eines Rechtecks, das zwei Punkte enthält, um mit den Koordinaten weitere Berechnungen durchzuführen. Hierbei lassen sich verschiedene Granularitätsstufen mischen. Wenn man sich also nur für die Koordinaten des ersten Punktes interessiert, kann der zweite quasi „anonym" bleiben. In Zukunft wird es da irgendwann auch ein Platzhalter-Pattern

```
record Point(int x, int y) {
}

record Rectangle(Point p1, Point p2) {
}

Object somePoint = new Point(0, 0);
Object someRectangle = new Rectangle(new Point(0, 1), new Point(10, 6));

if (somePoint instanceof Point(int x, int y)){
  System.out.println(x + y);
}

if (someRectangle instanceof Rectangle(Point(int x1, int y1), Point p2)) {
  int area = Math.abs(p2.x() - x1) * Math.abs(p2.y() - y1);
  System.out.println(String.format("Koordinaten des P1 vom Rechteck: (%s; %s)", x1, y1));
}
```

*Listing 9*

(beispielsweise mit dem Unterstrich „_") für nicht benötigte Referenzen geben, so wie das Scala oder andere funktionale Sprachen bereits anbieten. Man kann dieses Verhalten aber bereits jetzt mit einer Variablen mit dem Namen „__" (zwei Unterstriche) simulieren und diese Variable dann einfach ignorieren.

In Zukunft sind weitere Pattern-Typen denkbar. Die schon angesprochenen Array Patterns ermöglichen das Zerlegen von listenartigen Strukturen. Andere Ideen sind die sogenannten Map- und POJO-Patterns oder als Alternative zum starren Konstruktor die Definition von Patterns über Factory-Methoden. Dadurch können sich Entwickler selbst Dekonstruktions-Patterns definieren und sind nicht auf die Signatur des Konstruktors beschränkt. Pattern Matching wird uns sicher noch einige Releases und Jahre begleiten.

Auch in Java 20 gab es bei den Record Patterns nochmal Änderungen. Zum einen kann der Compiler ähnlich wie bereits beim Pattern Matching for switch nun generische Typen selbst herleiten, sodass die redundante Angabe in den Mustern entfallen kann. Das betrifft zum Beispiel Fälle, in denen ein Interface mit einem generischen Typ von einem Record implementiert wird.

Leider weggefallen ist die Unterstützung für benannte Record Patterns. Denn bisher konnte man Records auf drei Arten angeben *(siehe Listing 10)*.

```
if (somePoint instanceof Point p){
  System.out.println(p.x() + p.y());
}
if (somePoint instanceof Point(int x, int y)){
  System.out.println(x + y);
}
if (somePoint instanceof Point(int x, int y) p){
  System.out.println(x + p.y());
}
```

*Listing 10*

Bei der dritten Variante hatte man dann die Wahl, ob man direkt die extrahierten Werte verwendet oder über den Accessor des Records zugreift. Diese Variante hat man für überflüssig befunden und daher nun entfernt. Auch wenn es die Verwendung der Record Patterns etwas klarer macht, nimmt es in bestimmten Szenarien die Flexibilität. In *Listing 11* soll neben der Flächenberechnung (Methode auf dem Rechteck) im gleichen Block auch auf die Koordinaten der Eckpunkte zugegriffen werden. Das ist dann nicht mehr so einfach und pragmatisch möglich.

```
// Variable "r" ist nicht mehr erlaubt
if (someRectangle instanceof Rectangle(Point(int x1,
int y1), Point(var x2, var y2)) r) {
  System.out.println(String.format("Fläche des Rech-
tecks (%s): %s", r.area()));
}
```

*Listing 11*

Die letzte Neuerung zeigt einen interessanten Einsatzzweck, die Verwendung von Record Patterns in for-Schleifen. Statt mit der Referenz auf dem jeweils nächsten Record-Objekt zu arbeiten, kann der Record im Kopf der Schleife direkt dekonstruiert werden. Das macht den Code wieder ein Stück prägnanter und lesbarer, wie in *Listing 12* zu sehen.

```
if (Point(int x, int y): coordinates) {
  System.out.printf("(%d; %d)%n", x, y);
}
```

*Listing 12*

Diese Möglichkeit wird im OpenJDK 21 allerdings schon wieder entfernt werden, soll aber dann später in einem separaten JEP erneut auftauchen. Dafür werden die Record Patterns laut der Ankündigung im JEP 440 *[6]* im Herbst 2023 finalisiert.

## Was sonst noch passiert ist?

Obwohl das Vector-API seit Java 16 ein regelmäßiger Begleiter der halbjährlichen Releases ist, ist es diesmal nicht als eigenständiger JEP aufgeführt. Trotzdem gab es in diesem vierten Inkubator ein paar kleinere Bugfixes. Es geht dabei übrigens um die Unterstützung der modernen Möglichkeiten der SIMD-Rechnerarchitektur

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. Bleibt abzuwarten, wann es dort weitergeht beziehungsweise wann diese Thematik finalisiert wird.

Auch schon seit einigen Versionen dabei und diesmal wieder im Recall 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 jedoch relativ aufwendig 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 des 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-JEP gemeinsam enthalten. Mit dem JEP 434 wurden nochmal einige Änderungen am API gemacht. Vermutlich wird es sich aber langsam der Finalisierung nähern.

## Weitere Änderungen

In der Klassenbibliothek des JDK werden auch regelmäßig Änderungen vorgenommen, die allerdings nicht in JEPs auftauchen. Da lohnt ein Blick auf die Differenz zu früheren Java-Versionen durch den Java Almanac *[7]*.

Als veraltet markiert wurden die Konstruktoren der Klasse java.net.URL. Stattdessen soll zukünftig der URI-Builder verwendet und dann daraus eine URL erzeugt werden *(siehe Listing 13)*.

```
URL url = URI.create("https://www.embarc.de").toURL();
```

*Listing 13*

Einen Schritt weiter ist man bei zwei Methoden der Klasse Thread gegangen. Die Methoden `suspend` und `resume` wurden bereits in Java 1.2 deprecated, da sie anfällig für Deadlocks sind. In Java 14 wurden sie dann als „Deprecated for removal" markiert, konnten aber bis jetzt immer noch verwendet werden. Ab dem OpenJDK 20 werfen beide Methoden nun eine UnsupportedOperationException.
Die weitere Liste der vielen kleinen Änderungen ist lang. So wurde zum Beispiel die Default Keep Alive Time des http-Client auf 30 Sekunden verringert. Und die Unterstützung für Unicode 15.0 hinzugefügt. Es gab zudem kleinere Optimierungen an Garbage-Collectoren und Verbesserungen beim Monitoring/Logging von JVM-Laufzeitinformationen. Wer zu diesen und den vielen weiteren Themen mehr wissen möchte, sei auf die Release Notes *[8]* verwiesen.

## Fazit

Mit den Scoped Values hat es ein wichtiges Feature in das OpenJDK 20 geschafft, das die in Java 19 begonnene Revolution im Concurrency-Bereich fortsetzt. Sonst hat sich scheinbar wenig getan. Einige neue Funktionen wurden unverändert als weitere Preview mit dem Ziel zur Verfügung gestellt, weitere Erfahrungen und Feedback einsammeln zu können. Bei anderen gab es interessante, aber moderate Verbesserungen. So dürfen wir gespannt sein, ob mit dem nächsten LTS-Release (OpenJDK 21) im Herbst 2023 wirklich einige der jetzt noch offenen Baustellen in den finalen Stand übergehen werden.
Bereits angekündigt wurde, dass die noch offenen Themen zum Pattern Matching jetzt endgültig finalisiert werden. Dabei soll es nochmal drei Änderungen geben:

- Geklammerte Patternausdrücke werden entfernt.
- In `case`-Labels sollen neben Enum-Typen (als Type-Patterns) nun auch direkt Enum-Konstanten als konkrete Ausprägungen des Enum-Typs zulässig sein.
- Die Unterstützung von Record Patterns im Kopf von for-Schleifen soll zunächst entfallen (aber später in einem separatem JEP fortgeführt werden).

Außerdem warten viele andere interessante Themen aus den verschiedenen Inkubator-Projekten darauf, in den nächsten Releases in Erscheinung zu treten. So gibt es neben den Ideen für weitere Pattern-Matching-Typen (Unnamed, Array …) im Projekt Amber auch die Bestrebungen, Java-Code weiter zu entschlacken und lesbarer zu machen. Das Ziel von Projekt Amber ist die Umsetzung kleiner, die Produktivität steigernde Funktionen. Aktuell in der Diskussion sind eine kompaktere Definition von ausführbaren Java-Klassen *(siehe Listing 14)* und das lang ersehnte Thema der String-Interpolation. Denn das kennen mittlerweile viele moderne Programmiersprachen. Bereits in Java 12 sollten die sogenannten Raw String Literals eingeführt werden, die es dann jedoch aufgrund vieler Unklarheiten nicht geschafft haben. Sie wurden dann durch die Text-Blöcke ersetzt, die aber nur die mehrzeilige Definition von Zeichenketten ermöglichen. In Zukunft sollen Zeichenketten in Java auch um Template-Mechanismen ergänzt werden *(siehe Listing 15)*.

```
void main() {
  println("Hello World")
}

// statt

public class HelloWorld {
  public static void main(String[] args) {
     System.out.println("Hello World")
  }
}
```

*Listing 14*

```
String query =
  STR."""select * from User u
  where u.\{property} ='\{value}'""";
```

*Listing 15*

Außerdem erwarten viele schon sehnsüchtig 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 allerdings 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 sowohl Value Types (haben keine Identität) als auch Primitive Classes (haben keine Identität und keine Referenz) und Universal Generics (*List<int>*) ins JDK überführt werden und Javas Typsystem viel flexibler machen.

In den letzten Jahren standen zum Veröffentlichungszeitpunkt des aktuellen Java-Releases bereits die ersten JEPs für die zukünftige Version fest. Dadurch konnte man an dieser Stelle schon einen detaillierteren Ausblick auf das zukünftige Release geben. Da scheint der Enthusiasmus etwas nachgelassen zu haben und man muss jetzt schon genau die Mailinglisten studieren, um relativ frühzeitig die Tendenzen mitzubekommen. Oder man lehnt sich einfach zurück, verprobt zunächst die Änderungen der aktuellen Version und freut sich darauf, was dann in einem halben Jahr kommen wird. Denn das klappt wirklich gut, Oracle schafft es jetzt schon seit über fünf Jahren, die Veröffentlichungstermine auf den Tag pünktlich einzuhalten. Genau genommen beginnt immer bereits drei Monate vor dem Release-Datum die Rampdown Phase One und damit der Fork vom Main-Branch. Ab da stehen dann die Features für die nächste Version fest und es werden nur noch Fehler korrigiert. Das wird schon Anfang Juni für Java 21 *[9]* der Fall sein.

Es tut sich also einiges in der Java-Welt. Wir dürfen gespannt sein und können weiterhin optimistisch in die Zukunft blicken. Denn das Java-Ökosystem ist lebendiger denn je und weiterhin sehr innovativ. Konkurrenz wie beispielsweise Python (hauptsächlich wegen Machine/Deep Learning), JavaScript, Go und die C-basierten Sprachen beleben das Geschäft. Java, das besonders im Unternehmensanwendungsumfeld vertreten ist, wird aber weiterhin ein gehöriges Wort mitreden.

## Referenzen:


[1] *https://openjdk.java.net/projects/jdk/20/*
[2] *https://openjdk.java.net/projects/amber/*
[3] *https://www.infoq.com/articles/data-oriented-programming-java/*
[4] *https://openjdk.org/jeps/441*
[5] *https://docs.scala-lang.org/tour/pattern-matching.html*
[6] *https://openjdk.org/jeps/440*
[7] *https://javaalmanac.io/jdk/20/apidiff/19/*
[8] *https://jdk.java.net/20/release-notes*
[9] *https://openjdk.org/projects/jdk/21/*

**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.

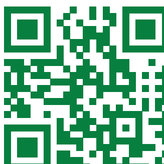
## JETZT TICKET SICHERN

**#JSD2023***

*Weinberge, Innovation und Vernetzung aller, deren Herz für die IT und Softwareentwicklung schlägt - Auf Sachsens größter IT-Community-Konferenz am 29. September 2023 in Radebeul bei Dresden.

JUG SAXONY **DAY**

**www.jugsaxony.day**



# Vom privaten Notizzettel zum Open-Data-Projekt javaalmanac.io

Marc Hoffmann, mtrail GmbH

*Wie auch in diesem Magazin häufig thematisiert, entwickelt sich das OpenJDK-Projekt kontinuierlich weiter. Zwei Releases im Jahr bringen jeweils viele Neuerungen bei den APIs, in der Sprache und auch bei den Tools. Um den Überblick zu behalten, hat Marc Hoffmann aus verschiedenen Quellen systematisch Informationen zu den Java-Releases zusammengetragen und öffentlich zugänglich gemacht. Was mit einem einfachen GitHub Repository begonnen hat, ist über die Jahre unter javaalmanac.io [1] zu einem umfangreichen Open-Data-Projekt herangewachsen.*

Ob als einzelner Java-Entwickler für ein kleines Projekt oder als Systemarchitekt für ein umfangreiches Vorhaben – es stellt sich fortlaufend die Frage, welche die passende Java-Version für ein neues Projekt ist und wo es Aktualisierungsbedarf bei bestehenden Projekten gibt. Für eine erste Auslegeordnung braucht man einen Überblick und sollte folgende Fragen beantworten können:

- Welche Java-Versionen gibt es überhaupt?
- Welche Versionen werden aktuell noch unterstützt?
- Was sind die jeweils letzten Patch-Versionen?
- Welche sind die zugehörigen Bytecode-Versionen?
- Was sind neue APIs und Features und welchen Nutzen bieten sie mir?
- Welche APIs sind deprecated oder werden gar bald entfernt?
- Woher kann ich OpenJDK-Distributionen zu welchen Bedingungen beziehen?

Um Antworten auf diese Fragen an zentraler Stelle zu erhalten, ist das Projekt Java Version Almanac *[1]* entstanden. Ziel war es einerseits, die Geschichte von Java nachzuzeichnen, sowie andererseits, alle Informationen über die aktuellen OpenJDK-Versionen zentral bereit zu halten und auch einen Ausblick auf alle aktuellen Entwicklungen (Early Access) zu geben. In diesem Artikel stellen wir das Projekt vor und zeigen auf, wie wir uns damit als Java-Praktiker jederzeit die oben genannten Fragen beantworten können.

## Java-Releases für den produktiven Einsatz

Für den produktiven Einsatz müssen wir – zumindest bei einem eher konservativen Vorgehen – sicherstellen, dass die verwendete OpenJDK-Version ausreichend stabil ist und längerfristig gewartet wird. Das OpenJDK-Projekt veröffentlicht zweimal im Jahr eine Hauptversion, die sogenannten Feature-Releases. Grundsätzlich gelten diese jeweils im März und September verfügbaren Releases als stabil und getestet und können für produktive Anwendungen verwendet werden. Allerdings werden nur ausgewählte Hauptversionen für einen längeren Zeitraum gewartet und regelmäßig zugehörige Patch-Versionen veröffentlicht. Die aktuellen Long-Term-Support(LTS)-Versionen sind Java 8, 11 und 17. Alle anderen Releases werden nicht mehr im OpenJDK-Projekt gewartet. Updates hierfür sind von verschiedenen Anbietern nur über kommerzielle Supportverträge möglich.

In der Praxis empfiehlt es sich, einen möglichst aktuellen LTS-Release einzusetzen, um von neuen APIs und Sprachmitteln sowie Performance-Verbesserungen zu profitieren. Allerdings gibt es je nach eingesetzten Frameworks oder Bibliotheken Einschränkungen, die zu beachten sind. Zum Beispiel gilt für Spring Boot 3.0 Java 17 als Mindestanforderung *[2]*. Umgekehrt kommen gerade ältere Bibliotheken nicht immer mit den neuesten JDKs zurecht. Das liegt primär daran, dass seit Java 9 mit der Einführung von JPMS (Java Platform Module System) der Zugriff auf interne JDK-Klassen sowie interne Felder und Methoden strikter geregelt ist und auf diese ohne zusätzliche Maßnahmen nicht mehr zugegriffen werden kann. In jedem Fall empfiehlt es sich, einen Blick auf die Release-Notes der eingesetzten Bibliotheken und Werkzeuge zu werfen und das Projekt für neue Java-Versionen zunächst zu testen.

Als erste Orientierung bietet der Java Version Almanac gleich auf der Startseite *(siehe Abbildung 1)* einen Statusüberblick für alle JDK-Releases. Die farblich angezeigten Status sind:

| Details | Status | Documentation | Download | Compare API to |
|---|---|---|---|---|
| Java 21 | DEV | API\| Notes | JDK\| JRE | 20 19 18 17 16 15 14 13 12 11 ... |
| Java 20 | REL | API\| Notes | JDK\| JRE | 19 18 17 16 15 14 13 12 11 10 ... |
| Java 19 | EOL | API\| Lang\| VM\| Notes | JDK\| JRE | 18 17 16 15 14 13 12 11 10 9 ... |
| Java 18 | EOL | API\| Lang\| VM\| Notes | JDK\| JRE | 17 16 15 14 13 12 11 10 9 8 ... |
| Java 17 | LTS | API\| Lang\| VM\| Notes | JDK\| JRE | 16 15 14 13 12 11 10 9 8 7 ... |
| Java 16 | EOL | API\| Lang\| VM\| Notes | JDK\| JRE | 15 14 13 12 11 10 9 8 7 6 ... |
| Java 15 | EOL | API\| Lang\| VM\| Notes | JDK\| JRE | 14 13 12 11 10 9 8 7 6 5 ... |
| Java 14 | EOL | API\| Lang\| VM\| Notes | JDK\| JRE | 13 12 11 10 9 8 7 6 5 1.4 ... |
| Java 13 | EOL | API\| Lang\| VM\| Notes | JDK\| JRE | 12 11 10 9 8 7 6 5 1.4 1.3 ... |
| Java 12 | EOL | API\| Lang\| VM\| Notes | JDK\| JRE | 11 10 9 8 7 6 5 1.4 1.3 1.2 ... |
| Java 11 | LTS | API\| Lang\| VM\| Notes | JDK\| JRE | 10 9 8 7 6 5 1.4 1.3 1.2 1.1 |
| Java 10 | EOL | API\| Lang\| VM\| Notes | JDK\| JRE | 9 8 7 6 5 1.4 1.3 1.2 1.1 |
| Java 9 | EOL | API\| Lang\| VM\| Notes | JDK\| JRE | 8 7 6 5 1.4 1.3 1.2 1.1 |
| Java 8 | LTS | API\| Lang\| VM\| Notes | JDK\| JRE | 7 6 5 1.4 1.3 1.2 1.1 |
| Java 7 | EOL | API\| Lang\| VM\| Notes | JDK\| JRE | 6 5 1.4 1.3 1.2 1.1 |
| Java 6 | EOL | API\| Lang\| VM\| Notes | JDK\| JRE | 5 1.4 1.3 1.2 1.1 |
| Java 5 | EOL | API\| Lang\| VM\| Notes | | 1.4 1.3 1.2 1.1 |
| Java 1.4 | EOL | API | | 1.3 1.2 1.1 |
| Java 1.3 | EOL | API | | 1.2 1.1 |
| Java 1.2 | EOL | API\| Lang | | 1.1 |
| Java 1.1 | EOL | API | | |
| Java 1.0 | EOL | API\| Lang\| VM | | |
| Pre 1.0 | EOL | | | |

Data Source

*Abbildung 1: Die Einstiegsseite listet alle Java-Hauptversionen und verlinkt relevante Informationen und die zugehörige Dokumentation (© javaalmanac.io)*

- **DEV** – Eine JDK-Version, die sich zurzeit noch in Entwicklung befindet.
- **REL** – Ein Feature-Release, der nur bis zum nächsten JDK-Release gewartet wird (also maximal sechs Monate).
- **LTS** – Ein Long-Term-Support-Release, der für mehrere Jahre gewartet wird.
- **EOL** – Diese JDK-Versionen werden nicht mehr gewartet (End of Life).

Um eine bestimmte JDK-Version dann auch tatsächlich nutzen zu können, benötigt man eine sogenannte Distribution. Das OpenJDK-Projekt selbst stellt lediglich den Source-Code des JDK auf GitHub zur Verfügung *[3]*. Diesen kann man entweder selbst zu ausführbarem Binärcode bauen (was in der Praxis eher selten vorkommt) oder aber fertige, getestete Artefakte von verschiedenen Projekten und Herstellern beziehen. Es gibt freie Distributionen sowie kommerzielle Angebote mit zusätzlichen Features und Support.

Der Java Version Almanac listet für jede Java-Hauptversion verfügbare Distributionen mit den jeweiligen Lizenzen und den jeweils unterstützten Hardware-Plattformen und Betriebssystemen. Dank der engen Zusammenarbeit mit dem foojay.io-Projekt werden auch direkt Downloadlinks angeboten, die mithilfe dessen Disco-API bezogen werden *[4]*.

## Was kommt, was geht?

Wer sich für neue Funktionen der Java-Versionen interessiert, findet in diesem Magazin sowie an vielen Stellen im Internet eine Vielzahl von guten Berichten über die aktuellen Java-Releases. Dabei kann man vier Bereiche unterscheiden, in denen sich das OpenJDK regelmäßig weiterentwickelt:

1. Java Virtual Machine (JVM)
2. Java-Sprache
3. Java-API
4. Werkzeuge

Die wichtigen Weiterentwicklungen werden im OpenJDK-Projekt zunächst in sogenannten Java Enhancement Proposals (JEP) spezifiziert *[5]*. Die JEPs sind eine hervorragende Quelle, um neue Funktionalitäten im Detail zu verstehen und sich auch in die Motivation dahinter und die bedachten Alternativen einzulesen.

Im Java Version Almanac sind neben Basisinformationen und den Referenzen auf die offiziellen Dokumentationen die JEPs pro Java-Release in einem einheitlichen Format aufgelistet und verlinkt. Tatsächlich sind auch die verlinkten Sprach- und JVM-Spezifikationen manchmal sehr nützliche Quellen, wenn man sich in gewisse Aspekte im Detail einlesen möchte. Zusatzinformationen wie zum Beispiel die jeweils zugehörige Bytecode-Version – die unglücklicherweise eine andere Zählweise als die JDK-Versionen hat – sind beim Debuggen von Inkompatibilitäten häufig hilfreich.

Auch das Java-API entwickelt sich kontinuierlich weiter. Teilweise sind es kleine praktische Ergänzungen wie `java.util.HexFormat` in Java 17 oder ganze neue Packages wie `java.util.random` im selben Release.

Im Java Version Almanac gibt es die Möglichkeit, die APIs beliebiger

**New APIs in Java 20**

Comparing **Java 20** (20+36-2344-open) with **Java 19** (19.0.2+7-tem).

| Element | Modification |
| --- | --- |
| java.base | |
| java.io | |
| BufferedInputStream | |
| transferTo(OutputStream) | added |
| Console | - final |
| PushbackInputStream | |
| transferTo(OutputStream) | added |
| SequenceInputStream | |
| transferTo(OutputStream) | added |
| java.lang.constant | |
| ClassDesc | |
| ofInternalName(String) | added |

*Abbildung 2: Dem detaillierten Vergleich der APIs aller Java-Versionen entgeht keine noch so kleine Änderung. Die farbigen Tags sind jeweils mit dem zugehörigen CSR verlinkt (© javaalmanac.io)*

Java-Hauptversionen zu vergleichen *(siehe Abbildung 2)*. Dazu werden im Hintergrund täglich für die jeweils aktuellen Patch-Versionen die APIs automatisch auf Bytecode-Ebene verglichen. Das OpenJDK-Projekt ist sehr behutsam mit API-Änderungen und erfordert für jede Änderung einen sogenannten Compatibility & Specification Review (CSR) *[6]*. Diese Reviews werden öffentlich zugänglich als JIRA-Ticket im OpenJDK-Projekt geführt und erhalten häufig nützliche Zusatzinformationen zum Grund der Änderung. Deshalb ist auf den API-Vergleichsseiten jede API-Änderung mit dem zugehörigen CSR verlinkt.

Die genauere Betrachtung der API-Vergleiche der letzten OpenJDK-Releases zeigt ein Novum in der Java-Plattform: Konnte man sich die ersten zwei Jahrzehnte darauf verlassen, dass Java immer aufwärtskompatibel ist, sehen wir jetzt die ersten deprecated APIs, die tatsächlich auch entfernt wurden. Obwohl diese APIs vorgängig mit dem Flag `forRemoval=true` versehen werden, ist man jetzt gezwungen, die Codebasis regelmäßig auf aktuelle Schnittstellen umzuschreiben, wenn man ein Projekt langfristig kompatibel zu neuen Java-Versionen halten möchte.

## Open Data als Grundprinzip

Das Projekt javaalmanac.io kommt zunächst als reine HTML-Oberfläche mit weitgehend statischen Inhalten daher. Im Hintergrund sind jedoch wo möglich sämtliche Informationen als strukturierte JSON-Daten in einem GitHub-Repository gehalten *[7]*. Diese dienen nicht nur als Quelle für die regelmäßige Generierung der HTML-Darstellung, sondern können direkt auch über eine in OpenAPI 3.0 spezifizierte Schnittstelle abgerufen und frei genutzt werden *[8]*. Das Grundprinzip, die zugrunde liegenden Daten in strukturierter Form frei zur Verfügung zu stellen, ermöglich Kooperation wie zum Beispiel mit dem foojay.io-Projekt, das die Daten in etwas anderer Form aufbereitet *[9]*. Umgekehrt ermöglicht die Ablage der Daten in einem öffentlichen Git-Repository regelmäßige Beiträge durch die Java-Community in Form von Ergänzungen oder Korrekturen.

*Abbildung 3: Die OpenAPI-3.0-Spezifikation der Datenschnittstelle im Swagger-Editor (© javaalmanac.io)*

Neben den bisher beschriebenen Aspekten beinhaltet der Java Version Almanac weitere Informationen (zum Beispiel eine Bytecode-Referenz) und stellt interaktive Elemente zur Verfügung: Mit sogenannten Sandboxes für alle aktuellen Java-Versionen kann Java-Code direkt im Browser editiert und ohne lokale Installation ausgeführt werden. Dies ermöglicht es, neue Sprach-Konstrukte und Schnittstellen direkt auszuprobieren. Erwähnenswert ist dabei die Zusammenarbeit mit Cay Horstmann, der regelmäßig ausführliche Beschreibungen für neue Sprach-Features beiträgt und dazu in seine Artikel interaktive Code-Beispiele mit den genannten Sandboxes einbettet *[10]*.

Für den Autor hat das Java-Version-Almanac-Projekt mal wieder bewiesen, welches Potenzial die Zusammenarbeit mit freien, quelloffenen Daten birgt. Dank zahlreicher Ideen und Beiträge der Entwicklergemeinde haben die Inhalte einen Umfang und eine Qualität erreicht, die der ursprünglichen privaten Notizensammlung weit überlegen ist.

## Quellen


[1] *https://javaalmanac.io/*
[2] *https://spring.io/blog/2022/05/24/preparing-for-spring-boot-3-0*
[3] *https://github.com/openjdk/*
[4] *https://github.com/foojayio/discoapi*
[5] *https://openjdk.org/jeps/0*
[6] *https://wiki.openjdk.org/display/csr*
[7] *https://github.com/marchof/java-almanac*
[8] *https://data.javaalmanac.io/v1/openapi.yaml*
[9] *https://foojay.io/almanac/java-17*
[10] *https://javaalmanac.io/features/*

**Marc Hoffmann**
mtrail GmbH
*marc.hoffmann@mtrail.ch*


Java Champion Marc Hoffmann war seit dem Erscheinen von Java 1.0 im Jahre 1996 in den unterschiedlichsten kommerziellen und quelloffenen Java-Projekten involviert. Unter anderem arbeitet er an dem quelloffenen Werkzeug JaCoCo zur Messung von Test-Abdeckung. Dies erfordert die fortlaufende Beschäftigung mit den neuesten Java-Versionen, um das Werkzeug kompatibel zu halten.

In seinem Hauptberuf ist Marc Hoffmann CTO der mtrail GmbH. Ein Teil seiner Verantwortung ist der technische Lebenszyklus von geschäftskritischen Anwendungen für Eisenbahnen, was auch das regelmäßige Update auf aktuelle Java-Versionen beinhaltet.

Community-Konferenz organisiert von Java User Groups aus dem Norden

# JAVA FORUM NORD

Hannover Congress Centrum, Dienstag 12. September 2023

Das Java Forum Nord ist die Community Konferenz organisiert von Java User Groups (nicht nur) aus dem Norden. Nach dem es pandemiebedingt in den letzten Jahren schwierig war, wollen wir in 2023 wieder das volle Programm auffahren. Informiere dich in unzähligen Vorträgen von hochkarätigen Speakern über neuste Technologien und die aktuellen Entwicklungen im Java Umfeld.

**Dieses Jahr wieder mit Social Event und JFN for Kids!**

Jetzt Tickets sichern unter javaforumnord.de

**Keynotes von**

Sandra Parsick

Christina Hartmann

javaforumnord.de @JavaForumNord@ijug.social

eck*cellent IT · bdr. · holisticon · Die Techniker · cloudogu · OPEN KNOWLEDGE · WPS Management · HEC IT-Engineering · CGS · Sparkasse Finanz Informatik · CNC Consult · Java aktuell · JavaSPEKTRUM · MICROMATA · embarc · msg David

# Ein Blick über den Java-Tellerrand: Kotlin und Clojure

Fredrik Winkler, WPS – Workplace Solutions GmbH

*In diesem Artikel werden zwei funktionale Java-Beispiele nach Kotlin und Clojure portiert und dafür relevante Sprachfeatures und APIs diskutiert, insbesondere Null Safety, Extension Functions, Makros und Transducer. Zielpublikum sind erfahrene Java-Programmierer, die an alternativen JVM-Sprachen interessiert sind, aber noch keine konkrete Vorstellung davon haben, was sie in Kotlin oder Clojure erwartet.*

Kotlin wird seit 2011 von der Firma JetBrains entwickelt, um die Weiterentwicklung der hauseigenen IDEs (unter anderem IntelliJ IDEA) zu vereinfachen. Industriell fand Kotlin zunächst eine Nische in Android-Frontends, mittlerweile werden aber auch JVM-Backends zunehmend in Kotlin geschrieben *[1]*. Interop zwischen Kotlin und Java ist in beide Richtungen möglich und weitestgehend reibungsfrei; Interop-Reibungen entstehen primär durch Kotlins Berücksichtigung von `null` im Typsystem.

Clojure (2007) ist ein dynamisch getypter LISP-Dialekt, der nach Java-Bytecode kompiliert wird. Clojure hat Interop-Syntax für den direkten Zugriff auf Konstruktoren, Methoden und Felder von Java. Für weitere Interop-Belange (Arrays, Interfaces…) existieren APIs. Clojures eigene Collections (PersistentList, PersistentVector, PersistentHashSet, PersistentHashMap…) sind unveränderlich; Funktionen zum Einfügen/Entfernen/Ersetzen eines Elements erzeugen neue Collections, dank *Structural Sharing* mit sublinearem Kopieraufwand. Dadurch wird ein funktionaler Programmierstil gefördert, aber nicht erzwungen. Für Robert „Uncle Bob" Martin (Clean Code, SOLID) ist Clojure seit 2010 Programmiersprache der Wahl *[2]*.

## Erstes Beispiel: Fakultät

Die Fakultät einer Zahl n ist das Produkt aller Zahlen von 1 bis n. In der Ausbildung programmiert man die Fakultät klassisch iterativ oder rekursiv. Mit heutigen funktionalen APIs geht das aber auch ohne explizite Schleifen oder Selbstaufrufe.

*Listing 1* zeigt eine mögliche Implementation in Java 8. `iterate` liefert einen unendlich langen Stream aller natürlicher Zahlen. Dieser wird durch `limit` auf die Zahlen von 1 bis n beschränkt. Anschließend werden diese Zahlen mittels `reduce` multipliziert. Bei n=0 ist der Stream allerdings leer; dann liefert `reduce` einen leeren Optional, welcher mithilfe `orElse` auf die Fakultät von 0 abgebildet wird, also 1.

## Fakultät in Kotlin

*Listing 2* zeigt eine mögliche Implementation in Kotlin. Im Vergleich mit Java fallen als Erstes die optionalen Semikola auf. Außerdem erfordern „statische" Imports in Kotlin kein `static`. Funktionen können außerhalb von Klassen definiert werden. In Deklarationen steht der Typ ganz rechts, hinter einem Doppelpunkt.

Scheinbar hat `generateSequence` nur ein Argument, tatsächlich sind es aber zwei: Das zweite Argument ist die Lambda-Funktion in den (beziehungsweise mit den) geschweiften Klammern. Wenn das letzte Argument eines Aufrufs ein Lambda ist, kann man dieses aus der Argumentliste herausziehen. Diese Syntaxregel hat sich in der Programmiersprache Groovy für eigene Kontrollstrukturen und DSLs bewährt.

Der Aufruf `x.inc()` funktioniert, obwohl die Klasse `BigInteger` gar keine `inc`-Methode beinhaltet. Stattdessen definiert Kotlin in der Standardbibliothek eine entsprechende *Extension Function*. Kompiliert werden Extension Functions zu den in Java üblichen statischen Hilfsmethoden; deshalb können Extension Functions *nicht* auf private Interna zugreifen, die Kapselung bleibt also gewährleistet!

`reduceOrNull` liefert `null`, falls die Sequence leer ist, und das spiegelt sich auch im Typsystem wider: `BigInteger?` kann `null` sein, während BigInteger niemals null ist. Mittels `?: ONE` wird `null` zu `ONE` konvertiert. Ohne diese Konvertierung wäre die Funktion nicht übersetzbar, weil `BigInteger?` kein Sub-

```
package user;

import         java.math.BigInteger;
// ...
import static  java.math.BigInteger.ONE;

public class Java {
  public static BigInteger fakultät(long n) {
      return Stream.iterate(ONE, x -> x.add(ONE))  // Stream<BigInteger>
          .limit(n)                                // Stream<BigInteger>
          .reduce(BigInteger::multiply)            // Optional<BigInteger>
          .orElse(ONE);                            // BigInteger
  }
}
```

*Listing 1: Fakultät in Java 8*

```
package user

import java.math.BigInteger
import java.math.BigInteger.ONE // ohne static

fun fakultät(n: Int): BigInteger {
  return generateSequence(ONE) { x -> x.inc() } // Sequence<BigInteger>
      .take(n)                                  // Sequence<BigInteger>
      .reduceOrNull(BigInteger::multiply)       // BigInteger?
      ?: ONE                                    // BigInteger
}
```

*Listing 2: Fakultät in Kotlin*

```
fun alleFakultäten(): Sequence<BigInteger> = sequence {
  var ergebnis = ONE
  yield(ergebnis)
  var index = ONE
  while (true) {
     ergebnis *= index
     yield(ergebnis)
     ++index
  }
}

fun main() {
  for (x in alleFakultäten()) {
     println(x)
  }
}
```

Listing 3: Alle Fakultäten mit yield

typ von `BigInteger` ist (sondern andersherum). Zugriffe auf `BigInteger?` mittels gewohnter Punktnotation sind statisch verboten, das geht nur nach expliziter Prüfung auf `null`. Daher treten `NullPointerExceptions` in reinem Kotlin praktisch nie auf.

## Alle Fakultäten mit yield

Wenn man mehrere aufeinanderfolgende Fakultäten berechnen möchte, dann ist es Rechenzeitverschwendung, die Produktberechnung für jede Fakultät immer wieder von vorne beim Faktor 1 anfangen zu lassen. Stattdessen kann man *alle* Fakultäten als unendliche Sequence berechnen, siehe Listing 3.

Das Gleichheitszeichen hinter dem Ergebnistyp verhält sich wie ein Funktionsrumpf, der nur aus einer einzigen `return`-Anweisung besteht. Der scheinbare Funktionsrumpf ist in Wirklichkeit ein *Extension-Lambda*-Argument an `sequence`.

Jedes `yield` pausiert den Kontrollfluss und liefert ein Ergebnis zurück. Sobald der Aufrufer sich für die nächste Fakultät interessiert, wird der Kontrollfluss nach dem zuletzt pausierten `yield` fortgesetzt. In Python, JavaScript und C# ist `yield` ein Schlüsselwort. In Kotlin ist `yield` dagegen eine Methode aus der Standardbibliothek, die auf einem fundamentaleren Sprachmittel aufbaut: Suspend Functions. Asynchrone Programmierung (`async/await`) basiert ebenfalls darauf.

Dank *Extension Operator Functions* funktionieren *= und ++ auch für `BigInteger`.

## Fakultät in Clojure

Listing 4 beginnt mit `ns`, das steht für „namespace" und erfüllt grob den Zweck von Javas `package` und `import`. Ein Namespace wird zu einer Klasse kompiliert und alle folgenden Definitionen zu statischen Feldern in dieser Klasse.

Mit `defn` definiert man eine Funktion, der Vector `[^long n]` ist die Parameter-„Liste". Die Typ-Angaben sind optional und abgesehen von Java-Interop kulturell eher unüblich. Jede Clojure-Funktion ist ein Java-Objekt mit 22 `invoke`-Methoden für 0 bis 20 Object-Parameter beziehungsweise mehr als 20 Object-Parameter („Varargs"). Um Boxing zu vermeiden, können Parametertypen und Ergebnistyp auf `long` oder `double` spezialisiert werden; aufgrund

```
(ns user
  (:require [clojure.test :refer [deftest is run-tests]]))

(defn fakultät ^Number [^long n]

  ; #1 von unten nach oben lesen
  (reduce *'
     (take n
        (iterate inc
           1)))

  ; #2 von oben nach unten lesen
  (->> 1
     (iterate inc)
     (take n)
     (reduce *')))

(deftest fakultät-test
  (is (= (fakultät 0) 1))
  (is (= (fakultät 1) 1))
  (is (= (fakultät 2) 2))
  (is (= (fakultät 3) 6))
  (is (= (fakultät 4) 24))
  (is (= (fakultät 5) 120))
  (is (= (fakultät 6) 720))
  (is (= (fakultät 7) 5040)))

(run-tests)
```

Listing 4: Fakultät in Clojure

der kombinatorischen Explosion ist diese Optimierung aber nur für Parameterlisten der Länge 0 bis 4 vorgesehen.

Bei Funktionsaufrufen steht die Funktion *in* den runden Klammern anstatt davor, das heißt, `f(x, y)` wird zu `(f x y)`. Aufgrund dieser Präfix-Syntax braucht Clojure nicht zwischen Operatoren und Funktionen zu unterscheiden: Scheinbare Operatoren sind einfach nur ganz normale Funktionen mit ungewöhnlichen Namen. Kommata werden vom Clojure-Lexer grundsätzlich ignoriert beziehungsweise wie Whitespace behandelt.

LISP-typisch muss man komplizierte Ausdrücke häufig von rechts nach links beziehungsweise von unten nach oben lesen: Man fängt bei 1 an, iteriert mit Erhöhung, nimmt die ersten n Zahlen und reduziert per Multiplikation. Das Makro –>> dreht den Lesefluss in die für Menschen angenehmere Richtung um. Makros sind im Prinzip ganz normale (jedoch speziell markierte) Funktionen, die Code vor der eigentlichen Kompilierung in anderen Code transformieren. Dank Makros ist der eigentliche Sprachkern von Clojure erstaunlich klein; selbst scheinbare Primitive wie `defn` oder `for` sind als Makros realisiert.

Clojure hat drei Multiplikationsfunktionen:

1. * wirft beim Überlauf eine Exception
2. *' wechselt bei drohendem Überlauf von java.lang.Long auf clojure.lang.BigInt
3. `unchecked-multiply` ignoriert den Überlauf, genau wie * in Java/Kotlin

Konkret liefert `fakultät` also kleine Longs oder große BigInts; daher der gemeinsame Supertyp Number als Ergebnistyp von `fakultät`.

Mit `deftest` definiert man einen Test. Die Gleichheit wird mit `(= x y)` geprüft, was im Wesentlichen auf `equals` basiert. Die

```
(defpure fakultät
  {[0] 1
   [1] 1
   [2] 2
   [3] 6
   [4] 24
   [5] 120
   [6] 720
   [7] 5040}
  [n]
  (->> 1
     (iterate inc)
     (take n)
     (reduce *')))
```

Listing 5: Testen purer Funktionen vereinfachen

Referenz-Gleichheit beziehungsweise Identität kann mit `(identical? x y)` geprüft werden, aber das benötigt man eigentlich nie.

`clojure.test/is` verhält sich ähnlich wie `assertTrue` aus JUnit, allerdings sind die Fehlermeldungen deutlich hilfreicher. Wenn man die Fakultätsfaktoren etwa versehentlich bei 0 statt 1 anfangen lässt, erhält man folgende Fehlermeldung:

FAIL in (fakultät-test)
expected: (= (fakultät 1) 1)
 actual: (not (= 0 1))

`clojure.test/is` zeigt also sowohl den Code als auch die ausgewerteten Argumente an. (Ohne Makros wäre das so nicht möglich!) Weitere Funktionen wie JUnits `assertEquals` beziehungsweise komplette `assertThat`-DSLs entfallen damit.

## Testen purer Funktionen vereinfachen

Beim Testen von puren Funktionen hat mich die Syntax `(is (= (f eingaben) ausgabe))` irgendwann genug gestört, um mir ein Makro `defpure` zu schreiben, das die Test-Eingaben/-Ausgaben aus der Definition der Funktion selbst extrahiert. *Listing 5* zeigt ein Anwendungsbeispiel.

Nach dem Funktionsnamen kommen die Testdaten in einer Map. Jeder Key ist ein Eingabewerte-Vector und jeder Value der erwartete Ausgabewert. Die separate Testfunktion entfällt. `defpure` erleichtert nicht nur das Schreiben von Tests, sondern dient auch der exemplarischen Dokumentation: Konkrete Beispieldaten sind oft eine sinnvolle Ergänzung zu abstrakten Beschreibungstexten, insbesondere für Randfälle.

```
(defn alle-fakultäten []
  (->> 1
     (iterate inc)
     (reductions *' 1)))

(deftest alle-fakultäten-test
  (is (=
         (take 8 (alle-fakultäten))
         [1 1 2 6 24 120 720 5040])))
```

Listing 6: Alle Fakultäten mit reductions

Die Implementation von `defpure` ist gar nicht so aufwendig *[3]*, aber alle Makro-Details von Clojure zu erklären, würde den Rahmen dieses Artikels sprengen.

## Alle Fakultäten mit reductions

Zum Berechnen *aller* Fakultäten muss man lediglich `reduce` durch `reductions` ersetzen, siehe *Listing 6*. Die zusätzliche 1 beim Aufruf von `reductions` ist erforderlich, weil ansonsten die Fakultät von 0 im Ergebnis fehlen würde. Ein Testen mittels `defpure` ist nicht möglich, weil das erwartete Ergebnis unendlich lang ist.

## Zweites Beispiel: Alte Namen

*Listing 7* berechnet die Namen von Menschen, die vor 1950 geboren wurden, ohne Duplikate. Java Streams sind lazy, das heißt, weder `filter` noch `map` rufen das übergebene Lambda beziehungsweise die übergebene Methodenreferenz selber auf. Erst `collect` triggert pro Mensch jeweils *abwechselnd* das Lambda und gegebenenfalls die Methodenreferenz. Das Set eliminiert wie gewünscht doppelte Namen. Java Streams dürfen nicht wiederverwendet werden, sonst droht eine `IllegalStateException`.

## Alte Namen in Kotlin

In *Listing 8* fällt zunächst auf, dass Lambdas ohne explizite Parameterdefinition einen impliziten Parameter namens `it` haben.

In Kotlin funktionieren `filter` und `map` dank Extension Functions direkt auf Iterables. Dann geschieht die Verarbeitung allerdings eager, das heißt `filter` befüllt sofort eine Liste alter Menschen und `map` befüllt sofort eine Liste derer Namen.

Durch das eingeschobene `asSequence` arbeiten `filter` und `map` lazy, das bedeutet, sie liefern selber nur Sequences zur späteren Verarbeitung zurück. Das Sequence-Interface beinhaltet genau wie Iterable (bis Java 7) nur eine `iterator`-Methode. Ein und dieselbe Sequence kann beliebig oft erneut verarbeitet werden, ähnlich wie C# IEnumerables.

```
record Mensch(String name, int jahrgang) {

     static Set<String> alteNamen(Collection<Mensch> menschen) {

     return menschen.stream()               // Stream<Mensch>
            .filter(m -> m.jahrgang < 1950) // Stream<Mensch>
            .map(Mensch::name)              // Stream<String>
            .collect(Collectors.toSet());   //    Set<String>
     }
}
```

Listing 7: Alte Namen in Java 8

```
data class Mensch(val name: String, val jahrgang: Int)

fun eagerNamen(menschen: Collection<Mensch>): Set<String> {

  return menschen                      // Collection<Mensch>
      .filter { it.jahrgang < 1950 }   //        List<Mensch>
      .map(Mensch::name)               //        List<String>
      .toSet()                         //        Set<String>
}

fun lazyNamen(menschen: Collection<Mensch>): Sequence<String> {

  return menschen.asSequence()         // Sequence<Mensch>
      .filter { it.jahrgang < 1950 }   // Sequence<Mensch>
      .map(Mensch::name)               // Sequence<String>
      .distinct()                      // Sequence<String>
}
```

Listing 8: Alte Namen in Kotlin

| Funktion | Typ |
|---|---|
| filter | Prädikat → Transducer |
| map | Selektor → Transducer |
| comp | Transducer, Transducer → Transducer |
| Transducer | Schritt → Schritt |
| Schritt | Zwischenergebnis, Element → Zwischenergebnis |
| reduce | Schritt, Startwert, Elemente → Ergebnis |
| transduce | Transducer, Schritt, Startwert, Elemente → Ergebnis |
| into | Collection, Transducer, Elemente → Collection |
| sequence | Transducer, Elemente → LazySeq |

## Alte Namen in Clojure

Listing 9 zeigt acht Varianten mit unterschiedlichem Verhalten. `filterv`, `mapv` und `set` arbeiten eager. (Der Buchstabe `v` am Ende von `filterv` und `mapv` steht für den Ergebnistyp PersistentVector.)

Die Raute vor der runden Klammer ist eine Kurzschreibweise für Funktionen. Deren Parameter heißen %1 bis %20, wobei %1 durch % abgekürzt werden kann, analog zu Kotlins `it`.

```
(defpure alte-namen
  {[[{:name "Alan", :jahrgang 1912}
     {:name "Alan", :jahrgang 1940}
     {:name "Bjarne", :jahrgang 1950}
     {:name "Brian", :jahrgang 1942}
     {:name "Dennis", :jahrgang 1941}
     {:name "James", :jahrgang 1955}]]
   #{"Alan" "Brian" "Dennis"}}
  [menschen]

  ;;; 3x EAGER
  (->> menschen
     (filterv #(< (:jahrgang %) 1950))   ; PersistentVector
     (mapv :name)                        ; PersistentVector
     set)                                ; PersistentHashSet

  ;;; 3x LAZY
  (->> menschen
     (filter #(< (:jahrgang %) 1950))    ; LazySeq
     (map :name)                         ; LazySeq
     distinct)                           ; LazySeq

  ;;; 1x EAGER
  (reduce                                ; PersistentHashSet
     (fn schritt [namen mensch]
        (if (< (:jahrgang mensch) 1950)
           (conj namen
              (:name mensch))
           namen))
     #{}                                 ; PersistentHashSet
     menschen)

  ;;; TRANSDUCER
  (let [deko (comp                                 ; Transducer
               (filter #(< (:jahrgang %) 1950))    ; Transducer
               (map :name))]                       ; Transducer
     ; 1x EAGER
     (reduce   (deko conj) #{} menschen)    ; PersistentHashSet

     (transduce deko conj  #{} menschen)    ; PersistentHashSet

     (into  #{} deko menschen)              ; PersistentHashSet
     ; 1x LAZY
     (sequence (comp deko (distinct)) menschen))  ; LazySeq

  ;;; COMPREHENSION
  (set                                             ; PersistentHashSet
     (for [mensch menschen                                   ; LazySeq
           :when (< (:jahrgang mensch) 1950)]
        (:name mensch))))
```

Listing 9: Alte Namen in Clojure

Die *Keywords* `:jahrgang` und `:name` werden hier als Funktionen verwendet. Dann suchen sie sich selber als Key in einer Map und liefern den entsprechenden Value.

`filter`, `map` und `distinct` arbeiten lazy, wenn man Elemente übergibt; dann ist der Ergebnistyp eine (einfach verkettete) LazySeq. Eine LazySeq berechnet jedes Element erst auf Anfrage und speichert es dann für eventuelle spätere Zugriffe. Theoretisch kann man LazySeqs also längerfristig speichern, das macht man aber eher selten. Im gezeigten Beispiel dienen sie der Einsparung kompletter Zwischen-Collections im Arbeitsspeicher: Bereits verarbeitete und nicht länger benötigte Präfixe einer LazySeq können frühzeitig vom Garbage Collector entsorgt werden.

Mit `reduce` kann man die temporären LazySeqs vermeiden. Dazu beginnt man mit der leeren Namensmenge `#{}` und übergibt eine Schritt-Funktion (im Code `schritt` genannt) an `reduce`, die bei einem jungen Menschen einfach die Eingangsmenge liefert (also dessen Namen ignoriert) und bei einem alten Menschen mittels `conj` eine neue Namensmenge berechnet, die auch dessen Namen enthält.

Anstatt komplizierte Schritt-Funktionen von Hand zu schreiben, kombiniert man Dekorierer-Funktionen namens *Transducer*. Ohne übergebene Elemente liefern `filter` und `map` (und `distinct`) keine LazySeqs, sondern Transducer. Deren Komposition ergibt einen weiteren Transducer (im Code `deko` genannt) mit der Eigenschaft, dass (`deko conj`) sich genau wie `schritt` verhält; Transducer-Aufrufe dekorieren also (im Sinne des Dekorierer-Musters) Schritt-Funktionen.

Anstatt `(deko conj)` an `reduce` zu übergeben, übergibt man `deko` und `conj` besser separat an `transduce`, weil das auch für Schritt-Funktionen mit Ausgabe-Puffern funktioniert, die nach der eigentlichen Reduktion noch geflusht werden müssen. (Ein Beispiel dafür wäre die Pufferung der aktuellen Partition durch `partition-by`.) Die schnellste Variante ist `into`, weil dann als Zwischenergebnis eine *Transient Collection* mit verringertem Kopieraufwand zum Einsatz kommt.

Um das Ergebnis lazy zu berechnen, verwendet man `sequence` statt `into`. Transducer selber sind also weder eager noch lazy; das hängt allein von ihrer Verwendung ab. Transducer sind nicht auf Collections oder LazySeqs beschränkt, sie werden beispielsweise auch für Channels und Observables eingesetzt. Transducer sind eine Meisterleistung der Entkopplung: Wer hätte gedacht, dass `filter` und `map` komplett losgelöst von Collections, Sequences, Streams und Ähnlichem implementierbar sind?

Falls Effizienz zweitrangig ist, freuen sich Pythonistas über Clojures lesbare `for`-Comprehension. Wie so häufig ist for aber nicht Teil des Sprachkerns, sondern nur ein Makro aus der Standardbibliothek.

## Fazit

Wer ein „Java nach dem Frühjahrsputz“ mit ein paar Killer-Features sucht (vor allem Null Safety und Extension Functions), ist mit Kotlin gut beraten. Wer dagegen tiefer in funktionale Programmierung einsteigen möchte, dem sei Clojure ans Herz gelegt.

## Quellen


[1] *https://twitter.com/tsmith/status/1581318139895156737*
[2] *https://blog.cleancoder.com/uncle-bob/2021/06/25/OnTypes.html*
[3] *https://github.com/fredoverflow/clopad/blob/master/samples/defpure.clj*

**Fredrik Winkler**
WPS – Workplace Solutions GmbH
*fred@wps.de*


Fredrik arbeitet mit Schwerpunkt Nachwuchsgewinnung und -förderung bei der WPS. 2015 entdecke er Kotlin und entwickelt seitdem zwei Lernumgebungen mit Kotlin („Karel“ und „Skorbut“) in seiner Freizeit. Den Advent of Code bestreitet er seit 2020 in seiner eigenen Clojure-Lernumgebung („Clopad“). Für Einsteiger mit Java-Hintergrund bietet Fredrik eine ganztägige Kotlin-Schulung sowie einen ganztägigen „Advent of Clojure“-Workshop an.

# NEWSLETTER

Anmeldung

**Java aktuell: der iJUG Newsletter informiert dich alle vier Wochen über das aktuelle Geschehen in der Java-Welt und im iJUG.**

**Abonniere ihn kostenfrei und bleibe immer auf dem Laufenden!**

https://shop.doag.org/newsletter/

oder nach dem Login mit euren Zugangsdaten direkt über euer Profil abonnieren.

# The Great Migration: Erfahrungen aus dem Umzug von Java nach Kotlin

Lars Adler, tarent solutions GmbH

*In diesem Artikel werden einige Hilfestellungen zur Migration für Code von Java nach Kotlin gegeben. Neben einer Übersicht über die Stolpersteine, die sich auf dem Weg befinden könnten, werden einige Beispiele und Anregungen aus der Erfahrung des Autors vorgestellt, die den Übergang sicherer und sanfter gestalten können.*

Als Google am 7. Mai 2019 ankündigte, bevorzugt Kotlin als Basis für Android zu verwenden, hinterließ dies einen bleibenden Eindruck in der App-Entwicklung. Der Schub war nicht nur in den Frontends wahrzunehmen, auch manches Backend wird inzwischen in der „neuen" Sprache geschrieben. (Anmerkung: Kotlin wurde von JetBrains erstmals 2011 präsentiert, die erste stabile Veröffentlichung kam allerdings erst 2016 heraus.) Drei Jahre (oder eine Pandemie) später ist sie bereits nicht mehr wegzudenken.

Derzeit werden neue Projekte meistens auf Kotlin 1.7 oder 1.8 entwickelt, Java kommt eigentlich nur noch infrage, wenn man die Version SE 8 besonders schätzt – das Android-Framework ist hier quasi in der Zeit eingefroren.

Was ist jedoch mit älteren Projekten zu tun? Zum Glück – oder auch nicht, das ist Ansichtssache – können Java und Kotlin in ein und demselben Projekt reibungslos nebeneinander existieren, da der Kotlin-Code einfach in Java-Klassen kompiliert werden kann. Es ist also jederzeit möglich, neue Elemente mit Kotlin hinzuzufügen, ohne dass davon die Entwicklung ausgebremst wird.

Mit der Zeit erzeugt das jedoch eine unübersichtliche und verwirrende Codebasis, die immer schwerer zu pflegen wird, also könnte es besser sein, noch zu warten. Sollte dann endlich das Signal kommen, dass der große Schalter umgelegt werden darf, kann man sich freuen – und die Frage stellen: Wo setzt man jetzt am besten an?

## Automatische Umwandlung

Dass uns die IDE anbietet, den Code selbst umzuschreiben (in IntelliJ IDEA unter: Code | Convert Java File to Kotlin File), ist beinahe ein Angebot, das man nicht ausschlagen kann. Allerdings ist es mit sehr viel Vorsicht zu genießen. So gut die Werkzeuge dafür auch sein mögen, es gibt immer wieder Fälle, in denen man den Code anschließend noch anpassen sollte oder sogar muss. Es bleiben sonst zu viele Artefakte zurück und manchmal kann der Code sogar völlig zerbrechen.

Einige der Konstrukte, die noch stark von Java geprägt sind, sollten auf jeden Fall durch äquivalente Kotlin-Phrasen ersetzt werden. Dies kann gegebenenfalls auch durch Automatisierungswerkzeuge wie zum Beispiel OpenRewrite erfolgen.

Niemals jedoch darf der gesamte Code in einem Aufwasch konvertiert werden! Es sei denn, man sucht gerne die Nadel im Heuhaufen…

Daher sollte man sich zuallererst einen Plan zurechtlegen:

1. Man entscheidet über die Reihenfolge, in der die Module und Packages gewandelt werden sollen. (Am besten eignen sich solche, die schon etwas Staub angesetzt haben.)
2. Die enthaltenen Klassen geht man einzeln durch und führt nach der Konvertierung die jeweils noch notwendigen Bereinigungen aus.
3. Klassen, von denen sich andere ableiten, und solche, die häufig von anderen referenziert werden, hebt man sich bis zum Schluss auf.

Regelmäßiges Testen und Absichern durch eine Testsuite sollte dabei Voraussetzung sein, andernfalls wäre jetzt die Gelegenheit, dies nachzuholen, bevor man die erste Konversion startet. Nur so bleibt sichergestellt, dass sich keine Fehler einschleichen, die am Ende das Kompilieren verhindern können. Eine konsequente Anwendung des Mikado-*[3]* oder Jenga-Prinzips ist hier definitiv zu empfehlen!

Kommen wir damit zu konkret zu empfehlenden Aufräumarbeiten.

## Null Checks

Eines der Hauptmerkmale von Kotlin ist Null Safety, das übersetzt sich jedoch nicht ganz so gut direkt aus Java. Sofern man nicht alle entsprechenden Variablen mit `@NonNull` versehen hat, werden hier noch viele Stellen auftauchen, in denen Kotlin Nullables (wie etwa `String?`) verwendet. Besonders häufig kommen diese bei Aufrufen externer Bibliotheken vor. Man sollte diese dann genauer betrachten und, wo möglich, entfernen. Eine aufmerksame IDE gibt häufig schon entsprechende Hinweise für unnötige Fragezeichen. Wenn die Prüfungen tatsächlich noch benötigt werden, können diese auch durch `?.let {…}` , `?.apply {…}` oder den Elvis-Operator (`?:`) ersetzt werden, sofern das den Code lesbarer macht *(siehe Listing 1)*.

```
// Java
if (text != null) answer = text;
else answer = "";

// Kotlin (elvis operator)
answer = text ?: ""
```

*Listing 1: null checks versus null safety*

Ansonsten gilt: Was nicht benötigt wird, kann raus. (Marie Kondo lässt grüßen.)

Wenn es beim Testlauf doch zu einer `NullPointerException` kommt – keine Panik! Das kann daran liegen, dass eine Java-Klasse eine Methode enthält, die zwar für ihren Rückgabewert `@NonNull` gesetzt hat, dann aber trotzdem wenige Zeilen tiefer `return null` ausgeführt wird. Ja, das hat der Autor persönlich gefunden. In einer offiziellen Java-Library. Von Android…

## Companion Objects

Da Kotlin kein `static` kennt, landen alle Variablen und Methoden mit diesem Modifier im sogenannten `companion object`, das man in der Regel am Ende der migrierten Klasse wiederfindet *(siehe Listing 2)*. Es lohnt sich meistens, auch dieses näher zu betrachten – `private` Variablen oder Ähnliches brauchen dort nicht zu liegen, sie können in die Klasse direkt verschoben werden. Ebenso Methoden und Felder, die nur von der Klasse selbst verwendet werden.

```
class WithStaticMembers {
     …

     companion object {
        @JvmStatic
        const val staticProperty = "statics"

        @JvmStatic
        fun staticFunction() { … }
     }
}
```

*Listing 2: Das Companion-Objekt, mit aus Java erreichbaren Elementen*

Etwas, das auch häufig hier auftaucht, sind statische Konstanten zur Vermeidung von „Magischen Werten“ beziehungsweise Magic Numbers *[1]*. Wenn diese tatsächlich nur selten (soll heißen: einmalig) verwendet werden, hat Kotlin auch hierfür eine Alternative parat: Denn oft reicht es, dem Parameter seinen Namen voranzustellen:

```
sendRequest(timeoutMs = 1000)
```

Wenn dieser sprechend genug ist, kann man sich das Auslagern in einer eigens benannten Referenz hier sparen.

Andere Methoden können gegebenenfalls in Extensions umgeschrieben werden, die dann auch global erreichbar werden *(siehe Listing 3)*.

```
// Java static
class Lookup {
     private static HashMap<String, String> map =
        new HashMap<String, String>() {
           { put("key", "value"); … }
        };

     public static String translate(String key) {
        String element = map.get(key)
        return (element != null) ? element : key
     }
}

// Kotlin Extension (outside class)
private val map = mapOf("key" to "value", … )
fun String.translate() = map[this] ?: this
```

*Listing 3: Lookup-Methode, liefert bei fehlender Antwort das Original zurück*

Sollte sich der Eindruck ergeben, dass hier jemand Companion Objects nicht besonders mag, dann ist das nicht ganz von der Hand zu weisen… Der Autor meint, außerhalb der Kommunikation zwischen Java- und Kotlin-Klassen kann man tatsächlich auf diese verzichten, da Kotlin hier oft bessere Alternativen bietet.

## Verzweigungen/Conditionals

Enthält der Code noch eine komplexe Folge von `if-else if-…`, für das bei der Erstellung ein `switch-case` nicht ausreichte? Mit `when` kann vieles davon jetzt auf die gleiche Weise behandelt werden *(siehe Listing 4)*. Generell empfiehlt es sich, für mehrfache Verzweigungen immer `when() {…}` zu verwenden und `if-else`-Statements nur noch für einfache Fälle zu nehmen.

```
val result = when {
     firstString.startsWith("prefix:") -> { … }
     lastString.endsWith("-suffix") -> { … }
     otherString.contains("_infix_") -> { … }
     else -> { … }
}
```

*Listing 4: Verzweigung mit argumentlosem „when“ und direkter Zuweisung*

Eine weitere Möglichkeit, den Code zu verbessern, ist, in der `if(condition)` immer den Positiv-Fall zu verarbeiten – das negierende Ausrufezeichen (!) übersieht man nicht mehr, wenn es gar nicht verwendet wird. Für den Fall, dass die Negierung doch benötigt wird, bietet Kotlin in ein paar Fällen auch entsprechende Extensions an (`.isNotNull(), .isNotEmpty()` etc.).

Das Einzige, was Kotlin hier „fehlt“, ist der ternäre Operator. Stattdessen wird einfach eine Zuweisung per `if()` verwendet, die ohnehin besser lesbar ist *(siehe Listing 5)*.

```
// Java (ternary operator)
final String result = isValid ? "Yes" : "No"

// Kotlin
val result = if (isValid) "Yes" else "No"
```

*Listing 5: Einfache Fallunterscheidung mit Zuweisung*

## Schleifen/Loops

Für Schleifen gab es bereits mit Java SE 8 Streams Alternativen zur Umsetzung. Kotlin bietet hier noch deutlich mehr Möglichkeiten, Listen zu mappen, zu filtern etc. Diese sollten auf jeden Fall eingesetzt werden, somit reicht oft eine Zeile, wozu früher ein längerer Algorithmus geschrieben werden musste *(siehe Listing 6)*.

Dabei gilt auch: Nicht über das Ziel hinausschießen. Manchmal ist eine einfache Schleife der bessere Weg. Kotlin bietet aber auch hier ein paar Tricks, mit denen man auf einfache Weise einen Wertebereich

```
IntRange(from, to) {i -> …}
```

oder eine Wiederholung programmieren kann:

```
repeat(x) {i -> …}
```

## Vererbung und Polymorphismus

Java und Kotlin enthalten sowohl objektorientierte als auch funktionale Elemente, legen jedoch verschiedene Schwerpunkte auf den jeweiligen Typ. Kotlin bietet unter anderem die Extension Functions an sowie einen `expect/actual`-Mechanismus als Alternative für Interfaces, die dabei helfen können, alte Vererbungshierarchien zu glätten oder sogar komplett aufzulösen.

Ein Grund, sich die vererbten Klassen für den Schluss aufzuheben, ist, dass diese dadurch noch während der Umstellung in den neuen Kotlin-Klassen einfach weiterverwendet werden können, während für das Aufrufen von Kotlin-Code in Java zusätzliche Annotations (`@JvmStatic` etc.) gesetzt werden müssen, die schließlich obsolet werden, nachdem auch die letzte Java-Klasse migriert worden ist. Das Gleiche gilt für Utils- und Helper-Klassen oder auch für die Companions, falls diese übergreifend verwendet werden. Spart man sich das bis zum Ende auf, kann man geschickt nach und nach diese Referenzen nach Java eliminieren.

## Java Standard Library

Was auch gerne zurückbleibt, sind Referenzen auf Java Libraries mit Klassen oder Objekten, für die Kotlin bereits eigene Varianten mitliefert. Diese sollten natürlich auf die entsprechenden Kotlin-Klassen

```
// Java
List<Display> filteredDisplays(List<String> list, String substring) {
      List<Display> filteredList = new ArrayList<>();
      for (String item: list) {
             if (item.contains(substring)) {
                    Display element = new Display(item);
                    filteredList.add(element);
             }
      }
      return filteredList;
}

// Kotlin (refactored to extension method & expression body)
fun List<String>.filteredDisplays(substring: String) =
         filter { it.contains(substring) }
             .map { Display(it) }
```

*Listing 6: Schleifen-Ersetzung*

umgestellt werden. Ein Beispiel ist das `java.time`-Package, das durch die entsprechende JetBrains-Library `kontlinx-datetime` ersetzt werden kann.

Doch nicht in jedem Fall ist das möglich. Kotlin besitzt beispielsweise keine eigene Reflection, sondern verwendet bei Bedarf die von Java. Durch inzwischen vorhandene Alternativen lohnt es sich aber, sie komplett zu entfernen. Das erfordert jedoch Ansätze wie die Verwendung von `inline` und `reified` für generische Methoden oder einen Symbol-Prozessor (ksp), der die Verbindungen bereits beim Kompilieren auflösen kann.

Generell gilt die Empfehlung, bestehende Abhängigkeiten zu Java-Referenzen rasch aufzulösen. Bietet Kotlin etwas an, das Java-Funktionen oder auch Utility-Bibliotheken von Drittanbietern ersetzen kann, sollte man dies auch verwenden, wie zum Beispiel: `String.format(...)` → String Templates.

Auch Setter- und Getter-Methoden sollte man zu ihrer Kotlin-Version umschreiben, wenn das vom Konverter übersehen wurde. Das gilt auch für parameterlose Methoden, also solche, die nur einen definierten oder berechneten Wert zurückgeben *(siehe Listing 7)*.

```
// Java
private int parameter
public int getParameter() {
      return parameter
}
public void setParameter(int value) {
      if (isValid(value)) param = value
}
public String getParameterAsString() {
      return String.valueOf(parameter)
}

// Kotlin
var parameter: Int
      set(value) = { if (value.isValid) field = value }

val parameterAsString get() = "$parameter"
```

*Listing 7: Encapsulation in Java und Kotlin*

## Framework Extensions

Viele Frameworks liefern zusätzliche Modul-Bibliotheken mit einer Reihe von Hilfs- und Convenience-Methoden. Man erkennt sie häufig an dem Suffix `-ktx` im Namen (für „Kotlin Extensions“).

Android *[4]* bietet unter anderem solche, die die Sichtbarkeit von Views per einfachem booleschem `true` oder `false` behandeln *(siehe Listing 8)*, lazy loading von ViewModels erlauben – `by viewModels / activityViewModels<>()` – und mehr. Oft können dadurch selbstgeschriebene Methoden und Boilerplate-Code stark reduziert oder sogar ganz hinfällig werden. (Beispielsweise mit `bundleOf()` für die Klasse `Bundle`, analog zu `listOf()` und `arrayOf()`.)

## Es wächst zusammen, was zusammengehört

Im Gegensatz zu Java erlaubt Kotlin auch die Definition mehrerer `public`-Klassen in derselben Datei. Dies kann sich bei vielen kleinen Klassen als nützlich erweisen, wie etwa Datenobjekte zur JSON-Deserialisierung, die sich über ein ganzes Package verteilen. Diese können nun übersichtlich und in logisch aufbauender Abfolge in einer einzigen Datei gesammelt werden – und am besten gleich als `data class`, damit man auch alle Vorzüge des automatischen Klonens und Vergleichens mit `.copy()`, `.equals()` und `.hashCode()` genießen kann *(siehe Listing 9)*.

Fußnote: `data class` kann keine Arrays klonen, diese sollte man daher besser auf Listen umstellen *[5]*.

## Fertig mit dem Application-Code – Was machen wir mit den Tests?

Nach jedem Migrationsschritt laufen ja die Tests mit (aber latürnich!) somit fällt jedes Mal sofort auf, sollte etwas brechen.

Häufig passiert das, wenn der Java-Code der Testklasse den Kotlin-Code nicht mehr erreicht, weil Teile verschoben wurden oder nicht mehr statisch sind. Dann müssen wir entweder in den sauren Apfel beißen und `@JvmStatic` und `companion objects` verwenden oder stattdessen den Testcode ebenfalls migrieren.

Dabei können wiederum Teile der Testklasse in ihrem eigenen `companion object` auftauchen – `@BeforeClass` und `@AfterClass` sind ja auch `static`-Methoden. Diese loszuwerden, ist unter JUnit 5 (alias „Jupiter“) möglich – das führt jedoch aufgrund der mangelnden Unterstützung durch Android & Robolectric gleich in den benachbarten Kaninchenbau (immer dem weißen Hasen nach ...).

```
// Java using View static constants
if (isValid) {
     sendButton.setVisibility(View.VISIBLE)
     warningTextView.setVisibility(View.GONE)
     warningImageView.setVisibility(View.INVISIBLE)
}

// Kotlin with extensions
import androidx.core.view.isGone
import androidx.core.view.isInvisible
import androidx.core.view.isVisible
…

if (isValid) {
     sendButton.isVisible = true              // false -> GONE
     warningTextView.isGone = true            // false -> VISIBLE
     warningImageView.isInvisible = true      // false -> VISIBLE
}
```

*Listing 8: Extensions für Android Klassen – androidx.core:core-ktx*

```
1    package com.example.datatransfer
2
3    data class Dataset (
4          val person: Person,
5          val address: Address
6    )
7
8    data class Person(
9          val firstName: String,
10         val lastName: String,
11         val title: String?
12   )
13
14   data class Address(
15         val street: String,
16         val city: String,
17         val zipCode: String
18   )
```

*Listing 9: DataSet.kt – Beispiel für eine Datenstruktur mit Immutable Objects*

Eine Alternative dazu bietet das ebenfalls die JUnit-Plattform verwendende Kotest. Damit ist es möglich, ebenfalls modular die Tests nach und nach in reines Kotlin umzuschreiben. Dies sollte allerdings erst in Angriff genommen werden, nachdem der Anwendungscode vollständig migriert wurde, um auf keinen Fall zu riskieren, dass einem während der Migration das Testgerüst zusammenbricht.

## Zusätzliche Refactorings

Natürlich kann auch bei schrittweiser Umstellung nicht alles gleich im ersten Durchgang bereinigt werden. Manche Dinge müssen einfach übergreifend überarbeitet werden, aber das ist ohnehin nichts Neues – dieser Fall tritt auch dann auf, wenn man nur die Version der Sprache ändert. Frameworks bekommen Updates oder können eventuell auch komplett ausgetauscht werden – das sollte auch bereits bei der Planung der Migration berücksichtigt werden. So kann man sich unter anderem frühzeitig darauf einigen, dass Dependency Injection mit Dagger oder Hilt abgelöst wird durch Koin oder Kodein.

Die Entwicklung von Kotlin selbst steht ebenfalls nicht still und neue Funktionen können den alten Code weiterhin außer Dienst stellen. Refactorings sind ein wichtiger Teil der Softwareentwicklung, sie sollten deswegen immer mit eingeplant werden.

Gerade im Hinblick auf die Entwicklung systemübergreifender Features wie Kotlin Multiplatform Mobile [6] ist es von Vorteil, die Codebasis frühzeitig auf eine gemeinsame Stufe zu stellen, sodass die hier vorgestellten Rezepte auch auf Sprachen wie Objective-C, Swift oder JavaScript angewendet werden können.

## Fazit

Es gibt wenige Gründe, die einen davon abhalten sollten, von Java auf Kotlin umzustellen. Bei entsprechender Planung können selbst komplexe Projekte in wenigen Wochen migriert werden, ohne dabei die Entwicklung auszubremsen. In manchen Fällen kann es allerdings nötig werden, auf einem getrennten Repository zu arbeiten und späte Änderungen am Schluss manuell zu übertragen.

Zudem bietet ein Migrationsprojekt auch immer Ein- und Umsteigern die Gelegenheit, aus ihrem Blickwinkel die neue Sprache kennenzulernen und, angeführt von ein paar Gurus, deren Eigenheiten gezeigt zu bekommen. Der Code sollte allen im Team die bevorzugten neuen Muster und Phrasen demonstrieren und nicht davor zurückschrecken, auch das eine oder andere neue Konzept vorzustellen.

Definitiv sollte man eine Migration nicht zu lange hinausschieben. Nicht nur ist das Mischen der Codebasis ein Dorn im Auge jedes Clean-Code-Entwicklers, es führt auch zur Verwirrung für Neueinsteiger im Projekt und verleitet dazu, veraltete Muster selbst in neuem Code weiterzuverwenden. Die leichte Integrierbarkeit von Kotlin mit Java sollte daher immer als Argument verstanden werden, die Migration zu beginnen, nicht ihre Vollendung zu verzögern.

## Quellen


[1] Robert C. Martin (2009): Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, USA.

[2] Bill Blunden (2003): Software Exorcism: A Handbook for Debugging and Optimizing Legacy Code. Apress, USA.

[3] Daniel Brolund, Ola Ellnestam (2014): The Mikado Method. Manning Publications, USA.

[4] *https://developer.android.com/kotlin/ktx*

[5] *https://kotlinlang.org/docs/data-classes.html*

[6] *https://kotlinlang.org/lp/mobile/*

**Lars Adler**

tarent solutions GmbH

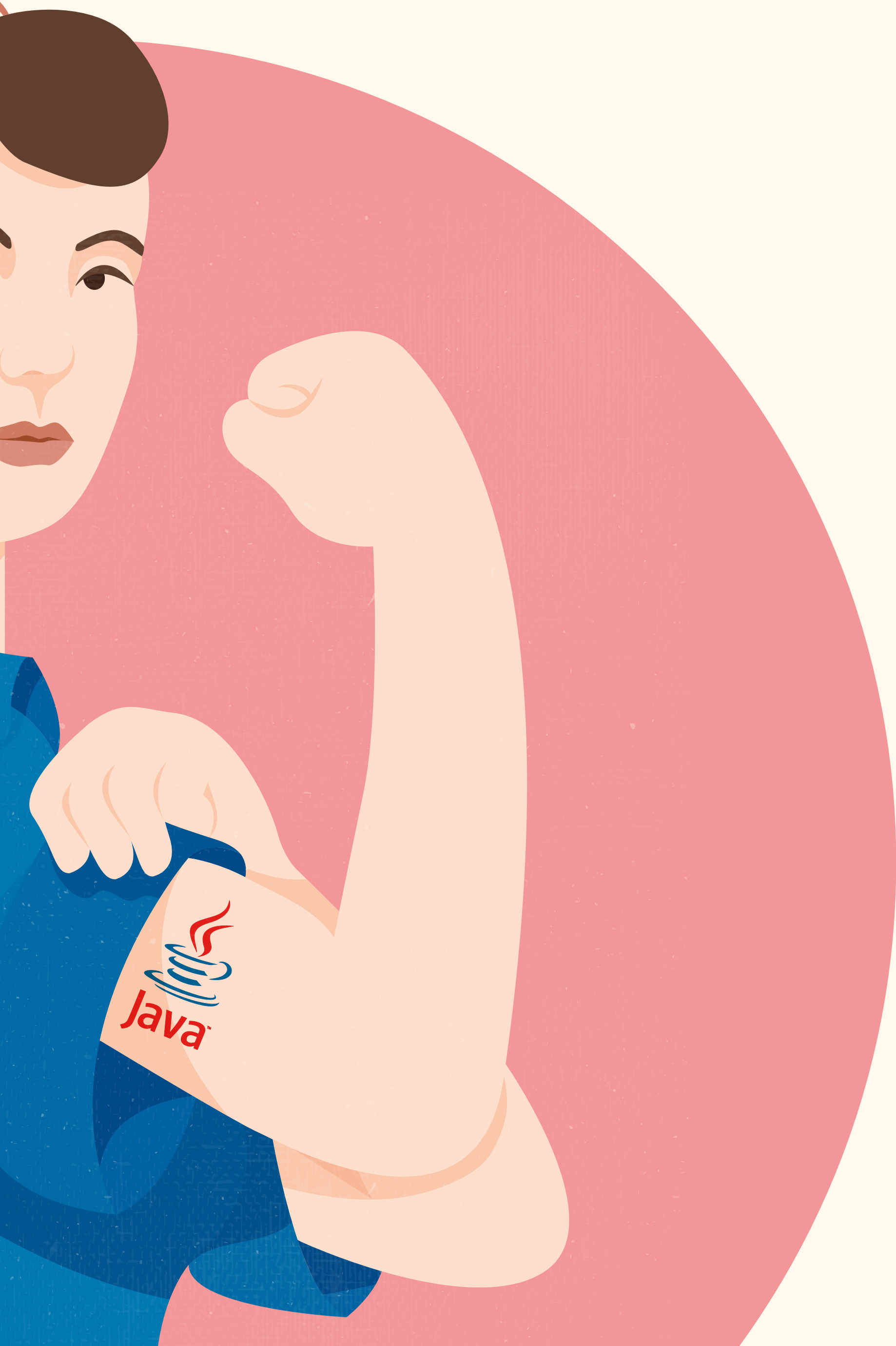
*l.adler@tarent.de*


Lars Adler, geboren 1971 und aufgewachsen in Köln, lernte Programmieren auf seinem ersten 8-Bit-Heimcomputer im Jahre 1985. Nach einem Studium in Physikalischer Chemie mit abschließendem Diplom ist er seit 2001 in vielfältigen Projekten in der Software-Entwicklung tätig, unter anderem mit Schwerpunkten in Java, Android, Kotlin und automatisiertem Testen. Weiterhin sieht er sich noch als Clean Code Developer [1] und Software Exorcist [2].

# Java can do that too! – Data Oriented Programming

**Merlin Bögershausen, adesso SE**

*Java 17 noch nicht richtig verdaut und schon kommt mit Java 21 ein neues LTS-Release auf uns zu und bringt wieder viele Neuerungen mit sich. Der Teilbereich von Projekt Amber ist dabei besonders interessant, denn unter anderem hier wird beeinflusst, wie sich die Sprache anfühlt und was sie alles kann. Sehen wir uns anhand eines Beispiels an, was sich rund um Sealed und Record Classes getan hat.*




AVA

DO IT!

Project Amber hat das Ziel, kleine Änderungen an der Sprache Java zu erforschen, die die Produktivität von Java-Entwickler/-innen verbessern. Von vielen Entwickler/innen wird die funktionale Programmierung (FP), nachdem sie gemeistert wurde, als hochproduktiv beschrieben. Sicher auch aus diesem Grund orientieren sich viele der neuen Sprachfeatures an der funktionalen Programmierung. Niemand muss jetzt Angst haben, Java würde seinen Fokus auf Objektorientierung (OO) verlieren. Die datenorientierte Programmierung (DOP) ermöglicht die Verbindung zwischen OO im Großen und FP im Kleinen. In diesem Artikel sollen die neuen Sprachfeatures mit DOP in Aktion gebracht und eine sehr vereinfachte Abrechnungskomponente implementiert werden.

## Das Beispiel der Rechnungsstellung

Es wurde ein relativ simples Exempel gewählt, das gleichzeitig Raum für die Umsetzung eigener Ideen bietet und einlädt, mit der Codebase auf GitHub *[1]* zu experimentieren.

Doch nun zum Inhalt. In dem vorliegenden Beispiel wird ein kleines Modul zur Rechnungserstellung für Unternehmen implementiert, auch wenn es in der Realität oft klüger wäre, so etwas zu kaufen. In *Abbildung 1* sieht man ein einfaches Klassendiagramm, das fast alle notwendigen Informationen enthält. Kern der Domain des Moduls ist die *Rechnung*. Diese kann in zwei Ausprägungen vorkommen. Um zwischen zwei Abteilungen innerhalb eines Unternehmens etwas zu verrechnen, kann die *InterneVerrechnung* verwendet werden. Hier fallen in der Regel keine Steuern an und es genügt, eine Rechnung in den Büchern abzulegen. Um externen Kunden eine Rechnung zu senden, kann die Rechnungsart *ExternVersandt* verwendet werden. Hierbei fällt je nach Kunde Mehrwertsteuer (MwSt.) an und es muss immer eine Rechnung versandt werden. Der Einfachheit halber werden in vorliegendem Beispiel nur E-Mail-Rechnungen versendet und ein Steuersatz von 10 % angenommen.

Der zweite zentrale Baustein der Domain sind die Kunden und deren Ausprägungen *Privatkunde* und *Businesskunde*. Für Rechnungen an Privatkunden muss immer MwSt. berechnet werden, bei Businesskunden ist es abhängig von ihrer Berechtigung zum Vorsteuerabzug. Ist ein Businesskunde vorsteuerabzugsberechtigt, so muss keine MwSt. berechnet werden, ist das Recht nicht vorhanden, ist der volle Satz anzuwenden.

Das Beispiel stellt die Realität nur sehr vereinfacht dar und würde sicher in einer realen Buchhaltungssoftware anders umgesetzt werden. In den folgenden Absätzen wird es verwendet, um möglichst viele der neuen Features von Java und die Grundsätze von datenorientierter Programmierung zu zeigen.

## Domain-Modellierung algebraischer Datentypen

Wie das Verb „datenorientiert" in DOP nahelegt, kommt der Modellierung der Daten beziehungsweise der Domain ein besonders hoher Stellenwert zu. Ähnlich wie auch bei Modellierungen in der funktionalen Programmierung wird die Domain mit allen Ausprägungen realisiert. Dabei steht die Frage „Gibt es für jede Ausprägung eine konkrete Modellierung?" im Vordergrund.

Mit Record Classes JEP-395 *[2]* und Sealed Classes JEP-409 *[3]* wurden mit Java 16 beziehungsweise 17 zwei neue Arten von Klas-

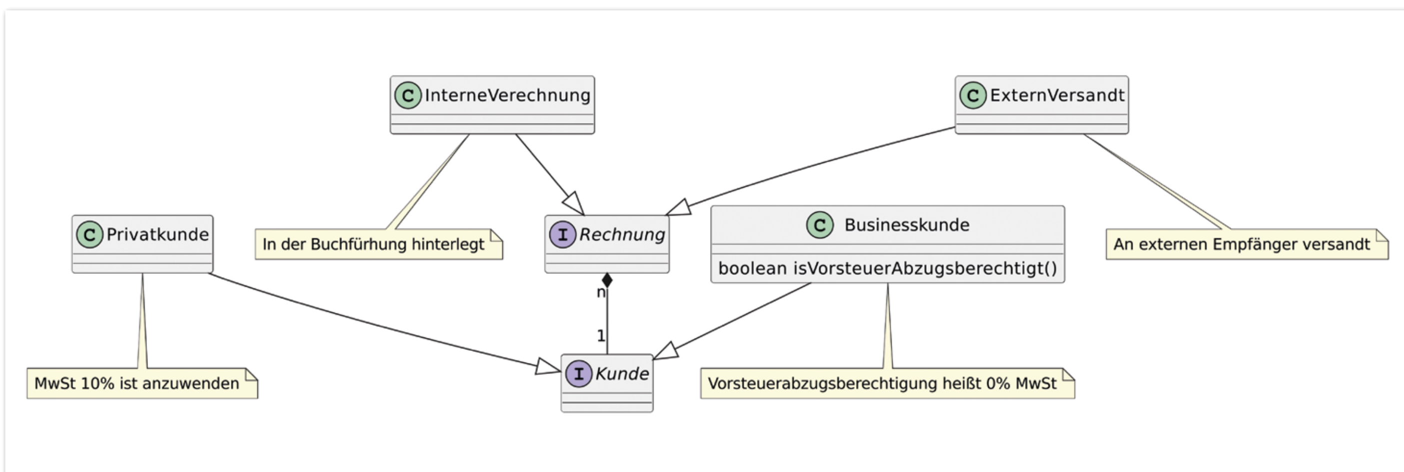



*Abbildung 1: Klassendiagramm zum Modul Rechnungsstellung (© Merlin Bögershausen)*

```
/* Rechnung.java */
sealed interface Rechnung permits InterneVerechnung, ExternVersandt { }
record InterneVerechnung(String abteilung, double wert) implements Rechnung { }
record ExternVersandt(Kunde kunde, double wert) implements Rechnung { }

/* Kunde.java */
sealed interface Kunde { String name(); String mail();}
record Privatkunde(String name, String mail) implements Kunde { }
record Businesskunde(String name, String mail, boolean isVorsteuerAbzugsberechtigt) implements Kunde { }
```

*Listing 1: Modellierung der Domain des Rechnungsmoduls mit Record und Sealed Classes*

```
/** berechne zu zahlende MwSt */
public double calculateMwSt(Kunde kunde, double wert) {
     if (kunde instanceof Privatkunde) {
        return calculateMwSt((Privatkunde) kunde, wert);
     } else if (kunde instanceof Businesskunde) {
          return calculateMwSt((Businesskunde) kunde, wert);
     } else {
          throw new IllegalArgumentException("Typ %s nicht implementiert".formatted(kunde));
     }
}

/** immer 10% des Kaufwertes für Privatkunden */
private double calculateMwSt(Privatkunde p, double wert) {
     return wert * 0.1d;
}

/** 0% wenn Vorsteuerabzugsberechtigt, sonst 10% */
private static calculateMwSt(Businesskunde b, double wert) {
     return b.isVorsteuerAbzugsberechtigt() ? 0d : wert * 0.1d;
}
```

*Listing 2: MwSt.-Berechnung mit Java-Sprachversion unter 13*

sen in die Sprache Java eingeführt. Mit ihnen ist eine Domainmodellierung möglich, die den Anforderungen von DOP genügt. In *Listing 1* habe ich die Domain des Rechnungsmoduls sehr kompakt mit Sealed und Record Classes umgesetzt. In Zeile 2 wird das Interface *Rechnung* definiert und mit dem `sealed`-Schlüsselwort wird signalisiert, dass es nur vorher bestimmte Ausprägungen geben kann. Die möglichen Ausprägungen sind im weiteren Verlauf explizit hinter dem `permits`-Schlüsselwert als *InterneVerrechnung* und *ExternVersandt* angegeben. Die beiden Rechnungsarten sind als transparente Datenklassen mit Records umgesetzt und implementieren das Interface *Rechnung*.

Um zu signalisieren, dass es sich um Records handelt, wird das Schlüsselwort `record` gefolgt vom Namen der neuen Klasse genutzt. Dem Namen folgt etwas, dass nach der Definition eines Konstruktors aussieht. Hier werden die Komponenten des Records mit ihren Datentypen aufgezählt. Zusätzliche private Instanzvariablen darf es nicht geben, statische Felder schon. Durch diese Einschränkung wird versteckter Datenbestand verhindert, der andernfalls zu Intransparenz führen würde. Die Klassen sind implizit final und bekommen `equals, hashCode, toString` sowie einen Konstruktor entsprechend der definierten Komponenten generiert. In der Implementierung der Kunden bedient man sich zweier kleiner Tricks. Zum einen wird der `permits`-Teil der `definition` des Sealed Interface *Kunde* weggelassen. Da die beiden implementierenden Klassen *Privatkunde* und *Businesskunde* in derselben Compilation-Unit liegen, ist dies möglich, im Allgemeinen eine Java-Datei (vgl. JLS 7.3 *[4]*). Der zweite Trick ist die Wahl der Methodennamen, um an den Namen und die Mailadresse des Kunden zu kommen. Hier wird auf das `get`-Präfix verzichtet. Damit decken sich die Namen der generierten Accessoren in den Records mit denen der im Interface geforderten und man muss weniger Code schreiben und warten.

## Implementierung der Rechnungslogik

Um eine Rechnung zu erstellen, wird die Höhe der Mehrwertsteuer benötigt. Sie hängt ab von der Art des Kunden:

- Privatkunden zahlen 10 % MwSt.
- Businesskunden zahlen ebenfalls 10 % MwSt.
- Mit Vorsteuerabzugsberechtigung zahlen Businesskunden 0 % MwSt.

Um also die Höhe korrekt zu berechnen, muss nach der Art des Kunden unterschieden werden, unter Umständen ein Feld aus der Klasse Businesskunden ausgelesen und mit dem Rechnungswert multipliziert werden.

In einem rein objektorientieren Ansatz könnte dies über eine Fassade mit polymorphen Implementierungen realisiert werden. In *Listing 2* wird die Berechnung der MwSt. in zwei Methoden realisiert. Durch die unterschiedlichen Typen des ersten Parameters in Zeile 7 und 12 wird die Unterscheidung nach der Art des Kunden realisiert. Dieser klassische und vollkommen valide Weg birgt zwei große Nachteile:

1. Die Logik verteil sich über mehrere Methoden. Dadurch wird es schwieriger, diese zu überblicken und verstehen. Des Weiteren ist es schwierig, bei Anpassungen alle Stellen zu finden.
2. Type Check mit nachgelagertem Type Cast. Obwohl sie immer als Double auftreten, sind sie nicht syntaktisch aneinander gebunden. Nicht selten führt dies bei Weiterentwicklungen zu unerwarteten Verhalten, weil der Cast oder der Check nicht angepasst wurde.

Die wenig elegante Behandlung von nicht implementieren Fällen ist Gegenstand eines späteren Absatzes.

Die verteilte Logik kann man sehr einfach auflösen, indem man das Instanceof-Pattern nutzt und eine zusammenhängende `if-else`-Kaskade aufbaut. Das Instanceof-Pattern ist mit JEP 394 *[5]* eingeführt worden und seit Java 16 fester Bestandteil der Sprache. Es bildet das erste und grundlegendste Pattern der Sprache Java. In *Listing 3* ist die Berechnung der MwSt. entsprechend umgebaut. Im `if` in Zeile 2 wird der altgediente Instanceof-Operator verwendet,

```java
public double calculateMwSt(Kunde kunde, double wert) {
     if (kunde instanceof Privatkunde) { // immer 10% des Kaufwertes
           return wert * 0.1d;
     } else if (kunde instanceof Businesskunde b) { // abhängig von dem Vorsteuerabzug
           if (b.isVorsteuerAbzugsberechtigt()) return 0.0d;
           else return wert * 0.1d;
     } else { // behandle fehlende Implementierung
           throw new IllegalArgumentException(„Typ %s nicht implementiert“.formatted(kunde));
     }
}
```

Listing 3: MwSt.-Berechnung mit Instanceof-Pattern

um Privatkunden zu verarbeiten, und in Zeile 4 das Instanceof-Pattern angewendet. In einem Schritt wird geprüft, ob die Instanz ein Businesskunde ist und wenn ja, direkt gecastet. Der Unterschied zwischen Zeile 2 und Zeile 4 liegt nur in der Angabe einer Zielvariable für den Typecast. Die neue Variable b ist vom Typ Businesskunde und im kompletten `if`-Zweig verfügbar.

Hier könnte man nun natürlich den „Codesmell"-Alarm auslösen – berechtigt. In dem Beispiel wird `instanceof` in einer `if-else`-Kaskade verwendet, die bei jeder Erweiterung neu betrachtet werden muss. Zumindest, wenn Exceptions vermieden werden sollen. Durch die Implementierung der Kunden als Sealed Interface weiß der Compiler ganz genau, welche möglichen Ausprägungen der Klasse Kunde es gibt. Leider ist es nicht möglich, dieses Wissen auf `if-else`-Kaskaden anzuwenden.

Möglich ist die sogenannte Exhaustiveness-Analyse mit der durch JEP 361 *[6]* eingeführten und in Java 14 finalisierten Switch Expression in Verbindung mit Pattern Matching aus JEP 433 *[7]*. In *Listing 4* wird die `if-else`-Kaskade durch eine Switch Expression mit Typ-Patterns ersetzt. In Zeile 3 wird ein einfacher Match auf den Typ Privatkunde durchgeführt und da man nicht auf die Instance zugreift, kann man den doppelten Unterstrich verwenden, um ein Ignorieren anzudeuten. Der Pfeil, den man bereits von Lambdas kennt, zeichnet das Switch als eine Expression aus. Das gefürchtete Fallthrough-Verhalten des Switch ist damit deaktiviert und das Angeben eines Ergebnisses der Expression ist möglich.

Im Fall des Businesskunden (Zeile 4) kommt man nicht mit einer Zeile aus, deshalb wird analog zu Lambdas ein Block verwendet, in dem die Unterscheidung entsprechend der Vorsteuer durchgeführt wird. In diesem Fall muss das Ergebnis explizit mit dem Schlüsselwort *yield* gekennzeichnet werden. Einen `default`-Fall benötigt man hier nicht, denn der Compiler kann verifizieren, dass alle möglichen Pfade abgedeckt werden. Würde man hier eine neue Art von Kunde hinzufügen, so kommt es zu einem Kompilierfehler mit dem Hinweis, dass der Switch-Ausdruck nicht alle möglichen Werte abdeckt. Dadurch fällt der Fehler nicht

```java
public double calculateMwSt(Kunde kunde, double wert) {
     return switch (kunde) {
           case Privatkunde __ -> wert * 0.1d;
           case Businesskunde b -> {
                 if (b.isVorsteuerAbzugsberechtigt()) yield 0.0d;
                 else yield wert * 0.1d;
           } // alternativ b.vorsteuerAbzug() ? 0.0d : wert * 0.1d;
     };
}
```

Listing 4: MwSt.-Berechnung Switch Expression und Pattern Matching

```java
public double calculateMwSt(Kunde kunde, double wert) {
     return switch (kunde) {
           case Businesskunde b when b.isVorsteuerAbzugsberechtigt() -> 0.0d;
           case Businesskunde b -> wert * 0.1d;
           case Privatkunde p -> wert * 0.1d;
     };
}
```

Listing5: MwSt.-Berechnung Switch Expression, Pattern Matching und When-Clause

```java
void processRechnung(Rechnung rechnung) {
     switch (rechnung) {
          case InterneVerechnung(var abt, double wert) -> Dummy.storeInDB(abt, wert);
          case ExternVersandt(Kunde kunde, var wert) -> {
               double mwst = MwStRechner.PlainOOP.calculateMwSt(kunde, wert);
               var txt = formatMail(kunde.name(), wert, mwst);
               Dummy.sendViaMail(kunde.mail(), txt);
          }
     }
}
```

*Listing 6: Verarbeitung von Rechnungen mit Switch Expression und Record Deconstruction*

erst bei der Verwendung durch eine *IlligalArgumentException* auf, sondern bereits zur Compile-Zeit.

Bei der Verschachtelung von Kontrollstrukturen läuft es in Fachkreisen vielen kalt den Rücken herunter. Eine solche Vermischung ist im vorliegenden Beispiel in *Listing 4* in Zeile 5 unterlaufen, als die Behandlung der Vorsteuerabzugsberechtigung durch ein `if` realisiert wurde. Viel besser dafür geeignet wäre eine When-Clause, die ebenfalls in JEP 433 *[7]* eingeführt wurde. Damit kann ein boolescher Ausdruck mit einem Pattern kombiniert werden, um eine Fallunterscheidung durchzuführen. In *Listing 5* wurde dann das geschachtelte `if` durch eine When-Clause ausgetauscht, um den Vorsteuerabzug zu behandeln. Hierzu muss nur ein `when` gefolgt von einem booleschen Ausdruck an das Typ-Pattern angehängt werden. Innerhalb der When-Clause sind alle Variablen zugreifbar, die im umgebenden Scope und dem Pattern definiert wurden.

Anschließend soll noch eine Methode implementiert werden, die die `calculateMwSt`-Methode verwendet, um eine Rechnung zu bearbeiten. Hierbei bestimmt die Art der Rechnung, wie sie verarbeitet wird. Rechnungen zur internen Verrechnung werden in einer Datenbank gespeichert, vereinfacht angedeutet wird dies durch die Methode `Dummy.storeInDB(String abteilung, double wert)`. Eine externe Rechnung wird mit der Methode `Dummy.sendViaMail(String mail, String text)` an die angegebene Mailadresse versandt. Gekapselt ist das Verhalten in der Methode `processRechnung(Rechnung rechnung)` aus *Listing 6*. Hier wird wieder eine Switch Expression verwendet, um Nutzen aus der Sealed-Klassenhierarchie der Rechnung zu ziehen. In der Expression verwendet man aber nicht das Typ-Pattern, sondern die Record Patterns aus JEP 432. Diese Operation wird auch Dekonstruktion genannt und man wendet sie an, wenn man auf die enthaltenen Daten zugreifen möchte. Die Dekonstruktion unterscheidet sich syntaktisch nur durch das Fehlen des `new`-Schlüsselwortes von der Erzeugung und stellt semantisch das genaue Gegenteil dar. Die Datentypen der Komponenten können explizit angegeben oder durch die Local-Variable-Type-Inferenz bestimmt werden. Anders als bei Lambdas ist das Weglassen einer Typdefinition beziehungsweise des `var`-Keywords nicht erlaubt.

Als letztes Puzzleteil fehlt noch die Formatierung der Rechnung als Text. Der Text einer Rechnung ist immer gleich und unterscheidet sich lediglich in der Ansprache und den Werten in der Rechnung. Die in *Listing 6* verwendete Methode `formatMail(String name, double wert, double mwst)` wird in *Listing 7* mit einem Textblock realisiert. Bei einem Textblock handelt es sich um einen mehrzeiligen String, der keine besonderen Steuerzeichen für Zeilenumbrüche und Einrückungen benötigt. Der Textblock wurde mit JEP 378 *[9]* eingeführt und in Java 15 finalisiert. Er kann mit dem String Formatter kombiniert werden, um wie in Zeile 9 des Listings 7 die notwendigen Werte einzufügen und zu formatieren.

## Zusammenfassung

Mit diesem datenorientierten Ansatz sollte gezeigt werden, wie eine einfache Rechnungserstellung implementierbar ist. Dabei


Bring your IntelliJ IDEA inspections to any CI pipeline — and a whole lot more

QD IJ

```
String formatMail(String name, double wert, double mwst) {
     return """
                   Hallo %s,
                   Bitte senden Sie mir den Rechnungsbetrag in Höhe von %.2f€ plus %.2f€ MwSt.

                   Mit freundlichen Grüßen
                      Merlin Bögershausen
                  """.formatted(name, wert, mwst);
}
```

Listing 7: Formatierung des E-Mail-Texts mit einem Textblock

wurden viele der von Projekt Amber entwickelten Sprachfeatures angewandt. Die beiden Preview Features Pattern Matching for Switch und Record Patterns sollen mit der nächsten LTS-Version 21 finalisiert werden [10] und sind danach permanente Features der Sprache Java. Mit ihnen ist die Anzahl an Zeilen deutlich kleiner als bei einem herkömmlichen OOP-Ansatz. Gleichzeitig steht verwandte Logik nah beieinander und ist dank der Unterstützung des Compilers einfacher erweiterbar. Die genutzten Konstrukte verkürzen und vereinfachen den Code dabei deutlich. Java-Entwickler/-innen werden in Zukunft sehr viel Konzepte der datenorientierten beziehungsweise funktionalen Programmierung im Kleinen mit objektorientierter Programmierung im Großen kombinieren. Für eine Diskussion über die aufgezeigten Ideen oder Fragen steht der Autor gern zur Verfügung – virtuell oder persönlich auf Konferenzen und JUG-Meetups.

## Quellen


[1] GitHub/Mboegers/DataOrientedJava *https://github.com/MBoegers/DataOrientedJava*
[2] JEP-395 *https://openjdk.org/jeps/395*
[3] JEP-409 *https://openjdk.org/jeps/409*
[4] JLS 7.3 Compilation Unit *https://docs.oracle.com/javase/specs/jls/se17/html/jls-7.html#jls-7.3*
[5] JEP 394: Pattern Matching for instanceof *https://openjdk.org/jeps/394*
[6] JEP 361: Switch Expressions *https://openjdk.org/jeps/361*
[7] JEP 433: Pattern Matching for switch (Fourth Preview) *https://openjdk.org/jeps/433*
[8] JEP 432: Record Patterns (Second Preview) *https://openjdk.org/jeps/432*
[9] JEP 378: Text Blocks *https://openjdk.org/jeps/378*
[10] G. Bierman in Amber-dev Mailingliste *https://mail.openjdk.org/pipermail/amber-dev/2023-February/007841.html*

**Merlin Bögershausen**
Adesso SE
*Merlin.boegershausen@rwth-aachen.de*


Merlin ist Vater, Segelfluglehrer, Volleyballsteller und Softwareengineer aus Aachen mit über 10 Jahren Erfahrung im Java-Bereich. Er schreibt Artikel und tritt auf Konferenzen oder JUG-Meetings als Speaker auf, wo er über seine Begeisterung für modernen Java-Code und einfache Lösungen spricht.

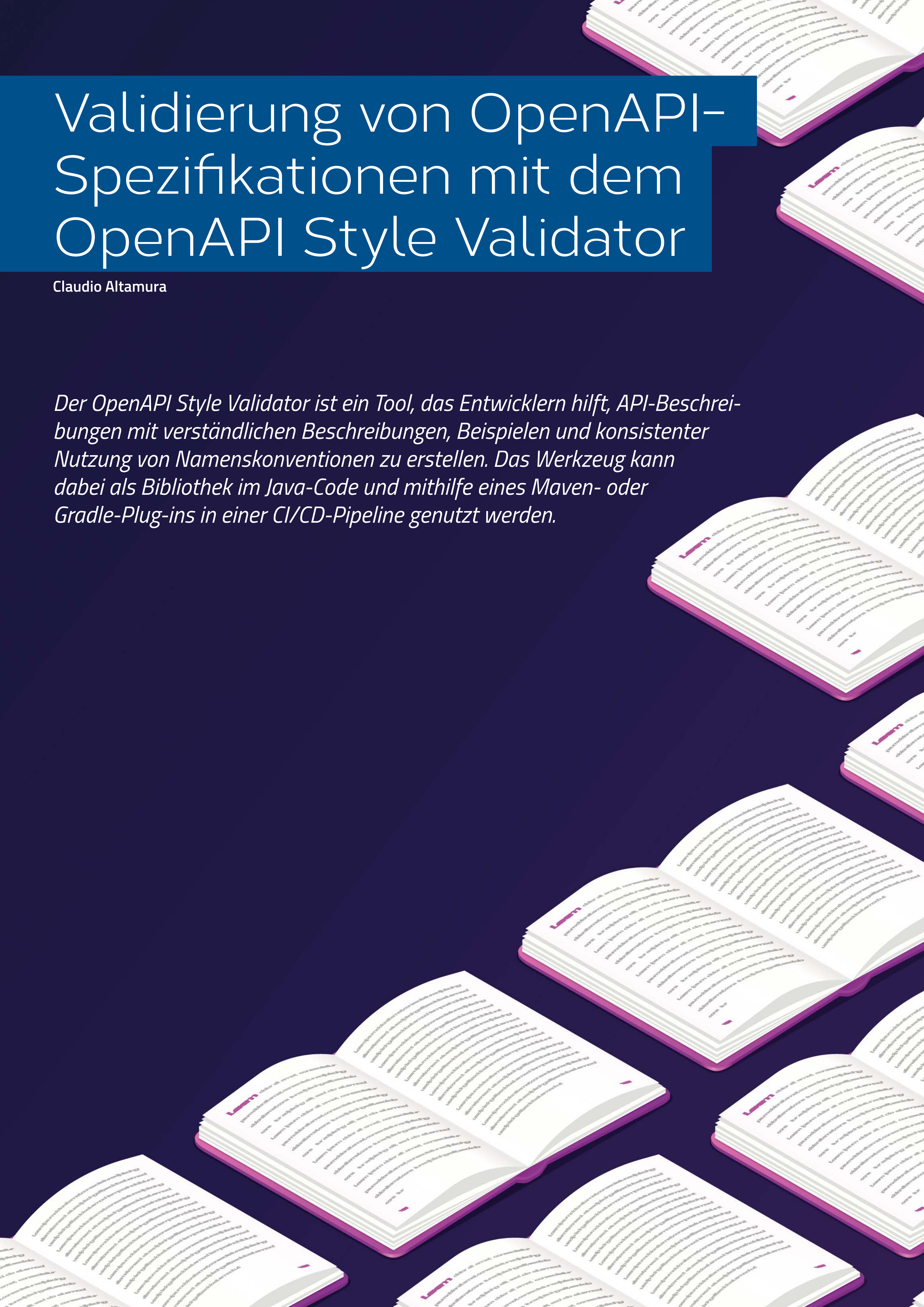
FÜR 29,00 €
BESTELLEN

Java aktuell

JAHRESABO

Mehr Informationen zum Magazin und Abo unter:

www.ijug.eu/de/java-aktuell



# Validierung von OpenAPI-Spezifikationen mit dem OpenAPI Style Validator

Claudio Altamura

*Der OpenAPI Style Validator ist ein Tool, das Entwicklern hilft, API-Beschreibungen mit verständlichen Beschreibungen, Beispielen und konsistenter Nutzung von Namenskonventionen zu erstellen. Das Werkzeug kann dabei als Bibliothek im Java-Code und mithilfe eines Maven- oder Gradle-Plug-ins in einer CI/CD-Pipeline genutzt werden.*

Sowohl bei einem API-First- als auch bei einem Code-First-Ansatz hilft der OpenAPI Style Validator *[1]* Entwicklern, Schwächen in einer OpenAPI-Spezifikation zu identifizieren. Damit kann der Validator beim Umsetzen und Durchsetzen von Richtlinien genutzt werden. Mit definierten Regeln kann genau beschrieben werden, wie eine API-Spezifikation auszusehen hat. Im Gegensatz zu einer Dokumentation in einem Wiki, die vielleicht nicht jeder Entwickler verinnerlicht hat, können mit dem Tool automatisiert Spezifikationen geprüft werden. Diese Informationen können in Code Reviews genutzt werden oder in einer Build-Pipeline, in der es bei Regelverstößen zum Build Break kommt.

## Vollständige Beschreibungen und Namenskonventionen

Was kann mit dem OpenAPI Style Validator geprüft werden? Für die nähere Erläuterung und dazu, was der Validator leisten kann, ziehen wir das beliebte Petstore-Beispiel *[2]* heran. Der Validator prüft verschiedene Objekte des OpenAPI-Schemas, angefangen mit dem Info-Objekt und den dazugehörigen Contact- und License-Objekt *(siehe Listing 1)*. Oftmals werden diese Informationen zusammen mit dem Titel und der Beschreibung vernachlässigt oder es wird gar nicht erst angegeben, was als Regelverstoß definiert werden kann.

Das nächste Objekt, das wir uns näher ansehen, ist das Operation-Objekt. Doch fangen wir zuerst mit dem Paths-Objekt an. Das Paths-Objekt beinhaltet, wie der Name schon aussagt, alle Pfade zu den existieren Endpunkten (Path Items). Ein einzelner Pfad (/pets) enthält Operationen, die näher beschreiben, was für http-Methoden erlaubt sind. Im Beispiel in *Listing 2* ist das GET.

Der OpenAPI Style Validator erkennt, ob bestimmte Properties im Operation-Objekt existieren und gefüllt sind. Beispielsweise fehlt im *Listing 2* das Property „summary". Dagegen ist eine „description" vorhanden. Die Abwesenheit eines Property wird als Fehler eingestuft, wenn der Validator entsprechend konfiguriert ist, aber dazu kommen wir später.

Schauen wir uns als Nächstes an, wie der OpenAPI Style Validator die Beschreibung von Datentypen überprüft. Datentypen werden bei der OpenAPI-Spezifikation als Schema-Objekte definiert, die in Requests oder Responses referenziert werden können (etwa

```
{
"info": {
     "version": "1.0.0",
     "title": "Swagger Petstore",
     "description": "A sample API that uses a petstore as an example to demonstrate features in the OpenAPI 3.0
specification",
     "termsOfService": "http://swagger.io/terms/",
     "contact": {
        "name": "Swagger API Team",
        "email": "apiteam@swagger.io",
        "url": "http://swagger.io"
     },
     "license": {
        "name": "Apache 2.0",
        "url": "https://www.apache.org/licenses/LICENSE-2.0.html"
     }
  }
…
}
```

*Listing 1: Das Info-Objekt*

```
{
…
"/pets/{id}": {
     "get": {
        "description": "Returns a user based on a single ID, if the user does not have access to the pet",
        "operationId": "find pet by id",
        "parameters": [
           {
              "name": "id",
              "in": "path",
              "description": "ID of pet to fetch",
              "required": true,
              "schema": {
                 "type": "integer",
                 "format": "int64"
              }
           }
        ],
      …
     }
   }
…
}
```

*Listing 2: Das Operation-Objekt*

$ref": "#/components/schemas/NewPet"). Das Werkzeug kann prüfen, ob für alle Schema-Properties Beschreibungen (description) und Beispiele (example) vorhanden und nicht leer sind.

So finden wir, wenn wir uns aus dem Petstore-Beispiel das NewPet-Schema-Objekt im *Listing 3* anschauen, keine Beschreibungen und Beispiele. Die Prüfung, ob `example` und `description` existieren, hat den Vorteil, dass API-Dokumentationen durch die Beschreibungen und Beispiele verständlicher werden. Zusätzlich können aus den Beispielen und Beschreibungen durch weitere Tools realistischere Testdaten generiert werden.

```
{
…
"NewPet": {
     "type": "object",
     "required": [
        "name"
     ],
     "properties": {
        "name": {
           "type": "string"
        },
        "tag": {
           "type": "string"
        }
     }
  },
…
}
```

*Listing 3: Das Schema-Objekt NewPet*

Kommen wir nun zu den Namenskonventionen. Namenskonventionen helfen dabei, dass ein API einfacher zu verstehen und zu verwenden ist. Der OpenAPI Style Validator unterstützt dabei eine Reihe von verschiedenen Konventionen, die wir für Pfade, Parameter (Path Parameter, Query Parameter und Cookies) sowie Header und Properties anwenden können. Dazu gehören:

- der Underscore Case (snake_case),
- Camel Case, wie wir es von Java und JavaScript kennen
- und natürlich der sogenannte Hyphen-Case, auch Kebab-Case genannt.

## Optionen und Aufruf des OpenAPI Style Validator

Mit den definierten Regeln, die wir jetzt kenngelernt haben, können wir steuern, wie eine OpenAPI-Spezifikation auszusehen hat. Welche Einstellungsmöglichkeiten haben wir? Es gehören boolesche Optionen dazu, wie etwa `validateOperationOperationId`, die durch ein `true` verlangt, dass jede Operation eine Id besitzt. Die Option `validateOperationSummary` fordert ein, dass Operationen auch eine Beschreibung haben. Aber es sind auch Optionen vom Typ String enthalten, wie

- pathNamingConvention,
- parameterNamingConvention,
- pathParamNamingConvention,
- queryParamNamingConvention,

auf die ich jetzt näher eingehen möchte. Mit diesen Optionen können wir für Elemente bestimmen, beispielsweise, ob sie der Underscore-Case- oder Camel-Case-Namenskonvention folgen sollen.

An dieser Stelle möchte ich ein paar reale Beispiele nennen, in denen unterschiedliche Namenskonventionen ihre Anwendung finden. In den Zalando Guidelines *[3]* können wir nachlesen, dass für URIs der Hyphen-Case und für query parameter der Underscore-Case angewendet werden soll (zum Beispiel /article-size-advices?skus=sku-1,sku-2). Das macht Adidas ein wenig anders, um hier ein weiteres Beispiel zu nennen *[4]*. Adidas verwendet für URIs ebenfalls den Hyphen-Case, erlaubt aber sonst den Camel-Case (zum Beispiel /system-orders/{orderId}). Zalando nutzt des Weiteren für Enums den UPPERCASE_SNAKE_CASE (etwa INCLUDED) und Adidas für http-Header den Hyphen-Uppercase (Beispiel: Order-Metadata-Header: 42).

Unter *[5]* sind alle Optionen genannt. Aber kommen wir nun nach der ganzen Theorie dazu, wie der Validator letztendlich genutzt wird.

Das Projekt besteht aus einer Bibliothek, einer Befehlszeilenschnittstelle (CLI) sowie einem Plug-in für Maven und Gradle. Schauen wir

```
…
<plugin>
<groupId>org.openapitools.openapistylevalidator</groupId>
<artifactId>openapi-style-validator-maven-plugin</artifactId>
<version>1.8</version>
<configuration>
     <inputFile>petstore-expanded.json</inputFile>
</configuration>
<dependencies>
     <dependency>
           <groupId>org.openapitools.empoa</groupId>
           <artifactId>empoa-swagger-core</artifactId>
           <version>2.0.0</version>
           <exclusions>
                 <exclusion>
                       <groupId>io.swagger.core.v3</groupId>
                       <artifactId>swagger-models</artifactId>
                 </exclusion>
           </exclusions>
     </dependency>
</dependencies>
</plugin>
…
```

*Listing 4: Konfiguration des Maven-Plug-ins in der pom.xml*

```
[INFO] --- openapi-style-validator:1.8:validate (default-cli) @ openapitools-validator-mvn-example ---
[INFO] Validating spec: petstore-expanded.json
[ERROR] OpenAPI Specification does not meet the requirements. Issues:

[ERROR]         *ERROR* in Operation GET /pets 'summary' -> This field should be present and not empty
[ERROR]         *ERROR* in Operation GET /pets 'tags' -> The collection should be present and there should be at least one item in it
[ERROR]         *ERROR* in Operation POST /pets 'summary' -> This field should be present and not empty
[ERROR]         *ERROR* in Operation POST /pets 'tags' -> The collection should be present and there should be at least one item in it
[ERROR]         *ERROR* in Operation GET /pets/{id} 'summary' -> This field should be present and not empty
[ERROR]         *ERROR* in Operation GET /pets/{id} 'tags' -> The collection should be present and there should be at least one item in it
[ERROR]         *ERROR* in Operation DELETE /pets/{id} 'summary' -> This field should be present and not empty
[ERROR]         *ERROR* in Operation DELETE /pets/{id} 'tags' -> The collection should be present and there should be at least one item in it
[ERROR]         *ERROR* in Model 'NewPet', property 'name', field 'example' -> This field should be present and not empty
[ERROR]         *ERROR* in Model 'NewPet', property 'name', field 'description' -> This field should be present and not empty
[ERROR]         *ERROR* in Model 'NewPet', property 'tag', field 'example' -> This field should be present and not empty
[ERROR]         *ERROR* in Model 'NewPet', property 'tag', field 'description' -> This field should be present and not empty
[ERROR]         *ERROR* in Model 'Error', property 'code', field 'example' -> This field should be present and not empty
[ERROR]         *ERROR* in Model 'Error', property 'code', field 'description' -> This field should be present and not empty
[ERROR]         *ERROR* in Model 'Error', property 'message', field 'example' -> This field should be present and not empty
[ERROR]         *ERROR* in Model 'Error', property 'message', field 'description' -> This field should be present and not empty
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
```

*Abbildung 1: Prüfung des Beispiel-OpenAPI und Abbruch aufgrund von Verstößen*

uns an, wie wir mit der CLI eine OpenAPI-Spezifikation prüfen können. Folgender Befehl startet den Validator:

```
java -jar openapi-style-validator.jar -s ./
petstore-expanded.json -o ./options.json
```

Der Aufruf ist nahezu selbsterklärend. Es wird ein Jar mit den Optionen -s (source) für die zu prüfende Spezifikation und -o für eine Optionsdatei aufgerufen, in der alle möglichen Optionen eingestellt werden können. Wird keine Datei mit Optionen angegeben, dann werden die Default-Einstellungen genommen. Es werden alle Fehler aufgelistet, die gegen die konfigurierten Optionen verstoßen.

Wie sieht der Aufruf per Maven aus? Für Maven muss ein Plug-in innerhalb von <build> konfiguriert werden *(siehe Listing 4)*.

Zurzeit muss für das Maven-Plug-in die Abhängigkeit `io.swagger.core.v3` exkludiert werden, da sonst eine neuere Version der Library genutzt wird, die inkompatibel zur Version 1.8 des OpenAPI Style Validator ist. Beim Maven-Plug-in kann jede Option als Parameter unter <configuration> als XML-Tag hinzugefügt werden, beispielsweise `validateOperationSummary` mit `true` oder `false` als Text-Content.

Fügt man die Exclusion hinzu, kann der Validator erfolgreich, beispielsweise mit folgendem Befehl, ausgeführt werden:

```
mvn openapi-style-validator:validate
```

Mit der Standardkonfiguration erhalten wir für das Petstore-Beispiel folgendes Prüfergebnis, das aufgrund von Regelverstößen zum Build Break führt *(siehe Abbildung 1)*.

## Fazit

Der OpenAPI Style Validator hilft Entwicklern, unvollständige Beschreibungen und inkonsistente Nutzung von Namenskonventionen zu finden. Der Validator kann genutzt werden, um API-Richtlinien umzusetzen. Solche Richtlinien sollen die Zusammenarbeit, die Stabilität und die Erweiterung von APIs gewährleisten. Denn ein API soll einfach zu verstehen und zu erlernen sein. Es muss einfach zu benutzen sein sowie ein einheitliches „Look and Feel" bieten. Genau hierbei hilft das Tool. Mit definierten Regeln kann genau beschrieben werden, wie eine API-Spezifikation auszusehen hat.

## Quellen


[1] OpenAPI Style Validator *https://github.com/OpenAPITools/openapi-style-validator*

[2] Petstore-Beispiel *https://github.com/OAI/OpenAPI-Specification/blob/main/examples/v3.0/petstore-expanded.json*

[3] OpenAPI Style Validator Options *https://github.com/OpenAPITools/openapi-style-validator#options-file*

[3] Zalando RESTFUL API Guidelines *https://opensource.zalando.com/restful-api-guidelines/*

[4] Adidas API Guidelines *https://adidas.gitbook.io/api-guidelines/rest-api-guidelines/evolution/naming-conventions#general-naming-rules*

[5] OpenAPI Style Validator Maven Plugin *https://central.sonatype.com/artifact/org.openapitools.openapistylevalidator/openapi-style-validator-maven-plugin/1.8*

**Claudio Altamura**

*me@claudioaltamura.de*


Claudio Altamura ist Softwareentwickler und Softwarearchitekt. Er unterstützt Kunden bei Design und Umsetzung von Applikationen mit Microservice-Architekturen und natürlich APIs in der Cloud. Claudio ist regelmäßiger Sprecher und Autor von verschiedenen Fach- und Blog-Artikeln.

# DEINE VORTEILE

**30 % Rabatt**
auf JavaLand-Tickets

**Java aktuell**
Jahres-Abonnement

**Java Community Process**
Mitgliedschaft

## JETZT MITGLIED WERDEN!

Ab 15 Euro im Jahr

www.ijug.eu

iJUG Verbund

# Remote Mob Programming – zuhause, aber nicht allein

Joshua Töpfer, INNOQ

*Das ganze Team sitzt in einem Online-Meeting und entwickelt gemeinsam. Einer tippt den Code, die anderen diskutieren. Klingt ungewöhnlich? Das ist Remote Mob Programming, eine spannende Arbeitsweise für verteilte Teams. Was es genau damit auf sich hat und welche Vor- und Nachteile es gibt, erfahren Sie in diesem Artikel.*

Remote Mob Programming *[1]* ist eine Methode der Softwareentwicklung, die das Konzept des Mob Programming mit dem der verteilten Teams kombiniert. Durch die Verbreitung von COVID-19 haben verteilte Teams in Deutschland und weltweit an Bedeutung gewonnen. Woody Zuill beschreibt Mob Programming oder auch Software Teaming wie folgt: „Mob Programming ist eine Softwareentwicklungsmethodik, bei der das gesamte Team zur gleichen Zeit, im gleichen Raum und am gleichen Computer an der gleichen Sache arbeitet." *[2]*

Remote Mob Programming überträgt diese Methode auf verteilte Teams, indem ein virtueller Raum geschaffen wird, in dem sich das Team trifft und gemeinsam programmiert. Remote Mob Programming ist jedoch kein Widerspruch in sich. Es ist eine neue Art der Zusammenarbeit, die die Vorteile beider Methoden vereint.

## Wie funktioniert Mob Programming?

Aber was genau ist Mob Programming? Stellen Sie sich vor, Sie und Ihr Team arbeiten an einer Aufgabe, aber anstatt dass jeder für sich arbeitet, arbeitet das ganze Team zusammen an dieser. Der Computerbildschirm wird an eine Wand projiziert und einer von Ihnen, der sogenannte Typist, sitzt an der Tastatur und tippt ein, was der Rest des Teams diskutiert und entscheidet. Jeder im Rest des Teams hat die Möglichkeit, am Entscheidungsprozess teilzunehmen und das Ergebnis zu beeinflussen. Der Typist ist jedoch bewusst aus der Entscheidungsfindung herausgenommen, da er sonst die Macht hätte zu tippen, was er will, ohne dass es vorher im Team diskutiert wurde. So kann es keine Alleingänge geben und die Mitglieder des Teams müssen sich vorher verständigen und die gewünschte Lösung diskutieren.

## Was ist Remote Mob Programming?

Remote Mob Programming überträgt Mob Programming ins Internet. Ein virtueller Raum, etwa ein Zoom-Meeting, ersetzt den physischen Raum. Das Team trifft sich dort, um die Software gemeinsam weiterzuentwickeln. Der Typist teilt seinen Bildschirm und kann als High-Level-API für die Entwicklungsumgebung gesehen werden. Er nimmt nicht direkt an der Diskussion und Lösungsfindung teil, sondern wartet auf Anweisungen und programmiert nur, wenn er vom Rest des Teams dazu aufgefordert wird. Je mehr Erfahrung der Typist hat, desto abstrakter können diese Anweisungen sein. So kann ein eher unerfahrener Typist genaue Anweisungen bekommen, was er zu tippen hat. Einem erfahrenen Typist würde beispielsweise nur gesagt werden, dass er einen Test schreiben soll, der Feature X mit den Parametern Y testet. Wenn der Typist gerade keine Anweisungen vom Rest des Teams erhält, wird meditativ das Blinken des Cursors betrachtet, da jede Mausbewegung den Rest des Teams von der Diskussion ablenken könnte.

Die Rolle des Typist wird regelmäßig gewechselt, sodass die Person ab diesem Zeitpunkt wieder aktiv an der Lösungsfindung teilnehmen kann. Bei einem Team von vier Personen hat es sich bewährt, alle zehn Minuten den Typist zu wechseln. So ist jeder im Team nach spätestens einer halben Stunde erneut Typist. Dies sorgt dafür, dass die Aufmerksamkeit der Teammitglieder aufrechterhalten wird. Beim Remote Mob Programming gestaltet sich der Wechsel des Typist etwas schwieriger, da nicht einfach die Tastatur an das nächste Teammitglied weitergegeben werden kann. Eine Lösung ist die Übergabe des Quellcodes über Git *[3]*. Damit der Code nicht jedes Mal erst mühsam manuell committet und dann vom nächsten Kollegen ausgecheckt werden muss, gibt es das Kommandozeilenprogramm mob *[4]*. Mob kapselt die Übergabe in einfachen Befehlen. Damit ist ein Handover des Codes in weniger als zehn Sekunden möglich. Um an den regelmäßigen Wechsel zu erinnern, bietet mob auch einen Timer.

## Konstanter Fortschritt und gute Qualität

Das gesamte Team arbeitet an einem Feature, sodass es maximal ein Ticket gibt, das Work in Progress ist. Der Fokus liegt somit immer auf dem wichtigsten Ticket. Um diesen Fokus beizubehalten, versucht Remote Mob Programming, jegliche Wartezeiten zu vermeiden, da diese meist zu Kontextwechseln führen und zusätzliche Zeit für die erneute Einarbeitung kosten. Daher ist es besonders wichtig, dass das Team möglichst unabhängig arbeiten kann. Ein guter Schnitt der Software und Unabhängigkeit von anderen Teams sowie gut vorbereitete Tickets hinsichtlich der Abhängigkeiten zu anderen Teams sind daher unbedingt notwendig.

Oft ist das Team Terminen ausgesetzt, um beispielsweise Abhängigkeiten mit anderen Teams zu koordinieren. Das Team tritt jedoch immer geschlossen auf, auch bei Terminen. Die große Zahl der Teilnehmer an einem Termin kann sich positiv auf die Termine auswirken, da die hohen Kosten des Termins Druck ausüben, diese möglichst effizient zu gestalten. Ist es nicht möglich, Termine effizienter zu gestalten, so kann das Team auch einen einzelnen Botschafter in einen Termin entsenden. Das Team beginnt, auch seine eigenen Termine infrage zu stellen. Das Daily Scrum, beispielsweise, ist ein Termin, der beim Remote Mob Programming nicht benötigt wird, da jeder im Team den aktuellen Stand kennt. Braucht der Product Owner ein Update zu einem Thema, kann er einfach jederzeit in den virtuellen Team-Raum kommen und sich dieses Update abholen.

In einigen Projekten gibt es Architekten, die wichtige architektonische Entscheidungen treffen. Beim Remote Mob Programming übernimmt das Team die Rolle des Architekten, wann immer dies erforderlich ist. Wenn eine Architekturentscheidung getroffen werden muss, erarbeitet das Team mögliche Alternativen, reflektiert die Vor- und Nachteile und dokumentiert diese in einem Architecture Decision Record. Da die Architekturentscheidung von allen getroffen wird, ist es wichtig, als Einzelner auch Entscheidungen mittragen zu können, die nicht die bevorzugte Lösung sind. Es empfiehlt sich daher nicht, einen Konsens im Team zu finden, sondern einen Konsent, also eine Lösung, bei der niemand ein Veto einlegt. Durch die vielen Beteiligten werden in der Regel mehr potenzielle Lösungen evaluiert und auch die Qualität der betrachteten Lösungen steigt.

Auch während der Programmierung werden viele kleine Entscheidungen implizit getroffen. Remote Mob Programming ermöglicht es, diese Entscheidungen explizit zu machen und zu diskutieren. Auch einfache Fehler werden durch die Teammitglieder meist direkt erkannt, während der Typist noch tippt. Dies spart viel Zeit, die sonst für Debugging oder Fehlersuche verloren gegangen wäre. Außerdem ist ein Code Review in der bekannten Form nicht mehr notwendig, da alle bereits an der Entwicklung beteiligt waren und ihre Meinung einbringen konnten. Trotzdem empfiehlt es sich, nach der Implementierung eines Features den Code noch einmal gemeinsam durchzugehen und zu refaktorieren.

All dies sorgt für eine starke Fokussierung auf das wichtigste Ticket und ermöglicht einen stetigen Fortschritt durch die Reduzierung von Wartezeiten und Kontextwechseln. Dies führt zu einer sehr guten Time-to-Market, was einer der Hauptvorteile von Remote Mob Programming ist.

## Von den anderen lernen

Beim Remote Mob Programming lernt man durch die enge Zusammenarbeit ständig dazu. So wird aus einem anfangs sehr heterogenen Team immer mehr ein homogenes Team. Das macht auch ein klassisches Onboarding überflüssig. Wenn ein neuer Entwickler ins Team kommt, ist er am ersten Tag der Typist. Gemeinsam werden sein Rechner und die Zugänge eingerichtet. Am Ende des Tages ist das Ziel, dass es eine Codeänderung des neuen Kollegen in Produktion schafft. Danach ist der neue Kollege einfach Teil des Teams. Wenn er etwas nicht versteht, kann er jederzeit nachfragen. Das bremst das Team am Anfang etwas aus, sorgt aber für eine schnellere und bessere Einarbeitung des Kollegen. Schon nach kurzer Zeit hat sich der Kollege ein breites Wissen über das Fachgebiet und die Anwendung angeeignet, sodass er effektiv im Team mitarbeiten kann.

## Zuhause, aber nicht allein

Die Vorteile des Remote Mob Programming sind vielfältig: Das Pendeln zur Arbeit fällt weg. Es kann gemeinsam mit der Familie zu Mittag gegessen werden. Private Termine, wie beispielsweise der Handwerker oder ein Arztbesuch, sind auch kein Problem, da der virtuelle Raum für diese Zeit einfach verlassen werden kann. Obwohl nur noch von zu Hause aus gearbeitet wird, entsteht trotzdem eine starke Bindung zu den Kollegen. Remote Mob Programming schweißt ein Team zusammen und schafft viel Vertrauen. Durch den regelmäßigen Small Talk lernt man sich auch persönlich sehr gut kennen. Unstimmigkeiten und Konflikte im Team können so leichter und schneller erkannt und gelöst werden. Allerdings gibt es auch einen Nachteil gegenüber der asynchronen Arbeitsweise: So muss sich das Team beim Remote Mob Programming auf einen gemeinsamen Arbeitsbeginn sowie auf Pausen und Arbeitsende einigen. Um das Vertrauen im Team weiter zu stärken, empfiehlt es sich dennoch, regelmäßige Workshops oder Teamevents vor Ort durchzuführen.

## Wann ist Remote Mob Programming keine gute Idee?

Remote Mob Programming kann eine gute Methode sein, aber, wie so oft in der Softwareentwicklung, ist es kein Silver Bullet. So gibt es bestimmte Rahmenbedingungen, unter denen der Einsatz von Remote Mob Programming nicht oder nur bedingt zu empfehlen ist. Ist das Team zum Beispiel weltweit über verschiedene Zeitzonen verteilt, ist es unmöglich, für alle ein gemeinsames Zeitfenster für Remote Mob Programming zu finden, und somit Remote Mob Programming praktisch unmöglich.

Bereits bestehende Probleme und Konflikte im Team kommen durch die Einführung von Remote Mob Programming schnell an die Oberfläche. Diese Probleme können dann entweder schnell gelöst werden oder die Fronten verhärten sich, was eine effektive Zusammenarbeit im Team unmöglich macht und alle Teammitglieder frustriert. Daher sollte die Einführung von einem Coach begleitet werden, der dem Team die Methodik näherbringt, regelmäßige Retrospektiven mit dem Team durchführt und so dafür sorgt, dass Konflikte frühzeitig gelöst werden.

Wenn ein Team in hohem Maße von anderen Teams abhängig und nicht in der Lage ist, ein Feature allein zu entwickeln, kann Remote Mob Programming nicht die Vorteile einer guten Time-to-Market anbringen. Wenn dies der Fall ist, ist es ratsam, zuerst über den Schnitt der Anwendungen nachzudenken. Denn für ein effektives Remote Mob Programming sollte das Team so wenig Abhängigkeiten wie möglich haben, um Kontextwechsel zu vermeiden, die den stetigen Fortschritt unterbrechen.

Remote Mob Programming ist eine sehr kommunikative Arbeitsweise. Es gibt Persönlichkeiten, die mit dieser Arbeitsweise nicht gut zurechtkommen, da sie soziale Interaktion als sehr anstrengend empfinden und so nicht lange arbeiten können. Auch wenn man das Bedürfnis hat, Probleme selbst zu durchdenken, wird man mit Remote Mob Programming nicht glücklich werden. Es empfiehlt sich daher, Remote Mob Programming als ein Experiment über einen begrenzten Zeitraum auszuprobieren und danach gemeinsam zu entscheiden, ob diese Arbeitsweise zum Team passt.

## Fazit

Remote Mob Programming ist eine neue und aufregende Arbeitsmethode, die verteiltes Arbeiten neu denkt und besonders geeignet ist, wenn eines der Hauptziele des Projekts eine schnelle Time-to-Market ist und auch der Wissenstransfer maximiert werden soll. Remote Mob Programming ermöglicht es dem Team, gemeinsam ein gutes und qualitativ hochwertiges Produkt zu erstellen, in dem neue Features schnell implementiert werden können, da Wartezeiten reduziert werden. Remote Mob Programming sollte jedoch nicht einfach als Wundermittel betrachtet werden. Es ist wichtig, im Vorfeld die Ziele des Teams und des Projekts mit den Vor- und Nachteilen des Remote Mob Programming abzugleichen und zu prüfen, ob die Persönlichkeiten im Team mit dieser Arbeitsweise kompatibel sind.

Weitere Informationen zu Remote Mob Programming finden Sie unter remotemobprogramming.org. Wenn Sie Remote Mob Programming im Team ausprobieren wollen, können Sie einen eintägigen Workshop bei Socreatory *[5]* buchen. Wenn Sie es alleine mit anderen ausprobieren wollen, können Sie kostenlos an einer Remote Mob Session unter mobusoperandi.com teilnehmen.

## Quellen


[1] *https://remotemobprogramming.com*

[2] Mob Programming – A Whole Team Approach by Woody Zuill, Woody Zuill, July 2014, Agile2014 Conference *(https://www.agilealliance.org/resources/experience-reports/mob-programming-agile2014/)*

[3] *https://git-scm.com/*

[4] *https://mob.sh/*

[5] *https://www.socreatory.com/de/trainings/remote-mob-programming*

**Joshua Töpfer**
INNOQ
*joshua.toepfer@innoq.com*


Joshua Töpfer ist Senior Consultant bei INNOQ. Seit über zwei Jahren arbeitet er ausschließlich mit der Methodik des Remote Mob Programming. Er ist einer der Maintainer von mob.sh und coacht regelmäßig Teams, die Remote Mob Programming ausprobieren wollen.

## 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

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, Melanie Andrisek, 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 © storyset
*https://freepik.com*
S. 10 + 11: Bild © tobrono
*https://123rf.com*
S. 20 + 21: Bild © emojoez
*https://123rf.com*
S. 26 + 27: Bild © Designed by Freepik
*https://freepik.com*
S. 34 + 35: Bild © bolina
*https://123rf.com*
S. 40 + 41: Bild © Designed by Freepik
*https://freepik.com*
S. 48 + 49: Bild © upklyak
*https://freepik.com*
S. 54 + 55: Bild © naum100
*https://123rf.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*

Alle Rechte vorbehalten. Jegliche Vervielfältigung oder Weiterverbreitung in jedem Medium als Ganzes oder in Teilen bedarf der schriftlichen Zustimmung des Verlags.

Die Informationen und Angaben in dieser Publikation wurden nach bestem Wissen und Gewissen recherchiert. Die Nutzung dieser Informationen und Angaben geschieht allein auf eigene Verantwortung. Eine Haftung für die Richtigkeit der Informationen und Angaben, insbesondere für die Anwendbarkeit im Einzelfall, wird nicht übernommen. Meinungen stellen die Ansichten der jeweiligen Autoren dar und geben nicht notwendigerweise die Ansicht der Herausgeber wieder.

## Inserentenverzeichnis

DOAG Dienstleistungen GmbH U 2
iJUG e.V. S. 33, S. 47, S. 53, U3
JUG Saxony e. V. S. 19
Java User Group Deutschland e. V. S. 25
JavaLand GmbH U 4
JetBrains GmbH S. 45

# MITMACHEN UND BEITRAG EINREICHEN!

Du kennst dich in einem bestimmten Gebiet aus dem Java-Themenbereich bestens aus und möchtest als Autorin oder Autor dein Wissen mit der Community teilen?

Nimm Kontakt zu uns auf und sende deinen Artikelvorschlag zur Abstimmung an redaktion@ijug.eu.

Wir freuen uns, von dir zu hören!

iJUG Verbund

# JavaLand

## on demand

## JavaLand 2023 verpasst?

**Jetzt On-demand-Ticket buchen und Vortragsaufzeichnungen anschauen!**

**Alle Angebote im On-demand-Ticket-Shop**

Präsentiert von:

Heise Medien
DOAG
Veranstalter: