OX DEVELOPER

Großes Java-EE-Tutorial

Java 2017

Sommer 2017

Zusatzmaterial auf DVD

Sponsored Software:
Open for Innovation
KNIME

- Entwicklerwerkzeuge
- Buchauszüge
- KNIME Analytics Platform
- Videos
- Podcast-Episoden
- Beispielcode

DVD ROM

Datenträger enthält **Info- und Lehrprogramme** gemäß § 14 JuSchG

Was Java-Entwickler wissen müssen!

Java 9: Was sich alles ändern wird

Project Jigsaw und die Kunst der Modularisierung

Zukunft: Value Types, Project Panama & HTTP/2

Java EE 8: Was bringt der Enterprise-Java-Standard?

Im Trend: **Microservices** mit Java EE

MicroProfile als (r)evolutionäre Ergänzung

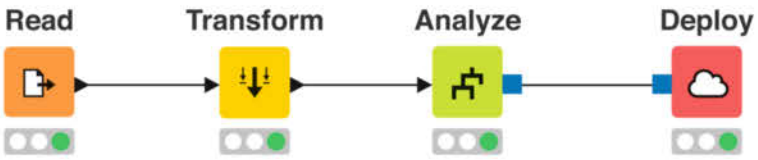
Lambda: Die nächste Generation von **JUnit**

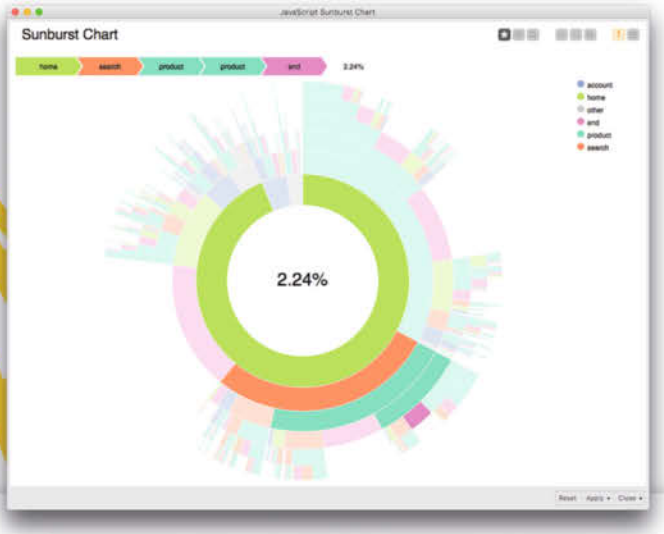
Reactive Programming mit dem **Spring**-Ökosystem

Erweiterte **Sicherheit** für Java-Anwendungen

Data Science, Analytics, KI, Deep Learning, Data Blending, ...

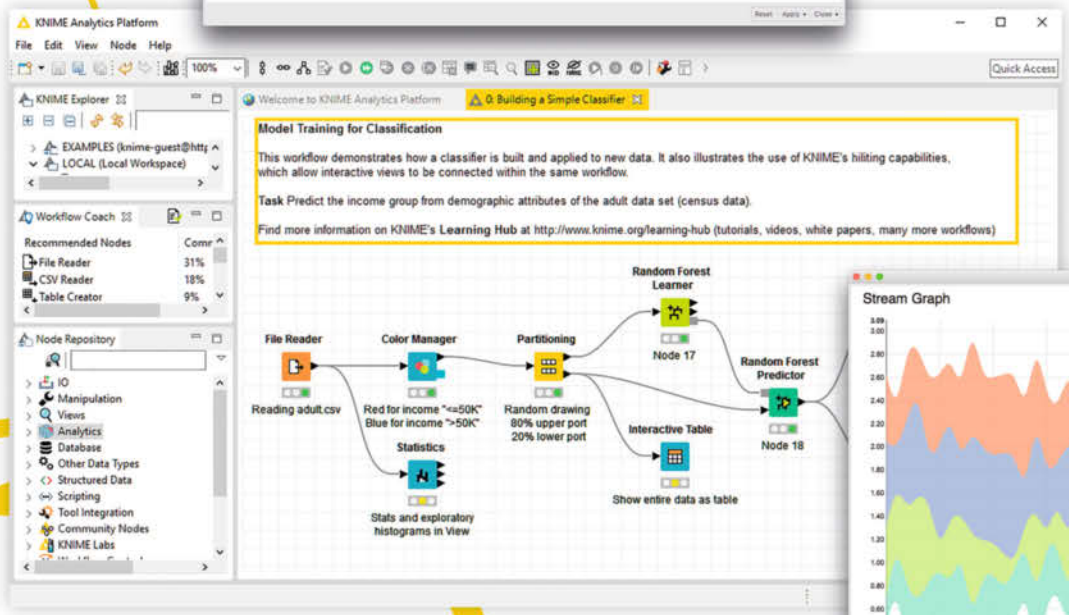
Für iX Leser GRATIS
"Beginner's Luck"
das eBook von Rosaria Silipo by KNIME Press
Promotion Code
iXJava17

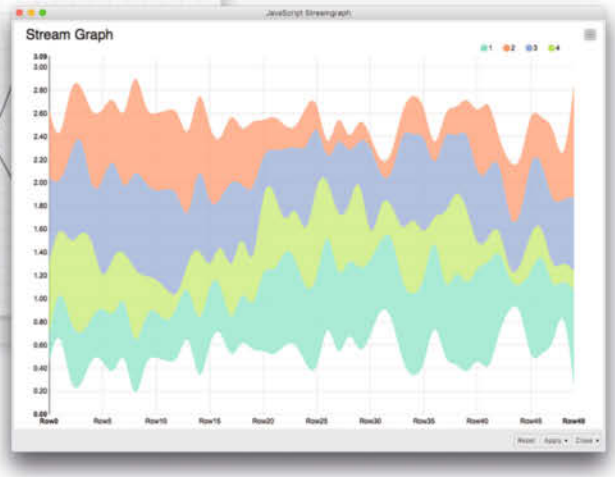




- Visuelle Workflows:
  von den Datenquellen bis zum Ergebnis in einem Fluss.
- Open Source:
  >1500 eigene Module und R, Python, JavaScript, Spark, ... Integrationen

Ausprobieren! Open Source KNIME Analytics Platform auf der Heft DVD





Open for Innovation ®
KNIME

Startup Feeling und Profitabel? Klar geht das!
Open Source Software Entwicklung mit Eclipse, Java, JavaScript, Apache Spark, uvm. In Berlin oder am Bodensee: Bewirb' Dich! → *knime.com/careers*

KNIME Standorte: **Zürich** - Technoparkstr. 1 I **Konstanz** - Reichenaustr. 11 I **Berlin** - Körtestr. 10 I **Austin, Texas** - 11801 Domain Blvd

Turbulente Erfolgsgeschichte

Java hat zweifelsohne eine äußerst erfolgreiche Laufbahn hinter sich. Als Sun Microsystems die Programmiersprache 1995 vorgestellt hat, ahnte kaum jemand, dass sie sich dauerhaft als eine der meist genutzten etablieren würde, während Sun in den Geschichtsbüchern verschwinden sollte. Seit Oracle das Unternehmen und damit das Ruder bei Java übernommen hat, haben sich auch die Kontroversen rund um Java verstärkt, was dessen Verbreitung jedoch kaum schaden konnte.

Selten waren die Zeiten rund um Java so turbulent wie in den letzten Monaten. Mit Java SE 9 steht ein großer Wurf an, der aber bis zur Zielgeraden viel Gegenwind bekam, der weniger technischer Natur war, sondern den Interessen der wichtigsten Beteiligten geschuldet.

Bei den Vorbereitungen zu diesem Sonderheft waren uns die Herausforderungen durchaus bewusst, aber wir waren zuversichtlich, dass Java 9 den nach zahlreichen Verschiebungen geplanten Termin Ende Juli einhalten kann. Dass es dazu nicht kommen wird, sondern (nach Stand beim Redaktionsschluss) Entwickler bis September warten müssen, liegt vor allem daran, dass sich die Verantwortlichen für das Platform Module System lange nicht einig wurden. Das als Project Jigsaw gestartete Modulsystem ist die prominenteste Neuerung in Java 9 und ohnehin schon überfällig, da es ursprünglich bereits in das 2011 veröffentlichte Java 7 einfließen sollte.

Die technischen Hürden haben die Plattformmacher inzwischen weitgehend überwunden, aber es blieben bis zum Schluss Bedenken der Entwickler, dass ihre Anwendungen aufgrund der Neuerungen nicht ohne Weiteres funktionieren. Dass die Macher darauf eingehen und die Veröffentlichung lieber verzögern, als zahlreiche Anwendungen ins Abseits zu befördern, ist grundsätzlich zu begrüßen. Ärgerlich ist vor allem der Umgang damit – und das nicht nur von Oracles Seite, sondern auch von der Seite der Kritiker, bei denen Red Hat und IBM an vorderster Stelle stehen, die durchaus mit dem Bremsmanöver während des Community Process eigene Interessen verfolgen.

2017 sagen Entwickler endlich zum Abschied leise Servus zu Java Applets. Zwar haben sie durchaus anfangs den ersten Erfolg der Programmiersprache mitbegründet, aber mittelfristig ihrem Ruf mehr geschadet, da sie zu zahlreichen Sicherheitsproblemen in Webbrowsern beigetragen haben.

Der treibende Motor ist die Java Virtual Machine: Die Plattformunabhängigkeit gehört zum Erfolgsrezept von Java, da sie garantiert, dass derselbe Code auf diversen Systemen erwartungsgemäß läuft. Inzwischen haben sich zudem andere JVM-Sprachen wie Kotlin, Scala und Clojure herausgebildet, die gänzlich andere Konzepte als das objektorientiere Java wie funktionale Programmierung verwenden, aber eben auf derselben Maschine laufen.

Mit diesem Heft möchten wir Ihnen einen umfassenden Überblick über Java im Jahr 2017 geben und dabei vor allem die Neuerungen von Java 9 und von Java EE 8 beleuchten, aber auch weitere APIs und Security-Neuerungen aufzeigen und einen Blick unter die Haube der JVM werfen. Auch ein Blick über den Tellerrand auf das Spring-Ökosystem, JUnit 5 und JavaScript als Frontend sollen nicht fehlen. *iX* und *heise Developer* wünschen Ihnen viel Spaß beim Lesen!

RAINALD MENGE-SONNENTAG UND
ALEXANDER NEUMANN

Java 9

Das neue Java Module System überstrahlt auf den ersten Blick alles bei Java 9. Und wie die Diskussionen bis zuletzt gezeigt haben, ist es auch die am kontroversesten diskutierte Neuerung der Java-Entwicklung. Doch die nächste Generation hat noch deutlich mehr zu bieten.

ab Seite 7

Java EE 8

Von den zahlreichen neuen Spezifikationen der kommenden Enterprise Edition sind einige bereits vor dem Release von Java EE 8 verfügbar. Die Context and Dependency Injection machte mit CDI 2.0 einen großen Schritt, und auch JavaServer Faces hat einige Neuerungen zu bieten.

ab Seite 69

Java SE 9

„Modularität als Sicherheitsgurt“
Im Gespräch mit Brian Goetz und Martin Lehmann **8**

Was ist neu in Java – Teil I?
Java 9 – mehr als ein neues Modulsystem **10**

Modularisierung
Java und die Tugend der Modularisierung **16**
Das Modulsystem Jigsaw in Java 9 **20**
Restrukturierung der Code-Basis mit Java 9 **30**

Was ist neu in Java – Teil II?
Multicore-Programmierung mit Java **37**
JShell – Read-Eval-Print Loop für Java **42**

Zukunftsmusik
Project Panama – das neue Java Native Interface **46**
Der HTTP/2-Client aus Java 9 (im Vergleich mit Jetty und OkHttp) **50**
Project Valhalla: Value Types in Java **56**

Standard Java

Laufzeitumgebung
Die Java Virtual Machine im Überblick **60**

Swing-Nachfolger
Status quo von JavaFX **64**

Java EE 8

Gegenwind aus der Community
Java EE – ein kritischer Lagebericht **70**

Was ist neu in Java EE?
Erweiterte Sicherheit für Java-Anwendungen **75**
CDI bringt große und kleine Änderungen **80**
JPA bewegt sich nur langsam, während Hibernate davoneilt **84**
Die Neuerungen in JavaServer Faces 2.3 **90**
JSON-P 1.1 in einer Microservice-Architektur **94**
MVC 1.0 als alternatives Webframework **100**

Enterprise Java

Java hat sich vor allem im Backend durchgesetzt. Die Java Enterprise Edition ist die tägliche Basis für viele Entwickler, denen ein Tutorial die Grundzüge und nützliche Erweiterungen zeigt. Java EE ist nicht mehr das Schwergewicht von früher, sondern taugt durchaus für schlanke Anwendungen. Im aktuellen Unternehmensumfeld gehören dazu auch Microservices.

ab Seite 107

Diverses

Nicht nur im Umfeld von Java SE und Java EE tut sich einiges, auch bei wichtigen Werkzeugen wie JUnit oder der Software aus dem Spring-Lager, das mit Version 5 des Frameworks das Thema Reactive Programming angeht. Und welcher Java-Entwickler kann noch von sich sagen, dass er ohne das im Webfrontend allmächtige JavaScript auskommt.

ab Seite 139

Enterprise Java

Java EE Tutorial
Teil 1: Dreigestirn aus JSF, CDI & JPA **108**
Teil 2: Asynchrone Nachrichtenverarbeitung **116**
Teil 3: Single Page & HTTP APIs **122**

Modulare Architektur
Microservices mit Java EE – das geht? **127**

Microservices
Evolution oder Revolution: MicroProfile **132**

Diverses

Test-Framework
Die nächste Generation von JUnit **140**

Webanwendungen
Enterprise JavaScript ohne Schrecken **146**

Framework
Das Spring-Ökosystem **150**

Sonstiges

Editorial **3**
DVD-Inhalt **6**
Inserentenverzeichnis **154**
Impressum **154**

Alle Links: www.ix.de/ix1715004 Artikel mit Verweisen ins Web enthalten am Ende einen Hinweis darauf, dass diese Webadressen auf dem Server der *iX* abrufbar sind. Dazu gibt man den *iX*-Link in der URL-Zeile des Browsers ein. Dann kann man auch die längsten Links bequem mit einem Klick ansteuern. Alternativ steht oben rechts auf der *iX*-Homepage ein Eingabefeld zur Verfügung.

Auf der Heft-DVD

Sponsored Software

KNIME Analytics Platform

Die Open-Source-Software zum Erstellen visueller Datenanalyse-Workflows mit Erweiterungen für die Integration zahlreicher Datenquellen (CSV, Excel, Datenbanken, Hive, Impala ...) und der Analyse von Texten, Bildern, Netzwerken und anderen Datentypen (für Windows, Linux, Mac).

Für Leser des Sonderhefts kostenlos:
das E-Book „Beginner's Luck" von KNIME Press (www.knime.org/knimepress/beginners-luck; Promotion Code ist „iXJava17").

Multimedia

Videos

Michael Wiedeking: **Programmiersprachen und parallele Programmierung** (parallel 2017)

Bibliotheken für die parallele Programmierung gibt es zuhauf, aber deren Benutzung erfordert oft viel zu viel Geschick und Sorgfalt, um sie vernünftig und korrekt einsetzen zu können. Werden aber die Mittel für die Parallelprogrammierung direkt in die Programmiersprache eingebettet, wird es oft sehr viel einfacher, deutlich weniger anfällig für Fehler und gelegentlich sogar noch besser bezüglich der Parallelisierung.

Jens Deters: **Einführung in MQTT – der Industriestandard für IoT-Kommunikation** (building IoT 2017)

Dieser Vortrag bietet eine umfassende Einführung in das im Internet der Dinge gesetzte MQTT-Protokoll sowie in MQTT.fx, eine plattformunabhängige Desktopanwendung zum grafischen Testen, Simulieren und Debuggen von MQTT-Kommunikation.

Dominik Obermaier: **MQTT 5 – What's New?** (building IoT 2017)

Das noch im Entstehen begriffene MQTT 5 kommt mit einigen neuen Features und Verbesserungen daher, was das schlanke Protokoll so vielseitig wie nie zuvor macht. Der Vortrag vermittelt, ob es sich bereits lohnt, auf das neue Release zu aktualisieren.

SoftwareArchitekTOUR-Podcast

- Pro und Contra von Web Components
- Einführung in die Programmiersprache Rust
- Wissenswertes zum Internet der Dinge
- Microservices und Self-contained Systems
- Softwareanalyse mit Graphendatenbanken
- Echte Cross-Platform-Anwendungsentwicklung
- Architekturanalyse und -bewertung
- Spring und Spring IO
- Know-how für Architekten
- Soziale Kompetenz für Architekten

Software

Ubuntu 16.04.2: Long-Term-Support-Release der populären Linux-Distribution

Eclipse 4.6.3: Auf der Heft-DVD finden Leser das dritte Update der Entwicklungsumgebung in der Neon-Distribution für Java-Entwickler.

NetBeans 8.2: Quelloffene Java-Entwicklungsumgebung aus dem Hause Oracle.

IntelliJ IDEA 2017.1.3: Community-Edition der Java-IDE, die viele Autoren des Sonderhefts bevorzugen.

Application Server: Apache Tomcat 8.5.15, Jetty 9.4.5, Payara 4.1.1.171.1, WildFly 10.1.0

Tools: Docker, JUnit 5, Gradle, Spring

Literatur

Nebenläufige Programmierung mit Java (Broschüre zum Buch)

Damit die Performance-Möglichkeiten moderner Multicore-Rechner effizient genutzt werden, muss die Software dafür entsprechend entworfen und entwickelt werden. Für diese Aufgabe bietet insbesondere Java vielfältige Konzepte an. Mit dieser Broschüre von Jörg Hettel und Manh Tien Tran erhalten Leser einen Überblick über die Abstraktionskonzepte Executor, Future und CompletableFuture für die nebenläufige Programmierung in Java.

Der Java-Profi: Persistenzlösungen und REST-Services (Buchauszüge)

Wer komplexe Java-Applikationen für den Desktop-Bereich schreibt, kann sich an Unternehmensanwendungen als weitere Herausforderung wagen. Dabei kommen Entwickler früher oder später mit Datenbanken, den Datenformaten XML oder JSON und auch mit verteilten Applikationen in Berührung. Einen Einstieg in das notwendige Wissen hierfür bieten die Buchauszüge aus Michael Indens Werk für Java-Profis.

API-Design (Buchauszüge)

Mit Schnittstellen zum Zwecke der Arbeitsteilung, Wiederverwendung oder beispielsweise zur Bildung einer modularen Architektur haben Entwickler täglich zu tun. Häufig werden hierbei jedoch nur unbewusst durch Erfahrung erlernte Konzepte und Best Practices genutzt. Auszüge aus Kai Spichales Buch schärfen den Blick für APIs und erläutern, welche Eigenschaften effektive APIs haben sollten.

Java 8 – die Neuerungen (Buchauszüge)

Auszüge des Buchs bieten einen fundierten Einstieg in Java 8 sowie einen Überblick über die umfangreichen Neuerungen im JDK 8. Damit eignen sich die Buchauszüge für all jene, die ihr Java-Wissen auffrischen und aktualisieren wollen.

Listings und Lizenzen

Die Listings und Beispiele zu den Heftartikeln sowie die Lizenzen zu den Softwarepaketen auf der Heft-DVD.

Hinweis für Käufer der digitalen Versionen

- PDF-, iPad- und Android-Version: In der *iX*-App finden Sie einen Button zum Download des DVD-Images.
- PDF-E-Book: Folgen Sie im Browser der unter „Alle Links" angegebenen URL.

Alle Links: www.ix.de/ix1715006

Java SE 9

Klar, das neue Java Module System überstrahlt auf den ersten Blick erst mal alles bei Java 9. Und wie die Diskussionen bis zuletzt gezeigt haben, ist es auch die am kontroversesten diskutierte Neuerung der Java-Entwicklung seit langem. Doch die nächste Generation hat noch deutlich mehr zu bieten, wie die folgenden Artikel zeigen werden. Selbst der Ausblick auf Java 10 verspricht, dass es nicht zum Stillstand kommt.

Im Gespräch mit Brian Goetz und Martin Lehmann	**8**
Java 9 – mehr als ein Modulsystem	**10**
Java und die Tugend der Modularisierung	**16**
Das Modulsystem in Java 9	**20**
Restrukturierung der Code-Basis mit Java 9	**30**
Multicore-Programmierung mit Java	**37**
JShell – Read-Eval-Print Loop für Java	**42**
Project Panama – das neue Java Native Interface	**46**
Der HTTP/2-Client aus Java 9 (im Vergleich mit Jetty und OkHttp)	**50**
Project Valhalla: Value Types in Java	**56**

iX Developer im Gespräch mit Brian Goetz und Martin Lehmann

„Modularität als Sicherheitsgurt“

Mit Java SE 9 und Java EE 9 erscheinen diesen Sommer zwei wichtige neue Releases. Bei aller Kontroverse dazu in der Java-Community, die das Sonderheft auch in den folgenden Artikeln thematisiert, stellen sie doch die Weichen für eine vermutlich weiterhin erfolgreiche Zukunft der Programmierplattform. *iX Developer* wollte es genau wissen und hat bei zwei wichtigen Vertretern aus dem Hause Oracle nachgefragt.

iX Developer: **Herr Goetz, Java 8 brachte den funktionalen Programmierstil, Java 9 endlich ein Modulsystem. Was hat mehr Einfluss auf die tägliche Arbeit von Java-Entwicklern?**
Brian Goetz: Jede dieser großen Plattformverbesserungen zielt darauf ab, die Leiden der Entwickler zu lindern. Wir haben Lambda-Funktionen nicht eingeführt, weil der funktionale Programmierstil populär ist. Sie wurden entworfen, weil sie den besten Weg darstellen, Java-APIs ausdrucksvoller, performanter und sicherer zu machen. Mit ihnen sorgen wir für ein besseres Java-Ökosystem.

Die primäre Motivation für das Modulsystem war die Erkenntnis, dass Java in beispiellos skalierbaren und komplexen Systemen eingesetzt wird. Java besitzt schon seit jeher Features zum Strukturieren großer Codebasen: Pakete für Namespaces, Zugangskontrollen für die Datenkapselung, Classloader zur Isolierung, Schnittstellen zur Abstraktion und so weiter. Aber letztlich verwenden die meisten Menschen eine Mischung aus Skripten und Tools, um die Codebasis in ein lauffähiges System zu verwandeln. Etwas im Kern von Java schien also zu fehlen.

Die Einführung eines Modulsystems bedeutet, dass sich jede Codebasis in Form von Modulen strukturieren lässt, die die Features geordnet zusammenfassen und sie alle besser machen können. Beispielsweise wird durch die Module ersichtlich, ob Pakete für den „internen“ Gebrauch bestimmt sind oder als APIs verwendet werden sollen. Das reduziert Bugs, erleichtert die Wartung und fördert die Wiederverwendung. Letztlich kommen die Module besser den Intentionen der Entwickler nach als Dinge wie Dokumentation, Tests, Skripte und Tools.

Wenn Features wie Lambda-Funktionen als „Jetpack“ für Entwickler beschrieben werden, dann ist die Modularität so etwas wie ein „Sicherheitsgurt“. Zwar bekommen Raketenrucksäcke mehr Aufsehen als Sicherheitsgurte, tatsächlich brauchen wir jedoch Leistungsfähigkeit und Sicherheit, um korrekte, zuverlässige und robuste Programme zu bauen.

iX Developer: **Warum hat es so lange gedauert, ein Modulsystem zu implementieren? Erste formale Ideen stammen ja bereits von 2005.**
Goetz: Die gleiche Frage ließe sich auch zu Generics oder Lambdas stellen – beide waren fast zehn Jahre im Gespräch, bevor sie in Java gelandet sind. Auf einer der letzten JavaOne-Konferenzen sprach James Gosling darüber, dass er und seine Entwickler sich vom ersten Tag an der Notwendigkeit der Generics bewusst gewesen wären. Sie wurden aber erst mal nicht Bestandteil der Sprache, weil Gosling und seine Mitstreiter keine zu Java passende Umsetzung gefunden hatten. Wenn Generics schon in Java 1.0 gelandet wären, hätten wir wahrscheinlich so etwas wie C++-Templates bekommen – und ich denke, dass das kein guter Weg gewesen wäre. Java hat eine lange Tradition bei der Suche nach einem effektiven 80/20-Prinzip, nach dem man den größten Vorteil für einen Bruchteil der Komplexität bekommt – und das geschieht in der Regel über langes und intensives Nachdenken darüber, was wesentlich ist und was nicht.

Es sei daran erinnert, dass „Java modular zu machen“ nicht einfach nur bedeutet, ein Modulsystem zu entwerfen – das ist wohl der „einfache“ Teil. Es dauerte allein schon mehrere Jahre, die JDK-Codebasis zu modularisieren. Verwendet man das System für eine Codebase, die so umfangreich ist wie das JDK, so erhält man im Gegenzug ein weiter verbessertes Design vom Modulsystem. Aber auch das ist nicht wirklich genug, um Entwicklern Modularität mit Java zu ermöglichen. Wie bei den Ge-

Brian Goetz …

… ist Java Language Architect bei Oracle und war für die Spezifikation des JSR 335 (Lambda Expressions für Java) zuständig. Er ist Autor des Bestsellers „Java Concurrency in Practice“ sowie von über 75 Artikeln zu Java-Entwicklung. Seine Begeisterung zur Programmierung geht bis in die Zeiten zurück, als Jimmy Carter Präsident wurde.

nerics war sicherzustellen, dass es einen sauberen Migrationspfad gibt, über den sich Codebasen allmählich umstellen lassen, da es keine Möglichkeit gibt, dass sofort und über Nacht eine bestehende Applikation modularisiert wird.

Bei Beobachtung der Entwicklung des Jigsaw-Designs in den letzten Jahren bin ich froh, dass wir so lange gewartet haben – das Modulsystem, das wir im JDK 9 bekommen, ist viel einfacher und konzeptuell schlüssiger als die früheren Iterationen des Designs oder als andere Modulsysteme, die für Java vorgeschlagen wurden – und der jetzige Stand ist viel besser dafür vorbereitet, existierende Anwendungen zu unterstützen.

Gewappnet für die Zukunft

iX Developer: **Mal JPMS ausgeklammert, was ist denn das wichtigste Feature von Java 9 für Sie?**
Goetz: JShell! Wer noch keine REPL (Read-Eval-Print Loop) aus anderen Sprachen kennt, kann sich wirklich darauf freuen. JShell ist eine interaktive Java-Shell, mit der Entwickler Ausdrücke und Deklarationen direkt eingeben und sie sofort und interaktiv auswerten können. Sie eignet sich wunderbar für das Erkunden von APIs, zum Ausprobieren und für prototypische Ideen. Aufgrund des interaktiven Charakters können Entwickler viel schneller ausliefern, und wenn etwas schiefgeht, können sie sofort eine andere Möglichkeit ausprobieren. Ich hoffe, dass die Shell ein Teil des täglichen Arbeitsprozesses jedes Java-Entwicklers wird.

iX Developer: **Warum ist Java im Jahr 2017 noch attraktiv? Und wie kann die Programmierplattform auf lange Zeit mit Sprachen wie Python und JavaScript mithalten, um nicht zum neuen Cobol zu werden?**
Goetz: Java als Sprache hat erfolgreich große Überarbeitungen (Generics, Lambdas) hinter sich – unter Beibehaltung der Kompatibilität mit bestehenden Code und Bibliotheken. Deswegen sind definitiv viele Entwickler weiterhin „im Boot“, die sonst über Bord gesprungen wären. Aber die Wahl einer Sprache geht weit tiefer als das Programmiermodell an der Oberfläche. Man muss die Qualität, Stabilität und Leistung der Laufzeitumgebung berücksichtigen: die Breite und die Zuverlässigkeit des Ökosystems an Bibliotheken, das der Werkzeuge sowie die Verfügbarkeit von Lernressourcen wie Bücher, Trainings und Q&As. In diesen Bereichen ist Java unübertroffen.

iX Developer: **Wie sehen Sie andere JVM-Sprachen wie Kotlin oder Clojure? Sind sie eine Stärkung der JVM oder eine Konkurrenz zu Java?**
Goetz: Wir sind glücklich über die Verbreitung von Sprachen auf der JVM. Sie bereichern das Ökosystem der JVM-Sprachen. Viele populäre Bibliotheken und Frameworks sind teilweise oder ganz in einer JVM-Sprache, also nicht in Java selbst, geschrieben. Trotzdem profitieren dann die Anwender von allen diesen JVM-Sprachen. Sie sind außerdem ein ausgezeichnetes Experimentierumfeld für die Programmierung von Sprachfeatures, da „Early Adopters“ versuchen, diese zu erforschen, und dadurch wertvolle Informationen darüber liefern, wie sich konservativere Sprachen wie Java entwickeln können.

iX Developer: **Nun ist es an der Zeit, sich Java EE zuzuwenden. Herr Lehmann, warum gibt es in Java EE 8 keine Features für Container- und Microservice-Anwendungen?**
Mike Lehmann: Zuerst sei gesagt, dass Java EE bereits das Erstellen von Microservices unterstützt, die in Docker-Containern laufen. Oracle, IBM und Red Hat haben alle Docker-Images mit ihren Java-EE-Anwendungsservern veröffentlicht, und wir sehen ein großes Interesse an Java EE mit Containern in Microservices-Architekturen. So stellt sich eher die Frage, wie wir Java EE für Cloud und Microservices verbessern können. Wir hatten letztes Jahr die Community befragt, was für sie die wichtigsten Features sind, und in der Folge beschlossen, so schnell wie möglich eine Implementierung der für sie wichtigsten Anforderungen mit Java EE 8 zu liefern. Sie umfasst Plattformmodernisierungen mit Features, die sich von Microservices nutzen lassen, zum Beispiel Aktualisierungen bei der Unterstützung für REST, JSON und das HTTP/2-Protokoll. Wir haben für Verbesserungen bei CDI, Bean Validation und den Security-Standards gesorgt. Und wir haben Java-EE-Entwicklern die Features von Java SE zur Verfügung gestellt. Alle diese Änderungen entsprechen den Bedürfnissen der Community und ermöglichen eine größere Verbreitung von Java EE in Microservices-Umgebungen.

Schritt für Schritt voran bei Java EE

Abgesehen davon sind sich sowohl Oracle als auch die Community bewusst, dass es in diesem Bereich eine riesige Menge an technischen Innovationen gibt. Standards wie Java EE formalisieren, was akzeptiert ist oder bald voraussichtlich akzeptiert wird, oder De-facto-Standards in der Branche. Die Definition eines Java-EE-Standards zu Serverless Computing wurde beispielsweise als zu früh für Java EE 8 erachtet und bekam deswegen eine niedrigere Priorität unter den Community-Anforderungen.

iX Developer: **Was halten Sie von der MicroProfile-Initiative? Ist es eine nützliche Ergänzung – oder wird sie in Zukunft durch Java-EE-Standards ersetzt?**
Lehmann: Wir sehen die MicroProfile-Initiative unter dem Dach der Eclipse Foundation als komplementär zu Java EE. Tatsächlich ist eines der wichtigsten Elemente der aktuellen MicroProfile-Roadmap die Unterstützung der Java EE 8 API. Es geht hier um die erwähnten Neuerungen bei REST, JSON und CDI. MicroProfile stellt einen von vielen Ansätzen für Microservices dar – wenn diese Ansätze reifen und weithin genutzt werden, nehmen Standards wie Java EE sie (oder die besten von ihnen) auf und machen sie zum Standard. Oracle freut sich darauf, die MicroProfile-Entwickler am Java-EE-Community-Prozess teilnehmen zu sehen.

iX Developer: **Wir bedanken uns für ihre Antworten.**

Die Fragen stellten Alexander Neumann und Rainald Menge-Sonnentag (ane)

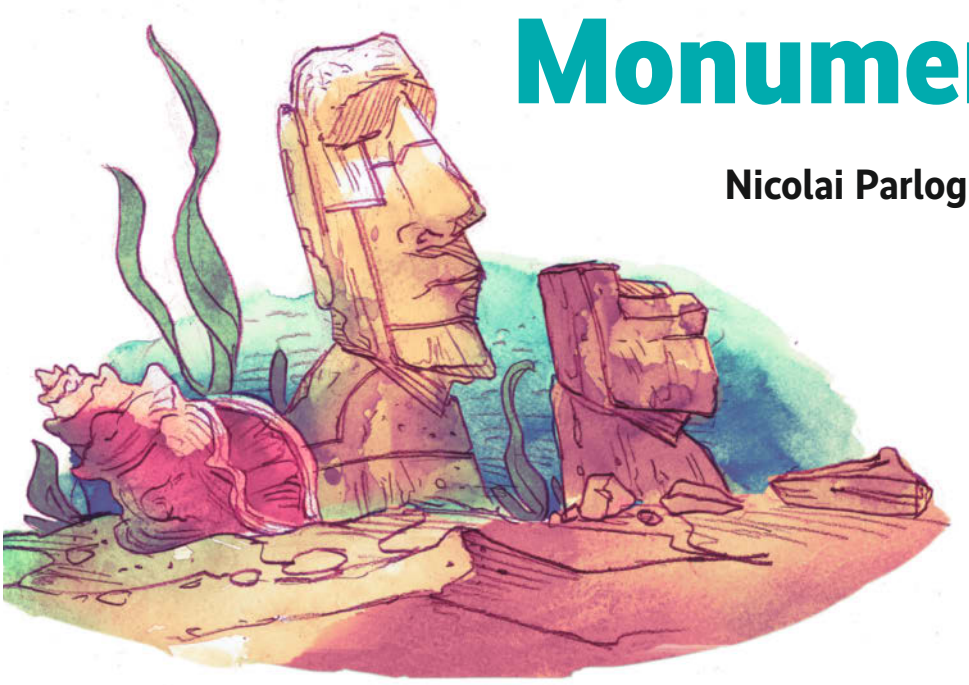
Mike Lehmann ...

... ist Vice President Product Management für das Oracle Cloud Application Development Portfolio und hier verantwortlich für Entwicklung und Bereitstellung mehrerer Container-Services für die native Anwendungsentwicklung und der Java-Plattform sowie von Oracle WebLogic Server und Coherence. Er hat 20 Jahre Erfahrung in Produktentwicklung, -marketing und -management.

Java 9 – mehr als ein neues Modulsystem

Monumental

Nicolai Parlog



Änderungen gibt es in Java 9 jede Menge: ein neues Modulsystem, private Methoden in Interfaces, eine verbesserte API für den Umgang mit Betriebssystemprozessen und Performanceverbesserungen sind vorrangig zu nennen. Sowohl bei der Sprache selbst als auch bei den Bibliotheken des Java Development Kit (JDK) und der Java Virtual Machine (JVM) sind viele Dinge hinzugekommen und verbessert worden.

Für den Artikel wurden die Early Access Builds des JDK 9 verwendet [a] [b]; Javadoc für Java 9 ist ebenfalls bereits verfügbar [c]. Für die Spracherweiterungen und einige der neuen APIs gibt es Beispiele in einem Java-9-Demo-Repository auf GitHub sowie auf der Heft-DVD [d].

Während Java 8 die Syntax mit Lambda-Ausdrücken in eine neue Epoche katapultiert hat, hält sich Java 9 hier zurück. Es gibt nur eine Handvoll Syntaxerweiterungen, und keine davon wird den Programmierstil nachhaltig verändern. Aber es sind angenehme, inkrementelle Überarbeitungen, die einige Kanten der Sprache glätten.

Private Methoden und Try-with-resources

Um Interfaces um neue Methoden erweitern zu können, etwa *stream* oder *forEach* auf *Collection*, hat Java 8 sogenannte Default Methods eingeführt. Sie hatten bisher den Nachteil, dass es keine angenehme Art und Weise gibt, Code zwischen solchen Methoden zu teilen, ohne ihn öffentlich verfügbar zu machen (s. Listing 1). Java 9 erlaubt nun private Methoden in Interfaces:

```
private int sum(int[] numbers) {
    return IntStream.of(numbers).sum();
}
```

Private Interface-Methoden werden genauso behandelt wie private Methoden in Klassen. Das heißt, weder Klassen, die das Interface aufrufen, noch solche, die es implementieren, können diese Methoden aufrufen beziehungsweise überschreiben.

Ein Try-with-resources-Ausdruck erlaubt, hinter dem *try* eine *AutoCloseable*-Variable zu deklarieren und sie innerhalb des Blocks mit der Sicherheit zu verwenden, dass auch im Fall einer Exception beim Verlassen *close* aufgerufen wird:

```
void doSomethingWith(Connection connection) throws Exception {
    try(Connection c = connection) {
        c.doSomething();
    }
}
```

Die Deklaration von *c* sieht allerdings ziemlich nutzlos aus. Warum ist sie also nötig? Der *try*-Ausdruck ist nur zu verstehen, wenn klar ist, auf welcher Instanz *close* aufgerufen wird. Die einfachste Art und Weise, das sicherzustellen, ist, die Variable final zu machen. Und *try* kümmert sich genau darum – *c* ist final, auch ohne das Schlüsselwort.

Seit Java 8 kennt der Compiler das Konzept „effectively final“. Das gilt für eine Variable, wenn sie final sein könnte und nur das Schlüsselwort fehlt. Mit dieser Weiterentwicklung müsste der Compiler in der Lage sein, selbst festzustellen, ob die Variable im Block neu zugewiesen wird oder nicht. Und tatsächlich ist genau das ab Java 9 der Fall:

```
void doSomethingWith(Connection connection) throws Exception {
    try(connection) {
        connection.doSomething();
    }
}
```

Hier ist *connection* „effectively final“, und es lässt sich somit direkt benutzen, ohne es erst einer Helfervariable zuzuweisen.

Anonyme Diamaten

Der mit Java 7 eingeführte Diamond-Operator erlaubt, eine Instanz zu erstellen, ohne generische Typparameter anzugeben:

```
java List<String> names = new ArrayList<>();
```

Wenn man allerdings eine Instanz einer anonymen Klasse erzeugen wollte, funktionierte das bisher leider nicht (s. Listing 2). Grund dafür ist, dass der Compiler ein umfangreicheres Typsystem kennt als die JVM und dementsprechend nicht alle Typen in Bytecode ausdrücken kann. Diejenigen, die sich nicht darstellen lassen, nennen sich „non-denotable types“. Beim Erstellen einer anonymen Klasse mit Diamond kann es passieren, dass der Compiler einen solchen Typ ableitet, deswegen wurde diese Kombination schlicht untersagt.

In Java 9 wird die Regel gelockert: Der Operator lässt sich verwenden, wenn der Typ ausgedrückt werden kann, andernfalls beschwert sich der Compiler. Das heißt, das Folgende ist jetzt möglich:

```
// in Java 9, the following compiles
List<String> names = new ArrayList<>() {
        // ... [whatever]
};
```

Listing 3 zeigt ein Beispiel für einen „non-denotable type“. Man erkennt, dass solche Fälle selten auftreten. Zum Beispiel ist im JDK nur eine kleine Minderheit der Typen „non-denotable“.

SaveVarargs und Deprecation-Warnings

Bei der Mischung von Varargs [e] und Generics [f] kann es passieren, dass der Compiler eine Warnung mit dem Stichwort Heap Pollution [g] ausgibt. Verkürzt gesagt weist sie darauf hin, dass der Compiler nicht beweisen kann, dass eine generische Variable ein Objekt vom korrekten Typ referenziert. Dadurch ist es möglich, dass der Code zur Laufzeit mit einem Typfehler scheitert – ein Ereignis, das Java als statisch typisierte Sprache zu verhindern versucht.

Hat man die Umstände geprüft und ist sich sicher, dass das Problem nicht auftreten kann, lässt sich die Warnung mit der Annotation *@SaveVarargs* unterbinden:

```
@SafeVarargs
private <T> Optional<T> firstNonNull(T... args) {
    return stream(args)
            .filter(Objects::nonNull)
            .findFirst();
}
```

Nur funktionierte das genau in solchen Fällen vor Java 9 nicht, weil die Annotation auf privaten, nicht finalen Methoden nicht verwendet werden darf. Das ist eher versehentlich daraus entstanden, dass *@SafeVarargs* nur auf nicht überschreibbaren Methoden eingesetzt werden darf [h], und für Instanzmethoden hat man als Kriterium für die An- oder Abwesenheit das Schlüsselwort *final* verwendet. Das lässt den Sonderfall von privaten, nicht finalen Methoden außen vor, denn hier ist das Schlüsselwort *final* laut Annahme abwesend, überschreiben lassen sie sich aber dennoch nicht. Java 9 behebt diese Lücke und lässt die *@SafeVarargs* auf privaten Methoden grundsätzlich zu, also unabhängig von *final*.

Wer gerne warnungsfreie Projekte hat, wird vielleicht schon darüber gestolpert sein, dass auch Importbefehle Deprecation Warnings auslösen können: Es erzeugt zum Beispiel ein Typ, der „deprecated“-Klassen verwendet, aber selbst als „deprecated“ markiert wird, Warnungen, wenn die veralteten Abhängigkeiten importiert werden (s. Listing 4). Hier sollte es keine Warnung geben, denn idealerweise sollten diese nur an der Grenze zwischen aktuellem und veraltetem Code auftreten. Tatsächlich bleibt im Beispiel aber nur, *LineNumberInputStream* voll qualifiziert zu verwenden und so den Import zu vermeiden, was unschön ist. Mit Java 9 ist das nicht mehr nötig, denn Importe erzeugen dann selbst keine Warnung mehr.

Listing 1: Vor Java 9 ist keine Wiederverwendung von Code in Interfaces möglich, ohne sie zum Teil der öffentlichen API zu machen.

```
default boolean evenSum(int... numbers) {
    return sum(numbers) % 2 == 0;
}

default boolean oddSum(int... numbers) {
    return sum(numbers) % 2 == 1;
}

// we don't want this to be public;
// but how else do we reuse?
default int sum(int[] numbers) {
    return IntStream.of(numbers).sum();
}
```

Listing 2: Vor Java 9 kann der Diamant-Operator nicht bei anonymen Klassen angewendet werden.

```
// the following does not compile
List<String> names = new ArrayList<>() {
    // ... [whatever]
};

// we have to do this
List<String> names = new ArrayList<String>() {
    // ... [whatever]
};
```

Listing 3: Auch in Java 9 ist der Diamant-Operator nicht bei allen anonymen Klassen möglich.

```
List<?> createCrazyList(Object content) {
    List<?> innerList = Arrays.asList(content);
    // we can't use the diamond operator
    // because the inferred type is non-denotable;
    return new ArrayList<List<?>>(innerList) {
        // ... [whatever]    };
}
```

Listing 4: Dieser Import löst eine Compiler-Warnung aus.

```
import java.io.LineNumberInputStream;

@Deprecated
public class DeprecatedImports {

    LineNumberInputStream stream;

}
```

Java Platform Module System

Das Java Platform Modulsystem (JPMS) ist das wichtigste neue Feature von Java 9 und wird ab Seite 16 im Detail vorgestellt – deswegen gibt es hier nur eine kurze Zusammenfassung. Im Kern des Modulsystems stehen selbstverständlich Module. Dabei handelt es sich letztlich um reguläre JARs, die mit Metainformationen versehen wurden, die Compiler und JVM auslesen können (sie werden modulare JARs genannt). Die wichtigsten Eigenschaften eines Moduls sind:

- Es hat einen Namen (möglichst global eindeutig).
- Es gibt Abhängigkeiten auf andere Module an.
- Seine öffentliche API wird durch die exportierten Pakete definiert.

Diese Metainformationen werden in der Datei *module-info.java* hinterlegt – ein Beispiel:

```
module org.codefx.foo {
    requires org.codefx.bar;
    exports org.codefx.foo.api;
}
```

Hier wird ein Modul namens *org.codefx.foo* definiert, das von einem anderen, nämlich *org.codefx.bar*, abhängt. Zusätzliche Abhängigkeiten lassen sich durch weitere *requires*-Klauseln angeben. Das Modul exportiert das Paket *org.codefx.foo.api*; weitere Pakete können durch mehr *exports*-Klauseln exportiert werden.

Compiler und JVM wurden parallel zum Class Path um ein neues Konzept erweitert, den Module Path. Er erwartet modulare statt reguläre JARs. Beim Kompilieren oder Starten eines Moduls stellt das Modulsystem sicher, dass alle benötigten Abhängigkeiten auf dem Modulpfad oder innerhalb des JDK vorhanden sind. Dadurch lassen sich einige Laufzeitfehler wie *NoClassDefFoundError* weitgehend vermeiden. Die *exports*-Klauseln werden verwendet, um die API eines Moduls zu definieren. Andere Module können nur *public*-Typen einsetzen, die in solchen Paketen enthalten sind. Das gilt auch für die Reflection API. Damit haben die Entwickler einer Bibliothek oder eines Frameworks endlich die Möglichkeit, Implementierungsdetails zuverlässig vor dem Zugriff Dritter zu schützen.

Kleinere Neu- und Weiterentwicklungen

Gerne übersehen Entwickler, dass es außer dem Modulsystem viele weitere Neuerungen und Überarbeitungen gibt – zu viele, um sie erschöpfend in diesem Artikel zu besprechen. Trotzdem seien sie zumindest im Folgenden aufgelistet. Tiefer lässt sich über die Verlinkung auf die jeweiligen JEPs oder andere Quellen über die am Ende des Artikels unter „Alle Links“ aufgeführte Webseite einsteigen.

APIs:

- Factory-Methoden für Collections auf den entsprechenden Interfaces (z.B. *List.of("a")*, JEP 269)
- Interfaces für Reactive Streams, die es entsprechenden Bibliotheken erlauben sollen, gemeinsame Typen zu verwenden (JEP 266)
- Erweiterungen existierender APIs (z. B. Optional und Stream)
- einige technische APIs, die für Anwendungsentwickler typischerweise nicht allzu interessant sind (z. B. Dynalink, Variable Handles und Method Handles)

Netzwerk:

- vorläufige HTTP/2-Unterstützung und -API (JEP 110)
- Datagram Transport Layer Security (DTLS, JEP 219)
- TLS Application-Layer Protocol Negotiation Extension (TLS ALPN, JEP 244)
- OCSP Stapling for TLS (JEP 244)

Grafik:

- Multi-Resolution Images (JEP 251)
- TIFF-Unterstützung (JEP 262)
- HiDPI-Unterstützung (Linux und Windows; JEP 263)
- Verwendung von GTK 3 auf Linux (optional; JEP 283)
- Update auf GStreamer 1.4.4 (JEP 257)
- HarfBuzz als neue OpenType Layout Engine (JEP 258)

Unicode:

- Unterstützung von Unicode 8.0 (JEP 227, JEP 267)
- Unicode in Property Files (JEP 226)

Sicherheit:

- Implementierung einiger SHA-3-Algorithmen (JEP 287)
- *SecureRandom*s-Java-Implementierung verwendet einen besseren Deterministic Random Bit Generator (JEP 273)

Performance:

- Compact Strings reduzieren den Speicherbedarf um durchschnittlich 10 bis 15 Prozent (JEP 254)
- schnellere und speicherschonendere String-Konkatenation (JEP 280)
- Der Performance-Verlust durch Lock Contention wurde reduziert (JEP 143)
- Die Verlangsamung durch den *SecurityManager* wurde reduziert (JEP 232)
- GHASH- und RSA-Berechnungen wurden durch Verwendung neuer CPU-Routinen auf modernen Prozessoren beschleunigt (JEP 246)
- Im OpenJDK wurde der Pisces-Renderer durch Marlin ersetzt, was die Performance grafikintensiver Anwendungen verbessert (JEP 265)

Kommandozeile:

- Neue Kommandozeilenparameter werden der GNU-Syntax (*–parameter*, *-p*) folgen (JEP 293)
- neuer Compiler-Parameter *–release*, der *-source*, *-target* und *bootclasspath* auf die passenden Werte setzt, um für alte Java-Versionen zu kompilieren (JEP 247)
- Werte von Kommandozeilenparametern werden jetzt beim Start der JVM geprüft (JEP 245)
- Alle Log-Meldungen der JDK-Bibliotheken, aber auch des Garbage Collector sind durch einheitliche Parameter auswähl- und formatierbar (JEP 158, JEP 271)

Tools:

- JShell, die REPL für Java (JEP 222; s. Artikel auf S. 42)
- JavaDoc kann jetzt HTML 5 erzeugen (JEP 224) und hat eine Suche (JEP 225) sowie eine neue Doclet API (JEP 221) spendiert bekommen

Zum Modulsystem lässt sich noch viel sagen, insbesondere zur Kompatibilität [i], Unterstützung von Reflection [j], Migration sowie weiteren Features wie Services, Layers [k] und jlink [l]. An dieser Stelle muss aber diese kurze Einführung genügen, damit genug Raum für andere spannende Neuerungen in Java 9 bleibt.

Das Modulsystem macht es leicht, die vielen anderen Änderungen zu übersehen, die Java 9 mit sich bringt. So viele, dass sie unmöglich alle in einem Artikel vorzustellen sind. Bevor einige etwas ausführlicher beschrieben werden, sei deswegen auf den Kasten „Kleinere Neu- und Weiterentwicklungen" hingewiesen, der eine Liste derjenigen enthält, die ausgelassen werden müssen.

Prozess-Interaktion und Stack Walking

Die durch JEP 102 erweiterte Process API vereinfacht das Erstellen, Verknüpfen und die Interaktion von und mit Prozessen deutlich. Zum Beispiel kann man mit der neuen Methode *ProcessBuilder::startPipeline* leicht verknüpfte Linux-Befehle wie *tree -i | grep pdf* ausführen (s. Listing 5). Der Aufruf von *startPipeline* ist asynchron. Das heißt, er startet die Prozesse, kehrt aber zurück, während sie noch ausgeführt werden. Um auf ihren Abschluss zu warten, lässt sich die neue Methode *Process::onExit* verwenden, die einen *CompletableFuture* zurückgibt (s. Listing 6).

Ebenfalls neu ist eine einfache Variante, die ID eines Prozesses zu erfahren:

```
lsThenGrep.forEach(
    process -> System.out.println("PID: " + process.getPid()));
```

Einige Tools und Frameworks benötigen Informationen über den aktuellen Execution Stack, um zum Beispiel zu wissen, welche Methoden aufgerufen wurden. Dazu war es bisher üblich, ein *Throwable* zu erzeugen. Denn dann evaluiert die JVM den Stack, um das Trace zu erzeugen, der von der Exception erfragt werden kann (s. Listing 7).

Allerdings ist das Evaluieren eines Stack Frame aufwendig, und in der Variante in Listing 7 wird das notgedrungen immer für jedes auf dem gesamten Stack durchgeführt. Der ist gerade in Enterprise-Anwendungen gerne recht tief, sodass die Performance darunter leiden kann. Um das zu verbessern sowie um eine zeitgemäßere API zu entwickeln, wurde JEP 259 ins Leben gerufen. In Java 9 kann der Stack nun Frame für Frame erkundet werden, wobei jeder erst ausgewertet wird, wenn er auch angefragt wurde. Für diese Art der Lazy Evaluation ist die Stream-API besonders geeignet (s. Listing 8). Interessant ist dabei, dass der Stack Walker den Stream von Frames erzeugt und eine Funktion aufruft. Der Grund dafür ist, dass der Walker das Stack Trace nicht komplett auswertet, sondern nur auf Anfrage erkunden soll. Das erfordert einen stabilen Stack, aber der lässt sich nur unter besonderen Umständen garantieren. Dafür muss die Methode *StackWalker::walk* die Kontrolle über den Stack haben, indem sie selbst enthalten ist.

Umleitung für JRE-Log-Nachrichten

Einige JDK-Klassen erzeugen eigene Log-Meldungen, aber bisher gab es keine Mittel, diese in das Logging-Framework der Anwendung (z. B. Log4J) umzuleiten. Java 9 behebt das, indem es Anwendungen die Möglichkeit gewährt, ein eigenes Logging-Backend für die JDK-Nachrichten anzugeben. Alle Nachrichten

Listing 5: Die Process API wurde für Java 9 unter anderem um *startPipeline* erweitert.

```
ProcessBuilder ls = new ProcessBuilder()
        .command("tree", "-i")
        .directory(Paths.get("/home/nipa").toFile());
ProcessBuilder grepPdf = new ProcessBuilder()
        .command("grep", "pdf")
        .redirectOutput(Redirect.INHERIT);
List<Process> lsThenGrep = ProcessBuilder
        .startPipeline(asList(ls, grepPdf));
```

Listing 6: Neu ist auch *onExit*, das einen CompletableFuture für den OS-Prozess zurückgibt.

```
CompletableFuture[] lsThenGrepFutures = lsThenGrep.stream()
        .map(Process::onExit)
        .toArray(CompletableFuture[]::new));
// wait until all processes are finished
CompletableFuture
        .allOf(lsThenGrepFutures)
        .join();
```

Listing 7: Vor Java 9 wurde zum Erkunden des Stack Trace meist *Throwable* bemüht, das immer den gesamten Stack erfasst.

```
private void printLineNumber() {
    StackTraceElement[] stackTrace = new Throwable().getStackTrace();
    String line = walk(stackTrace);
    System.out.println(line);
}

private String walk(StackTraceElement[] stackTrace) {
    return Arrays.stream(stackTrace)
            .filter(frame -> frame
                    .getMethodName()
                    .contains("callingMethod"))
            .map(frame -> "Line " + frame.getLineNumber())
            .findFirst()
            .orElse("Unknown line");
}
```

Listing 8: Die neue StackWalker API macht den Zugriff auf Stack Frames angenehmer und performanter.

```
private void printLineNumber() {
    String line = StackWalker.getInstance().walk(this::walkStack);
    System.out.println(line);
}

private String walkStack(Stream<StackFrame> stackTrace) {
    return stackTrace
            .filter(frame -> frame
                    .getMethodName()
                    .contains("callingMethod"))
            .map(frame -> "Line " + frame.getLineNumber())
            .findFirst()
            .orElse("Unknown line");
}
```

werden durch ein Interface *Logger* geleitet, wovon ein *LoggerFinder* Implementierungen zur Verfügung stellen muss. Der Finder ist ebenfalls ein Interface, und Implementierungen davon werden über ein Feature des neuen Modulsystems, nämlich Services, lokalisiert.

Services geben Modulen die Option, mit der erweiterten Service Loader API [m] zu interagieren und Implementierungen von Interfaces bereitzustellen. Konsumenten einer Schnittstelle können sich vom Modulsystem alle registrierten Implementierungen zur Verfügung stellen lassen, ohne diese selbst kennen zu müssen (klassische Entkopplung mit dem Service Locator Pattern [n]).

Zusammengefasst heißt das, dass Maintainer von Logging-Frameworks ein Modul erstellen können, das eine Framework-spezifische Implementierung des *LoggerFinder* als Service deklariert. Die Runtime wird dann den Finder verwenden, um Logger für die JDK-Bibliotheken zu beziehen und so Frame-

work-spezifische Logger zu erhalten, die nichts anderes als Adapter für die existierenden Logger sind. Ein Beispiel dazu ist im Demo-Projekt auf der Heft-DVD zu finden.

Version Strings

Bisher hat die System Property *java.version*-Strings der Form *1.x…* zurückgegeben. Das entspricht schon lange nicht mehr dem Sprachgebrauch, der zum Beispiel von Java 9 redet. Das JDK stimmt damit nun endlich überein und gibt Strings der Form *$MAJOR.$MINOR.$SECURITY* zurück:

- *$MAJOR* gibt die Hauptversion an, also derzeit 9.
- *$MINOR* markiert die regelmäßigen Releases einer Hauptversion und beginnt nach *9.x* wieder bei *10.0*.
- *$SECURITY* wird bei jeder sicherheitsrelevanten Veröffentlichung erhöht und bei einer Erhöhung von *$MINOR* nicht wieder zurückgesetzt. Auf *9.2.4* folgt also *9.3.4* oder *9.3.5*, je nachdem ob das Release Sicherheits-Patches enthält.

Listing 9: Multi-Release-JARs legen versionsspezifische Class Files in *META-INF/versions* ab.

```
JAR root
org/codefx/... (mehr Ordner)
  Main.class
  VersionDependent.class (auf Java 8)
META-INF
  versions
    9
      org/codefx/... (mehr Ordner)
        VersionDependent.class (auf Java 9)
```

Der Grund für das ungewöhnliche Verhalten von *$SECURITY* liegt darin begründet, dass so an der Nummer allein erkennbar ist, welche Version die aktuellsten Sicherheits-Patches enthält – etwas, das derzeit nicht der Fall ist.

Der String *java.version* muss übrigens nicht geparst werden. *Runtime.version()* gibt ihn in Form eines Objekts vom Typ Version zurück.

Multi-Release-JARs und G1

Wer schon einmal Code geschrieben hat, der je nach Java-Version unterschiedlich ausfallen sollte, weiß, dass das nicht leicht ist. Ab Java 9 gibt es nun Unterstützung für dieses Szenario. Sogenannte Multi-Release-JARs können im Verzeichnis *META-INF/versions* *class*-Dateien enthalten, die je nach Version die sonst verwendeten Dateien ersetzen. In Listing 9 wird auf Java 8 die erste Variante von *VersionDependent* verwendet, auf Java 9 die zweite. Jetzt müssen nur noch Build-Werkzeuge und IDEs damit umgehen können.

Der Garbage Collector G1 ist seit Java 8 Teil des JDK. Die große Änderung in Java 9 ist, dass er zum Standard-GC wird. Das heißt, ohne besondere Flags wird er beim Start des JDK ausgewählt. Gegenüber dem bisher verwendeten Parallel Collector, der weiterhin verfügbar ist, zeichnet er sich durch kürzere Stop-the-world-Pausen aus, hat dafür aber auch einen etwas schlechteren Durchsatz. Damit ist er eher für latenzsensitive Anwendungen (z. B. Backends) interessant als für laufzeitsensitive (z. B. Simulationen).

Fazit

Von kleinen Sprachänderungen über viele neue und erweiterte APIs hin zu neuen JVM-Features und besserer Performance bringt Java 9 viele Änderungen. Dennoch wird das nächste Release den Programmieralltag weniger dramatisch verändern als Java 8 – einzig das Modulsystem wird wohl eine fundamentale Umstellung bewirken, aber die wird sich eher langfristig zeigen. Dennoch sind viele tolle Neu- und Weiterentwicklungen dabei, und allein das Modulsystem, die Performance-Verbesserungen sowie der freie Weg zu Value Types in Java 10 (Daumen drücken!) machen ein Update lohnenswert. (ane)

Onlinequellen

[a] JDK 9 Early Access Releases
jdk9.java.net/download/

[b] JDK 9 Early Access mit Project Jigsaw
jdk9.java.net/jigsaw/

[c] Java SE 9 API Specification
download.java.net/java/jdk9/docs/api/overview-summary.html

[d] Demo-Repository für Java 9
github.com/CodeFX-org/demo-java-9

[e] Varargs
docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html

[f] Generics
docs.oracle.com/javase/tutorial/java/generics/

[g] Warnung mit dem Stichwort Heap Pollution
docs.oracle.com/javase/tutorial/java/generics/nonReifiableVarargsType.html#heap_pollution

[h] Project Coin: Allow @SafeVarargs on private methods
bugs.openjdk.java.net/browse/JDK-7196160

[i] Nicolai Parlog; How Java 9 and Project Jigsaw May Break Your Code
blog.codefx.org/java/dev/how-java-9-and-project-jigsaw-may-break-your-code/

[j] Nicolai Parlog; Reflection vs Encapsulation – Stand Off in the Java Module System
www.sitepoint.com/reflection-vs-encapsulation-in-the-java-module-system/

[k] Mark Reinhold; The State of the Module System (s. Kapitel „Services“ und „Layers“)
openjdk.java.net/projects/jigsaw/spec/sotms/

[l] Project Jigsaw: Module System Quick-Start Guide
openjdk.java.net/projects/jigsaw/quick-start#linker

[m] Erweiterte Service-Loader-API
download.java.net/java/jdk9/docs/api/java/util/ServiceLoader.html

[n] Service locator pattern
en.wikipedia.org/wiki/Service_locator_pattern

[o] Beispiel zum Plattform-Logging
github.com/CodeFX-org/demo-java-9/tree/master/src/org/codefx/demo/java9/api/platform_logging

Nicolai Parlog
ist selbstständiger Softwareentwickler, Autor und Trainer. Er lebt in Karlsruhe, bloggt auf codefx.org und schreibt im Manning-Verlag „The Java 9 Module System“.

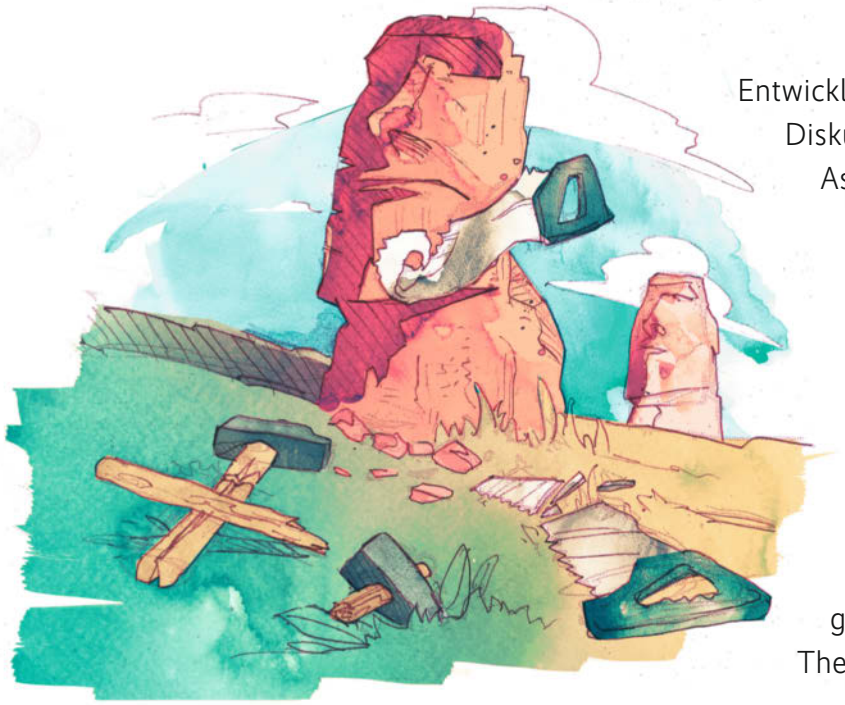
Alle Links: www.ix.de/ix1715010


JavaLand
2018
13. - 15. März 2018 in Brühl bei Köln
Ab sofort Ticket & Hotel buchen!
Call for Papers
23. Juni - 10. August
JavaLand
www.javaland.eu

Java und die Tugend der Modularisierung

Saubere Trennung

Jochen Mader



Entwickler lieben leidenschaftliche Diskussionen über technische Aspekte. Mitunter nehmen solche Auseinandersetzungen fast schon religiösen Charakter an. Das eigentliche Thema aber verschwindet hinter hitzig geführten Debatten. Die Unterstützung von Modulkonzepten auf der JVM gehört definitiv zu diesem Themenkreis.

Die modulare Anwendungsentwickung ist so etwas wie der heilige Gral der Entwicklerzunft. Das Streben nach Modularisierung ist kein neues Thema in der Java-Welt. Bereits 2000 erschien mit OSGi ein vollständiges Modulsystem mit allerlei Zusatzfeatures, das sich bis heute großer Beliebtheit erfreut. Auch die Java Enterprise Edition bietet seit 2003 ein einfaches Modulsystem. Durch die Kombination von EAR, WAR und JAR ist es möglich, Anwendungen in Module aufzuteilen und bereitzustellen. Neben diesen großen Initiativen haben sich andere Frameworks mehr oder weniger erfolgreich versucht, das Fehlen expliziter Module auf der JVM auszugleichen. JBoss Modules dürfte wohl das prominenteste Beispiel sein. Bis auf OSGi und Java EE konnte aber keines einen bleibenden Eindruck auf der JVM hinterlassen.

Schon 2005 formulierte die Java-Welt im Rahmen des JSR 277 (Java Module System) deshalb ihren Wunsch nach echter, von der JVM unterstützter Modularisierung. Fast zwölf Jahre später – die Community hatte den Glauben daran fast verloren – wird das sich seit 2008 der Modularisierung annehmende Project Jigsaw endlich zum festen Bestandteil des JDK. Es ist nicht die Absicht des Artikels zu klären, ob Jigsaw alle Anforderungen erfüllt und OSGi somit überflüssig wird (zu diesem Thema gibt es auf S. 30 den passenden Artikel). Vielmehr soll die aktuelle Aufmerksamkeit genutzt werden, generell auf Modularisierungskonzepte einzugehen und zu beleuchten, warum die grundlegenden Bausteine der JVM nicht ausreichen.

Das Modul

Zuerst ist aber zu klären, was unter einem Modul verstanden wird. Deshalb hier eine persönliche Definition als Grundlage für die weitere Diskussion:

> Ein Modul umfasst einen funktionalen Aspekt einer Anwendung. Es stellt Informationen über seine Abhängigkeiten bereit und unterscheidet zwischen öffentlichen sowie privaten Elementen.

Die primäre Aufgabe eines Moduls ist es, einen Aspekt einer Anwendung zu kapseln. Ob es sich dabei um einen technischen (Logging, REST-Lib …) oder einen fachlichen (Buchungslogik, Eingabemasken …) handelt, ist für diese Definition erst mal zweitrangig. Wichtig ist die Zahl: Nicht zwei oder drei, sondern ein Aspekt gehört hinein. Des Weiteren stellt das Modul Informationen zu seinen Abhängigkeiten bereit. Ein fachliches Modul zur Abwicklung von Buchungen wird auf ein technisches zur Interaktion mit entsprechenden Backend-Techniken (JDBC-Treiber, Middleware-Schnittstellen) zurückgreifen müssen. Module bilden also einen Abhängigkeitsgraphen, aus dem klar