Michael Inden

Java

Die Neuerungen in Version 9 bis 12

Modularisierung, Syntax- und API-Erweiterungen

dpunkt.verlag

Dipl.-Inform. Michael Inden ist Oracle-zertifizierter Java-Entwickler. Nach seinem Studium in Oldenburg hat er bei diversen internationalen Firmen in verschiedenen Rollen etwa als Softwareentwickler, -architekt, Consultant, Teamleiter sowie Trainer gearbeitet. Zurzeit ist er als CTO und Leiter Academy in Zürich tätig.

Michael Inden hat über zwanzig Jahre Berufserfahrung beim Entwurf komplexer Softwaresysteme gesammelt, an diversen Fortbildungen und mehreren Java-One-Konferenzen teilgenommen. Sein besonderes Interesse gilt dem Design qualitativ hochwertiger Applikationen mit ergonomischen GUIs sowie dem Coaching. Sein Wissen gibt er gerne als Trainer in internen und externen Schulungen und auf Konferenzen weiter, etwa bei der Java User Group Switzerland, bei der JAX/W-JAX, ch.open und den IT-Tagen.

Michael Inden

Java – die Neuerungen in Version 9 bis 12

Modularisierung, Syntax- und API-Erweiterungen

dpunkt.verlag

Michael Inden
michael_inden@hotmail.com

Lektorat: Dr. Michael Barabas
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Michael Inden
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, *www.exclam.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-86490-672-5
PDF 978-3-96088-777-5
ePub 978-3-96088-778-2
mobi 978-3-96088-779-9

1. Auflage 2019

Wieblinger Weg 17
69123 Heidelberg

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: hallo@dpunkt.de.

5 4 3 2 1 0

Inhaltsverzeichnis

Vorwort

Zunächst einmal bedanke ich mich bei Ihnen, dass Sie sich für dieses Buch entschieden haben. Hierin finden Sie eine Vielzahl an Informationen zu den Neuerungen in der aktuellen Java-Version 12 und in den Vorgängern. Aufgrund des nun halbjährlichen Releasezyklus sind in den Java-Versionen 10, 11 und 12 jeweils weniger Änderungen als in früheren Releases enthalten. In diesem Buch werden auch diverse Neuerungen aus Java 9 beschrieben, weil Java 9 das letzte große Update nach Java 8 war und eine Vielzahl an relevanten Erweiterungen mitbringt. Eine weitreichende Neuerung von Java 9 ist sicherlich die Modularisierung, die es erlaubt, eigene Programme in kleinere Softwarekomponenten, sogenannte Module, zu unterteilen. Zudem wurde auch das JDK in Module aufgeteilt.

An wen richtet sich dieses Buch?

Dieses Buch ist kein Buch für Programmierneulinge, sondern richtet sich an diejenigen Leser, die bereits solides Java-Know-how besitzen und sich nun kurz und prägnant über die wichtigsten Neuerungen in den Java-Versionen 9 bis 12 informieren wollen.

Um die Beispiele des Buchs möglichst präzise und elegant zu halten, verwende ich diverse Features aus Java 8. Deshalb setzt der Text voraus, dass Sie sich schon mit den Neuerungen von Java 8 beschäftigt haben. Alle, die eine kleine Auffrischung benötigen, finden zum leichteren Einstieg im Anhang einen Crashkurs zu Java 8. Für einen fundierten Einstieg in Java 8 möchte ich Sie auf meine Bücher »Java 8 – Die Neuerungen« [2] oder alternativ »Der Weg zum Java-Profi« [4] verweisen.

Zielgruppe

Dieses Buch richtet sich im Speziellen an zwei Zielgruppen:

1. Zum einen sind dies engagierte Hobbyprogrammierer und Informatikstudenten, aber auch Berufseinsteiger, die Java als Sprache beherrschen und an den Neuerungen in Java 9 bis 12 interessiert sind.
2. Zum anderen ist das Buch für erfahrene Softwareentwickler und -architekten gedacht, die ihr Wissen ergänzen oder auffrischen wollen, um für zukünftige Projekte abschätzen zu können, ob und – wenn ja – für welche Anforderungen die neuen Java-Versionen eine gewinnbringende Alternative darstellen können.

Was vermittelt dieses Buch?

Sie als Leser erhalten in diesem Buch neben Theoriewissen eine Vertiefung durch praktische Beispiele, sodass der Umstieg auf Java 9 bis 12 in eigenen Projekten erfolgreich gemeistert werden kann.

Ich setze zwar ein gutes Java-Grundwissen voraus, allerdings werden ausgewählte Themengebiete etwas genauer und gegebenenfalls einführend betrachtet, wenn dies das Verständnis der nachfolgenden Inhalte erleichtert.

Aufbau dieses Buchs

Nachdem Sie eine grobe Vorstellung über den Inhalt dieses Buchs haben, möchte ich die Themen der einzelnen Kapitel kurz vorstellen.

Kapitel 1 – Einleitung Die Einleitung stimmt Sie auf Java 9 bis 12 ein und gibt einen groben Überblick, was Sie so alles in diesem Buch bzw. als Neuerungen erwartet.

Kapitel 2 – Syntaxerweiterungen in JDK 9 Zunächst widmen wir uns verschiedenen Änderungen an der Syntax von Java. Neben Details zu Bezeichnern, dem Diamond Operator und Ergänzungen bei Annotations gehe ich vor allem kritisch auf das neue Feature privater Methoden in Interfaces ein.

Kapitel 3 – Neues und Änderungen in JDK 9 In den APIs des JDKs finden sich diverse Neuerungen. Dieses Potpourri habe ich thematisch ein wenig gegliedert. Neben Vereinfachungen beim Prozess-Handling, der Verarbeitung mit `Optional<T>` oder von Daten mit `InputStream`s schauen wir auf fundamentale Neuerungen im Bereich der Concurrency durch Reactive Streams.

Kapitel 4 – Änderungen in der JVM in JDK 9 In diesem Kapitel beschäftigen wir uns mit Änderungen in der JVM, etwa bei der Garbage Collection oder der Einführung der `jshell`. Auch in Bezug auf `javadoc` und der Nummerierung von Java-Versionen finden wir in Java 9 Änderungen, die thematisiert werden.

Kapitel 5 – Übungen zu den Neuerungen in JDK 9 In diesem Kapitel werden Übungsaufgaben zu den Themen der vorangegangenen Kapitel 2 bis 4 präsentiert. Deren Bearbeitung sollte Ihr Wissen zu den Neuerungen aus Java 9 vertiefen.

Kapitel 6 – Neues und Änderungen in Java 10 Seit Java 10 verfolgt man bei Oracle die Strategie, Java in kleinen, aber feinen Iterationen um nützliche Funktionalität zu ergänzen und auch in Bezug auf die Syntax zu modernisieren. In diesem Kapitel werden die Syntaxerweiterung `var` zur Definition lokaler Variablen sowie einige API-Erweiterungen vorgestellt.

Kapitel 7 – Neues und Änderungen in Java 11 Nachdem Java 9 und 10 jeweils nur 6 Monate aktuell waren, stellt Java 11 wieder ein über Jahre supportetes, sogenanntes LTS-Release dar, wobei LTS für Long Term Support steht. Neben einigen API-Erweiterungen wurden vor allem kleinere Bereinigungen im JDK vorgenommen, unter anderem sind die Module zu CORBA, JavaFX und in Teilen auch zu XML, speziell JAXB, nun nicht mehr Bestandteil des JDKs.

Kapitel 8 – Neues und Änderungen in Java 12 Zunächst waren zwei Syntaxerweiterungen für Java 12 vorgesehen. Leider die sogenannten »Raw String Literals« als Feature gestrichen, sodass für uns als Entwickler vor allem ein Preview auf Syntaxänderungen bezüglich `switch` verbleibt. Sofern Sie Optimierungen auf Mikroebene vornehmen mussten, um das letzte Quäntchen an Performance herauszuholen, interessiert es Sie bestimmt, dass das Microbenchmark-Framework JMH (Java Microbenchmarking Harness) ins JDK integriert wurde.

Kapitel 9 – Übungen zu den Neuerungen in JDK 10 und 11 In diesem Kapitel werden Übungsaufgaben zu den Themen der vorangegangenen Kapitel 6 bis 8 präsentiert. Deren Bearbeitung sollte Ihr Wissen zu den Neuerungen aus Java 10 und 11 vertiefen – wobei keine Übungen zu Java 12 angeboten werden, da hier kaum relevante API-Erweiterungen stattgefunden haben.

Kapitel 10 – Modularisierung mit Project Jigsaw Klar strukturierte Softwarearchitekturen mit sauber definierten Abhängigkeiten sind erstrebenswert, um selbst größere Softwaresysteme möglichst beherrschbar zu machen und Teile unabhängig voneinander änderbar zu halten. Seit Java 9 helfen dabei Module als eigenständige Softwarekomponenten. In diesem Kapitel wird die Thematik Modularisierung eingeführt und anhand von Beispielen vorgestellt. Im Speziellen werden auch Themen wie Sichtbarkeit und Zugriffsschutz behandelt.

Kapitel 11 – Weiterführende Themen zur Modularisierung In diesem Kapitel schauen wir uns einige fortgeschrittenere Themen zur Modularisierung an. Zunächst stelle ich Ihnen eine Alternative zum von Oracle propagierten, aber in der Praxis hinderlichen Verzeichnislayout für Module vor. Danach betrachten wir die Abhängigkeitssteuerung in größerer Tiefe: Zwar hilft die Modularisierung bei der Strukturierung eines Systems, jedoch besitzen die Module oftmals direkte Abhängigkeiten bereits zur Kompilierzeit. Wird eine losere Kopplung benötigt, so kann man dafür Services nutzen. Zudem ändern sich durch die Modularisierung ein paar Dinge bezüglich Reflection, beispielsweise lassen sich neue Eigenschaften ermitteln, etwa die Moduldaten zu einer Klasse. Verbleibt noch ein wichtiges Thema, nämlich die Migration einer bestehenden Applikation in eine modularisierte. Weil dabei doch ein paar Dinge zu beachten sind, ist diesem Thema ein ausführlicher Abschnitt gewidmet, der insbesondere die verschiedenen Arten von Modulen und ihre Eigenschaften behandelt.

Kapitel 12 – Übungen zur Modularisierung Wie für die API-Erweiterungen werden auch für die Modularisierung verschiedene Übungsaufgaben in einem Kapitel zusammengestellt.

Kapitel 13 – Build-Tools und IDEs mit Java 11 Während frühere Versionen von Java der Rückwärtskompatibilität viel Aufmerksamkeit geschenkt haben und sich dadurch die notwendigen Anpassungen in IDEs und Build-Tools in Grenzen hielten, führt Java 9 durch die Modularisierung zum ersten Mal zu einem größeren Bruch. Die neue Art und Weise, wie Module den Sourcecode strukturieren, wie Klassen geladen werden und wie Zugriffe eingeschränkt werden können, und vor allem das Verbot zum Zugriff auf interne Klassen des JDKs führen zu Inkompatibilitäten und erfordern einige Anstrengungen bei Toolherstellern. Zudem wird das JDK mittlerweile zum Teil nicht mehr rückwärtskompatibel weiter entwickelt: In JDK 11 wurden verschiedene Module aus dem JDK entfernt, wodurch externe Bibliotheken genutzt werden müssen, um etwa JAXB oder JavaFX einzubinden. Dieses Kapitel gibt einen Überblick über den derzeitigen Stand und betrachtet Java 11.

Kapitel 14 – Zusammenfassung Dieses Kapitel fasst die Themen rund um die vielfältigen Neuerungen aus Java 9, 10, 11 und 12 noch einmal kurz zusammen.

Anhang A – Schnelleinsteg in Java 8 In Anhang A werden für dieses Buch wesentliche Ergänzungen aus Java 8 rekapituliert. Das erleichtert Ihnen das Verständnis der Neuerungen in aktuellen Java-Versionen, selbst dann, wenn Sie sich noch nicht eingehend mit Java 8 beschäftigt haben. Neben einer Vorstellung der funktionalen Programmierung mit Lambdas widmen wir uns den Streams, einer wesentlichen Neuerung in JDK 8 zur Verarbeitung von Daten. Abgerundet wird Anhang A durch einen kurzen Blick auf das Date and Time API und verschiedene API-Erweiterungen.

Anhang B – Einführung Gradle Anhang B liefert eine kurze Einführung in das Build-Tool Gradle, mit dem die Beispiele dieses Buchs übersetzt wurden. Mit dem vermittelten Wissen sollten Sie dann auch kleinere eigene Projekte mit einem Build-System ausstatten können.

Anhang C – Einführung Maven In diesem Anhang wird Maven als Build-Tool kurz vorgestellt. Derzeit bietet es die beste Unterstützung für modularisierte Applikationen in Java. Zudem kann man Maven-Projekte einfach in gängige IDEs importieren.

Sourcecode und ausführbare Programme

Um den Rahmen des Buchs nicht zu sprengen, stellen die Listings häufig nur Ausschnitte aus lauffähigen Programmen dar, wobei wichtige Passagen zum besseren Verständnis mitunter fett hervorgehoben sind. Auf der Webseite zu diesem Buch

www.dpunkt.de/java-die-neuerungen steht dann der vollständige, kompilierbare Sourcecode zu den Programmen zum Download bereit. Neben dem Sourcecode befindet sich auf der Webseite auch ein Eclipse-Projekt, über das sich alle Programme ausführen lassen. Idealerweise nutzen Sie dazu Eclipse 2018-12 oder IntelliJ 2018.3.

Ergänzend wird die Datei build.gradle mitgeliefert, die den Ablauf des Builds für Gradle beschreibt. Dieses Build-Tool besitzt viele Vorzüge, wie die kompakte und gut lesbare Notation, und vereinfacht die Verwaltung von Abhängigkeiten enorm. Gradle erlaubt aber auch das Starten von Programmen, wobei der jeweilige Programmname in Kapitälchenschrift, etwa DATETIMEEXAMPLE, angegeben wird.

Blockkommentare in Listings Beachten Sie bitte, dass sich in den Listings diverse Blockkommentare finden, die der Orientierung und dem besseren Verständnis dienen. In der Praxis sollte man derartige Kommentierungen mit Bedacht einsetzen und lieber einzelne Sourcecode-Abschnitte in Methoden auslagern. Für die Beispiele des Buchs dienen diese Kommentare aber als Anhaltspunkte, weil die eingeführten oder dargestellten Sachverhalte für Sie als Leser vermutlich noch neu und ungewohnt sind.

```
public static void main(final String[] args) throws InterruptedException,
                                                     IOException
{
    // Prozess erzeugen
    final String command = "sleep 60s";
    final String commandWin = "cmd timeout 60";
    final Process sleeper = Runtime.getRuntime().exec(command);
    ...

    // Process => ProcessHandle
    final ProcessHandle sleeperHandle = ProcessHandle.of(sleeper.pid()).
                                        orElseThrow(IllegalStateException::new);
    ...
}
```

Konventionen

Verwendete Zeichensätze

In diesem Buch gelten folgende Konventionen bezüglich der Schriftart: Neben der vorliegenden Schriftart werden wichtige Textpassagen *kursiv* oder ***kursiv und fett*** markiert. Englische Fachbegriffe werden eingedeutscht großgeschrieben, etwa Event Handling. Zusammensetzungen aus englischen und deutschen (oder eingedeutschten) Begriffen werden mit Bindestrich verbunden, z. B. Plugin-Manager. Namen von Programmen und Entwurfsmustern werden in KAPITÄLCHEN geschrieben. Listings mit Sourcecode sind in der Schrift Courier gesetzt, um zu verdeutlichen, dass dies einen Ausschnitt aus einem Java-Programm darstellt. Auch im normalen Text wird für Klassen, Methoden, Konstanten und Parameter diese Schriftart genutzt.

Tipps und Hinweise aus der Praxis

Dieses Buch ist mit diversen Praxistipps gespickt. In diesen werden interessante Hintergrundinformationen präsentiert oder es wird auf Fallstricke hingewiesen.

Tipp: Praxistipp

In derart formatierten Kästen finden sich im späteren Verlauf des Buchs immer wieder einige wissenswerte Tipps und ergänzende Hinweise zum eigentlichen Text.

Verwendete Klassen aus dem JDK

Werden Klassen des JDKs erstmalig im Text erwähnt, so wird deren voll qualifizierter Name, d. h. inklusive der Package-Struktur, angegeben: Die Klasse `String` würde demnach als `java.lang.String` notiert – alle weiteren Nennungen erfolgen dann ohne Angabe des Package-Namens. Diese Regelung erleichtert initial die Orientierung und ein Auffinden im JDK und zudem wird der nachfolgende Text nicht zu sehr aufgebläht. Die voll qualifizierte Angabe hilft insbesondere, da in den Listings eher selten `import`-Anweisungen abgebildet werden.

Im Text beschriebene Methodenaufrufe enthalten in der Regel die Typen der Übergabeparameter, etwa `substring(int, int)`. Sind die Parameter in einem Kontext nicht entscheidend, wird mitunter auf deren Angabe aus Gründen der besseren Lesbarkeit verzichtet – das gilt ganz besonders für Methoden mit generischen Parametern.

Verwendete Abkürzungen

Im Buch verwende ich die in der nachfolgenden Tabelle aufgelisteten Abkürzungen. Weitere Abkürzungen werden im laufenden Text in Klammern nach ihrer ersten Definition aufgeführt und anschließend bei Bedarf genutzt.

Abkürzung	Bedeutung
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
(G)UI	(Graphical) User Interface
IDE	Integrated Development Environment
JDK	Java Development Kit
JLS	Java Language Specification
JRE	Java Runtime Environment
JSR	Java Specification Request
JVM	Java Virtual Machine

Danksagung

Ein Fachbuch zu schreiben ist eine schöne, aber arbeitsreiche und langwierige Aufgabe. Alleine kann man dies kaum bewältigen. Daher möchte ich mich an dieser Stelle bei allen bedanken, die direkt oder indirekt zum Gelingen des Buchs beigetragen haben. Insbesondere konnte ich bei der Erstellung des Manuskripts auf ein starkes Team an Korrekturlesern zurückgreifen. Es ist hilfreich, von den unterschiedlichen Sichtweisen und Erfahrungen profitieren zu dürfen.

Zunächst einmal möchte ich mich bei Michael Kulla, der als Trainer für Java SE und Java EE bekannt ist, für sein mehrmaliges, gründliches Review vieler Kapitel und die fundierten Anmerkungen bedanken.

Nachfolgende Danksagung bezieht sich auf den Java-9-Teil sowie die Texte zur Modularisierung. Merten Driemeyer, Dr. Clemens Gugenberger, Prof. Dr. Carsten Kern sowie Andreas Schöneck haben mit verschiedenen hilfreichen Anmerkungen zu einer Verbesserung beigetragen. Zudem hat Ralph Willenborg dort mal wieder ganz genau gelesen und so diverse Tippfehler gefunden. Vielen Dank dafür! Auch von Albrecht Ermgassen erhielt ich den einen oder anderen Hinweis.

Schließlich bedanke ich mich bei einigen ehemaligen Arbeitskollegen der Zühlke Engineering AG: Jeton Memeti und Marius Reusch trugen durch ihre Kommentare zur Klarheit und Präzisierung bei. Auch von Hermann Schnyder von der Swisscom erhielt ich ein paar Anregungen.

Ebenso geht ein Dankeschön an das Team des dpunkt.verlags (Dr. Michael Barabas, Martin Wohlrab, Miriam Metsch und Birgit Bäuerlein) für die tolle Zusammenarbeit. Außerdem möchte ich mich bei Torsten Horn für die fundierte fachliche Durchsicht sowie bei Ursula Zimpfer für ihre Adleraugen beim Copy-Editing bedanken.

Abschließend geht ein lieber Dank an meine Frau Lilija für ihr Verständnis und die Unterstützung. Glücklicherweise musste sie beim Entstehen dieses Buchs zu den Neuerungen in Java 9 bis 12 einen weit weniger gestressten Autor ertragen, als dies früher beim Schreiben meines Buchs »Der Weg zum Java-Profi« der Fall war.

Anregungen und Kritik

Trotz großer Sorgfalt und mehrfachen Korrekturlesens lassen sich missverständliche Formulierungen oder sogar Fehler leider nicht vollständig ausschließen. Falls Ihnen etwas Derartiges auffallen sollte, so zögern Sie bitte nicht, mir dies mitzuteilen. Gerne nehme ich auch sonstige Anregungen oder Verbesserungsvorschläge entgegen. Kontaktieren Sie mich bitte per Mail unter:

michael_inden@hotmail.com

Zürich, im Februar 2019
Michael Inden

1 Einleitung

Während früher die Java-Releases aufgrund unfertiger Features häufig verschoben wurden, hat Oracle mit Java 10 auf halbjährliche Releases umgestellt, die jeweils die bis zu diesem Zeitpunkt fertig implementierten Features bereitstellen. Dadurch wurden sowohl Java 10 als auch Java 11 beide pünktlich im Abstand von rund 6 Monaten veröffentlicht und dies wird vermutlich auch für Java 12 gelten, dessen Releasetermin für den 19. März 2019 geplant ist.

Während diese schnelle Releasefolge eine größere Herausforderung für Toolhersteller ist, kann dies für uns als Entwickler aber positiv sein, weil wir potenziell weniger lang auf neue Features warten müssen. Das konnte früher recht mühsam sein, wie die letzten Jahre gezeigt haben.

Rund 3,5 Jahre nach dem Erscheinen von JDK 8 am 18. März 2014 ging Java mit Version 9 im September 2017 an den Start. Wieder einmal musste die Java-Gemeinde auf die Veröffentlichung der Version 9 des JDKs länger warten – es gab gleich mehrere Verschiebungen, zunächst von September 2016 auf März 2017, dann auf Juli 2017 und schließlich auf September 2017. Aber immerhin hat sich das Warten gelohnt: Neben diversen Verbesserungen im JDK selbst lag bei Java 9 der Hauptfokus auf der Modularisierung, die eine verlässliche Konfiguration und besser strukturierte Programme mit klaren Abhängigkeitsbeziehungen begünstigt.[1]

Was erwartet Sie im Folgenden?

Dieses Buch gibt einen Überblick über diverse wesentliche Erweiterungen in den JDKs 9, 10, 11 und 12. Es werden unter anderem folgende Themen behandelt:

API- und Syntaxerweiterungen Wir schauen uns verschiedene Änderungen an der Syntax von Java an. Neben Erweiterungen bei der `@Deprecated`-Annotation widmen wir uns Details zu Bezeichnern, dem Diamond Operator und vor allem gehe ich kritisch auf das Feature privater Methoden in Interfaces ein. Für Java 10 und 11 thematisiere ich die Syntaxerweiterung `var` als Möglichkeit zur Definition lokaler Variablen bzw. zur Verwendung in Lambdas.

Kommen wir zu den APIs: In Java 9 wurden diverse APIs ergänzt oder neu eingeführt. Auch Bestehendes, wie z. B. das Stream-API oder die Klasse `Optional<T>`,

[1] Allerdings sollte man bedenken, dass sowohl die funktionale Programmierung mit Lambdas als auch die Modularisierung bereits für JDK 7 angekündigt waren.

wurde um Funktionalität ergänzt. Neben Vereinfachungen beim Prozess-Handling, der Verarbeitung mit `Optional<T>` oder von Daten mit `InputStream`s schauen wir auf fundamentale Neuerungen im Bereich der Concurrency durch Reactive Streams. Darüber hinaus enthalten Java 10 und 11 eine Vielzahl kleinerer weiterer Neuerungen. Eine größere Änderung ist der mit Java 11 offiziell ins JDK 11 aufgenommene HTTP/2-Support. Eher schmerzlich könnte der Wegfall verschiedener Funktionalitäten, etwa rund um JAXB sowie JavaFX, aufgenommen werden. In Java 12 wurden leider die sogenannten »Raw String Literals« als Feature gestrichen, sodass für uns als Entwickler vor allem ein Preview auf Syntaxänderungen bezüglich `switch` verbleibt.

JVM-Änderungen In jeweils eigenen Abschnitten beschäftigen wir uns mit Änderungen in der JVM, die in den neuen Java-Versionen enthalten sind, für JDK 9 etwa in Bezug auf die Nummerierung von Java-Versionen oder `javadoc`. Zudem kann für Quereinsteiger und Neulinge die durch das Tool `jshell` bereitgestellte Java-Konsole mit REPL-Unterstützung (Read-Eval-Print-Loop) erste Experimente und Gehversuche erleichtern, ohne dafür den Compiler oder eine IDE bemühen zu müssen. Mit Java 11 kommt ein neuer Garbage Collector, mit dem Flight Recoder ein neues Tool sowie mit dem Feature »Launch Single-File Source-Code Programs« die Möglichkeit, Java-Klassen ohne explizite vorherige Kompilierung ausführen zu lassen und somit für Scripting einsetzen zu können. Mit Java 12 kann man die Integration des Microbenchmark-Frameworks JMH (Java Microbenchmarking Harness) als wesentliche Neuerung ansehen.

Modularisierung Die Modularisierung adressiert zwei typische Probleme größerer Java-Applikationen. Zum einen ist dies die sogenannte JAR-Hell, womit gemeint ist, dass sich im `CLASSPATH` verschiedene JARs mit zum Teil inhaltlichen Überschneidungen (unterschiedliche Versionen mit Abweichungen in Packages oder gleiche Klassen, aber anderem Bytecode) befinden. Dabei kann aber nicht sichergestellt werden, wann welche Klasse aus welchem JAR eingebunden wird. Zum anderen sind als `public` definierte Typen beliebig von anderen Packages aus zugreifbar. Das erschwert die Kapselung. Mit JDK 9 kann man nun eigenständige Softwarekomponenten (Module) mit einer Sichtbarkeitssteuerung definieren. Das hat allerdings weitreichende Konsequenzen: Sofern man Module verwendet, lassen sich Programme mit JDK 9 nicht mehr ohne Weiteres wie gewohnt starten, wenn diese externe Abhängigkeiten besitzen. Das liegt vor allem daran, dass Abhängigkeiten nun beim Programmstart geprüft und dazu explizit beschrieben werden müssen.

Es gibt aber einen rein auf dem `CLASSPATH` basierenden Kompatibilitätsmodus, der ein Arbeiten wie bis einschließlich JDK 8 gewohnt ermöglicht.

Tipp: Beispiele und der Kompatibilitätsmodus

Zum Ausprobieren einiger Neuerungen aus JDK 9, 10, 11 und 12 werden wir kleine Beispielapplikationen in `main()`-Methoden erstellen. Dabei ist es für erste Experimente und für die Migration bestehender Anwendungen von großem Vorteil, dass man das an sich modularisierte JDK auch ohne eigene Module und ihre Sichtbarkeitsbeschränkungen betreiben kann. In diesem Kompatibilitätsmodus wird wie zuvor bei Java 8 mit `.class`-Dateien, JARs und dem `CLASSPATH` gearbeitet. Für zukünftige Projekte wird man aber mitunter Module nutzen wollen. Das schauen wir uns in eigenen Kapiteln an.

Entdeckungsreise JDK 9, 10, 11 und 12 – Wünsche an die Leser

Ich wünsche allen Lesern viel Freude mit diesem Buch sowie einige neue Erkenntnisse und viel Spaß beim Experimentieren mit JDK 9, 10, 11 und 12. Möge Ihnen der Umstieg auf die neue Java-Version und die Erstellung modularer Applikationen oder die Migration bestehender Anwendungen durch die Lektüre meines Buchs leichter fallen.

Wenn Sie zunächst eine Auffrischung Ihres Wissens zu Java 8 und seinen Neuerungen benötigen, bietet sich ein Blick in den Anhang A an.

Teil I

Sprach- und API-Erweiterungen in Java 9

2 Syntaxerweiterungen in JDK 9

Bereits in JDK 7 wurden unter dem Projektnamen Coin verschiedene kleinere Syntaxerweiterungen in Java integriert. Für JDK 9 gab es ein Nachfolgeprojekt, dessen Neuerungen wir uns jetzt anschauen.

2.1 Anonyme innere Klassen und der Diamond Operator

Bei der Definition anonymer innerer Klassen konnte man den Diamond Operator bis JDK 8 leider nicht nutzen, sondern der Typ aus der Deklaration war auch bei der Definition explizit anzugeben. Praktischerweise ist es mit JDK 9 (endlich) möglich, auf diese redundante Typangabe zu verzichten. Als Beispiel dient die Definition eines Komparators mit dem Interface `java.util.Comparator<T>`.

Beispiel mit JDK 8

Bis JDK 8 musste man bei der Definition einer anonymen inneren Klasse den Typ noch wie folgt angeben:

```
final Comparator<String> byLengthJdk8 = new Comparator<String>()
{
    ...
};
```

Beispiel mit JDK 9

Die Änderung zu JDK 8 ist kaum sichtbar: Mit JDK 9 ist es nun erlaubt, die Typangabe wegzulassen und somit den Diamond Operator zu verwenden, wie wir dies von anderen Variablendefinitionen bereits gewohnt sind:

```
final Comparator<String> byLength = new Comparator<>()
{
    ...
};
```

Tipp: Alternative Definitionsvarianten von Komparatoren seit JDK 8

Für Komparatoren bietet es sich an, folgende Neuerungen aus JDK 8 zu nutzen:

1. Einen Lambda-Ausdruck

```
Comparator<String> byLength = (str1, str2) ->
                  Integer.compare(str1.length(), str2.length());
```

2. Die Methode `comparing()` aus dem Interface `Comparator<T>`

```
Comparator<String> byLength = Comparator.comparing(String::length);
```

Im Anhang A gehe ich auf einige Neuerungen aus JDK 8 ein. Dabei behandle ich unter anderem auch die Erweiterungen bei Komparatoren.

2.2 Erweiterung der `@Deprecated`-Annotation

Die `@Deprecated`-Annotation dient bekanntlich zum Markieren von obsoletem Sourcecode und besaß bislang keine Parameter. Das ändert sich mit JDK 9: Die `@Deprecated`-Annotation wurde um die zwei Parameter `since` und `forRemoval` erweitert. Die Annotation ist nun im JDK wie folgt definiert:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER,
      TYPE})
public @interface Deprecated {
    /**
     * Returns the version in which the annotated element became deprecated.
     * The version string is in the same format and namespace as the value of
     * the {@code @since} javadoc tag. The default value is the empty
     * string.
     *
     * @return the version string
     * @since 9
     */
    String since() default "";

    /**
     * Indicates whether the annotated element is subject to removal in a
     * future version. The default value is {@code false}.
     *
     * @return whether the element is subject to removal
     * @since 9
     */
    boolean forRemoval() default false;
}
```

Diese Erweiterung wurde nötig, weil in Zukunft geplant ist, veraltete Funktionalität aus dem JDK zu entfernen, statt sie – wie bislang für Java üblich – aus Rückwärtskompati-

bilitätsgründen ewig beizubehalten. Das folgende Beispiel zeigt eine Anwendung, wie sie aus dem JDK stammen könnte:

```
@Deprecated(since = "1.5", forRemoval = true)
```

Mithilfe der neuen Parameter kann man für veralteten Sourcecode angeben, in welcher Version (`since`) dieser mit der Markierung als `@Deprecated` versehen wurde und ob der Wunsch besteht, die markierten Sourcecode-Teile in zukünftigen Versionen zu entfernen (`forRemoval`). Weil beide Parameter Defaultwerte besitzen (`since = ""` und `forRemoval = false`), können die Angaben jeweils für sich alleine stehen oder ganz entfallen.

Diese Erweiterung der `@Deprecated`-Annotation kann man selbstverständlich auch für eigenen Sourcecode nutzen und so anzeigen, dass gewisse Funktionalitäten für die Zukunft nicht mehr angeboten werden sollen. Darüber hinaus empfiehlt es sich, in einem Javadoc-Kommentar das `@deprecated`-Tag zu verwenden und dort den Grund der Deprecation und eine empfohlene Alternative aufzuführen. Nachfolgend ist dies exemplarisch für eine veraltete Methode `someOldMethod()` gezeigt:

```
/**
 * @deprecated this method is replaced by someNewMethod()
 * ({@link #someNewMethod()}) which is more stable
 */
@Deprecated(since = "7.2", forRemoval = true)
private static void someOldMethod()
{
    // ...
}
```

2.3 Private Methoden in Interfaces

Allgemein bekannt ist, dass Interfaces der Definition von Schnittstellen dienen. Leider verlieren in Java die Interfaces immer mehr von ihrer eigentlichen Bedeutung. Unter anderem wurden mit JDK 8 statische Methoden und Defaultmethoden in Interfaces erlaubt. Mit beiden kann man Implementierungen in Interfaces vorgeben.[1] Das führt allerdings dazu, dass sich Interfaces kaum mehr von einer abstrakten Klasse unterscheiden: Abstrakte Klassen können ergänzend einen Zustand in Form von Attributen besitzen, was in Interfaces (noch) nicht geht.

Mit JDK 9 wurde der Unterschied zwischen Interfaces und abstrakten Klassen nochmals verringert, weil nun auch die Definition privater Methoden in Interfaces erlaubt ist. Das Argument dafür war, dass sich damit die Duplikation von Sourcecode in Defaultmethoden reduzieren ließe. Das mag richtig sein. Allerdings ist es für die meisten Anwendungsprogrammierer eher fraglich, ob diese jemals Defaultmethoden selbst

[1] Dieser Schritt war designtechnisch nicht schön, aber nötig, um Rückwärtskompatibilität und doch Erweiterbarkeit zu erreichen und um vor allem die Neuerungen im Bereich der Streams nahtlos ins JDK 8 integrieren zu können.

implementieren sollten. Trotz dieser Kritik möchte ich Ihnen das Feature anhand eines Beispiels vorstellen, da es eventuell für Framework-Entwickler von Nutzen sein kann.

Beispiel

Schauen wir uns zur Demonstration privater Methoden in Interfaces das nachfolgende Listing und vor allem die private Methode `myPrivateCalcSum(int, int)` sowie deren Aufruf aus den beiden öffentlichen Defaultmethoden an:

```
public interface PrivateMethodsExample
{
    // Tatsächliche Schnittstellendefinition - public abstract ist optional
    public abstract int method1();
    public abstract String method2();

    public default int sum(final String num1, final String num2)
    {
        final int value1 = Integer.parseInt(num1);
        final int value2 = Integer.parseInt(num2);

        return myPrivateCalcSum(value1, value2);
    }

    public default int sum(final int value1, final int value2)
    {
        return myPrivateCalcSum(value1, value2);
    }

    // Neu und unschön in JDK 9
    private int myPrivateCalcSum(final int value1, final int value2)
    {
        return value1 + value2;
    }
}
```

Kommentar

Vielleicht fragen Sie sich, warum ich den privaten Methoden in Interfaces so ablehnend gegenüberstehe. Tatsächlich wurde die Büchse der Pandora bereits mit JDK 8 und den Defaultmethoden geöffnet. Die privaten Methoden mögen für Framework-Entwickler mitunter praktisch sein, jedoch besteht die Gefahr, dass sie für »normale« Entwickler noch attraktiver werden und von diesen somit ohne großes Hinterfragen zur Applikationsentwicklung eingesetzt werden. Das wäre aber im Hinblick auf das Design und die Klarheit von Business-Applikationen ein Schritt in die falsche Richtung.[2] Dadurch wird unter Umständen dem Schnittstellenentwurf weniger Aufmerksamkeit gewidmet, basierend auf der Annahme, dass benötigte Funktionalität immer noch nachträglich hinzugefügt werden kann.

[2] Dieser Nachteil verliert durch Nutzung einer modernen Microservice-Architektur etwas an Gewicht, da die Designsünde dann relativ isoliert existiert.

2.4 Verbotener Bezeichner '_'

Bei den Bezeichnern gibt es eine kleine Änderung: Der Compiler erlaubt mit JDK 9 das Zeichen _ (Unterstrich) nicht mehr als Bezeichner.

Während folgende Zeile mit JDK 8 noch kompilierte

```
final String _ = "Underline";
```

produziert der Java-Compiler mit JDK 9 folgende Fehlermeldung:

```
as of release 9, '_' is a keyword, and may not be used as an identifier
```

Ich persönlich halte ein einzelnes Zeichen als Variablenbezeichner fast immer für einen Bad Smell und insbesondere gilt dies für den Unterstrich. Vermutlich sehen Sie dies ähnlich. Insofern stellt diese Änderung wohl eher selten ein Problem dar.

3 Neues und Änderungen in JDK 9

Nachdem wir diverse kleinere Änderungen in der Syntax der Sprache Java kennengelernt haben, wollen wir uns in diesem Kapitel relevante Erweiterungen im JDK anschauen. Erwähnenswert sind das neue Process-API, die Ergänzungen im Stream-API sowie in `java.util.Optional<T>`, aber auch die neuen Collection-Factory-Methoden. Darüber hinaus findet man Ergänzungen in den Klassen `java.io.InputStream` und `java.util.ResourceBundle` sowie diverse Neuerungen, unter anderem auch im Bereich Concurrency: Die Klasse `java.util.concurrent.CompletableFuture<T>` bietet mehr Funktionalität. Wesentlicher ist vermutlich aber die Unterstützung von Reactive Streams durch die im Package `java.util.concurrent` neu eingeführte Klasse `Flow`. Auch im Bereich von Unicode, Grafikformaten sowie Desktop-Technologie mit HiDPI-Support und Unterstützung der Taskbar bzw. des Docks hat sich einiges weiterentwickelt. Außerdem wurden Utility-Klassen wie `java.util.Objects` und `java.util.Arrays` sinnvoll ergänzt.

3.1 Neue und erweiterte APIs

Wie bereits angedeutet, wurden in den APIs des JDKs diverse Verbesserungen vorgenommen und Neuerungen integriert, die wir nun thematisch gruppiert anschauen wollen. Dabei beginnen wir mit den Neuerungen im Process-API.

3.1.1 Das neue Process-API

Bis einschließlich JDK 8 sind die Möglichkeiten recht eingeschränkt, wenn es darum geht, Prozesse des Betriebssystems zu kontrollieren und zu verwalten.

PID mit JDK 8 ermitteln

Ein simples Beispiel ist die Ermittlung der ID eines Prozesses, kurz PID genannt. Je nach Plattform muss man dies mit Java 8 unterschiedlich implementieren. Für Linux und Mac OS führt man dazu ein Shell-Kommando mit der Methode `exec()` aus der Klasse `java.lang.Runtime` aus – das korrespondierende Windows-Kommando unter Einsatz der Powershell ist als `commandsWin` definiert:

```
private static long getPidJdk8Style() throws InterruptedException,
                                             IOException
{
    // $PPID steht für Parent Process ID, also hier derjenigen der JVM
    final String[] commands = new String[]{ "/bin/sh", "-c", "echo $PPID" };

    // Komplexere Windows-Variante
    final String[] commandsWin = new String[]{ "powershell",
                          "gwmi win32_process | ?{$_.ProcessID -eq $pid} | " +
                          "select ParentProcessID | fw -c 2"};

    // Für Windows: commandsWin nutzen
    final Process proc = Runtime.getRuntime().exec(commands);

    if (proc.waitFor() == 0)
    {
        try (final InputStream in = proc.getInputStream())
        {
            final int available = in.available();
            final byte[] outputBytes = new byte[available];
            in.read(outputBytes);

            final String pid = new String(outputBytes);
            // rein theoretisch wäre hier eine NumberFormatException abzufangen
            return Long.parseLong(pid.trim());
        }
    }
    throw new IllegalStateException("PID is not accessible");
}
```

Im Listing wird ersichtlich, wie aufwendig das Auslesen der Informationen aus dem `InputStream` des erzeugten Prozesses ist und dass sogar rein theoretisch der Fall behandelt werden müsste, dass keine gültige Zahl zurückgeliefert wird – was wohl nur in Ausnahmesituationen bei Lesefehlern der Streams geschehen könnte und was wir hier der Vereinfachung halber außer Acht lassen.

Hinweis: Die Klasse `Runtime`

Java ist weitestgehend betriebssystemunabhängig. Manchmal ist es aber wünschenswert, externe Programme in Form von Prozessen auszuführen. Dies ist durch den Aufruf von `exec()` der Klasse `Runtime` möglich. Dadurch entsteht ein neues `java.lang.Process`-Objekt. Mit dessen Methode `waitFor()` kann man ***blockierend*** auf das Ende des Prozesses warten und anschließend mit der Methode `exitValue()` den Rückgabewert erfragen.

Zudem erhält man über `getOutputStream()`, `getInputStream()` und `getErrorStream()` Zugriff auf die dem `Process` zugeordneten Ein- und Ausgabeströme. Dabei sind allerdings einige Details zu beachten. Zunächst ist der Zugriff auf diese Streams insofern wichtig, als dass der erzeugte Prozess keine Konsole zur Ausgabe hat und dadurch die Ausgaben auf `System.out` an den Vaterprozess weitergeleitet werden. Auch kann man Eingaben an einen Subprozess weiterleiten. Dabei empfiehlt es sich, möglichst zeitnah aus den Streams zu lesen und nicht erst nach Terminierung des Prozesses, da es ansonsten zu Blockierungen und Deadlocks kommen kann.

PID mit JDK 9 ermitteln

Die Abfrage der PID mit Java 9 wird mithilfe der Klasse `java.lang.ProcessHandle` nun deutlich kürzer, besser lesbar und verständlich:

```
private static long getPidJdk9Style()
{
    return ProcessHandle.current().pid();
}
```

Neben den genannten Vorteilen bietet die Methode `pid()` einen betriebssystemunabhängigen Weg zur Ermittlung der Prozess-ID (zumindest aus Sicht des Aufrufers).

Beispielprogramm

Wir wollen die beiden Methoden in Aktion erleben und deren Ergebnis prüfen. Dazu schreiben wir folgendes Programm:

```
public static void main(final String[] args) throws InterruptedException,
                                                     IOException
{
    System.out.println("PID old: " + getPidJdk8Style());
    System.out.println("PID new: " + getPidJdk9Style());
}
```

Listing 3.1 *Ausführbar als* **'PIDEXAMPLE'**

Startet man das Programm PIDEXAMPLE, so sieht man anhand der Ausgabe identischer Prozess-IDs, dass beide Varianten funktional übereinstimmen, beispielsweise wie folgt:

```
PID old: 41948
PID new: 41948
```

Das Interface `ProcessHandle`

Neben der PID kann man mithilfe von `ProcessHandle` noch diverse weitere Informationen zu Prozessen auslesen. Dazu gibt es unter anderem folgende Methoden:

- `current()` – Ermittelt den aktuellen Prozess als `ProcessHandle`.
- `info()` – Stellt Infos zum Prozess in Form des inneren Interface `ProcessHandle.Info` bereit, etwa zu Benutzer, Kommando usw.
- `info().command()` – Gibt das Kommando als `Optional<String>` aus einem `ProcessHandle.Info` zurück.
- `info().user()` – Liefert den Benutzer als `Optional<String>` aus einem `ProcessHandle.Info`.
- `info().totalCpuDuration()` – Ermittelt aus den Infos die benötigte CPU-Zeit als `Optional<Duration>`. Die Klasse `java.time.Duration` entstammt dem mit JDK 8 neu eingeführten Date and Time API.[1]

[1] Einen Kurzüberblick für dieses Datums-API bietet Anhang A.

Zum besseren Verständnis dieser Methoden betrachten wir ein Beispiel:

```
public static void main(final String[] args)
{
    final ProcessHandle current = ProcessHandle.current();
    printInfo(current);
}

private static void printInfo(final ProcessHandle current)
{
    System.out.println("PID:       " + current.pid());
    System.out.println("Info:      " + current.info());
    System.out.println("Command:   " + current.info().command());
    System.out.println("CPU-Usage: " + current.info().totalCpuDuration());
}
```

Listing 3.2 *Ausführbar als* **'PROCESSHANDLEEXAMPLE'**

Das Programm PROCESSHANDLEEXAMPLE gibt in etwa Folgendes aus (gekürzt):

```
PID:       13670
Info:      [user: Optional[michaeli], cmd: /Library/Java/JavaVirtualMachines/jdk
      -9.jdk/Contents/Home/bin/java, args: [-Dfile.encoding=UTF-8, -Duser.country
      =DE, -Duser.language=de, -Duser.variant, -cp, /Users/michaeli/Desktop/
      PureJava9/quelltext/build/libs/Java9-all.jar:/Users/michaeli/Desktop/
      PureJava9/quelltext/build/requiredLibs, ch3_1.processapi.
      ProcessHandleExample], startTime: Optional[2017-04-06T17:21:56.927Z],
      totalTime: Optional[PT0.230888S]]
Command:   Optional[/Library/Java/JavaVirtualMachines/jdk-9.jdk/Contents/Home/
      bin/java]
CPU-Usage: Optional[PT0.308141S]
```

Neben der PID werden diverse Informationen aus dem `Info`-Objekt aufgelistet, exemplarisch separat nochmals die Werte für `command()` und `totalCpuDuration()`. Für das mit `command()` als `Optional<String>` ermittelte Kommando erkennen wir, dass es sich um das Programm `java` aus JDK 9 handelt, das laut `totalCpuDuration()` etwa 0.31 Sekunden CPU-Zeit verbraucht hat, wie man anhand von `Optional<Duration>` sieht.

Alle Prozesse mit `ProcessHandle` abfragen

Neben Informationen zum aktuellen Prozess lassen sich Informationen für alle Prozesse des Benutzers sowie alle Subprozesse zu einem Prozess wie folgt ermitteln:

- `allProcesses()` – Liefert alle Prozesse als `Stream<ProcessHandle>`.
- `children()` – Ermittelt zu einem Prozess alle seine (direkten) Subprozesse als `Stream<ProcessHandle>`.

Im nachfolgenden Beispiel iterieren wir über das Ergebnis von `allProcesses()` und geben Infos zu solchen Prozessen aus, die Subprozesse besitzen. Die Anzahl an Subprozessen können wir durch Aufruf von `children().count()` erfragen:

```
public static void main(final String[] args)
{
    System.out.println("All Processes:");
    showInfoForAllProcesses();
}

private static void showInfoForAllProcesses()
{
    ProcessHandle.allProcesses().forEach(processHandle ->
    {
        final Stream<ProcessHandle> children = processHandle.children();
        final long count = children.count();
        if (count > 0)
        {
            System.out.println("Info: " + processHandle.info() +
                               " has " + count + " children");
        }
    });
}
```

Listing 3.3 *Ausführbar als '***ALLPROCESSHANDLESEXAMPLE***'*

Das Programm ALLPROCESSHANDLESEXAMPLE produziert die folgenden Ausgaben (gekürzt), die eine Liste der zurückgelieferten Informationen widerspiegeln:

```
All Processes:
Info: [user: Optional[michaeli], cmd: /Applications/Adobe Acrobat Reader DC.app/
      Contents/MacOS/AdobeReader, args: [-psn_0_3822501], startTime: Optional
      [2016-08-02T21:16:30.322Z]] has 3 children
...
Info: [user: Optional[michaeli], cmd: /System/Library/CoreServices/Dock.app/
      Contents/MacOS/Dock, startTime: Optional[2016-07-24T08:17:12.938Z]] has 1
      children
Info: [user: Optional[root], startTime: Optional[2016-07-24T08:16:40.564Z]] has
      285 children
...
```

Prozesse mit `ProcessHandle` kontrollieren

Neben der Bereitstellung und Abfrage von Informationen zu Prozessen existieren auch verschiedene Möglichkeiten, Prozesse zu beenden sowie auf das Ende eines Prozesses zu reagieren. Dazu findet man im Interface `ProcessHandle` folgende Methoden:

- `of(long)` – Liefert ein `Optional<ProcessHandle>` zu einer gegebenen PID.
- `destroy()` – Terminiert einen Prozess, sofern dies erlaubt ist. Ansonsten, etwa für den mit `current()` ermittelten Prozess, wird eine Exception ausgelöst:

  ```
  Exception in thread "main" java.lang.IllegalStateException: destroy of
        current process not allowed
  ```

- `onExit()` – Liefert ein `CompletableFuture<ProcessHandle>` zurück, das man dazu nutzen kann, verschiedene Aktionen als Reaktion auf das Ende eines Prozesses auszuführen.