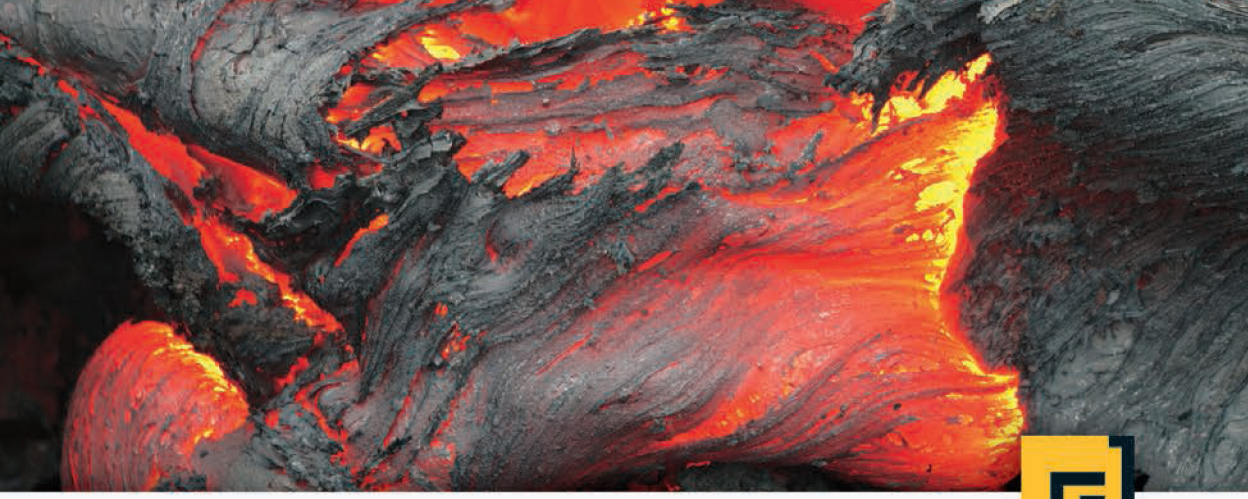

Michael Inden

Java 9

Die Neuerungen

Syntax- und API-Erweiterungen
und Modularisierung im Überblick

dpunkt.verlag

Dipl.-Inform. Michael Inden ist Oracle-zertifizierter Java-Entwickler für JDK 6. Nach seinem Studium in Oldenburg war er lange Zeit als Softwareentwickler und -architekt bei verschiedenen internationalen Firmen tätig und arbeitet derzeit als Teamleiter Softwareentwicklung in Zürich.

Michael Inden hat rund 20 Jahre Erfahrung beim Entwurf komplexer Softwaresysteme gesammelt, an diversen Fortbildungen und an mehreren Java-One-Konferenzen in San Francisco teilgenommen. Sein Wissen gibt er gerne als Trainer in Schulungen und auf Konferenzen weiter. Sein besonderes Interesse gilt dem Design qualitativ hochwertiger Applikationen mit ergonomischen, grafischen Oberflächen sowie dem Coaching von Kollegen.

Michael Inden

Java 9 – Die Neuerungen

Syntax- und API-Erweiterungen und Modularisierung im Überblick

dpunkt.verlag

Michael Inden
michael_inden@hotmail.com

Lektorat: Dr. Michael Barabas
Projektkoordination: Miriam Metsch
Technischer Review: Torsten Horn, Aachen
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Michael Inden
Herstellung: Susanne Bröckelmann
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie 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 978-3-86490-451-6
PDF 978-3-96088-378-4
ePub 978-3-96088-379-1
mobi 978-3-96088-380-7

1. Auflage 2018

Wieblinger Weg 17
69123 Heidelberg

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 der brandaktuellen Version 9 von Java. Neben einigen Detailveränderungen in der Sprache selbst gibt es in vielen APIs diverse kleine und größere praktische Neuerungen – auch solche, die die mit JDK 8 eingeführten Erweiterungen ergänzen und abrunden. Ab und an kann das neu eingeführte Programm `jshell` interessant sein, das einen REPL (Read-Eval-Print-Loop) in die JVM integriert. Das bedeutendste Feature 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 viele kleine Bausteine 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 Neuerungen in Java 9 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 auseinandergesetzt 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« [3] oder alternativ »Der Weg zum Java-Profi« [5] verweisen.

Zielgruppe

Dieses Buch richtet sich im Speziellen an zwei Zielgruppen:

1. Zum einen sind dies engagierte Hobbyprogrammierer, Informatikstudenten und Berufseinsteiger, die Java als Sprache beherrschen und an den Neuerungen in Java 9 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 Java 9 eine gewinnbringende Alternative darstellen kann.

Was vermittelt dieses Buch?

Sie als Leser erhalten in diesem Buch neben Theoriewissen eine Vertiefung durch praktische Beispiele, sodass der Umstieg auf Java 9 in eigenen Projekten erfolgreich gemeistert werden kann. Sofern hilfreich, stelle ich zunächst eine herkömmliche Lösung mit Java 8 vor und vergleiche diese dann mit der Java-9-Variante. 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 ein und nennt auch Hintergründe zur Entstehungsgeschichte.

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

Kapitel 3 – Neues und Änderungen im JDK 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. Darüber hinaus enthält Java 9 eine Vielzahl weiterer Neuerungen, beispielsweise im Bereich Desktop und Unterstützung von HiDPI und Multi-Resolution Images.

Kapitel 4 – Änderungen in der JVM In diesem Kapitel beschäftigen wir uns mit ein paar Änderungen in der JVM, die im JDK 9 enthalten sind, 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 – 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. Dazu wird Java 9 um Module als eigenständige Softwarekomponenten erweitert. In diesem Kapitel wird die Thematik Modularisierung zunächst eingeführt und anhand von Beispielen vorgestellt. Im Speziellen werden auch Themen wie Sichtbarkeit und Zugriffsschutz behandelt.

Kapitel 7 – Weiterführende Themen zur Modularisierung In diesem Kapitel schauen wir uns einige fortgeschrittenere Themen zur Modularisierung an. Zwar hilft die Modularisierung bei der Strukturierung eines Systems, jedoch besitzen die Module dann auch 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. Das werden wir zur Erstellung von Tools nutzen. 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 8 – Übungen zur Modularisierung Wie für die API-Erweiterungen werden auch für die Modularisierung verschiedene Übungsaufgaben in einem Kapitel zusammengestellt.

Kapitel 9 – Build-Tools und IDEs 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. Dieses Kapitel gibt einen Überblick über den derzeitigen Stand.

Kapitel 10 – Zusammenfassung Dieses Kapitel fasst die Themen rund um die vielfältigen Neuerungen aus Java 9 noch einmal kurz zusammen. Dabei nenne ich auch einige mit JDK 9 geplante, aber leider nicht umgesetzte Sprachfeatures.

Anhang A – Schnelleinsteg in Java 8 In Anhang A werden für dieses Buch wesentliche Neuerungen aus Java 8 rekapituliert. Damit können Sie von diesem Buch selbst dann profitieren, wenn Sie sich noch nicht eingehend mit Java 8 beschäftigt haben. Neben einem Einstieg in die funktionale Programmierung mit Lambdas widmen wir uns den Streams, einer weiteren 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 können Sie dann auch kleinere eigene Projekte mit einem Build-System ausstatten.

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 derartige 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-9` 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 Oxygen 1A oder neuer, weil diese Version der IDE bereits Java 9 unterstützt. Aber auch NetBeans und IntelliJ IDEA sind gute Alternativen.

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.

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 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 Swissom 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 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 November 2017
Michael Inden

1 Einleitung

Rund 3,5 Jahre nach dem Erscheinen von JDK 8 am 18. März 2014 geht nun Java mit Version 9 im September 2017 an den Start. Wieder einmal hat die Java-Gemeinde auf die Veröffentlichung der neuen Version des JDKs etwas länger warten müssen. Zuletzt wurde sie von September 2016 auf März 2017, dann auf Juli 2017 und schließlich auf September 2017 und damit rund 1 Jahr verschoben. Aber das Warten hat sich gelohnt: Neben diversen Verbesserungen im JDK selbst liegt der Hauptfokus auf der Modularisierung, die eine verlässliche Konfiguration und besser strukturierte Programme mit klaren Abhängigkeitsbeziehungen begünstigt.[1]

Dieses Buch gibt einen Überblick über wesentliche Erweiterungen in JDK 9. Es werden unter anderem folgende Themen behandelt:

API- und Syntaxerweiterungen Zunächst schauen wir uns verschiedene Änderungen an der Syntax von Java an. Neben Erleichterungen beim ARM (Automatic Resource Management) sowie Erweiterungen bei `@Deprecated`-Annotations widmen wir uns Details zu Bezeichnern, dem Diamond Operator und vor allem gehe ich kritisch auf das neue Feature privater Methoden in Interfaces ein.

Ebenfalls wurden diverse APIs ergänzt oder neu eingeführt. Auch Bestehendes, wie z. B. das Stream-API oder die Klasse `Optional<T>`, wurde adäquat um Funktionalität ergänzt. 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. Darüber hinaus enthält Java 9 eine Vielzahl weiterer Neuerungen, beispielsweise im Bereich Desktop und zur Unterstützung von HiDPI und Multi-Resolution Images.

JVM-Änderungen In einem eigenen Kapitel beschäftigen wir uns mit Änderungen in der JVM, die im JDK 9 enthalten sind, etwa bei der Garbage Collection oder in Bezug auf `javadoc` und der Nummerierung von Java-Versionen. 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.

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

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.

Entdeckungsreise JDK 9 – 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. Möge Ihnen der Umstieg auf die neue Java-Version und die Migration bestehender Anwendungen in modulare Applikationen 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.

Tipp: IDE-Support und Kompatbilitätsmodus

Alle derzeit relevanten IDEs, also Eclipse, IntelliJ IDEA und NetBeans, erlauben es in ihren aktuellen Versionen, Java 9 zu nutzen. Die IDEs lassen sich allerdings eventuell (noch) nicht ohne Weiteres mit JDK 9 starten. Mitunter ist es ratsam, zusätzlich zu JDK 9 ein aktuelles JDK 8 installiert zu haben. Weitere Informationen zu Konfigurationen finden Sie in Kapitel 9.

Kompatbilitätsmodus

Zum Ausprobieren einiger Neuerungen aus JDK 9 werden wir kleine Applikationen 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 9 auch ohne eigene Module und ihre Sichtbarkeitsbeschränkungen betreiben kann. In diesem Kompatbilitätsmodus wird wie zuvor bei Java 8 mit `.class`-Dateien, JARs und dem `CLASSPATH` gearbeitet. Für zukünftige Projekte wird man aber bevorzugt Module nutzen wollen. Das schauen wir uns nach der Vorstellung einiger wichtiger Änderungen in JDK 9 in eigenen Kapiteln an.

Teil I

Sprach- und API-Erweiterungen

2 Syntaxerweiterungen

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 Nutzung von »effectively final«-Variablen im ARM

Das mit JDK 7 eingeführte Automatic Resource Management (ARM), auch try-with-resources genannt, erleichtert die Arbeit mit und die Freigabe von Ressourcen, indem automatisch die `close()`-Methode der korrespondierenden Ressourcenvariablen aufgerufen wird. Damit solche Variablen in ARM nutzbar sind, müssen diese das Interface `java.lang.AutoCloseable` erfüllen und zudem unveränderlich sein. Mit JDK 9 gibt es eine Vereinfachung für den Fall, dass bereits Ressourcenvariablen außerhalb des ARM-Blocks definiert sind. Das ist jedoch eher eine Ausnahme. Häufiger findet man wohl die Definition von Ressourcenvariablen direkt im ARM, die von dieser Neuerung nicht betroffen ist.

Beispiel mit JDK 8

Im folgenden Listing sind zwei Ressourcenvariablen `bufferedIs` und `bufferedOs` außerhalb des ARM-Blocks definiert. Bis einschließlich JDK 8 musste man für derartige Szenarien beim Einsatz von ARM immer (künstlich) noch jeweils eine weitere Variable im `try`-Block definieren, selbst dann, wenn die eigentliche Ressourcenvariable bereits unveränderlich war. Das Ganze war etwas schwierig zu lesen:

```
final BufferedInputStream bufferedIs = ...;
BufferedOutputStream bufferedOs = ...

try (final BufferedInputStream bis = bufferedIs;
     BufferedOutputStream bos = bufferedOs)
{
    // ....
}
```

Im Beispiel sind bewusst die Variablen vom Typ `java.io.BufferedInputStream` als `final` und `java.io.BufferedOutputStream` als nicht `final` definiert, um zu zeigen, dass ARM sowohl mit `final`- als auch mit »effectively final«-Variablen verwendbar ist: Der Compiler kann die Unveränderlichkeit einer Variablen selbst dann erkennen, wenn diese nicht explizit `final` definiert ist, sondern lediglich ihren Wert nicht mehr ändert, also als »effectively final« zu betrachten ist.

Beispiel mit JDK 9

Für außerhalb des ARM-Blocks definierte Ressourcenvariablen wird mit JDK 9 die Syntax und die Nutzung ein wenig erleichtert. Sofern man bereits eine `final` oder »effectively final« definierte Ressourcenvariable besitzt, kann diese direkt im `try`-Block von ARM verwendet werden. Dadurch wird der Sourcecode ein wenig kürzer und man vermeidet die erneute Definition einer Variablen:

```
final BufferedInputStream bufferedIs = ...;
BufferedOutputStream bufferedOs = ...;

try (bufferedIs;
     bufferedOs)
{
    // ....
}
```

Achtung: Einsatz von ARM

In der Praxis sieht man statt des obigen Aufrufs häufig einen verschachtelten Aufruf wie folgt:

```
try (final BufferedInputStream bis =
                                new BufferedInputStream(inputStream))
```

Bei solchen Konstrukten muss man vorsichtig sein. ARM ruft die im Interface `AutoCloseable` definierte Methode `close()` nur von den explizit deklarierten Ressourcen auf, hier also `bis`. In diesem Fall funktioniert das Schließen, weil `BufferedInputStream.close()` wiederum die Methode `close()` des gewrappten `java.io.InputStream`s aufruft und zudem der Konstruktor des `BufferedInputStream`s keine Exception auslöst.

Wenn jedoch in einer ummantelnden Klasse im Konstruktor eine Exception ausgelöst werden kann, dann würde das Objekt nicht erzeugt und dessen `close()`-Methode auch niemals aufgerufen. Damit würde dann `close()` ebenfalls nicht von dem ummantelten `InputStream` aufgerufen. Um sicherzustellen, dass `close()` korrekt für alle Ressourcen aufgerufen wird, sollte die Definition wie folgt – und hier nochmals explizit gezeigt – aussehen:

```
try (final InputStream is = inputStream;
     final SomeWrappingStream wrapperStream = new SomeWrappingStream(is))
```

2.3 Neuerung bei der `@SafeVarargs`-Annotation

Mit der Annotation `@SafeVarargs` kann man dem Compiler im Falle von Vararg-Parametern eine Hilfestellung geben, wenn es Uneindeutigkeiten bei der Verwendung gibt. Was ist damit gemeint und wie kann es zu dem Problem kommen? Schauen wir uns dazu ein einführendes Beispiel an:

```
// Simple Methode mit Vararg-Parameter
private static <T> void printAll(T... elements)
{
    for (final T elem : elements)
    {
        System.out.println(elem + " of type " + elem.getClass());
    }
}

public static void main(final String[] args)
{
    // Keine Compiler-Warnung
    printAll("Hello", "Varargs");

    // Type safety: A generic array of List<String> is created
    // for a varargs parameter
    printAll(Arrays.asList("A", "B"), Arrays.asList("C", "D"));
}
```

Weil ein Vararg-Parameter intern wie ein Array gehandhabt wird und Generics und Arrays leider nicht ideal zusammenarbeiten, kommt es bei der Kombination von Generics und Varargs immer wieder zu Problemen, was oftmals – wie auch hier – zu Compiler-Warnungen führt. Das liegt insbesondere daran, dass die mit Generics angegebenen Typinformationen beim Kompilieren weitestgehend durch die sogenannte Type Erasure aus dem Bytecode entfernt werden. Darauf gehe ich im Anschluss an die Vorstellung von `@SafeVarargs` in einem Praxishinweis nochmals ein.

Wenn man als Entwickler einer Methode sicher ist, dass deren Verarbeitung ihres Vararg-Parameters keine Typprobleme verursacht, kann man die Annotation `@SafeVarargs` angeben – beachten Sie dazu bitte auch den folgenden Abschnitt »Vorsicht bei Vergabe der Annotation `@SafeVarargs`«. Für das Beispiel lässt sich die Compiler-Warnung wie folgt vermeiden:

```
@SafeVarargs
private static <T> void printAll(T... elements)
{
    for (final T elem : elements)
    {
        System.out.println(elem + " of type " + elem.getClass());
    }
}
```

Damit scheint das Problem gelöst. Doch was wäre, wenn die Methode nicht `static` wäre, sondern eine Instanzmethode? Dann käme es bis JDK 8 zu der Fehlermeldung `@SafeVarargs annotation cannot be applied to non-final instance method printAll`. Das liegt schlicht daran, dass bis JDK 8 die Annotation `@Safe-`

Varargs nur für finale oder statische Methoden verwendet werden kann, weil sich nur so Effekte durch Overriding verhindern lassen. Private Methoden kann man auch ohne explizite Angabe von final nicht überschreiben, weshalb hier auch die Annotation anwendbar sein sollte. Das wird mit JDK 9 möglich.

Neuerung in JDK 9: @SafeVarargs für private Methoden

Als Beispiel für die Neuerung in JDK 9 betrachten wir nachfolgende Hilfsmethode firstOrEmpty(T...), die das erste Element liefert, sofern das übergebene Array nicht leer ist, und ansonsten den Rückgabewert von java.util.Optional.empty():

```
@SafeVarargs
private <T> Optional<T> firstOrEmpty(T... args)
{
    if (args.length == 0)
    {
        return Optional.empty();
    }
    return Optional.of(args[0]);
}
```

Vorsicht bei Vergabe der Annotation @SafeVarargs

Prüfen Sie immer genau, ob die Annotation @SafeVarargs wirklich vergeben werden kann und dadurch tatsächlich keine Probleme ausgelöst werden. Im JDK findet man in der Onlinedokumentation folgendes Gegenbeispiel, in dem als Vararg-Parameter eine List<String>... entgegengenommen und dann mutwillig in den allgemeinsten Typ Object[] umgewandelt wird:

```
// Achtung: Gegenbeispiel
@SafeVarargs // Not actually safe!
static void m(final List<String>... stringLists)
{
   final Object[] array = stringLists;

   // Semantically invalid, but compiles without warnings
   final List<Integer> tmpList = Arrays.asList(42);
   array[0] = tmpList;

   // ClassCastException at runtime!
   final String s = stringLists[0].get(0);
}
```

Die Zuweisung einer List<Integer> wird vom Compiler zwar erlaubt, ist aber unsinnig und nicht typsicher. Für Arrays würde zur Laufzeit eine java.lang.ArrayStoreException ausgelöst.[1] Im Beispiel kommt es für die generische Liste dagegen beim Auslesen zu einer java.lang.ClassCastException.

[1] Details zur Typsicherheit mit Arrays und Generics sowie den dafür wichtigen Themen wie In-, Ko- und Kontravarianz finden Sie in meinem Buch »Der Weg zum Java-Profi« [5].

Hinweis: Fallstricke durch Type Erasure

Ich hatte zuvor angedeutet, dass Arrays und Generics in Kombination immer mal wieder für Probleme sorgen. Schauen wir zunächst eine kleine Abwandlung unseres initialen Beispiels an und rufen unsere `printAll(T...)`-Methode folgendermaßen auf:

```
public static void main(final String[] args)
{
    final List<Integer> integers = new ArrayList<>();
    integers.add(Integer.valueOf(1));
    integers.add(2);

    printAll(integers, "End", 4711);
}
```

Dadurch kommt es zu folgenden Ausgaben, die den Verlust der Typinformationen bei der `ArrayList<E>` zeigen:

```
[1, 2] of type class java.util.ArrayList
End of type class java.lang.String
4711 of type class java.lang.Integer
```

Knifflige Probleme durch Type Erasure Solange man lediglich Ausgaben auf der Konsole vornimmt, wirkt der Verlust von Typinformationen nicht so tragisch. Um ernsthaftere Probleme aufzudecken, wandeln wir das Beispiel ab. Dabei greife ich auf die Ideen aus Angelika Langers Artikel `http://www.angelikalanger.com/Articles/EffectiveJava/59.Java7.Coin2/59.Java7.Coin2.html` zurück. Wir definieren zwei Methoden, die zwar harmlos aussehen, aber jeweils Compiler-Warnungen produzieren:

```
// Type safety: Potential heap pollution via varargs parameter elements
private static <T> T[] toArray(final T... elements)
{
    return elements;
}

// Type safety: A generic array of T is created for a varargs parameter
private static <T> T[] combine(final T t1, final T t2)
{
    return toArray( t1, t2 );
}

public static void main(final String[] args)
{
    final String[] strings = toArray("bad", "karma");
    final String[] strings2 = combine("bad", "karma"); // ClassCastException
}
```

Beim Ausführen kommt es zu einer `ClassCastException` mit folgendem Hinweis: `[Ljava.lang.Object; cannot be cast to [Ljava.lang.String;`. Das wiederum liegt daran, dass durch die Type Erasure nur mit dem Typ `Object` gearbeitet wird und vor allem die Methode `combine()` beim Aufruf ein neues Array vom Typ `Object[]` wie folgt erzeugt:

```
Object[] combine(final Object  t1, final Object t2)
{
   return toArray(new Object[] {t1, t2}); // Warnung: generic array creation
}
```

Schließlich wird beim zweiten Aufruf die `ClassCastException` ausgelöst, weil der vom Compiler automatisch erzeugte Cast des zurückgelieferten `Object[]` auf ein `String[]` erfolgt:

```
final String[] strings2 = (String[]) combine("bad", "karma");
```

2.4 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 Rückwärtskompatibilitätsgründen 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()
{
    // ...
}
```

2.5 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.[2] 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 implementieren sollten. Trotz dieser Kritik möchte ich Ihnen das Feature anhand eines Beispiels vorstellen, da es eventuell für Framework-Entwickler von Nutzen sein kann.

[2] 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.