1 Einleitung

Früher wurden Java-Releases aufgrund unfertiger Features häufiger verschoben. Um dem entgegenzuwirken, hat Oracle nach dem Erscheinen von Java 9 auf einen halbjährlichen Releasezyklus umgestellt. Das erlaubt es, die jeweils bis zu diesem Zeitpunkt fertig implementierten Funktionalitäten zu veröffentlichen. Zwar kann diese schnelle Releasefolge eine größere Herausforderung für Toolhersteller sein, für uns als Entwickler ist es aber oftmals positiv, weil wir potenziell weniger lang auf neue Features warten müssen. Das konnte früher recht mühsam sein, wie die letzten Jahre gezeigt haben. Ein paar Gedanken dazu greift der nachfolgende Hinweiskasten auf.

Allerdings gilt das Positive vor allem für eigene Hobbyprojekte, weil man dort mit den Neuerungen experimentieren kann und weniger durch Restriktionen eingeschränkt ist. Im professionellen Einsatz wird man eher auf Kontinuität und die Verfügbarkeit von Security Updates setzen, weshalb in diesem Kontext vermutlich nur LTS-Versionen in Betracht kommen, um Migrationsaufwände kalkulierbar und zeitlich besser planbar zu halten.

Hinweis: Oracles neue Releasepolitik

Bis einschließlich Java 9 wurden neue Java-Versionen immer Feature-basiert veröffentlicht. Das hatte in der Vergangenheit oftmals und mitunter auch beträchtliche Verschiebungen des geplanten Releasetermins zur Folge, wenn für die Version wesentliche Features noch nicht fertig waren. Insbesondere deshalb verzögerten sich Java 8 und Java 9 um mehrere Monate bzw. sogar über ein Jahr: Rund 3,5 Jahre nach dem Erscheinen von JDK 8 im 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.

Mit einer zeitbasierten Releasestrategie möchte man derartigen Verzögerungen entgegenwirken, indem jedes halbe Jahr eine neue Java-Version veröffentlicht wird, die all jene Features enthält, die bereits fertig sind. Alle drei Jahre ist dann eine LTS-Version (Long Term Support) geplant. Eine solche ist in etwa vergleichbar mit den früheren Major-Versionen.

Was erwartet Sie im Folgenden?

Dieses Buch gibt einen fundierten Überblick über diverse wesentliche Erweiterungen in den JDKs 9 bis 14. 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. Im Kontext von `instanceof` ist es mit Java 14 möglich, künstliche Hilfsvariablen und unschöne Casts zu vermeiden.

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>`, wurde um Funktionalität erweitert. 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. In Java 12 wurden ein paar API-Erweiterungen integriert. Mit Java 13 finden wir zwei Previews auf Syntaxänderungen, nämlich einmal bezüglich `switch` und zudem die sogenannten »Text Blocks«, die mehrzeilige Strings erlauben. Mit Java 14 werden die Neuerungen bei `switch` endlich in den Sprachstandard aufgenommen. Außerdem finden wir eine hilfreiche Neuerung zur Fehleranalyse bei `NullPointerException`s. Zusätzlich bietet Java 14 zwei Erweiterungen bei der Definition mehrzeiliger Texte. Ganz besonders interessant sind sogenannte Records, die eine extrem kompakte Schreibweise zum Deklarieren spezieller Klassen mit unveränderlichen Daten ermöglichen.

JVM-Änderungen In jeweils eigenen Abschnitten beschäftigen wir uns mit Änderungen in der JVM, 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 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. Java 12 bietet als wesentliche Neuerung die Integration des Microbenchmark-Frameworks JMH (Java Microbenchmarking Harness).

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, jedoch mit anderem Bytecode) befinden. Dabei kann allerdings 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 lassen sich eigenständige Softwarekomponenten (Module) mit einer Sichtbarkeitssteuerung definieren. Das hat allerdings weitreichende Konsequenzen: Sofern man Module verwendet, kann man 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 verschiedener Neuerungen aus JDK 9 bis 14 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 mitunter Module nutzen wollen. Das schauen wir uns in eigenen Kapiteln an.

Ausprobieren der Java-14-Beispiele

Durch die kurzen Releasezyklen werden mittlerweile einige Features als Previews der Entwicklergemeinde vorgestellt. Möchte man diese nutzen, so sind in den IDEs und Build-Tools gewisse Parametrierungen sowohl beim Kompilieren als auch beim Ausführen nötig, etwa wie im folgenden Beispiel:

```
java --enable-preview -cp build/libs/Java14Examples.jar \
     java14.RecordExamples
```

Details dazu werden in Abschnitt 9.4 beschrieben.

Entdeckungsreise JDK 9 bis 14 – 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 bis 14. Möge Ihnen der Umstieg auf die von Ihnen bevorzugte 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.