Und halbjährlich grüßt das Java Release

Falk Sippach, Orientation in Objects GmbH

Mittlerweile haben wir Java-Entwickler uns an die kurzen Release-Zyklen gewöhnt. Wie geplant wurde im September 2019 das JDK 13 veröffentlicht. Und auch wenn es keine ganz großen Änderungen gibt, lohnt sich doch ein Blick auf die neue Version.

Seit Java 10 können wir uns zweimal im Jahr – im März und September – über neue Java-Releases freuen. Da natürlich in einem halben Jahr nicht so viel in der Entwicklung passieren kann, sind die jeweiligen Feature-Listen zum Glück überschaubar. Aber immerhin können wir jetzt regelmäßig wenige neue Funktionen ausprobieren, und sie werden nicht alle paar Jahre von einem riesigen Funktionsumfang neuer Features erschlagen. Das Java-Team bei Oracle bekommt zudem schnelleres und wertvolleres Feedback zu den neuen Funktionen, von denen sich einige zunächst nur im Preview-Status befinden. Auf dem Weg zum nächsten LTS (Long Term Support) Release werden sie durch Feedback der Nutzer gegebenenfalls noch angepasst. Letztlich hat man auch beim aktuellen LTS-Release (Java 11) die in Version 9 und 10 begonnenen Arbeiten finalisiert, damit es für die nächsten drei Jahre gut dasteht.

Im Jahr 2021 erscheint mit Java 17 dann die nächste Version mit längerer Support-Dauer.

Für den produktiven Einsatz seiner Java-Anwendungen kann man zwischen zwei Varianten wählen. Entweder man bleibt für drei Jahre beim aktuellen LTS-Release oder man aktualisiert zweimal im Jahr auf die jeweils für ein halbes Jahr von Oracle mit Updates versorgte Major-Version des OpenJDK. Letzteres darf man frei einsetzen, muss aber nach sechs Monaten auf die nächste Version aktualisieren, um weiterhin freie Security Patches zu erhalten. Bei der LTS-Variante gibt es sowohl kostenpflichtige Angebote (von Oracle und alternativen Herstellern) als auch freie Installationen, wie zum Beispiel von AdoptOpenJDK, Amazon Corretto, Red Hat und anderen.

Im JDK 13 wurden die folgenden Java Enhancement Proposals umgesetzt *[1]*:

- JEP 350: Dynamic CDS Archives
- JEP 351: ZGC: Uncommit Unused Memory
- JEP 353: Reimplement the Legacy Socket API
- JEP 354: Switch Expressions (Preview)
- JEP 355: Text Blocks (Preview)

```
// Switch-Statement mit break
private static String statementMultilabel(int switchArg){
     String str = "not set";
     switch (switchArg){
         case 1,2:
             str = "one or two";
             break;
         case 3:
             str = "three";
             break;
     };
     return str;
}

// Switch-Expression mit yield
private static String expressionBreakWithValue(int switchArg){
     return switch (switchArg){
         case 1, 2: yield "one or two";
         case 3: yield "three";
         default: yield "smaller than one or bigger than three";
     };
}
```

Listing 1

```
javac --release 13 --enable-preview Examples.java
java --enable-preview Examples
```

Listing 2

Dynamic CDS Archives

Bereits in Java 5 wurde Class Data Sharing (CDS) eingeführt. Bis Java 10 waren die geteilten Archive jedoch nur für den Bootstrap-Classloader zugänglich. Das Ziel von CDS ist die Verkürzung der Startzeiten von Java-Anwendungen, indem bestimmte Informationen über Klassen in sogenannten Class-Data-Sharing-Archiven abgelegt werden. Diese Daten können dann zur Laufzeit geladen und auch von mehreren JVMs benutzt werden. Mit Java 10 wurde CDS um Application-Class-Data-Sharing (AppCDS) erweitert. AppCDS ermöglicht dem eingebauten System- und Plattform-Classloader sowie benutzerdefinierten Classloadern, auf die CDS-Archive zuzugreifen. Zum Anlegen der CDS-Archive werden Klassenlisten benötigt, um die zu ladenden Klassen identifizieren zu können. Bisher mussten diese Klassenlisten durch Probeläufe der Anwendung ermittelt werden, um festzustellen, welche Klassen während der Ausführung tatsächlich geladen werden. Seit Java 12 werden Default-CDS-Archives standardmäßig mit dem JDK ausgeliefert, die auf der Klassenliste des JDK basieren.

Dynamic CDS Archives bauen nun darauf auf. Ziel ist es, sich die zusätzlichen Probeläufe der Anwendung zu sparen. Nach der Ausführung einer Anwendung werden nur noch die neu geladenen Anwendungs- und Bibliotheksklassen, die nicht bereits im Default-/Base-Layer-CDS enthalten sind, archiviert. Aktiviert wird das dynamische Archivieren mit Kommandozeilenbefehlen. In einer zukünftigen Erweiterung könnte es vollständig automatisch und transparent ablaufen. Weitere Details finden sich auf der offiziellen Seite des JEP 350 *[2]*.

ZGC kann ungenutzten Speicher wieder freigeben

Den Z Garbage Collector (ZGC) gibt es seit Java 11. Er wurde von Oracle entwickelt und verspricht sehr kurze Pausen beim Aufräumen von Heap-Speichern mit mehreren Terabytes. Bisher gab er den für die Anwendung reservierten Heap-Speicher jedoch nicht wieder frei. Das führte dazu, dass Anwendungen über ihre Lebensdauer hinweg für gewöhnlich weit mehr Speicher verbrauchen als notwendig. Besonders betroffen sind Anwendungen, die in Ressourcen-armen Umgebungen ausgeführt werden. Andere Garbage-Collectoren wie der G1 und Shanandoah unterstützen bereits das Freigeben von ungenutztem Speicher.

ZGC unterteilt den Heap in Z-Pages. Leere Z-Pages werden nach dem Garbage-Collector-Lauf in einem Cache abgelegt, um sie später für die Neuzuweisung wiederzuverwenden. Das Zuweisen und Freigeben neuer Z-Pages ist nämlich sehr kostenintensiv. Der Cache repräsentiert also eine Liste der bereits beanspruchten, aber derzeit ungenutzten Speicherabschnitte. Mit Java 13 soll eine Z-Page im Cache nun nach einem simplen Timeout wieder freigegeben werden. Der Timeout-Value kann per Kommandozeile überschrieben werden. In zukünftigen Versionen könnten zudem noch weitere verfeinerte Entscheidungsmechanismen eingeführt werden. Für mehr Details siehe JEP 351 *[3]*.

Socket APIs wurden überarbeitet

Die java.net.Socket und java.net.ServerSocket APIs und deren zugrunde liegenden Implementierungen sind noch Überbleibsel aus dem JDK 1.0. Zu großen Teilen bestehen sie aus Legacy-Java- und C-Code. Das erschwert die Wart- und Erweiterbarkeit deutlich. Die NioSocketImpl soll die veraltete PlainSocketImpl nun ablösen. NioSocketImpl ist an die bereits vorhandene New-I/O-Implementierung angelehnt und benutzt deren vorhandene Infrastruktur im JDK. Die bisherige Implementierung ist außerdem nicht kompatibel zu den geplanten Erweiterungen der Sprache im Rahmen des Projekts Loom. Die leichtgewichtigen User-Threads (Fiber) werden durch Concurrency-Probleme des Socket API behindert. Für mehr Details siehe JEP 353 *[4]*.

Switch Expressions wurden leicht angepasst

Die bisher beschriebenen Änderungen betreffen die JVM und die Klassenbibliothek. Es gibt allerdings auch einige Syntax-Erweiterungen. So wurden die in Java 12 als Preview eingeführten Switch Expressions nochmals angepasst.

Die Switch Expression ist eine Alternative zu dem ausschweifenden und für Fehler anfälligen Switch Statement. Eine ausführliche Übersicht über die Verwendung findet sich in diversen Artikeln zu Java 12 (zum Beispiel in Java aktuell 04/2019). Die größte Änderung in Java 13 ist das Ersetzen des Schlüsselwortes „break" in der Switch Expression durch „yield". Der Hintergrund ist die bessere Unterscheidbarkeit zwischen Statement (mit möglichem „break") und den Expressions (mit „yield"). Die „yield"-Anweisung verhält sich dabei wie ein „return", verlässt den Switch und liefert das Ergebnis des aktuellen Zweiges zurück.

Die zweite neue Variante mit der Arrow-Syntax funktioniert übrigens weiterhin wie in Java 12 eingeführt. Die Switch Expressions bleiben jedoch vorerst noch ein Preview-Feature, somit könnte es in den nächsten Java-Versionen weitere Anpassungen geben. In *Listing 1* sind zwei Code-Beispiele aufgeführt. Auf der einen Seite sehen wir ein Statement mit „breaks", das aber bereits mehrere Labels pro Zweig als redundanzfreie Fall-Through-Variante verwendet. Im direkten Vergleich folgt eine Switch Expression mit dem neuen Schlüsselwort „yield".

Beim Kompilieren und zum Starten unter JDK 13 müssen die jeweiligen Preview-Flags angegeben werden *(siehe Listing 2)*. Die Build-Tools (Maven, Gradle etc.) bringen natürlich auch entsprechende Konfigurationsschalter mit. Für mehr Details zu den Änderungen an den Switch Expressions, siehe JEP 354 *[5]*.

Raw String Literals kommen als Text Blocks

Ursprünglich schon für Java 12 geplant, wurde der JEP 326 (Raw String Literals) dann doch kurzfristig zurückgezogen. Die Details kann man in der Mailingliste *[6]* nachlesen. Letztlich hatte man sich aufgrund des Feedbacks und der noch nicht bis ins letzte Detail durchdachten Umsetzung zu diesem Schritt entschlossen. Mit Java 13 hat es nun aber der JEP 355 (Text Blocks) als Preview Feature in das Release geschafft. Hier konzentriert man sich zunächst auf einen Teil der ursprünglichen Raw String Literals, nämlich die mehrzeiligen Texte. In vielen Java-Anwendungen wird Code aus anderen Sprachen wie HTML oder SQL verarbeitet. Bisher waren Strings mit mehreren Zeilen weder gut zu lesen noch einfach aufzuschreiben. Für Zeilenumbrüche müssen beispielsweise extra Steuerbefehle (Escapes mit \n) eingesetzt werden. Andere Sprache wie Groovy oder Kotlin bieten längst die Möglichkeit, mehrzeilige Texte zu definieren.

Um die jetzigen Entscheidungen für Text Blocks nachvollziehen zu können, lohnt sich ein Blick auf die ursprüngliche Implementierungsidee der Raw String Literals (JEP 326). Diese speziellen Zeichenketten konnten sich auch über mehrere Zeilen erstrecken und durften zudem keine Escape-Sequenzen interpretieren. Als Begrenzungszeichen konnte man eine beliebige Anzahl von Backticks (`) verwenden. Dadurch hätte man sogar Backticks im Inhalt des Strings verwenden können, solange die Begrenzer immer mindestens ein Zeichen mehr enthalten. Wenn also im Text einzelne

```
// Ohne Text Blocks
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello, Escapes</p>\n" +
              "    </body>\n" +
              "</html>\n";

// Mit Text Blocks
String html = """
              <html>
                  <body>
                      <p>Hello, Text Blocks</p>
                  </body>
              </html>""";
```

Listing 3

Backticks enthalten sind, hätte man als Begrenzer mindestens doppelte Backticks verwenden müssen. Dieser Ansatz wurde jedoch kritisiert. Es könnte bei vielen Entwicklern zu Verwirrung führen, so die Befürchtung.

Bei den Text Blocks hat man sich darum für dreifache Anführungszeichen als einzige Art von Begrenzungszeichen entschlossen. Die öffnenden müssen im Gegensatz zu den schließenden Anführungszeichen in einer eigenen Zeile stehen. Nichtsdestotrotz beginnt der eigentliche Inhalt erst mit der zweiten Zeile, also ohne die scheinbare Leerzeile mit Zeilenumbruch. Das erhöht die Lesbarkeit des Quellcodes, da so die Einrückung der ersten Zeile korrekt im Quelltext dargestellt wird.

Textblöcke können überall da benutzt werden, wo auch reguläre Strings erlaubt sind. Ein einfaches Beispiel zeigt die Unterschiede zwischen traditioneller und neuer Syntax *(siehe Listing 3)*.

Die Textblöcke werden übrigens bereits zur Compile-Zeit ersetzt. Zunächst erfolgt die Umwandlung der Umbrüche in Betriebssystem-unabhängige Line-Feeds. Dann werden unnötige Whitespaces, die aufgrund der Code-Formatierung entstanden sind, entfernt. Zuletzt werden noch vorhandene Escape-Sequenzen aufgelöst. Nach dem Kompilieren kann man darum nicht mehr herausfinden, wie der String ursprünglich definiert war. Für mehr Details siehe JEP 355 *[7]*.

Ausblick

Die in diesem Artikel angesprochenen Änderungen waren durch JEPs (Java Enhancement Proposals) beschrieben. Weitere Anpassungen direkt an der Klassenbibliothek kann man sich entweder über den JDK API Diff Report Generator *[8]* oder den Java Almanac *[9]* auflisten lassen. Dort entdeckt man zum Beispiel, dass das alte Doclet API unter com.sun.javadoc entfernt wurde *[10]*. Sonst gab es im Gegensatz zu den letzten Releases keine weiteren nennenswerten Änderungen.

Während Java 13 gerade final erschienen ist und zum Ausprobieren heruntergeladen werden kann *[11]*, laufen auch schon die Arbeiten an der nächsten Version *[12]*. Zum Zeitpunkt der Entstehung dieses Artikels war nur der JEP 352 (Non-Volatile Mapped Byte Buffers) eingeplant. Gegebenenfalls werden die Previews von Switch Expressions und Text Blocks finalisiert. Außerdem könnte im Rahmen des JEP 343 ein neues Tool zum Verpacken von in sich abgeschlossenen Java-Anwendungen in Erscheinung treten. So ungewiss der Blick in die Glaskugel im Moment noch scheinen mag, so schnell wird das nächste Release letztlich da sein. Denn schon im März 2020 grüßt uns Java 14. Wir dürfen also gespannt sein und freuen uns auf die nächsten Neuerungen.

Referenzen:

[1] JDK 13 Projektseite: *https://openjdk.java.net/projects/jdk/13/*
[2] JEP 350: *https://openjdk.java.net/jeps/350*
[3] JEP 351: *https://openjdk.java.net/jeps/351*
[4] JEP 353: *https://openjdk.java.net/jeps/353*
[5] JEP 354: *https://openjdk.java.net/jeps/354*
[6] *https://mail.openjdk.java.net/pipermail/jdk-dev/2018-December/002402.html*
[7] JEP 355: *https://openjdk.java.net/jeps/355*
[8] API-Änderungen: *https://github.com/AdoptOpenJDK/jdk-api-diff*
[9] Java Almanac: *http://download.eclipselab.org/jdkdiff/V12/V13/index.html*
[10] *https://bugs.openjdk.java.net/browse/JDK-8215584*
[11] JDK 13 Download: *http://jdk.java.net/13/*
[12] JDK 14 Projektseite: *https://openjdk.java.net/projects/jdk/14/*

Falk Sippach
Orientation in Objects GmbH
falk.sippach@oio.de

Falk Sippach hat zwanzig Jahre Erfahrung mit Java und ist bei der Mannheimer Firma OIO Orientation in Objects GmbH als Trainer, Softwareentwickler und -architekt tätig. Er publiziert regelmäßig in Blogs, Fachartikeln und auf Konferenzen. In seiner Wahlheimat Darmstadt organisiert er mit anderen die örtliche Java User Group. Falk twittert unter @sippsack.