War denn schon wieder Java-Release-Zeit?

Falk Sippach, OIO - die Java-Experten der Trivadis

An die halbjährlichen Updates von Java haben wir uns mittlerweile gewöhnt – sie tun Sprache und Plattform gut. Zuletzt waren die Änderungen jedoch überschaubar. Mit dem Mitte März 2020 erschienenen Java 14 gab es nun wieder ein regelrechtes Feuerwerk an neuen spannenden Features. In diesem Artikel wollen wir die aus Entwicklersicht besonders relevanten Themen unter die Lupe nehmen. Schließlich steht im September mit Java 15 bereits die nächste Version vor der Tür.

Die folgenden Java Enhancement Proposals (JEP) wurden umgesetzt *[1]*:

- JEP 305: Pattern Matching for instanceof (Preview)
- JEP 343: Packaging Tool (Incubator)
- JEP 345: NUMA-Aware Memory Allocation for G1
- JEP 349: JFR Event Streaming
- JEP 352: Non-Volatile Mapped Byte Buffers
- JEP 358: Helpful NullPointerExceptions
- JEP 359: Records (Preview)
- JEP 361: Switch Expressions (Standard)
- JEP 362: Deprecate the Solaris and SPARC Ports
- JEP 363: Remove the Concurrent Mark Sweep (CMS) Garbage Collector
- JEP 364: ZGC on macOS
- JEP 365: ZGC on Windows
- JEP 366: Deprecate the ParallelScavenge + SerialOld GC Combination
- JEP 367: Remove the Pack200 Tools and API
- JEP 368: Text Blocks (Second Preview)
- JEP 370: Foreign-Memory Access API (Incubator)

JEP 359: Records

Die wahrscheinlich spannendste und gleichzeitig auch überraschendste Neuerung dürfte die Einführung der Record-Typen sein. Sie wurden noch relativ spät in das Release von Java 14 aufgenommen. Dabei handelt es sich um eine eingeschränkte Form der Klas-

sendeklaration, ähnlich zu den Enums. Entwickelt wurden Records im Rahmen des Projekts Valhalla. Es gibt gewisse Ähnlichkeiten zu Data Classes in Kotlin und Case Classes in Scala. Die kompakte Syntax könnte Bibliotheken wie Lombok in Zukunft zum Teil überflüssig machen. Kevlin Henney sieht außerdem noch folgenden Vorteil *[2]*: „I think one of the interesting side effects of the Java record feature is that, in practice, it will help expose how much Java code really is getter/setter-oriented rather than object-oriented."

Die einfache Definition einer Person mit zwei Feldern wird in *Listing 1* gezeigt. Eine erweiterte Variante mit einem zusätzlichen Konstruktor, sodass nur das Feld `name` Pflicht ist, lässt sich ebenfalls realisieren *(siehe Listing 2)*. Vom Compiler wird eine unveränderbare („immutable") Klasse erzeugt, die neben den beiden Attributen und den eigenen Methoden natürlich auch noch die Implementierungen für die Accessors (allerdings keine Getter!), den Konstruktor sowie `equals/hashcode` und `toString` enthält *(siehe Listing 3)*. Die Verwendung sieht erwartungsgemäß aus. Man sieht dem Aufrufer dabei nicht an, dass Record-Typen instanziiert werden *(siehe Listing 4)*.

Records sind übrigens keine klassischen Java Beans, da sie keine echten Getter enthalten. Man kann aber über die gleichnamigen Methoden auf die Membervariablen zugreifen. Records kann man auch mit Annotationen und JavaDocs erweitern. Im Body dürfen zudem statische Felder sowie Methoden, Konstruktoren oder Instanzmethoden deklariert werden. Die Definition von weiteren Instanzfeldern außerhalb des Record Header ist nicht erlaubt.

JEP 305: Pattern Matching for `instanceof`

Das Konzept des Pattern Matching kommt bereits seit den 1960er Jahren bei diversen Programmiersprachen zum Einsatz. Zu den modernen Vertretern zählen Haskell und Scala. Ein Pattern ist eine Kombination aus einem Prädikat, das auf eine Zielstruktur passt, und einer Menge von Variablen innerhalb dieses Musters. Diesen Variablen werden bei passenden Treffern die entsprechenden Inhalte zugewiesen. Die Intention ist demnach die Destrukturierung von Objekten, also das Aufspalten in seine Bestandteile.

In Java kann man für solche Anwendungsfälle das Switch-Statement benutzen, ist dabei aber auf die Datentypen Integer, String und Enum beschränkt. Es wird noch einige Zeit dauern, bis es echtes Pattern Matching in Java gibt. Durch die Einführung der Switch Expression in Java 12 wurde jedoch bereits der erste Schritt dahin vollzogen. Mit Java 14 können wir nun zusätzlich Pattern Matching beim `instanceof`-Operator nutzen. Dabei werden unnötige Casts vermieden, zudem erhöht sich durch die verringerte Redundanz die Lesbarkeit.

Vorher musste man beispielsweise für das Prüfen auf einen leeren String beziehungsweise eine leere Collection wie in *Listing 5* gezeigt vorgehen. Jetzt kann man beim Check mit `instanceof` den Wert direkt einer Variablen zuweisen und darauf weitere Operationen ausführen *(siehe Listing 6)*. Der Unterschied mag auf den ersten Blick marginal erscheinen. Für die Puristen unter den Java-Entwicklern spart das allerdings eine kleine, aber dennoch lästige Redundanz ein.

Übrigens, die Switch Expression wurde zunächst in Java 12 und 13 jeweils als Preview-Feature eingeführt. Sie wurde nun im JEP 361

```
public record Person(String name, Person partner) {}
```

Listing 1

```
public record Person(String name, Person partner) {
  public Person(String name) {
    this(name, null);
  }
  public String getNameInUppercase() {
    return name.toUpperCase();
  }
}
```

Listing 2

```
public final class Person extends Record {
  private final String name;
  private final Person partner;

  public Person(String name) {
    this(name, null);
  }
  public Person(String name, Person partner){
    this.name = name; this.partner = partner;
  }

  public String getNameInUppercase(){
    return name.toUpperCase();
  }
  public String toString(){ /* ... */ }
  public final int hashCode(){ /* ... */ }
  public final boolean equals(Object o){ /* ... */ }
  public String name(){ return name; }
  public Person partner(){ return partner; }
}
```

Listing 3

```
var man = new Person("Adam");
var woman = new Person("Eve", man);
woman.toString(); // ==> "Person[name=Eve, partner=…]"

woman.partner().name(); // ==> "Adam"
woman.getNameInUppercase(); // ==> "EVE"

// Deep equals
// ==> true
new Person("Eve", new Person("Adam")).equals(woman);
```

Listing 4

```
boolean isNullOrEmpty(Object o){
  return == null ||
    instanceof String && ((String) o).isBlank() ||
    instanceof Collection && ((Collection) o).isEmpty();
}
```

Listing 5

```
boolean isNullOrEmpty(Object o){
  return o == null ||
    o instanceof String s && s.isBlank() ||
    o instanceof Collection c && c.isEmpty();
}
```

Listing 6

```
String developerRating(int numberOfChildren){
  return switch(numberOfChildren){
    case 0 -> "open source contributor";
    case 1, 2 -> "junior";
    case 3 -> "senior";
    default -> {
      if(numberOfChildren < 0){
        throw new IndexOutOfBoundsException(numberOfChildren);
      }
      yield "manager";
    }
  };
}
```

Listing 7

```
man.partner().name()

Result: java.lang.NullPointerException: Cannot invoke "Person.name()" because the return value of "Person.partner()" is null
```

Listing 8

finalisiert. Dadurch stehen Entwicklern zwei neue Syntaxvarianten mit einer kürzeren und klareren sowie weniger fehleranfälligen Semantik zur Verfügung. Das Ergebnis der Expression kann einer Variablen zugewiesen oder als Wert aus einer Methode zurückgegeben werden *(siehe Listing 7)*. Weitere Details können dem Artikel zu Java 13 *[3]* entnommen werden.

JEP 358: Helpful NullPointerExceptions

Der unbeabsichtigte Zugriff auf leere Referenzen ist auch bei Java-Entwicklern gefürchtet. Nach eigener Aussage von Sir Tony Hoare war seine Erfindung der Null-Referenz ein Fehler mit Folgen in Höhe von vielen Milliarden Dollar. Und das nur, weil es bei der Entwicklung der Sprache Algol in den 1960er Jahren einfach so leicht zu implementieren war.

In Java gibt es weder vom Compiler noch von der Laufzeitumgebung Unterstützung beim Umgang mit Null-Referenzen. Mit diversen Mustern und Vorgehensweisen lassen sich diese leidigen Exceptions allerdings vermeiden. Den einfachsten Weg stellt die Prüfung auf `null` dar. Leider ist dieses Vorgehen sehr mühselig und wird immer genau dann vergessen, wenn man den Check gebraucht hätte. Mit der seit dem JDK 8 enthaltenen Wrapper-Klasse „optional" kann man über das API den Aufrufer darauf hinweisen, dass ein Wert `null` sein kann und er darauf reagieren muss. Somit kann man nicht mehr aus Versehen in eine Null-Referenz hineinlaufen, sondern muss explizit mit dem möglicherweise leeren Wert umgehen. Dieses Vorgehen bietet sich unter anderem bei Rückgabewerten von öffentlichen Schnittstellen an, kostet jedoch auch eine extra Indirektionsschicht, da man den eigentlichen Wert immer auspacken muss.

In anderen Sprachen wurden längst Hilfsmittel in die Syntax und den Compiler eingebaut, wie zum Beispiel in Groovy das NullObjectPattern und der Safe Navigation Operator (*some?.method()*). Bei Kotlin kann man explizit zwischen Typen, die nicht leer sein dürfen, und solchen, bei deren Referenz auch `null` erlaubt ist, unterscheiden. Lange Rede, kurzer Sinn: Mit den NullPointerExceptions werden wir auch zukünftig in Java leben müssen. Aber immerhin erleichtern uns die als Preview-Feature eingeführten „Helpful NullPointerExceptions" nun die Fehlersuche im Ausnahmefall. Damit beim Werfen einer NullPointerException die notwendigen Informationen eingefügt werden, muss man beim Starten die folgende Option aktivieren: `-XX:+ShowCodeDetailsInExceptionMessages`. Wenn dann in einer Aufrufkette ein Wert `null` ist, bekommt man die in *Listing 8* gezeigte aussagekräftige Meldung.

Lambda-Ausdrücke benötigen eine Spezialbehandlung. Ist zum Beispiel der Parameter einer Lambda-Funktion `null`, bekommt man standardmäßig eine unzureichende Fehlermeldung *(siehe Listing 9)*. Damit der korrekte Parametername angezeigt wird, muss der Quellcode mit der Option `-g:vars` kompiliert werden. Das Resultat ist in *Listing 10* zu sehen. Bei Methodenreferenzen gibt es aktuell leider noch keinen Hinweis im Fall eines leeren Parameters *(siehe Listing 11)*.

Wenn man aber wie hier im Beispiel jeden Stream-Methodenaufruf auf eine neue Zeile setzt, lässt sich die problematische Codezeile schnell eingrenzen. Herausfordernd waren NullPointerExceptions bisher beim automatischen Boxing/Unboxing. Wird hier nun auch der Compiler-Parameter `-g:vars` aktiviert, bekommt man ebenfalls eine neue hilfreiche Fehlermeldung *(siehe Listing 12)*.

```
Stream.of(man, woman)
  .map(p -> p.partner())
  .map(p -> p.name())
  .collect(Collectors.toUnmodifiableList());

Result: java.lang.NullPointerException: Cannot invoke "Person.name()" because "<parameter1>" is null
```

Listing 9

```
java.lang.NullPointerException: Cannot invoke "Person.name()" because "p" is null
```

Listing 10

```
Stream.of(man, woman)
  .map(Person::partner)
  .map(Person::name)
  .collect(Collectors.toUnmodifiableList())
Result: java.lang.NullPointerException
```

Listing 11

JEP 368: Text Blocks

Ursprünglich als Raw String Literals bereits für Java 12 geplant, wurde in Java 13 zunächst eine abgespeckte Variante in Form von mehrzeiligen Strings namens Text Blocks eingeführt. Insbesondere für HTML-Templates und SQL-Skripte erhöht sich dadurch die Lesbarkeit enorm *(siehe Listing 13)*.

Neu hinzugekommen sind jetzt zwei Escape-Sequenzen, mit denen man die Formatierung eines Text Block anpassen kann. Um zum Beispiel einen Zeilenumbruch zu verwenden, der aber nicht explizit in der Ausgabe erscheinen soll, kann man am Zeilenende einfach ein „\" (Backslash) einfügen. Im in *Listing 14* Gezeigten bekommt man trotz Umbrüchen einen String mit einer langen Zeile.

Die zweite neue Escape-Sequenz „\s" wird zu einem Leerzeichen übersetzt. Dadurch kann man erreichen, dass Leerzeichen am Zeilenende nicht automatisch abgeschnitten (getrimmt) werden und man beispielsweise eine feste Zeichenbreite je Zeile erhält *(siehe Listing 15)*.

Was gibt es noch Neues?

Neben den gerade beschriebenen Features, die hauptsächlich für Entwickler interessant sind, gibt es natürlich diverse andere Änderungen. Im JEP 352 wurde das FileChannel-API erweitert, um die Erzeugung von MappedByteBuffer-Instanzen zu ermöglichen. Die arbeiten auf nichtflüchtigen Datenspeichern (NVM: non-volatile memory), im Gegensatz zum volatilen Speicher, dem RAM. Die Zielplattform ist allerdings auf Linux x64 beschränkt. Auch bei den Garbage Collectors hat sich wieder etwas getan. So wurde der Concurrent Mark Sweep (CMS) Garbage Collector entfernt und den ZGC gibt es jetzt auch für macOS und Windows.

Bei kritischen Java-Anwendung wird empfohlen, die Flight-Recording-Funktion in Produktion zu aktivieren. Der in *Listing 16* gezeigte Befehl startet eine Java-Anwendung mit Flight Recording und schreibt die Informationen in die recording.jfr, wobei immer die Daten eines Tages aufgehoben werden.

Normalerweise liest man die Daten dann mit dem Tool JDK Mission Control (JMC) aus, um sie zu analysieren. Neu im JDK 14 ist, dass man auch aus der Anwendung heraus asynchron die Events abfragen kann *(siehe Listing 17)*.

```
int calculate(){
  Integer a = 2, b = 4, x = null;
  return a + b * x;
}
calculate();
Result: java.lang.NullPointerException: Cannot invoke "java.lang.Integer.intValue()" because "x" is null
```

Listing 12

```
// 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 13

```
String text = """
  Lorem ipsum dolor sit amet, consectetur adipiscing \
  elit, sed do eiusmod tempor incididunt ut labore \
  et dolore magna aliqua.\
""";
// statt
String literal = "Lorem ipsum dolor sit amet, consectetur adipiscing " +
  "elit, sed do eiusmod tempor incididunt ut labore " +
  "et dolore magna aliqua.";
```

Listing 14

```
String colors = """
    red  \s
    green\s
    blue \s
    """;
```

Listing 15

```
java \
-XX:+FlightRecorder \
-XX:StartFlightRecording=disk=true, \
filename=recording.jfr,dumponexit=true,maxage=1d \
-jar application.jar
```

Listing 16

```
import jdk.jfr.consumer.RecordingStream;
import java.time.Duration;

try(var rs = new RecordingStream()){
  rs.enable("jdk.CPULoad").withPeriod(Duration.ofSeconds(1));
  rs.onEvent("jdk.CPULoad", event -> {
    System.out.printf("%.1f %% %n",
      event.getFloat("machineTotal") * 100);
  });
  rs.start();
}
```

Listing 17

Im JDK 8 gab es das Tool „javapackager", das leider mitsamt JavaFX in den neueren Java-Versionen entfernt wurde. In Java 14 wird nun der Nachfolger „jpackage" eingeführt (JEP 343: Packaging Tool), mit dem wieder eigenständige Java-Installationsdateien erstellt werden können. Der Inhalt ist die Java-Anwendung mitsamt einer Laufzeit-Umgebung. Das Tool baut aus diesem Input ein lauffähiges Binärartefakt, das sämtliche Abhängigkeiten enthält (Formate: msi, exe, pkg als dmg, app als dmg, deb und rpm).

Ausblick

Vor anderthalb Jahren ist im Herbst 2019 mit Java 11 die letzte LTS-Version erschienen. Seitdem gab es bei den beiden folgenden Major-Releases jeweils nur eine überschaubare Menge an neuen Features. In den JDK-Inkubator-Projekten (Amber, Valhalla, Loom etc.) wird jedoch bereits an vielen neuen Ideen gearbeitet und so verwundert es nicht, dass der Funktionsumfang beim gerade veröffentlichten JDK 14 wieder deutlich größer ausfällt. Und auch, wenn nur wenige die neue Version in Produktion einsetzen werden, sollte man trotzdem frühzeitig einen Blick auf die Neuerungen werfen und gegebenenfalls zu den Preview-Funktionen Feedback geben. Nur so ist sichergestellt, dass sie bis zur Finalisierung im nächsten LTS-Release, das als Java 17 im Herbst 2021 erscheinen wird, gerüstet sind.

„Languages must evolve, or they risk becoming irrelevant", sagte Brian Goetz von Oracle im November 2019 in seiner Präsentation „Java Language Futures" bei der Devoxx in Belgien. Er ist als Language Architect maßgeblich daran beteiligt, dass Java trotz seiner 25 Jahre noch lange nicht zum alten Eisen gehört. Oracle hat dazu in den vergangenen Jahren einige wegweisende Entscheidungen getroffen. Die halbjährlichen OpenJDK-Releases mit den Preview-Features wurden gut angenommen. Zudem hat Oracle sein Lizenzmodell geändert. Das Oracle JDK wird jetzt zwar nicht mehr kostenfrei angeboten, das hat den Wettbewerb allerdings angekurbelt. Und so bekommt man nun von diversen Anbietern, auch noch von Oracle, freie Distributionen des OpenJDK. Das ist seit Java 11 binärkompatibel zum Oracle JDK und steht unter einer Open-Source-Lizenz.

Wir können also festhalten: Java ist noch lange nicht tot. Außerdem sind aktuell noch viele Features in Arbeit, die in zukünftigen Versionen auf ihren Einsatz warten. Uns Java-Entwicklern wird also nicht langweilig werden, die Zukunft sieht weiterhin rosig aus. Und im September 2020 erwartet uns bereits Java 15!

Referenzen:

[1] JDK 14 Projektseite: *https://openjdk.java.net/projects/jdk/14/*

[2] Zitat Kevlin Henney: *https://twitter.com/KevlinHenney/status/1226094836404670464?s=03*

[3] Rückblick Java 13: *https://jaxenter.de/java-13-neue-features-87053*

[4] Codebeispiele: *https://github.com/jonatan-kazmierczak/java-new-features/blob/master/java14.md*

Falk Sippach

JUG Darmstadt

falk@jug-da.de

Falk Sippach hat 20 Jahre Erfahrung mit Java und ist bei der OIO - den Java-Experten der Trivadis - als Trainer, Software-Entwickler 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.