Java 12 ist da

Falk Sippach, Orientation in Objects

Mitte März erfolgte wie geplant die Veröffentlichung des JDK 12 mit den folgenden Java Enhancement Proposals (JEP) [1]:

- *JEP 189: Shenandoah: A Low-Pause-Time Garbage Collector (Experimental)*
- *JEP 230: Microbenchmark Suite*
- *JEP 325: Switch Expressions (Preview)*
- *JEP 334: JVM Constants API*
- *JEP 340: One AArch64 Port, Not Two*
- *JEP 341: Default CDS Archives*
- *JEP 344: Abortable Mixed Collections for G1*
- *JEP 346: Promptly Return Unused Committed Memory from G1*

Auf den ersten Blick sieht es nach wenig aus. Neben internen Änderungen und Performanceverbesserungen sticht aus Entwicklersicht hauptsächlich ein Punkt ins Auge: Switch Expressions. Im Rahmen dieses Artikels wollen wir das neue Feature näher beleuchten und auch einen Blick auf die weiteren Neuerungen werfen.

Probleme mit dem Switch Statement

Das Konstrukt „Switch" ist eigentlich eine Idee aus der prozeduralen Programmierung und ein Überbleibsel aus der Tatsache, dass die Syntax von Java initial stark an C angelehnt wurde. Für den erfahrenen Programmierer ist das Switch Statement jedoch ein Code Smell und kann im Sinne der Objektorientierung besser mit dem Strategie-Entwurfsmuster ausgedrückt werden. Nichtsdestotrotz gibt es Situationen, in denen man mit einem Switch sehr effizient

und übersichtlich Fallunterscheidungen umsetzen kann. Allerdings hatte die bisherige Implementierung einige Schwächen. Werfen wir darum zunächst einen Blick auf das klassische Switch Statement *(siehe Listing 1)*.

Diese Variante wirkt etwas unhandlich und wird bei komplexeren `case`-Blöcken schnell unübersichtlich. Zudem können kleine Flüchtigkeitsfehler oder Unwissenheit über die genaue Funktionsweise zu schlecht nachvollziehbaren Bugs führen. Ein möglicherweise unabsichtlicher Fall-Through entsteht, wenn man einen `case`-Block nicht mit einem `break` beendet. So springt die Ausführung weiter in den nächsten Block, auch wenn dessen Bedingung gar nicht zur Switch-Variablen passt. In unserem Beispiel würde bei der Ausführung mit `i = 1` als Ergebnis `two` zurückgegeben. Sinnvoll kann dieses Verhalten wiederum sein, wenn mehrere Switch-Bedingungen auf das gleiche Ergebnis abgebildet werden sollen, wobei dann die durchfallenden `case`-Zweige explizit leer bleiben *(siehe Listing 2)*.

Wenn man vergisst, einen bestimmten Fall abzudecken, wird der Switch scheinbar nicht ausgeführt. Daher empfiehlt es sich immer, einen `Default`-Zweig zu definieren. Dieser kann dann wenigstens zur Laufzeit einen Fehler für nicht definierte Input-Parameter werfen. Denn leider warnt uns der Compiler nicht, wenn wir den Switch mit Werten kleiner als eins oder größer als drei aufrufen. In den unterschiedlichen `case`-Blöcken wird typischerweise redundant immer wieder die gleiche temporäre Variable mit natürlich unterschiedlichen Werten befüllt. Das widerspricht dem DRY-Prinzip ("Don't Repeat Yourself") und erschwert die Nachvollziehbarkeit, insbesondere wenn mit dem Fall-Through gearbeitet wird. Temporäre Variablen sind nicht umsonst ein Code Smell. Idealerweise kann man komplett auf sie verzichten.

Zu guter Letzt führt der globale Scope des Switch Statement dazu, dass in den einzelnen Blöcken möglicherweise unterschiedliche Variablen definiert werden müssen, um ungewollte Nebeneffekte bei der Verwendung der gleichen Namen zu vermeiden.

Alles neu macht die Switch Expression

Ab Java 12 können die Case-Zweige jetzt mehrere Labels haben. Das ermöglicht eine redundanzfreie und kompakte Schreibweise, um einen Fall-Through zu verwenden *(siehe Listing 3)*.

Mit der Pfeil-Syntax existiert eine Variante, die der Definition von Lambda-Ausdrücken ähnelt. Jeder Zweig wird auf eine Zeile reduziert und das `break` entfällt. Das spart wieder Boiler Plate Code und macht Switch Statements kompakter und übersichtlicher. Ein unabsichtlicher Fall-Through ist so nicht mehr möglich. Es wird immer nur genau das eine Statement ausgeführt. Sobald mehr als eine Anweisung auf der rechten Seite benötigt wird, muss der Code innerhalb eines Blockes stehen. Damit wird ein eigener Scope geschaffen, der zugleich eine kollisionsfreie Variablendeklaration ermöglicht *(siehe Listing 4)*.

In der funktionalen Programmierung gibt es typischerweise keine Zustandsänderungen in Form von Variablenzuweisungen. Vielmehr wird

```
public String describeInt(int i) {
  String str = "not set";
  switch(i) {
    case 1:
      str = "one";
    case 2:
      str = "two";
      break;
    case 3:
      str = "three";
      break;
  }
  return str;
}
```

Listing 1

```
public String describeInt(int i) {
  String str = "not set";
  switch(i) {
    case 1:
    case 2:
      str = "one or two";
      break;
    case 3:
      str = "three";
      break;
  }
  return str;
}
```

Listing 2

```
public String describeInt(int i) {
  String str = "not set";
  switch (i) {
     case 1, 2:
       str = "one or two";
       break;
    case 3:
       str = "three";
       break;
  }
  return str;
}
```

Listing 3

mit Ausdrücken gearbeitet, die als Inputparameter wiederum anderen Funktionen zur weiteren Bearbeitung übergeben werden können. Die Switch Expression ist jetzt auch ein Ausdruck, der ein Ergebnis zurückliefert. So kann man das Resultat zum Beispiel direkt einer Variablen (besser Konstanten) zuweisen oder einem Methodenaufruf als Parameter übergeben beziehungsweise per `return` zurückgeben *(siehe Listing 5)*.

Übrigens muss jeder Zweig mit dem `break` jeweils genau einen Wert zurückliefern („break-with-value"-Semantik). Dadurch kann der Compiler prüfen, ob wir alle möglichen Fälle behandeln oder wenigstens einen `Default`-Block als Fallback angegeben haben. Damit ist sichergestellt, dass immer genau ein Zweig abgearbeitet und dessen Ergebnis zurückgeliefert wird. Switch Statements ohne

```
public String describeInt(int i) {
  String str = "not set";
  switch (i) {
    case 1, 2 -> str = "one or two";
    case 3 -> {
    var i = complexComputation();
    str = "three" + i;
  }
  }
  return str;
}
```

Listing 4

```
public static String describeInt(int i) {
  return switch (i) {
    case 1, 2:
      break "one or two";
    case 3:
      break "three";
    default:
      break "smaller than one or bigger than three";
  };
}
```

Listing 5

```
public static String describeInt(int i) {
  return switch (i) {
    case 1, 2 -> "one or two";
    case 3 -> "three";
    default -> "smaller than one or more than three";
  };
}
```

Listing 6

sichtbaren Output gehören somit der Vergangenheit an. Durch die schon angesprochene alternative Pfeil-Notation verkürzt sich die Syntax der Switch-Expression nochmals *(siehe Listing 6)*.

Die Switch Expression arbeitet natürlich auch nahtlos mit der in Java 10 eingeführten Local Variable Type Inference zusammen. Bei der Zuweisung zu einer mit „var" deklarierten Variable wird der spezifischste gemeinsame Typ aller `case`-Zweige (kleinster gemeinsamer Nenner) ausgewählt. Ein Nachteil von Switch Expressions ist aber, dass ein Ausdruck letztendlich immer zu einem Wert aufgelöst werden muss. Dadurch ist es nicht möglich, innerhalb der Expression `return` oder `continue` zu benutzen. Das wurde häufig verwendet, um direkt aus einer Methode beziehungsweise Iteration zu springen. Durch das Werfen einer Exception kann man allerdings weiterhin den Ausdruck vorzeitig beenden.

Als Datentypen sind bisher `byte`, `short`, `char`, `int`, deren Wrapper-Klassen sowie Enums und Strings in Switch Statements erlaubt. Es gibt bereits Pläne, zusätzlich `float`, `double` und `long` zu unterstützen. Mit dem JEP 305 soll in zukünftigen Versionen zudem das Pattern Matching – eine Art Switch on Steroids – Einzug in das JDK halten. Die Intention des Pattern Matching ist eigentlich die Destrukturierung von Werten, also viel mehr als das reine Unterscheiden beliebiger Datentypen. Nichtsdestotrotz kann man die Switch Expressions in der heutigen Form bereits als eine Art Vorgeschmack auf das Pattern Matching sehen. Nicht vergessen sollten wir allerdings, dass es sich aktuell im JDK 12 nur um eine Preview handelt. Bis zum nächsten LTS-Release (JDK 17) kann sich durch Feedback aus der Community an der Umsetzung nochmal einiges ändern.

Weitere Neuerungen

Werfen wir noch einen Blick auf einige der anderen Punkte aus den Release Notes *[2]*. So wird ab dem JDK 12 im Sourcecode eine Suite von etwa 100 Microbenchmarks ausgeliefert, die auf dem bekannten Java Microbenchmark Harness (JMH) basieren. Die Suite soll die Ausführung von existierenden und die Erstellung von neuen Microbenchmarks erleichtern.

Das JVM Constants API ermöglicht das typsichere und damit weniger fehleranfällige Laden von Werten aus dem Java Constants Pool (`int`, `float`, `String`, `Class`). Dieses API ist vor allem für Werkzeuge hilfreich, die Klassen und Methoden manipulieren. Bei den Garbage Collectors gibt es ebenfalls wieder einige Neuerungen. So hat der Default-GC G1 diverse Performance-Verbesserungen erfahren. Mit Shenandoah wurde zudem ein neuer Garbage Collector eingeführt. Er wurde ursprünglich von Red Hat entwickelt und ist eine Alternative zum bereits existierenden ZGC *[3]*, verwendet jedoch einen anderen Algorithmus. Dabei sind die STW-Pausen (Stop-the-World) sehr kurz und zudem können große Mengen an Hauptspeicher (mehrere Terabyte Heap) effizient verwaltet werden. Dazu werden die Aufräumarbeiten konkurrierend zur eigentlichen Anwendung erledigt, was diese dann allerdings etwas langsamer macht. Dieses Feature ist noch experimentell und darum nicht im Standard (Oracle) OpenJDK enthalten.

Um beim Starten der JVM nicht immer alle Metadaten der Klassen erneut laden zu müssen, können diese Informationen in sogenannten Class-Data-Sharing-Archiven abgelegt werden. Das verkürzt die Startzeit, verringert den Memory Footprint und verbessert die Performance der Garbage Collection. Für 64-Bit-Builds ist ab Java 12 das Abspeichern dieser Informationen der Standard, um die Startup-Zeit von Java-Anwendungen out of the box verbessern zu können.

In der JDK-Klassenbibliothek gab es auch ein paar Änderungen *[4]*. Die Klasse String hat zum Beispiel zwei neue Methoden spendiert bekommen, um einen Text entweder einzurücken (indent) oder umzuwandeln (transform). Die Methode „mismatch" in Files findet das erste Byte, das sich zwischen zwei Dateien unterscheidet *(siehe Listing 7)*.

Die neue Klasse "CompactNumberFormat" formatiert Zahlen abhängig vom Locale basierend auf dem Unicode Common Locale Data Repository (CLDR) *[5] (siehe Listing 8)*.

Ursprünglich sollten neben den Switch Expressions auch die Raw-String-Literals in Java 12 erscheinen. Allerdings wurde die Veröffentlichung dieses Features auf unbestimmte Zeit verschoben, weil noch der nötige Feinschliff gefehlt hat: „... in reviewing the feedback we have received, I am no longer convinced that we've yet got to the

```
// "     foobar"
System.out.println("foobar".indent(5));

// java.lang.Integer
var clazz = "42".transform(Integer::valueOf).getClass();

final long mismatch = Files.mismatch(Path.of("a.out"),
Path.of("b.out"));
```

Listing 7

```
import java.text.*;
var cnf = NumberFormat.getCompactNumberInstance(
Locale.GERMAN, NumberFormat.Style.LONG );

cnf.format(1L << 10); // ==> 1 Tausend
cnf.format(1920); // ==> 2 Tausend
cnf.format(1L << 20); // ==> 1 Million
cnf.format(1L << 30); // ==> 1 Milliarde
```

Listing 8

right set of tradeoffs between complexity and expressiveness, or that we've explored enough of the design space to be confident that the current design is the best we can do." – Brian Goetz *[6]*.

Trotzdem können wir bereits jetzt einen Blick darauf werfen, was uns in einem der nächsten Java Releases erwarten wird. Die Hauptprobleme bei den klassischen Java Strings sind der nicht intuitive Einsatz von Zeilenumbrüchen („1\n2\n3") und das schlecht lesbare „Escapen" des Backslash, was insbesondere Windows-Dateipfade und reguläre Ausdrücke unnötig verkompliziert *(siehe Listing 9)*.

Raw-String-Literale interpretieren abgesehen von CRLF beziehungsweise LF keinerlei Escape-Sequenzen. Als Trennzeichen fungiert der Backtick *(siehe Listing 10)*. Übrigens kann ein Raw-String-Literal auch mit mehrfachen Backticks umschlossen sein *(Listing 11)*. Das ermöglicht die Verwendung von Backticks im Text, ohne sie „escapen" zu müssen. Weitere Informationen finden sich in einem Blog-Post *[7]*.

```
var dir = "c:\\tmp";
var regexp = "\\d{2}\\.\\d{3}");
```

Listing 9

```
var string = `1. Zeile \d{2}\.\d{3}
2. Zeile: c:\windows`
```

Listing 10

```
var textWithBackticks = ``Enthält ` im Inhalt``
```

Listing 11

Fazit

Möglicherweise ist man weiterhin auf Java 8 beschränkt oder möchte bis 2021 zunächst die aktuelle LTS (Long Term Support) Version JDK 11 verwenden. Trotzdem empfiehlt es sich, das JDK 12 bereits heute herunterzuladen *[8]* und sich mit den Neuerungen, insbesondere den Switch Expressions, vertraut zu machen. Alternativ kann man auch die neuen halbjährlichen Releases des Oracle OpenJDK mitgehen. Dann muss man zwar häufig migrieren, vermeidet jedoch die aufwendigen Big-Bang-Umstiege auf eine neue Java-LTS-Version alle drei Jahre. Zudem erhält man so für das freie Oracle OpenJDK noch für ein halbes Jahr kostenlose Updates. Denn das Oracle JDK 11 darf in Produktion nur noch kostenpflichtig eingesetzt werden. Auch für das Oracle JDK 8 gibt es seit Januar 2019 keine freien Updates mehr. Möchte man lieber über mehrere Jahre eine aktuelle, aber freie JDK-LTS-Version mit regelmäßigen Updates einsetzen, muss man auf alternative OpenJDK-Distributionen wie AdoptOpenJDK oder Amazon Corretto ausweichen. Auf jeden Fall scheint Oracles Idee mit den halbjährlichen Major-Releases und damit regelmäßigen – wenn auch wenigen – Änderungen aufzugehen. Bisher wurden diese kleinen Releases sehr pünktlich geliefert und so dürfen wir bereits auf die nächsten Neuerungen von Java 13 im Oktober 2019 gespannt sein.

Referenzen

[1] JDK-12-Projektseite: *https://openjdk.java.net/projects/jdk/12/*
[2] Release Notes: *http://jdk.java.net/12/release-notes*
[3] ZGC: *https://wiki.openjdk.java.net/display/zgc/Main*
[4] API-Änderungen: *https://github.com/AdoptOpenJDK/jdk-api-diff*
[5] Unicode Common Locale Data Repository: *https://unicode.org/reports/tr35/tr35-numbers.html#Compact_Number_Formats*
[6] Drop Raw String Literals: *https://mail.openjdk.java.net/pipermail/jdk-dev/2018-December/002402.html*
[7] Raw String Literals: *https://blog.oio.de/2019/03/04/raw-string-literals-geplant-nach-java-12/*
[8] JDK 12 Download: *http://jdk.java.net/12/*

Falk Sippach
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.