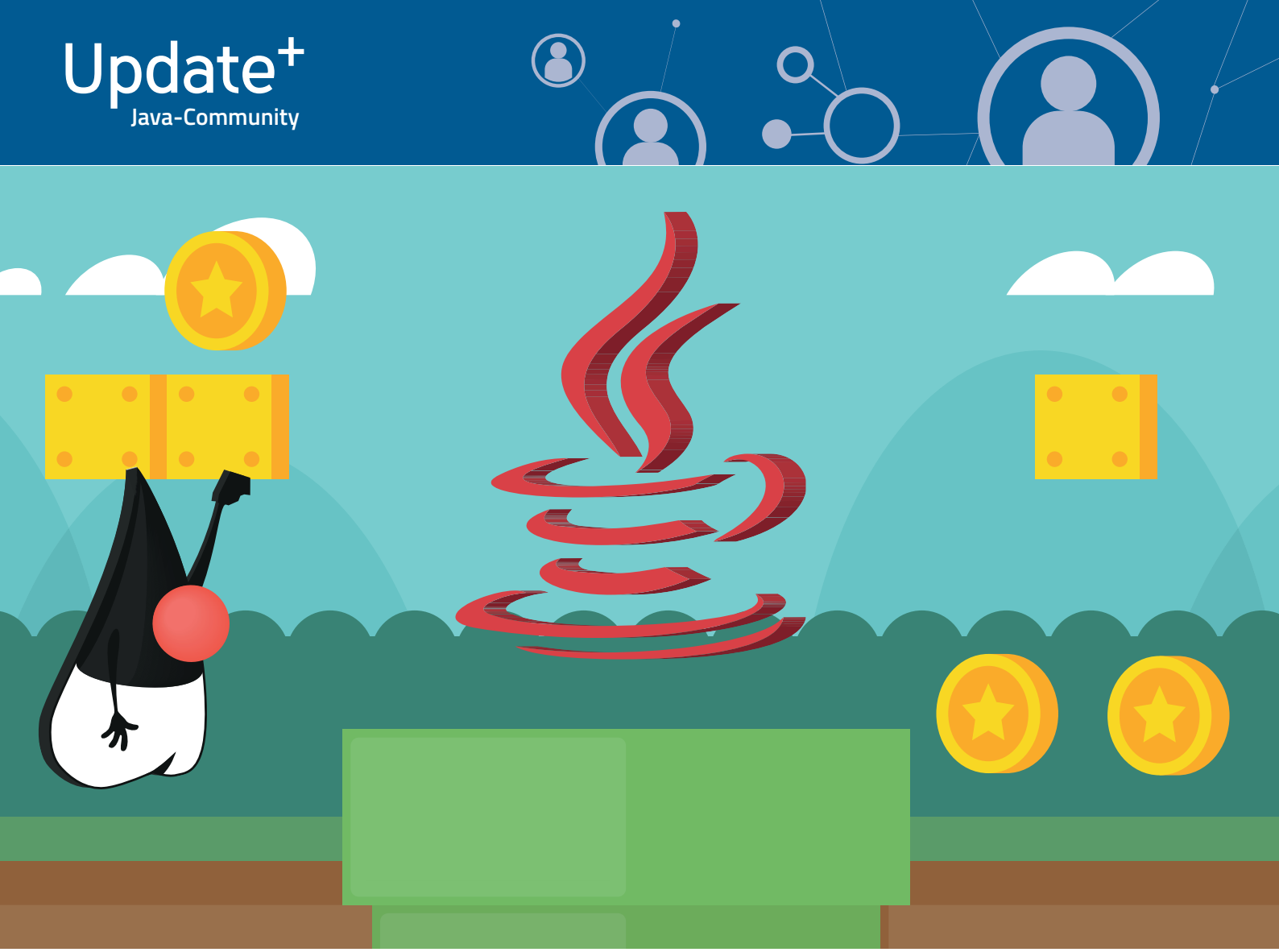

Java geht in die nächste Runde

Falk Sippach, Orientation in Objects GmbH

Nur die wenigsten haben die Neuerungen von Java 9 bereits verdaut, von daher kam das Release von Java 10 für viele überraschend. Das liegt einmal an der ungewohnt überpünktlichen Auslieferung, hauptsächlich aber an dem neuen, halbjährlichen Release-Zyklus. Natürlich darf man sich aufgrund der kurzen Zeitspanne keine großen Änderungen erwarten; doch es gibt einige interessante Features.

Erst vor einem halben Jahr hatte Java im September 2017 den letzten großen Versionssprung erlebt. Die damals eingeführten Modularisierungsmöglichkeiten rund um Jigsaw haben die meisten Entwickler bisher kaum verarbeitet und es gibt nur wenige kommerzielle Projekte, die bereits auf Java 9 migriert sind. Selbst viele Hersteller von Bibliotheken und Werkzeugen sind noch mit den Anpassungen auf das neue Modulsystem beschäftigt. So hat das JUnit-Team erst kürzlich die Version 5.1 veröffentlicht, die Tests in Jigsaw-Modulen unterstützt.

Dass viele Entwicklungsabteilungen die Migration nach Java 9 bisher gescheut haben, liegt nicht nur an der hohen Komplexität des Modulsystems; vielmehr erhalten durch das neue, halbjährliche Release-Modell nicht mehr alle Java-Versionen den gleichen langfristigen Support. Nach Java 8 wird Oracle voraussichtlich erst wieder für Java 11, das für September 2018 geplant ist, einen Long-Term-Support anbieten.

Für Entwickler haben die kürzeren Release-Zyklen den Vorteil, dass sie viel früher kleinere Änderungen ausprobieren können, auf die sie sonst wieder mehrere Jahre hätten warten müssen, weil sie im Schatten von großen Features (wie Generics, Lambdas, Streams,

```
// Type inference bei Parametern von Lambda Ausdrücken (Java 8)
Function<String, String> helloFunction = s -> "Hello " + s;

// Inference von Generics (Diamond Operator, seit Java 7)
List<String> strings = new ArrayList<>();
strings.add(helloFunction.apply("World"));

// Inference von Generics (Diamond Operator) bei anonymen inneren Klassen (Java 9 -> JEP 213)
Consumer<String> printer = new Consumer<>() {
  @Override
  public void accept(String string) {
     System.out.println(string);
  }
};
strings.forEach(printer::accept);
```

Listing 1

Jigsaw etc.) standen und sich dadurch die Auslieferung ungewollt immer wieder verzögerte.

Die neuen Funktionen

Kommen wir zurück zu Java 10 – dank des kurzen Zeitrahmens muss man sich nicht durch allzu lange Release Notes quälen. Die Liste auf der OpenJDK-Projektseite *[1]* ist überschaubar:

- 286: Local-Variable Type Inference
- 296: Consolidate the JDK Forest into a Single Repository
- 304: Garbage-Collector Interface
- 307: Parallel Full GC for G1
- 310: Application Class-Data Sharing
- 312: Thread-Local Handshakes
- 313: Remove the Native-Header Generation Tool (javah)
- 314: Additional Unicode Language-Tag Extensions
- 316: Heap Allocation on Alternative Memory Devices
- 317: Experimental Java-Based JIT Compiler
- 319: Root Certificates
- 322: Time-Based Release Versioning

Die Nummern vor den neuen Funktionen sind die Java Enhancement Proposals (JEP); aus Entwicklersicht ist insbesondere der erste am interessantesten, die Local-Variable Type Inference. Typ-Inferenz ist das Schlussfolgern auf Datentypen aus den restlichen Angaben des Codes und den Typisierungsregeln heraus. Dadurch kann man im Code gewisse Typ-Angaben weglassen. Die Lesbarkeit erhöht sich, weil der Quellcode nicht unnötig aufgebläht wird. Man kennt das in Java bereits bei Lambda-Parametern und beim Diamond-Operator für generische Daten-Container beziehungsweise seit Java 9 auch für anonyme innere Klassen *(siehe Listing 1)*.

```
// int
var zahl = 5;
// String
var string = "Hello World";
// BigDecimal
var objekt = BigDecimal.ONE;
```

Listing 2

Mit dem Schlüsselwort „var" lassen sich jetzt sehr prägnant lokale Variablen definieren, deren Datentyp sich direkt aus der Zuweisung des Werts ergibt. Während beim „Diamond"-Operator die Typ-Information aus der linken Seite der Zuweisung geschlussfolgert wird, ist es bei Local-Variable Type Inference genau andersherum *(siehe Listing 2)*.

Einmal deklarierte „var"-Variablen sind auf den zugewiesenen Datentyp festgelegt. Möchte man nachträglich Werte anderer Typen zuweisen, erhält man Compiler-Fehler aufgrund der fehlschlagenden Typ-Konvertierung *(siehe Listing 3)*.

Bei der Deklaration muss also immer ein Wert zugewiesen werden, sonst kommt es ebenfalls zu Compiler-Fehlern, auch wenn wie im folgenden Beispiel kurz darauf garantiert ein Wert zugewiesen werden würde *(siehe Listing 4)*.

Mit „var" deklarierte Variablen sind, wie der Name vermutet lässt, nicht automatisch unveränderbar. Man kann sie jedoch mit dem Schlüsselwort „final" kombinieren. Außerdem sind sie „effectively final" (wenn es nur eine Zuweisung gibt) und können somit auch aus inneren Klassen und Lambda-Ausdrücken verwendet werden, ohne sie explizit als final zu deklarieren *(siehe Listing 5)*.

```
// Type wird bei Deklaration und Erstzuweisung vom Compiler festgelegt
var zahl = 5;
zahl = 7L; // incompatible types: possible lossy conversion from long to int

var objekt = BigDecimal.ONE;
objekt = BigInteger.TEN; // incompatible types: BigInteger cannot be converted to BigDecimal
```

Listing 3

Die Typ-Inferenz funktioniert auch mit generischen Typen und innerhalb der „foreach"-Schleife. Die Kombination mit dem „Diamond"-Operator führt allerdings aufgrund der fehlenden Typ-Information zu einem Container von Objekt-Referenzen und somit zu weniger Typsicherheit *(siehe Listing 6)*.

Die Typ-Inferenz nutzt im Übrigen den konkreten Typ, bei anonymen inneren Klassen dürfen also zwei vom selben Interface abgeleitete Instanzen nicht derselben „var"-Variablen zugewiesen werden *(siehe Listing 7)*. Das bedeutet allerdings auch, dass bei lokalen Typen (anonymen inneren Klassen-Implementierungen) neu implementierte Methoden ohne Compiler-Fehler aufgerufen werden können („reverseMe()", *siehe Listing 8)*.

Für Java 11 wird noch an einer Erweiterung der Local-Variable Type Inference gearbeitet, die in Lambda-Ausdrücken funktioniert. Das ist für die Kombination von Typ-Inferenz mit Typ-Annotationen notwendig *(siehe Listing 9)*.

Die restlichen Punkte der Release Notes *[2]* betreffen eher den infrastrukturellen Bereich und den Betrieb von Java-Anwendungen. So kann der G1, der seit Java 9 der Standard GC ist, die Full Garbage Collection parallelisieren und dadurch die Stop-the-world-Zyklen verkürzen. Der Memory Footprint wird durch das Teilen von geladenen Klassen zwischen mehreren Java-Anwendungen verringert. Darüber hinaus gibt es einen neuen, noch experimentellen JIT- (Just-in-time) Compiler (Graal) und Verbesserungen an der JVM für die Zusammenarbeit mit Docker-Containern. Zudem wird der Trust Store des OpenJDK jetzt mit einer gewissen Menge an Root-Zertifikaten ausgeliefert, was sonst nur den Oracle-Java-SE-Versionen vorbehalten war.

An der Klassen-Bibliothek (JDK) gibt es auch kleine Änderungen. Das umfasst beispielsweise eine überladene Version von „orElseThrow()" in der Klasse „Optional" und diverse Fabrik-Methoden zur Erzeugung von „unmodifiable" Collections und Stream-Kollektoren. Weitere Änderungen kann man den Release Notes *[2]* entnehmen oder über das Tool „JDK-API-Diff" *[3]* herausfinden.

Fazit

Schlussendlich lässt sich sagen, dass Java 10 ein unaufgeregtes Release ist. Beeindruckend ist jedoch, dass es wie angekündigt just in time geliefert wurde. Nun muss sich in den nächsten Jahren

```
// Deklaration von "var" nur bei direkter Initialisierung der Variable
var flag = true;
var number; // Compiler-Fehler
if (flag) {
  number = 5;
} else {
  number = 7;
}
```

Listing 4

```
// Inference bei Neuzuweisung eines Wertes (var impliziert nicht "final")
var zahl = 5;
zahl = 7;

// Inference auch mit "final" nutzbar, ansonsten gilt seit Java 8 ohnehin die "effectively final" Semantik
final var zahl = 5;
```

Listing 5

```
// Inference generischer Typen (List<String>)
var strings = Arrays.asList("World", "Java 10");

// Inference in Schleifen
for (var string : strings) {
  System.out.println("Hello " + string);
}

// Kombination mit Diamond Operator führt zu Inferenz von List<Object>
var strings = new ArrayList<>();
strings.add("Hello World");
for (var string : strings) {
  System.out.println(string.replace("World", "Java 10")); // cannot find symbol 'replace'
}

var strings2 = new ArrayList<String>();
strings2.add("Hello World");
for (var string : strings2) {
  System.out.println(string.replace("World", "Java 10"));
}
```

Listing 6

```
// Inference nutzt konkrete Typisierung
var runnable = new Runnable() {
  @Override
  public void run() {
  }
};
// incompatible types: <anonymous Runnable> cannot be converted to <anonymous Runnable>
runnable = new Runnable() {
  @Override
  public void run() {
  }
};
```

Listing 7

```
// Inference von lokalen Typen
var myReversibleStringList = new ArrayList<String>() {
  List<String> reverseMe() {
     var reversed = new ArrayList<String>(this);
     Collections.reverse(reversed);
     return reversed;
  }
};
myReversibleStringList.add("World");
myReversibleStringList.add("Hello");

System.out.println(myReversibleStringList.reverseMe());
```

Listing 8

```
// Ausblick Java 11 (JEP 323)
// Inference von Lambda Parametern
Consumer<String> printer = (var s) -> System.out.println(s); // statt s -> System.out.println(s);

// aber keine Mischung von "var" und deklarierten Typen möglich
// BiConsumer<String, String> printer = (var s1, String s2) -> System.out.println(s1 + " " + s2);

// Nützlich für Type Annotations
BiConsumer<String, String> printer = (@Nonnull var s1, @Nullable var s2) -> System.out.println(s1 + (s2 == null ? ""
: " " + s2));
```

Listing 9

zeigen, wie die Java-Welt dieses neue Auslieferungsmodell annimmt. Als Entwickler dürfen wir jetzt häufiger mit kleinen, nützlichen Sprach-Features spielen. Das sind auf jeden Fall gute Aussichten. Dann viel Spaß beim Runterladen *[4]* und Ausprobieren. Nicht vergessen, im September 2018 steht mit Java 11 bereits das nächste Release auf dem Plan.

Referenzen

[1] OpenJDK-Projektseite: *http://openjdk.java.net/projects/jdk/10/*

[2] Release Notes: *http://www.oracle.com/technetwork/java/javase/10-relnote-issues-4108729.html*

[3] JDK-API-Diff Tool: *https://gunnarmorling.github.io/jdk-api-diff/jdk9-jdk10-api-diff.html*

[4] Download: *http://www.oracle.com/technetwork/java/javase/downloads/index.html*

Falk Sippach

falk.sippach@oio.de

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