Unter dem Radar

Features in Java 9

Michael Vitz

Passend zum Schwerpunktthema „Java – die Neunte“ dreht sich auch diese Kolumne um Neuigkeiten, die uns mit Java 9 zur Verfügung gestellt werden. Da sich den großen neuen Themen wie Jigsaw, JShell und HTTP/2 in diesem Heft eigene Artikel widmen, stelle ich Ihnen ein paar der vielen weiteren, jedoch deutlich kleineren Neuerungen vor. Freuen Sie sich also auf ein buntes Sammelsurium.

Bereits seit dem 26 Mai 2016 befindet sich Java 9 im „Feature Complete“-Stadium [JDK9]. Somit ist es bereits jetzt möglich, sich einen Build des Early Access Release herunterzuladen [JDK9-EA] und neue Features auszuprobieren.

Da die vier großen Änderungen Jigsaw [JSR376], JShell [JEP222], HTTP/2 [JEP110] und das Money and Currency API [JSR354] bereits Gegenstand zahlreicher Artikel und Beiträge waren und auch in dieser Ausgabe des JavaSPEKTRUMs mit eigenen Beiträgen gewürdigt werden, stellt diese Ausgabe des Praktikers Ihnen sechs eher unbekannte neue Features vor.

Stream-Erweiterungen

Das mit Java 8 eingeführte Stream-API erhält vier neue erwähnenswerte Methoden: **`takeWhile`** und **`dropWhile`** sowie eine überladene **`iterate`**-Methode und **`ofNullable`**.

`takeWhile` und **`dropWhile`** helfen, auf einem vorhanden Stream nur eine Teilmenge zu verarbeiten. **`takeWhile`** verarbeitet auf einem geordneten Stream (s. Kasten „Streams und Ordnung“) alle Elemente solange, bis das übergebene **`Predicate`** **`true`** ergibt. Anschließend ist der Stream zu Ende. Sollte der Stream ungeordnet sein, so ist das Verhalten der Methode nicht genau spezifiziert. Eine Ausnahme ist der Fall, in dem alle Elemente verarbeitet werden, dies muss auch für einen ungeordneten Stream sichergestellt werden.

```
package de.mvitz.javaspektrum.java9.stream;
import java.util.stream.Stream;

public class TakeAndDropWhileExample {
  public static void main(String[] args) {
    Stream.of("<html>",
              "<head>",
                "<title>Foo</title>",
              "</head>",
              "<body>",
                "<h1>Foo</h1>",
                "<p>Some text</p>",
              "</body>",
            "</html>")
          .dropWhile(s -> !"<body>".equals(s))
          .skip(1)
          .takeWhile(s -> !"</body>".equals(s))
          .forEach(System.out::println);
  }
}
```

Listing 1: Verwendung von takeWhile und dropWhile

Im Gegensatz hierzu sorgt **`dropWhile`** dafür, dass solange alle Elemente verworfen werden, bis das übergebene **`Predicate`** das erste mal **`false`** zurückgibt. Auch für diese Methode gilt, dass ihr Ergebnis für ungeordnete Streams nicht spezifiziert ist, außer alle Elemente werden verworfen.

Ein Beispiel für die Verwendung von **`take`**- und **`dropWhile`** zeigt Listing 1. Das Beispiel verwirft solange alle Elemente bis zum öffnenden Body-Tag. Anschließend überspringt es dieses Element und verarbeitet alle weiteren Elemente, bis das schließende Body-Tag auftaucht. Die Ausgabe enthält somit nur noch das h1- und p-Tag.

Die statische Methode **`iterate`** auf **`java.util.stream.Stream`** kann bereits seit Java 8 dazu genutzt werden, unendlich lange Streams zu erzeugen. Java 9 fügt nun eine überladene **`iterate`**-Methode hinzu, mit der endliche Streams erzeugt werden können (s. Listing 2).

```
package de.mvitz.javaspektrum.java9.stream;
import java.util.stream.Stream;

public class IterateExample {
  public static void main(String[] args) {
    // infinite stream with limit
    Stream.iterate(1, i -> i + 1).limit(5)
      .forEach(System.out::print);
    System.out.println();

    // finite stream
    Stream.iterate(1, i -> i < 6, i -> i + 1)
      .forEach(System.out::print);
  }
}
```

Listing 2: Endliche Streams mit iterate erzeugen

Eine praktische Verwendung hiervon ist vor allem an Stellen gegeben, die mit **`Enumeration`** arbeiten. Listing 3 zeigt, wie man mittels **`iterate`** alle JVM-Properties ausgeben kann.

```
package de.mvitz.javaspektrum.java9.stream;
import java.util.Enumeration;
import java.util.stream.Stream;

public class IterateEnumertionExample {
  public static void main(String[] args) {
    final Enumeration<Object> properties = System.getProperties()
      .elements();
    if (properties.hasMoreElements()) {
      Stream.iterate(properties.nextElement(),
          p -> properties.hasMoreElements(),
          p -> properties.nextElement())
        .forEach(System.out::println);
    }
  }
}
```

Listing 3: Verarbeitung von Enumeration mit iterate

Die vierte neue Methode **`ofNullable`** hilft an allen Stellen, an denen **`null`** zurückgegeben werden kann und anschließend über eine Collection iteriert werden soll (s. Listing 4). Ähnliches konnte auch in Java 8 bereits über **`Optional::ofNullable`** erreicht werden, man spart sich jedoch den Aufruf von **`.orElse(Stream.empty())`**.

Streams und Ordnung

Ich bin bei der Erstellung dieses Artikels darüber gestolpert, dass es auch ungeordnete Streams geben kann [StreamOrder].

Das für uns relevante Merkmal ist das deterministische Verhalten. Es gibt Streams, die bei mehrmaliger Verarbeitung dieses nicht aufweisen. Dies führt dazu, dass bei der Verarbeitung von ungeordneten Streams jeder Durchlauf zu einem anderen Ergebnis führen kann.

```
package de.mvitz.javaspektrum.java9.stream;
import java.util.List;
import java.util.stream.Stream;

public class OfNullableExample {
  interface Order {}
  interface Customer {
    List<Order> getOrders();
  }
  public static void main(String[] args) {
    final Customer customer = findCustomerById(4711);
    final Stream<Order> orderStream =
      Stream.ofNullable(customer)
       .flatMap(c -> c.getOrders().stream());
  }
  private static Customer findCustomerById(long id) {
    return null; // only for demo
  }
}
```

Listing 4: Nutzung von Stream::ofNullable

Optionale Erweiterungen

Auch **java.util.Optional** wird mit Java 9 um einige hilfreiche Dinge erweitert. So lässt sich durch die neue Methode **stream()** jetzt einfach aus einem **Optional** ein **Stream** erzeugen, der entweder leer ist oder genau ein Element enthält. Viele der hierzu passenden Anwendungsfälle lassen sich zwar auch schon in Java 8 mit Aufrufen von **map** oder **flatMap** lösen, an manchen Stellen ist die Verwendung von **stream** jedoch eleganter. Zudem werden die Operationen auf einem **Stream** erst bei einem Aufruf einer terminierenden Methode ausgeführt (s. Listing 5).

```
package de.mvitz.javaspektrum.java9.optional;
import java.util.Collections;
import java.util.List;
import java.util.Optional;
import java.util.stream.Stream;

public class StreamExample {
  interface Order {}
  interface Customer {
    List<Order> getOrders();
  }
  private static Optional<Customer> findCustomerById(long id) {
    return Optional.empty();
  }
  public static void main(String[] args) {
    final Stream<Order> lazy = findCustomerById(42).stream()
      .flatMap(c -> c.getOrders().stream());
    final List<Order> eager = findCustomerById(42)
      .map(Customer::getOrders)
      .orElse(Collections.emptyList());
  }
}
```

Listing 5: Nutzung von Optional::stream

Als zweite Neuerung gibt es nun neben **orElse** auch die Methode **or**. **or** bekommt als Argument einen **Supplier** übergeben, welcher ein **Optional** vom gleichen Typ erzeugen muss. Besonders hilfreich ist dieses Muster, wenn wir Daten aus verschiedenen Quellen bekommen können (s. Listing 6).

```
package de.mvitz.javaspektrum.java9.optional;
import java.util.Optional;

public class OrExample {
  interface Customer {}
  Optional<Customer> findCustomerInMemoryCache() {
    // some implementation
  }
  Optional<Customer> findCustomerOnDiskCache() {
    // some implementation
  }
  Optional<Customer> findCustomerInDatabase() {
    // some implementation
  }
  public static void main(String[] args) {
    Optional<Customer> customer = findCustomerInMemoryCache()
     .or(() -> findCustomerOnDiskCache())
     .or(() -> findCustomerInDatabase());
  }
}
```

Listing 6: Nutzung von Optional::or

Die dritte und letzte Neuerung in **Optional** stellt die Methode **ifPresentOrElse** dar. Ziel dieser Methode ist es, Fälle zu vereinfachen, in denen etwas anderes passieren soll, wenn ein leeres **Optional** vorliegt. Sie ergänzt somit die Methode **ifPresent** (s. Listing 7).

```
package de.mvitz.javaspektrum.java9.optional;
import java.util.Optional;

public class IfPresentOrElseExample {
  interface Spoon {}
  public static void main(String[] args) {
    Optional.<Spoon>empty().ifPresentOrElse(
      (s) -> System.out.println("Found spoon:" + s),
      () -> System.out.println("There is no spoon!"));
  }
}
```

Listing 7: Nutzung von Optional::ifPresentOrElse

Process-API-Erweiterungen

Unter dem JDK Enhancement Proposal 102 [JEP102] wird das bereits existierende Process-API (**java.lang.Process**) erweitert. Eines der Ziele ist hierbei, sowohl für den JVM-Prozess, in dem man selber läuft, als auch für Prozesse, die man selber startet, mehr Informationen zur Verfügung zu stellen. Wollte man zum Beispiel bisher innerhalb seines eigenen Codes wissen, unter welcher Prozess-ID (PID) die JVM läuft, ließ sich dies nur über Umwege (z. B. Native Code per JNI) herausfinden. Mit Java 9 ist dies nun einfach und standardisiert auf jedem Betriebssystem möglich (s. Listing 8).

```
package de.mvitz.javaspektrum.java9.process;

public class PidExample {
  public static void main(String[] args) {
    System.out.println(ProcessHandle.current().getPid());
  }
}
```

Listing 8: Ausgabe der JVM-PID durch das Process-API

Neben der PID kommt man über einen Aufruf der Methode **ProcessHandle::info()** auch an detaillierte Informationen, wie den User, unter dem der Prozess läuft, wann der Prozess gestartet wurde, wie viel CPU-Zeit er bisher verbraucht hat und auch welches Kommando mit welchen Argumenten zum Starten verwendet wurde. Zur bequemen Verwendung wurde der vorhandene **java.lang.Process** so erweitert, dass man diesen nicht immer in ein **ProcessHandle** umwandeln muss. Neben dem aktuellen Prozess bietet **ProcessHandle** auch die Möglichkeit, an alle aktuell laufenden Prozesse zu gelangen (s. Listing 9).

```
package de.mvitz.javaspektrum.java9.process;

public class ListProcessesExample {
  public static void main(String[] args) {
```

```
    ProcessHandle.allProcesses()
      .map(ProcessHandle::getPid)
      .forEach(System.out::println);
  }
}
```

Listing 9: Ausgabe der PIDs für jeden laufenden Prozess

Zusätzlich geben einem die Methoden **`parent()`**, **`children()`** und **`descendants()`** die Möglichkeit, sich durch den Prozessbaum zu hangeln. Ein Finden aller Root-Prozesse auf dem System lässt sich somit wie in Listing 10 implementieren.

```
package de.mvitz.javaspektrum.java9.process;

public class FindRootProcess {
  public static void main(String[] args) {
    ProcessHandle.allProcesses()
      .filter(p -> !p. parent().isPresent())
      .map(ProcessHandle::info)
      .forEach(System.out::println);
  }
}
```

Listing 10: Finden aller Root-Prozesse

Neben dem reinen Abfragen von Prozessinformationen bietet uns das erweiterte API auch die Möglichkeit, mit **`destroy()`** und **`destroyForcibly()`** Prozesse zu beenden. Der eigene Prozess, also die JVM, in der man selber läuft, lässt sich dabei nicht beenden (s. Listing 11).

```
package de.mvitz.javaspektrum.java9.process;

public class DestroyExample {
  public static void main(String[] args) {
    ProcessHandle.current().destroy();
    // throws java.lang.IllegalStateException: destroy of
    // current process not allowed
    System.out.println("Hello, world!");
  }
}
```

Listing 11: Beenden des JVM-Prozesses per Process-API

Aus Sicherheitsgründen möchte man natürlich die Möglichkeit haben, dieses API für den Nutzer einzuschränken. Dafür prüft die Programmierschnittstelle über den SecurityManager der JVM den Wert der Runtime-Permission **`manageProcess`**. Ist dieses Recht nicht vorhanden, wird verhindert, dass der Nutzer eine **`ProcessHandle`**-Instanz erlangen kann. Feingranularer kann dieses API nicht abgesichert werden. Es greifen jedoch natürlich noch die vom Betriebssystem bereitgestellten Sicherheitskonzepte. Somit sind hier nicht mehr Möglichkeiten vorhanden, als ein nativer Prozess auch hätte.

Factory-Methoden für Collections

Häufig wird an den Java Collections kritisiert, dass die Erzeugung zu kompliziert sei. Diesem Thema widmet sich innerhalb von Java 9 der JEP 269 [JEP269]. Dieser spezifiziert für **`List`**, **`Set`** und **`Map`** statische Methoden, die diese erzeugen (s. Listing 12).

```
package de.mvitz.javaspektrum.java9.collection;
import java.util.*;

public class CollectionFactoryExample {
  public static void main(String[] args) {
    List.of(1, 2, 3);
    Set.of(1, 2, 1, 4, 5);
    Map.of("foo", "bar");
    Map.ofEntries(Map.entry("foo", "bar"));
  }
}
```

Listing 12: Factory-Methoden für Collections

Hierbei ist zu beachten, dass alle hiermit erzeugten Collections Immutable sind. Dieser JEP hat sich explizit nicht zum Ziel gesetzt, ein großes Builder-API für alle Arten von Collections zu sein. Aus diesem Grunde werden hier auch nicht bereits bekannte Implementierungen zurückgegeben, sondern es wurden neue implementiert. Diese Implementierungen sind speziell auf Collections mit wenigen Elementen abgestimmt und können bei größeren Mengen unperformantes Verhalten aufweisen.

Neues Versionsnummernschema

Seit der Übernahme von Java durch Oracle wurde das Versionierungsschema bereits mehrfach geändert. Allerdings haben diese Änderungen wenig zum Verständnis beigetragen. Aktuell gibt es beispielsweise das Oracle Java 8 JDK in den beiden Versionen 8u101 und 8u102. Ich muss bei jedem Update erst einmal nachgucken, welche von den beiden Versionen ich haben möchte und was genau der Unterschied ist.

Genau diese Probleme zu beseitigen, hat sich JEP 223 [JEP223] zum obersten Ziel gesetzt. Die Versionsnummer besteht nun aus einer nicht leeren Menge von Elementen, die durch Punkte getrennt werden. Jedes Element ist hierbei entweder 0 oder eine ganze positive Zahl ohne führende Nullen. Zudem ist das letzte Element niemals 0. Weiterhin haben die ersten drei Elemente der Versionsnummer eine spezielle Bedeutung.

Das erste Element der Versionsnummer ist der Major-Teil. Für Java 9 ist dies die 9. Erhöht wird dieses Element bei jedem Release, das signifikante Änderungen beinhaltet. Zudem sind bei einer Erhöhung dieses Elementes auch inkompatible Änderungen erlaubt und es kann angekündigt werden, dass gewisse Features in Zukunft, das heißt frühestens mit dem nächsten Major-Release, entfernt werden. Wird dieses Element erhöht, so werden alle folgenden Elemente auf 0 gesetzt und somit entfernt.

Als zweites Element kann ein Minor-Teil auftauchen. Dieser Teil wird immer dann erhöht, wenn kleinere kompatible Änderungen oder Updates von APIs vorgenommen werden.

Durch das dritte Element wird das Security-Level angezeigt. Dieser Teil wird mit jedem Sicherheitsupdate erhöht und wird *nicht* auf 0 gesetzt, wenn es eine neue Minor-Version gibt. Somit kann man alleine anhand dieses Elementes sehen, welche von zwei Versionen desselben Major-Releases mehr Sicherheitsupdates erhalten hat.

Die erste Java 9-Version sollte somit die Versionsnummer 9 erhalten.

Neben der Versionsnummer wird auch noch ein sogenannter Versionsstring definiert. Dieser besteht aus der Versionsnummer, dem Pre-Release, der Build-Nummer und gegebenenfalls auch noch weiteren Informationen. So ergibt die Ausgabe für das aktuelle Early Access Release: 9-ea+128.

Die nächste durch diesen JEP spezifizierte Änderung ergibt sich automatisch durch diese Regeln. Java 9 wird das erste Release, das nicht mehr mit *1.x* beginnt. Bemerkenswert ist diese Änderung, da sie die durchaus reale Gefahr beinhaltet, dass Code, der bisher geschrieben wurde, um Java-Versionen zu vergleichen, nicht mehr funktioniert [ANT], oder Tooling von Entwicklern gestört wird. Siehe Listing 13 für die unter Mac gebräuchlichen Bash-Aliasse, um die Java-Version zu ändern. Bereits hier hat die Änderung der Versionsnummer eine Auswirkung.

```
alias java8="export JAVA_HOME=$(/usr/libexec/java_home -v1.8)"
alias java9="export JAVA_HOME=$(/usr/libexec/java_home -v9)"
```

Listing 13: Mac-Bash-Aliasse für Java-Versionswechsel

Neben der reinen Definition bietet Java 9 mittels `java.lang.Runtime.Version` auch eine Programmierschnittstelle an, um Java-Versionsnummern zu parsen und zu vergleichen (s. Listing 14

```
package de.mvitz.javaspektrum.java9.version;

public class VersionExample {
  public static void main(String[] args) {
    final Runtime.Version version = Runtime.version();
    System.out.println(version.toString());
    System.out.println(version.major());
    System.out.println(version.minor());
    System.out.println(version. security ());

    final Runtime.Version newVersion = Runtime.Version.parse("9.1");
    System.out.println(newVersion.compareTo(version));
  }
}
```

Listing 14: API für Java-Versionsnummern

Zu guter Letzt standardisiert dieser JEP noch die Rückgabewerte für die folgenden fünf System-Properties:

- java.version
- java.runtime.version
- java.vm.version
- java.specification.version
- java.vm.specification.version

Multi-Release-JAR-Dateien

Die letzte Neuerung, die ich Ihnen in dieser Kolumne vorstellen möchte, wird im JEP 238 [JEP238] spezifiziert. Die Umsetzung dieses JEPs erlaubt, dass man innerhalb einer JAR-Datei kompilierte Class-Dateien hat, die nur in bestimmten Java-Versionen genutzt werden.

Für Applikationen hat dieses Feature nur wenig Relevanz, Bibliotheken hingegen können hiervon profitieren. Beispielsweise könnte man, um die aktuelle PID abzufragen, ab Java 9 die neuen Features des Process-API nutzen, gleichzeitig jedoch für ältere JDKs andere Wege einschlagen.

Hierzu muss das Manifest in der JAR-Datei das Property `Multi-Release` auf `true` setzen. Anschließend können verschiedene Versionen derselben Klassen ausgeliefert werden. Listing 15 zeigt, wie eine solche JAR-Datei aussieht.

```
jar root
  - A.class
  - B.class
  - C.class
  - META-INF
    - versions
      - 8
        - A.class
        - B.class
      - 9
        - A.class
```

Listing 15: Aufbau einer Multi-Release-JAR-Datei

Wird dieses JAR mit Java 9 verwendet, wird die spezifisch für Java 9 abgelegte Klasse A, die für Java 8 erstellte Klasse B und die von allen Versionen genutzte Klasse C verwendet. Mit Java 8 wird die dort abgelegte Klasse A genutzt. Alle älteren Java-Runtimes (und damit alle, die dieses Format nicht unterstützen) nutzen nur die im Root gefundenen Klassen. Ein Beispiel, wie dies aktuell mit Maven machbar ist, finden Sie im Projekt hboutemy/maven-jep238 [MRDemo].

Fazit

Leider hat auch diese Kolumne nur Platz, um fünf kleine und eher unbekannte Features vorzustellen. Die Webseite zum JDK 9 [JDK9] listet alle JEPs auf und ist einen Blick wert. Ich bin mir sicher, Sie finden dort noch weitere interessante Neuerungen.

Ich hoffe, ich konnte Ihnen noch die ein oder andere bisher unbekannte Neuerung vorstellen, und freue mich wie immer auf Feedback.

Links

[ANT] ASF Bugzilla – Bug 59863 – JDK9 version number drops the leading "1.“, `https://bz.apache.org/bugzilla/show_bug.cgi?id=59863`
[JDK9] JDK 9, `http://openjdk.java.net/projects/jdk9/`
[JDK9-EA] JDK™ 9 Early Access Releases, `https://jdk9.java.net/download/`
[JEP102] JDK Enhancement Proposal 102: Process API Updates, `http://openjdk.java.net/jeps/102`
[JEP110] JDK Enhancement Proposal 110: HTTP/2 Client, `http://openjdk.java.net/jeps/110`
[JEP222] JDK Enhancement Proposal 222: jshell: The Java Shell (Read-Eval-Print Loop), `http://openjdk.java.net/jeps/222`
[JEP223] JDK Enhancement Proposal 223: New Version-String Scheme, `http://openjdk.java.net/jeps/223`
[JEP238] JDK Enhancement Proposal 238: Multi-Release JAR Files, `http://openjdk.java.net/jeps/238`
[JEP269] JDK Enhancement Proposal 269: Convenience Factory Methods for Collections, `http://openjdk.java.net/jeps/269`
[JSR354] Java Specification Request 354: Money and Currency API, `https://jcp.org/en/jsr/detail?id=354`
[JSR376] Java Platform Module System (JSR 376), `http://openjdk.java.net/projects/jigsaw/spec/`
[JSR379] Java SE 9 Platform Umbrella JSR (379), `http://openjdk.java.net/projects/jdk9/spec/`
[MRDemo] `https://github.com/hboutemy/maven-jep238`
[StreamOrder] `https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Ordering`

Enthaltene Beispiele

Sämtliche Beispiele dieser Kolumne können natürlich von Ihnen ausprobiert werden. Dazu finden Sie sämtliche Sourcen unter `https://github.com/mvitz/javaspektrum-java9`.

Diese wurden mit Build 128 des JDK 9 EA Releases gebaut und nutzen Apache Maven 3.3.3 als Build-Tool. Nach einem Import dieses Projektes in meine IDE (IntelliJ IDEA 2016.2) und dem Hinzufügen des JDKs hat alles ohne weitere Konfiguration funktioniert.

Michael Vitz ist Consultant bei innoQ und verfügt über mehrjährige Erfahrung in der Entwicklung und im Betrieb von JVM-basierten Systemen. Zurzeit beschäftigt er sich vor allem mit den Themen DevOps, Continuous Delivery und Cloud-Architekturen sowie Clojure. E-Mail: michael.vitz@innoq.com