3 Neues und Änderungen in JDK 9

Nachdem wir diverse kleinere Änderungen in der Syntax der Sprache Java kennengelernt haben, wollen wir uns in diesem Kapitel relevante Erweiterungen im JDK anschauen. Erwähnenswert sind das neue Process-API, die Ergänzungen im Stream-API sowie in java.util.Optional<T>, aber auch die neuen Collection-Factory-Methoden. Darüber hinaus findet man Ergänzungen in den Klassen java.io.InputStream und java.util.ResourceBundle sowie diverse Neuerungen, unter anderem auch im Bereich Concurrency: Die Klasse java.util.concurrent.CompletableFuture<T> bietet mehr Funktionalität. Wesentlicher ist vermutlich aber die Unterstützung von Reactive Streams durch die im Package java.util.concurrent neu eingeführte Klasse Flow. Auch im Bereich von Unicode, Grafikformaten sowie Desktop-Technologie mit HiDPI-Support und Unterstützung der Taskbar bzw. des Docks hat sich einiges weiterentwickelt. Außerdem wurden Utility-Klassen wie java.util.Objects und java.util.Arrays sinnvoll ergänzt.

3.1 Neue und erweiterte APIs

Wie bereits angedeutet, wurden in den APIs des JDKs diverse Verbesserungen vorgenommen und Neuerungen integriert, die wir nun thematisch gruppiert anschauen wollen. Dabei beginnen wir mit den Neuerungen im Process-API.

3.1.1 Das neue Process-API

Bis einschließlich JDK 8 sind die Möglichkeiten recht eingeschränkt, wenn es darum geht, Prozesse des Betriebssystems zu kontrollieren und zu verwalten.

PID mit JDK 8 ermitteln

Ein simples Beispiel ist die Ermittlung der ID eines Prozesses, kurz PID genannt. Je nach Plattform muss man dies mit Java 8 unterschiedlich implementieren. Für Linux und Mac OS führt man dazu ein Shell-Kommando mit der Methode exec() aus der Klasse java.lang.Runtime aus – das korrespondierende Windows-Kommando unter Einsatz der Powershell ist als commandsWin definiert:

```
private static long getPidJdk8Style() throws InterruptedException,
                                              IOException
{
    // $PPID steht für Parent Process ID, also hier derjenigen der JVM
    final String[] commands = new String[]{ "/bin/sh", "-c", "echo $PPID" };

    // Komplexere Windows-Variante
    final String[] commandsWin = new String[]{ "powershell",
                           "gwmi win32_process | ?{$_.ProcessID -eq $pid} | " +
                           "select ParentProcessID | fw -c 2"};

    // Für Windows: commandsWin nutzen
    final Process proc = Runtime.getRuntime().exec(commands);

    if (proc.waitFor() == 0)
    {
        try (final InputStream in = proc.getInputStream())
        {
            final int available = in.available();
            final byte[] outputBytes = new byte[available];
            in.read(outputBytes);

            final String pid = new String(outputBytes);
            // rein theoretisch wäre hier eine NumberFormatException abzufangen
            return Long.parseLong(pid.trim());
        }
    }
    throw new IllegalStateException("PID is not accessible");
}
```

Im Listing wird ersichtlich, wie aufwendig das Auslesen der Informationen aus dem `InputStream` des erzeugten Prozesses ist und dass sogar rein theoretisch der Fall behandelt werden müsste, dass keine gültige Zahl zurückgeliefert wird – was wohl nur in Ausnahmesituationen bei Lesefehlern der Streams geschehen könnte und was wir hier der Vereinfachung halber außer Acht lassen.

Hinweis: Die Klasse `Runtime`

Java ist weitestgehend betriebssystemunabhängig. Manchmal ist es aber wünschenswert, externe Programme in Form von Prozessen auszuführen. Dies ist durch den Aufruf von `exec()` der Klasse `Runtime` möglich. Dadurch entsteht ein neues `java.lang.Process`-Objekt. Mit dessen Methode `waitFor()` kann man ***blockierend*** auf das Ende des Prozesses warten und anschließend mit der Methode `exitValue()` den Rückgabewert erfragen.

Zudem erhält man über `getOutputStream()`, `getInputStream()` und `getErrorStream()` Zugriff auf die dem `Process` zugeordneten Ein- und Ausgabeströme. Dabei sind allerdings einige Details zu beachten. Zunächst ist der Zugriff auf diese Streams insofern wichtig, als dass der erzeugte Prozess keine Konsole zur Ausgabe hat und dadurch die Ausgaben auf `System.out` an den Vaterprozess weitergeleitet werden. Auch kann man Eingaben an einen Subprozess weiterleiten. Dabei empfiehlt es sich, möglichst zeitnah aus den Streams zu lesen und nicht erst nach Terminierung des Prozesses, da es ansonsten zu Blockierungen und Deadlocks kommen kann.

PID mit JDK 9 ermitteln

Die Abfrage der PID mit Java 9 wird mithilfe der Klasse `java.lang.ProcessHandle` nun deutlich kürzer, besser lesbar und verständlich:

```
private static long getPidJdk9Style()
{
    return ProcessHandle.current().pid();
}
```

Neben den genannten Vorteilen bietet die Methode `pid()` einen betriebssystemunabhängigen Weg zur Ermittlung der Prozess-ID (zumindest aus Sicht des Aufrufers).

Beispielprogramm

Wir wollen die beiden Methoden in Aktion erleben und deren Ergebnis prüfen. Dazu schreiben wir folgendes Programm:

```
public static void main(final String[] args) throws InterruptedException,
                                                    IOException
{
    System.out.println("PID old: " + getPidJdk8Style());
    System.out.println("PID new: " + getPidJdk9Style());
}
```

Listing 3.1 *Ausführbar als* '**PIDEXAMPLE**'

Startet man das Programm PIDEXAMPLE, so sieht man anhand der Ausgabe identischer Prozess-IDs, dass beide Varianten funktional übereinstimmen, beispielsweise wie folgt:

```
PID old: 41948
PID new: 41948
```

Das Interface `ProcessHandle`

Neben der PID kann man mithilfe von `ProcessHandle` noch diverse weitere Informationen zu Prozessen auslesen. Dazu gibt es unter anderem folgende Methoden:

- `current()` – Ermittelt den aktuellen Prozess als `ProcessHandle`.
- `info()` – Stellt Infos zum Prozess in Form des inneren Interface `ProcessHandle.Info` bereit, etwa zu Benutzer, Kommando usw.
- `info().command()` – Gibt das Kommando als `Optional<String>` aus einem `ProcessHandle.Info` zurück.
- `info().user()` – Liefert den Benutzer als `Optional<String>` aus einem `ProcessHandle.Info`.
- `info().totalCpuDuration()` – Ermittelt aus den Infos die benötigte CPU-Zeit als `Optional<Duration>`. Die Klasse `java.time.Duration` entstammt dem mit JDK 8 neu eingeführten Date and Time API.[1]

[1] Einen Kurzüberblick für dieses Datums-API bietet Anhang A.

Zum besseren Verständnis dieser Methoden betrachten wir ein Beispiel:

```
public static void main(final String[] args)
{
    final ProcessHandle current = ProcessHandle.current();
    printInfo(current);
}

private static void printInfo(final ProcessHandle current)
{
    System.out.println("PID:       " + current.pid());
    System.out.println("Info:      " + current.info());
    System.out.println("Command:   " + current.info().command());
    System.out.println("CPU-Usage: " + current.info().totalCpuDuration());
}
```

Listing 3.2 *Ausführbar als* '**PROCESSHANDLEEXAMPLE**'

Das Programm PROCESSHANDLEEXAMPLE gibt in etwa Folgendes aus (gekürzt):

```
PID:       13670
Info:      [user: Optional[michaeli], cmd: /Library/Java/JavaVirtualMachines/jdk
      -9.jdk/Contents/Home/bin/java, args: [-Dfile.encoding=UTF-8, -Duser.country
      =DE, -Duser.language=de, -Duser.variant, -cp, /Users/michaeli/Desktop/
      PureJava9/quelltext/build/libs/Java9-all.jar:/Users/michaeli/Desktop/
      PureJava9/quelltext/build/requiredLibs, ch3_1.processapi.
      ProcessHandleExample], startTime: Optional[2017-04-06T17:21:56.927Z],
      totalTime: Optional[PT0.230888S]]
Command:   Optional[/Library/Java/JavaVirtualMachines/jdk-9.jdk/Contents/Home/
      bin/java]
CPU-Usage: Optional[PT0.308141S]
```

Neben der PID werden diverse Informationen aus dem `Info`-Objekt aufgelistet, exemplarisch separat nochmals die Werte für `command()` und `totalCpuDuration()`. Für das mit `command()` als `Optional<String>` ermittelte Kommando erkennen wir, dass es sich um das Programm `java` aus JDK 9 handelt, das laut `totalCpuDuration()` etwa 0.31 Sekunden CPU-Zeit verbraucht hat, wie man anhand von `Optional<Duration>` sieht.

Alle Prozesse mit `ProcessHandle` abfragen

Neben Informationen zum aktuellen Prozess lassen sich Informationen für alle Prozesse des Benutzers sowie alle Subprozesse zu einem Prozess wie folgt ermitteln:

- `allProcesses()` – Liefert alle Prozesse als `Stream<ProcessHandle>`.
- `children()` – Ermittelt zu einem Prozess alle seine (direkten) Subprozesse als `Stream<ProcessHandle>`.

Im nachfolgenden Beispiel iterieren wir über das Ergebnis von `allProcesses()` und geben Infos zu solchen Prozessen aus, die Subprozesse besitzen. Die Anzahl an Subprozessen können wir durch Aufruf von `children().count()` erfragen:

```
public static void main(final String[] args)
{
    System.out.println("All Processes:");
    showInfoForAllProcesses();
}

private static void showInfoForAllProcesses()
{
    ProcessHandle.allProcesses().forEach(processHandle ->
    {
        final Stream<ProcessHandle> children = processHandle.children();
        final long count = children.count();
        if (count > 0)
        {
            System.out.println("Info: " + processHandle.info() +
                               " has " + count + " children");
        }
    });
}
```

Listing 3.3 *Ausführbar als* '**ALLPROCESSHANDLESEXAMPLE**'

Das Programm ALLPROCESSHANDLESEXAMPLE produziert die folgenden Ausgaben (gekürzt), die eine Liste der zurückgelieferten Informationen widerspiegeln:

```
All Processes:
Info: [user: Optional[michaeli], cmd: /Applications/Adobe Acrobat Reader DC.app/
      Contents/MacOS/AdobeReader, args: [-psn_0_3822501], startTime: Optional
      [2016-08-02T21:16:30.322Z]] has 3 children
...
Info: [user: Optional[michaeli], cmd: /System/Library/CoreServices/Dock.app/
      Contents/MacOS/Dock, startTime: Optional[2016-07-24T08:17:12.938Z]] has 1
      children
Info: [user: Optional[root], startTime: Optional[2016-07-24T08:16:40.564Z]] has
      285 children
...
```

Prozesse mit `ProcessHandle` kontrollieren

Neben der Bereitstellung und Abfrage von Informationen zu Prozessen existieren auch verschiedene Möglichkeiten, Prozesse zu beenden sowie auf das Ende eines Prozesses zu reagieren. Dazu findet man im Interface `ProcessHandle` folgende Methoden:

- `of(long)` – Liefert ein `Optional<ProcessHandle>` zu einer gegebenen PID.
- `destroy()` – Terminiert einen Prozess, sofern dies erlaubt ist. Ansonsten, etwa für den mit `current()` ermittelten Prozess, wird eine Exception ausgelöst:

```
Exception in thread "main" java.lang.IllegalStateException: destroy of
      current process not allowed
```

- `onExit()` – Liefert ein `CompletableFuture<ProcessHandle>` zurück, das man dazu nutzen kann, verschiedene Aktionen als Reaktion auf das Ende eines Prozesses auszuführen.

Zur Demonstration dieser Methoden wollen wir ein Beispiel erstellen. Es soll zunächst mit `Runtime.exec()` ein Prozess gestartet werden. Als Rückgabe erhält man ein `Process`-Objekt. Dieses bietet seit JDK 9 ebenfalls die Methode `pid()` sowie diverse andere, die auch durch `ProcessHandle` bereitgestellt werden. Auch kann man ein `Process`-Objekt durch einen Aufruf von `toHandle()` in ein `ProcessHandle`-Objekt transformieren:

```
public static void main(final String[] args) throws InterruptedException,
                                                    IOException
{
    // Prozess erzeugen
    final String command = "sleep 60s";
    final String commandWin = "cmd timeout 60";
    final Process sleeper = Runtime.getRuntime().exec(command);
    System.out.println("Started process is " + sleeper.pid());

    // Process => ProcessHandle
    final ProcessHandle sleeperHandle = ProcessHandle.of(sleeper.pid()).
                                  orElseThrow(IllegalStateException::new);
    final ProcessHandle sleeperHandle2 = sleeper.toHandle();
    System.out.println("Same handle? " + sleeperHandle.equals(sleeperHandle2));

    // Exit Handler registrieren
    final Runnable exitHandler = () -> System.out.println("exitHandler called");
    sleeperHandle.onExit().thenRun(exitHandler);
    System.out.println("Registered exitHandler");

    // Den Prozess zerstören und ein wenig warten,
    // damit onExit() ausgeführt werden kann
    System.out.println("Destroying process " + sleeperHandle.pid());
    sleeperHandle.destroy();
    Thread.sleep(500);
}
```

Listing 3.4 *Ausführbar als* **'CONTROLPROCESSEXAMPLE'**

Startet man das Programm CONTROLPROCESSEXAMPLE, so können wir anhand der folgenden Ausgaben recht gut die im Listing definierten Aktionen nachvollziehen:

```
Started process is 60392
Same handle? true
Registered exitHandler
Destroying process 60392
exitHandler called
```

Fazit

Wir haben in verschiedenen Beispielen kennengelernt, wie man mit dem neuen Process-API mit Prozessen des Betriebssystems interagieren oder zumindest Informationen darüber gewinnen kann. Die große Stärke des Process-APIs ist, dass die Aktion aus Sicht eines Java-Entwicklers betriebssystemunabhängig erfolgen kann. Damit kommt man Javas Versprechen von einer weitestgehend betriebssystemunabhängigen Programmierung wieder ein Stück näher.

3.1.2 Collection-Factory-Methoden

Das Erzeugen von Collections für ein paar vordefinierter Werte ist in Java mitunter etwas umständlich. Sprachen wie Groovy oder Python bieten dafür eine spezielle Syntax, sogenannte ***Collection-Literale***. Bereits 2009 hat man auch für Java über eine Integration einer einfacheren Schreibweise zur Erzeugung von und zum Zugriff auf Collections nachgedacht. Leider wurde nichts Derartiges realisiert, obwohl es einige, vielversprechende Vorschläge gab.

Vorschlag: Collection-Literale und Collection-Erzeugung

Nachfolgendes Listing zeigt, wie eine mögliche Syntax für Collection-Literale für die im Package `java.util` definierten Collections `List<E>`, `Set<E>` und `Map<K,V>` aussehen könnte. Dabei werden die Elemente der Collection in geschweifte oder eckige Klammern eingeschlossen:

```
// Leider weder mit JDK 8 noch JDK 9 umgesetzt
final List<String> newStyleList = ["item1", "item2"];
final Set<String> newStyleSet = {"Tim", "Mike" };
final Map<String, String> newStyleMap = ["key1" : "value1", "key2" : "value2"];
```

Vorschlag: Collection-Literale und Datenzugriff

Während man auf Arrays indiziert mit eckigen Klammern zugreift, muss man für Listen die Methode `get(int)` und für Maps die Methode `get(K)` nutzen. Wertzuweisungen erfolgen durch Aufruf von `set(int,E)` bzw. `put(K,V)`.

Für indizierte Zugriffe und Wertzuweisungen auf Listen bzw. Zugriffe auf Schlüssel und Werte bei Maps war für Java eine Notation analog zu indizierten Array-Zugriffen vorgesehen, wodurch explizite Methodenaufrufe überflüssig geworden wären:

```
// Collection-Literale leider weder mit JDK 8 noch JDK 9 umgesetzt
newStyleList[0] = "newValue";
final String valueAtPos0 = newStyleList[0]; // => newValue
```

Realisierung mit JDK 9

In Java 9 gibt es leider keine Collection-Literale in der zuvor beschriebenen Form. Stattdessen wurde eine Armada an Factory-Methoden mit den Namen `of()` die Interfaces `List<E>`, `Set<E>` und `Map<K,V>` integriert, die sich wie folgt zur Erzeugung von Collections nutzen lassen:

```
final List<String> names = List.of("Mike", "Tim", "Tom");
final Set<Integer> primeNumbers = Set.of(2, 3, 5, 7, 11);
final Map<Integer, String> numberMapping = Map.of(5, "five", 6, "six");
```

Schauen wir uns dazu ein Beispiel für jede der Methoden an, dass darüber hinaus die zusätzliche Methode `ofEntries()` im Interface `Map<K,V>` und deren Nutzung verdeutlicht:[2]

```
public static void main(final String[] args)
{
    // Collection Factory Methods
    final List<String> names = List.of("MAX", "MORITZ", "MIKE");
    names.forEach(name -> System.out.println(name)); // oder System.out::println

    final Set<Integer> numbers = Set.of(1, 2, 3);
    numbers.forEach(number -> System.out.println(number));

    final Map<Integer, String> mapping = Map.of(5, "five", 6, "six");
    final Map<Integer, String> mapping2 = Map.ofEntries(entry(5, "five"),
                                                        entry(6, "six"));
    mapping.forEach((key, value) -> System.out.println(key + ":" + value));
    mapping2.forEach((key, value) -> System.out.println(key + ":" + value));
}
```

Listing 3.5 *Ausführbar als* '**COLLECTIONFACTORYMETHODSEXAMPLE**'

Das Programm COLLECTIONFACTORYMETHODSEXAMPLE gibt Folgendes aus:

```
MAX
MORITZ
MIKE
1
2
3
6:six
5:five
6:six
5:five
```

Die Reihenfolge der Elemente bei Sets und Maps ist bei der Nutzung der Collection-Factory-Methoden allerdings nicht definiert und insbesondere wird die Einfügereihenfolge oftmals nicht beibehalten. Auch die Reihenfolge bei einer Iteration ist zufällig, sodass es durchaus etwa zu folgenden Ausgaben kommen kann:

```
MAX
MORITZ
MIKE
3
2
1
5:five
6:six
5:five
6:six
```

[2] Zur besseren Lesbarkeit erfolgt dazu ein statischer Import von `java.util.Map.entry`. Zudem sieht man, dass `ofEntries()` eine klarere Gruppierung von Schlüssel und Wert erlaubt, was vor allem dann hilfreich ist, wenn diese vom selben Typ sind.

Besonderheiten bei der Konstruktion In den jeweiligen Collection-Interfaces sind die Collection-Factory-Methoden explizit für 0 bis 10 Parameter sowie als Vararg-Variante definiert. Allerdings führt diese performanceoptimierte Realisierung zu einer Menge an Methoden. Zudem gibt es für den Fall mit dem Vararg-Parameter noch einen speziellen `switch-case`, der die spezialisierten Methoden aufruft:

```
static <E> List<E> of() {
    return ImmutableCollections.List0.instance();
}

static <E> List<E> of(E e1) {
    return new ImmutableCollections.List1<>(e1);
}

...

@SafeVarargs
@SuppressWarnings("varargs")
static <E> List<E> of(E... elements) {
    switch (elements.length) {
        case 0:
            return new ImmutableCollections.List0<>();
        case 1:
            return new ImmutableCollections.List1<>(elements[0]);
        case 2:
            return new ImmutableCollections.List2<>(elements[0], elements[1]);
        default:
            return new ImmutableCollections.ListN<>(elements);
    }
}
```

Glücklicherweise sieht man diese Implementierungsdetails nur dann, wenn man in den JDK-Sourcecode schaut. Allerdings scheint diese Unschönheit zumindest in der IDE bei der Auto-Completion durch, die eine Horde gleichnamiger Methoden anbietet.

Space Efficiency Das obige Listing zeigt, dass spezielle innere Klassen aus der Klasse `java.util.ImmutableCollections` erzeugt werden. Neben ihrer Unveränderlichkeit und auf Arrays basierenden Spezialimplementierungen zeichnen sich die durch die Collection-Factory-Methoden erzeugten Collections dadurch aus, dass sie viel weniger Speicherplatz verbrauchen als diejenigen aus dem Collections-Framework.

Betrachten wir ein `java.util.HashSet<String>` mit zwei Elementen bzw. ein mit den neuen Collection-Factory-Methoden erzeugtes `Set<String>`. Die folgende Variante für JDK 8 benötigt über 150 Bytes, wogegen diejenige für JDK 9 lediglich um 20 Bytes verbraucht:[3]

```
final Set<String> namesSet = new HashSet<>();
names.add("MAX");
names.add("TIM");
final Set<String> unmodifiableNamesJDK8 = Collections.unmodifiableSet(namesSet);

final Set<String> unmodifiableNamesJDK9 = Set.of("MAX", "TIM");
```

[3] Die Zahlen basieren auf Stuart Marks' Vortrag »New Collections APIs for Java 9«, der online unter `https://www.youtube.com/watch?v=OJrIMv4dAek` verfügbar ist.

Besonderheiten bei Duplikaten

Beim Einsatz der Collection-Factory-Methoden sollte man ein Detail für `Set<E>` und `Map<K,V>` kennen: Bekanntermaßen modelliert ein `Set<E>` das mathematische Konzept einer Menge und enthält somit keine Duplikate. Das gilt auch für die Schlüssel in Maps. Diese Eigenschaft wurde bei den bisherigen Collections automatisch sichergestellt, indem beim Einfügen von Elementen gegebenenfalls Duplikate aussortiert wurden.[4] Das war für diverse Anwendungsfälle ein recht praktisches Feature. Die Collection-Factory-Methoden weisen allerdings eine nicht überraschungsfreie Besonderheit auf: ***Für Sets wird zum Konstruktionszeitpunkt die Duplikatfreiheit geprüft. Ist diese nicht gegeben, so wird eine Exception ausgelöst. Gleiches gilt auch für Duplikate bei den Schlüsseln von Maps.*** Bei Listen findet dagegen – wie erwartet – keine Duplikatsprüfung statt. Schauen wir uns ein Beispiel an:

```
final Set<String> names = Set.of("MAX", "Moritz", "MAX");
```

Bei dieser Variablendefinition kommt es zur Laufzeit zu folgender Exception:

```
java.lang.IllegalArgumentException: duplicate element: MAX
    at java.base/java.util.ImmutableCollections$SetN.<init>(ImmutableCollections.
          java:462)
```

Weil die Collections direkt anhand der übergebenen Werte konstruiert werden, kann man jedoch auch einen Grund für dieses Verhalten finden, nämlich die Vermeidung von Inkonsistenzen durch Flüchtigkeitsfehler in Form einer Mehrfachangabe von Werten.

Hinweis: Spezifische Typen erzeugee

Die durch die inneren Klassen von `ImmutableCollections` erzeugten Collections erfüllen die jeweiligen Interfaces aus dem Collections-Framework. Wenn man jedoch eine Sortierung wie bei `java.util.TreeSet<E>` und `java.util.TreeMap<K,V>` wünscht, wird dies nicht direkt unterstützt.

Fazit

Die Collection-Factory-Methoden können mich nicht vollständig überzeugen. Für die Definition kleiner Wertbestände gefallen Sie mir eigentlich schon recht gut, auch wenn sie nicht ganz so elegant in der Schreibweise sind, wie es Collection-Literale wären. Positiv können die Collection-Factory-Methoden ihre enorme Speichereffizienz ins Feld führen. Oftmals ist zudem die Unveränderlichkeit von Vorteil, etwa wenn man diese Collections an andere Programmteile weitergibt.

[4] Das erfordert, dass die steuernden Methoden wie `equals()`, `hashCode()` usw. korrekt implementiert sind. Details finden Sie in meinem Buch »Der Weg zum Java-Profi« [4].

Durchaus praktisch wäre es gewesen, auch in den spezifischen Collections, wie ArrayList<E> oder TreeSet<E>, jeweils of()-Methoden anzubieten, um bei Bedarf einen speziell benötigten Typ erzeugen zu können.

Als potenziellen Negativpunkt sehe ich, dass die Duplikatbehandlung für Set<E> zumindest nicht überraschungsfrei ist – die ausgelöste Exception ist meiner Meinung nach sogar kontraintuitiv. Auch sollte man nicht zu sehr hinter die Kulissen der Collection-Factory-Methoden schauen, denn deren Realisierung als eine Horde überladener, statischer Methoden in Interfaces ist eher fragwürdig.

Insgesamt lässt sich festhalten, dass die zuvor genannten Negativpunkte für den Einsatz in der Praxis eine untergeordnete Rolle spielen. Als Nutzer fällt vor allem die Vereinfachung bei der Konstruktion von Collections positiv ins Gewicht.

3.1.3 Reactive Streams und die Klasse Flow

Die Unterstützung von Reactive Streams[5] ist die größte Neuerung in JDK 9 im Bereich der Concurrency. Dabei spielen die Klasse java.util.concurrent.Flow und deren innere Interfaces eine wesentliche Rolle.

Schnelleinstieg Reactive Streams

Themen wie Performance, Skalierbarkeit und Ausfallsicherheit gewinnen heutzutage zunehmend an Bedeutung. Das adressieren Frameworks wie Akka, RxJava, Vert.x und Reactor, indem sie eine asynchrone, eventgetriebene, nicht blockierende Verarbeitung unterstützen – man spricht von Reactive Programming.[6]

Das Essenzielle ist dabei, dass voneinander unabhängig ausgeführte Verarbeitungseinheiten über Events bzw. Messages miteinander kommunizieren. Dazu registrieren sich Subscriber (Empfänger) bei einem Publisher (Sender). Bei der Kommunikation kann es zu Engpässen kommen, wenn ein Publisher zu langsam oder zu schnell arbeitet. Im ersten Fall werden Subscriber eventuell nur teilweise ausgelastet, im zweiten Fall überlastet. Früher hat man als Abhilfe einen Eingangspuffer aufseiten des Empfängers genutzt. Um Speicherprobleme zu vermeiden, sollte der Eingangspuffer größenbeschränkt sein. Alternativ kann der Sender auch nur eine gewisse Maximalmenge an Daten verschicken. Dadurch verschwendet man aber potenziell Performance, wenn der Empfänger eine schnellere Abarbeitung bewerkstelligen könnte.

Reactive Streams adressieren die beschriebenen Engpässe im Datenfluss. Dazu verfolgen Reactive Streams das Konzept der ***Backpressure*** (Gegendruck oder Rückstau). Die Idee ist recht simpel: Der Empfänger kann dem Sender mitteilen, wie viele Daten er verarbeiten kann. Der Subscriber fordert aktiv Daten an und der Publisher schickt maximal so viele Daten wie angefordert. Dadurch ermöglichen Reactive Streams eine Selbstregulierung, bei der die Datenverarbeitung an einen möglicherweise temporär

[5] http://www.reactive-streams.org/

[6] Details dazu finden Sie unter http://www.reactivemanifesto.org/.

langsamen Verarbeiter angepasst wird. Schauen wir uns im Anschluss die mit JDK 9 eingeführte Umsetzung an.

Abbildung 3-1 *Schematische Zusammenarbeit zwischen Publisher und Subscriber*

Die Klassen `Flow` und `SubmissionPublisher` im Überblick

Java 9 unterstützt Reactive Streams mit der Klasse `Flow`. Das geschieht relativ leichtgewichtig und basiert auf den vier Interfaces `Publisher<T>`, `Subscriber<T>`, `Subscription` und `Processor<T,R>`, die innerhalb von `Flow` definiert sind. `Publisher<T>` und `Subscriber<T>` implementiert man gewöhnlich selbst und eine `Subscription` hilft beim Austausch von Daten. Schließlich kombiniert ein `Processor<T,R>` sowohl `Publisher<T>` als auch `Subscriber<T>` in einem Interface.

Das Interface `Publisher<T>` Ein `Publisher<T>` ist ein Veröffentlicher (mitunter auch Erzeuger) von Dingen, die von einem oder mehreren `Subscriber<T>`-Objekten verarbeitet werden. Dazu können sich diese bei dem `Publisher<T>` registrieren. Das zugehörige Interface ist minimalistisch wie folgt definiert:

```
@FunctionalInterface
public static interface Publisher<T>
{
    public void subscribe(Subscriber<? super T> subscriber);
}
```

Das Interface `Subscriber<T>` Ein `Subscriber<T>` ist ein Empfänger von Nachrichten. Die Methoden dieses Interface werden je nach Situation aufgerufen. Dabei wird vom `Publisher<T>` zunächst eine `Subscription` an einen `Subscriber<T>` gesendet und in der Folge werden Daten mit `onNext(T)` übertragen. Zudem lassen sich Fehlersituationen oder das Ende einer Datenübertragung durch das folgende Interface `Subscriber<T>` abbilden:

```
public static interface Subscriber<T>
{
    public void onSubscribe(Subscription subscription);
    public void onNext(T item);
    public void onError(Throwable throwable);
    public void onComplete();
}
```

Wie im Listing gezeigt, besitzt der `Subscriber<T>` vier Methoden:

- `onSubscribe(Subscription subscription)` – Dient zur Registrierung und wird vor der eigentlichen Kommunikation aufgerufen.
- `onNext(T item)` – Wird aufgerufen, wenn ein neues Item verfügbar ist.
- `onError(Throwable throwable)` – Für den Fall, dass ein Fehler auftritt, wird diese Methode durch den `Publisher<T>` aufgerufen, um dies dem jeweiligen `Subscriber<T>` mitzuteilen.
- `onComplete()` – Falls die Datenübertragung beendet werden soll, kann ein Aufruf dieser Methode durch den `Publisher<T>` erfolgen.

Das Interface `Subscription` Eine `Subscription` dient zur Verknüpfung zwischen `Publisher<T>` und `Subscriber<T>` und ist wie folgt definiert:

```
public static interface Subscription
{
    public void request(long n);
    public void cancel();
}
```

Subscriber erhalten allerdings nur dann Items, wenn diese explizit angefordert wurden. Dazu dient die `request(long)`-Methode. Deren Parameter beschreibt, wie viele Daten vom `Producer<T>` angefordert und als Reaktion auf diesen Aufruf maximal geliefert werden dürfen. Weitere Anforderungen können ergänzend per `request(long)` erfolgen. Die Gesamtzahl der per `onNext(T)` gelieferten Items wird kumuliert. Weil der Parameter die maximale Anzahl an Daten, die der `Subscriber<T>` verarbeiten kann oder will, bestimmt, lassen sich der Datenfluss steuern und Überlastsituationen vermeiden. Für eine unlimitierte Datenübertragung muss der Wert `Long.MAX_VALUE` übergeben werden. Schließlich kann die Verarbeitung vom `Subscriber<T>` durch Aufruf von `cancel()` bei Bedarf abgebrochen werden.

Das Interface `Processor<T,R>` Der `Processor<T,R>` kombiniert die beiden Interfaces `Subscriber<T>` sowie `Publisher<R>` und ist folgendermaßen definiert:

```
public static interface Processor<T,R> extends Subscriber<T>, Publisher<R>
{
}
```

Diese Kombination erlaubt es, dass sich damit Verarbeitungsketten nach folgendem Muster zusammenfügen lassen:

***Abbildung 3-2** Zusammenarbeit zwischen Publisher, Processor und Subscriber*

Die Klassen `SubmissionPublisher` und `BufferedSubscription` Die Klasse `java.util.concurrent.SubmissionPublisher` implementiert das Interface `Publisher<T>` und dient dazu, `Subscriber<T>` zu verwalten, insbesondere, um asynchron Daten an registrierte `Subscriber<T>` publizieren zu können. Die private statische innere Klasse `BufferedSubscription` implementiert das Interface `Subscription` und ermöglicht es, `Subscriptions` zu verarbeiten. Zudem können registrierte `Subscriber<T>` mithilfe einer `Subscription` neue Daten anfordern oder die Verarbeitung abbrechen.

Die Klassen und Interfaces im Überblick Abbildung 3-3 zeigt, wie die zuvor beschriebenen Klassen und Interfaces miteinander in Verbindung stehen. Dieses Klassendiagramm sollte das Verständnis für die Zusammenhänge und die im Anschluss vorgestellten Abläufe bei der Kommunikation erleichtern.

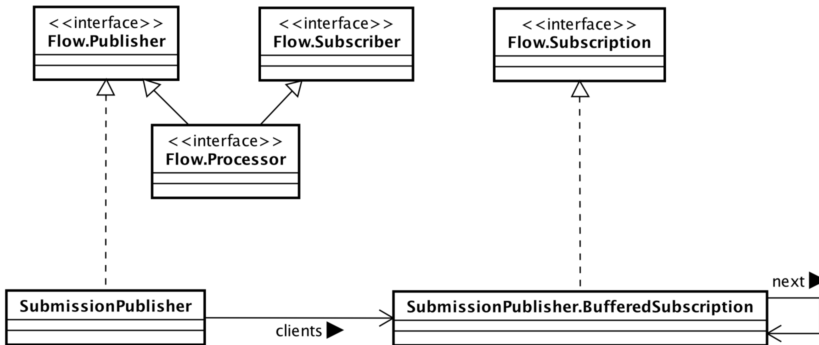

***Abbildung 3-3** Die `Flow`-Interfaces und die Klasse `SubmissionPublisher`*

Verarbeitungen mit der Klasse `Flow`

Die vorgestellten Interfaces ermöglichen eine flexible Kommunikation. Eine Variante davon ist in Abbildung 3-4 dargestellt und zeigt folgende Schritte:

1. Initial wird ein `Subscriber<T>` erzeugt und bei einem `Publisher<T>` registriert.
2. Zur Komplettierung ruft der `Publisher<T>` die Methode `onSubscribe()` des Interface `Subscriber<T>` auf und übergibt dabei ein `Subscription`-Objekt.
3. Die Kommunikation beginnt, wenn ein `Subscriber<T>` Daten anfordert, indem er die Methode `request(long)` aufruft.
4. Die Datenübergabe erfolgt durch Aufruf von `onNext(T)`. Diese Methode von `Subscriber<T>` darf jedoch vom `Publisher<T>` nur maximal so oft aufgerufen werden, wie zuvor in `request(long)` als Wert übergeben wurde – bei mehrfachen Aufrufen von `request(long)` wird die Anzahl aufsummiert.

5. Schließlich kann ein `Subscriber<T>` die Kommunikation durch Aufruf von `cancel()` beenden. Alternativ kann dies durch den `Publisher<T>` geschehen, indem für jeden registrierten `Subscriber<T>` dessen Methode `onComplete()` aufgerufen wird. Nutzt man die Klasse `SubmissionPublisher` lässt sich dies einfacher durch einen Aufruf von `close()` erledigen.

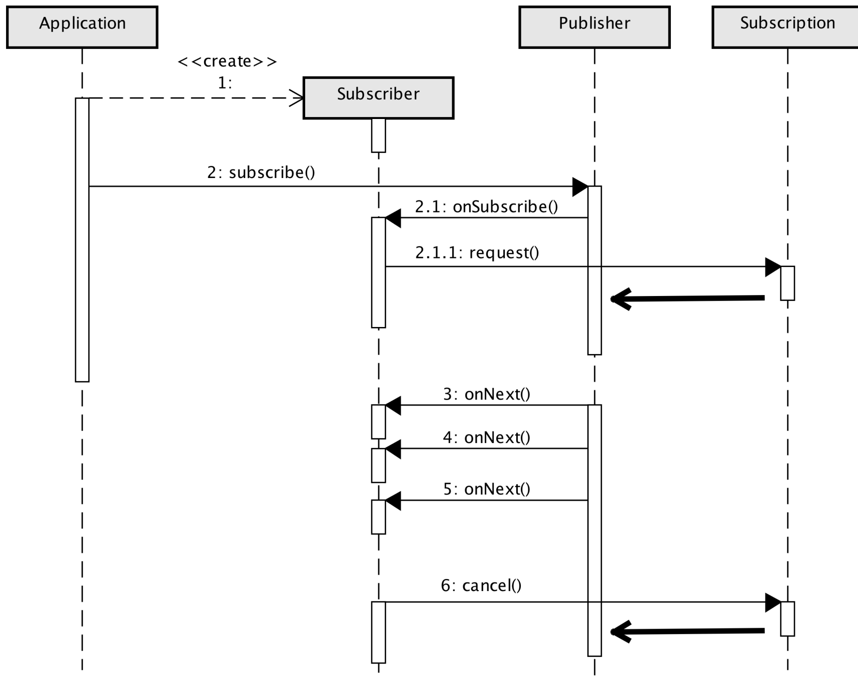

Abbildung 3-4 *Darstellung der Abläufe mit der Klasse* `Flow`

Es sei darauf hingewiesen, dass ein `Subscriber<T>` jederzeit erneut die Methode `request(long)` aufrufen kann, um weitere Daten anzufordern. Das kann selbst dann geschehen, wenn der `Publisher<T>` noch gar nicht alle aus dem letzten Aufruf von `request(long)` gewünschten Daten geliefert hat. Somit ist der `Subscriber<T>` nicht daran gebunden, zu warten, bis er alle Datensätze aus dem letzten Request erhalten oder verarbeitet hat. Dies ist ein Mittel, um dafür zu sorgen, dass es für ihn immer genügend Arbeit gibt. Wichtig dabei ist, dass sich die Anzahl der angeforderten Datensätze addiert. Werden beim ersten Mal zehn und danach fünf Datensätze angefordert, so liefert der `Publisher<T>` insgesamt 15 Datensätze, sofern er nicht selbst per `onComplete()` signalisiert, dass er keine Daten mehr an den `Subscriber<T>` liefern kann. Ein anderer Grund, weniger Daten zu liefern als angefordert, ist das Auftreten eines Fehlers, der über `onError(Throwable)` kommuniziert wird. Eine weitere Zusammenarbeit ist dann nicht mehr sinnvoll. Schließlich ist es dem `Subscriber<T>`

möglich, beim `Publisher<T>` indirekt über `subscription.cancel()` das Ende der Kommunikation anzufordern.

Im obigen Diagramm habe ich bewusst zwei dickere Pfeile von der `Subscription` auf den `Publisher<T>` eingezeichnet. Tatsächlich ist anhand der Interfaces nicht nachvollziehbar, wie ein `Publisher<T>` darüber informiert wird, dass ein `Subscriber<T>` bei der `Subscription` eine der Methoden `request(long)` oder `cancel()` aufgerufen hat. Dies geschieht nur indirekt und muss durch die jeweilige Implementierung sichergestellt werden. Dies wird durch Einsatz der Klasse `SubmissionPublisher` aus dem JDK erleichtert.

Beispiel zur Klasse `Flow`

Nachdem wir grob die Abläufe kennengelernt haben, wollen wir Reactive Streams zur parallelisierten Ausführung des Zählens von Wörtern in mehreren Dateien einsetzen. Diese Funktionalität habe ich bereits zur Beschreibung einiger Neuerungen in JDK 8 für verschiedene Klassen in meinem Buch »Java 8 – Die Neuerungen« [2] genutzt. Für die Klasse `Flow` haben es Hettel und Tran in ihrem Buch »Nebenläufige Programmierung mit Java« [1] als Beispiel gewählt. Das greife ich hier auf, verändere aber die Implementierung an diversen Stellen.

Die Applikation Für ein besseres Verständnis des Zusammenspiels der einzelnen Klassen beginnen wir mit dem Hauptprogramm. Hier sehen wir sowohl das Erzeugen und den Einsatz von `Publisher<T>` als auch von `Subscriber<T>`. Der `WordPublisher` durchsucht eine übergebene Liste von Dateien nach einem speziellen Wort, hier »`private`«. Die Suche wird durch Aufruf von `performSearch()` gestartet. Bei Auffinden eines passenden Worts wird der zuvor mit `subscribe()` registrierte `WordSubscriber` informiert:

```java
public static void main(final String[] args) throws Exception
{
    final WordPublisher finder = new WordPublisher("private", getInputFiles());
    finder.subscribe(new WordSubscriber());
    finder.performSearch();

    Thread.sleep(2_000);  // auf das Ende der Verarbeitung warten
    finder.terminate();
}

private static List<Path> getInputFiles()
{
    return List.of(
      Paths.get("src/main/java/ch3_1_3/reactivestreams/WordPublisher.java"),
      Paths.get("src/main/java/ch3_1_3/reactivestreams/WordFinderClient.java"));
}
```

Listing 3.6 *Ausführbar als* **'WORDFINDERCLIENT'**

Der Publisher Der `Publisher<T>` wird so implementiert, dass er in einer Menge von Dateien nach einem übergebenen Wort suchen kann. Diese Suche erfolgt parallel mithilfe eines Thread-Pools. Das wichtigste Detail ist aber die im JDK definierte Klasse `SubmissionPublisher`. Mit deren Hilfe kann man Implementierungen des Interface `Publisher<T>` einfach umsetzen: Zum einen bietet diese Klasse die Möglichkeit zur Verwaltung von mehreren `Subscriber<T>`-Objekten und zum anderen kann man damit registrierte `Subscriber<T>` benachrichtigen:

```
public class WordPublisher implements Flow.Publisher<String>
{
    private final String word;
    private final List<Path> paths;
    private final SubmissionPublisher<String> publisher;

    private final ExecutorService executor = Executors.newFixedThreadPool(4);

    public WordPublisher(final String word, final List<Path> paths)
    {
        this.word = word;
        this.paths = Collections.unmodifiableList(paths);
        this.publisher = new SubmissionPublisher<>();
    }

    @Override
    public void subscribe(final Subscriber<? super String> subscriber)
    {
        publisher.subscribe(subscriber);
    }

    public void performSearch() throws InterruptedException
    {
        for (final Path path : paths)
        {
            executor.execute(() ->
            {
               final Stream<String> occurences = findWord(word, path);
               occurences.forEach(line -> publisher.submit("file: " + path +
                                                           " : " + line));
            });
        }
    }

    public void terminate() throws InterruptedException
    {
        // führt zum Aufruf von onComplete()
        publisher.close();

        executor.shutdown();
    }

    private Stream<String> findWord(final String wordToSearch,
                                    final Path path) throws IOException
    {
        try
        {
            final Charset utf8 = StandardCharsets.UTF_8;
            final List<String> lines = Files.readAllLines(path, utf8);

            return lines.stream().filter(line -> line.contains(wordToSearch));
        }
```

```
        catch (final IOException e)
        {
            return Stream.of();
        }
    }
}
```

Der Subscriber Der im Listing realisierte `Subscriber<T>` protokolliert alle Methodenaufrufe auf der Konsole. Für diese einfache Funktionalität benötigt man keinen Zugriff auf die `Subscription`. Normalerweise würde man diese in einem Attribut speichern, damit man beliebig Daten anfordern oder die Verarbeitung abbrechen kann. Das lassen wir hier zunächst noch aus und heben uns dies für eine Erweiterung auf:

```
public class WordSubscriber implements Subscriber<String>
{
    @Override
    public void onSubscribe(final Subscription subscription)
    {
        System.out.println(LocalDateTime.now() + " onSubscribe()");

        subscription.request(Long.MAX_VALUE);
    }

    @Override
    public void onNext(final String item)
    {
        System.out.println(LocalDateTime.now() + " onNext(): " + item);
    }

    @Override
    public void onComplete()
    {
        System.out.println(LocalDateTime.now() + " onComplete()");
    }

    @Override
    public void onError(final Throwable throwable)
    {
        throwable.printStackTrace();
    }
}
```

Programmausführung Beim Start des Programms WORDFINDERCLIENT kommt es in etwa zu folgenden Ausgaben, die die Ergebnisse der Suche protokollieren:

```
2016-12-04T12:11:49.872009 onSubscribe()
2016-12-04T12:11:49.878411 onNext(): file: src/main/java/ch3_1_3/reactivestreams
      /WordPublisher.java : private final String word;
2016-12-04T12:11:49.878503 onNext(): file: src/main/java/ch3_1_3/reactivestreams
      /WordFinderClient.java :     final WordPublisher finder = new WordPublisher
      ("private", getInputFiles());
2016-12-04T12:11:49.878572 onNext(): file: src/main/java/ch3_1_3/reactivestreams
      /WordPublisher.java :     private final List<Path> paths;
2016-12-04T12:11:49.878637 onNext(): file: src/main/java/ch3_1_3/reactivestreams
      /WordPublisher.java : private final SubmissionPublisher<String> publisher;
```

```
2016-12-04T12:11:49.878708 onNext(): file: src/main/java/ch3_1_3/reactivestreams
      /WordPublisher.java :     private final ExecutorService executor =
      Executors.newFixedThreadPool(4);
2016-12-04T12:11:49.878782 onNext(): file: src/main/java/ch3_1_3/reactivestreams
      /WordFinderClient.java : private static List<Path> getInputFiles()
2016-12-04T12:11:49.878848 onNext(): file: src/main/java/ch3_1_3/reactivestreams
      /WordPublisher.java : private Stream<String> findWord(final String
      wordToSearch, final Path path) throws IOException
2016-12-04T12:11:51.859276 onComplete
```

Man erkennt die Start- und Registrierungsphase, auf die die Verarbeitung mit `onNext(T)` folgt. Zum Abschluss wird `onComplete()` aufgerufen.

Ein weiterer Subscriber Nun schauen wir uns eine Variante von `Subscriber<T>` an, die nur die ersten fünf Items konsumiert. In Abwandlung des sehr einfachen ersten `Subscriber<T>`-Interface wird hier die `Subscription` als Attribut definiert, und es lässt sich ein `Consumer<String>` mitgeben, der die Aktionen auf den Items beschreibt:

```
public class First5Subscriber implements Subscriber<String>
{
    private final Consumer<String> consumer;
    private Subscription subscription;
    private int count = 1;

    First5Subscriber(final Consumer<String> consumer)
    {
        this.consumer = consumer;
    }

    @Override
    public void onSubscribe(final Subscription subscription)
    {
        System.out.println("F5 Subscription: " + subscription);

        this.subscription = subscription;
        this.subscription.request(5);
    }

    @Override
    public void onNext(final String item)
    {
        System.err.println("F5 " + count + "x onNext(): " + item);

        consumer.accept(item);
        count++;
        if (count >= 5)
        {
            subscription.cancel();
        }
    }

    @Override
    public void onComplete()
    {
        System.out.println("onComplete()");
    }
```

```
    @Override
    public void onError(final Throwable throwable)
    {
        throwable.printStackTrace();
    }
}
```

Anhand dieses Beispiels lernt man zwei Dinge: Zum einen, wie die Verarbeitung als `Consumer<String>` an den `Subscriber<String>` übergeben werden kann, und zum anderen, wie mithilfe von `onNext(String)` eine etwas komplexere und flexiblere Verarbeitung realisiert werden kann: Hier wird das Zählen der Einträge sowie der Abbruch durch den `Subscriber<String>` mit einem Aufruf von `subscription.cancel()` bei fünf erhaltenen Einträgen demonstriert.

Dieser `Subscriber<String>` lässt sich problemlos parallel zu dem anderen registrieren. Dazu wandelt man die bereits gezeigte `main()`-Methode wie folgt leicht ab:

```
public static void main(final String[] args) throws IOException,
                                                     InterruptedException
{
    final WordPublisher finder = new WordPublisher("private", getInputFiles());
    finder.subscribe(new WordSubscriber());
    finder.subscribe(new First5Subscriber(System.err::println));
    finder.performSearch();

    TimeUnit.SECONDS.sleep(2); // auf das Ende der Verarbeitung warten

    finder.terminate();
}
```

Listing 3.7 *Ausführbar als* **'WORDFINDERTWOSUBSCRIBERSEXAMPLE'**

Fazit

Dieses Unterkapitel hat einen Einstieg in Reactive Streams gegeben und die grundlegenden Konzepte kurz vorgestellt. Sie sollten mit diesem Wissen in der Lage sein, erste eigene Experimente zu starten. Richtig spannend wird das Ganze, wenn man größere Verarbeitungsketten mithilfe mehrerer `Publisher<T>`, `Processor<T,R>` und `Subscriber<T>` beschreibt. Das würde jedoch den Rahmen dieses Buchs sprengen. Leider bietet die Implementierung der Reactive Streams im JDK momentan noch kein API zur Verkettung von Operationen. Es bleibt für die Zukunft zu hoffen und zu erwarten, dass hier noch Erweiterungen folgen.