
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 Streams⁵ 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 größtenteils beschränkt 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/>

⁶Details dazu finden Sie unter <http://www.reactivemanifesto.org/>.

langsamem 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 `SubmissionPublisher.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 vor gestellten Abläufe bei der Kommunikation erleichtern.

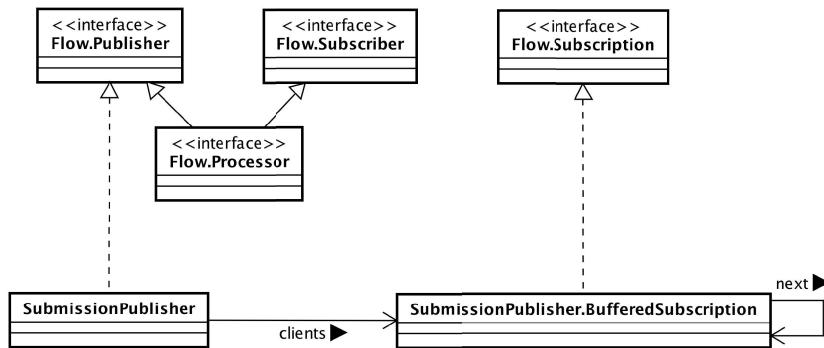


Abbildung 3-3 Die `Flow`-Interfaces und die Klasse `SubmissionPublisher`

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 über gibt 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.

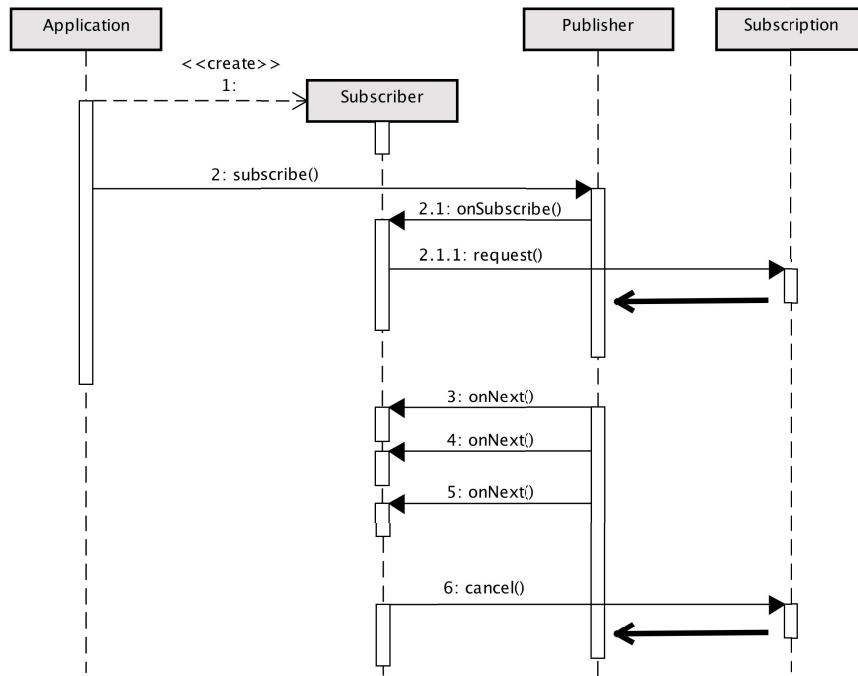


Abbildung 3-4 Darstellung der Abläufe mit der Klasse Flow

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 dicke 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:

```
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> occurrences = findWord(word, path);
                occurrences.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(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.out::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.