Wir bauen uns eine Monade – Railway Oriented Programming statt Exception-Handling

Stefan Macke, ALTE OLDENBURGER Krankenversicherung AG

Dieser Artikel zeigt an praktischen Beispielen den inzwischen alltäglichen Einsatz von Monaden in Java 8. Dieses Konzept aus der funktionalen Programmierung erleichtert den Umgang mit teilweise komplexen Problemen wie der asynchronen Programmierung. Zusätzlich wird eine eigene Monade entwickelt, die eine Alternative zur Verwendung von Exceptions zur Fehlerbehandlung bietet.

Monaden sind vielen objektorientierten Softwareentwicklern nur als kryptisches Konstrukt der funktionalen Programmierung bekannt. Dabei arbeiten auch Java-Entwickler inzwischen fast täglich damit, wenn sie die neuen Funktionen in Java 8 einsetzen. So zählen beispielsweise „Optional" und „Stream" zur Gruppe der monadischen Datenstrukturen. Sie sind also alles andere als exotisch, sondern helfen uns bei der täglichen Arbeit.

Darüber hinaus ist es auch gar nicht so schwierig, sich selbst eigene Monaden zu programmieren. Dies werden wir in diesem Artikel tun. Analog zum „Optional", das die Behandlung von „null"-Werten verhindern soll, wird eine Monade programmiert, die die Fehlerbehandlung mit Exceptions ersetzen kann. Der resultierende Code wird ein wenig funktionaler, einfacher zu testen und vielleicht sogar etwas verständlicher. Die Idee hinter diesem Ansatz heißt „Railway Oriented Programming". Doch zunächst eine kurze Einführung in Monaden.

Monaden im Java-Alltag

Wer schon einmal Code wie den in *Listing 1* geschrieben hat, verwendet bereits zwei Monaden: „Optional" und „Stream". Im Beispiel werden Benutzer anhand ihrer Benutzernamen aus einer Datenbank gelesen. Dabei wird geprüft, ob die jeweiligen Benutzer über-

haupt vorhanden sind. Zuletzt werden dann die Namen der vorhandenen Benutzer auf der Konsole ausgegeben. Interessant an diesem Code ist, dass weder eine explizite Iteration zu sehen ist, noch „null" als Repräsentation eines nicht vorhandenen Benutzers verwendet wird. Stattdessen wird das Durchlaufen der Liste dem „Stream" überlassen und „Optional" als Rückgabewert des Repository macht deutlich, dass es gegebenenfalls zum angegebenen Benutzernamen keinen Benutzer gibt.

Die beiden Datenstrukturen repräsentieren zwei allgemeine Konzepte: das Vorhandensein mehrerer Werte, über die iteriert werden kann, und das Nicht-Vorhandensein eines Werts. Zusätzlich stellen sie Funktionen bereit, die den Entwicklern die Arbeit mit beiden Konzepten einfacher machen: Es ist keine explizite Iteration mehr nötig und „null"-Checks sind überflüssig.

Java 8 bringt gleich mehrere solcher Datenstrukturen mit. Sie umschließen jeweils einen zugrunde liegenden Datentyp (im Beispiel „User") mit einem operationalen Kontext, dessen teilweise recht hohe Komplexität dadurch vor dem Aufrufer verborgen bleibt. Stattdessen kann die einheitliche Schnittstelle dieser Monaden – zum Beispiel „map()"und „filter()" – genutzt werden, um unterschiedliche Konzepte im Code sehr ähnlich zu verwenden. Folgende Liste zeigt drei bekannte Monaden in Java 8 und beschreibt, welches allgemeine Konzept sie repräsentieren;

- **Optional**
  Repräsentiert das Nicht-/Vorhandensein eines Werts und vermeidet „null"-Checks
- **Stream**
  Repräsentiert potenziell mehrere Werte und vermeidet explizite Iterationen
- **CompletableFuture**
  Repräsentiert das Ergebnis einer asynchronen Operation und vermeidet Thread-Programmierung

Monaden im Java-Alltag

Untersucht man die vorgestellten Datenstrukturen etwas genauer, stellt man fest, dass sie sich gewisse Eigenschaften teilen. Alle Monaden haben eine gemeinsame Basis an Funktionen, die sie anbieten müssen:

- Einen Typ-Konstruktor, der die Monade mit einem Datentyp parametrisiert. Im Beispiel sind das die generischen Klassen „Optional<T>" beziehungsweise „Stream<T>".
- Eine Funktion, die aus einem normalen Datentyp eine Monade dieses Datentyps macht, ihn also quasi in der Monade verpackt. Diese Funktion heißt in funktionalen Programmiersprachen wie Haskell häufig „unit()". Die Methode „Optional.of("Mein Text")" macht beispielsweise aus dem „String" mit dem Wert „Mein Text" ein „Optional<String>".
- Eine Funktion, die eine andere Funktion als Parameter bekommt, die aus dem inneren Datentyp der Monade eine neue Monade eines beliebigen Datentyps erzeugt und diese neue Monade zurückgibt. In funktionalen Sprachen heißt diese Funktion beispielsweise „bind()", in Java „flatMap()".

Zum dritten Punkt ein Beispiel: „Optional.of("Mein Text"). flatMap(text –> Optional.of(text.length()))" liefert als Ergebnis ein

```
List<String> usernames = new ArrayList<>();
usernames.add("user1");
usernames.add("user2");
usernames.add("user3");

UserRepository repo = new UserRepository();

usernames.stream()                    // Stream<String>
    .map(repo::findUserByUsername)    //
Stream<Optional<User>>
    .filter(Optional::isPresent)
    .map(Optional::get)               // Stream<User>
    .map(User::getName)               // Stream<String>
    .forEach(System.out::println);
```

Listing 1: Einsatz der Monaden „Optional" und „Stream" im Java-Code

„Optional<Integer>". Die an „flatMap()" übergebene Funktion (der Lambda-Ausdruck) bekommt einen „String" als Parameter und liefert ein „Optional<Integer>" zurück. „flatMap()"nimmt also den im „Optional" enthaltenen „String" mit dem Wert „Mein Text", wendet die übergebene Funktion „Optional.of(text.length())" darauf an, erhält als Ergebnis ein „Optional<Integer>", das die Länge des Strings („9") enthält, und gibt dieses dann zurück.

Das Vorhandensein dieser Funktionen allein reicht jedoch noch nicht aus, um von einer echten Monade sprechen zu können. Die Funktionen müssen zusätzlich folgende Gesetze einhalten:

- Die „unit()"-Funktion muss das linksneutrale Element der „bind()"-Funktion sein.
- Die „unit()"-Funktion muss das rechtsneutrale Element der „bind()"-Funktion sein.
- Die „bind()"-Funktion muss assoziativ sein.

Diese recht mathematischen Definitionen sind für die Theorie hinter den Monaden sicherlich sehr wichtig, für ihren Praxiseinsatz muss man allerdings nicht unbedingt jedes Detail verstehen. Da dieser Artikel keine wissenschaftliche Einführung in die mathematische Kategorien-Theorie ist, in der die Monaden ihren Ursprung haben, sondern ihre Vorteile in der Programmierpraxis zeigen soll, sei daher an dieser Stelle auf eine weitere Vertiefung der Monadengesetze verzichtet. Wer Interesse an den mathematischen Hintergründen hat, dem ist der entsprechende Wikipedia-Artikel *(siehe „https://en.wikipedia.org/wiki/Monad_(functional_programming)")* zum Einstieg empfohlen.

Wichtig ist an dieser Stelle noch der Hinweis, dass die hier vorgestellten Datenstrukturen eventuell keine echten Monaden im funktionalen Sinne sind. Insbesondere „Optional" bricht mindestens eines der drei Gesetze. Eine gute Erklärung liefert Marcello La Rocca in seinem Artikel „How Optional Breaks the Monad Laws and Why It Matters" *(siehe „https://www.sitepoint.com/how-optional-breaks-the-monad-laws-and-why-it-matters/").* Dort sind auch die Monadengesetze noch einmal im Detail erklärt.

Monaden statt Exceptions

Bereits im Jahr 2003 schrieb Joel Spolsky, der Gründer von StackOverflow, in einem Blog-Artikel *(siehe „https://www.joelonsoftware.com/2003/10/13/13/"),* dass er selbst niemals Exceptions verwen-

```
usernames.stream()
    .map(repo::findUserByUsername)  // Unhandled exception type DatabaseException
    .map(User::getName)
    .forEach(System.out::println);
```

Listing 2: Exceptions und Streams vertragen sich nicht

den würde. Sie seien im Quelltext unsichtbar, da man einer Methode nicht ansehen könne, ob sie eine Exception werfe (in Java gilt das natürlich nur für Unchecked Exceptions). Daher sei es für Leser des Codes schwierig zu erkennen, ob ein Fehler auftreten kann oder nicht. Außerdem ähnelten sie dem berüchtigten „goto"-Statement, da nicht offensichtlich sei, zu welcher Stelle im Code nach dem Auftreten einer Exception gesprungen würde.

Als Alternative schlägt Spolsky die Verwendung von Rückgabewerten vor, die auf einen potenziellen Fehler hinweisen. Diesen Vorschlag greifen wir nun auf und entwickeln eine monadische Datenstruktur, die den potenziell fehlerhaften Ausgang einer Funktion repräsentiert. Wir können die obige Liste somit um folgenden Punkt erweitern:

- **Result**
  Repräsentiert das potenziell fehlerhafte Ergebnis einer Operation und vermeidet Exception-Handling

Der Code in *Listing 2* greift das obige Beispiel wieder auf und erweitert es um eine Checked Exception in „UserRepository.findUserByUsername()". Da diese Methode auf eine Datenbank zugreift, die ausfallen kann, wirft sie nun eine „DatabaseException". Das macht ihre direkte Verwendung im Stream leider unmöglich, da „map()" keine Methoden mit Checked Exceptions erlaubt.

Listing 3 zeigt eine mögliche Lösung des Problems: Die Checked Exception wird gefangen und in eine „RuntimeException" verpackt. Der Code wird dadurch allerdings nicht gerade verständlicher. Die Schachtelungstiefe steigt und ein komplexer Lambda-Ausdruck wird programmiert, wo vorher nur eine einzige Zeile Code nötig war.

Eine Alternative zu obigem Vorgehen steht in *Listing 4*. Statt eine Checked Exception zu werfen, gibt „findUserByUsername()" nun ein „Result<User>" zurück. Diese noch recht rudimentäre Implementierung der Idee lässt sich analog zu „Optional" im ersten Beispiel verwenden und verhindert bereits die umständliche Fehlerbehandlung

```
usernames.stream()
    .map(username -> {
        try
        {
            return repo.findUserByUsername(username);
        }
        catch (DatabaseException e)
        {
            throw new RuntimeException(e);
        }
    })
    .map(User::getName)
    .forEach(System.out::println);
```

Listing 3: Eine mögliche, aber unschöne Lösung

aus *Listing 3*. Allerdings ist als Fehlertyp nur ein „String" möglich und auch die Monaden-Funktionen wie „flatMap()" fehlen noch. Außerdem wird keine interne Fehlerbehandlung durchgeführt und „null" als Repräsentation fehlender Werte ist auch nicht die beste Idee.

Railway Oriented Programming

Man kann und sollte die Klasse „Result" noch um die fehlenden Methoden und einen weiteren Typ-Parameter für den Fehlerfall ergänzen. Da dies jedoch den Rahmen dieses Artikels sprengen würde, sei auf das Projekt „ao-railway" bei GitHub *(siehe „https://github.com/StefanMacke/ao-railway")* verwiesen. Dort stellt der Autor eine komplette Implementierung des Konzepts bereit, die zusätzliche Methoden enthält, die den Umgang mit Fehlern weiter vereinfachen. *Listing 5* zeigt als Beispiel das Ändern eines Benutzer-Passworts. In wenigen Zeilen sprechenden Codes (dank des Einsatzes eines Fluent-API *(siehe „http://martinfowler.com/bliki/FluentInterface.html")* wird die komplette Geschäftslogik inklusive Fehlerbehandlung implementiert.

Der Name des Projekts „ao-railway" ist eine Referenz auf die Idee, den Ablauf eines Programms analog zu Bahnschienen in einen Erfolgs- („onSuccess()") und einen Fehlerweg („onFailure()") einzu-

```
usernames.stream()                  // Stream<String>
  .map(repo::findUserByUsername)    //
Stream<Result<User>>
  .filter(Result::isSuccess)
  .map(Result::getValue)            // Stream<User>
  .map(User::getName)               // Stream<String>
  .forEach(System.out::println);

public class Result<T> {
  private T value;
  private String error;

  private Result(T value, String error) {
     this.value = value;
     this.error = error;
  }

  public static <T> Result<T> withValue(T value) {
     return new Result<T>(value, null);
  }

  public static <T> Result<T> withError(String error)
{
     return new Result<T>(null, error);
  }

  public boolean isSuccess() {
     return error == null;
  }

  public T getValue() {
     return value;
  }
}
```

Listing 4: „Result" als Rückgabewert statt Exception

```
public Result<User> changePassword(
    Username username, Password oldPasswort, Password newPassword) {
    return Result.combine(
        Result.of(username, "Username cannot be empty"),
        Result.of(oldPassword, "Old password cannot be empty"),
        Result.of(newPassword, "New password cannot be empty"))
    .onSuccess(() -> userRepo.find(username))
    .ensure(user -> user.isCorrectPassword(oldPassword), "Invalid password")
    .onSuccess(user -> user.changePassword(newPassword))
    .onSuccess(user -> userRepo.update(user))
    .onFailure(() -> logger.error("Password could not be changed"));
}
```

Listing 5: „ao-railway" im Praxiseinsatz

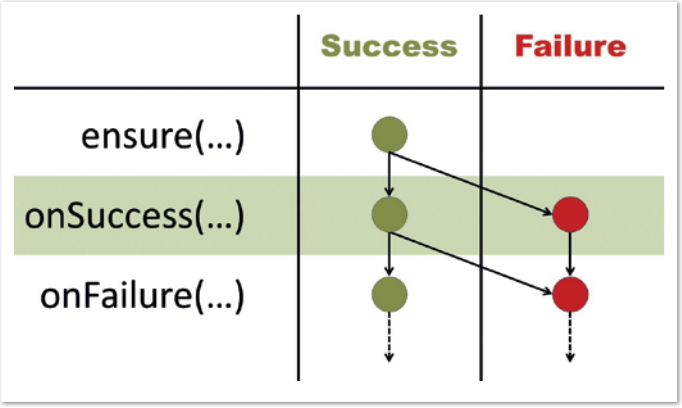

Abbildung 1: Programmfluss beim Railway Oriented Programming

teilen. Sie stammt von Scott Wlaschin, der sie bei der funktionalen Softwareentwicklung mit F# einsetzt. In einem Video zeigt er ausführlich ihren Praxiseinsatz *(siehe „https://vimeo.com/97344498")*.

Sobald an irgendeiner Stelle der Aufrufkette vom erwarteten Verhalten abgewichen wird (etwa weil das alte Passwort nicht gültig oder ein Datenbank-Fehler aufgetreten ist), soll der restliche Code nicht mehr durchlaufen werden. Der Fehlerfall führt das Programm also quasi wie bei einer Weiche auf eine andere Bahnschiene, die parallel zum normalen Programmfluss verläuft, aber eben komplett daran vorbeiführt und bis zum Ende der Befehlskette nicht wieder verlassen werden kann *(siehe Abbildung 1)*.

Die Klasse „Result" übernimmt dabei die gesamte Steuerung des Programmflusses, indem sie je nach internem Zustand (Erfolg oder Fehler) die ihr übergebenen Operationen ausführt oder nicht. Damit muss sich der Aufrufer nicht mehr um die Fehlerbehandlung kümmern, sondern nur noch entscheiden, welche Aktionen in den beiden Zuständen ausgeführt werden sollen.

Fazit

Dieser Artikel zeigt, dass Monaden nicht nur abstrakte Gebilde aus der funktionalen Programmierung sind, sondern auch bei der alltäglichen Arbeit in Java eingesetzt werden können und dabei auch konkrete Vorteile für den Entwickler mitbringen. Sie abstrahieren von teilweise sehr komplexen Problemen wie der asynchronen Programmierung mit Threads und machen den Code oftmals kompakter und verständlicher für den Leser.

Java 8 bringt bereits eine Menge monadischer Datenstrukturen mit, auch wenn diese gegebenenfalls nicht alle Monadengesetze einhalten. Dennoch erleichtern sie uns an vielen Stellen die Programmierung und sind aus modernem Java-Code kaum mehr wegzudenken. Auch das Erstellen einer eigenen Monade ist kein Hexenwerk. Das Beispiel der Klasse „Result" zeigt, wie einfach der Einstieg in die eher funktionale Problemlösung sein kann. Die komplette Klasse *(siehe „https://github.com/StefanMacke/ao-railway/blob/master/src/main/java/net/aokv/railway/result/Result.java")* inklusive zusätzlicher Methoden für das Fluent-API im Sinne des Railway Oriented Programming ist lediglich etwa hundert Zeilen lang (ohne Kommentare, Umbrüche etc.) und für erfahrene Java-Entwickler gut zu verstehen.

Viele objektorientierte Programmiersprachen lassen funktionale Konzepte in ihre neuen Versionen einfließen. Zuletzt wurden in Java 8 Lambda-Ausdrücke und Streams hinzugefügt sowie Funktionen zu Bürgern erster Klasse erhoben. Das sollte für alle Java-Entwickler ein Anreiz sein, sich intensiver mit den Ideen der funktionalen Programmierung auseinanderzusetzen. Dazu ist kein abgeschlossenes Mathematikstudium nötig, sondern nur das Interesse für neue Wege der Problemlösung. Monaden bilden dabei eine wichtige Grundlage. Vielleicht regt dieser Artikel dazu an, die altbewährten Exceptions abzulösen oder ihren Einsatz zumindest zu überdenken. Schließlich verwenden wir ja auch schon lange keine „goto"-Statements mehr.

Stefan Macke
mail@anwendungsentwicklerpodcast.de

Stefan Macke ist Software-Entwickler und -Architekt bei der ALTE OLDENBURGER Krankenversicherung AG aus Vechta. Seit dem Jahr 2007 ist er dort außerdem Ausbilder für Anwendungsentwickler und seit dem Jahr 2009 IHK-Prüfer in Oldenburg. Seine Erfahrungen in der Ausbildung kommuniziert er in seinem Podcast unter „http://anwendungsentwicklerpodcast.de". In seinen aktuellen Projekten beschäftigt er sich vor allem mit der Modernisierung von Alt-Anwendungen auf Basis einer serviceorientierten Architektur mithilfe von Java.