# Vavr.io – Objekt-funktionale Programmierung leicht gemacht

*David Schmitz, Senacor Techologies*

*Funktionale Programmierung hat in den letzten Jahren eine Art zweite Blüte erlebt und ist nun auch im Mainstream angekommen. In Folge hat Java 8 einige Konstrukte der funktionalen Programmierung in Java eingeführt, die im Tagesgeschäft jedoch Lücken offenbaren. Das Open-Source-Projekt Vavr (siehe „http://www.vavr.io“) will genau diese Lücken schließen. Sinnvolle, praktische Ideen und Konzepte aus Sprachen wie Clojure und Scala können mit Vavr nun auch in Java auf einfache Art genutzt werden.*

Wir wollen hier keinen theoretischen Exkurs in die Vor- und Nachteile funktionaler Programmierung machen. Dennoch sei der Vorteil der funktionalen Programmierung anhand eines kleinen Beispiels verdeutlicht. Funktional geschriebene Programme vermeiden unnötige Seiteneffekte. Dazu zählen auch die im Java-Umfeld standardmäßig genutzten Collections, die eine beliebte Fehlerquelle sind. Der Code konzentriert sich auf das, was man eigentlich erreichen will, statt in den Details des „Wie“ zu ertrinken. Deshalb spricht man hier von deklarativer Programmierung. Ein einfaches Beispiel ist die Verwendung von „for“-Schleifen im Vergleich zu Operationen wie „map“ *(siehe Listing 1)*.

```
String[] names = new String[]{"Foo", "Bar", "Baz"};

// imperativ
String[] upper = new String[names.length];
for (int i = 0; i < names.length; i++) {
  upper[i] = names[i].toUpperCase();
}

// funktional mit Vavr
String[] upper2 = Array.of(names)
                       .map(String::toUpperCase)
                       .toJavaArray(String.class);
```

*Listing 1*

```
listOfUsers
  .stream()
  .filter(user -> {....})
  .collect(Collectors.toList());
```

*Listing 2*

```
listOfUsers
  .stream()
  .filter(user -> {
     try {
        return user.validate(); // throws Exception
     } catch (Exception ex) {
        return false;
     }})
  .collect(Collectors.toList());
```

*Listing 3*

Im imperativen Fall ist die eigentliche Logik „Umwandlung in Großbuchstaben“ in einem Wust von Boilerplate-Code verborgen. Der funktional geschriebene Code liest sich wie Prosa: Wir nehmen ein Array und wandeln jedes Element in Großbuchstaben um. Funktionale Programmierung hilft uns dabei, die Komplexität unserer Programme zu reduzieren.

## Das Problem mit Java 8

Java 8 hat viele Sprach-Erweiterungen eingeführt, die aus anderen Sprachen lange bekannt waren und auf die viele Entwickler gewartet haben, wie die Erweiterung der Collections um Operationen wie „filter“ und „map“ oder die Einführung von Lambdas. Leider hat sich dabei auch schnell Ernüchterung breit gemacht. Die Verbindung von „Streams“ und „Collectors“, um überhaupt „filter“ nutzen zu können, führt beispielsweise oft zur „Pipeline-Hölle“ *(siehe Listing 2)*.

Auch die Verwendung von Lambdas mit Checked Exceptions ist nicht ganz so einfach, wie initial gehofft, und zwingt den Programmierer zu umständlichen Konstrukten wie im Beispiel von *Listing 3*.

Da Lambdas in Streams keine Checked Exceptions werfen dürfen, müssen diese umständlich gefangen und gekapselt werden. All dies hat zur Entwicklung einer Vielzahl von Bibliotheken geführt, die unter anderem diese Lücken schließen wollen. Wie auch vor knapp vier Jahren Javaslang, das nun als Vavr fortgesetzt wird. Im Kern konzentriert sich Vavr auf:

- Funktionale Datenstrukturen, also persistente, unveränderliche Datenstrukturen wie „List“, „Set“ oder „Seq“
- Abstraktionen für Werteklassen, beispielsweise Konstrukte wie „Option“ und „Tuple“
- Funktionalen Zucker, also „Lift“, „Curry“ und andere Konstrukte, die aus funktionalen Sprachen bekannt sind
- Funktionale Ausnahmebehandlung
- Strukturelle Dekomposition wie Pattern Matching auf Objekt-Instanzen

Wir können nicht auf alle Aspekte eingehen, sondern werden nur ein paar Highlights beleuchten, die im Tagesgeschäft direkten Nutzen bringen.

## Funktionales Java, aber bitte ohne großen Ballast

Um die grundlegenden Ideen von Vavr zu verstehen, nimmt man am besten die Tastatur in die Hand und fängt an zu programmieren. Unter *„https://github.com/koenighotze/vavr-kata-demo“* findet sich ein Demo-Projekt, das wir für unsere Diskussion nutzen. Ich lade dazu ein, das Repository zu klonen und die Anpassungen schrittweise selber auszuprobieren. Die Demo ist eine Spring-Boot-Anwendung *(siehe „https://projects.spring.io/spring-boot“)*, die Sport-Teams verwalten soll. Wie bei Spring Boot üblich, startet man die Anwendung mit „gradle bootRun“.

Die Anwendung exponiert die Ressource „Team“ über einen entsprechenden Endpunkt *„http://localhost:8080/teams“*. Dieser unterstützt lesende Zugriffe über „GET“ auf *„http://localhost:8080/teams“* für alle Teams und „GET“ auf *„http://localhost:8080/teams/{id}“* für das Team mit der Id „{id}“. Über den Pfad *„http://localhost:8080/teams/{id}/logo“* kann zu dem Team mit der Id „{id}“ das zugehörige Wappen angefragt werden.

Wir nutzen die Anwendungen, um punktuell den Nutzen von Vavr zu verdeutlichen. Dabei legen wir bewusst den Blick auf einfache Vavr-Funktionen, die es erlauben, Schritt für Schritt Code zu verbessern und die Vorteile funktionaler Programmierung nutzen zu können. Die jeweiligen Anpassungen können auf den Branches „step1-*“ bis „step7-*“ nachvollzogen werden.

## Optionale Ergebnisse

Als erstes Beispiel schauen wir uns die typische Nutzung von Null-Werten an. Null ist der größte Fehler aller Zeiten, wenn man der allgemeinen Meinung Glauben schenkt. Leider ist null häufig noch immer Teil der täglichen Arbeit. Dazu ein Beispiel aus der Demo-Anwendung: Die Methode „TeamInMemoryRepository#findById“ liefert „null“, wenn das Team nicht gefunden wurde. Der Controller muss entsprechend reagieren *(siehe Listing 4)*.

Funktionale Programmiersprachen umgehen diese „null-pointer“-Semantik durch den „Maybe“- beziehungsweise „Option“-Typ. Dieser repräsentiert entweder einen Wert, falls verfügbar, oder eben die Abwesenheit eines Wertes.

Auch Vavr hat seine Variante; wir nutzen hier „Option<T>“. Sie repräsentiert die Anwesenheit eines Wertes durch die Unterklasse „Some<T>“ und die Abwesenheit durch „None<T>“. Wir verbessern nun die Semantik und ändern das Repository, sodass es ein „Option<Team>“ zurückliefert *(siehe Listing 5)*.

Um nun das Ergebnis zu nutzen, verwenden wir "map", das Schweizer-Armee-Messer der funktionalen Programmierung *(siehe Listing 6)*.

Das Ergebnis liest sich wie ein einfacher Text: „Suche ein Team mit der id „id“. Falls es gefunden wird, liefere das Team und den Status „ok“, ansonsten liefere „not found“ zurück“.

## Persistente Collections

Javas Collections sind ein häufiger Quell für Fehler *(siehe Listing 7)*. Die Collection mit „Collections.unmodifiableList“ unveränderbar zu machen, ist leider nicht hilfreich. Für den Nutzer der unveränderbaren Liste ist das nicht sichtbar und erst zur Laufzeit offenbart sich das Problem mit einer hässlichen Ausnahme *(siehe Listing 8)*.

```
Team team = teamRepository.findById(id);
if (null == team) {
  return notFound().build();
}
return ResponseEntity.ok(team);
```

*Listing 4*

```
Option<Team> team = teamRepository.findById(id);
```

*Listing 5*

```
teamRepository
  .findById(id)
  .map(ResponseEntity::ok)
  .getOrElse(() -> notFound().build());
```

*Listing 6*

```
Set<String> teams = new HashSet<>(asList("F95", "FCK"));
Set<String> moreTeams = teams;
moreTeams.add("STP");

// Ooops...Test schlägt fehlt
assertThat(teams).hasSize(2);
```

*Listing 7*

```
Set<String> teams = new HashSet<>(asList("F95", "FCK"));
Set<String> unmodTeams = unmodifiableSet(teams);

// Ooops...Exception!
unmodTeams.add("Boom");
```

*Listing 8*

```
TreeSet<String> moreteams = teams.add("STP");

assertThat(teams).hasSize(3);
assertThat(moreteams).hasSize(4);
```

*Listing 9*

```
String readTeamLogo (Team t) throws InterruptedExcep-
tion,… {
  return supplyAsync(() -> {
     try {
        return fetchRemoteLogo(t);
     } catch (IOException e) {
        throw new RuntimeException(e);
     }
  })
  .get(3000, MILLISECONDS);
}

String fetchRemoteLogo (Team t) throws IOException {
  ...
}
```

*Listing 10*

Vavr vermeidet solche Probleme und bringt eine Vielzahl eigener Collections mit. Diese reichen von einer einfachen „List“ bis hin zu eher spezialisierten Datenstrukturen wie „PriorityQueue“. Alle Vavr-Collections sind funktionale Datenstrukturen, sie sind also unveränderlich und persistent. Was das bedeutet, wird an einem einfachen Beispiel klar. Angenommen, wir haben eine „TreeSet“ von Teams

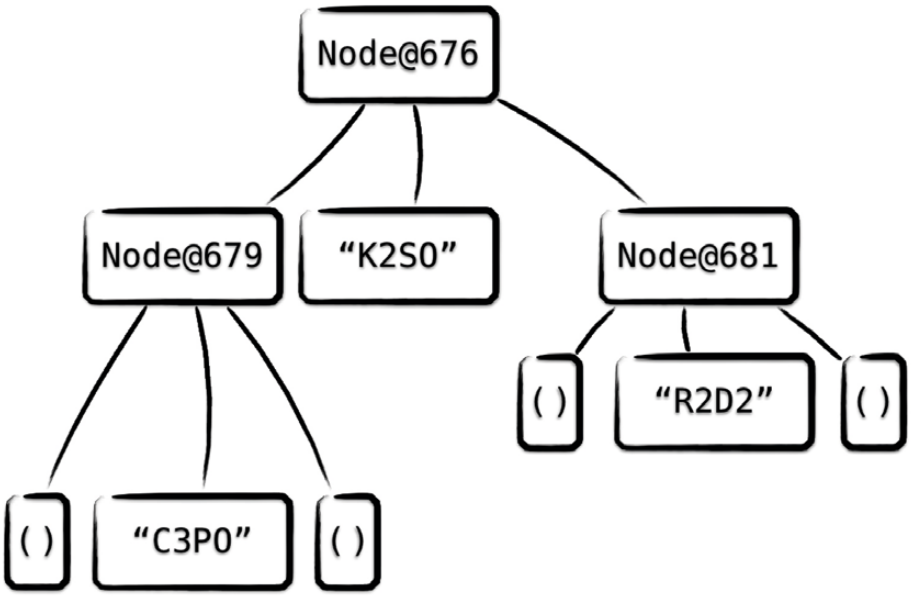


*Abbildung 1: Vavrs TreeSet ist ein Baum, der aus Knoten (wie „Node@681“) und Werten (wie „R2D2“) besteht. Leere (End-) Knoten sind als „()“ dargestellt*

„TreeSet<String> teams = TreeSet.of(“F95”, “FCK”, “FCB”);“ und fügen dieser Menge ein Element hinzu *(siehe Listing 9)*.

Vavr verändert nicht die ursprüngliche „TreeSet“, es wird eine neue erzeugt und durch geschickte Zeiger-Manipulation ein Großteil des Ursprungsbaums wiederverwendet. *Abbildung 1* zeigt, über die „TreeSet“ von „teams“, wie das konkret funktioniert.

Wird ein Element zugefügt, kann Vavr einen Großteil der ursprünglichen Datenstruktur weiter nutzen und muss nur geschickt das neue Element einfügen *(siehe Abbildung 2)*.

Es sollte klar sein, dass solche Operationen in der Implementierung von Vavr recht komplex sind. Für den Nutzer ist die Komplexität glücklicherweise völlig transparent. Wer sich um die Performance sorgt, sei hier auf die Micro-Benchmarks von Vavr verwiesen. Wir bevorzugen immer eine seiteneffektfreie Implementierung und optimieren nur, wo es Sinn ergibt. Im Zweifel ist der Übergang zwischen Vavr- und Java-Collections immer nur ein „toJavaSet“ entfernt.

## Funktionale Ausnahmebehandlung

Exceptions und funktionale Programmierung sind in Java 8 ein trauriges Kapitel. Werfen wir einen Blick auf die Methode „TeamsController#readTeamLogo“ *(siehe Listing 10)*.

Hier wird versucht, über die Methode „fetchRemoteLogo“ das Wappen eines Teams zu laden. Dabei kann eine „IOException“ geworfen werden. Das Ganze soll asynchron über „CompletableFuture#supplyAsync“ mit einem Timeout von drei Sekunden geschehen („CompletableFuture#get“). Daher muss die „IOException“ gefangen und irgendwie sinnvoll behandelt werden: Hier wird sie als „RuntimeException“ gekapselt.

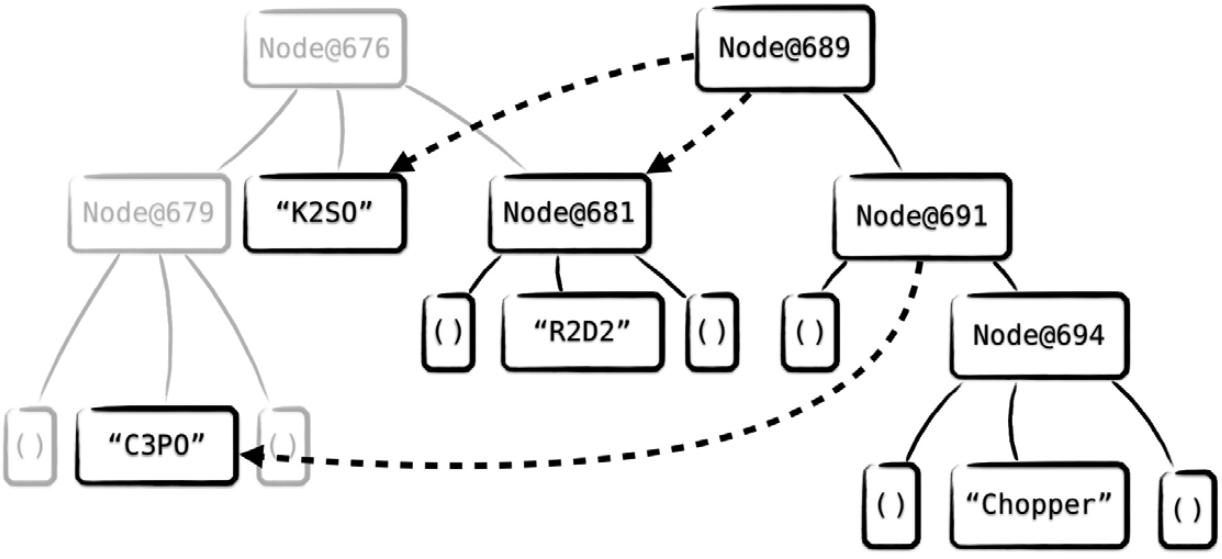


*Abbildung 2: Wird der Wert „STP“ hinzugefügt, verwendet Vavr einen Großteil des Ur-Baums wieder – hier als gestrichelte Pfeile illustriert*

Schließlich kann die Verwendung von „CompletableFuture“ selbst noch einen ganzen Zoo an Exceptions werfen, unter anderem „InterrupedException“. Offensichtlich ist das ganz schön viel Code für eine doch recht simple Semantik. Mit den Vavr-„Try“- und -„Lift“-Konstrukten wird die Fehlerbehandlung funktional und übersichtlich.

Als Erstes kapseln wir „fetchRemoteLogo“ so, dass keine Exception mehr geworfen wird. „fetchRemoteLogo“ ist eine partielle Funktion: Sie ist nur für Teams definiert, deren Logo geladen werden kann. Für alle anderen ist der Rückgabewert nicht definiert. Mit Vavr „CheckedFunction1.lift“ machen wir aus dieser partiellen Funktion eine totale Funktion; die obskure Syntax ist leider Java geschuldet *(siehe Listing 11)*.

„liftedReader“ liefert als Ergebnis ein „Option<String>“, sodass wir einfach im Fehlerfall ein Default-Wappen nutzen können: „liftedReader.apply(team).getOrElse(default);“. Lift ist ein wunderbares Mittel, um Code zu kapseln, den man selber nicht mehr modifizieren kann – beispielsweise wenn man in einem Brownfield-Projekt arbeitet und Legacy Code nutzen muss. Wir nutzen die geliftete Variante nun in der Implementierung von „readTeamLogo“ *(siehe Listing 12)*.

Besser, aber wir haben noch immer die ganzen Exceptions der „CompletableFuture“. Hier kommt Vavr „Try“ zu Hilfe. „Try<T>“ kapselt einen Funktionsaufruf, der eine Exception werfen kann. Das Ergebnis ist dann entweder ein „Success<T>“ mit dem Ergebnis oder ein „Failure<T>“ mit der geworfenen Exception. Sowohl „Success“ als auch „Failure“ lassen sich wie „Option“ nutzen. Das nutzen wir nun in der Implementierung von „readTeamLogo“ *(siehe Listing 13)*.

Diese Funktion drückt bereits durch die Signatur und den Typ der Rückgabe aus, was zu erwarten ist: „Try<String>“ – ein Versuch, einen String zu liefern, der entweder glückt oder eben nicht. Darüber hinaus bietet „Try“ Operationen, die man auch von „composable functions“ kennt. Man kann ohne Weiteres mit „andThen“ etc. mehrere Aufrufe nacheinander funktional kombinieren, ohne in einen Catch-Sumpf abzutauchen. „Try“ bietet aber noch viel mehr. So kann man damit die explizite Behandlung von Ausnahmen sauberer als mit „catch“-Blöcken ausdrücken *(siehe Listing 14)*.

Mit „recover(Exceptiontype, recoverFunction)“ kann auf Ausnahmen vom Typ „Exceptiontype“ ein Ersatzergebnis mithilfe der „recoverFunction“ konstruiert werden.

## Rund um Vavr

Im Vavr-Ökosystem ist noch einiges mehr zu finden. Vavr-Jackson *(siehe „https://github.com/vavr-io/vavr-jackson“)* ist eine Bibliothek, um Vavr-Datentypen in REST-Schnittstellen direkt nutzen zu können. Vavr-Test ist eine Erweiterung für Property Based Testing, ähnlich wie ScalaCheck. Resilience4J *(siehe „https://github.com/resilience4j/resilience4j“)* ist eine Circuit-Breaker-Implementierung auf Basis von Vavr. Damit kann man mit einfachen Mitteln Java-Funktionen mit Circuit-Breakern ausstatten, ohne auf komplexere Frameworks wie Hystrix zurückgreifen zu müssen. Mit Spring Data 2 lassen sich Vavr-Datentypen sogar direkt in Repositories nutzen; *Listing 15* zeigt ein Beispiel. Spring Data generiert die notwendigen Implementierungen, die die gewünschten Vavr-Strukturen liefern.

## Ausblick auf 1.0

Während Javaslang bereits auf dem Weg in Richtung Version 3 war, hat Vavr nun ein paar Schritte in der Versionsnummer zurückgemacht. Aktuell ist Version 0.9x verfügbar, und es wird sicher durch die Refactorings noch ein paar Wochen bis zu einer 1.0 dauern. Diese Version 1.0 steht ganz im Zeichen von Aufräumarbeiten und Vorbereitungen in Hinblick auf Java 9 – insbesondere das Modulsystem. Weiter werden einige Konzepte potenziell „deprecated“, etwa das Pattern Matching auf Objektebene *(siehe „http://blog.vavr.io/pattern-matching-starter“)*, da zukünftige Java-Versionen diese Konzepte nativ unterstützen werden *(siehe „http://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html“)*.

```
Function1<Team, Option<String>> liftedReader =
lift(this:: fetchRemoteLogo);
```

*Listing 11*

```
String readTeamLogo(Team t) throws InterruptedExcep-
tion,… {
  Function1<Team, Option<String>> lifted = …;

  return supplyAsync(() -> lifted.apply(t)
                            .getOrElse(default))
        .get(3000, MILLISECONDS);
}
```

*Listing 12*

```
Try<String> readTeamLogo(Team t) {
  Function1<Team, Option<String>> lifted = …;

  return Try.of(
        () -> supplyAsync(
              () -> lifted.apply(t)
                  .getOrElse(default)
            ).get(3000, MILLISECONDS)
     );
}
```

*Listing 13*

```
readTeamLogo(team)
  .map( /* Erfolgsfall */ )
  // Behandlung der Ausnahme innerhalb der Completa-
bleFuture
  .recover(ExecutionException.class, logoFetch-
Failed())
  // sonstige Fehlerfälle
  .getOrElse(TeamsController::logoFetchTimedoutRespon
se);
```

*Listing 14*

## Fazit

Vavr ist keine Allzweckwaffe. Man wird sicher nicht die Ausdruckstärke von Scala, Haskell oder ähnlichen Sprachen in Java erwarten dürfen. Dafür ist Java auch niemals gedacht gewesen. Es ist auch klar, dass für die Nutzung mit Hibernate oder MongoDB Typ-Konverter geschrieben werden müssen, was man auf punktueller Ebene abwägen muss. Am Ende macht Vavr den Übergang zwischen Java- und Vavr-Datenstrukturen extrem einfach, sodass man immer noch ein Sicherheitsnetz hat.

Schließlich muss man in seinen Projekten entscheiden, wie viele Collection-Bibliotheken parallel sinnvoll sind. Vavr, Guava, Commons Collections, Eclipse Collections etc.; irgendwann ist es halt