04-2016 | Winter | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

Java ist vielseitig

JUnit 5
Das nächste große Release steht vor der Tür

Ansible
Konfigurationsmanagement auch für Entwickler

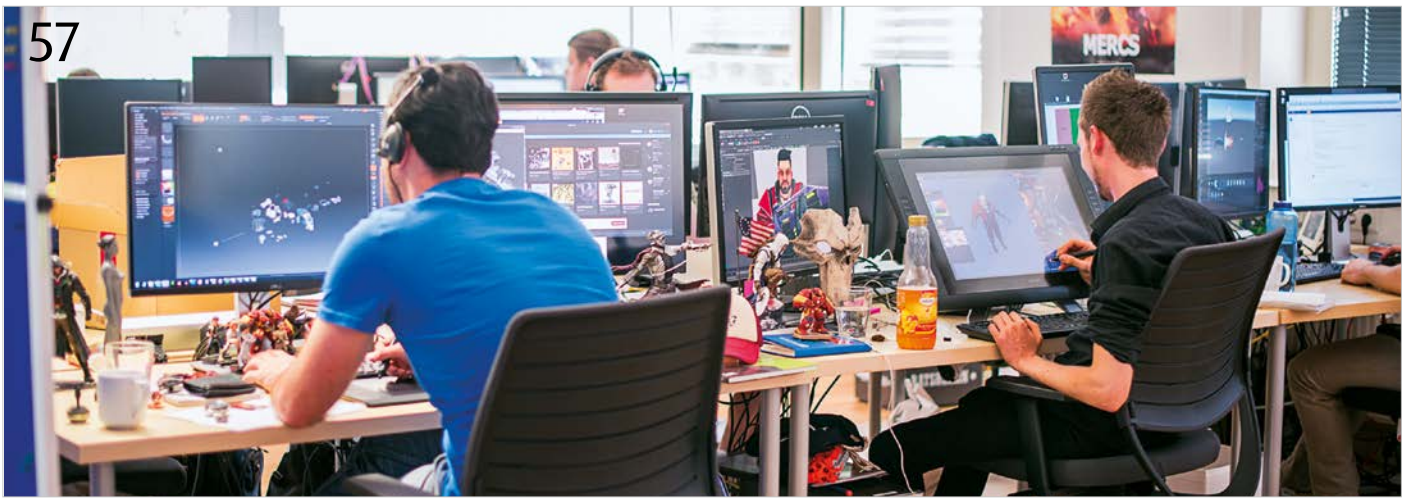
Spring Boot Starter
Komfortable Modularisierung und Konfiguration

4 191978 304903 04

iJUG Verbund

Anwendung im laufenden Betrieb verwalten

Business Process Management auf Open-Source-Basis



Ein Blick hinter die Kulissen eines Spiele-Unternehmens

Optional <Titel>

Dr. Frank Raiser, Konzept Informationssysteme GmbH

Eine der bedeutendsten Änderungen von Java 8 für den Entwickleralltag stellt die Klasse „java.util.Optional" dar. Damit lassen sich zahlreiche Fehlerquellen systematisch vermeiden, wobei gleichzeitig der Quellcode kürzer und verständlicher wird. Dieser Artikel zeigt die Historie der Klasse in anderen Sprachen, ihre Anwendungsmöglichkeiten sowie die Vor- und Nachteile auf. An einem durchgängigen Beispiel werden die verschiedenen Methoden von „Optional" und die daraus resultierende Umstellung auf eine Datenfluss-orientierte Entwicklung erläutert.

Blickt man über die Java-Welt hinaus, erkennt man, dass die Klasse „java.util.Optional" bei Weitem nichts Neues ist. Über lange Zeit war die bekannteste Variante die „Maybe"-Monade in Haskell. Die Programmierung mit Monaden geht auf Arbeiten von Moggi *[1]* Anfang der 1990er-Jahre zurück, sie ist jedoch viel weitreichender. Da man für ein Konzept wie „Optional" aber nicht auf deren Kenntnis angewiesen ist, sind ähnliche Umsetzungen bereits früher zu finden.

Im Laufe der Zeit haben immer mehr Sprachen die Idee eines Typs wie „Optional" aufgenommen. *Tabelle 1* zeigt, wie dieser Typ in verschiedenen Sprachen unter jeweils anderem Namen Verwendung findet. Man sieht auch, dass dies vor allem in statisch getypten Sprachen der Fall ist.

Nur wenige moderne Sprachen lassen einen entsprechenden Typ noch vermissen, darunter C# und JavaScript. Ein Blick auf die Quellen von „java.util.Optional" offenbart jedoch, dass man eine Implementierung mit wenig Aufwand selbst entwickeln kann. Daher gibt es in den entsprechenden Sprachen meist Ersatz in Form von Bibliotheken. Für Java 7 war die Verwendung bereits durch „com.google.guava.Optional" möglich. Die Implementierung in Java 8 ersetzt diese mittlerweile vollständig, da sie nicht nur Teil der JVM, sondern auch deutlich mächtiger ist.

Warum überhaupt „Optional"?

Wer sich mit „Optional" und dessen Einsatzmöglichkeiten noch nicht auseinandergesetzt hat, fragt sich vielleicht, warum wir diese Klasse als Java-Entwickler überhaupt benötigen. Die Frage ist durchaus berechtigt, konnten doch Java-Programme auch ohne diesen Typ schon seit zwei Jahrzehnten erfolgreich entwickelt werden.

Sprachen ohne Unterstützung eines „null"-Werts waren im Gegensatz zu Java dazu gezwungen. Beispielsweise gibt es sonst in Haskell keine Möglichkeit, eine Funktion zu implementieren, die aus einer Liste von Werten einen Wert mit einer bestimmten Eigenschaft heraussucht. Naturgemäß kann die Funktion so aufgerufen werden, dass der gefragte Wert in der Liste nicht vorkommt. Was soll dann aber das Ergebnis der Funktion sein? Sprachen wie Java haben zu diesem Zweck den speziellen Wert „null" eingeführt. Für Haskell wurde stattdessen mithilfe des „Maybe"-Typs ein Weg geschaffen, um auszudrücken, dass kein Ergebnis berechnet werden konnte.

Nun könnte man sich zurücklehnen mit der Erkenntnis, dass „null" das Problem bereits bestens löst. Wäre da nicht ein nagendes, schlechtes Gefühl, das bei jeder „NullPointerException" (NPE) stärker wird.

Die Idee, einen speziellen „null"-Wert in einer Sprache zu definieren, geht auf den mit dem Turing-Award ausgezeichneten Sir Charles Anthony Richard Hoare zurück, der einen solchen im Jahr 1965 erstmals für Algol W einführte. Knapp 40 Jahre später kommentierte er diese Idee als seinen „Milliarden Dollar Fehler" *[2]* und das nur, weil es „so einfach zu implementieren" war. Dass die Verwendung von „null" nicht der Weisheit letzter Schluss sein kann, weiß jeder erfahrene Entwickler. Was liegt also näher, als sich die Lösung anzusehen, die andere Sprachen, ohne „null"-Werte, verwenden?

Erste Verwendung von „Optional"

Um „Optional" als Ersatz für „null" zu verwenden, braucht es nicht viel. In diesem Artikel wird ein einfaches Programmstück bezüglich seiner Umstellung auf „Optional" untersucht. Es soll über die Kommandozeile beim Start einen Parameter „-f" und einen Dateinamen erhalten, um aus dieser Datei einen Konfigurationswert auszulesen. Die Methode in *Listing 1* zeigt, wie man mit klassischem „null"-Ansatz den Dateinamen aus den Kommandozeilen-Parametern ermitteln könnte. Selbstverständlich gibt es bessere Implementierungen und Bibliotheken zu diesem Zweck, aber uns interessiert hier lediglich der Umgang mit „null".

Bei der Verwendung einer solchen Methode muss der Aufrufer anschließend daran denken, das Ergebnis auf „null" zu prüfen. Dies kann leicht vergessen werden, was die Methode automatisch zu einer Fehlerquelle macht. Da „null" die Ursache dieses Problems ist, eliminiert der Einsatz von „Optional" die Fehlerquelle.

Sprache	Bezeichnung
Java, Swift	Optional
Haskell, Idris, Agda	Maybe
Coq, OCaml, Standard ML, F#	Option
Scala, Rust	Option
C++17	optional

Tabelle 1: „Optional"-Typen in verschiedenen Sprachen

```
public String getFilename(String [] arguments) {
  for (int i = 0; i < arguments.length; i++) {
    if ("-f".equals(arguments[i]) && i+1 < arguments.length) {
      return arguments[i+1];
    }
  }
  return null;
}
```

Listing 1: Klassische Methode mit „return null"

```
public Optional<String> findFilename(String [] arguments) {
  for (int i = 0; i < arguments.length; i++) {
    if ("-f".equals(arguments[i]) && i+1 < arguments.length) {
      return Optional.of(arguments[i+1]);
    }
  }
  return Optional.empty();
}
```

Listing 2: Die gleiche Methode mit „Optional"

Listing 2 zeigt, wie die gleiche Funktionalität mit Verwendung von „java.util.Optional" aussehen kann. Der Methodenname wurde geändert, um bereits deutlich zu machen, dass eine Suche nach einem Dateinamen stattfindet, die auch erfolglos bleiben kann. Der Rückgabetyp ist entsprechend auf „Optional<String>" gesetzt. Die „null"-Verwendung wurde durch Optional.empty() ersetzt und der eigentliche Ergebniswert mittels „Optional.of(...)" verpackt.

Für die Konstruktion eines „Optional"-Objekts gibt es neben diesen beiden Factory-Methoden noch „Optional.ofNullable", falls nicht bekannt ist, ob das übergebene Objekt „null" ist oder nicht. Nach der Erzeugung eines solchen „Optional"-Objekts kann man sich dieses als einen Wrapper vorstellen, der die Referenz auf einen Wert beinhaltet. Im Falle von „Optional.empty()" ist diese Referenz selbst „null"; bei „Optional.of(...)" findet sich dort eben der entsprechende Wert. Wer glaubt, dass es so einfach doch nicht sein kann, dem sei ein Blick auf die Quellen der Klasse „java.util.Optional" empfohlen. Tatsächlich findet sich dort wenig Überraschendes.

Die Vorteile

Nun liefert eine Methode „Optional<String>" anstelle eines einfachen „String" – was bringt uns das? Die „Optional"-Klasse bietet eine Methode „isPresent", mit der geprüft werden kann, ob sich in dem gekapselten Objekt ein gültiger Wert befindet. Damit lässt sich eine „null"-Prüfung umsetzen; eine Verwendung des „String"-Objekts ist anschließend mit „Optional#get" ebenfalls möglich. Bei dieser Umstellung wäre somit nur der Code ein wenig komplizierter geworden, aber Ablauf und Prüfungen blieben im Wesentlichen unverändert. Wer „Optional" nur so weit kennt oder die Klasse nur so verwendet, sollte unbedingt weiterlesen.

Wie bereits gesehen, ist eine Fehlerquelle vollständig ausgeschlossen: Ein Entwickler ist nun nicht mehr in der Lage zu vergessen, dass der Dateiname eventuell gar nicht erfolgreich ermittelt werden konnte. Wird etwa ein „null-String" an den „java.io.File"-Konstruktor übergeben, führt dies zur Ausführungszeit zu einer „NullPointerException". „Optional<String>" schließt dies aus, da der Compiler eine direkte Übergabe an den „File"-Konstruktor nicht akzeptiert.

Ein „Optional<T>" ist kein „T"-Objekt. Daher sind auch die NPEs, die von ei-

nem Methodenaufruf eines „null T"-Objekts wie „String#length" stammen, mit „Optional<T>" nicht möglich – mit Ausnahme der wenigen Methoden, die die Klasse „Optional" selbst anbietet.

Richtig interessant wird „Optional" aber erst, wenn man dessen Methoden mit ins Spiel bringt. Die Kurzfassung für funktionale Programmierer lautet: „java.util.Optional" ist eine Monade. Für alle anderen werden diese Vorteile in den folgenden Abschnitten am Beispiel Punkt für Punkt aufgezeigt.

Das Beispiel: Aus einem erfolgreich gelesenen Dateinamen soll ein „java.io.File"-Objekt entstehen. In einer klassischen Implementierung benötigt man dazu die „null"-Prüfung und kann dann den „File"-Konstruktor aufrufen. Das Ergebnis vom Typ „File" kann somit wiederum „null" sein. Anstatt dies analog mit „Optional<String>" nachzuprogrammieren, zeigt *Listing 3*, dass es mit „Optional#map" deutlich kürzer geht.

Die Methode „map" überprüft, ob ein Wert vorhanden ist. Falls ja, wird dieser an die übergebene Funktion – in diesem Fall an den Konstruktor von „File" – übergeben. Das Ergebnis von „map" ist somit entweder das – wieder in „Optional" verpackte – Ergebnis dieser Funktion oder „Optional.empty()".

Als Nächstes wird die Datei nur ausgelesen, falls sie auch wirklich existiert. Da man mit „File" auch nicht vorhandene Dateien repräsentieren kann, lässt sich dies mit „File#exists" prüfen. Im klassischen, „null"-basierten Java-Code stände hier eine weitere Fallunterscheidung an. *Listing 4* zeigt, wie „Optional#filter" das wiederum kürzer und im gleichen Kontrollfluss erledigen kann.

Wie genau der Konfigurationswert aus der Datei ausgelesen wird, ist hier nicht von Bedeutung. Nehmen wir daher an, dass eine Methode „readConfigValue(File)" existiert, die einen „Integer"-Wert als Ergebnis liefern kann. Sehen wir uns zuerst in *Listing 5* eine vollständige Implementierung der Beispielfunktion mithilfe von „null"-Rückgabewerten an. Sollte etwas schiefgehen, so soll dieser Konfigurationswert standardmäßig auf „0" gesetzt sein. Man sieht gut, dass sich selbst in einem so einfachen Beispiel bereits drei Fall-Unterscheidungen im Quellcode ergeben; das ist ein gutes Indiz für die Komplexität.

Selbstverständlich könnte der entsprechende Eintrag in der Datei selbst fehlen. Dementsprechend benennen wir die Methode „readConfigValue" in „findConfigValue" um und ändern den Rückgabetyp zu „Optional<Integer>". Somit entsteht ein „Optional<File>"-Objekt und mithilfe der „map"-Methode wäre das Ergebnis nach dem Aufruf von „findConfigValue" ein „Optional<Optional<Integer>>"-Objekt. Um eine derart ungewünschte Verschachtelung zu vermeiden, gibt es die Methode „Optional#flatMap". Sie funktioniert genauso wie „map", entfernt aber eine der Schachtelungen, sodass wir ein „Optional<Integer>"-Objekt erhalten.

Ferner gibt es die Methode „Optional#orElse" für den oft benötigten Fall, dass bei Abwesenheit eines Werts ein Standardwert herangezogen werden soll. Sollte in dem „Optional"-Objekt ein gültiger Wert enthalten sein, wird dieser zurückgeliefert, andernfalls der an „orElse" übergebene Wert.

Der Aufruf von „orElse" ist ein Methodenaufruf, dessen Argument fertig berechnet übergeben werden muss. Soll aber beispielsweise ein Standardwert aus einer langsamen Datenbank ermittelt werden, will man diese Auswertung möglichst einsparen können. Zu diesem Zweck kommt „Optional#orElseGet(Supplier<? extends T>)" zum Einsatz, um erst bei feststehender Abwesenheit des Werts den teuren Standardwert zu berechnen. Bringt man dies alles zusammen, erhält man die Methode aus *Listing 6*. Es bleibt lediglich eine einzelne „return"-Anweisung übrig. Diese ist verhältnismäßig leicht zu verstehen.

Die meisten Entwickler benötigen eine kurze Eingewöhnungszeit, aber dann sind insbesondere „map", „flatMap" und „filter" genauso natürlich zu lesen und zu verstehen wie klassische Methodenaufrufe oder Operatoren. Insbesondere, da auch das in Java 8 eingeführte Stream-API auf Methoden mit den gleichen Namen und einer ähnlichen Bedeutung basiert.

Interessant für Java-Entwickler ist auch die Erkenntnis, dass mit Java 8 ein Wechsel von Kontrollfluss- hin zu Datenfluss-orientierter Entwicklung stattfindet. Beim Vergleich der *Listings 5* und *6* sieht man gut, wie im ersten Fall die Kontroll-Struk-

```
Optional<File> file = findFilename(arguments).map(File::new);
```

Listing 3: Verwendung von „Optional#map"

```
Optional<File> existingFile = findFilename(arguments)
      .map(File::new)
      .filter(File::exists);
```

Listing 4: Verwendung von „Optional#filter"

```
private Integer getConfigValue(String[] arguments) {
  String filename = getFilename(arguments);
  if (filename != null) {
    File file = new File(filename);
    if (file.exists()) {
      Integer configValue = readConfigValue(file);
      if (configValue != null) {
        return configValue;
      }
    }
  }
  return 0;
}
```

Listing 5: Vollständige Implementierung mit „null"

```
private Integer getConfigurationValue(String[] arguments) {
  return findFilename(arguments)
          .map(File::new)
          .filter(File::exists)
          .flatMap(this::findConfigValue)
          .orElse(0);
}
```

Listing 6: Vollständige Implementierung mit „Optional"

turen zur Ablaufsteuerung dominieren. Im zweiten Listing hingegen kann man gut von oben nach unten lesend die Verarbeitungskette der Daten sehen:

- Dateiname aus Kommandozeilen-Argumenten heraussuchen
- File-Objekt daraus erstellen
- Sicherstellen, dass die Datei existiert
- Den Konfigurationswert aus der Datei lesen
- Im Fehlerfall den Standardwert „0" verwenden

Die funktionale Programmierung wirbt schon lange zu Recht damit, dass ein Entwickler mit Fokus auf den Datenfluss näher an der eigentlichen Problemstellung bleibt. Letzten Endes geht es bei Software primär um das Verarbeiten von Daten; Kontrollflüsse waren immer nur Mittel zu diesem Zweck. Auch das Stream-API verfolgt diese Philosophie und abstrahiert deswegen von Ablauf-Logiken und Parallelität, um dem Entwickler die Möglichkeit zu geben, sich auf den Datenfluss zu konzentrieren.

Die Nachteile

Nachdem nun zahlreiche Vorteile der Verwendung von „java.util.Optional" identifiziert sind, soll an dieser Stelle auch auf die Nachteile eingegangen werden. Zuallererst sind dies die offensichtlichen Nachteile der meisten Abstraktionen: Laufzeit- und Speicher-Kosten. Einen Wert in ein „Optional"-Objekt zu verpacken, kostet offensichtlich mehr Speicher, als nur für den Wert selbst benötigt wird. Der Overhead ist hier allerdings ähnlich gering wie etwa beim Boxing von „int" auf „Integer". Auch die Laufzeit ist geringfügig langsamer, da eine zusätzliche Indirektion über die „Optional"-Klasse stattfindet. In der Praxis sind diese Kosten jedoch in den allermeisten Fällen vernachlässigbar gering.

Interessanter ist der Nachteil, dass die Grundidee von „Optional", also die Vermeidung von „null" und NPEs, allein durch diese Klasse nicht verwirklicht werden kann. Es gibt den pathologischen Fall, dass eine Methode mit Rückgabetyp „Optional<T>" mithilfe eines „return null;" implementiert wird. Durch Reviews und das Wissen aus Artikeln wie diesem lässt sich einer derart bösartigen Verwendung von „Optional" jedoch effizient entgegenwirken. Nichtsdestotrotz ist und bleibt „null" auf absehbare Zeit ein Bestandteil von Java und somit wird auch die „NullPointerException" nicht vollständig verschwinden.

Insbesondere in neuen Projekten gibt es seit Java 8 kaum noch gute Gründe, um das „null"-Schlüsselwort zu verwenden. Ein Fall, in dem diese Verwendung beispielsweise noch immer unvermeidbar sein kann, ist die Ansteuerung von Bibliotheken. Dabei kann ein Methoden-Argument mit „null" übergeben werden müssen, um die gewünschte Funktionalität nutzen zu können.

Aber auch das JDK selbst hat in Version 8 seine Nachteile bezüglich „Optional". Aus Gründen der Abwärtskompatibilität wurde darauf verzichtet, bestehende APIs auf „Optional" umzuziehen. So kann etwa die Anfrage an eine „Map" mit einem darin nicht enthaltenen Schlüssel weiterhin „null" liefern. In modernen Sprachen wie Scala ist die „get"-Methode mit einem Rückgabetyp „Optional" definiert. In Java 8 muss man stattdessen daran denken, „Optional.ofNullable" für das Ergebnis von „Map#get" zu nutzen.

Wer einmal mit einem „Stream <Optional<T>>" hantiert hat, wird sich zudem fragen, warum es kein „Optional #stream" gibt, um eine Vereinfachung durch „Stream#flatMap" erreichen zu können. Dieser Defekt *[3]* wird leider erst mit Java 9 behoben sein.

Ähnlich wie beim Stream-API gestaltet sich das Debuggen von Lambda-Ausdrücken, wie sie „map", „flatMap" und „filter" verwenden, in den aktuellen IDEs häufig noch unnötig schwierig. Dies liegt daran, dass die IDEs in ihrer Entwicklung zum Teil noch nicht so weit sind, und sollte sich im Laufe der Zeit durch entsprechende Updates erübrigen.

Abschließend ist noch anzumerken, dass die Verwendung von „Optional" Fehlerursachen automatisch unterdrückt. Man sieht einem „Optional.empty()" nicht an, weswegen der Wert fehlt. Soll etwa in obigem Beispiel eine Meldung ausgegeben werden, falls die angegebene Datei nicht existiert, ist die Datenverarbeitungskette zu diesem Zweck zu unterbrechen. Wer oft solche Anforderungen hat, kann sich von fortgeschritteneren Techniken wie zum Beispiel der „Validation"-Monade von scalaz *[4]* inspirieren lassen.

Fazit

Die Vorteile des „Optional"-Einsatzes überwiegen bei Weitem die Nachteile. Nicht umsonst ist dieses Konzept in derart vielen Programmiersprachen zu finden. Seit der Einführung von Lambda-Ausdrücken und Streams mit Java 8 ist nun auch eine elegante monadische Verwendung möglich. Dadurch gelingt es in zunehmendem Maße, in Java 8 Datenfluss-orientiert zu entwickeln, was kürzere, verständlichere und weniger komplexe Programme ergibt.

Der Autor setzt „java.util.Optional", dessen Vorgänger „com.google.guava.Optional" und Varianten in anderen Sprachen seit Jahren in verschiedenen Projekten mit großem Erfolg ein. Die Klasse ist für Team-Mitglieder schnell zu erlernen und einfach zu verwenden; die Vorteile sind in der Praxis schon nach kurzer Zeit deutlich zu spüren. Aufgrund dieser Erfahrungen hat der Autor ein Checkstyle-Plug-in *[5]* geschrieben, das Verwendungen des „null"-Schlüsselworts im Quellcode findet und beanstandet. Bei bestehenden Projekten lässt sich somit die Zahl der Verwendungen stetig reduzieren. Seit etwa einem Jahr entwickelt der Autor zudem mit einem kleinen Team ein Java-Projekt, in dessen Quellcode kein einziges „null" mehr zu finden ist. „NullPointerExceptions" sind dementsprechend selten.

Quellen

[1] Moggi, Eugenio (1991), „Notions of computation and monads". Information and Computation 93 (1)

[2] Hoare, Tony (25 August 2009). „Null References: The Billion Dollar Mistake". *InfoQ.com*.

[3] *https://bugs.openjdk.java.net/browse/JDK-8050820*

[4] *https://github.com/scalaz/scalaz*

[5] *https://github.com/FrankRaiser/checkstyle-null-literal*

Dr. Frank Raiser
frank.raiser@konzept-is.de

Dr. Frank Raiser arbeitet als Software-Entwicklungsingenieur bei der Konzept Informationssysteme GmbH in Ulm. Er berät und unterstützt Projekte sowie Kunden bei technischen Fragestellungen und führt Schulungen und Coachings zu Themen wie „Java", „Clean Code" oder „Agile Entwicklung" durch. Seine Schwerpunkte sind moderne Ansätze in Programmiersprachen und Software-Entwicklungsprozessen.