ToiletPaper #131

jambit WHERE INNOVATION WORKS

Try but don't catch: Elegantes Exception Handling mit der Try-Monade

Autor: André Petermann / Software Architect / Standort Leipzig

✗ Problem

Für eine robuste Software ist der sorgfältige Umgang mit Exceptions unumgänglich. Gängige Strategien sind entweder lokales Behandeln mit **try{}catch(**SpecificException ex**){}** oder zentrales Behandeln. Letzteres bedeutet, dass Exceptions nach oben gegeben und zentral mit mehreren **catch**-Blöcken behandelt werden. Lokales Behandeln stört die Lesbarkeit des Programmflusses. Besonders unbeliebt sind Checked Exceptions, da diese lokales Behandeln erzwingen oder mit der **throws**-Klausel explizit nach oben gegeben werden müssen. In Java hat dies zu Entwicklungen wie Lombok's **@SneakyThrows** geführt. Dabei werden Checked Exceptions per Code-Generierung in Runtime Exceptions überführt. Konzeptionell ist zentrales Behandeln jedoch nicht weit entfernt von **GOTO**, da der eigentliche Programmfluss abgebrochen werden muss. In der Praxis findet man meist eine Kombination aus beiden Ansätzen. In gruseligen Fällen werden sogar neue Programmflüsse aus zentral behandelten Exceptions heraus gestartet.

✓ Lösung

In den letzten Jahren haben verschiedene Konzepte der funktionalen Programmierung Einzug in den Mainstream der Softwareentwicklung gehalten. Beispiele in Java sind Lambda-Ausdrücke oder die Monaden **Optional** und **Stream**. Für Exception Handling gibt es in der Welt der funktionalen Programmierung das Konzept *Railway Oriented Programming* mit Monaden wie **Try** oder **Result**. Mit der Bibliothek **VAVR** kann man Java einfach um die **Try**-Monade erweitern. Ihre Verwendung ist nicht komplizierter als die (richtige) Verwendung von **Optional**. Hier wird mit **Try.of(**() -> ...**)** die Ausführung einer Funktion in einen monadischen Kontext **Try<T>** überführt, der entweder **Success<T>** oder **Failure** sein kann. Dabei wird entweder das Ergebnis der Funktion (Typ **T**) oder ein **Throwable** gespeichert, dass später behandelt werden kann. Zudem können mit **try.map(**x -> ...**)** weitere Operationen auf den inneren Wert angewendet werden, welche nur im Kontext **Success** ausgeführt wird. Ist vorab schon eine Exception aufgetreten, so wird diese bei **map** einfach "weitergereicht". Analog dazu, können mit **mapFailure** auch Operationen auf dem Throwable ausgeführt oder mit **flatMap** der Kontext gewechselt werden.

➔ Beispiel

Success

```
Try<Integer> myTry = Try
 .of(() -> 10 / 2)
 .map(i -> i + 1);

assertTrue(myTry.isSuccess());
myTry.onSuccess(
 i -> assertEquals(6, i));
```

Failure

```
Try<Integer> myTry = Try
 .of(() -> 10 / 0)
 .map(i -> i + 1);

assertTrue(myTry.isFailure());
myTry.onFailure(
  ex -> assertTrue(
    ex instanceof
      ArithmeticException));
```

Von Failure zu Success

```
// Für alle Fälle
Try<Integer> try2 = myTry
  .orElse(() -> 0)

// Für bestimmte Exceptions
Try<Integer> try2 = myTry.recover(
  ArithmeticException.class,
  ex -> 0)
```

Von Success zu Failure

```
Try.of(() -> send(request))
   .flatMap(resp -> resp.isEmpty()
     ? Try.failure(new MyExptn())
     : Try.success(resp));
```

Ergebnis-Zugriff

```
T t = myTry.getOrElseGet(ex -> ...)

Either<Throwable, T> =
  myTry.toEither()
```

Exception Handling in Streams

```
List<Object> input = List.of(...);

List<Try<Object>> trys =
  input.stream()
       .map(o ->
         Try.of(() -> o)
            .map(this::validate)
            .map(this::convert)
            .map(this::saveToDb))
       .collect(
         Collectors.toList());

trys.stream()
    .filter(Try::isFailure)
    .forEach(
      t -> t.onFailure(log::warn));

List<Object> output =
  trys.stream()
      .filter(Try::isSuccess)
      .map(t -> t.getOrElseGet(
        ex -> ...))
      .collect(Collectors.toList());
```

✚ Weiterführende Aspekte

- Im VAVR user guide findet ihr leider nur eine kurze Abhandlung, einen besseren Guide gibt es bei Baeldung.
- Am besten werft ihr einen Blick in den gut dokumentierten Quellcode!
- Das generelle Konzept von Railway Oriented Programming wird in diesem Artikel gut erklärt.
- Eine weitere Try-Implementierung findet ihr in Scala.