ToiletPaper #131

jambit WHERE INNOVATION WORKS

Try but don't catch: Elegant Exception Handling Using the Try Monad

Author: André Petermann / Software Architect / Office Leipzig

✗ Problem

Thorough exception handling is essential to achieve robust software. Typically, exceptions are either handled locally by **try{}catch(**SpecificException ex**){}** or in a centralized manner. The latter means that exceptions are propagated up the call hierarchy and actual handling is implemented by multiple **catch**-blocks at a single place. Local handling has a negative impact on code readability. In particular, checked exceptions are quite unpopular among programmers since they enforce either local handling or explicit upwards propagation by the **throws**-clause. This situation even lead to Java developments such as Lombok's **@SneakyThrows** where checked exceptions are converted into runtime exceptions by automated code generation. On the other hand, centralized handling is conceptually similar to **GOTO** because the actual program flow is interrupted. In professional practice, we mostly find a combination of both approaches. In scary cases, centralized handling is even used to start new program flows.

✓ Solution

In recent years, several concepts that originate from functional programming made their way to the mainstream of software development. In the Java language, examples are lambda expressions as well as the monads **Optional** and **Stream**. With regard to exception handling, there is a functional concept: Railway Oriented Programming. It is based on monads such as **Try** or **Result**. The Java library VAVR includes an implementation of the **Try** monad. Its usage is not more complex than the (intended) usage of Optional. By calling **Try.of(**() -> ...**)** a supplier function is wrapped in the monadic context **Try<T>** which is either **Success<T>** or **Failure.** After execution, either the function's result (Type **T**) or a **Throwable** will be stored for further processing. For example, operators such as **try.map(**x -> ...**)**, whose input is the inner value, can be specified but will only be executed in the context of **Success**. However, if an exception has occurred, the operator will do nothing but just passing on the Throwable. Similar to this, it is also possible to specify operations on the Throwable with **mapFailure** or to change context with **flatMap**.

➔ Example

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));
```

From Failure to Success

```
// Generally
Try<Integer> try2 = myTry
  .orElse(() -> 0)

// For specific exceptions
Try<Integer> try2 = myTry.recover(
  ArithmeticException.class,
  ex -> 0)
```

From Success to Failure

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

Access to inner value

```
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());
```

+ Further Aspects

- The VAVR user guide contains a short introduction but Baeldung provides a better guide.
- It's best to have a look at the well documented source code.
- The general concept of Railway Oriented Programming will be explained in this video.
- A further implementation of Try can be found in Scala.