Combining `Try` and `Either`

Effective Programming in Scala

Combining `Try` and `Either`

In the previous lessons, we have seen how to manage exceptions with `Try`, and how to manage data validation with `Either`.

In a complete program that manages both exceptions and validations, how do we combine both `Try` and `Either`?

Parsing a List of Dates, Revisited

Let us have a look at our example where we want to parse an arbitrary number of dates from a file. Each date is written in a new line.

We want to return an error message for each date that was not properly formatted.

In case another error happens (e.g., unable to read the file) the program should log the error in the standard error stream and stop.

Parsing a List of Dates: Program Structure

We break down our program into two main components. The first component, `readAndParseDates`, tries to read the file and parse its content. The second component, `run`, calls the first component and shows the error message in case of failure:

```
def readAndParseDates(fileName: String): Try[Validated[Seq[LocalDate]]]

def run(fileName: String): Unit
```

The return type of `readAndParseDates` explicitly models the two types of errors we want to handle. `Try` models errors that can abort our program execution, whereas `Validated` models the fact that the dates read in the file can possibly be invalid.

Parsing a List of Dates: Reading the File

The first step for implementing `readAndParseDates` is to read the file content, which we do in a separate method, `readDateStrings`.

Reading a file with `Source.fromFile` can throw an exception.

We explicitly signal to our callers that these errors can happen by having the return type `Try`:

```
def readDateStrings(fileName: String): Try[Seq[String]] =
  Using(Source.fromFile(fileName)) { source =>
    source.getLines().toSeq
  }
```

Parsing a List of Dates: Validating Dates

The Java operation `LocalDate.parse` signals failures by throwing exceptions. We want to handle such errors by turning them into invalid results (`Left`).

We achieve that by wrapping the call to `LocalDate.parse` in a `Try` and then by converting failures into `Left` values, and successes into `Right` values:

```
def parseDate(str: String): Validated[LocalDate] =
  Try(LocalDate.parse(str)) match
    case Failure(exception) => Left(Seq(exception.toString()))
    case Success(date)      => Right(date)
```

Parsing a List of Dates: `readAndParseDates` Implementation

We implement `readAndParseDates` by reusing `readDateStrings` and `parseDate`.

```
def readAndParseDates(fileName: String): Try[Validated[Seq[LocalDate]]] =
  readDateStrings(fileName).map { dateStrings =>
    validateEach(dateStrings)(parseDate)
  }
```

Parsing a List of Dates: run

```
def run(fileName: String): Unit =
  readAndParseDates(fileName) match
    case Failure(exception) =>
      System.err.println(s"Unable to parse dates file: $exception")
    case Success(validatedDates) =>
      validatedDates match
        case Left(errors) =>
          println(s"Invalid data: ${errors.mkString(", ")}")
        case Right(dates) =>
          println(s"Successfully parsed dates: ${dates.mkString(", ")}")
```

Summary

The more types of errors we model, the more complex our types become.

In this case, we have used the type `Try[Validated[A]]` to model the fact that there are two types of errors:

1. validation errors that we want to report to the user for them to fix; and
2. other errors that the user cannot deal with.