Manipulating `Validated` Values

Effective Programming in Scala

Reminder: Operations on Type `Validated`

We have seen several building blocks for manipulating validated values.

- *transform* valid values with `map`,
- *chain* validation rules with `flatMap`,
- *aggregate* valid values (accumulating validation errors) with `validateBoth`.

In this lesson, we will discover another building block for aggregating an arbitrary number of valid values.

Aggregating an Arbitrary Number of Valid Values

Let us revisit the problem of parsing an arbitrary number of dates in a file.

For now, let us assume that we have already read each line of the file, so we have a `List[String]`.

We want to parse each of these `String` elements as a date, and return a valid list of dates in case of success, or a list of errors in case there is at least one failure.

Aggregating an Arbitrary Number of Valid Values

Let us revisit the problem of parsing an arbitrary number of dates in a file.

For now, let us assume that we have already read each line of the file, so we have a `List[String]`.

We want to parse each of these `String` elements as a date, and return a valid list of dates in case of success, or a list of errors in case there is at least one failure.

```
def parseDates(strs: Seq[String]): Validated[Seq[LocalDate]]
```

Aggregating an Arbitrary Number of Valid Values

Let us revisit the problem of parsing an arbitrary number of dates in a file.

For now, let us assume that we have already read each line of the file, so we have a `List[String]`.

We want to parse each of these `String` elements as a date, and return a valid list of dates in case of success, or a list of errors in case there is at least one failure.

```
def parseDates(strs: Seq[String]): Validated[Seq[LocalDate]]
```

We already know how to parse a single `String` as a date:

```
def parseDate(str: String): Validated[LocalDate]
```

Aggregating an Arbitrary Number of Valid Values (Solution)

```
def parseDates(strs: Seq[String]): Validated[Seq[LocalDate]] =
  strs.foldLeft[Validated[Seq[LocalDate]]](Right(Vector.empty)) {
    (validatedDates, str) =>
      val validatedDate: Validated[LocalDate] = parseDate(str)
      validateBoth(validatedDates, validatedDate)
        .map((dates, date) => dates :+ date)
  }
```

Generalizing To Any Validation Rule

The method `parseDates` validates that each `String` element of a list is a date. It returns either a valid list of `LocalDate`, or a list of validation errors.

In practice, validating each element of a collection is a common thing to do. Could we generalize what `parseDates` does to an arbitrary type of elements `A`, to which we want to apply an arbitrary validation rule `A => Validated[B]`?

Generalizing To Any Validation Rule

The method `parseDates` validates that each `String` element of a list is a date. It returns either a valid list of `LocalDate`, or a list of validation errors.

In practice, validating each element of a collection is a common thing to do. Could we generalize what `parseDates` does to an arbitrary type of elements `A`, to which we want to apply an arbitrary validation rule `A => Validated[B]`?

```
def validateEach[A, B](as: Seq[A])(f: A => Validated[B]): Validated[Seq[B]]
```

Generalizing To Any Validation Rule (Solution)

```scala
def validateEach[A, B](
  as: Seq[A]
)(
  validate: A => Validated[B]
): Validated[Seq[B]] =
  as.foldLeft[Validated[Seq[B]]](Right(Vector.empty)) {
    (validatedBs, a) =>
      val validatedB: Validated[B] = validate(a)
      validateBoth(validatedBs, validatedB)
        .map((bs, b) => bs :+ b)
  }
```

Revisiting parseDates

```
def parseDates(strs: Seq[String]): Validated[Seq[LocalDate]] =
  validateEach(strs)(parseDate)
```

Revisiting parseDates

```
def parseDates(strs: Seq[String]): Validated[Seq[LocalDate]] =
  validateEach(strs)(parseDate)
```

Luckily, third-party libraries already implement `validateEach` for you. This method is often called `traverse`, or `foreach`.

Summary

- Transform valid data with `map`
- Aggregate valid data with `validateBoth`
- Chain validation rules with `flatMap`
- Validate a collection of values with `validateEach`

These operations are the building blocks for implementing complex validation scenarios.

(In practice, these operations might be provided with different names by third-party libraries)