



# Validating Data

Effective Programming in Scala

# Introduction

We have seen how to use exceptions (or Try) to handle errors.

These solutions have the property of stopping the execution flow of the program as soon as there is an error.

As a consequence, they might not be appropriate for reporting validation errors to the end-users of an application. Indeed, in this case we might want to provide a **list of errors** (e.g., a list of properties that are invalid in a JSON object, or a list of invalid fields in a form).

# Validation Result

In practice, your company or project might use a third-party library to handle validation errors.

In this lesson, we introduce a general framework to reason about validation errors, without relying on a specific library.

First, we model validation errors as a collection of String messages:

```
type Errors = Seq[String]
```

Then, we model the type of “a validated value of type A” to be either Errors or A:

```
type Validated[A] = Either[Errors, A]
```

## Aside: Type Aliases

Type aliases are handy to provide a shorthand name for a complex type expression. In our case, we can just write `Validated[Int]` in place of `Either[Seq[String], Int]`.

Type aliases incur no runtime costs (there is no class added), and can be interchangeably used with the type they are an alias to.

# Constructing Either Values

The type `Either` comes from the Scala standard library and models a value that can be either of one type or of another type.

`Either` takes two type parameters, the *left* type, and the *right* type.

```
type Validated[A] = Either[Errors, A]
```

In our example, a valid value of type `Validated[Int]` can be constructed like this:

```
val validInt: Validated[Int] = Right(42)
```

Conversely, an invalid value of type `Validated[Int]` would be constructed as follows:

```
val invalidInt: Validated[Int] = Left(Seq("Invalid integer"))
```

## Consuming Either Values

We can use pattern matching to inspect the content of a validated value:

```
validInt match  
  case Right(n)      => println(s"Got a valid integer: $n")  
  case Left(errors) => println(s"Got validation errors: $errors")
```

## Transforming Validated Values

Like with Try, we can use the operation map to transform a valid (*right*) value without having to deal with the error cases:

```
validInt.map(n => n + n) // : Validated[Int] = Right(84)
```

```
invalidInt.map(n => n + n)  
// : Validated[Int] = Left(Seq("Invalid integer"))
```

## Transforming Validated Values

Like with Try, we can use the operation map to transform a valid (*right*) value without having to deal with the error cases:

```
validInt.map(n => n + n) // : Validated[Int] = Right(84)
```

```
invalidInt.map(n => n + n)  
// : Validated[Int] = Left(Seq("Invalid integer"))
```

- ▶ The operation map only transforms the “right” value of an Either. For this reason, our Validated[A] type sets the error on the “left” side, and the success on the “right” side.

```
type Validated[A] = Either[Errors, A]
```



## Chaining Validations

Again, similar to Try, we can use the operation flatMap to transform a valid value with a function that also returns a validated value.

For instance, to parse two dates and compute the period between them:

```
def parseDate(string: String): Validated[LocalDate] = ...

def validatePeriod(str1: String, str2: String): Validated[Period] =
  parseDate(str1).flatMap { date1 =>
    parseDate(str2).map { date2 =>
      Period.between(date1, date2)
    }
  }
}
```

## Chaining Validations

Again, similar to Try, we can use the operation flatMap to transform a valid value with a function that also returns a validated value.

For instance, to parse two dates and compute the period between them:

```
def parseDate(string: String): Validated[LocalDate] = ...

def validatePeriod(str1: String, str2: String): Validated[Period] =
  parseDate(str1).flatMap { date1 =>
    parseDate(str2).map { date2 =>
      Period.between(date1, date2)
    }
  }
```

It is worth noting that this implementation is very similar to the one that uses Try (in the previous lesson).

## Chaining Validations (2)

```
validatePeriod("not a date", "not a date either")  
// : Validated[Period] = Left(Seq("Text 'not a date' could not be parsed"))
```



Only one error is returned! Our goal is to report all the possible errors to the user.

## Meet validateBoth

We need an operation for combining validated values such that:


- ▶ two valid values produce a valid value containing a pair,
- ▶ one or two invalid values produce an invalid value containing all the validation errors.

The signature of this operation would be the following:

```
def validateBoth[A, B](  
  validatedA: Validated[A],  
  validatedB: Validated[B]  
): Validated[(A, B)]
```

## Using validateBoth

```
def validatePeriod(str1: String, str2: String): Validated[Period] =  
  validateBoth(parseDate(str1), parseDate(str2))  
    .map((date1, date2) => Period.between(date1, date2))
```



## Using validateBoth

```
def validatePeriod(str1: String, str2: String): Validated[Period] =  
    validateBoth(parseDate(str1), parseDate(str2))  
        .map((date1, date2) => Period.between(date1, date2))
```

```
validateDuration("not a date", "not a date either")  
// : Validated[Period] = Left(  
//   List(  
//     "Text 'not a date' could not be parsed",  
//     "Text 'not a date either' could not be parsed"  
//   )  
// )
```

Both errors are reported!

# Implementation of validateBoth

```
def validateBoth[A, B](  
  validatedA: Validated[A],  
  validatedB: Validated[B]  
): Validated[(A, B)] =  
  (validatedA, validatedB) match  
    case (Right(a), Right(b)) => Right((a, b))  
    case (Left(e), Right(_)) => Left(e)  
    case (Right(_), Left(e)) => Left(e)  
    case (Left(e1), Left(e2)) => Left(e1 ++ e2)
```

## Summary

We have discovered a useful operation for combining validation result, which accumulates validation errors.

We have called it `validateBoth`, but it may exist under the name `product` or `zip` in third-party libraries.

In addition to `validateBoth`, transforming valid values (`map`), and chaining operations that validate values (`flatMap`) are still useful operations.