

# Data And Scala

## Tech Blogs

- [Case Classes](#)
  - [Quick Intro To Case Classes](#)
  - [Separation Of Concerns: Case Class Access Scope, Validation, And Derived Fields](#)
- [Data Validation](#)
  - [Avoid Throwing Exceptions](#)
  - [Using Either\[E, A\] And MapN For Data Validation](#)
  - [Use Union Types Instead Of Either\[E, A\] For A More Efficient Validation?](#)

# Case Classes

# Quick Intro To Case Classes

The case class in scala is the most relevant feature in scala for handling data effectively.

```
final case class Employee(firstName: String, lastName: String, ssn: String)
```

The **final** qualifier makes sense because extending a case class may lead to inconsistencies and performance issues as well (

<https://gist.github.com/chaotic3quilibrium/58e78a2e21ce43bfe0042bbfbb93e7dc>)

A case class provides a swiss knife of features to make your life easier when handling data.

## Covering The Key Features

Here is the list of the most popular features provided by the case class:

- A convenient toString() method that will display all it's field contents
- A compare by-field-values, not by reference
- A copy method for handling immutable data
- Pattern matching for field extraction (perhaps better avoided)

## Convenience toString Method

With case classes, the toString method is invoked when you need to evaluate any object to a string eg

```
final case class Employee(firstName: String, lastName: String, ssn: String)
class EmployeeClass(firstName: String, lastName: String, ssn: String)

val employee = Employee("john", "wick", "111-11-1111")

val employeeClass = new EmployeeClass("john", "wick", "111-11-1111")

println(employee)
```

```
// Cool
// Employee( john, wick, 111-11-1111)

println(employeeClass)
// Not cool. Prints a representation of the reference for this object
// Main$EmployeeClass$1@3f3afe78
```

## Equality By Structure Not By Reference

With case classes, you can compare objects by their structure not by reference (default when using plain classes). Here is an illustration on how it works vs plain classes:

```
final case class Employee(firstName: String, lastName: String, ssn: String)
class EmployeeClass(firstName: String, lastName: String, ssn: String)

val employee1 = Employee("john", "wick", "111-11-1111")
val employee2 = Employee("john", "wick", "111-11-1111")
val employee3 = Employee("robert", "mccall", "222-222-2222")
println(employee1 == employee2)
// true
println(employee1 == employee3)
// false

val employeeClass1 = new EmployeeClass("john", "wick", "111-11-1111")
val employeeClass2 = new EmployeeClass("john", "wick", "111-11-1111")
println(employeeClass1 == employeeClass2)
// false
```

## Built-In copy Method

Case classes are immutable by default. This means that modifying fields is not possible. However, you can copy-modify case classes. eg:

```
val employeeWithModifiedLastName = employee1.copy(firstName = "Jon")
println(employeeWithModifiedLastName)
// Employee( Jon, wick, 111-11-1111)
println(employee1)
// FYI, employee1 is not changed
```

```
// Employee( john, wick, 111-11-1111)
```

**Note:** cases classes may not be mutable by default, but they can be mutable eg:

```
final case class MutableEmployee(var firstName: String, var lastName:
    String, var ssn: String)

val mutableEmployee = MutableEmployee("gravik", "skrull", "333-33-3333")
println(mutableEmployee)
// MutableEmployee( gravik, skrull, 333-33-3333)

mutableEmployee.firstName = "talos"
println(mutableEmployee)
// MutableEmployee( talos, skrull, 333-33-3333)
```

## Using Pattern Matching For Field Extraction

You can extract the fields of a case class by using pattern matching eg:

```
employee1 match {
  case Employee(firstName, lastName, ssn) =>
    println(s"Name is $firstName $lastName and ssn is $ssn")
    // Name is john wick and ssn is 111-11-1111
}

// No need to define unused fields
employee1 match {
  case Employee(firstName, _, _) =>
    println(s"First name is $firstName")
    // First name is john
}
```

**The reason I discourage pattern match extractions for scala case classes is that correct extractions depends on the correct order of the fields. Also, adding an extra field to the case class will cause a compile error. Here is an example where the wrong order creates a bug:**

```
// wrong order!
employee1 match {
```

```

    case Employee(lastName, firstName, ssn) =>
    println(s"First name is $firstName, last name is $lastName and ssn is $ssn")
    // First name is wick, last name is john and ssn is 111-11-1111
}

final case class WideCaseClass(name: String, s1: String, s2: String, s3:
String,
                                r1: String, r2: String, r3: String, t1: String, t2: String,
t3: String)

val wideClass: WideCaseClass = WideCaseClass("a", "b", "c", "d", "e", "f", "g", "h", "i", "j")
wideClass match
case WideCaseClass(_, _, _, _, _, r1, _, _, _, _) => println(r1)
// You've got it wrong by one position! Too bad!!!
// Type safety won't help you here.

```

In this page we did the basic thing: introduced case classes and it's basic features: The toString method, the equality by structure, the copy method and my least favorite field extraction by pattern matching.

# Separation Of Concerns: Case Class Access Scope, Validation, And Derived Fields

It may not be intuitive why scope and validation are together at this moment. Let me explain why these two topics should go together.

## Basic Case Class With Validation

The most basic form of case class validation looks like the following. It works well but may not be ideal.

```
final case class EmployeeValidated(firstName: String, lastName: String, ssn: String):  
  require(firstName.nonEmpty)  
  require(lastName.nonEmpty)  
  require(ssn.nonEmpty)  
  
// Runs ok  
val employee1: EmployeeValidated = EmployeeValidated("John", "Wick", "111-11-1111")  
  
// Runtime exception thrown due to "require(lastName.nonEmpty)" validation above  
val employee2: EmployeeValidated = EmployeeValidated("Michael", "", "222-22-2222")
```

## Separating Validation And Case Classes

We typically want the case class to just be a placeholder of data. Mixing up validation, calculations and data in the same place will make case classes larger and bloated. For this purpose we want to

use companion objects eg

```
final case class Employee(firstName: String, lastName: String, ssn: String)

object Employee:
  val ssnToFullName: Map[String, (String, String)] = Map(
    "111-11-1111" -> ("John", "Wick"),
    "222-22-2222" -> ("Michael", "Bubble")
  )
  val fullNameToSsn: Map[(String, String), String] = ssnToFullName.map {case (k,v) =>
    (v,k)}

  def fromAllFields(firstName: String, lastName: String, ssn: String): Employee =
    require(firstName.nonEmpty)
    require(lastName.nonEmpty)
    require(ssn.nonEmpty)
    Employee(firstName, lastName, ssn)

  def fromSsn(ssn: String): Employee =
    require(ssn.nonEmpty)
    val (firstName: String, lastName: String) = ssnToFullName(ssn)
    Employee(firstName, lastName, ssn)

  def fromFullName(firstName: String, lastName: String): Employee =
    require(firstName.nonEmpty)
    require(lastName.nonEmpty)
    val ssn: String = fullNameToSsn((firstName, lastName))
    Employee(firstName, lastName, ssn)

// The following 3 println() should print the same data
val employee1FromBuilder = Employee.fromSsn("111-11-1111")
println(employee1FromBuilder)
val employee2FromBuilder = Employee.fromAllFields("John", "Wick", "111-11-1111")
println(employee2FromBuilder)
val employee3FromBuilder = Employee.fromFullName("John", "Wick")
println(employee3FromBuilder)
```

## Restricting Access To Case Class Constructor



Now the problem with the code above is that anyone outside this file can directly build `Employee` directly without validation. Ideally, we want to restrict this to guarantee data integrity. A way to address this issue is to add a **private** qualifier to the case class right before the parenthesis eg

```
final case class EmployeePrivate private (firstName: String, lastName: String, ssn: String)

object EmployeePrivate:
  val ssnToFullName: Map[String, (String, String)] = Map(
    "111-11-1111" -> ("John", "Wick"),
    "222-22-2222" -> ("Michael", "Bubble")
  )
  val fullNameToSsn: Map[(String, String), String] = ssnToFullName.map {case (k,v) =>
    (v,k)}

  def fromAllFields(firstName: String, lastName: String, ssn: String): EmployeePrivate =
    require(firstName.nonEmpty)
    require(lastName.nonEmpty)
    require(ssn.nonEmpty)
    EmployeePrivate(firstName, lastName, ssn)

  def fromSsn(ssn: String): EmployeePrivate =
    require(ssn.nonEmpty)
    val (firstName: String, lastName: String) = ssnToFullName(ssn)
    EmployeePrivate(firstName, lastName, ssn)

  def fromFullName(firstName: String, lastName: String): Employee =
    require(firstName.nonEmpty)
    require(lastName.nonEmpty)
    val ssn: String = fullNameToSsn((firstName, lastName))
    Employee(firstName, lastName, ssn)

// The following 3 println() should print the same data
val employeePrivate1FromBuilder = EmployeePrivate.fromSsn("111-11-1111")
println(employee1FromBuilder)
val employeePrivate2FromBuilder = EmployeePrivate.fromAllFields("John", "Wick", "111-11-1111")
println(employee2FromBuilder)
val employeePrivate3FromBuilder = EmployeePrivate.fromFullName("John", "Wick")
println(employeePrivate3FromBuilder)
```

Now you won't be able to instantiate `EmployeePrivate` directly. You will only be able to do it through the companion object or any other method in the same file.

```
// Code from a separate file
import CaseClassAccessScopeAndValidation.EmployeePrivate

// The following will create a compile error
val employee = EmployeePrivate("John", "Wick", "111-11-1111")
// method apply cannot be accessed as a member of
CaseClassAccessScopeAndValidation.EmployeePrivate.
// type from module class CaseClassAccessScopeAndValidation_AccessAttempt$.

// But the following will work just fine
val employee2 = EmployeePrivate.fromAllFields("John", "Wick", "111-11-1111")
```

**Note:** be aware that the case class convenience methods, namely `copy`, will not be available when you add the **private** qualifier eg:

```
// Code from same separate file as code sample above

val employee2 = EmployeePrivate.fromAllFields("John", "Wick", "111-11-1111")

// Following code will generate a compile error
val employee3 = employee2.copy(firstName = "hernan")
// from module class CaseClassAccessScopeAndValidation_AccessAttempt$
// method copy cannot be accessed as a member of
// (CaseClassAccessScopeAndValidation_AccessAttempt.employee :
CaseClassAccessScopeAndValidation.EmployeePrivate )
```

In that case, you will have to implement the `copy` command in a method inside the companion object. Keep in mind the companion object has access to all the private methods of the case class and vice-versa.

## Keep Derived Fields Outside The Case Class

We still want to stick to the idea of only keeping data in the case class and keep any logic outside of it. Same thing applies to derived fields (fields that are a calculation of other fields in the case class).

There are at least 2 ways we can do this. **Method 1**: Create a class that extends the case class and a trait containing the derived field. And **Method 2**: Use scala **extension**.

```
case class NonFinalEmployee(firstName: String, lastName: String, ssn: String)

// Method 1 for adding derived fields
// Use a trait and extend Employee case class with trait containing the derived field
trait FullNameDerived:
  self: NonFinalEmployee =>
    def fullNameDerived1 =
      s"$firstName $lastName"

class EmployeeData(firstName: String, lastName: String, ssn: String)
  extends NonFinalEmployee(firstName, lastName, ssn) with FullNameDerived

val employeeData: EmployeeData = new EmployeeData("John", "Wick", "111-11-1111")
println(employeeData.fullNameDerived1)

// Method 2 for adding derived fields
// Use extension method and extend Employee with derived field instead
extension (c: Employee)
  def fullNameDerived2: String =
    s"${c.firstName} ${c.lastName}"

val employee = Employee ("John", "Wick", "111-11-1111")
println(employee.fullNameDerived2)
```

**Note:** The author of this article prefers **method 2** for looking cleaner and being more extensible. Besides, we can still keep the case class **final** as well.



# Data Validation

In this chapter we will cover different aspects of data validation.

# Avoid Throwing Exceptions

## Why Throwing Exceptions Are Bad

Exceptions Are Like GOTOs To Somewhere....

Anywhere...Or Nowhere

At least GOTOs go to known places in the code. Well.. it ain't that bad. Exceptions also carry data describing the nature of the exception.

But just like GOTOs exceptions break the normal logical flow of your code.

## Exceptions Should Be Used For Exceptional Reasons

Let's be honest: Is an invalid email considered an **EXCEPTIONAL** condition in your code? I doubt it. In reality, **an invalid email is just that, an invalid email and your application should be able to handle it harmoniously**. Treating mundane conditions like an invalid email should never be an **EXCEPTIONAL** situation. Same goes for any other invalid data handled in your application. Your code shouldn't be hard to understand or maintain because your fields require validation or don't conform to the expectations of your application.

Conditions like running out of memory or not having enough CPUs to run your code successfully are exceptional conditions. And in those cases, perhaps we should just exit the program or thread instead, not just throwing an exception.

## You Lose Referential Transparency Thus Functional Purity

From [Avoid Throwing Exceptions In Medium](#)

Throwing an `Exception` breaks [referential transparency](#).

This can be demonstrated fairly easily. If `throw` was [referentially transparent](#), by definition, the

two following methods would be equivalent:

```
def foo1() = if(false) throw new Exception else 2

def getA(): Int = throw new Exception
def getAOr2(a: Int): Int = if (false) a else 2

def foo2() =
  val a = getA()
  getAOr2(a)

// foo3 and foo2 should be identical but aren't/
```

But they aren't foo1() will return 2 and foo2() will throw an exception.

If we need to factor out code (imagine we want to factor out foo1() to improve testability) the code will suddenly behave differently!

Lacking referential transparency makes refactoring/ testing and debugging more difficult. It also makes it harder to use functional libraries such as ZIO which relies on your functions being referentially transparent.

Since scala is a functional language. When we write functional code, we want our functions to be **as pure as possible**, making our code more resilient and testable. Functions that throw exceptions are not pure functions (because it may not return an output): Method getA() returns an Int, but in reality it doesn't return anything during execution. Having functional purity provides clear benefits (see <https://alvinalexander.com/scala/fp-book/benefits-of-pure-functions/>)

## It Encourages Code Duplication For Data Validation

When throwing exceptions for data validation, It's common practice to validate arguments at the beginning of a method. We like to keep things tidy eg

```
def validateEmail(string: String): Unit =
  if (string == null)
    throw new Error("Email is null")
  else
    val split = string.split("@")
```

```
    if (split.size != 2)
        throw new Error(s"Email ${string} is malformed")
    else ()

case class Email(user: String, domain: String)

def produceEmail(email: String): Email =
    validateEmail(email)
    // But we already did the same split in validateEmail(email)!
    val splitEmail: Array[String] = email.split("@")
    val emailUser = splitEmail(0)
    val emailDomain = splitEmail(1)
    Email(emailUser, emailDomain)
```

Many of us do this separate validation so we can then have a clean "happy path" code afterwards.

This replication is not specific to the example above. Validation often requires decomposing data to inspect for its validity, the same goes for data parsing. Therefore, it often occurs that clean code may mean duplicated code when throwing exceptions for data validation.

## Exceptions Are Slow

[According to this article](#) throwing a freshly created exception has been benchmarked to be more than 100 times slower than just returning an exception object. Imagine a service running 150 times slower just because most of the requests have invalid fields. It will create a chain effect where bad data lowers the performance of your application, apparently, for no good reason!



# Using Either[E, A] And MapN For Data Validation

## Quick Intro To Either[E, A] type

In addition to throwing exceptions to handle errors, scala also offers the **Either[E, A]** data type to perform the equivalent task of handling errors as well.

The email validation method we had:

```
def validateEmail(string: String): Unit =  
  if (string == null)  
    throw new Throwable("Email is null")  
  else  
    val split = string.split("@")  
    if (split.size != 2)  
      throw new Throwable(s"Email ${string} is malformed")  
    else ()
```

Can now be redefined to return an **Either[Error, Unit]**. Additionally, I introduce an, IMO, more readable way to check for null fields:

```
def validateEmailEither(string: String): Either[Throwable, Unit] = string match  
  case null =>  
    Left(Error("Email is null"))  
  case _ =>  
    val split = string.split("@")  
    if (split.size != 2)  
      Left(Error(s"Email ${string} is malformed"))  
    else Right(())
```

```

validateEmail("hernan#email.com")
// Exception in thread "main" java.lang.Error: Email hernan#email.com is malformed

// The following throws no exception
val validated: Either[Throwable, Unit] = validateEmailEither("hernan#email.com")
// We can then selectively handle the error when it occurs while staying in functional
programming paradigm

// The error handling uses the same runtime stack as "normal code" now
validated match
  case Left(f) => println(s"validation fail: ${f.getMessage}")
// validation fail: Email hernan#email.com is malformed

```

## Building Validated Case Classes For Cleaner Code

Perhaps, you can't see yet the benefit of using **Either[A, E]** yet with such small example.

Here we are using case classes to return validated fields instead of still using String types.

```

final case class SSN private(area: Int, group: Int, serial: Int)
object SSN:
  def fromString(string: String): Either[Throwable, SSN] = string match
    case null =>
      Left(Throwable("Social security is null"))
    case _ =>
      val split = string.split("-")
      if (split.size != 3)
        Left(Throwable(s"Three different sets of digits expected but ${split.size}
found"))
      else if (split(0).filter(_._isDigit).isEmpty)
        Left(Throwable(s"No digits found in area position '${string}'"))
      else if (split(1).filter(_._isDigit).isEmpty)
        Left(Throwable(s"No digits found in group position '${string}'"))
      else if (split(2).filter(_._isDigit).isEmpty)
        Left(Throwable(s"No digits found in serial position '${string}'"))
      else if (split(0).filter(!_._isDigit).nonEmpty)
        Left(Throwable(s"Invalid digit found in area position '${string}'"))
      else if (split(1).filter(!_._isDigit).nonEmpty)
        Left(Throwable(s"Invalid digit found in group position '${string}'"))

```

```

    else if (split(2).filter(!_._.isDigit).nonEmpty)
      Left(Throwable(s"Invalid digit found in serial position '${string}'"))
    else
      Right(SSN(area = split(0).toInt, group = split(1).toInt, serial = split(2).toInt))

final case class Email private(user: String, domain: String)
object Email:
  def fromString(string: String): Either[Throwable, Email] = string match
    case null =>
      Left(Throwable("Email is null"))
    case _ =>
      val split = string.split("@")
      if (split.size != 2)
        Left(Throwable(s"Email '${string}' is malformed"))
      else
        Right(Email(user = split(0), domain = split(1)))

```

## Putting It Together With For Comprehensions

Now we can validate our fields while expressing the "happy path" clearly eg

```

final case class Employee(ssn: SSN, email: Email)
val employee: Either[Throwable, Employee] = for
  email <- Email.fromString("hernan@email.com")
  ssn <- SSN.fromString("111-11-1111")
yield Employee(ssn = ssn, email = email)

val employeeBadEmail: Either[Throwable, Employee] = for
  email <- Email.fromString("hernan#email.com")
  ssn <- SSN.fromString("111-11-1111")
yield Employee(ssn = ssn, email = email)

val employeeBadSsn: Either[Throwable, Employee] = for
  email <- Email.fromString("hernan@email.com")
  ssn <- SSN.fromString("11111-1111")
yield Employee(ssn = ssn, email = email)

// Again, we handle errors using normal control data flow
employee match {

```

```

    case Right(o) => println(s"employee: Validated employee: $o")
    case Left(e) => println(s"employee: Validation error: $e")
}
// employee: Validated employee: Employee(SSN(111,11,1111),Email(hernan,email.com))

employeeBadEmail match {
    case Right(o) => println(s"employeeBadEmail: Validated employee: $o")
    case Left(e) => println(s"employeeBadEmail: Validation error: $e")
}
// employeeBadEmail: Validation error: java.lang.Throwable: Email 'hernan#email.com' is
malformed

employeeBadSsn match {
    case Right(o) => println(s"employeeBadSsn: Validated employee: $o")
    case Left(e) => println(s"employeeBadSsn: Validation error: $e")
}
// employeeBadSsn: Validation error: java.lang.Throwable: Three different sets of digits
expected but 2 found

```

## But There Is A Problem With For Comprehensions: It's Sequential Nature

I love **for comprehensions** because enables me to clearly express the happy path while handling potential errors. But not everything is perfect here. The **sequential nature of for comprehensions doesn't allow us to catch all errors** if that is what we need. eg, how do we know if both fields ssn and email are incorrect?

```

val employeeBadEmailAndBadSsn: Either[Throwable, Employee] = for
    email <- Email.fromString("hernan#email.com")
    ssn <- SSN.fromString("11111-1111")
yield Employee(ssn = ssn, email = email)

// Two fields are invalid but only one will be evaluated. Therefore, you will only be able to
collect one error
employeeBadEmailAndBadSsn match {
    case Right(o) => println(s"employeeBadEmailAndBadSsn: Validated employee: $o")
    case Left(e) => println(s"employeeBadEmailAndBadSsn: Validation error: $e")
}

```

```
// employeeBadEmailAndBadSsn: Validation error: java.lang.Throwable: Email 'hernan#email.com'
is malformed
```

The printed errors above only shows the first invalid field, the email. However, the format of the ssn is also incorrect. But due to the sequential nature of for comprehensions, all computations after the first error are cancelled and email doesn't get a change to get evaluated.

For comprehensions are useful for the common use case when you need to fail fast with no need to evaluate other bad fields.

For comprehensions will cancel the next steps when the first error is generated. This means it's not equipped to evaluate multiple fields.

## Using **Either[List[Throwable], A]** instead of **Either[Throwable, A]**

If we want to collect many validation errors, we first need a data type able to handle them. Enter **Either[List[Throwable], A]**

```
// Convenience type alias
type EitherError[A] = Either[List[Throwable], A]

// Usage
val goodText1: Either[List[Throwable], String] = Right("good text")
val badText1: Either[List[Throwable], String] = Left(List(Throwable("bad text found")))
// Or
val goodText2: EitherError[String] = Right("good text")
val badText2: EitherError[String] = Left(List(Throwable("bad text found")))
```

In the example above I provided a simplified version of **Either[List[Throwable], A]** to reduce verbosity, alias type **EitherError[A]**.

Here is how our case classes and builders would look after modifying `Either[E, A]`

```
final case class SSN2 private(area: Int, group: Int, serial: Int)
object SSN2:
  def fromString(string: String): Either[List[Throwable], SSN2] = string match
    case null =>
```

```

    Left(List(Throwable("Social security is null")))
case _ =>
    val split = string.split("-")
    if (split.size != 3)
        Left(List(Throwable(s"Three different sets of digits expected but ${split.size}
found")))
    else if (split(0).filter(_._isDigit).isEmpty)
        Left(List(Throwable(s"No digits found in area position '${string}'")))
    else if (split(1).filter(_._isDigit).isEmpty)
        Left(List(Throwable(s"No digits found in group position '${string}'")))
    else if (split(2).filter(_._isDigit).isEmpty)
        Left(List(Throwable(s"No digits found in serial position '${string}'")))
    else if (split(0).filter(!_._isDigit).nonEmpty)
        Left(List(Throwable(s"Invalid digit found in area position '${string}'")))
    else if (split(1).filter(!_._isDigit).nonEmpty)
        Left(List(Throwable(s"Invalid digit found in group position '${string}'")))
    else if (split(2).filter(!_._isDigit).nonEmpty)
        Left(List(Throwable(s"Invalid digit found in serial position '${string}'")))
    else
        Right(SSN2(area = split(0).toInt, group = split(1).toInt, serial = split(2).toInt))

final case class Email2 private(user: String, domain: String)
object Email2:
    def fromString(string: String): Either[List[Throwable], Email2] = string match
        case null =>
            Left(List(Throwable("Email is null")))
        case _ =>
            val split = string.split("@")
            if (split.size != 2)
                Left(List(Throwable(s"Email '${string}' is malformed")))
            else
                Right(Email2(user = split(0), domain = split(1)))

```

Since I don't like the ugly nested **Left(List(Throwable**, I created a [convenience method wrapper](#) called **LeftThrowable** that you can find in [this repo](#). This is how the code above will look like:

```

final case class SSN3 private(area: Int, group: Int, serial: Int)
object SSN3:
    def fromString(string: String): Either[List[Throwable], SSN3] = string match

```

```

case null =>
  LeftThrowable("Social security is null")
case _ =>
  val split = string.split("-")
  if (split.size != 3)
    LeftThrowable(s"Three different sets of digits expected but ${split.size}
found")
  else if (split(0).filter(_._isDigit).isEmpty)
    LeftThrowable(s"No digits found in area position '${string}'")
  else if (split(1).filter(_._isDigit).isEmpty)
    LeftThrowable(s"No digits found in group position '${string}'")
  else if (split(2).filter(_._isDigit).isEmpty)
    LeftThrowable(s"No digits found in serial position '${string}'")
  else if (split(0).filter(!_._isDigit).nonEmpty)
    LeftThrowable(s"Invalid digit found in area position '${string}'")
  else if (split(1).filter(!_._isDigit).nonEmpty)
    LeftThrowable(s"Invalid digit found in group position '${string}'")
  else if (split(2).filter(!_._isDigit).nonEmpty)
    LeftThrowable(s"Invalid digit found in serial position '${string}'")
  else
    Right(SSN3(area = split(0).toInt, group = split(1).toInt, serial = split(2).toInt))

final case class Email3 private(user: String, domain: String)
object Email3:
  def fromString(string: String): Either[List[Throwable], Email3] = string match
    case null =>
      LeftThrowable("Email is null")
    case _ =>
      val split = string.split("@")
      if (split.size != 2)
        LeftThrowable(s"Email '${string}' is malformed")
      else
        Right(Email3(user = split(0), domain = split(1)))

```

## Introducing mapN (AKA Applicatives)

Applicatives are helpful when all fields need to be evaluated. Unfortunately, this capability is not included in the scala standard library. [Libraries like cats provide it](#). I am using a [homegrown version of applicatives you can use in this link](#) in case you don't want to deal with the somehow

heavy cats library.

Here is how it would look when putting together these validated fields into a case class. Similar to the example above:

```
final case class Employee3 private(ssn: SSN3, email: Email3)

val employeeGood: EitherError[Employee3] = Applicative.mapN(
  Email3.fromString("hernan@gmail.com"),
  SSN3.fromString("111-11-1111")
)((email, ssn) => Employee3(email = email, ssn = ssn))
println(employeeGood)
// Right(Employee3(SSN2(111,11,1111),Email2(hernan@gmail.com)))

val employeeBadEmail: EitherError[Employee3] = Applicative.mapN(
  Email3.fromString("hernan#gmail.com"),
  SSN3.fromString("111-11-1111")
)((email, ssn) => Employee3(email = email, ssn = ssn))
println(employeeBadEmail)
// Left(List(java.lang.Throwable: Email 'hernan#gmail.com' is malformed))

val employeeBadEmailAndSsn: EitherError[Employee3] = Applicative.mapN(
  Email3.fromString("hernan#gmail.com"),
  SSN3.fromString("111111111")
)((email, ssn) => Employee3(email = email, ssn = ssn))
println(employeeBadEmailAndSsn)
// Left(List(java.lang.Throwable: Email 'hernan#gmail.com' is malformed, java.lang.Throwable:
// Three different sets of digits expected but 1 found))
```

So far, we've learned two useful techniques for evaluating and validating fields, for comprehensions for the simple stop-on-first-fail and mapN when catching multiple errors is beneficial.



# Use Union Types Instead Of Either[E, A] For A More Efficient Validation?

The reason I put together this article is because I couldn't help to notice how similar Either[E, A] is to union type E | A.

And then I asked myself, could we use union types instead of Either[E, A] with a goal of getting a leaner and faster equivalent functionality?

Enough of clickbait, the answer is yes, but not by a lot, **we are talking about 4% improvement in runtime and memory usage**. Now you can chose to continue reading this blog with the right expectations.

Let's revisit how we would do validation using Either[E, A] and for comprehension. Here is a piece of code from [this blog's repo](#)

```
val employeeGood = for
  email <- UsingEither.EmailBuilder.fromString("x@dd.com")
  ssn <- UsingEither.SsnBuilder.fromString("111-11-1111")
yield Employee(email=email, ssn=ssn)
println(s"employeeGood $employeeGood")
```

I would like to clarify that in the example above we are generating Either[E, A] objects that are only used for validation and eventually thrown away once their value is extracted.

```
val employeeGood = for
  // The Either[E, A] object from UsingUnionType.EmailBuilder.fromString
  // is thrown away immediately
  email <- UsingEither.EmailBuilder.fromString("x@dd.com")
```

```
// The Either[E, A] object from UsingUnionType.SsnBuilder.fromString
// is also thrown away immediately
ssn <- UsingEither.SsnBuilder.fromString("111-11-1111")
yield Employee(email=email, ssn=ssn)
println(s"employeeGood $employeeGood")
```

If we do this enough time in our code, we maybe be giving our garbage collector a lot of work just to cleanup these intermediate Either[E, A] wrappers!

Thus, here is the question I will try to answer in this blog:

## Can we use for comprehensions for validating data without using a wrapper object like Either[E, A]?

I am thinking that the best candidate for this solution would be the [union types introduced in scala 3](#):

```
object UsingUnionType:
  type unionWithErrorList[A] = List[Throwable] | A
  object SsnBuilder:
    def fromString(string: String): unionWithErrorList[SSN] = {
      string match
        case null =>
          List(Throwable("Social security is null"))
        case _ =>
          val split = string.split("-")
          if (split.size != 3)
            List(Throwable(s"Three different sets of digits expected but ${split.size} found"))
          else if (split(0).filter(_._isDigit).isEmpty)
            List(Throwable(s"No digits found in area position '${string}'"))
          else if (split(1).filter(_._isDigit).isEmpty)
            List(Throwable(s"No digits found in group position '${string}'"))
          else if (split(2).filter(_._isDigit).isEmpty)
            List(Throwable(s"No digits found in serial position '${string}'"))
          else if (split(0).filter(!_._isDigit).nonEmpty)
            List(Throwable(s"Invalid digit found in area position '${string}'"))
          else if (split(1).filter(!_._isDigit).nonEmpty)
```

```
List(Throwable(s"Invalid digit found in group position '${string}'"))
    else if (split(2).filter(!_._isDigit).nonEmpty)
        List(Throwable(s"Invalid digit found in serial position
        '${string}'"))
    else
        SSN(area = split(0).toInt, group = split(1).toInt, serial =
        split(2).toInt)
    }

object EmailBuilder:
    def fromString(string: String): unionWithErrorList[Email] = string match
        case null =>
            List(Throwable("Email is null"))
        case _ =>
            val split = string.split("@")
            if (split.size != 2)
                List(Throwable(s"Email '${string}' is malformed"))
            else
                Email(user = split(0), domain = split(1))
```

But will the following work?

```
val employeeGood = for
    email <- UsingUnionType.EmailBuilder.fromString("x@dd.com")
    ssn <- UsingUnionType.SsnBuilder.fromString("111-11-1111")
yield Employee(email=email, ssn=ssn)
println(s"employeeGood $employeeGood")
```

**Nope! You will get compile errors! There Union types don't have their own flatmap!**

**value flatMap is not a member of  
datavalidation.UnionTypeVsEither.UsingUnionType.unionWithErrorList[  
datavalidation.UnionTypeVsEither.Email  
, but could be made available as an extension method.**

Well, then let's add a flatmap and a map to the union type to make for comprehensions work!

```
extension[B] (or: UsingUnionType.unionWithErrorList[B])
    def flatMap[B1](f: B => List[Throwable] | B1): UsingUnionType.unionWithErrorList[B1] = or
```

```
match
  [case e: List[Throwable] => e
  [case o: B => f(o)

def map[B1](f: B => B1): UsingUnionType.unionWithErrorList[B1] = or match
  [case e: List[Throwable] => or.asInstanceOf[UsingUnionType.unionWithErrorList[B1]]
  case o: B => f(o).asInstanceOf[UsingUnionType.unionWithErrorList[B1]]
```

It works!

```
val employeeGood = for
  email <- UsingUnionType.EmailBuilder.fromString("x@dd.com")
  ssn <- UsingUnionType.SsnBuilder.fromString("111-11-1111")
yield Employee(email=email, ssn=ssn)
println(s"employeeGood $employeeGood")
// prints employeeGood Employee(Email(x, dd.com), SSN(111,11,1111))
```

### But does it consume less memory? Yes, a little bit

I setup 2 benchmarks and setup the following jvm settings to collect memory usage. Make sure you are using java 11 or newer and java 8 will not recognize

**"-XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC"**

```
-XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC -Xmx32g
```

The idea is that we accumulate memory usage and we track memory usage through each incremental step. Be aware this benchmark eats up a lot of memory **since we are turning off the garbage collector.**

More details about this benchmark code can be seen at the [repo file for this blog](#). Feel free to download them and run them.

```
object BenchmarkEither extends App:
  UnionTypeVsEither.employeeGenerateValidateWithEither(4_000_000)

  val t1 = System.currentTimeMillis()
  val m1 = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()

  val list = UnionTypeVsEither.employeeGenerateValidateWithEither(4_000_000)
```

```

val t2 = System.currentTimeMillis()
val m2 = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()

println(s"employeeGenerateValidateWithEither: mem: gb: ${(m2 - m1) / 1_000_000_000.0}")
println(s"employeeGenerateValidateWithEither: ms: ${(t2 - t1)}")

object BenchmarkUnionType extends App:
  UnionTypeVsEither.employeeGenerateValidateWithUnionTypes(4_000_000)

  val t1 = System.currentTimeMillis()
  val m1 = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()

  val list = UnionTypeVsEither.employeeGenerateValidateWithUnionTypes(4_000_000)
  val t2 = System.currentTimeMillis()
  val m2 = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()

  println(s"employeeGenerateValidateWithUnionTypes: mem: gb: ${(m2-m1)/1_000_000_000.0}")
  println(s"employeeGenerateValidateWithUnionTypes: ms: ${(t2 - t1)}")

object NoWrappers extends App:
  UnionTypeVsEither.employeeGenerateNoValidationWrappers(4_000_000)

  val t1 = System.currentTimeMillis()
  val m1 = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()

  val list = UnionTypeVsEither.employeeGenerateNoValidationWrappers(4_000_000)
  val t2 = System.currentTimeMillis()
  val m2 = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()

  println(f"employeeGenerateNoValidationWrappers: mem: gb: ${(m2 - m1) /
1_000_000_000.0}% 2f")
  println(s"employeeGenerateNoValidationWrappers: ms: ${(t2 - t1)}")

```

Both benchmarks, the one for union types and the one for Either[E, A] yielded the following results rather consistently:

```

union types
employeeGenerateValidateWithUnionTypes: mem: gb: 4.148166656
employeeGenerateValidateWithUnionTypes: ms: 2545

```

```
either
employeeGenerateValidateWithEither: mem: gb: 4.273995776
employeeGenerateValidateWithEither: ms: 2639
```

One can speculate the difference should deepen the longer the for-comprehension gets (meaning more Either types being garbage collected).

## Will memory efficiency help if we use opaque union types? Haven't seen any improvements

Let's try something else, **opaque types for union types**. Scala 3 introduced opaque types. The claim of opaque types is that they "provide type abstraction without any **overhead**". In Scala 2, a similar result could be achieved with [value classes](https://docs.scala-lang.org/overview/implicit-classes.html)." (from [docs.scala-lang.org](https://docs.scala-lang.org/overview/opaque-types.html))

Does that mean that we could use opaque types for union types? Let's try it!

```
object UsingOpaqueUnionType:
  opaque type OpaqueUnionWithErrorList[A] = List[Throwable] | A

  object SsnBuilder:
    def fromString(string: String): OpaqueUnionWithErrorList[SSN] = {
      string match
        case null =>
          List(Throwables("Social security is null"))
        case _ =>
          val split = string.split("-")
          if (split.size != 3)
            List(Throwables(s"Three different sets of digits expected but ${split.size}
found"))
          .....
          else if (split(2).filter(!_._.isDigit).nonEmpty)
            List(Throwables(s"Invalid digit found in serial position
'${string}'"))
          else
            SSN(area = split(0).toInt, group = split(1).toInt, serial =
split(2).toInt)
    }

  object EmailBuilder:
```

```

def fromString(string: String): OpaqueUnionWithErrorList[Email] = string match
  case null =>
    List(Throwable("Email is null"))
  case _ =>
    val split = string.split("@")
    if (split.size != 2)
      List(Throwable(s"Email '${string}' is malformed"))
    else
      Email(user = split(0), domain = split(1))

extension[B] (or: OpaqueUnionWithErrorList[B])
  def flatMap[B1](f: B => List[Throwable] | B1): OpaqueUnionWithErrorList[B1] = or
match
  case e: List[Throwable] => e.asInstanceOf[OpaqueUnionWithErrorList[B1]]
  case o: B => f(o).asInstanceOf[OpaqueUnionWithErrorList[B1]]

def map[B1](f: B => B1): OpaqueUnionWithErrorList[B1] =
  or match
    case e: List[Throwable] => or.asInstanceOf[OpaqueUnionWithErrorList[B1]]
    case o: B => f(o).asInstanceOf[OpaqueUnionWithErrorList[B1]]

```

Be warned your IDE may not like parsing the code above, you will likely see strange errors reporting as we are not writing standard scala.

After benchmarking opaque union types vs Either vs just union types we get the following results more or less consistently:

#### union types

```

employeeGenerateValidateWithUnionTypes: mem: gb: 4.148166656
employeeGenerateValidateWithUnionTypes: ms: 2545

```

#### either

```

employeeGenerateValidateWithEither: mem: gb: 4.273995776
employeeGenerateValidateWithEither: ms: 2639

```

#### opaque union type

```

employeeGenerateValidateWithOpaqueUnionTypes: mem: gb: 4.15
employeeGenerateValidateWithOpaqueUnionTypes: ms: 2493

```

## The conclusion so far:

- I observed an improvement of about 4% on performance and memory savings by using union types or opaque types instead of `Either[E, A]`
- If we do the math on object creation (Either objects that get trashed right away) the math doesn't add up. There are clearly scala specific memory optimizations (outside the garbage collector) that make the use of `Either[E, A]` very efficient.

## Is it justifiable to use union types instead of Either?

The answer would be: **not today.**

- Clearly, this is not a commonly accepted way of evaluating data today
- There maybe unexpected side effects when writing code like this
- IntelliJ IDE doesn't like it. You will see lots of red in the code
- I can't think of many places where it's worth the risk using Union types for a 4% improvement in performance and memory efficiency.

[Access to repo for this blog](#)