

ゼロから始めるScala体験会

2025年4月24日 (木)

株式会社ネクストビート

本日のゴールと対象者

- ターゲット:
 - Scalaに興味がある方
 - プログラミング経験はあるが、Scalaは初心者レベル
- ゴール: 1時間で...
 - Scalaの魅力（特に **型安全性, 表現力, 関数型** の考え方）の一端を体験
 - `opaque type` や `Either`、関数合成を使って **堅牢なコード** を書くメリットを体感してもらう

ツールと進め方

- ツール: **Scastie** (ブラウザで動く Scala環境)
 - URL: <https://scastie.scala-lang.org/>
 - Scala 3モードで利用します
- 形式:
 - 説明とデモが中心
 - ときどき、Scastieで簡単なコードを試す 演習時間 を設けます

時間配分（目安）

- Scala紹介と導入（10分）
- 値オブジェクト: `case class` vs `opaque type`（10分）
- `Either` と関数合成による安全な生成（15分）
- 値オブジェクトを使った型安全な処理（15分）
- まとめ・質疑応答（10分）

※Scastieの使い方とScalaの基本的な構文は補足資料をご覧ください

Scala紹介と導入

Scalaとは？(1/2)

- JVM (Java Virtual Machine) 上で動作する **静的型付け** 言語
- オブジェクト指向 + 関数型プログラミング の融合
 - 両方の良いところを活用できる！
- 強力な **型システム**
 - コンパイル時にエラーを発見しやすく、堅牢なコードを書ける
 - 今日、このメリットを体験します！
- **型推論**
 - 型を明記しなくても、コンパイラが型を推測してくれる
 - コードが簡潔に

Scalaとは？(2/2)

- Javaとの高い **相互運用性**
 - Javaのライブラリをそのまま利用できる
- **今日の焦点:**
 - Scalaの **型安全性**
 - 関数型プログラミング の考え方の一部
- **体験するメリット:**
 - **堅牢性:** バグりにくいコード
 - **表現力:** やりたいことを明確にコードで表現できる
 - **組み立てやすさ:** 部品を組み合わせて安全なプログラムを作る

Hello, Scala! (Scastieで試そう！)

まずは定番の "Hello, World!"

```
// このコードをScastieに貼り付けて実行してみましょう!
println("Hello, World!")
```

ちょっとScalaらしいコード

フィボナッチ数列を生成する例 (雰囲気を掴むだけでOK)

```
val fibs: LazyList[BigInt] = {  
    BigInt(0) #::: BigInt(1) #::: fibs.zip(fibs.tail).map { case (a, b) =>  
        a + b  
    }  
}  
println(fibs.take(10).toList)  
// 出力: List(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
```

- `LazyList`: 必要になるまで計算しないリスト
- `.map`: 関数型らしいデータ変換

値オブジェクト: case class vs opaque type

「型」でドメイン概念を表す

問題: メールアドレスをただの `String` で扱うと？

```
def sendEmail(address: String, subject: String): Unit = ???  
  
// こんな呼び出し方ができるてしまう...  
sendEmail("これはメールアドレスじゃない", "テスト")  
sendEmail("", "件名") // 空文字列も渡せる  
sendEmail("user@domain.com", "user-name") // 引数の順番ミスにも気づきにくい
```

課題:

- 不正な値（空文字列、形式が違う文字列）を代入できてしまう
- 引数が `String` だと、何を表す文字列なのか分かりにくい

解決策: メールアドレス専用の 型 を作る！ => 値オブジェクト

方法1: `case class` (シンプル!)

`case class` を使うと、簡単に独自の型を定義できます。

```
// Emailという名前の case class を定義
case class Email(value: String)
// Email型として値を保持
val email: Email = Email("test@example.com")
println(email)          // 出力: Email(test@example.com)
println(email.value)    // 出力: test@example.com
// 型が違うので、Stringを直接代入できない! (コンパイルエラー)
// val wrong: Email = "test@example.com"
```

- メリット: 簡単、シンプル
- デメリット: 実行時に `Email` オブジェクト生成の一バーヘッド

方法2: opaque type (Scala 3 の機能)

実行時オーバーヘッドなしで型安全性を実現！

```
object Email {  
    // opaque type を定義（実体は String）  
    opaque type Email = String  
    // ファクトリメソッド（インスタンス生成用、今は仮実装）  
    def from(value: String): Email = value  
    // 拡張メソッド（内部の値へのアクセス用）  
    extension (self: Email) def value: String = self  
}  
import Email.*  
val e: Email = Email.from("test@example.com")  
println(e.value) // 出力: test@example.com  
// 直接 String は代入できない！(コンパイルエラー)  
// val eBad: Email = "test@example.com"
```

opaque type の特徴と比較

- `opaque type Email = String`
 - `Email` 型を作るが、コンパイル後は `String` として扱われる
- メリット:
 - 実行時オーバーヘッドゼロ！（オブジェクト生成コストがない）
 - コンパイル時に `String` と `Email` の混同を防げる（型安全性）
- デメリット:
 - `case class` より少し記述が増える

今回は `opaque type` を使って進めます！

Either と関数合成による安全な生成

安全な値オブジェクト生成へ

目標: `opaque type Email` を、 不正な文字列 (例: `""`, `"not-email"`) からは 生成できないようにしたい。

アイデア:

1. `Email` を生成する前に、 入力文字列を バリデーション する
2. バリデーションの結果を `Either` で返すようとする
 - 成功: `Right(検証済み文字列)`
 - 失敗: `Left(エラーメッセージ)`
3. 成功時に `opaque type Email` として値を返す 安全なファクトリメソッド を作る。

バリデーション関数の準備

今回はシンプルなチェック関数を自作します。

(実務ではバリデーションライブラリを使うことが多いです)

- シグネチャ: `String => Either[String, String]`
 - 入力: `String`
 - 出力: `Either[エラーメッセージ, 検証済み文字列]`
- 作るチェック関数:
 1. `nonEmpty`: 空文字列 (" " や " ") でないか？
 2. `containsAtMark`: @ マークを含んでいるか？

バリデーション関数 (1/2)

Email オブジェクト内に private で定義します。

```
// object Email { ... の中に追加

// private: Email オブジェクトの中からしか呼べない
private def nonEmpty(value: String): Either[String, String] = {
    // value.trim で前後の空白を除去してから isEmpty でチェック
    if (value.trim.isEmpty) {
        Left("Email cannot be empty") // 失敗 -> Left
    } else {
        Right(value) // 成功 -> Right
    }
}

// }
```

バリデーション関数(コード 2/2)

```
// object Email { ... の中に追加

private def containsAtMark(value: String): Either[String, String] = {
  if (value.contains('@')) {
    Right(value) // 成功 -> Right
  } else {
    Left("Email must contain '@'") // 失敗 -> Left
  }
}

// }
```

バリデーションの「合成」

`nonEmpty` と `containsAtMark` の両方を順番に適用したい。

- `nonEmpty` で失敗したら、処理を中断して `Left` を返したい。
- `nonEmpty` が成功したら、結果を使って `containsAtMark` を実行したい。

ここで `Either` の関数合成が役立ちます！

とくに `for` 式 (for comprehension) を使うと宣言的に書けます。

バリデーションの合成: `for` 式

`Either` に対する `for` 式は、途中で `Left` になったら、その後ろの処理は実行されずに、その `Left` が最終結果となります。

```
// object Email { ... の中に追加

// 複数のバリデーションを合成する関数
private def validate(input: String): Either[String, String] = {
  for {
    s1 <- nonEmpty(input)          // 1. nonEmpty を実行。失敗(Left)ならここで終了
    s2 <- containsAtMark(s1)      // 2. s1が成功(Right)の場合のみ実行。失敗ならここで終了
    // 他のチェックもここに追加できる
    // s3 <- checkDomain(s2)
  } yield s2 // 3. 全てのチェックが成功(Right)した場合、最後の結果(s2)が Right で包まれて返る
}

// }
```

安全なファクトリメソッド `from` の実装

仮実装した `from` メソッドを、`validate` を使うように修正します。

```
// object Email { ... の from を修正

// 安全なファクトリメソッド
def from(value: String): Either[String, Email] = {
  validate(value) match {
    // validate が成功(Right)した場合のみ Email 型にして返す
    case Right(validatedValue) => Right(validatedValue) // String を Email 型にキャスト (OpaqueなのでOK)
    // validate が失敗(Left)した場合は、そのまま Left を返す
    case Left(error)           => Left(error)
  }
  // Note: .map(identity) や .map(v => v) と書いても同じ意味になります
  // validate(value).map(identity)
}

// }
```

全体のコード (Email オブジェクト)

```
object Email {  
    opaque type Email = String  
  
    // --- バリデーション関数 ---  
    private def nonEmpty(value: String): Either[String, String] =  
        if (value.trim.isEmpty) Left("Email cannot be empty") else Right(value)  
  
    private def containsAtMark(value: String): Either[String, String] =  
        if (value.contains('@')) Right(value) else Left("Email must contain '@'")  
  
    // --- バリデーション合成 ---  
    private def validate(input: String): Either[String, String] =  
        for {  
            s1 <- nonEmpty(input)  
            s2 <- containsAtMark(s1)  
        } yield s2  
  
    // --- 安全なファクトリメソッド ---  
    def from(value: String): Either[String, Email] =  
        validate(value).map(identity) // .map(identity) は成功時のみ値を適用  
  
    // --- 拡張メソッド ---  
    extension (self: Email) def value: String = self  
}
```

試してみよう！(デモ/演習 in Scastie)

安全なファクトリメソッド `Email.from` を使ってみましょう。

```
import Email.* // Email オブジェクトの中身を使えるようにする

val validEmailResult    = Email.from("test@example.com")
val emptyEmailResult   = Email.from("")
val noAtMarkResult     = Email.from("testexample.com")
val emptyThenNoAtMark = Email.from(" ") // 空白のみ

println(s"Valid: $validEmailResult")
println(s"Empty: $emptyEmailResult")
println(s"No '@': $noAtMarkResult")
println(s"Empty then No '@': $emptyThenNoAtMark")
```

実行結果

```
Valid: Right(test@example.com)
Empty: Left>Email cannot be empty)
No '@': Left>Email must contain '@')
Empty then No '@': Left>Email cannot be empty) // 最初の nonEmpty で失敗
```

- 不正な値からは `Left` が返るようになりました！

値オブジェクトを使った型安全な処理

型安全な関数の定義

`Email.from` で安全に生成された `Email` 型だけを受け取る関数を定義してみましょう。

```
import Email.*  
def processEmail(email: Email): Unit = {  
    // Email.from によってバリデーション済みであることが保証されている!  
    println(s"Processing valid email: ${email.value}") // .value で String 値を取得  
}
```

- メリット:
 - コンパイラが `Email` 型以外の値(例: `String`)を渡そうとするとエラーにしてくれる。

型安全性の確認（1/2）

コンパイラが間違いを防いでくれる様子を見てみましょう。

```
import Email.*  
def processEmail(email: Email): Unit = {  
    println(s"Processing valid email: ${email.value}")  
}  
val validEmailResult = Email.from("test@example.com")  
val invalidEmailResult = Email.from("invalid-email")  
// --- これはコンパイルエラーになる! ---  
// processEmail("test@example.com") // String は渡せない!  
// --- Email.from の結果 (Either) を使う必要がある ---  
println("Handling results...")
```

- `processEmail` に直接 `String` を渡せない！

型安全性の確認 (2/2)

`Email.from` の結果は `Either[String, Email]` なので、`match` を使って安全に処理を分岐します。

```
// 前のスライドの続き
validEmailResult match {
  case Right(emailInstance) => processEmail(emailInstance)
  case Left(error)          => println(s"Match Failed: $error")
}
// 出力: Processing valid email: test@example.com
invalidEmailResult match {
  case Right(emailInstance) => processEmail(emailInstance)
  // Left(error) の場合は失敗処理
  case Left(error)          => println(s"Match Failed: $error")
}
// 出力: Match Failed: Email must contain '@'
```

Either の結果を安全に使う (fold)

`match` の代わりに `fold` メソッドもよく使われます。

```
// 前のスライドの続き
println("\nUsing fold:")
validEmailResult.fold(
  error => println(s"Fold Failed: $error"), // 第1引数: Left の場合の処理
  email => processEmail(email)           // 第2引数: Right の場合の処理
)
// 出力: Processing valid email: test@example.com
invalidEmailResult.fold(
  error => println(s"Fold Failed: $error"),
  email => processEmail(email)
)
// 出力: Fold Failed: Email must contain '@'
```

まとめ・質疑応答

まとめ: 体験したこと ✨

- 値オブジェクト: ドメインの概念（例: Email）を型で表現する手法。
 - `case class` (シンプル) vs `opaque type` (実行時コストゼロ！)
- `Either[L, R]`: 成功(**Right**) または 失敗(**Left**) を型レベルで表現。
- 関数合成 (`for` 式): 複数の処理（バリデーションなど）を宣言的かつ安全に組み合わせる。途中で失敗したら中断できる！
- 型安全性:
 - `opaque type` で `String` と `Email` の混同をコンパイル時に防ぐ。
 - 関数の引数型 (`def f(e: Email)`) で意図を明確にし、不正な利用をコンパイル時に防ぐ。
 - `Either` の `match` や `fold` で、成功/失敗の処理漏れをコンパイル時に防ぐ。

Scalaの魅力 再確認

今日体験した機能はScalaの魅力のほんの一部です。

- **堅牢性:** 型システムや `Either` などが、バグを未然に防ぎ、信頼性の高いコードを書く助けになる。
- **表現力:** `opaque type` や `for` 式のように、プログラマの意図をコードで明確に表現しやすい。
- **組み立てやすさ:** 小さな関数（バリデーション）や型（`Either`）を組み合わせて、安全で複雑な処理を構築できる（関数型プログラミングの考え方）。

皆さんのScalaへの興味を深めるきっかけになれば幸いです！

(任意) 次のステップ

- **Scala公式サイト:**
 - Scala 3 Book: <https://docs.scala-lang.org/scala3/book/introduction.html>
 - Tour of Scala: <https://docs.scala-lang.org/tour/tour-of-scala.html>
- オンライン学習:
 - Scala Exercises: <https://www.scala-exercises.org/>
- **Scastie**で色々試してみる！

質疑応答

ご清聴ありがとうございました！