# Error Handling in Scala 3

## Modern Functional Error Management

Said BOUDJELDA

Senior Software Engineer @SCIAM
Email : mohamed-said.boudjelda@intervenants.efrei.net
Follow me on GitHub @bmscomp

Course, May 2025

# Overview

Modern error handling in Scala 3 emphasizes:

- **Type Safety**: Errors visible in signatures
- **Union Types**: Native error representation
- **Enums**: Structured error hierarchies
- **Composition**: Functional error chaining
- **Performance**: Zero-cost abstractions

*From exceptions to functional types*

# Traditional vs Modern

```scala
// Old way: Exceptions (not type-safe)
def divide(a: Int, b: Int): Int =
  if b == 0 then throw ArithmeticException("Division
      by zero")
  else a / b

// Scala 3 way: Union types (type-safe)
type Result[T] = T | String

def safeDivide(a: Int, b: Int): Result[Int] =
  if b == 0 then "Division by zero" else a / b

// Usage
safeDivide(10, 2) match
  case result: Int => println(s"Success: $result")
  case error: String => println(s"Error: $error")
```

# Union Types - The Modern Way

```scala
// Simple union types for errors
type ParseResult = Int | String

def parseNumber(s: String): ParseResult =
  try s.toInt
  catch case _: NumberFormatException => s"Invalid: $s
      "

// Pattern matching is seamless
def processNumber(input: String) = parseNumber(input)
    match
  case num: Int => num * 2
  case error: String => 0

// Chaining operations
def calculate(x: String, y: String): ParseResult =
  (parseNumber(x), parseNumber(y)) match
    case (a: Int, b: Int) => a + b
    case (error: String, _) => error
```

# Enums for Error Hierarchies

```scala
// Structured errors with enums
enum AppError(val message: String):
  case ValidationError(msg: String) extends AppError(
      msg)
  case NetworkError(msg: String) extends AppError(msg)
  case ParseError(msg: String) extends AppError(msg)

  def isRetryable: Boolean = this match
    case NetworkError(_) => true
    case _ => false

type ApiResult[T] = T | AppError

def validateAge(age: Int): ApiResult[Int] =
  if age >= 0 && age <= 150 then age
  else AppError.ValidationError(s"Invalid age: $age")

def parseAge(s: String): ApiResult[Int] =
  try validateAge(s.toInt)
```

# Extension Methods for Fluent APIs

```scala
// Extensions for composable error handling
extension [T](value: T)
  def validateThat(condition: T => Boolean,
                   error: String): T | String =
    if condition(value) then value else error

extension [T](result: T | String)
  def andThen[U](f: T => U | String): U | String =
    result match
    case value: T => f(value)
    case error: String => error

// Fluent usage
def processInput(input: String): String | Int =
  input.validateThat(_.nonEmpty, "Input required")
    .andThen(_.validateThat(_.forall(_.isDigit), "Not
       a number"))
    .andThen(_.toInt)
```

# Option - Still Useful

```scala
// Option for null safety
def findUser(id: Int): Option[String] =
  if id > 0 then Some(s"User$id") else None

// Functional operations
val result = findUser(1)
  .map(_.toUpperCase)
  .filter(_.startsWith("USER"))
  .getOrElse("Unknown")

// For-comprehension
def getUserData(id: Int, role: String) = for
  user <- findUser(id)
  validRole <- if role.nonEmpty then Some(role) else
    None
yield s"$user has role $validRole"

println(getUserData(1, "admin")) // Some(USER1 has
  role admin)
```

# Either - When You Need Details

```scala
// Either for rich error information
def validateUser(name: String, age: Int): Either[
    String, String] =
  for
    validName <- if name.nonEmpty then Right(name)
                 else Left("Name required")
    validAge <- if age > 0 then Right(age)
                else Left("Age must be positive")
  yield s"$validName is $validAge years old"

// Error accumulation (custom approach)
def combineValidations[A, B](
  va: Either[String, A],
  vb: Either[String, B]
): Either[List[String], (A, B)] = (va, vb) match
  case (Right(a), Right(b)) => Right((a, b))
  case (Left(e1), Left(e2)) => Left(List(e1, e2))
  case (Left(e), _) => Left(List(e))
  case (_, Left(e)) => Left(List(e))
```

# Resource Management

```scala
import scala.util.Using

// Automatic resource cleanup
def readConfig(file: String): String | AppError =
  Using(scala.io.Source.fromFile(file)) { source =>
    source.mkString
  }.toEither match
    case Right(content) => content
    case Left(ex) => AppError.NetworkError(ex.
        getMessage)

// Multiple resources
def copyFile(from: String, to: String): Unit |
    AppError =
  Using.Manager { use =>
    val source = use(scala.io.Source.fromFile(from))
    val writer = use(java.io.PrintWriter(to))
    source.getLines().foreach(writer.println)
  }.toEither match
```

# Given/Using for Error Context

```scala
// Context for error handling
trait ErrorReporter:
  def report(error: String): Unit

given consoleReporter: ErrorReporter with
  def report(error: String): Unit = println(s"[ERROR]
      $error")

def validateWithContext[T](value: T,
                           condition: T => Boolean,
                           errorMsg: String)
                           (using reporter:
                              ErrorReporter): Option[T]
                               =
  if condition(value) then Some(value)
  else
    reporter.report(errorMsg)
    None
```

# Async Error Handling

```scala
import scala.concurrent.Future

// Future with union types
def fetchUser(id: Int): Future[String | AppError] =
  Future {
    if id > 0 then s"User$id"
    else AppError.ValidationError("Invalid ID")
  }

// Combining async operations
def getUserProfile(id: Int): Future[String | AppError]
    =
  fetchUser(id).map {
    case user: String => s"Profile for $user"
    case error: AppError => error
  }

// Error recovery
def fetchUserWithRetry(id: Int): Future[String |
```

# Pattern Matching Improvements

```scala
// Scala 3 pattern matching enhancements
def handleResult(result: String | Int | AppError):
    String = result match
  case s: String => s"Text: $s"
  case n: Int => s"Number: $n"
  case AppError.ValidationError(msg) => s"Validation:
      $msg"
  case AppError.NetworkError(msg) => s"Network: $msg"
  case AppError.ParseError(msg) => s"Parse: $msg"

// Guard patterns with union types
def categorizeValue(value: String | Int): String =
    value match
  case s: String if s.isEmpty => "Empty string"
  case s: String if s.length > 10 => "Long string"
  case s: String => "Short string"
  case n: Int if n < 0 => "Negative number"
  case n: Int => "Positive number"
```

# Error Boundaries

```scala
// Error boundary pattern for isolation
def withErrorBoundary[T](operation: () => T): T |
    AppError =
  try operation()
  catch
    case ex: IllegalArgumentException =>
      AppError.ValidationError(ex.getMessage)
    case ex: NumberFormatException =>
      AppError.ParseError(ex.getMessage)
    case ex: Exception =>
      AppError.NetworkError(ex.getMessage)

// Usage isolates errors
def riskyOperation(): String = throw RuntimeException(
    "Boom!")

val result = withErrorBoundary(() => riskyOperation())
result match
  case value: String => println(s"Success: $value")
```

# Custom Validation DSL

```scala
// Build a validation DSL with Scala 3
case class ValidationResult[T](value: T, errors: List[
    String])

extension [T](value: T)
  def validate: ValidationResult[T] = ValidationResult
      (value, Nil)

extension [T](vr: ValidationResult[T])
  def check(condition: T => Boolean, error: String):
      ValidationResult[T] =
    if condition(vr.value) then vr
    else vr.copy(errors = vr.errors :+ error)

  def result: T | List[String] =
    if vr.errors.isEmpty then vr.value else vr.errors

// Usage
def validateUser(name: String, age: Int) =
```

# Testing Error Scenarios

```scala
// Simple testing approach
def testValidation(): Unit =
  // Test success
  val success = validateAge(25)
  assert(success == 25)

  // Test errors
  validateAge(-5) match
    case error: AppError.ValidationError =>
      assert(error.message.contains("Invalid"))
    case _ => sys.error("Expected validation error")

  // Test parsing
  parseAge("abc") match
    case AppError.ParseError(msg) => assert(msg.
        contains("number"))
    case _ => sys.error("Expected parse error")

// Property-based testing
```

# Performance Considerations

```scala
// Union types are zero-cost
type FastResult = String | Int // No boxing!

// Avoid Option allocation in hot paths
def fastParse(s: String): FastResult =
  var i = 0
  var result = 0
  while i < s.length do
    val c = s.charAt(i)
    if c.isDigit then
      result = result * 10 + (c - '0')
      i += 1
    else
      return s"Invalid char at $i: $c"
  result

// Value classes for domain types
opaque type UserId = Int
object UserId:
```

# Real-World API Example

```scala
// Complete API with modern error handling
case class User(name: String, age: Int, email: String)

def createUser(data: Map[String, String]): User | List
    [String] =
  val validations = List(
    data.get("name").toRight("Missing name")
      .flatMap(n => if n.nonEmpty then Right(n) else
          Left("Empty name")),
    data.get("age").toRight("Missing age")
      .flatMap(a => try Right(a.toInt) catch case _ =>
          Left("Invalid age")),
    data.get("email").toRight("Missing email")
      .flatMap(e => if e.contains("@") then Right(e)
          else Left("Invalid email"))
  )

  val errors = validations.collect { case Left(err) =>
      err }
```

# Migration Strategy

```scala
// Step 1: Wrap existing exceptions
def legacyCode(): String = throw RuntimeException("Old
    code")

def wrapped(): String | String =
  try legacyCode() catch case ex => ex.getMessage

// Step 2: Use union types
def modernVersion(): String | AppError =
  try legacyCode()
  catch case ex => AppError.NetworkError(ex.getMessage
      )

// Step 3: Pure functional (no exceptions)
def pureVersion(input: String): String | AppError =
  input.validateThat(_.nonEmpty, "Input required")
    .andThen(processInput)

def processInput(s: String): String | String =
```

# Monitoring and Logging

```scala
// Structured error tracking
def logError(error: AppError, context: Map[String,
    String] = Map.empty): Unit =
  val logData = Map(
    "error_type" -> error.getClass.getSimpleName,
    "message" -> error.message,
    "timestamp" -> java.time.Instant.now().toString
  ) ++ context

  println(s"ERROR: ${logData.map((k, v) => s"$k=$v").
      mkString(", ")}")

// Usage with error handling
def processWithLogging[T](operation: String)(thunk: =>
    T | AppError): T | AppError =
  val startTime = System.currentTimeMillis()
  thunk match
    case result: T =>
      println(s"$operation completed in ${System.
```

# Error Handling Patterns Summary

| Pattern | Use Case | Scala 3 Feature |
|---|---|---|
| Union Types | Simple errors | Native type system |
| Enums | Error hierarchies | Structured ADTs |
| Option | Null safety | Built-in monad |
| Either | Rich errors | Right-biased |
| Extensions | Fluent APIs | Extension methods |
| Given/Using | Error context | Context functions |

**Recommendation:** Start with union types, add structure with enums

# Best Practices

1. **Union types first** for simple error cases
2. **Enums** for structured error hierarchies
3. **Extension methods** for composable APIs
4. **Pattern matching** over exception handling
5. **Value types** for performance-critical code
6. **Test error paths** as thoroughly as success paths

```
//      Good: Type-safe and composable
def parseConfig(file: String): Config | AppError

//      Avoid: Hidden exceptions
def parseConfig(file: String): Config // can throw!

//      Good: Composable validation
input.validateThat(_.nonEmpty, "Required")
  .andThen(parseNumber)
  .andThen(validateRange)
```

# Key Takeaways

- **Union types** are the modern Scala 3 way
- **Zero runtime cost** compared to exceptions
- **Type safety** prevents runtime surprises
- **Composition** enables building complex validations
- **Pattern matching** provides elegant error handling
- **Extensions** create fluent, readable APIs

*Embrace Scala 3's type system for robust, efficient error handling*

# References


[Odersky, 2023] Odersky, M. *Scala 3 Reference: Union Types*. 2023.

[EPFL, 2023] EPFL Team *Scala 3 Book: Error Handling*. 2023.

[Wampler, 2021] Wampler, D. *Programming Scala 3: Functional Error Handling*. 2021.

[Spiewak, 2020] Spiewak, D. *Error Handling in Modern Scala*. 2020.