Functional programming with Scala

Error Handling

Said BOUDJELDA

Senior Software Engineer @SCIAM
GitHub @bmscomp

Course, May 2025

Overview

Modern error handling in Scala 3:

- **Union Types**: Native error representation
- **Enums**: Structured error hierarchies
- **Type Safety**: Errors visible in signatures
- **Zero Cost**: No performance overhead
- **Composition**: Functional error chaining

From exceptions to types

Traditional vs Modern

```scala
// Old way: Hidden exceptions
def divide(a: Int, b: Int): Int =
  if b == 0 then throw ArithmeticException("Zero")
  else a / b

// Scala 3 way: Union types
def safeDivide(a: Int, b: Int): Int | String =
  if b == 0 then "Division by zero" else a / b
```

Modern approach: Type-safe, no hidden failures

Problems with Exception-Based Error Handling

- **Hidden Failures**: No compile-time visibility of errors
- **Type System Bypass**: Exceptions break normal type flow
- **Performance Overhead**: Stack unwinding is expensive
- **Resource Leaks**: Finally blocks can be error-prone
- **Poor Composability**: Hard to chain operations safely
- **Testing Difficulties**: Exception paths often forgotten
- **Maintenance Issues**: Catching wrong exception types

Modern functional approaches solve these problems

Union Types

```scala
type Result[T] = T | String

def parse(s: String): Result[Int] =
  try s.toInt catch case _ => "Invalid number"

parse("42") match
  case num: Int => println(s"Success: $num")
  case err: String => println(s"Error: $err")
```

Simple, type-safe error handling

Enums for Error Types

```scala
enum AppError:
  case ValidationError(msg: String)
  case NetworkError(msg: String)
  case ParseError(msg: String)

def validateAge(age: Int): Int | AppError =
  if age > 0 then age
  else AppError.ValidationError("Invalid age")
```

Structured error hierarchies

Extension Methods

```scala
extension [T](value: T)
  def ensure(condition: T => Boolean,
             error: String): T | String =
    if condition(value) then value else error

// Usage
"hello".ensure(_.nonEmpty, "Empty string")
42.ensure(_ > 0, "Must be positive")
```

Fluent validation APIs

Chaining Operations

```
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

"42".ensure(_.nonEmpty, "Empty")
  .andThen(s => parse(s))
  .andThen(n => n.ensure(_ > 0, "Negative"))
```

Composable error handling

Option for Null Safety

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

val result = findUser(1)
  .map(_.toUpperCase)
  .getOrElse("Not found")

println(result) // USER1
```

Clean null handling

Either for Rich Errors

```scala
def validateUser(name: String, age: Int): Either[
    String, String] =
  for
    n <- if name.nonEmpty then Right(name) else Left("
        No name")
    a <- if age > 0 then Right(age) else Left("Bad age
        ")
  yield s"$n is $a years old"

println(validateUser("John", 25))
// Right(John is 25 years old)
```

For-comprehension error handling

Resource Management

```scala
import scala.util.Using

def readFile(name: String): String | String =
  Using(scala.io.Source.fromFile(name)) { source =>
    source.mkString
  }.toEither match
    case Right(content) => content
    case Left(ex) => s"Error: ${ex.getMessage}"
```

Automatic cleanup

Given/Using Context

```scala
trait Logger:
  def log(msg: String): Unit

given Logger with
  def log(msg: String) = println(s"[LOG] $msg")

def validate[T](value: T)(using logger: Logger):
    Option[T] =
  logger.log(s"Validating $value")
  Some(value)
```

Contextual error handling

Async Error Handling

```scala
import scala.concurrent.Future

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

fetchUser(1).map {
  case user: String => s"Found: $user"
  case error: AppError => s"Error: $error"
}
```

Future with union types

Pattern Matching

```scala
def handle(result: String | Int | AppError): String =
  result match
  case s: String => s"Text: $s"
  case n: Int => s"Number: $n"
  case AppError.ValidationError(msg) => s"Invalid:
    $msg"
  case AppError.NetworkError(msg) => s"Network: $msg"
  case AppError.ParseError(msg) => s"Parse: $msg"
```

Exhaustive error handling

Error Boundaries

```scala
def safe[T](operation: () => T): T | AppError =
  try operation()
  catch
    case _: NumberFormatException => AppError.
        ParseError("Bad format")
    case ex: Exception => AppError.NetworkError(ex.
        getMessage)

val result = safe(() => "42".toInt)
// Success: 42
```

Isolate failures

Validation DSL

```scala
case class Validated[T](value: T, errors: List[String
    ])

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

extension [T](v: Validated[T])
  def check(cond: T => Boolean, err: String):
      Validated[T] =
    if cond(v.value) then v else v.copy(errors = v.
        errors :+ err)
```

Custom validation builder

Testing Errors

```scala
def testValidation(): Unit =
  // Test success
  assert(validateAge(25) == 25)

  // Test error
  validateAge(-5) match
    case AppError.ValidationError(msg) => assert(msg.
        contains("Invalid"))
    case _ => sys.error("Expected error")
```

Simple error testing

Performance: Zero Cost

```scala
// Union types - no boxing overhead
type FastResult = String | Int

def quickParse(s: String): FastResult =
  if s.forall(_.isDigit) then s.toInt else "Invalid"

// Opaque types for domain safety
opaque type UserId = Int
object UserId:
  def apply(id: Int): UserId | String =
    if id > 0 then id else "Invalid ID"
```

Efficient error handling

Real-World API

```scala
case class User(name: String, age: Int)

def createUser(name: String, age: String): User | List
    [String] =
  val validations = List(
    if name.nonEmpty then None else Some("Name
        required"),
    try { age.toInt; None } catch case _ => Some("
        Invalid age")
  ).flatten

  if validations.isEmpty then User(name, age.toInt)
     else validations
```

Complete validation example

Migration Strategy

```scala
// Step 1: Wrap exceptions
def legacy(): String = throw RuntimeException("Old")
def wrapped(): String | String =
  try legacy() catch case ex => ex.getMessage

// Step 2: Use union types
def modern(): String | AppError =
  AppError.NetworkError("Converted")
```

Gradual migration path

Logging and Monitoring

```scala
def withLogging[T](op: String)(f: => T | AppError): T
    | AppError =
  val start = System.currentTimeMillis()
  f match
    case result: T =>
      println(s"$op: success (${System.
          currentTimeMillis() - start}ms)")
      result
    case error: AppError =>
      println(s"$op: error - ${error}")
      error
```

Structured error tracking

Comparison Table

Pattern	Use Case	Performance
Union Types	Simple errors	Zero cost
Enums	Structured errors	Zero cost
Option	Null safety	Minimal overhead
Either	Rich errors	Some allocation
Try	Exception wrapping	Exception cost

Recommendation: Union types for most cases

Best Practices

1. **Union types** for simple cases
2. **Enums** for error hierarchies
3. **Extensions** for fluent APIs
4. **Pattern matching** over try-catch
5. **Compose** operations safely
6. **Test** error scenarios

```
// Good: Explicit errors
def parse(s: String): Int | String

// Bad: Hidden exceptions
def parse(s: String): Int
```

Advanced: Custom Error Monad

```scala
case class Result[T](value: T | String):
  def map[U](f: T => U): Result[U] = value match
    case v: T => Result(f(v))
    case e: String => Result(e)

  def flatMap[U](f: T => Result[U]): Result[U] = value
      match
    case v: T => f(v)
    case e: String => Result(e)
```

Build your own error monad

HTTP Client Example

```
enum HttpError:
  case NotFound, Unauthorized, ServerError

def get(url: String): String | HttpError =
  if url.startsWith("https://") then "Response data"
  else HttpError.NotFound

get("https://api.example.com") match
  case data: String => println(s"Success: $data")
  case HttpError.NotFound => println("URL not found")
```

Simple HTTP error handling

Database Operations

```
enum DbError:
  case ConnectionFailed, QueryFailed, NotFound

def findById(id: Int): User | DbError =
  if id > 0 then User("John", 25)
  else DbError.NotFound

def updateUser(user: User): Unit | DbError =
  if user.name.nonEmpty then () else DbError.
     QueryFailed
```

Database error modeling

Validation Pipeline

```scala
def validateEmail(email: String): String | String =
  email.ensure(_.nonEmpty, "Email required")
    .andThen(_.ensure(_.contains("@"), "Invalid format
       "))
    .andThen(_.ensure(_.length < 100, "Too long"))

def processSignup(email: String) = validateEmail(email
   ) match
  case valid: String => s"Welcome: $valid"
  case error: String => s"Error: $error"
```

Composable validation chain

Configuration Loading

```scala
case class Config(host: String, port: Int)

def loadConfig(): Config | String =
  for
    host <- sys.env.get("HOST").toRight("Missing HOST"
        )
    portStr <- sys.env.get("PORT").toRight("Missing
        PORT")
    port <- try Right(portStr.toInt) catch case _ =>
        Left("Invalid PORT")
  yield Config(host, port)
```

Environment configuration

JSON Parsing

```
def parseJson(json: String): Map[String, String] |
    String =
  try
    val data = json.split(",").map(_.split(":")).map {
        arr =>
      arr(0).trim -> arr(1).trim
    }.toMap
    data
  catch case _ => "Invalid JSON format"

parseJson("name:John,age:25") match
  case data: Map[String, String] => println(data)
  case error: String => println(s"Parse error: $error"
    )
```

Simple JSON parsing with errors

Key Takeaways

- **Union types** are the modern Scala 3 way
- **Zero cost** error handling
- **Type safety** prevents surprises
- **Pattern matching** handles all cases
- **Composition** builds complex validations
- **Migration** can be gradual

Make errors visible in types, not hidden in exceptions

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*. 2021.

[Spiewak, 2020] Spiewak, D. *Functional Error Handling*. 2020.