Error Handling in Scala 3

From Exceptions to 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

Error handling is crucial in any programming language. Scala 3 offers multiple approaches:

- **Traditional**: Exceptions and try-catch blocks
- **Functional**: Option, Either, Try types
- **Modern**: Union types and improved pattern matching

We'll explore evolution from imperative to functional error handling.

Traditional Exception Handling

The Old Way

```scala
// Traditional Java-style exception handling
def divide(a: Int, b: Int): Int = {
  if (b == 0)
    throw new ArithmeticException("Division by zero")
  else
    a / b
}

try {
  val result = divide(10, 0)
  println(s"Result: $result")
} catch {
  case e: ArithmeticException =>
    println(s"Error: ${e.getMessage}")
} finally {
  println("Cleanup operations")
}
```

Problems with Exceptions

- **Not type-safe**: Exceptions are not tracked in method signatures
- **Control flow**: Breaks normal program flow
- **Performance**: Stack unwinding is expensive
- **Composition**: Hard to compose operations that might fail

```scala
// Signature doesn't tell us this method can fail
def parseNumber(s: String): Int = s.toInt // Can throw!

// Callers might forget to handle exceptions
val num = parseNumber("not-a-number") // Runtime crash!
```

Option Type - Handling Null Values

Functional Approach

```scala
// Option represents optional values - Some or None
def safeDivide(a: Int, b: Int): Option[Int] =
  if (b == 0) None else Some(a / b)

// Pattern matching
safeDivide(10, 2) match {
  case Some(result) => println(s"Result: $result")
  case None => println("Division by zero")
}

// Functional operations
val result = safeDivide(10, 2)
  .map(_ * 2)           // Transform if present
  .filter(_ > 5)        // Filter condition
  .getOrElse(0)         // Default value

println(result) // 10
```

Option - Advanced Operations

```scala
// Chaining operations that might fail
def parseAge(s: String): Option[Int] =
  try Some(s.toInt) catch case _ => None

def validateAge(age: Int): Option[Int] =
  if (age >= 0 && age <= 150) Some(age) else None

// Composition using flatMap
def processAge(input: String): Option[String] =
  parseAge(input)
    .flatMap(validateAge)
    .map(age => s"Valid age: $age")

println(processAge("25"))    // Some(Valid age: 25)
println(processAge("200"))   // None
println(processAge("abc"))   // None
```

Either Type - Rich Error Information

Left = Error, Right = Success

```
sealed trait AppError
case class ValidationError(msg: String) extends
    AppError
case class ParseError(msg: String) extends AppError

def parseAndValidateAge(s: String): Either[AppError,
    Int] =
  try {
    val age = s.toInt
    if (age >= 0 && age <= 150)
      Right(age)
    else
      Left(ValidationError(s"Invalid age: $age"))
  } catch {
    case _: NumberFormatException =>
      Left(ParseError(s"Not a number: $s"))
  }
```

Either - Functional Operations

```scala
// Either is right-biased in Scala 2.12+
val result = parseAndValidateAge("25")
  .map(_ * 2)                          // Only if Right
  .flatMap(age =>
    if (age < 100) Right(s"Young: $age")
    else Left(ValidationError("Too old")))

// For-comprehension with Either
def processUser(name: String, ageStr: String) =
  for {
    age <- parseAndValidateAge(ageStr)
    validName <- if (name.nonEmpty) Right(name)
                 else Left(ValidationError("Empty name"))
  } yield User(validName, age)

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

println(processUser("John", "25")) // Right(User(John
```

Try Type - Exception Wrapping

```scala
import scala.util.{Try, Success, Failure}

// Try wraps operations that might throw exceptions
def safeParse(s: String): Try[Int] = Try(s.toInt)

def safeFileRead(filename: String): Try[String] =
  Try(scala.io.Source.fromFile(filename).mkString)

// Pattern matching
safeParse("123") match {
  case Success(num) => println(s"Parsed: $num")
  case Failure(ex) => println(s"Failed: ${ex.getMessage}")
}

// Functional operations
val result = safeParse("42")
  .map(_ * 2)
  .recover { case _: NumberFormatException => 0 }
```

Scala 3 Union Types for Errors

Modern Approach

```scala
// Union types in Scala 3
type ParseResult = Int | String

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

// Pattern matching with union types
parseNumber("42") match {
  case num: Int => println(s"Parsed: $num")
  case error: String => println(s"Error: $error")
}

// More complex union types
type Result[T] = T | Exception

def divide(a: Int, b: Int): Result[Double] =
```

Error Accumulation with Validated

Collecting Multiple Errors

```scala
// Using cats library for error accumulation
import cats.data.Validated
import cats.syntax.all._

type ValidationResult[T] = Validated[List[String], T]

def validateName(name: String): ValidationResult[
    String] =
  if (name.nonEmpty) name.valid
  else List("Name cannot be empty").invalid

def validateAge(age: Int): ValidationResult[Int] =
  if (age >= 0 && age <= 150) age.valid
  else List(s"Invalid age: $age").invalid

// Accumulate all errors
(validateName(""), validateAge(200)).mapN(User.apply)
    match {
```

Custom Error ADTs

Algebraic Data Types for Errors

```scala
// Define comprehensive error hierarchy
sealed trait DatabaseError extends Exception
case class ConnectionError(msg: String) extends DatabaseError
case class QueryError(sql: String, msg: String) extends DatabaseError
case class TimeoutError(seconds: Int) extends DatabaseError

sealed trait ValidationError extends Exception
case class InvalidEmail(email: String) extends ValidationError
case class InvalidPassword(reason: String) extends ValidationError

// Combine different error types
type AppError = DatabaseError | ValidationError
```

Error Handling with For-Comprehensions

```scala
// Sequential error handling
def processOrder(): Either[String, Order] =
  for {
    user <- findUser("john@example.com")
    product <- findProduct("laptop")
    inventory <- checkInventory(product.id)
    order <- createOrder(user, product) if inventory >
        0
  } yield order

// With custom error types
def processOrderAdvanced(): Either[AppError, Order] =
  for {
    user <- findUser("john@example.com")
              .toRight(UserNotFound("john@example.com"
                  ))
    product <- findProduct("laptop")
                  .toRight(ProductNotFound("laptop"))
    _ <- validateInventory(product)
```

Resource Management with Using

Scala 3 Automatic Resource Management

```
import scala.util.Using

// Automatic resource cleanup
def readFileContent(filename: String): Try[String] =
  Using(scala.io.Source.fromFile(filename)) { source =>
    source.getLines().mkString("\n")
  }

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

// Resource is automatically closed even if exception
```

Error Recovery Strategies

```scala
// Retry mechanism
def withRetry[T](maxAttempts: Int)(operation: () =>
    Try[T]): Try[T] =
  operation() match {
    case success @ Success(_) => success
    case Failure(_) if maxAttempts > 1 =>
      withRetry(maxAttempts - 1)(operation)
    case failure => failure
  }

// Circuit breaker pattern
class CircuitBreaker(failureThreshold: Int) {
  private var failureCount = 0
  private var state: State = Closed

  def execute[T](operation: () => T): Try[T] =
    state match {
      case Closed => Try(operation()).recoverWith(
          handleFailure)
```

Async Error Handling with Future

```scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

// Async operations with error handling
def fetchUser(id: Int): Future[Either[String, User]] =
  Future {
    // Simulate network call
    Thread.sleep(100)
    if (id > 0) Right(User(s"user$id", 25))
    else Left("Invalid user ID")
  }

// Combine async operations
def getUserProfile(id: Int): Future[Either[String, Profile]] =
  for {
    userResult <- fetchUser(id)
    profile <- userResult match {
```

Monadic Error Handling

Composing Operations

```scala
// Error monad for chaining operations
case class Result[+T](value: Either[String, T]) {
  def map[U](f: T => U): Result[U] =
    Result(value.map(f))

  def flatMap[U](f: T => Result[U]): Result[U] =
    value match {
      case Right(v) => f(v)
      case Left(e) => Result(Left(e))
    }
}

object Result {
  def success[T](value: T): Result[T] = Result(Right(
     value))
  def failure[T](error: String): Result[T] = Result(
     Left(error))
}
```

Error Handling Best Practices

- **Use types**: Make errors explicit in function signatures
- **Avoid exceptions**: For predictable failures, use Option/Either
- **Fail fast**: Validate inputs early
- **Error accumulation**: Collect all validation errors
- **Recovery**: Provide fallback mechanisms

```scala
// Good: Error is explicit in return type
def parseConfig(file: String): Either[ConfigError, Config]

// Bad: Exception not visible in signature
def parseConfig(file: String): Config // throws ConfigException

// Good: Accumulate validation errors
def validateUser(data: UserData): ValidatedNel[Error, User]

// Bad: Stop at first error
```

Performance Considerations

```
// Option/Either allocation overhead
def heavyComputation(): Option[Int] = {
  // Avoid creating Option for every intermediate step
  val intermediate = computeValue()
  if (isValid(intermediate)) Some(intermediate) else None
}

// Use specialized collections for performance
import scala.collection.mutable

// For high-performance scenarios, consider using:
// - Specialized Option types (OptionalInt, etc.)
// - Custom Result types with value classes
// - Unboxed union types in Scala 3

value class UserId(val value: Int) extends AnyVal
type UserResult = UserId | String // Union type - no boxing!
```

Scala 3 Improvements

New Features for Error Handling

```scala
// Union types for error handling
type ParseError = NumberFormatException | IllegalArgumentException

// Improved pattern matching
def handleError(error: Throwable): String = error match {
  case _: NumberFormatException => "Invalid number format"
  case _: IllegalArgumentException => "Invalid argument"
  case _ => "Unknown error"
}

// Enums for error codes
enum ErrorCode {
  case ValidationFailed, NetworkTimeout, DatabaseError
```

Error Boundary Pattern

```scala
// Error boundary for isolating failures
trait ErrorBoundary[F[_]] {
  def handle[A](fa: F[A])(recover: Throwable => A): F[A]
}

// Implementation for Future
given ErrorBoundary[Future] with {
  def handle[A](fa: Future[A])(recover: Throwable => A): Future[A] =
    fa.recover { case ex => recover(ex) }
}

// Usage
def safeOperation[F[_]: ErrorBoundary](computation: F[String]): F[String] =
  summon[ErrorBoundary[F]].handle(computation) { ex =>
    s"Operation failed: ${ex.getMessage}"
  }
```

Testing Error Scenarios

```scala
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers

class ErrorHandlingSpec extends AnyFlatSpec with Matchers {

  "safeDivide" should "return None for division by zero" in {
    safeDivide(10, 0) shouldBe None
  }

  it should "return Some for valid division" in {
    safeDivide(10, 2) shouldBe Some(5)
  }

  "parseAndValidateAge" should "accumulate multiple errors" in {
    val result = validateUser(UserData("", -5))
    result.isInvalid shouldBe true
```

Migration Strategy

From Exceptions to Functional

```scala
// Phase 1: Wrap existing exception-throwing code
def legacyOperation(): String = throw new
    RuntimeException("Legacy!")

def wrappedLegacy(): Try[String] = Try(legacyOperation
    ())

// Phase 2: Introduce Either for domain errors
def improvedOperation(): Either[String, String] =
  wrappedLegacy().toEither.left.map(_.getMessage)

// Phase 3: Use custom error types
sealed trait DomainError
case class LegacyError(msg: String) extends
    DomainError

def modernOperation(): Either[DomainError, String] =
  improvedOperation().left.map(LegacyError.apply)
```

Real-World Example: HTTP Client

```
import sttp.client3._

sealed trait HttpError
case class NetworkError(cause: Throwable) extends
   HttpError
case class InvalidResponse(code: Int, body: String)
   extends HttpError
case class ParseError(json: String, cause: Throwable)
   extends HttpError

def fetchUser(id: Int): IO[Either[HttpError, User]] =
   {
  val request = basicRequest
    .get(uri"https://api.example.com/users/$id")
    .response(asString)

  for {
    response <- request.send(backend).attempt.map(_.
       left.map(NetworkError.apply))
```

Error Handling in Web Applications

```
// Using Tapir for HTTP API error handling
import sttp.tapir._

sealed trait ApiError
case class ValidationError(field: String, message:
   String) extends ApiError
case class NotFoundError(resource: String, id: String)
    extends ApiError
case class ServerError(message: String) extends
   ApiError

val getUserEndpoint = endpoint.get
  .in("users" / path[String]("id"))
  .out(jsonBody[User])
  .errorOut(oneOf[ApiError](
    oneOfVariant(statusCode(StatusCode.BadRequest).and
        (jsonBody[ValidationError])),
    oneOfVariant(statusCode(StatusCode.NotFound).and(
        jsonBody[NotFoundError])),
```

Monitoring and Observability

```
// Error tracking with structured logging
import org.slf4j.LoggerFactory
import io.circe.syntax._

val logger = LoggerFactory.getLogger(this.getClass)

def processWithLogging[A](operation: String)(thunk: =>
    Either[AppError, A]): Either[AppError, A] = {
  val startTime = System.currentTimeMillis()

  thunk match {
    case Right(result) =>
      logger.info(s"$operation completed successfully
          in ${System.currentTimeMillis() - startTime}
          ms")
      Right(result)

    case Left(error) =>
      logger.error(s"$operation failed after ${System.
```

Error Handling Patterns Summary

Pattern	**Use Case**	**Pros**	**Cons**
Try-Catch	Legacy code	Familiar	Not type-safe
Option	Null safety	Simple	No error info
Either	Rich errors	Type-safe	Right-biased only
Validated	Error accumulation	Collects all errors	More complex
Union Types	Scala 3 errors	Modern, efficient	New syntax
IO/Effect	Async + Sync	Composable	Learning curve

- Choose based on your specific requirements
- Migrate gradually from exceptions to functional types
- Consider performance implications

Key Takeaways

1. **Make errors explicit** in function signatures
2. **Use Option** for simple null/missing value cases
3. **Use Either** when you need error information
4. **Use Validated** when you need to accumulate errors
5. **Consider Union types** in Scala 3 for performance
6. **Design error hierarchies** with sealed traits
7. **Test error scenarios** thoroughly
8. **Provide recovery mechanisms** where appropriate

Functional error handling leads to more robust, composable, and maintainable code.

References and Further Reading

[Odersky, 2021] Odersky, M. *Scala 3 Reference: Union Types*. 2021.
Official documentation on union types for error handling

[Spiewak, 2018] Spiewak, D. *Functional Error Handling with Monads*. 2018.
Comprehensive guide to monadic error handling

[Lipovaca, 2011] Lipovaca, M. *Learn You a Haskell: Error Handling*. 2011.
Fundamental concepts of functional error handling

[Cats Contributors, 2023] Cats Contributors *Cats Documentation: Error Handling*. 2023.
Practical patterns with cats library

[Wampler, 2020] Wampler, D. *Programming Scala: Error Handling*. 2020.
Industry best practices for Scala error handling