ADVENTURES IN TYPE-SAFE ERROR HANDLING

How do we handle errors in Scala today?

EITHER[+E, +A]

EITHER[+E, +A]

- Use any error type you want!

EITHER[+E, +A]

- Use any error type you want!

```scala
sealed trait AllErrors extends Throwable
final case class E1() extends AllErrors
final case class E2() extends AllErrors
final case class E3() extends AllErrors
```

```scala
def maybeError1: Either[E1, Unit] = ???
def maybeError2: Either[E2, Unit] = ???

val result: Either[AllErrors, Unit] =
  for {
    _ <- maybeError1
    _ <- maybeError2
  } yield ()
```

EITHER[+E, +A]

- Use any error type you want!

```scala
sealed trait AllErrors extends Throwable
final case class E1() extends AllErrors
final case class E2() extends AllErrors
final case class E3() extends AllErrors
```

```scala
def maybeError1: Either[E1, Unit] = ???
def maybeError2: Either[E2, Unit] = ???

val result: Either[AllErrors, Unit] =
  for {
    _ <- maybeError1
    _ <- maybeError2
  } yield ()
```

- Covariant - Compiler will find **Least Upper Bound** (LUB) to reconcile the error type

CATS IO[A]

CATS IO[A]

- Any **IO[A]** can fail with a **Throwable** using **IO.raiseError**

CATS IO[A]

- Any **IO[A]** can fail with a **Throwable** using **IO.raiseError**
- Have to rely on documentation, not types

CATS IO[A]

- Any **IO[A]** can fail with a **Throwable** using **IO.raiseError**
- Have to rely on documentation, not types

```scala
def io1: IO[Unit] = ???
def io2: IO[Unit] = ???

val ioResult: IO[Unit] =
  for {
    _ <- io1
    _ <- io2
  } yield ()
```

EITHER IN IO

- Need to check the **Either** result from the previous step
- Error prone and verbose - Not recommended

```
def io1: IO[Either[E1, Unit]] = ???
def io2: IO[Either[E2, Unit]] = ???

for {
  result <- io1
  result2 <- result match {
    case Left(e1) => IO.pure(Left(e1))
    case Right(_) => io2
  }
} yield {
  result2 match {
    case Left(e2) => Left(e2)
    case Right(_) => Right(())
  }
}
```

EITHERT[IO, E, A]

Similar to **IO[Either[E, A]]** but “short-circuits” when you have a `Left`

EITHERT[IO, E, A]

Similar to **IO[Either[E, A]]** but “short-circuits” when you have a `Left`

```scala
def et1: EitherT[IO, E1, Unit] = ???
def et2: EitherT[IO, E2, Unit] = ???

val eitherTResult: EitherT[IO, AllErrors, Unit] =
  for {
    _ <- et1.leftWiden[AllErrors]
    _ <- et2.leftWiden[AllErrors]
  } yield ()
```

EITHERT[IO, E, A]

Similar to **IO[Either[E, A]]** but “short-circuits” when you have a `Left`

```
def et1: EitherT[IO, E1, Unit] = ???
def et2: EitherT[IO, E2, Unit] = ???

val eitherTResult: EitherT[IO, AllErrors, Unit] =
  for {
    _ <- et1.leftWiden[AllErrors]
    _ <- et2.leftWiden[AllErrors]
  } yield ()
```

- Invariant - no auto upcasting but you can use `leftWiden`

EITHERT[IO, E, A]

Similar to **IO[Either[E, A]]** but “short-circuits” when you have a `Left`

```scala
def et1: EitherT[IO, E1, Unit] = ???
def et2: EitherT[IO, E2, Unit] = ???

val eitherTResult: EitherT[IO, AllErrors, Unit] =
  for {
    _ <- et1.leftWiden[AllErrors]
    _ <- et2.leftWiden[AllErrors]
  } yield ()
```

- Invariant - no auto upcasting but you can use `leftWiden`
- **IO.raiseError** reserved for defects or unhandleable errors

BIFUNCTOR IO[+E, +A] (ZIO)

- Similar to EitherT, but better ergonomic

BIFUNCTOR IO[+E, +A] (ZIO)

- Similar to EitherT, but better ergonomic

```
def zio1: IO[E1, Unit] = ???
def zio2: IO[E2, Unit] = ???

val eitherTResult: IO[AllErrors, Unit] =
  for {
    _ <- zio1
    _ <- zio2
  } yield ()
```

BIFUNCTOR IO[+E, +A] (ZIO)

- Similar to EitherT, but better ergonomic

```
def zio1: IO[E1, Unit] = ???
def zio2: IO[E2, Unit] = ???

val eitherTResult: IO[AllErrors, Unit] =
  for {
    _ <- zio1
    _ <- zio2
  } yield ()
```

- Can terminate the execution chain with a `Throwable` (**IO.die**)

JAVA CHECKED EXCEPTIONS!

```java
void method1() throws E1 { ... }

void method2() throws E2 { ... }

void onlyE1Handled() throws E2 { // E2 not handled, must declare!
  try {
    method1();
    method2();
  }
  catch (E1 e1) { ... }
}

void allHandled() { // All errors handled!
  try {
    method1();
    method2();
  }
  catch (E1 e1) { ... }
  catch (E2 e2) { ... }
}
```

THE TROUBLE WITH CHECKED EXCEPTIONS

THE TROUBLE WITH CHECKED EXCEPTIONS

- Not available in Scala 🙃

THE TROUBLE WITH CHECKED EXCEPTIONS

- Not available in Scala 🙃
- Errors are not values. Doesn’t work well with many newer language features such as anonymous functions

THE TROUBLE WITH CHECKED EXCEPTIONS

- Not available in Scala 🙃
- Errors are not values. Doesn’t work well with many newer language features such as anonymous functions
- Type system special case - no abstraction or reuse

BUT IT HAS MANY COOL IDEAS TOO

BUT IT HAS MANY COOL IDEAS TOO

- Exhaustive handling

BUT IT HAS MANY COOL IDEAS TOO

- Exhaustive handling
- Partial handling

BUT IT HAS MANY COOL IDEAS TOO

- Exhaustive handling
- Partial handling
- Open union of errors

BUT IT HAS MANY COOL IDEAS TOO

- Exhaustive handling
- Partial handling
- Open union of errors
- Can we have these in Scala?

SHAPELESS COPRODUCT!

SHAPELESS COPRODUCT!

```scala
import shapeless._
type E123 = E1 :+: E2 :+: E3 :+: CNil
// Similar to Either[E1, Either[E2, Either[E3, CNil]]

import shapeless.syntax.inject._

val e1InCoproduct: E1 :+: E2 :+: E3 :+: CNil = E1().inject[E1 :+: E2 :+: E3 :+: CNil]
// e1InCoproduct: E1 :+: E2 :+: E3 :+: CNil = Inl(E1())
val e2InCoproduct: E1 :+: E2 :+: E3 :+: CNil = E2().inject[E1 :+: E2 :+: E3 :+: CNil]
// e2InCoproduct: E1 :+: E2 :+: E3 :+: CNil = Inr(Inl(E2()))

e2InCoproduct match {
  case Inl(E1()) => println("it's E1!")
  case Inr(Inl(E2())) => println("it's E2!")
  case Inr(Inr(Inl(E3()))) => println("it's E3!")
  case Inr(Inr(Inr(cnil))) => cnil.impossible // To satisfy exhaustiveness check
}
// it's E2!
```

COPRODUCTS ARE FLEXIBLE!

Let’s extract a particular cases from a coproduct!

```scala
import shapeless.ops.coproduct._

// Returns a Left(E1()) if we have an E1
Remove[E1 :+: E2 :+: E3 :+: CNil, E1].apply(e1InCoproduct)
// res8: Either[E1, E2 :+: E3 :+: CNil] = Left(E1())

// Otherwise return the rest in Right(..)
Remove[E1 :+: E2 :+: E3 :+: CNil, E2].apply(e1InCoproduct)
// res9: Either[E2, E1 :+: E3 :+: CNil] = Right(Inl(E1()))
```

COPRODUCTS ARE FLEXIBLE!

Let’s extract a particular cases from a coproduct!

```scala
import shapeless.ops.coproduct._

// Returns a Left(E1()) if we have an E1
Remove[E1 :+: E2 :+: E3 :+: CNil, E1].apply(e1InCoproduct)
// res8: Either[E1, E2 :+: E3 :+: CNil] = Left(E1())

// Otherwise return the rest in Right(..)
Remove[E1 :+: E2 :+: E3 :+: CNil, E2].apply(e1InCoproduct)
// res9: Either[E2, E1 :+: E3 :+: CNil] = Right(Inl(E1()))
```

…and you can do many, many things with Coproducts!

Using **Coproducts** directly feels cumbersome

Can we make it nicer?

HOTPOTATO

A library for type-safe, ergonomic and readable error handling!

HOTPOTATO

A library for type-safe, ergonomic and readable error handling!

- Based on Shapeless coproducts

HOTPOTATO

A library for type-safe, ergonomic and readable error handling!

- Based on Shapeless coproducts
- Integrates with ZIO and Cats

FIRST, A BIT OF SIMPLIFICATION

Coproducts can be a bit tedious to read and write, so Hotpotato provides some type aliases for coproducts

```scala
import hotpotato._

type ErrorsSimple = OneOf3[E1, E2, E3] // is equivalent to E1 :+: E2 :+: E3 :+: CNil
```

HANDLING ERRORS - EXHAUSTIVE

- Convert all errors into one single type
- **OR** each to its own type

```scala
import hotpotato._
import shapeless.syntax.inject._
import zio._

val io: IO[OneOf3[E1, E2, E3], Unit] = IO.fail(E1().inject[OneOf3[E1, E2, E3]])

// Turn every error into String
val resString: IO[String, Unit] = io.mapErrorAllInto(
  (e1: E1) => "e1",
  (e2: E2) => "e2",
  (e3: E3) => "e3",
)

// Turn every error into some other type
val result: IO[OneOf2[X2, X1], Unit] = io.mapErrorAll(
  (e1: E1) => X1(),
  (e2: E2) => X2(),
  (e3: E3) => X1(),
)
```

HANDLING ERRORS - PARTIAL

```scala
import hotpotato._

val ioE123: IO[OneOf3[E1, E2, E3], String] = ???

// Turn some error into String
val result: IO[OneOf3[String, Int, E3], String] = ioE123.mapErrorSome(
  (e1: E1) => "e1",
  (e2: E2) => 12,
)
```

ERROR HANDLING WITH SIDE-EFFECTS

Very often error recovery/handling requires side-effect (e.g. logging)

```scala
import hotpotato._

val ioE123: IO[OneOf3[E1, E2, E3], String] = ???
val fallbackIO: E1 => IO[Int, String] = ???

val result: IO[OneOf3[Int, E2, E3], String] = ioE123.flatMapErrorSome(
  (e1: E1) => fallbackIO(e1),
)
```

ERROR HANDLING WITH SIDE-EFFECTS

Very often error recovery/handling requires side-effect (e.g. logging)

```scala
import hotpotato._

val ioE123: IO[OneOf3[E1, E2, E3], String] = ???
val fallbackIO: E1 => IO[Int, String] = ???

val result: IO[OneOf3[Int, E2, E3], String] = ioE123.flatMapErrorSome(
  (e1: E1) => fallbackIO(e1),
)
```

flatMapErrorAll, flatMapErrorAllInto are provided for exhaustive handling too

COMBINING ERRORS

We often have a series of steps and each step may have different errors

```scala
import hotpotato._

val ioE1: IO[E1, Unit] = ???
val ioE23: IO[OneOf2[E2, E3], Unit] = ???

// An embedder tells the compiler what types we want all errors to embed to
implicit val embedder: Embedder[OneOf3[E1, E2, E3]] = Embedder.make

val result: IO[OneOf3[E1, E2, E3], Unit] = for {
  _ <- ioE1.embedError
  _ <- ioE23.embedError
} yield ()
```

INTERFACING WITH SEALED TRAIT ERRORS

Easy conversion from/to sealed traits

```scala
import hotpotato._

// Recall that E1, E2 and E3 all extends AllErrors
val ioAllErrors: IO[AllErrors, String] = ???

val ioE123: IO[OneOf3[E1, E2, E3], String] = ioAllErrors.errorAsCoproduct

val ioAllErrorsAgain: IO[AllErrors, String] = ioE123.unifyError
```

SUMMARY

	ChckEx	EitherT	ZIO	With Hotpotato
Composable	✗	✓	✓	✓
Error type unification	✓	🥲	✓	😅
Open error union	✓	✗	✗	✓
Handling - Exhaustive	✓	✓	✓	✓
Handling - Partial	✓	✗	✗	✓

IT'S JUST THE BEGINNING!

- Hotpotato is available now!
- Your ideas, feedback and use cases are welcome!
- Docs: jatcwang.github.io/hotpotato/
- Gitter: jatcwang.github.io/hotpotato/

THANK YOU!

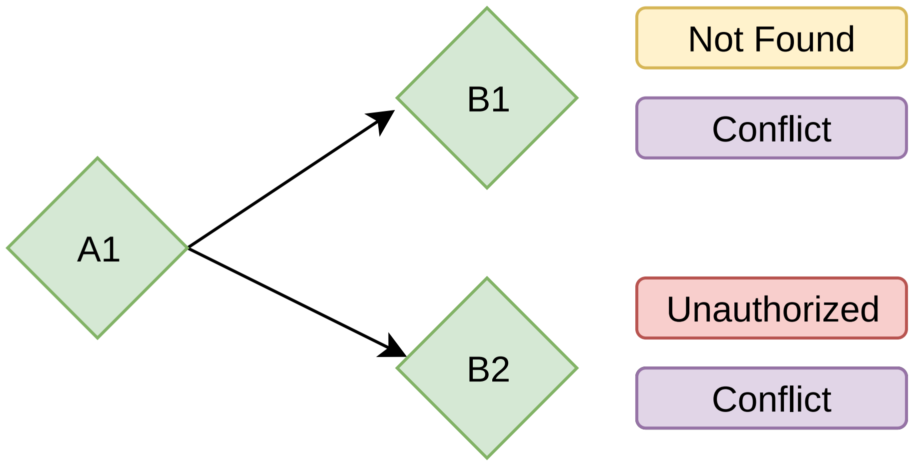
- Twitter / Github: @jatcwang

WHY SEALED TRAIT ISN’T ENOUGH

Let’s look at an example

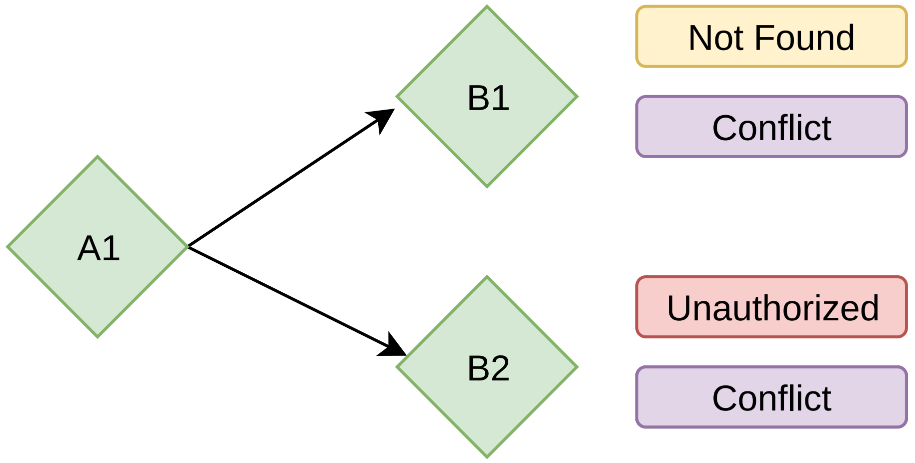
WHY SEALED TRAIT ISN’T ENOUGH

Let’s look at an example

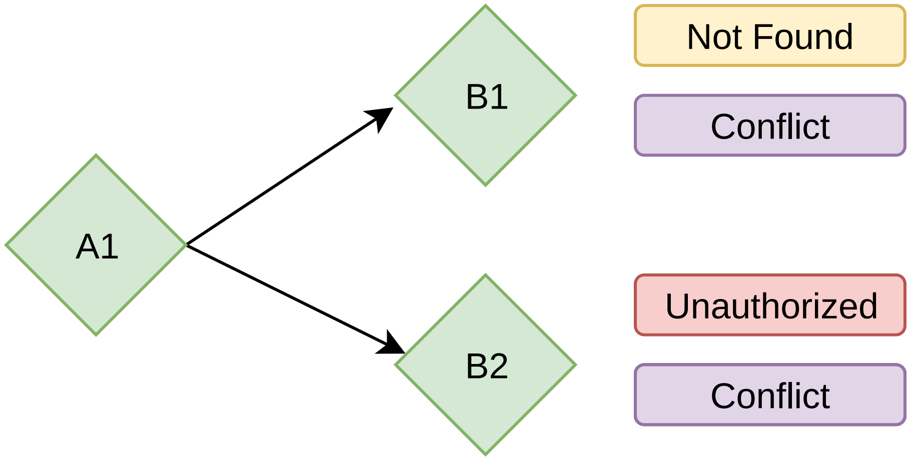


WHY SEALED TRAIT ISN’T ENOUGH

Let’s look at an example



How should we model the errors for **B1** and **B2**?



```scala
sealed trait B1Errors
sealed trait B2Errors

case class Conflict() extends B1Errors with B2Errors
case class NotFound() extends B1Errors
case class Unauthorized() extends B2Errors
```

```scala
sealed trait B1Errors
sealed trait B2Errors

case class Conflict() extends B1Errors with B2Errors
case class NotFound() extends B1Errors
case class Unauthorized() extends B2Errors
```

```scala
sealed trait B1Errors
sealed trait B2Errors

case class Conflict() extends B1Errors with B2Errors
case class NotFound() extends B1Errors
case class Unauthorized() extends B2Errors
```

- Error class declaration now need to be in the same file

```scala
sealed trait B1Errors
sealed trait B2Errors

case class Conflict() extends B1Errors with B2Errors
case class NotFound() extends B1Errors
case class Unauthorized() extends B2Errors
```

- Error class declaration now need to be in the same file
- You cannot use these error classes in another error hierarchy

```scala
sealed trait B1Errors
sealed trait B2Errors

case class Conflict() extends B1Errors with B2Errors
case class NotFound() extends B1Errors
case class Unauthorized() extends B2Errors
```

- Error class declaration now need to be in the same file
- You cannot use these error classes in another error hierarchy
- We want:
    - Exhaustive matching
    - Partial elimination
    - Use types we don’t own

// reveal.js plugins