

# FOUNDATION

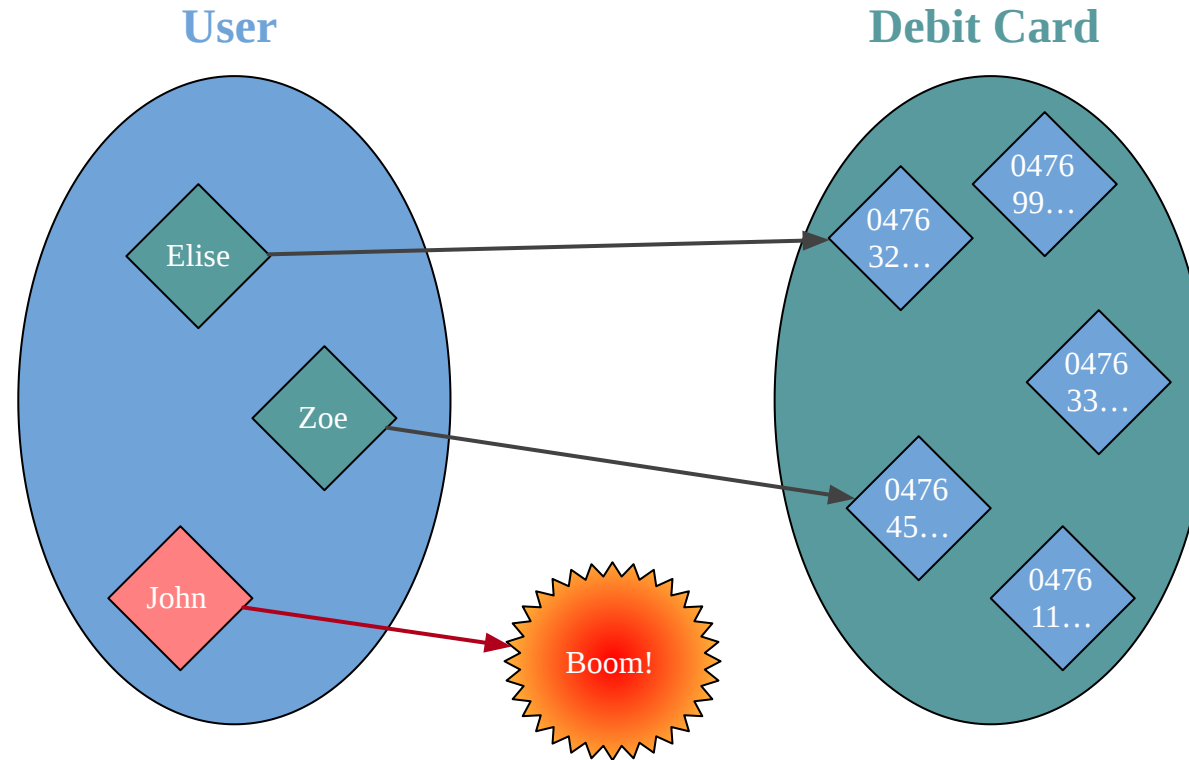


## Error Handling

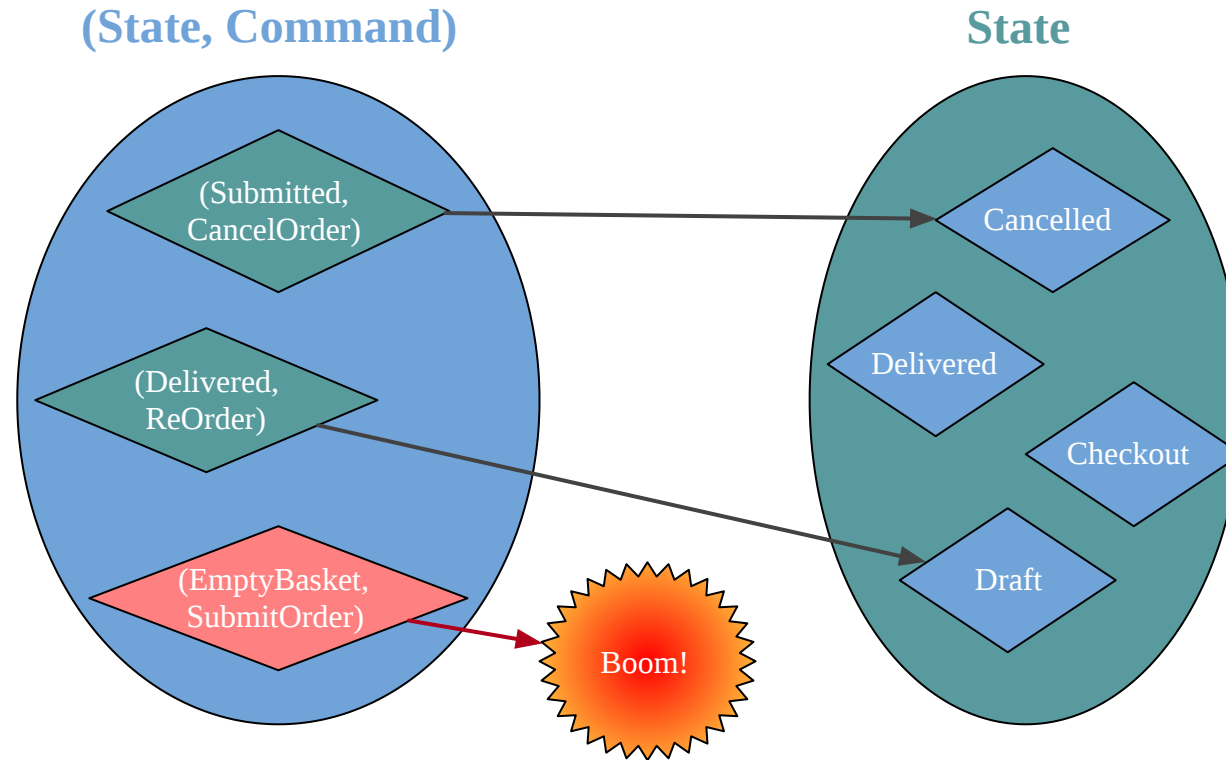
# How to deal with runtime errors



# Partial Function



# Partial Function



# Error handling objectives

1. Document when and what type of errors can occur
2. Force caller to deal with errors
3. Make it easy to fail



# Exception

```
case class Item(id: Long, unitPrice: Double, quantity: Int)

case class Order(status: String, basket: List[Item])

def submit(order: Order): Order =
  order.status match {
    case "Draft" if order.basket.nonEmpty =>
      order.copy(status = "Submitted")
    case other =>
      throw new Exception("Invalid Command")
  }
```

```
scala> submit(Order("Draft", Nil))
java.lang.Exception: Invalid Command
  at .submit(<console>:7)
  ... 42 elided
```



# Exception

```
case object EmptyBasketError extends Exception
case class InvalidCommandError(command: String, order: Order) extends Exception

def submit(order: Order): Order =
  order.status match {
    case "Draft" =>
      if(order.basket.isEmpty) throw EmptyBasketError
      else order.copy(status = "Submitted")
    case other =>
      throw new InvalidCommandError("submit", order)
  }
```

```
scala> submit(Order("Draft", Nil))
EmptyBasketError$
... 44 elided

scala> submit(Order("Delivered", Nil))
InvalidCommandError
at .submit(<console>:8)
... 42 elided
```



# Exceptions are not documented

```
def submit(order: Order): Order = ???

def canSubmit(order: Order): Boolean =
  try {
    submit(order)
    true
  } catch {
    case EmptyBasketError      => false
    case _: InvalidCommandError => false
    case _: ArithmeticException => true
    case _: Exception          => false
  }
```





# Exceptions are not documented

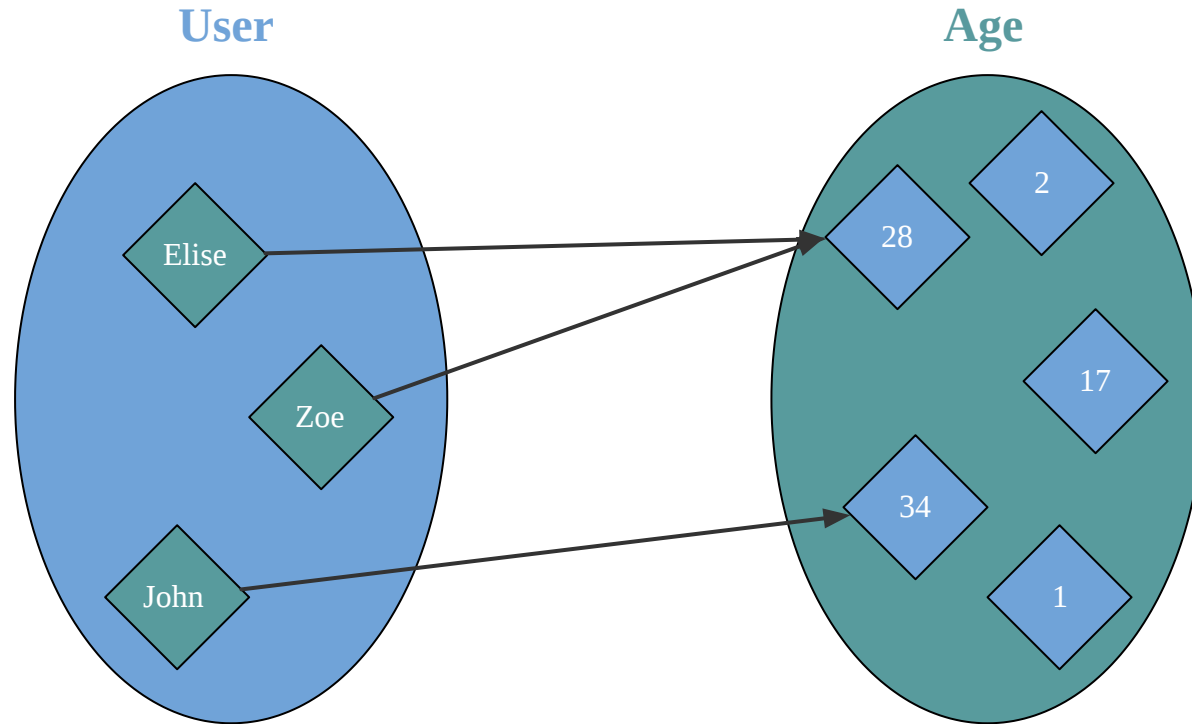
```
def submit(order: Order): Order = ???  
  
def canSubmit(order: Order): Boolean =  
  try {  
    submit(order)  
    true  
  } catch {  
    case EmptyBasketError      => false  
    case _: InvalidCommandError => false  
    case _: ArithmeticException => true  
    case _: Exception          => false  
  }
```

## In Java, you have checked Exception

```
public Order submit(Order order) throws EmptyBasketError, InvalidCommandError
```



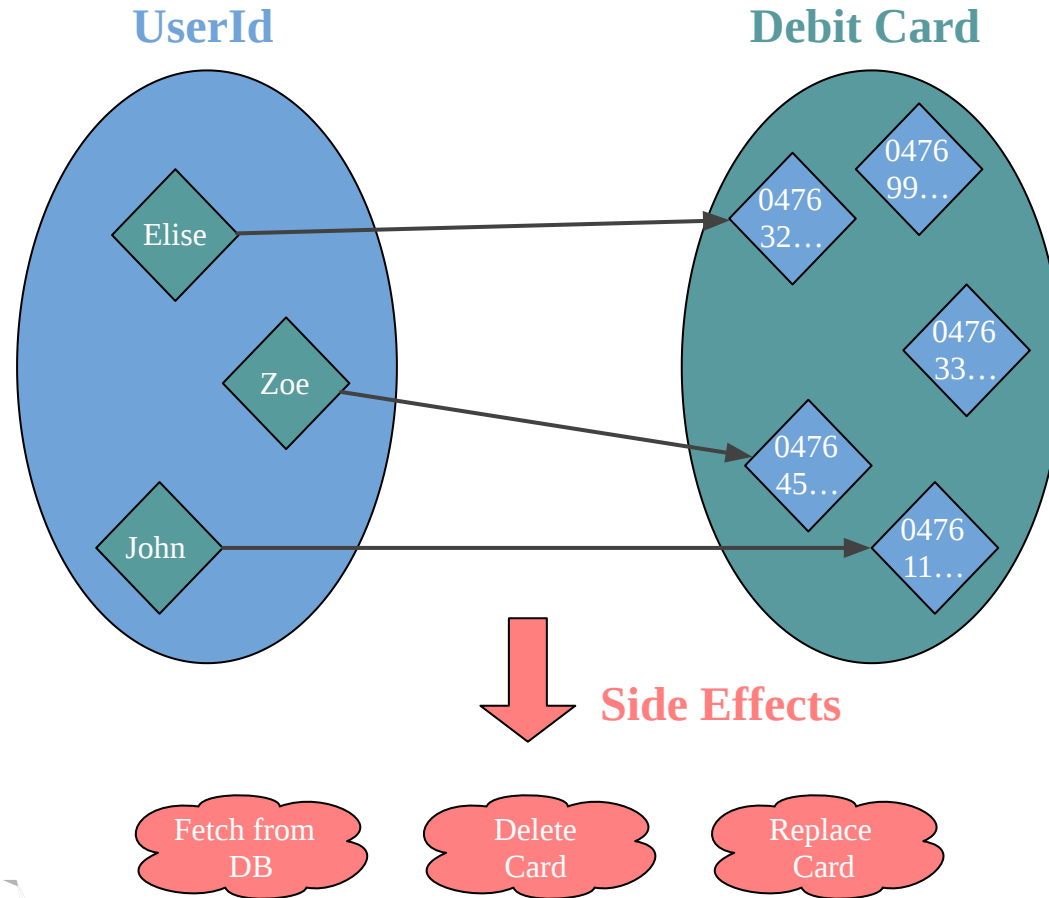
# Pure Function is a mapping between types



- Powerful refactoring
- Local reasoning
- Easier to test
- Potential performance optimisation
- Better documentation



# Functions with side effects are not pure

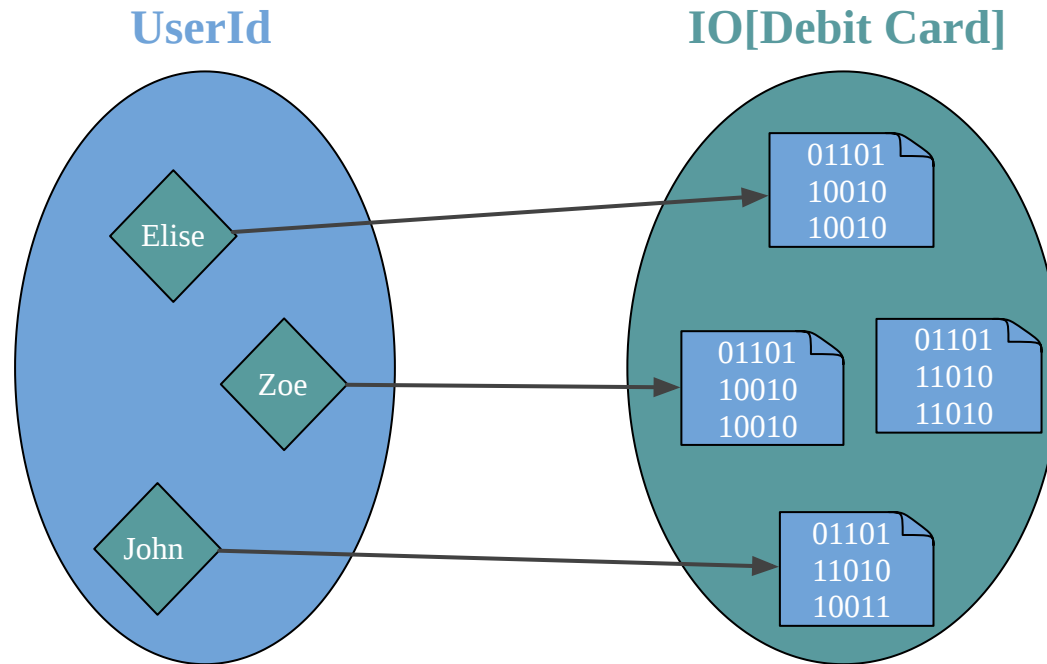


```
def getDebitCard(userId: UserId): DebitCard = {  
  val user = db.getUser(userId)  
  user.debitCard  
}
```

```
def deleteDebitCard(userId: UserId): DebitCard = {  
  val user = db.getUser(userId)  
  val debitCard = user.debitCard  
  db.upsertDebitCard(userId, null)  
  debitCard  
}
```



# Pure side effects and exceptions with IO



```
def extractDebitCard(user: User): IO[DebitCard] = {  
  if(user.debitCard == null)  
    IO.fail(new Exception("No debit card"))  
  else  
    IO.succeed(user.debitCard)  
}  
  
def deleteDebitCard(db: DbApi, userId: UserId): IO[Deb  
  for {  
    user      <- db.getUser(userId)  
    debitCard <- extractDebitCard(user)  
    _        <- db.upsertDebitCard(userId, null)  
  } yield debitCard
```



# IO error handling is fragile

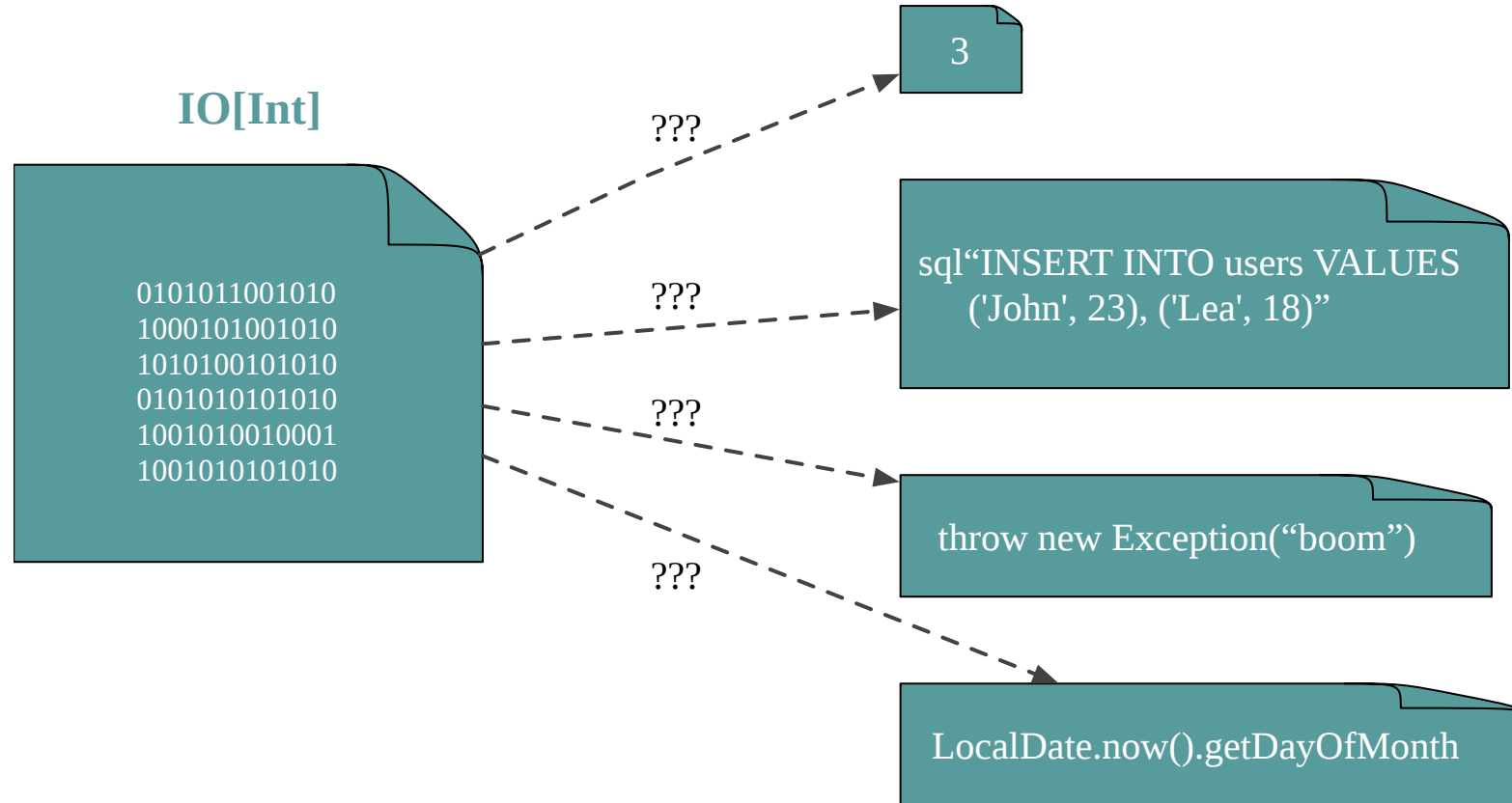
```
def deleteCard(userId: UserId): IO[DebitCard] = ???

val httpRoute = {
  case DELETE -> Root / "user" / UserId(x) / "card" =>
    deleteCard(x)
      .flatMap(Ok(_))
      .handleErrorWith {
        case _: UserMissing | _: CardMissing => NotFound()
        case _: ExpiredCard                => BadRequest()
        case _: Throwable                  => InternalServerError()
      }
}
```

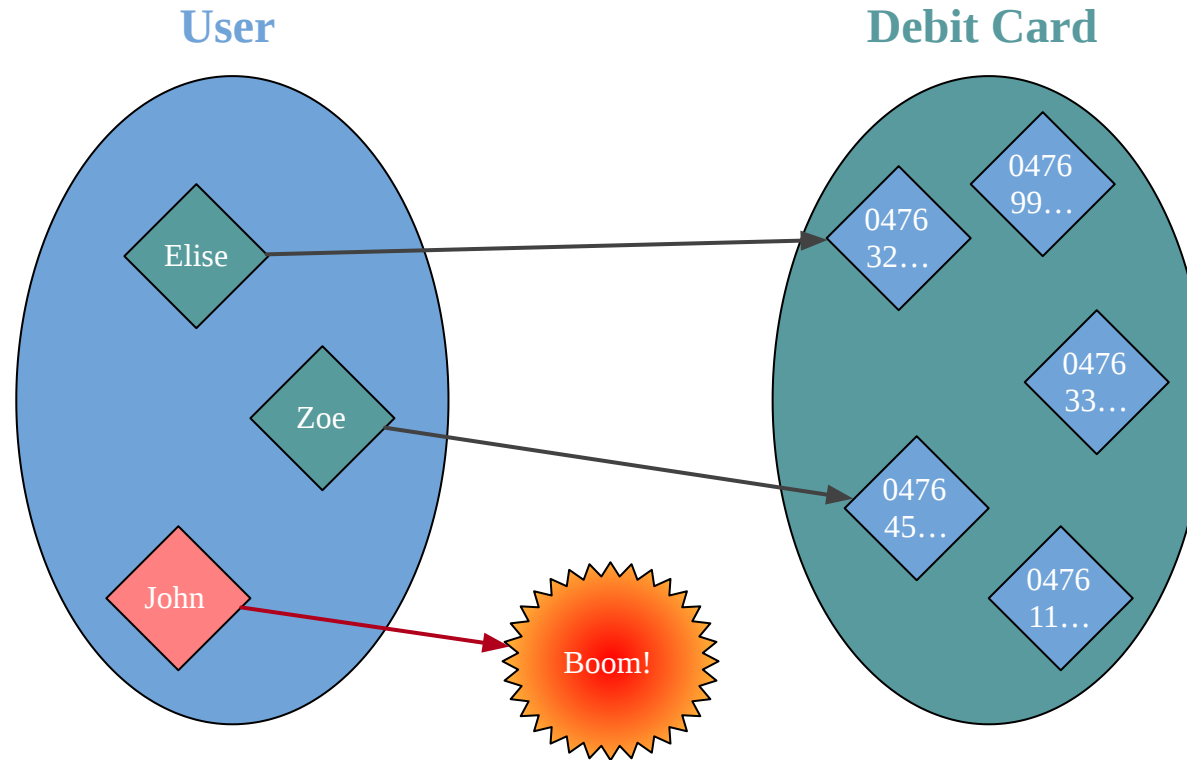
```
def handleErrorWith[A, B](io: IO[A])(f: Throwable => IO[B]): IO[B] = ???
```



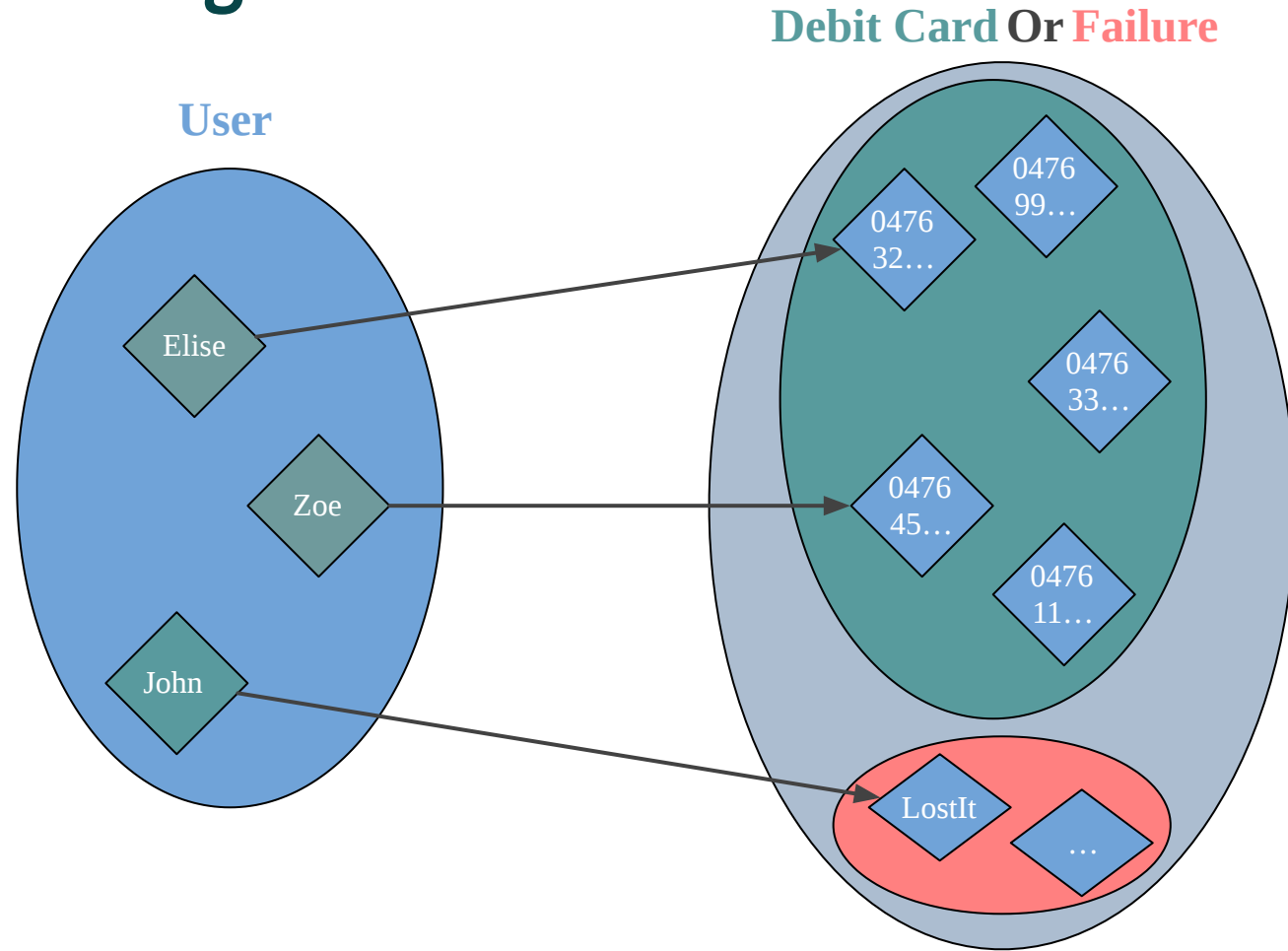
# IO can be too many things



# Can we use a type dedicated to error handling?



# Pure Error Handling





Which types can we use?



# Type constructors with an error channel

- Option
- Try
- Either



# Plan

- Look at use cases for `Option`, `Try` and `Either`
- Practice the design of error ADTs (Algebraic Data Type)
- How to use `Option` and `Either` in conjunction with `IO`

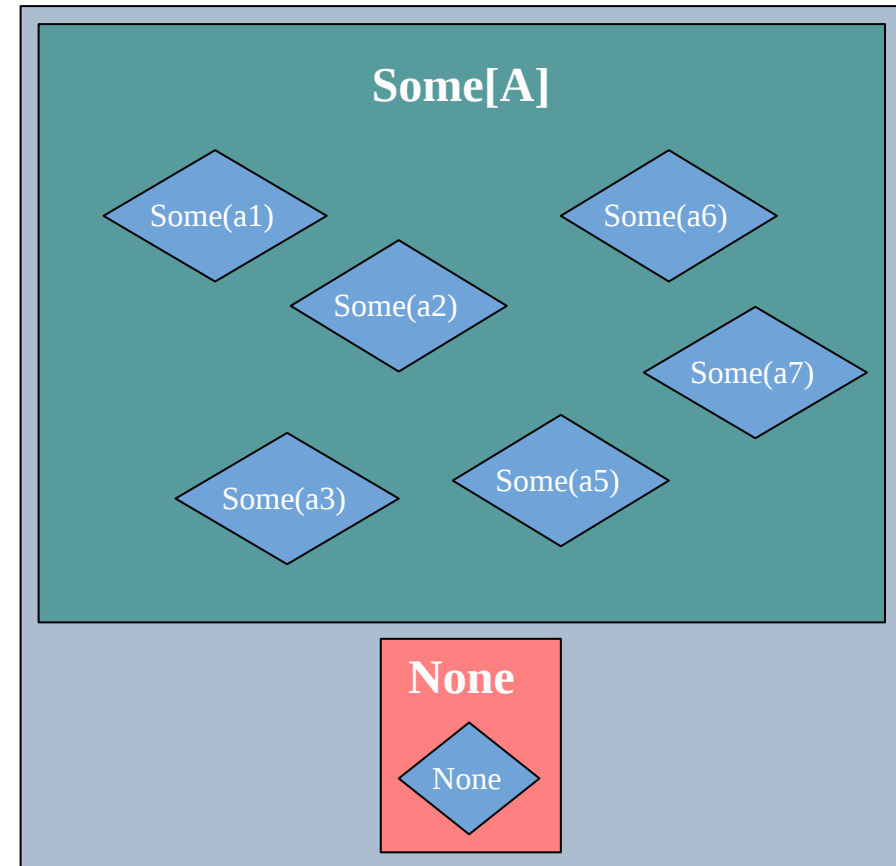


# Option

```
sealed trait Option[+A]

object Option {
  case class Some[+A](value: A) extends Option[A]
  case object None extends Option[Nothing]
}
```

Option[A]



# Option documents which values are optional

```
case class User(  
  id      : java.util.UUID,  
  name    : String,  
  age     : Int,  
  email   : Option[String],  
  address: Option[String]  
)
```

```
CREATE TABLE users (  
  id      UUID NOT NULL,  
  name    TEXT NOT NULL,  
  age     INT  NOT NULL,  
  email   TEXT,  
  address TEXT  
)
```



# Option forces us to think about empty case

```
def longest(xs: List[String]): Option[String] = {  
  var current: Option[String] = None  
  
  for (x <- xs) {  
    current match {  
      case Some(max) if max.length > x.length =>  
        () // do nothing  
      case _ =>  
        current = Some(x)  
    }  
  }  
  
  current  
}
```

```
def longest(xs: List[String]): String = {  
  var current: String = null  
  
  for (x <- xs) {  
    if(current != null && current.length > x.length) {  
      () // do nothing  
    } else {  
      current = x  
    }  
  }  
  
  current  
}
```



# Option is a List with at most one element

```
scala> Some("hello").toList  
res3: List[String] = List(hello)  
  
scala> None.toList  
res4: List[Nothing] = List()  
  
scala> List(Some(3), None, Some(4), None, None, Some(5)).flatMap(_.toList)  
res5: List[Int] = List(3, 4, 5)
```



# Option Exercise 1

`exercises.errorhandling.OptionExercises.scala`





# Variance

```
sealed trait Option[+A]  
object Option {  
  case class Some[+A](value: A) extends Option[A]  
  case object None extends Option[Nothing]  
}
```



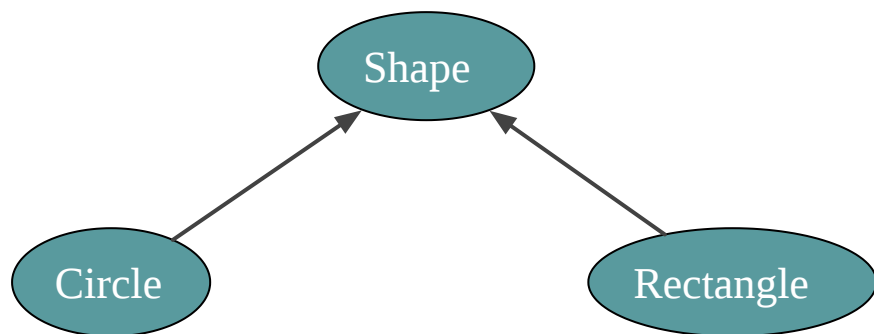
# Variance

```
sealed trait Option[+A]
object Option {
  case class Some[+A](value: A) extends Option[A]
  case object None extends Option[Nothing]
}
```

```
trait Foo[+A] // Foo is covariant
trait Foo[-A] // Foo is contravariant
trait Foo[A] // Foo is invariant
```



# Shape



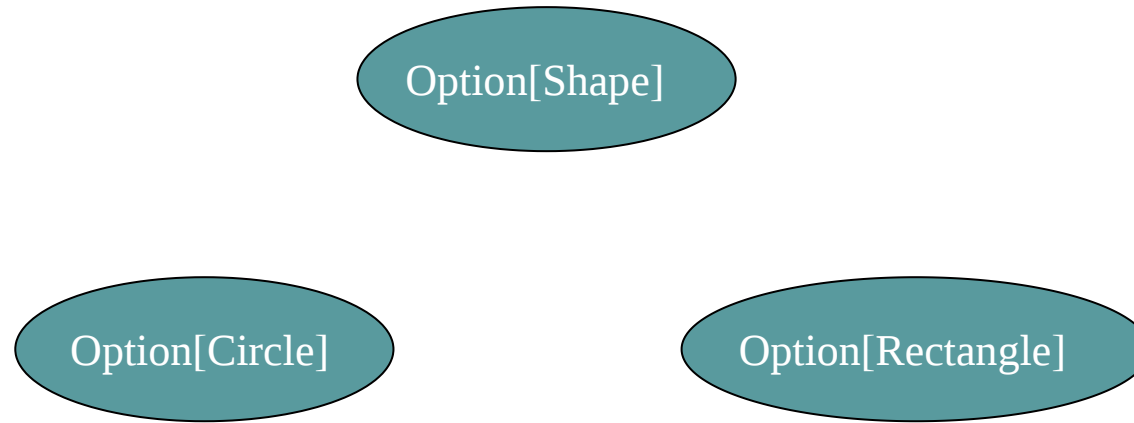
```
sealed trait Shape
```

```
case class Circle(radius: Int) extends Shape
```

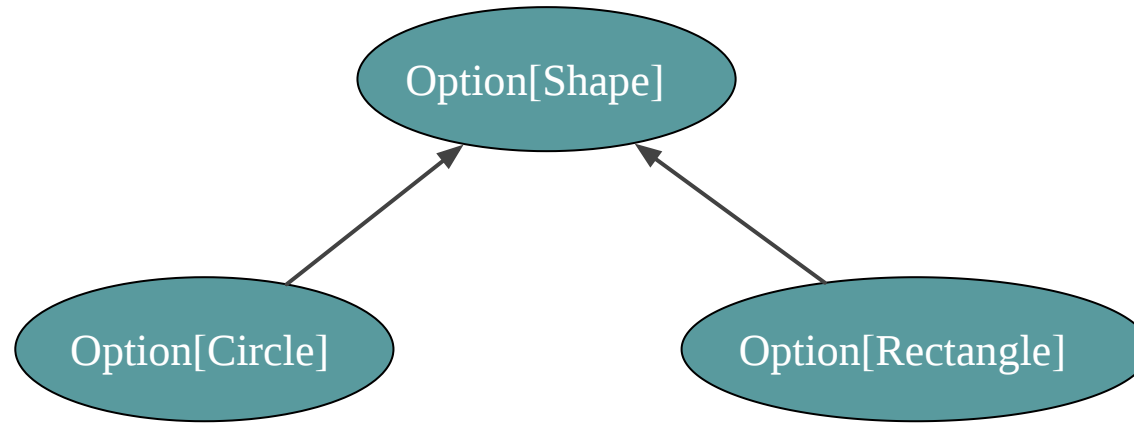
```
case class Rectangle(width: Int, height: Int) extends Shape
```



# What is the relationship for Option[Shape]?



# Option is covariant



# If Option were invariant

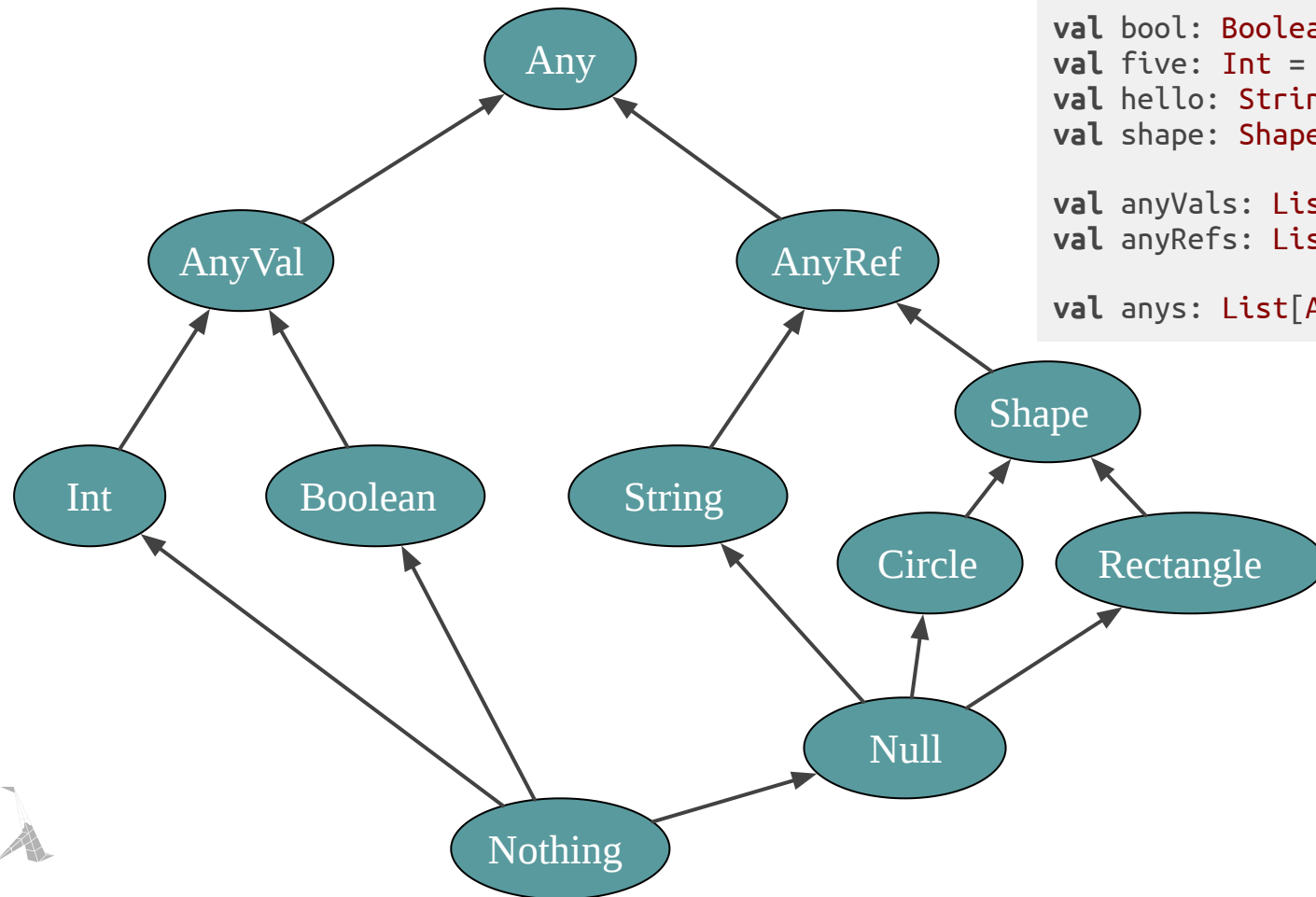
Option[Circle]

Option[Shape]

Option[Rectangle]



# Type hierarchy



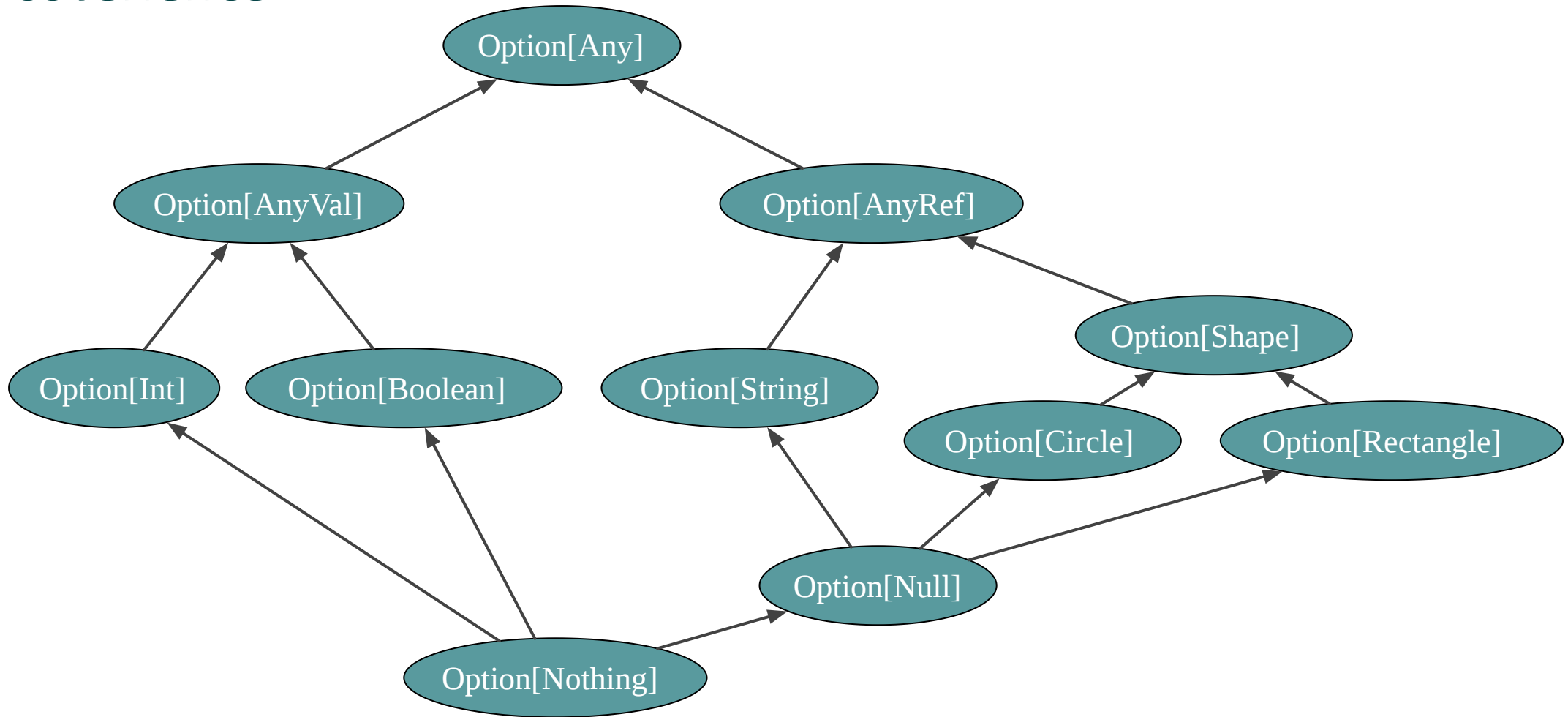
```
val bool: Boolean = true
val five: Int = 5
val hello: String = "Hello"
val shape: Shape = Circle(5)

val anyVals: List[AnyVal] = List(bool, five)
val anyRefs: List[AnyRef] = List(hello, shape, null)

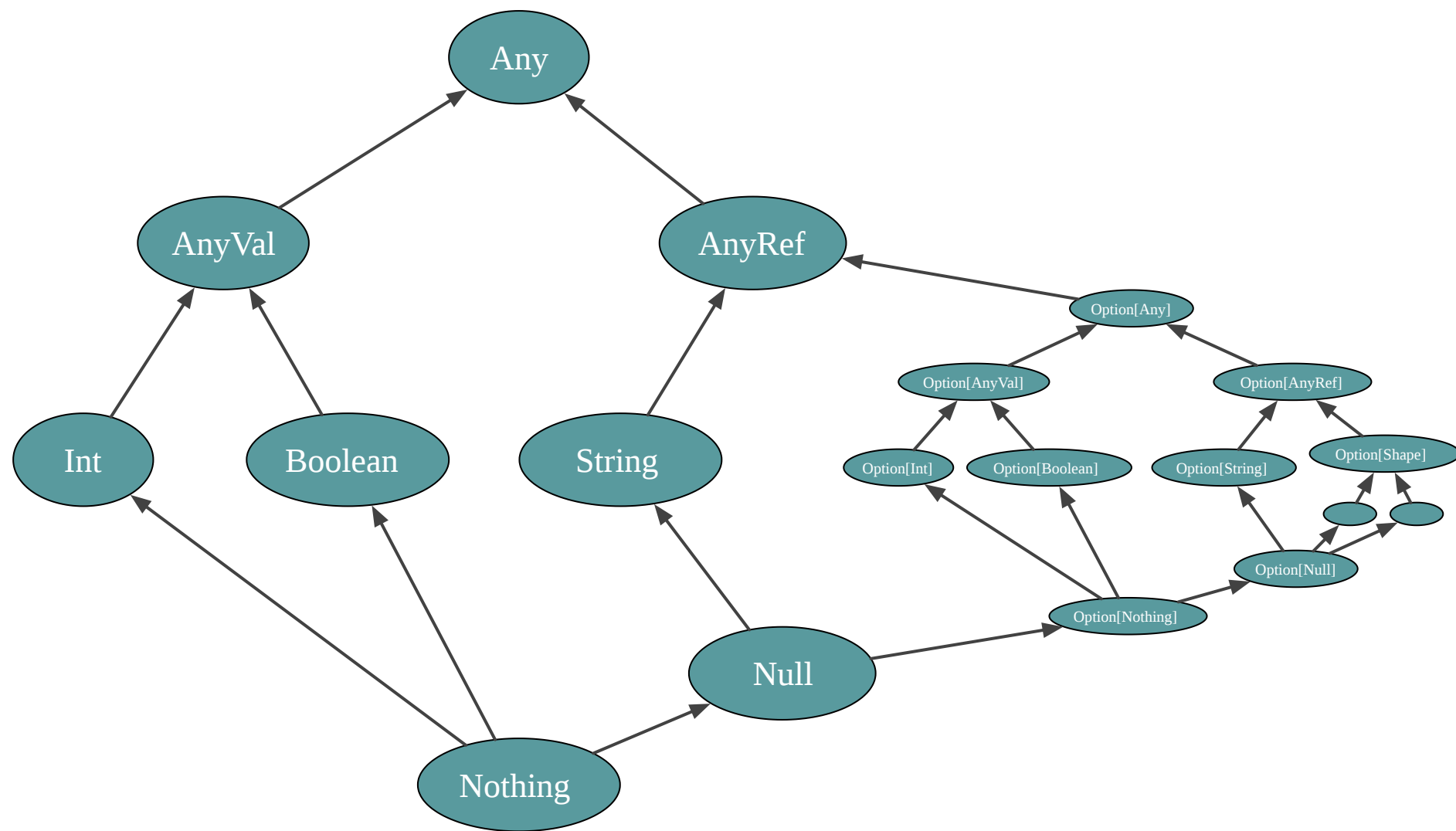
val anys: List[Any] = anyVals ++ anyRefs
```



# Covariance







# Variance is type checked

```
sealed trait Option[-A]

object Option {
  case class Some[-A](value: A) extends Option[A]
  case object None extends Option[Nothing]
}
```

```
scala> contravariant type A occurs in covariant position in type => A of value value
case class Some[-A](value: A) extends Option[A]
```



# Option Exercises 2, 3 and 4

`exercises.errorhandling.OptionExercises.scala`



# Use Option when

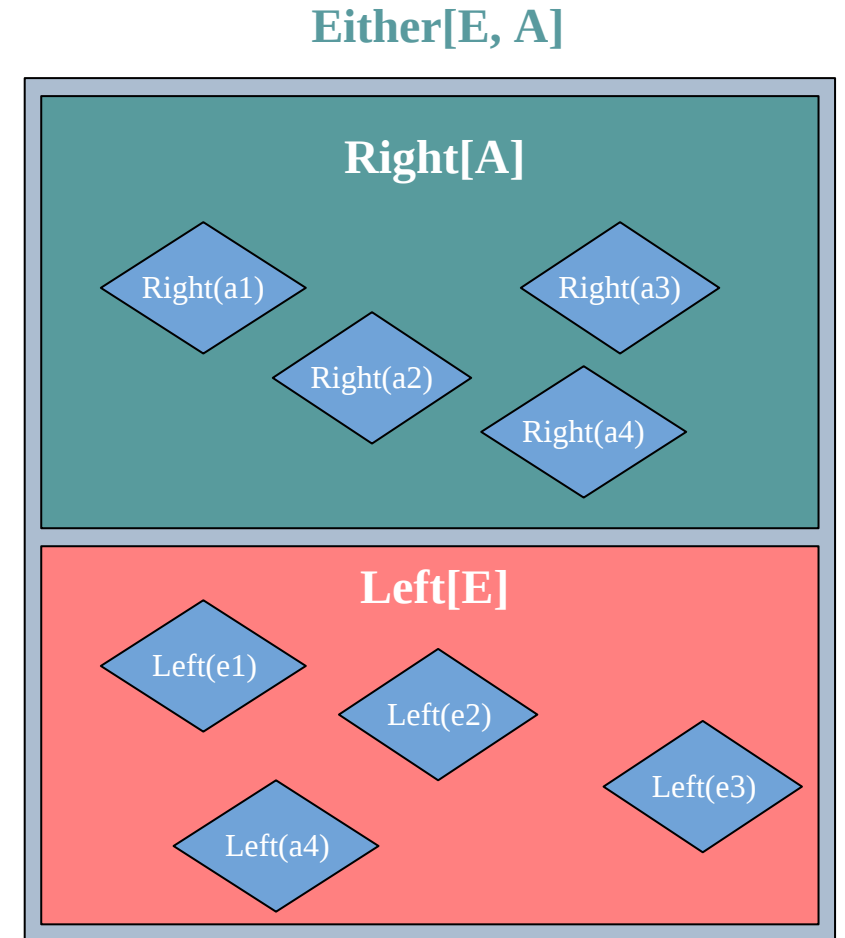
- A value may be missing
- An operation can fail in a unique obvious way
- An operation can fail in many ways but we don't need any information about the error



# Either

```
sealed trait Either[+E, +A]

object Either {
  case class Left[+E](value: E) extends Either[E, Nothing]
  case class Right[+A](value: A) extends Either[Nothing, A]
}
```



# Either is the canonical encoding of OR

```
def getUser(userIdOrEmail: Either[UserId, Email]): IO[User] =  
  userIdOrEmail match {  
    case Left(userId) => db.getUserById(userId)  
    case Right(email) => db.getUserByEmail(email)  
  }
```

`Either[UserId, Email]` represents a `UserId` OR an `Email`



# Either is the canonical encoding of OR

```
def getUser(userIdOrEmail: Either[UserId, Email]): IO[User] =  
  userIdOrEmail match {  
    case Left(userId) => db.getUserById(userId)  
    case Right(email) => db.getUserByEmail(email)  
  }
```

`Either[UserId, Email]` represents a `UserId` OR an `Email`

How would you encode a `UserId` AND an `Email`?



# Either is the canonical encoding of OR

```
def getUser(userIdOrEmail: Either[UserId, Email]): IO[User] =  
  userIdOrEmail match {  
    case Left(userId) => db.getUserById(userId)  
    case Right(email) => db.getUserByEmail(email)  
  }
```

`Either[UserId, Email]` represents a `UserId` OR an `Email`

`(UserId, Email)` represents a `UserId` AND an `Email`





Either[???, A]



# String Error

```
def submit(order: Order): Either[String, Order] =  
  order.status match {  
    case "Draft" =>  
      if(order.basket.isEmpty) Left("Basket is empty")  
      else Right(order.copy(status = "Submitted"))  
    case _ =>  
      Left(s"Cannot submit an order in ${order.status}")  
  }
```

```
scala> submit(Order("Draft", List(Item(111, 12.25, 2))))  
res6: Either[String,Order] = Right(Order(Submitted,List(Item(111,12.25,2))))
```

```
scala> submit(Order("Draft", Nil))  
res7: Either[String,Order] = Left(Basket is empty)
```

```
scala> submit(Order("Delivered", Nil))  
res8: Either[String,Order] = Left(Cannot submit an order in Delivered)
```



# Enum Error

```
sealed trait OrderError
case object EmptyBasketError extends OrderError
case class InvalidAction(action: String, status: String) extends OrderError

def submit(order: Order): Either[OrderError, Order] =
  order.status match {
    case "Draft" =>
      if(order.basket.isEmpty) Left(EmptyBasketError)
      else Right(order.copy(status = "Submitted"))
    case _ =>
      Left(InvalidAction("submit", order.status))
  }
```

```
scala> submit(Order("Draft", List(Item(111, 12.25, 2))))
res9: Either[OrderError,Order] = Right(Order(Submitted,List(Item(111,12.25,2))))

scala> submit(Order("Draft", Nil))
res10: Either[OrderError,Order] = Left(EmptyBasketError)

scala> submit(Order("Delivered", Nil))
res11: Either[OrderError,Order] = Left(InvalidAction(submit,Delivered))
```



# Enum Error

```
def canSubmit(order: Order): Boolean =  
  submit(order) match {  
    case Right(_)           => true  
    case Left(EmptyBasket)  => false  
    case Left(_: InvalidAction) => false  
  }
```



# Enum Error

```
def canSubmit(order: Order): Boolean =  
  submit(order) match {  
    case Right(_)           => true  
    case Left(_: InvalidAction) => false  
  }
```

On line 3: warning: **match** may not be exhaustive.  
It would fail on the following input: **Left(EmptyBasketError)**



# Throwable Error

```
import java.time.LocalDate
import java.time.format.DateTimeFormatter
import scala.util.Try

val formatter = DateTimeFormatter.ofPattern("uuuu-MM-dd")

def parseLocalDate(dateStr: String): Either[Throwable, LocalDate] =
  Try(LocalDate.parse(dateStr, formatter)).toEither
```

```
scala> parseLocalDate("2019-09-12")
res12: Either[Throwable, java.time.LocalDate] = Right(2019-09-12)

scala> parseLocalDate("12 July 1996")
res13: Either[Throwable, java.time.LocalDate] = Left(java.time.format.DateTimeParseException: Text '12 July 1996' could not be parsed to LocalDate)
```



Why do we use Left for error and Right for success?



It is completely arbitrary





# Either is Right biased

```
def map[E, A, B](either: Either[E, A])(f: A => B): Either[E, B] = ???
```

```
scala> parseLocalDate("2019-09-12").map(_.plusDays(2))  
res14: scala.util.Either[Throwable, java.time.LocalDate] = Right(2019-09-14)
```



# Either is Right biased

```
def map[E, A, B](either: Either[E, A])(f: A => B): Either[E, B] = ???
```

```
scala> parseLocalDate("2019-09-12").map(_.plusDays(2))  
res14: scala.util.Either[Throwable, java.time.LocalDate] = Right(2019-09-14)
```

```
def flatMap[E, A, B](either: Either[E, A])(f: A => Either[E, B]): Either[E, B] = ???
```

```
scala> for {  
  |   date1 <- parseLocalDate("2019-01-24")  
  |   date2 <- parseLocalDate("2020-09-12")  
  | } yield date1.isBefore(date2)  
res15: scala.util.Either[Throwable, Boolean] = Right(true)
```



# Either Exercises

`exercises.errorhandling.EitherExercises.scala`



Either is an Option with polymorphic error type



Option is a special case of Either

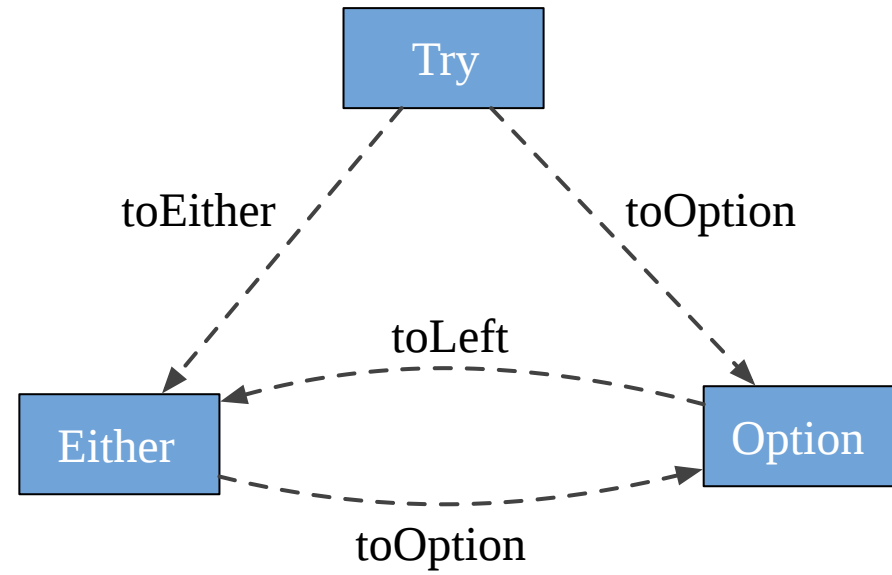


`type Option[+A] = Either[Unit, A]`



type Try[+A] = Either[Throwable, A]



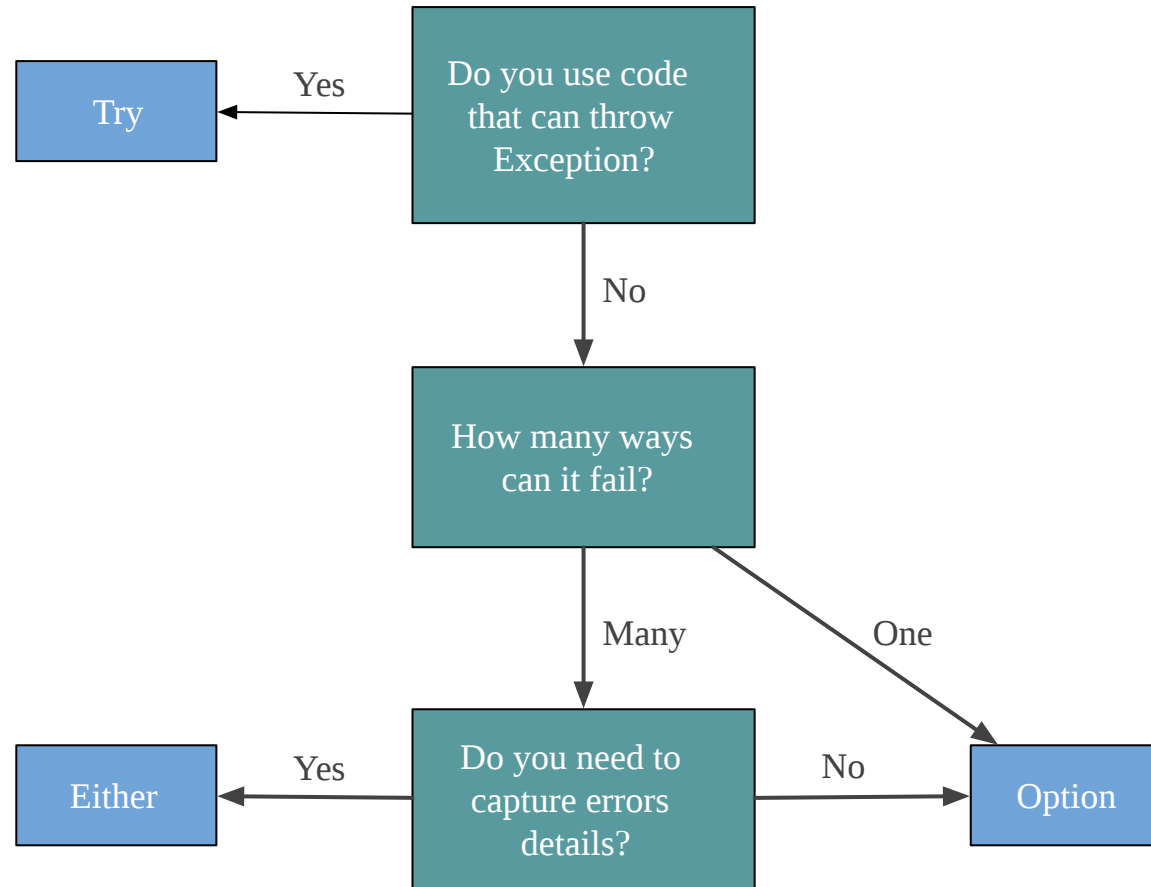


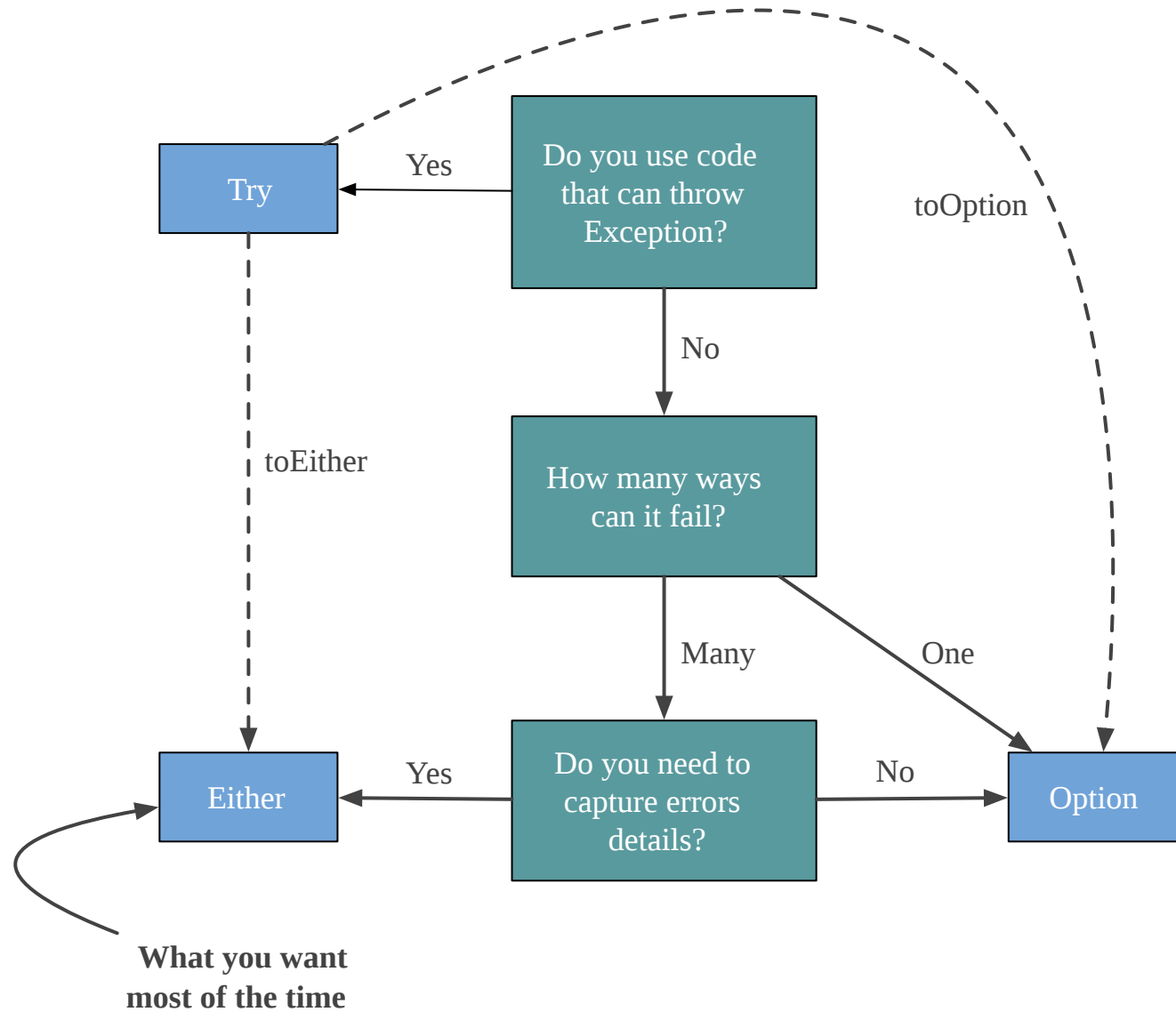


# Either Summary

- Use when you need to capture details about failure
- ADTs are generally the best way to encode errors
- Two modes:
  - Fail early with `flatMap`
  - Accumulate failures with `parMap`, `parSequence`







# Option with IO

```
trait OrderApi {  
  def getUser(userId: UserId): IO[Option[User]]  
  def getOrder(orderId: OrderId): IO[Option[Order]]  
}  
  
def getOrderDetails(api: OrderApi)(userId: UserId, orderId: OrderId): IO[OrderDetails] =  
  for {  
    optUser  <- api.getUser(userId)  
    optOrder <- api.getOrder(orderId)  
  } yield ???
```



# Option with IO

```
trait OrderApi {  
  def getUser(userId: UserId): IO[Option[User]]  
  def getOrder(orderId: OrderId): IO[Option[Order]]  
}  
  
def getOrderDetails(api: OrderApi)(userId: UserId, orderId: OrderId): IO[OrderDetails] =  
  for {  
    optUser  <- api.getUser(userId)  
    optOrder <- api.getOrder(orderId)  
    user     <- optUser match {  
      case None    => IO.fail(new Exception(s"User not found $userId"))  
      case Some(x) => IO.succeed(x)  
    }  
    order    <- optOrder match {  
      case None    => IO.fail(new Exception(s"Order not found $orderId"))  
      case Some(x) => IO.succeed(x)  
    }  
  } yield ???
```



# Option with IO

```
trait OrderApi {  
  def getUser(userId: UserId): IO[Option[User]]  
  def getOrder(orderId: OrderId): IO[Option[Order]]  
}  
  
def fromOption[A](opt: Option[A])(error: => Throwable): IO[A] =  
  opt match {  
    case None    => IO.fail(error)  
    case Some(x) => IO.succeed(x)  
  }  
  
def getOrderDetails(api: OrderApi)(userId: UserId, orderId: OrderId): IO[OrderDetails] =  
  for {  
    user  <- api.getUser(userId).flatMap(fromOption(_)(new Exception(s"User not found $userId")))  
    order <- api.getOrder(orderId).flatMap(fromOption(_)(new Exception(s"Order not found $orderId")))  
  } yield ???
```



# Option with IO

```
sealed trait ApplicationError extends Exception
case class UserNotFound(userId: UserId) extends ApplicationError
case class OrderNotFound(orderId: OrderId) extends ApplicationError

trait OrderApi {
  def getUser(userId: UserId): IO[Option[User]]
  def getOrder(orderId: OrderId): IO[Option[Order]]
}

def getOrderDetails(api: OrderApi)(userId: UserId, orderId: OrderId): IO[OrderDetails] =
  for {
    user  <- api.getUser(userId).flatMap(fromOption(_)(UserNotFound(userId)))
    order <- api.getOrder(orderId).flatMap(fromOption(_)(OrderNotFound(orderId)))
  } yield ???
```



# Option with IO

```
sealed trait ApplicationError extends Exception
case class UserNotFound(userId: UserId) extends ApplicationError
case class OrderNotFound(orderId: OrderId) extends ApplicationError

trait OrderApi {
  def getUserOpt(userId: UserId): IO[Option[User]]
  def getOrderOpt(orderId: OrderId): IO[Option[Order]]

  def getUser(userId: UserId): IO[User]      = getUserOpt(userId).flatMap(fromOption(_)(UserNotFound(userId)))
  def getOrder(orderId: OrderId): IO[Order] = getOrderOpt(orderId).flatMap(fromOption(_)(OrderNotFound(orderId)))
}

def getOrderDetails(api: OrderApi)(userId: UserId, orderId: OrderId): IO[OrderDetails] =
  for {
    user  <- api.getUser(userId)
    order <- api.getOrder(orderId)
  } yield ???
```





# OptionT

```
case class OptionT[+A](value: IO[Option[A]]) {  
  def map[B](f: A => B): OptionT[B] =  
    OptionT(value.map(_.map(f)))  
  
  def flatMap[B](f: A => OptionT[B]): OptionT[B] =  
    OptionT(value.flatMap {  
      case None    => IO.succeed(None)  
      case Some(a) => f(a).value  
    })  
}
```



# OptionT

```
case class OptionT[+A](value: IO[Option[A]]) {  
  def map[B](f: A => B): OptionT[B] =  
    OptionT(value.map(_.map(f)))  
  
  def flatMap[B](f: A => OptionT[B]): OptionT[B] =  
    OptionT(value.flatMap {  
      case None    => IO.succeed(None)  
      case Some(a) => f(a).value  
    })  
}
```

```
trait DbApi {  
  def getUser(userId: UserId): OptionT[User]  
  def getOrder(orderId: OrderId): OptionT[Order]  
}  
  
def getOrderDetails(db: DbApi)(userId: UserId, orderId: OrderId): OptionT[OrderDetails] =  
  for {  
    user  <- db.getUser(userId)  
    order <- db.getOrder(orderId)  
  } yield ???
```



# More general pattern

```
case class OptionT[F[+_], +A](value: F[Option[A]])
```

```
type IOOptionT[A] = OptionT[IO, A]
```

```
type ListOptionT[A] = OptionT[List, A]
```

```
case class EitherT[F[+_], +E, +A](value: F[Either[E, A]])
```

```
type IOEitherT[E, A] = EitherT[IO, E, A]
```

```
type ListEitherT[E, A] = EitherT[List, E, A]
```



# Conclusion

- Option and Either are the two main types to encode failures
- Try is a helpful tool to catch Exception
- Error ADTs can be as granular as we want
- Option and Either can be used in conjunction with IO



# Resources and further study

- [Scala Best Practices I Wish Someone'd Told Me About](#)



# Module 4: Type

