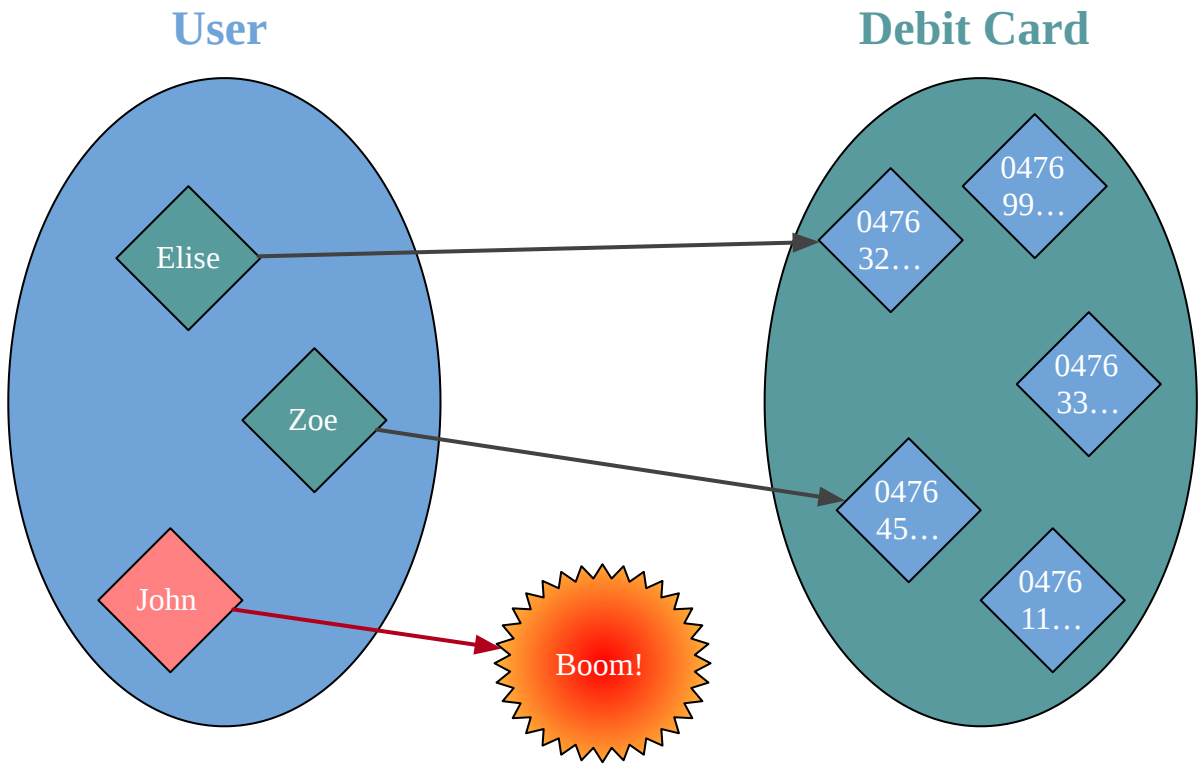
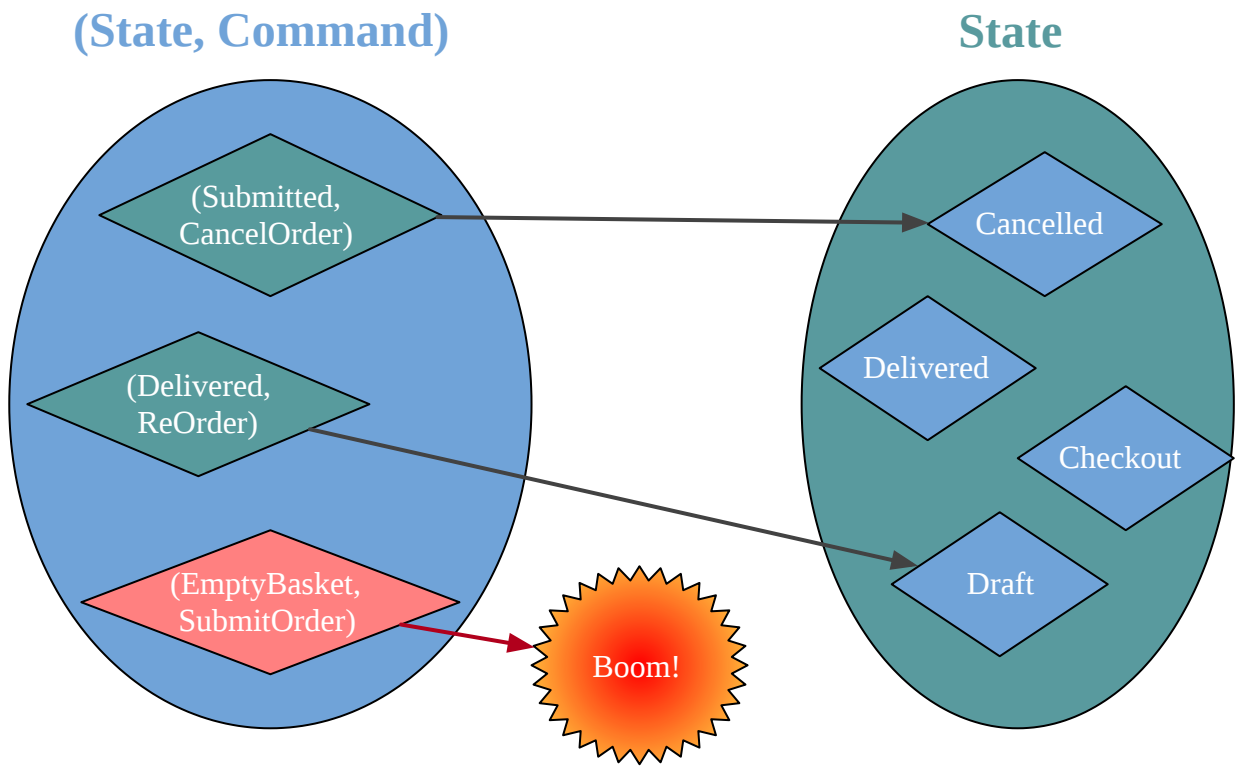
Error Handling

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
  }
```

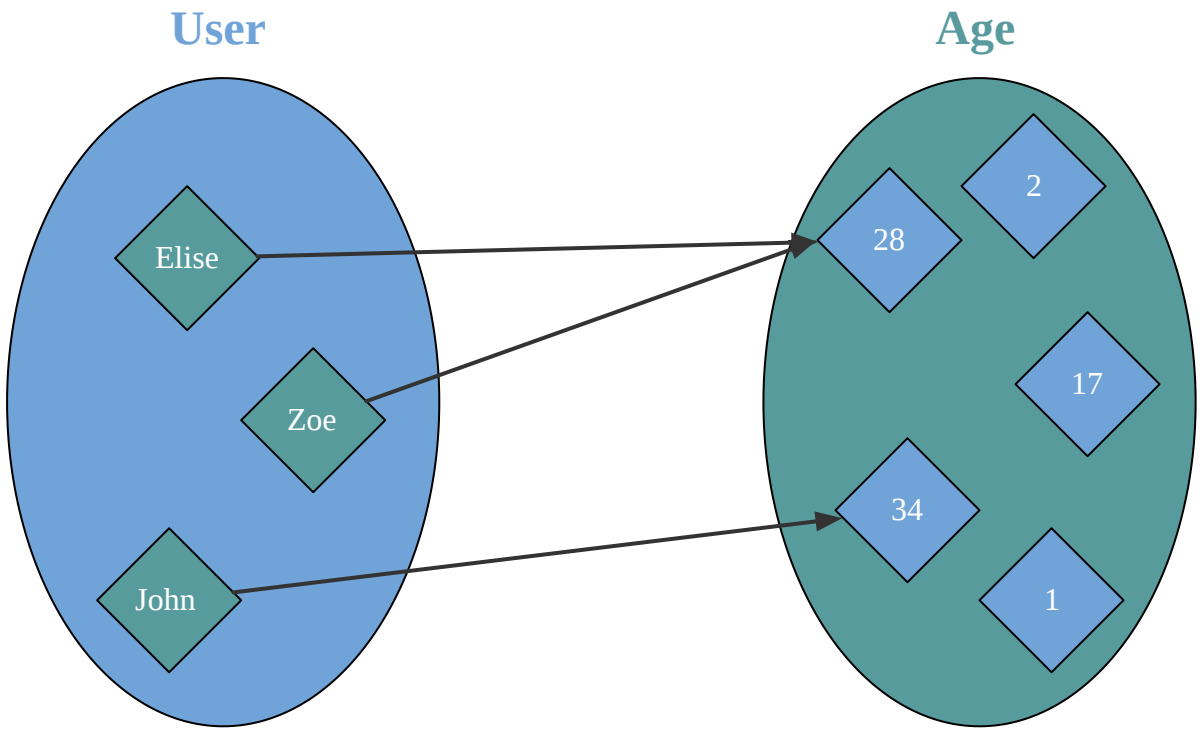
In Java, you have checked Exception

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

Functional subset

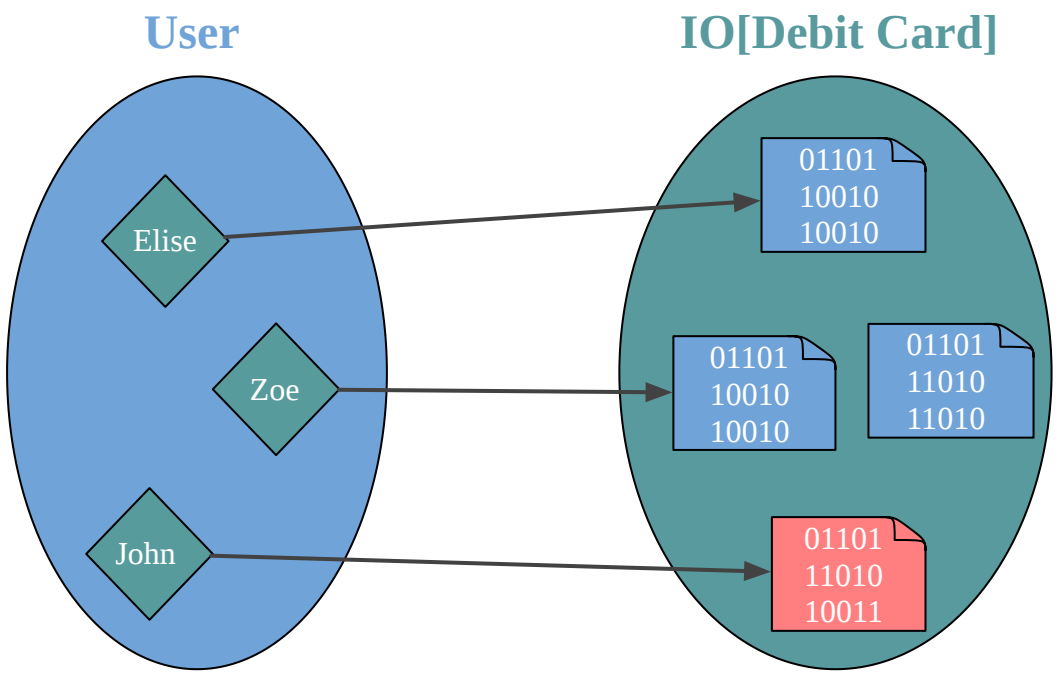
- Total
- No exception
- Deterministic
- No mutation
- No side effect
- No null
- No reflection

Functional subset



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

IO workaround



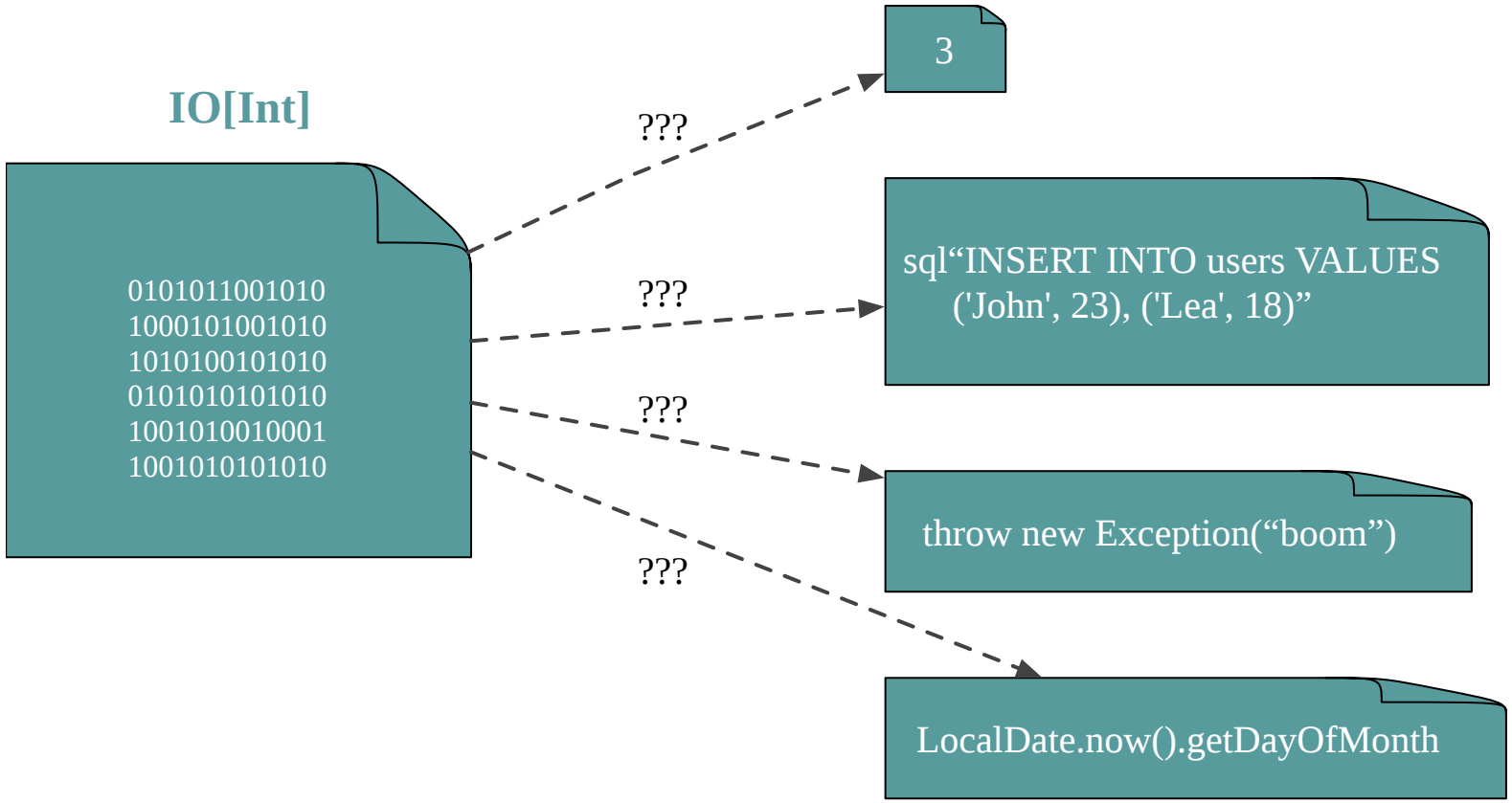

```
def getDebitCard(user: User): IO[DebitCard] = {
  if(user.debitCard == null)
    IO.fail(new Exception("No debit card"))
  else if (user.debitCard.hasExpired)
    IO.fail(new Exception("Expired debit card"))
  else
    IO.succeed(user.debitCard)
}
```

IO does not document errors

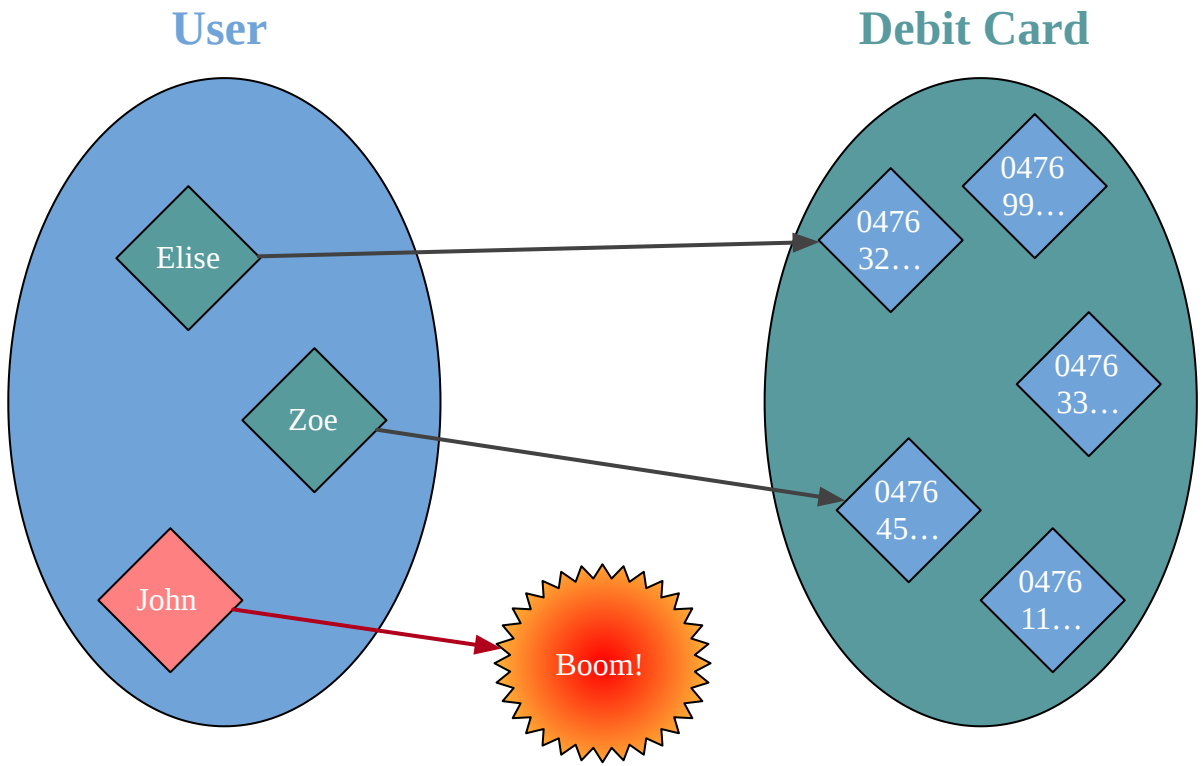
```
def deleteCard(userId: UserId): IO[DebitCard] = ???

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] = ???
```

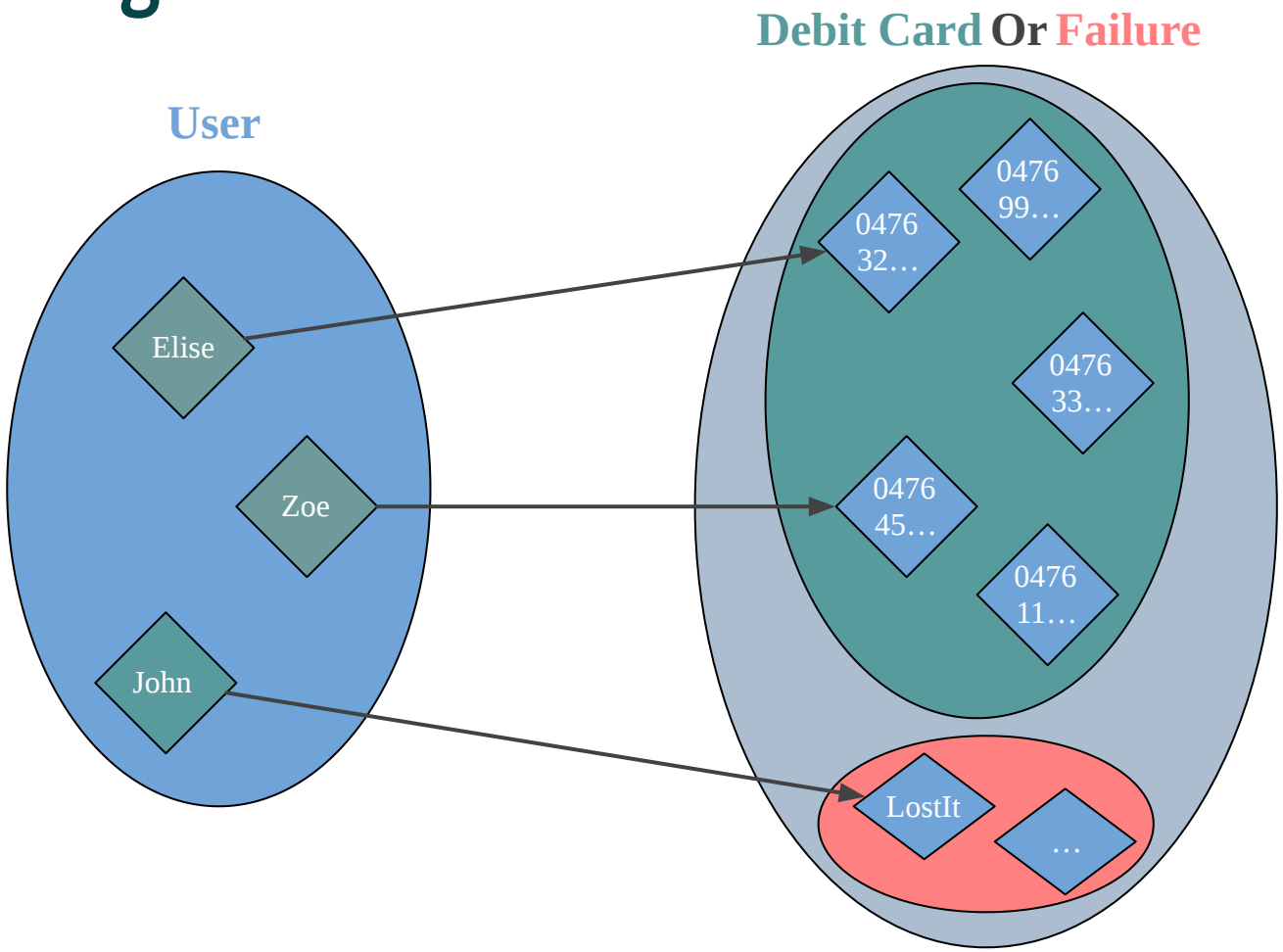
IO can be too many things



Can we do pure error handling without IO?



FP error handling



Types with an error channel

- `Option`
- `Try`
- `Either`

Plan

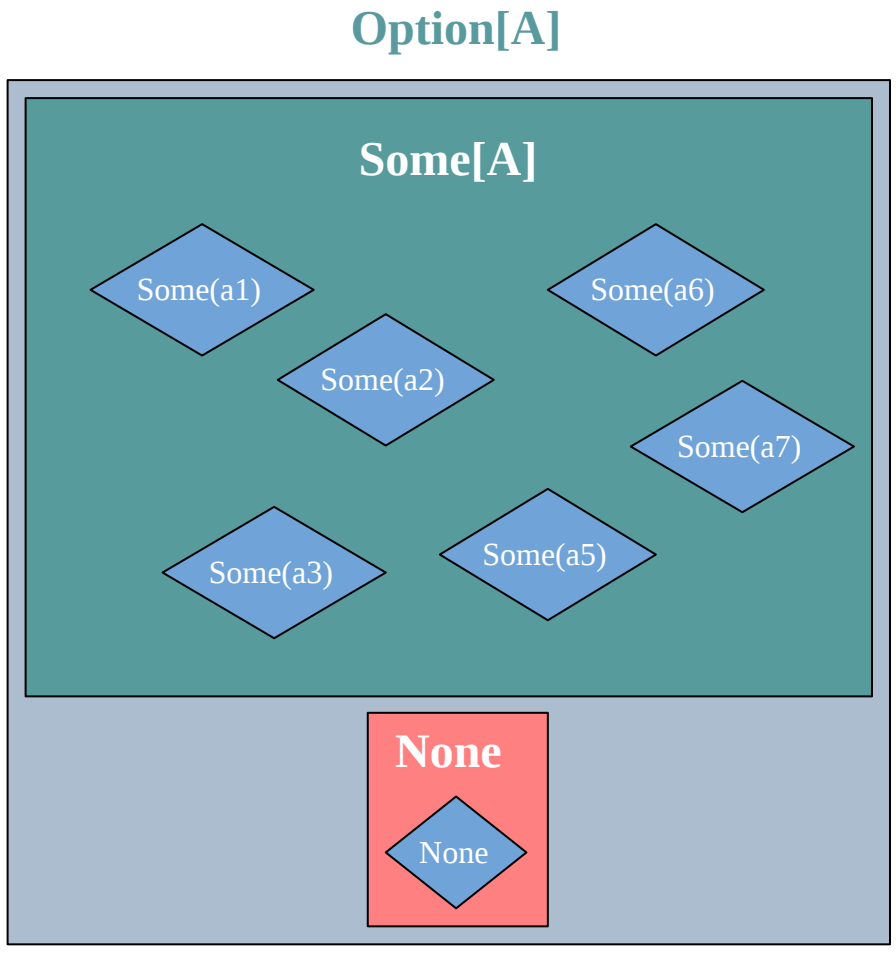
- Look at use cases for `Option`, `Try` and `Either`
- Practice the design of error types
- How to use `Option` and `Either` in conjunction with `IO`

Option

```scala
sealed trait Option[+A]

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

In Scala 3

```scala
enum Option[+A] {
  case Some(value: A)
  case None
}
```



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

```scala
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
}
```

```scala
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 digression

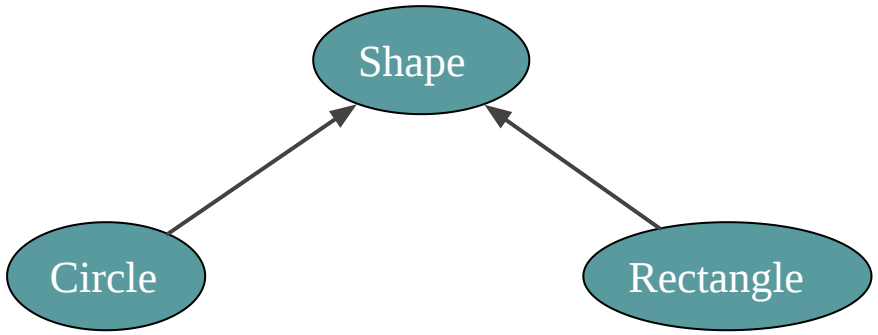
```
sealed trait Option[+A]

object Option {

  case class Some[+A](value: A) extends Option[A]

  case object None extends Option[Nothing]

}
```

Variance digression

```
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 is an enumeration




```
sealed trait Shape

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

What is the inferred type of circle?

```
val circle = Circle(12)
```

What is the inferred type of circle?

```
scala> val circle = Circle(12)
circle: Circle = Circle(12)
```

What is the inferred type of shapes?

```
val optCircle   : Option[Circle]    = Some(Circle(12))
val optRectangle: Option[Rectangle] = Some(Rectangle(5, 8))
```

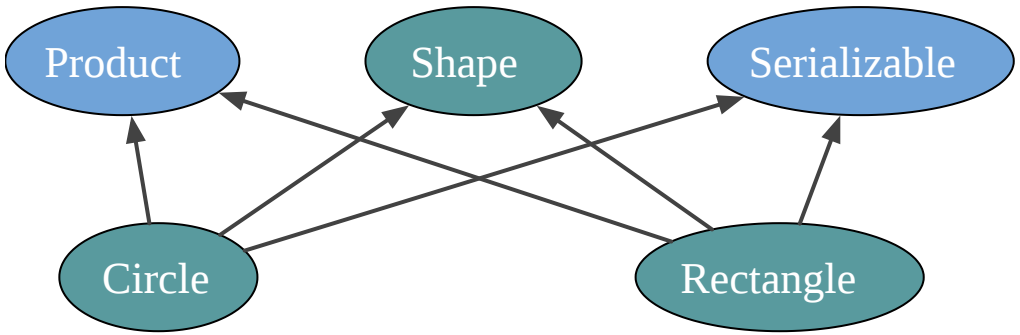
```
val shape = optCircle.orElse(optRectangle)
```

What is the inferred type of shape?

```
val optCircle   : Option[Circle]    = Some(Circle(12))
val optRectangle: Option[Rectangle] = Some(Rectangle(5, 8))
```

```
scala> val shape = optCircle.orElse(optRectangle)
shape: Option[Product with Shape with java.io.Serializable] = Some(Circle(12))
```

Shape




```
sealed trait Shape

case class Circle(radius: Int)                 extends Shape with Product with Serializable
case class Rectangle(width: Int, height: Int) extends Shape with Product with Serializable
```

Shape

```
sealed trait Shape extends Product with Serializable

case class Circle(radius: Int)                extends Shape with Product with Serializable
case class Rectangle(width: Int, height: Int) extends Shape with Product with Serializable
```

```
val optCircle   : Option[Circle]    = Some(Circle(12))
val optRectangle: Option[Rectangle] = Some(Rectangle(5, 8))
```

```
scala> val shape = optCircle.orElse(optRectangle)
shape: Option[Shape] = Some(Circle(12))
```

Shape

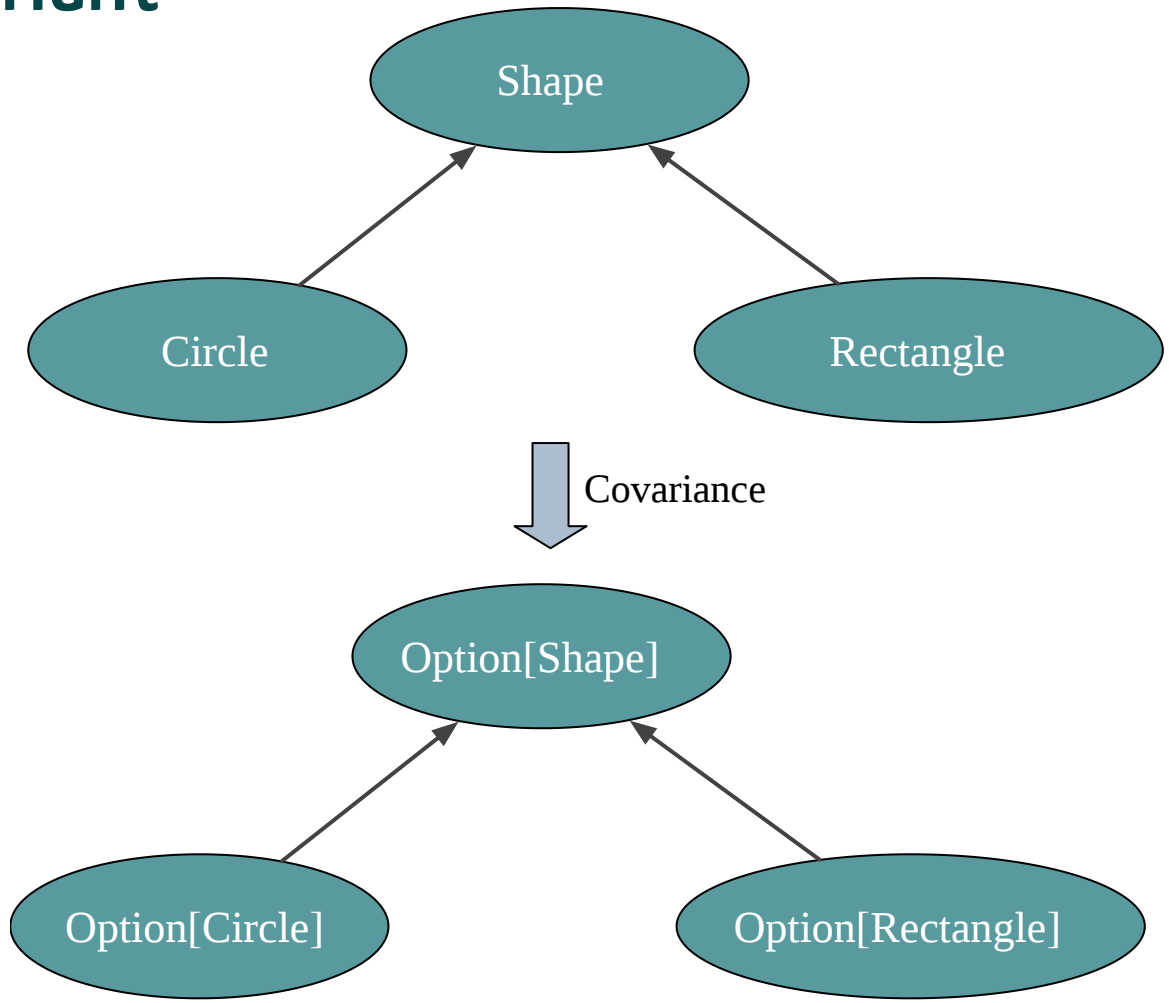
```
sealed trait Shape extends Product with Serializable

case class Circle(radius: Int)                extends Shape with Product with Serializable
case class Rectangle(width: Int, height: Int) extends Shape with Product with Serializable
```

```
val optCircle   : Option[Circle]    = Some(Circle(12))
val optRectangle: Option[Rectangle] = Some(Rectangle(5, 8))
```

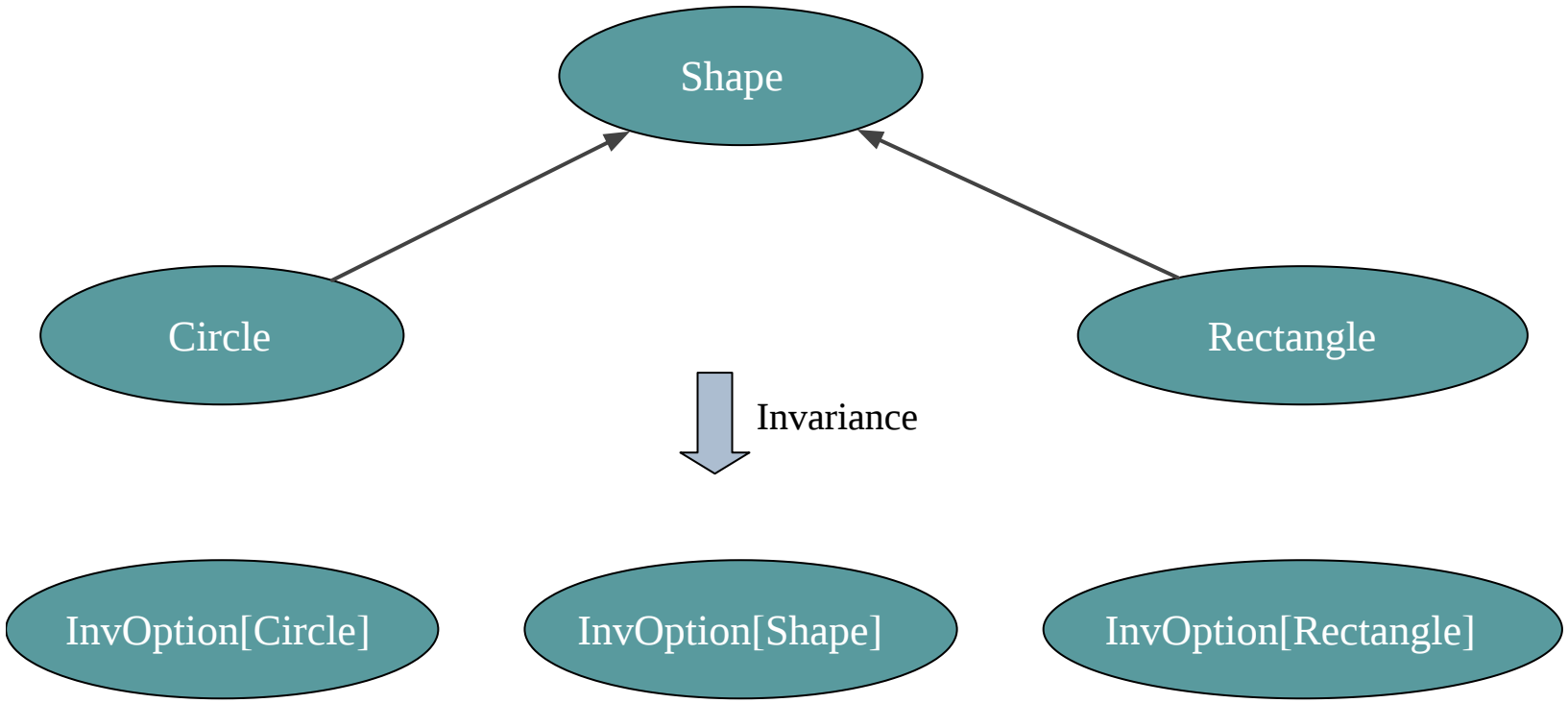
```
scala> val shape = optCircle.orElse(optRectangle)
shape: Option[Shape] = Some(Circle(12))
```

Why shape is an Option[Shape]?

Option is covariant



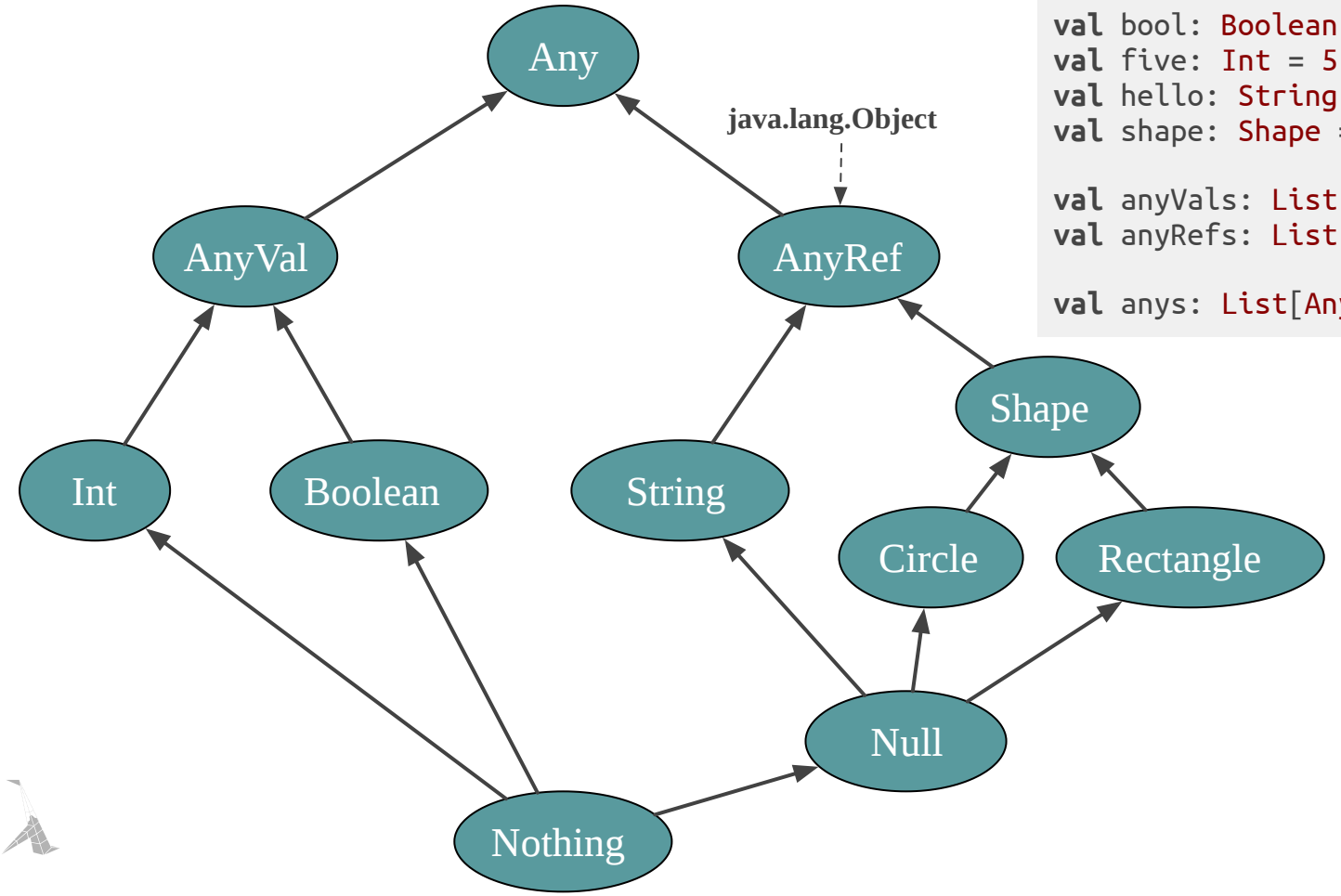
If Option were invariant



What about Nothing?

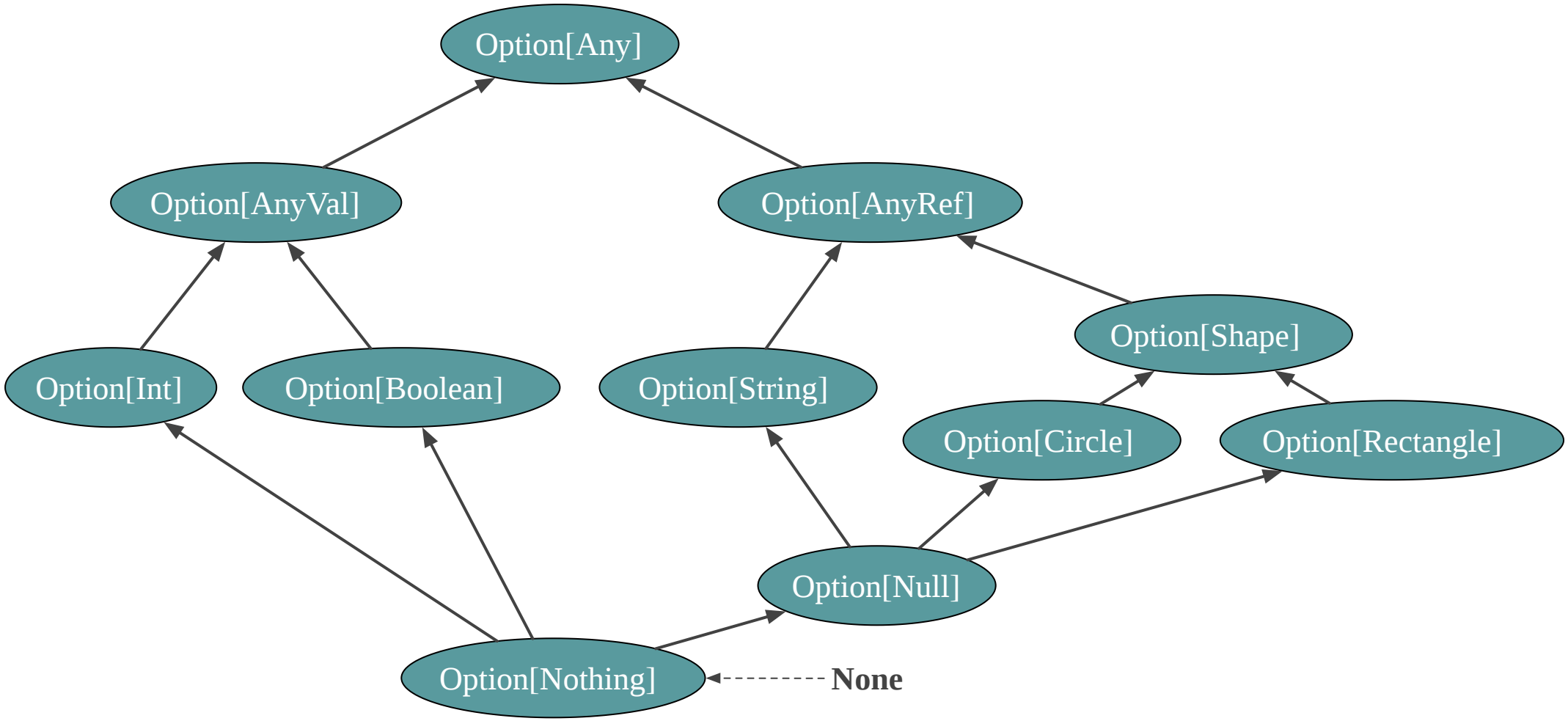
```
sealed trait Option[+A]

object Option {

  case class Some[+A](value: A) extends Option[A]

  case object None              extends Option[Nothing]

}
```

Scala Type hierarchy




```scala
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


Option[Any]
Option[AnyVal]
Option[AnyRef]
Option[Int]
Option[Boolean]
Option[String]
Option[Shape]
Option[Circle]
Option[Rectangle]
Option[Null]
Option[Nothing]
None

None

```
sealed trait Option[+A]

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

```
val optNothing: Option[Nothing] = None
val optInt    : Option[Int]     = None
val optCircle : Option[Circle]  = None
```

If Option were invariant

```
sealed trait InvOption[A]

object InvOption {
  case class Some[A](value: A) extends InvOption[A]
  case class None[A]()         extends InvOption[A]
}

val optInt   : InvOption[Int]   = InvOption.None()
val optCircle: InvOption[Circle]= InvOption.None()
```

Option Exercise 2

`exercises.errorhandling.OptionExercises.scala`

Variance rules

Type	A	B
(A, B)	Covariant	Covariant
Either[A, B]	Covariant	Covariant
A => B	Contravariant	Covariant
A => A	Invariant	N/A

Thinking with types by Sandy Maguire

What is the variance of JsonDecoder?

```
trait JsonDecoder[A]{
  def decode(value: Json): A
}
```

JsonDecoder is covariant

```
trait JsonDecoder[+A]{
  def decode(value: Json): A
}

val circleDecoder: JsonDecoder[Circle] = (value: Json) => ???
val shapeDecoder : JsonDecoder[Shape]  = circleDecoder
```

But

```
scala> val rectangleDecoder: JsonDecoder[Rectangle] = shapeDecoder
       val rectangleDecoder: JsonDecoder[Rectangle] = shapeDecoder
                                                      ^
On line 2: error: type mismatch;
        found   : JsonDecoder[Shape]
        required: JsonDecoder[Rectangle]
```

What is the variance of JsonEncoder?

```
trait JsonEncoder[A]{
  def encode(value: A): Json
}
```

JsonEncoder is contravariant

```
trait JsonEncoder[-A]{
  def encode(value: A): Json
}

val shapeDecoder : JsonEncoder[Shape]  = (value: Shape) => ???
val circleDecoder: JsonEncoder[Circle] = shapeDecoder
```

But

```
scala> val shapeDecoder2: JsonEncoder[Shape] = circleDecoder
       val shapeDecoder2: JsonEncoder[Shape] = circleDecoder
                                               ^
On line 2: error: type mismatch;
        found   : JsonEncoder[Circle]
        required: JsonEncoder[Shape]
```

What is the variance of JsonCodec?

```
trait JsonCodec[A] extends JsonDecoder[A] with JsonEncoder[A]
```

```
trait JsonCodec[A]{
  def decode(value: Json): A
  def encode(value: A   ): Json
}
```

JsonCodec is invariant

```
trait JsonCodec[A] extends JsonDecoder[A] with JsonEncoder[A]
```

```
trait JsonCodec[A]{
  def decode(value: Json): A
  def encode(value: A   ): Json
}
```

Variance summary

1. It is the only way to get an ergonomic API in Scala

2. Variance is type checked

```
scala> trait JsonEncoder[+A]{
     |   def encode(value: A): Json
     | }
         def encode(value: A): Json
                           ^
On line 3: error: covariant type A occurs in contravariant position in type A of value value
```

Option Exercises 3 and 4

`exercises.errorhandling.OptionExercises.scala`

Use Option when

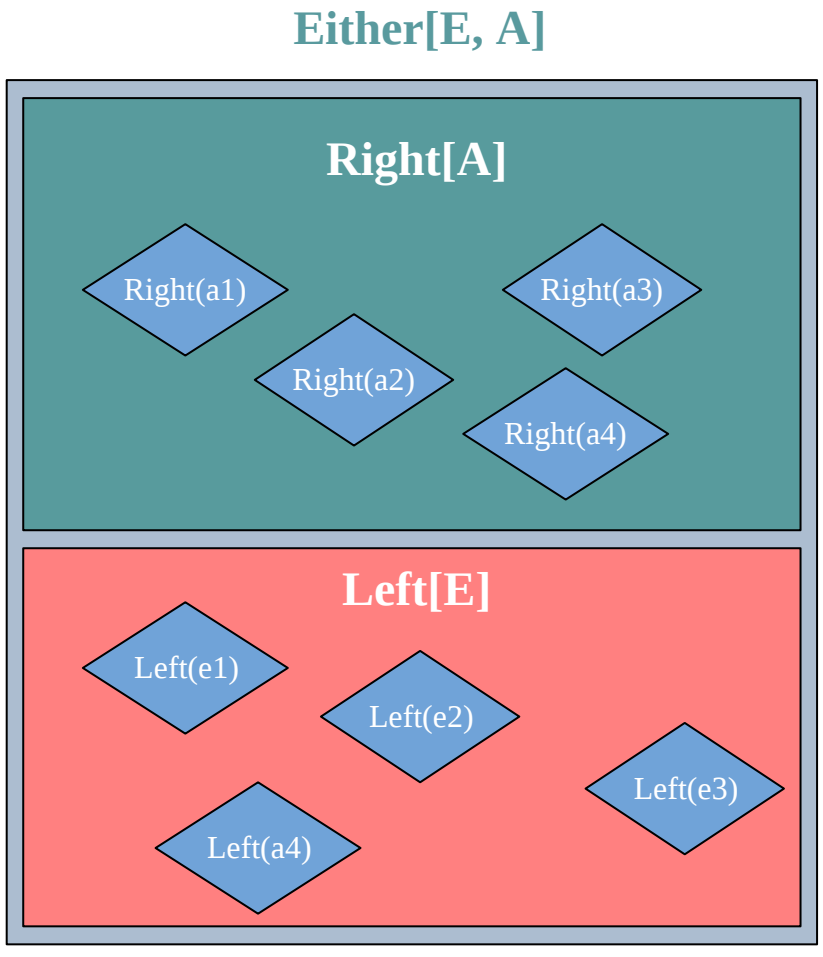
- A value may be missing
- An operation can fail in a unique obvious way
- An operation can fail in many ways but we don't need any information about the error

Either

```scala
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]
}
```

In Scala 3

```scala
enum Either[+E, +A] {
  case Left (value: E)
  case Right(value: 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' cou
```

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)
```

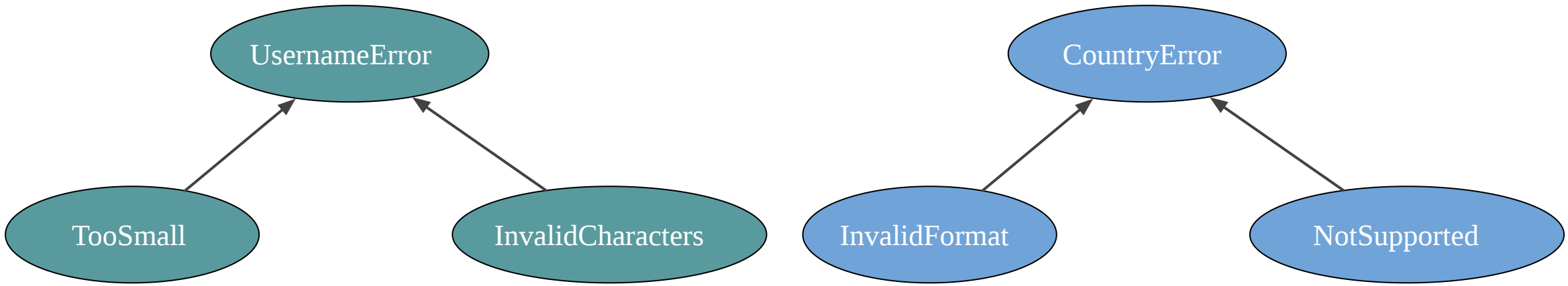
Either Exercises 1 to 3

`exercises.errorhandling.EitherExercises.scala`

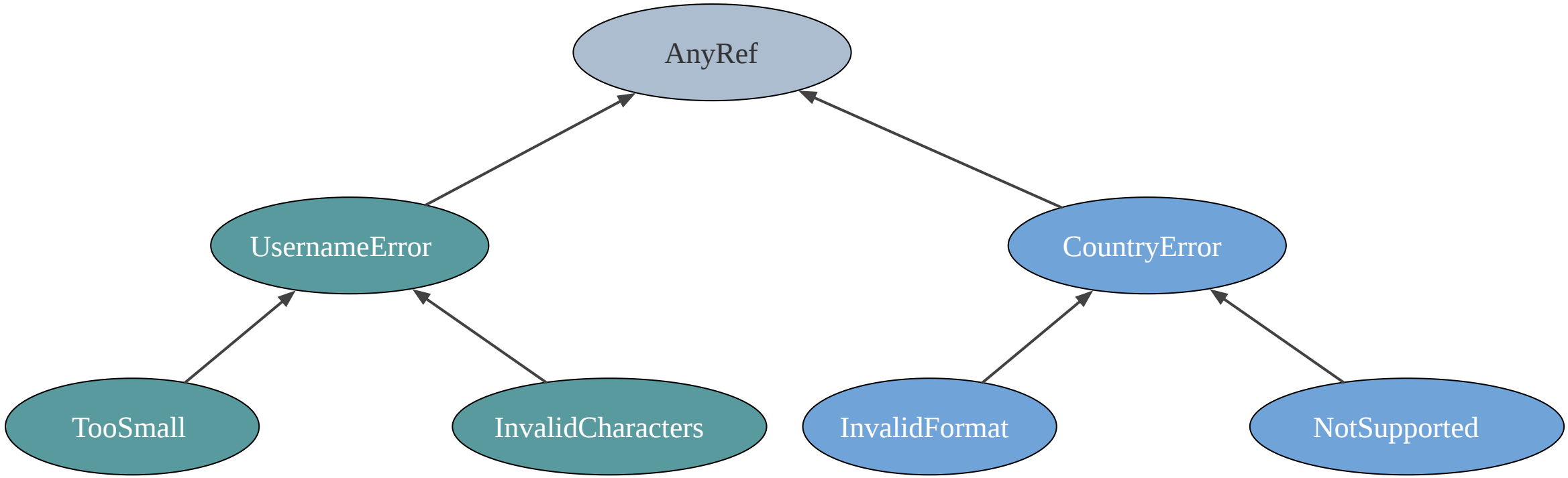
Error Hierarchy

```
sealed trait UsernameError
object UsernameError {
  case class TooSmall(length: Int)               extends UsernameError
  case class InvalidCharacters(char: List[Char]) extends UsernameError
}

sealed trait CountryError
object CountryError {
  case class InvalidFormat(country: String) extends CountryError
  case class NotSupported(country: String)  extends CountryError
}
```

Error Hierarchy



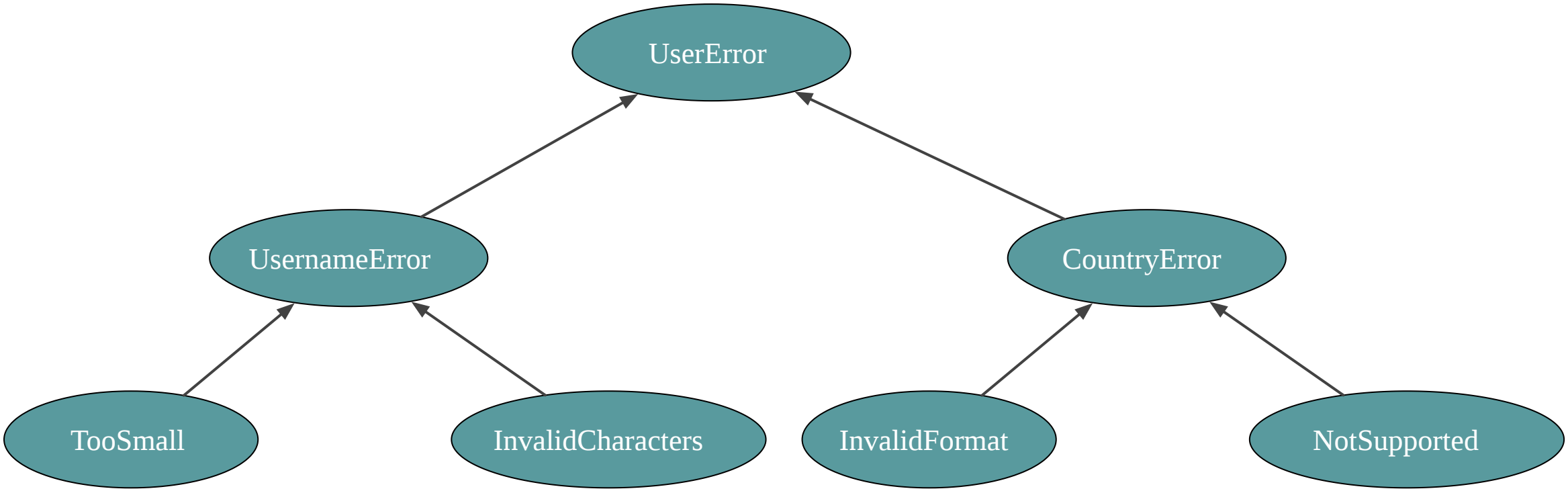
Error Hierarchy



Error Hierarchy

```
sealed trait UserError

sealed trait UsernameError extends UserError
object UsernameError {
  case class TooSmall(length: Int)              extends UsernameError
  case class InvalidCharacters(char: List[Char]) extends UsernameError
}

sealed trait CountryError extends UserError
object CountryError {
  case class InvalidFormat(country: String) extends CountryError
  case class NotSupported(country: String)  extends CountryError
}
```

Error Hierarchy



Union types in Scala 3

```
type UserError = UsernameError | CountryError | OtherError
```

Either Exercise 4

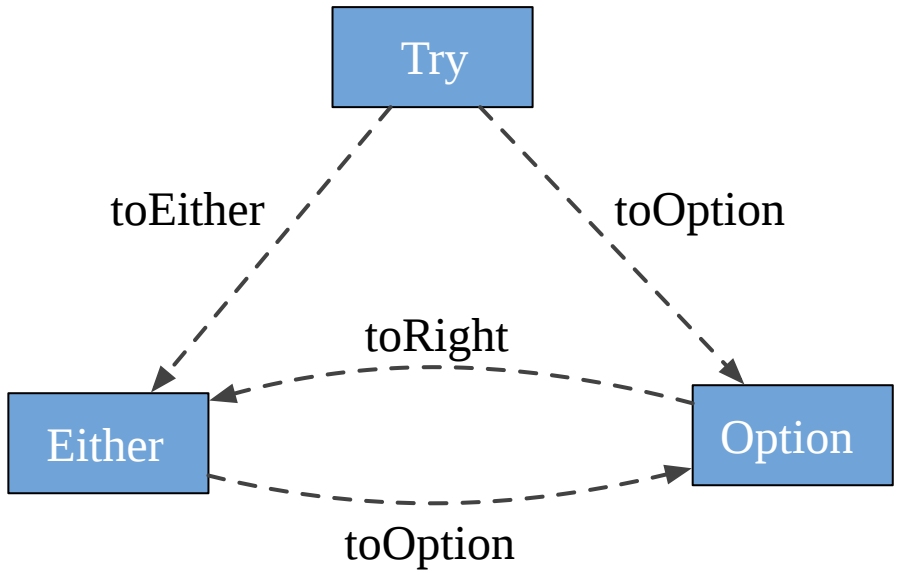
`exercises.errorhandling.EitherExercises.scala`

Either is an Option with polymorphic error type

Option is a special case of Either

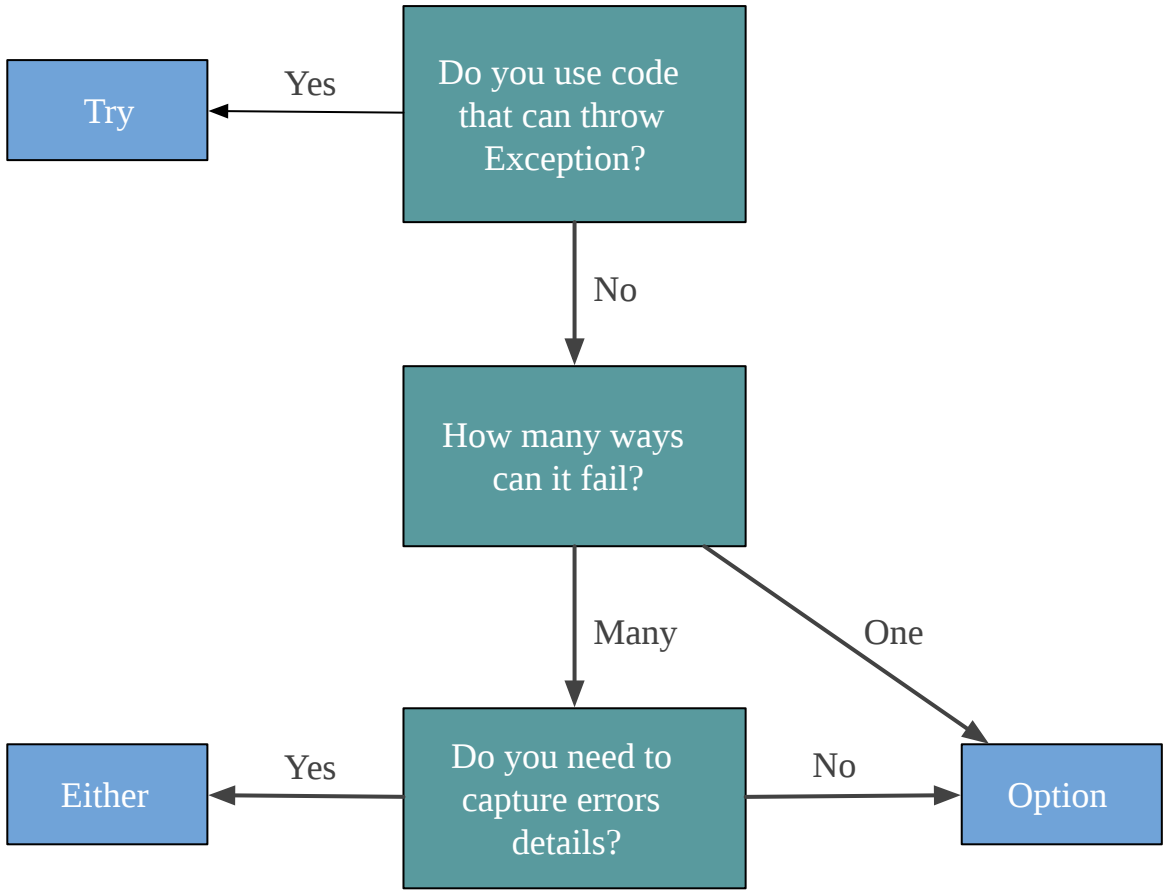
```
type Option[+A] = Either[Unit, A]
```

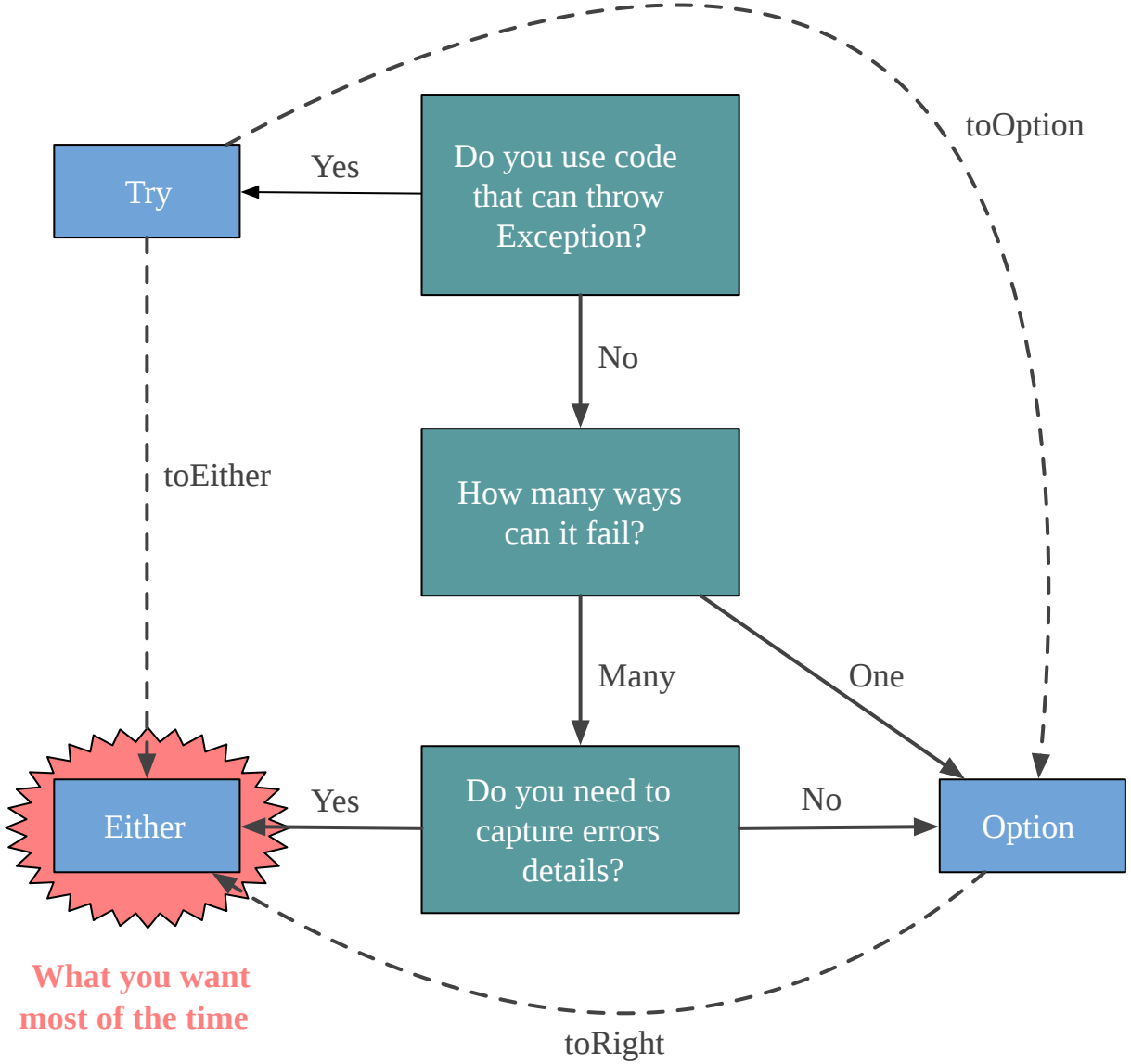
```
type Try[+A] = Either[Throwable, A]
```


Try
toEither
toOption
toRight
Either
Option
toOption

Either Summary

- Use when you need to capture details about failure
- Enums are generally the best way to encode errors
- Two modes:
  - Fail early with `flatMap`
  - Accumulate failures with `map2Acc`, `sequenceAcc`


Do you use code that can throw Exception?
Yes
Try
No
How many ways can it fail?
Many
One
Do you need to capture errors details?
Yes
Either
No
Option


toOption
Try
Yes
Do you use code that can throw Exception?
No
toEither
How many ways can it fail?
Many
One
Either
Yes
Do you need to capture errors details?
No
Option
What you want most of the time
toRight

IO with Option

```
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)
    order <- api.getOrder(orderId)
  } yield ???
```

What is the type of user and order?

IO with Option

```
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  /* Option[User]  */ <- api.getUser(userId)
    order /* Option[Order] */ <- api.getOrder(orderId)
  } yield ???
```

IO with Option

```
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 ???
```

IO with Option

```
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(a) => IO.succeed(a)
  }

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 ???
```

IO with Either

```
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[Either[UserNotFound , User]]
  def getOrder(orderId: OrderId): IO[Either[OrderNotFound, Order]]
}
```

IO with Either

```
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[Either[UserNotFound , User]]
  def getOrder(orderId: OrderId): IO[Either[OrderNotFound, Order]]
}
```

```
def failOnLeft[E <: Exception, A](io: IO[Either[E, A]]): IO[A] =
  io.flatMap {
    case Left(e)  => IO.fail(e)
    case Right(a) => IO.succeed(a)
  }
```

IO with Either

```
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[Either[UserNotFound , User]]
  def getOrder(orderId: OrderId): IO[Either[OrderNotFound, Order]]
}
```

```
def failOnLeft[E <: Exception, A](io: IO[Either[E, A]]): IO[A] =
  io.flatMap {
    case Left(e)  => IO.fail(e)
    case Right(a) => IO.succeed(a)
  }
```

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

Extension methods

```
implicit class IOEitherSyntax[E <: Exception, A](io: IO[Either[E, A]]){
  def failOnLeft: IO[A] =
    io.flatMap {
      case Left(e)  => IO.fail(e)
      case Right(a) => IO.succeed(a)
    }
}
```

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

Conclusion

- Option and Either are the two main types to encode failures
- Try is a helpful tool to catch Exception
- Enums can encode errors as precisely as we want (trade-offs)
- 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