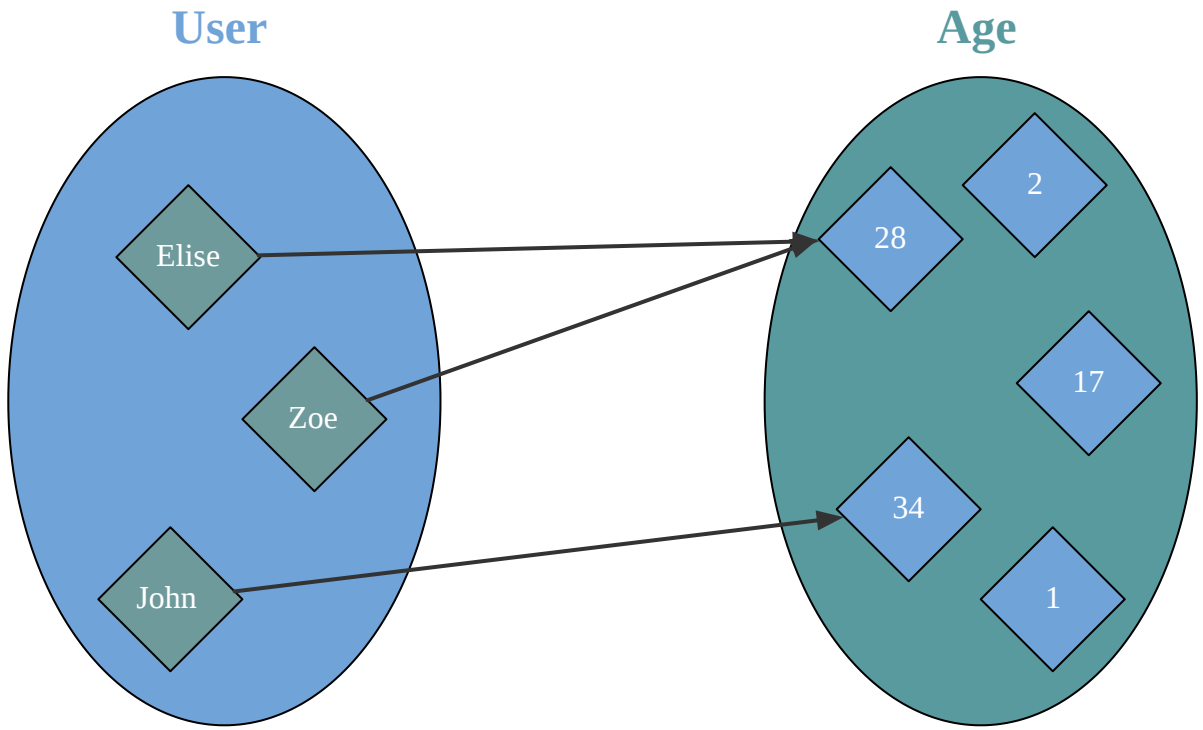
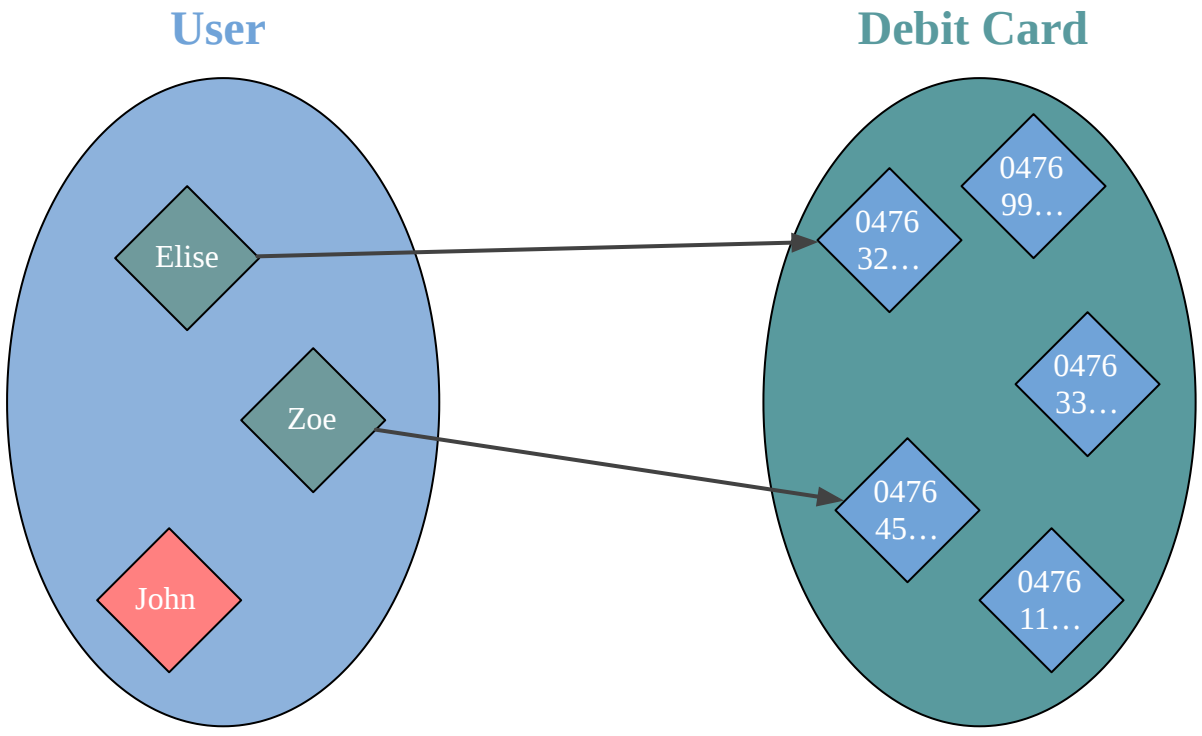
Error Handling

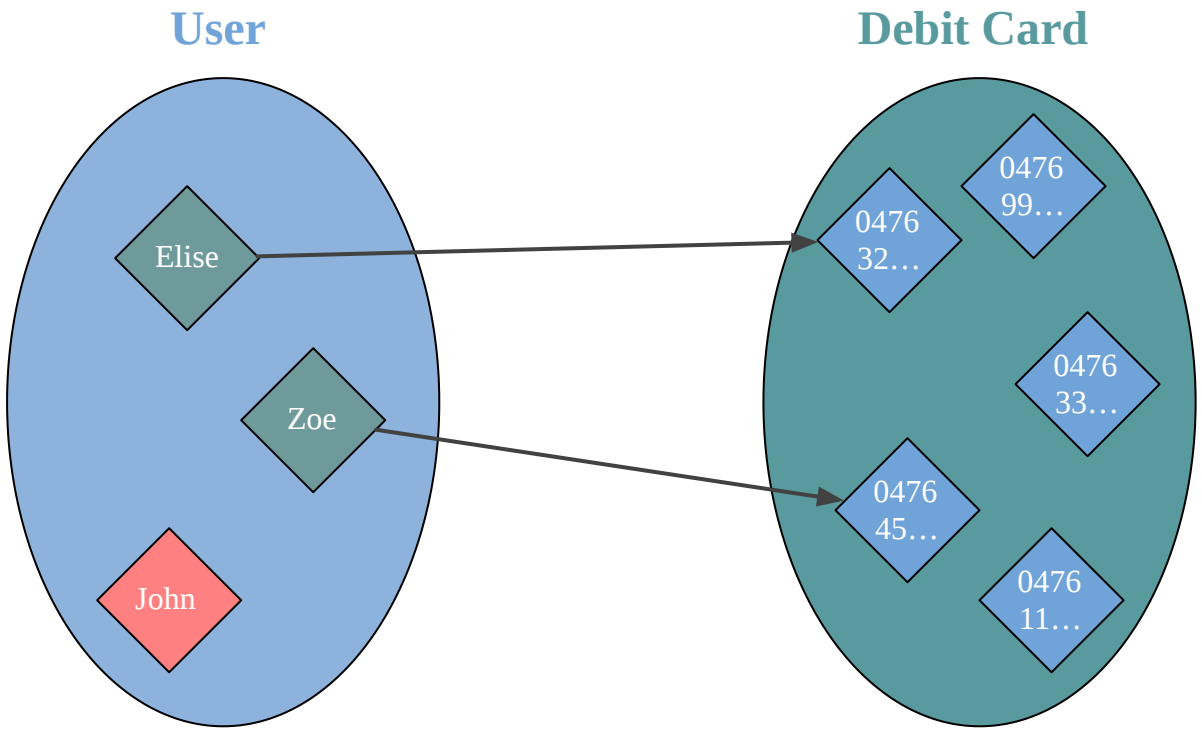
Function



Partial Function



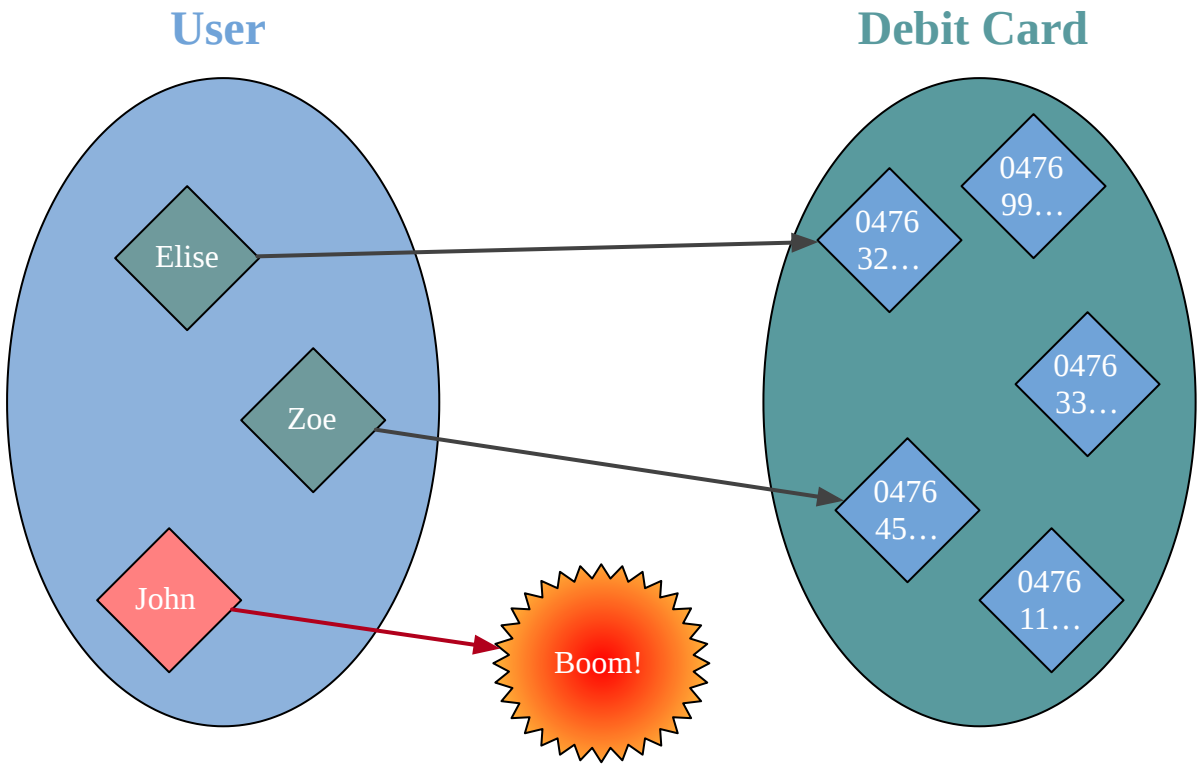
Partial Function



Maybe John:

- doesn't have a card
- didn't share it with us
- lost it
- card expired
- …

Partial Function



Maybe John:

- doesn't have a card
- didn't share it with us
- lost it
- card expired
- …

Partial Function is lie

```
// If DebitCard is missing throw a NoSuchElementException
def getDebitCard(person: Person): DebitCard = ???
```

How have you seen error scenarios handled?

[2-3 min] to write down answers

Null

```
import java.time.LocalDate

case class DebitCard(number: String, expiry: LocalDate)

case class User(name: String, debitCard: DebitCard)

val elise = User("Elise", DebitCard("0476-9999-9999-9999", LocalDate.of(2019,6,8)))
val john  = User("John" , null)

def debitCardHint(x: User): String =
  x.debitCard.number.take(4)
```

```
scala> debitCardHint(john)
java.lang.NullPointerException
  at .debitCardHint(<console>:16)
  ... 43 elided
```

Option in data type

```
case class DebitCard(number: String, expiry: LocalDate)

case class User(name: String, debitCard: Option[DebitCard])

val elise = User("Elise", Some(DebitCard("0476-9999-9999-9999", LocalDate.of(2019,6,8))))
val john  = User("John" , None)
```

Exception in function

```
case class DebitCard(number: String, expiry: LocalDate)

case class User(name: String, debitCard: Option[DebitCard])

val elise = User("Elise", Some(DebitCard("0476-9999-9999-9999", LocalDate.of(2019,6,8))))
val john  = User("John" , None)
```

```
def getValidDebitCard(user: User, today: LocalDate): DebitCard = {
  if(user.debitCard.isEmpty)
    throw new Exception("No debit card")
  else if(user.debitCard.get.expiry.isAfter(today))
    throw new Exception("Expired debit card")
  else
    user.debitCard.get
}
```

```
scala> getValidDebitCard(elise, today = LocalDate.of(2019,5,1))
java.lang.Exception: Expired debit card
  at .getValidDebitCard(<console>:21)
  ... 43 elided
```

Boolean flag (Boolean blindness)

```
case class DebitCard(number: String, expiry: LocalDate)

case class User(name: String, debitCard: Option[DebitCard])

val elise = User("Elise", Some(DebitCard("0476-9999-9999-9999", LocalDate.of(2019,6,8))))
val john  = User("John" , None)
```

```
def isValidDebitCard(user: User, today: LocalDate): Boolean = {
  if(user.debitCard.isEmpty) false
  else user.debitCard.get.expiry.isBefore(today)
}
```

```
scala> isValidDebitCard(elise, today = LocalDate.of(2019,5,1))
res2: Boolean = false

scala> isValidDebitCard(elise, today = LocalDate.of(2020,1,1))
res3: Boolean = true
```

Number status

```
case class DebitCard(number: String, expiry: LocalDate)

case class User(name: String, debitCard: Option[DebitCard])

val elise = User("Elise", Some(DebitCard("0476-9999-9999-9999", LocalDate.of(2019,6,8))))
val john  = User("John" , None)
```

```
def validateDebitCard(user: User, today: LocalDate): Int = {
  if(user.debitCard.isEmpty) -1
  else if(user.debitCard.get.expiry.isAfter(today)) -2
  else 1
}
```

```
scala> validateDebitCard(elise, today = LocalDate.of(2019,5,1))
res4: Int = -2

scala> validateDebitCard(elise, today = LocalDate.of(2020,1,1))
res5: Int = 1

scala> validateDebitCard(john, today = LocalDate.of(2020,1,1))
res6: Int = -1
```

Option or Either in function

```
sealed trait DebitCardError
case object MissingDebitCard extends DebitCardError
case object ExpiredDebitCard extends DebitCardError

def getValidDebitCard(user: User, today: LocalDate): Either[DebitCardError, DebitCard] =
  user.debitCard match {
    case None     => Left(MissingDebitCard)
    case Some(dc) =>
      if(dc.expiry.isAfter(today)) Left(ExpiredDebitCard)
      else Right(dc)
  }
```

```
scala> getValidDebitCard(elise, today = LocalDate.of(2019,5,1))
res7: Either[DebitCardError,DebitCard] = Left(ExpiredDebitCard)

scala> getValidDebitCard(john , today = LocalDate.of(2019,5,1))
res8: Either[DebitCardError,DebitCard] = Left(MissingDebitCard)
```

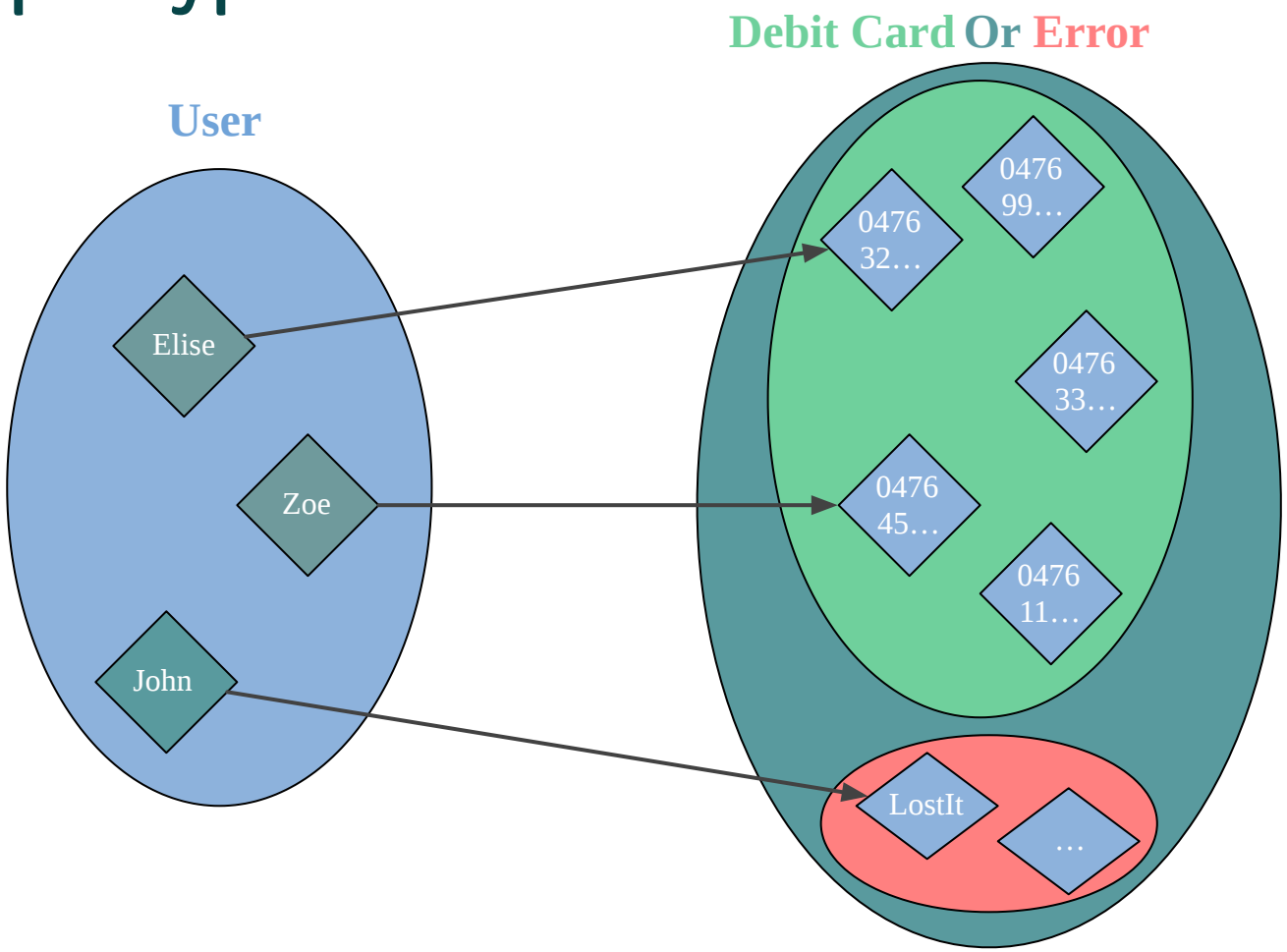
Make the error unrepresentable

```
sealed trait User

case class UserWithDc(name: String, debitCard: DebitCard) extends User
case class InvalidUser(name: String) extends User
```

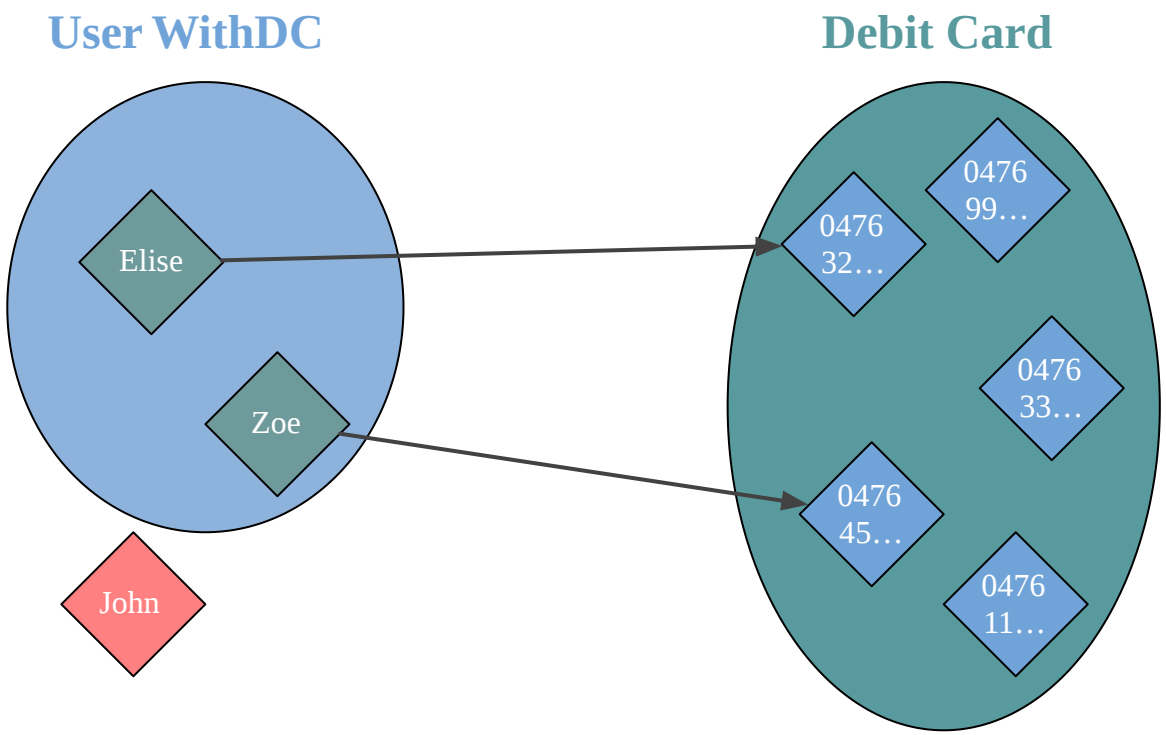
```
def getDebitCard(user: UserWithDc): DebitCard = user.debitCard

def getValidDebitCard(user: UserWithDc, today: LocalDate): Option[DebitCard] = {
  if(user.debitCard.expiry.isAfter(today)) None
  else Some(user.debitCard)
}
```

Two strategies to fix it

Increase output type



Reduce input type



Which approach is preferable? Why?

[2-3 min] discuss with neighbour

$$|A \Rightarrow B| = |B| \hat{} |A|$$

Cardinality

Increase output type

```
|A => (B | E)| = |(B | E)| ^ |A|
                = (|B| + |E|) ^ |A|
```

Reduce input type

```
|A1 => B| = |B| ^ |A1|
           = |B| ^ |(A | A2)|
           = |B| ^ (|A| - |A2|)
```

assume `A = A1 | A2`

Unrepresentable Exercises

Making state unrepresentable

- More upfront work but easier to write business logic
- Different tools with various complexities:
  - Enum
  - ADT / GADTs
  - Parametric / Refinement / Dependent types
  - HList, HSet

Increase output type

Which type constructor has an error channel?

[2-3 min] to write down answers

Type constructors with an error channel

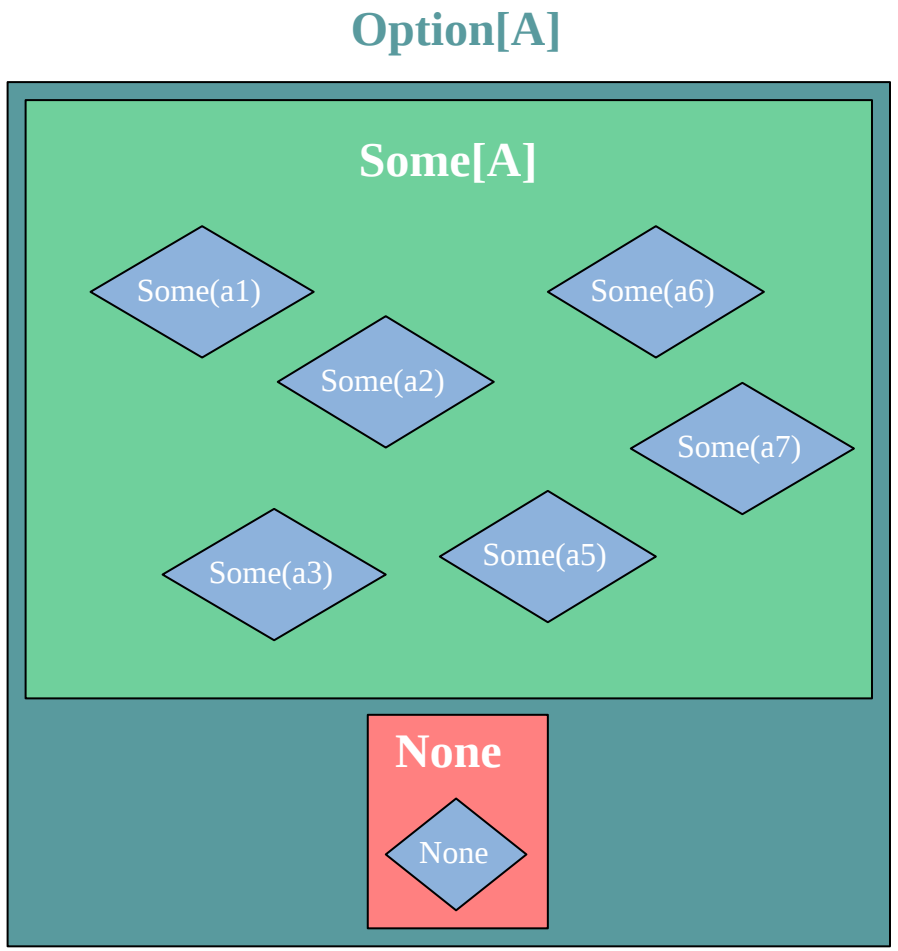
- `Option`
- `Either`
- `Try`
- `Future` or `IO` (from cats, Monix, ZIO)
- `Validated` (from cats, scalaz)

Type constructors with an error channel

1. Option
2. Either
3. Validated

Option

```
sealed trait Option[+A]

object Option {

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

  case object None extends Option[Nothing]

}
```



Option variance

```
sealed trait Option[+A]

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

```
sealed trait Option[A]

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

Option variance

```
sealed trait Shape extends Product with Serializable

object Shape {
  case class Circle(radius: Double) extends Shape
  case class Rectangle(width: Double, height: Double) extends Shape
}
```

Option variance

```
sealed trait Option[+A]

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

```
scala> Some(Circle(5.2)).getOrElse(Rectangle(4,2))
res9: Shape = Circle(5.2)
```

```
scala> Some(Circle(5.2)).getOrElse("foo")
res10: java.io.Serializable = Circle(5.2)
```

```
sealed trait Option[A]

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

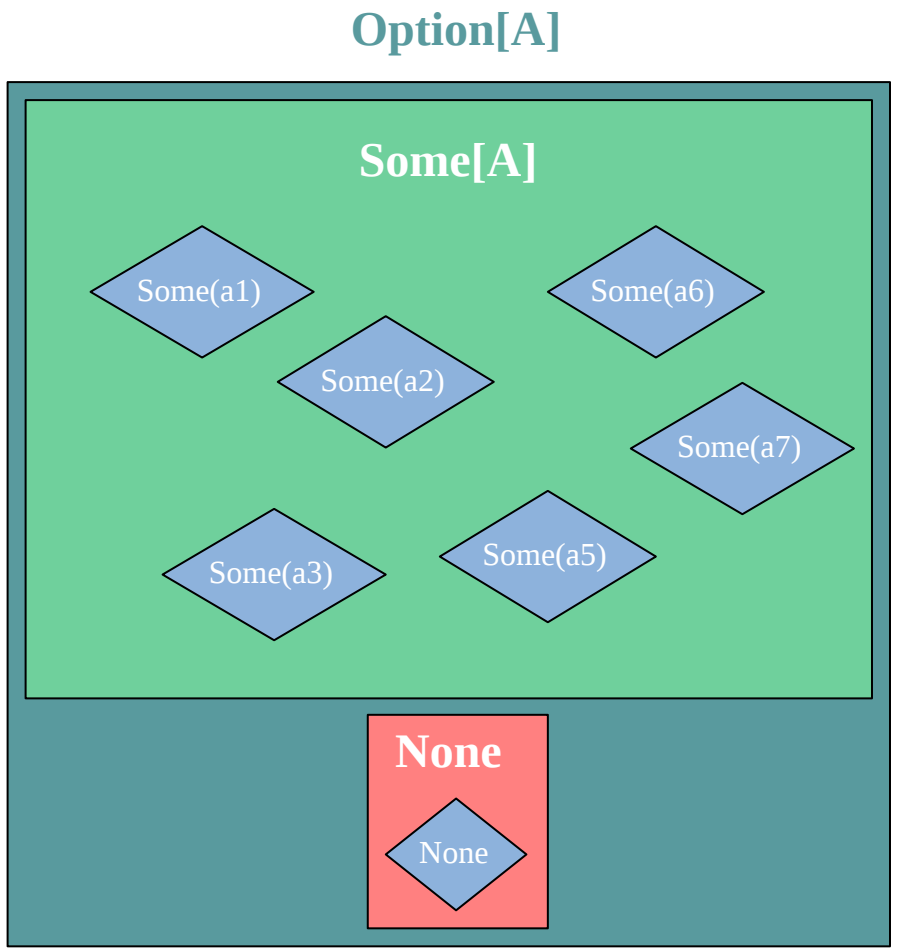
```
scala> Some(Circle(5.2): Shape).getOrElse(
     |   Rectangle(4,2): Shape)
res11: Shape = Circle(5.2)
```

```
scala> Some(Circle(5.2)).getOrElse("foo")
<console>:24: error: type mismatch;
 found   : String("foo")
 required: Shape.Circle
       Some(Circle(5.2)).getOrElse("foo")
                                   ^
```

Option Exercises

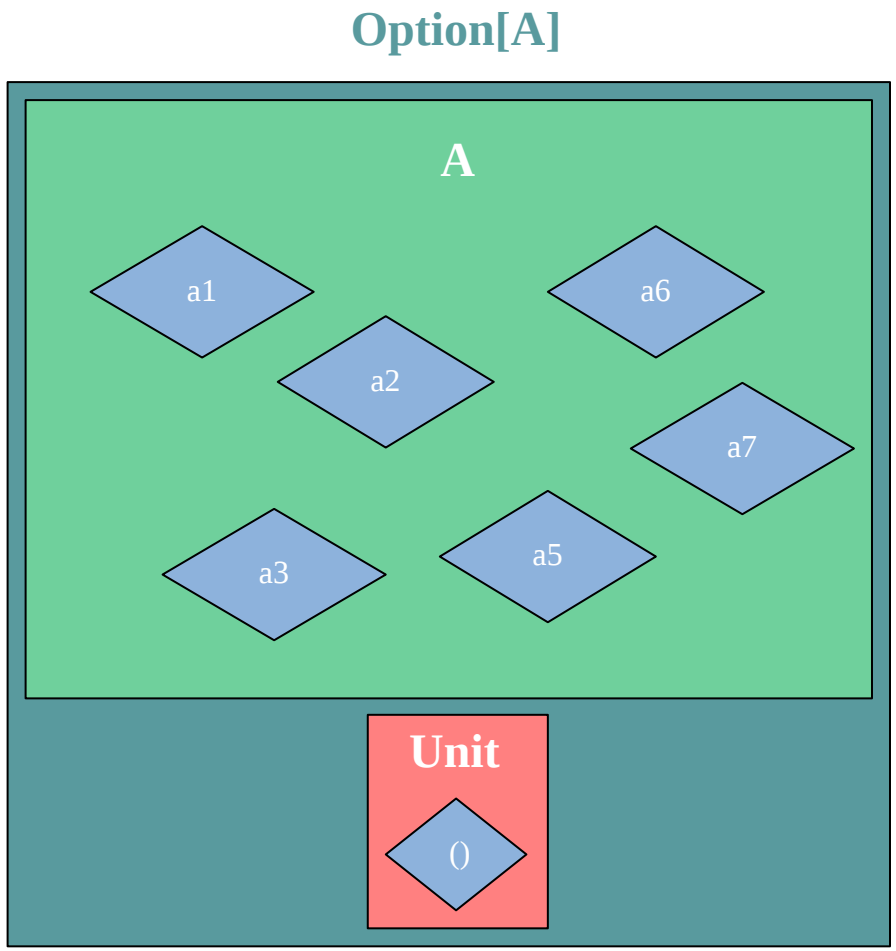
Option

```
sealed trait Option[+A]

object Option {

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

  case object None extends Option[Nothing]

}
```



Option

```
sealed trait Option[+A]

object Option {

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

  case object None extends Option[Nothing]

}
```

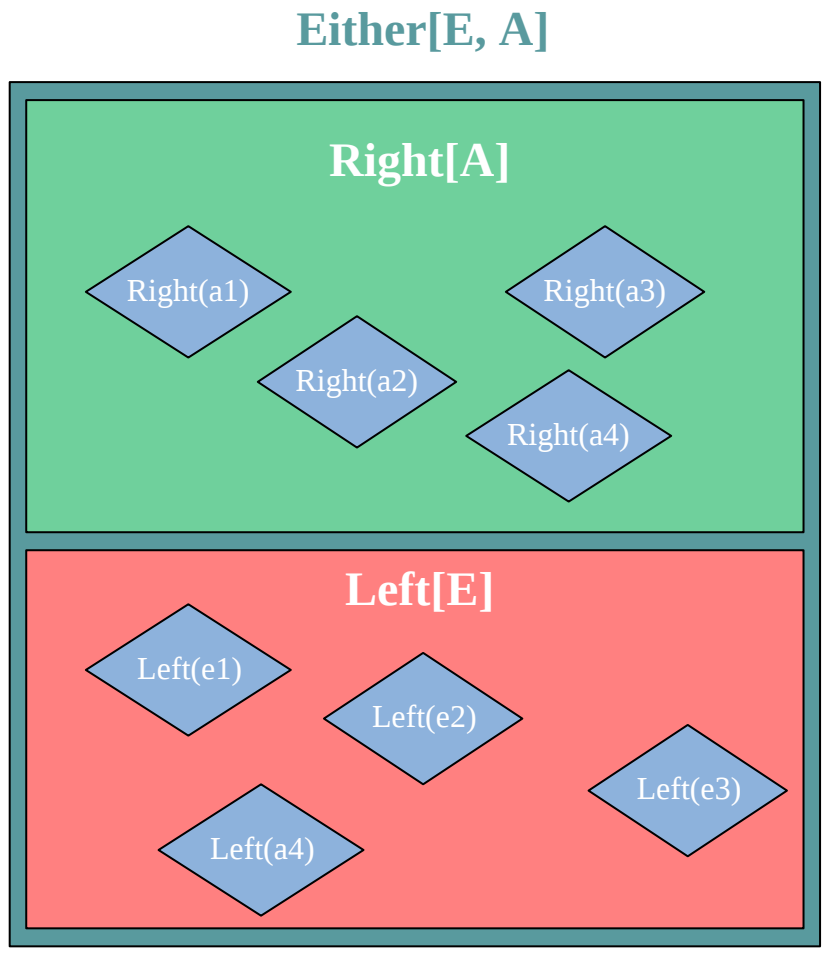


Use Option when

- Failure is unique, e.g. key is not in the Map
- Or you don't need granular error

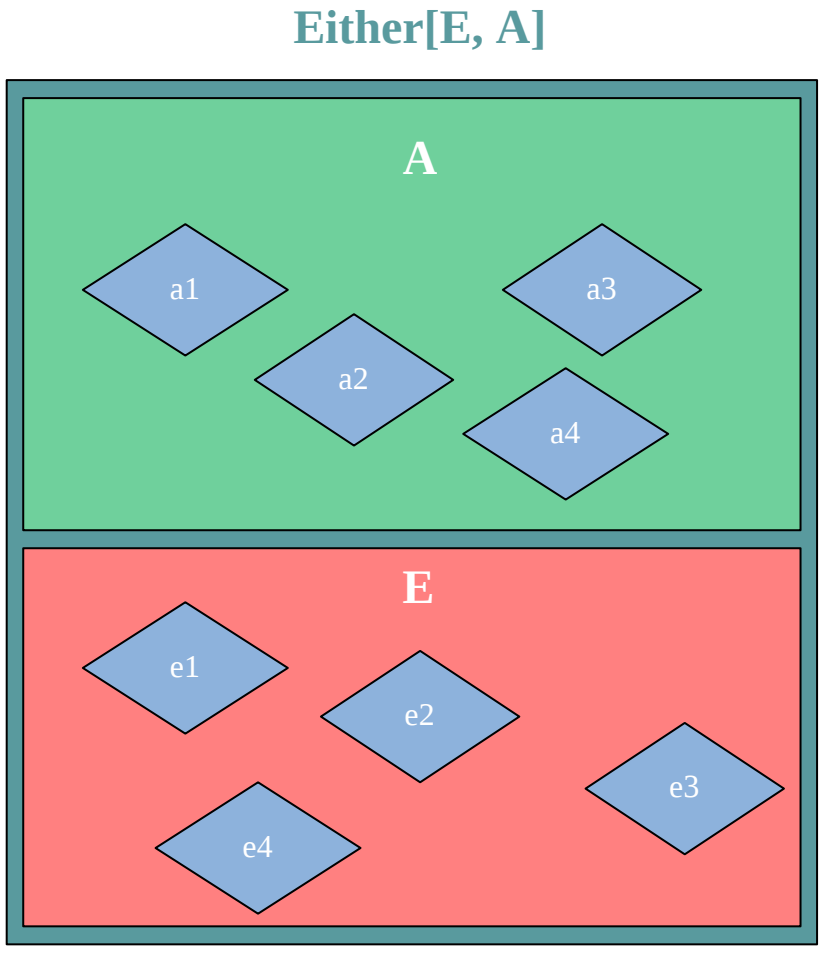
Either

```
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

```
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 variance

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

```
sealed trait MyError
case class Error1(name: String) extends MyError
case object Error2 extends MyError

sealed trait Shape
case class Circle(radius: Double) extends Shape
case class Rectangle(width: Double, height: Double) extends Shape
```

```
val right1: Either[MyError, Shape] = Right(Circle(5.2))
val right2: Either[MyError, Shape] = Right(Rectangle(2, 3))
val left1 : Either[MyError, Shape] = Left(Error1("foo"))
val left2 : Either[MyError, Shape] = Left(Error2)
```

Either Exercises

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]
```

Use Either when

- Many causes of failure and you want to keep track of them
- Fail early, do not accumulate errors

Validated

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

object Validated {
  case class Invalid[+E](value: E) extends Validated[E, Nothing]
  case class   Valid[+A](value: A) extends Validated[Nothing, A]
}
```

Isn't Validated the same as an Either?

```haskell
data Either    e a = Left    e | Right a
data Validated e a = Invalid e | Valid a
```

Isn't Validated the same as an Either?

```
data Either    e a = Left    e | Right a
data Validated e a = Invalid e | Valid a
```

Yes, but Validated accumulates failure

```
scala> tuple2(Left("id not found"), Left("username too small"))
res0: Either[String,(Nothing, Nothing)] = Left(id not found)
```

```
scala> tuple2(Invalid("id not found"), Invalid("username too small"))(_ ++  ", " ++ _)
res1: exercises.errorhandling.Validated[String,(Nothing, Nothing)] = Invalid(id not found, username too small)
```

ValidatedNel

```
import cats.data.NonEmptyList
```

```
type ValidatedNel[+E, +A] = Validated[NonEmptyList[E], A]
```

ValidatedNel

```
import cats.data.NonEmptyList
```

```
type ValidatedNel[+E, +A] = Validated[NonEmptyList[E], A]
```

```
def invalidNel[E](e: E): ValidatedNel[E, Nothing] =
  Invalid(NonEmptyList.of(e))
```

ValidatedNel

```
import cats.data.NonEmptyList
```

```
type ValidatedNel[+E, +A] = Validated[NonEmptyList[E], A]
```

```
def invalidNel[E](e: E): ValidatedNel[E, Nothing] =
  Invalid(NonEmptyList.of(e))
```

```
scala> println(tuple2(Invalid("id not found"), Invalid("username too small"))(_ ++  ", " ++ _))
Invalid(id not found, username too small)

scala> println(tuple2(invalidNel("id not found"), invalidNel("username too small"))(_ ::: _))
Invalid(NonEmptyList(id not found, username too small))
```

Validated Exercises

Use Validated when

- Many causes of failure and you want to keep track of them
- Accumulate errors, do not fail early

Error accumulation

```
def tuple2[E, A, B](fa: Validated[E, A], fb: Validated[E, B])
                   (combineError: (E, E) => E): Validated[E, (A, B)] = ???

tuple2(invalidNel("id not found"), invalidNel("username too small"))(_ ::: _)
```

Error accumulation

```
def tuple2[E, A, B](fa: Validated[E, A], fb: Validated[E, B])
                   (combineError: (E, E) => E): Validated[E, (A, B)] = ???

tuple2(invalidNel("id not found"), invalidNel("username too small"))(_ ::: _)
```

```
def tuple2[E: Semigroup, A, B](fa: Validated[E, A], fb: Validated[E, B]): Validated[E, (A, B)] = ???

tuple2(invalidNel("id not found"), invalidNel("username too small"))
```

More details in Typeclass module

Review

Algebraic Data Type (ADT)

```
class Option[+A]        = None                  | Some(a: A)
class Try[+A]           = Failure(e: Throwable) | Success(a: A)
class Either[+E, +A]    = Left(e: E)            | Right(a: A)
class Validated[+E, +A] = Invalid(e: E)         | Valid(a: A)
```

Algebraic Data Type (ADT)

```
class Option[+A]        = None                   | Some(a: A)
class Try[+A]           = Failure(e: Throwable)  | Success(a: A)
class Either[+E, +A]    = Left(e: E)             | Right(a: A)
class Validated[+E, +A] = Invalid(e: E)          | Valid(a: A)
```

They are all variations of Either

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

case class Validation[+E, +A](value: Either[E, A])
```

Decision Tree