Handling Errors without Exceptions

Goals

- Learn the basic principles for **raising and handling errors functionally**.
- The **big idea** is that we can **represent failures and exceptions with ordinary values**, and we can write **higher-order functions that *abstract out* common patterns of error handling and recovery,** while preserving the benefit of consolidation of error-handling logic.
- Recreate **Option** and **Either** to enhance your understanding of how these types can be used for handling errors.

Review: Referential Transparency

- Any RT expression may be **substituted with the value it refers to**, and this substitution should **preserve program meaning**.
- The meaning of RT expressions **does not depend on context** and may be **reasoned about locally**, whereas the meaning of non-RT expressions is **context-dependent** and requires more **global reasoning**.


```
def failingFn(i: Int): Int =
  val y: Int = throw new Exception("fail!")
  try
    val x = 42 + 5
    x+y
  catch
    case e: Exception => 43
```

`val y: Int = ...` declares `y` as having type `Int` and sets it equal to the right-hand side of `=`.

A `catch` block is just a pattern-matching block like the ones we've seen. `case e: Exception` is a pattern that matches any `Exception`, and it binds this value to the identifier `e`. The match returns the value `43`.

```
def failingFn2(i: Int): Int =
  try
    val x = 42 + 5
    x + ((throw new Exception("fail!")): Int) }
  catch
    case e: Exception => 43
```

A thrown `Exception` can be given any type; here we're annotating it with the type `Int`.

The Good and The Bad of Exceptions

- ***Exceptions break RT and introduce context dependence***, requiring non-local reasoning.
  - Prevent simple reasoning of the substitution model.
  - May lead us to write confusing exception-based code.
- ***Exceptions are not type-safe***.
  - Without checking for an exception in call site, it won't be detected until runtime.

☞***Consolidate and centralize error-handling logic***, rather than being forced to distribute this logic throughout our codebase.

Alternatives to Exceptions #1

```
def mean(xs: Seq[Double]): Double =
  if (xs.isEmpty) then
    throw new ArithmeticException("mean of empty list!")
  else xs.sum / xs.length
```

sum is defined as a method on Seq only if the elements of the sequence are numeric. The standard library accomplishes this trick with implicits, which we won't go into here.

*The **mean** function is an example of what's called a **partial function**: it's not defined for some inputs.*

The first possibility is to return some sort of bogus value of type **Double**.

- simply return **xs.sum / xs.length** in all cases, and have it result in 0.0/0.0 when the input is empty, which is **Double.NaN**;
- or we could return some other sentinel value.

Reasons to Reject This Solution

- It allows errors to silently propagate.
- It results in a fair amount of boilerplate code at call sites, with explicit if statements to check the result.
- It's not applicable to polymorphic code.
- It demands a special policy or calling convention of callers.

Alternatives to Exceptions #2

- Force the caller to supply an argument for the case we cannot handle the input:

```
def mean(xs: Seq[Double], onEmpty: Double): Double =
  if (xs.isEmpty) then onEmpty
  else xs.sum / xs.length
```

- **Drawbacks**: it requires that *immediate* callers have direct knowledge of how to handle the undefined case and limits them to returning a **Double**.

Solutions?

- Better to be dealt with at the most appropriate level.
- The solution is to **represent explicitly in the return type** that a function may **not** always have an answer.
- We can think of this as **deferring to the caller for the error-handling strategy**.

Benefits of Returning Errors as Values

- The functional solution is **safer** and **retains referential transparency**.
- Through the use of higher-order functions, **preserve the primary benefit** of **exceptions—*consolidation of error-handling logic*.**

- Our example: **Option** and **Either**

Option Type

- Represent explicitly in the return type that a function may not always have an answer.
  - Deferring to the caller for the error-handling strategy.

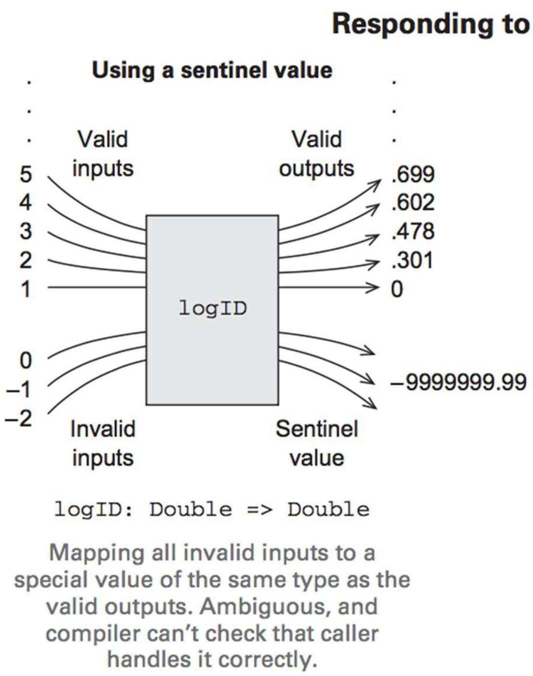
```
enum Option[+A]:
  case Some(get: A)
  case None
```

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


Partial Function vs Total Functions

```
def mean(xs: Seq[Int]): Double =
  if (xs.isEmpty) then
     throw new ArithmeticException("mean of empty list!")
  else xs.sum / xs.length
```

```
import Option.{Some, None}
def mean(xs: Seq[Double]): Option[Double] =
  if (xs.isEmpty) then None
  else Some(xs.sum / xs.length)
```



Usage patterns for Option

Used throughout the Scala standard library, for instance:

- Map lookup for a given key returns **Option**.
- **headOption** and **lastOption** defined for lists and other iterables return an **Option** containing the first or last elements of a sequence if it's nonempty.

Through **Option** (and **Either**), we can **factor out common patterns of error handling** *via* **higher-order functions**, freeing us from writing the usual boiler-plate that comes with exception-handling code.

Option Data Type

```
enum Option[+A]:
  case Some(get: A)
  case None

  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
  def getOrElse[B >: A](default: => B): B
  def orElse[B >: A](ob: => Option[B]): Option[B]
  def filter(f: A => Boolean): Option[A]
```

Don't evaluate ob unless needed.

The B >: A says that the B type parameter must be a supertype of A.

Implementation Guide for Combinators for Option Type

- Fine to use pattern matching for **map** and **getOrElse**.
- For **map** and **flatMap**, the type signature should be enough to determine the implementation.
- **getOrElse** returns the result inside the **Some** case of the **Option**, or if the **Option** is **None**, returns the given default value.
- **orElse** returns the first **Option** if it's defined; otherwise, it returns the second **Option**.

Common Usage Pattern for Option

A common pattern is to transform an **Option** via calls to **map**, **flatMap**, and/or **filter**, and then use **getOrElse** to do error handling at the end:

```
case class Employee(name: String, department: String)
def lookupByName(name: String): Option[Employee] = ...

val dept: String = lookupByName("Joe")
                   .map(_.deptartment)
                   .filter(_ != "Accounting")
                   .getOrElse("Default Dept")
```


Common Usage Pattern for Option

- The **map** function transforms the result inside an **Option**, if it exists. It defers the error handling to later code.
- With **flatMap**, we can construct a *computation with multiple stages*, and the **computation will abort as soon as the first failure is encountered**, since **None.flatMap(f)** will immediately return **None**, without running **f**.

```
def variance(xs: Seq[Double]): Option[Double] =
  mean(xs).flatMap(m => mean(xs.map(x => math.pow(x - m, 2))))
```

Common Usage Pattern for Option

- **orElse** is useful when we need to chain together possibly failing computations, trying the second if the first hasn't succeeded.
- A common idiom is to do **o.getOrElse(throw new Exception("FAIL"))** to convert the **None** case of an **Option** back to an exception.

Lifting

map transforms **F[A]** to **F[B]** by applying **f: A => B**

```
def map[A, B](fa: F[A])(f: A => B): F[B]
```

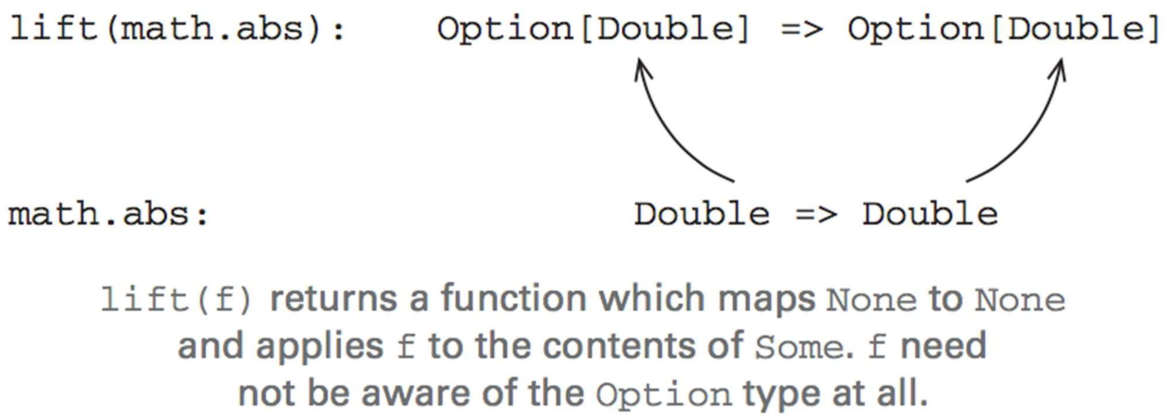
Another way of looking at this is to view **map** as follow:

```
(A => B) => (F[A] => F[B])
```

We can **lift** ordinary functions to ones that operate on **Option**.

```
def lift[A,B](f: A => B): Option[A] => Option[B] = _ map f
```

Lifting Functions



Lifting Example

Any function that already exists can be transformed (via **lift**) to operate **within the context of** a single **Option** value.

```
scala> val absO: Option[Double] => Option[Double] = lift(math.abs)
absO: Option[Double] => Option[Double] = Option$$$Lambda$1914/1358135423@5b576cf8

scala> absO(Option(-33))
res12: Option[Double] = Some(33.0)
```

Try Function

Note: This discards information about the error. We'll improve on this later in the chapter.

```
def Try[A](a: => A): Option[A] =
  try Some(a)
  catch
    case e: Exception => None
```

We accept the A argument non-strictly, so we can catch any exceptions that occur while evaluating a and convert them to None.

A general-purpose function to convert from an exception-based API to an ***Option-oriented API***.

Uses a non-strict or lazy argument, as indicated by the **=> A** as the type of **a**.


```
def insuranceRateQuote(
  age: Int, numberOfSpeedingTickets: Int): Double

def parseInsuranceRateQuote(
    age: String,
    numberOfSpeedingTickets: String): Option[Double] =
  val optAge: Option[Int] = Try(age.toInt)
  val optTickets: Option[Int] =
                Try(numberOfSpeedingTickets.toInt)
  insuranceRateQuote(optAge, optTickets)
```

Doesn't type check! See following discussion.

*we'd like to lift **insuranceRateQuote** to operate in the context of two optional values.*

Combinator: map2

```
def map2[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C): Option[C]
```

Functions accepting a single argument may be called with braces instead of parentheses; this is equivalent to `Try(age.toInt)`.

```
def parseInsuranceRateQuote(
    age: String,
    numberOfSpeedingTickets: String): Option[Double] =
  val optAge: Option[Int] = Try { age.toInt }
  val optTickets: Option[Int] =
              Try{ numberOfSpeedingTickets.toInt }
  map2(optAge, optTickets)(insuranceRateQuote)
```

If either parse fails, this will immediately return `None`.

The map2 makes any existing functions of two arguments to become “***Option-aware***”, without modifying them.

For-comprehensions

```
def map2[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C): Option[C] =
  a flatMap (aa =>
    b map (bb =>
      f(aa, bb)))
```

And here’s the *exact same code* written as a for-comprehension:

```
def map2_[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C): Option[C] =
  for
    aa <- a
    bb <- b
  yield f(aa, bb)
```

The compiler desugars the bindings to **flatMap** calls, with the final binding and **yield** being converted to a call to **map**.

sequence

```
def sequence[A](a: List[Option[A]]): Option[List[A]]
```

Write a function **sequence** that combines a list of **Option**s into one **Option** containing a list of all the **Some** values in the original list.

If the original list contains **None** even once, the result of the function should be **None**; otherwise the result should be **Some** with a list of all the values.

```
List(Some(1), Some(2), Some(3)) => Some(List(1, 2, 3))
List(Some(1), None, Some(3)) => None
```


What if we have a whole list of **String** values that we wish to parse to **Option[List[Int]]**?

```
def parseInts(a: List[String]): Option[List[Int]] =
  sequence(a map (i => Try(i.toInt)))
```

Unfortunately, this is inefficient, since it traverses the list twice,

- first to convert each **String** to an **Option[Int]**, and
- a second pass to combine these **Option[Int]** values into an **Option[List[Int]]**.

traverse

```
def traverse[A, B](fa: List[A])(f: A => Option[B]):
  Option[List[B]] =  sequence(fa.map(f))
```

It's straightforward to do using **map** and **sequence**, but try for a more efficient implementation that only looks at the list once. In fact, implement **sequence** in terms of **traverse**.

Limitations of Option

- **Option** doesn't tell us anything about what went wrong in the case of an exceptional condition.
- A simple extension to **Option** is the **Either** data type, which lets us track a ***reason*** for the failure.

Either Type

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

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

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

By convention

- **Right** constructor is reserved for the **success** case (a pun on “right,” meaning correct), and
- **Left** is used for **failure**. Note the suggestive name E (for *error*).

Usage Example of Either

```scala
import Either.{Right, Left}

def mean(xs: Seq[Double]): Either[String, Double] =
  if (xs.isEmpty) then Left("mean of empty list!")
  else Right(xs.sum / xs.length)

import scala.util.control.NonFatal

def safeDiv(x: Int, y: Int): Either[Throwable, Int] =
  try Right(x / y)
  catch case e: NonFatal(e) => Left(e)

// Can be extracted as a reusable and general function
def catchNonFatal[A](a: => A): Either[Throwable, A] =
  try Right(a)
  catch case e: NonFatal => Left(e)
```

Implement versions of **map**, **flatMap**, **orElse**, and **map2** on **Either** that operate on the **Right** value.

When mapping over the right side, we must promote the left type parameter to some supertype, to satisfy the +E variance annotation.

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

  def map[B](f: A => B): Either[E, B]
  def flatMap[EE >: E, B](f: A => Either[EE, B]): Either[EE, B]
  def orElse[EE >: E, B >: A](f: => Either[EE, B]): Either[EE, B]
  def map2[EE >: E, B, C](eb: Either[EE, B])(f: (A,B) => C): Either[EE, C]
end Either
```


Syntax before Scala 2.12

- In Scala 2.10.x and 2.11.x, **Either** is unbiased.
- That is, usual combinators like **flatMap** and **map** are missing from it.
- Instead, you call **.right** or **.left** to get a **RightProjection** or **LeftProjection** (respectively) which does have the combinators.
- The direction of the projection indicates the direction of bias. For instance, calling **map** on a **RightProjection** acts on the **Right** of an **Either**.

Syntax before Scala 2.12

```
val e1: Either[String, Int] = Right(5)
// e1: Either[String,Int] = Right(5)

e1.right.map(_ + 1)
// res0: scala.util.Either[String,Int] = Right(6)

val e2: Either[String, Int] = Left("hello")
// e2: Either[String,Int] = Left(hello)

e2.right.map(_ + 1)
// res1: scala.util.Either[String,Int] = Left(hello)
```

Note the return types are themselves back to **Either**, so if we want to make more calls to **flatMap** or **map** then we again must call **right** or **left**.

Syntax as of Scala 2.12

- In Scala 2.12.x, **Either** is **right-biased**.
- Because **Either** is right-biased, it is possible to define a **Monad** instance for it. Since we only ever want the computation to continue in the case of **Right**, we fix the left type parameter and leave the right one free ➔ More on this later!

Note that with these definitions, **Either** can now be used in for-comprehensions. For instance:

```
def parseInsuranceRateQuote( age: String,
  numberOfSpeedingTickets: String): Either[Exception,Double] = for
    a <- Either.catchNonFatal { age.toInt }
    tickets <- Either.catchNonFatal { numberOfSpeedingTickes.toInt }
  yield insuranceRateQuote(a, tickets)
```

Now we get information about the actual exception that occurred, rather than just getting back **None** in the event of a failure.

Exercise

Implement **sequence** and **traverse** for **Either**. These should return the first error that's encountered, if there is one.

```
def sequence[E, A](es: List[Either[E, A]]): Either[E, List[A]]
def traverse[E, A, B](as: List[A])(
                             f: A => Either[E, B]): Either[E, List[B]]
```

Using Either to validate data

```
case class Name private (value: String)

object Name:
  def apply(name: String): Either[String, Name] =
    if name == "" || name == null then Left("Name is empty.")
    else Right(new Name(name))

case class Age private (value: Int)
object Age:
  def apply(age: Int): Either[String, Age] =
    if age < 0 then Left("Age is out of range.")
    else Right(new Age(age))

case class Person(name: Name, age: Age)
object Person:
  def make(name: String, age: Int): Either[String, Person] =
    Name(name).map2(Age(age))(Person(_, _))
```


Accumulating Errors

- The implementation of **map2** is only able to report *one error*, even if both arguments are invalid (that is, both arguments are **Left**`s). It would be *more useful if we could report both errors*.
- What would you need to change in order to report both errors? Could you create a new data type that captures this requirement better than **Either** does, with some additional structure? How would **orElse**, **traverse**, and **sequence** behave differently for that data type?

Validated

```
enum Validated[+E, +A]:
  case Valid(get: A)
  case Invalid(errors: List[E])

  def toEither: Either[List[E], A] = this match
    case Valid(a)    => Either.Right(a)
    case Invalid(es) => Either.Left(es)

  def map[B](f: A => B): Validated[E, B] = this match
    case Valid(a)    => Valid(f(a))
    case Invalid(es) => Invalid(es)

  def map2[EE >: E, B, C](b: Validated[EE, B])(f: (A, B) => C):
    Validated[EE, C] = (this, b) match
      case (Valid(aa), Valid(bb))         => Valid(f(aa, bb))
      case (Invalid(es), Valid(_))        => Invalid(es)
      case (Valid(_), Invalid(es))        => Invalid(es)
      case (Invalid(es1), Invalid(es2)) => Invalid(es1 ++ es2)
```


Validated (Cont'd)

```
object Validated:
  def fromEither[E, A](e: Either[List[E], A]): Validated[E, A] =
    e match
      case Right(a) => Valid(a)
      case Left(es) => Invalid(es)

  def traverse[E, A, B](
      as: List[A],
      f: A => Validated[E, B]
  ): Validated[E, List[B]] =
    as.foldRight(Valid(Nil): Validated[E, List[B]])((a, acc) =>
      f(a).map2(acc)(_ :: _)
    )

  def sequence[E, A](vs: List[Validated[E, A]]): Validated[E, List[A]] =
    traverse(vs, identity)
```