Handling Failures

06016415 Functional Programming

Outline

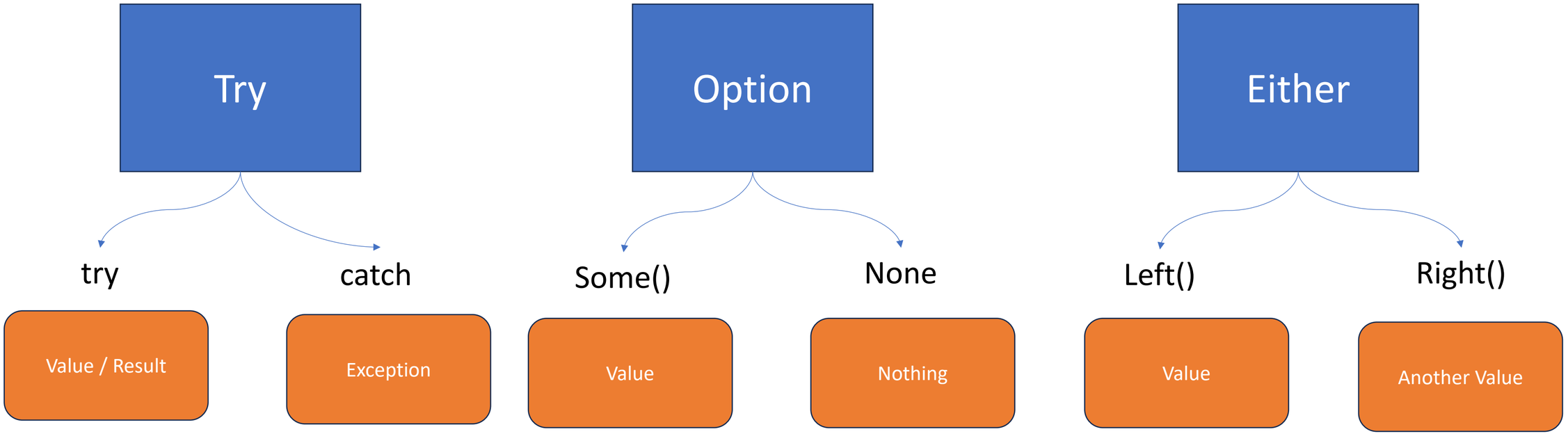
- Exceptions
- Using Try/Catch
- Using Option/Some/None
- Using Either/Left/Right

Exceptions

- ใน FP มุ่งเน้นการใช้ value ซึ่งถูกสร้าง และใช้งานโดยฟังก์ชันใด
- ดังนั้น จึงอาจเกิดข้อผิดพลาดในกรณีพิเศษ เมื่อเขียนโปรแกรมในฟังก์ชัน ซึ่งเป็นเรื่องปกติที่อาจเกิดขึ้นได้

- Exception (การยกเว้น) จึงถูกนำมาใช้
  - **Try** (also called Result) / Catch
  - **Option** / Some / None
  - **Either** / Left / Right

- วิธีการข้างต้น ช่วยจัดการกับข้อผิดพลาดตามสถานการณ์ แต่การพิจารณาข้อยกเว้นอาจซับซ้อน จึงต้องคำนึงถึงประเภทข้อยกเว้น และคำสั่งที่ฝังอยู่ภายในด้วย


Try
try
catch
Value / Result
Exception
Option
Some()
None
Value
Nothing
Either
Left()
Right()
Value
Another Value

```scala
//Exceptions and Special Values
def search_0[A](values: List[A], target: A): Int =  // returns -1 when not found
  values.indexOf(target)
```

```scala
// DON'T DO THIS!
def between_0[A](values: List[A], from: A, to: A): List[A] =
  val i = search_0(values, from)
  val j = search_0(values, to)
  values.slice(i min j, (i max j) + 1)
```

```scala
val words = List("one", "two", "three", "four")
```

```scala
scala> between_0(words, "two", "four")
val res1: List[String] = List(two, three, four)

scala> between_0(words, "four", "two")
val res2: List[String] = List(two, three, four)
```

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

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

```
scala> failingFn(12)
java.lang.Exception: fail!
  at Main$package$.failingFn(Main.scala:10)
  ... 66 elided
```

```
scala> failingFn2(12)
val res17: Int = 43
```

```scala
// Seq is the common interface of various linear sequence-like collections.
def mean_0(xs: Seq[Double], onEmpty: Double): Double =
  if xs.isEmpty then onEmpty
  else xs.sum / xs.length
```

```
scala> mean_0(Seq(1,2,3),1)
val res4: Double = 2.0

scala> mean_0(Seq(),1)
val res5: Double = 1.0
```

```scala
def mean_1(xs: Seq[Double]): Double =
  if xs.isEmpty then
    throw new ArithmeticException("mean of empty list!")
  else xs.sum / xs.length
```

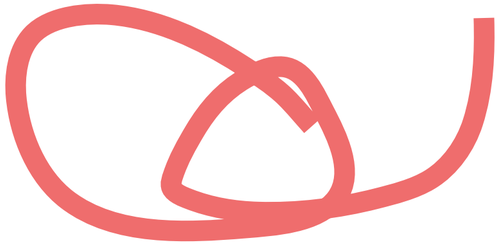
```
scala> mean_1(Seq(1,2,3))
val res12: Double = 2.0

scala> mean_1(Seq())
java.lang.ArithmeticException: mean of empty list!
  at Main$package$.mean_1(Main.scala:69)
  ... 66 elided
```

Using Try/Catch

General syntax:

```
try
   // some code that can throw 1+ exception types
catch
   // catch and handle the exceptions
   case e1: ExceptionType1 =>
      e1...
   case e2: ExceptionType2 =>
      e2...
finally
   // clean up your resources, i.e., call `.close`
```

```scala
def makeInt(s: String): Int =
    try
        s.toInt
    catch
        case e: NumberFormatException => 0
    finally
        println("finally!")
```



```scala
def readFileString(filename: String): Unit =
  val bufferedSource = scala.io.Source.fromFile(filename)
  for (line <- bufferedSource.getLines)
    println(line)
  bufferedSource.close
  println("End of file")
```

```scala
def readFile(filename: String): Unit =
  try
    scala.io.Source.fromFile(filename).getLines.foreach(line =>
      println(line)
    )
  catch
    case e: Exception => throw new Exception("cannot read")
  finally
    println("End of file")
```

Using Option/Some/None

```scala
//The Option data type
enum Option[+A]:
  case Some(get: A)
  case None

import Option.{Some, None}

def StringToInt(s: String): Option[Int] =
    try
        Some(s.toInt)
    catch
        case e: NumberFormatException => None
```

```scala
def CallMakeInt(aString: String): Unit =
  StringToInt(aString) match
      case Some(i) => println(s"Conversion worked. i = $i")
      case None => println("The conversion failed.")
```

```scala
def search_OP[A](values: List[A], target: A): Option[Int] =
  Some(values.indexOf(target))
```

```scala
def between_2[A](values: List[A], from: A, to: A): List[A] =
  (search_OP(values, from), search_OP(values, to)) match
    case (Some(i), Some(j))    => values.slice(i min j, (i max j) + 1)
    case _                     => List.empty


val words = List("one", "two", "three", "four")
```

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

```
scala> mean(Seq(1,2,3))
val res14: Option[Double] = Some(2.0)

scala> mean(Seq())
val res15: Option[Double] = None
```

Using Either/Left/Right

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

import Either.{Left, Right}

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

```
scala> mean_E(Seq(1,2,3))
val res11: Either[String, Double] = Right(2.0)
```

```scala
def divideXByY(x: Int, y: Int): Either[String, Int] =
    if (y == 0) Left("Hey, It can't divide by 0")
    else Right(x / y)
```

```scala
def checkDivideByY(x: Int, y: Int) : Unit =
  divideXByY(x, y) match {
    case Left(s) => println(s"Error: $s")
    case Right(i) => println(s"Answer: $i")
  }
```

Summary

- Functions can deal with failure by returning special values.

- To produce error values, functional programs often rely on specific types, usually in the form of alternatives.
  - The simplest of those types is **Option**, which can represent either a value or the absence of a value.
  - When more failure-related information is needed, other types can be used, including **Try** (which stores an exception) and
  - **Either** (which contains a substitute value).

- In general, exceptions are not well suited to a FP style.
  - They deviate from the core principle of value-returning functions, and often disrupt control flow embedded in higher-order functions.