

INFO0054-1
Programmation Fonctionnelle
Chapter 04: Exception handling

Christophe Debruyne
(c.debruyne@uliege.be)

References

- Chapter 4: Handling errors without exceptions.
Paul Chiusano and Runar Bjarnason. Functional Programming in Scala, Manning Publications, 2015.
- Recommended to consult:
 - [Option](#) in Scala's standard library
 - [Either](#) in Scala's standard library

Overview

- Exceptions cause side-effects
- Representing exceptions as values
- The Option data type
- The Either data type

Exception handling

- The term **exception handling** is preferred over error handling. Semantically speaking, not all exceptions are errors.
- **Throwing** and **catching exceptions** are side effects and thus break referential transparency. We will later see an example of that.
- But how do we handle exceptions in FP (and Scala)? The solution is surprisingly simple; **we will treat exceptions as values that are part of our functions' domain**. Those values can be processed by our pure functions or returned to the outer (non-pure) layer of our program.
- In this chapter, we will implement our own data types for handling exceptions, even though they are available in Scala's standard library. Again, this exercise allows us to gain a better understanding of the concept.

An example

- The function `foobar` will divide by 0 when we provide 5 as an argument. Notice that this happens outside the `try` block.
- A `catch` block is like a pattern matching block. The exception is matched against patterns. Whenever a match is found, the instructions corresponding with that match are executed.
- As expected, whenever 5 is provided as input, the function throws an error that is not handled.
- What if we substitute all occurrences of `x` with its definition? Is our program equivalent?

```
def foobar(n: Int): Int =  
  val x: Int = n / (n - 5)  
  try  
    x + 1  
  catch  
    case e: Exception => -9999
```

```
scala> :load ExceptionHandling.scala  
def foobar(n: Int): Int
```

```
scala> foobar(10)  
val res0: Int = 3
```

```
scala> foobar(5)  
java.lang.ArithmeticException: / by  
zero  
    at rs$line$1$.foobar(rs$line$1:2)  
    ... 39 elided
```

An example

- What if we substitute all occurrences of `x` with its definition? Is our program equivalent?
- No! **Exceptions thus break referential transparency.**
- There is another problem: **exceptions are not type-safe.**
 - Look at the signature of the function `foobar`. The function takes as input integers and returns integers. Nowhere is specified that exceptions may occur.

```
def foobar(n: Int): Int =  
  // val x: Int = n / (n - 5)  
  try  
    (n / (n - 5)) + 1  
  catch  
    case e: Exception => -9999
```

```
scala> :load ExceptionHandling.scala  
def foobar(n: Int): Int
```

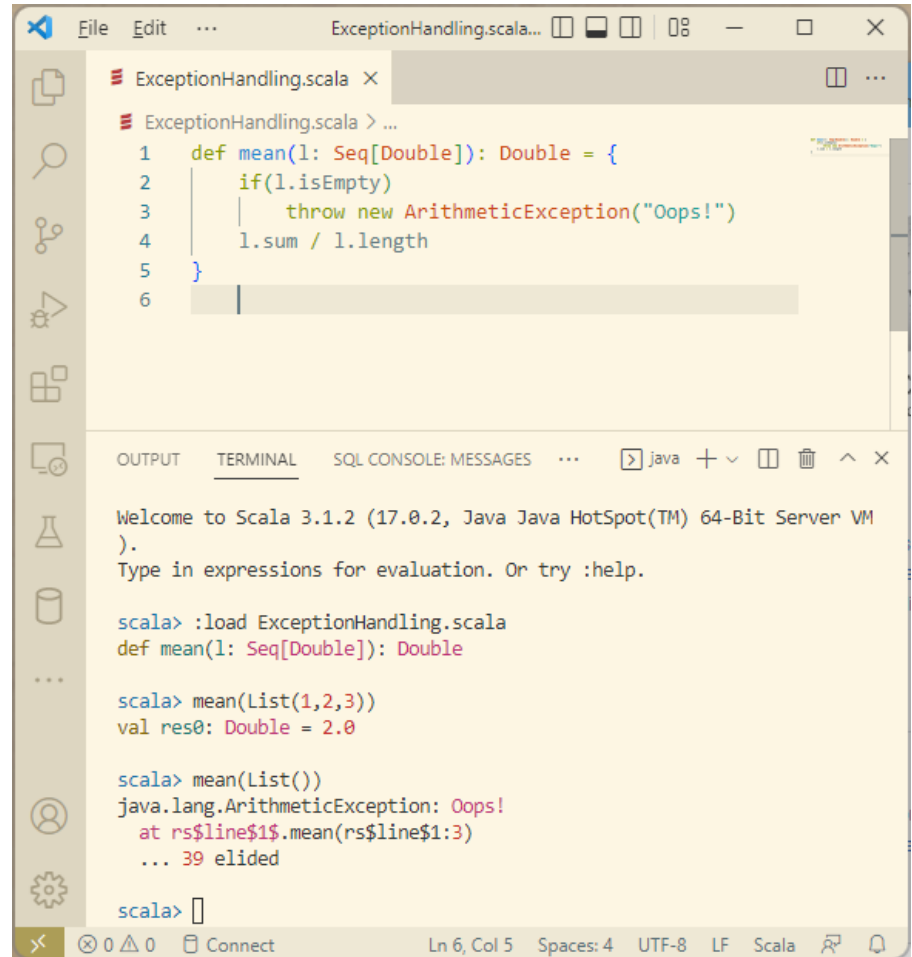
```
scala> foobar(10)  
val res0: Int = 3
```

```
scala> foobar(5)  
val res1: Int = -9999
```

How can we handle errors without losing the ability to centralize and consolidate "error logic"?

Calculating the mean

- Let's create a function that computes the mean of a sequence of doubles. The function takes as input a Seq of doubles, which is the common interface for lists, arrays, etc.
- Also notice that there are convenience functions for processing sequences of numeric values!
- The function mean makes assumptions: the list cannot be empty. It is therefore not defined for some inputs. We call such functions **partial functions**.
 - A function is typically partial when it makes assumptions about its inputs that are not implied by the input types.



```
ExceptionHandling.scala > ...  
1 def mean(l: Seq[Double]): Double = {  
2   if(l.isEmpty)  
3     throw new ArithmeticException("Oops!")  
4   l.sum / l.length  
5 }  
6  
  
Welcome to Scala 3.1.2 (17.0.2, Java Java HotSpot(TM) 64-Bit Server VM  
)  
Type in expressions for evaluation. Or try :help.  
  
scala> :load ExceptionHandling.scala  
def mean(l: Seq[Double]): Double  
  
scala> mean(List(1,2,3))  
val res0: Double = 2.0  
  
scala> mean(List())  
java.lang.ArithmeticException: Oops!  
  at rs$line$1$.mean(rs$line$1:3)  
  ... 39 elided  
  
scala> 
```

Avoiding throwing errors

Sentinel values

(e.g., null, NaN, -9999999.99, ...)

Problems:

- **Errors can silently propagate** because the caller forgot to check the value and the code may become **ambiguous**. E.g., when 0.0 is used as the bogus value.
- **Callers need to implement additional code to check the value**. This may grow rapidly if many such functions are used. This makes them furthermore **unfit for use by higher-order functions**.
- **Not applicable to polymorphic code**. For example, max returns the maximum value of a list of object of a particular type. If the list is empty, Scala does not know whether it is a list of Doubles, a list of a custom data type, ...

Additional arguments

Providing an additional argument specifying what should be returned on an exception makes the function total, but also has its problems:

- **Callers must know before hand of how to handle exceptions**, and
- **Callers are limited to the types of arguments that can be passed**.
- **We have not as much control over computation**. What if mean is used as part of a larger computation (e.g., computing standard deviations) and would like to abort everything if the mean is not defined? An additional parameter does not provide us that possibility.

By the way

During Object-Oriented Programming, you will see Exception Handling in the Java Programming Language. In Java, you have checked and unchecked exceptions. There are two disadvantages with *checked exceptions*:

- First, they force you to use try-catch blocks or have the method throw the same exception.
 - You will thus either create a bunch of catch blocks for each type of exception (except when you aim to catch all exceptions in one catch block, which may indicate bad code).
 - Or you will delegate all exception handling to outer layers of your code.
- Secondly, checked exceptions do not play well with generic functions. Exceptions may depend on the types on which generic functions operate. A generic function cannot anticipate all possible errors and providing errors as separate arguments is tedious. When combining Java with higher-order programming, we tend to resort to `RuntimeExceptions`. These exceptions do not need to be checked and can be thrown from anywhere in the program

The Option data type

- Representing in the return type that a function may not always have an answer.

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

- The return type reflects the possibility that a result may not be defined.
 - If a result is defined, we can use a **Some** to hold the value.
 - If a result is undefined, we use **None**.
- We are now able to **transform partial functions into total functions**.

The Option data type

```
import Option.{Some, None}
```

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

```
scala> mean(List(1,2,3,4,5))  
val res0: Option[Double] = Some(3.0)
```

```
scala> mean(List())  
val res1: Option[Double] = None
```

A lot of functions are partial. This example demonstrates how we can easily transform partial functions in total functions. In Scala, many of the functions in the standard library return **Option** (and **Either**) objects.

How are these **Option** objects used? These objects have functions. Funnily enough, many of these functions are shared with lists. You can think of **Option** objects as lists that can contain at most one value.

Basic functions on Option

In this example, we do not create a companion object. Instead, we define our functions in the body of our trait.

This will allow us to call these functions on instances of our traits, e.g.,

```
mean(List()).getOrElse("Oh no!")
```

Since those functions are applied on an instance of a trait, we need to specify that those functions have to operate on this object. Therefore, we must use `this`.

```
enum Option[+A]:  
  case Some(get: A)  
  case None  
  
  def map[B](f: A => B): Option[B] = this match  
    case None => None  
    case Some(a) => Some(f(a))  
  
  def getOrElse[B>:A](default: => B): B = this match  
    case None => default  
    case Some(a) => a  
  
  def flatMap[B](f: A => Option[B]): Option[B] =  
    map(f).getOrElse(None)  
  
  def orElse[B>:A](op: => Option[B]): Option[B] =  
    map(x => Some(x)).getOrElse(op)  
  
  def filter(f: A => Boolean): Option[A] =  
    flatMap(a => if (f(a)) Some(a) else None)
```

Basic functions on Option

`map` applies a function on the value of a `Some` and otherwise returns `None`. Notice that the function returns an `Option`.

```
def map[B](f: A => B): Option[B] = this match
  case None => None
  case Some(a) => Some(f(a))
```

`getOrElse` returns the result inside the `Some` case of the `Option` or a given default value if the option is `None`.

`B` must be equal to or a supertype of `A`.

```
def getOrElse[B>:A](default: => B): B = this match
  case None => default
  case Some(a) => a
```

This means that the default value is of the type `B`, but that the expression will not be evaluated until needed by `getOrElse`. For now, this is a detail. We will cover strictness and laziness next week.

Basic functions on Option

`flatMap` is similar to `map`. In `flatMap`, the function provided can itself fail. In other words, the function that is provided returns `Option` objects.

```
def flatMap[B](f: A => Option[B]): Option[B] =  
  map(f).getOrElse(None)
```

`orElse` returns the `Option` if its defined, and otherwise returns the `Option` that has been provided to `orElse`.

```
def orElse[B>:A](op: => Option[B]): Option[B] =  
  map(x => Some(x)).getOrElse(op)
```

Basic functions on Option

`filter` returns an `Option` value that satisfies a predicate. It can be used to convert successes into failures, for instance.

```
def filter(f: A => Boolean): Option[A] =  
    flatMap(a => if (f(a)) Some(a) else None)
```

Using Option

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

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

```
scala> variance(List())  
val res0: Option[Double] = None
```

```
scala> variance(List(1,2,3,4,5))  
val res1: Option[Double] = Some(2.0)
```

```
scala> variance(List(1,2,3,4,5)).getOrElse("Invalid!")  
val res2: Matchable = 2.0
```


Lifting

- Does this mean we have to write all our functions in terms of `Option`? No.
- Why not? Because we can lift "ordinary" functions into functions that operate on `Option`.
- How? We can **lift** any function that we have to operate withing the context of a single `Option` value.

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

- `lift(f)` returns a function that maps `None` to `None` and applies `f` to the contents of `Some`. `f` does not need to be aware of `Option`.
- Demonstration

```
def ceil0 = lift(math.ceil)
-----
scala> ceil0(variance(List()))
val res0: Option[Double] = None
scala> ceil0(mean(List(2,3,2,3,2)))
val res1: Option[Double] = Some(3.0)
```

Notice how we did not need to rewrite the `math.ceil` function to work with `Option` values.

We can do this to any function!

Wrapping exceptions

- How do we handle APIs that throw exceptions? To illustrate the idea, we assume that we have a game that requests a user to enter their year of birth (YOB) via a terminal. The YOB is used to determine whether the user is an adult by testing $YOB + 18 \leq 2022$.

```
def isAdult(yob: Int): Boolean = yob + 18 <= 2022
```

- The YOB is an integer, but a terminal does not prevent a user from entering arbitrary strings such as "Let's break this!" Input provided by a terminal are read as strings. String objects in Scala have a function `toInt` that tries to parse the string value as an integer. If the string does not contain a valid string-representation of an integer, it throws a `NumberFormatException`.

```
scala> "2002".toInt  
val res0: Int = 2002
```

```
scala> "Let's break this!".toInt  
java.lang.NumberFormatException: For input string: "Let's break this!"  
...
```

- How to convert the exception-based API of `toInt` to `Option`?

Wrapping exceptions

- We accept `a` in a `non-strict` manner. The evaluation of the expression is not passed as an argument, but the expression. The expression will be evaluated in the function when it is used. This allows us to catch any exceptions that occur while evaluating `a` and convert those exceptional cases to `None`.

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

- Notice how we pass an expression to `Try`. If `Try` was strict, it would have raised an error before calling `Try`. The evaluation of that expression happens inside `Try`. We then use `map` on your `Option` object to determine whether the user is an adult.

```
def parseIsAdult(yob: String): Option[Boolean] =  
  val yob0 = Try(yob.toInt)  
  yob0.map(isAdult)
```

Wrapping exceptions: demonstration

```
scala> parseIsAdult("test")  
val res0: Option[Boolean] = None
```

```
scala> parseIsAdult("1999")  
val res1: Option[Boolean] = Some(true)
```

```
scala> parseIsAdult("test").getOrElse(false)  
val res2: Boolean = false
```

With the data types and functions provided by Scala, you should never have to modify a function to work with optional values.

The Either data type

- Look at

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

- Notice how we discard information about the exception. With **Option**, we do not "care" about the specific exception. We just want to represent successes and failures in some way.
- What if we want to keep track of the (type of) exception? Well, we can avail of **Either**. An **Either** object allows us to represent successes and failures, just like **Option**. The difference between the two is that **Either** allows us to store values for exceptions.

The Either data type

- Again, **Either** is part of Scala's standard library, but we will implement our own version to gain a better understanding of the concept.

```
sealed trait Either[+E, +A] {  
  // TO BE COMPLETED :-)  
}
```

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

The book uses an **enum** instead of a **sealed trait**. There is no real difference. This is just to show you an alternative.

- The "E" stands for "error."
- "Right" is a play on words as it refers to "correct" on a success.
- The opposite of "right" is "left."

Note: Either is not only used for exception handling. It is also used when one needs to process one of two possible cases without resorting to custom data types.

Either: example 1

```
def mean(l: Seq[Double]): Either[String, Double] = {  
    if(l.isEmpty) Left("Empty list.")  
    else Right(l.sum / l.length)  
}
```

```
scala> mean(List(1,2,3,4,5))
```

```
val res0: Either[String, Double] = Right(3.0)
```

```
scala> mean(List())
```

```
val res1: Either[String, Double] = Left(Empty list.)
```

Either: example 2

```
def safeDiv(n: Int, m: Int): Either[Exception, Int] = {  
  try Right(n / m)  
  catch { case e: Exception => Left(e) }  
}
```

```
scala> safeDiv(20, 4)  
val res0: Either[Exception, Double] = Right(5.0)  
  
scala> safeDiv(20, 0)  
val res1: Either[Exception, Double] = Left(...: / by zero)
```


Using Either, Option, ...

```
> val x = List.range(0, 50, 3).map((x) => safeDivision(x, x % 4))
```

```
val x: List[Either[Exception, Int]] =  
List(Left(java.lang.ArithmeticException: / by zero), Right(1), Right(3),  
Right(9), Left(java.lang.ArithmeticException: / by zero), Right(5),  
Right(9), Right(21), Left(java.lang.ArithmeticException: / by zero),  
Right(9), Right(15), Right(33), Left(java.lang.ArithmeticException: / by  
zero), Right(13), Right(21), Right(45), Left(java.lang.ArithmeticException:  
/ by zero))
```

By including exceptions are part of our domain, we can write functional programs that can elegantly process those. In this example, we compute an average "safely" ignoring the exceptions.

```
> val (tot, count) =
```

```
    x.map(_._1.toList).flatten.foldRight((0.0, 0))((x, y) => (x+y._1, 1+y._2))
```

```
val tot: Double = 184.0
```

```
val count: Int = 12
```

```
> val avg = tot / count
```

```
val avg: Double = 15.333333333333334
```

Lexicon

- Exception handling – Gestion d'exceptions