Lecture 6
Handling Errors

18.4.2023

Iflaah Salman

LUT University

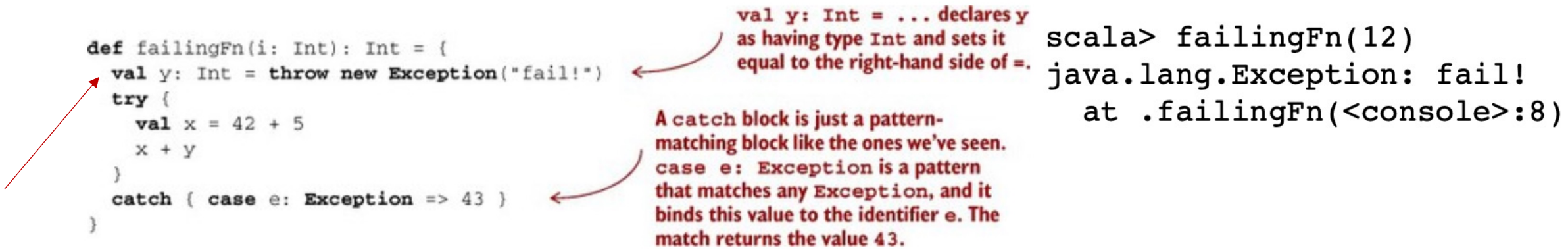
Handling Errors

- Raising and handling errors functionally!
  - Throwing exceptions is a side effect.
- Represent failures and exceptions with ordinary values.
- Enables writing higher-order functions
  - abstract out common patterns of error handling and recovery.
- The functional solution:
  - Returning errors as values
    - is safer
    - retains referential transparency,
    - and through the use of higher-order functions.

LUT University

Why do exceptions break referential transparency?

Listing 4.1. Throwing and catching an exception



- **y** is not referentially transparent.
- An RT expression may be substituted with the value it refers to
  - this substitution should preserve program meaning

LUT University

The good and Bad Aspects of Exceptions

```
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`.

```
scala> failingFn2(12)
res1: Int = 43
```

- RT expression does not depend on context and can be reasoned locally, whereas the meaning of non-RT expressions is context-dependent and requires more global reasoning.
- The expression **throw new Exception("fail")** is context-dependent
  - it takes on different meanings depending on which try block (if any) it's nested within.

- If we substitute throw new Exception("fail!") for **y** in x + y,
- it produced a different result,
- *because* the exception is now raised inside a try block that will catch the exception and return 43.

LUT University

Problems with Exceptions

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

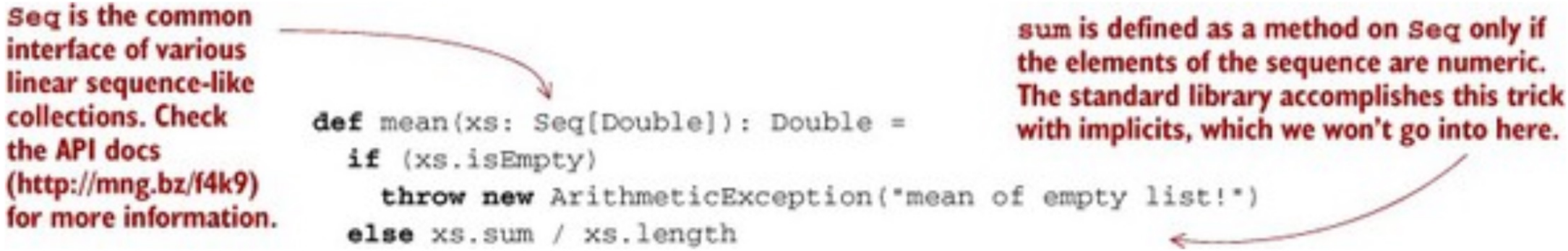
- Exceptions break RT and introduce context dependence
  - moving us away from the simple reasoning of the substitution model
- Exceptions are not type-safe.
  - The type of **failingFn, Int => Int** tells us nothing about the fact that exceptions may occur
  - the compiler will certainly not force callers of **failingFn** to make a decision about how to handle those exceptions.
  - If we forget to check for an exception in **failingFn**,
    - this won't be detected until runtime.

The Technique

Exceptions allow us to consolidate and centralize error-handling logic

- The technique is based on an old idea:
  - instead of throwing an exception
    - we return a value that indicates an exceptional condition
- There's a new generic type
  - for these "possibly defined values",
  - and use of higher-order functions to encapsulate common patterns of handling errors.
- Completely type safe
- full assistance from the type-checker in forcing us to deal with errors

Possible Alternatives



First possibility:

1. Return a bogus value of type Double.
   - Either 0.0/0.0 –> Double.Nan or other sentinal value
2. Return Null – instead of the value of the needed type.

We reject this solution.

Reasons for Rejection: for simply returning a value

- the caller can forget to check this condition and won't be alerted by the compiler, thus affecting the subsequent code.
- It results in a fair amount of code at all sites
  - With If statements to validate whether the caller has received the correct result.
- It is not applicable to polymorphic code.

$$\text{def max[A](xs: Seq[A])(greater: (A,A) => Boolean): A}$$

  - If the input is empty, we can't invent a value of type A.
  - Nor can null be used here, since null is only valid for non-primitive types (Double, Int)
- Defining policies for the function/method callers to deal with the result.

Second Possiblity & it's rejection

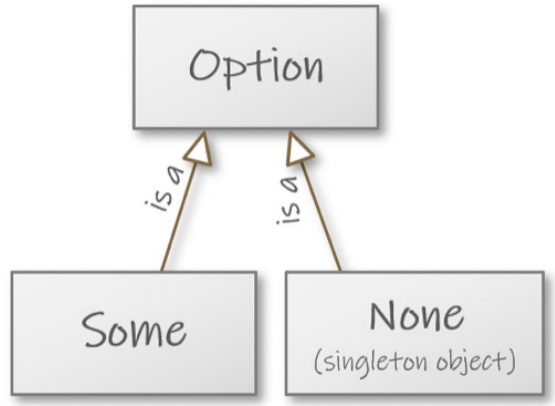
- Force the caller to supply an argument that tells us what to do in case the method doesn't know how to handle the input.

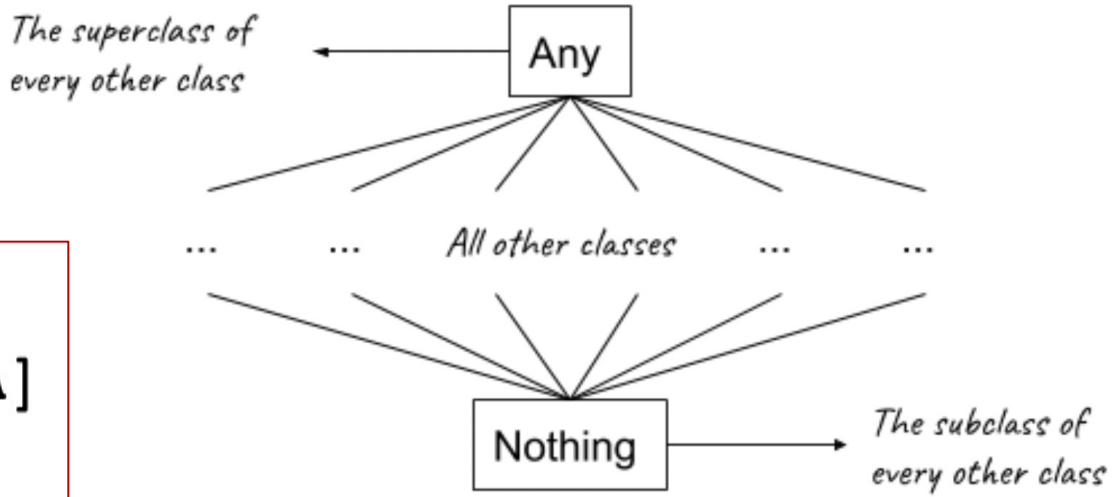
```
def mean_1(xs: IndexedSeq[Double], onEmpty:
Double): Double =
  if (xs.isEmpty) onEmpty
  else xs.sum / xs.length
```

- A total function, but with drawbacks:
  - The immediate caller should have direct knowledge of how to handle undefined cases and limit (the result) to returning Double.
    - What if we want to handle larger computations, and abort in case of undefined mean?
    - Or, we want to take a different logical branch.

Option type

- The solution is to represent explicitly in the return type that a function may not always have an answer.
- Scala library provides a type, Option.
- Can be thought of like a List,
  - But contains only one element.






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

```
package scala

sealed abstract class Option[A]

case class Some[A](a: A) extends Option[A]
case object None extends Option[Nothing]
```

Option has two cases:

- Some – defined (or have something)
- None – undefined (has nothing)

https://plus.cs.aalto.fi/o1/2022/w07/ch05/

Option type

```
scala> val optInt: Option[Int] = Some(1)
optInt: Option[Int] = Some(1)

scala> val optString: Option[String] = Some(1)
<console>:11: error: type mismatch;
 found   : Int(1)
 required: String
       val optString: Option[String] = Some(1)
```

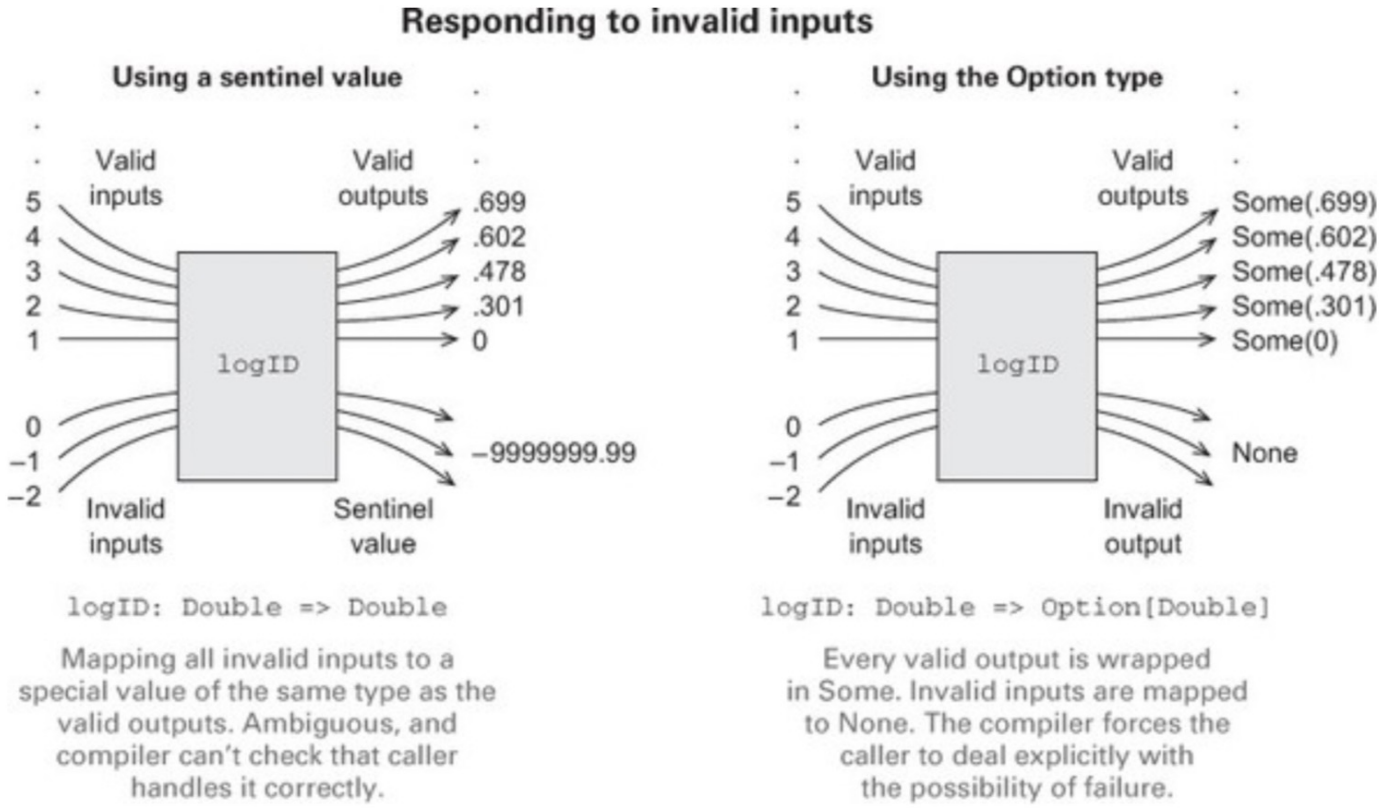
- Some[A] is an implementation of Option[A]
- Some[Int] is a valid implementation for Option[Int]
- Not for Option[String]

```
scala> val optInt: Option[Int] = None
optInt: Option[Int] = None

scala> val optString: Option[String] = None
optString: Option[String] = None
```

- None doesn't have a type parameter
- None extends Option[Nothing]
- It is valid
  - because Nothing is a subclass of every other class.

Option type

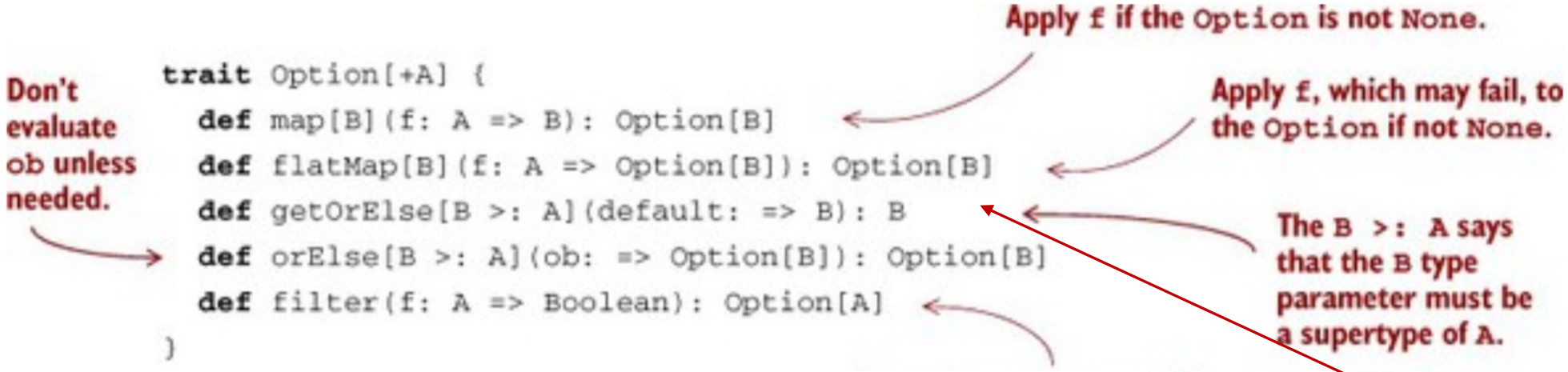



```
def mean(xs: Seq[Double]): Option[Double] =
  if (xs.isEmpty) None
  else Some(xs.sum / xs.length)
```

- Using Option for the **mean** method
- The return type now reflects the possibility that the result may not always be defined.
- We still always return a result of type Option[Double]

Basic Functions on Option

Option is used throughout Scala standard library [*Lecture 4*]




```
case class Employee(name: String, department:
String)

def lookupByName(name: String): Option[Employee] =
...

val joeDepartment: Option[String]
```

indicates that the argument is of type B, but won't be evaluated until it's needed by the function.
[it relates to the concept of non-strictness]

User Scenarios on Option


```
case class Employee(name: String, department:
String)

def lookupByName(name: String): Option[Employee] =
...

val joeDepartment: Option[String] =lookupByName("Joe").map(_.department)
```

- **lookupByName("Joe")** returns an **Option[Employee]**
  - **Transform**ing using **map** to pull out the **Option[String]** representing the department.
- No need to explicitly check the result of **lookupByName("Joe")**
  - simply continue the computation as if no error occurred, inside the argument to map.
- If it returns None, this will abort the rest of the computation
  - map will not call the _.department function.

User Scenarios on Option

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

- Using **filter** to convert successes into failures (only a learning-case)
  - if the successful values don't match the given predicate.
- A common pattern is to
  - transform an Option via calls to **map**, **flatMap**, and/or **filter**
  - *Then,* use **getOrElse** to do error handling at the end.
- **getOrElse** is used here to convert from an Option[String] to a String

Option type: error handling

- Returning errors as ordinary values can be convenient
- The use of higher-order functions enables achieving the same sort of consolidation of error-handling logic we would get from using exceptions.
- No need to check for None at each stage of the computation
  - we can apply several transformations and
    - Then, check for and handle None when we're ready.

References

- Hunt, J. (2018). A Beginner's Guide to Scala, Object Orientation and Functional Programming. In *A Beginner's Guide to Scala, Object Orientation and Functional Programming*. Springer International Publishing. https://doi.org/10.1007/978-3-319-75771-1
- Chiusano, P., & Bjarnason, R. (2014). *Functional Programming in Scala*. Manning publications.
- Sfregola, D. (2021). *Get Programming with Scala*. Manning.