```
mirror_mod.mirror_object
  million object to mirror
peration == "MIRROR_X":
mirror_mod.use_x = True
mirror_mod.use_y = False
mirror_mod.use_z = False
 _operation == "MIRROR_Y"
lrror_mod.use_x = False
Lrror_mod.use_y = True
mirror_mod.use_z = False
  operation == "MIRROR_Z";
  rror mod.use x = False
  lrror_mod.use_y = False
 rror_mod.use_z = True_ecture 6
  mtext scene Flandling Errors
   irror_ob.select = 0
  bpy.context.selected_obj
   ata.objects[one.name].se
                          18.4.2023
  int("please select exactle
  -- OPERATOR CLASSES --- Iflaah Salman
   vpes.Operator):
```

x mirror to the select
ject.mirror\_mirror\_x"
or X"



## 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.



# Why do exceptions break referential transparency?

#### Listing 4.1. Throwing and catching an exception

```
val y: Int = ... declares y
                                                                                      scala> failingFn(12)
                                                          as having type Int and sets it
def failingFn(i: Int): Int = {
                                                          equal to the right-hand side of =.
                                                                                      java.lang.Exception: fail!
  val v: Int = throw new Exception ("fail
                                                                                         at .failingFn(<console>:8)
                                                   A catch block is just a pattern-
    val x = 42 + 5
                                                   matching block like the ones we've seen.
                                                   case e: Exception is a pattern
                                                   that matches any Exception, and it
  catch { case e: Exception =>
                                                   binds this value to the identifier e. The
                                                   match returns the value 43.
```

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



## 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.
```

- 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.

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

- 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.



#### Problems with Exceptions

- 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

```
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.

if (xs.isEmpty)

(http://mng.bz/f4k9)
for more information.

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.

if (xs.isEmpty)

throw new ArithmeticException("mean of empty list!")

else xs.sum / xs.length
```

#### 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.

```
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.

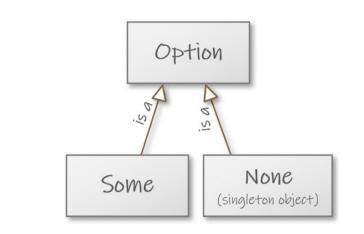
#### Option type

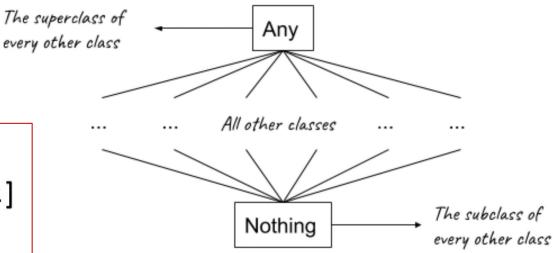
- The solution is to represent explicitly in the return type that a function may not always have an answer.
- Scala library provides a type, Option.
- Can be thought of like a List,
  - But contains only one element.

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)
```

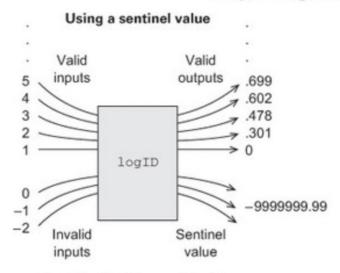
- Some[A] is an implementation of Option[A]
- Some[Int] is a valid implementation for Option[Int]
- Not for Option[String]

```
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.

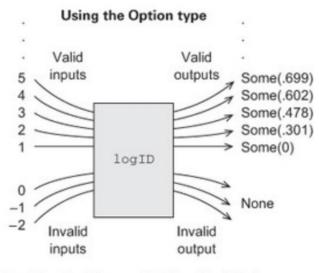
### Option type

#### Responding to invalid inputs



logID: Double => Double

Mapping all invalid inputs to a special value of the same type as the valid outputs. Ambiguous, and compiler can't check that caller handles it correctly.



logID: Double => Option[Double]

Every valid output is wrapped in Some. Invalid inputs are mapped to None. The compiler forces the caller to deal explicitly with the possibility of failure.

```
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

val joeDepartment: Option[String]

. . .

Option is used throughout Scala standard library [Lecture 4]

```
Apply f if the Option is not None.
           trait Option[+A] {
 Don't
                                                                        Apply £, which may fail, to
             def map[B] (f: A => B): Option[B]
                                                                        the Option if not None.
 evaluate
 ob unless
             def flatMap[B](f: A => Option[B]): Option[B]
 needed.
             def getOrElse[B >: A] (default: => B): B
             def orElse[B >: A](ob: => Option[B]): Option[B]
                                                                           that the B type
                                                                           parameter must be
             def filter(f: A => Boolean): Option[A]
                                                                           a supertype of A.
                                                                    e if
case class Employee(name: String, department:
                                                                    fy f.
String)
def lookupByName(name: String): Option[Employee] =
```

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

```
Joe's dept. if Joe is an employee

lookupByName("Joe").map(_.department)

None if Joe is not an employee
```

```
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

- 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. <a href="https://doi.org/10.1007/978-3-319-75771-1">https://doi.org/10.1007/978-3-319-75771-1</a>
- Chiusano, P., & Bjarnason, R. (2014).
   Functional Programming in Scala.
   Manning publications.
- Sfregola, D. (2021). *Get Programming with Scala*. Manning.