Functional Programming in Scala

Chapter 4
Handling errors without exceptions

Jordan Moldow

Oct. 9, 2014

Handling errors without exceptions

- Pure functions are like mathematical functions: $f(x)$
  - Always returns the same single result
  - Produces no side effects in the outside world
- Throwing exceptions is a side effect, breaks referential transparency

Handling errors without exceptions

Key ideas:

- Use container type to expand codomain (range) of functions
- Return errors as values
- Use higher-order functions to
  - consolidate of error handling logic
  - preserve composability
  - "lift" normal functions to error handling functions

Handling errors without exceptions

- The good and bad aspects of exceptions
- Possible alternatives to exceptions
- The Option data type
  - Usage patterns for Option - the Option functor
  - Option composition and lifting - the monad laws
  - Wrapping exception-oriented APIs
- The Either data type
- Exercises

Throwing exceptions breaks referential transparency

```
1  def failingFn(i: Int): Int = {
2    val y: Int = throw new Exception("fail!")
3    try {
4      val x = 42 + 5
5      x + y
6    }
7    catch { case e: Exception => 43 }
8  }
9
10 scala> failingFn(12)
11 java.lang.Exception: fail!
12
13 def failingFn2(i: Int): Int = {
14   try {
15     val x = 42 + 5
16     x + ((throw new Exception("fail!")): Int)
17   }
18   catch { case e: Exception => 43 }
19 }
20
21 scala> failingFn2(12)
22 res1: Int = 43
```

The bad aspects of exceptions

- Exceptions break the substitution model of reasoning
  - `throw new Exception("fail")` is context-dependent, taking on different meanings depending on which block it's in
- Exceptions can't be described in the type system
  - Does `f: Int => Int` always return? Might it fail? What exceptions might it throw? Who knows!
  - Java checked exceptions don't work with higher-order functions

The good aspects of exceptions

- Consolidate, centralize error-handling logic
- Error info (messages, stack traces, memory dumps)
- Exception subclasses
- Functions don't have to handle callee errors

Problem: Procedures aren't always total

- Total function: always has an output (like a mathematical function)
- Partial function: output undefined for some inputs
  - `mean: List[Double] => Double`
  - `sqrt: Double => Double`
  - (Not to be confused with partially applied functions)
- Pure functions must be total
- Need strategy for turning partial function into total function

Option 1 - Return bogus value in error case

- Return a sentinel value, or `NaN`, or `null`
- Can't attach extra information to errors
- Must manually check result at call sites / before uses of value
- No applicable in polymorphic code
- Requires special calling convention
- Not easy to compose
- Not easy to pass to higher-order functions

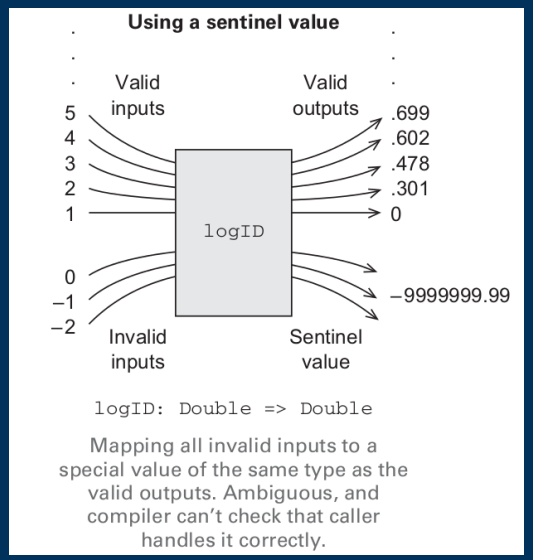
Option 2 - Return integer error codes

- Like assembly, C, Unix programs, etc.
- Not compatible with type system
- Plus all the bad things about Option 1
  - Especially bugs with not correctly error checking at call sites
  - `kill(fork())` bug - http://rachelbythebay.com/w/2014/08/19/fork/

Option 3 - Caller-provided default values

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

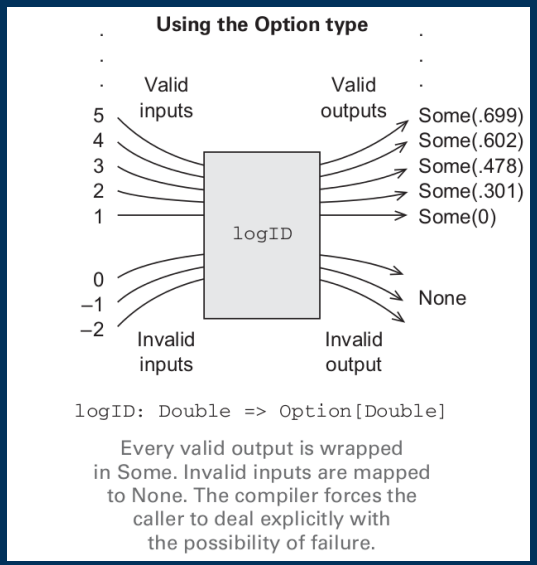
- Limited to passing / returning `Double`
- Parameter can only be used as a default value
- In error cases, can't branch or abort
- Immediate callers must decide default value



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.

Option 4 - The Option data type

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




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

`mean` is now total

None is not null!

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

- There is no such thing as a "generic" None
  - None $\notin$ A
  - Can't return None from function that returns an A
- Every usage of None must be assigned to a specific type
  - None:Option[A] $\neq$ None:Option[B], None:Option[A] $\notin$ Option[B]
- Type system prevents null pointer dereference

`Option` as a container

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

Think of `Option[A]` as a `List[A]` with length ≤ 1

- `None:Option[A]` $\approx$ `Nil:List[A]`
- `Some(a:A)` $\approx$ `List(a:A)`

Usage patterns for Option

```
1  sealed trait Option[+A] {
2    // Apply f if the Option is not None.
3    def map[B](f: A => B): Option[B]
4
5    // The B >: A says that the B type parameter must be
6    // a supertype of A.
7    def getOrElse[B>:A](default: => B): B
8
9    // Apply f, which may fail, to the Option if not None.
10   def flatMap[B](f: A => Option[B]): Option[B]
11
12   // `ob: => Option[B]` means don't evaluate ob unless needed.
13   // The argument is non-strict / evaulated lazily
14   // (just like if-else short-circuiting) - see chapter 5!
15   def orElse[B>:A](ob: => Option[B]): Option[B]
16
17   // Convert Some to None if the value doesn't satisfy f.
18   def filter(f: A => Boolean): Option[A]
19 }
20 case class Some[+A](get: A) extends Option[A]
21 case object None extends Option[Nothing]
```

Exercise 4.1

```scala
sealed trait Option[+A] {
  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](ob: => Option[B]): Option[B]
    map(Some(_)) getOrElse ob

  def filter(f: A => Boolean): Option[A] = {
    flatMap(a => if (f(a)) Some(a) else None)
  }
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

Usage scenarios for Option

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def getOrElse[B>:A](default: => B): B
  def flatMap[B](f: A => Option[B]): Option[B]
  def orElse[B>:A](ob: => Option[B]): Option[B]
  def filter(f: A => Boolean): Option[A]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

Usage patterns for Option

```
1  sealed trait Option[+A] {
2    def map[B](f: A => B): Option[B]
3    def getOrElse[B>:A](default: => B): B
4    def flatMap[B](f: A => Option[B]): Option[B]
5    def orElse[B>:A](ob: => Option[B]): Option[B]
6    def filter(f: A => Boolean): Option[A]
7  }
8  case class Some[+A](get: A) extends Option[A]
9  case object None extends Option[Nothing]
```

1. Some initial computation `f: A => Option[B]` may fail
2. Apply further computations with `map`, `flatMap`
   - Subsequent computations only run when there is still a value
   - In error cases, `None` is carried through the computations
3. Optionally `filter` on predicates to generate error
4. Do error handling at end with `getOrElse` or `orElse`
   - `getOrElse` provides default value
   - `OrElse` provides new chain of computations to try

Language Comparison

Scala:

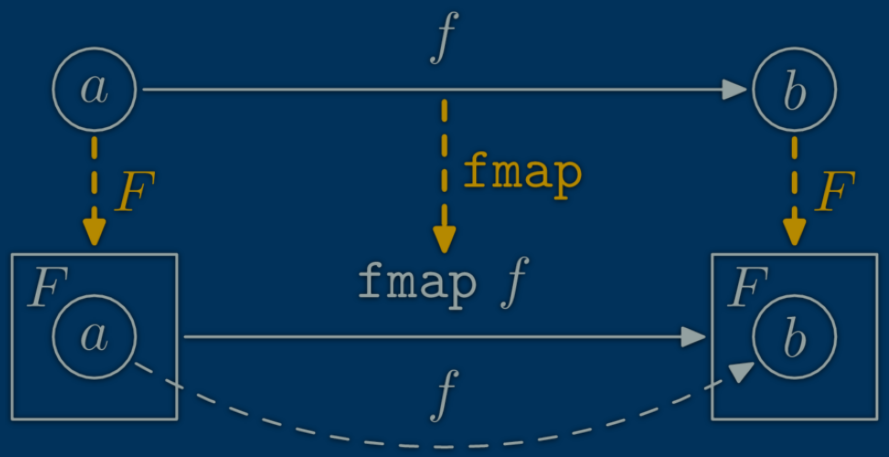
```scala
val dept: String =
  lookupByName("Joe").   // Impossible to forget None check.
  flatMap(_.dept).       // Type system does not allow you to.
  filter(_ != "Accounting").
  getOrElse("Default Dept")
```

Python:

```python
employee = lookupByName("Joe")
# If you forget this line
if employee is not None:
  # this will raise AttributeError.
  department = employee.dept
  if (department is not None) and (department != "Accounting"):
    dept = department
```

The Option functor

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```



1. `map identity = identity`
2. `map (f compose g) = (map f) compose (map g)`

More in chapter 10!

Option composition - Exercise 4.3

```
1  sealed trait Option[+A] {
2    def map[B](f: A => B): Option[B]
3    def flatMap[B](f: A => Option[B]): Option[B]
4  }
5  case class Some[+A](get: A) extends Option[A]
6  case object None extends Option[Nothing]
7
8
9  // Return None if any input is None.
10 // Otherwise, apply the function to the values.
11 def map2[A,B,C](a:Option[A],b:Option[B])(f:(A,B)=>C):Option[C]
12 def flatMap2[A,B,C](a:Option[A],b:Option[B])(f:(A,B)=>Option[C])
       :Option[C]
13 def map3[A,B,C,D](a:Option[A],b:Option[B],c:Option[C])
14                   (f:(A,B,C)=>D):Option[D]
```

Option composition - Exercise 4.3

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]


// Return None if any input is None.
// Otherwise, apply the function to the values.
def map2[A,B,C](a:Option[A],b:Option[B])(f:(A,B)=>C):Option[C]
def flatMap2[A,B,C](a:Option[A],b:Option[B])(f:(A,B)=>Option[C])
    :Option[C]
def map3[A,B,C,D](a:Option[A],b:Option[B],c:Option[C])
                 (f:(A,B,C)=>D):Option[D]
```

```scala
map2(a,b)(f) = a flatMap (aa => b map (bb => f(aa, bb)))
flatMap2(a,b)(f) = a flatMap (aa => b flatMap (bb => f(aa, bb)))
map3(a,b,c)(f) = a flatMap (aa => b flatMap(bb => c map (cc => f
    (aa, bb, cc))))
```

Option lifting

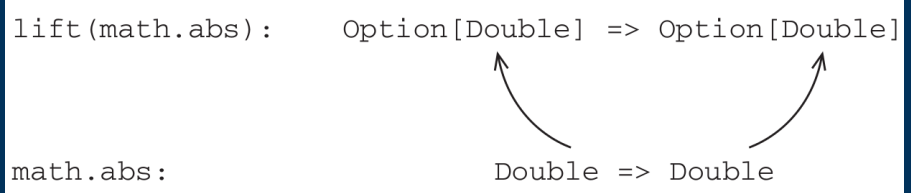
```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]

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

- Take ordinary functions, and lift them to functions on `Option`
- Can design API of `A => B`, `A => Option[B]` functions
- Don't about writing all your functions to accept `Option[A]`
- Can lift buildin Java, Scala functions

Option lifting

```
1  sealed trait Option[+A] {
2    def map[B](f: A => B): Option[B]
3    def flatMap[B](f: A => Option[B]): Option[B]
4  }
5  case class Some[+A](get: A) extends Option[A]
6  case object None extends Option[Nothing]
7
8  def lift[A,B](f: A => B): Option[A] => Option[B] =
9    _ map f
10 def flatLift[A,B](f: A => Option[B]): Option[A] => Option[B] =
11   _ flatMap f
12
13 val absO: Option[Double] => Option[Double] = lift(math.abs)
```

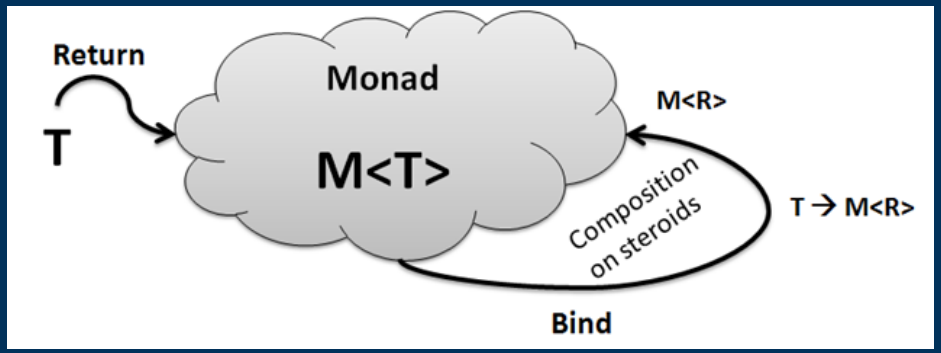
Lifting functions



`lift(f)` returns a function which maps `None` to `None` and applies `f` to the contents of `Some`. `f` need not be aware of the `Option` type at all.

Option monad

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
  val bind = flatMap
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]

def unit[A](a: A): Option[A] =
  Some(a)
val return = unit
```



Option monad

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
  val bind = flatMap
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]

def unit[A](a: A): Option[A] =
  Some(a)
val return = unit
```

```
unit(aa) bind f == f(aa)
a bind unit == a
a bind (aa=>f(aa) bind g) == (a bind f) bind g
```

Wrapping exception-oriented APIs

```
1  // We accept the A argument non-strictly,
2  // so we can catch any exceptions that
3  // occur while evaluating a and convert them to None.
4  def Try[A](a: => A): Option[A] =
5    try Some(a)
6    catch { case e: Exception => None }
```

Exercises