Updated: 2021-10-20

4.0
Functional Composition

Northeastern University

What exactly is functional composition?

- We’ve already seen “higher-order functions/methods”. These are methods like ***map*** for ***List*** which takes a ***function*** as one or more of its parameters.
  - But what if we apply a function to a function/method? I think we can call that “functional composition.”
  - Here are a couple of simple examples:
    - `f andThen g`
    - `f compose g`
  - These are functional composition because we start with a function, apply it to a parameter which is also a function and the result is yet another function!

Example of functional composition

- Let's suppose we have a function *f* which takes two parameters, ***x*** and ***y***. But what we really want is a function ***g*** that takes the same two parameters, but in the order ***y*** then ***x***?
- Let's write a method/function which will convert one form to the other:

```scala
def swapParams[T1, T2, R](f: (T1, T2) => R): (T2, T1) => R = ???
```

Example of functional composition: swapParams

```
def swapParams[T1, T2, R](f: (T1, T2) => R): (T2, T1) => R =
      (t2,t1) => f(t1,t2)

def div(x: Double, y: Double) =
   x / y
val g = swapParams(div)
div(1, 2)
g(2, 1)
```

The pattern (left side of =>) pertains to the Resulting function; the expression (right side of =>) pertains to the original function.

Review: Option/Try
Introduce: Either

- We use *Option[T]*
  - to make it explicit when we may or may not have a *T* value;
  - thus we avoid the use of null;
  - to “wrap” an object returned from a java method which is ***Nullable***.
- We use *Try[T]*
  - to make it explicit when we may have a *T* value or instead have an exceptional condition;
  - thus we generally avoid throwing exceptions;
  - to “wrap” an expression that might throw an exception.
- We use *Either[P,Q]*
  - when we might have *either* a *P* or a *Q*.
  - as usual, we have two cases:
    - *case class Left[P](p: P) extends Either[P,Nothing];*
    - *case class Right[Q](q: Q) extends Either[Nothing,Q].*
  - the *Right* case is (asymmetrically) treated as the *right* case.

Either

- For example, a numeric String can be parsed as a ***Double*** or an ***Int*** (or neither).

```
scala> :paste
// Entering paste mode (ctrl-D to finish)
  val rDouble = """(-?)([0-9]*)\.([0-9]+)""".r
  val rInt = """(-?)([0-9]+)""".r
  def parse(s: String): Option[Either[Int,Double]] = s match {
    case rDouble(_, _, _) => Some(Right(s.toDouble))
    case rInt(_, _) => Some(Left(s.toInt))
    case _ => None
  }
// Exiting paste mode, now interpreting.
scala> parse("3.1415927")
res0: Option[Either[Int,Double]] = Some(Right(3.1415927))
scala> parse("3")
res1: Option[Either[Int,Double]] = Some(Left(3))
scala> parse("X")
res2: Option[Either[Int,Double]] = None
```

Option Review (1)

- Avoiding exceptions/nulls using *Option*
  - First, what's wrong with nulls (and exception)?
    - nulls (in Java) are for lazy programmers who don't mind running into a null-pointer-exception every now and then. The problem is that they don't ***force*** the caller to check the result.
    - exceptions are side-effects!
  - We've briefly seen this before, for example, in the ***List*** method ***find***:

    def find(p: (A) ⇒ Boolean): Option[A]
    Finds the first element of the list satisfying a predicate, if any.
    p the predicate used to test elements.
    returns an option value containing the first element in the list that satisfies p, or *None* if none exists.

  - *Option*, therefore, is a **container** whose value is either a *Some* (a wrapper) of a valid value, or *None*.

Option (2)

- Creating *Option* values:

```
scala> Some("hello")
res1: Some[String] = Some(hello)
scala> None
res2: None.type = None
scala> Option(null)
res3: Option[Null] = None
```

Useful if using a Java library that might return a *null* value

- Using *Option* values — simple ways:

```
scala> val l = List(1,2,3)
l: List[Int] = List(1, 2, 3)
scala> val y = 3
y: Int = 3
scala> val x = l.find{_==y}
x: Option[Int] = Some(3)
scala> x.isDefined
res11: Boolean = true
scala> x.get
res10: Int = 3
scala> x match {case Some(n) => println(s"found $n"); case None => println("not found")}
found 3
scala> x.getOrElse("not found")
res12: Any = 3
scala> val y = 5
y: Int = 5
scala> val x = l.find{_==y}
x: Option[Int] = None
scala> x match {case Some(n) => println(s"found $n"); case None => println("not found")}
not found
scala> x.getOrElse("not found")
res13: Any = not found
```

It's possible to use *Option* values this way but definitely not recommended!

Try

- Similar to *Option[T]*, *Try[T]* is a container that has one of two possible values: a *T* <u>or</u> an exception
  - The successful form is *Success(t)* where t: T
  - The unsuccessful form is *Failure(x)* where x: Throwable
- As we discussed before, *Try(expression)* is a factory method which evaluates *expression* lazily (call-by-name) thus being able to catch any exceptions <u>inside</u> *Try.apply*.

Lift, map2, flatMap, “for comprehensions”

Fasten your seat belts!

A simple conversion tool

- Since the customary unit for temperature in the US is Fahrenheit, we decide to write a converter.
- We type in the temperature and out comes the value in Celsius. Simple, right?
- We know that sometimes people make mistakes and type in the wrong thing. Like “82F” instead of “82”; or “”; or “covfefe”
- We should try to take care of such situations.

fToC

```
object TemperatureConverter extends App {
    def fToC(x: Double): Double = (x -32) * 5 / 9
    def fToC(x: String): String = x.toDoubleOption match {
        case Some(f) => fToC(f).toString
        case None => "invalid input"
    }
    val scanner = new java.util.Scanner(System.in)
    System.err.print("Temperature in Fahrenheit? ")
    val f = scanner.nextLine()
    println(fToC(f))
}
```

Running it…

Temperature in Fahrenheit? 90

32.22222222222222

Temperature in Fahrenheit? covfefe

invalid input

fToC

```
object TemperatureConverter extends App {
    def cToF(x: Double): Double = x * 9 / 5 + 32
    def cToF(x: String): String = x.toDoubleOption match {
        case Some(c) => cToF(c).toString
        case None => "invalid input"
    }
    val scanner = new java.util.Scanner(System.in)
    System.err.print("Temperature in Celsius? ")
    val c = scanner.nextLine()
    println(cToF(c))
}
```

Thoughts?

- The logic of the *cToF(Double)* method is obviously necessary;
- But the logic of *cToF(String)* method is rather repetitive. And we hate to repeat ourselves (DRY).
- Wouldn’t it be nice if there was a function that could take an *Option[X]* and a function *X=>Y*, resulting in an *Option[Y]*?
- Then, we’d be able to write the *fToC* and *cToF* methods with String parameters much more easily (and elegantly).

A better way of dealing with instances of *Option** (1)

- Lift
  - First, wouldn't it be nice if, whenever we had a function ***f:*** *A=>B*, we could create a function *g: Option[A]=>Option[B]* ?
  - That would mean that, whenever we had a function ***f*** and a variable ***ao*** of type *Option[A]*, we could do something with it which retained the optional aspect.

    ```
    def lift[A,B](f: A => B): Option[A] => Option[B] = ???
    ```

  - What can we put on the right-hand-side that could possibly make sense? Remember our mantra: ***simple, obvious, elegant.***

* and other container types

A better way of dealing with containers (1a)

- So, our lift method should look something like this:

```
def lift[A,B](f: A => B): Option[A] => Option[B] = _ match {
  case Some(a) => Some(f(a))
  case None => None
}
```

  - Does that "_" bother you at all? It shouldn't. It just represents the input to the resulting function.
  - But does that code look familiar at all?

```
sealed abstract class Option[+A] extends IterableOnce[A] with Product with Serializable {…}
final case class Some[+A](a: A) extends Option[A] { …
final def map[B](f: A => B): Option[B] = this match {
  case Some(a) => Some(f(a))
  case None => None
}}
```

A better way of dealing with containers (1b)

- So, given that the logic is identical to the map method lift method should look something like this:

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

  - Huh? Surely it can’t be that simple?? **It is that simple!!**

A better way of dealing with containers (1c)

- What about lifting a function to a function on ***List***, ***Try***, or ***Seq***?

```
def lift[A,B](f: A => B): List[A] => List[B] = _ map f
def lift[A,B](f: A => B): Try[A] => Try[B] = _ map f
def lift[A,B](f: A => B): Seq[A] => Seq[B] = _ map f
```

- Whoa! Is it really that simple?
    - Yes!

Using *lift*

```
object TemperatureConverter extends App {
    def fToC(x: Double): Double = (x - 32) * 5 / 9
    def lift[A, B](f: A => B): Option[A]=>Option[B] = _ map f
    val fToCOption: Option[Double] => Option[Double] = lift(fToC)
    def fToC(x: String): String =
        fToCOption(x.toDoubleOption) map (c => c.toString+"C")
            getOrElse "invalid input”
    val scanner = new java.util.Scanner(System.in)
    System.err.print("Temperature in Fahrenheit? ")
    val f = scanner.nextLine()
    println(fToC(f))}
```

A better way…(1d)

- We can apply *lift* to any *Function1*.
- Incidentally, we could also write ***lift*** as follows:

```
def lift[A,B](f: A => B): List[A] => List[B] = a => a map f
```

  - We will have to use this less elegant form in the following functions…
- Could we also apply our *lift* mechanism to a *Function2*? Yes, we can but, to do it elegantly, requires knowledge of another higher-level function called *tupled**:

```
def lift2[A,B,C](f:(A, B)=>C):List[(A,B)]=>List[C] = _ map f.tupled
```

- Later, we'll create a similar method we're going to call *map2.*

* that's because *f* has type (A, B) => C whereas we need an ((A, B)) => C

Option and Try—in greater depth

- For example, let's look at *Rating* from the *Movie* assignment.

```scala
case class Rating(code: String, age: Option[Int]) {
    override def toString = code + (age match {
        case Some(x) => "-" + x
        case _ => ""
    })
}

object Rating {
    val rRating = """^(\w+)(-(\d\d))?$""".r

    def parse(s: String): Try[Rating] =
        s match {
            case rRating(code, _, age) =>
                Success(apply(code, Try(age.toInt).toOption))
            case _ =>
                Failure(new Exception(s"parse error in Rating: $s"))
        }
}
```

Option and Try (2)

What are these names all about?

- So, we have a method called ***parse*** which will take a *String* and yield a *Try[Rating]*.
  - Now, we want to add that rating, along with other element(s) to something called *Reviews:* (simplified)

```
case class Reviews(imdbScore: Double, contentRating: Rating)
val xy = Try("97.5".toDouble)
val ry = Rating.parse("PG-13")
Reviews(xy,ry)
```

  - Oops! we don't have a *Double* and a *Rating*. We have a *Try[Double]* and a *Try[Rating]* instead.
    - So, why not write?

```
val r = Reviews(xy.get,ry.get)
```

Bad idea! Remember, we never want to invoke ***get*** on these containers

    - In any case, if we do that, we essentially lose all the advantage of *Try*. We just simply throw exceptions now if there were failures.

Sidebar: naming identifiers

- Isn't it better if there's a consistent naming convention for the variables which don't have an obvious identifier to use?
  - See http://scalaprof.blogspot.com/2015/12/naming-of-identifiers.html
  - Very briefly, the scheme is that we go in reverse order of the types in the type of the variable.
  - So, a sequence of ***X***, such as ***Seq[X]*** (or ***List[X]***, etc.) would be called ***xs***. This much is totally standard in Scala. The rest is non-standard: my own scheme:
    - So, *xy* represents a *Try[X]* (we use "t" for a *Tuple*);
    - *xo*: Option[X]
    - *kvm* (or *kVm* or *k_vm* or even *`k,vm`*) is used for a *Map[K,V]* (here, the type parameters of *Map* are not reversed since they're at the same level.
    - etc. You get the idea.

Option and Try (2a)

- Wouldn’t it be nice if we had a method that took the parameters we actually have and returned a *Try[Reviews]*?
- Let’s write it…

Option and Try (2b)

- So, let's try to write the method we need (it's simple stuff)…

```
def makeTryReview(xy: Try[Double], ry: Try[Rating]): Try[Reviews] =
  xy match {
      case Success(x) =>
          ry match {
              case Success(r) => Success(Reviews(x,r))
              case Failure(e) => Failure(e)
          }
      case Failure(e) => Failure(e)
  }

val vy = makeTryReview(xy,ry)
```

These Failure(e) cases could just yield *ry*, right? Well, no, because they are the wrong type.

- That's just what we need! Great…
- Wait a moment! Do we have to write something like this method every time we want to create a *Try[Z]* from a *Try[X]* and a *Try[Y]*??? Aaaaaargh!
- Of course not! Help is on the way.

A better way…(2c)

- Similarly, it would be very convenient if we had a way of combining, say, two ***Option*** values into one single ***Option*** value, given a function that can combine the two underlying values. What we need is something like this:

```
def map2[A,B,C](ao: Option[A], bo: Option[B])(f: (A,B)=>C): Option[C] =
    ao match {
      case Some(a) => bo match {
        case Some(b) => Some(f(a,b))
        case _ => None
      }
      case _ => None
    }
```

  - OK, this is nice and general. But for ***Reviews***, we need ***map2*** that works with ***Try*** instead of ***Option***.

A better way…(2d)

- Here, we do the exact same thing for ***Try***:

```
def map2[A,B,C](ay: Try[A], by: Try[B])(f: (A, B) => C): Try[C] =
    ay match {
      case Success(a) => by match {
        case Success(b) => Success(f(a,b))
        case Failure(e) => Failure(e)
      }
      case Failure(e) => Failure(e)
    }
```

- Now, we can rewrite ***makeTryReview***:

```
    def makeTryReview(xy: Try[Double], ry: Try[Rating]): Try[Reviews] =
map2(xy, ry)(Reviews.apply)
```

Actually, we can drop the ".apply" part
and just write `(Reviews)`

A better way…(2e)

- OK, our *map2* method for *Try* is nice and general.
- But can we do better? What do you think *map* and *flatMap* do on an *Option[A]*?

```
def map[B](f: (A) => B): Option[B] = ???
def flatMap[B](f: (A) => Option[B]): Option[B] = ???
```

A better way…(2f)

- Continuing with our *map2* on *Option*…

```
def map2[A,B,C](ao: Option[A], bo: Option[B])(f: (A, B) => C): Option[C] =
    ao match {
      case Some(a) => bo match {
        case Some(b) => Some(f(a,b))
        case _ => None
      }
      case _ => None
    }
```

- Let's write out *map* and *flatMap* as object (non-instance) methods (and with a minor rename in the *map* signature):

```
def map[B,C](bo: Option[B])(f: (B) => C): Option[C] =
  bo match {
    case Some(b) => Some(f(b))
    case _ => None
  }
def flatMap[A,B](ao: Option[A])(f: (A) => Option[B]): Option[B] =
  ao match {
    case Some(a) => f(a)
    case _ => None
  }
```

- Are these looking a little bit similar to *map2*? Kind of…

A better way… (2g)

- Suppose we substitute for *flatMap* in the previous slide…
- And where we see *f(a)* we substitute *map* applied to *bo*.
- We'll call this new method "***map2a***":

```
    def map2a[A,B,C](ao: Option[A], bo: Option[B])(f: (A, B) => C): Option[C] =
    ao flatMap (a => bo map (b => f(a, b)))
```

A better way… (2h)

- Now, let's evaluate *map2a* using our own object-methods and then substituting…

```
def map2a[A,B,C](ao: Option[A], bo: Option[B])(f: (A, B) => C):
Option[C] = ao flatMap (a => bo map (b => f(a, b)))
```

```
ao match {
    case Some(a) => bo map (b => f(a, b))
    case _ => None
  }
```

```
ao match {
    case Some(a) => bo match {
     case Some(b) => Some(f(a,b))
     case _ => None
   }
    case _ => None
  }
```

Look familiar???

A better way… (2i)

- Thus we have shown that our *map2* function can be re-written more simply as…

```
def map2[A,B,C](ao: Option[A], bo: Option[B])(f: (A, B) =>C):Option[C] =
    ao flatMap (a => bo map (b => f(a, b)))
```

Whoa!! That's neat.

And what about…

```
def map2[A,B,C](ay: Try[A], by: Try[B])(f: (A, B) => C): Option[C] =
    ay flatMap (a => by map (b => f(a, b)))
```

```
def map2[A,B,C](as: List[A], bs: List[B])(f: (A, B) => C): Option[C] =
    as flatMap (a => bs map (b => f(a, b)))
```

A better way… (2j)

- And an even better way:

```
def map2[A,B,C](ao:Option[A],bo:Option[B])(f:(A,B)=>C): Option[C] =
  for (a <- ao
       b <- bo
  ) yield f(a,b)
```

  - This is called a “for-comprehension” and works for any container type where the container is a monad! It is syntactic sugar for:

```
ao flatMap (a => bo map (b => f(a, b)))
```

- Going back to our original problem…

```
val vy: Try[Reviews] =
  for (x <- xy
       r <- ry
  ) yield Reviews(x, r)
```

- Phew! That was hard getting there.
  - But so simple in the end. And very important!

Quick summary

- We created a method called *lift* that takes an *A=>B* function and returns a *M[A]=>M[B]* function where *M* is some container type like *Option*, *Try*, *List*, etc.
  - The body of this method is always the same:

    ```
    _ map f
    ```

- Then we created a method called *map2* that takes, an M[A], an M[B], a function (A,B)=>C and returns an M[C], where *M* is a container as above.
  - The body of this method is always the same:

    ```
    _ flatMap (a => _ map (b => f(a, b)))
    ```

  - Which we can re-write very nicely as:

    ```
    for (a: A <- _; b: B <- _) yield f(a,b)
    ```

"for comprehensions (1)"

- There are two forms of "for comprehension" [we already covered this]:
    - Without *yield* (i.e. relying on side-effect):

      for (*seq* **)** *body*

    - With *yield* (returns value—no side effects):

      for (*seq* **) yield** *expr*

“for comprehensions”

- In each case, ***seq*** represents a sequence of ***generators***, ***definitions*** and ***filters***, separated by semi-colon (or newline)
  - A generator is of form:

    `pattern <- container`

    - where pattern is matched against each item generated from the container (most of the time, the pattern is simply an identifier which matches everything)
  - A definition is of form (exactly like a variable declaration, but without “val”):

    `identifier = expr`
  - A filter (“guard”) is of form (just like the guard clause on a match/case pattern):

    `if` *`expr`*

Putting it all together

```scala
object ReadURL {
    import scala.util._
    import scala.io.Source
    import java.net.URL

    def getURLContent(url: String): Try[Iterator[String]] =
      for {
          u <- Try(new URL(url))
          connection <- Try(u.openConnection())
          is <- Try(connection.getInputStream)
          source = Source.fromInputStream(is)
      } yield source.getLines()

    def wget(args: Array[String]): Unit = {
        val maybePages = for {
            arg <- args
            x = getURLContent(arg)
        } yield x
        for {
            Success(p) <- maybePages
            l <- p
        } println(l)
    }

    def main(args: Array[String]): Unit = {
          println(s"web reader: ${args.toList}")
          wget(args)
    }
}
```

Here we are using the real *Try* class in *scala.util*

Instead of *Try[Try[Try[Iterator[String]]]]*, the *Try* classes are collapsed into one—because of the way *flatMap* operates.

From *The Neophyte's Guide to Scala*—this can be improved: we don't close the source for instance.

Note that we can even create the equivalent of a "val" inside a for-comprehension. We can also do filtering, for instance.

This for-comprehension has no **yield** therefore relies on side-effect

Here's an example of a pattern match

For now, we throw away any error messages.

I ran this with arguments:

- http://htmldog.com/examples/lists0.html
- http://htmldog.com/examples/lists1.html

Some other handy methods:

- What if you had a *Seq[Option[X]]* and you wanted an *Option[Seq[X]]?*
  - *sequence*:

    ```
    def sequence[X](xos: Seq[Option[X]]): Option[Seq[X]] = ???
    ```

  - this method should iterate through x*os* and, if all elements are *Some(x)*, collect them into a sequence x*s* then return *Some(xs)*. If any of the elements are *None*, return *None*.
  - We're not quite ready to implement this one.
- What if you had a *Seq[X]* and a function *f: X=>Option[Y]* and you wanted an *Option[Seq[Y]]?*
  - *traverse:*

```
def traverse[X,Y](xs: Seq[X])(f: X=>Option[Y]): Option[Seq[Y]] = ???
```

In general, lots of these functional compositions

- You will be working with some of these in an upcoming assignment.