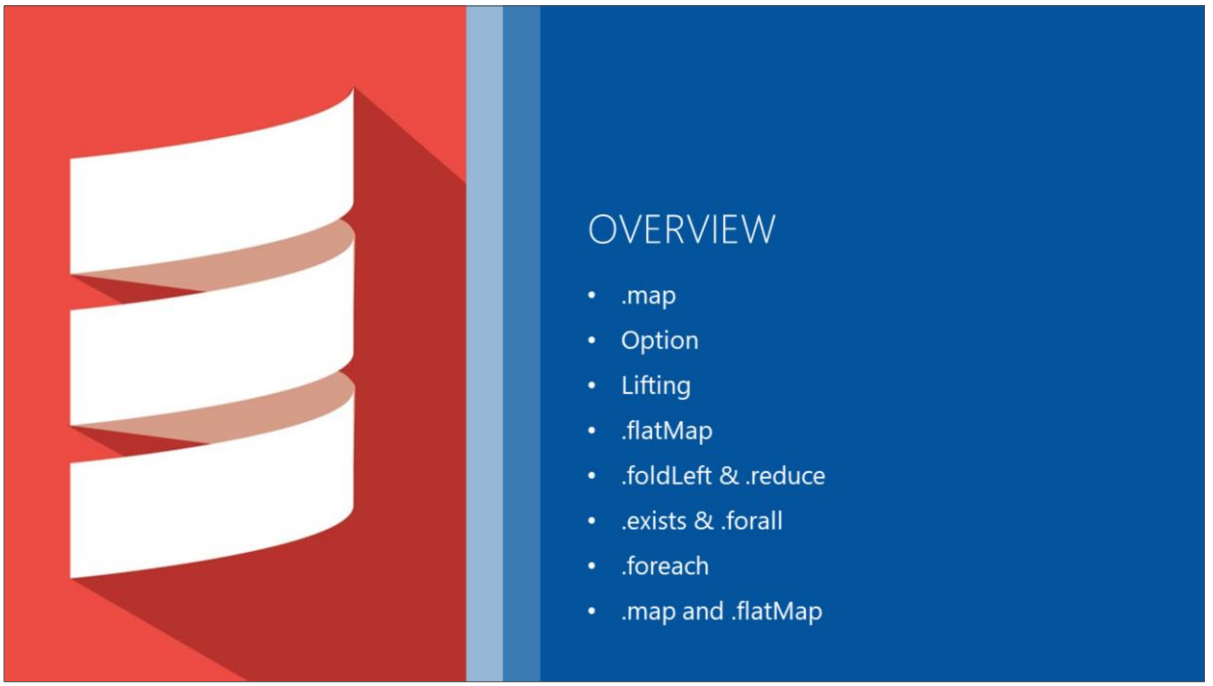
Combinators

SCALA PROGRAMMING
A DATA SCIENCE AND MACHINE LEARNING COURSE


OVERVIEW
.map
Option
Lifting
.flatMap
.foldLeft & .reduce
.exists & .forall
.foreach
.map and .flatMap

.map

- Combinators are HOFs
  - use lambda to restructure
- .map is not a loop
  - only on collections
- C[X] to a C[Y]
  - via f : X => Y
  - Eg. List[String] to List[Int] via _.length

```
val names = List("Michael", "Sherlock",
"Watson")
val prepMs = (str: String) => "Ms. " + str

def prepMr(str: String) = "Mr. " + str

println(names.map(prepMs))
println(names.map(prepMr))

names.map( (e: String) => "Mr. " + e )
names map( (e: String) => "Mr. " + e )
names map( e => "Mr. " + e )
names map( "Mr. " + _ )
names map { "Mr. " + _ }

for( e <- names ) yield "Mr. " + e

val ages = List(18, 19, 20, 21)
ages map { _ > 18 }

// OUTPUT :
List(Ms. Michael, Ms. Sherlock, Ms. Watson)

List(Mr. Michael, Mr. Sherlock, Mr. Watson)

List(false, true, true, true)
```

.map is a combinator

a function (, method) which accepts another function as an argument and uses it to restructure (recombine) the data

.map, in general, can only be understood in terms of types

it takes a C[X] to a C[Y] by applying a f : X => Y

.map is often first introduced on lists where it is a loop

it is not a loop on other data structures, and may be a very different operation

.map applied to a list takes a

List[A] to a List[B] by applying a f: A => B

Ie., you get a new List[B] from a List[A] by running f

on every element

Option

- .map should be understood in terms of types
  - *not loops!*
- e.g., Option[A] = Some(value: A) OR None[A]
  - There is no Option[A] constructor
- X[A] doesn't mean *has* an a value inside
  - v : X[A] is not v: X[B]
  - regardless of what actual data is in-memory
  - None[Int] is not None[String]
  - despite neither holding any data
- Options represent possibly empty data

```
val sname: Option[String] = Some("Michael")
val nname: Option[String] = None

println(sname map { "Mr." + _ })
println(nname map { "Mr." + _ })

def loc(name: String): Option[String] = Map(
"Michael" -> "London",
"Andy" -> "Cardiff"
).get(name)

val names = List("Michael", "Andy")
val locations = (names map loc)

println((names map loc ) map {
  opt => opt map { _.toUpperCase }
})

// OUTPUT :
Some(Mr.Michael)

None
List(Some(London), Some(Cardiff))
List(Some(LONDON), Some(CARDIFF))
```

To understand the type-rule structure of map we need to consider a type which does not have a looping-map

Option[A] is a good example

It can be created with the Some(value: A) constructor

Or None[A] which accepts no value

There is no Option[A] constructor

, it is an abstract parent type

Note that the [A] doesn't mean has an a value inside

It means any value : X[A] should be differentiated form an X[B] regardless of what actual data is in-memory

Here None[Int] differs from a None[String] despite neither holding any data

Options are used to represent possibly empty / invalid data

They will be discussed in detail later, for now here as an example for map

Options

Val x : Option[A]

'A' can be anything
Eg. Option[Int]
Option[String]
etc.

this term has ! this type

How do you make one?

Either

= Some[A](3)

a value of some kind

OR

= None[A]

these are 'constructors' for Option

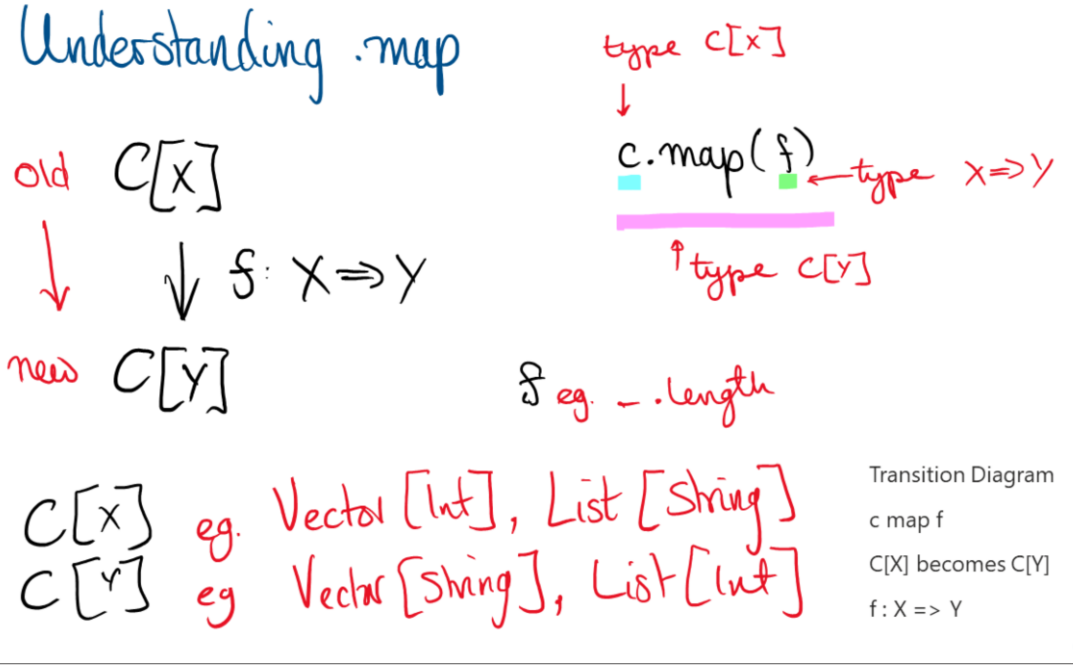
: Type = Construtor()

: Option[String] = Some("Hi")

: Option[String] = None

5

Understanding .map



C[X] eg. Vector[Int], List[String]
C[Y] eg Vector[String], List[Int]

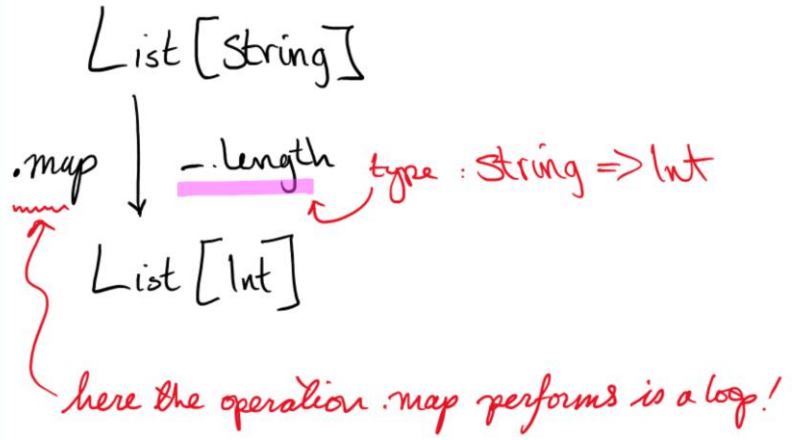
Transition Diagram

c map f

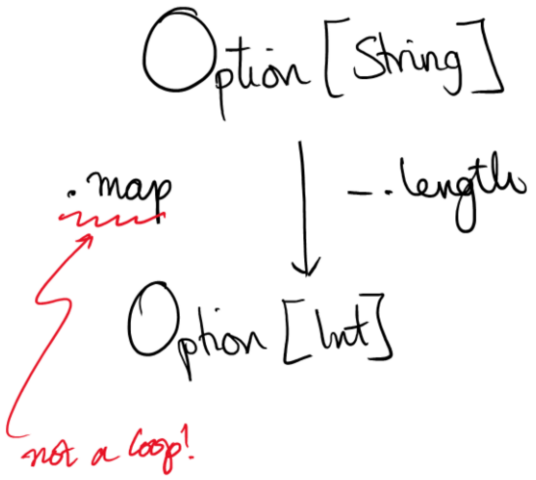
C[X] becomes C[Y]

f : X => Y

Understanding list and .map



Understanding Option and .map



here .map does:

```
if( v == None)
  None
else
  Some( f(v))
```

where
f = _.length
and v is the nested value

.flatMap

- .flatMap
  - runs a function just like map
  - C[A] to a C[B] via f
  - However f isn't A to B
    - f: A => C[B]
- Consider _.split : Array[String]
  - _.map { _.split(" ") }
    - List[List[String]]
  - .flatMap { _.split(" ") }
    - List[String]
    - via concatenation of inner lists

```
val names = List(
  "Fluffy Jefferson",
  "Fido Holmes",
  "Spot Sinatra")

names map { _.split(" ") }

//peeling away the type
names flatMap { _.split(" ") }

names flatMap {
  _.split(" ")
} map { _.toUpperCase }

// OUTPUT :

names: List[String] =
 List(
 Fluffy Jefferson,
 Fido Holmes,
 Spot Sinatra
)

res0: List[String] = List(
  FLUFFY, JEFFERSON,
  FIDO, HOLMES,
  SPOT, SINATRA
)
```

.flatMap is another important combinator

It runs a function just like map

Taking a C[A] to a C[B] with an f

However this time the f isn't A => B

It returns a C[B]

Consider the .split method, it returns a Seq[String]

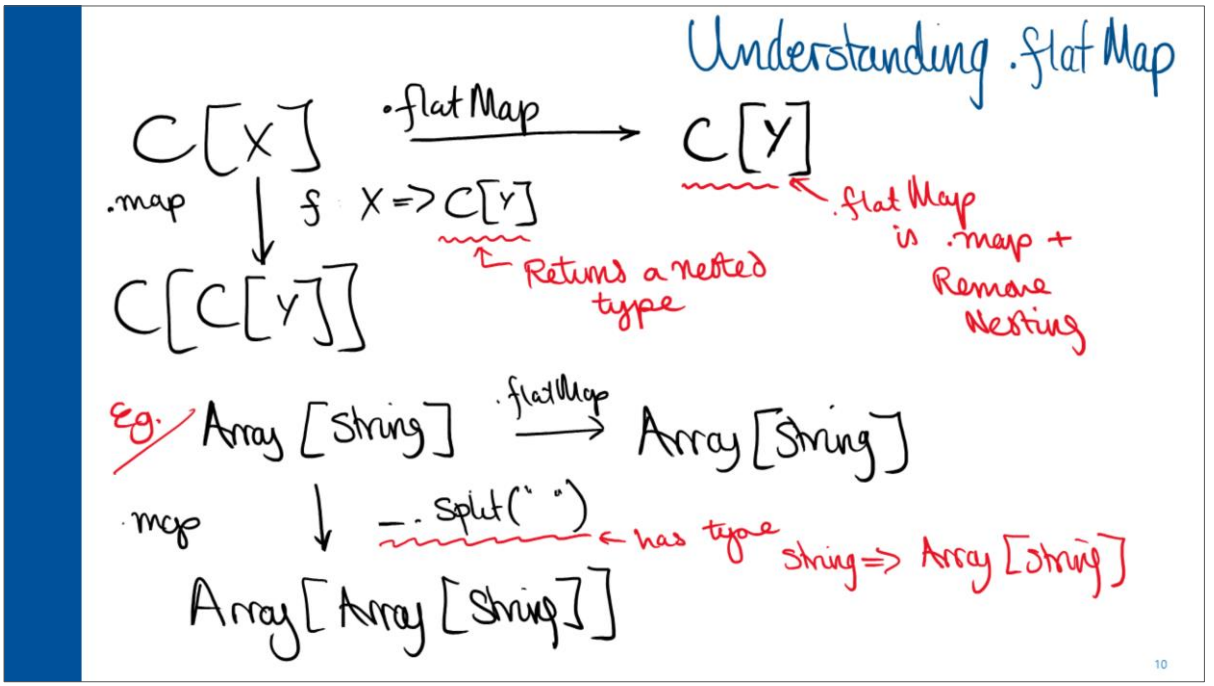
In particular, an Array[String] for the moment we can treat Array = List = Seq

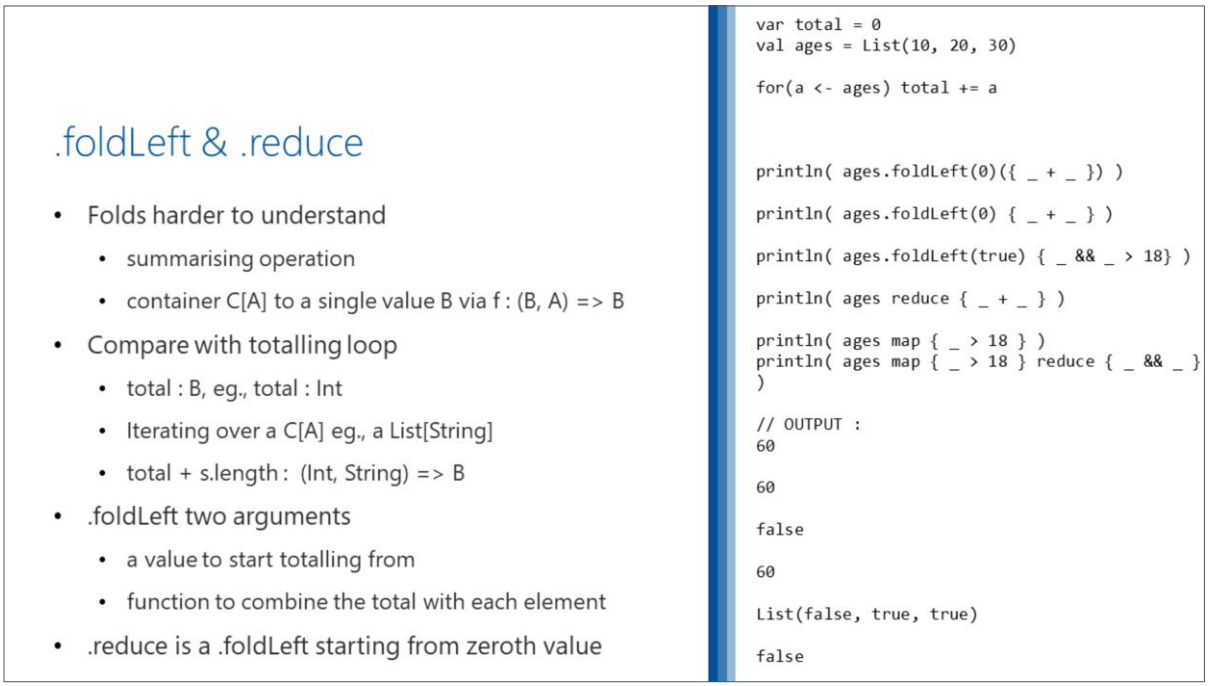
Mapping .split : String => List[String] over strings gives

List[List[String]]

Flat Mapping gives List[String]

In the case of strings this is with a nested concatenating inner loop


Understanding .flatMap
C[X]
.flatMap
C[Y]
.map
f X => C[Y]
C[C[Y]]
Returns a nested type
.flatMap is .map + Remove Nesting
eg.
Array [String]
.flatMap
Array [String]
.map
_.split(" ")
has type
String => Array [String]
Array[Array [String]]
10



Folds are a little harder to understand

Roughly they can be seen as a summarising operation

Which takes a container C[A] to a value B by combing the data of C[A] into a B with an operation f : (B, A) => B

An imperative version of a fold is totalling loop

Where total : B, eg., total : Int

Iterating over a C[A] eg., a List[String]

By total = total + s.length : (Int, String) => Int

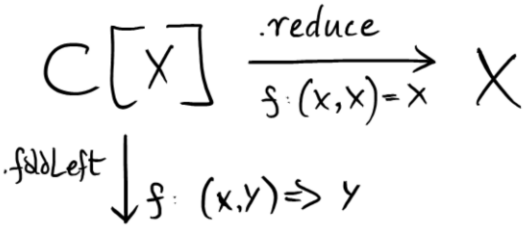
Ie., (B, A) => B

.foldLeft accepts two arguments a value to start totalling from and a function to combine the total with each element

Reduce just takes the function and assumes the first element is the starting value

Understanding Folds

A fold 'simplifies' a nested data structure, aggregating it.



a fold "reduces" a container-ish type C[X] down to a unnested "contents" type Y

type (Int, Int) => Int

{ _ + _ }

eg.

List[Int] → Int

List(5, 3) →

.exists & .forall

- test contents of container
  - run a function which returns true or false
  - .exists at least one element passes
  - .forall all elements pass the test
- Just a convenient fold
  - true for forall, false for exists
- Boolean-returning functions
  - called predicates
  - describe data

```
// PREDICATES
val ages = List(10, 20, 30)
val isAdult = (age: Int) => age > 18

println(ages forall { _ > 18 })
println(ages exists { _ < 18 })

println(ages forall isAdult)
println(ages exists isAdult)

// OUTPUT :
false
true
false
true
```

.exists and .forall provide information about the contents of a data type

They run a function across the contents which returns true or false

.exists tells you if there is at least one true element

.forall tells you if all elements pass the test

These two combinators are just a convenient fold

They fold starting with true for forall, false for exists

Applying the function to the total and each element

Boolean-returning functions are called predicates

Because they are descriptions of data

For Comprehensions

- arrow unwraps, e.g. a <- src
  - a : A and src : C[A]
- body may contain
  - A yield – the whole expression is then C[B]
  - No yield – the whole operation is Unit
- Subsequent extractions depend on the former
  - for(a <- as; b <- f(a)) yield b
  - takes f : A => C[B] to make a C[B]
- Comprehension desugars
  - yielding .map and .flatMap
  - no-yield .foreach

```
for(person <- List("Matt", "Mark", "Luke"))
   println(person)

for(pair <-
   Map("Cat" -> "Mammal", "Raven" -> "Bird"))
    println(pair)

for((animal, kind) <-
  Map("Dog" -> "Mammal", "Crow" -> "Bird"))
    println(s"$animal is a $kind")

val simple = for((animal, kind) <-
  Map("Dog" -> "Mammal", "Crow" -> "Bird")
) yield kind + ":" + animal

// OUTPUT :
Matt
Mark
Luke
(Raven,Bird)
(Cat,Mammal)
Dog is a Mammal
Crow is a Bird
List(Mammal:Dog, Bird:Crow)
```

Recall for-comprehensions

a <- src – is an unwrapping

a : A and src : C[A]

The body can contain

A yield – the whole expression is then C[B]

Where the yield operation A => B

No yield – the whole operation is Unit

Subsequent extractions depend on the former

for(a <- as; b <- f(a)) yield b

has the form of taking an f : A => C[B]

And making a C[B]

The for comprehension then desugars to .map .flatMap and .foreach depending on the syntax of its use

.foreach

- .foreach is a higher order method
  - takes a unitary function as an argument
  - runs the function on the object's data structure
    - throwing away the return value
  - So that .foreach itself returns : Unit
- For comprehensions
  - with no yield
  - desugar into .foreach

```
for(pet <- List("Fluffy", "Spot"))
println(pet)

List("Fluffy", "Spot") foreach {
  pet => println(pet)
}

List()

// yield and map
for(id <- List(1, 2, 3)) yield id + 1000

List(1, 2, 3) map { id => id + 1000 }

// OUTPUT :

Fluffy
Spot

Fluffy
Spot

List(1001,
1002, 1003)

List(1001,
1002, 1003)
```

.foreach is a higher order method which takes a unitary function as an argument

It runs this function on the object's data structure throwing away the return value of each

So that .foreach itself returns : Unit

For comprehensions with no yield are therefore desugared into .foreach

.map and .flatMap

- Roughly
  - yield desugars to .map
  - additional extractions are .flatMap
- Single Extractions
  - best written a map or a flatMap
- Multiple extractions
  - for comprehensions become much clearer

```
val ids = List(1, 2, 3)
val pets = List("Fluffy", "Spot")

for {
id <- ids
pet <- pets
} yield s"${id + 1000}: $pet"

val petids = ids.flatMap { id =>
  pets.map { pet => s"${id + 1000}: $pet" }
}

println(petids)

// OUTPUT :

List(
1001: Fluffy,
1001: Spot, 1002: Fluffy, 1002: Spot, 1003:
Fluffy, 1003: Spot)
```

Roughly, the yield can be considered to desugar into a .map and any additional extractions are a .flatMap

Simple for comprehensions are often best written as just a map or a flatMap

However with multiple extractions for comprehensions become much clearer

.map/.flatMap parings are rarer than for comprehension use and best avoided

Aside: Lifting

- standard: (f: O => N) => xs map f
  - List(1, 2, 3) map len
  - run function f over given List
- lifting: (xs: List[O]) => xs map f
  - lift f into _
  - convert given function to new type
  - "upgrading" its argument and return type
- Given
  - f: Int => String
- Then
  - listyF: List[Int] => List[String] = _ map f

```
//we know how to convert values:
println(1.toDouble)

//but how do you convert functions?
val addOne =
  (i: Int) => i + 1
// : Int => Int

//can we make it
//  Option[Int] => Option[Int]
val _optAdd1 = (oi: Option[Int]) => ???

val optAdd1: Option[Int] => Option[Int] =
  _ map addOne

println(optAdd1(Some(5)))
//should be: Some(6)

println(optAdd1(None))
//should be: None

//"converting a function"
// is called lifting
//map is just lift!
// (for one-arg functions)

// OUTPUT :
1.0
Some(6)
None
```

Lifting is the process of converting a function

"upgrading" its argument and return type

By wrapping them in the same type

Eg. Lifting Int => String into List makes

List[Int] => List[String]

Two interpretations of the role of .map depending whether the container varies or the function

_ map f

lift f into _ -- convert f to work with type of _

List(...) map _

run function _ within List

Exercise