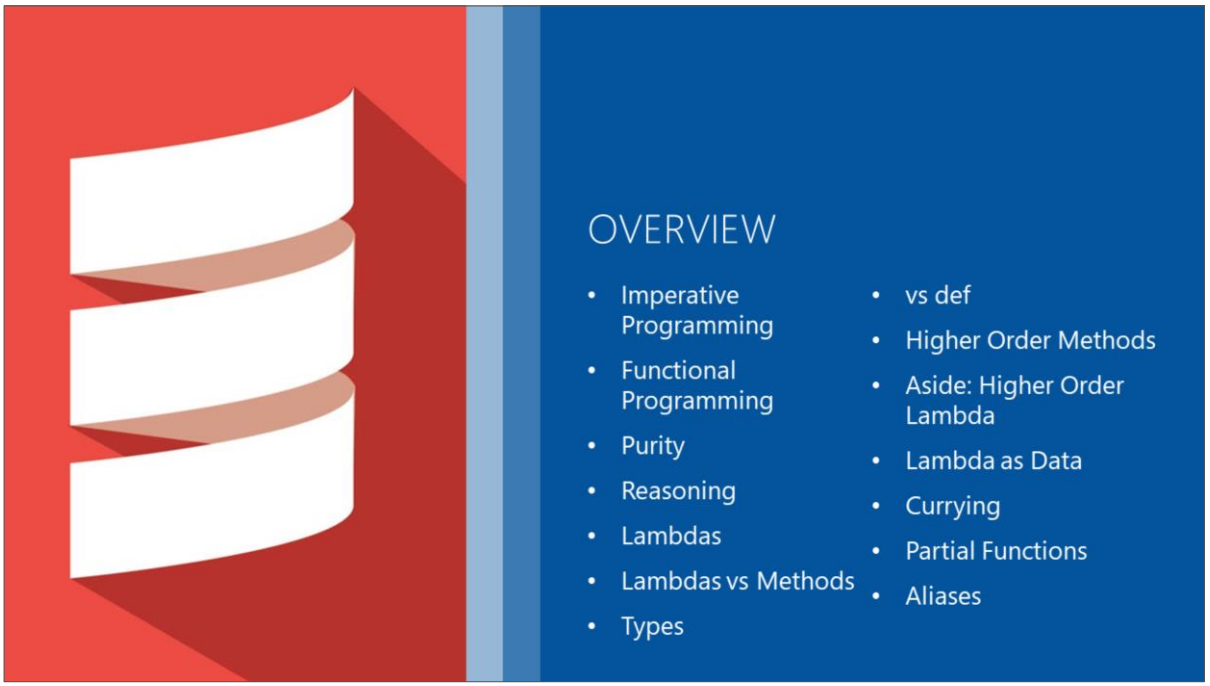
Functions

SCALA PROGRAMMING
A DATA SCIENCE AND MACHINE LEARNING COURSE


OVERVIEW
Imperative Programming
Functional Programming
Purity
Reasoning
Lambdas
Lambdas vs Methods
Types
vs def
Higher Order Methods
Aside: Higher Order Lambda
Lambda as Data
Currying
Partial Functions
Aliases

Imperative Programming

- Revise device state
  - build by mutation
  - destroying originals
- Every line
  - serial order
  - ordering must be followed
  - no way to determine reordering

```
// imperative program: actions on state

class Ingredients(var amount: Double) {
  def prep() = amount *= 2
  def mix() = amount += 2
  def bake() = amount /= 2.0
}

val ing = new Ingredients(100)
ing.prep()
ing.mix()
ing.bake()

println(ing.amount) // why..?!

// OUTPUT :

101.0
```

Imperative programs revised pre-existing state

revisions build up values over time

destroying original values

Each line in the program may revise a value

the order of revisions is important

each line may modify anything established in any other

There is no way of knowing without running the program what modifications take place

Functional Programming

- Pure
  - no state required
  - state never revised
- Single calculation
  - f(g(h (input)))
- Syntax to facilitate
  - abstraction with functions
  - abstraction over functions
  - abstraction with types
  - abstraction over types

```
// functional programming:
//all data flow is explicit

def prep(amount: Double) = amount * 2
def mix(amount: Double) = amount + 2
def bake(amount: Double) = amount / 2.0

val ing = 100.0
val prepd = prep(ing)
val mixd = mix(prepd)
val baked = bake(mixd)

println(baked)

//the entire application is one expression
println( bake(mix(prep(100.0))) )


// OUTPUT :
101.0
```

Pure functional programs do not revise state

every new value is created from an old value

without any state changes

Every pure functional program is a single expression

f(g(h (input)))

It is difficult to design programs with this syntax alone

to define reusable functions

to define types correctly that allow one return to be passed to one argument

Functional languages provide syntax to ease reuse and design

Purity

- Pure functions are maps
  - Out = In
  - In => Out substitution
- Impure functions
  - no substitutions
  - e.g., device access
- Compare
  - println(3)
    - screen modification
  - "abc".length
    - temporary value store

```
/* EG: */
// a pure function is a mapping
// from inputs to outputs

def prep(amount: Double): Double = amount * 2

// “Double to Double” means:
// any single Double to a single Double

// equivalent to a(n infinite) set of pairs:

val prep = Map(
         0.0 ->  0.0,
         1.0 ->  2.0,
         1.1 ->  2.2,
         //... for every double
)
```

A function is pure if it only maps input arguments to an output value

the function call can then be substituted for this value

Compare println() with "abc".length

there is no value a call to println() can be substituted for without changing the program

"abc".length can be everywhere substituted for 3

without changing the program

therefore .length is pure

Reasoning

- What will happen next?
- Why?
  - What state?
  - What connections?
- Purity
  - equational reasoning

```
def prep(amount: Double) = amount * 2.0
def mix(amount: Double) = amount + 2.0
def bake(amount: Double) = amount / 2.0

// like is substitutable for like
println(bake(mix(200.0)))
println(bake(202))
println(101.5)


//easy to reason about
//these are totally independent
//and so may be run in parallel
val serialA = bake(mix(prep(100)))
val serialB = bake(mix(prep(100)))


// OUTPUT:
101.0
101.0
101.5
```

The difficult task in programming is reasoning about the behaviour of a program

what will happen next?

why?

what parts are involved?

how are those parts connected?

etc.

Pure functional programs make answering these questions much easier

pure functions are identical to their return values

pure functions which are not called by others have no effect on them

Lambdas

- Functions are Values
  - Lambda are anonymous
- Values are Objects
  - Lambda are objects
- Lambdas
  - defined
    - (args) => return
  - typed
    - f : ArgumentT => ReturnT
- L1 : L2 =>L3 = R
- L1 = R1 => R2

```
// behaviour can be wrapped up
// into objects:

val square     = (x: Int) => x * x

val printHello = () => println("Hello")

val add = (x: Int, y: Int) => x + y
val sub = (x: Int, y: Int) => x - y


printHello()

println(square(add(10, 10)))


// OUTPUT :

Hello
400
```

Lambda functions are values

functions that may be assigned to variables

these are sometimes called anonymous

as they do not have their own name

All values in scala are objects

therefore lambdas are objects

Lambdas are defined with =>

the argument list precedes

the return value follows

Lambdas vs Methods

- Methods
  - call on name
  - compile time
- Lambda
  - object
  - call on .apply()
  - () rewrite rule

```
/* ERROR: */

val first = (name: String) => name.split("
")(0)
def firstName(name: String) = name.split("
")(0)

// called the same way:
println(firstName("Fido Holmes"))
println(first("Fluffy Jefferson"))

//the second call is rewritten to:
println(first.apply("Fido Holmes"))

// reads a variable containing a value
first

// a command to call a method
println(firstName)              // ERROR!

// first.apply is a metod of first
// methods are not objects

// OUTPUT:
Fido
Fluffy
Fido
```

Compare lambda with methods

methods are blocks of code

whenever a method name is used the compiler calls

lambda are objects

whenever a lambda name is used nothing is called

Lambda are only called when followed by ()

an object followed by () is always rewritten to .apply()

"calling an object" is therefore just calling o.apply()

Types

- Lambda
  - => constructor syntax
- Type
  - => 'to'

```
// method syntax
def m(x: String): Int = x.length

// function syntax
val f: String => Int =
  (x: String) => x.length

// note type of functions:
// String => Int
// for every string,
// a single integer is defined

println(
  m("Hello")
)

println(
  f("Hello")
)

// OUTPUT :

5
5
```

Lambda have their own type syntax

the => occurs in both the type definition

and the definition of the lambda

In a type => means 'to'

for every set of arguments there is one value produced

The => in the type is always after a :

a type differentiates kinds of objects

lambda are differentiated by argument and return types

⇒ vs def

- Method
  - conversation to lambda
  - automatic on assignment
  - _
  - new + .apply

```
// scala prefers the name 'function'
// to mean a lambda
// run-time object which wraps behaviour

// scala prefers the name 'method' to mean a
// compile-time label for behaviour

val function = (name: String) =>
name.toUpperCase

def method(name: String) = name.toUpperCase

println(function("victoria"))
println(method("victoria"))

// wrap up method into a function:
val fnFromMethod = method _

// OUTPUT :
VICTORIA
VICTORIA
VICTORIA
```

A method may be converted to lambda

follow the method name with _

or assign the method to a variable with a lambda type

Scala wraps the method up in an object

ie., roughly, cut/pastes the code into a .apply()

Higher Order Methods

- Passing processes
  - inversion of control
- Standard syntax
  - () becomes {}
  - (f) => becomes _

```
def sendMessage(formatter: String => String) =
  println(formatter("How are you?"))

sendMessage( (m: String) => m.toUpperCase )

sendMessage( m => m.toUpperCase )

sendMessage( _.toUpperCase )

// standard form
sendMessage { _.toUpperCase }

// OUTPUT:

HOW ARE YOU?
HOW ARE YOU?
HOW ARE YOU?
HOW ARE YOU?
HOW ARE YOU?
```

One of the major uses of lambda is to pass them to other processes

a method may be defined to require some other process to complete its job – a Higher Order Method

that is, a parameter may have a lambda type

This is an inversion of control

the method allows the called to decide how it will work

Standard form for calling methods with lambda:

a single underscore generates a lambda of one argument where the underscore is the argument

parentheses around a single argument replaced with braces

Aside: Compare with Javascript

- Some developers may be more familiar with javascript's notation for creating functions

```
// JAVASCRIPT:

function send(m, f) {
 console.log( "<html>" + f(m) + "</html>" );
}


send("embrace",
 function (s){ return s.toLowerCase() + "?";}
);


// OUTPUT:

<html>embrace?</html>


// SCALA:

def send(m: String, f: String => String) =
  println("<html>" +  f(m) + "</html>")


send("Embrace Life",
  (s: String) => s.toLowerCase + "?"
)

// OUTPUT:
<html>embrace life?</html>
```

Lambda as Data

- 3 + 4 = 7
- f + g = f compose g

```
val first = (n: String) => n.split(' ')(0)
val upper = (n: String) => n.toUpperCase

// data can be combined
// so can functions:

val total = 5 + 5
val fnFirstUpper = first andThen upper
val fnUpperFirst = first compose upper  //
f(u())

println(fnFirstUpper("John Watson"))

// why? for example, to use with HOFs ..

def sendMessage(formatter: String => String) =
 println(formatter("Hello -- How are you?"))


sendMessage(first compose upper)

// OUTPUT :

JOHN
HELLO
```

Data has an arithmetic:

methods and operations that can act on it

Functions are data

they have operation which can be performed on them

The most important of these is composition

f compose g makes a new function which calls g then f

g andThen f makes the same function

some prefer andThen as the reading order follows calling

Aside: Higher Order Lambda

- composition is higher-order
  - z = c(f, g)
  - transforms f, g into z
- one of many transforms

```
val add = (x: Int, y: Int) => x + y
val sub = (x: Int, y: Int) => x - y

def reop(op: (Int, Int) => Int, x: Int, y:
Int) =
  op(op(x, y), op(y, x))

println( reop(add, 1, 2) )
println( reop(sub, 2, 4) )

// OUTPUT :

6
-4
```

Lambda may accept other lambda as arguments

a function may return another

this can be used to transform functions

eg., accept a function f and return a function g

where g has the arguments of f in reverse order

Currying

- Reshaping
  - too many arguments
  - reordering arguments
- Method currying
  - mostly for syntax

```
def price(p: Double)(disc: Double => Double) =
  println(disc(p * 2))

price(10)( p => p * 0.3 )
price(10) { _ * 0.3 }

//currying with partially applying
def configure(h: String)(u: String)(p: String)
=
  s"Configuring ${h} (u: ${u} p: ${p})"

val configureUK = configure("uk-host") _
val configureFrance = configure("french-host")
_

println(configureUK("user")("pa$$"))
println(configureFrance("user")("pa$$"))

// OUTPUT :

6.0
6.0
Configuring uk-host (u: user p: pa$$)
Configuring french-host (u: user p: pa$$)
```

Sometimes functions have too many arguments

when passing a function to a higher order method

it may require a function of fewer arguments than available

Functions may have their argument lists broken apart

currying a function separates its argument lists

so that each argument can be filled in separately

reshaping the function so that it fits into the required slot

Methods may be defined in a curried style

Aside: Partial Functions

- map { case 0 => 1 }
  - equivalent to map { _ match case 0 => 1 }

```
/* ERROR: */

List(1, 2, 3) map { _ match { case _ => 'a' }
}
List(1, 2, 3) map { case _ => 'a' }

List(
  "Bahn Mi, UK", "Soup, UK", "Steak, US"
) map { _ split ", " } map {
 case Array(f, "UK")    => "London"
 case Array("Steak", _) => "Texas"
}

List(1, 2, 3) map { case 1 => 'a' } //error

// OUTPUT :

scala.MatchError: 2 (of class
java.lang.Integer) …

res0: List[Char] = List(a, a, a)

res1: List[Char] = List(a, a, a)

res2: List[String] = List(UK, UK, US)
```

map { case 0 => 1 }

equivalent to map { _ match case 0 => 1 }

Aside: Aliases

- f : Function
  - type Function = From => To

```
//functions can be type-aliased:

type PairToInt = (Int, Int) => Int

def reop(op: PairToInt, x: Int, y: Int): Int =
  op(op(x, y), op(x, y))

println(  reop( { _ + _ }, 5, 5)  )

// OUTPUT :

20
```

Exercise