

Jan 9, 2024

It is a general-purpose programming language

It is not designed for a specific platform

There are a few arbitrary limitations ex: tuple with more than 22 elements (use a list lol) - (unlike java)

It is very easy to extend

It is extremely suitable for concurrent & parallel processing

Scala runs on any jvm or browser

Scala can be compiled into 3 ways:

Scala to bytecode - can run any machine that use bytecode (portable way) .scala

Scala to JS - can run on any browser (why was this invented ?? - something like spark shell)

Scala to Machine code - using Native Java compilation - runs on only particular systems (newest way)

REPL - cmd line

REPL Notebook - Zeppelin

Scala Worksheet - .sc

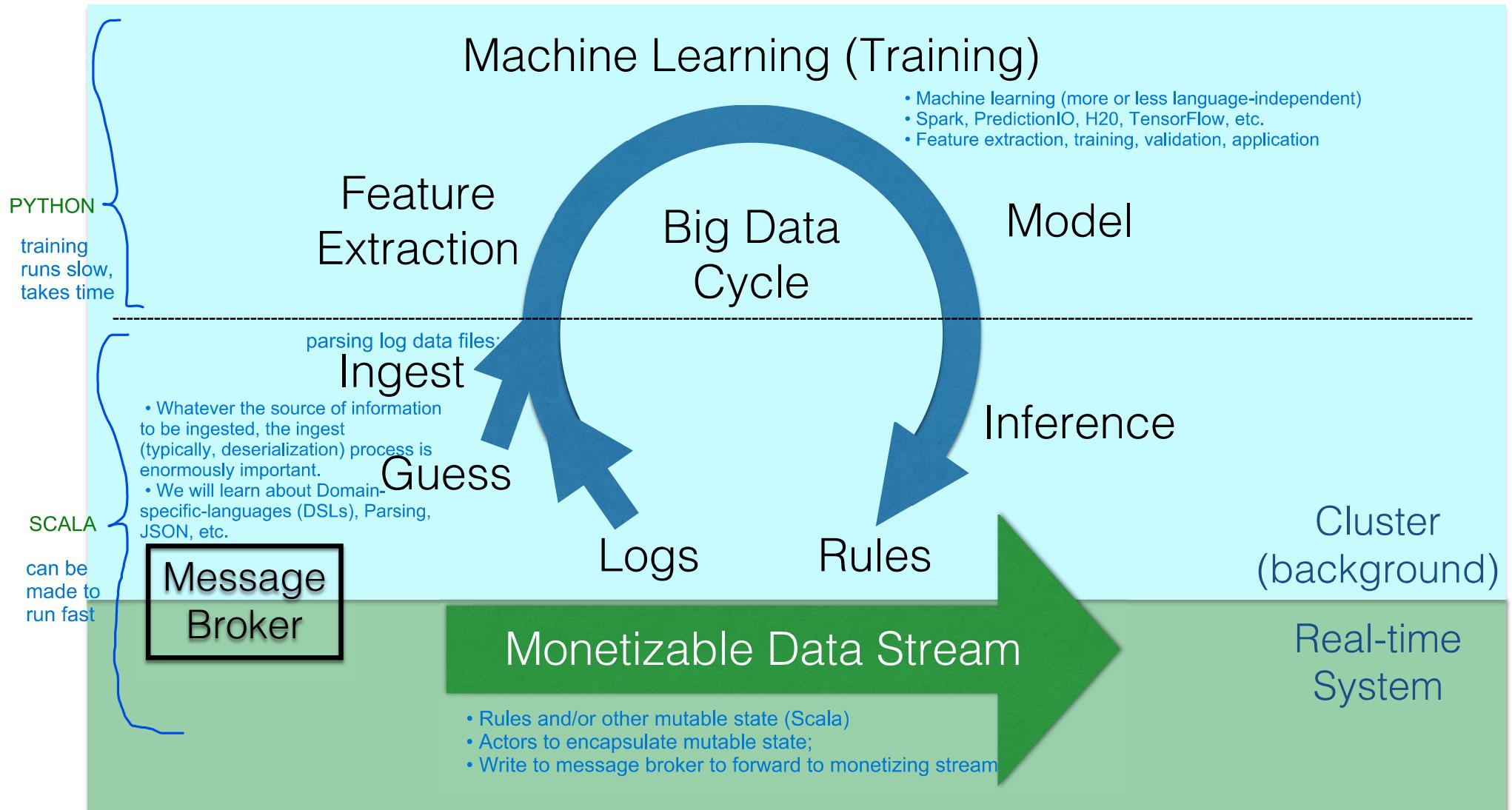
0.1 What is Scala?

A quick look at some real code

object MergeSort will be represented as \$MergeSort
\$MergeSort is a companion

- Massive amounts of data: But, given the nature of this data, it is usually possible to parallelize the processing of it because, typically, chunks of data are independent.
- Many concurrent inputs to a program: Databases, users, web services, It's really important that we design for multiple, asynchronous threads.
- Scala (Scalable Language) is a language that is designed for these situations.
- And (Apache) Spark is a high performance computing platform which is built with Scala to, as much as possible, insulate programmers from the intricacies of parallelization and concurrency.
- Whether you're someone who thinks that Java and Python will eventually catch up to Scala, it doesn't matter. You need to understand how functional programming works because it is THE WAY OF THE FUTURE

Big Data information cycle



Everything is an expression except...

```
new *
object FizzBuzz extends App {
    new *
    def fizzBuzz(x: Int): String =
        (x % 3 == 0, x % 5 == 0) match {
            case (true, true) => "FizzBuzz"
            case (true, _) => "Fizz"
            case (_, true) => "Buzz"
            case _ => x.toString
        }
    private val strings = for (x <- 1 ≤ to ≤ 100) yield fizzBuzz(x)
    println(strings mkString("", "\n", ""))
}
```

- ... the last statement.
- Although there is no explicit “*main*” method here, extending *App* makes this object into a main program.
- A Java Virtual Machine main program has no way to return anything to the operating system (other than an exit code) so we must persist the final object somehow: here we write it to the console.

- *strings* is a *List[String]* and can be combined into a single string using *mkString*.

Here's a slightly more elegant solution:

```
package edu.neu.coe.csye7200.assthw

object FizzBuzz extends App {
  def fizzBuzz(x: Int): String = {
    lazy val dividesBy3 = Factor(3)
    lazy val dividesBy5 = Factor(5)
    lazy val dividesBy3And5 = Factor(3 * 5)
    x match {
      case dividesBy3And5(_) => "FizzBuzz"
      case dividesBy3(_) => "Fizz"
      case dividesBy5(_) => "Buzz"
      case _ => x.toString
    }
  }
  println((for (x <- 1 to 100) yield fizzBuzz(x)) mkString("", "\n", ""))
}

case class Factor(f: Int) {
  def isMultiple(x: Int): Boolean = x % f == 0
  def unapply(x: Int): Option[Int] =
    if (isMultiple(x)) Some(x / f) else None
}
```

- Here we use a more elegant pattern-matching approach.
 - For this strategy, we need another class: *Factor*, which you can see abides quite happily in the same file (module).
 - The “under-the-hood” part of the pattern-matching is implemented via the *unapply* method.
 - A match is said to happen if the result of *unapply* is *Some(...)*. If the result is *None*, then there is no match.
 - We will learn about *Some* and *None* later.
 - Each case is followed by => *expression*.
 - The first match is the one taken.
 - We need to declare *dividesBy3*, etc. in advance because cases are “stable references”

Here's the code with better error handling

The screenshot shows an IntelliJ IDEA interface with a Scala code editor and a run log window.

Code Editor Content:

```
1 package edu.neu.coe.csye7200.assthw
2
3 import edu.neu.coe.csye7200.assthw.Tries.{tryEquals, tryNotEquals}
4 import scala.util.{Failure, Success, Try}
5 import spray.json._
6
7 object WhatsTheTime extends App {
8
9     import TimeJsonProtocol._
10
11     private val maybeTime: Try[Time] = for {
12         response <- Try(requests.get("https://worldtimeapi.org/api/timezone/America/New_Yor"))
13         _ <- tryEquals(response.statusCode, expected = 200, message = "invalid status")
14         _ <- tryEquals(response.headers("content-type"), List("application/json; charset=utf-8"), message = "bad content type")
15         json <- tryNotEquals(response.text(), expected = "", message = "empty json")
16         time <- Try(json.parseJson.convertTo[Time])
17     } yield time
18
19     // show the result
20     maybeTime foreach println
21
22     // log any failures.
23     maybeTime.recoverWith { case x: Exception => System.err.println(s"Failure: ${x.getLocalizedMessage}"); Failure(x) }
24 }
```

Run Log:

```
WhatsAppTime
Run: WhatsAppTime x
↑ /Users/rhillyard/Library/Java/JavaVirtualMachines/openjdk-19.0.1/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA.app/Contents/MacOS/lib/idea_rt.jar -Dfile.encoding=UTF-8
Failure: Request to https://worldtimeapi.org/api/timezone/America/New_Yor failed with status code 404
{"error":"unknown location America/New_Yor"}
```

Process finished with exit code 0

- Lots to explain here:
- We deliberately misspelled “New_York” so that we would see the error handling in practice.
- The most significant thing is the “for-comprehension” that essentially sequences the various assignments.
- If any stage results in an error (a *Failure*), the following steps will not occur.
- The resulting *maybeTime* will have either a *Time* or an exception.

Fixed

The screenshot shows the IntelliJ IDEA interface. The top part is a code editor with a dark theme, displaying Scala code. The code imports `edu.neu.coe.csye7200.assthw.Tries` and `scala.util.{Failure, Success, Try}` from the `assthw` package. It also imports `spray.json._`. The code defines an object `WhatsTheTime` that extends `App`. Inside, it uses `TimeJsonProtocol._` and defines a private val `maybeTime` as a `Try[Time]`. This value is created by a for comprehension that makes a GET request to https://worldtimeapi.org/api/timezone/America/New_York. It checks the response status code (expected 200), content type (application/json; charset=utf-8), and text (empty). The response is then parsed into a `Time` object. Finally, it prints the time and handles any failures. The bottom part is a terminal window titled "WhatsTheTime" showing the command run and the output: the current date and time (2023-01-03T18:48:19.779640-05:00 EST) and a message indicating the process finished with exit code 0.

```
package edu.neu.coe.csye7200.assthw

import edu.neu.coe.csye7200.assthw.Tries.{tryEquals, tryNotEquals}
import scala.util.{Failure, Success, Try}
import spray.json._

object WhatsTheTime extends App {
    import TimeJsonProtocol._

    private val maybeTime: Try[Time] = for {
        response <- Try(requests.get("https://worldtimeapi.org/api/timezone/America/New_York"))
        _ <- tryEquals(response.statusCode, expected = 200, message = "invalid status")
        _ <- tryEquals(response.headers("content-type"), List("application/json; charset=utf-8"), message = "bad content type")
        json <- tryNotEquals(response.text(), expected = "", message = "empty json")
        time <- Try(json.parseJson.convertTo[Time])
    } yield time
    // show the result
    maybeTime foreach println
    // log any failures.
    maybeTime.recoverWith { case x: Exception => System.err.println(s"Failure: ${x.getLocalizedMessage}"); Failure(x) }
}
```

```
WhatsTheTime
WhatstheTime x
/Users/rhillyard/Library/Java/JavaVirtualMachines/openjdk-19.0.1/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA.app/Contents/lib/idea_rt.jar@63345
2023-01-03T18:48:19.779640-05:00 EST

Process finished with exit code 0
```

- We needed to use some libraries and some extra classes to do all this.
- The libraries are **requests** to send/receive the HTTP response;
- And `spray.json` to convert Json into a *Time* object.
- Notice how we print the time by invoking *foreach* with the *println* method.
 - Nothing is printed if there was a failure
 - To get the failure details, we use *recoverWith*.

Implementing merge safely

```
case _ => @tailrec
  def merge(result: List[X], l: List[X], r: List[X]): List[X] =
    (l, r) match {
      case (Nil, Nil) => result.reverse
      case (Nil, _) => result.reverse ++ r
      case (_, Nil) => result.reverse ++ l
      case (h1 :: t1, h2 :: t2) =>
        if (implicitly[Ordering[X]].compare(h1, h2) <= 0)
          merge(h1 :: result, t1, r)
        else
          merge(h2 :: result, l, t2)
    }

  val (l, r) = xs.splitAt(xs.length / 2)
  val (ls, rs) = (sort(l), sort(r))not tail recursed, will be O(log n) would not really cause stack overflow due to log n
  merge(Nil, ls, rs)tailrecursed
```

Here tail recursion occurs, this is seen with Nil variable
Nil is an empty list

Scala has a pointer to head & tail so operations on the start or
the end of the list is O(1)

- We use a technique called **tail recursion** to prevent the compiler putting stuff on the stack.
- Note that when we call *merge* recursively here, there is no more work to do.
- We do the work when we add the extra *result* parameter.
- More on this later.

- One more thing: because it's most efficient to prepend an element to a list, we end up having to do some calls to *reverse*.

A recursive function is said to be tail recursive if the recursive call is the last thing done by the function.

A few more observations

- What was that code: *implicitly[Ordering[X]].compare(h1, h2)*
- Scala has a feature called “**implicits**” which makes it extremely powerful. Here, we used **the feature to allow X to be generic**—just so long as there was available an implicit value of *Ordering[X]*.
- *Ordering* is a trait that has just one method: *compare*. Scala pre-defines a value of *Ordering* for *Int*, *String*, *Boolean*, etc.
- What if we wanted to sort *Strings*? No problem: the only difference in the code would be in the main program: declaring the *sorter* and setting up a list of *Strings* instead of *Ints*.

Big Data Lab 1

```
// Partial functions

val f = (x: Int) => x * x +1

f(3)

val g = (x: Int) => (w: String) => w(x)

→ g(3)("Hello")
```

The screenshot shows a Scala code editor with a command-line interface below it. The code defines two functions: `f` which takes an integer and returns its square plus one, and `g` which takes an integer and a string, returning the string. The command `→ g(3)("Hello")` is being typed, and the output shows the function `g` being applied to 3 and "Hello". The output also includes variable definitions `val f: Int => Int = <function>`, `val res0: Int = 10`, `val g: Int => (String => Char) = <function>`, and `val res1: Char = l`.

This is a function

Val `f = (x : Int) => x * x + 1`

This is a method

Def `f(x : Int):`

Return `x * x + 1;`

In the output its harder to understand what the function really is unless we look at the code, so its usually easier to implement methods

```
val f = (x: Int) => x * x +1

f(3)

val g = (x: Int) => (w: String) => w(x)

g(7)("Hello")
```

The screenshot shows a Scala code editor with a command-line interface below it. The code defines the same `f` and `g` functions as before. The command `g(7)("Hello")` is being typed, and the output shows an error: `java.lang.StringIndexOutOfBoundsException: Index 7 out of bounds for length 7`. The error stack trace is displayed, showing the call chain from `String.charAt` through `StringOps$apply$extension` and `$anonfun$g2adapted` to the final call at index 7.

To avoid throwing errors, or rather handling errors, we want to implement a certain check

(See next implementation)

```
// Partial functions

val f = (x: Int) => x * x +1

f(3)

val g = new PartialFunction[(Int, String), Char] {

  def isDefinedAt(x: (Int, String)): Boolean =
    (x._1 >= 0 && x._1 < x._2.length)

  def apply(x: (Int, String)): Char = x._2(x._1)
}

if (g.isDefinedAt((7,"Hello")))
  g.apply((7,"Hello"))
else '?'
```

```
// Partial functions

val f = (x: Int) => x * x +1

f(3)

val g: PartialFunction[(Int, String), Char] =
{
  case (x,_) if x < 0 => '?'
  case (x,y) if x >= y.length => '?'
  case (x,y) => y.charAt(x)
}

g(3, "hello")
g(-1, "hello")
> g(7, "hello")
```

We usually don't use isDefinedAt(), in case its handled for us internally. Its easier to write code this way

Unapply lets us unpack the applied, here in this case the tuple

Partially applied functions

```
// Partial functions

val f = (x: Int) => x * x +1

f(3)

val h: Int => String => Char =
  (x: Int) => (y: String) => y.charAt(x)

h(3)("hello")
val h3 = h(3)
h3("hello")
```

```
val f: Int => Int = <function>

val res0: Int = 10

val h: Int => (String => Char) = <function>

val res1: Char = l
val h3: String => Char = <function>
val res2: Char = l

val g: PartialFunction[(Int, String),Char] = <function1>
```

```
val res3: Char = l
val res4: Char = ?
val res5: Char = ?
```

h is a curried function -> it's a representation only

g

h3 is a partially applied function

```
val j: (Int, String) => Char =
  (x: Int, y: String) => y.charAt(x)

val myTuple = (3, "hello")
j(myTuple)
```

```
j(3, "hello")
```

```

val f = (x: Int) => x * x +1
f(3)

val h: Int => String => Char =
  (x: Int) => (y: String) => y.charAt(x)

h(3)("hello")
val h3 = h(3)
h3("hello")

val j: (Int, String) => Char =
  (x: Int, y: String) => y.charAt(x)

val myTuple = (4,"hello")
j.tupled(myTuple)
j(1,"hello")

```

```

val f: Int => Int = <function>
val res0: Int = 10

val h: Int => (String => Char) = <function>

val res1: Char = l
val h3: String => Char = <function>
val res2: Char = l

val j: (Int, String) => Char = <function>
+
val myTuple: (Int, String) = (4,hello)
val res3: Char = o
val res4: Char = e

val g: PartialFunction[(Int, String), Char] = <function1>

```

```

val f = (x: Int) => x * x +1
f(3)

val h: Int => String => Char =
  (x: Int) => (y: String) => y.charAt(x)

h(3)("hello")
val h3 = h(3)
h3("hello")

val j: ((Int, String)) => Char =
  (x: (Int, String)) => x._2.charAt(x._1)

val myTuple = (4,"hello")
//j.tupled(myTuple)
j(1,"hello")

```

```

val f: Int => Int = <function>
val res0: Int = 10

val h: Int => (String => Char) = <function>

val res1: Char = l
val h3: String => Char = <function>
val res2: Char = l

val j: ((Int, String)) => Char = <function>

val myTuple: (Int, String) = (4,hello)
val res3: Char = e

val g: PartialFunction[(Int, String), Char] = <function1>

```

```

val f = (x: Int) => x * x +1
f(3)

val h: Int => String => Char =
(x: Int) => (y: String) => y.charAt(x)

h(3)("hello")
val h3 = h(3)
h3("hello")

val j: (Int, String) => Char =
(x: Int, y: String) => y.charAt(x)

val myTuple = (4, "hello")
j.tupled(myTuple)
j(_, "hello")

```

```

val f: Int => Int = <function>
val res0: Int = 10

val h: Int => (String => Char) = <function>

val res1: Char = l
val h3: String => Char = <function>
val res2: Char = l

val j: (Int, String) => Char = <function>

val myTuple: (Int, String) = (4,hello)
val res3: Char = o
val res4: Int => Char = <function>

val g: PartialFunction[(Int, String), Char] = <function1>

```

```

val f = (x: Int) => x * x +1
f(3)

val h: Int => String => Char =
(x: Int) => (y: String) => y.charAt(x)

h(3)("hello")
val h3 = h(3)
h3("hello")

val j: (Int, String) => Char =
(x: Int, y: String) => y.charAt(x)

val myTuple = (4, "hello")
j.tupled(myTuple)
val partialFunction = j(_, "hello")
partialFunction(3)

```

```

val f: Int => Int = <function>
val res0: Int = 10

val h: Int => (String => Char) = <function>

val res1: Char = l
val h3: String => Char = <function>
val res2: Char = l

val j: (Int, String) => Char = <function>

val myTuple: (Int, String) = (4,hello)
val res3: Char = o
val partialFunction: Int => Char = <function>
val res4: Char = l

val g: PartialFunction[(Int, String), Char] = <function1>

```

val h3 = h(3)

h3 is partially applied curried function

val partialFunction = j(_, "hello")

partialFunction is partially applied uncurried function

```

val jAsAFunction: (Int, String) => Char =
  (x: Int, y: String) => y.charAt(x)

def jAsAMethod(x: Int, y: String): Char =
  y.charAt(x)

```

There's no partial method

But these are the same thing

Call by name Vs Call by value

```

def logTime(b: Boolean, time: String): Unit =
  if (b) println(time)

→ logTime(true, LocalTime.now().toString)

def logTime(b: Boolean, time: String): Unit =
  if (b) println(time)

+ logTime(false, LocalTime.now().toString)

```

```

def logTime(b: Boolean, time: String): Unit
  09:54:46.088913

def logTime(b: Boolean, time: String): Unit
def logTime(b: Boolean, time: String): Unit

```

Even though b is false, LocalTime.now().toString is calculated, this is a hit on performance

```

def logTime(b: Boolean, time: String): Unit =
  if (b) println(time)
def getTime: String = {
  println("getTime: sleeping for 2 second")
  Thread.sleep(2000)
  LocalTime.now().toString
}
logTime(false, getTime)
println("all done")

```

```

val res6: Char = ?
val res7: Char = ?
def logTime(b: Boolean, time: String): Unit
def getTime: String

getTime: sleeping for 2 second
all done

```

So even though b is false, while fetching getTime, its sleeping for 2 seconds, we don't want that

```

def logTime(b: Boolean, timeFunction: () => String): Unit =
  if (b) println(timeFunction.apply)
def getTime: String = {
  println("getTime: sleeping for 5 second")
  Thread.sleep(5000)
  LocalTime.now().toString
}
logTime(false, () => getTime)
println("all done")

```

```

val res6: Char = ?
val res7: Char = ?
def logTime(b: Boolean, timeFunction: () =>
def getTime: String

all done

```

Here `getTime` is not invoked, since `b` is false, so its more efficient

```
def logTime(b: Boolean, timeFunction: () => String): Unit =  
  if (b) println(timeFunction.apply)  
  def getTime: String = {  
    println("getTime: sleeping for 5 second")  
    Thread.sleep(5000)  
    LocalTime.now().toString  
  }  
  logTime(true, () => getTime)  
  println("all done")
```

```
val res5: Char = :  
val res6: Char = ?  
val res7: Char = ?  
def logTime(b: Boolean, timeFunction: () => String): Unit =  
  def getTime: String = {  
    println("getTime: sleeping for 5 second")  
    Thread.sleep(5000)  
    LocalTime.now().toString  
  }  
  logTime(true, () => getTime)  
  println("all done")
```

Here `b` is true, so `getTime` is invoked and it sleeps for 5 seconds

`()` essentially means `Unit`, it can be present as a `String` here, or it can be `None`

We do not need to specify `()` & `.apply` since writing with just the arrow `=>`, the compiler does things for us

```
def logTime(b: Boolean, time: => String): Unit =  
  if (b) println(time)  
  def getTime: String = {  
    println("getTime: sleeping for 5 second")  
    Thread.sleep(5000)  
    LocalTime.now().toString  
  }  
  logTime(false, getTime)  
  println("all done")
```

```
val res5: Char = l  
val res6: Char = ?  
val res7: Char = ?  
def logTime(b: Boolean, time: => String): Unit =  
  def getTime: String = {  
    println("getTime: sleeping for 5 second")  
    Thread.sleep(5000)  
    LocalTime.now().toString  
  }  
  logTime(false, getTime)  
  println("all done")
```

`time: => String`

`symbol =>` is call by name, it is only evaluated only when required

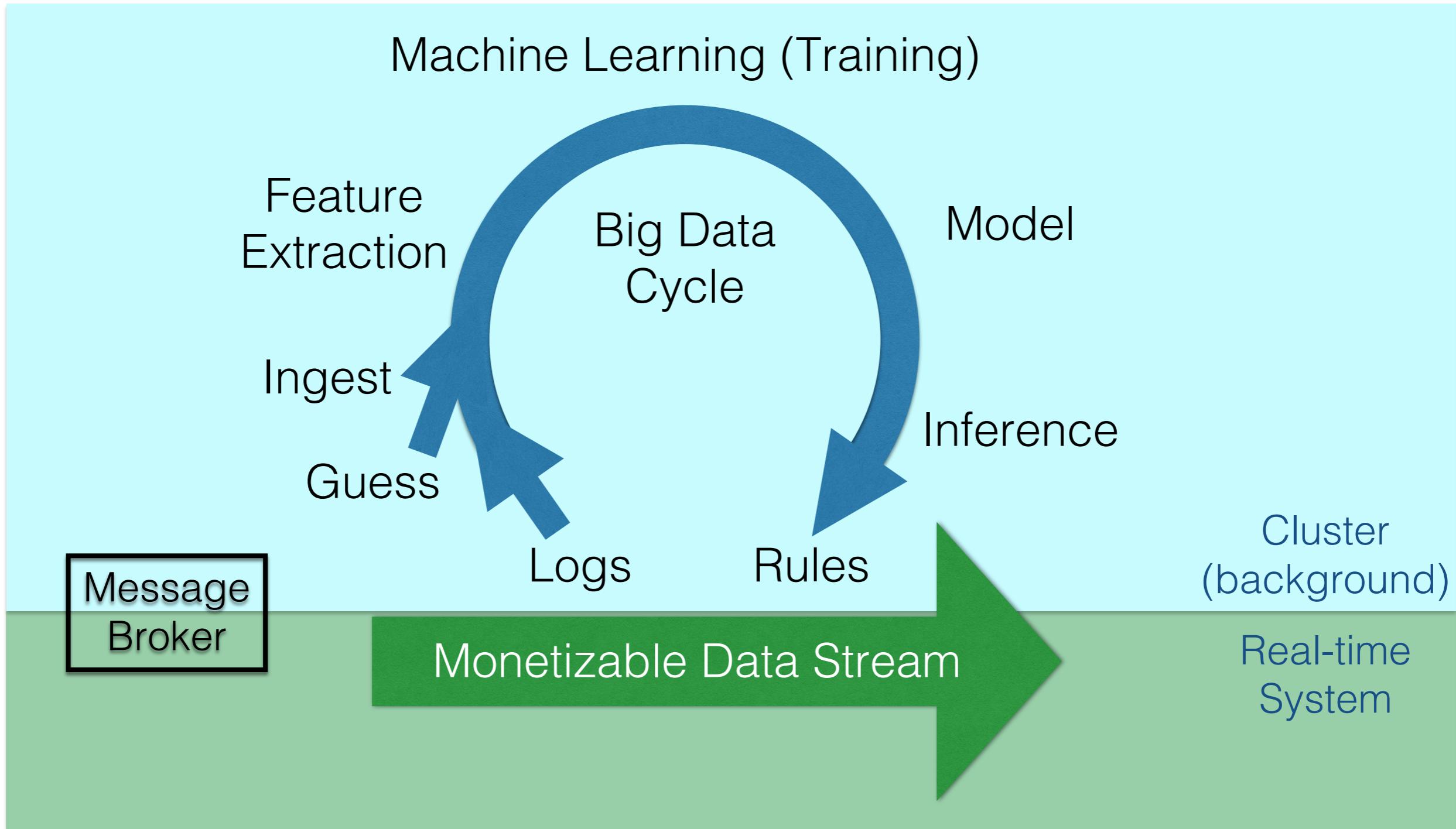
Updated: 2023-01-06

0.2 Big Data: What's it all about?

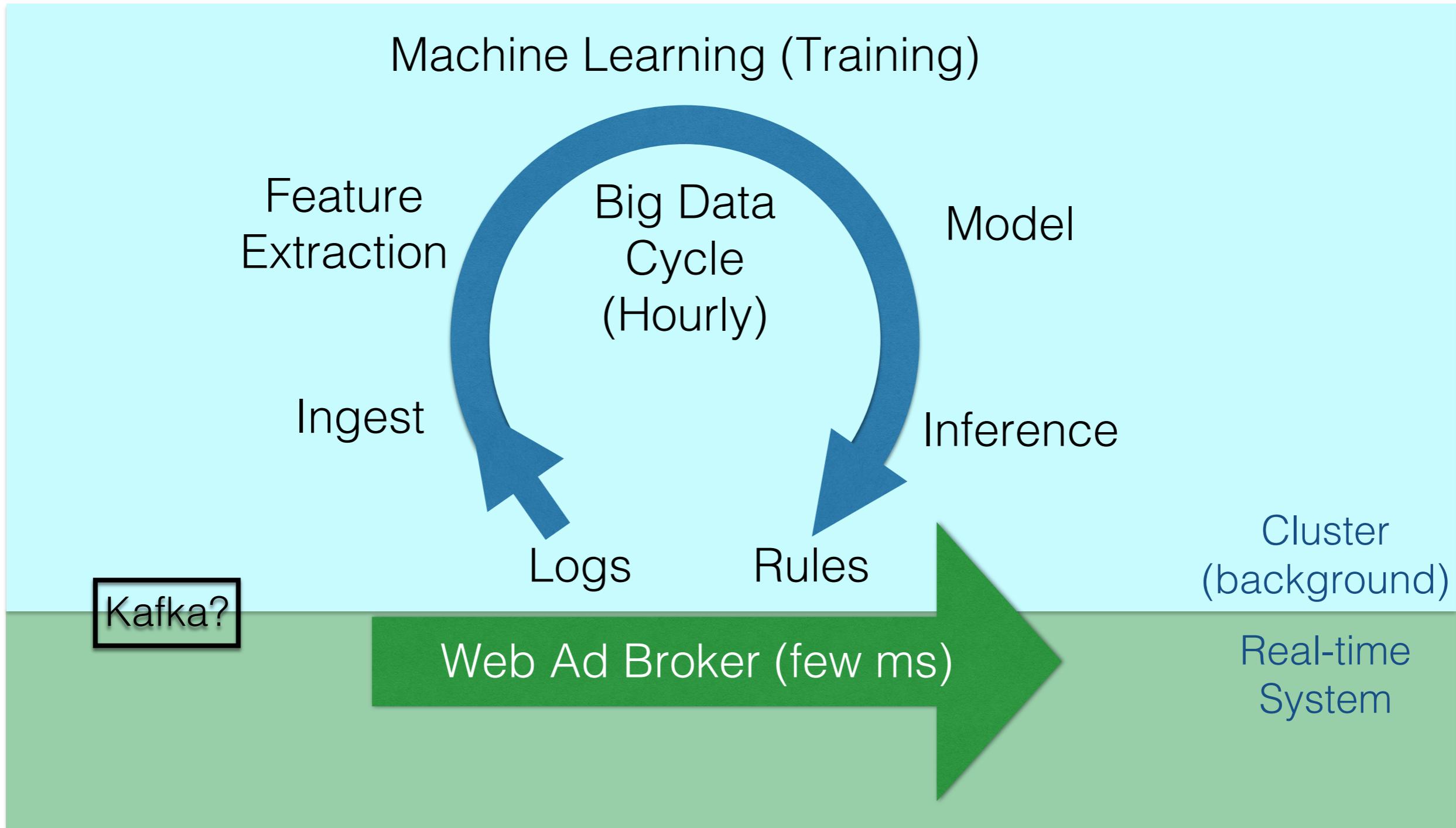
© 2017-22 Robin Hillyard



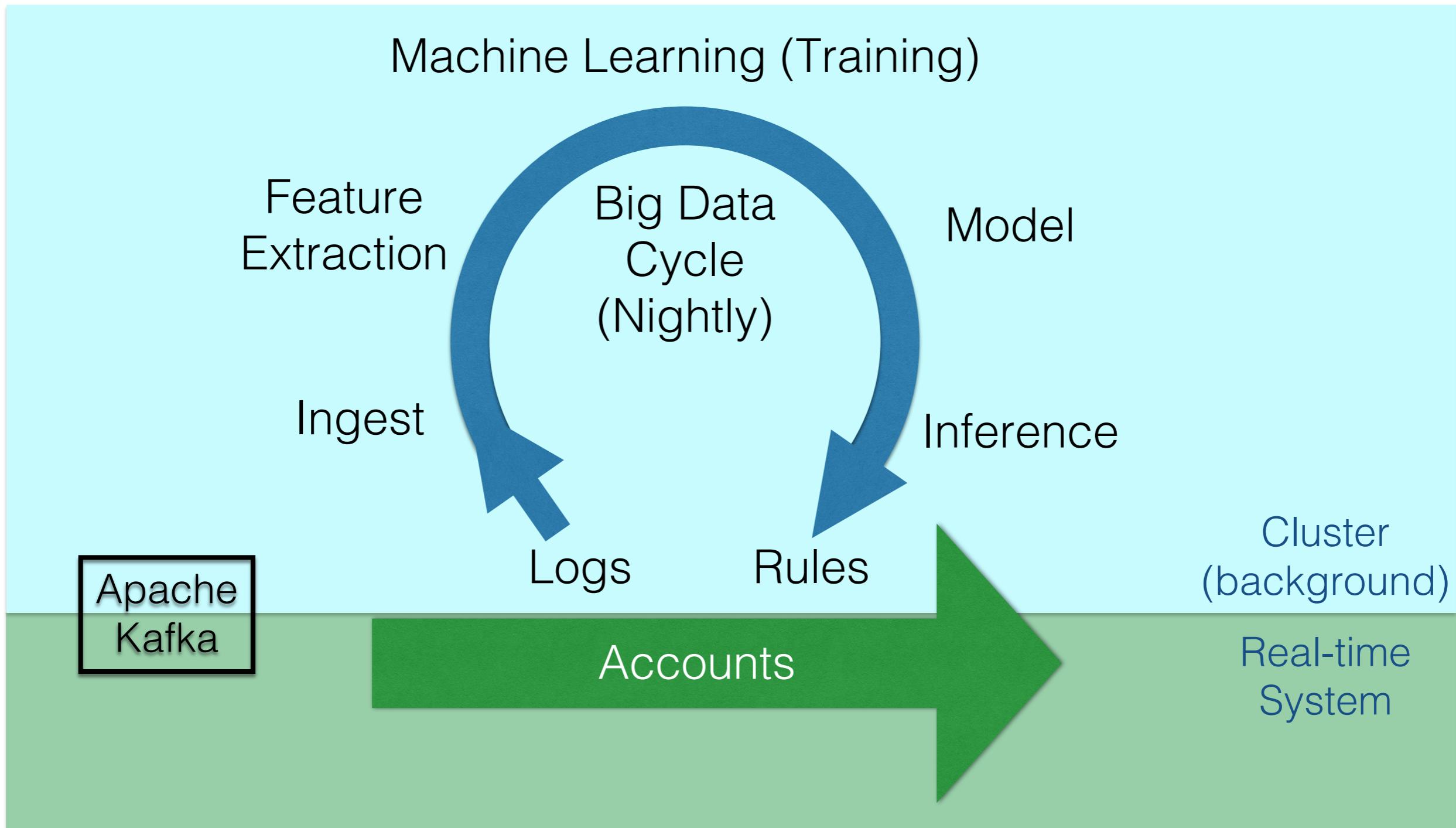
Big Data information cycle



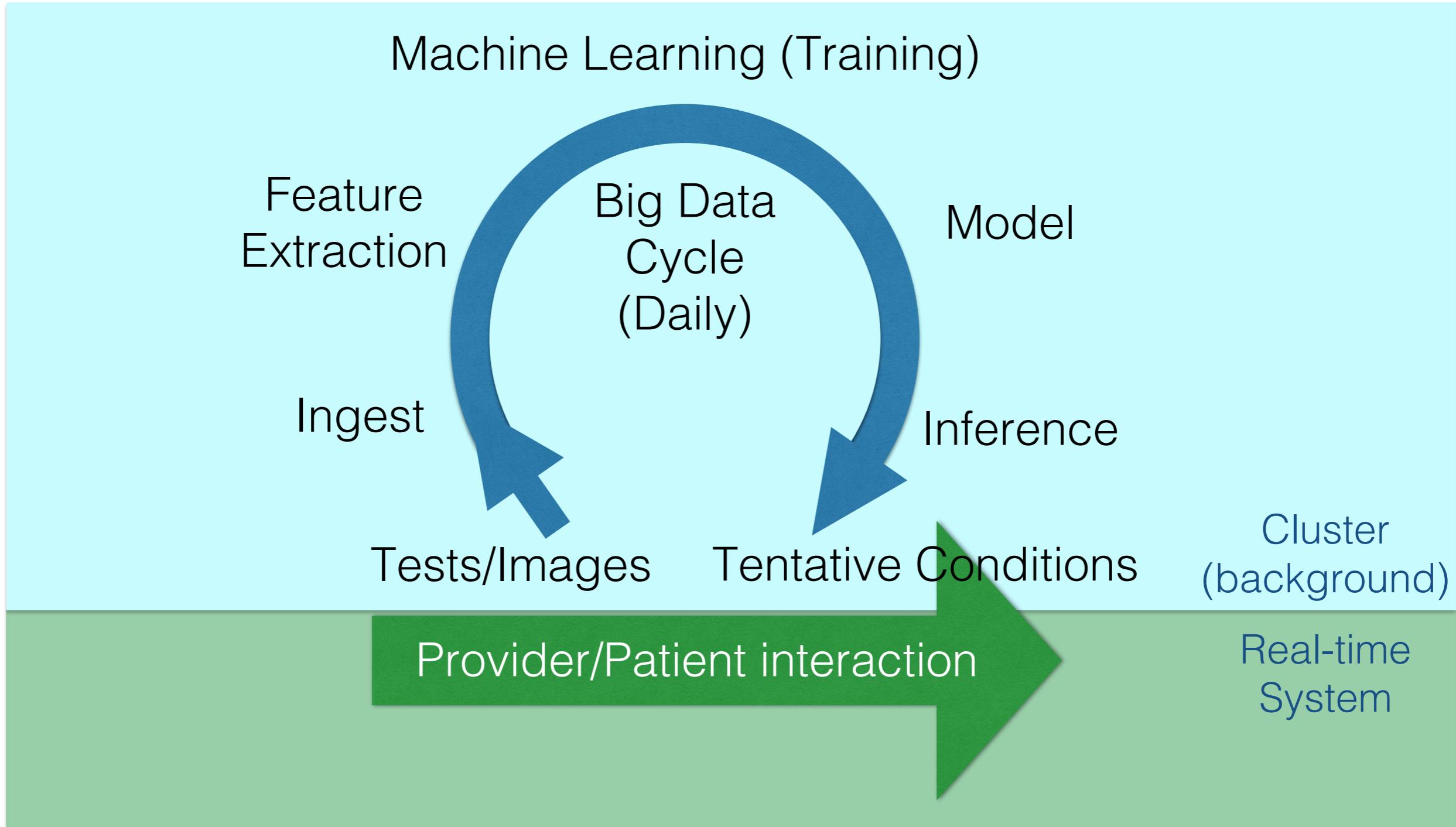
Big Data in web ad service



Big Data in Finance



Big Data in Healthcare



So, what's important?

- Ingest (Scala)
 - Whatever the source of information to be ingested, the ingest (typically, deserialization) process is *enormously* important.
 - We will learn about Domain-specific-languages (DSLs), Parsing, JSON, etc.
- Machine learning (more or less language-independent)
 - *Spark, PredictionIO, H2O, TensorFlow*, etc.
 - Feature extraction, training, validation, application, etc.
- Rules and/or other mutable state (Scala)
 - Actors to encapsulate mutable state;
 - Write to message broker to forward to monetizing stream.

Ingest, etc.

- Scala is very well suited to the tasks required for ingest.
- This is why we will be spending a lot of time in this class on ingest and related techniques.
- Especially, the assignments.

A language for Big Data

(Spark, in particular)

Designing a Language for Big Data

- What constructs do we need? We need the ability to:
 - Define expressions;
 - Execute instructions *in parallel*;
 - Communicate with other threads/cores/nodes;
 - *Pushdown computing*: manipulate collections (or wrappers) by specifying an action to be taken on a single element; even though we do not get access to the whole collection or entity
(ex: Healthcare applications, PII, etc.)
 - Handle errors and missing values; usually happens with raw data
 - Recognize patterns;
 - Deal with *effects*; In scala, we want to use more pure functions, a function must give us back something meaningful, but also deal with a few effects
 - Lazy evaluation. Improves performance

Expressions allow us to compose "values" into one value:
here, f is function, x & y are expressions too

x
f(x)
f x
x f y
f(x, y, ...)
f(x) (y) -> curried form

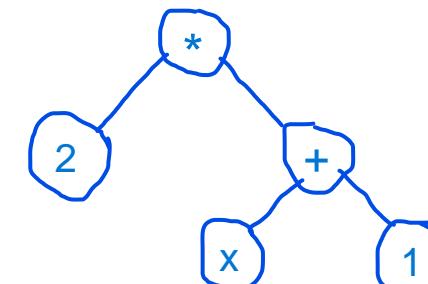
(variations: 'this' and 'apply')

Expressions, methods and types

- Why expressions?
 - Because every “block” of code should yield a value and we *express* that value with... An *expression*.
 - Functions, methods, etc. all yield values (as expressions).
- Expressions:
 - The simplest expressions will be constants;
 - Others will be the results of invoking a method (or function);
 - Others will be compositions of values, for example, arithmetic expressions like $2(x + 1)$.
Compilers convert expressions into Abstract Syntax Trees (AST)

Sugaring?

Desugaring?



Expressions, methods and types

- Others should be parameterizable (where values depend on other expressions that may not be known at programming time):
 - But, if an expression is parameterizable (we would call that a **method**, subroutine, procedure, etc.) then we need a way to express the domain (range, behavior, etc.) of the parameterized value;
 - We can define the domain of a parameterized value by “**type**.”
 - OK, so we need types.
 - Should we just stick to things like integer type (-2³¹ to 2³¹-1) and real type? No.
 - How about (Cartesian) “product” types such as a Complex number?
 - How about collection types? Wrapper types?
 - ex: generic implementation that work for integers, doubles, strings, etc.
 - And, for the underlying type of a collection? That should be parametric, too.

lists types

generic list type

List[Int]
List[String]

Expressions continued

```
case class Complex( r : Double, i : Double) {  
    def magnitude : Double = sqrt ( r * r + i * i) // here we don't need to  
    } // provide params,  
      since its a part of Complex
```

- But expressions are even more fundamental than this.
- When we divide up a program (in just the same way that we might use divide-and-conquer reduction when solving a problem), we need to divide it into some convenient sub-programs.
- These “sub-programs” *are* expressions.
- Thus, in functional programming, every program, every method body, and the right-hand-side of every lambda, is an expression.

$\lambda x . x^* x$

`x => x * x` // But we need types for compiler to understand

`(x : Int) => x * x` // Lamda expression

`def (x : Int) : Int = x * x` // Usually we don't need to write : Int, copiler usually figures it out, unless its in a recursive manner

Parallelism vs Concurrency

parallelism, different threads can interact with separate data
concurrency, different threads interact with same data
ex: Buying tickets together, updating the DB concurrently
not parallelly

Parallelism and concurrency

- For parallel processing:
 - Divide and conquer—requires unlimited recursion;
 - Map/reduce—requires the ability to join threads together;
 - Dynamic programming—requires efficient lazy evaluation, and memoization
- For concurrency:
 - Pub/sub (publish-subscribe)—or message-passing and queuing, in general—ok for independent topics;
 - Actor model—also message-passing and queuing—but with rigorous safeguards to protect mutable state;
 - Mutex, locks, synchronization—very difficult to get right but necessary for the underpinning of the actor model.

Communication ???

- The primary mode of *inter-process* communication is through sockets.
- But, sockets really only know how to transmit bits and bytes, they don't know how to transmit anything else.
- Therefore, we require some mechanism for serialization-deserialization.
 - This requires passing a value into a method—and returning a value, just like we talked about with expressions, and that in turn requires that we need types.
 - If we need to send/receive type information, how can that be done? We need a way of serializing types also!

Pushdown Computing

- What do we mean by “pushdown” computing?
 - Suppose that a column in a database table is made up of numbers over some arbitrary range.
 - We would like to “normalize” these number so that they are in the range -1 thru +1.
 - We can identify the table, and we can identify the column, but we don’t have direct access to any of the values.
 - One possible solution is that we retrieve all the values from the column as an array, then we measure the actual range; then, for each element of the array, we invoke a method which scales that value appropriately; then we copy the array back into the database table.
 - But wouldn’t it be better just to create a function (the equivalent of invoking the method) and apply it to the column as a whole (after another call to get the range)?

Pushdown Computing continued

- We established that we're going to need functions.
- But let's say we need to apply two functions in our pushdown computing model: multiply by 2 and increment by 1.
- That pushdown function is likely to be quite expensive, right?
 - We have to get every element from memory and apply our function to it.
 - If we have two functions to be applied, wouldn't it be a lot more efficient if we *composed* the two functions into one function and then applied that?
- Thus, we need *functional composition!*

Patterns

What is the inverse of an expression?

In an expression, we compose one or more values into one value

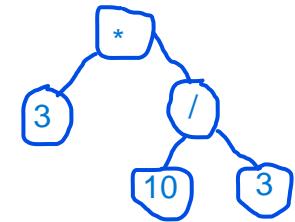
But surely, we also want to do the opposite

Take a value and de-compose it into the values that composed it originally (if possible)

If we provide $3 * (10 / 3)$

We might get output: 9.99999999

But if we express it as AST,
we can get output: 10



- Much of machine learning is about recognizing patterns:
 - We need a way to determine what expression to evaluate when a particular pattern is recognized
 - An “if” clause can do that: but it only recognizes two distinct patterns;
 - A “switch” clause can do that: but it only recognizes integer patterns;
- We will need a way to recognize an object that is a *compound* object—whose characteristics can *each* be matched independently.

```
val c = Complex(1, 0)
```

```
val Complex(r, i) = c
```

```
println(r) //prints 1, pattern matching
```

Effects

- (Pure) expressions can only take you so far.
 - If you are able to calculate an approximation of π , for example, it will do you no good if the run-time system can't share the value with *the outside world* (including *you*) and the program simply terminates.
- Sometimes a value you need has to come from a file, or a database, or a web service on another computer.
 - In other words, you will have to get information from *the outside world*.
 - This is likely to take a long time, quite possibly billions of clock ticks while your thread does nothing.
ex: logging, reasonable effect
- Strict functional programming disallows "side-effects" (the most common types of "effect"), so we need a way to manage these effects safely (not as side-effects but as well-behaved predictable effects).

Lazy Evaluation

ex: call by name: instead of evaluating the expression then passing it into function,
we can see if the expression is required and then only evaluate the expression in the function

- We've already mentioned lazy evaluation in the context of dynamic programming and waiting for effects.
- But there's another more logic-based reason for lazy evaluation:
 - Let's say we have two expressions a and b , each of which yields a Boolean and we want to combine their values:
 - `val result = a & b`
 - That's fine, right? But what if a evaluates as false? It doesn't matter what the value of b is: the final result will always be false.
 - So, it's a complete waste of time evaluating b .
 - Thus, we need a way to defer the evaluation of b until we know it's actually necessary—*lazy* evaluation.

Designing an actual language

- The first big question: How/Where
 - How will this language run on the machines that we need it to run on?
 - Follow the model of “C” and have different run-time implementations for different microprocessors and operating systems?
 - That’s likely to get very good performance, but it makes every application that a programmer might build have to be built for each and every platform. **it's not portable**
 - There’s another drawback to this approach: libraries.
 - Where are the libraries that application programmers need to come from? This is especially important in the early days of our language. People won’t adopt the language if it’s too difficult to get libraries for it.
- Answer: the Java Virtual Machine (JVM)
 - compilers can use new Java versions, but some constructs may not be present for an old Java version for a given Scala version

jvm's bytecode does not really know whether a lang was written in Java, Scala, Groovy, etc.

What is this JVM?*

- Java is two different things (three if you include the run-time library):
 - The Java language (in which you write your programs):
 - The compiler takes your source code and outputs “bytecode,” in *.class files;
 - You can run the compiler on any machine that is convenient to you.
 - The Java Virtual Machine (JVM) jvm doesn't load class until it needs to
if it needs to use that class, it stops program execution until that class is loaded
 - The JVM actually runs your code on a specific machine;
 - Your code can therefore run on *any* machine for which there is an implementation of the JVM (without any recompilation and, in theory at least, without any extra testing);
 - The JVM knows how to deal with *int*, *double*, *char*, *byte*, etc. but, for more complex O-O programs, it must load the appropriate .class files from the disk. ex: String -> ubiquitous class
 - As long as it gets valid bytecode, the JVM *doesn't care which language* produced it.
 - The Java run-time library:
 - This has been split up recently, but basically is all the standard library stuff you need.

* If you're not familiar with Java (or even if you are)

Object-oriented?

Should our language be “object-oriented”? What does that even mean?

- We naturally assume that our language will have objects like 0, 1, π , “Hello World!”, true, etc. All languages have these, so the term O-O is a bit confusing!
- However, so far, our objects don’t have any defined behavior. We can’t actually *do* anything with them. We could try writing something like $\text{sum}(0, 1)$ and expect that 1 will be returned. However, in mathematics, we’re more used to writing $0 + 1$.
- It seems obvious that 0 and 1 should share much behavior—they are both non-negative integers (i.e. counting numbers), something well understood. The only real difference is in their actual value: where they appear in the range $0..\infty$.

Object-oriented? continued

Instance constructor
new Integer(37)
evaluated in runtime

Type constructor
List[Int]
evaluated before runtime
in compile time, from List[X]

How can we define this idea of behavior?

List[X] in a library, it won't evaluate it, it won't have a main function, but say we List[Int] from List[X], it will be evaluated in compile time

- Recall from our discussion of expressions that we need “types.” Types define behavior (and range). For example, all counting numbers behave the same and so could be described by the “counting number” type.
- A language could define that any positive integer has type “counting number.”
- In an O-O language, we can define a (parametric*) class. Here, the parameter of a parametric class of type “counting number” is its value. Once it’s been constructed with that value, it becomes an instance of the class—also known as an “object.” An object is thus non-parametric, i.e. a “singleton”.
- In this case, we could also define a singleton of type “counting number” to represent the negative numbers (i.e. an error condition).

* Known in Java as “generic”

List <: Seq
String <: Char Seq

<: means sub-type

Object-oriented? (2)

The Liskov Substitution Principle

- Suppose we have our counting number type but we recognize that there is a special sub-type of these numbers called “primes.” Such instances would have an additional property that the regular counting numbers don’t have: primality.
- If we have an expression that expects a counting number, we can substitute a prime (because it is also a counting number, its supertype). But, if we expect a prime, we cannot substitute just any counting number because instances of that type don’t have the primality property.
- This is the essence of the Liskov Substitution Principle.
- And it is this idea of sub-types, super-types and substitution that is one of the core features of object-oriented programming.

```
def print( xs : Seq [ Char Se ] ) : Unit  
    print(xs)
```

val x = List [String] () We can use pass x in print, since we are passing its sub-type => this is true since Liskov's Substitution principle

Object-oriented? (3)

Dot

- There's another significant aspect of O-O.
- Suppose we want to get the *value* of a counting number object. We can write:
`value(x)`
- We simply pass a reference to *x* (an instance of counting number) to a method (or function) called *value* which is defined for counting numbers. But, in practice, that would mean writing something like*:
`CountingNumber::value(x)`
- There's a convenient shorthand for this:
`x.value`
- Note that this is really just a syntactic convention. But, it makes code a lot easier to read and, IMO, has a lot to do with why O-O languages took off as they did.

* This is how we did it before O-O

Object-oriented? (4)

Final decision?

- So far, everything we've discussed about O-O fits in perfectly with what we're trying to accomplish: we like the idea of types, classes and objects. And we like the syntactic convention regarding the dot.
- There's one more factor: the JVM (which we earlier decided to use) is based on this concept of classes (types, really). In other words, we can get all this stuff more or less for free.
We also need to send the types, since the JVM compiler would not know about it
But later once its in the bytecode, it really doesn't matter in the JVM
- However, we don't have to define huge hierarchies of types like Java programmers do. We have the option to enable a different method of constructing types: called a “type constructor”. More on that later.

Let's get down to details

Expressions:

- Think of an expression such as $x+y$. There are three logical ways to implement this:
 - A (global) sort of method called “+” which takes two parameters (one pre- and one post-)
 - Treat it like an instance method on x as if it is really $x.+(y)$ — or even $y.+(x)$
 - The first approach (the Java way) requires that the language itself knows all the types of object that can be added together using “+”.
 - The second approach (the Scala way) requires only that each type that supports addition should define the + method. Does it bother you that “+” isn’t a proper identifier? Relax!
- Now, what about $1+2$?
 - “1” is a constant, not a variable, so how would that work? How about: $1.+(2)$?
 - Yes, just the same as “ x ”—they both are of type *Int*, which supports the “+” operator.

Expressions continued

- What about $x+y+z$?
 - We do have to set some precedence rules:
 - Most methods bind to the left so this expression is actually $x.(y.(z))$ [binding rule](#)
 - This raises a bit of a problem when we dealing with containers.
 - Suppose we have a list xs of type $List[Int]$. And we want to append another Int x . We could define a method like $+$ and write $xs + x$. That would work.
 - But it isn't exactly the best because sometimes we would want to *prepend* x as in $x + xs$. In this case, x for an Int is regular addition and it expects an Int parameter.
 - We want to invoke a method on xs itself. So, we could just write $xs.prepend(x)$
 - But that's just not as nice as we'd like: let's write $x +: xs$ and define that methods whose names end (or begin) “ $:$ ”, associate with the “ $:$ ”.

[overrules, precedence binding](#)

More Complex Expressions

Complex expressions

- We can now write some very complex programs simply by nesting expressions to any depth we like.
- But, ultimately, the terminal tokens of such an expression will need to be known values such as:
 - 1, 3.12415927, true, or “Hello World!”
 - Or constant (or “literal”) values of bits, bytes, longs, etc. (easily defined).
- If we’ve got two objects we want to keep together (like a “key-value pair”)?
 - How can we represent that?
 - Well, what if we just wrote something like: (“pi”, 3.1415927)?
 - Those parentheses essentially group objects: and we call the group a “tuple”
 - We can do it with more than two... (1, “pi”, 3.1415927, true) up to 22 elements*

* This limit is very arbitrary but there has to be a limit somewhere.

More Complex Expressions (2)

But, didn't we say we also needed to be able to specify functions?
How?

- Let's go directly to Church's Lambda calculus:
 - A function to multiply a value by 2 would be written thus: $\lambda x.x^*2$
 - Let's use a similar syntax: $x \Rightarrow x^*2$.

More Complex Expressions (3)

The need for extraction:

- Take a look at this expression to determine the number of meeting rooms necessary to accommodate a list of meetings (given by *ms*):

```
(ms flatMap {  
    m => Seq(Transition(m.start, start = true), Transition(m.stop, start = false))  
}).sorted.scanLeft(0)((d, t) => if (t.start) d + 1 else d - 1).max
```

- Of course, the compiler will lap this up! But, for a human, this might be a bit more challenging. Wouldn't it be nicer if we could *extract* expressions and give them a name? How about this (see next slide)?

More Complex Expressions (4)

```
val transitions = ms flatMap {  
    m => Seq(Transition(m.start, start = true), Transition(m.stop, start = false))  
}  
val orderedTransitions = transitions.sorted  
val rooms = orderedTransitions.scanLeft(0)((d, t) => if (t.start) d + 1 else d - 1)  
val totalRooms = rooms.max
```

- This is the exact same program, only it declares “variables” (like algebraic variables) that, through their names, give some sense of their purpose.
- Note that we can also ensure DRY by using such declarations. [DRY - don't repeat yourself](#)
- Furthermore, because the compiler effectively substitutes the right-hand-side of any declaration wherever it sees the left-hand-side, the final run-time code is more like the original version than the human-readable version.

Complete programs

A complete program is actually wrapped inside an “Object”

- Recall that we have adopted the JVM as our platform.
 - A source file in the JVM is associated with a “.class” file which contains the executable byte-code corresponding to the source file (it’s what the compiler generates).
 - In order to follow that scheme, all our program’s source code* is wrapped in an “Object”.
 - The “value” of that object is an expression as in “(ms flatMap ... max)”
 - But to help us humans, and to follow DRY, an object will usually consist of an expression, preceded by a set of declarations.
 - Such declarations may define variables, methods, classes, objects, traits.
 - Actually, classes, objects, and traits are going to be allowed at the top-level of a module, while variables and methods must belong to one of those top-level constructs.

[grouping of multiple groups/traits module](#)

* In Scala, we allow a source file to contain any positive number of types

Parametric or non-parametric?

So far, every example has been non-parametric

- But, remember from earlier, we said we would need declarations to be parameterizable, perhaps both in values and in type.
- Objects, including functions, are not parametric.
- But, part of DRY is not repeating code that differs only in type.
- So, if we have a function to sum the values of a *List[Int]*, we want to be able to define a *method* on a *List* that yields the sum, whatever the underlying type.

Parametric or non-parametric?

Define own declarations?

| Construct | Parametric types? | Parametric values? | Can define own declarations? |
|-----------|---|---|------------------------------|
| Class | yes | yes | Yes: for instances |
| Trait | yes <small>trait Show[T] { // Parametric type T def show(value: T): String }</small> | no* | Yes: for instances |
| Object | no | no | Yes: for singletons |
| Method | yes | yes | Yes (private declarations) |
| Function | no | yes <small>def apply(x: Int): Int = x * factor // factor is a parametric value</small> | no |
| Variable | yes (one) <small>val someNumber: Option[Int] = Some(42) // Parametric Type Int</small> | n/a | no |

Methods are to *functions* as classes are to *objects*: methods are more variable / parametric than functions

- Re-writing a function as a method has two possible advantages:
 - A more human-readable syntax with names (i.e. *signature* = *body*);
 - The option of making types parametric.
- It is trivially easy to turn a method into a function when the context requires it.

* Traits can have values in Scala 3

Execution in parallel

Parallel execution is an *effect*.

- For this, we will need some sort of asynchronous method invocation.
 - Let's put that off until later, although in the worst case, we can surely make use of the Java *Future* and *Call* classes. In fact, since Java8, we have many higher-level functions such as *CompletableFuture*. Remember, we will have access to all the classes defined by the standard Java run-time library.
- Threads, etc.
 - If we like, we can use the same low-level mechanisms available to Java programmers. But, if we use *CompletableFuture*, etc. we should be able to do everything we need.
- (De-)Serialization:
 - Java gives us this for free because, each JVM can get access to the byte code of a class as long as the .class file is on the machine.

Pushdown computing and functions

Let's say we have a container such as a *List[T]*?

- We will need a way to invoke a function on each element and return a new *List[R]*.

- We want to call this *map* and its signature will be something like:

`def map[R](f: T => R): List[R]` In Java, we usually get a type iter-able or stream not the same object type,
But, in scala, we can get the result in the same object type

- But, we want to do that for every sort of container that there is, right?
 - So, this will be one of the more common methods and is defined for a high-level trait called *Iterable*.
 - The logic is that, if you can iterate over a collection, you can also “map” it.
 - Similarly, we will define *flatMap*:

`def flatMap[R](f: T => List[R]): List[R]`

Functional Composition

- For functional composition, all we really need is to be able to write lambdas:
- Here, we compose three functions together into one:

```
val f: Int => Double = _ * 2
val g: Double => Double = _ + 1
val h: Double => String = _.toString
val fgh: Int => String = x => h(g(f(x)))
```

f(...) means compose f
means h compose g compose f

- This is the equivalent of using the built-in functional composition method *andThen*:

```
val fgh = f andThen g andThen h
```

Errors and missing values

- We'll leave handling exceptions for now, but what about missing values? It's a very common problem in datasets.
- Suppose that a row of a dataset is supposed to have a value for gender: "M," "F," or "O." But this is often left blank. We could store the value as an optional Boolean. It's either true (M), false (F), or other/missing.
- Let's do that with a container called *Option[Boolean]*. *Some* a sub type of Option
- There will be three possibilities: *Some(true)*, *Some(false)*, *None*.
- Note that *None* doesn't have any value. But it is an instance of the type *Option[Boolean]* or *Option[X]* where *X* is any other type.

Let's talk about patterns

```
p match {  
    case x => Int : x  
}  
  
// if we had to cast a var in Java,  
// we would have done:  
if (p instanceof Integer) {  
    Integer i = (Integer) p  
}
```

In which contexts would it be useful to allow patterns?

- Well, for a start, we need to be able to pattern-match an object x :

```
x match { ... }
```

- Inside the brackets, we will want to define a set of partial functions each of which will yield a different result (one for each of the different patterns of x).

- Each partial function will need to take the form of a lambda with a set of “bound” variable(s) which derive from the value of x .

- We'll need the following types of pattern:

- Constants, extractors, guarded patterns, nested patterns, typed patterns...

```
if (x) y else z
```

Pattern types

Where patterns are used the most:

`val <pattern> = expression`

// val (x, y) = method that results
in a tuple

`x match { case <pattern> => expression }`

`for { <pattern> <- expression } yield x`

- A constant pattern is obvious;
- But what is an “extractor?”
 - An extractor is the opposite of a constructor: instead of passing in parameters and getting back an instance of a class, an extractor matches an instance of a class and gets back the parameters that were used to construct it.
 - For instance, if we had previously created a variable as a tuple, we can match it as a tuple...

```
val x = (1, "Hello")
```

```
x match { case (n,s) => println(s"$n:$s"); }    1 : Hello
```

- A guarded pattern is one that matches only if a predicate is true.

Companion Objects

These extractors sound simple, but they still have to be defined and, before we do that, I want to suggest another way to take advantage of the JVM class structure.

- Java allows you to create properties that belong to a singleton class related to a particular class by using the keyword “static.”
- We will do something similar but we’ll split the code into their two logical parts: instance properties (class) and singleton properties (object). They will have the same name and coexist in a module.
- A useful method which we can define in the companion object of class T is $apply(...):T$ whose purpose is to construct a T without “new.”

“apply” methods

- Let's say we define:

```
class Complex(real: Double, imag: Double)
```

when we create case class M, compiler automatically provides us with a companion object & apply for class M

- We can define in the companion object *Complex* the following:

```
def apply(real: Double, imag: Double): Complex = new Complex(real, double)
```

- And we propose one little syntactic simplification:

$t(p)$ is shorthand for $t.apply(p)$

Where t is an instance of class T which defines a method *apply*(...); and:

$T(p)$ is shorthand for $T.apply(p)$

Where object T defines a method *apply*(...); and:

In scala 3, sometimes we don't need to use, apply & unapply - provided by compiler automatically

Now, to instantiate a Complex, we can write something like:

```
Complex(1, 0)
```

Extractors

Back to extractors...

- We implement them with an *unapply* method which, essentially, returns the (optional) list of the original parameters used to create the instance.
*if its Any[],
ex:: Any = compiler always returns None*
- This is kind of the opposite of the *apply* method which we just met.
`def unapply(z:Complex):Option[(Double,Double)] = Some((z.real,z.imag))`
- Because *apply* and *unapply* are so useful, we're going to define a special type of *class* called a *case class* which automagically defines these methods (and others).

Lazy code*

- How are we going to enable lazy (deferred) evaluation?

- What's the difference between the following two declarations?

```
val x = math.PI  
def y = math.PI
```

- The biggest difference is that the variable “x” takes on the value of π immediately after the declaration, but “y” is a “method” and so has no value of its own until somebody *invokes* y. Recall that methods usually have parameters so there *is* no result until an invocation is made.

- Actually, in this case, we can rewrite the second declaration:

```
def y = math.PI
```

- can be rewritten in the following form:

```
lazy val y = math.PI.
```

* We're going to skip effects for now.

Lazy evaluation continued

- When you pass a parameter into a method, there are two logical ways to do it:

- Call-by-value (cbv)
- Call-by-name (cbn)

| | num of times vars are evaluated |
|----------|---|
| val | 1 time |
| def | 0 to infinity times present in the code |
| lazy val | 0 or 1 time |

- So, for example,

- Suppose you pass the following expression as a parameter: $3 + 4$.
- Do you think that the compiler should sum 3 and 4 and pass 7 (cbv) into the method? Or pass $3+4$ as an expression (cbn) into the method?
- Does it matter? `def square (x : Int) : Int = x * x` // here x & y need not to be evaluated until it reaches & references rms function
- It matters a lot! `def rms (x: => Int, y : = > Int) : Double = sqrt (square (x) + square (y))`

- Since “no size fits all,” we need to specify which form in the declaration of the method.

// if lazy val y is written with
`def y() ->` compiler doesn't know what value of y would
come - > is y lazy ???

Lazy evaluation continued...

- Let's go back to our earlier example: a method to yield the logical *and* of two Boolean values:

```
def and(a: Boolean, b: Boolean): Boolean = a & b
```

- Well? Are a and b passed by name or by value in this method?
 - The “default” is call-by-value. So, this won’t help. The value of b will still have to be figured out and passed into the method regardless of the value of a .
 - We’re going to need a syntax to tell the compiler to pass the parameter by name (essentially, pass a function):

```
def and(a: Boolean, b: => Boolean): Boolean = a & b
```

```
() => thunk // used in pass by name
```

And this will allow for lazy lists

- If we declare a collection type called a lazy list something like the following:

```
class LazyList[T](head: T, tail: => LazyList[T])
```

- Then, when we construct a new *LazyList*, the tail of the list won't be evaluated until (and unless) a method is called that references the tail. *In other languages, like Java it will be evaluated and runs till infinity, but will throw memory problems like stack overflow*
- This will allow us to declare the following:

```
def positiveInts(from: Int) = new LazyList[Int](from, positiveInts(from+1))
```

- When we call this method with *from* = 1, it will define an infinitely long list of integers starting from 1 without any memory problems!

2.1

Spark Intro to Scala

From the point of view of Spark

© 2019-22 Robin Hillyard



So, what actually *is* Spark?

- Spark is
 - A fast, general-purpose cluster-computing platform
 - Based on map/reduce paradigm
 - In that sense, similar to Hadoop.
 - Optimized for clusters with lots of memory
 - An open-source Apache project
 - Part of the ^{JVM} ~~JDK~~ eco-system
 - Implemented in Scala

Typical applications?

- What sort of applications are typical for Spark?
 - Machine learning: classification, clustering, recommenders, etc.;
 - Natural language processing, sentiment analysis;
 - Mathematical modeling—matrix manipulation, anything that can be done in parallel batches;
 - Message subscription;
 - Graphs; need parallel execution - using scala
 - Relational information processing (using SQL); 90% of applications
 - Just about any Big Data application...

Word Count (python)

- Let's say we want to count the words in a document:

```
import sys
from pyspark import SparkContext, SparkConf
if __name__ == "__main__":
    # create Spark context with necessary configuration
    sc = SparkContext("local", "PySpark Word Count Example")
    # read data from text file and split each line into words
    words = sc.textFile("D:/workspace/spark/input.txt").flatMap(lambda line:
        line.split(" "))
    # count the occurrence of each word
    wordCounts = words.map(lambda word: (word, 1)).reduceByKey(lambda a,b:a
        +b)
    # save the counts to output
    wordCounts.saveAsTextFile("D:/workspace/spark/output/")

    Spark Context, uses a core  
split by 2 or 2 cores for  
efficient processing
```

- And to run this program:

> spark-submit pyspark_example.py

Why python?

- Well, for now, we're pretending that we want to execute this code but so far we don't have a Spark framework to run it on:
- Because we haven't invented Scala yet ;)

What would we need to implement such a framework?

- A cluster manager (execution system);
- An optimizing execution planner; DAG so that dependencies can be applied appropriately
- A partitionable lazily-evaluated collection type; RDD
- A way to compose and to (de-)serialize functions.

Details?

- Cluster manager: External (YARN, Mesos, Kubernetes, stand-alone, etc.)
- Execution planner: Internal (map/reduce-like); optimizable for some jobs, e.g. SQL
- Why do we say “Partitionable and lazily-evaluated collection type?”
 - More detail in next slide.
- A way to compose and to serialize functions.
 - More detail in subsequent slide.

Why do we say “Partitionable and lazily-evaluated collection type?”

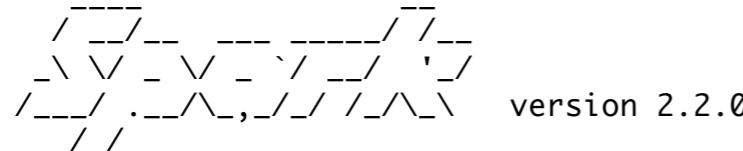
- Unlike Hadoop, which tends to do each map/reduce as a separate task, with persistence and replication to the HDFS, the Spark execution planner builds a DAG (directed, acyclic graph) of stages, the whole of which is implemented in one remote “job.”
- In Spark, the map/reduce pipeline that corresponds to one “action” is represented by a directed acyclic graph of map and reduce stages.
- The map stage may actually invoke many different transformations, each represented by a function, and which can be composed together into a single transformation for “lazy*” evaluation.
- A stage is broken up into tasks, typically one per partition.
- The data structure which enables these operations is called a Resilient Distributed Dataset (RDD)
 - * *lazy evaluation is an important aspect of functional programming: delaying evaluation until absolutely necessary.*

A way to compose and to serialize functions.

- Composing functions
 - Each task is physically executed with the appropriate partition, the appropriate function, and on the appropriate executor (i.e. worker node). We therefore prefer to compose functions together if possible to avoid extra performance overhead.
 - For example, suppose we wish to transform a partition with a function which doubles the value and then we wish to transform the same partition with a function which increments the result. Instead we can compose these two functions into one function of form $x \Rightarrow x * 2 + 1$
- Closures
 - A “closure” is a function which is partially applied so that all variables, except the one(s) expected as input, are “captured”, i.e. bound to values `val f = 2 ; val k = 3 // f & k are captured & bound to function transform`
 - For example, suppose we wish to transform a partition with a function which multiplies the value by a factor f and adds a constant k to the result: we will then need a function like: `def transform(x: Int) = // partial function`
 - But in order for this to be usable out of context (i.e. on a remote node), we must capture the values f and k . This is what a closure does.
- Serializing functions In a broadcast, a constant is applied in each lambda (not really captured?)
 - These functions will do us no good at all if we can't send them, like ordinary objects, over the network as part of the task definition.
 - We therefore need a way to serialize these functions. But the driver and the executors are all running the JVM which means that they all understand byte code. Pure functions can be serialized and deserialized as byte code!

An example Spark job*

```
Robins-MacBook-Pro:spark-2.2.0-bin-hadoop2.7 scalaprof$ bin/spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
18/01/02 20:28:25 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes
where applicable
18/01/02 20:28:31 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObjectException
Spark context Web UI available at http://172.20.20.20:4040
Spark context available as 'sc' (master = local[*], app id = local-1514942907077).
Spark session available as 'spark'.
Welcome to
```



```
Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_144)
Type in expressions to have them evaluated.
```

```
Type :help for more information.
```

```
scala> val rdd = sc.textFile("/Users/scalaprof/flatland.txt")
rdd: org.apache.spark.rdd.RDD[String] = /Users/scalaprof/flatland.txt MapPartitionsRDD[1] at textFile at <console>:24
scala> rdd.flatMap(_.split(" ")).map((_,1)).reduceByKey((x, y) => x + y).sortBy(-_._2).take(10).foreach (println _)
(the,15)
(of,14)
(and,10)
(a,9)
(to,9)
(will,8)
(you,8)
(have,6)
(at,5)
(I,5)
scala> :quit
```

evaluates until foreach or save as
text file is called

* This actually is in Scala

Another example

```
package com.project.csye7200

import Data.preprocess_data
import TFIDF.{pipeline_stages, tf_idf}
import org.apache.spark.ml.PipelineModel
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.col

object ModelLoad extends App {
    val session = SparkSession.builder().master("local").appName("Load Model").getOrCreate()
    session.sparkContext.setLogLevel("ERROR")
    val test = session.read.options(Map("header" -> "true", "quote" -> "\"", "escape" ->
    "\"")).csv("src/test/scala/resources/datasets/test.csv")
    val dfs = session.createDataFrame(Seq("",  

        "On Christmas day, Donald Trump announced that he would ..."  

        , "", "user", null.asInstanceOf[Integer])
    ).toDF("title", "text", "subject", "date", "target")
    val df = dfs.union(test)
    df.show()
    val text_preprocessed = preprocess_data(df, "text")
    val text_TFIDF = tf_idf(text_preprocessed, "text")
    val final_df = text_TFIDF.withColumnRenamed("text_tfidf", "features")
    val (indexer, assembler) = pipeline_stages(2)
    val title_preprocessed = preprocess_data(text_TFIDF, "title")
    val title_TFIDF = tf_idf(title_preprocessed, "title")
    val nvmodel = PipelineModel.load("src/test/scala/resources/model/NaiveBayes")
    val start_time = System.currentTimeMillis()
    val nv_prediction = nvmodel.transform(title_TFIDF)
    val stop_time = System.currentTimeMillis()
    println("Model Prediction Time - " + (stop_time - start_time) + "ms")
    nv_prediction.filter((col("date") === "user")).show(true)
    val nv_json = nv_prediction.filter((col("date") === "user")).select("prediction", "probability"). toJSON
    nv_json.show(false)
}
```

Summary

- What we need to implement Spark:
 - lazy evaluation
 - functional composition
 - closures
 - serializable functions
 - JVM
 - Other stuff that we will discuss next.
- These all come built-in with **Scala** but not with Java (at least not until very recently).
- For another (better) take on this, please refer to: [A Newbie's guide to Scala...](#)

2.1 What's the big deal? About functional programming?

Copyright © 2019-2022 Robin Hillyard



What's the big deal about
functional programming?

What does functional programming even mean?

What does “functional programming” mean?

- "What's in a name? That which we call a rose
By any other name would smell as sweet."

Juliet from *Romeo and Juliet*, William Shakespeare.

- It's not just about functions, despite the name.
- There used to be a type of programming called “symbolic programming” (You could argue that Perl is symbolic programming):
 - The tokens in such a language aren't just thrown away as code is generated: the tokens are a living part of the language.
- To me, that's the essence of “functional programming.”

Let's just look at one tiny (but significant) detail

- In a language like Java or C, an assignment is just what it sounds like: a value is “assigned” to a variable:

```
double x = Math.PI;
```

- This “statement” combines a “declaration” of *x* and an assignment (in Java, you can split those two parts up if you like).
- The constant value *pi* has been given to *x* and that will be its value until changed.
- In Java, if you never want to allow *x*’s value to be changed, then you write *final* in front of the statement:

```
final double x = Math.PI;
```

- But, even in the second form, you would never really say that *x* and *Math.PI* were actually the same thing, would you? It’s essentially a temporary relationship.

But in Scala, things are a bit different:

- In Scala, we can write the following:

```
val pi = math.PI
```

- This declaration essentially says that, from here on, we treat *pi* and *math.PI* identically. We could, at any point, choose either one. In other words, *pi* has simply become an alias for *math.PI*.
- And here's the really important point:
 - At any future time, we can replace *pi* by *math.PI* (or we can replace *math.PI* with *pi*) and our program will be absolutely identical: it will in every respect behave the same!
 - So, what's the point of this alias?

Declarations as aliases

- Here are some of the reasons why this is a “good thing:”
 - Let’s say that an important piece of information in our program is the circumference of a circle.
 - And let’s say that we write:

```
val circumference = d * math.PI
```

- ... and then, in several places, we write *circumference* instead of *d * math.PI*.
 1. We have avoided repeating ourselves (the DRY principle of programming);
 2. We have identified the concept of circumference with an appropriate name;
 3. We’ve (hopefully) made our program more understandable and, at the same time, more mathematical.

Don't
Repeat
Yourself

What about functions, though?

- Let's say you have developed a matrix manipulation framework that is designed to run on a cluster of 1000 nodes.
- It can multiply matrices together, transpose them, invert them, transform their elements from one domain to another, all that stuff.
- You will need a driver which allows the programmer to set up the matrices and define operations such as multiplication, etc.
- All of those are easy to implement except *transform*. How exactly are you going to let your programmer specify what operation should be performed on an element of a matrix *when he doesn't have direct access to an element of a matrix?*
- This is a case of “pushdown” logic. We want the system to push our function down into the depths of its data structures.
- Therefore, we need to be able to treat functions just like other objects.

Functions are first-class objects

- If functions are objects, just like "Hello, World!" or `math.PI`, then we can operate on functions and apply them to (sets of) parameters.
- How can we operate on functions, though?
- Operations on functions are not well-known obvious stuff like "*", "+", etc. which are suitable for arithmetic objects;
- Instead, we can define our own functions and *compose* them.
- Why do we need to **compose functions**?
 - Because composition is the chief mechanism that allows **re-use**.
 - And code re-use is the chief contributor to **writing robust, efficient code with consistent behavior**.

Functional composition and higher-order functions

- A higher order function is a function, at least one of whose parameters is a function (where xs is a collection, f is a function).
 - $xs \text{ map } f$
 - $xs \text{ reduce } f$
- Functional composition is where the result of a higher order function is itself a function.
 - $g \text{ andThen } h$

But how do we define the functions we need?

- Lambda calculus (lambdas for short).
- A lambda is an anonymous function (or “functional literal”) which defines how its parameters “bound variables” are transformed into its result.
- A lambda also “closes” on any free variables in scope.
- You can find lambdas in the following languages...
 - Java8 (actually there were special cases of lambdas all the way back in Java 1);
 - Python
 - Scala
 - Haskell
 - Pharo
 - All functional programming languages

But that's not all...

- There are many other aspects of functional programming, many of which we can't find in Java8 or Python:
 - Lazy (deferred) evaluation;
 - Type inference and shape preservation;
 - Pattern-matching;
 - Referential transparency:
 - Immutability by default;
 - Pure functions (lack of side-effects);
 - Tail recursion;
 - Tuples;
 - Monads, etc.
 - Higher-kinded types.

More later...

See 2.3 Functional Programming in Scala

2.2 Let's write some code

Copyright © 2020-22 Robin Hillyard

A large, semi-transparent watermark of the Northeastern University logo is positioned in the lower right quadrant of the slide. The logo consists of a stylized 'N' formed by two overlapping arches, with the university's name 'Northeastern University' written in white serif capital letters across the center.

Northeastern
University

Hello World!

“Hello World!”

- That’s a perfectly fine Scala expression. But it’s not very useful as is because, if we do nothing else, we will never actually see anything.
- But that’s because we never really specified how we wanted to “see” the string.
- If we were writing a microservice, we would probably want to pack that string into an HTTP response object (or similar).
- But let’s just follow the convention and write it to the console:
`println ("Hello World!")`

Organizing our code for the compiler

- That last bit of code still isn't really useful if we want to compile it because the compiler needs a little bit of “boilerplate”:

```
object HelloWorld {  
    println ("Hello World!")  
}
```

- This tells the compiler that we are defining an “object” and, within it, we have created an expression which prints the message. However, this code will actually do nothing at all (!) because it's not a *executable program*.
- However, if you were running the “REPL” (read-evaluate-print-loop) you wouldn't need all this object stuff.

Hello World in the REPL

Welcome to Scala 2.13.4 (OpenJDK 64-Bit Server VM, Java 11.0.8).

Type in expressions for evaluation. Or try :help.

```
scala> "Hello World!"
```

```
val res0: String = Hello World!
```

```
scala> println ("Hello World!")
```

```
Hello World!
```

```
scala> println (res0)
```

```
Hello World!
```

```
scala>
```

Worksheets

can use it for unit testing)

- In IntelliJ IDEA (and maybe Eclipse and/or Metals):
 - We can test ideas by coding in a Scala worksheet (extension *.sc).
 - We can write it like a program—not in an object—but, when we run it, each line that defines a value prints that value on the right-hand side of the screen.

The screenshot shows a Scala worksheet named "Factorial.sc" in IntelliJ IDEA. The code defines a tail-recursive factorial function using an inner helper function. It also calculates the factorials of 5 and 20.

```
import scala.annotation.tailrec

println("Welcome to the Factorial worksheet")

def factorial(x: Int): Long = {
    @tailrec def inner(r: Long, i: Int): Long =
        if (i == 0) r else inner(r * i, i - 1)

    inner(1L, x)
}

val f5 = factorial(5)
val f20 = factorial(20)
```

Output:

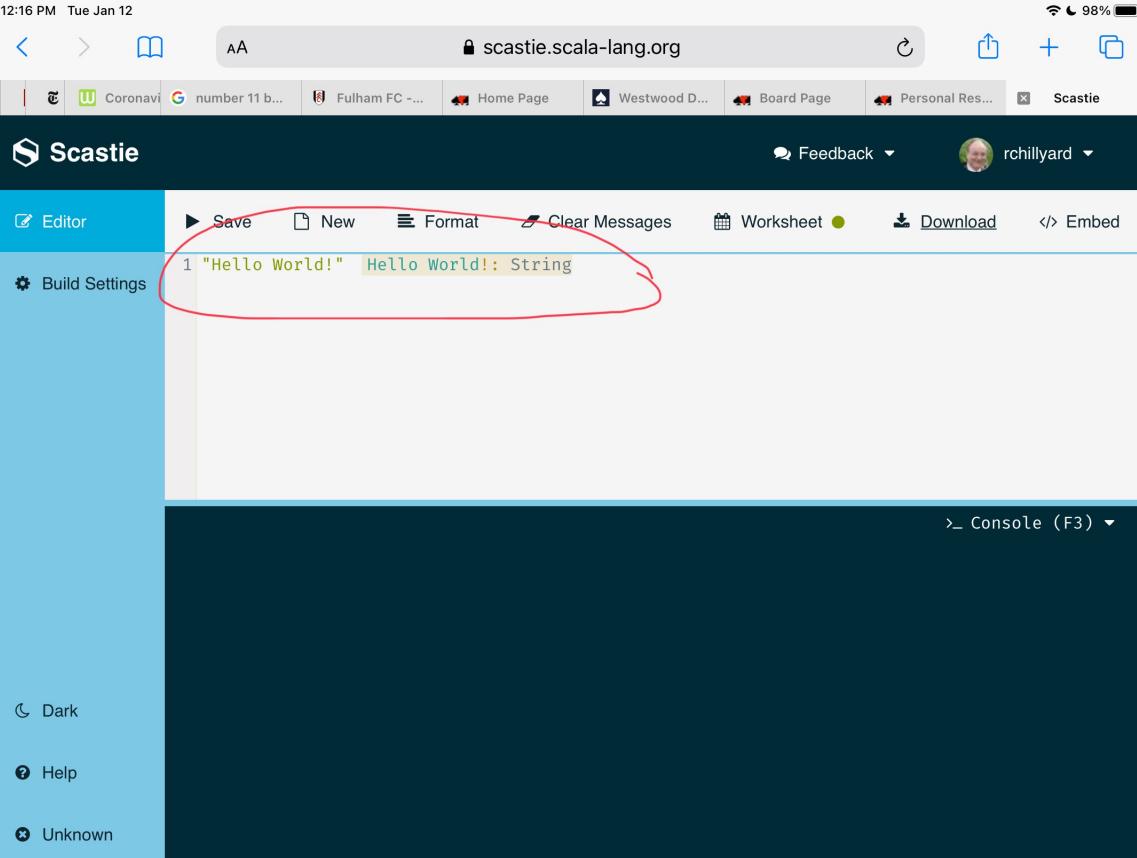
```
Welcome to the Factorial worksheet

def factorial(x: Int): Long

val f5: Long = 120
val f20: Long = 2432902008176640000
```

Scastie

- You can play with Scala on the web at <https://scastie.scala-lang.org>



A screenshot of a web browser displaying the Scastie website (<https://scastie.scala-lang.org>). The browser's address bar shows the URL. The Scastie interface has a dark-themed header with a logo, user profile, and navigation links like 'Feedback' and 'Scastie'. Below the header is a toolbar with buttons for 'Save', 'New', 'Format', 'Clear Messages', 'Worksheet' (which is selected), 'Download', and 'Embed'. On the left, there's a sidebar with 'Editor' and 'Build Settings' tabs, and buttons for 'Dark', 'Help', and 'Unknown'. The main workspace shows a single line of Scala code: '1 "Hello World!" Hello World!: String'. A red oval highlights this code line. At the bottom right of the workspace is a 'Console (F3)' dropdown.

not like REPL / worksheet

will only show console output

Scalafiddle

- You can do the same thing with [scalafiddle](#), even better: it allows you to easily include functional libraries, such as Cats, Scalaz, Shapeless, Monocle, etc.
- However, it's stuck on Scala 2.12 (and we should learn 2.13, or 3.0).
- We'll come back to this later.

Back to Scastie...

- Did you notice that we didn't need to say *println* because we saw the evaluated expression in a comment on the right?
- But, normally, we'll want to show things in the console (lower, black, part of the page).
- So, we'll do `println("hello world!")` as before [please follow along with your own Scastie session].

Our program isn't very interesting

- Because it always does the same thing and that thing is pretty simple.
- Let's do something more interesting:

```
val hw = "Hello World!"  
def time() = java.time.LocalDateTime.now  
def greet = s"$hw it's ${time()}"  
println(greet)  
println(greet)
```

this is a method, since we have used def
it will be evaluated twice in println (re-evaluate it)

but if we had used val instead of def
it will be evaluated then and there
println output will not change

Variables and methods

- The `val` keyword in front of `hw`, declares it as a “value” – `hw` is really best thought of as an alias for “Hello World!” (More on this later).
- But we call these “variables” in Scala because they serve the same purpose as variables in algebra. They don’t change their value once set up (they are immutable), but they can stand in for anything.
- But what about `time` and `greet`?
- Again, in each case, we have defined an alias for an expression. The difference here is that the evaluation of the expression won’t always be the same.
 - *But, in functional programming, we normally try to have methods which always return the same result for a given parameter set. What’s up with this?*

Pure and impure methods

- There are two reasons why a method might not always yield the same result:
 - The method takes parameters (which, of course, can have different values);
 - The method might not be a *pure* method.
- A **pure** method is one which, for a given set of input parameter(s), will always yield the same result AND there must be no side-effects*.
- A less strict kind of method is an **idempotent** method.
 - An idempotent method typically has side-effects, but it can be called any number of times and the effect on the rest of the world will be the same as calling it only once.

* See next slide

Effects

- Functional programs are all about *expressions*.
 - But, if we don't actually use that expression anywhere, it will not affect the outside world (we are part of the outside world).
 - Therefore, we will never see the result.
 - What use is that?
- So, any real-life computer program must cause some *effect* to happen in the outside world.
 - How can we do this functionally?
- In a pure functional language, such as Haskell, then all I/O (input/output) is achieved through *effects*. iThe Performance is provided in cats = category theory f to f =
- But, Scala is a lot less strict about effects.*

* There is an excellent library called Cats which includes easy-to-use effects

Side-effects

- A method which obviously is designed to cause an effect, such as *println*, is not—in and of itself—a side-effect.
 - That's because *println* yields *Unit* (the empty type) so a programmer just knows that, if the method is to achieve anything at all, it must be through an effect—and the very name of it makes that obvious.
- But, a method which logs some string *and* yields a result, is causing a side-effect. It's better for a method to do one thing *or* the other.
 - Effects are tricky because we can't easily reason about them. And, if we can't reason about them, we can't prove a program that involves them.
- Just imagine testing the system clock.
 - Invoking, e.g. *System.currentTimeMillis()*, relies on an effect (the ticking of the system clock).
 - How could you ever test it? And you definitely can't prove such a program.

Back to Hello World

- We have embedded the current time into our greeting.
- The time method has no parameters but it's *not pure*: it relies on an independent source of information: the system clock (it's an effect).
- That's why we should define it using "def" instead of "val." A **val** is evaluated only once, while a **def** is evaluated every time it is invoked (which makes a lot more sense when there are parameters).
- It may come as a bit of a surprise that we can define a "def" (i.e. a method) without any parameters at all.
 - By convention*, if the method relies on a side-effect (i.e. not pure), then we define (and invoke) it with empty parentheses—example: `close()`.
 - if the method is pure, we define/invoke it without parentheses.

* The compiler doesn't get too upset if we get this wrong.

Other details

- We didn't have to specify the type of the `val` and `defs`: that's because **Scala has type inference.**

```
val hw = "Hello World!"  
def time() = java.time.LocalDateTime.now()  
def greet() = s"$hw it's ${time()}"  
println(greet())  
println(greet())
```

- *Time()* relies on a Java class and is not pure. There's no problem with that.
- *greet* relies on *string interpolation* (the `s"..."` construct).
- Each time we invoke *greet*, the result is slightly different.
- BTW, I've tried to emphasize the keyword/tokens by putting them in **bold**.

Back to using the compiler

delayed Serialization

- If we want to compile this program and be able to run it in order to get the time, we are going to have to put the boiler plate back...

```
object HelloWorld extends App {  
    val hw = "Hello World!"  
    def time = java.time.LocalDateTime.now  
    def greet = s"$hw it's $time"  
    println(greet)  
    println(greet)  
}
```

- Notice the “extends App” in the definition of *HelloWorld*. Without that, still nothing would happen because there would be no “main” program.
- *App* is a class which causes the expression to be evaluated when you run the program. [In this case, the “expression” is the last *println*, a *Unit*].

Note that I skipped the () from now, time, and greet. No big deal.

What about control structures?

- Other than these declarations, there really aren't any statements in Scala code. Everything is an expression.
- So, it probably won't come as a big surprise that control structures are all basically functions which yield a value and which are invoked via "keywords"^{*}. For example:

```
val msg = if (time.getDayOfWeek.getValue==0) "happy rest day" else "keep working"
```

- Or, we could write:

```
val message = time.getDayOfWeek match {  
    case java.time.DayOfWeek.MONDAY => "Monday's child is fair of face"  
    case java.time.DayOfWeek.TUESDAY => "Tuesday's child is full of grace"  
    case java.time.DayOfWeek.WEDNESDAY => "Wednesday's child is full of woe"  
    case java.time.DayOfWeek.THURSDAY => "Thursday's child has far to go"  
    case _ => "I didn't have enough space to do these"  
}
```

- Note how all of these are expressions which yield a result.

they are keywords not functions

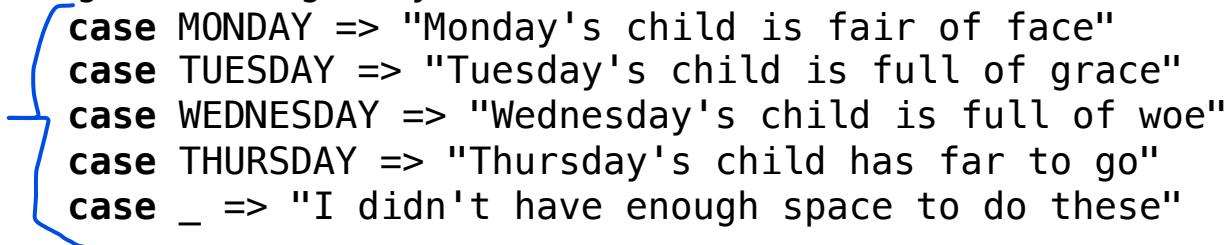
* Are if/else/match keywords? Or functions?

Imports

- BTW, there's another kind of aliasing that we haven't come across yet. It's invoked when we use *import**. It's a lot like "alias" in SQL. It's just a way of shortening what otherwise would be quite a long type name.

```
import java.time.LocalDateTime
object HelloWorld extends App {
    val hw = "Hello World!"
    def time = LocalDateTime.now
    def greet = s"$hw it's $time"
    println(greet)
    println(greet)
    val msg = if (time.getDayOfWeek.getValue==0) "happy rest day" else "keep working"
    import java.time.DayOfWeek._
    val message = time.getDayOfWeek match {
        case MONDAY => "Monday's child is fair of face"
        case TUESDAY => "Tuesday's child is full of grace"
        case WEDNESDAY => "Wednesday's child is full of woe"
        case THURSDAY => "Thursday's child has far to go"
        case _ => "I didn't have enough space to do these"
    }
}
```

alias to DayOfWeek



* Note that **import** doesn't have to come at the start

What if we want to get the time repeatedly?

- There are a couple of different ways, depending on whether we want to invoke something with effects (like printing on the console) or we want to get a list of results:

| | | |
|---|----------------------------|-------------------------|
| ^{0, 1, 2} <code>(0 until 3) foreach (_ => println(greet))</code> | <code>yields unit</code> | not good implementation |
| <code>for (_ <- 0 until 3) println(greet)</code> | <code>yields unit</code> | not good implementation |
| <code>for (_ <- 0 until 3) yield greet</code> | <code>yields String</code> | good implementation |

- The first two are pretty much equivalent. Notice the “yield” in the third one. Each results in three greetings (but the last doesn’t print anything—it is an expression).
- BTW, you might notice the use of “_” in each of these expressions. This basically represents an unidentified variable, that’s to say a variable whose name we don’t need because we never are going to refer to it.

Anything else?

not there in Scala 3

- I'm told there's a **do while** construct but you never need to use it!
- And there's an **if** clause which, like **for** without **yield**, conditionally invokes a side-effect. But you really shouldn't use it.
- Note that the normal **if** expression is like “?” in Java. So, you must always provide an **else** expression (if you see a compiler message about *Any* type or perhaps *Serialized* or *Product* type, check your **ifs**).
- Are you wondering if it's possible to declare mutable variables?
 - Good question. We'll discuss in much more detail later.
 - There are such things—but you should never use them!

2.3 Let's write some more code

© 2019-2022 Robin Hillyard



Scala expressions

- First of all, as we will see in 2.3, **Scala programs are also expressions.**
- So let's take a look at an expression. Follow along with me in the REPL.

```
scala> 1 + 1
```

```
res0: Int = 2
```

```
scala> res0 * res0
```

```
res1: Int = 4
```

```
scala> 2 * 2
```

```
res2: Int = 4
```

Expressions (2)

```
scala> "Hello World!"  
res3: String = Hello World!
```

```
scala> "Hello World!\n" * 3  
res4: String =  
"Hello World!  
Hello World!  
Hello World!  
"  
"
```

```
scala> s"$hw\n" * 3  
res5: String =  
"Hello World!  
Hello World!  
Hello World!  
"  
"
```

```
scala> val hw = "Hello World!"  
hw: String = Hello World!
```

Refactoring: extraction

```
scala> 2 * 2
```

```
res6: Int = 4
```

```
def square(x: Int) = x * x
```

```
square: (x: Int)Int
```

```
scala> square(2)
```

```
res7: Int = 4
```

```
scala> square("Hello World!")
```

```
<console>:13: error: type mismatch;
```

```
  found  : String("Hello World!")
```

```
  required: Int
```

```
    square("Hello World!")
```

```
    ^
```

```
scala> def square(x: Double) = x * x
```

```
square: (x: Double)Double
```

```
scala> square(2)
```

```
res8: Double = 4.0
```

Lists

```
scala> List(1,2,3)
```

```
res9: List[Int] = List(1, 2, 3)
```

```
scala> res9 foreach println
```

```
1
```

```
2
```

```
3
```

```
scala> def square(x: Int) = x * x
```

```
square: (x: Int)Int
```

```
scala> val xs = res9
```

```
xs: List[Int] = List(1, 2, 3)
```

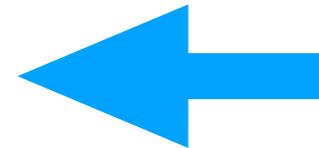
```
scala> xs map square
```

```
res10: List[Int] = List(1, 4, 9)
```

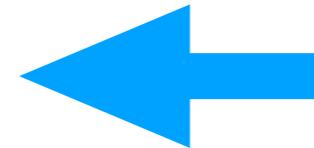
```
scala> xs sum
```

```
res11: Int = 6
```

This is *not* an expression!
We are invoking a side-effect
while yielding *Unit*.



This is an expression!
See explanation of *map*
in following slide



Looping over Lists

- Transformations:
here we don't need direct access to xs -> x
 - `for (x <- xs) yield f(x)` here we don't need direct access to xs
`xs map (x => f(x))`
- Cartesian products:
`xs map f`
`xs flatMap (x => ys map (y => (x, y)))`
 - `for (x<-xs; y<-ys) yield (x,y)` [List of n^2 elements]
 - `for (x<-xs) yield for (y<-ys) yield (x,y)` [List of lists]
- Zip:
`xs zip ys`
`xs map (x => ys map(y => (x,y)))`
- Printing:
`for (x <- xs) println x`
`xs foreach println`
`xs foreach (println _)`
`xs foreach (x => println (x))`
Compiler throws warning,
we need not write _

Lists: map

- So, what's this *map* method all about?
 - *map* takes an iterable sequence of some sort (a collection) and yields the same sort of sequence except that each element in the result is formed by applying a function to the corresponding element in the original sequence. In the example from the previous slide, *square* is the function.

```
scala> xs map square  
res10: List[Int] = List(1, 4, 9)
```

Lists (2)

```
scala> xs map square  
res10: List[Int] = List(1, 4, 9)
```

Repeated

```
scala> for (x <- xs) yield square(x)  
res19: List[Int] = List(1, 4, 9)
```

This generates identical code to the map expression above

```
scala> xs ::= 4  
res20: List[Int] = List(1, 2, 3, 4)
```

This is also an expression: the original xs doesn't change

```
scala> 0 +: xs  
res21: List[Int] = List(0, 1, 2, 3)
```

Note that the ":" in an operator associates left or right with the collection

Empty list?

- How can we get an empty list? There are several ways:
 - ***Nil***: this is the special name of the empty list—it is a case object that extends the trait *List* with type *Nothing*.
Sub Type of any Type
 - ***List()***: this is the *apply* method that takes a comma-separated sequence of elements: in this case, none.
Example: `xs.empty`
 - ***List.empty***: this is a standard technique for most collections—if it's possible to have an empty one, then this constant in the companion object will give it.

Lists (3)

- Writing our own *sum* method:

```
scala> def sum(xs: Seq[Int]) = xs match { case Nil => 0; case h :: t => h + sum(t) }
<console>:13: error: recursive method sum needs result type
      def sum(xs: Seq[Int]) = xs match { case Nil => 0; case h :: t => h + sum(t) }
                                         ^
                                         ^
```

```
scala> def sum(xs: Seq[Int]): Int = xs match { case Nil => 0; case h :: t => h + sum(t) }
sum: (xs: Seq[Int])Int
scala> sum(res20)
res22: Int = 10
```

- Note that we have used *pattern-matching* here. More later

Sum by iteration

- It is of course possible to use a **var*** to calculate the sum of a collection...

```
def sumByIteration: Int = {  
    var result = 0  
    foreach (a => result += a)  
    result  
}
```

- ... but we should learn **not** to use **var** in our programs.

* actually, that's what the low-level methods do

Lists (4)

This will give you
your “10 times table”
But as a list of 100
Items, not as a table.

```
scala> for (i <- 1 to 10; j <- 1 to 10) yield i * j  
val res1: IndexedSeq[Int] = Vector(1,2,3,4,5,6,7,8,9,10,2,4,...)
```

desugaring

This gives an
identical result

```
scala> (1 to 10).flatMap(i => (1 to 10).map(j => i * j))  
val res2: IndexedSeq[Int] = Vector(1,2,3,4,5,6,7,8,9,10,2,4,...)
```

- So, what is this *flatMap* method all about?
- *map* takes an iterable sequence... (as you already know).

Lists: flatMap

- So, what's this *flatMap* method all about?
 - *flatMap* is like *map*, except that the result of the function is not a single element but is a collection of elements (in fact, a collection of the same shape as the original iterable). If we used *map* with the same function, we would end up with a (two-dimensional) table. But *flatMap flattens* the table into a sequence. *Factors* returns a list of all factors (inc. 1).

```
scala> List(1, 2, 3, 4) flatMap factors  
res16: List[Int] = List(1, 1, 2, 1, 3, 1, 2, 4)
```

Map, flatMap and filter

for comprehension

predicates like if

- We will find that these methods, *map*, *flatMap*, and *filter* are **really important** in functional programming. They crop up everywhere, and not just in sequences or collections.
- Indeed, any container that supports these methods is called a *monad*. More on that later.

Infix, postfix, and dot notation

- Let's talk about the syntax of method invocations:
 - `xs.length`: give us the length of some iterable collection `xs`;
 - This (dot notation) follows a Java-like syntax where the dot implies that the method `length` is invoked on the object `xs`. Standard O-O syntax, in other words.
 - `xs length`: is legal but somewhat discouraged.
 - It's called postfix notation and is semantically identical to the dot notation. Indeed, you must `import scala.lang.postfixOps` otherwise you will get a compiler warning.
 - `xs.length()`: is legal but discouraged unless `length` is invoking a side-effect.
 - It's also discouraged to invoke (or override) a method that has/hasn't parentheses with an invocation (or signature) which hasn't/has parentheses. Stay consistent, in other words.
 - Example: `inputStream.close()`

1-ary methods

- A very common situation is where we have an object (the “receiver”), a method and one parameter to that method, for example:
 - $\begin{array}{ll} \text{pre operand} & \text{xs} \\ \text{operator} & + \\ \text{post operand} & x \end{array}$
 - $\text{xs} :+ x$
 - (“Infix” notation): this adds an x to the right-hand end of collection xs .
 - We could also write it as follows: $xs.:+(x)$ but we prefer to use the more natural “infix” notation, especially when the method is symbolic (looks like an operator). We generally use the dot notation when the method is verbal.
 - $\text{xs} :+ (x, y)$
 - It’s even possible to have 2-ary methods written in infix notation.
 - In this context, (x, y) is a “tuple” and we might write it this way to add a key-value pair to a map-type object like *HashMap*.
 - Actually, we’d be more likely to write this equivalent form:
 - $\text{xs} :+ x \rightarrow y$ $\begin{array}{ll} \text{key} & x \\ \text{value} & y \end{array}$

Chaining method calls

- There are some situations where it's very typical to use dot notation in a chain.
 - This typically happens with so-called “lens” functions which return a modified version of an object. For example, setting up a Spark session:
 - `val spark: SparkSession = SparkSession.builder().appName("WordCount").master("local[*]").getOrCreate()`
 - It can also be used when combining *map* methods or other shape-preserving methods on collections, for example:
 - `lines.flatMap(_.split(separator)).map(_._1).reduceByKey(_ + _).sortBy(-_._2)`

Associativity

- Normally, these methods associate to the left, as we'd expect in object-oriented code.
- But Scala also allows us to define methods which associate to the right, by ending the method name with ":"
- For example: $x +: xs$ adds the element x to head (left-hand end) of the collection xs . This can be rewritten in O-O style as $xs.+:(x)$ but now it's not so obvious that x should end up to the left of xs .
- It's easy to remember: in infix notation, the colon (:) always is adjacent to the collection.

| | | | |
|-----------|--------------------------------------|---------------|------------------------------|
| $xs :+ x$ | is same as | $xs . :+ (x)$ | adds x to the tail of xs |
| $x :+ xs$ | is same as | $xs . +: (x)$ | adds x to the head of xs |
| $x :: xs$ | pattern matching x at head of xs | | |

Summary

- $1 + 2$ is just the more familiar way of writing:
- $1.+(2)$ which is the object-oriented way of saying that we want to invoke the `+` method on object 1 (i.e. `this = 1`) with the parameter (the “addend”) of 2.
- In Scala there are very few true operators (maybe none)—everything that looks like an operator is in fact a method on the first of the parameters.

2.4 Functional Programming in Scala

© 2019-2023 Robin Hillyard



Functional Programming in Scala

- This lesson is long but it contains most of the features which (in my opinion) make functional programming (and Scala in particular) work so well.
- You should take good notes on all this.
- There may be some repetition of a couple of features I've already mentioned briefly (either in “A language for...” or “What's the Big Deal?”

What exactly is functional programming? (repeat)

- Functional programs are *expressions*;
- Higher-order functions and functional composition;
- Parametric expressions (function “literals”) encoded as “lambdas”;
- Lazy evaluation;
- Recursion as the primary re-use mechanism;
- Immutability, pure functions, referential transparency, zero side-effects;
- Pattern-matching;
- Type inference.

What are examples of FP languages?

- Lisp, Scheme, Miranda, Clojure, Erlang, OCaml, Haskell, F#
- What about Scala?
 - Scala is a *hybrid* language: it has features of FP, but it also has features of O-O;
 - Some people think this is a good thing (me);
 - Others think it's a terrible thing (FP purists);
 - More on this later.

Functional Programs are expressions

- A Java (or other “imperative” language) program is a series of statements that accomplish some task such as printing a value or updating a database.
- A functional program is an expression, such as $f(x)$.
- What good is that? What use is a program that just provides some value that has no physical presence?
- Not much good as a main program perhaps, but providing a value to something that can display it/store it/print it, whatever is just fine!
- And, in any case, FP languages allow for I/O, but with stricter rules.

Lambda calculus

- In the 1930s, Alonzo Church developed a notation and a set of simple rules that governed the way computable expressions could be defined.
- In particular, the way he defined the abstraction of a function, e.g. multiply by 2, is: $\lambda x. x * 2$
- This expression gives a name to the bound variable (x) but not to the function itself. Thus, it is anonymous.
- Thus, we use the terms *lambda* and *anonymous function* as synonyms (mean the same).

Parametric expressions encoded as lambdas

- Church's lambda calculus does not take into account the types (or domains) of the variables and terms. But Scala is strictly typed so we must specify the types. And, if we need to refer to a function, we can give it a name.
- For example:

```
val y = 2  
val f1: Double=>String = x => (x/y).toString
```

- In this example, the lambda is $x \Rightarrow (x/y).toString$ and this is treated as a function of x (the “bound” variable) and which can be passed around a program as an object. Because the lambda is a “closure,” it has the value of 2 for y (the “free” variable), wherever it is used. And, you can see from the type of the **val** part, that x is of type *Double* and the result is of type *String*.

* I added extra space around the = of f1 just to clearly separate the two sides.

Composing functions

- Once you can define and reference lambdas as functions, you can *compose* them!
- Just like you can compose values (with operators such as `+`, `/`, etc.), you can also compose functions (with operators such as `compose` or `andThen`*).
- Example of composition (Scala):
`val h: Int=>Double = (x => x/2.) andThen (x => x+1)`
 - Where `h` is a function such that $h(x) = 1+x/2$.
- And why is this useful?** Because, using something like Spark, invoking each function on all the partitions of a dataset over all the remote executors can be very expensive—but if we can compose two functions into one, we’ve halved the problem!

* Those are the Scala names—they have different names in other languages

Lazy evaluation

- “Lazy” is good (in a *program*, not in a *programmer*);
- Lazy is also known as “non-strict” or delayed/deferred evaluation:
 - A strict parameter or a strict variable is one which is always evaluated (when it is passed in, or when it is declared);
 - A non-strict (lazy) parameter/variable is one which is only evaluated if and when required.
- This may not seem like a big deal—but it is. It has the potential for huge improvements in performance;
- In Scala, a lazy list is called a “LazyList” (from 2.13*), and it is possible to define a *LazyList* of *all* the positive integers, for example. We can use that to filter out all non-primes, for instance, and yield the list of all primes. Of course, we can’t evaluate the whole list but we can evaluate as many as we *actually need*.
- Java 11 has some lazy features (via Streams) but not as much flexibility as Scala.

* previously, a lazy list was called a *Stream*.

Lazy evaluation (2)

- But laziness is much more important than just for building lazy lists.
- Examples:
 - when logging to debug, for example, the parameter which builds the string to be logged can be passed in lazily (“call by name”): it’s never evaluated if it’s not needed (i.e. debug logging is off);
 - when passing an expression x to $\text{Try.apply}(x)$, the x is evaluated inside the apply method and so any exception can be caught there and turned into a Failure^* .
 - In Spark, for example, an RDD (and therefore a Dataframe/Dataset) is lazy:
 $\text{rdd.map}(f).\text{map}(g).\text{collect}()$
 - is equivalent to:
 $\text{rdd.map}(f \text{ andThen } g).\text{collect}()$
 - In other words, Spark can compose the functions f and g and visit all the elements of the RDD only once!

* We will explain Try , apply , Failure , collect later

Recursion

- In an "imperative" language such as Java, the primary mechanism for doing things many times (i.e. repeating code) is *iteration*.
- In FP, the primary mechanism is *recursion*.
- Does it matter? Well, it often doesn't matter much. Other times it can make a big difference. You can't do iteration without mutable variables (think about how you might sum the elements of a list).
- Recursion is typically a more mathematical way of expressing some sort of aggregate function*:

```
def sum(xs: Seq[Int]): Int = if (xs.isEmpty) 0 else xs.head +  
sum(xs.tail)
```

* *Head* and *tail* refer to the first element of the list, and the rest.

Recursion (2)

- But isn't recursion a BAD thing?
- Recursion is like iteration but, since you are not mutating a variable each time around, you need *history*. This history is the breadcrumbs you need to find your way out of the recursion.
- The trouble is that this history takes up space: on the very limited system stack*.
- But, for many applications (most, actually), you don't really need the history and in this case you can make your recursion *tail*-recursive:

```
def factorial(n: Int) = {  
    def inner(r: Long, n: Int): Long =  
        if (n <= 1) r  
        else inner(n * r, n - 1)  
    inner(1L, n)  
}
```

@Scala.annotations.TailRec

- So, in practice, FP uses tail-recursion wherever it can and the problems of stack overflow **do not arise**.

* And when you exhaust the stack, you suffer a *StackOverflow* 😞

Recursion (3)

- There's another advantage of recursion: when you *reduce* a problem into a set of easier problems, one of the ways to do that is simply to arrange for each sub-problem to operate on a subset. This is known as "divide and conquer."
- There are two obvious ways to do this for a set (collection) of size N :
 - A. Make the sets of size 1 and $N-1$; *insertion sort*
 - B. Make the sets of size N/k where k is some integer, usually 2. *merge sort*
- The first strategy (A) naturally gives rise to iteration; the second (B) naturally gives rise to recursion.
- Algorithms which use A tend to be $O(N)$; algorithms that use B tend to be $O(\log N)$.

Immutability, pure functions, RT, zero side-effects

- A cornerstone of functional programming is that when you write $f(x)$, you expect the result always to be the same, assuming x is the same, i.e., $f(x)$ is **pure**.
 - But if f involved some other variable y that was free to mutate, then the result wouldn't always be the same.
- Functions which behave this way (the pure way) are so predictable that you can *prove* functional programs to be correct. Yes, you read that right! Imperative programs* cannot be *proven*. They can only be *tested*.
- So, pure functional programs do not allow variables to mutate or cause side-effects.
- And, if you have $\text{def } f(x) = x + 1$ (for example), the program is the *same* whether you write $f(x)$ or $x + 1$ in some expression (“substitution principle”).
- This concept is also known as referential transparency.

* By imperative, we mean the paradigm where the programmer instructs the compiler *how* to perform a task.

Immutability (2)

- So, what exactly do we mean by immutability?
 - When is a variable (or collection) mutable and when is it immutable?
- For variables, it's easy: the variable must be declared as **var** and we will see two or more *assignments*:
 - `var x = 1`
 - `x = 2`
- For collections, it's a bit harder to distinguish:
 - First, a **mutable collection** is instantiated from the `scala.collection.mutable` package, for instance:
 - `val xs = mutable.Map(1 -> "A", 2 -> "B", 3 -> "C")`
 - But, notice that `xs` is still typically a *val*, not a *var*, because it's the collection itself that can be mutated, not the reference to the collection.

Immutability (3)

- What are the mutating operators on, say, *Map*?
 - `xs += 4 -> "D"`
- So, how would we add an element to an immutable Map?
 - `val map3 = Map(1 -> "A", 2 -> "B", 3 -> "C")`
 - `map3 += 4 -> "D"` 
 - This results in a compiler error. Instead, we write something like this:
 - `val map4 = map3 + (4 -> "D")`
 - Now, we have a new identifier *map4* which has four key-value pairs, while *map3* still has only three.

Immutability (4)

- What about a *LazyList*? Are the head and tail mutable? Because they do appear to change value at some later time than their construction.
- No, lazy values are not mutable, just lazy (deferred evaluation). It's just that, when first declared, a lazy value has no value at all. Once a lazy value gets evaluated, it never changes.
- What about things like recursive methods?
 - `def factorial(x: Int) = if (x<=1) 1 else x * factorial(x-1)`
- There's nothing here that's mutable. Each time *factorial** is invoked, it has a new, immutable, value of *x*.

* But this factorial is not tail-recursive

Pattern-matching

- Pattern-matching is another big difference between functional and non-functional code:
 - You can match on constant- or variable- values;
 - You can match on types (so you don't need to dynamically cast anything);
 - You can match on the de-struction of objects (the opposite of a constructor—called an extractor).
- In functional programming, you can think of pattern-matching like a powerful, generalized, and glorified *switch* (or *if*) statement.
- But, another way to think about it is that Pattern-matching is the opposite of a variable declaration.
 - Instead of taking an expression and assigning it to an identifier...
 - Pattern-matching allows you to take an identifier and match it as an expression.

Pattern-matching: Scala examples

- Getting the contents of an optional value (Scala has a special type for this-- Java has it now, too):

```
Option(maybeX) match {  
    case Some(x) => println(x)  
    case None =>  
}
```

- Remember that *sum* method from before? How about this?

```
def sum(xs: Seq[Int]): Int = xs match {  
    case Nil => 0  
    case h :: t => h + sum(t)  
}
```

- *Nil* is simply the name of the empty list;
- *h :: t* is a pattern* that says match *xs* as the head element *h* followed by the tail *t* (i.e. the rest of the list).

```
def sum(xs: Seq[Int]): Int = {  
    @tailrec  
    def inner(xsum, xs): Int = xs match {  
        case Nil => xsum  
        case h :: t => inner(xsum + h, t)  
    }  
    inner(0, xs)
```

For List, we can use *h :: t*

* In fact, `::` is actually a case class (yeah, really) and its *unapply* method is used here.

Type inference

- This might just seem like a convenience to the programmer so you don't have to explicitly mention the type of a variable or method.
- But, actually, it's much more than that.
- If the type of the variable or method is known, type inference can be used by the compiler to determine which of several methods or variables can form part of the expression on the right of the = sign.
- And, it actually goes further than that (we will eventually cover that).

Lambda calculus: a brief discussion

- In the 1930s, Alonzo Church developed a notation and a set of simple rules that governed the way computable expressions could be defined. He called it the lambda calculus because he used the Greek symbol λ . See https://en.wikipedia.org/wiki/Lambda_calculus.
- The three rules:
 - If x is a variable then x is a valid lambda term;
 - If t is a lambda term, and x is a variable, then $\lambda t.x$ is a valid lambda term (an abstraction);
 - If t and s are lambda terms, then $(t s)$ is a valid lambda term (an application).

Lambda calculus continued

- Abstractions:
 - $\lambda x.x^2$ represents a function which takes an input (here denoted by x , but we could use any symbol we like, except λ) and which yields the square of the input. We consider x to be bound in the expression x^2 . In other words, we have no choice about x , it is provided for us by the input to the function.
 - $\lambda x.xy$ represents a function which takes an input x and which yields the product of x and y . Since y could take on any value as far as our function is concerned, we say that y is free.
- Applications:
 - $(t s)$ represents the application of function t to input s , that's to say we pass s into t .

Functional logging: an example

- Suppose you have some code like the following:

```
for (y <- ys; z <- f(y)) yield z
```

- Perhaps you'd like to log the resulting value from $f(y)$.

- To do that with normal logging, you would have to break up the “for” construct and extract a variable, log the variable, and then pass its value to the yield clause.
- Instead, you could write something like the following with a functional logger:

```
val flog = Flog[MyClass]
import flog._

for (y <- ys; z <- s"f($y): " !! f(y)) yield z
```

ys flatMap (y => f(y) map (z => z))

ys flatMap (y => f(y))

ys flatMap f

Functional logging: under the hood

```
implicit class Flogger(message: => String) extends Loggables {
    def !![X: Loggable](x: => X): X = info(x)

}

private def logLoggable[X: Loggable](function: LogFunction)(prefix: => String)(x: => X): X =
    tee[X](y => function(s"$prefix: ${implicitly[Loggable[X]].toLog(y)}"))(x)

private def tee[X](f: X => Unit)(x: => X): X = {
    lazy val xx: X = x
    Try(f(xx)) match {
        case Failure(e) => Logger[Flog].error("Exception thrown in tee function", e)
        case _ =>
    }
    xx
}
```

Summary

- Functions are just objects, like 1, “abc,” math.Pi, etc.
- They can be found as parameters (or results) of another function (or method) and they can be composed into expressions using “higher-order” functions.
- You do not have to remember anything of the previous four slides—those are just for a little more background.

2.5

More Detail on Scala Syntax

With emphasis on the functional way of
doing things

© 2019-2023 Robin Hillyard



Classes and Objects

- Classes (and Objects) in Scala follow the same principles as the Java Virtual Machine. That's to say code for a class/object is stored as bytecode is in a *.class file.
- However, at the language level (remember Java is both a language and the JVM), things are a little different.
- So, it shouldn't come as a big surprise that all useful code that is compiled and built must be in a “class.”
- But, “class” implies that there could be many instances of the class, each with different types or parameters. For Java, that makes perfect sense.
- However, Scala distinguishes between an “Object” (not parameterized) and a “Class” which can have many different instances because they are parametric in value(s) and type(s).

Objects are not parameterized

Objects are usually created with Constructors, but we use Factory Method to create Objects in Scala

accessing objects in trait, use <object name>.property

private properties are more flexible in scala

Objects

- All Scala code is defined within either an *Object* or a *Class*. But we will start with *Objects* because you always need at least one *Object* to run a Scala program:

- All *values* (including functions and instances of classes) are objects in the general (Java) sense, for example:

- "Hello, World!"
- math.PI
- 1
- **new List[Int]()**

- An *object* is a singleton (i.e. there is only one copy of it);

- But objects can be more complex than simple value: you can define your own object as follows:

- **object identifier**
- **object identifier extends supertype**
- **object identifier extends supertype { code }**
- **object identifier { code }**

we cannot extend with multiple supertypes
but we can extend with multiple super traits
similar to Java

- The *object* definition takes no parameters and doesn't need a companion class;
- An *object* has a type (which defaults to *AnyRef*)

mymodule.mytrait.scala

```
trait myTrait {  
    val x = myTrait.hello  
}
```

```
object myTrait {  
    private [myTrait] def hello = "helllo"  
}
```

Objects (2)

- An object which extends *App* invokes its initialization code within the (invisible) *main* method.
- The *main* method does not yield a value (technically, it yields *Unit*) so, unlike everywhere else in Scala, it must contain side-effects, otherwise it would do nothing..

```
object Newton extends App {  
    val newton = Newton("cos(x)-x", x => math.cos(x) - x, x => -math.sin(x) - 1)  
    newton.solve(10, 1E-10, 1) match {  
        case Success(x) => println(s"""The solution to "$newton=0" is $x""")  
        case Failure(t) => System.err.println(s"""$newton unsuccessful: ${t.getLocalizedMessage}""")  
    }  
}
```

- An object is useful for defining non-instance methods, such as *math.sqrt*.

```

def listOfDuplicates[A](x: A, length: Int): List[A] = {
  if (length < 1)
    Nil
  else
    x :: listOfDuplicates(x, length - 1)
}
println(listOfDuplicates[Int](3, 4)) // List(3, 3, 3, 3)
println(listOfDuplicates("La", 8)) // List(La, La, La, La, La, La, La, La)

```

The method `listOfDuplicates` takes a type parameter `A` and value parameters `x` and `length`. Value `x` is of type `A`. If `length < 1` we return an empty list. Otherwise we prepend `x` to the list of duplicates returned by the recursive call.

Classes

- Classes

- Like objects, classes can have initialization code (what would be in a Java constructor or within `{}`). But generally, all the useful code of a class is in its methods.
- All fields and methods of a class are *instance* fields/methods. If you want “class” fields/methods then you need to declare a companion object (one with the same name and in the same module).
- A class generally takes both value parameters and type parameters; Unless it is a “case” class, you will need to invoke the constructor using the `new` keyword.

```

case class Newton(w: String, f: Double => Double, dfbydx: Double => Double) {
  override def toString: String = w
  private def step(xy: Try[Double], yy: Try[Double]) = for (x <- xy; y <- yy) yield x - y / dfbydx(x)
  def solve(tries: Int, threshold: Double, initial: Double): Try[Double] = {
    @tailrec def inner(ry: Try[Double], n: Int): Try[Double] = {
      val yy = for (r <- ry) yield f(r)
      (for (y <- yy) yield math.abs(y) < threshold) match {
        case Success(true) => ry
        case _ =>
          if (n == 0) Failure(new Exception(s"failed to converge in $tries tries, " +
            s"starting from x=$initial and where threshold=$threshold"))
          else inner(step(ry, yy), n - 1)
      }
    }
    inner(Success(initial), tries)
  }
}

```

Constructors

- Classes define (class) constructors:
 - when you instantiate a new class, you give it a set of *values* and, if appropriate, *types* to correspond to the parameters/parametric-types of the constructor. These values are just objects. We normally give the parameters lower case identifiers. And the parameters are enclosed in parentheses:
 - for example: **class** Complex(real: Double, imag: Double)
- Similarly, types can have type constructors:
 - when you construct a concrete type, you give it a set of concrete types to correspond to the parametric types of the type constructor. These are types. We normally give the type parameters upper case identifiers. And the parameters are enclosed in square brackets:
 - for example: **trait** Ingestible[T]

Classes, Methods & Traits can be parametric
Objects , _ cannot be parametric

scala.lang._
java.lang._
predef._

Examples

```
trait EmptyTrait
object Junk extends EmptyTrait
trait EmptyTrait[T]
object Junk extends EmptyTrait[Int]
  val x = Junk // doesn't compile - check this???
object Junk[T] extends EmptyTrait[T] (doesn't compile)
val x = EmptyTrait[Int] (doesn't compile)
val x = new EmptyTrait[Int]
val f = Int => String (doesn't compile)
val f: Int => String = _.toString
```

```
trait Function1 [R, T] {
  def apply(r: R): T
}

new Function1 [Int, String] {
  def apply(r: Int): String = r.toString
}
// instead of above defined trait,
// compiler suggests us to use a
// lambda instead, using predef
{ r: Int => r: toString }
```

- Defining either an *object* or a *val* (or *var*) can only be done when there is a right-hand-side that can be evaluated to an object.

```
package foo.scala  
private [foo] def f1 = ...
```

```
// f1 method is private for the whole module foo  
// usually private is for the object
```

Modules

Best Practice:
Have only 1 trait per module
Derive classes & objects from that trait
This ensures that each file relates to different behavior

- The code in one file (module) is treated like being in its own (Java-style) package.
 - Privacy rules can apply at the module level.
 - A module may contain any number of traits, classes or objects.

```
sealed trait Foo {  
    def a: String  
    def create(a: String): Foo  
}  
  
abstract class AbstractFoo(val a :String) extends Foo
```

```
abstract class AbstractFoo(a: String) extends Foo
```

// val is not necessarily required, we would need val when:
// AbstractFoo is defined as a class
// When we want to expose a variable outside the class

```
case class Bar(a: String,b: Option[Int]) extends AbstractFoo(a) {  
    def create(y: String) = Bar(y,None)
```

}

Since Bar & Buzz are case classes,
if we don't create our own companion class, the compiler automatically gives us a companion class

```
case class Buzz(a: String, b: Boolean) extends AbstractFoo(a) {  
    def create(y: String) = Buzz(y,false)
```



Abstract methods simply lack an expression

* A sealed trait cannot be extended from a different module

Traits

- A trait defines some behavior (a bit like an interface in Java):
 - Traits have type parameters (typically) but cannot have value parameters*.
 - Methods and fields of traits may have concrete values.
 - A trait can be instantiated providing that you define all abstract properties:
 - **val s = new MyTrait {def method...}**
 - A trait which may only be extended *in-module* is marked as “sealed”.

```
sealed trait TraitExample[T] extends Comparable[TraitExample[T]] {
    def name: String
    def property: T
    def >(o: TraitExample[T]): Boolean = compareTo(o)>0
    def <(o: TraitExample[T]): Boolean = compareTo(o)<0
    def >=(o: TraitExample[T]): Boolean = compareTo(o)>=0
    def <=(o: TraitExample[T]): Boolean = compareTo(o)<=0
    def ==(o: TraitExample[T]): Boolean = compareTo(o)==0
}
abstract class Base[T: Ordering] extends TraitExample[T] {
    def compareTo(o: TraitExample[T]) = implicitly[Ordering[T]].compare(property, o.property)
} with the use of implicitly, compiler adds () so its methods, here, Ordering methods
case class Telephone(name: String, number: String) extends Base[String] { such as .compareTo() can be used
    def property: String = number
} For Base[String], compiler would have implemented Ordering[String]
case class Age(name: String, age: Int) extends Base[Int] {
    def property: Int = age For Base[Int], compiler would have implemented Ordering[Int]
}
```

* Except in Scala 3

file 1.scala

package foo

object bar { ... }

file 2.scala

import foo.Bar

val x: Bar = { ... }

Expressions

- So, now we know where we can write code, what sort of code can we write?
- Basically, we write expressions:
 - An expression yields a result (of any type, including “Unit”, a non-result);
 - An expression can be preceded by definitions of “memoizing” variables (i.e., “extraction” in refactoring terminology);  val circum= 2 * pi * r
Can be written as:
// diameter gives better
understanding
val diameter = 2 * r
val circum = d * pi
 - An expression can be preceded or followed by definitions of methods; ex: lazy
 - An expression can be preceded by *import* statement(s) which allow us essentially to create aliases of types; imports are best way to introduce implicits to our local context,
see left corner
 - An expression is a series of identifiers/literals/method invocations interspersed with operators;
 - When a method invocation takes parameters, the values of those parameters will also be expressions (maybe just identifiers).

Variable definitions

- The following are examples of variable definitions:
 - `val x = Math.PI or`
 - `val x: Double = Math.PI`
 - `val x = Math.PI/2 + 1`
 - `var x = 0` it is mutable, though it is not recommended for functional programming
 - `lazy val x = connection.get("date")`

`val x = ...` → evaluated 1 time

`lazy val x = ...` → evaluated 0 or 1 time

`def x = { ... }` → evaluated every time x is called

Variable definitions with patterns

- When we wrote:
 - `val x: Double = Math.Pi`
- We used a very simple pattern as the identifier (`x`) and that pattern is now in scope for the remainder of the current context, but:
 - We can do more generalized pattern-matching as in, for example:
 - `val h :: t = List(1,2,3)`
 - `println h`
 - You should see “1” printed on the console because `h` matches 1 and `t` matches `List(2, 3)`
 - What if the expression on the right isn’t a list at all?
 - The compiler will warn you.
 - What if it’s an empty list (but that’s not known until run-time)?
 - Then it will throw a *MatchError* at run-time. basically, you should be quite sure that these patterns in “`val`” definitions are really what they are expected to be.

Variable definitions with patterns (2)

- We can use very general complex patterns of the form:
 - `val pattern = expression`
 - For example (the first one is a very common situation*):
 - `val (odd, even) = xs.partition(_ % 2 == 1)`
 - `println(s"odd: $odd, even: $even")`
 - `val Some(x) = methodWithOptionalResult()`
 - `println x`
 - In the second example, what if the result of calling the method is `None`?
 - In that case, you will get a *MatchError* at run-time.
 - But, if the compiler knows that the pattern won't match, it will let you know at compile time. Again, it's best to be sure when you use these patterns. This one is probably a bad idea.

* Partition puts all elements evaluating to true in the first sequence and the rest in the second sequence.

Method definitions

- The following are examples of method definitions:

- def x = Math.PI
- def x: Double = Math.PI/2 + 1
- def x(s: String) = connection.get(s)
- def x(s: String) = {

val connection = makeConnection("myServer")

val r = connection.get(s)

connection.close()

r

}

We can also write it as:

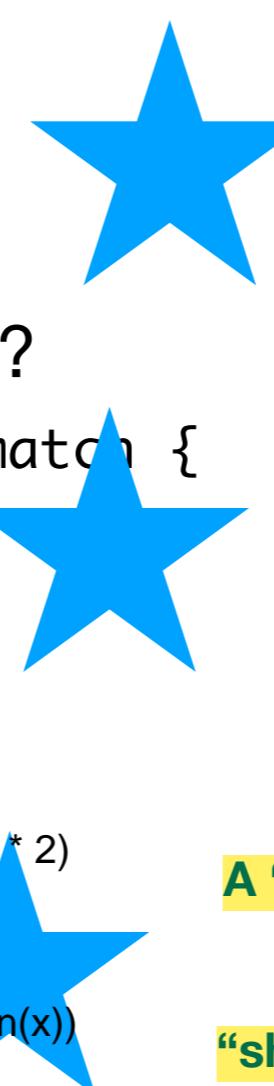
```
using(makeConnection("myServer")) {  
    connection => connection.get(s)  
}
```

// here with the help of using, after its
block, runtime ensures connection is
closed

- The following can also be used when there are no parameters and no side-effects:

- lazy val x = Math.PI

Control flow*?

- OK, that's great but what about control flow?
 - Well, in a functional programming language, we define **expressions**, we don't put together a series of statements interspersed with control flows.
 - But what about a simple *if*?
 - `if (x>=0) x else -x`
 - `if (x>=0) println(x)`
 - And what about some kind of switch?
 - `def length(xs: Seq[X]): Int = xs match { case Nil => 0 case _ :: t => length(t) + 1 }`
 - And what about some kind of loop?
 - `for (x <- xs) yield x * 2`
 - `for (x <- xs) println(x)`
 - `xs foreach println`
- 
- An “if” expression should always have an “else”.
The Unit form of if really should never be used!
- This is called pattern-matching and is *much* more powerful than a switch statement in Java
- A “for comprehension” with “yield” always returns a result of the same “shape” as its generator (xs)

* Refresher: we covered this earlier in 2.0

Updated: 2021-09-27

2.6 Scala in context

A little history, a little general
description

© 2016 Robin Hillyard



What is Scala?

- Scala
 - stands for *Scalable Language*
 - is a “blend of **object-oriented** and **functional programming** concepts in a **statically typed** language”
statically typed means compiler beforehand knows the type for each object
 - is a general-purpose language
python is a dynamically typed language,
type is assigned during runtime ?
 - runs on the JVM (Java Virtual Machine)* via “bytecode,” just the same as Java, Groovy, Clojure, Kotlin, etc.
 - is bi-directionally compatible with Java (and other JVM languages). Although Scala has its own definitions for things such as *List*, you can, if you like, use *java.util.List*

* Note that *Scala.js* can also compile to Javascript in your browser

Very quick History of Scala

- Design began in 2001 by Martin Odersky at EPFL
 - Many ideas from existing languages contributed
- First public release 2004
- Version 2.0 in 2006
- Typesafe (now Lightbend) launched in 2011
- Scala.js version 0.1 released November 2013
- 2.11 released April 2014
- 2.12 released November 2016 (mainly internal improvements)
- 2.13 released 2019.
- Scala 3 (code name Dotty) is released (latest is 3.0.2)

Code Example – interoperability with Java

```
package edu.neu.coe.scala
case class Mailer(server: String) {
    import java.net._
    val s = new Socket(InetAddress.getByName(server), 587)
    val out = new java.io.PrintStream(s.getOutputStream)
    def doMail(message: String, filename: String) = {
        val src = scala.io.Source.fromFile(filename)
        for (entry <- src.getLines.map(_.split(","))) out.println(s"To: ${entry(0)}\nDear
${entry(1)},\n$message")
        src.close
        out.flush()
    }
    def close() = {
        out.close()
        s.close()
    }
}
object EmailApp {
    def main(args: Array[String]): Unit = {
        val mailer = new Mailer("smtp.google.com")
        mailer.doMail(args(0), "mailinglist.csv")
        mailer.close
    }
}
```

Notice that we can place imports wherever needed.

Notice how we are including some Java types here, although that is unusual.

Notice how easy it is to print strings with the values of variables.

Notice that there are no semi-colons

Notice that Scala uses [] for types, not <>

Here we create a main program which is just like one in Java, except here it is not marked “static”. There is a better way, however.

Another code example

```
object Ratios {  
    /**  
     * Method to calculate the Sharpe ratio, given some history and a constant risk-free rate  
     * @param investmentHistory the history of prices of the investment, one entry per period, starting with the most recent price  
     * @param periodsPerYear the number of periods per year  
     * @param riskFreeRate the risk-free rate, each period (annualized)  
     * @return the Sharpe ratio  
    */  
    def sharpeRatio(investmentHistory: Seq[Double], periodsPerYear: Int, riskFreeRate: Double): Double =  
        sharpeRatio(investmentHistory, periodsPerYear, LazyList.continually(riskFreeRate))  
    /**  
     * Method to calculate the Sharpe ratio, given some history and a set of benchmark returns  
     * @param investmentHistory the history of prices of the investment, one entry per period, starting with the most recent price  
     * @param periodsPerYear the number of periods per year  
     * @param riskFreeReturns the actual (annualized) returns on the risk-free benchmark  
     * @return the Sharpe ratio  
    */  
    def sharpeRatio(investmentHistory: Seq[Double], periodsPerYear: Int, riskFreeReturns: Stream[Double]): Double = {  
        // calculate the net gains of the investment  
        val xs = for (x <- investmentHistory.sliding(2)) yield x.head - x.last  
        // calculate the net returns on the investment  
        val ys = (xs zip investmentHistory.iterator) map { case (x, p) => x / p }  
        // calculate the annualized returns  
        val rs = for (y <- ys) yield math.pow(1 + y, periodsPerYear) - 1  
        // calculate the Sharpe ratio  
        sharpeRatio(rs.toSeq, riskFreeReturns)  
    }  
    /**  
     * Method to calculate the Sharpe ratio, given actual and benchmark returns  
     * @param xs the actual historical returns (expressed as a fraction, plus or minus) of the given investment  
     * @param rs the actual historical returns (expressed as a fraction, plus or minus) of the benchmark  
     * @return the Sharpe ratio  
    */  
    def sharpeRatio(xs: Seq[Double], rs: Stream[Double]): Double = {  
        val count = xs.size  
        // calculate the excess returns  
        val zs = (xs zip rs) map { case (r, f) => r - f }  
        // calculate the Sharpe ratio  
        zs.sum / count / math.sqrt((xs map (r => r * r)).sum / count)  
    }  
}
```

Notice that wherever we define a variable or parameter,
it's **always** of form *name : Type*

Look at these *for* loops. Each of them
returns a value via the *yield* keyword

Notice that “variables” must be preceded by “val” or “var”
but don’t necessarily need a type annotation.

Notice that we don’t need a *return* keyword.

Newton's Approximation (FORTRAN)

```
program newton
    Print *, "Newton's Approximation"
    x = 1.0
    maxTries = 100
10   y = cos(x) - x
        if (abs(y)<1E-7) goto 20
        x = x + y / (sin(x) + 1)
        maxTries = maxTries - 1
        if (maxTries>0) goto 10
    Print *, "failed"
    goto 30
20   Print *, x
30   Print *, "Goodbye"
end program newton
```

Newton's Approximation (Java8)

```

import java.util.function.DoubleFunction;
public class Newton {
    public Newton(final String equation, final DoubleFunction<Double> f, final DoubleFunction<Double> dfbydx) {
        this.equation = equation;
        this.f = f;
        this.dfbydx = dfbydx;
    }
    public Either<String, Double> solve(final double x0, final int maxTries, final double tolerance) {
        double x = x0;
        int tries = maxTries;
        for (; tries > 0; tries--) {
            try {
                final double y = f.apply(x);
                if (Math.abs(y) < tolerance) return Either.right(x);
                x = x - y / dfbydx.apply(x);
            } catch (Exception e) {
                return Either.left("Exception thrown solving " + equation + "=0, given x0=" + x0 + ", maxTries=" + maxTries + ", and tolerance=" +
tolerance + " because" + e.getLocalizedMessage());
            }
        }
        return Either.left(equation + "=0 did not converge given x0=" + x0 + ", maxTries=" + maxTries + ", and tolerance=" + tolerance);
    }
    public static void main(String[] args) {
        Newton newton = new Newton("cos(x) - x", (double x) -> Math.cos(x) - x, (double x) -> -Math.sin(x) - 1);
        Either<String, Double> result = newton.solve(1.0, 200, 1E-7);
        result.apply(
            System.err::println,
            aDouble -> {
                System.out.println("Good news! " + newton.equation + " was solved: " + aDouble);
            });
    }
    private final String equation;
    private final DoubleFunction<Double> f;
    private final DoubleFunction<Double> dfbydx;
}

```

Newton (Scala)

```
package edu.neu.coe.csye7200
import scala.annotation.tailrec
import scala.util._

case class Newton(f: Double => Double, dfbydx: Double => Double) {
    private def step(x: Double, y: Double) = x - y / dfbydx(x)
    def solve(tries: Int, threshold: Double, initial: Double): Try[Double] = {
        @tailrec def inner(r: Double, n: Int): Try[Double] = {
            val y = f(r)
            if (math.abs(y) < threshold) Success(r)
            else if (n == 0) Failure(new Exception("failed to converge"))
            else inner(step(r, y), n - 1)
        }
        inner(initial, tries)
    }
}

object Newton extends App {
    val newton = Newton(x => math.cos(x) - x, x => -math.sin(x) - 1)
    newton.solve(10, 1E-10, 1.0) match {
        case Success(x) => println(s"the solution to math.cos(x) - x is $x")
        case Failure(t) => System.err.println(t.getLocalizedMessage)
    }
}
```

A case class is a class but with fields that are visible both for construction and extraction; it also comes with other useful methods.

the result of *inner* and so of *solve* is a *Try[Double]*. This is a bit like the *Either* of the Java version, but it has value which is either a Double (*Success*) or an Exception (*Failure*).

Lambdas define f and dfbydx

case Success(x) => println(s"the solution to math.cos(x) - x is \$x")
case Failure(t) => System.err.println(t.getLocalizedMessage)

Another Example: Ingest

```
package edu.neu.coe.scala.ingest

import scala.io.Source

class Ingest[T : Ingestible] extends (Source => Iterator[T]) {
  def apply(source: Source): Iterator[T] = source.getLines.map(e =>
  implicitly[Ingestible[T]].fromStrings(e.split(",").toSeq))
}

trait Ingestible[X] {
  def fromStrings(ws: Seq[String]): X
}

case class Movie(properties: Seq[String])

object Ingest extends App {
  trait IngestibleMovie extends Ingestible[Movie] {
    def fromStrings(ws: Seq[String]): Movie = Movie.apply(ws)
  }
  implicit object IngestibleMovie extends IngestibleMovie

  val ingestor = new Ingest[Movie]()
  if (args.length > 0) {
    val source = Source.fromFile(args.head)
    for (m <- ingestor(source)) println(m.properties.mkString(","))
    source.close()
  }
}
```

This example of Scala uses some advanced features of the language, including *type classes* and *implicits* to allow us a truly generic solution to ingestion. Don't expect to understand it just yet.

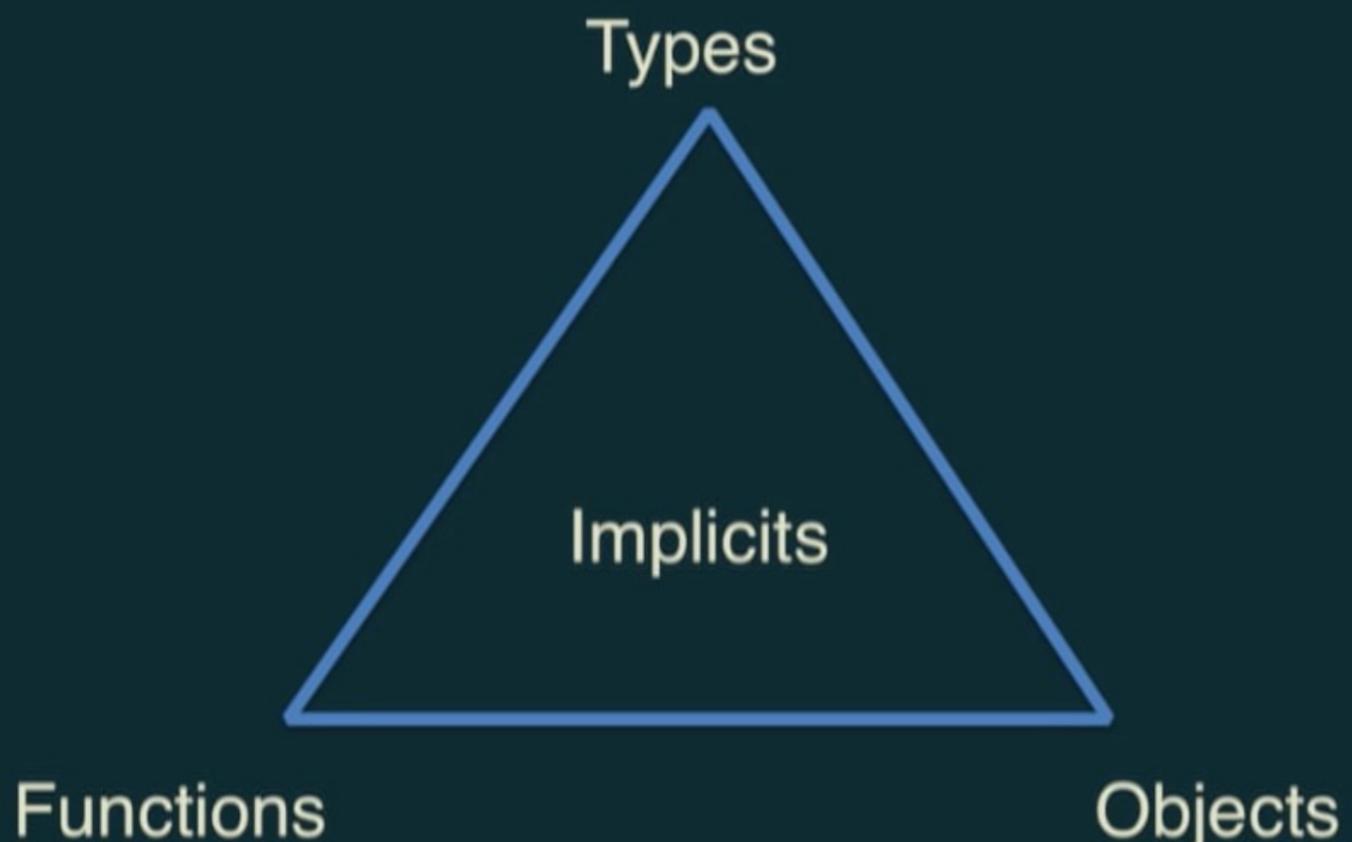
What Scala is not?

- Scala *is not*
 - a “scripting language,” although it can be and is used in many places scripts might be found, such as in the *build.sbt* file (used to build Scala applications)
 - a domain-specific-language (DSL) although it can be used to build DSLs
 - too esoteric for people to learn!

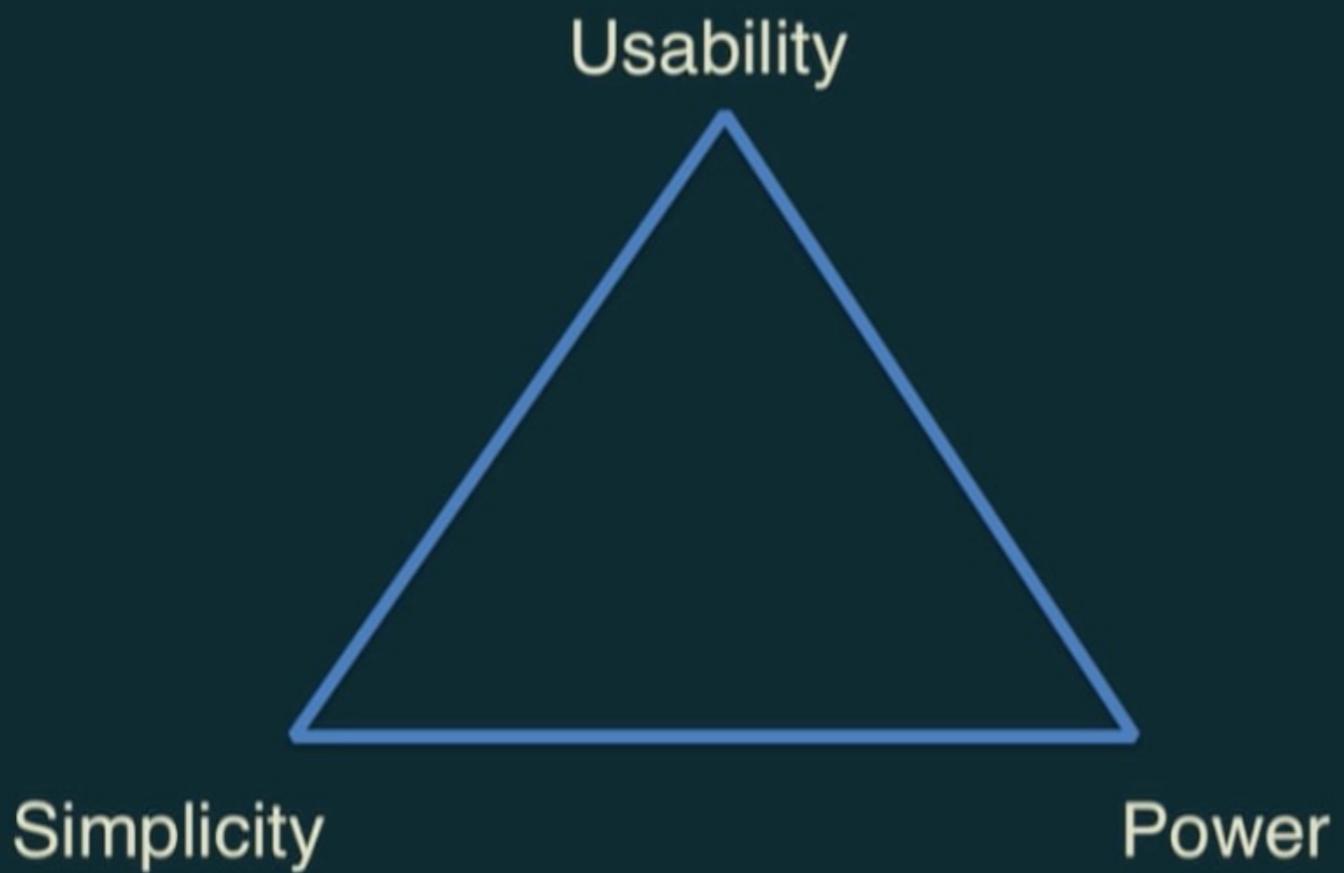
Functional vs. Procedural

| | Functional | Procedural |
|-----------|---|---|
| Paradigm | Declarative (you describe/declare problem) | Imperative (you explicitly specify the steps to be taken to solve problem) |
| Functions | First-class objects, composable | “callbacks”, anonymous functions, etc. |
| Style | Recursive, mapping, pattern matching | Iterative, loops, if... |
| Examples | Lisp, Haskell, Scala... | Almost all other mainstream languages, including Java* *Java8 has many FP concepts |

Scalastic Principles



Scalastic Pragmatics



Why Scala?

- Scala:
 - is fun (!) and very elegant;
 - is mathematically grounded;
 - is extensible;
 - is powerful (many built-in features);
 - has syntactic sugar: sometimes you can express something in different ways—however, usually, one form is “syntactic sugar” for the other form and the two expressions are entirely equivalent;
 - is predictable: once your program is compiling without warnings, it *usually* does what you expect;
 - is ideal for reactive programming (concurrency);
 - is ideal for parallel programming (map/reduce, etc.);
 - used by Twitter, LinkedIn, Foursquare, Netflix, Tumblr, Databricks (Spark), etc.

Is Scala really an important language today?

- Scala is presently in 31st position in the TIOBE index (<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>)
 - Up two places since September 2015
- In 17th place (more or less stable) on the PYPL index (<http://pypl.github.io/PYPL.html>)-up one places in five years.
- Ratios of Java : Scala
 - Google search
 - 10:1 in general (“xxx programming”)—was 20:1 five years ago.
 - “stackoverflow.com” in last two years
 - 15:1 (but Scala is growing relative to Java—it was 22 the first time I checked)

Why not Python?

- Python is a very popular language which is used by programmers in many disciplines, including Data Science.
- Python is interpreted, whereas Scala is compiled:
 - What difference does this make in practice? Compiled is better if you are running in a production environment; interpreted is better if you are in research/data science mode.
- Scala is strictly typed whereas Python is dynamically typed:
 - In practice, this means that you can probably write Python more quickly (less time spent trying to keep the compiler/interpreter happy) but strict typing helps ensure your code does what you expect and won't crash!

References: [Quora](#); [6 points](#); [Argument against Python](#)

My thoughts on Python

- I can't take seriously a language that requires specific patterns of white space* in its source code!
- Python can't decide whether it's object-oriented or not.
 - For example, to get the length of a string s , you write:
 - $\text{len}(s)$
 - But to find the index of the first occurrence of a substring t in string s , you write:
 - $s.\text{find}(t)$
- I hate inconsistencies like that!

* it's true that Scala also pays attention to newlines and indents but it doesn't *rely* on them.

How do we use Scala? (review)

- Very briefly, there are at least *five** ways:
 - Create a module (file) called **`xxx.scala`** and compile it, build it, run it (using shell, **`sbt`**, or IDE such as IntelliJ);
 - The “REPL” (*read-execute-print-loop*) (using shell commands `scala` or `sbt console`)—or IDE—**the REPL is your friend**;
 - Compile it in-browser with, e.g. <https://scalafiddle.io> or <https://scastie.scala-lang.org>;
 - Create a “worksheet” called **`xxx.sc`** and save it (IDE only);
 - Use a *REPL-with-Notebook* such as Zeppelin.
Zeppelin is similar to Jupyter Notebook, or Databricks
- Two popular IDEs: ***Scala-IDE*** (based on *Eclipse*) and ***IntelliJ-IDEA*** (what I recommend).

* You can also dynamically inject code using Twitter’s “eval” utility

Object-oriented programming

- The central concept of O-O is that methods (operations, functions, procedures, etc.) are invoked ***on objects*** (as opposed to local/global data structures in memory).
- A class is defined as the set of methods that can be invoked on its instances and the set of properties that these instances exhibit (aka fields).
- Objects are instances of a ***class*** and contain both methods and fields (properties).
- Classes form an inheritance hierarchy such that sub-classes inherit and/or override methods and fields from their super-classes.
- Examples: Smalltalk, Scala, Java, C++, C#, Objective-C, Python, PHP5, Ruby.

def vs. *val/var*

- The obvious way to think of the difference between *def* and *val* is to say: *def* declares a method (function) and *val* declares a variable.
- But there's a bit more to it:
 - *def* is (usually) parameterized and will have a different value for each combination of parameter(s), thus:
 - *def f(x: double) = ??? // something having to do with x*
 - but even with zero parameters, *def* is evaluated every time it is referenced, so
 - *def x = y is evaluated every time x is referenced, even though y may not have changed.*
 - *but a variable definition:*
 - *val x = y*
 - is evaluated once and once only
 - Later, we will learn about *lazy val* and *var*.

Functional Programming Cont'd

- Functional Programming:
 - Functions are first-class objects:
 - Is a function ***data or program?***
 - Actually, it's both: it's all *relative*:
 - If you are *creating* the function, think of it as *program*, but another function which receives it as a parameter sees it as an object (i.e. *data*.)
 - functions, treated as data, can be *composed* and *serialized*.
 - Avoids side-effects—variables, mutable collections, even I/O are the ***exception*** rather than the rule in the vast majority of Scala code:
 - functions **may not** change their input parameters; this, in turn:
 - leads to *referential transparency* (provable via substitution—Turing complete—“ λ -calculus”)
 - and to logic—i.e. predictability (and, thus, testability)
 - that's to say $f(x) == f(x)$
 - and so is inherently **scalable and parallelizable**

It's a bit like a photon: wave?
particle? or both?



A brief diversion...

- What's so special about $f(x) == f(x)$?
 - Wouldn't we always expect that to be true?

```
Robins-MacBook-Pro:LaScala scalaprof$ date
Sat Jan  7 11:17:58 EST 2017
Robins-MacBook-Pro:LaScala scalaprof$ date
Sat Jan  7 11:18:01 EST 2017
```

- Well, here we invoked the same function twice but it didn't give the same answer each time. WUWT? What's up with that?
- In this case, the date function is not *pure**. date depends on clock
- But, in general, functions defined in a functional programming language *are* pure.

* that's because it depends on something *external*; see <https://stackoverflow.com>

A brief diversion contd.

- Take a look at this Java program:

```
public class Purity {  
    class MyInteger {  
        int i;  
        public MyInteger(int x) { i = x; }  
        public int getInt() { return i; }  
        public void setInt(int x) { i = x; }  
    }  
    public int getNext(MyInteger x) {  
        int x1 = x.getInt() + 1;  
        x.setInt(x1);  
        return x1;  
    }  
    public int getNext(Integer x) throws Exception {  
        int result = x.intValue() + 1;  
        Field f = x.getClass().getDeclaredField("value");  
        f.set(x, result);  
        return result;  
    }  
    public int getNext(int x) {  
        x = x + 1;  
        return x;  
    }  
    public static void main(String[] args) {  
        Purity n = new Purity();  
        int i1 = 0;  
        testPurity(n.getNext(i1) == n.getNext(i1), "getNext(int)");  
        MyInteger i2 = n.new MyInteger(0);  
        testPurity(n.getNext(i2) == n.getNext(i2), "getNext(MyInteger)");  
        Integer i3 = 0;  
        try { testPurity(n.getNext(i3) == n.getNext(i3), "getNext(Integer)); } catch (Exception e) { e.printStackTrace();}  
    }  
    public static void testPurity(boolean b, String message) {  
        if (b) System.out.println(message + " is pure");  
        else System.out.println(message + " is not pure");  
    }  
}
```

What's happening here? Is *x* updated or what?

Or here?

What about here?

It's awkward outputting the message in the exceptional case because Java doesn't support call-by-name

A brief diversion—result

- Here, we annotate the program with comments on the result:

```
public class Purity {  
    class MyInteger {  
        int i;  
        public MyInteger(int x) { i = x; }  
        public int getInt() { return i; }  
        public void setInt(int x) { i = x; }  
    }  
    public int getNext(MyInteger x) {  
        int x1 = x.getInt() + 1;  
        x.setInt(x1);  
        return x1;  
    }  
    public int getNext(Integer x) throws Exception {  
        int result = x.intValue() + 1;  
        Field f = x.getClass().getDeclaredField("value");  
        f.set(x, result);  
        return result;  
    }  
    public int getNext(int x) {  
        x = x + 1;  
        return x;  
    }  
    public static void main(String[] args) {  
        Purity n = new Purity();  
        int i1 = 0;  
        testPurity(n.getNext(i1) == n.getNext(i1), "getNext(int)");  
        MyInteger i2 = n.new MyInteger(0);  
        testPurity(n.getNext(i2) == n.getNext(i2), "getNext(MyInteger)");  
        Integer i3 = 0;  
        try { testPurity(n.getNext(i3) == n.getNext(i3), "getNext(Integer)); } catch (Exception e) { e.printStackTrace(); }  
    }  
    public static void testPurity(boolean b, String message) {  
        if (b) System.out.println(message + " is pure");  
        else System.out.println(message + " is not pure");  
    }  
}
```

int does not get updated
MyInteger gets updated
This (using Integer) throws an exception

A brief diversion Contd. (2)

- We got three different results with this Java program:
 - *getNext(int)* is pure;
 - *getNext(MyInteger)* is not pure;
 - *getNext(Integer)* results in: *java.lang.IllegalAccessException*: Class *edu.neu.coe.scala.Purity* can not access a member of class *java.lang.Integer* with modifiers "private final" at *sun.reflect.Reflection.ensureMemberAccess(Reflection.java:101)*
- In other words, you cannot guarantee the truth of the test $f(x) == f(x)$ in Java (or C++, etc.)
- **But in functional programming languages, such as *Scala*, you can!**

A brief diversion from the brief diversion

- Why does all this matter?
- What's the future of computers? Can we continue with Moore's law?
 - No, we can't! Because the speed of light is too slow, amongst other physical limitations.
 - On my computer, in one clock cycle (0.38 ns), light can travel about 115 mm — that's only about three times the size of the processor!
 - If we speed the processor up a bit, electric field fluctuations won't have time to get from one end of the chip to the other!

Continuing the brief diversion from the brief diversion

- So, to make computers run faster is going to cost many \$\$.
But there's another way:
 - Instead of speeding up *one* computer by 10x, why not have 10 computers? or 100? or 1000?
 - But we *must have* the following:
 - the ability to pass a function over the network to a different computer, treating the function itself as *data* (requires serialization);
 - the computing environment on the other computer must be identical (that's where the JVM comes in);
 - and, of course, we need the result of invoking that function multiple times (on any computer) to be identical each time, i.e. $f(x) == f(x)$.
- Now, I want you to just think about this and its implications.

Typed functional programming

- Typing adds a new dimension to FP
 - ML, Miranda, Haskell, Scala, Clojure, etc. are all typed
 - Variables have types; types have kinds; kinds have...?
- Typing eliminates* “casting” errors at run-time

* well, Scala does allow a certain amount of circumventing strict typing—mainly for compatibility with Java—so it’s still possible (but rare) to run into class cast errors.

Quick introduction to tuples

- Language types:
 - All languages have types such as Int, Double, String, etc.
 - And languages such as Java allow you to create your own types where you can have any number of fields, each potentially of a different type. However, that requires a lot of “boiler-plate” code.
 - But this sort of thing is so common that Scala (and Python) allow you to create these types on-the-fly with no special boilerplate code. You don't even have to declare the type. It's called a “tuple” and it corresponds precisely to the kind of information contained in a row of a database table (where it's also called a tuple).
- Examples:
 - (1, “One”, true)
 - 1 -> “One”

Combining O-O and FP

- In a pure functional language we *only* have functions:

val g = f(x)

- But, recall that *x* can be anything, including a tuple (a group of disparate things). So, if *x* happened to be a tuple of *a*, *b*, and *c*. Then *val g = f((a, b, c))* which we can rewrite as *val g = f(a, b, c)*.
- We call these parameters *a*, *b*, and *c* a “parameter set” although as you can see, they really are just one parameter, functionally. However, most languages allow these comma-separated parameters so Scala does too.
- We can have more than one parameter (in the functional sense), eg. *val z = f(x)(y)*
 - The interpretation of this is that *f(x)* yields a function *g* and therefore *z = g(y)*.

Combining O-O and FP (contd.)

- So, a Scala function invocation can have any number of these “parameter sets.”

val z = f(x)(a,b,c)(d)(e)...

- Which we can re-write as:

val z = g(a,b,c)(d)(e)...

- where *g* is the function yielded by *f(x)*

Combining O-O and FP (contd.)

- Methods
 - If we write, in O-O notation, the following expression:
 $\text{val } z = x.m(a,b,c)$
 - We know that this means invoke method m on object x and pass parameters a , b , and c .
 - But if we simply rewrite $x.m$ as $f(x)$
 - Then we can interpret $f(x)$ as the function that we obtain when we apply the method m to object x .
 - So that:
 $\text{val } z = f(x)(a,b,c)$ curried form of, val z = d(x)=f(a,b,c, d,f)
 - In the simple case where the right-hand parameter set is just one parameter we can write it either as:
 $x.m(a)$ or
 $x \ m \ a$ *
 - Now, you can see how methods and functions are, more-or-less, the same thing. More on this later.

One more thing...

- Remember the slide with Martin Odersky? He mentioned *implicits*. Is this a big deal? Is it as important as: FP+OO+Types+JVM?
- Not quite. But it's really important:
 - Configuration: *implicits* enable library classes to behave differently without having to pile on extra parameters all the time;
 - generalizes the concept of “widening”;
 - is as powerful and sweeping as allowing objects as the (implicit) first parameter of a function. In fact, any *implicits* are always the *last* parameter of a function.

2.7 Scala, O-O and Java: Part 1

© 2019 Robin Hillyard



How does O-O mix with FP?*

- In O-O/FP language (like Scala):
 - Classes have fields (just like Java, say) but these fields can represent any object (including *functions*):
 - We have a special way of defining *function* fields: we can define them using *methods* (similar to Java) with a special declaration syntax;
 - Incidentally, we can also just have a value field which represents a function.
 - The syntax for method declarations is a mix of a *function* has 0..N parameter sets—each enclosed in “()”—see next slide for detail;
 - a *method* has, additionally, “this” as a *parameter*; and, by convention, “this” is invisibly *prefixed* to the method name (other parameter sets *follow* the method name)—(“infix” notation)—standard O-O stuff;
 - example method *apply* on an indexed sequence (with one parameter set) can be invoked thus (all equivalent):
 - `this apply x`
 - `this.apply(x)`
 - `this(x)`
 - `apply(x)`
 - `(x)` // yes, this really works.

all function types - have apply

* These three slides are very similar to end of 2.6

Functions and methods

Functions in Scala are objects that extend FunctionN.
Lambda expressions are just syntactic sugar for FunctionN.
Scala supports up to Function22 for functions with 22 parameters.
Methods (def) are not functions until converted using _.
FunctionN enables functional programming in a type-safe way.
FunctionN allows higher-order functions (functions that take functions as parameters).
FunctionN allows for function composition (f andThen g, f compose g).

- Functions follow the form of mathematical functions:
 - $f(x, y)$
 - $f(x)(y)$
- These two forms are essentially the same in terms of the result. But, mathematically, they are different. We will talk about this in more detail later.
- But, recall that x and y can represent any object, including a “tuple”. So, if $x = (i, j)$ and $y = (k, l)$ then we can write our function as $f((i, j))((k, l))$. Scala simplifies this to $f(i, j)(k, l)$.
- Bottom line is that a method in Scala can have any number of “parameter sets”:
 - `def f(i, j)(k, l)... = ...`

What's imperative programming?

- Imperative programming comes from an understanding of the hardware architecture of a computer, whether a Turing machine*, a Von-Neumann machine**, or whatever:
 - The notion is that you have a block of addressable memory and at least one register (the accumulator).
 - The program, as well as the data, is stored in this memory
 - another register (the program counter—PC) points to the current instruction;
 - the system steps through the program by changing the value of the PC.
 - To *use* a value in your program you must:
 - find its address;
 - issue a fetch (or load) statement to get the value into the accumulator;
 - perform some arithmetic (or store) operation on it.
 - To *store* a value from the accumulator into memory you must:
 - determine an appropriate address.
 - issue a store statement to that address.
- All programs written in imperative style *ultimately* perform these operations.

* A [Turing machine](#) is a hypothetical computing apparatus; ** [The Von Neumann architecture](#) is a more concrete version.

Other differences

- Pointers:
 - In procedural programming, pointers are used everywhere (although good O-O style masks use of some pointers by *this*, etc.)
 - Unfortunately pointers which are mutable can be null—result: *NullPointerException* (NPE).
 - Scala allows *null* for compatibility with Java, but you should *never* use it! (There are better ways)
- Types:
 - With non-strict typing (e.g. as in Java), generic types are used
 - Unfortunately, sometimes these are wrong—result: *ClassCastException*.
 - **Scala has *strict* typing**

Scala vs. Java 8/11/14

- There are four concepts that are common to both languages:
 - Lambdas (anonymous functions)
 - Methods in traits/interfaces
 - “Stream” operations on collections
 - SAM (single-abstract-method) types (used for all *FunctionN*)
- Otherwise, Scala and Java 8 are completely different: Java 8 lacks most of the other functional programming “goodies” that we will learn about.
- However, Java 11 and Java 14 have quite a few more functional “goodies.”

Java 17:

records in Java => case classes in Scala

Scala, Java & the JVM

- Being on the JVM is one of the **great strengths** of Scala.
 - It opened up the entire Hadoop/BigData/Spark world to Scala:
 - There's no question in my mind that, otherwise, the language would never have "taken off";
 - But... The cosy relationship between Java and Scala comes with some "baggage":
 - exceptions, nulls, auto-boxing, mutable arrays, erasure, even O-O in the eyes of purists, etc.
 - "pure" *fp* languages like Haskell do not suffer from these issues so Scala does its best to work around them.
- Auto-boxing is the automatic conversion between primitive types (like Int, Double) and their corresponding reference types (java.lang.Integer, java.lang.Double) when interacting with Java APIs or generic types. Since Scala runs on the JVM, it must sometimes convert between Scala primitive types and Java boxed types automatically.
- Type ERASURE is a process where generic type parameters are removed (erased) at runtime by the JVM. This means Scala's type system exists at compile-time, but generic type information is not available at runtime.

Declarative Programming

- If you go back in history to the dawn of programming* you will find that (almost) all programs** were written in the imperative style***:

```
package edu.neu.coe.scala;
public class NewtonApproximation {
    public static void main(String[] args) {
        // Newton's Approximation to solve cos(x) = x
        double x = 1.0;
        int tries = 200;
        for (; tries > 0; tries--) {
            final double y = Math.cos(x)-x;
            if (Math.abs(y)<1E-7) {
                System.out.println("the solution to cos(x)=x is: "+x);
                System.exit(0);
            }
            x = x + y/(Math.sin(x)+1);
        }
    }
}
```

* actually, I do go back almost that far ;)

** does this language look familiar? It should—it's Java.

*** you could also call this the “Von Neumann” style (search for “Backus Turing Award”)

Observations on Newton-Raphson Approximation

- This is written in Java but the style is similar to the original Fortran (Backus et al)—the loop would have been a GOTO originally and tries would have been called “I” or “N”
- If you’re unsure about the method, see:
 - [http://en.wikipedia.org/wiki/Newton's method](http://en.wikipedia.org/wiki/Newton's_method)
- Everything is static (there is no O-O here, as you’d expect)
- There’s not just one but *two* mutable variables (*x* and *tries*)
- The program tells the system *exactly* how to run the calculation
 - there’s a loop until some terminating condition is met;
 - meanwhile, the best estimate (*x*) is explicitly updated.

x

So, what's wrong with the imperative/declarative style?

- In the beginning a program was loaded into memory (from magnetic/paper tape or punched cards) and ran until it was finished (or “abend”ed).
- As computers became more shareable, used disks, had other devices attached, the idea of interrupts was born:
 - An interrupt put the current program on hold and started running a different program for a while—when finished it would go back to the original program, once started, is didn't have any break, there was no interrupts
 - The less tolerant of delay was the device, the higher the priority of the interrupt (a card or paper tape reader was intolerant, a disk a little more tolerant).
- This worked fine until networks and shared databases came along (early 70s)—even then it worked OK until the sheer volume and frequency of network interrupts—and the number of database users got too high.
- This problem was “solved” by inventing “threads” (sub-processes)—and using a programming language that explicitly allows threads to be programmed at a low level, e.g. Java.

The straw that broke the camel's back

- Have you ever tried to develop or, more importantly, *test* a threaded application?
- I have—it can be maddeningly frustrating.
- Typical “solutions”:
 - synchronize mutable state—but be careful not to synchronize too much at once lest you run into race conditions and deadlocks
 - custom Executor services and a greater number of processors (make the problem “go away”—for a while)
 - defensive deep copying (Aaargh!)

2.8 Objects, types and the Scala compiler

© 2021-22 Robin Hillyard



$R \Rightarrow T$
function type
parametric type

Int \rightarrow String (n)
concrete type

domain ranges:

$() \Rightarrow 1$ value
Boolean $\Rightarrow 2$ values
int (4.4 billion types)

Types

- What is a type? What does *type* denote?
 - The domain of possible values;
 - The operations that can be performed on that type.
- Types in Scala come in two forms:
 - A “concrete” type is one that is fully known and has a physical representation in the Java Virtual Machine (either a “class” or a “primitive.” For example, *String* or *Int*.
 - A parametric or existential type is type a that still needs to be evaluated in order to become concrete. For example, *List[X]* is not concrete (assuming that *X* is a parametric type).
- For a lot more discussion on types, see
<https://kubuszok.com/2018/kinds-of-types-in-scala-part-1/>

Objects

- What is an object?
 - It is one instance of a particular (concrete) type...
 - ... whose value is one of the possible values in the domain of that type.
- The value of an object may not be known at compile-time—it typically is not known until run-time.
- But the type of an object is known at compile-time.

The Scala Compiler

- So, what exactly does the Scala compiler do?
 - It takes text and parses it into a legal abstract syntax tree (AST);
 - Each of about a dozen passes, refines the tree;
 - By the end, every node in the tree is *typed*;
 - The AST results in a set of ".class" files (bytecode);
 - See: <https://www.iteratorshq.com/blog/scala-compiler-phases-with-pictures/>

Spark - DataFrame API

Brief overview

- Inspired by python pandas
- Spark DataFrames are like distributed in-memory tables with named columns and schemas, where each column has a specific data type: integer, string, array, map, real, date, timestamp, etc.
- DataFrames are immutable and Spark keeps a lineage of all transformations.
- You can add or change the names and data types of the columns, creating new DataFrames while the previous versions are preserved.
- A named column in a DataFrame and its associated Spark data type can be declared in the schema.

Table format of a DataFrame

| Id (Int) | First (String) | Last (String) | Url (String) | Published (Date) | Hits (Int) | Campaigns (List[Strings]) |
|---------------------|---------------------------|--------------------------|---|-----------------------------|-----------------------|--------------------------------------|
| 1 | Jules | Damji | https://tinyurl.1 | 1/4/2016 | 4535 | [twitter, LinkedIn] |
| 2 | Brooke | Wenig | https://tinyurl.2 | 5/5/2018 | 8908 | [twitter, LinkedIn] |
| 3 | Denny | Lee | https://tinyurl.3 | 6/7/2019 | 7659 | [web, twitter, FB, LinkedIn] |
| 4 | Tathagata | Das | https://tinyurl.4 | 5/12/2018 | 10568 | [twitter, FB] |

Spark Dataframe API code

```
import org.apache.spark.sql.functions.avg
import org.apache.spark.sql.SparkSession
// Create a DataFrame using SparkSession
val spark = SparkSession
    .builder
    .appName("AuthorsAges")
    .getOrCreate()
// Create a DataFrame of names and ages
val dataDF = spark.createDataFrame(Seq(("Brooke", 20), ("Brooke", 25), ("Denny", 31),
("Jules", 30), ("TD", 35))).toDF("name", "age")
// Group the same names together, aggregate their ages, and compute an average
val avgDF = dataDF.groupBy("name").agg(avg("age")) // Show the results of the final
execution
avgDF.show()
```

Output

| name | avg(age) |
|--------|----------|
| Brooke | 22.5 |
| Jules | 30.0 |
| TD | 35.0 |
| Denny | 31.0 |

Spark's Basic Data Types

| Data type | Value assigned in Scala | API to instantiate |
|-------------|-------------------------|-----------------------|
| ByteType | Byte | DataTypes.ByteType |
| ShortType | Short | DataTypes.ShortType |
| IntegerType | Int | DataTypes.IntegerType |
| LongType | Long | DataTypes.LongType |
| FloatType | Float | DataTypes.FloatType |
| DoubleType | Double | DataTypes.DoubleType |
| StringType | String | DataTypes.StringType |
| BooleanType | Boolean | DataTypes.BooleanType |
| DecimalType | java.math.BigDecimal | DecimalType |

Complex Data Types

| Data type | Value assigned in Scala | API to instantiate |
|----------------|--|---|
| BinaryType | Array[Byte] | DataTypes.BinaryType |
| Timestamp Type | java.sql.Timestamp | DataTypes.TimestampType |
| DateType | java.sql.Date | DataTypes.DateType |
| ArrayType | scala.collection.Seq | DataTypes.createArrayType(Element Type) |
| MapType | scala.collection.Map | DataTypes.createMapType(keyType, valueType) |
| StructType | org.apache.spark.sql.Row | StructType(ArrayType[fieldTypes]) |
| StructField | A value type corresponding to the type of this field | StructField(name, dataType, [nullable]) |

```
import org.apache.spark.sql.types._  
import org.apache.spark.sql.SparkSession  
import org.apache.spark.sql.types._  
  
val schema = StructType(Array(StructField("author", StringType, false),  
StructField("title", StringType, false), StructField("pages", IntegerType, false)))  
val schema = "author STRING, title STRING, pages INT"  
object Example3_7 {  
    def main(args: Array[String]) {  
        val spark = SparkSession .builder  
            .appName("Example-3_7")  
            .getOrCreate()  
        if (args.length <= 0) {  
            println("usage Example3_7 <file path to blogs.json>") System.exit(1)  
        }  
        // Get the path to the JSON file  
        val jsonFile = args(0)  
        // Define our schema programmatically  
        val schema = StructType(Array(StructField("Id", IntegerType, false),  
StructField("First", StringType, false), StructField("Last", StringType, false), StructField("Url", StringType, false),  
StructField("Published", StringType, false), StructField("Hits", IntegerType, false), StructField("Campaigns", ArrayType(StringType), false)))  
        // Create a DataFrame by reading from the JSON file // with a predefined schema  
        val blogsDF = spark.read.schema(schema).json(jsonFile) // Show the DataFrame schema as output blogsDF.show(false)  
        // Print the schema  
        println(blogsDF.printSchema)  
        println(blogsDF.schema)  
    }  
}
```

Output

| Id | First | Last | Url | Published | Hits | Campaigns |
|-----------|--------------|-------------|-------------------|------------------|-------------|------------------------|
| 1 | Jules | Damji | https://tinyurl.1 | 1/4/2016 | 4535 | [twitter, LinkedIn] |
| 2 | Brooke | Wenig | https://tinyurl.2 | 5/5/2018 | 8908 | [twitter, LinkedIn] |
| 3 | Denny | Lee | https://tinyurl.3 | 6/7/2019 | 7659 | [web, twitter,...] |
| 4 | Tathagata | Das | https://tinyurl.4 | 5/12/2018 | 10568 | [twitter, FB] |
| 5 | Matei | Zaharia | https://tinyurl.5 | 5/14/2014 | 40578 | [web, twitter, FB,...] |
| 6 | Reynold | Xin | https://tinyurl.6 | 3/2/2015 | 25568 | [twitter, LinkedIn] |

root

```
|-- Id: integer (nullable = true)
|-- First: string (nullable = true)
|-- Last: string (nullable = true)
|-- Url: string (nullable = true)
|-- Published: string (nullable = true)
|-- Hits: integer (nullable = true)
|-- Campaigns: array (nullable = true)
|   |-- element: string (containsNull = true)
```

3.0 Containers and Exceptional Conditions

© 2019 Robin Hillyard



Containers

- Most real-world computing involves using **containers**:
 - Containers are wrappers around zero or more values
 - The most obvious and common container is *List[X]*, which represents zero or more elements of type *X*, in some order from *head* to *last*.
 - What exactly is *List[X]*? It's a trait defining methods such as *apply(x: Int): X* and *head: X*.
 - But is there a concrete type of List? Well, actually, there are two:
 - *case object Nil* (i.e. the empty list) which is a sub-class of *List[Nothing]*;
 - *case class ::[X] (h: X, t: List[X])* which is a sub-class of *List[X]*.

The functional programming way of dealing with abnormal situations

- In O-O, there are two constructs that were developed to allow for a deviation from the expected flow:
 - *null*
 - *Exception*.
- These are not FP-friendly. Why?
 - two reasons why *null* is not FP-friendly:
 - *null* is not an object and so does *not conform to type*;
 - if you try to dereference a *null* you get a NPE (null-pointer-exception)
 - *Exception* is not FP-friendly because throwing an exception is a *side-effect*!

Dealing with *null** in O-O

- Where do nulls come from in O-O?
 - missing optional values (for example in a database).
 - Basically, nulls are for lazy programmers: but they are dangerous!!
- How do we typically deal with a *null*?

```
public class Phones {  
    public static void main(String[] args) {  
  
        Map<String, Long> phones = new HashMap<>();  
        phones.put("Prof. Hillyard", 6173733972L);  
        // ...  
        String professor = "...";  
        Long x = phones.get(professor);  
        if (x != null) System.out.println(professor+"'s Phone number is: "+x);  
        else System.out.println("No phone number for "+professor);  
    }  
}
```



Before we can do anything with x, we first must ensure that it is not *null*. And yet, *null* is an expected value, not unusual, at all.

* Tony Hoare (quicksort) called “null” his \$1B mistake. Methods that can yield null do not advertise the same (unless they are strictly using @Nullable annotation).

Optional values—The Scala* way

- If the return type of a method is optional, then why not make it *explicitly* optional?



“foreach” is a pretty basic method available on all container-types. Here it means do it once or not at all, as the case may be.

```
val phones = Map("Prof. Hillyard" -> 6173733972L)
phones.get("Prof. Hillyard") foreach (x => println(s"found phone: $x"))

found phone: 6173733972
```

- In other words:
 - force the caller to deal with the possibility that there might not be a value returned...
 - ...*but* make it *easy* for the user to deal with that returned value.

* Java now has an *Optional* type also.

Dealing with exceptional conditions in O-O

- What if something goes wrong in O-O and we want to know what actually happened?
 - real life example: unable to get connection to remote database.
 - we catch/handle the exception (e.g. print stack trace) if we can, otherwise, we pass it up to the caller.
- What does this look like in practice?

```
Map<String,Long> phones = new HashMap<>();  
phones.put("Prof. Hillyard", 6173733972L);  
// ...  
String professor = "...";  
long x = phones.get(professor);
```



Unboxing to *long* (which the compiler has no problem with) causes a null-pointer-exception when the professor is not found.

Exceptions—The Scala way

```
trait Option[x]
case object None extends Option[Nothing]
case object Some[x](x:X) extends Option[x]
```

```
trait Try[X]
case class Success[X](x: X) extends Try[X]
case class Failure[X](e: Throwable) extends Try[X]
```

- Let's deal with these errors (exceptional conditions) in a calm, *referentially-transparent* way, with no loss of information:

```
def log(x: Double) = if (x > 0) Math.log(x) else throw new Exception("x must be positive")
def tryLog(x: Double) = Try(log(x))
val result = tryLog(-1)
result foreach (x => println(s"log value is $x"))
tryLog(math.E) foreach (x => println(s"log value is $x"))
log value is 1.0
```

- In other words:
 - this looks just like the situation where we returned *Option[Long]*
 - the difference is that we have ways to recover the exception from the *Try* object (including “throwing” it if we really want to).

// use try by name



“foreach” is used in the same way as in *Option*. There are ways to recover the actual cause of the *Failure* if we need to.

Option

- Suppose we want to find an element in a list that satisfies a predicate?
 - What if there's no such element (the list might be empty, or the predicate simply never yields *true* for any element)?
 - In Scala, the *find* method on *List[X]* returns an *Option[X]*.
 - How should we implement *Option[X]*?
 - What should its API be?
 - *isDefined: Boolean*
 - *isEmpty: Boolean = !isDefined*
 - *get: X [will throw exception if empty]* // never call get on an option
 - *getOrElse[Y >: X](default: => Y): Y = if (isDefined) get else default*
 - *map[Y](f: X=>Y): Option[Y]*

Option (2)

- So, we can define some methods we will want to call. Let's make them into a **trait**.

```
trait Option[X] {  
    def isDefined: Boolean  
    def get: X  
  
    ...  
}
```

- What have we got? A container based on a parametric type X and of which essentially there are two* types: an empty container and a non-empty container that holds an X in it.

* it's nearly always *two* types

Option (3)

- Let's call our two containers *Some* and *None* and implement them as *case classes/objects* extending *Option*.

```
case class Some[X](x: X) extends Option[X] {  
    def isDefined: Boolean = true  
    def get: X = x  
}  
case object None extends Option[Nothing] {  
    def isDefined: Boolean = false  
    def get: Nothing = throw new NoSuchElementException("None.get")  
}
```

- Now, we can use pattern matching to figure what we've got:

```
List(1,2,3).find(_%2==0) match {  
    case Some(x) => println(x)  
    case None => println("no even number found")  
}
```

Option (4)

- Using *Option* to handle objects returned from Java methods:

```
object Option {  
  
    import scala.language.implicitConversions  
  
    /** An implicit conversion that converts an option to an iterable value // here, converts java types to scala types */  
    implicit def option2Iterable[A](xo: Option[A]): Iterable[A] = xo.toList  
  
    /** An Option factory which creates Some(x) if the argument is not null,  
     * and None if it is null.  
     */  
    * @param x the value  
    * @return Some(value) if value != null, None if value == null  
    */  
    def apply[A](x: A): Option[A] = if (x == null) None else Some(x)  
}
```

- val x = javaFunction(); // could return null
- val xo = Option(x)

Try

- Suppose we want to convert a *String* to an *Int* and know that it might throw an exception?
 - In Java we can wrap the expression in try..catch..finally
 - In Scala, we can actually do the same thing. But there's a much better, more functional way: *Try[X]*.
 - How should we implement *Try[X]*?
 - What should its API be?
 - *isSuccess: Boolean*
 - *isFailure: Boolean = !isSuccess*
 - *get: X [will throw exception if failure]*
 - *getOrElse[Y >: X](default: => Y): Y = if (isSuccess) get else default*
 - *map[Y](f: X=>Y): Try[Y]* recover() & recoverWith() can handle exceptions
 - ...

Try (2)

- So, we can define some methods we will want to call. Let's make them into a **trait**.

```
trait Try[X] {  
    def isSuccess: Boolean  
    def get: X  
    ...  
}
```

Sound familiar? We defined *Option* just like this.



- What have we got? A container with a parametric type X and of which essentially there are two types: a successful container with an X in it and a failure container that holds the exception.

Try (3)

- Let's call our two containers *Success* and *Failure*. They will be case classes extending *Try*.

```
case class Success[X](x: X) extends Try[X] {  
    def isSuccess: Boolean = true  
    def get: X = x  
}  
case class Failure[X](e: Throwable) extends Try[X] {  
    def isSuccess: Boolean = false  
    def get: X = throw e  
}
```

- Now, we can use pattern matching to figure what we've got:

```
Try("a".toInt) match {  
    case Success(x) => println(x)  
    case Failure(e) => e.printStackTrace()  
}
```

Look familiar? We extracted the contents of *Option* rather like this. The big difference is that in the fail case, we actually have an exception



Try (4)

- Using *Try* to handle methods which may throw an exception (usually but not always Java methods):

```
object Try {  
    /** Constructs a `Try` using the by-name parameter. This  
     * method will ensure any non-fatal exception is caught and a  
     * `Failure` object is returned.  
     */  
    def apply[T](r: => T): Try[T] =  
        try Success(r) catch {  
            case NonFatal(e) => Failure(e)  
        }  
}
```

Note that the parameter to the *Try.apply* method is call-by-name: this allows the actual exception to be caught *inside* the apply method.



- val xy = Try("s".toInt)

Summary

- Scala defines two traits, each with two case classes/objects to represent the exceptional conditions corresponding to Java's *null* and *Exception*.
- As we shall see, these containers are very natural and easy to use and they are *referentially transparent!* // using mathematical induction, if we substitute the recursive values as a chain, we can get back the answer, ex: tailrecursive factorial
- Oh yes, and they are *monads!* (We'll talk about these later).
// used in for comprehensionsthis idea is called referentially transparency and this allows us to use pure functionsthis is not possible with impure functions -> when variables mutate we might get different answers

Updated: 2022-10-06

3.1 Methods, functions, types, etc.

© 2019-22 Robin Hillyard



Values, variables and expressions

- As mentioned previously, a functional program is essentially an *expression* which yields a *value*:
 - That expression is *expressed* in terms of *functions* of values. Semi-formally (using *productions*):
 - `expression ::= term | function(term, term, ...)`
 - `term ::= value | "(" expression ")"`
 - `value ::= constant | variable`
 - `variable ::= identifier`
 - A *variable* is something which we use to simplify an expression by extracting a sub-expression and giving it an identifier with which to refer to it.
 - Note that what we are used to thinking of as “operators” are just functions under another name (and with a different style of invocation);
 $1 + 2$ is equivalent to $1.+ (2)$ §
 - Also note that in Scala, we also have classes and objects which complicate the structure noted above—but in pure functional programming, we just have expressions as described.
 - What about keywords like *if*, *for*, *match*, *case*, etc.? These can all be treated, loosely, as “syntactic sugar” to define expressions.

Methods and functions

- A *function* is an object whose behavior is that it can transform an object of one type into an object of (maybe) another type:
 - so, for example, `Int=>String` is the type of a function which transforms an `Int` into a `String`
 - one way* to define a function is by declaring a variable with an appropriate type and provide an expression which says what the function does (where “`_`” is a placeholder for the value of the input to the function);
 - e.g.:
`val f: Int=>Int = _*2`
- A *method* is a property of a class or an object (it is not itself an object) which does the following:
 - it defines an identifier (so we can refer to the method);
 - It (optionally) declares one or more parametric types;
 - it declares a list of object (value) parameters (essentially, these are “variables” available to the function);
 - it defines a function (via the method body) in a convenient and readily understood format;
 - and—by convention, if the method belongs to a *class* (rather than an *object*)—implicitly adds another parameter set: *(this)*.
 - e.g.:
`object MyMath { def double(x: Int): Int = x*2; val f: Int=>Int = double }`

* the other way is by defining a method

Types, values, etc.

- The strength of Scala rests to a large extent on its strict type safety.
How do types, values and other properties relate to each other?
- Variables* all have six important aspects—which are shared knowledge between you and the compiler:
 - **name**: the *identifier* of the variable (i.e. how it gets referenced);
 - **type**: the *domain* the value belongs to, i.e. properties the variable supports—methods, range of legal values, etc.;
 - **scope**: *where* the variable can be referenced;
 - **mutability**: *whether* the value can be changed;
 - **value**: the *value* of the variable—(if mutable, then the *current* value);
 - **evaluation mechanism**: *how/when* the variable “evaluates” its value (if all functions are *pure* functions, it won’t matter when it gets evaluated).

*By “variable”, a Scala programmer doesn’t mean something that can change its value during a run. A variable is used in the sense of algebra—it’s something that stands for some sort of quantity.

A quick explanation: types

- Different styles of type:
 - **functions**: these can transform value(s) of one (or more) types into a value of some other type.
e.g. `(x: Int) => x.toString`
 - **scalars**: ordinary value types such as *Int*, *Double*, *String*, *LocalDate*, etc.
e.g. `3`
 - **containers**: wrappers around groups of values which may contain zero thru N members:
 - **longitudinal*** (collections): e.g. *Iterator*, *List*, *Array*, etc.
e.g. `List("a", "b", "c")`
 - **transverse***: e.g. *Option*, *Tuple*, *Try*, *Future*, *Either*, etc.
e.g. `Tuple("a", 1, 3.1415927)`
 - **hybrid***: e.g. *Map[String, Int]*, *Seq[(String, Int)]* etc.
e.g. `Map("a" -> 1, "b" -> 2)`

*These terms are not in common use: I use them to help differentiate different families of containers.

A quick explanation: Scope

- Scope in Scala is similar (but not the same) as in Java.
- Example of legal (but not good) code:

```
Object Z {  
    val x = 3  
  
    def y = {  
        val x = 5  ← This is called “shadowing”  
        x + 8  
    }  
}
```

The value of *y* is 13 (not 11).

Evaluation mechanism

- A variable or parameter can be *evaluated* in several ways, each by a different mechanism:
 - **direct reference (call-by-value)**: a variable (or method parameter) has a value and that value is effectively *substituted* for the variable/parameter wherever that variable/parameter is referenced.
 - `val x: X = expression`
 - `def y(x: X)`
 - **indirect reference (call-by-name)**: an indirect reference is like a reference via a pointer. We don't actually need to evaluate the pointer until we refer to it.
 - `def x = expression` which is equivalent to `val x: Unit=>Double = { _ => expression }`
 - `def y(x: => X)`
 - **lazy**: if a variable is lazily-evaluated, evaluation is *deferred* until it is needed. But unlike an indirect reference, which is evaluated each and every time it is referenced, a lazy variable is evaluated only when it is *first* referenced.

Exercise 1 - REPL

- A. def f(x: Int) = x*x
- B. f(9)
- C. def f(x: Int) = {println(x); x*x}
- D. f(9)
- E. val y = f(9)
- F. lazy val z = f(9)
- G. z + 19
- H. z + 20
- I. f{println("hello"); 9}
- J. def f(x: () => Int) = x()*x()
- K. f{} => println("hello"); 9}
- L. def f(x: => Int) = x*x
- M. f{println("hello"); 9}
- N. val g = {println(9); 9*9}

Compare, particularly, J/L, and K/M

Constructors & Extractors

- You’re familiar with the idea of *constructors* such as:
 - `List(1,2,3)`
 - `Complex(1.0,-1.0)`
- But, surely, if you can construct objects, you ought to be able to “deconstruct” (or extract) them:

```
case class Complex(real: Double, imag: Double)  
  
val z = Complex(1.0,-1.0)  
  
z match {  
  case Complex(r,i) => println(s"$r+$i")  
}  
  
def show(l: List[Int]): String =  
  l match {  
    case Nil => ""  
    case h::t => s"$h,"+show(t)  
  }
```

- Yes you can! This is how pattern matching works.

Exercise 2 - REPL

- A. case class Complex(real: Double, imag: Double)
- B. val z = Complex(1,0)
- C. z match {case Complex(r,i) => println(s"\$r \$i"); case _ => println(s"exception: \$z") }
- D. val l = List(1,2,3)
- E. l match { case h :: t => println(s"head: \$h; tail: \$t"); case Nil => println(s"empty") }

3.2

For-Comprehensions Quick Introduction/Refresher

© 2018-22 Robin Hillyard



For Comprehensions

- We already met the following construct where we print each element of a list on a separate line using the *foreach* method which is defined in *Iterable*.

```
List(1,2,3) foreach println
```

- But we could also write this, which has the same effect:

```
for (i <- List(1,2,3)) println(i)  
List (1,2,3) map println
```

We can spell this with
or without the hyphen

- This form is called a “for-comprehension” and it is “syntactic sugar” for the first statement (the one involving *foreach*).
- Thus, the type yielded by this for-comprehension is: *Unit*.

For Comprehensions (2)

- Actually, there are two quite different forms of for-comprehension:

```
for (i <- List(1,2,3)) println(i)  bad form - isn't supposed to be used
```

```
for (i <- List(1,2,3)) yield i.toString  good form
```

- As noted in the previous slide, the first form yields *Unit*. (i.e. it works through side-effects).
- Notice the “yield” in the second form. What does it do?
 - The result of the second statement is a *List[String]* with the elements “1”, “2”, and “3”
- We will discover that this second form is "syntactic sugar" for:

```
List(1,2,3) map (_.toString)  
      iterable          lambda
```

For Comprehensions (3)

- What if we wanted to “zip” two lists together? `xs zip List("A", "B", "C")`

```
val xs = List(1,2,3)  
for (x <- xs; y <- List("A", "B", "C")) yield (x, y)
```

- The result would be a *List[(Int, String)]*, in other words a list of tuples
- And, we will discover that this form is "syntactic sugar" for:

```
xs.flatMap(x => List("A", "B", "C").map(y => (x, y)))
```

`val op = for (gen 1, gen 2, .. gen n) yield func`

`shape of gen1 should be same as gen 2, gen 3 ... gen n, and op will be the same shape`

`when we use the for comprehension, we would have 1 map & n - 1 flatmaps`

For Comprehensions (4)

- What's *flatMap*?
 - *flatMap* is a lot like *map* except that it flattens the output.
 - its signature, for a container $C[X]$, is:
 - `def flatMap[Y](f: X => C[Y]): C[Y]`
 - recall that *map*'s signature is:
 - `def map[Y](f: X => Y): C[Y]`

For Comprehensions (5)

- We will find that there is much more to these for-comprehensions. They are an important way of making functional code more readable by a human.

Not always a loop

- So far, all of the usages of for-comprehensions have been looping over lists (0 thru N elements);
- But, what about something that might have 0 or 1 element (such as *Option*)?

```
private val map = Map("A" -> "alpha", "B" -> "bravo", "C" -> "Charlie",  
"D" -> "delta", "X" -> "xray")
```

```
private val ao = map.get("A")  
private val xo = map.get("X")  
ao.flatMap(a => xo.map(x => x + a))  
private val z1 = for (a <- ao; x <- xo) yield a + x Some ("alphaxray")
```

// The for comprehension above is more or less equivalent to the pattern match below.

```
private val z2 = (ao, xo) match {  
  case (Some(a), Some(x)) => Some(a + x)  
  case _ => None  
}
```

. get(<key>) returns Option(<value>)

Updated: 2022-10-11

3.3 Parallel Processing, Mutable State and Immutability

How to avoid using mutable anything.

© 2021-22 Robin Hillyard

A large, semi-transparent watermark of the Northeastern University logo is positioned in the lower right quadrant of the slide. The logo consists of a stylized 'N' formed by two overlapping white shapes: a trapezoid pointing down and a rounded triangle pointing up. To the right of the logo, the words "Northeastern University" are written in a white serif font.

Northeastern
University

Parallel Processing (1)

- Modern computers have multiple “cores” and are typically arrayed in clusters. This allows us (in practice, forces us) to do things in parallel. There are two types of parallel processing and Scala can be effective in both:
 - immutable state—the definition of the problem is known at the start and does not change until the problem is solved
 - e.g., count the total number of words in 1,000,000 documents
 - Amdahl’s law applies to this type of problem:
 - maximum possible speedup $\sigma = 1/\alpha$ where α is the proportion of total time which is non-parallelizable
 - Because each document is independent of the others, a master node can divide up (this part is not parallelizable) the documents to be processed by a set of worker nodes/threads;
 - Then (in parallel) each worker counts the words in the documents it was given and returns the result, asynchronously of course, as a future value
 - Once all of these future values are realized, they can be summed to get the grand total (that final step is also non-parallelizable)
 - This is (more or less) the principle on which Map/Reduce works

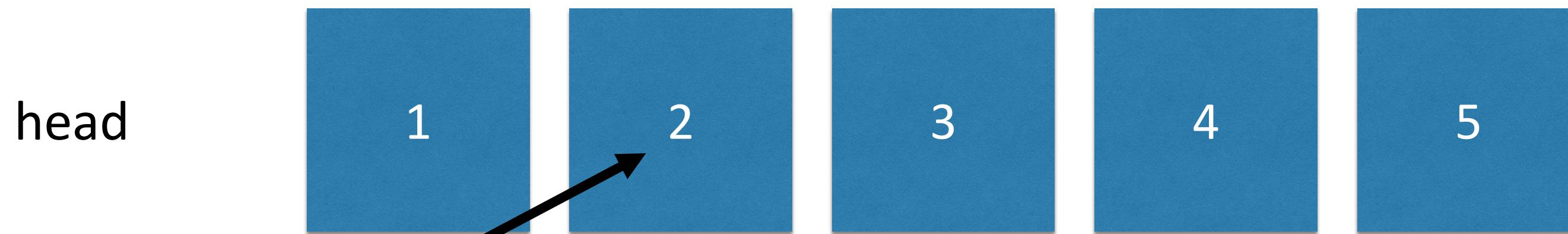
Parallel Processing (2)

- continuing:
 - *mutable state*—the conditions are inherently changing at all times
 - e.g., ticket agency
 - there are two fundamental mutable states: the pool of tickets and the bank account of agency
 - it is therefore essential that these can be updated only by one thread at a time
 - there are other temporary mutable states—e.g., the status of a shopping cart—but these, if lost or corrupted, can be restarted
 - for these applications, we use *actors* (more about this later)

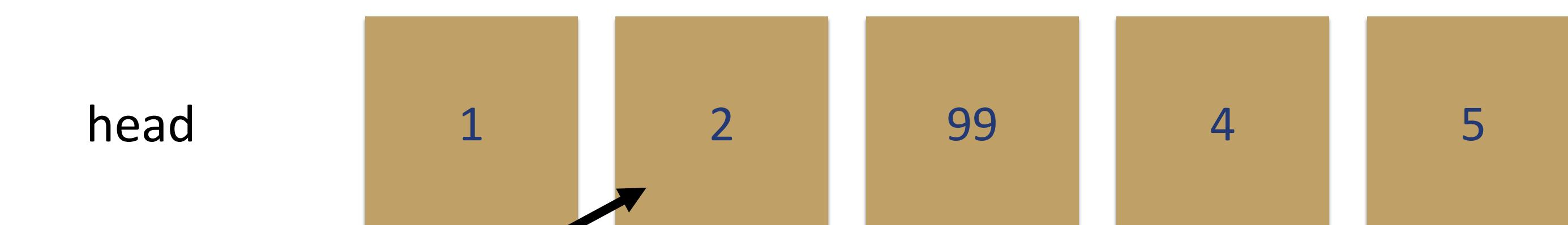
Parallel Processing (3)

- Apart from the internals of actors, is there any need to use mutable variables (vars) or mutable containers/collections?
- Caching (memoization)?
 - Maybe. But this can also be done inside an actor.
- Aggregation?
 - Normally, in functional programming we perform aggregation by recursion—no mutable state required.
- Sorting as a prelude to searching?
 - Yes—we do typically perform sorting on a mutable array but this again can/should be done inside an actor.
- Non-deterministic algorithms?
 - Yes, like the Newton Approximation or maybe genetic algorithms, etc.

Immutable list



Mutable list



Oops!

But surely we need *some* variables?

- Rarely! Although Scala provides *var* and has a library of mutable collections—you typically don't need them!
- Let's think about a box office app (see next slide):
 - if we are careful we can combine all mutable state into one *var* (the “state” of an actor);
 - when we take a list of items from another list (for example, we remove some tickets from the pool), surely we end up with lots of redundant copies of tickets! No, we don't.
 - in fact, the opposite is true: in an imperative program (e.g. Java) we typically end up doing a lot of list copying because our lists are mutable when they don't need to be!

Ticket Agency

Lightbend :
Scala
Akka
Play2

```
package edu.neu.coe.scala
package boxOffice
object BoxOffice extends App {
    val initialState = State(List(Ticket(1,1,100),Ticket(1,2,100)), List(), List())
    Tickets(initialState).start
}
case class Tickets(private var state: State) extends scala.actors.Actor {
    def act() {
        while(state.availability) {
            receive {
                case sale: Sale => state = state.makeTransaction(sale)
                case Status => sender ! state // here, rather should have sent state.toString , so that state cannot be modified since its mutable
                case _ => println("unknown message")
            }
        }
    }
}
case class State(tickets: List[Ticket], transactions: List[Transaction]) {
    def availability = tickets.length > 0
    def makeTransaction(sale: Sale): State = ??? // Take some tickets and add it to Transactions
}
case class Ticket(row: Int, seat: Int, price: Int)
case class Sale(tickets: List[Ticket], transaction: Transaction)
case class Transaction(creditCard: Long, total: Int, timestamp: Long, confirmation: String)
case object Status
```

Scala actors have been deprecated in favor of Akka

Now, Actors looks different, its from Akka

The field(s) of a case class can be marked “var”

??? is a boon to designers who want to create stubs to be implemented later

Updated TicketAgency (part thereof)

```
object Agency {

    def apply(): Behavior[Request] = create(List.empty)

    private def create(transactions: List[ActorRef[Transaction]]): Behavior[Request] =
        Behaviors.receive { (context, message) =>
            message match {
                case CreateTicketPool(ts, _) =>
                    context.log.info(s"CreateTicketPool(${ts.size}) received")
                    context.log.info(s"create with ticket transaction no. ${transactions.size}")
                    create(transactions.appended(context.spawn(TicketPool(ts, Nil), s"ticketPool-${transactions.size}")))
                case SeatRequest(x, p, replyTo) =>
                    if (x > 0)
                        if (transactions.nonEmpty) {
                            implicit val timeout: akka.util.Timeout = 15.seconds
                            val transaction = scala.util.Random.shuffle(transactions).head
                            context.log.info(s"SeatRequest($x, $p) received with transaction = $transaction")
                            transaction.ref ! ProformaTransaction(x, p, replyTo)
                            Behaviors.same
                        }
                        else throw TicketAgencyException("no tickets available")
                    else throw TicketAgencyException("seats requested must be greater positive")
                case seats: Seats =>
                    context.log.info(s"Seats booked ${seats.ts}")
                    Behaviors.same
            }
        }
}
```

* See <https://github.com/rchillyard/TicketAgency>

Take care of mutable state when testing

```
package edu.neu.coe.scala;

import static org.junit.Assert.assertEquals;
import org.junit.BeforeClass;
import org.junit.Test;
import scala.util.Random;

public class GenericTest {
    private static Random random;
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        random = new Random(0L);
    }
    @Test
    public void test1() {
        int x = 0;
        for (int i = 0; i < 2; i++)
            x = random.nextInt();
        assertEquals(x, -723955400);
    }
    @Test
    public void test2() {
        int x = random.nextInt();
        assertEquals(x, 1033096058);
    }
}
```

This is Java, by the way

what if we change this number? How will test2 work out?

what if the test runner decides to run test2 before test1?

- We will come back to this problem later.

So, you really want to use mutable variables

Nothing I can do to dissuade you? Always avoid using var & Array
Array - mutable collection of objects

- I recently was working on a method to force x into the range 0 through 2pi. I was in a hurry and couldn't think how best to do it without a var. This has rarely happened to me but I did it anyway:

```
// NOTE: using a var here!
var result = z
while (result < min) result = result + max - min
while (result > max) result = result + min - max
result
```

- Note that I wrote a “NOTE” comment to be sure that everyone looking at this code would be warned about me using var.
- A little later, I decide to clean it up. This is what I wrote:

```
@tailrec
def inner(result: X): X =
  if (result < min) inner(result + max - min)
  else if (result > max) inner(result + min - max)
  else result
inner(z)
```

Was that so hard?

- No, it wasn't hard. It was easy. And it just looked so much more elegant.
- Here's another situation where you will probably want to use a mutable collection:

```
case class Mill (stack: mutable.Stack[Number]) { //ensure to include "mutable ." if you really want to use mutable objects
  def push(n: Number): Unit = {
    stack.push(n)
  }
  def pop(): Option[Number] = if (stack.isEmpty) None else Some(stack.pop())
  def isEmpty: Boolean = stack.isEmpty
}
val mill = Mill()
mill.push(Number.one)
mill.pop() match {
  case None => fail("logic error")
  case Some(x) => x shouldBe Number.one
}
```

- But, surprisingly perhaps, you don't need to do it this way.

Proper way to define Mill

In scala, there is no Stack [T], since its the same as List [T]

- Here we use an immutable List:

```
case class Mill(stack: List[Number]) {  
    def push (x: Number): Mill = Mill(x :: stack)  
    def pop: (Option[Number], Mill) = stack match {  
        case Nil => (None, this)  
        case h :: t => (Some(h), Mill(t))  
    }  
    def isEmpty: Boolean = stack.isEmpty  
}  
val mill = Mill().push(Number.one)  
mill.pop match {  
    case (None, _) => fail("logic error")  
    case (Some(x), _) => x shouldBe Number.one  
}
```

- Notice that there is no immutable Stack class in Scala. Why? Because a stack and a list are exactly the same thing!
- Notice also that both *push* and *pop* must return a *Mill*. In the case of *pop*, we must return a tuple of *Mill* and *Option[Number]*.

Hash tables and caches

- Another situation where you'll be tempted to use a mutable collection is when you set up a Hash table.
- Now, depending on your actual use case, this may be appropriate. But like the use of the stack/list in the *Mill*, you don't have to do it that way.
- The place where you will need to make it mutable is when you are defining a cache which has a relatively global scope. It might be awkward and inelegant to carry the latest version of the cache around in all of your logic. One way to get around this is to make the pointer to the cache a var.
- Fortunately, there's a pattern in functional programming which takes care of this situation. It's called an actor. An actor *hides* its state from other code because that other code does not have direct access to the state: the access is *indirect* via the actor manager (or actor system). Actor is referentially transparent, in the code we get a pseudo reference to the actor
Actor is opaque, we cannot change working of actor directly

Let's set up a cache: first the traits

```
trait Cache[K, V] extends (K => Future[V])
```

- This trait simply defines a cache as a function that takes a *K* (key type) and returns a *Future[V]* (the value type wrapped in the asynchronous wrapper called *Future*).

```
trait ExpiringKey[K] {  
    def expire(k: K): Unit  
}
```

- This trait defines a method which can be used to expire (remove) a key from a cache. We haven't shown how this method might get called, based perhaps on a timer.

```
trait Fulfillment[K, V]{  
    val fulfill: K => Future[V]  
}
```

- This trait describes how a key-value pair actually gets fulfilled such that it can be stored in a cache.

Let's set up a cache: next the case class

```
case class FulfillingCache[K, V](fulfill: K => Future[V]) extends Cache[K, V] with ExpiringKey[K] with Fulfillment[K, V] {  
    private def put(k: K, v: V): Unit = cache += (k -> v)  
    def apply(k: K): Future[V] = if (cache.contains(k)) Future(cache(k))  
    else for (v <- fulfill(k); _ = put(k, v)) yield v  
    def expire(k: K): Unit = cache -= k  
    private val cache: mutable.Map[K, V] = mutable.Map.empty  
}
```

- It doesn't matter that this class has a difficult name. We will never actually use it in application code.
- Two code fragments to note: in method *put*, we see “*cache* += (k -> v)”
- That += method is the one that should be used when adding something to a mutable collection.
- Similarly, in method *expire*, we see “*cache* -= k” where the -= method is the one to remove an element from a mutable collection.

Using a factory class

- Factory classes are well-known from object-oriented programming.

```
object CacheFactory {  
    def createCache[K, V](fulfill: K => Future[V]): Cache[K, V] =  
        FulfillingCache(fulfill)  
    def lookupStock(k: String): Future[Double] = Future(MockStock.lookupStock(k))  
    def createStockCache: Cache[String, Double] = createCache(lookupStock)  
}
```

- We define a base factory method *createCache* and a specific factory method *createStockCache*, which defines the *fulfill* method based on some method called *lookupStock*, which we don't really need to know about.
- Because of encapsulation, and because *createCache* returns a *Cache*, not a *FulfillingCache*, the caller does not know anything about fields such as *cache* field.

Iteration and Recursion

Church-Turing Thesis

stack

- Iteration requires mutation; recursion requires ~~immutability~~^{stack}:
- In general, we can always replace iteration with recursion or *vice versa*.
- For detail, see [https://en.wikipedia.org/wiki/Recursion_\(computer_science\)](https://en.wikipedia.org/wiki/Recursion_(computer_science))

3.4 Type Declarations

© 2017-23 Robin Hillyard



What are types?

- A type defines the range of values (domain) and the set of operations that may operate on the given values.
- What does a program do at run-time?
 - It uses machine instructions such as + to operate on *values*.
- What does a compiler* do at compile-time?
 - It uses grammar and inference to operate on *types*;
 - Then it generates machine instructions to execute during run-time.

* of a typed-language

Defining Types* (1)

- “type”, trait, class, object, case class, abstract class, value class:
 - what do they all mean? what’s the difference?
 - “type” is to an object, class or trait what “val” is to value (i.e. an alias)

```
object List {  
    type IntList = List[Int]  
    def sum(ints: IntList): Int = ints match {  
        case Nil => 0  
        case Cons(x, xs) => x + sum(xs)  
    }  
}
```

- “type” can also be used to refer to a (singleton) object, as in `List.type`. same as "List\$",
but we don't use it
- a trait (or object) can also define a “type” member

```
trait Base {  
    type T  
    def method: T  
}
```

- where `T` must be defined as a concrete type wherever `Base` is subclassed concretely, e.g.

```
class Dog extends Base {  
    type T = String  
    def method: String = "woof!"  
}
```

* Some of this will be familiar from 2.5 More Detail on Scala Syntax

Defining Types (2)

Traits

- define *behavior*;
- are basic to type hierarchies and can be “mixed in” to provide multiple inheritance (with rules on exactly how methods are overridden if necessary);
- allow the definition of abstract/concrete types/properties/methods (features are abstract if they lack concreteness);
- traits can define both concrete and abstract methods/values (example: abstract compare function);

Abstract:

`def hello: String`

Concrete:

`def hello = "Hello"`

Defining Types (3)

Traits (continued)

- can have type parameters (“parametric polymorphism”), e.g. A in following example (not exactly like in Scala package):

```
trait Monad[+A] {  
    def map[B, C](f: A => B): C  
    def flatMap[B, C](f: A => C): C  
    def foreach[U](f: A => U): Unit  
}
```

+A Monad is covariant in A

- But *cannot* have value parameters (you need an abstract class for that) [Note that this implies that you can't define a parametric type of a trait to be a member of a *type class*];
- *Note: Scala 3 allows traits with parameters.* in scala 2, if we need a value parameter, we need to create an abstract class
`trait Monad[A : String] // not possible in scala 2`

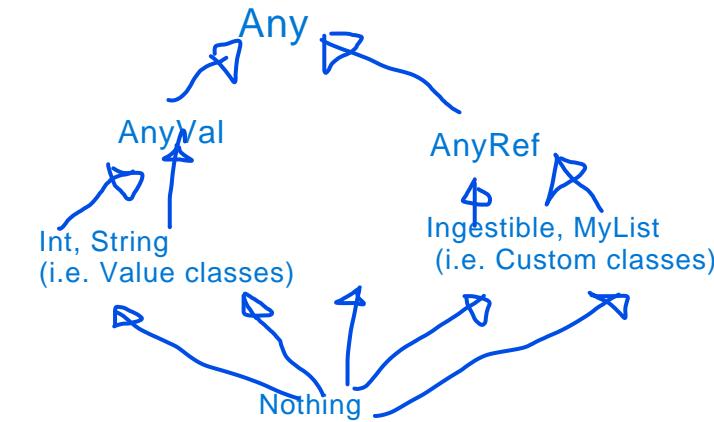
Defining Types (4)

Value classes

- *Int, Boolean, Double*, etc.
- Programmers can define Value Classes (since 2.10):
 - some efficiency improvements
 - extend *AnyVal* (rather than *AnyRef*)
 - may not contain references to non-Value objects
 - various other restrictions
 - only one actual value
 - no “require”
- Example: *UniformDouble* in an obsolete assignment:

```
case class UniformDouble(x: Double) extends AnyVal with Ordered[UniformDouble] {  
    def + (y: Double) = x + y  
    def compare(that: UniformDouble): Int = x.compare(that.x)  
}
```

Any is superclass of all scala classes



Nothing is a sub class of every class

Defining Types (5)

Abstract classes

- Are similar to traits but with minor differences:
 - can define constructor parameters (in addition to type parameters as in traits)
 - cannot be mixed in like traits (a sub-class can have only one super-class but many super-trait);
 - polymorphism is slightly more efficient (because *super* is known and is statically bound);
- Other unimportant observations:
 - can be sub-classed by Java-7 classes;
 - is binary-compatible: if members are added to an abstract class that is used by other modules, those modules do not need to be recompiled (assuming they don't reference the new member) —but if new member was added to trait, modules would need to be recompiled.

Defining Types (6)

Object (companion object or ordinary singleton)

- companion object is where we put the Scala equivalent of “class” (static) methods
- singleton objects are just that, e.g., a message type (“Status”) that we sent to our Ticket Agency actor in week 1.

Case classes/objects `case object Nil extends List[Nothing]`

- extend *Product* (the trait extended by Tuples) but cannot normally be sub-classed (they are leaves of the class hierarchy);
- case *objects* have no parameters (that’s why they’re objects, not classes).

Defining Types (7): Case classes continued

When you use a case class, the compiler provides a lot of “boiler-plate” code for you:

- It provides **val** for each “member”—this essentially exposes the members to the public scope
- You can override this **val** in several ways:
 - Add the **private** keyword (or **private[scope]**);
 - You can make it mutable by using the **var** keyword;
 - You can put the member in a second parameter set—but be careful as that parameter is really not a *member*—but one time you might want to do that is if you want the parameter to be *call-by-name* or *implicit*.

Defining Types (8): Case classes continued

```
case class Complex ( r : Double, i : Double ) {  
    def unapply (c : Complex) : Some ( (c.r, c.i) ) // given by compiler  
}
```

Continuing with what the compiler provides:

- It provides *equals*, *hashCode*, and *toString* as instance methods;
- It provides *apply* and *unapply* in the companion object (even though you may not have defined a companion object, the compiler provides an invisible one).
 - *apply*: takes the exact same parameters as the members and essentially exists so that you don't have to keep using the **new** keyword;
 - *unapply*: takes an object and returns *Some(tuple)* where the tuple corresponds to the members. If the object isn't of the same type as the case class, then *None* will be returned. *Unapply* is primarily used by the **match...case** construction.
- It provides a default implementation of *Serializable* (true also for a case object).

Defining Types (9): Case classes continued

There are one or two other aspects of case classes:

- Case classes support the *copy* method;

```
case class Complex(real: Double, imag: Double) {  
    def moveHorizontal(dx: Double): Complex = copy(real = real + dx)  
    def moveVertical(dy: Double): Complex = copy(imag = imag + dy)  
}
```

- The *apply* method is often used for the purpose of type inference, rather than being invoked. For example:

```
implicit val complexFormat: RootJsonFormat[Complex] = jsonFormat2(Complex.apply)
```

- In this case, *Complex.apply* will only be invoked directly when *reading* from Json. When *writing* to Json, the *apply* method is used solely for type inference.

Defining Types (10): Case classes continued

Case class constructors can have additional parameter sets:

- However, the standard methods already described, such as *toString*, *equals*, *apply*, *unapply*, etc. do not “see” the additional parameters (they are not true *members*).
- Sometimes, you do want to add a parameter set, especially to allow “implicit” parameters:

```
case class CsvTableFileRenderer[T: CsvRenderer : CsvGenerator](file: File)(implicit csvAttributes: CsvAttributes) extends CsvTableRenderer[T, FileWriter]()(implicitly[CsvRenderer[T]], implicitly[CsvGenerator[T]], Writable.fileWritable(file), csvAttributes)
```

- In the above declaration, the implicit parameters (generated by the compiler) of *CsvRenderer[T]* and *CsvGenerator[T]* are added to the implicit parameter set and not “seen” by *equals*, *apply*, etc.

Defining Types (11): Tuples

| | Pros | Cons |
|------------|-------------------------------------|--|
| case class | choose appropriate names for member | explicitly needs to be declared |
| tuple | don't need any declaration | names are ugly <code>_1, _2, _n</code> |

Tuples are case classes (and so extend *Product*) but with names chosen for their position.

- A case class allows you to name the members—a *TupleN* has member names like `_1, _2`, etc.;
- Note that (T,R) is syntactic sugar for *Tuple2[T,R]*.

Defining Types (12): Extending functions

What about extending function types? Functions aren't classes,
are they?

Functions are traits

- No, but they are types and types can be extended;
- See the *lab-sorted* code for some examples.
- Note that $T \Rightarrow R$ is syntactic sugar for *Function1[T, R]*.

There is always one result: R

```
trait Function1[T, R] {  
    def apply(t: T) : R  
    def compose ...  
    def andThen  
}
```

Function1, - only has
compose & andThen,
rest function dont
have it:
Function2,
... Function22

Defining Types (13): Structural types

Suppose that we need to invoke a method on an object that is not defined by a trait extended by or evidenced by the class of the object.

We can nevertheless require that the type of the object defines the method we need.

This is called a Structural Type. [different in Scala 3](#)

[See our repository.]

Modules/Packages

- A Scala file is called a “module”
 - A module corresponds more to a Java package than a Java class (whereas Java packages are directories of files)
 - Thus, a module in Scala can define multiple traits, classes, objects, etc.
 - You can use the keyword *package* to define a more specific package.
 - *However*, if you want to execute the main method of a module, or run tests defined in a module, you must ensure that the (first) package statement corresponds to the position in the directory structure. Otherwise, things get confused because Java is quite strict about the class packaging.

Package object

[not there anymore in Scala 3](#)

- As in Java, there is also a package “object” which can contain definitions that are common to the whole package.
[all modules of](#)
- I find this particularly useful for defining type aliases as in:

```
type Ints = Seq[Int]
```

- because you can't do that inside a module. [in scala 2](#)

3.5 ADT Algebra

© 2021-22 Robin Hillyard



Abstract Data Types*

- Basically, an ADT is simply three things:
 - *Data structure*;
 - *Algorithm(s)*;
 - *Invariant(s)*.
- An invariant is something that must be true when the ADT is in a steady state.
 - For example, the data structure of a *Map* is simply a sequence of key-value pairs.
 - The invariant requires that there are no duplicates among keys.
 - Otherwise, a *Map* would be identical with a sequence of key-value pairs.

`val map = Seq [(K, V)]`

both are identical

`val map = Map [K, V] , but here we cannot have duplicate key`

* In functional programming contexts, you might see this as “algebraic data types”

But, first, a little algebra

- I'd like to cover something that isn't unique to functional programming but is more relevant than with other paradigms.
- Suppose I am defining a method *m* as follows:
 - `def m(x: Int): String = ???`
- I may not know much about what to put instead of `???`. But here's what I do know:
 - it must involve *x*. Why?
 - it must yield a *String*. Why?
 - it *should* not involve anything else, apart from *this* (if *m* is an instance method). Why?
 - in practice, we are allowed to use identifiers from *Objects* but we should only do so if it's completely obvious why we're doing it. For example we might reference *Math.PI* or *System.out.println*.
 - in particular, we should avoid using any "global" values unless they are totally obvious.

Exceptions: `implicits`, `Math.pi`, `System.get` etc. we can reference methods from objects

Simple, Obvious, Elegant

- This is what I call the SOE (Simple, Obvious, Elegant) principle.
- Whenever we are writing an expression, we will:
 - Write code that is as *simple* as possible;
 - Use only *obvious* variables and method invocations;
 - End up with *elegant* code.
- Although this has little to do with Scala programming, I am always reminded of [Occam's Razor](#).
- Don't forget that you will be guided entirely by the types of the variables and methods available.

ADT algebra

- So, what are the operators that we can apply to an ADT? That's to ask what is its *algebra*?
- That's going to depend obviously on what the ADT does and whether or not it's an immutable ADT.
- What good is an immutable ADT? I hear you ask.
- Well, we can easily just copy an existing instance of an ADT and make some modifications to it if we want the equivalent of mutation. This sounds like a lot of copying, perhaps, but actually it isn't.
- Let's start out with a nice simple ADT: *List[A]*.
 - BTW, *List* is an abstract class which implements the *Seq* trait. Its two concrete implementation classes are `::` and *Nil*.
 - As we invent an algebra, refer to the actual [List algebra](#).

Dreaming up all possible signatures for *List[A]*

- What possible return types are there?
 - **Unit**
 - **Boolean**
 - **Int**
 - **String**
 - **A**
 - **Option[A]**
 - **List[A]**
 - **(List[A],List[A])** List [A] or List [B] where B >: A i.e. B is a super type of A
 - *Does anything else make sense? Not really!*

First: Methods with no parameters

- What names and semantics might we give these according to the return type?
 - **Unit**: ? There doesn't seem to be an obvious candidate; how about *println*?
 - **Boolean**: *isEmpty*;
 - **Int**: *length*;
 - **String**: *toString*;
 - **A**: *head* [but what does it do on an empty list? See below: *headOption*];
 - **Option[A]**: *headOption*;
 - **List[A]**: *tail*.
 - **(List[A], List[A])**: *first half, second half*.

One parameter of a function type:

- What possible return types are there with one function parameter of type **A=>X**?

- **Unit**: *foreach* [where X is *Unit*]; `def foreach[U] (f : A => U) : Unit // U can be unit, but not always`
- **Boolean**: *exists* [where X is *Boolean*]; `def exists (f : A => Boolean) : Boolean`
- **Int**: *count* [where X is *Boolean*]; `def count (f: A => Boolean) : Int`
- **Option[A]**: *find* [where X is *Boolean*]; `def find (f : A => Boolean) : Option [A]`
- **List[B]**: *map* [where X is *B*];
- **List[B]**: *flatMap* [where X is *List[B]*];
- **(List[A], List[A])**: *partition* [where X is *Boolean*]; `def partition (f : A => Boolean) : List [B]`
`List[B] where B is a parametric type: map[where X is B] def map[B] (f : A => B) : List [B]`

`list(3) => 3 indexed = 3`

`0 1 2 3 n - 1`

One parameter of an *Int* type:

- What possible return types are there with one parameter of type *Int*?

- **Unit**: doesn't make sense; `println(get(x))`

Option Try
don't use .get => so we
don't encounter Nil and
exception is not thrown

- **Boolean**: *isDefinedAt*;

- **Int**: no obvious method;

- **String**: no obvious method; `makefromStringN`

- **A**: *apply*;

- **Option[A]**: *get*;

For iterables,
.get returns Option
but apply might throw
exception, so .get is safer

- **List[A]**: *take* (or *drop*);

- **(List[A], List[A])**: *splitAt*.

- **val a = xs(1) // unsafe**

3.6

Updated: 2022-10-13

Asynchronous Execution: Futures

© 2015-22 Robin Hillyard



Asynchronous Execution and Futures

- What does a processor/thread do most of the time?
 - It waits to be asked to do something (unless, that is, we are very smart and sufficiently determined to keep it busy).
 - One way we can try to keep our thread busy is to ensure it doesn't have to wait for slow operations. In other words, we use asynchronous calls (i.e. non-blocking) whenever we want to do two or more things in parallel.
 - The time-honored way to implement asynchronous calls (e.g. Ajax or Swing in Java7) is by defining a *callback* method. You provide a callback that will be invoked when the result is ready.
 - That's similar to waiting for the user to do something in a UI
 - But, although ultimately you will have to do some waiting, you would prefer to minimize the number of callbacks to one (per transaction).
 - In Scala, we make asynchronous calls using a *Future*. Guess what? It's a monad.

Let's design our own type: *Par*

- This is an exercise in designing an algebra.
- First let's think about the kinds of things we want to do in parallel:
 - How about summing the elements of a *List*?

```
def sum(is: IndexedSeq[Int]): Int =  
  if (is.size <= 1)  
    is.headOption getOrElse 0  
  else {  
    val (l, r) = is.splitAt(is.length/2)  
    sum(l) + sum(r)  
  }
```



For example, let's split our list of integers and sum each half independently (divide and conquer).

Let's design our own type: *Par* (1)

- Perhaps we could define a trait *Par*[*T*]
 - Clearly, our *Par*[*T*] will have to be able to hold a result of some type *T*. That's to say it will “wrap” a *T*.

Let's design our own Par (2)

- So, let's create a trait and some methods:

```
trait Par[T] {  
    def get: T  
    def pure(t: T): Par[T]  
}  
  
object Par {  
    def apply[T](t: T): Par[T] = new Par[T] { def get = t; def pure(u: T): Par[T] =  
Par.apply(u) }  
}
```

- Now, we can rewrite our sum method:

```
def sum(is: IndexedSeq[Int]): Int =  
    if (is.size <= 1)  
        is.headOption getOrElse 0  
    else {  
        val (l, r) = is.splitAt(is.length/2)  
        val sumL = Par(sum(l))  
        val sumR = Par(sum(r))  
        sumL.get + sumR.get  
    }
```

Notice that a `val` declaration is essentially a pattern—we can declare a tuple made up of two `vals`: `l` and `r`.

- Note that we haven't said anything yet about how we might implement the `get` or the `apply` methods.

Let's design our own Par (3)

- This is fine, but if we substitute the right-hand-side of *get* for the invocation(s) of *get*, we basically force evaluation—but not in parallel.
- We need a way to combine the results of the two parallel computations, leaving *get* to be called later when we really need to know the answer.
- Can you think of a method where we can combine two *containers* (like *Par*), while knowing only a function that can be applied to combine the values of the containers? Sound familiar??

```
trait Par[T] {  
    def get: T  
    def map2(p: Par[T])(f: (T,T)=>T): Par[T]  
}  
  
object Par {  
    def apply[T](t: => T): Par[T] = ???  
    def sum(is: IndexedSeq[Int]): Int =  
        if (is.size <= 1)  
            is.headOption getOrElse 0  
        else {  
            val (l,r) = is.splitAt(is.length/2)  
            val sumL = Par(sum(l))  
            val sumR = Par(sum(r))  
            val result = sumL.map2(sumR)(_+_).get  
            result.get  
        }  
}
```

Movie database

Reviews

Rating score

```
trait List[T] {  
    def map[U] (f : T => U ) : List [T]  
    def map2[U, V] ( f : (T, U) => V ) ( List [U] ) : List [V]  
}
```

caller

(_ + _) means (sumL + sumR)

Let's design our own Par (4)

- What have we got?
- *Par* is a data structure that allows us to set up lazy calculations which can, we hope, be implemented in parallel.
 - But *Par* doesn't know how to do this.
- What does?
 - *ExecutorService* knows.
 - It's a Java class with a Scala wrapper (kind of)

```
class ExecutorService {  
    abstract def submit[T](arg0: Callable[T]): Future[T]  
}  
trait Callable[T] { def call: T }  
trait Future[T] {  
    def get: T  
    def isDone: Boolean  
    // etc.  
}
```

Let's design our own Par (5)

- So, when we actually *get* the value of our *Par* object, we will have to *run* it with an *ExecutorService*.
- So, it turns out that *get* isn't so useful and we will replace it with *run*:

```
trait Par[T] {  
    def run(implicit ec: ExecutorService): T  
    def map2(p: Par[T])(f: (T, T) => T): Par[T]  
}
```



Let's pass the *ExecutorService* implicitly since it's not really part of the logic that we want to make obvious.

- So, now we can rewrite our *sum* method:

```
trait Par[T] {  
    def run(implicit ec: ExecutionContext): Future[T]  
    def map2(p: Par[T])(f: (T, T) => T): Par[T]  
}
```



Notice that, instead of having *run* return a *T*, we return a *Future[T]*. In this context, *Future* is a Java interface.

```
object Par {  
    def apply[T](t: => T): Par[T] = ???  
    def sum(is: IndexedSeq[Int]): Int =  
        if (is.size <= 1) is.headOption getOrElse 0  
        else {  
            import scala.concurrent.ExecutionContext.Implicits.global  
            val (l, r) = is.splitAt(is.length / 2)  
            val sumL = Par(sum(l))  
            val sumR = Par(sum(r))  
            val result = sumL.map2(sumR)(_ + _)  
            result.run.get  
        }  
}
```



Also, we are passing in an *ExecutionContext*, whence we can derive an *ExecutorService*.

Let's design our own Par (6)

- OK, this isn't bad.

for (t1 <- this; t2 <- p) // is actually:

this.flatMap(t1 => p.map(t2 => f (t1 : ???))

- How should we go about implementing *map2*?

`def map2(p: Par[T])(f: (T,T)=>T): Par[T] = for (t1 <- this; t2 <- p) yield f(t1,t2)` would be done sequentially

- What will we need *Par* to implement for this to work?

- *Par* needs to be a monad: i.e. it must implement *map* and *flatMap*

- In addition to *map2*, we'll need something that will take an arbitrary number of segments, our old friend *sequence*:

need to convert List[Par[T]] into Par[List[T]]

`def sequence[T](tps: List[Par[T]]): Par[List[T]] = ???`

- In practice, however, there is *ParSeq*, *ParMap*, etc.

though Par classes does allow to use it without the use of Future

- In reality, I haven't found much use for these *Par*-type classes.

- And, also in reality, *Future[T]* isn't exactly like Java's *Future* object. It is much more flexible and powerful.

- You can learn much more about this idea from *Functional Programming in Scala*.

Futures (0)

- We've done enough on that algebra-design exercise.
Let's now talk about the real Scala *Future* object...

JVM & Bytecode doesn't know about type,
ex: `List<String> xs` : Compiler only knows about the address of List,
doesn't know about String type (Type eraser)
But, Scala's compiler knows about the type
We can query it for the run time class

Futures (1)

- *Future[T]* is a trait (and is also a monad!—no big surprise there)
 - What does that mean in practice?
Below is returned from Future:
`None`,
`Some(Success(T))`,
`Some(Failure(T))` // if the future results in exception
- The trait has some other very useful methods:
 - *value: Option[Try[T]]* ← **This is more or less the equivalent of `get`**
 - *onComplete[U](f: Try[T]=>U)(implicit ec: ExecutionContext): Unit*
 - *mapTo[S : ClassTag]: Future[S]*
- And *Future*'s companion object has some other good methods:
 - *def apply[T](t :=> T)(implicit ec: ExecutionContext): Future[T]*
 - *def firstCompletedOf[T](futures: IterableOnce[Future[T]])(implicit ec: ExecutionContext): Future[T]*
 - *sequence, reduce, foldLeft, fromTry, traverse, etc.*

```
val xfs: List[Future[Int]] = for {
  x <- List(1,2,3)
} yield Future(x)
// here we will have 3 future objects to test

val xsf: Future[List[Int]] =
Future.sequence(xfs)
// here we will only have 1 future of list
```

Futures (2)

- Let's open a connection to a web page and await the result:

```
scala> val url = new java.net.URL("http://www.htmldog.com/examples/")
url: java.net.URL = http://www.htmldog.com/examples/
scala> import scala.concurrent.Future
import scala.concurrent.Future
scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global
scala> import scala.util._
import scala.util._
```

We know that opening the connection will take a while. This statement, however, will return immediately. Meanwhile, we print our welcome

```
scala> val eventualConnection = Future(url.openConnection)
eventualConnection: scala.concurrent.Future[java.net.URLConnection] =
scala.concurrent.impl.Promise$DefaultPromise@439896bf
scala> println("welcome to my web crawler")
welcome to my web crawler
```

Now, we try to get the result. Oops!

```
scala> Try(eventualConnection.getInputStream)
<console>:27: error: value getInputStream is not a member of
scala.concurrent.Future[java.net.URLConnection]
Try(eventualConnection.getInputStream)
^
```

Futures (3)

- Of course, *Future* is a container! We have to actually get the *value* from it. But that value might not exist: we might have failed to open the connection.

```
scala> eventualConnection.value
res5: Option[scala.util.Try[java.net.URLConnection]] =
Some(Success(sun.net.www.protocol.http.HttpURLConnection:http://www.htmldog.com/
examples/))
```

- Note that the type of *value* is *Option[Try[URLConnection]]*. If action is not yet complete, we get *None*. If it's complete we get either *Success(u)* or *Failure(e)*.

Futures (4)

- Continuing...
 - We can first ask *eventualConnection.isCompleted*. Or we can await the result:

```
scala> import scala.concurrent._  
import scala.concurrent._  
scala> import scala.concurrent.duration._  
import scala.concurrent.duration._  
scala> Await.result(eventualConnection,100 millis)  
res1: java.net.URLConnection =  
sun.net.www.protocol.http.HttpURLConnection:http://www.htmldog.com/examples/
```

- But we won't normally use either method here. It's better to compose all of our *Futures* together and set up a function to act accordingly (a callback, actually).

```
scala> eventualConnection.onComplete {  
  case Success(_) => println("OK")  
  case _ => println("failed")  
}  
OK
```

Futures (5)

- Continuing...
 - Here, we compose a couple of futures using a for-comprehension:

```
scala> import scala.io.Source
import scala.io.Source
scala> for {
  connection <- Future(url.openConnection())
  is <- Future(connection.getInputStream)
  source = Source.fromInputStream(is)
} yield source.mkString
res18: scala.concurrent.Future[String] =
scala.concurrent.impl.Promise$DefaultPromise@efe19b1
```



It's a bit ugly having to wrap things in Future but, in practice, we will use something like Akka Http to do this sort of thing.

Futures (6)

- Review:
 - As much as possible, compose all of your *Future* objects together (use a for-comprehension or the *sequence* method);
 - As with *get* for *Option*, *Try*, you should **never** call *value* on a *Future* (instead, use a for-comprehension or set up a callback);
 - If you do call *value*, realize that the result can be any of three possibilities:
 - *None*
 - *Some(Success(x))*
 - *Some(Failure(e))*
 - Normally, you will only actually **await** the result of a single *Future* when to do otherwise will terminate your program. If you have more than one *Future* in your program then try to compose them into just one *Future* that you can await on.

Futures (7)

- Exercise (enter this into the REPL or [Scastie](#) or ScalaFiddle^{*}):

```
import scala.collection.immutable.IndexedSeq
import scala.concurrent._
import scala.concurrent.duration._
import scala.util._

import scala.concurrent.ExecutionContext.Implicits.global
val chunk = 10000 // Try it first with chunk = 10000 and build up to 1000000
def integers(i: Int, n: Int): LazyList[Int] = LazyList.from(i) take n
def sum[N : Numeric](is: LazyList[N]): BigInt = is.foldLeft(BigInt(0))(_+implicitly[Numeric[N]].toLong(_))
def asyncSum(is: LazyList[Int]): Future[BigInt] = Future {val x = sum(is); System.err.println(s"${is.head} is done with sum $x"); x}
val xfs = for (i <- 0 to 10) yield asyncSum(integers(i * chunk, chunk))
val xsf = Future.sequence(xfs)
val xf: Future[BigInt] = for (ls <- xsf) yield ls.sum
```

- It's your job to process the result *xf* appropriately.
- What extra statement must you provide if you are compiling/running this in a *main* program (or Scastie)?
- See also *FutureExercise* in the REPL.

* But, if you use ScalaFiddle you will have to revert LazyList to Stream

Future: usage

- Futures are used in many contexts:
 - In the source code for Spark (although not in your application code because of the Spark architecture);
 - interacting with Akka (HTTP, Actors or Streams);
 - *ad hoc asynchronous calls*, such as when opening a stream at a URL;
 - in Play applications;
 - Database interactions, such as Slick;
- Example: [Majabigwaduce](#) (map/reduce with actors)

3.7

Managing State

© 2021 Robin Hillyard



Continuing our discussion of mutability

- Whether we like it or not, applications typically have *state*.
- Remember how we defined push and pop in the Mill class (3.4 Mutable vs. Immutable)?
- We will continue that theme here.

Problem with Random Number Generation

- Random numbers (for testing)
 - are usually generated by a PRNG (pseudo-random-number-generator), e.g. *java.util.Random* scala.util.Random is a wrapper on java.util.Random
(it's essentially the same thing)
 - But, by definition, such a PRNG is **non-idempotent** and therefore **non-referentially-transparent**;
 - That's to say:
`random.getNext != random.getNext`
 - and this is anathema to functional programming.
 - In the first week, I referred to the “evils” of mutable state for testing purposes.

Problem with PRNGs continued

- Is there anything we can do about this?
 - Or, are we forever going to be limited by this lack of referential transparency?
- Let's think about the root cause of the problem?
 - It's that a mutable object can change its state without a referrer knowing about it—the referrer is *in the dark*:
 - There are two ways this can happen:
 - the mutable object *spontaneously* mutates (e.g. the system clock or a remote web service);
 - the mutable object is referenced in another thread and is updated there (e.g. draws the next random number);
 - In a testing framework, there are no guarantees about the order of execution of tests—therefore another test may draw a random number.

Carry your protection with you

- Maybe tortoises used to protect themselves by hiding under a rock—then they found it more convenient to carry the rock around with them on their backs ?!
 - BTW, I don't believe this of course!



What if we carry our state with us?

- Let's define the following trait:

```
trait RNG[A] {  
    def next: RNG[A]  
    def value: A  
}
```

- It has two properties: its (random) value and the next RNG in the series...
 - ... from which, of course, we can get another random value.
- What we've defined is like the old concept of a “one-time pad” used for setting an encryption key.
 - We only ever call *value* on an instance of *RNG once* (it always yields the same result because it's immutable)!
 - To get the next value in the series we invoke *next.value*

Using it in practice

- Here we test in a Spec file:

```
behavior of "RNG"
it should "allow predictable sequential usage" in {
    val r0 = LongRNG(0L)
    val r1 = r0.next
    r1.value shouldBe -4962768465676381896L
    val r2 = r1.next
    r2.value shouldBe 4804307197456638271L
    val r3 = r2.next
    r3.value shouldBe -1034601897293430941L
}
```

- Still, that's not super-useful. For practical purposes, we will need either:
 - a (mutable) *LazyList* of random values, batches of which can be consumed in different places; *Or*
 - an (immutable) *LazyList* of random values that can be consumed in one place.

Streamer

- I call the first of these a *Streamer* (a general concept):

```
/**  
 * This class is based on a mutable LazyList.  
 * Its purpose is like a one-time-pad: each value is yielded by the Streamer once and once only.  
 *  
 * @param s the LazyList  
 * @tparam X the underlying type and the type of the result  
 */  
case class Streamer[X] (private var s: LazyList[X]) extends (( )=>X) {  
    apply() // We need to skip over the first value  
  
    /**  
     * This method mutates this Streamer by resetting the value of s to its tail and returning its head.  
     * @return the head of the Stream  
     */  
    override def apply(): X = s match {  
        case x #:: tail => s = tail; x  
    }  
    def take(n: Int): Seq[X] = {  
        @tailrec def inner(xs: Seq[X], i: Int): Seq[X] = if (i==0) xs else inner(xs :+ this(), i-1)  
        inner(Seq.empty, n)  
    }  
=====  
behavior of "take"  
it should "work with random number generator" in {  
    val target = Streamer(RNG.values(LongRNG(0L)))  
    target.take(4) shouldBe Seq(-4962768465676381896L, 4804307197456638271L, -1034601897293430941L, 7848011421992302230L)  
}
```

LazyList of Random

- The second is simply an (immutable) *LazyList* of random numbers, generated from an *RNG[T]* instance:

```
trait RNG [T] {  
    /**  
     * @return the next random state in the pseudo-random series  
     */  
    def next: RNG [T]  
  
    /**  
     * @return the value of this random state (renamed from value)  
     */  
    def get: T  
  
    /**  
     * @return a lazy list of T values  
     */  
    def toLazyList: LazyList [T]  
}
```

Guess what?

- You're going to implement the second of these in an assignment.

3.8

Writing Good Scala code

© 2019 Robin Hillyard

A large, semi-transparent watermark of the Northeastern University logo is positioned in the lower right quadrant of the slide. The logo consists of a stylized 'N' formed by two overlapping arches, with the university's name 'Northeastern University' written in white serif capital letters to its right.

Northeastern
University

Making the transition from Java to Scala (1)

- I would estimate that 99.9% of Scala programmers were (or still are) Java programmers;
- There are some things that need getting used to:
 - Perhaps the number one thing is the use of `var`. Many new Scala programmers don't know how to do things without mutable variables and collections. That's not surprising. It takes training and a knowledge of FP to be able to avoid `vars`.
 - Immutable collections: Scala collections are immutable by default (you have to import `mutable.xxx` in order to get the mutating version). This takes a little getting used to and causes confusion about the operators such as `++`, `+=`, etc.
 - Equality: Java has primitives (`int`, `double`, etc.) while Scala doesn't. Therefore, Java provides two methods for testing equality: “`==`” for primitives and “`equals`” for objects. In Scala, the “`==`” method simply delegates to Java's `equals` method. If you want to test that two objects are the same object in Scala (rare), you use `eq`.

Making the transition from Java to Scala (2)

- More things that need getting used to:
 - Java loves to wrap everything in {} and separate statements with ";" —in Scala, you can eliminate most of this clutter.
 - In Java, you're allowed multiple returns from a method—in FP, this is very much frowned on. You don't need the *return* keyword in Scala—the final expression in a method or block is what is returned: if you ever find yourself using *return*, ask yourself why.
 - In Java8, functions are all defined as lambdas—but in Scala you can take advantage of the so-called *eta* expansion—that's to say that wherever Scala expects a function, you can provide a method (it's much clearer what's going on when you define your function as a method—for one thing a method has a name).
 - Defining classes in Scala: it's rare to define a straightforward class in Scala: a class is either an abstract class or a case class. You can do it, but there's no good reason to.

Making the transition from Java to Scala (3)

- Yet more things that need getting used to:
 - Class constructors: in Java it's common practice to provide several different constructors for a class. In Scala, you can do this too (syntax is different), but it's better practice to code these as additional "apply" methods in the companion object.
 - Case classes: Get used to using these for just about everything. If you just need a temporary way to combine values (say for the return from a method), you can use a tuple (you don't have to name the fields or give them types). But be aware that case classes are also tuples.
 - Modules: a single file in Scala can have any number of related classes (don't abuse this, though). Typically, you will have one (sealed) trait and a number of case classes which extend that trait. Each of these case classes may also have a companion object.

Making the transition from Java to Scala (4)

- Even more things that need getting used to:
 - In Java, it can be a pain to initialize a list. Not so in Scala. You can just write `List(1,2,3)`
 - How about adding an element to a list? In Java, you typically do that with *add* (a mutating method). In Scala, you do it immutably:

```
val xs1 = List(1)  
val xs2 = xs1 :+ 2
```

- Note that the `:` is always on the side of the collection.

```
val xs2 = 2 +: xs1
```

How to write good Scala code (1)

- Perhaps the first difference you notice from Java is the syntax:
 - No semi-colons (unless you need them to fit stuff on one line);
 - Fewer braces and parentheses (no-arg methods usually don't need parentheses);
- and built-in semantics:
 - Case classes, for instance, provide all the getters (possibly setters, too), *equals*, *hashCode*, *toString*, all that jazz—and the definition is really the constructor—and you don't need "new".
- In short, Scala code is a lot more compact than the equivalent Java code.
- So, try to use these improvements to the look-and-feel of the code to write really elegant, clear code.
- Have each class know about one domain and put all the methods that use that knowledge in the class—and have each method do only one simple thing.

The principle of Simple, Obvious, Elegant

- Most of the time when writing, say, a method in FP, you have a very limited set of variables which are in scope and, because each has some particular type, they can only be combined together in a small number of ways:
 - Don't be afraid to follow where your instinct is leading you.
 - Let the IDE guide you by using ctrl-space (or whatever): it will show you the possible expansions, with the types that they result in.

Getting the help you need

- Your first place to go for anything is: <https://www.scala-lang.org/>
- The definitive book is *Programming in Scala* (3rd edition) by Odersky et al.
- But the book that will really teach you what's going on is *Functional Programming in Scala* by Rúnar Bjarnesson and Paul Chiusano (the “red book”). It's fun to read but not for the faint of heart.
- The [Lightbend site](#) is also good (more application-oriented).
- Of course, [StackOverflow](#) is great for all Scala questions.
- There are some great blogs out there too:
 - [The Scala Times](#)
 - [The Neophyte's guide to Scala](#)
 - [\(my\) Scalaprof blog](#)

Updated: 2021-02-18

3.9

Containers, Collections, Wrappers, etc.

© 2019 Robin Hillyard



Real-life programming

- Unlike the Newton's method, for example, just about every other program involves grouping things together under one identifier.
- These groups are known as containers, collections and wrappers.

What is the difference between a container, a collection and a wrapper?

- A *collection* is a data structure that collects together some data objects (duh!).
- A *container* is a data structure that contains some data (duh!)
- So, how does a collection differ from a container?
- These terms overlap quite a bit, but usually:
 - a collection is made up of 0 thru many elements *of the same type*. Such elements are usually accessed by key or index.
 - a container is made up of *disparate elements* (a container is typically a tuple with one or more fields). Such elements are usually accessed by name.
- a *wrapper* is a specialized container that, conventionally, wraps either a single value or nothing at all.

What is a container?

- Collections and containers (including wrappers) are classes and are constructed by a constructor (!)
- If, for example, we say:
 - `val y = Bag(x)`
- what we mean is `val y = Bag.apply(x)` and that means construct a *Bag* with *x* as its contents. Typically (if *Bag* is a case class) that will be the same as `val y = new Bag(x)`.
- A case class is basically a tuple with named fields, and a bunch of compiler-provided methods.
- A wrapper may contain an element, or some indication of a non-element.

What about a collection?

- A collection is usually linear in its “shape”:
 - Elements can be accessed by position, by index, or by key (or all of these);
 - There may be zero through infinity elements;
 - All the elements conform to a particular “underlying” type.

Methods on these aggregate types

- *val aw: Wrapper[A]*
- How do we get the value out of *aw* when *aw* is a wrapper?
 - *val a: A = aw()* ?
 - No, we use *aw()* when *aw* is a function that take no parameters
 - *val a: A = aw get* ?
 - Yes, we use *aw get* when *aw* is a wrapper.
 - But! if *aw* is empty (or there was an error creating it), then the *get* method will typically throw an exception. We will learn therefore never to actually use *get*.

More methods on these aggregate types

- *val as: Seq[A]*
- How do we get a value out of as when as is a collection?
 - *val a: A = as(x) ?*
 - Yes, if x is an *Int* (for the index) or perhaps a key, then we can use *as(x)*.
 - This is short for *as.apply(x)*
 - But note that an exception might be thrown if x is not a valid index.
 - *val a: A = as method ?*
 - Yes, we use *as head* when we want the first element (the head).
 - But! if *as* is empty, then the *head* method will typically throw an exception. We will learn therefore to never actually use *head*, unless it's part of a pattern match.

Even more methods on these aggregate types

- *val as: Seq[A]*
- What about other types of result (different from *A*)?
 - *val x: Int = as method ?*
 - Yes, we can use *as size* to get the length of *as*.
 - *val b: Boolean = as method ?*
 - Yes, we use *as isEmpty* to find out whether *as* is empty.

Yet more methods on these aggregate types

- $\text{val as: Seq}[A]$
- What about other types of result (different from A)?
 - $\text{val xs: Seq}[A] = \text{as method}$?
 - We can use as tail to get the tail of as .
 - $\text{val ao: Option}[A] = \text{as get } x$?
 - An alternative to $\text{as}(x)$ is get (not available in all collections).
 - But note that what we actually get back is the value wrapped in *Option*.
- $\text{val xs: Seq}[A] = \text{as method predicate}$?
 - This method is *filter* which takes a predicate (a function which yields a *Boolean*) and returns a $\text{Seq}[A]$ which may be shorter (but not longer) than *this*.

Still more methods on these aggregate types

- $\text{val as: Seq}[A]$
- What about other types of result (different from A)?
 - $\text{val bs: Seq}[B] = \text{as method}[B] f \text{ where } f: A=>B?$
 - This method is *map*, one of the most important methods.
 - We don't normally need to explicitly state the [B] after the method name.
 - $\text{val bs: Seq}[B] = \text{as method}[B] f \text{ where } f: A=>\text{Seq}[B]?$
 - This method is *flatMap*, one of the most important methods.
 - We don't normally need to explicitly state the [B] after the method name.
 - $\text{val xs: Option}[A] = \text{as method predicate ?}$
 - This method is *find* which takes a predicate (a function which yields a *Boolean*) and returns an *Option[A]* which is *Some(x)* if found, else *None*.

3.10 Lazy Lists

© 2017-2023 Robin Hillyard



Non-strict collections

```
val xs = LazyList.from(0)
```

```
val ten = xs(10)
```

```
val four = xs(4)
```

```
// won't be evaluated again, since part of it was evaluated in ten
```

- There are three special types of collection which are *non-strict* (lazy to you and me):
 - *Iterator*: evaluates elements as needed but they cannot be revisited;
 - *LazyList*: evaluates element as needed and they can be revisited [*LazyList* was formerly *Stream* in 2.12 and earlier);
 - *SeqView* (actually, there are other types of view, too): essentially just “decorates” a collection with a transformation function
- Each of these has different behavior but what is generally common is that an element in the sequence will not be evaluated if you never actually need it.

LazyLists are lazy lists

- We've briefly mentioned *LazyList* before:
 - Like a *List*, a *LazyList* has a *head* and a *tail* but...
 - ...Unlike in a *List*, the *tail* (*and head*) of a *LazyList* are call-by-name parameters.

```
trait LazyList[A] {  
    def head: A  
    def tail: LazyList[A]  
}  
#::  
case class Cons[A](head:>A, tail:>LazyList[A]) extends LazyList[A]  
case object empty extends LazyList[Nothing] { Nothing is a subtype of every type  
    def head: throw NoSuchElementException("head of empty lazy list")  
    def tail: throw UnsupportedOperationException("tail...lazy list")  
}
```

- A *LazyList* is ideal for memoizing something.

Working with LazyLists

- Ways to create a *LazyList*:
 - `import LazyList._`
 - `1 #:: 2 #:: empty`
 - `cons(1, cons(2, empty))`
 - `from(1)` library implementation includes `iterate`
 - `continually(9)` gives a list with all elements = 9
 - `range(1, 20, 3)` 1, 4, 7, 10, 13, 16, 19
- A *LazyList* has no definite length.
 - In order to turn an (infinite) *LazyList* into a (finite) *List*, you need to force a definite size. Sometimes, it's convenient to convert it to a *List*:
 - `from(1) take 10 to List`

`take`: converts *LazyList* without size => *LazyList* with known size
`to` : converts *LazyList* with size => *List* (ofc it has size)

What do you think this function does?

Fibonacii Numbers

Note: recursive
even though **f** is
a val.

A bit like
foldLeft but
retains shape

```
val f: LazyList[Long] = 0L #:: f.scanLeft(1L)(_ + _)
```

$$f(n) = f(n - 1) + f(n - 2)$$

```
val g: LazyList[Long] = 0L #:: 1L #:: g.zip(g.tail).map (n =>  
n._1 + n._2)
```

Should be a bit
easier to
understand.

Fibonacci

```
scala> val f: LazyList[BigInt] = BigInt(0) #:: f.scanLeft(BigInt(1))(_ + _)
f: LazyList[BigInt] = LazyList(0, ?)
```

```
scala> f zipWithIndex take 100 foreach println
```

```
(0,0)          def zipWithIndex[X] (xs : Seq [X], Seq [(Int, X)]:
(1,1)          xs zip LazyList.from (0)
(2,1)
(3,2)
(4,3)
(5,5)
(6,8)
(7,13)
(8,21)
(9,34)
etc. etc.
(99,218922995834555169026)
```

Changes in 2.13

- *Stream* has been deprecated in favor of *LazyList*.
 - In *LazyList*, both *tail* and *head* are lazily evaluated.
- There's no *Traversable* any more:
 - They decided that *Iterable* and *Traversable* were so similar that it wasn't worth maintaining a distinction.
- *StringOps* has methods *toIntOption*, etc.
 - Converting collections is now done as follows:
 - `xs.toList`, or [xs to List](#)
 - `xs to List`

`"A" . toIntOption => None`
`"1" . toIntOption => Some(1)`

3.11

Pattern Matching

The “dark” side of expressions

© 2018, 2021 Robin Hillyard



A deeper dive into Pattern Matching

- We looked at pattern matching in 2.3 Functional Programming in Scala.
- Let's do a “deeper dive” into the subject and see how expressions and pattern-matching are really two sides to the same coin.

Expressions

- Expressions
 - One of the central concepts in almost all languages is the *expression*.
 - An expression is essentially a function which takes some number of variables/constants/expressions and yields a result, e.g. $f(x,y,z)$ or $x+z^2$. Remember that (in Scala) this is just a human-readable form of $x.+(z.^*(2))$.
 - In procedural languages, expressions form the right-hand-sides of statements such as assignments, return statements, constant declarations, the branches of an if/else/for/while/do construct, etc.
 - In Scala, the yield (return value) of *every part of the language is an expression*. The only exceptions are:
 - declarations of traits/classes/objects, methods and variables;
 - other non-expression constructs: import, package, “type” statements.

Patterns

- *Patterns* are the antithesis (i.e. opposite) of expressions and are found only in functional programming languages.
- A pattern allows you to take a variable and *produce* several variables (exactly the opposite of what an expression does).
- Let's look at an example:

```
def decode(x: (Int, String)): Unit =  
  x match {  
    case (k, v) => println(s"$k:$v")  
  }
```

- In this example, we are pattern-matching on a *Tuple* of *Int* and *String*. We only need one case whose pattern looks exactly like an expression forming a tuple from *k* and *v*. But it is a pattern instead and so *k* and *v* are extracted extractor pattern from the given tuple *x*, and become in-scope variables which can be used in the expression on the right-hand side, in this case *println*...
- Thus, a pattern which looks like an expression is actually an *extractor*.

Patterns (1)

- Let's look at another example:

```
def print(xs: List[Int]): Unit =  
  xs match {  
    case h :: t => println(h); print(t)  
    case _ =>  
  }
```

- In this example, the construct $h :: t$ is a pattern. It looks *exactly* like an expression BUT instead of h and t being existing in-scope variables which will be applied to the $::$ operator to yield a value, instead the value is matched against the pattern and, if there is a match, variables h and t are, magically if you like, declared and in scope.
- In this situation, it is the variable xs which is matched against the patterns—in the order defined—and, if a match is found, the expression on the RHS of the “ $=>$ ” (“rocket symbol”) is invoked. Here, h is passed to $println$ and t is passed, recursively, to $print$.

Patterns (2)

- Here again is our example:

```
def print(xs: List[Int]): Unit =  
  xs match {  
    case h :: t => println(h); print(t)  
    case _ =>  
  }
```

- There is also a catch-all pattern “`_`”, although any identifier name would do, such as “`a`”, “`x`”, etc. The unique thing about the “`_`” is that the pattern matches as it would to an identifier, but no variable is synthesized.
- I need to explain what $h :: t$ actually represents (whether as an expression or a pattern).
 - In the context of an expression, “`::`” is actually the case class “`::`” (which we pronounce “`cons`”) or, more precisely, the auto-magically generated *apply* method of the companion object “`::`”

Patterns (3)

- Here again is our example (but slightly altered):

```
def copy(xs: List[Int]): List[Int] =  
  xs match {  
    case h :: t => h :: t  
    case Nil => Nil  
  }
```

- When we write $h :: t$ as an expression the compiler actually expands this into $::apply(h, t)$.
- But when we write $h :: t$ as a pattern, a completely different mechanism is used. The compiler invokes the following method on “ $::$ ”:
 - $unapply[X](xs: List[X]): Option[(X, List[X])]$ which results in an (optional) tuple of two parameters: an X (the head) and a $List[X]$ (the tail). Any object can have an $unapply$ method, but the magic of case classes is that they auto-generate the appropriate $unapply$ method. That's to say that case classes are designed for use in pattern-matching!
 - Can you see that $h :: t$ as an expression and $h :: t$ as a pattern are simply two sides to the same coin? One expands to $::apply(h, t)$ and the other, effectively, to $val (h,t) = ::unapply(xs).get$

Patterns (4)

- Here again is our example (but altered again):

```
def reverse(xs: List[Int]): List[Int] =  
  xs match {  
    case h :: t => reverse(t) :+ h  
    case Nil => Nil  
  }
```

- A **case class** is just like an ordinary class but it has some very special additional properties. Our particular case class is defined in the [API](#) thus:

`final case class ::[B](head: B, tl: List[B]) extends List[B] with Product with Serializable`

- The additional properties that the compiler provides for a case class are:

- the members of the case class are, by default, available as methods of the class (e.g. `xs.head`); the case class instances also behave as *Tuples* (as provided for by the `Product` trait); Product trait allows us to examine member position of a case class
- `toString`, `equals` and `hashCode` methods are all automagic;
- the companion object provides an *apply* method (which effectively obviates the need to use the “new” keyword) and an *unapply* method which, given an instance of the case class, will yield an optional tuple of the parameters. **This** is how the variables `h` and `t` are automagically generated from the `h :: t` pattern.

Patterns (5)

- Back to patterns. What other sorts of patterns are there?
 - constant patterns, for example using *Nil* (the empty list):

```
xs match {
  case Nil =>
    case h :: t => println(h); print(t)
}
```
 - typed patterns, for example (the parentheses are not always required) where we are only interested in *Int* values:

```
xs match {
  case Nil =>
    case (h: Int) :: t => println(h); print(t)
  case _ =>
}
```
 - arbitrarily-nested patterns, for example:

```
xs match {
  case Nil =>
    case h :: k :: t => println(h+k); print(t)
  case _ =>
}
```
 - guarded patterns, for example:

```
xs match {
  case Nil =>
    case h :: t if h > 0 => println(h); print(t)
  case _ => println("head was not positive")
}
```
- variable patterns, for example, matching an in-scope variable *y*:
- ```
val y = 2
x match {
 case `y` => print("matched")
```
- alternative patterns
- ```
x match {
  case 1 | 2 | 3 | 5 | 7 | 11 => println("Prime numbers")
}
```
- identified patterns, for example:
- ```
Complex(1, 1) match {
 case q @ Complex(r, i) => println(q)
}
```

# Where can we use patterns?

- The most obvious context for a pattern is after the *case* keyword in a *match* clause.
- But, don't forget about patterns after **val** or before “**<-**” in a *for-comprehension*:

```
case class Complex(real: Double, imag: Double)
val pi = Complex(-1, 0)
val Complex(r, i) = pi
for (x <- xs) println(x)
```
- Do you see the elegant symmetry of the two middle lines? In particular, note that:
  - Every value on the *right* of the “**=**” must evaluate to something;
  - Every value on the *left* of the “**=**” actually defines a “bound variable”.

# Wrap-up

- We will come across patterns again when we talk about for comprehensions. Note that you can see the details of my examples in running code in the REPO. Just look for *PatternExample*.

# 3.12 Scala, O-O and Java: Part 2

© 2019 Robin Hillyard



# Scala and functional programming

- There are five key features of Scala:
  - Functional Programming;
  - Object-oriented;
  - Strict types (with type inference);
  - Java virtual machine;
  - Implicits.
- We already discussed Functional Programming.

# Object-oriented

- Type hierarchy—basically the same as for Java;
- Polymorphism, encapsulation, all that good stuff;
- Classes, traits, objects.
  - traits instead of Interface (traits can have default implementations, similar to what's now in Java8);
  - “objects” in Scala are singleton classes (similar to marking stuff static in Java);
  - Also “case” classes:
    - Provide all of the standard boiler-plate code, including *unapply* for pattern-matching;
- There are no “primitives” in Scala (at least you don't have to be aware of them);
- The Class Loader (doesn't distinguish between Scala and Java code);
- The ubiquitous “dot” operator.

# Static Types

- Parametric (as opposed to generic) types
  - Liskov substitution principle (defines what sub- and super-types mean);
  - Full treatment of variance (where  $S$  is a sub-type of  $T$ ):
    - Invariant:  $\text{Array}[T]$  is neither subtype nor supertype of  $\text{Array}[S]$
    - Covariant:  $\text{List}[S]$  is a subtype of  $\text{List}[T]$
    - Contravariant:  $T \Rightarrow U$  is a subtype of  $S \Rightarrow U$
  - This allows us always to be very strict about types (unlike Java or Python, for example)
- Type classes and other type constructors
  - $T[X]$
- **Once you have succeeded in compiling a Scala program, it usually does what you want (run-time errors are less common than with Python or Java).**

# Java Virtual Machine

- Scala wouldn't be a serious contender in the language space if it didn't run on the JVM.
- Running on the JVM gives access to thousands of Java libraries, especially important in the early days of Scala.
- The JVM, lazy evaluation, functions, class loader, etc. are what powers Spark. There's nothing magic in Spark.

# Implicits

- One of the big problem areas in any language is dealing with (configuring, searching) “global” variables;
- Implicits not only provide a solution to these global variables, but they give us many other benefits (e.g. “extension” methods);
- Type classes;
- Implicit classes and conversions (instead of just “widening” as in Java).

# What's different from other FP languages?

- Scala does allow *var* (mutable variable) and mutable collections:
  - but you are discouraged from using them! And you really don't need them!
- Scala has similar ways of defining new types (*Option*, *Either*, etc.) as well as “type constructors” and lots of arcane stuff that you will never actually come across—however,
  - Scala also allows you to define new types by inheritance (just like O-O)
- Scala doesn't actually have a monadic type—
  - if you really want that, you have to use one of the libraries such as ScalaZ

# What's different from Java?

- Scala avoids *nulls* and exceptions via built-in types:
  - *Option[X]* which has two cases: *Some(x)* and *None*;
  - *Try[X]* which has two cases: *Success(x)* and *Failure(e)*;
- Other “union” types like *Either[L,R]*;
- *Future[X]* is used to handle asynchronous results
  - different from Java’s *Future*.
- Scala doesn’t have primitives like *int, double*;
- Collections are different;
- Function types are much more general than Java8’s function types:
  - multiple parameters easy to define;
  - “curried” functions for example;
- Other types, e.g. *String, Array, MyClass*, are the same;
- Traits can have default methods (Java8 introduced this);
- Type *inference* (now available in Java9).

# What's different from Java (continued)?

- You can define related types (traits, classes, etc.) in one module
  - i.e. without having to make private inner classes;
- There are no static (“class”) methods: all singletons\* are “objects”;
- In Java, you can’t return more than one value from a method unless you go to the trouble of defining a class—in Scala you just return a tuple.
- In Java, generics are kind of a mess (they were an after-thought).
  - You can cheat, or you can just make everything a “?”.
  - In Scala, parametric types are very strictly enforced. So, you can’t be surprised at run-time by a value that doesn’t conform to the proper type.

\* Have we talked about singletons?

# Working with REST APIs

# Scala Requests

- Requests-Scala is a Scala port of the popular Python Requests HTTP client
- Requests-Scala exposes the `requests.*` functions to make HTTP requests to a URL.
- The main advantage is no boilerplate code, simple setup, and intuitive and simpler API. ( lesser and easier code)
- Can be used instead of other libraries such as Scala Play
- Detailed explanation at URL : <https://github.com/com-lihaoyi/requests-scala>

# Sample Get request

```
val r = requests.get("https://api.github.com/users/rchillyard")
r.statusCode
// 200
r.headers("content-type")
// Buffer("application/json; charset=utf-8")
r.text
// prints the text
```

# Output

```
{
 "login": "rchillyard",
 "id": 4192733,
 "node_id": "MDQ6VXNlcjQxOTI3MzM=",
 "avatar_url": "https://avatars.githubusercontent.com/u/4192733?v=4",
 "gravatar_id": "",
 "url": "https://api.github.com/users/rchillyard",
 "html_url": "https://github.com/rchillyard",
 "followers_url": "https://api.github.com/users/rchillyard/followers",
 "following_url": "https://api.github.com/users/rchillyard/following{/other_user}",
 "gists_url": "https://api.github.com/users/rchillyard/gists{/gist_id}",
 "starred_url": "https://api.github.com/users/rchillyard/starred{/owner}{/repo}",
 "subscriptions_url": "https://api.github.com/users/rchillyard/subscriptions",
 "organizations_url": "https://api.github.com/users/rchillyard/orgs",
 "repos_url": "https://api.github.com/users/rchillyard/repos",
 "events_url": "https://api.github.com/users/rchillyard/events{/privacy}",
 "received_events_url": "https://api.github.com/users/rchillyard/received_events",
 "type": "User",
 "site_admin": false,
 "name": "Robin Hillyard",
 "company": "Northeastern University",
 "blog": "www.phasmidsoftware.com",
 "location": "Boston, MA",
 "email": null,
 "hireable": true,
 "bio": "Scala, Spark and Java developer and teacher (Northeastern University). Open source developer of many Scala libraries.",
 "twitter_username": "PhasmId",
 "public_repos": 46,
 "public_gists": 0,
 "followers": 137,
 "following": 5,
 "created_at": "2013-04-18T15:05:53Z",
 "updated_at": "2022-09-17T20:34:20Z"
}
```

# Getting Data & Processing it using uJson

```
@ val resp = requests.get(
 "https://api.github.com/repos/lihaoyi/upickle/issues",
 params = Map("state" -> "all"),
 headers = Map("Authorization" -> s"token $token")
)
resp: requests.Response = Response(
 "https://api.github.com/repos/lihaoyi/upickle/issues",
 200,
 "OK",
 [{"url": "https://api.github.com/repos/Lihaoyi/upickle/issues/294", "repository_url": ...
@ resp.text()
res4: String = "[{\\"url\\": \"https://api.github.com/repos/Lihaoyi/upickle/issues/620\", ...
```

# JSON Parsing

```
@ val parsed = ujson.read(resp.text())
parsed: ujson.Value.Value = Arr(
 ArrayBuffer(
 Obj(
 Map(
 "url" -> Str("https://api.github.com/repos/Lihaoyi/upickle/issues/620"),
 "repository_url" -> Str("https://api.github.com/repos/Lihaoyi/upickle"),
 ...
)
)
)
)
@ println(parsed.render(indent = 4))
[
 {
 "id": 449398451,
 "number": 620,
 "title": "Issue with custom repositories when trying to use Scoverage",
 "user": {
 "Login": "jacarey",
 "id": 6933549,
 }
 ...
]
```

# URL parameters

- If you try the above code, you will receive only 30 records.
- This is because the API is paginated.
- In order to fetch all the records, we have to pass a ?page parameter
- ?page=1,?page=2, ?page=3, stopping when there are no more pages to fetch.
- This can be done by using a simple while loop.

# Code

```
@ def fetchPaginated(url: String, params: (String, String)*) = {
 var done = false
 var page = 1
 val responses = collection.mutable.Buffer.empty[ujson.Value]
 while (!done) {
 println("page " + page + "...")
 val resp = requests.get(
 url,
 params = Map("page" -> page.toString) ++ params,
 headers = Map("Authorization" -> s"token $token")
)
 val parsed = ujson.read(resp.text()).arr
 if (parsed.length == 0) done = true
 else responses.appendAll(parsed)
 page += 1
 }
 responses
}
```

## Note

- The Map("key"-> "value") is a shorthand syntax for a tuple ("key", "value").
- The argument params of type (String, String)\* allows us to pass an arbitrary number of key- value pairs.
- There is no need to manually wrap them as a Seq.

# Extracting Data

```
@ val nonPullRequests = issues.filter(!_.obj.contains("pull_request"))
```

```
@ val issueData = for (issue <- nonPullRequests) yield (
 issue("number").num.toInt,
 issue("title").str,
 issue("body").str,
 issue("user")("login").str
)
```

# Output.

```
issueData: collection.mutable.Buffer[(Int, String, String, String)] = ArrayBuffer(
 (
 272,
 "Custom pickling for sealed hierarchies",
 """Citing the manual, "sealed hierarchies are serialized as tagged values,
 "
```

Updated: 2024-02-06

4.0

# Functional Composition

© 2017-2024 Robin Hillyard



Northeastern  
University

# How do we declare a function?

- There are essentially three ways to yield a function:
  - Lambda (typically for simple functions)
    - Example:

```
val f: Int => Int = _ * 2
```

eta
  - $\eta$ -conversion of method (for more complex functions)
    - Example:

```
def doubleIt(x: Int): Int = x * 2
val f = doubleIt _
```

partially applied function
  - Composition, i.e., the result of a higher-order method
    - Example: `val h = f andThen g`

val f: Int => Int = x => x \* 2  
val g: Int => String = x.toString

// below values are the same:  
val h: Int => String = g(f(x))  
val h: Int => String = g compose f  
val h: Int => String = f andThen g

# 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
    - g compose f
  - 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:

- Or perhaps we want a curried version:

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

# 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*” (correct) 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)) .unapplySeq() , we get back Option(Tuple)
 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

Option.when(b: Boolean)(t => T)  
val to: Option[T] =

```
val x = to.getOrElse("Hello")
val xo = to.orElse("Hello")
```

- 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* or 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 inside *Try.apply*.

Lift, map2, flatMap,  
“for comprehensions”

Fasten your seat belts!

# First: let's bake some cookies

- 1 cup brown sugar
- 2 eggs
- 2 teaspoons vanilla extract
- 1 teaspoon baking soda
- 2 teaspoons hot water
- $\frac{1}{2}$  teaspoon salt
- 3 cups flour
- 2 cups chocolate chips
- 1 cup chopped walnuts

# Let's bake some other cookies

- 1 cup brown sugar
- 2 eggs
- 2 teaspoons vanilla extract
- 1 teaspoon baking soda
- 2 teaspoons hot water
- $\frac{1}{2}$  teaspoon salt
- 3 cups flour
- 2 cups chocolate chips (optional)
- 1 cup chopped walnuts (optional)

# 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

# cToF

```
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 a function  $X \Rightarrow Y$ , and return an *Option[X] => 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  $f: A \Rightarrow B$ , we could create a function  $g: Option[A] \Rightarrow Option[B]$  ?
  - 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, the *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!!**
- Hopefully, you can see how to desugar this expression:  
 $a \Rightarrow a \text{ map } f$
- In other words, take whatever is the input and invoke *map* on it, passing in the parameter *f*.

# 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  $(A, B) \Rightarrow C$  whereas we need an  $((A, B)) \Rightarrow C$

# Option and Try—in greater depth

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

```
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, age.toIntOption))
 case _ =>
 Failure(new Exception(s"parse error in Rating: $s"))
 }
}
```

r allows compiler to understand this as a regular expression

in """ ... """ triple quoted expression, we don't need to add escape character, ex: \\\d , just as \d

# 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)`
    - 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.

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

# 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  $\text{Seq}[X]$  (or  $\text{List}[X]$ , etc.) would be called  $xs$ . This much is totally standard in Scala. The rest is non-standard: my own scheme:
    - So,  $xy$  represents a  $\text{Try}[X]$  (we use "t" for a  $\text{Tuple}$ );
    - $xo$ :  $\text{Option}[X]$
    - $kvm$  (or  $kVm$  or  $k\_vm$  or even  $'k,vm'$ ) is used for a  $\text{Map}[K,V]$  (here, the type parameters of  $\text{Map}$  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] = ??? }
```

```
this match {
 case Some(value) => f(value)
 case None => None
}
```

# 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)

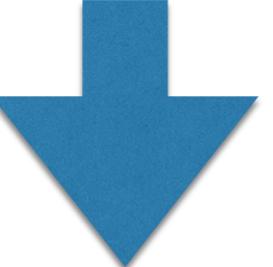
- Suppose we substitute for *flatMap* in the previous slide...
- And where we see *f(a)* we substitute *map* applied to *bo*.
- We'll call this new method “*map2a*”:

```
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)))
```

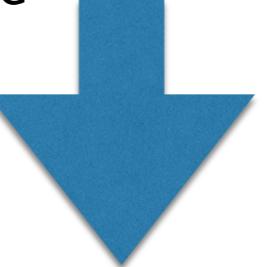
# A better way... (2h)

- 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): Try[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): List[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  $A \Rightarrow B$  function and returns a  $F[A] \Rightarrow F[B]$  function where  $F$  is some container type like *Option*, *Try*, *List*, etc. (a “functor”)
  - 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  $(A, B) \Rightarrow C$  and returns an  $M[C]$ , where  $M$  is a container as above (a “monad”)
  - 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

```
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)
 }
}
```

we can't have diff monads in the same container

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

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*:

```
xos.foldLeft(Some(Seq.empty[X]): Option[Seq[X]])
{ (acc, xo) =>
 for {
 xs <- acc
 x <- xo
 } yield xs :+ x
}
```

```
xos.foldRight(Some(Seq.empty[X]): Option[Seq[X]])
{ (xo, acc) =>
 for {
 xs <- acc
 x <- xo
 } yield x +: xs
}
```

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

- this method should iterate through *xos* and, if all elements are *Some(x)*, collect them into a sequence *xs* then return *Some(xs)*. If any of the elements are *None*, return *None*.
  - We're not quite ready to implement this one.

```
xs.foldRight(Some(Seq.empty[Y]): Option[Seq[Y]]) { (x, acc) =>
 for {
 ys <- acc
 y <- f(x)
 } yield y +: ys
}
```

- 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]] = ???
```

```
xs.foldLeft(Some(Seq.empty[Y]): Option[Seq[Y]]) { (acc, x) =>
 for {
 ys <- acc
 y <- f(x)
 } yield ys :+ y
}
```

# In general, lots of these functional compositions

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

Updated: 2021-02-25

4.1

## The Substitution Principle

© 2017 Robin Hillyard



# Floating Point Problem

```
def evaluate_3_tenths = 1.0/10 + 2.0/10

def multiply_by_10_over_3(x: Double) = x / 3 * 10

"doublePrecision" should "work properly" in {
 val x = FunctionalProgramming.evaluate_3_tenths
 val y = FunctionalProgramming.multiply_by_10_over_3(x)
 y should be(1)
}
```

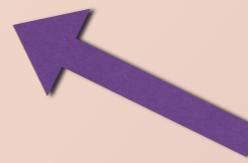
This test fails. Why? Because, unlike pure algebraic numbers, floating point numbers are *not associative*.

That's to say that  $(a+b)+c \neq a+(b+c)$ , at least in general. Much of the time, it will hold true but not always.

# Floating Point Solution 1

ScalaTest:

```
"Floating Point Problem" should "be OK" in {
 val x = Rational(1,10)+Rational.normalize(2,10)
 val y = x * 10 / 3
 y shouldBe 'unity
}
```



*Rational* is a class defined in the class git repo.  
There is also a chapter on a *Rational* class in the text book.

ScalaTest works fine for particular cases. We could do more testing to cover the domain, especially using *scalacheck*. But none of this is as good as...

# Floating Point Solution 2

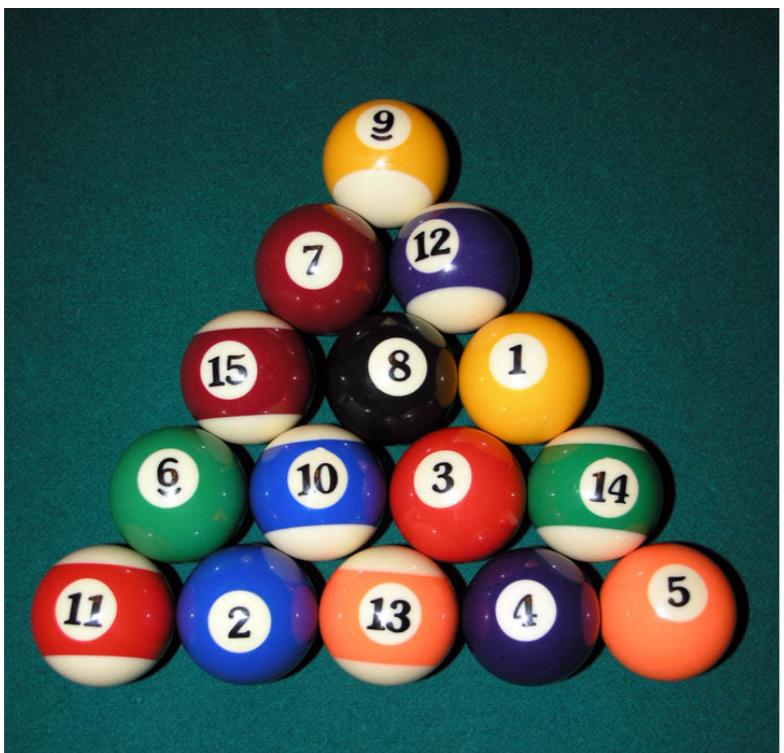
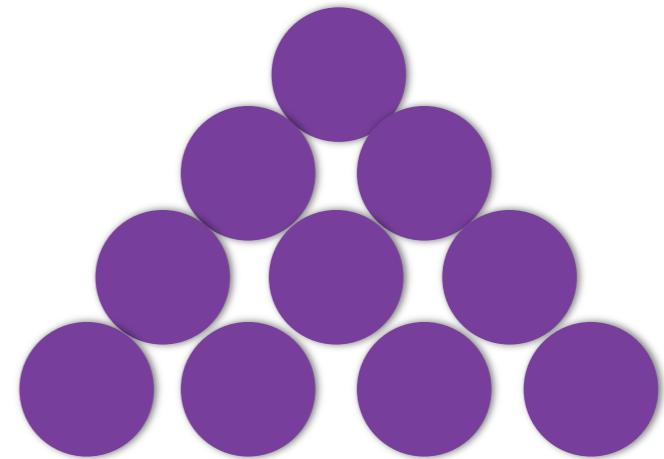
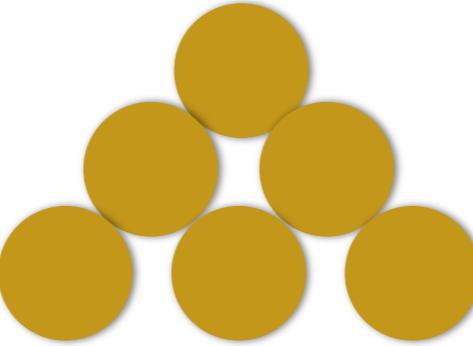
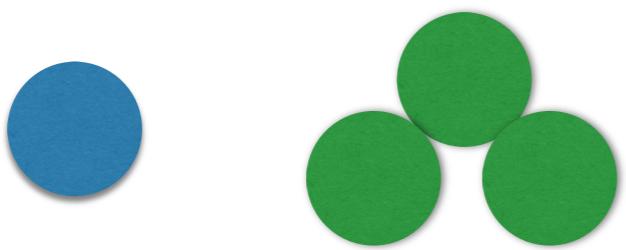
We can do better with substitution\*:

- (1) Rational.normalize(2,10) → Rational(2/gcd(2,10),10/gcd(2,10))  
→ Rational(2/gcd(2,0),10/gcd(2,0)) → Rational(2/2,10/5) →  
Rational(1,5)
- (2) Rational(1,10)+Rational(1,5) →  
Rational.normalize(1\*5+10\*1,10\*5) → Rational.normalize(15,50) →  
Rational(15/gcd(15,50),50/gcd(15,50)) → Rational(3,10)
- (3) Rational(3,10) \* 10 → Rational(3,1)
- (4) Rational(3,1) / 3 → Rational(1,1)
- (5) Rational(1,1).isUnity → 1==1 && isWhole → 1==1 && 1==1 → true

\* The “Substitution Model” is an important aspect of functional programming which we will refer to again and again.

# The Substitution Model

- Formalized in the  $\lambda$  calculus, introduced by Church (1932)
- Can be used to **prove** the equivalence of expressions *provided* all expressions reduce to “pure” functions (or constants), i.e. **there are no side-effects!**
- Termination: not all expressions can be substituted in a finite number of steps: e.g. `def x = x`
- In which order should we evaluate expressions?
  - `val x = 3*4*5`: two options:  $(3*4)*5$  and  $3*(4*5)$
  - `val x = square(3+4)`: two options:  $(3+4)*(3+4)$  and  $7*7$ 
    - These are known as call-by-name and call-by-value
    - Which do you think is fastest? Best?



[https://commons.wikimedia.org/wiki/File:Eight\\_Ball\\_Rack\\_2005\\_SeanMcClean.jpg](https://commons.wikimedia.org/wiki/File:Eight_Ball_Rack_2005_SeanMcClean.jpg)

|    |    |    |    |    |    |
|----|----|----|----|----|----|
| 9  | 5  | 4  | 13 | 2  | 11 |
| 7  | 12 | 14 | 3  | 10 | 6  |
| 15 | 8  | 1  | 1  | 8  | 15 |
| 6  | 10 | 3  | 14 | 12 | 7  |
| 11 | 2  | 13 | 4  | 5  | 9  |

$$N(N+1)$$

Sum of 1 thru N

# Side-bar on “Proof by Induction”

- Consider the “discrete sum” of consecutive whole numbers:
  - $P(n) = 0 + 1 + 2 + \dots + n$
  - I will assert (see previous slide) that  $P(n) = n(n+1)/2$  for any positive integer  $n$ .
    - This seems to work for  $n = 4$ :  $0+1+2+3+4 = 10 = 4(5)/2$
    - But can we *prove* it for all  $n$ ?
- Proof by induction involves proving, independently, two cases: the base case and the inductive step:
  - Base case:  $n = 1$ 
    - $0 + 1 = 1$  and  $1(1+1)/2 = 1$
  - Inductive step:
    - *if  $P(n) = n(n+1)/2$  then  $P(n+1)$  should equal  $(n+1)(n+2)/2$*
    - i.e.  $(n+1)(n+2)/2 - n(n+1)/2$  should equal  $n+1$
    - i.e.  $(n+2-n)(n+1)/2$  should equal  $n+1$
    - i.e.  $2(n+1)/2$  should equal  $n+1$
- Now, we have proven that the identity holds for  $n=1$  and that, if it holds for  $n$ , it also holds for  $n+1$
- Therefore:  $P(n) = n(n+1)/2$



# Substitution proof

- Let us prove a definition of *List.length*:

```
package edu.neu.coe.csye7200.list

trait List[+A] {
 def length: Int
}

case object Nil extends List[Nothing] {
 def length: Int = 0
}
:: ex: x = h :: t
case class Cons[+A](head: A, tail: List[A]) extends List[A] {
 def length: Int = 1 + tail.length
}

object List { A* means Seq[A]
 def apply[A](as: A*): List[A] =
 if (as.isEmpty) Nil
 else Cons(as.head, apply(as.tail: _*))_* means convert to Seq of arguments
}
```

- By *induction*: two parts:

- Prove the case for Nil (0)
- Prove that: *if* it holds for list of length  $N$ , *then* it holds for list of length  $N+1$

Do this proof by induction. You may confer.



# Substitution proof (2)

- (1):  
 $\text{Nil.length} \rightarrow 0$
- (2):
  - Assume that  $/\text{listN.length}$  yields the length of  $\text{ListN}$ , i.e.  $N$   
 $\text{Cons("a", listN).length} \rightarrow 1 + \text{listN.length} \rightarrow 1 + N$
  - Statement is proved because of (1) and (2)

# Call-by-name/value

- Remember from a few slides ago?
  - In which order should we evaluate expressions?
    - `val x = 3*4*5`: two options:  $(3*4)*5$  and  $3*(4*5)$
    - `val x = square(3+4)`: two options:  $(3+4)*(3+4)$  and  $7*7$ 
      - These are known as call-by-name and call-by-value
      - Which do you think is fastest? Best?
  - Did you decide what was best?

# Call-by-name/value

- If both CBN and CBV terminate, then the result is equivalent...
  - So, does it matter which we use?
  - It can matter a lot!!
  - if CBV terminates, then so must CBN (but not other way around)
  - `def zip[A,B](x: Seq[A], y: Seq[B]): Seq[(A,B)]` 

“Seq” is a Trait extended by List, LazyList, etc.

```
Welcome to Scala version 2.11.6 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_05).
Type in expressions to have them evaluated.
Type :help for more information.
```

```
scala> val x = List("a","b")
x: List[String] = List(a, b)
```

```
scala> val y = LazyList from 1
Y: scala.collection.immutable.LazyList[Int] = LazyList(1, ?)
```

```
scala> x.zip(y)
res6: List[(String, Int)] = List((a,1), (b,2))
```

# Strict/non-strict functions

- First off—a question:
  - You've been tasked with implementing a method which takes two boolean arguments and yields the “and” of the two values.
  - However, with a mind to efficiency, your “customer” says that he doesn't want to evaluate the second argument if the first one is false.
  - Your first idea is this:

```
def and(a: Boolean, b: Boolean): Boolean =
 if (a) b else false
```
  - How can you change the definition to achieve your goal?

# Non-strict (lazy) functions

```
scala> def and(a: Boolean, b: Boolean): Boolean = if (a) b else false
and: (a: Boolean, b: Boolean)Boolean
```

```
scala> def myTrue = {println("hello"); true}
myTrue: Boolean
```

```
scala> and(false, myTrue)
hello
res9: Boolean = false
```

Note that when we define *and* this way, we have to “partially apply” the value of *myTrue*.

```
scala> def and(a: Boolean, b: () => Boolean): Boolean = if (a) b() else false
and: (a: Boolean, b: () => Boolean)Boolean
```

```
scala> and(false, myTrue _)
res10: Boolean = false
```

However, we did have to explicitly invoke *b* to yield a *Boolean* from a  $\Rightarrow \text{Boolean}$ .

- Note that we replaced the “normal” value-type Boolean parameter “*b*” with a function that takes no arguments but returns a Boolean.
- We could also write the “*and*” method and its invocation thus:

```
def and(a: Boolean, b: => Boolean): Boolean = if (a) b else false
and(false, myTrue)
```

Syntactic sugar for the form involving *Function0[Boolean]*

# Type inference with isomorphism

- Let's say we have a *List* of *Ints* and we convert it to a list of *Strings*:
  - `val x = List(1,2,3)`
  - `val y = x map {n => n.toString}`
  - The compiler knows that *y* is a *List[String]* because of **Type Inference\*** so we don't have to explicitly write `val y: List[String]=...`
- Let's say we need a greatest common denominator method:
  - `@tailrec private def gcd(a: Long, b: Long) = if (b==0) a else gcd(b, a % b)`
  - The compiler cannot figure out what the return type of *gcd* is supposed to be
  - `@tailrec private def gcd(a: Long, b: Long): Long = if (b==0) a else gcd(b, a % b)`
  - More on this later...

However, explicitly specifying types can be a very helpful aid to getting programs to compile

We must explicitly specify the return type of a recursive method

\* and because of type isomorphism, *map* retains the “shape” of *x* in *y*

# Type inference/isomorphism continued

- We just saw this (where annotating the type of *y* is optional):
  - `val y: List[String] = List(1,2,3) map (_.toString)`
- But we could also have written any of the following:
  - `val y: Seq[String] = Seq(1,2,3) map (_.toString)`
  - `val y: LazyList[String] = LazyList.from(1) map (_.toString)`
  - `val y: Option[String] = Option(1) map (_.toString)`
- But there is really only one definition of *map*: it's in *IterableOnce*...
- How does it work? It's a little bit of magic and some ordinary code. You can look at the source code yourself. Rather advanced for now, though.

4.2

## Functions, Methods and Operators

Updated: 2021-02-25

© 2018 Robin Hillyard

The logo of Northeastern University, featuring a large white stylized letter 'N' on a black background. The 'N' is composed of two main vertical bars with a diagonal bar connecting them. Below the 'N', the words "Northeastern University" are written in a white serif font.

Northeastern  
University

# Functions, methods and operators

methods are more parametric, methods are a behavior of that class or object

- These are, *more or less*, the same thing:
  - You can think of a “method” is an aspect of object-oriented programming: essentially, it’s a “field” of the class (or object) whose expression (the right-hand-side) happens to be lazily evaluated and, usually, parameterized in either/both types and parameters;
  - In other words, methods have a convenient syntax which is very human-friendly and which allows you to name the parameters (in addition to specifying their types) and refer to those names in the expression;

# Functions, methods and operators (2)

- Classes (and objects) have methods...
  - of which their bodies define functions so...

```
scala> def increment(n: Int) = n+1
increment: (n: Int)Int
scala> increment(2)
res13: Int = 3
```

- which is equivalent to...

```
scala> val f1: Int=>Int = x=>x+1
f1: Int => Int = <function1>
scala> def increment(n: Int) = f1(n)
increment: (n: Int)Int
scala> increment(2)
res12: Int = 3
```

- but you can't do it this way...

```
scala> def increment(n: Int) = f1
increment: (n: Int)Int => Int
scala> increment(2)
res11: Int => Int = <function1>
```

Equivalent to: `f1.apply(n)`

`f1` is a partially applied function

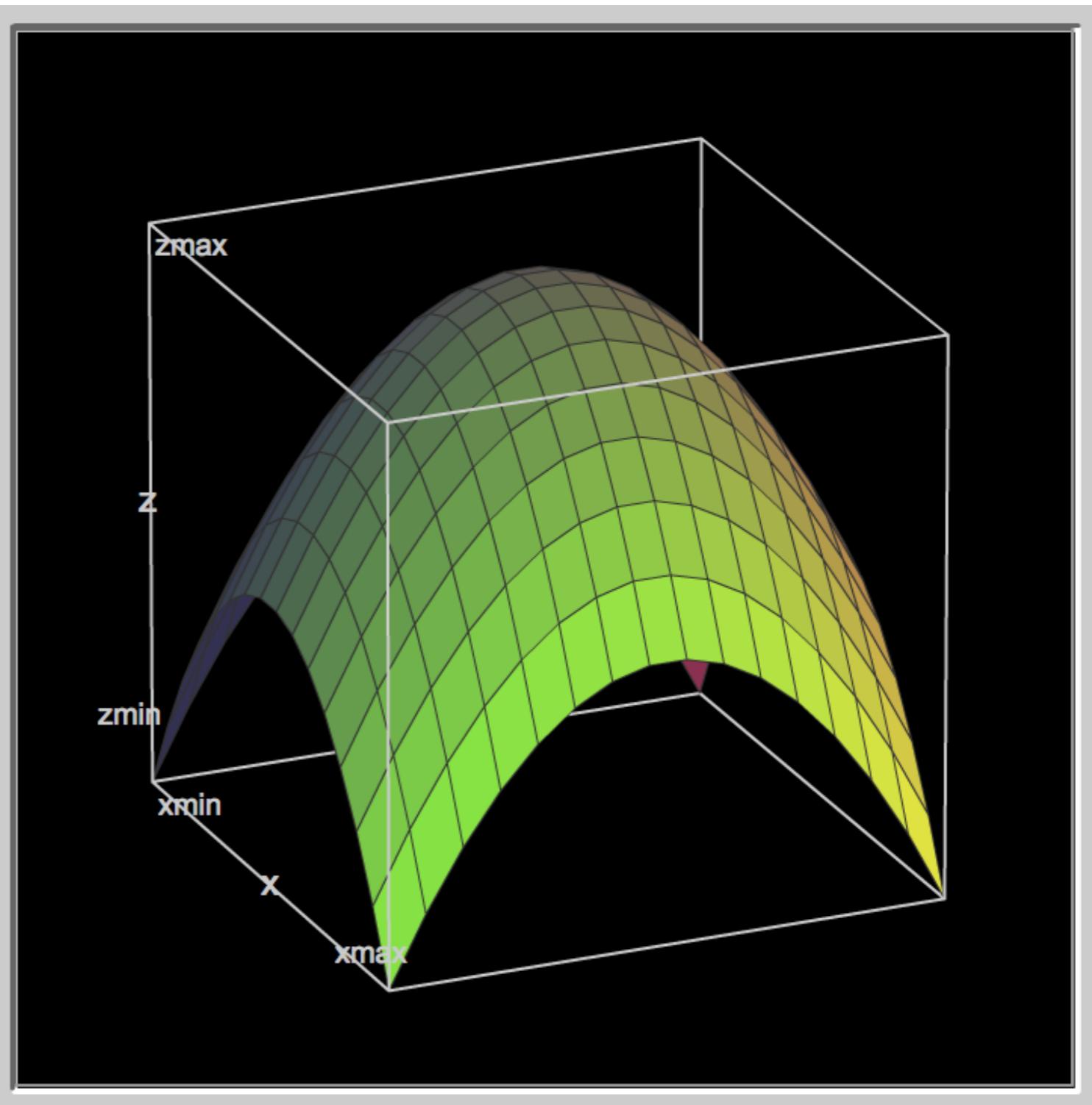
What's going on here?

# A little mathematics

- Suppose:  $z = f(x, y)$
- and also that:  $z' = f'(y)$ .
- If  $z = z'$ , for all  $x, y$ , then what does that tell us about the relationship of  $f(x, y)$  and  $f'(y)$ ?
- Obviously,  $f'$  must itself be a function of  $x$
- Let's call  $f'$  by  $g(x)$
- In that case,  $z' = g(x)(y) = f(x, y)$
- And so  $f(x, y) = g(x)(y)$
- $g(x)(y)$  is called the “curried” version of  $f(x, y)$

# Let's take a look...

- $z = f(x, y)$
  - $f(x, y) = 2-x^2-y^2$
  - $g(x)(y) = 2-x^2-y^2$
  - $g(x) = (2-y^2)-x^2$
- here, x is a bound variable for g(x),  
g(x) is  
dependent  
on x
- where y is a constant  
as far as the function g  
is concerned (also  
referred to,  
confusingly, as a “free”  
variable).
- Of course:
    - $h(y) = (2-x^2)-y^2$



# Partially-applied functions

- What if you leave off the parameter(s) of a Scala method?

```
scala> def sqr(x: Int): Int = x*x
sqr: (x: Int)Int
scala> sqr
<console>:12: error: missing arguments for method sqr;
follow this method with `_` if you want to treat it as a partially applied function
 sqr
 ^
scala> sqr _
res0: Int => Int = <function1>
scala> res0(2)
res1: Int = 4
scala> def add(x: Int, y: Int) = x+y
add: (x: Int, y: Int)Int
scala> add _
 tupled form of the function
res2: (Int, Int) => Int = <function2>
scala> res2(1,3)
res3: Int = 4
scala> val xs = Seq(1,2,3)
xs: Seq[Int] = List(1, 2, 3)
scala> xs map res0
res4: Seq[Int] = List(1, 4, 9)
scala> xs map sqr
res5: Seq[Int] = List(1, 4, 9)
```

This is a partially-applied function (indicated by “\_”). Technically, this is the  $\eta$  expansion.

Huh? surely we can't pass a *method* as a parameter? Well, yes we can, where the context is clear. The compiler invokes the  $\eta$ -expansion for us.

# Partially-applied functions (2)

```
scala> val y = 1
y: Int = 1
scala> xs map (add(_,y))
res6: Seq[Int] = List(2, 3, 4)
scala> val f: Int=>Int = y.+
f: Int => Int = <function1>
scala> f(3)
res7: Int = 4
scala> xs map { f }
res8: Seq[Int] = List(2, 3, 4)
scala> xs map (f)
res9: Seq[Int] = List(2, 3, 4)
scala> xs map f
res10: Seq[Int] = List(2, 3, 4)
```

- ← This is called a “closure” because it “closes” on y
- ← OK provided that we explicitly annotate type of f.  
This is also a closure.
- ← apply 3 to f.
- ← map xs on f.
- ← notice we can substitute parentheses for the  
braces
- ← indeed we can dispense with the parentheses too

- What we’ve done here is to separate the two parameter “sets” of the + operator and close on one and bind the other to the values of xs. We haven’t been able to split parameter sets up (the compiler won’t allow it). In a moment, we’ll see how.

# Partially-applied functions (3)

- Let's try it now for a method where we explicitly use two parameter sets:

```
scala> def gMethod(x: Int)(y: Int) = x+y
gMethod: (x: Int)(y: Int)Int
scala> val g = gMethod _
g: Int => (Int => Int) = <function1>
scala> g(1)
res11: Int => Int = <function1>
scala> g(1)(2)
res12: Int = 3
scala> val h = g _
h: () => Int => (Int => Int) = <function0>
scala> h()(1)(2)
res13: Int = 3
```

This time when we partially apply the function we again get a function (*g*) but one which has an (unbound) parameter.

But now we can partially apply *that* function (*g*) and this time we get a function (*h*) which has no parameters but which yields a function.

# Partially-applied functions (4)

- But you can always simply substitute an underscore for a parameter and make it partially-applied:

```
case class Complex(r: Double, i: Double)
val g: (Double, Double) => Complex = Complex(_, _)
```

# Partially defined functions

- In mathematics a function which is valid for all possible inputs is called a “total” function.
  - Contrarily, if a function only works for certain values, it is called a “partial” function (or partially-defined function). [subclass of partial defined Function1](#)
  - For example,  $\cos^{-1}(x)$  is only defined for  $-1 \leq x \leq 1$
  - Scala also has partially-defined functions. You've met some of them already (from week 1 - ticket agency):

```
while (state.availability) {
 receive {
 case sale: Sale => state = BoxOffice.makeTransaction(state, sale)
 case Status => sender ! state
 case _ => throw new Exception("unknown message")
 }
}
```

if this case was not present, match exception will be thrown

- The function which is provided as the body of receive is a partially-defined function: it only yields a valid result when the message received is of type *Sale* or *Status*.
- And of course our old friend *match*.

# Currying (1)

- This same notion of substituting a partially-applied function such that we end up with multiple parentheses is called Currying.
- No, it's not a culinary reference to delicious Indian food. Nor is it a reference to the Student Center at Northeastern. Or a certain talented basketball player. It is named after [Haskell Curry](#) (1900-1982). Wouldn't you feel cheated and disappointed if Scala **didn't** work this way?
- Actually, we can take an *uncurried* function definition and make it curried simply by calling the curried method:

```
scala> def f(x: Double, y: Double) = math.sqrt(x*x + y*y)
f: (x: Double, y: Double)Double
scala> val fDash = (f _).curried
fDash: Double => (Double => Double) = <function1>
scala> def g(x: Double)(y: Double) = math.sqrt(x*x + y*y)
g: (x: Double)(y: Double)Double
scala> val gDash = g _
gDash: Double => (Double => Double) = <function1>
```

function2 onwards supports  
curried functions

# Currying (2)

- Let's say we have the following matrix:

```
scala> val matrix = List(List(1,2,3),List(2,3,1),List(3,1,2))
matrix: List[List[Int]] = List(List(1, 2, 3), List(2, 3, 1), List(3, 1, 2))
```

- We can access an element of the matrix using this method:

```
scala> def element(r: Int)(c: Int) = matrix(r)(c)
element: (r: Int)(c: Int)Int
```

- In general, if we have a function with multiple parameter sets:

- def f(args1)...(argsN-1)(argsN) = E

E is some expression  
g is an arbitrary identifier

- then we can also write:

- def f(args1)...(argsN-1) = { def g(argsN) = E; g }

- def f(args1)...(argsN-1) = ( argsN => E )

which simplifies to

```
scala> def row(r: Int) = element(r) -
row: (r: Int)Int => Int
```

```
scala> val row1 = row(1)
row1: Int => Int = <function1>
```

```
scala> row1(2)
res17: Int = 1
```

# Currying (3)

- You may already have noticed that Scala methods allow for several parameter lists, as in the example from the RNG assignment (method meanU).
  - Or, take a look at this from the [Scala Documentation](#):

```
object Currying extends App {

 def filter(xs: List[Int], p: Int => Boolean): List[Int] =
 if (xs.isEmpty) xs
 else if (p(xs.head)) xs.head :: filter(xs.tail, p) parametric sets
 else filter(xs.tail, p)

 def modN(n: Int)(x: Int) = ((x % n) == 0)

 val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
 println(filter(nums, modN(2)))
 println(filter(nums, modN(3)))
}
```

modN has two parameter lists, in this case each of them has just one parameter.

Here modN is given only one parameter list!  
What's a poor method to do?

# Currying (4)

- Well, we recognize this as a “partially-applied-function”, except that here, context is unambiguous and we don’t need to provide “\_” to confirm that we want a PAF.
- Look at *modN* (which you may describe as a “curried” function) in the REPL (you will want to use “paste mode” to enter the whole program):  
`modN: (n: Int)(x: Int)Boolean`
- But *modN(2)* is a *Int=>Boolean* so in order for *modN(2)* to yield a result of *Boolean*, it must be given another *Int*. That’s exactly what the context of *filter* does: it successively applies the *Int* elements of *nums* to *modN(2)* yielding *true* if the element is even.

# Lambdas, also known as anonymous functions; function literals

- Like in Java, we can use anonymous functions

```
scala> val list = List(1,2,3)
list: List[Int] = List(1, 2, 3)
scala> list map (_.toString)
res0: List[String] = List(1, 2, 3)
scala> list map {_.toString}
res1: List[String] = List(1, 2, 3)
scala> list map {x => x.toString}
res2: List[String] = List(1, 2, 3)
scala> list map ((x: Int) => x.toString)
res3: List[String] = List(1, 2, 3)
```

This is an anonymous function. It is also a closure, although in this case, it doesn't capture any values.

Alternative syntax: with anonymous variable  
(can use `_` once only for each parameter)

Here we name our identifier as `x`

We can even specify the type of `x` if we like—but careful—you need parentheses.

- We could also do this of course (use a named function):

```
scala> def stringify(x: Int) = x.toString
stringify: (x: Int)String
scala> list map stringify
res4: List[String] = List(1, 2, 3)
scala> list.map(stringify)
res5: List[String] = List(1, 2, 3)
```

- And we could also do this (defines a “partial function”):

```
scala> list map {case x: Int => x.toString}
res0: List[String] = List(1, 2, 3)
```

Updated: 2023-11-09

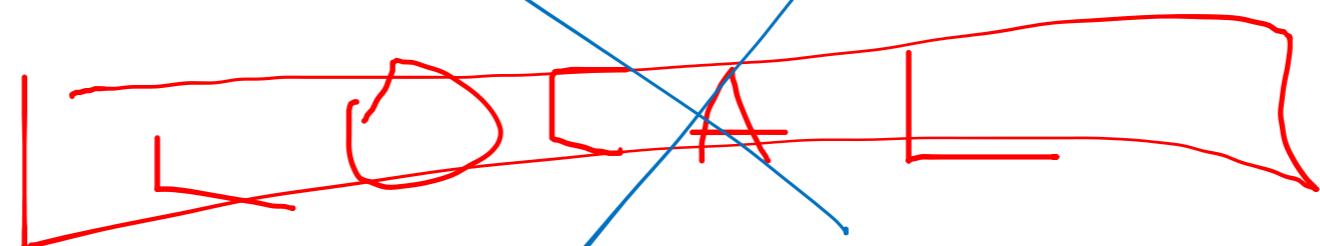
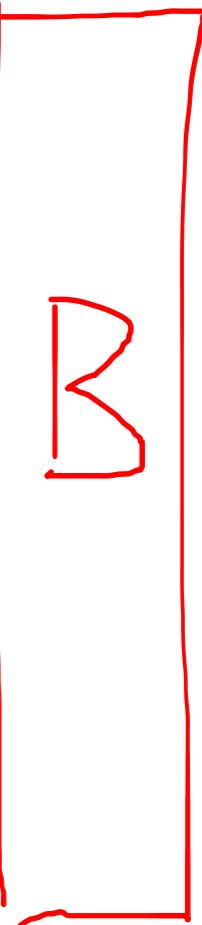
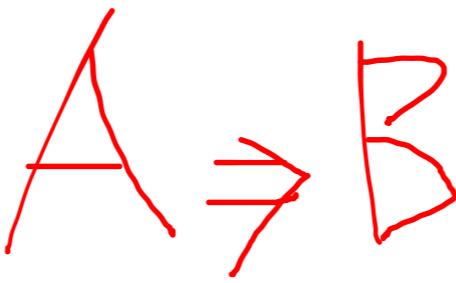
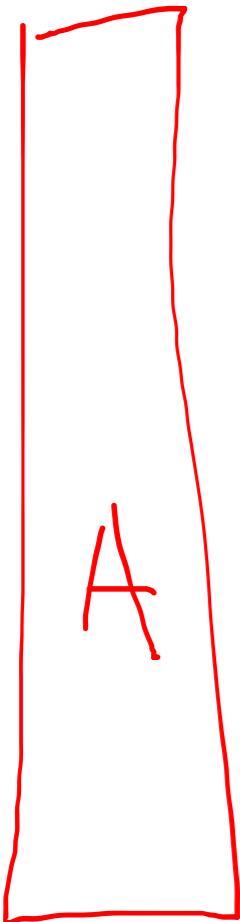
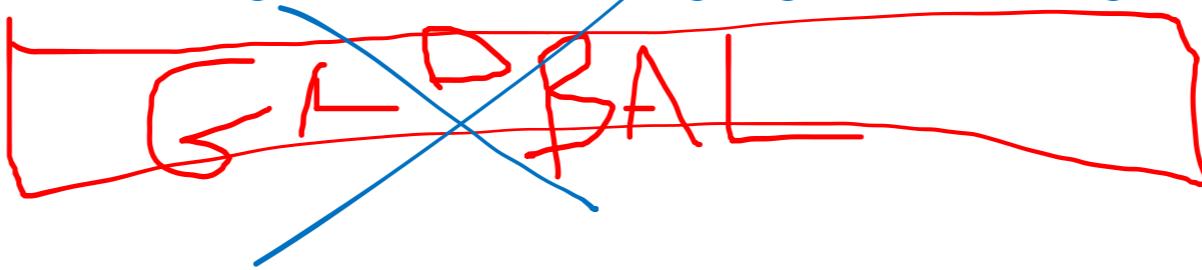
## 4.3 Implicits

© 2018-23 Robin Hillyard



expression refer to identifiers in "scope"

how should we reference identifiers in global? - some languages allow things to be declared in global, but Scala doesn't  
explicit reference? -  
Ex: math.pi



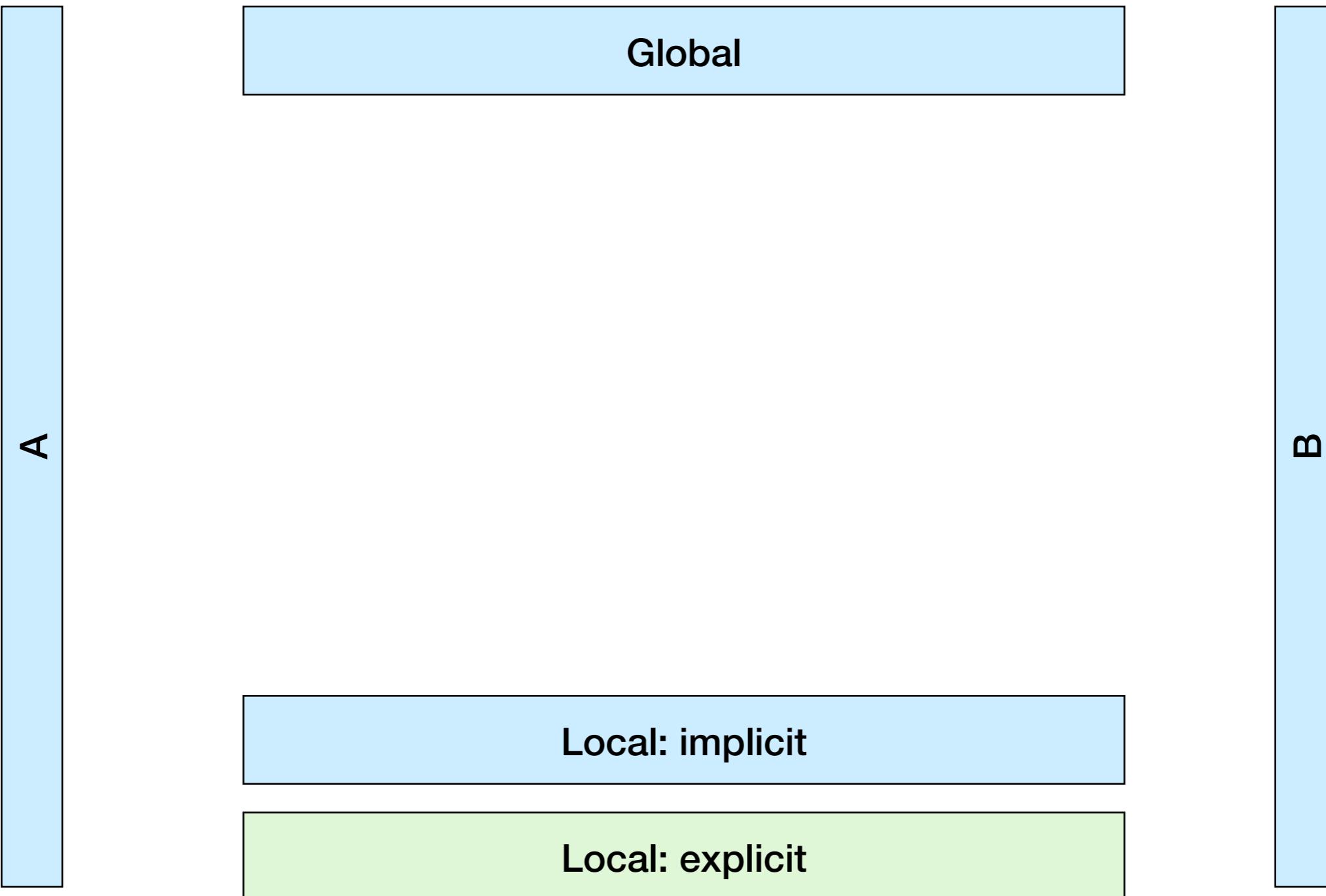
local values - parameters of the method, vals & defs inside the method

# Scope

- Where do we get values from?
  - The obvious place is that we get them from our local scope (context) or
  - ... from a parent/grandparent scope.
- But functional programs don't have deep scopes:
  - In effect, functional programs have two scopes:
    - Global scope
    - Local scope

# Scope (2)

- Is there any way we could use class/object structure to add other scopes?
  - Suppose we need a converter from type A to type B and it's not in local scope?
  - We could look in the global scope perhaps? But that would tend to clutter the global scope with a lot of junk.
  - What if we looked scopes unique to A and/or B?
  - We still need a way to mark these values as being eligible for use in general code (i.e. *implicit*).



# Implicits

- Remember from the first lecture: Odersky says that “implicits” are one of the major pillars of Scala
- See my Quora answer to [Why should I learn Scala in 2018?](#)

# Implicits (1)

- What happens when you pass an *Int* to a method that expects a *Double*?

```
scala> def cToFConverter(c: Double) = 9*c/5+32
cToFConverter: (c: Double)Double
scala> cToFConverter(10)
res1: Double = 50.0
```

- It just works! If you are coming from a Java background, this will be no big surprise (and no big deal). There's a set of language rules including that *int* will be “widened” to *double* if appropriate. But these rules are arbitrarily defined by the language designers.
- In Scala, the designers wanted programmers to have more control over this type of thing: Scala has a much more general mechanism called “implicits.”
- Think about why you want to convert the type: because you need to invoke some method that is only available in the converted type.

```
def ok(b: Boolean = True) : Boolean = b
ok() // returns True
ok(True) // returns True
ok(False) // returns False
```

```
def ok (tz: TimeZone = NY): Boolean
def ok(implicit tz: timezone): Boolean
```

# Implicits (2)

- What about using someone else's date-time library that is written for a world-wide audience but in your application of it, you never have to worry about timezones. It's tedious having to pass in a *tz* parameter to all of the methods. And what if the library is all sealed traits and classes? You can't even add your own non-tz-dependent methods.
  - Scala allows you, as a library designer, to specify certain parameters like this as "implicit".
- **Implicits can be tricky!**

# Implicits (3)

- Defining a method that adds two numbers:

```
def add(x: Int, y: Int): Int = x+y
val r = add("1","2")
```



Does not compile

- Defining an implicit converter:

```
scala> implicit def stringToInt(x: String) = x.toInt
stringToInt: (x: String)Int
scala> add("1","2")
res0: Int = 3
```

- Definition must be:

- a *val*, *def*, *class*, or a (final) parameter set of a method;
- marked *implicit*;
- in scope—scope rules for implicits are different: see Implicits (5);
- a single identifier (not something like *x.y*);
- non-ambiguous (exactly one implicit definition in scope);
- non-pipelined, i.e. *x+y* can't be replaced by *conv1(conv2(x))+y*.

# Implicits (4)

- Where can implicit conversions occur?
  - implicit conversion to an expected type: when compiler sees an  $X$  but needs a  $Y$ , it will look for an implicit  $X \Rightarrow Y$ .
  - implicit conversion of a receiver: e.g  $Y$  has a method *value* but  $X$  does not. So,  $X.value$  will not compile. Unless you provide an implicit  $X \Rightarrow Y$ .
  - implicit parameter sets: a method call *value(x,y)* can be converted to *value(x,y)(z)* if the method is defined thus:

```
def add(x: Int, y: Int)(implicit z: Int): Int = x+y+z
 //> add: (x: Int, y: Int)(implicit z: Int)Int
implicit def stringToInt(x: String) = x.toInt; //> stringToInt: (x: String)Int
implicit val z: Int = 4 //> z : Int = 4
val r = add("1","2") //> r : Int = 7
```

- implicit parameter sets are always:
  - an entire parameter set
  - the last parameter set
  - marked “implicit”

# Implicits (5)

- Here's an example where we define the *locale* implicitly:

```
package edu.neu.coe.scala.scaladate
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

trait LocaleDependent {
 def toStringForLocale(implicit locale: Locale): String
}

case class ScalaDate(date: Date) extends LocaleDependent {
 import ScalaDate.locale
 def toStringForLocale(implicit locale: Locale): String =
 getDateInstance(LONG, locale).format(date)
 override def toString: String = toStringForLocale(locale)
}

object ScalaDate {
 def apply(): ScalaDate = ScalaDate(new Date)
 implicit def locale = Locale.FRANCE
}
```

- In the REPL:

```
scala> ScalaDate()
res1: ScalaDate = 1 octobre 2015
```

```
implicitly[Numeric[Int]]
```

```
implicit object extends Numeric[Int]] {...}
```

# Implicits (6)

```
Numeric[T] {
```

```
 def zero: T
```

```
 def plus(t1:) : t1: T, t2 : T {...}
```

- Scope rules for implicits:
  - In the *current* scope, an implicit must be declared above the place it is to be used. Important!
    - Basically, it's the same as the rule for *vals*, but it applies to every implicit object, including those defined by *def* or *class*.
    - If the implicit is actually defined somewhere else that is not in scope, then you can get it into scope using *import*.
  - An implicit involving a class *C* may be found in the companion object of *C*.

# Implicits (7)

- You can even have implicit classes!
  - Constructor must have exactly one parameter: this is the value that will be “converted” implicitly into an instance of the class.
  - Example: Benchmark class:

```
object Benchmark extends App {
 implicit class Rep(n: Int) {
 /**
 * Method which can be invoked, provided that Benchmark._ has been imported.
 * See for example BenchmarkSpec
 * @param f the function to be invoked
 * @tparam A the result type of f
 * @return the average number of nano-seconds per run
 */
 def times[A](f: => A): Double = {
 // Warmup phase: do at least 20% of repetitions before starting the clock
 1 to (1+n/5) foreach (_ => f)
 val start = System.nanoTime()
 1 to n foreach (_ => f)
 (System.nanoTime() - start) / n.toDouble
 }
 }
 println(s"ave time for 40! is ${10000.times(Factorial.factorial(40))} nanosecs")
}
```

# Implicits (8)

- A more common example of this is *StringOps*.
- There are many methods that you would like to have in a *String* but, because it is *final* (as in Java), you cannot add any of your own behavior to the *String* class.
  - For example, suppose you want to create a padding string of exactly n spaces?
  - There's no good way to do this with *String*.
  - But, in Scala, you can just write::

```
val n: Int = ???
val padding = " " * n
implicit class
```
- A new *StringOps*(" ") is constructed and its \* method is invoked with parameter n.

# Implicits (9)

- Another example comes from my crypto project (for [CSYE7374](#)).

- Because modular arithmetic is so common in cryptography, I defined an implicit class thus:

```
implicit class Divides(x: Int) {
 def |>(y: Int): Boolean = |>(y.toLong)
 def |>(y: Long): Boolean = y % x == 0
 def |>(y: BigInt): Boolean = y % x == 0
}
```

- Thus, we can write, in a unit test of “7 | 49”:

```
7 |> 49L shouldBe true
```

# Polymorphism

- Polymorphism is perhaps the most important aspect of object-oriented programming:
  - It allows us to refer to an individual instance of something using a label (interface, super-type, whatever) that is less specific, i.e. more generic, than the actual instance.
  - This allows for things like dependency injection and factories.
  - It's also the basis for encapsulation (although that's possibly an orthogonal concept in O-O).

# Polymorphism (2)

- In a purely object-oriented language, polymorphism is implemented via inheritance, i.e. by sub-typing:
  - A class extends another class;
  - A class implements an interface;
  - An interface extends another interface.
- There are times when this won't work very well:
  - The superclass is final;
  - You can extend a class from a third-party library (e.g. open source) but unfortunately, a new version of that library makes nonsense of your sub-class;
  - When it just doesn't make sense: i.e. it's better to use composition rather than inheritance.

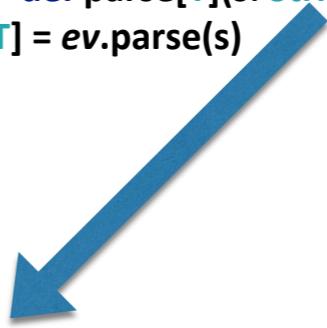
# Polymorphism (3)

- Typeclasses are the functional-programming way of accomplishing this notion of polymorphism:
  - Let's say you have in mind a trait but there's nothing appropriate for it to extend:

```
trait Parseable[T] {
 def parse(s: String): Try[T]
}
object Parseable {
 trait ParseableInt extends Parseable[Int] {
 def parse(s: String): Try[Int] = Try(s.toInt)
 }
 implicit object ParseableInt extends ParseableInt
}
object TestParseable {
 def parse[T : Parseable](s: String): Try[T] = implicitly[Parseable[T]].parse(s)
}
```

This form is called a “context bound”.  
But we can also write it as follows:

```
def parse[T](s: String)(implicit ev: Parseable[T]):
 Try[T] = ev.parse(s)
```


  - What we are doing here is adding the behavior of *Parseable* to type *T* without requiring *T* to extend anything (without using inheritance).
    - Note that you cannot add a context bound to a trait. Why not?

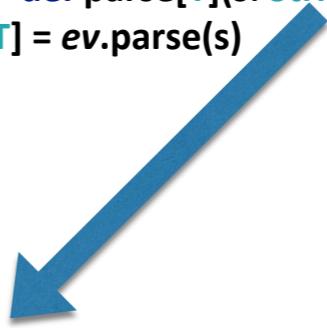
# Polymorphism (3)

- Typeclasses are the functional-programming way of accomplishing this notion of polymorphism:
  - Let's say you have in mind a trait but there's nothing appropriate for it to extend:

```
trait Parseable[T] {
 def parse(s: String): Try[T]
}
object Parseable {
 trait ParseableInt extends Parseable[Int] {
 def parse(s: String): Try[Int] = Try(s.toInt)
 }
 implicit object ParseableInt extends ParseableInt
}
object TestParseable {
 def parse[T : Parseable](s: String): Try[T] = implicitly[Parseable[T]].parse(s)
}
```

This form is called a “context bound”.  
But we can also write it as follows:

```
def parse[T](s: String)(implicit ev: Parseable[T]):
 Try[T] = ev.parse(s)
```


  - What we are doing here is adding the behavior of *Parseable* to type *T* without requiring *T* to extend anything (without using inheritance).
    - Note that you cannot add a context bound to a trait. Why not?

```
Iterable[T] {
```

```
 ...
 def sorted(implicit to : Ordering[T]) = to.compare(t1, t2)
```

```
}
```

# Sorting

- Unlike in Java where we need an explicit *Comparable* (or *Comparator*), ordering in Scala is done implicitly.

```
scala> List(1,3,2).sorted
res2: List[Int] = List(1, 2, 3)
```

Use *Ordering* since 2.8

- but you can provide an explicit ordering method → this works because operator “<“ is implemented by the *Ordered* trait:

```
scala> List(1,3,2).sortWith(_ < _)
res3: List[Int] = List(1, 2, 3)
```

- you can mix in *Ordered[A]*\* with your own trait or class based on *A*, which defines the abstract method `def compare(that: A): Int`

```
case class UniformDouble(x: Double) extends AnyVal with Ordered[UniformDouble] {
 def + (y: Double) = x + y
 def compare(that: UniformDouble): Int = x.compare(that.x)
}
```

```
(scalaTest...)
val y = RNG.randoms(new UniformDoubleRNG(0L)) take 10 toList;
y.sorted.head should equal (UniformDouble(0.052988271629967366))
```

\* see:

<https://github.com/scala/scala/blob/v2.13.5/src/library/scala/math/Ordered.scala>

# Ordering

- Since 2.8, Scala has used *Ordering* as the primary mechanism for sorting.
- *Ordering* is a typeclass whereas *Ordered* is a supertype.
  - There are implicit conversions between *Ordered* and *Ordering*, however.
  - For example:
    - `trait Numeric[T] extends Ordering[T]`

# Use of Ordering

```
sealed trait TraitExample[T] extends Comparable[TraitExample[T]] {
 def name: String
 def property: T
 def >(o: TraitExample[T]): Boolean = compareTo(o)>0
 def <(o: TraitExample[T]): Boolean = compareTo(o)<0
 def >=(o: TraitExample[T]): Boolean = compareTo(o)>=0
 def <=(o: TraitExample[T]): Boolean = compareTo(o)<=0
 def ==(o: TraitExample[T]): Boolean = compareTo(o)==0
}
abstract class Base[T: Ordering] extends TraitExample[T] {
 def compareTo(o: TraitExample[T]): Int = implicitly[Ordering[T]].compare(property, o.property)
}
case class Telephone(name: String, number: String) extends Base[String] {
 def property: String = number
}
case class Age(name: String, age: Int) extends Base[Int] {
 def property: Int = age
}
object TraitExample extends App {
 val telephone1 = Telephone("Robin", "123456789")
 val telephone2 = Telephone("Hillyard", "0123456789")
 if (telephone1 > telephone2) println("Robin's phone comes after Hillyard's phone")
}
```

# Getting IDE help with implicits

- [https://confluence.jetbrains.com/display/IntelliJIDE  
A/Working+with+Scala+Implicit+Conversions](https://confluence.jetbrains.com/display/IntelliJIDE/A/Working+with+Scala+Implicit+Conversions)
- <https://blog.jetbrains.com/scala/2018/07/25/intellij-scala-plugin-2018-2-advanced-implicit-support-improved-patterns-autocompletion-semantic-highlighting-scalafmt-and-more/>

Updated: 2022-03-21

## 4.4 Serialization

© 2015-22 Robin Hillyard



# Why is this important?

- Serialization/deserialization in parallel processing:
  - Marshalling/unmarshalling—required for crossing processor boundaries (thread boundaries too where there is little shared data between threads, as is typical with FP).
  - Examples:
    - server/client (web using REST, SOAP, whatever), messaging subscriptions, etc.
    - inter-processor for map/reduce, actor replication, segmentation of Spark datasets, etc.
  - Technologies:
    - typical server/client solution uses JSON (Javascript Object Notation) but HTML and XML also used (esp. for SOAP/WSDL).
    - JVM solution for inter-processor is to use binary (*Serializable* interface).
- Serialization/deserialization for data acquisition/persistence:
  - Streaming data, and regular data ingest (ETL<sup>\*</sup>):
    - use XML (e.g. RSS), JSON, CSV or plain text.
  - Persistence is oriented towards easy analysis:
    - typically use JSON (NoSQL database) but can also be in (append-only) log files, as well as XML.

<sup>\*</sup> Extract, transform, load

# Different serialized formats (1)

- Binary
  - The original (but always with problems). Still used for JVM serialization.  
Advantage: opaque; Disadvantages: opaque; versioning problems.
- XML
  - Successor to “SML”, mainly used for legacy situations. Adv: legible, model-driven, flexible; Disadv: too flexible in some respects, verbose, model-driven.
- HTML
  - Originally extension to “XML” but long-since diverged; Adv: legible, standardized, ubiquitous; Disadv: not standardized.
- JSON
  - Originally developed for Javascript, but now almost *de facto* standard. Adv: legible, very flexible (simple model), objects, ubiquitous; Disadv: slightly verbose.

# Different serialized formats (2)

- Avro, Parquet
  - Serialization file format for Hadoop
- CSV
  - Originally designed for Excel, but now almost *de facto* standard for tabular data. Adv: legible, ubiquitous; Disadv: no formal model (e.g. separators, headers, quotes).
- Text, log files
  - Anything goes. Requires NLP/regex to interpret (but Scala very good at that)
- Database
  - Relational, NoSQL or Search Engine: indexed for fast query response but frequently that's not required.
- Other cross-platform capabilities
  - RPC (remote procedure call), Thrift, etc.

# **Serialization/deserialization:**

# **XML**

# XML: built-in support

```
scala> val x1 = <xml><title>Hello, World!</title></xml>
x1: scala.xml.Elem = <xml><title>Hello, World!</title></xml>
```

Notice that you don't have to put quotes around the XML string.

```
scala> x1 \ "xml"
res0: scala.xml.NodeSeq = NodeSeq()
```

The "\ operator gives you a sequence (NodeSeq) of matching nodes.

```
scala> x1 \ "title"
res1: scala.xml.NodeSeq = NodeSeq(<title>Hello, World!</title>)
```

The "\\ operator gives you a sequence (NodeSeq) of matching nodes including their children.

```
scala> x1 \\ "xml"
res3: scala.xml.NodeSeq = NodeSeq(<xml><title>Hello, World!</title></xml>)
```

```
scala> val x2 = <xml><title language="English">Hello, World!</title></xml>
x2: scala.xml.Elem = <xml><title language="English">Hello, World!</title></xml>
```

```
scala> x2 \ "title" map {_ \@ "language"}
res7: scala.collection.immutable.Seq[String] = List(English)
```

The "\\@ operator gives you the value of the specified attribute.

- This literal XML feature is great, but not especially useful for reading xml (which we normally do from a file)

```
scala> val xml = XML.loadFile("poets.xml")
xml: scala.xml.Elem = etc...
```

# More XML processing

- Of course, we can still use our for-comprehensions:

```
scala> val poets = for (poets <- XML.loadFile("poets.xml")\\\"poets"; poet <- poets\\\"poet") yield poet
poets: scala.xml.NodeSeq = NodeSeq(<poet>
 <name language="en">Wang Wei</name>
 <name language="zh">王維</name>
</poet>, <poet>
 <name language="en">Li Bai</name>
 <name language="zh">李白</name>
</poet>)
```

Notice that we (finally) put in an “if” condition into our for-comprehension

- To get the text of a node(s), we just use *node.text*:

```
scala> val names = for (poets <- XML.loadFile("poets.xml")\\\"poets"; poet <- poets\\\"poet"; name <- poet\\\"name"; lang = name\\\"language"; if lang=="en") yield name.text
names: scala.collection.immutable.Seq[String] = List(Wang Wei, Li Bai)
```

- But suppose we want to write out xml for an object?

```
scala> case class Name(name: String, language: String) {
 | def toXML = <name language={language}>{name}</name>
 | }
```

defined class Name

```
scala> val weiWang = Name("Wei Wang", "en")
weiWang: Name = Name(Wei Wang, en)
```

```
scala> weiWang.toXML
```

```
res3: scala.xml.Elem = <name language="en">Wei Wang</name>
```

We use curly brackets {} to insert expressions.

# XML in both directions

- Here's the full poets class:

```
package edu.neu.coe.scala.poets
import scala.xml.{XML, Node, NodeSeq}
case class Name(name: String, language: String) {
 def toXML = <name language={language}>{name}</name>
}
case class Poet(names: Seq[Name]) {
 def toXML = <poet>{names map (_.toXML)}</poet>
}
object Poet {
 def fromXML(node: Node) = Poet(Name.fromXML(node \
"name"))
}
object Name {
 def getLanguage(x: Option[Seq[Node]]) = x match {case
Some(Seq(y)) => y.text; case _ => ""}
 def fromXML(nodes: NodeSeq): Seq[Name] = for {
 node <- nodes
 } yield
Name(node.text, getLanguage(node.attribute("language")))
}
object Poets extends App {
 def toXML(poets: Seq[Poet]) = poets map {_ toXML}
 val xml = XML.loadFile("poets.xml")
 val poets = for (poet <- xml \\ "poet") yield
Poet.fromXML(poet)
 println(poets)
 println(toXML(poets))
}
```

- Here is the poets file:

```
<?xml version="1.0" encoding="UTF-8"?>
<class>
 <name>Tang Dynasty Poetry</name>
 <description language="en">This is just an
example XML file</description>
 <instructor>Robin Hillyard</instructor>
 <poets>
 <poet>
 <name language="en">Wang
Wei</name>
 <name language="zh">王維</name>
 </poet>
 <poet>
 <name language="en">Li Bai</name>
 <name language="zh">李白</name>
 </poet>
 </poets>
</class>
```

# **Serialization/deserialization: JSON**

# JSON libraries

- There is no built-in support for JSON in Scala (never was)— that's appropriate. There are many libraries but we will concentrate on *spray-json* (<https://github.com/spray/spray-json>).
  - *build.sbt* file:

```
val sprayGroup = "io.spray"
val sprayJsonVersion = "1.3.6"
libraryDependencies ++= List("spray-json") map {c => sprayGroup %% c % sprayJsonVersion}
```

Other JSON libraries:  
uJson  
circe

# JSON processing

- Here is our *Poets* object from last time, but with JSON output enabled

```
object Poets extends App {
 import spray.json._
 import scala.xml.XML
 type PoetSeq = Seq[Poet]
 def toXML(poets: PoetSeq) = poets map {_ toXML}
 val xml = XML.loadFile("poets.xml")
 val poets: PoetSeq = for (poet <- xml \\ "poet") yield Poet.fromXML(poet)
 println(poets)
 println(toXML(poets))

 case class Poets(poets: PoetSeq)

 object MyJsonProtocol extends DefaultJsonProtocol {
 implicit val nameFormat = jsonFormat2(Name.apply)
 implicit val poetFormat = jsonFormat1(Poet.apply)
 implicit val poetsFormat = jsonFormat1(Poets)
 }

 import MyJsonProtocol._

 println(poets.toJson)
}
```

We import the *spray.json* classes and, just for convenience, define this *PoetSeq* type.

New *Poets.Poets* class

Here we must create appropriate implicit values. You need them all, else compile errors. Note that for *Name* and *Poet*, which have explicit companion objects, we must reference the apply method explicitly. Note also that *Name* has two fields so requires *jsonFormat2*.

Very important! These *must* be listed in the correct order least-dependent first, most-dependent last. Otherwise it will not compile. That is one aspect of *implicits* that we have to watch out for!

# JSON in REPL

- Run the *main* program:

```
scala> import edu.neu.coe.scala.poets._
import edu.neu.coe.scala.poets._

scala> Poets.main(Array[String]())
List(Poet(List(Name(Wang Wei,en), Name(王維,zh))), Poet(List(Name(Li Bai,en), Name(李白,zh))))
List(<poet><name language="en">Wang Wei</name><name language="zh">王維</name></poet>, <poet><name language="en">Li
Bai</name><name language="zh">李白</name></poet>)
JSON: [{"names": [{"name": "Wang Wei", "language": "en"}, {"name": "王維", "language": "zh"}]}, {"names": [{"name": "Li
Bai", "language": "en"}, {"name": "李白", "language": "zh"}]}]
```

- Add a *fromJson* method to do the opposite transformation:

```
def fromJson (string: String) = string.parseJson.convertTo[PoetSeq]
```

- Back in the REPL:

```
scala> import edu.neu.coe.scala.poets._
import edu.neu.coe.scala.poets._

scala> import spray.json._
import spray.json._

scala> val source = """[{"names": [{"name": "Wang Wei", "language": "en"}, {"name": "王維", "language": "zh"}]}, {"names": [{"name": "Li Bai", "language": "en"}, {"name": "李白", "language": "zh"}]}]"""
source: String = [{"names": [{"name": "Wang Wei", "language": "en"}, {"name": "王維", "language": "zh"}]}, {"names": [{"name": "Li
Bai", "language": "en"}, {"name": "李白", "language": "zh"}]}]

scala> Poets.fromJson(source)
res0: edu.neu.coe.scala.poets.Poets.PoetSeq = List(Poet(List(Name(Wang Wei,en), Name(王維,zh))), Poet(List(Name(Li
Bai,en), Name(李白,zh))))
```

- It's that easy, particularly if you use case classes!

# TableParser

- *TableParser* is an open-source project of mine which does for CSV files (and similar) as *spray/json* does for JSON.
  - You can see an example of its use in the *MovieRating* assignment.
  - The key feature is that it will read/write nested case class instances.
  - Take a look: <https://github.com/rchillyard/TableParser>
  - It's available via Maven Central:

```
libraryDependencies += "com.phasmidsoftware" %% "tableparser" % "1.1.1"
```

Updated: 2021-11-08

# 4.5 Recursion

© 2018 Robin Hillyard

A large, semi-transparent watermark of the Northeastern University logo is positioned in the lower right quadrant of the slide. The logo consists of a stylized 'N' and 'U' formed by white, angular shapes on a black background.

Northeastern  
University

# Iteration

- You're all familiar with iteration in Java:

```
public class Iteration {

 public static int sum(int[] array) {
 int result = 0;
 for (int i : array) result += i;
 return result;
 }

 public static void main(String[] args) {
 int[] array = new int[10];
 for (int i = 0; i < 10; i++) array[i] = i+1;
 System.out.println(sum(array));
 }
}
```

- Note that this involves the use of a mutable variable called *result*.

# Addition by recursion

- Iteration vs. Recursion
  - In functional programming we like to avoid the use of mutable variables so that we can prove programs using *referential transparency* (a.k.a. the substitution principle).
  - But what is the more natural way to do the sum? iteration or recursion?
  - A discrete series is often defined recursively (e.g. Fibonacci sequence). I suggest that iteration is a consequence of the Turing/von Neumann architecture.
  - A more mathematical way of defining the sum of a list  $xs$  of numbers would be:
    - $\text{sum}(xs) = xs.\text{head} + \text{sum}(xs.\text{tail})$

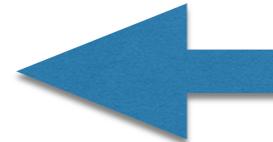
# Sum by recursion

- Here's the simplest way to sum in Scala:

```
object Sum extends App {

 def sum(xs: Seq[Int]): Int = xs match {
 case Nil => 0
 case h :: t => h + sum(t)
 }

 val xs = LazyList.from(1) take 10
 println(sum(xs.toList))
}
```



*Recursively call sum*

- But haven't we always been taught not to use recursion in our programs if we can avoid it?
  - What's wrong with recursion?
  - Think about what would happen if instead of 10 numbers, we summed 10 *billion* numbers.

# Stack Overflow

- We'd get a stack overflow. Try it for yourself.
  - That's really bad news. That's why we were taught not to use recursion!
- But in functional programming we can actually avoid using the stack provided that the recursion is *tail* recursive.
  - Suppose that the *last* thing we do in our code is the recursive call itself (that's called *tail* recursion)? We'd also say that the recursive call is in *tail position*. In that case, there'd be nothing we'd need to store on the stack, right?
  - So, we can “unroll” a tail-recursive call into a kind of iteration.

# Sum by recursion (take 2)

- Let's take another look

```
object Sum extends App {

 def sum(xs: Seq[Int]): Int = xs match {
 case Nil => 0
 case h :: t => h + sum(t)
 }

 val xs = LazyList.from(1) take 10
 println(sum(xs.toList))
}
```



*Recursively call sum but the last thing we do is to add h to the result of the recursive call—“+” is in tail position.*

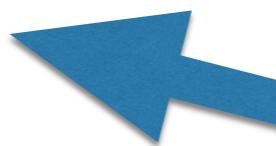
- How can we make this tail-recursive?
  - Actually, it's quite easy...

# Sum by tail-recursion\*

- We create an “inner” method which is tail-recursive
  - Its signature is based on two<sup>†</sup> things:
    - the current value of the result (and which will be yielded when the recursion terminates, in this case when `work` is `Nil`);
    - the work still to do.
  - Here’s our new `sum` (note we use `BigInt` because we no longer have a restriction on the size of `xs`):

```
object Sum extends App {
 def sum(xs: Seq[Int]): BigInt = {
 def inner(result: BigInt, work: Seq[Int]): BigInt = work match {
 case Nil => result
 case h :: t => inner(result+h, t)
 }
 inner(0, xs)
 }

 val xs = LazyList.from(1) take 10000000
 println(sum(xs.toList))
}
```



**Tail-recursively call inner which is the last thing we do.**

\* also known as tail call recursion

<sup>†</sup> three if you’re processing something 2-dimensional like a tree

# How can we be sure it's tail-recursive?

- The compiler will optimize it provided that it really is tail-recursive. But what if it's not?
  - In that case, you risk a stack overflow at run-time!
  - But here's what you can do: just add the *tailrec* annotation which asserts that the method *is* tail-recursive and, if it's not, the compiler will warn you:

```
import scala.annotation.tailrec
object Sum extends App {
 def sum(xs: Seq[Int]): BigInt = {
 @tailrec def inner(r: BigInt, work: Seq[Int]): BigInt = work match {
 case Nil => r
 case h :: t => inner(r+h, t)
 }
 inner(0, xs)
 }
 val xs = LazyList.from(1).take 10000000
 println(sum(xs.toList))
}
```

# Tail Recursion

## — factorial

- Let's take a look at perhaps the most obvious recursive function, *factorial*:

```
scala> def badFactorial(x: Int): Long = if (x<=1) 1 else x*badFactorial(x-1)
badFactorial: (x: Int)Long
```

- This isn't tail-recursive. Why not?
- But the following *is* tail-recursive (which we can assert with the annotation):

```
def factorial(n: Int) = {
 @scala.annotation.tailrec def inner(r: Long, n: Int): Long =
 if (n <= 1) r
 else inner(n * r, n - 1)
 inner(1L, n)
}
```

When we create one of these tail-recursive inner methods, we usually have two parameters: the first (*r*) is the current result; the second (*n*) represents the work still to do. Sometimes, there is a third parameter which controls some other aspect of the recursion.

# What else can we do that's tail-recursive?

- *sum* and *factorial*
  - In the *sum* case, we added *h* to *result*,
  - In the *factorial* case, we multiplied *n* by *result*.
- *FoldLeft* for a more general aggregation function
  - In general, we can provide our own function to aggregate the current result with the current head (we will also need a value to use as the starting result).

```
def foldLeft[X,Y](xs: Seq[X])(y: Y)(f: (Y,X)=>Y): Y = {
 @tailrec def inner(r: Y, work: Seq[X]): Y = work match {
 case Nil => r
 case x :: t => inner(f(r,x),t)
 }
 inner(y, xs)
}

def sum(xs: Seq[Int]): BigInt = foldLeft(xs)(0)(_+_)

val xs = LazyList.from(1) take 10000
println(sum(xs.toList))
```

# Other recursive methods

- Of course, we can also define *foldRight* but it can't be efficient and tail-recursive unless we are operating on a backwards list.
- We can also define *reduce* like *foldLeft* but in *reduce*, the initial value is inferred to be the zero value in the *Y* type. This constrains *Y* such that we can create a *Y* based on zero. More on that when we get to implicits...
- Another important recursive method which is similar but is not reducing (as in *Seq[A] => A*) but composing (*Seq[A] => Seq[A]*) is *scanLeft* (and *scanRight*). Elements of the result are compositions of adjacent elements of the input, where the composition is defined by the given function. Remember this?  
`0L #:: f.scanLeft(1L)(_ + _)`

# Remember sequence, traverse?

- 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 *xos* and, if all elements are *Some(x)*, collect them into a sequence *xs* then return *Some(xs)*. If any of the elements are *None*, return *None*.
  - Now, we're ready to implement this one:

```
def sequence[X](xos: Seq[Option[X]]): Option[Seq[X]] = xos.foldLeft(Option(Seq[X]())) {
 (xso, xo) => for (xs <- xso; x <- xo) yield xs :+ x
}
```



I don't expect you to remember this!!!

# See also...

- <http://scalaprof.blogspot.com/2016/05/transforming-iteration-into-tail.html>
- <http://scalaprof.blogspot.com/2016/11/a-generic-tail-recursive-method-for.html>
- You can also transform a recursive call in tail position into a loop yourself: you can do it in Java or Scala. But keep in mind that the Java compiler does not optimize tail calls.
- Java8 also allows you to do tail-call recursion but it uses a technique called *trampolining* which is beyond our scope.

4.6

Updated: 2021-03-08

# Review: Syntax and Collections

© 2016 Robin Hillyard

The logo of Northeastern University, featuring a large white 'N' and 'U' on a black background.

Northeastern  
University

**Part one**

*Syntax*

# Scala programs and syntax

- I've explained before about the nature of Scala programs:
  - Basically, a Scala program is an expression that yields a result\* but...
  - there are other constructs which you can insert before the expression such as:
    - type definitions (traits, classes, etc.) (including methods and initialization—expressions yielding Unit);
    - method definitions (where the RHS is an expression);
    - val/var definitions (where the RHS is an expression);
    - imports;
    - syntactic sugar such as for-comprehension, case clause.
  - Usually, if you add such things to your expression you will need to create a block with {}

\* So, you don't need to end an expression with "return" since that's what's expected.

# Lines and blocks

- Lines:
  - Scala lines don't normally need a semi-colon at the end of each line because they are not normally *statements* (but there are exceptions to this which we will cover later—and which the compiler will warn you about)
  - The compiler pseudo-inserts a semi-colon for you at the end of each line if it thinks it's a statement rather than an expression.
  - So, if you write:

```
expression1
+expression2
```

it will be interpreted as a statement followed by an expression

```
expression1; and +expression2
```
- You can fix this by putting parentheses around your expression or by moving the operator up to the first line:

```
(expression1
+expression2)
expression1+
expression2
```

# Lines and blocks (contd.)

- Blocks:
  - they allow you to precede your expression by val/var/def statements;
  - they reduce namespace conflicts;
  - they allow for encapsulation (information hiding);
  - there is no requirement for method or identifier definitions (defs, vals, etc.) to be at any particular level: private methods will normally be inside a public method definition (or a class)

- Even import statements can be inside blocks

```
val x = 3
def y = {
 val z = sqr(y)*x
 import math.round
 def sqr(p: Double) = round(p*p).toInt
 z
}
```

# Basic Scala syntax\*

- module ::= prolog type-definition\*
- prolog ::= package import\*
- type-definition ::= header definition\* “}”
- header ::= trait identifier mixins “{“ |  
  [“abstract”|“case”] “class” identifier type-declaration  
parameter-set\* mixins “{“ auxiliary-constructor\* initialization |  
  object identifier “{“
- mixins ::= “extends” type [“with” type]\*
- definition ::= method-definition | variable-definition | type-definition
- method-definition ::= “def” identifier [parameter-set]\* “:” return-type [  
  “=” expression ]
- variable-definition ::= [“lazy”] “val”|“var” identifier “:” return-type [ “=”  
expression ]
- expression ::= identifier | invocation | “{“ definition\* expression “}”
- invocation ::= [ receiver ] [“.”] identifier [ identifier | [“(“ expression\* “)”)”]\*

\* For the true syntax see <http://www.scala-lang.org/docu/files/ScalaReference.pdf> p. 159

# Parentheses

- Parentheses are generally there to override the precedence rules for expressions. But occasionally, there's a bit more to it.
  - IntelliJ/IDEA and Eclipse have analyzers which will tell you if you have superfluous parentheses.
    - Joke: *what does LISP stand for?* Lots of irritating superfluous parentheses
    - For example, you don't need parentheses around a singleton parameter type of a function type—these are the same:
  - But, it's fairly conventional to put the parentheses there, in parallel so to speak with the function invocation. And if your parameter list is a tuple (“parameter set”), you must use parentheses.
  - You will need parentheses here, though:

```
trait Cache[K, V] extends (K => Future[V])
```

# Syntax of lambdas

- Lambdas involve pattern-matching so here's a summary of their rules:

- You don't need parentheses for a lambda provided that the context clearly requires a function and the type of the “`_`” can be inferred:

```
scala> def doubleMap(f: Int=>Int, g: Int=>Int)(x: Int) = (f andThen g)(x)
doubleMap: (f: Int => Int, g: Int => Int)(x: Int)Int
scala> doubleMap(_+1, _*2)(3)
res9: Int = 8
```

- If the context isn't clear, you will need parentheses:

```
scala> val ys = xs map _*2
<console>:13: error: missing parameter type for expanded function ((x$1: <error>) =>
xs.map(x$1.$times(2)))
 val ys = xs map _*2
 ^
scala> val ys = xs map (_*2)
ys: List[Int] = List(2, 4, 6)
```

- If you need to specify the type, you will need braces and, maybe, “case”:

```
scala> val xs: List[Any] = List(1,2,3)
xs: List[Any] = List(1, 2, 3)
scala> val ys: List[Int] = xs map { case x: Int => x * 2}
ys: List[Int] = List(2, 4, 6)
```

# Patterns—Review (1)

- In Scala, pattern-matching plays a big part. Patterns are found:
  - In a variable definition:

```
val Some(x) = Option(methodCall); println(x)
```
  - in a case clause (within a match);

```
case Some(x) => println(x)
```
  - in a lambda;

```
map (x => 2*x)
map {x: Int => 2*x}
```
  - in a for-comprehension.

```
for (x <- xs) yield x*2
for (Some(x) <- xos) yield x*2
```

    - *BTW, some of these are very subtly similar (I don't even understand some of the distinctions—I use the source-code analyzer to help in some situations)*
- The important thing is that a pattern not only *matches* but also serves as a pseudo-variable within its scope.

# Patterns (2)

- Example:

```
def map[U](f: (T) => U): RandomState[U] =
JavaRandomState[U](n, n => f(g(n)))
```

- In this fragment of code, there are two “n”s. The first *n* is a variable in the scope of the *map* method and its enclosing class. The second *n* is a pattern (within a lambda). It could equally have been *x* (probably should have). The lambda could also have been written (with no explicit pattern):  
*f(g(\_))*
- Another form that is basically the same:

```
def map[U](f: (T) => U): RandomState[U] =
JavaRandomState[U](n, {m:Long => f(g(m))})
```

# Patterns (3) and “\_”

- The anonymous match-everything pattern: `_`
  - Similar to a simple identifier like `x`, which also matches everything, the “`_`” does not define a bound variable, e.g. `case _ => None`
- But the underscore `_` has several other meanings:
  - Anonymous bound variable in a lambda, e.g. `_ + _`
  - The wildcard in an import statement (like `*` in Java)
  - Higher-kinded type parameter, e.g. `def f[M[_]]`
  - $\eta$ -expansion of method into function, e.g. `apply _`
  - conversion of sequence to varargs: as in `f(xs: _*)` or as in `case Seq(xs @ _*)`

# The IDE is there to help!

- If all of this syntax stuff seems confusing, recall that the IDE and its built-in compiler is there to help you.
- If types are refusing to match, look at some of your intermediate variables and, if necessary, explicitly specify a type annotation (easy to do with an IDE “quick fix,” i.e. the light bulb in IntelliJ IDEA).
- Do you have a strange type being mentioned like *Any* or *Product*? Chances are you are missing the *else* part of an *if* clause.

**Part two**

*Collections*

# Let's talk some more about collections

- Real-life software generally involves collections.
  - Sometimes, we want to deal with one thing a time, more usually many things at once.
  - These are the kinds of things we want to do with collections:
    - find if the collection is empty (*isEmpty*);
    - find the number of elements in the collection (*length/size*);
    - (optionally) select a specific element by position or key (*get*);
    - traverse each element in succession, applying a side-effect function to each (*foreach*);
    - create a new collection based on the original, but with only those elements that satisfy a predicate (*filter*);
    - traverse each element in succession, applying a function to each, thus yielding a new collection (*map*, *flatMap*);
    - traverse each element in succession, applying a function to each element and an accumulator, thus yielding a value (*reduce*);
    - create a new collection based on the original, but with new element(s) (concatenate or “*cons*”).

# A simple definition of List

- This is (more or less) what I wrote on the board last time:

```
package edu.neu.coe.scala.list
trait List {
 def length: Int
}
case object Nil extends List {
 def length: Int = 0
}
case class Cons (head: Int, tail: List) extends List {
 def length: Int = 1 + tail.length
}
object List {
 def apply(as: Int*): List =
 if (as.isEmpty) Nil
 else Cons(as.head, apply(as.tail: _*))
}
```

First, we define some behavior in a trait. For now, the only behavior we've defined for our *List* is the ability to get its length.

Remember *proof by induction*? There are two cases: the “base case” and the “inductive step”. We will typically (but not always) have two “case class/object” extensions of a trait.

This is the “companion” object for *List*. Any class or trait can have such a companion object. Case classes always have one (via “syntactic sugar”)

# Parametric Types

- A very quick observation before we get into lists.
- We can define a *List* of *Int*, a *List* of *String*, etc.
- But we'd end up having to write all the same methods for each! That would be no good!!
- So, in Scala, all containers have an *underlying* type, including *List*. Such types are known as **Parametric types\*** because that's what they are. Scala doesn't really use the term *generics* (partly because it wasn't an afterthought).
- Unlike in Java, we cannot define a *List* without a parametric type because this would break type inference.

\* such types make up the *parameters* of a type—and are enclosed in `[]`, just like parameters of a method are enclosed in `()`.

# Lists and their methods

- Recap:

```
package edu.neu.coe.scala.list
trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A] (head: A, tail: List[A]) extends List[A]
object List {
 def apply[A](as: A*): List[A] =
 if (as.isEmpty) Nil
 else Cons(as.head, apply(as.tail: _*))
}
```

We have called the parametric type of the *List* “A”.  
A stands for any type, even a *List*[*B*]. I will explain  
the “+” shortly.

The name for *Cons* in the Scala library is “::”

- What are the methods that we expect *List* to implement?

- Let’s try a few signatures and think what they might mean:

- def x0: Boolean
- def x1: Int
- def x2: A
- def x3: List[A]
- def x4(x: Int): Option[A]
- def x5(f: A=>Boolean): List[A]
- def x6(f: A=>Boolean): Option[A]
- def x7[B](f: A=>B): List[B]
- def x8[B](f: A=>List[B]): List[B]
- def x9(f: A=>Unit): Unit

actually, there are a couple of plausible methods  
which yield an A

we are defining an “algebra”  
for the *List* type

# List methods (“SOE”)

```
def x0: Boolean = this match {case Nil => true; case _ => false }

def x1: Int = this match {
 case Nil => 0
 case Cons(hd, tl) => 1 + tl.x1
}

def x2a: A = this match {
 case Nil => throw new Exception("logic error")
 case Cons(hd, tl) => hd
}

// Alternative interpretation
def x2b: A = this match {
 case Nil => unit(0)
 case Cons(hd, tl) => hd + tl.x2
}

def x3: List[A] = this match {
 case Nil => Nil;
 case Cons(hd, tl) => tl
}

def x4(x: Int): Option[A] = {
 @tailrec def inner(as: List[A], x: Int): Option[A] = as match {
 case Nil => None
 case Cons(hd, tl) => if (x == 0) Some(hd) else inner(tl, x - 1)
 }
 if (x < 0) None else inner(this, x)
}
```

Will not compile: need an operator + that can combine two A objects into another A. Need unit function, too.



# List methods (higher-order functions)

- ```
def x5(f: A=>Boolean): List[A] = this match {
    case Cons(hd,tl) => val ftl = tl.x5(f); if (f(hd)) Cons(hd, ftl) else ftl
    case Nil => Nil
}
```
 - ```
def x6(f: A=>Boolean): Option[A] = this match {
 case Cons(hd,tl) => if (f(hd)) Some(hd) else tl.x6(f)
 case Nil => None
}
```
  - ```
def x7[B](f: A=>B): List[B] = this match {
    case Cons(hd,tl) => Cons(f(hd),tl.x7(f))
    case Nil => List[B]()
}
```
 - ```
def ++[B >: A](x: List[B]): List[B] = this match {
 case Nil => x
 case Cons(hd,tl) => Cons(hd, tl ++ x)
}
```
  - ```
def x8[B](f: A=>List[B]): List[B] = this match {
    case Cons(hd,tl) => f(hd) ++ tl.x8(f)
    case Nil => List[B]()
}
```
 - ```
def x9(f: A=>Unit): Unit = this match {
 case Cons(hd,tl) => f(hd); tl.x9(f)
 case Nil => Unit
}
```
- We need a way to concatenate two lists.
- But can't we just use *A* as the type of both lists? No: co-/contra-variance

# Giving the methods names:

- `def isEmpty: Boolean = this match {case Nil => true; case _ => false }`
- `def length: Int = this match {  
 case Nil => 0  
 case Cons(hd, tl) => 1 + tl.length  
}`
- `def head: A = this match {  
 case Nil => throw new Exception("logic error")  
 case Cons(hd, tl) => hd  
}`
- `def sum: A = this match {  
 case Nil => unit(0)  
 case Cons(hd, tl) => hd + tl.sum  
}`
- `def tail: List[A] = this match {  
 case Nil => Nil;  
 case Cons(hd, tl) => tl  
}`
- `def get(x: Int): Option[A] = {  
 @tailrec def inner(as: List[A], x: Int): Option[A] = as match {  
 case Nil => None  
 case Cons(hd, tl) => if (x == 0) Some(hd) else inner(tl, x - 1)  
 }  
 if (x < 0) None else inner(this, x)  
}`

Will not compile: need an operator + that can combine two A objects into another A. Need *unit* function, too.



# and names for the higher-order functions...

- ```
def filter(f: A=>Boolean): List[A] = this match {
    case Cons(hd,tl) => val ftl = tl.filter(f); if (f(hd)) Cons(hd, ftl) else ftl
    case Nil => Nil
}
```
 - ```
def find(f: A=>Boolean): Option[A] = this match {
 case Cons(hd,tl) => if (f(hd)) Some(hd) else tl.find(f)
 case Nil => None
}
```
  - ```
def map[B](f: A=>B): List[B] = this match {
    case Cons(hd,tl) => Cons(f(hd), tl.map(f))
    case Nil => List[B]()
}
```
 - ```
def ++[B >: A](x: List[B]): List[B] = this match {
 case Nil => x
 case Cons(hd,tl) => Cons(hd, tl ++ x)
}
```
  - ```
def flatMap[B](f: A=>List[B]): List[B] = this match {
    case Cons(hd,tl) => f(hd) ++ tl.flatMap(f)
    case Nil => List[B]()
}
```
 - ```
def foreach(f: A=>Unit): Unit = this match {
 case Cons(hd,tl) => f(hd); tl.foreach(f)
 case Nil => Unit
}
```
- We need a way to concatenate two lists.
- But can't we just use **A** as the type of both lists? No: co-/contra-variance

# Method categories

- (Refer to my [StackOverflow answer](#) for the original)
- Let's assume a type *Bunch[T]* which extends *Iterable[T]* (the base trait for all Scala collections)
- In the following, *U* is a supertype of *T*, *V* is any *T-related type*:
  - *traversing*: there are actually two subclasses:
    - *shape-preserving*: these define a return type of *Bunch[U]*; example: *map*;
    - *non-shape-preserving*: these define a return type of *Iterator[T]*, *Iterable[T]*, *Iterable[U]*, etc.; example: *iterator*;
  - *side-effecting*: these define a return type of *Unit*; example *foreach*;
  - *selecting*: these define a return type of *T*; example *head*;
  - *maybeSelecting*: these define a return type of *Option[T]*
  - *aggregation*: these define a return type of *V*; example: *foldLeft*;
  - *testing*: these define a return type of *Boolean*; example: *isEmpty*;
  - etc. etc.

4.7

Updated: 2023-03-28

# Monoids, Functors and Monads

© 2015-23 Robin Hillyard



# Monoids, Functors and Monads

- These terms come from category theory. But really, their definitions as far as Scala is concerned are quite simple:
    - for instance, a **monoid** is just the type of thing we were getting at with our *List.x2* method (a.k.a *sum*)—where List’s underlying type must be a *monoid*:  
using 2 operands together
    - something that can be operated on dyadic-ally and something which has a “zero” (identity) value:  
or 1 for multiplication
  - You don’t really need to remember all this stuff. I will review what’s important at the end.
- ```
def sum: A = {  
    @tailrec def inner(as: List[A], x: A): A = as match {  
        case Nil => x  
        case Cons(hd, tl) => inner(tl, x++hd)  
    }  
    inner(this, 0)  
}
```

Monoids

- A **monoid** consists of the following:
 - a type A
 - an associative binary operation, op , that takes two values of type A and combines them into one such that:
$$op(op(x, y), z) == op(x, op(y, z))$$
for any $x: A$, $y: A$, $z: A$.
 - an “identity” (zero) value: $\text{identity}: A$ that is an identity for op , i.e.
$$op(x, \text{identity}) == x \text{ for any } x: A.$$
- For example:

```
trait Monoid[A] {  
    def op(a1: A, a2: A): A  
    def identity: A  
}  
object Monoid {  
    val stringMonoid = new Monoid[String] {  
        def op(a1: String, a2: String) = a1 + a2  
        val identity = ""  
    }  
    def listMonoid[A] = new Monoid[List[A]] {  
        def op(a1: List[A], a2: List[A]) = a1 ++ a2  
        val identity = Nil  
    }  
}
```

Monoids (2)

```
def reduce(f: (A,B) => A + B )
```

- There are two important functions on collections that we met briefly in Recursion: *foldRight* and *foldLeft*:

```
def foldLeft[A,B](z: B)(f: (B, A) => B): B  
def foldRight[A,B](z: B)(fr: (A, B) => B): B
```

- These methods are equivalent providing that *f* and *fr* are associative—they simply work through a container in different directions—but only *foldLeft* is tail-recursive for *List* so we'll concentrate on that.
- Here's *foldLeft* implemented for our own *List* class from week 2:

```
case class Cons[A](h: A, t: List[A]) extends List[A] {  
    def foldLeft[B](z: B)(f: (B, A) => B): B = t.foldLeft(f(z,h))(f)  
}  
  
case object Nil extends List[Nothing] {  
    def foldLeft[B](z: B)(f: (B, Nothing) => B): B = z  
}
```

Let's reimplement sum on List in terms of *foldLeft*...

```
def sum[B](xs: List[A]): B = xs.foldLeft(B.zero)(B.plus)
```

- Just one snag: *B.zero* and *B.plus* are not defined. But if, for any *B*, *zero* and *plus* were defined we'd be all set.

Monoids (2a)

- We can do it with implicits (and we can keep B the same as A)...

```
def sum[A: Numeric](xs: List[A]): A = {  
    val an = implicitly[Numeric[A]]  
    xs.foldLeft(an.zero)(an.plus)  
}
```

Numeric (Ex: Int) provides zero and plus method in predef of Scala

Monoids (3)

- Let's go back to our *stringMonoid* and create an *intMonoid* too (we'll also make *op* explicit as *plus*):

```
object Monoid {  
    val stringMonoid = new Monoid[String] {  
        def plus(a1: String, a2: String) = a1 + a2  
        val zero = ""  
    }  
    def intMonoid[A] = new Monoid[Int] {  
        def plus(a1: Int, a2: Int) = a1 + a2  
        val zero = 0  
    }  
}
```

- So we would have:

```
def sum[B]: B = foldLeft(intMonoid.zero)(intMonoid.plus(_,_))
```

- Therefore, in general, a *Monoid* is something that is *foldable*. We could make this explicit by defining the following type constructor:

```
trait Foldable[F[_]] extends Functor[F] {  
    def foldLeft[A,B](z: B)(f: (B, A) => B): B  
    def foldRight[A,B](z: B)(f: (A, B) => B): B  
}  
def foldableList[A] = new Foldable[List] {  
    def map[A,B](as: List[A])(f: A => B): List[B] = as map f  
    def foldLeft[A,B](z: B)(f: (B, A) => B): B = ??? // tail recursive  
    def foldRight[A,B](z: B)(f: (A, B) => B): B = ??? // NOT tail recursive  
}
```

Functor defines *map*—we'll meet it next. “F[_]" is a type constructor that takes an argument of a type, i.e. a higher-kinded type.

Functors

- Functor is another term from category theory:
 - a *functor* is just a mapping between two categories (or types in Scala):

```
trait List[+A] { def map[B](f: A=>B): List[B] }
trait Option[+A] { def map[B](f: A=>B): Option[B] }
trait LazyList[+A] { def map[B](f: A=>B): LazyList[B] }
etc. etc. etc.
```
 - Notice anything?
 - All the definitions are identical—only the implementations differ.
 - Repetitive code like this is anathema to functional programmers!
 - Let's define a functor trait:

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
object Functor {
  def listFunctor = new Functor[List] {
    def map[A,B](as: List[A])(f: A => B): List[B] = as map f
  }
}
```

Monads

- Do Monads matter?
- Yes, they are perhaps the most important aspect of functional programming. Here are some of the things they help with:
- Sequencing:
 - Because functional programs don't really have expressions, there isn't an obvious way to make sure one thing happens after another: monads let us do that.
- Wrapping:
 - We frequently need to wrap one or more values inside some sort of container: to protect it/them, to group the elements together, to model union types (where the value is one thing or another), to model values that aren't materialized yet—future values, in other words.

Monads (1)

- Monad, too, comes from category theory. You've heard me mention monads already:
 - A *Monad* is a Functor (i.e. it implements *map*):

```
trait List[+A] { def map[B](f: A=>B): List[B] }
trait Option[+A] { def map[B](f: A=>B): Option[B] }
trait LazyList[+A] { def map[B](f: A=>B): LazyList[B] }
etc. etc. etc.
```
 - Remember how *map2* was basically identical for several different container types? These were all *monads*! So let's define *map2* once and for all...

```
trait Monad[F[_]] extends Functor[F] {
  def map2[A,B,C](ma: F[A], mb: F[B])(f: (A,B)=>C): F[C] =
    flatMap(ma)(a => map(mb)(b => f(a,b)))
}
```
 - But *flatMap* isn't defined :(Except that since this definition is the canonical definition of *map2* and we are essentially defining monad as things which support *map2*, ergo *Monad* must define *flatMap* too:

```
trait Monad[F[_]] extends Functor[F] {
  def flatMap[A,B](ma: F[A])(f: A=>F[B]): F[B]
  def map2[A,B,C](ma: F[A], mb: F[B])(f: (A,B)=>C): F[C] =
    flatMap(ma)(a => map(mb)(b => f(a,b)))
}
```

Monads (2)

- Actually, there's a certain flexibility in exactly how we define the primitive methods of *Monad*. Let's think about *map*. If we had a method *unit* which took a value and simply wrapped it (like a single-element *List*) then we could actually write *map* in terms of *flatMap*:

```
trait Monad[F[_]] extends Functor[F] {  
    def unit[A](a: => A): F[A] // abstract  
    def flatMap[A,B](ma: F[A])(f: A=>F[B]): F[B] // abstract  
    def map[A,B](ma: F[A])(f: A=>B): F[B] = flatMap(ma)(a => unit(f(a)))  
    def map2[A,B,C](ma: F[A], mb: F[B])(f: (A,B)=>C): F[C] =  
        flatMap(ma)(a => map(mb)(b => f(a,b)))  
}
```



In this definition of *Monad*, *unit* and *flatMap* are abstract methods—they must be defined by implementers of *Monad*. The other two methods are concrete methods defined in terms of the first two.

- So, *unit+flatMap* is one possible form.
unit is an instance method because it needs parametric types, we do not reference any state from its surrounding class
- But why do we care about all this stuff? Because of compositability.

Example

Here's an example of using a for-comprehension (with IO monad) to parse and process a CSV file (from TableParser):

```
def doMain(triedResource: Try[URL])(implicit random: Random): IO[String] =  
  for {  
    // get the URL for either the complete file or a sample file.  
    url <- IO.fromTry(triedResource)  
    // open/close resource and parse it as a Table[Crime]  
    ct <- IOUsing(Try(Source.fromURL(url)))(x => Table.parseSource(x))  
    // filter according to validity and then convert rows to CrimeBrief.  
    lt <- IO(ct.filterValid.mapOptional(m => m.brief))  
    // sample 1 in every (approximately) 450 rows.  
    st <- IO(lt.sample(450))  
    // write the table as a String in CSV format.  
    w <- st.toCSV  
  } yield w
```

Using[R <: Autocloseable, T]
(r :=> R)(f : R => Try[T]) : Try[T]

Autocloseable ensures that once its done, all resources are closed

Monads (4)

- Articles/videos which (try to) explain monads:
 - [Monads Everywhere \(Rock the JVM\)](#)
- If that doesn't do it for you, you can try these others:
 - <https://youtu.be/d-dy1x33moA>
 - <https://medium.com/@lettier/your-easy-guide-to-monads-applicatives-functors-862048d61610>
 - <https://medium.com/@sinisalouc/demystifying-the-monad-in-scala-cc716bb6f534>
 - <https://medium.com/@yuriigorbylov/monads-and-why-do-they-matter-9a285862e8b4>
 - <https://medium.com/zendesk-engineering/dont-fear-the-monad-f424260f29f6>
 - <https://medium.com/@evinsellin/teaching-monads-slightly-differently-2af62c4af8ce>

Review: what do you need to remember?

- A type that implements *map* is a **functor**.
- A **monad** is a functor but not all functors are monads.
- For-comprehensions work on *monads*.
- The *reduce* method works when the underlying type is a **monoid**.
- Any type which defines both *map* and *flatMap* **is a monad**—you don't have to declare it explicitly as a monad!
- *map* can be defined as *flatMap*($a \Rightarrow \text{unit}(\underline{f(a)})$)

4.9

Parsing and DSLs

© 2015,2024 Robin Hillyard



Northeastern
University

What is a Domain-specific language?

- Any time you create a structured format for writing/reading particular objects, it's a DSL.
- XML, JSON, etc. are *not* themselves DSLs but they can be used to create DSLs. However, you don't need either of these types of markup language to create a DSL
- Examples:
 - KML
 - Product inventories;
 - Lens definitions;
 - Workflows...

What exactly is parsing and why do we need it?

- Any time we have some input in some sort of serialized form (such as a *CharSequence*), and we want to turn it into an object that we can reference in an expression, we need a parser. In other words, we wish to create a domain-specific language (DSL).
 - Some examples we've already encountered:
 - Creating *Rational* objects;
 - Reading the CSV file into our *Movie* program;
 - Reading JSON strings returned by Google Finance (*lab-actors*) (or from the *Poets*);
 - Defining the rules for dealing with "options" (*lab-actors*).
 - In all these cases, we used a **regular expression** with a match:

```
object NumberPredicate {  
    def apply...  
    def apply(predicate: String): NumberPredicate = {  
        val rPredicate = """^\\s*(\\w+)\\s*([=>]{1,2})\\s*(-?[0-9]+\\.?[0-9]* )\\s*$""".r  
        predicate match {  
            case rPredicate(v, o, n) => apply(v, o, n)  
            case _ => throw new Exception(s"predicate: $predicate is malformed")  
        }  
    }  
}
```

Example: Rational (1)

```
implicit class RationalHelper(val sc: StringContext) extends AnyVal {  
    def r(args: Any*): Rational = {  
        val strings = sc.parts.iterator  
        val expressions = args.iterator  
        val sb = new StringBuffer()  
        while(strings.hasNext) {  
            val s = strings.next  
            if (s.isEmpty) {  
                if(expressions.hasNext)  
                    sb.append(expressions.next)  
            }  
            else  
                sb.append(s)  
        }  
        if(expressions.hasNext)  
            throw new RationalException(s"r: ignored: ${expressions.next}")  
        else  
            Rational(sb.toString)  
    }  
}
```

Example: Rational (2)

- def apply(x: String): Rational = {
 val rRat = """^\\s*(\\d+)\\s*(/\\s*(\\d+)\\s*)?""".r
 val rDec = """^-?(\\d|(\\d+,?\\d+))*\\.\\d+(e\\d+)?""".r
 x match {
 case rRat(n, _, null) => *Rational*(n.toLong)
 case rRat(n, _, d) => *normalize*(n.toLong, d.toLong)
 case rRat(n) => *Rational*(n.toLong) // shouldn't need this but we do
 case rDec(w, _, f, null) => *Rational*(*BigDecimal*.apply(w + f))
 case rDec(w, _, f, e) => *Rational*(*BigDecimal*.apply(w + f + e))
 case _ => throw new RationalException(s"invalid: \$x")
 }
}

Example: Rating

```
object Rating {  
    val rRating = """^(\w*)(-(\d\d))?""".r  
    /**  
     * Alternative apply method for the Rating class such that a single  
     * String is decoded  
     * @param s a String made up of a code, optionally followed by a dash  
     * and a number, e.g. "R" or "PG-13"  
     * @return a Rating  
     */  
    def apply(s: String): Rating = s match {  
        case rRating(code, _, null) => apply(code, None)  
        case rRating(code, _, age) => apply(code, Try(age.toInt).toOption)  
        case _ => throw new Exception(s"parse error in Rating: $s")  
    }  
}
```

Parsing (2)

- Matching on regular expressions works OK...
 - and REs are the basis of all parsing...
 - but the API is Java-centric and involves dealing with null...
 - and, moreover, the API isn't easy to use and such parsers don't *compose* very well.

Parsing (2a)

- Let's try a more functional parser:

```
trait Parser[-S,+T] extends (S => T)
```

That's to say, we extend *Function1[S,T]*

- This would work fine: it takes input of type *S* and returns a result of type *T*. But what if we want to combine this parser with another which takes whatever input is left over and then returns something of type *U* ?
- We're going to need something that returns not a *T* but a tuple of *S* and *T*. Or we could define a trait *ParseResult[S,T]*. Then, from this result, we could get both our *T* value and the rest of the input. What do we need for that?

```
trait Parser[S,+T] extends (S => ParseResult[S,T])
trait ParseResult[S,+T]
case class Success[S,+T](result: T, nextInput: S) extends ParseResult[S,T]
case class Failure[S,+T](message: String, nextInput: S) extends ParseResult[S,T]
```

- We could write our own parser that way. In fact, the Scala classes are similar but not quite the same: *Parser* takes only one parametric type *T* because input is defined via an abstract type defined in *Parser*.

Parsing (2b)

- We're going to need some new compound types:
 - We will need to be able to represent the following standard grammatical constructs in our *Parser*:
 - $T_1 \xrightarrow{T_1 \sim T_2}$ followed by T_2 —we could simply use (T_1, T_2) but Scala defines a type constructor \sim so we can write $T_1 \sim T_2$.

$T_1 \mid T_2$



This is effectively just a case class

- $T_1 \text{ otherwise } T_2$ —in other words, alternation: if we can parse the input as a T_1 , that's what we get, otherwise we try to parse it as a T_2 .
- $\text{Maybe } T$, that's to say 0 or 1 T s, equivalent to $\text{Option}[T]$.
- $\text{Sequence of } T$, that's to say any number of T s (including zero), equivalent to $\text{Seq}[T]$.

Parsing (3)

- OK, now we just need to be able to define the grammar that our parser can operate on:
 - Take a look at this set of “productions” in BNF (Backus-Naur form) followed by examples: { } means any number of enclosed string that is followed by

```
expr ::= term { "+" term | "-" term }.
term ::= factor {"*" factor | "/" factor}.
factor ::= floatingPointNumber | "(" expr ")".
```

- 1+4.5-3 is an *expr*, 2*3.14/5 is a *term*; 3.1415927 is a *factor*, (7-5) is also a *factor*.
- The **Scala Parser Combinator** library allows us to code this parser with only a few substitutions:

```
import scala.util.parsing.combinator._
class Arith extends JavaTokenParser {
    def expr: Parser[Any] = term ~ rep(+"~term | "-~term);
    def term: Parser[Any] = factor ~ rep(*~factor | /*~factor);
    def factor: Parser[Any] = floatingPointNumber | "(" ~expr ~")";
}
```

“~” replaces “”; “rep(“ replaces “{”; “”)
replaces “””; “;” replaces “.” [although
those “;” are entirely optional]

each method defines a *Parser[Any]*

Parsing (4)

- Let's try it in the REPL:

```
scala> val p = new Arith  
p: Arith = Arith@78291b30  
  
scala> val x = p.parseAll(p.expr,"1")  
x: p.ParseResult[Any] = [1.2] parsed: ((1~List())~List())
```

We want to apply, specifically, the `expr` parser to “1”

TMI: Not very helpful output but it can be useful when debugging!

consumed text up to line 1, column 2

- That's not quite what we want!

- We can get the result's “value”:

```
scala> x.get  
res1: Any = ((1~List())~List())
```

- But that's not super useful either. For a start, it's an “Any” and secondly, what we've got is the concatenation of all the intermediate parse results. But, for now, we're not so interested in the internal workings of the parser.
- So, how can we get the value “1” out of this?

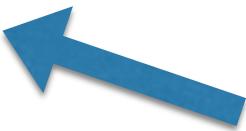
- First, we need to understand how the *Parser* operators work:
 - https://www.javadoc.io/doc/org.scala-lang.modules/scala-parser-combinators_2.13/latest/scala/util/parsing/combinator/index.html will take you to the root package
 - From there, you can click on *Parsers* to find:

```
p1 ~ p2 // sequencing: must match p1 followed by p2
p1 | p2 // alternation: must match either p1 or p2, with preference given to p1
p1.? // optionality: may match p1 or not
p1.* // repetition: matches any number of repetitions of p1
```

- Now, you can understand what the parsers we defined before do:

```
def expr: Parser[Any] = term~rep("+"~term | "-"~term)
def term: Parser[Any] = factor~rep("*"~factor | "/"~factor)
def factor: Parser[Any] = floatingPointNumber | "("~":"expr~":"")"
```

floatingPointNumber is itself a *Parser*, defined in *JavaTokenParsers*,
https://www.javadoc.io/doc/org.scala-lang.modules/scala-parser-combinators_2.13/latest/scala/util/parsing/combinator/index.html



- There are many methods defined in *Parsers*, for example *rep*.

Parsing (6)

- These operators/methods work as follows:
 - Any (constant) string returns itself (as a *String*)
 - Any regular expression parser similarly returns the matched string(s)
 - A sequential composition $P \sim Q$ returns both P and Q . This returns a “tilde” class written $P \sim Q$ or, if you prefer, $\sim[P, Q]$
 - An alternation returns $P | Q$ which is either P or Q but preferably P
 - A repetition $rep(P)$ or $repsep(P, \text{separator})$ returns a $List[P]$
 - An option $opt(P)$ returns an $Option[P]$
java tokens usually disregards whitespaces

Parsing (7)

- We're getting close but not quite there yet...
 - *Parser* defines the `^^` operator such that a parser definition of the form `P ^^ f` parses the input just like `P` (yielding result `R`) but the result of the `^^` operator is actually `f(R)`.
 - For example:
 - `floatingPointNumber ^^ (_.toDouble)`



Does it bother you that `^^` seems to work just like `map`? It bothered me! Then I found that `^^` actually invokes `map` but also records a name.

- Now we're ready to implement our arithmetic parser...

```
expr ::= term { "+" term | "-" term }.
term ::= factor {"*" factor | "/" factor}.
factor ::= floatingPointNumber | "(" expr ")".
```

Parsing (8)

```
class Arith extends JavaTokenParsers {

trait Expression {
  def eval: Double
}

trait Factor extends Expression

case class Expr(t: Term, ts: List[String ~ Term]) extends Expression {
  def term(t: String ~ Term): Double = t match {
    case "+" ~ x => x.eval;
    case "-" ~ x => -x.eval
    case x ~ _ => throw ParseException(s"Expr: $x unsupported")
  }

  def eval: Double = ts.foldLeft(t.eval)(_ + term(_))
}

case class Term(f: Factor, fs: List[String ~ Factor]) extends Expression {
  def factor(t: String ~ Factor): Double = t match {
    case "*" ~ x => x.eval;
    case "/" ~ x => 1 / x.eval
    case x ~ _ => throw ParseException(s"Term: $x unsupported")
  }

  def eval: Double = fs.foldLeft(f.eval)(_ * factor(_))
}

case class FloatingPoint(x: Any) extends Factor {
  def eval: Double = x match {
    case x: String => x.toDouble
    case _ => throw new RuntimeException("FloatingPoint: logic error: x is not a String")
  }
}

case class Parentheses(e: Expr) extends Factor {
  def eval: Double = e.eval
}

def expr: Parser[Expr] = term ~ rep("+~term | "-~term) ^^ {
  case t ~ (r: List[String ~ Term]) => Expr(t, r)
}

def term: Parser[Term] = factor ~ rep("*~factor | "/"~factor) ^^ {
  case f ~ (r: List[String ~ Factor]) => Term(f, r)
}

def factor: Parser[Factor] = (floatingPointNumber | "(" ~> expr <~ ")") ^^ {
  case e: Expr => Parentheses(e)
  case s => FloatingPoint(s)
}
}
```

Parsing (9)

```
scala> import edu.neu.coe.scala.parse._  
import edu.neu.coe.scala.parse._  
scala> val parser = new Arith  
parser: edu.neu.coe.scala.parse.Arith = edu.neu.coe.scala.parse.Arith@48326b9d  
scala> parser.parseAll(parser.expr, "1").get.eval  
res0: Double = 1.0  
scala> parser.parseAll(parser.expr, "1*2+1-3/2").get.eval  
res1: Double = 1.5  
scala> parser.parseAll(parser.expr, "1*2+1-pi/2").get.eval  
java.lang.RuntimeException: No result when parsing failed  
  at scala.sys.package$.error(package.scala:27)  
  at scala.util.parsing.combinator.Parsers$NoSuccess.get(Parsers.scala:176)  
  at scala.util.parsing.combinator.Parsers$NoSuccess.get(Parsers.scala:162)  
... 43 elided
```

pi is not defined

Oops! throwing an exception isn't very nice—but that's to be expected when we invoke get.

- We can build in a little error handling to avoid this:

```
def expr: Parser[Expr] = term ~ rep("+") ~ term | "-" ~ term | failure("expr")) ^^ {  
  case t ~ (r: List[String ~ Term]) => Expr(t, r)  
}  
  
def term: Parser[Term] = factor ~ rep("*") ~ factor | "/" ~ factor | failure("term")) ^^ {  
  case f ~ (r: List[String ~ Factor]) => Term(f, r)  
}  
  
def factor: Parser[Factor] = (floatingPointNumber | "(" ~> expr <~ ")" | failure("factor")) ^^ {  
  case e: Expr => Parentheses(e)  
  case s => FloatingPoint(s)  
}  
  
scala> parser.parseAll(parser.expr, "1*2+1-pi/2")  
res1: parser.ParseResult[parser.Expr] =  
[1.7] failure: factor  
1*2+1-pi/2  
^
```

unable parse at position 7 o f1st string

Parsing (10)—Rational

```
trait RationalNumber { def value: Try[Rational] }
class RationalParser extends JavaTokenParsers {
  def parse(w: String): Try[RationalNumber] = parseAll(number, w) match {
    case Success(t, _) => scala.util.Success(t)
    case Failure(m, _) => scala.util.Failure(RationalParserException(m))
    case Error(m, _) => scala.util.Failure(RationalParserException(m))
  }
  case class WholeNumber(sign: Boolean, digits: String) extends RationalNumber {
    override def value: Try[Rational] = scala.util.Success(Rational(BigInt(digits)).applySign(sign))
  }
  object WholeNumber {
    val one: WholeNumber = WholeNumber(sign = false, "1")
  }
  case class RatioNumber(numerator: WholeNumber, denominator: WholeNumber) extends RationalNumber {
    override def value: Try[Rational] = for (n <- numerator.value; d <- denominator.value) yield n / d
  }
  case class RealNumber(sign: Boolean, integerPart: String, fractionalPart: String, exponent: Option[String]) extends RationalNumber {
    override def value: Try[Rational] = {
      val bigint = BigInt(integerPart + fractionalPart)
      val exp = exponent.getOrElse("0").toInt
      Try(Rational(bigint).applySign(sign).applyExponent(exp - fractionalPart.length))
    }
  }
  def number: Parser[RationalNumber] = realNumber | ratioNumber
  def ratioNumber: Parser[RatioNumber] = simpleNumber ~ opt("/") ~> simpleNumber ^^ { case n ~ maybeD => RatioNumber(n, maybeD.getOrElse(WholeNumber.one)) }
  def simpleNumber: Parser[WholeNumber] = opt("-") ~ wholeNumber ^^ { case so ~ n => WholeNumber(so.isDefined, n) }
  def realNumber: Parser[RealNumber] = opt("-") ~ wholeNumber ~ ("." ~> wholeNumber) ~ opt(E ~> wholeNumber) ^^ { case so ~ integerPart ~ fractionalPart ~ expo =>
    RealNumber(so.isDefined, integerPart, fractionalPart, expo)
  }
  private val E = "[eE]" .r
}
object RationalParser {
  val parser = new RationalParser
  def parse(s: String): Try[Rational] = parser.parse(s).flatMap(_.value)
}
case class RationalParserException(m: String) extends Exception(m)
```

Parsing (wrap-up)

- Best sources of information for Parsing:
 - *Programming in Scala* (Odersky & Spoon)
 - [Latest Release](#)
 - Code examples:
http://booksites.artima.com/programming_in_scala_2ed/examples/html/ch3_3.html
 - A somewhat more practical document on this (though I say so myself):
<http://scalaprof.blogspot.com/2015/10/scalas-parser-combinators.html>
 - And a rather more advanced parser problem written up here:
https://www.javadoc.io/doc/org.scala-lang.modules/scala-parser-combinators_2.13/latest/scala/util/parsing/combinator/index.html
 - My projects that do a lot of parsing:
 - [TableParser](#)
 - [Matchers](#)
 - [KmlDoc](#)
 - [Number](#)

Updated: 2021-04-05

4.10

Tour of the Scala API

© 2018 Robin Hillyard



Brief interlude on partial functions

- A partial function is a function which is defined for a subset of possible input values.
trait *PartialFunction* extends *Function1*:

```
trait PartialFunction[-A, +B] extends (A) ⇒ B
abstract def isDefinedAt(x: A): Boolean
```

- Some examples:

```
scala> val fraction = new Function[Int, Int] {
    |   def apply(d: Int) = 1 / d
    |
fraction: Function[Int,Int] = <function1>
scala> List(1,0) map fraction
java.lang.ArithmetricException: / by zero
  at $anon$1.apply$mcII$sp(<console>:11)
  at $anon$1.apply(<console>:11)
  at $anon$1.apply(<console>:10)
  at scala.collection.immutable.List.map(List.scala:277)
... 33 elided
scala> val fraction = new PartialFunction[Int, Int] {
    |   def apply(d: Int) = 1 / d
    |   def isDefinedAt(d: Int) = d != 0
    |
fraction: PartialFunction[Int,Int] = <function1>
scala> List(1,0) collect fraction
res1: List[Int] = List(1)
scala> List(1, "1") collect { case i: Int ⇒ i + 1 }
res2: List[Int] = List(2)
```



Note that, if we use *map* instead of *collect*, it will compile just fine.
But each of these would throw an exception.

Collections (1)

- ***Iterable[A]***
 - Methods defined (in *IterableLike*, *IterableOnce*, etc.):
 - (iteration) `foreach`, `iterator`, `grouped`, `slice`, `sliding`
 - (concatenation) `++` and `++:` append two iterables together.
 - (monad) `map`, `flatMap`, `filter/Not` and `collect`: `collect` takes a partial function
 - (conversions) `to[Array]`, `to[List]`, `to[Iterable]`, `to[Seq]`, `to[IndexedSeq]`, `to[LazyList]`, `to[Set]`, `to[Map]`: all convert (but only if necessary) to the appropriate type
 - (copying) `copyToBuffer`, `copyToArray`
 - (size) `isEmpty`, `nonEmpty`, `size`, and `hasDefiniteSize`
 - (element retrieval) `head`, `last`, `headOption`, `lastOption`, and `find`

caters to infinite collections like LazyList

may not be meaningful if collection is not ordered.

Collections (2)

- Iterable continued...
 - Continuing other methods defined:
 - (sub-collection retrieval) *tail*, *init*, *slice*, *take*, *drop*, *takeWhile*, *dropWhile*, *filter*, *filterNot*, *withFilter* These are similar: see below
 - (subdivision) *splitAt*, *span*, *partition*, *groupBy*
 - element tests (by predicate) *exists*, *forall*, *count*
 - (accumulating) *foldLeft*, *foldRight*, *reduceLeft*, *reduceRight*, *scan*, *scanLeft*, *scanRight* works if underlying type is *Numeric* or *Ordered*
 - (specific folds) *sum*, *product*, *min*, *max* optional
 - (string operations) *mkString(start, sep, end)*, *asString*, *stringPrefix*
 - (views) *view*, *view(from,to)*

partition(p: A => Boolean) => (T[A],T[A])

span(p: A => Boolean) => (T[A],T[A])

splitAt(n: Int) => (T[A],T[A])

groupBy(f: A => K) => Map[K,T[A]]

Collections (3)

- ***Iterable***

- Provides an iterator:

```
def foreach(f: Elem => Unit): Unit = {  
    val it = iterator  
    while (it.hasNext) f(it.next())  
}
```

- (other iterators) *grouped*, *sliding*
 - (*GenIterable*):
 - *zip*

may be overridden by subclasses;
an *Iterator* is not itself a collection
but can be a generator in for-comp



Collections (4)

- Sequence traits:
 - *Seq* (extends *Iterable*), *IndexedSeq*, *LinearSeq* (both extend *Seq*)
 - (indexing and length) *apply*, *isDefinedAt*, *length*, *indices*, and *lengthCompare*
 - (index searches) *indexOf*, *lastIndexOf*, *indexOfSlice*, *lastIndexOfSlice*, *indexWhere*, *lastIndexWhere*, *segmentLength*, *prefixLength*
 - (element addition) *+:*, *:+*, *padTo*
 - (updates) *updated*, *patch*
 - (sorting) *sorted*, *sortWith*, *sortBy*
 - (reversal) *reverse*, *reverseIterator*, *reverseMap*
 - (comparison) *startsWith*, *endsWith*, *contains*, *containsSlice*, *corresponds*
 - (multiset) *intersect*, *diff*, *union*, *distinct*
- extends *PartialFunction[Int]*. Methods all the same but efficiency of operations varies
- 

Collections (5)

- Other traits/types:
 - *List, Map, Set, Array**
 - Immutable collections
 - Mutable collections
 - See it all here: <https://docs.scala-lang.org/overviews/collections-2.13/introduction.html>
 - And look up individual types/methods here: <http://www.scala-lang.org/api/2.13.5/#package>

* *Array[T]* has two implicit conversions: to *ArrayOps[T]* and *WrappedArray[T]* which extends *Seq[T]*.

Updated: 2021-03-25

4.11

Syntactic Sugar

© 2018 Robin Hillyard



Syntactic Sugar

- Scala has a neat feature called “syntactic sugar” which:
 - allows us to write a program in a very human-readable form that gets converted into more computer-readable form;
 - perhaps the most obvious example is the for-comprehension:

```
for(x <- List(1,2,3)) yield x*x
```

=> (i.e. is de-sugared into)

```
List(1,2,3).map {x => x*x}
```

and

```
for(x <- List(1,2,3); y <- List(4,5,6)) yield x*y
```

=>

```
List(1,2,3).flatMap { x => List(4,5,6).map { y => x*y } }
```

Syntactic Sugar (2)

- Desugaring continued...

- other examples:

```
val x = 1; x + x
```

=> (i.e. is de-sugared into)

```
val x: Int = 1; x.+ (x)
```

and

```
class X[Y : Numeric] {}
```

yields a constructor of the form

```
def <init>()(implicit ev$1: Numeric[Y]): this.X[Y] = {X.super.<init>();()}
```

and

```
List(1,2,3).map(_*2)
```

=>

```
val x = List(1, 2, 3).map(((x$1) => x$1.$times(2)))
```

Syntactic Sugar (3)

- Unary functions (i.e. instances of *Function1*):
 - $f(x) \rightarrow f.\text{apply}(x)$
 - most collection-type containers implement *Function1*, e.g. *List*, *Map*, *Set* but not *Option*, *Try*, *Either*.
- A unary method on an object (like *map*, for example) is a binary (dyadic) function on the object and the parameter:
 - $a \text{ map } f \rightarrow a.\text{map}(f)$
- But, a method/operator whose name ends in “：“, associates to the *right*:
 - $h :: t \rightarrow t.::(h)$
- And, for mutable objects, there is a built-in assignment method:
 - $x(y) = z \rightarrow x.\text{update}(y, z)$
- And, obviously, tuples:
 - $(x, y) \rightarrow \text{Tuple2}[X, Y](x, y)$
 - $(x, y, z) \rightarrow \text{Tuple3}[X, Y, Z](x, y, z)$



Note that *map* associates to the left: in this case, *a* is the receiver; *f* is the parameter

Syntactic sugar (4)

- And another example of our old friend the for-comprehension ([How does yield work](#)):

```
for { i <- List(1,2,3); x = i*3; if (x%2 == 0)} yield x →  
List(1,2,3) map (_ * 3) filter (_ %2 == 0)
```

```
for(x <- c1; y <- c2; z <- c3) yield {...} →  
c1.flatMap(x => c2.flatMap(y => c3.map(z => {...})))
```

Syntactic Sugar (5)

- It's important to realize that there is **no magic** in syntactic sugar (or any other aspect of Scala).
 - The desugaring process (the first process in the compiler's workflow) is a formal substitution scheme just as you'd expect
 - IntelliJ/IDEA offers a desugaring option;
 - You can see the result of desugaring a module by (e.g.):

```
scalac -Xprint:parser src/main/scala/edu/neu/coe/scala/FutureExercise.scala
```
 - You can see other options from the compiler by using:

```
scalac -Xshow-phases
```
 - You can see, for example, the “trees” for an expression:

```
scala -Xprint:typer -e "val x = Option(1); x match { case Some(y) => println(y); case _ => }"
```

Other Scala constructs

- call-by-value and call-by-name:

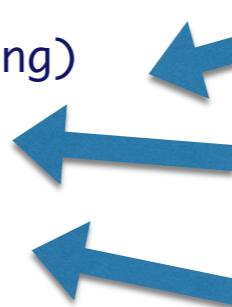
```
def f(x: X)  
def f(x: => X)
```



`=> X` is a nullary function that results in an `X`. Can also be written `() => X`

- anonymous functions:

```
List(1,2,3) (_.toString)  
List(1,2,3) (_*_ )  
List(1,2,3) (x =>x*x)
```



`_` stands for the obvious: a particular element of `m`

No. Doesn't work with `List(Int)` but would work with `List((Int,Int))` for instance.

This is OK as we have explicitly named the value

- varargs methods:

```
def sum(xs: Int*) = xs reduce (_+_)  
sum(List(1,2,3): _*)
```



`*` tells the compiler that the parameter `args` is a variable sequence of `Int`, not just one `Int`.

- tuples: defining

```
scala> val x = 1->"a"  
x: (Int, String) = (1,a)
```



For example, initializing a `Map`:
`val x = Map(1->"a", 2->"b", ...)`

Standard Imports

- When looking for operators, methods, implicit functions, values, it makes sense to know what's automatically imported:

```
import java.lang._    // http://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html
import scala._         // http://www.scala-lang.org/api/current/#scala.package
import scala.Predef._  // http://www.scala-lang.org/api/current/#scala.Predef\$
```

Pattern Matching (review)

- First: expressions:
 - The most basic aspect of functional programming is that it allows you to define expressions which yield some result.
 - You can split a complicated expression up using *val* or *def* and make things easier to understand (and usually shorter):
 - Let's say we want to know the sum of the integers 1 thru 20:

```
scala> 1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18+19+20
res0: Int = 210
scala> 20*(20+1)/2
res1: Int = 210
scala> val n = 20
n: Int = 20
scala> n*(n+1)/2
res2: Int = 210
scala> def sumOfNIntegers(n: Int) = n*(n+1)/2
sumOfNIntegers: (n: Int)Int
scala> sumOfNIntegers(20)
res3: Int = 210
```

- We've gone from the most specific form to the most general form. All give the same answer: 210. That's because all expressions are mathematically identical. The final answer is easiest to read, however. The first answer could easily be in error and we humans might not notice (a number skipped or repeated, for example).

Pattern Matching (2)

- There's another way we could have defined the sum...
 - Suppose we create a *case class* to represent a range of numbers starting with 1:

```
scala> case class Range(n: Int) { def sum = n*(n+1)/2 }
defined class Range
scala> val r = Range(20)
r: range = Range(20)
scala> r.sum
res4: Int = 210
```
 - We've talked about extractors and pattern matching before. Essentially, they are the opposite of the substitution principle that we use in expressions. Extractors are like a “what if?” scenario:

```
scala> r match { case Range(m) => println(m) }
20
```
 - **What if** we had a *Range* such that its *n* value was represented by a variable called *m*? We could print *m* to see what it was or do anything else with *m*.
 - It's important to understand that *m* is a “variable” (in the algebraic sense) but its value is fixed by the pattern-matching code to be whatever *n* was in the *Range*.

5.1

Actors and Akka

© 2015 Robin Hillyard



What is an Actor?

*All the world's a stage,
And all the men and women merely players;
They have their exits and their entrances,
And one man in his time plays many parts,
His acts being seven ages.*

Shakespeare, *As You Like It.*

- **Actor** is a mechanism that enables the separation of state from other state--and from a main (stateless) program--in both space and time;
- The concept was first introduced in Erlang;
- Actors form the foundation of the entire Akka system.

Why Actors?

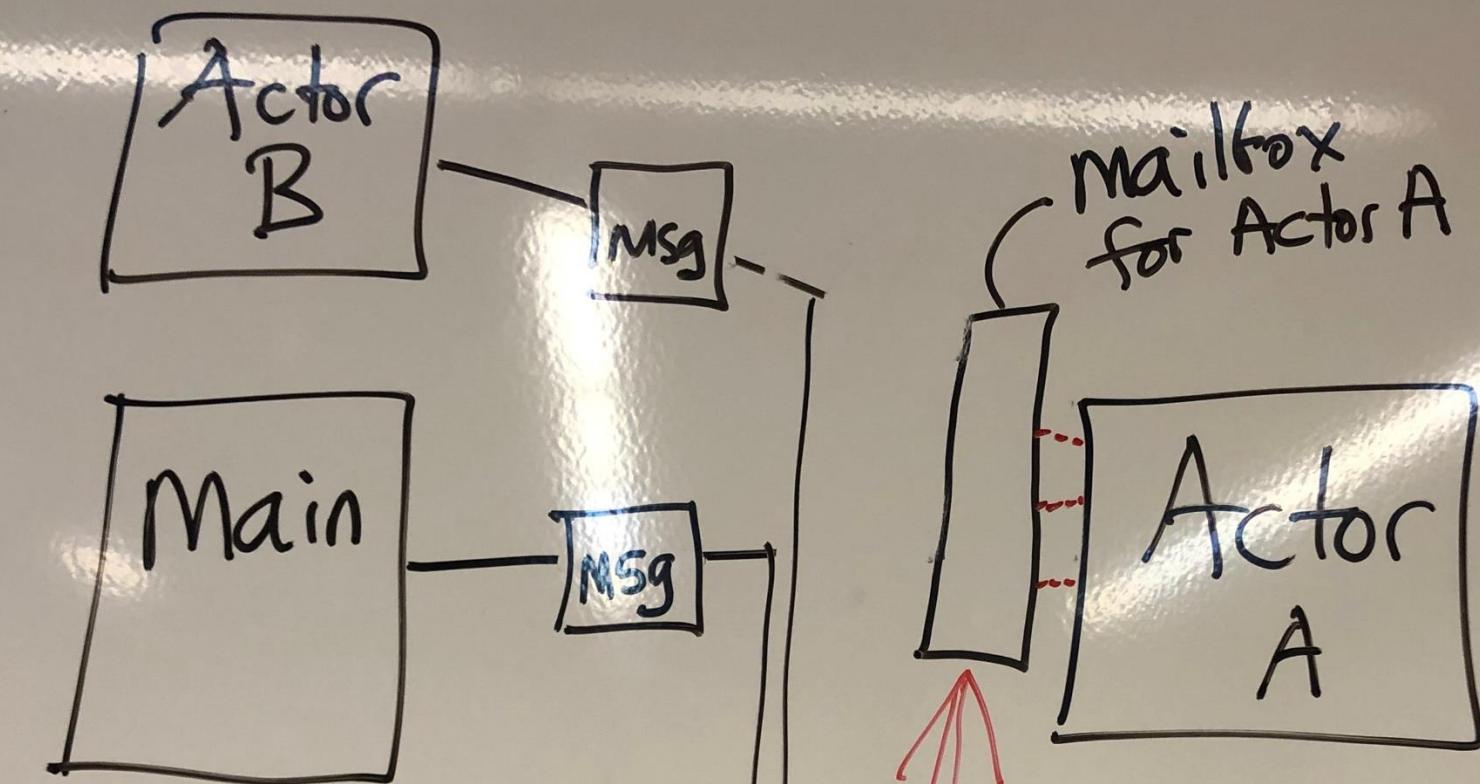
- We know that side-effects* (including I/O) and mutable state don't support referential transparency, i.e. we cannot substitute expressions for identifiers when side-effects/mutable state are involved. We cannot provably compose functions (e.g. **for-comprehensions**) when the functions are not R.T.
 - But just about all complex systems involve some side-effects and mutable state. So, how can we make these "rogue" components safe for Functional Programming?
 - We can create concepts such as I/O Monads and state-preserving concepts such as our RNG class. By doing this, we can isolate all the non-pure code into very clear chunks, leaving all the rest of the logic to be **composable** in our FP way. BTW, I/O Monads are not easy and you can't find them in the standard Scala library.
 - What about **mutable** state? Another technique is to isolate mutable state in a way that other parts of the system can never observe a mutation. For instance, we want to implement **quicksort** by creating an **Array** of the elements and mutating their positions. Then we copy the array back to the original form. As long as the rest of the system can never substitute that array, we should be safe.

* **side-effect** is basically any interaction with the world outside our program and, in particular, the thread it inhabits.

Why Actors? (2)

- And, there is one other practical consideration. When we are dealing with side-effects, those actions typically take enormous amounts of time compared with “normal” programming. We typically want to interact with the world outside asynchronously, i.e. using a *Future*.
- There is one further golden rule of functional programming: let each method (or function) do one thing and do it well. [Actually, this should be the rule for all programming, period.] Remember our mantra:

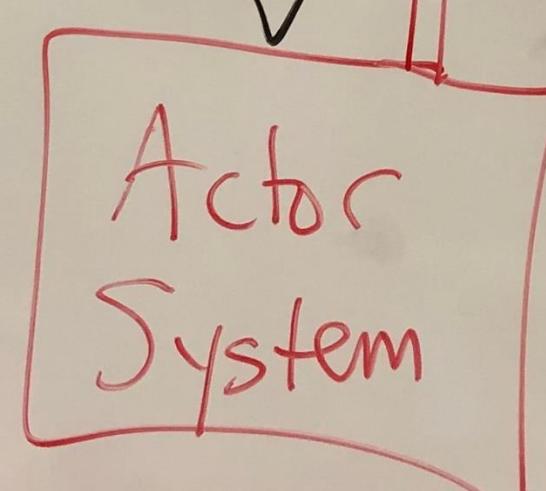
simple, obvious, elegant.
- *Actors* embody all the properties so far mentioned (although they are admittedly not the only way to do so). But actors can do a lot more...
- Actors were introduced to the computer programming world by *Erlang* (“Erlang concurrency model”).
- Scala has its own actors but they are now deprecated in favor of...



Actor B & main are in one address space

Actor A processes every message in a Q,
but one after the other in a same thread

Actor A is in another address space
compared to the one main's address space



Akka: "Classic" actors

- Akka actors are:
 - **Message receivers and senders:** [PubSub](#)
 - That is, they can communicate with the rest of the system only via messages—this helps reduce *coupling*;
 - Each actor has its own “mailbox” into which messages are deposited by the actors system.
 - **Encapsulating:**
 - In the sense we talked about—the only handle that we, as programmers, have to an actor (except when testing) is an *ActorRef* which does not give access to any actor internals such as mutable state. Therefore the rest of the system cannot observe any side-effects or mutability directly and can therefore follow normal FP composition rules and patterns.
 - **Thread-safe:**
 - In the sense that the entire processing of any one message is completed before any other message from the mailbox can be read or processed.
 - **Untyped***:
 - That's to say an actor has no intrinsic knowledge about the types of its properties—all actors are essentially the same in this regard. Information about typed objects is confined to the messages sent and received.
 - **Replicable:**
 - That's to say that the actor system can make copies of an actor to improve performance. These copies can be on remote systems of course.
 - **Resilient:**
 - That's to say that the actor system monitors the health of actors and will restart an actor if required, potentially bubbling up to the surface any exception.
 - **Lightweight:**
 - The boiler-plate overhead for an actor is only about 1k bytes so it's practical to have millions of them in an application.

* typed actors are here!.

Akka Classic (2)

- Overall, these properties make *Akka* the perfect system for *reactive programming*!
 - So, how do we get started? First go to <http://akka.io> for documentation, tutorials, patterns, etc.
 - Then add the following to your *build.sbt*

```
val akkaGroup = "com.typesafe.akka"  
val akkaVersion = "2.6.5"  
libraryDependencies ++= Seq(  
    akkaGroup %% "akka-actor" % akkaVersion,  
    akkaGroup %% "akka-testkit" % akkaVersion % "test",  
    akkaGroup %% "akka-slf4j" % akkaVersion,  
    "com.typesafe" %% "config" % "1.4.0",  
    "ch.qos.logback" %% "logback-classic" % "1.2.3" % "runtime"  
)
```
 - Let's take a look at an existing system: using Akka to solve Map-Reduce problems ([Majabigwaduce](#))

Akka Classic (3)

- An actor receives *typed* messages via its *receive* method*:

```
/*
 * The purpose of this mapper is to convert a sequence of objects into several sequences,
 * each of which is associated with a key.

*/
          Actor with logging
class Mapper[K1,V1,K2,W](f: (K1,V1)=>(K2,W)) extends MapReduceActor {

  override def receive = {
    case i: Incoming[K1,V1] =>
      log.info(s"received $i")
      val wk2ts = for ((k1,v1) <- i.m) yield Try(f(k1,v1))
      sender ! prepareReply(wk2ts)           ! means to the action here - prepareReply don't wait for the output
    case q =>
      super.receive(q)
  }
}

case class Incoming[K, V](m: Seq[(K,V)]) {
  override def toString = s"Incoming: with ${m.size} elements"
}
```

- Typically, the actor will prepare a response and send it either to the sender or another actor. A logging actor is able to log the messages as they arrive.

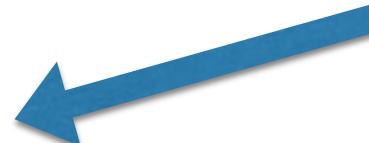
* This is taken from [Majabigwaduce](#)

Akka Classic (4)

```
import akka.actor.{ Actor, ActorLogging, ActorSystem, Props }
import scala.concurrent.duration._
import scala.concurrent._
import akka.util.Timeout
import akka.pattern.ask
import scala.util._

import ExecutionContext.Implicits.global
case object QuickSort extends Actor with ActorLogging {
  override def receive = {
    case input: Seq[Int] =>
      log.info(s"received $input")
      val array = input.toArray
      Sorting.quickSort(array)
      val response = array.toSeq
      sender ! response
    case _ => println("unknown message")
  }
}

object QuickSortApp extends App {
  implicit val timeout: Timeout = 10.seconds
  implicit val system = ActorSystem("QuickSort")
  val quickSort = system.actorOf(Props.create(classOf[QuickSort]), "sorter")
  val ints = args map _.toInt
  val f = quickSort ? ints.toSeq
  f.onComplete { x => x match {
    case Success(s) => println(s"received response of sorted sequence: $s")
    case Failure(e) => System.err.println(s"exception: $e")
  }
}
Await.ready(f, 10.second)
system.stop(quickSort)
system.shutdown
}
```



Quicksort is most efficient for large N when it is operating on a (mutable) array. Encapsulating array inside an *Actor* is *Referentially Transparent*. Quicksort is one of the best-known and most efficient sorting methods: it is generally $\mathbf{O}(N \log N)$ but in worst case is $\mathbf{O}(N^2)$.

Actor Ref is returned, it is an indirect reference to the actor

Akka Examples

- Our repository has only one place remaining that uses Classic actors:
 - MapReduce
- And two examples of typed actors, both in lab-actors:
 - Stocks
 - TicketAgency
- But there are better examples at:
 - <https://github.com/rchillyard/HedgeFund>
 - <https://github.com/rchillyard/Stocks>

Map-Reduce

- Map-reduce is a pattern for parallel processing:
 - it is based on the notion that you can divide a problem into N tasks:
 - provided that each of the tasks is truly independent;
 - where you have a “map” function which yields a “key” for each element (sub-division) of the problem;
 - and where you have a “reduce(k)” function which evaluates all elements with key “ k ” in parallel with (and independent of) all other elements;
 - once all of the N values are available, they are collected together and combined for the final report/result;
 - it is the pattern on which *Hadoop* and *Spark* are based;
 - it is very amenable to functional programming.

Map-Reduce (2)

- The n^{th} stage of the process looks like this:
 - $\text{Map}[K_{n-1}, V_{n-1}] \rightarrow \text{Map}[K_n, \text{Seq}[W_n]] \rightarrow \text{Map}[K_n, V_n]$
“mapper/groupBy” “reducer(s)”
 - But, typically, the input data to the first stage has no natural key so we use the fact that $\text{Map}[K_0, V_0]$ can be transformed directly to $\text{Seq}[(K_0, V_0)]$ where $\text{Seq}[(\emptyset, V_0)]$ in turn is the equivalent of $\text{Seq}[V_0]$. 
But we can't generally go in the reverse direction.
Why not?
- Assuming, then, that each (n^{th}) stage of the pipeline works as expected, then overall, we can transform:
 - $\text{Seq}[V_0] \rightarrow \text{Map}[K_n, V_n]$

Map-Reduce (3)

- For example: WebCrawler application...
 - $\text{Map}[\emptyset, \text{String}] \rightarrow \text{Map}[\text{URI}, \text{Seq}[\text{URI}]] \rightarrow \text{Map}[\text{URI}, \text{Seq}[\text{String}]]$
 $\rightarrow \text{Map}[\text{URI}, \text{Seq}[\text{String}]] \rightarrow \text{Map}[\text{Seq}[\text{String}], \text{Seq}[\text{String}]]$
 $\rightarrow \text{Seq}[\text{String}]$

CountWords app

```
/**  
 * CountWords: an example application of the MapReduce framework.  
 * This application is a three-stage map-reduce process (the final stage is a pure reduce process).  
 * Stage 1 takes a list of Strings representing URIs, converts to URLs, opens each as a stream, reading the contents and finally returns a map of URI->Seq[String]  
 * where the key is the URI of a server, and the Strings are the contents of each of the documents retrieved from that server.  
 * Stage 2 takes the map of URI->Seq[String] resulting from stage 1 and adds the lengths of the documents (in words) to each other. The final result is a map of  
 * URI->Int where the value is the total number of words read from the server represented by the key.  
 * Stage 3 then sums these values together to yield a grand total.  
 */  
case class CountWords(resourceFunc: String => Resource)(implicit system: ActorSystem, config: Config, timeout: Timeout, ec: ExecutionContext) extends (Seq[String] =>  
Future[Int]) {  
  override def apply(v1: Seq[String]): Future[Int] = {  
    def init = Seq[String]()  
  
    val stage1: MapReduce[String, URI, Seq[String]] = MapReduceFirstFold  
    { w: String => val u = resourceFunc(w); system.log.debug(s"stage1 map: $w"); (u.getServer, u.getContent) }, { (a: Seq[String], v: String) => a :+ v },  
    init _  
  }  
  val stage2: MapReduce[(URI, Seq[String]), URI, Int] = MapReducePipe  
  { (w: URI, gs: Seq[String]) => (w, for (g <- gs) yield g.split("""\s+""").length) reduce (_ + _) }, { (x: Int, y: Int) => x + y },  
  1  
}  
  val stage3 = Reduce[Int, Int](_ + _)  
  val countWords = stage1 | stage2 | stage3  
  countWords.apply(v1)  
}
```

CountWords app

```
object CountWords {  
    def apply(hc: HttpClient, args: Array[String]): Future[Int] = {  
        val configRoot = ConfigFactory.load  
        implicit val config: Config = configRoot.getConfig("CountWords")  
        implicit val system: ActorSystem = ActorSystem(config.getString("name"))  
        implicit val timeout: Timeout = getTimeout(config.getString("timeout"))  
        import ExecutionContext.Implicits.global  
  
        val ws = if (args.length > 0) args.toSeq else Seq("http://www.bbc.com/doc1", "http://www.cnn.com/doc2", "http://default/doc3",  
"http://www.bbc.com/doc2", "http://www.bbc.com/doc3")  
        CountWords(hc.getResource).apply(ws)  
    }  
  
    // TODO try to combine this with the same method in MapReduceActor  
    def getTimeout(t: String): Timeout = {  
        val durationR = """(\d+)\s*(\w+)""".r  
        t match {  
            case durationR(n, s) => new Timeout(FiniteDuration(n.toLong, s))  
            case _ => Timeout(10 seconds)  
        }  
    }  
}
```

CountWords unit test

```
class CountWordsSpec extends FlatSpec with Matchers with Futures with ScalaFutures with Inside with MockFactory {
  "CountWords" should "work for http://www.bbc.com/ http://www.cnn.com/ http://default/" in {
    val wBBC = "http://www.bbc.com/"
    val wCNN = "http://www.cnn.com/"
    val wDef = "http://default/"
    val uBBC = new URI(wBBC)
    val uCNN = new URI(wCNN)
    val uDef = new URI(wDef)
    val hc = mock[HttpClient]
    val rBBC = mock[Resource]
    (rBBC.getServer _).expects().returning(uBBC)
    rBBC.getContent _ expects() returning CountWordsSpec.bbcText
    val rCNN = mock[Resource]
    rCNN.getServer _ expects() returning uCNN
    rCNN.getContent _ expects() returning CountWordsSpec.cnnText
    val rDef = mock[Resource]
    rDef.getServer _ expects() returning uDef
    rDef.getContent _ expects() returning CountWordsSpec.defaultText
    hc.getResource _ expects wBBC returning rBBC
    hc.getResource _ expects wCNN returning rCNN
    hc.getResource _ expects wDef returning rDef
    val nf = CountWords(hc, Array(wBBC, wCNN, wDef))
    whenReady(nf, timeout(Span(6, Seconds))) {
      case i => assert(i == 556)
    }
  }
}
```

Akka: typed actors

- Akka typed actors are:
 - **Message receivers and senders:**
 - As before.
 - **Encapsulating:**
 - As before.
 - **Thread-safe:**
 - As before.
 - **Typed:**
 - Typed actors have a very different API. See next slide.
 - **Replicable:**
 - As before.
 - **Resilient:**
 - As before.
 - **Lightweight:**
 - As before.

Akka: Typed Actors (2)

- Essentials of the API:
 - Actors are created through factory methods (unlike untyped actors which were constructed);
 - An actor has (typed) *behavior*;
 - The actor's receive method returns a (potentially different) behavior after doing any of the following:
 - creating and messaging other actors;
 - changing internal state

Exercise:

- Go to <https://developer.lightbend.com/start/?group=akka&project=akka-quickstart-scala>
 - Click CREATE A PROJECT FOR ME
 - Make a New project in your IDE using the existing source directory that you just downloaded;
 - When offered a choice of project type, use SBT (or BSP);
 - You will probably have to specify a JDK (use whatever you normally use);
 - Run *AkkaQuickStart* (e.g. right-click on the source file in the Project window and select Run).
 - You've just created and used typed actors!

```

package com.example
import akka.actor.typed._
import akka.actor.typed.scaladsl.Behaviors
import com.example.GreeterMain.SayHello
object Greeter {
  final case class Greet(whom: String, replyTo: ActorRef[Greeted])
  final case class Greeted(whom: String, from: ActorRef[Greet])
  def apply(): Behavior[Greet] = Behaviors.receive { (context, message) =>
    context.log.info("Hello {}!", message.whom)
    message.replyTo ! Greeted(message.whom, context.self)
    Behaviors.same
  }
}
object GreeterBot {
  def apply(max: Int): Behavior[Greeter.Greeted] = { bot(0, max) }
  private def bot(greetingCounter: Int, max: Int): Behavior[Greeter.Greeted] = Behaviors.receive { (context, message) =>
    val n = greetingCounter + 1
    context.log.info("Greeting {} for {}", n, message.whom)
    if (n == max) Behaviors.stopped
    else { message.from ! Greeter.Greet(message.whom, context.self); bot(n, max) }
  }
}
object GreeterMain {
  final case class SayHello(name: String)
  def apply(): Behavior[SayHello] = Behaviors.setup { context =>
    val greeter = context.spawn(Greeter(), "greeter")
    Behaviors.receiveMessage { message =>
      val replyTo = context.spawn(GreeterBot(max = 3), message.name)
      greeter ! Greeter.Greet(message.name, replyTo)
      Behaviors.same
    }
  }
}
object AkkaQuickstart extends App {
  val greeterMain: ActorSystem[GreeterMain.SayHello] = ActorSystem(GreeterMain(), "AkkaQuickStart")
  greeterMain ! SayHello("Charles")
}

```

Akka

The industry's leading cloud-native frameworks and runtimes.



Full suite of reactive microservices frameworks and runtimes
for building cloud-native applications



The Power of Akka optimized for the cloud



Akka Streams, Spark, Flink and everything you need to rapidly
build and operate streaming data applications on Kubernetes



Next generation, stateful serverless computing,
powered by Akka



<https://www.lightbend.com>

Examples

- See the repository <https://github.com/rchillyard/TicketAgency> for an example of using typed actors with the ticket agency that we talked about before.
- Rock the JVM:
 - [Stateful vs. Stateless \(intro to typed actors\)](#)
 - [3 Reasons why...](#)
 - [The Pipe pattern \(a useful pattern for working with typed actors\)](#)
 - [Adapt me... \(an important principle if you want to do anything useful with Akka actors\)](#)
 - [Akka, Cats and Cassandra in a project](#)

Updated: 2022-11-08

5.2

Spark Introduction

© 2015-22 Robin Hillyard



Northeastern
University

What is Spark?

- Apache *Spark* is a fast, general-purpose, cluster-computing platform. Or:
- **Unified engine for large-scale data analytics**
 - It abstracts out the map/reduce model* in such a way that a programmer or shell user is simply unaware of it. [This is similar to the way actors abstract out the threading model of Java]
 - It does this in a way that takes more advantage of memory (as opposed to persistent storage). It's also more natural.
 - *Spark* is written in **Scala** and therefore takes full advantage of the functional programming paradigm: providing better performance than competing solutions.
 - *Spark* does not require *Hadoop*, although it is very compatible and is happy to use **YARN** and **HDFS** if they're available.

* typified by *Hadoop*

What is Spark? (2)

- **Spark** provides API bindings for *Scala*, *Python*, *R*, *SQL* and *Java*. So, you don't have to use *Scala*. But by now, surely you'd want to, right? I definitely wouldn't recommend the *Java* API because it can be quite messy to get things right for *Java*.
- Spark documentation can be found here:
<https://spark.apache.org/docs/latest/>. The current latest version is 3.3.1
- Download from here: <https://spark.apache.org/downloads.html>
- **Spark 3.3.1 runs with Java 8/11* and Scala 2.12 or 2.13** :)**
- *Spark* provides a platform (“stack” if you prefer) which includes several components:

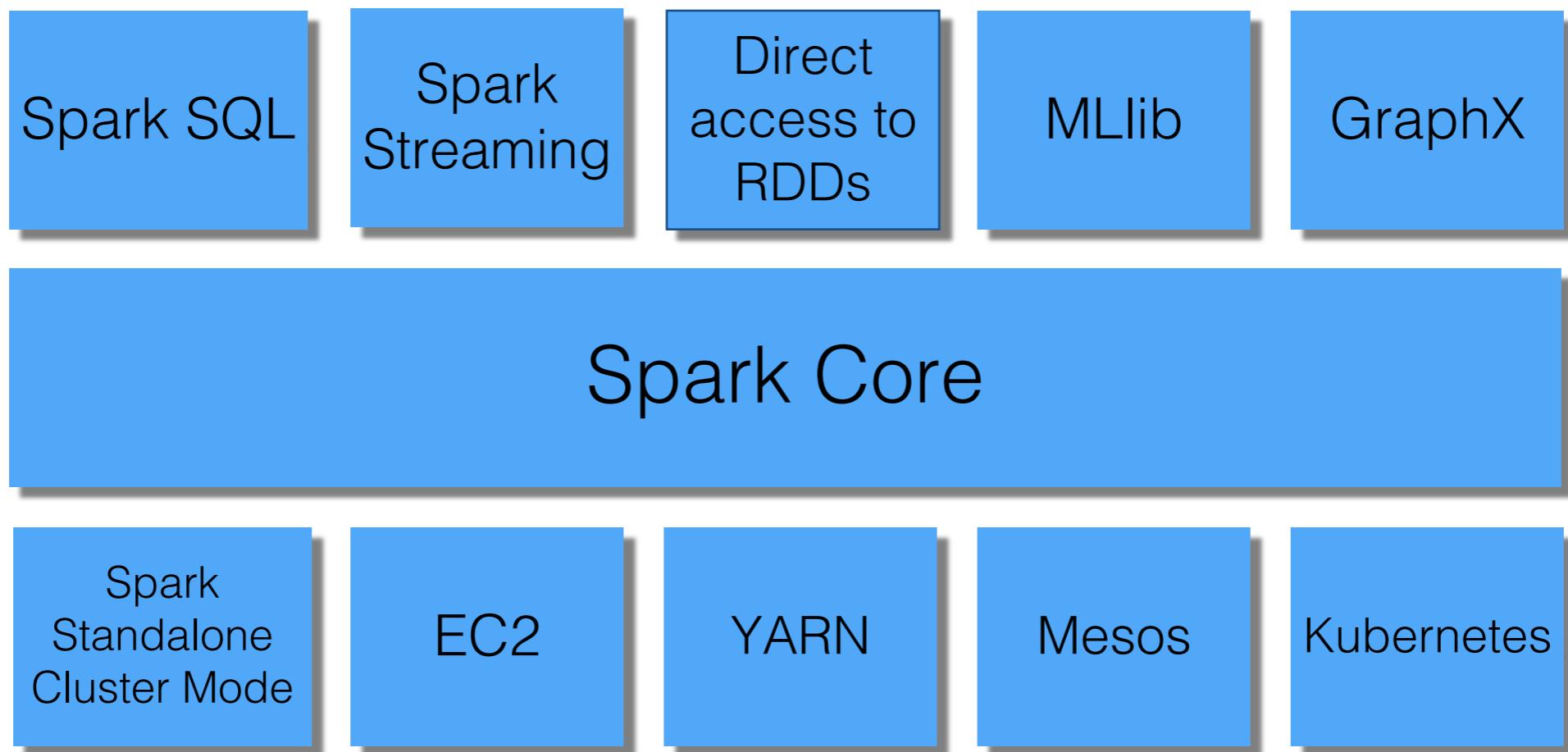
SAM was introduced in scala 2.10

There were some issues with spark,
since spark had not updated its code

* Version 8u92 and onwards;

** make sure you download the 2.13 version

Spark Platform



Books: [Learning Spark, Karau et al, \(O'Reilly\)](#);
Spark in Action (Manning).

Spark components

- You can work directly with Spark's RDDs **but there's really no need to...**
- **Spark-SQL**
 - Essentially a memory-intensive implementation of HIVE:
 - Or you can think of Spark-SQL as an alternative implementation language to *Scala*, *Python*, *Java*, *R*.
 - Spark-SQL in 2.0 and later is highly optimized. It may well be that you should always use Spark-SQL unless you have a good reason not to.
- **Spark Streaming**
 - This is how you would deal with a stream of events, e.g. messaging or processing log files from a web system, database, or message broker like Kafka
- **MLlib**
 - Spark's ML library, including classification, regression, clustering, etc.
- **GraphX**
 - Spark's graph-database (Pregel*-oriented)

* Name of river that flowed in Euler's time through Königsberg

RDDs

(you should still understand these).

- The *Spark API* is founded on a simple, elegant, obvious data type (everything is an RDD):

- **Resilient Distributed Dataset (RDD)**

```
abstract class RDD[T] extends Serializable with Logging
```

- Guess what? It's a monad! So it supports *map*, *flatMap*, *filter*, etc. [it's a container similar to *Future*, *Par*, *Option*, *Try* in that it doesn't extend *FilterMonadic*]

[Stream in Java i.e. LazyList in Scala](#)

- Like *Stream*, *RDD* is lazy.

Actually, there is no *README.md* file available but *Spark* doesn't worry about that yet because it's lazy!

- As far as the user is concerned, an *RDD* is just a collection that you can do stuff with. The fact that it may potentially be distributed across thousands of nodes isn't of any immediate concern to you.

```
scala> val lines = sc.textFile("README.md")
lines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at textFile at <console>:21
scala> val list = List(1,2,3)
list: List[Int] = List(1, 2, 3)
scala> sc.parallelize(list)
res0: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:24
```

Working with RDDs

- You already know how to do this!
 - A couple of details:

```
scala> res0 map (_ .toString )
res1: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at map at <console>:26
scala> println(res1)
MapPartitionsRDD[3] at map at <console>:26
scala> res1 foreach (println)
3
2
1
```

- Hmm, that's a bit of a surprise! Not really. The method `sc.parallelize` by default splits into two slices which are recombined for the final `println` step. Specifying one slice, `sc.parallelize(list, 1)`, maintains the order. But most of the time, you don't care!
- If we want to use an *RDD* again, you can persist it:

```
scala> res1.persist
res2: res1.type = MapPartitionsRDD[3] at map at <console>:26
```

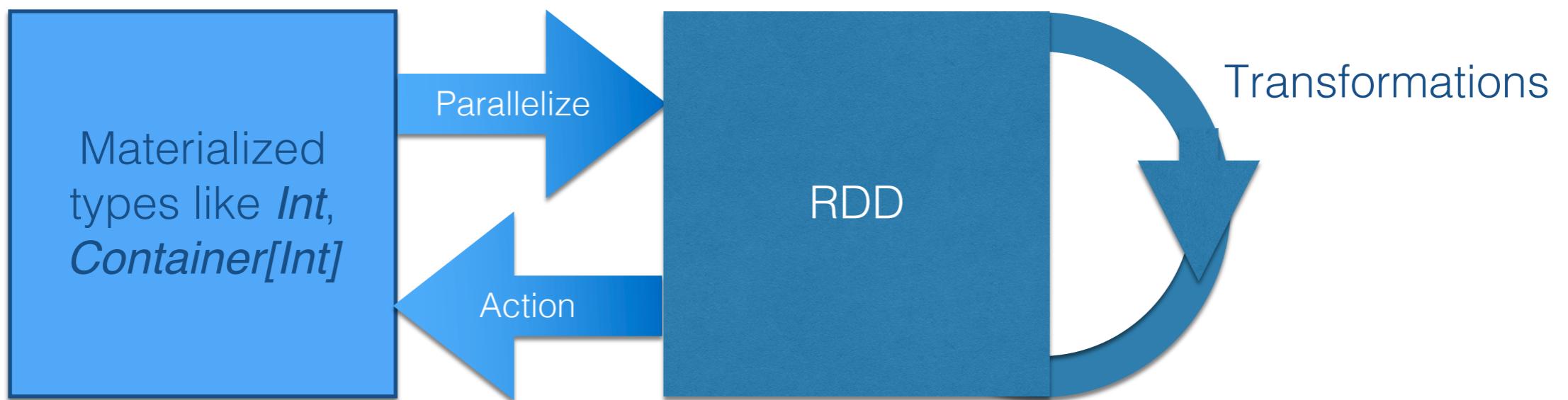
- There is also a signature of `persist` which allows you to specify the storage level, e.g. `res1.persist(DISK_ONLY)`
- Note that the `persist` call itself doesn't force evaluation.
- If you just want to get back a collection from an *RDD*, invoke `collect` (without a function parameter):

```
scala> res0.collect
res3: Array[Int] = Array(1, 2, 3)
```

We just pass in a function to do stuff as you would expect. Note, however, that said function must be *serializable* since it has to be passed to remote worker nodes. Don't pass in instance methods.

Working with RDDs (2)

- Because $\text{RDD}[\text{T}]$ is lazy, it's essentially *opaque*



- Once we have parallelized a container as an RDD, we can apply transformations to RDDs, creating new RDDs. Since these of course are lazy, the RDDs are efficiently decorated. In order to materialize something from an RDD, we need to apply an “action”.

RDD methods

- What can you do with *RDDs*?
 - Actions
 - Extractions:
 - *collect* (not the same as the *Collection* method we've met before—which takes a partial function—think of this if you like as the opposite of *sc.parallelize*)
 - *foreach, saveAsTextFile, saveAsObjectFile,*
 - Aggregations—measure an *RDD* perhaps of an appropriate type:
 - *count, aggregate, max, min, reduce, treeReduce, fold*
 - Transformations
 - single *RDDs*:
 - *map, flatMap, filter, distinct, sample, take, drop, collect(f),*
 - single *RDDs* of an appropriate type:
 - *top, takeOrdered, takeSample, countByValue, groupBy, sortBy*
 - Combinations—two *RDDs* (of same underlying type):
 - *union, intersection, subtract, cartesian, zip*

Pair RDDs

- What are *Pair RDDs*?
 - It shouldn't come as a big surprise that building RDDs of key/value pairs works well with the map/reduce paradigm. That's how map/reduce works.
 - Furthermore, you can explicitly control partitioning on such pair RDDs which can give you the location-based performance boost that is a key feature of Hadoop.

```
scala> val lines = sc.textFile("flatland.txt")
lines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[18] at textFile at <console>:21
scala> val pairs = lines.map(x => (x.split(" ")(0), x))
pairs: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[19] at map at <console>:23
scala> pairs.groupByKey
res24: org.apache.spark.rdd.RDD[(String, Iterable[String])] = ShuffledRDD[20] at groupByKey at <console>:26
scala> res24.collect
res25: Array[(String, Iterable[String])] = Array((is,CompactBuffer(is impossible that there should be anything of what)), (luminous,CompactBuffer(luminous edges-and you will then have a pretty)), (readers,,CompactBuffer(readers, who are privileged to live in Space.)), (their,CompactBuffer(their places, move freely about, on or in the surface,)), (last,CompactBuffer(last when you have placed your eye exactly on the)), (one,CompactBuffer(one figure from another. Nothing was visible, nor could be visible,)), (as,CompactBuffer(as I have described them. On the contrary, we could)), ("",CompactBuffer(, , , )), (correct,CompactBuffer(correct notion of my country and countrymen. Alas,)), (becoming,CompactBuffer(becoming more and more oval to your view, and at)), (Imagine,CompactBuffer(Imagine a...))
```

- Here are the methods that operate on pair RDDs:
 - reduceByKey, groupByKey, combineByKey(...), mapValues(f), flatMapValues(f), keys, values, sortByKey
 - (on two pair RDDs) subtractByKey, join, rightOuterJoin, leftOuterJoin, cogroup
- Incidentally, you will find that the methods on GraphX are very similar!

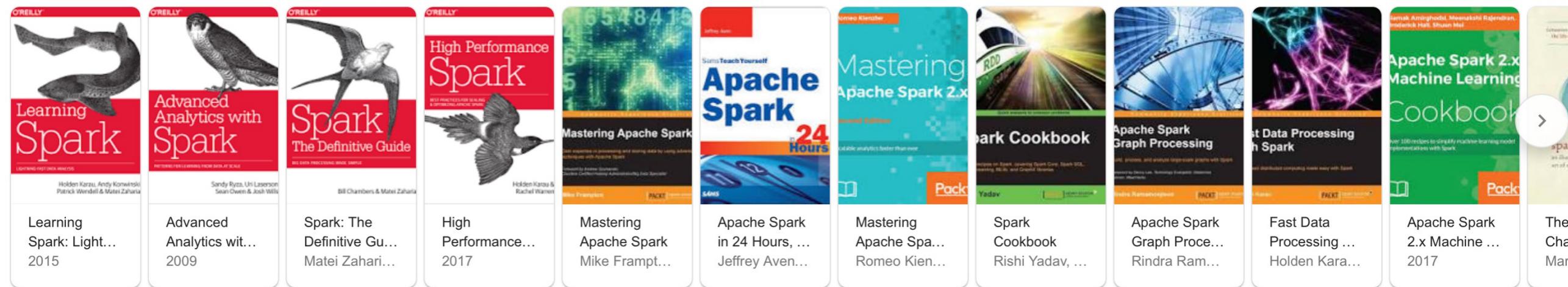
Word count by Spark

- Do you remember the code we looked at to do word count by map/reduce?
- Look how complicated it is in Spark (not!):

```
scala> val words = lines.flatMap(x => x.split(" "))  
words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[21] at flatMap at <console>:23  
scala> val result = words.map(x => (x,1)).reduceByKey((x, y) => x + y)  
result: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[23] at reduceByKey at  
<console>:25  
scala> result.collect  
res26: Array[(String, Int)] = Array((table,1), (circle.,1), (call,3), (paper,1), (country,1),  
(is,1), (penny,3), (its,1), (Flatland,,2), (fixed,1), (now,,1), (oval,2), (have,6), (upon,1),  
(this,1), (countrymen.,1), (one,2), (mind,1), (with,1), (live,1), (we,3), (straight,2),  
(been,1), (dare,1), (us,,1), (who,1), (correct,1), (places,,1), (over,1), (without,1), (my,4),  
(rising,1), (exactly,1), (so,,1), (make,1), (instead,1), (what,1), (years,1), (becoming,1),  
(are,,1), (other,2), (from,1), (now,1), (has,1), (table,,1), (leaning,1), (happy,1), (vast,1),  
(world,1), (contrary,,1), (drawing,1), (demonstrate.,1), (are,1), (kind,,1), (few,1),  
(luminous,1), (readers,,1), (because,1), (can,1), (their,1), (moving,1), (country,,1), (down,1),  
(anything,1), (remaining,1), (last,1), (will,8), (our,1)...
```

Books

Books / Apache Spark



Updated: 2022-03-28

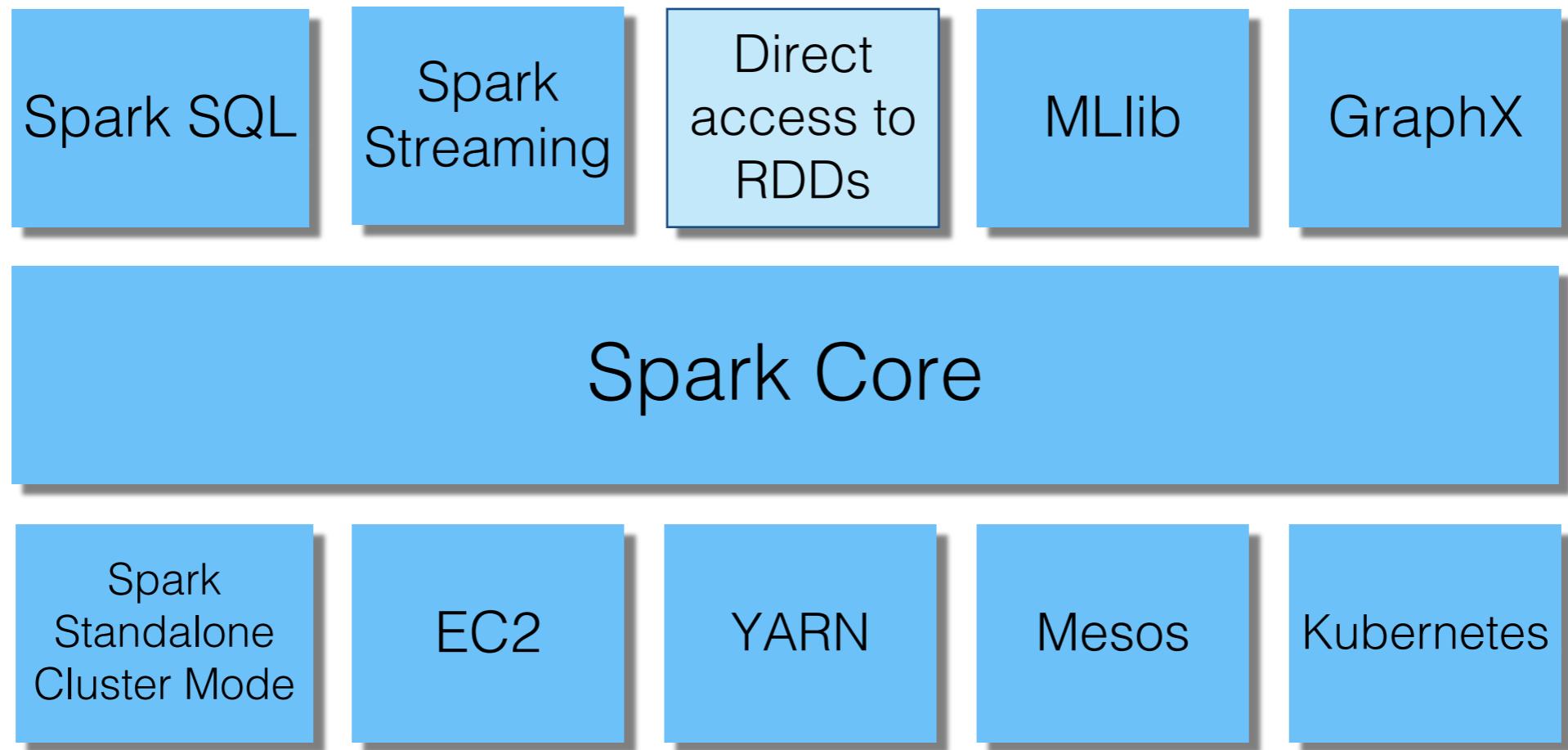
5.3

Spark Continued

© 2015-22 Robin Hillyard



Spark Platform



Books: [Learning Spark, Karau et al. \(O'Reilly\)](#);
Spark in Action (Manning).

Under the hood

- As mentioned previously, Spark is built in Scala
 - You could easily write your own version of Spark
 - (in fact, that's kind of what *Majabigwaduce* is)
 - Basically, Spark is:
 - A “resilient distributed dataset” container (RDD);
 - A DAG (directed-acyclic graph) generator;
 - An interface to a resource manager (Yarn, MESOS, etc.);
 - A higher-level API for working with SQL;
 - A few other bits and pieces.
 - But getting the details right would take a lot of work, obviously — but the point is that Spark is just Scala set up to make parallel processing (map/reduce) easy

Example of using RDD (SparkContext)

```
package edu.neu.csye._7200

import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}

object WordCount extends App {

    def wordCount(lines: RDD[String], separator: String) = {
        lines.flatMap(_.split(separator))
            .map((_, 1))
            .reduceByKey(_ + _)
    }

    //For Spark 1.0-1.9
    val sc = new SparkContext(new SparkConf().setAppName("WordCount").setMaster("local[*]"))

    wordCount(sc.textFile("input/WordCount.txt"), " ").foreach(println(_))

    sc.stop()
}
```

How to invoke Spark?

- There are lots of ways:
 - spark-shell (like we did last week)
 - spark-submit –jar
 - Docker...
 - Databricks notebook
 - Zeppelin
 - AWS, etc. know how to run spark.

Example of using RDD from Dataset/Dataframe

(using SparkSession: spark-sql)

```
package edu.neu.csye._7200
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.SparkSession
object WordCount extends App {
  def wordCount(lines: RDD[String], separator: String) = {
    lines.flatMap(_.split(separator))
      .map((_, 1))
      .reduceByKey(_ + _)
  }
  val spark = SparkSession
    .builder()
    .appName("WordCount")
    .master("local[*]")
    .getOrCreate()
  wordCount(spark.read.textFile("input/WordCount.txt").rdd, " ")
    .collect().foreach(println(_))
  spark.stop()
}
```

How does RDD work?

- As always if you want to answer a question like this: go to the [source](#)!

```
// Transformations (return a new RDD)

/**
 * Return a new RDD by applying a function to all elements of this RDD.
 */
class tag is an implicit, that is provided
def map[U: ClassTag](f: T => U): RDD[U] = withScope {
    val cleanF = sc.clean(f)  clean makes sure it works i.e by serialization
    new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.map(cleanF))
}
```



withScope ensures that this *RDD* stays within the same hierarchy; *ClassTag* give us information about the class *U* at runtime; *sc.clean* ensures that the function *f* is serializable, etc.

- The point is that *map* doesn't really “do” anything: it simply creates a new *RDD* with function *f* and a reference to *this*.

Persistence

- Basically:
 - since everything is done in memory, an RDD will be garbage-collected when there are no RDDs referencing it (that's to say until you create an *action* which corresponds to a *task*).
 - If you want to avoid this: and keep an RDD around for longer, you can use *cache* or *persist*. (*cache* is just a form of *persist* but memory only—*persist* allows some or all to be save to disk).

Broadcasting

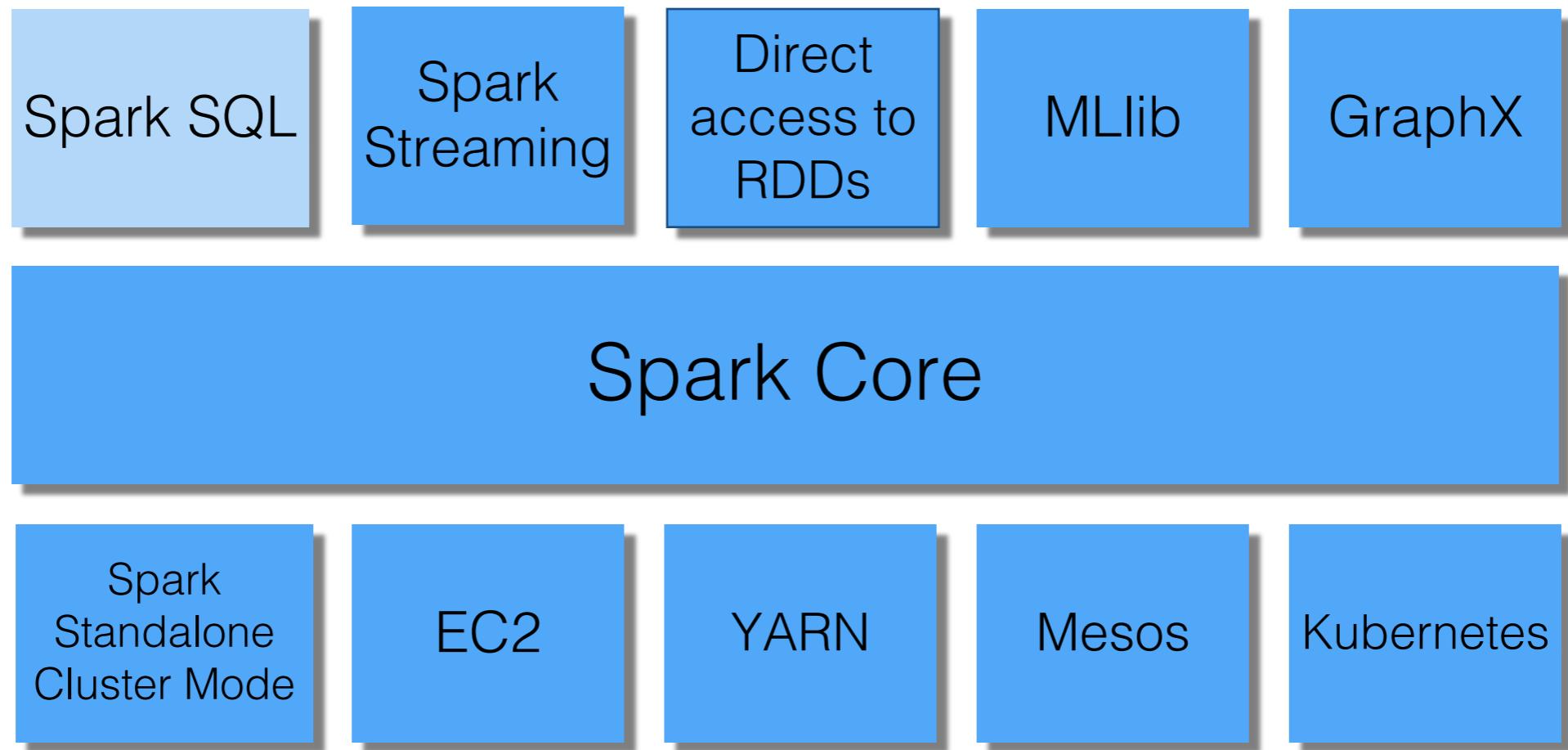
- Suppose that you have a lookup table (or something similar) that will need to be used by each of the executors?
 - The table will have to be sent over the network for every task to be run on each executor.
 - If you know that will happen ahead of time, you can broadcast the table so that it only has to be sent to each executor once.
 - `val xb = sc.broadcast(x)`
 - Now, in our code, we refer to `xb.value` instead of `x`

Accumulators

- Information flows from the driver to the executors mostly in only one direction (other than the result of running a Spark task).
 - But suppose we want to keep count of the number of operations that happen on the executors, or the number of ***None*** values, ***Failure*** values, whatever...
 - It would be awkward to include this information in the return type, and that wouldn't really work if the executor threw an exception and failed.
 - The answer is to set up an accumulator: these are write-only objects that you set up in the driver and which are updated by the executors.

Spark Modules

Spark Platform



Books: [Learning Spark, Karau et al. \(O'Reilly\)](#);
Spark in Action (Manning).

SparkSQL

- What exactly is SparkSQL and why would you want to use it?
 - At first, SparkSQL was fairly primitive and it was better to use RDDs (or Hive).
 - But now (especially in Spark 2.0), SparkSQL has a very good optimizer which will create an execution plan for Spark which is potentially very efficient
 - Spark 2.x (and 1.6.3?) even allows you extend the optimizer with your own rules and node types.
 - Consequently, more and more Spark work is being done not, in Scala, not in Python, Java or R: but in SQL

Datasets/Dataframes

- An alternative to using SQL is to set up a *Dataset* (or *Dataframe* in 1.6.1) and treat it similarly to an *RDD* (i.e. with Scala)
 - A *Dataframe* is untyped (basically a collection of tuples) but a *Dataset* has a type:
 - In 2.0 (+), type *Dataframe* = *Dataset[Row]*
 - *Dataframe/Dataset* do not extend *RDD*. But you can get the underlying *RDD* with the *rdd* method.

Dataset - has type safety

Spark SQL

- You can run SQL in several ways:
 - get a *spark* and make **SQL** queries;
 - get a *spark* and use the **DataFrame** or **Dataset** [API](#):
 - *DataFrames* provide a DSL for structured data manipulation.

```
scala> val df = sqlContext.read.json("examples/src/main/resources/people.json")
df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]
scala> df.show
+---+---+
| age| name|
+---+---+
| null| Michael|
| 30 | Andy |
| 19 | Justin|
+---+---+
scala> df.printSchema
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
scala> df.select("name").show
+---+
| name|
+---+
| Michael|
| Andy |
| Justin|
+---+
```

Joining

- SparkSQL supports various types of Join:
 - INNER, LEFT, OUTER, etc. etc.
 - There will be times when you can improve performance by using a *broadcast* join (aka “map” join) — same idea as broadcast variable.
 - However, Spark will do the broadcast for you if it knows the sizes of the tables in advance (e.g. you use a persistence format such as ORC).

Reading CSV as DataFrame

- The standard way to read a CSV file as a *DataFrame* is*:

```
object Diamonds extends App {  
    val spark: SparkSession = SparkSession  
        .builder()  
        .appName("WordCount")  
        .master("local[*]")  
        .getOrCreate()  
  
    val diamonds = spark.read.format("csv")  
        .option("header", "true")  
        .option("inferSchema", "true")  
        .load("assignment-spark-  
wordcount/diamonds.csv")  
  
    diamonds.printSchema()  
    diamonds.show()  
}
```

* see: <https://docs.databricks.com/data/data-sources/read-csv.html>

Diamonds

```
root
|-- _c0: integer (nullable = true)
|-- carat: double (nullable = true)
|-- cut: string (nullable = true)
|-- color: string (nullable = true)
|-- clarity: string (nullable = true)
|-- depth: double (nullable = true)
|-- table: double (nullable = true)
|-- price: integer (nullable = true)
|-- x: double (nullable = true)
|-- y: double (nullable = true)
|-- z: double (nullable = true)

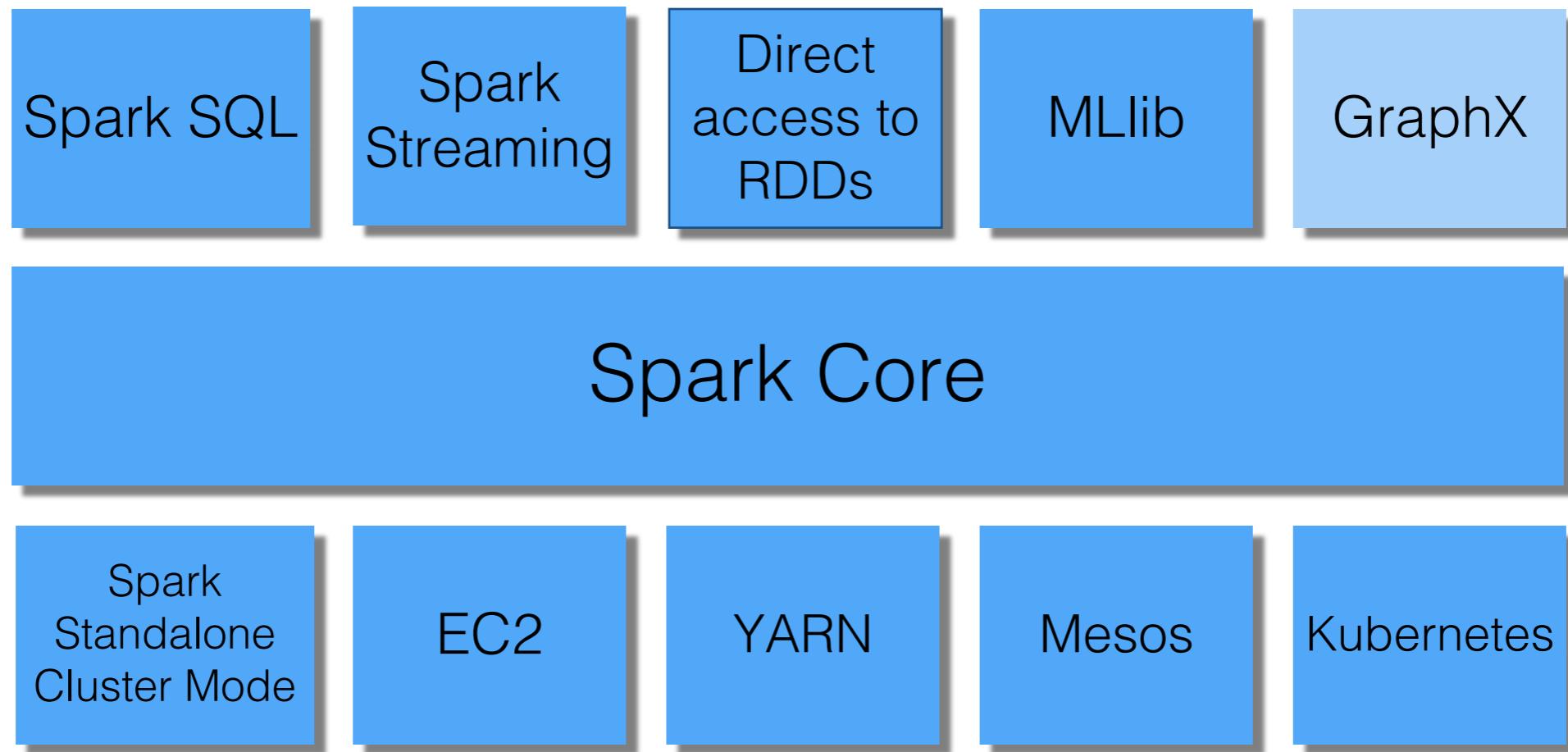
+---+-----+-----+-----+-----+-----+-----+
|_c0|carat|      cut|color|clarity|depth|table|price|   x|   y|   z|
+---+-----+-----+-----+-----+-----+-----+
 1 | 0.23| Ideal| E| SI2| 61.5| 55.0| 326|3.95|3.98|2.43|
 2 | 0.21| Premium| E| SI1| 59.8| 61.0| 326|3.89|3.84|2.31|
 3 | 0.23| Good| E| VS1| 56.9| 65.0| 327|4.05|4.07|2.31|
 4 | 0.29| Premium| I| VS2| 62.4| 58.0| 334|4.2|4.23|2.63|
 5 | 0.31| Good| J| SI2| 63.3| 58.0| 335|4.34|4.35|2.75|
 6 | 0.24| Very Good| J| VVS2| 62.8| 57.0| 336|3.94|3.96|2.48|
 7 | 0.24| Very Good| I| VVS1| 62.3| 57.0| 336|3.95|3.98|2.47|
 8 | 0.26| Very Good| H| SI1| 61.9| 55.0| 337|4.07|4.11|2.53|
 9 | 0.22| Fair| E| VS2| 65.1| 61.0| 337|3.87|3.78|2.49|
10 | 0.23| Very Good| H| VS1| 59.4| 61.0| 338|4.0|4.05|2.39|
11 | 0.3| Good| J| SI1| 64.0| 55.0| 339|4.25|4.28|2.73|
12 | 0.23| Ideal| J| VS1| 62.8| 56.0| 340|3.93|3.9|2.46|
13 | 0.22| Premium| F| SI1| 60.4| 61.0| 342|3.88|3.84|2.33|
14 | 0.31| Ideal| J| SI2| 62.2| 54.0| 344|4.35|4.37|2.71|
15 | 0.2| Premium| E| SI2| 60.2| 62.0| 345|3.79|3.75|2.27|
16 | 0.32| Premium| E| I1| 60.9| 58.0| 345|4.38|4.42|2.68|
17 | 0.3| Ideal| I| SI2| 62.0| 54.0| 348|4.31|4.34|2.68|
18 | 0.3| Good| J| SI1| 63.4| 54.0| 351|4.23|4.29|2.7|
19 | 0.3| Good| J| SI1| 63.8| 56.0| 351|4.23|4.26|2.71|
20 | 0.3| Very Good| J| SI1| 62.7| 59.0| 351|4.21|4.27|2.66|
+---+-----+-----+-----+-----+-----+-----+
only showing top 20 rows
```

Reading CSV as DataSet

- There is no standard way to read a CSV file as a *DataSet*.
- You could use my *TableParser* library:
 - <https://index.scala-lang.org/rchillyard/tableparser/tableparser/1.1.1?binaryVersion=2.13>
 - <https://github.com/rchillyard/TableParser>
 - <https://scalaprof.blogspot.com/2019/04/new-projects.html>
- This is the "proper" way to deal with a CSV file in Spark.

```
import MovieParser._  
val mty: Try[Table[Movie]] = Table.parse("movies.csv")  
val dy: Try[Seq[Movie]] = mty map (spark.createDataset(_.toSeq))
```

Spark Platform



Books: [Learning Spark, Karau et al, \(O'Reilly\)](#);
Spark in Action (Manning).

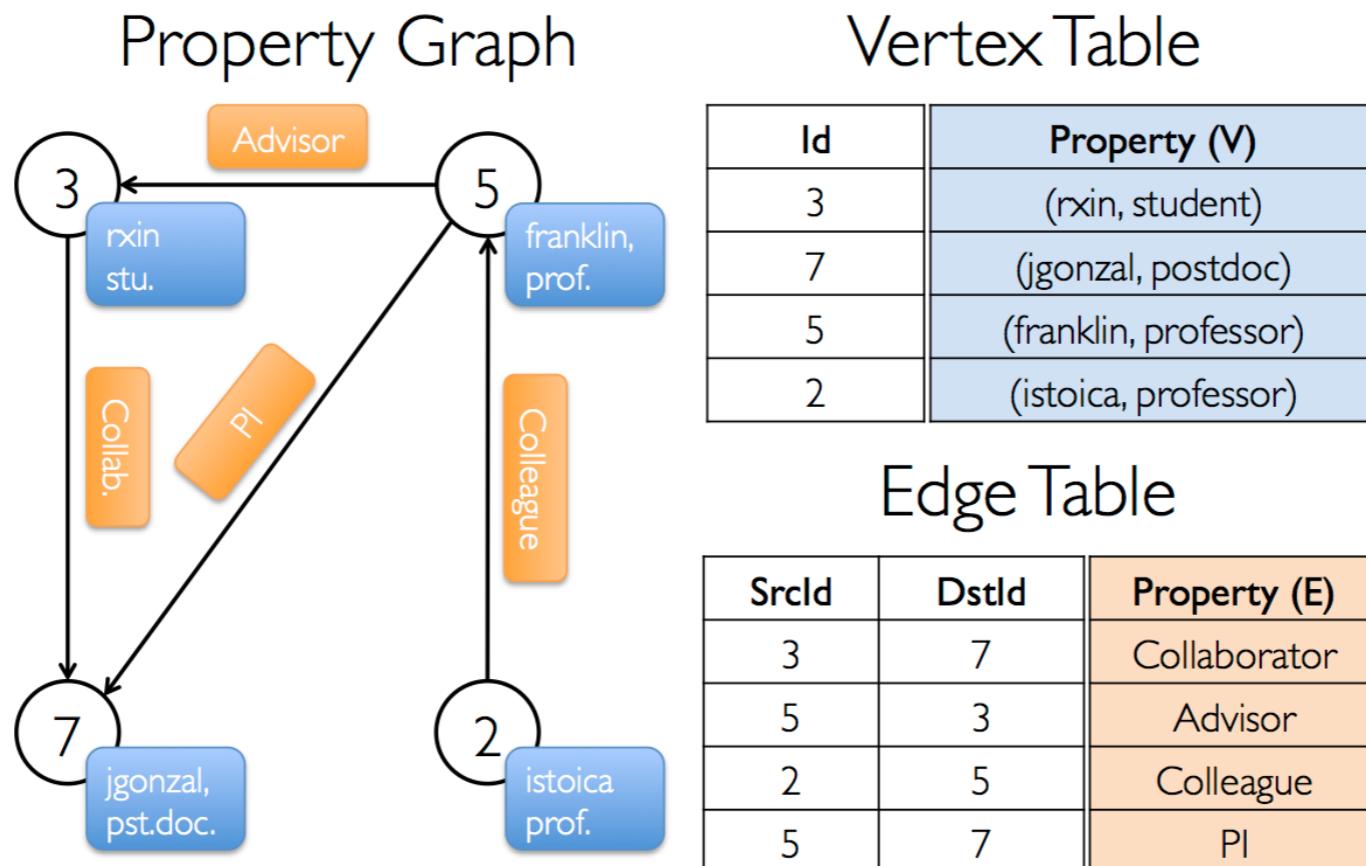
Why is GraphX interesting?

- *GraphX* is interesting because...
 - Graphs are interesting on their own.
 - Graphs represent *relationships*—and relationships are an important type of information.
 - Over the years, we have been seduced into thinking that tables are the most important way of modeling data (relational databases)
 - Actually, before relational databases, we had so-called codasyl databases which could represent graphs.
 - Graphs can store data whose structure is totally arbitrary and dynamic: trees, sparse matrices, key-value stores, tables, etc. (you might not always *want* to use a graph of course, but you could)
 - *GraphX* extends the *Spark* infrastructure (based on linear, but segmentable datasets—*RDDs*) and patterns to graph information.

Some GraphX resources

- [GraphX Programming Guide](#)
- [GraphX: Graph Analytics in Spark- Ankur Dave \(UC Berkeley\)](#)

An example (from programming guide)



An example

```
scala> import org.apache.spark._  
import org.apache.spark._  
scala> import org.apache.spark.graphx._  
import org.apache.spark.graphx._  
scala> import org.apache.spark.rdd.RDD  
import org.apache.spark.rdd.RDD  
scala> val users: RDD[(VertexId, (String, String))] =  
|   sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),  
|                     (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))  
users: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, (String, String))] = ParallelCollectionRDD[0] at  
parallelize at <console>:29  
scala> val relationships: RDD[Edge[String]] =  
|   sc.parallelize(Array(Edge(3L, 7L, "collab"), Edge(5L, 3L, "advisor"),  
|                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))  
relationships: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[String]] = ParallelCollectionRDD[1] at  
parallelize at <console>:29  
scala> val defaultUser = ("John Doe", "Missing")  
defaultUser: (String, String) = (John Doe, Missing)  
scala> val graph = Graph(users, relationships, defaultUser)  
graph: org.apache.spark.graphx.Graph[(String, String),String] = org.apache.spark.graphx.impl.GraphImpl@35ca1e22  
scala> graph.triplets.collect  
res1: Array[org.apache.spark.graphx.EdgeTriplet[(String, String),String]] =  
Array(((3,(rxin,student)),(7,(jgonzal,postdoc)),collab), ((5,(franklin,prof)),(3,(rxin,student)),advisor),  
((2,(istoica,prof)),(5,(franklin,prof)),colleague), ((5,(franklin,prof)),(7,(jgonzal,postdoc)),pi))  
scala> graph.edges  
res2: org.apache.spark.graphx.EdgeRDD[String] = EdgeRDDImpl[13] at RDD at EdgeRDD.scala:40  
scala> res2.reverse  
res3: org.apache.spark.graphx.EdgeRDD[String] = EdgeRDDImpl[20] at RDD at EdgeRDD.scala:40
```

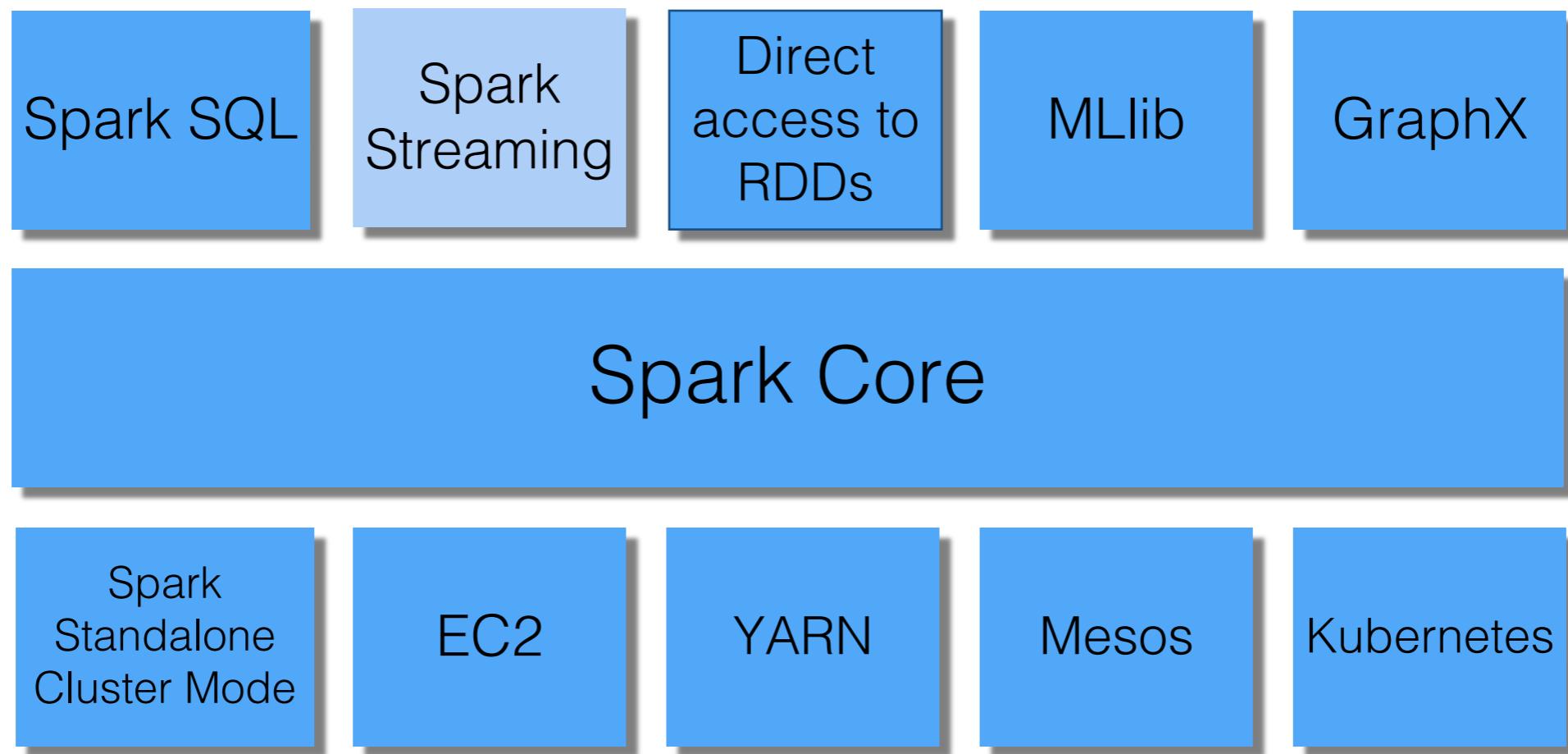
Note that *VertexId* is a type alias for *Long*.

```
case class Edge[ED](srcId: VertexId = 0, dstId: VertexId = 0,  
attr: ED = null.asInstanceOf[ED]) extends Serializable with  
Product
```

Edge Triplet: *Vertex1*, *Vertex2*, *Edge*

Note that *EdgeRDD[X]* is an abstract class
which extends *RDD[Edge[X]]*, with
reverse, *mapValues* and *innerJoin*

Spark Platform



Books: [Learning Spark, Karau et al, \(O'Reilly\)](#);
Spark in Action (Manning).

Spark Streaming

- Spark Streaming...

- is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams;
- data can be ingested from many sources like Kafka, etc.
- finally, processed data can be pushed out to filesystems, databases, and live dashboards.
- *in fact, you can apply Spark's machine learning and graph processing algorithms on data streams.*



Spark Streaming (contd.)

- Internally, it works as follows:
 - Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.



Updated: 2021-03-29

5.4

Machine Learning & MLlib

© 2015-21 Robin Hillyard



What is Machine Learning?

- Machine learning (“ML”) is:
 - when we teach (or train) a software system to perform repetitive tasks such as classification while only providing the system the vaguest idea of how to perform the work;
 - then running that system so that we don’t have to spend the time doing it ourselves.
- Other names have been used for this, such as *expert systems*, even *AI* generally.

ML Components

- Raw data
- Ingestor (also known as *ETL* process: extract-transform-load) which can take the form of:
 - a batch ingestor; or
 - a streaming ingestor (mini-batches separated in the time domain with *exactly-once* guarantee).
- Trainer:
 - training and validation datasets in the form of rows of feature vectors with, optionally, labels;
 - configured algorithm, e.g. linear regression, perceptron.
- Model: the end result of training (and validating). *pmml - models can be moved into another system, by importing model & running it on spark*
- Runner:
 - query processor which transforms a query into a feature vector;
 - model runner (applies the feature vector to the model).

ML types

- Supervised learning:
 - Features are selected by the supervisor;
 - Training/validation sets are *labeled* and supervisor trains the system to do its own labeling
 - Example: classification, regression.
- Unsupervised learning:
 - The training/validation sets are *unlabeled*;
 - We seek *patterns* or *insights* from the data.
 - Example: clustering.
- Deep learning:
 - Features are selected by the system;
 - Example: multi-layer neural nets, “Watson”.

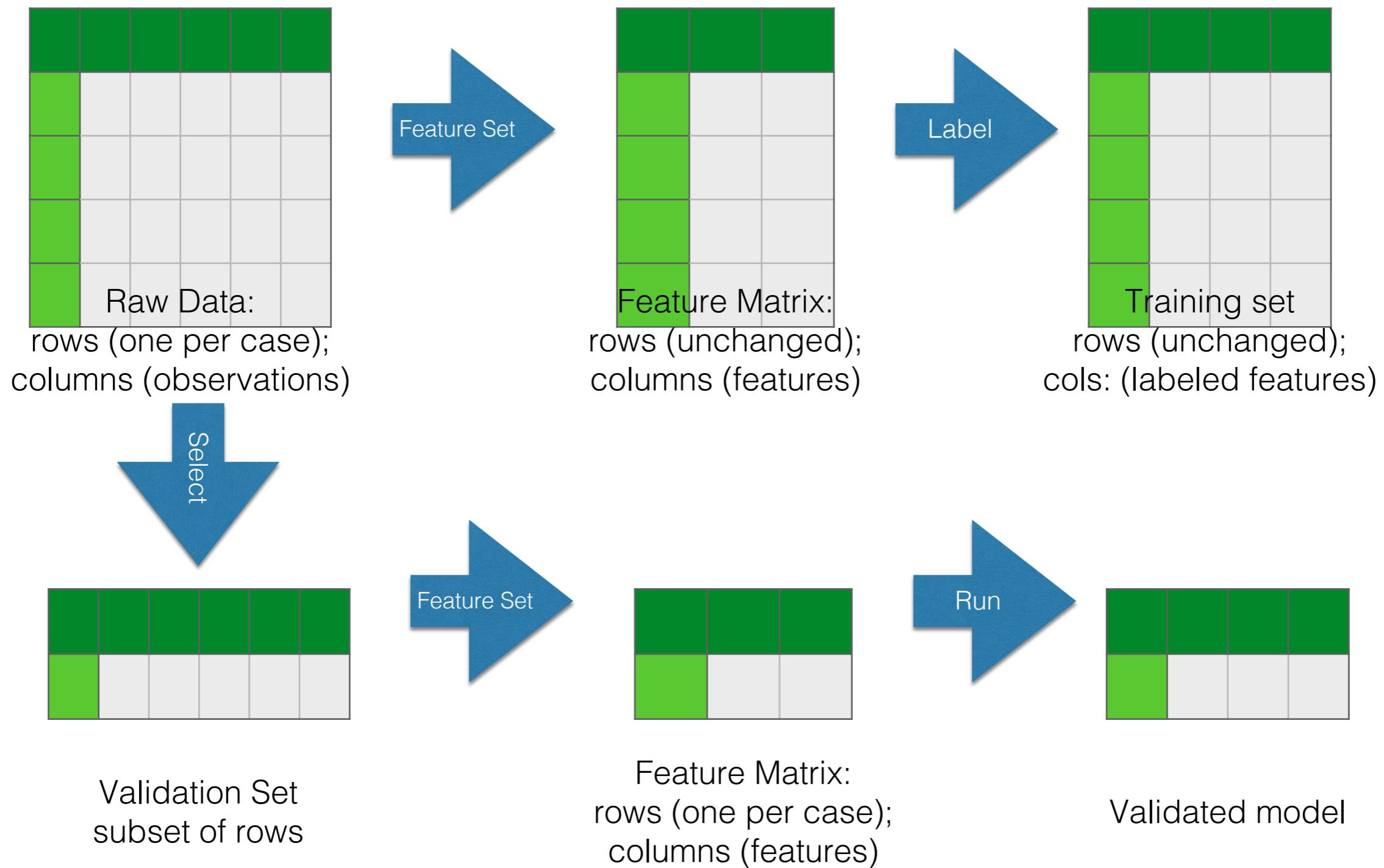
What is ML really doing?

- It is trying to “fit” a model to the training data that you provide;
- By “fit”, we mean that a set of inferences made by the model should have *minimal variance* from the true facts;
- Variance is the square of the Euclidean distance from the point inferred (predicted) from the actual point, i.e. $\sum (y_i - x_i)^2$ where i is over all the “dimensions”/features.
- All ML algorithms are essentially minimizing this total variance in one way or another.

Some notes on entropy

- Entropy in information theory is, like its thermodynamics counterpart, a measure of disorder (unknown variables);
- A ML system starts out with large entropy and that entropy is reduced through training
 - Each feature (a column in the training set) yields some information based on the range of values it can assume: Ex: cols with height in feet & height in meters have mutual information
 - But this can be diminished by “mutual information” with another feature;
 - the information provided by a feature is also proportional to the *number of rows of quality data* provided;
 - But each feature adds an extra degree of freedom to the model (or you can think of it as another dimension)—this *increases* the total entropy of the model
 - Bottom line: additional features that provide little or no information content actually have a positive (i.e. bad) effect on the entropy of the system
 - Very very approximate rule of thumb: you need at least 1000 rows for every feature that you plan to model.

Strategy for (Supervised) ML



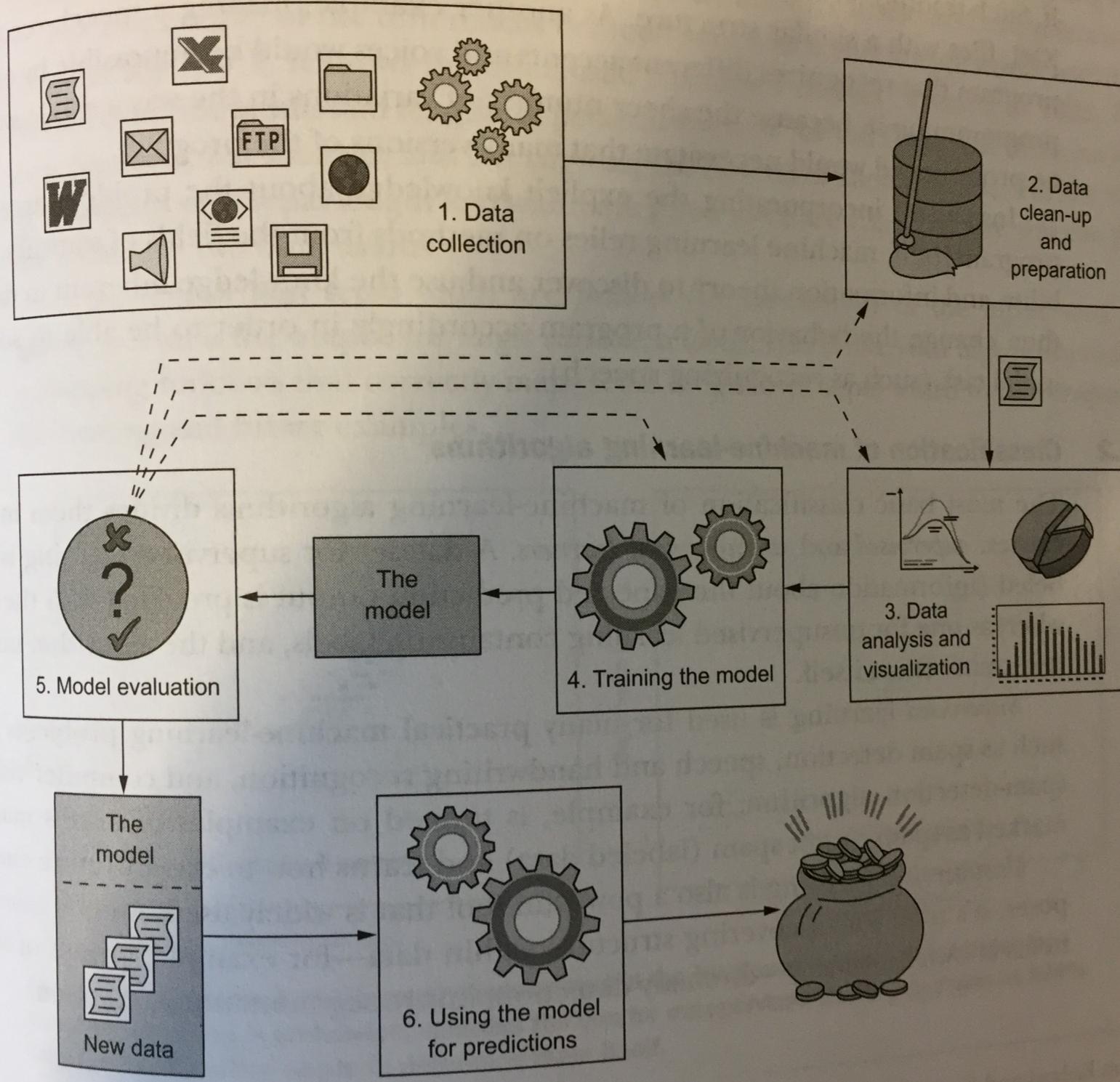


Figure 7.1 Typical steps in a machine-learning project

Another way of looking at it

Figure 7.1 from *Spark in Action*

Data and Features

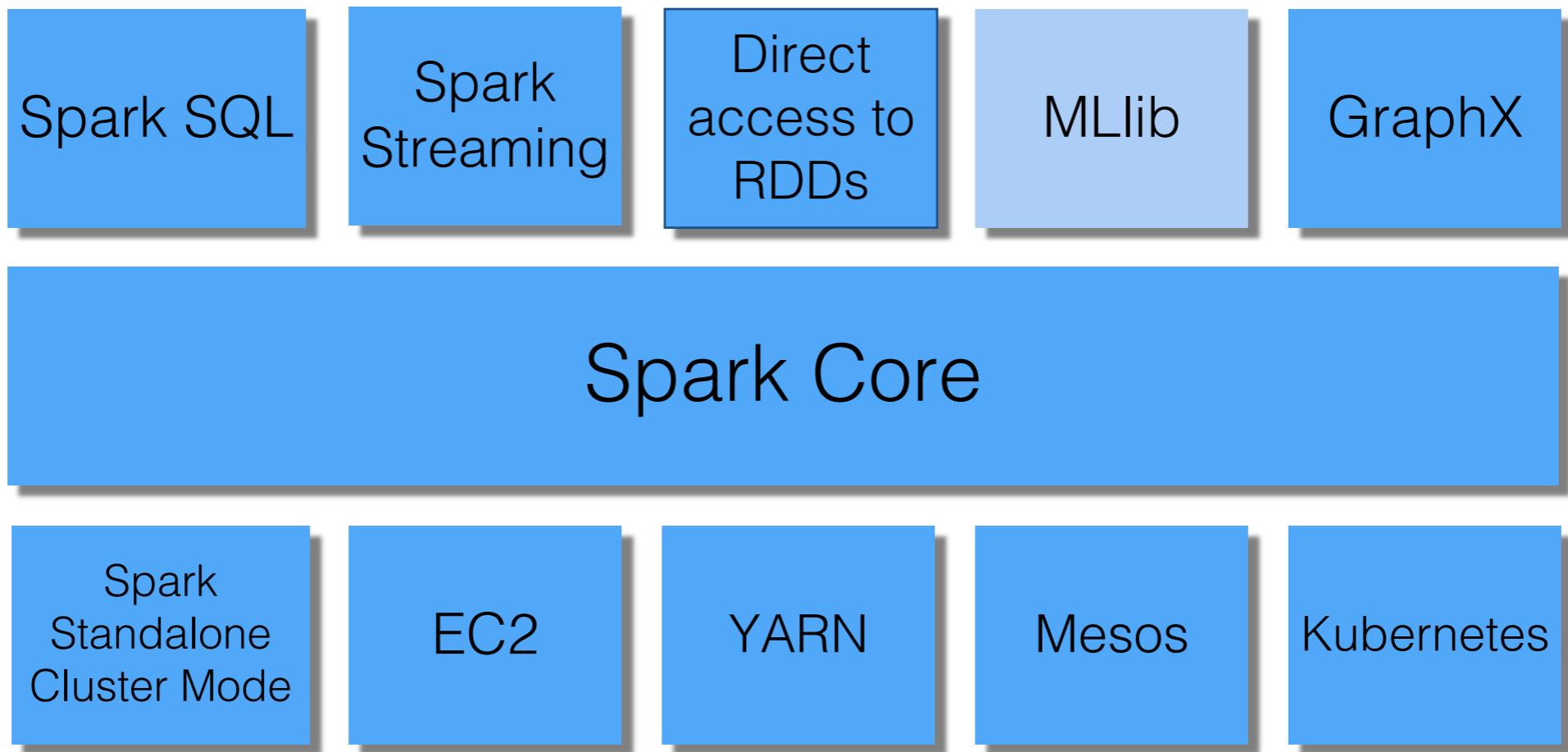
- Scaling:
Standardizes features by removing the mean and scaling to unit variance using column summary statistics on the samples in the training set.
- Statistics:
 - *Statistics.colStats(rdd)* gives column statistics for an RDD of vectors
 - *Statistics.corr(rdd, method)* computes correlation matrix
 - *Statistics.chiSqTest(rdd)* computes independence test for each feature
- Principal Component Analysis ([PCA](#))
 - reduces dimensionality of features to a few uncorrelated dimensions
 - For example, you might reduce to two dimensions and then perform clustering to see if your data appears to divide naturally.

Feature Selection

gif udvd

- Feature selection is the essence of modeling
 - Suppose you have 100 observations but that your model (e.g. Neural Net, SVM, whatever) operates best with 10 inputs/dimensions.
 - You will choose 10 features from your 100 observations (x_j):
 - $f_i = f(x_0, x_1, \dots x_n)$
 - $I(f_i; f_j) \approx 0$ for all i, j where $I(X; Y)$ is the *mutual information* of X & Y
 - i.e. f_i and f_j should be, as far as possible, *independent*.
 - Furthermore, x_j should be clean and unbiased: there should be no x_j values which are out of the x_j domain and:
 - x_j values should be uniformly distributed over its domain.
 - Principle Component Analysis (PCA) will do a decent job of feature selection for you.

Spark Platform



Spark 3.1

- Since Spark 2.0, the primary API is via *DataFrames* (“Spark ML”), although the RDD-based API is extant.
- Features:
 - Learning algorithms for classification, regression, clustering and collaborative filtering,
 - Featurization: PCA, etc,
 - Pipelines
 - Persistence: models, etc.
 - Utilities: built-in image reading, basic statistics.

Introduction to (artificial) neural networks

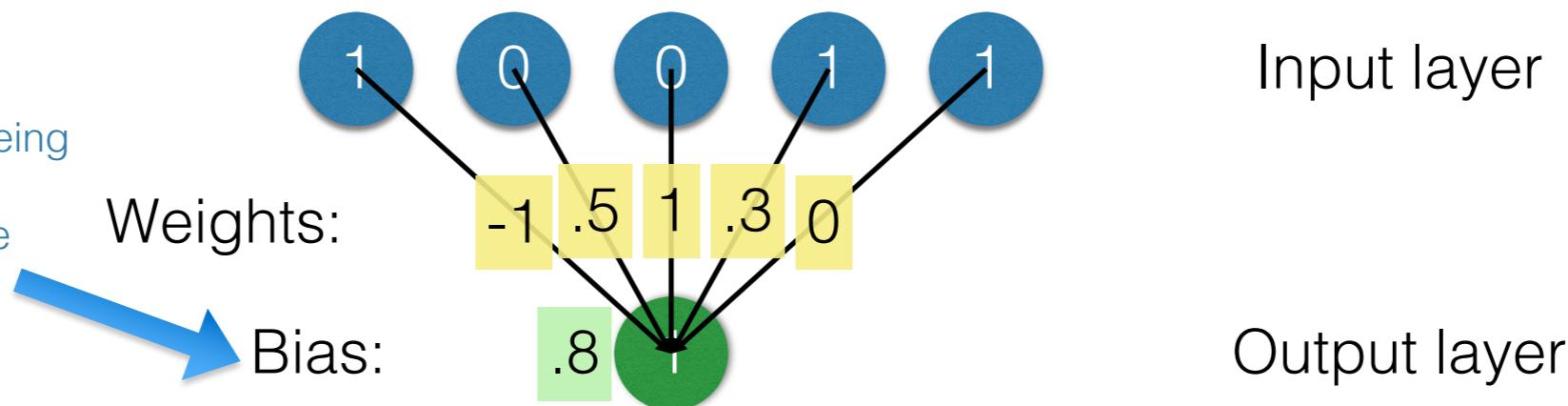
Perceptron Classifier

- What is an Artificial Neural Network (ANN or “Neural Net”)?
 - It’s a general purpose network of quasi-analog “neurons,” which pass and combine “messages” to other neurons. The parameters of the connections and neurons are mutable—that’s to say they can learn.
- What is a Perceptron?
 - A *perceptron* is a very simple form of ANN
 - with neurons arranged in layers such that its “feed-forward” connections are formed only between layers (not between neurons of the same layer)
 - each neuron outputs a binary signal (the “message”)
 - one of the first types of neural net to be built (1957)
 - and able to *classify* input sets with one or more (orthogonal) labels
 - for each neuron, output is calculated as follows:
 - $f(x) = 1 \text{ if } \mathbf{w} \cdot \mathbf{x} + b > 0; \text{ otherwise } 0$
 - where **w** is the weight vector of connections from the previous layer, and
 - where **x** is the (binary) set of outputs at the previous layer,
 - and b is the bias

Training the perceptron

- Two-layer perceptron:

can also think of bias as being
the weight of an additional
input connection with value
constrained to be 1

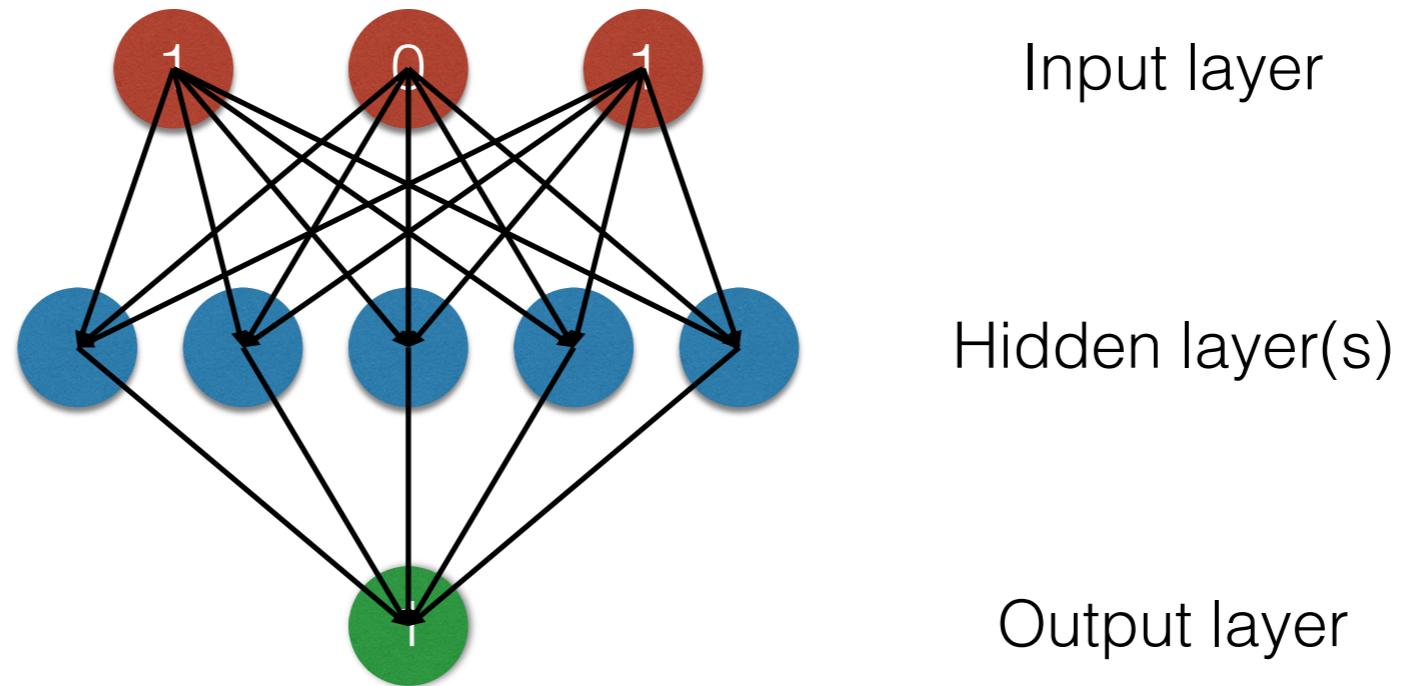


- The weights start out at random (and bias values usually zero):
 - As each new input set is labeled (supervised learning), the weights/bias are adjusted (by back-propagation of errors) until output values match as closely as possible. *has niasl usyll*
 - Provided the input is *linearly separable**^{*}, the perceptron will converge (with some set of weights). The optimally stable solution is known as a *support vector machine*.

* if a hyperplane can be drawn separating the inputs

Solving non-linearly-separable problems

- Use a multi-layer perceptron:



- Now you can solve problems such as XOR

Back-propagation, etc.

- Remember our Newton-Raphson convergence from the first week or two?
 - Essentially, perceptrons use an N-dimensional version of the same thing where N is the number of weights (including bias) that must be adjusted and where the function is linear.
- In practice, perceptrons aren't strictly binary at each neuron
 - Hidden neurons use “sigmoid” function;
 - Output neurons use “softmax” function.

Perceptron code

```
package edu.neu.coe.scala.spark.nn
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.ml.classification.MultilayerPerceptronClassifier
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.sql.Row

object PerceptronClassifier extends App {
    val conf = new SparkConf().setAppName("perceptron")
    val sc = new SparkContext(conf)
    val sqlContext = new org.apache.spark.sql.SQLContext(sc)
    val sparkHome = "/Applications/spark-1.5.1-bin-hadoop2.6/"
    val trainingFile = "data/mllib/sample_multiclass_classification_data.txt"
    // this is used to implicitly convert an RDD to a DataFrame.
    import sqlContext.implicits._

    // Load training data
    val data = MLUtils.loadLibSVMFile(sc, s"$sparkHome$trainingFile").toDF()
    // Split the data into train and test
    val splits = data.randomSplit(Array(0.6, 0.4), seed = 1234L)
    val train = splits(0)
    val test = splits(1)
    // specify layers for the neural network:
    // input layer of size 4 (features), two intermediate of size 5 and 4 and output of size 3 (classes)
    val layers = Array[Int](4, 5, 4, 3)
    // create the trainer and set its parameters
    val trainer = new MultilayerPerceptronClassifier()
        .setLayers(layers)
        .setBlockSize(128)
        .setSeed(1234L)
        .setMaxIter(100)
    // train the model
    val model = trainer.fit(train)
    // compute precision on the test set
    val result = model.transform(test)
    val predictionAndLabels = result.select("prediction", "label")
    val evaluator = new MulticlassClassificationEvaluator()
        .setMetricName("precision")
    println("Precision:" + evaluator.evaluate(predictionAndLabels))
}
```

```

1 1:-0.222222 2:0.5 3:-0.762712 4:-0.833333
1 1:-0.555556 2:0.25 3:-0.864407 4:-0.916667
1 1:-0.722222 2:-0.166667 3:-0.864407 4:-0.833333
1 1:-0.722222 2:0.166667 3:-0.694915 4:-0.916667
0 1:0.166667 2:-0.416667 3:0.457627 4:0.5
1 1:-0.833333 3:-0.864407 4:-0.916667
2 1:-1.32455e-07 2:-0.166667 3:0.220339 4:0.0833333
2 1:-1.32455e-07 2:-0.333333 3:0.0169491 4:-4.03573e-08
1 1:-0.5 2:0.75 3:-0.830508 4:-1
0 1:0.611111 3:0.694915 4:0.416667

```

result: 

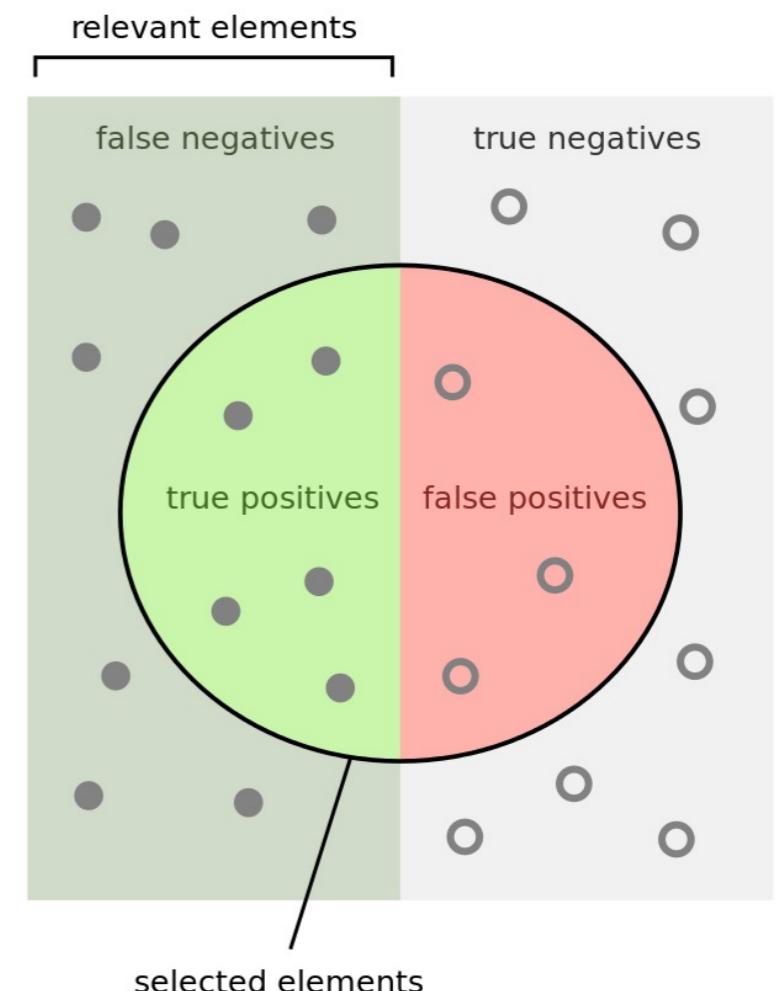
| | prediction | label |
|---|------------|-------|
| 1 | 1.0 | 1.0 |
| 1 | 1.0 | 1.0 |
| 1 | 1.0 | 1.0 |
| 1 | 1.0 | 1.0 |
| 1 | 1.0 | 1.0 |
| 1 | 1.0 | 1.0 |
| 2 | 2.0 | 2.0 |
| 2 | 2.0 | 2.0 |
| 2 | 2.0 | 2.0 |
| 2 | 2.0 | 0.0 |
| 2 | 2.0 | 2.0 |
| 0 | 0.0 | 2.0 |
| 1 | 1.0 | 1.0 |
| 0 | 0.0 | 0.0 |
| 1 | 1.0 | 1.0 |
| 1 | 1.0 | 1.0 |
| 0 | 0.0 | 0.0 |
| 2 | 2.0 | 2.0 |
| 1 | 1.0 | 1.0 |
| 0 | 0.0 | 0.0 |
| 1 | 0.0 | 0.0 |

Why Neural Nets?

- I believe that ANNs are primarily useful because they:
 - are very general;
 - do not require much advance modeling thought;
 - are resistant to mutual information (co-variance);
 - can be used with very large feature sets and training sets—
 - in fact, multi-layer ANNs essentially do the feature selection for you;
 - can be used to gain insights for better modeling.
- What to avoid:
 - without sophisticated optimization (gradient descent, etc.) algorithms, convergence may be slow—or worse
 - MLlib uses L-BFGS (limited memory version of Broyden–Fletcher–Goldfarb–Shanno algorithm)

Classification

- Measurements
 - Precision/Recall (applicable for two classes)
 - e.g. for a recall of 50%, we want a precision of 60%
 - Confusion matrix (for > two classes)



$$\text{Precision} = \frac{\text{How many selected items are relevant?}}{\text{How many relevant items are selected?}}$$
$$\text{Recall} = \frac{\text{How many selected items are relevant?}}{\text{How many relevant items are selected?}}$$

Spark MLlib tools for modeling

- In addition to *RDD*, MLlib uses the following types:

1. Vector: dense and sparse

```
import org.apache.spark.mllib.linalg.{Vector, Vectors}
vectors.dense(1.0, 2.0, 3.0...)
Vectors.dense(array)
vectors.sparse(4, Array(0, 2), Array(1.0, 2.0))
```

2. Matrix:

```
import org.apache.spark.mllib.linalg.{Matrix, Matrices}

// Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))

// Create a sparse matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))
val sm: Matrix = Matrices.sparse(3, 2, Array(0, 1, 3), Array(0, 2, 1), Array(9, 6, 8))
```

3. LabeledPoint

```
new LabeledPoint(label: Double, features: Vector)
```

4. Rating

5. Model

```
e.g. MultilayerPerceptronClassificationModel
```

Models, Algorithms (but see latest Spark doc)

• [Classification](#)

- [Logistic regression](#)
 - [Binomial logistic regression](#)
 - [Multinomial logistic regression](#)
- [Decision tree classifier](#)
- [Random forest classifier](#)
- [Gradient-boosted tree classifier](#)
- [Multilayer perceptron classifier](#)
- [Linear Support Vector Machine](#)
- [One-vs-Rest classifier \(a.k.a. One-vs-All\)](#)
- [Naive Bayes](#)
- [Factorization machines classifier](#)

• [Regression](#)

- [Linear regression](#)
- [Generalized linear regression](#)
 - [Available families](#)
- [Decision tree regression](#)
- [Random forest regression](#)
- [Gradient-boosted tree regression](#)
- [Survival regression](#)
- [Isotonic regression](#)
- [Factorization machines regressor](#)

• [Linear methods](#)

• [Factorization Machines](#)

• [Decision trees](#)

- [Inputs and Outputs](#)
 - [Input Columns](#)
 - [Output Columns](#)

• [Tree Ensembles](#)

- [Random Forests](#)
 - [Inputs and Outputs](#)
 - [Input Columns](#)
 - [Output Columns \(Predictions\)](#)
- [Gradient-Boosted Trees \(GBTs\)](#)
 - [Inputs and Outputs](#)
 - [Input Columns](#)
 - [Output Columns \(Predictions\)](#)

Working with Text

- Stemming: use NLTK
 - alternatively could simply make all lower-case and remove punctuation
- Feature extraction using TF-IDF
 - use *HashingTF* model
 - Compute the IDF
 - Then the TF-IDF vectors
- Now do classification/regression to, for example, distinguish between spam and ham.
 - Use LogisticRegressionModel, for example

TF-IDF

- Term frequency—inverse document frequency
 - See <https://en.wikipedia.org/wiki/Tf–idf>
 - Karen Spärck Jones developed the idea in 1972
 - Essentially tf-idf measures the *contribution to relevant information* summed over all terms in a document. Common words (as measured by their frequency in some *corpus* of documents) are discounted while rare words are emphasized.
 - Often used with k-means clustering to cluster text documents
 - See [spam filter in our repository](#) (taken from *Learning Spark*—Karau et al).

Spam Analyzer



```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.feature.HashingTF
import org.apache.spark.mllib.classification.LogisticRegressionWithSGD

val spam = sc.textFile("/Users/scalaprof/bigdatascalaclass/SparkApp/spam.txt")
val norm = sc.textFile("/Users/scalaprof/bigdatascalaclass/SparkApp/normal.txt")
val tf = new HashingTF(10000)
val spamFeatures = spam.map(email => tf.transform(email.split(" ")))
val normFeatures = norm.map(email => tf.transform(email.split(" ")))
val posExamples = spamFeatures.map(f => LabeledPoint(1, f))
val negExamples = normFeatures.map(f => LabeledPoint(0, f))
val trainingData = posExamples.union(negExamples)
trainingData.cache()
val model = new LogisticRegressionWithSGD().run(trainingData)
val posTest = tf.transform("Subject: Cheap Stuff From: <omg.fu> O M G GET cheap stuff by sending money to Robin Hillyard".split(" "))
val negTest = tf.transform("Subject: Spark From: Robin Hillyard<scalaprof@gmail.com> Hi Adam, I started studying Spark the other day".split(" "))
println(s"Prediction for positive test example: ${model.predict(posTest)}")
println(s"Prediction for negative test example: ${model.predict(negTest)}")
```

FINISHED ▶ ✎ 📄 ⚙

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.feature.HashingTF
import org.apache.spark.mllib.classification.LogisticRegressionWithSGD
spam: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at textFile at <console>:27
norm: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at textFile at <console>:26
tf: org.apache.spark.mllib.feature.HashingTF = org.apache.spark.mllib.feature.HashingTF@1701e4c1
spamFeatures: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector] = MapPartitionsRDD[4] at map at <console>:30
normFeatures: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector] = MapPartitionsRDD[5] at map at <console>:30
posExamples: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] = MapPartitionsRDD[6] at map at <console>:33
negExamples: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] = MapPartitionsRDD[7] at map at <console>:32
trainingData: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPoint] = UnionRDD[8] at union at <console>:40
res3: trainingData.type = UnionRDD[8] at union at <console>:40
model: org.apache.spark.mllib.classification.LogisticRegressionModel = org.apache.spark.mllib.classification.LogisticRegressionModel: intercept = 0.0, numFeatures = 10000, numClasses = 2, threshold = 0.5
posTest: org.apache.spark.mllib.linalg.Vector = (10000,[71,77,79,454,702,1371,2752,3066,3159,3290,3700,3707,4351,5726,6372,7023,9552],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0])
negTest: org.apache.spark.mllib.linalg.Vector = (10000,[73,575,2337,2752,3066,4605,4801,4849,5693,5726,9228,9401,9776],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0])
Prediction for positive test example: 1.0
```

Spam filter

Updated: 2021-03-31

5.5 Play

© 2015-21 Robin Hillyard



What is Play?

- In the beginning (2005)...
 - there was Django (Python) and Rails (Ruby) (followed by Grails (Groovy), etc.)
 - MVC (model-view-controller) frameworks
 - COC (convention-over-configuration): configuration only necessary when deviating from standard pattern
 - DRY (don't-repeat-yourself): minimal coupling
 - ARP (active-record-pattern): object-relational mappings, CRUD
 - Play (2007) (groovy for templates), Play2 (2012)
 - Stateless (*RESTful*: “representational state transfer” or, more simply, using HTTP properly)
 - Asynchronous I/O
 - Scala template engine
 - Native Scala support, etc.

Play/Activator components

- *Play* is based on the following:
 - JBoss *Netty* (non-blocking IO client-server framework, i.e. it's a web-server for JVM apps)
 - built-in hot reloading
 - choice of persistence layer: Slick, H2, Anorm, etc.
- *Activator* is deprecated:
 - <https://groups.google.com/forum/#topic/play-framework/NeMD04W6bN4>
- Now, to create a new Play application based on a template, go to:
 - <https://developer.lightbend.com/start/?group=play>

Key Features

- Declarative application URL scheme configuration
- Type-safe mapping from HTTP to an idiomatic Scala API
- Type-safe template syntax
- Architecture that embraces HTML5 client technologies
- Live code changes when you reload the page in your browser
- Full-stack web framework features, including persistence, security, and internationalization.

Updated: 2021-05-5

5.6

Java-Scala Interoperability

© 2021 Robin Hillyard

The Northeastern University logo is a large, stylized white 'N' on a black background. Inside the 'N', the words "Northeastern University" are written in a white serif font.

Northeastern
University

Interoperating Java with Scala

- The ClassLoader is language-agnostic:
 - Java classes are Scala classes and *vice versa*. The JVM (classloader) doesn't know the difference.
- So, what's the problem?
 - One type of problem arises when you want to extend a Java class with a Scala class, or *vice versa*. This can be done in many situations without problems, but I don't believe there's really any compelling reason to do so.
 - Another issue arises when you have a Java *List* and a Scala *List*. These are not the same class: `java.util.List` and `scala.collection.immutable.List`. Therefore, they cannot be used as if they were. But Scala provides a simple way to convert between the corresponding collection types (see following slides).
 - And then there's the fact that you can pass one of the standard monadic wrappers, e.g. `Try`, back to a Java method, but it won't be able to access it in a monadic way* (not really a major issue).

* actually, this may no longer be true with Java 14

Interoperating Java with Scala (2)

- Best practices:
 - Define cross-language APIs as much as possible using compatible types:
 - Simple, unwrapped, scalar types (*String*, *Int/Integer*, *Double*, etc.)—automatically converted;
 - “our own” data Structures;
 - Tend to call Scala code from Java but not the other way around:
 - Provide additional Java-centric signatures which call their corresponding Scala methods;
 - Don’t return *Try[X]*: instead return *Option[X]* after logging the exception in the Scala method;
 - But if you return *Try[Boolean]* from Scala, it will not be converted to *Try<java.lang.Boolean>* in Java. You can only wrap explicit classes (that are the same).
 - For instance, a wrapped *Unit* gets converted to a wrapped *BoxedUnit*.
 - When it is necessary to reference a collection in a method, I recommend doing the conversion in Scala code and providing a Java-specific API for that method (in addition to the Scala API).

Interoperating Java with Scala (3)

- Best practices, continued:
 - When defining a Scala method with a function parameter, use only simple functions:
 - non-curried functions: i.e. those that correspond to Java8's function types.
 - Don't define a tuple as the return type from a Scala method:
 - instead, define a case class in your Scala code and return that so that it can be easily referenced on the Java side.
 - Avoid defining parameters with default values (Java can't omit the argument):
 - Instead, just create a method signature without the default value.
 - It is possible to use the `object.method$default$Number()` mechanism but that is extremely inelegant!
 - If your Scala method expects an `Option[T]`, then pass values as either `Option.apply(t)` or `Option.empty()`:
 - Alternatively, use `Optional<T>` in your signature.

Example of passing in *Optional<T>*

- If you pass in *Optional<T>*, then you will need to convert to *Option[T]*:
 - You can do this by code:
 - `def toJavaOptional[A](maybeA: Option[A]): Optional[A] = maybeA match {
 case Some(a) => Optional.of(a)
 case _ => Optional.empty()
}`
 - `def toScalaOption[A](maybeA: Optional[A]): Option[A] = if (maybeA.isPresent)
 Some(maybeA.get) else None`
 - Or you can add a dependency (sbt):
`"org.scala-lang.modules" %% "scala-java8-compat" % "0.9.0" (sbt)`
 - Or (maven):
`<dependency>
 <groupId>org.scala-lang.modules</groupId>
 <artifactId>scala-java8-compat_2.11</artifactId>
 <version>0.9.0</version>
</dependency>`
 - And then, in your code:
`import scala.compat.java8.OptionConverters._
javaOptional.asScala`
 - Or
`scalaOption.asJava`

Interoperating Java with Scala: CollectionConverters

- Here is the definitive list of implicit conversions:
 - [https://www.scala-lang.org/api/2.13.5/scala/jdk/CollectionConverters\\$.html](https://www.scala-lang.org/api/2.13.5/scala/jdk/CollectionConverters$.html)
 - Import the converters and add `asScala` or `asJava` where appropriate.
 - Note that `Seq` in Scala is a trait, not a class. You cannot instantiate `Seq` like `List`.
 - However, in Scala, `Seq(1,2,3)` gets desugared into `List(1,2,3)` so it seems like it's a class with its own constructor or `apply` method.
 - Thus if you have a `java.util.List<A>` *list* and need a `Seq[A]`, just import the converters and write `list.asScala`.

Example: collections (Scala 2.12)

- Scala:

```
object Collections {  
  
    def show[A](xs: Iterable[A]): Unit = xs foreach println  
  
    def showJava[A](xs: java.util.Collection[A]): Unit = {  
        import collection.JavaConverters._  
        show(xs.asScala)  
    }  
}
```

- Java:

```
public class CollectionsJ {  
  
    public static void main(String[] args) {  
        Collection<String> strings = new ArrayList<>();  
        strings.add("Curriculum");  
        strings.add("Associates");  
        Collections.showJava(strings);  
    }  
}
```

Example: Shuffle (Scala 2.13)

- Scala:

```
package com.phasmidsoftware.util

import java.util
import java.util.Random
import scala.collection.mutable.ListBuffer

object Shuffle {

    def apply[X](xs: Iterable[X]): Seq[X] = {
        import scala.jdk.CollectionConverters.-
        val listBuffer: ListBuffer[X] = new ListBuffer().appendAll(xs)
        val jList: util.List[X] = listBuffer.asJava
        java.util.Collections.shuffle(jList, new Random())
        jList.asScala.toSeq
    }
}
```

Interoperating Java with Scala: Collections (2)

- Here's a neat trick that allows you to construct Java lists*:

- It uses the same ability you use to construct Scala lists:
 - *Collections.scala*

```
import collection.JavaConverters._  
import scala.annotation._  
@varargs def createJavaList[A](xs: A*): java.util.List[A] = xs.asJava
```

- *CollectionsJ.java*

```
List<String> ca = Collections.createJavaList("Curriculum", "Associates");  
Collections.showJava(ca);
```

- The type of *ca* is a *Wrappers\$SeqWrapper* which may not be what you want. But you can always get, say, an *ArrayList* like so:

```
new ArrayList<>(Collections.createJavaList("Curriculum", "Associates"));
```

*** For some reason, the Java designers forgot to give us this ability**

Example: dealing with *Try[Unit]* in Java*:

- Scala code:

```
object Trial {  
    def trial(b: Boolean): Try[Unit] = if (b) Success()  
        else Failure(new Exception("b was false"))  
}
```

- Java code (version 1):

```
public static void main(String[] args) {  
    Try<BoxedUnit> good = Trial.trial(true);  
    if (good.isSuccess()) System.out.println("good is OK");  
  
    Try<BoxedUnit> bad = Trial.trial(false);  
    if (bad.isFailure()) System.out.println("bad is OK");  
    String msg = bad.failed().get().getLocalizedMessage();  
    System.out.println("failure message: "+msg);  
}
```

* If you feel you really have to

Example: dealing with *Try[Unit]* in Java* (2):

- Scala code:

```
object Trial {  
    def trial(b: Boolean): Try[Unit] = if (b) Success() else Failure(new  
Exception("b was false"))  
}
```

- Java code (version 2):

```
public static void main(String[] args) {  
    Try<BoxedUnit> tried = Trial.trial(false);  
    tried.transform(TrialJ::processUnit, TrialJ::handleException);  
}  
private static Try<BoxedUnit> processUnit(BoxedUnit x) {  
    System.out.println("OK");  
    return new scala.util.Success<>( x );  
}  
  
private static Try<BoxedUnit> handleException(Throwable t) {  
    System.out.println("Exception thrown: "+t.getLocalizedMessage());  
    BoxedUnit x = BoxedUnit.UNIT;  
    return new scala.util.Success<>( x );  
}
```

* If you feel you really have to

Example:

- Java code fragment:

```
// Get the datasets.  
private Iterable<SourceDataSet> datasets = CleverDataSets.allSetsJava();  
// Set up the flow.  
private Flow<Progenitor, Account> flow = CleverAccountFlow.createFromJava(datasets, batchSize,  
persister, akkaSystemName);
```

- Scala code fragments

```
lazy val allSetsJava: java.lang.Iterable[SourceDataSet] = {  
  import scala.collection.JavaConverters.  
  import scala.language.implicitConversions  
  allSets.asJava  
}  
  
def createFromJava(datasets: java.lang.Iterable[SourceDataSet], batchSize: Int, persister:  
  Persister, akkaSystemName: String): CleverAccountFlow =  
  new CleverAccountFlow(datasets.asScala, batchSize, persister, akkaSystemName)
```

Updated: 2025-04-08

5.3 Scala 3

© 2015 Robin Hillyard

A large, semi-transparent watermark of the Northeastern University logo is positioned in the lower right quadrant of the slide. The logo consists of a stylized 'N' and 'U' formed by white shapes on a black background.

Northeastern
University

What's wrong with Scala 2?

- People complained about the speed of compilation
 - But it's always seemed fast to me. It's rare that I compile a whole project but when I do, it's fast!
- People complained that implicits were difficult to understand.
 - And, in particular, traits can't take implicit evidence (as in context bounds like [A: Numeric])
- Some people wanted Scala to be more like Python
 - And, in particular, they apparently didn't like having to put in those (rare) curly braces.
- Enums weren't great (based on Java enums)

Scala 3 Timeline

- The first commit occurred in 2012 (!)
 - That's about the time I started programming in Scala
- Dotty 0.1.2-RC1 in 2017
- Scala 3 was officially [announced](#) in May 2021
 - Supported by libraries such as Cats at the same time
- A few articles:
 - [Scala 3 Book: Introduction](#) (docs.scala-lang.org)
 - [Scala 3 Migration: New in Scala 3](#) (docs.scala-lang.org)
 - [A detailed comparison between Scala 2 and Scala 3](#) (Medium)
 - [Scala 3 Overview](#) (DE Academy)
 - Let me know if you find other useful articles.

Overview

- “New and Shiny”
 - “Quiet” control syntax for if, while, for, etc.
 - `new` keyword is optional (it was rare anyway)
 - Optional braces (i.e., Python style)
 - Wildcard in type definitions changed from `_` to `?`
 - “Heavily” revised implicits

Quiet Control Syntax

- Can drop parentheses on if conditions, and curly braces/parentheses for for-comprehensions.
- Examples:

```
if x < 0 then "negative" else if x == 0 then "zero" else "positive"
```

```
for x <- xs if x > 0 yield x * x
```

```
for x <- xs y <- ys do println(x + y)
```

Contextual Abstractions

- Using clauses use implicits
- Given Instances declare implicits
- Extension Methods uniform way of extending behaviour of types
- Etc.

Type System Improvements

- Enumerations
- Opaque Types
- Intersection and Union Types
 - either type - union
 - tuple - contains all types
- Etc.

Object Oriented Programming

- Traits are more like classes; trait can have parameters in scala 3
- Open classes; allows for a lib class to be extended with behavior by declaring it explicitly
- Etc.

Metaprogramming

- **Inline** reduces overhead at run time
- **Compile-time** can create compile time constants
- Quoted code blocks
- Reflection

Migration

- Easy to prepare for migration by adding compiler flag
`-Xsource:3` to 2.13 code (see [Scala 3 Migration Guide](#))
- Alternatively, jump right into Scala 3 but ask for forgiveness with invalid syntax using another compiler flag.

Continuous Integration (CI)

CSYE 7200

Continuous Integration (CI)

- Why CI?
- GitHub
- CircleCI
- Lifecycle of CircleCI
- Demo
- More

Why CI?

- Maintain a code repository
- Automate the build
- Self-testing
- Every commit should be built
- Test in a clone of the production/clean environment
- Make it easy to get the latest deliverables
- Everyone can see the results of the latest build
- Automate deployment

GitHub

- GitHub Education Pack
- <https://education.github.com/pack>

CI Tools

- Jenkins — self-contained, not cloud based, more control
- Travis CI — cloud based, free for public project
- Circle CI — cloud based, free for one project both public and private
- You may use either TravisCI or CircleCI for your Final Project as long as you can prove your project compiled and passed all tests in CI env
- This demo is based on CircleCI

Circle CI

- GitHub->Marketplace->CircleCI
- <https://github.com/marketplace/circleci>
- Free for both public repo and private repo
- Support Scala
- v2.0— — Faster, configuration file required
- Run on Ubuntu 14.04 (Trusty)

Lifecycle of CircleCI

- Environment setup
- Checkout
- Fetch cache*
- Compile
- Store cache*
- Test
- Publish*
- Upload*

Demo

- Link IntelliJ IDEA to GitHub
- Create a new project with unit test
- Import into GitHub
- Apply Circle CI to the project
- Add badge and Slack notification for your project

Link IntelliJ IDEA to GitHub

- File -> Settings -> Version Control -> GitHub
- Host: github.com
- Login: your GitHub user name
- Password: your GitHub password
- Click Test button it should success

Import into GitHub

- VCS -> Import into Version Control -> Share project on GitHub
- New repository name: your project name on GitHub
- Remote name: used for IntelliJ, not branch name
- Only Commit your source file(src) and build.sbt for now
- *Add .gitignore in your local repo

Apply Circle CI to the project(1)

- Login GitHub: <https://github.com/>
- Marketplace -> Continuous integration -> CircleCI
- Choose Free Plan -> Install -> Authorize
- Goto CircleCI Dashboard: [https://circleci.com/
dashboard](https://circleci.com/dashboard)
- Projects -> Add Project
- Find your project -> Setup project

Apply Circle CI to the project(2)

- Open a new tab in your browser and goto the GitHub page of your project
- Create new file
- Name of file: `.circleci/config.yml` (Yes, there is a `.` at beginning)
- Copy the content of `Sample.yml` to your `config.yml`
- Change line 17: `working_directory: ~/repo` to `working_directory: ~/your project name`
- Commit new file
- Go back to CircleCI page and click Start building

Add badge and Slack notification for your project

- Goto CircleCI Dashboard
- Projects -> Find your project -> Settings
- Notifications -> Status Badges
- Copy Embed Code to your README.md
- Notifications -> Chat Notifications -> Slack -> CircleCI Integration
- Choose a channel (can be your team channel) -> Add CircleCI Integration
- Copy the Webhook link back to your CircleCI page and Save

More

- Skip Tests
- Run Linux Command
- Download Artifacts
- Workflow
- Refer to LaScala & Darwin repo

Q&A

Proof by Induction

Induction

- Often in mathematics or computer science, we need to prove a relationship involving some number n :
 - $S_n = f(n)$ where:
 - S_n represents the left-hand-side of the relationship and where
 - $f(n)$, some polynomial function of n , represents the right-hand side.
- Induction allows us to prove our relationship using two steps:
 - Base case
 - Inductive step

Base case

- If, for example, we want to prove $S_n = f(n)$ for all positive n , then we should choose for our base case $n=0$ or perhaps $n=1$.
- We simply verify that the relationship holds in the base case.

Inductive step

- If, for example, we want to prove $\mathbf{S}_n = f(n)$ for all positive n , then we assume that $\mathbf{S}_n = f(n)$ and then, using that as a relationship as if it were fact, we show that $\mathbf{S}_{n+1} = f(n+1)$.
- Our ability to prove the inductive step will depend on our knowledge of the behavior of \mathbf{S}_n .

The proof

- If we confirm the base case ($n=0$) and we confirm the inductive step such that if the relationship is true for n , it is true for $n+1$, then we can combine these “facts” and assert that the relationship is *true for all positive integers*.

A simple example

- We will prove a formula for the sum of all integers 1 through n .
- Relationship to prove: $\mathbf{S}_n = n(n+1)/2$
- Base case ($n=1$): $1 = 1(2)/2$ (confirmed)
- Inductive step: $\mathbf{S}_{n+1} - \mathbf{S}_n = n+1$ (by definition of \mathbf{S}_n)
 - Using the given relationship, we have:
 - $\mathbf{S}_{n+1} - \mathbf{S}_n = (n+1)(n+2)/2 - n(n+1)/2 = (n+1)/2 * (n+2-n)$
 - $\mathbf{S}_{n+1} - \mathbf{S}_n = n+1$
 - *QED, 证毕, इति सिद्धम्*

Spark - DataFrame API

Brief overview

- Inspired by python pandas
- Spark DataFrames are like distributed in-memory tables with named columns and schemas, where each column has a specific data type: integer, string, array, map, real, date, timestamp, etc.
- DataFrames are immutable and Spark keeps a lineage of all transformations.
- You can add or change the names and data types of the columns, creating new DataFrames while the previous versions are preserved.
- A named column in a DataFrame and its associated Spark data type can be declared in the schema.

Table format of a DataFrame

| Id (Int) | First (String) | Last (String) | Url (String) | Published (Date) | Hits (Int) | Campaigns (List[Strings]) |
|---------------------|---------------------------|--------------------------|---|-----------------------------|-----------------------|--------------------------------------|
| 1 | Jules | Damji | https://tinyurl.1 | 1/4/2016 | 4535 | [twitter, LinkedIn] |
| 2 | Brooke | Wenig | https://tinyurl.2 | 5/5/2018 | 8908 | [twitter, LinkedIn] |
| 3 | Denny | Lee | https://tinyurl.3 | 6/7/2019 | 7659 | [web, twitter, FB, LinkedIn] |
| 4 | Tathagata | Das | https://tinyurl.4 | 5/12/2018 | 10568 | [twitter, FB] |

Spark Dataframe API code

```
import org.apache.spark.sql.functions.avg
import org.apache.spark.sql.SparkSession
// Create a DataFrame using SparkSession
val spark = SparkSession
    .builder
    .appName("AuthorsAges")
    .getOrCreate()
// Create a DataFrame of names and ages
val dataDF = spark.createDataFrame(Seq(("Brooke", 20), ("Brooke", 25), ("Denny", 31),
("Jules", 30), ("TD", 35))).toDF("name", "age")
// Group the same names together, aggregate their ages, and compute an average
val avgDF = dataDF.groupBy("name").agg(avg("age")) // Show the results of the final
execution
avgDF.show()
```

Output

| name | avg(age) |
|--------|----------|
| Brooke | 22.5 |
| Jules | 30.0 |
| TD | 35.0 |
| Denny | 31.0 |

Spark's Basic Data Types

| Data type | Value assigned in Scala | API to instantiate |
|-------------|-------------------------|-----------------------|
| ByteType | Byte | DataTypes.ByteType |
| ShortType | Short | DataTypes.ShortType |
| IntegerType | Int | DataTypes.IntegerType |
| LongType | Long | DataTypes.LongType |
| FloatType | Float | DataTypes.FloatType |
| DoubleType | Double | DataTypes.DoubleType |
| StringType | String | DataTypes.StringType |
| BooleanType | Boolean | DataTypes.BooleanType |
| DecimalType | java.math.BigDecimal | DecimalType |

Complex Data Types

| Data type | Value assigned in Scala | API to instantiate |
|----------------|--|---|
| BinaryType | Array[Byte] | DataTypes.BinaryType |
| Timestamp Type | java.sql.Timestamp | DataTypes.TimestampType |
| DateType | java.sql.Date | DataTypes.DateType |
| ArrayType | scala.collection.Seq | DataTypes.createArrayType(Element Type) |
| MapType | scala.collection.Map | DataTypes.createMapType(keyType, valueType) |
| StructType | org.apache.spark.sql.Row | StructType(ArrayType[fieldTypes]) |
| StructField | A value type corresponding to the type of this field | StructField(name, dataType, [nullable]) |

```
import org.apache.spark.sql.types._  
import org.apache.spark.sql.SparkSession  
import org.apache.spark.sql.types._  
  
val schema = StructType(Array(StructField("author", StringType, false),  
StructField("title", StringType, false), StructField("pages", IntegerType, false)))  
val schema = "author STRING, title STRING, pages INT"  
object Example3_7 {  
    def main(args: Array[String]) {  
        val spark = SparkSession .builder  
            .appName("Example3_7")  
            .getOrCreate()  
        if (args.length <= 0) {  
            println("usage Example3_7 <file path to blogs.json>") System.exit(1)  
        }  
        // Get the path to the JSON file  
        val jsonFile = args(0)  
        // Define our schema programmatically  
        val schema = StructType(Array(StructField("Id", IntegerType, false),  
StructField("First", StringType, false), StructField("Last", StringType, false), StructField("Url", StringType, false),  
StructField("Published", StringType, false), StructField("Hits", IntegerType, false), StructField("Campaigns", ArrayType(StringType), false)))  
        // Create a DataFrame by reading from the JSON file // with a predefined schema  
        val blogsDF = spark.read.schema(schema).json(jsonFile) // Show the DataFrame schema as output blogsDF.show(false)  
        // Print the schema  
        println(blogsDF.printSchema)  
        println(blogsDF.schema)  
    }  
}
```

Output

| Id | First | Last | Url | Published | Hits | Campaigns |
|-----------|--------------|-------------|-------------------|------------------|-------------|------------------------|
| 1 | Jules | Damji | https://tinyurl.1 | 1/4/2016 | 4535 | [twitter, LinkedIn] |
| 2 | Brooke | Wenig | https://tinyurl.2 | 5/5/2018 | 8908 | [twitter, LinkedIn] |
| 3 | Denny | Lee | https://tinyurl.3 | 6/7/2019 | 7659 | [web, twitter,...] |
| 4 | Tathagata | Das | https://tinyurl.4 | 5/12/2018 | 10568 | [twitter, FB] |
| 5 | Matei | Zaharia | https://tinyurl.5 | 5/14/2014 | 40578 | [web, twitter, FB,...] |
| 6 | Reynold | Xin | https://tinyurl.6 | 3/2/2015 | 25568 | [twitter, LinkedIn] |

root

```
|-- Id: integer (nullable = true)
|-- First: string (nullable = true)
|-- Last: string (nullable = true)
|-- Url: string (nullable = true)
|-- Published: string (nullable = true)
|-- Hits: integer (nullable = true)
|-- Campaigns: array (nullable = true)
|   |-- element: string (containsNull = true)
```

Spark Demo on AWS

CSYE7200

Spark Example

- WordCount Example
- Setup AWS EMR & AWS S3
- Run Spark application on AWS EMR

WordCount Example

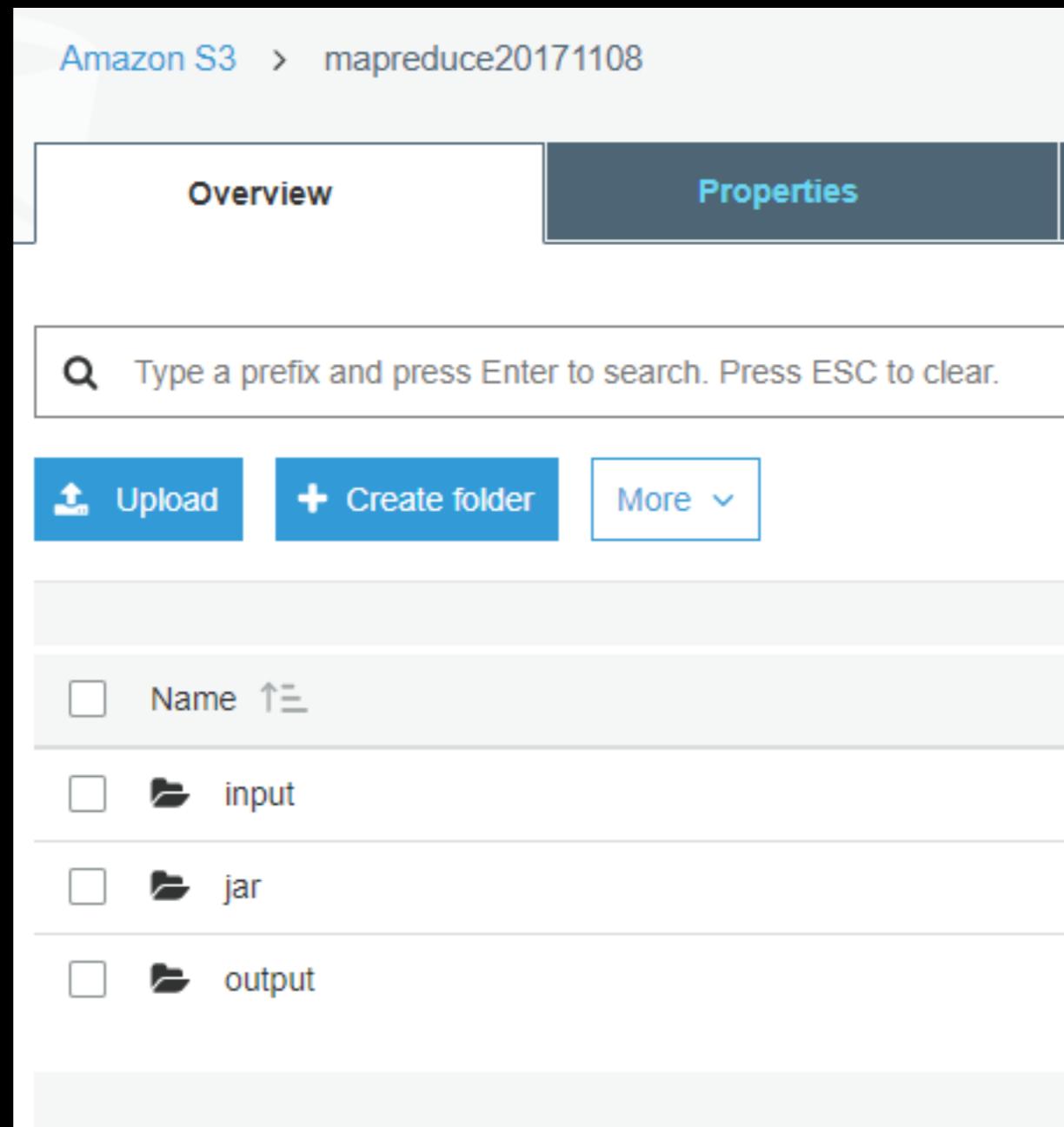
- <https://github.com/rchillyard/CSYE7200/tree/master/spark-example>
- sbt package
- Get your jar file from projectRoot/target/scala-2.11

Setup AWS S3

- Create a bucket
- Upload your jar file & input file
- Make sure that output path for next job does NOT exist

Setup AWS S3

- Use default for all upload file



Setup AWS EMR

- <https://aws.amazon.com/emr/>
- Create Cluster
- Add Step

Add Step

Add step X

Step type Spark application Run Spark application using spark-submit. [Learn more](#)

Name WordCountExample

Deploy mode Cluster Run your driver on a slave node (cluster mode) or on the master node as an external client (client mode).

Spark-submit options `--class edu.neu.csye..7200.WordCountAWS` Specify other options for spark-submit.

Application location* `s3://mapreduce20171108/jar/sparkexample_2.11-0.1.jar` Path to a JAR with your application and dependencies (client deploy mode only supports a local path).

Arguments `s3n://mapreduce20171108/input/WordCount.txt
s3n://mapreduce20171108/output/7` Specify optional arguments for your application.

Action on failure Continue What to do if the step fails.

[Cancel](#) [Add](#)

Wait Result

Clone Terminate AWS CLI export

Cluster: WordCountExample Terminated Steps completed

Summary Application history Monitoring Hardware Events **Steps** Configurations Bootstrap actions

Add step Clone step Cancel step

Steps

| | ID | Name | Status | Start time (UTC+8) | Elapsed time |
|-----------------------|----------------|------------------|-----------|--------------------------|--------------|
| <input type="radio"/> | s-2I8HO2B47BX6 | WordCountExample | Completed | 2017-11-16 07:47 (UTC+8) | 1 minute |

JAR location : command-runner.jar
Main class : None
spark-submit --deploy-mode cluster --class edu.neu.csye._7200.WordCountAWS s3://mapreduce20171108/jar/sparkexample_2.11-0.1.jar s3n://mapreduce20171108/output/7
Arguments : s3n://mapreduce20171108/output/7

Action on failure: Continue

| | ID | Name | Status | Start time (UTC+8) | Elapsed time |
|-----------------------|-----------------|------------------------|-----------|--------------------------|--------------|
| <input type="radio"/> | s-3VW59GDMK5RG9 | Setup hadoop debugging | Completed | 2017-11-16 07:47 (UTC+8) | 2 seconds |

Result

Amazon S3 > mapreduce20171108 / output / 7

Overview

Type a prefix and press Enter to search. Press ESC to clear.

Upload **Create folder** **More**

| <input type="checkbox"/> | Name | Size |
|--------------------------|------------|------|
| <input type="checkbox"/> | _SUCCESS | |
| <input type="checkbox"/> | part-00000 | |

Amazon S3 > mapreduce20171108 / output / 7sql

Overview

Type a prefix and press Enter to search. Press ESC to clear.

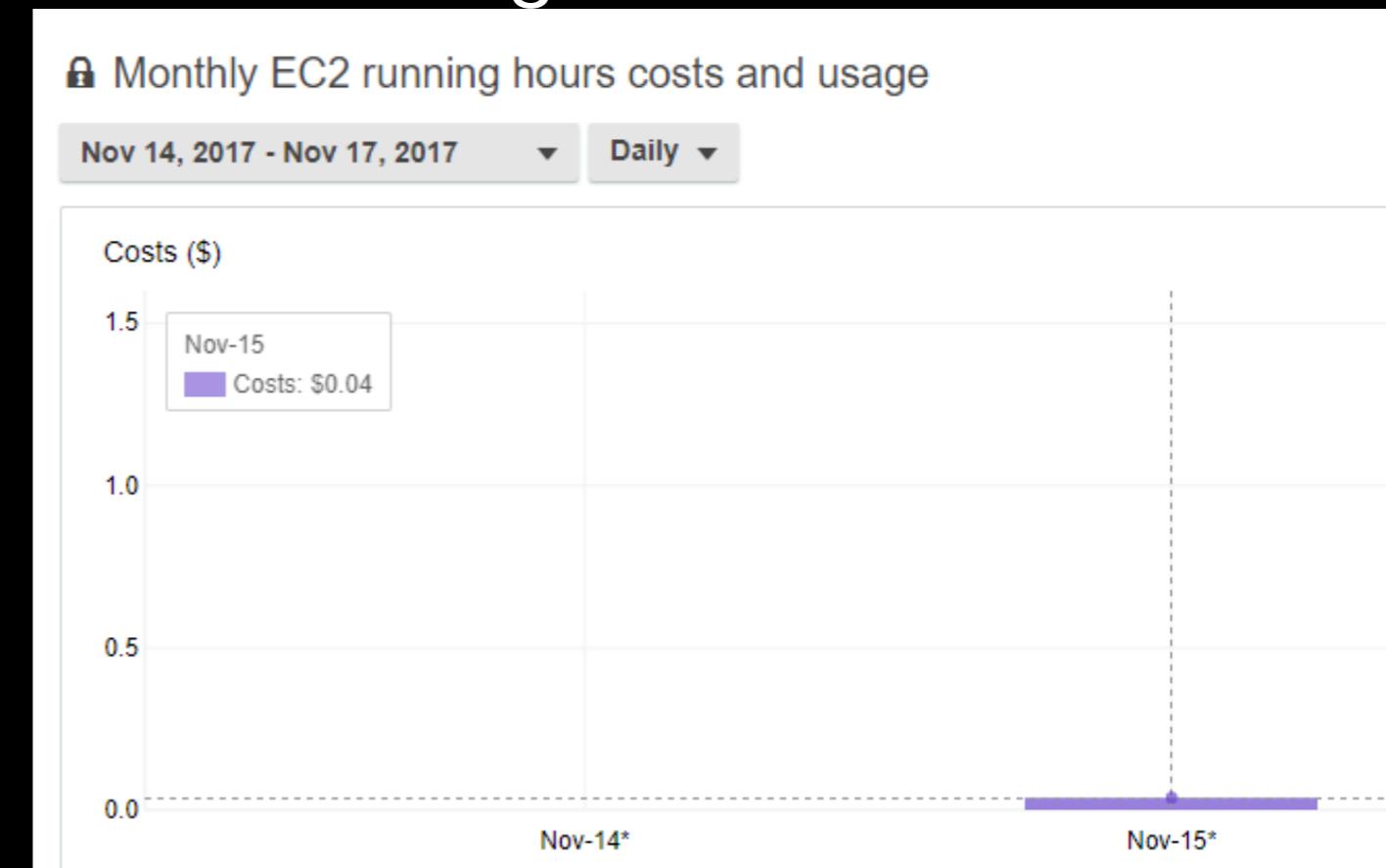
Upload **Create folder** **More**

| <input type="checkbox"/> | Name | Size |
|--------------------------|--|------|
| <input type="checkbox"/> | _SUCCESS | |
| <input type="checkbox"/> | part-00000-4f122006-a2f8-4c77-81ab-9f77dd40e6e5-c000.csv | |

About Bill

- EMR is NOT included in Free Tier
- It is included in your Education Credit
- A running cluster of 3 nodes(1 master, 2 slave on m4large) will charge about 1 quarter per hour, make sure you terminate your cluster if you are not using it

- This example(10 mins)
- cost about \$0.04
- Other charge may apply



Q&A

Unit Testing

CSYE 7200

Testing

- Unit Test
- System Integration Test (SIT)
- User Acceptance Test (UAT)

Unit Test

- Single logic
 - Eg. Each filter and map in Spark processing
- Cover all branches
 - Eg. If statement: true and false
 - Eg. Case match: all scenario
- Coverage

Unit Test

- Input: should be small but representative
 - Create test data
 - Random Sampling real data
- Should NOT use absolute path for test file
- Should NOT use standalone database(eg. MySQL)

Unit Test Example

- In class repo:
- <https://github.com/rchillyard/CSYE7200/blob/master/spark-example/src/test/scala/edu/neu/coe/csye7200/WordCountSpec.scala>
- Eg. myFilter and myReplacer

SIT

- A high-level software test
 - Eg. A test which simulate Spark inputs and check output

SIT Example

- In class repo:
- <https://github.com/rchillyard/CSYE7200/blob/master/spark-example/src/it/scala/edu/neu/coe/csye7200/WordCountSpark2ItSpec.scala>
- Eg. wordCount3

Mocking

- org.scalamock
- https://github.com/yangy4/CSYE7200_FinalProject_Team2_Spring2017/blob/master/src/test/scala/edu/neu/coe/scala/retrieval/UsecasesSpec.scala

Q&A

Updated: 2022-04-21

6.0

Variance

© 2015 Robin Hillyard

A large, semi-transparent watermark of the Northeastern University logo is positioned in the lower right quadrant. The logo consists of a stylized 'N' and 'U' formed by white diagonal bars on a black background.

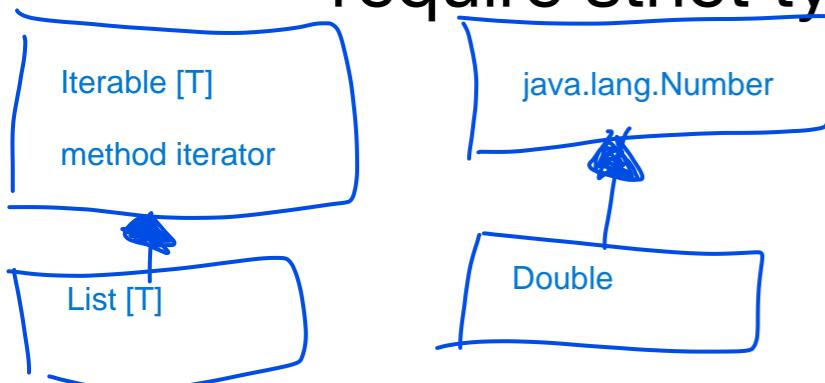
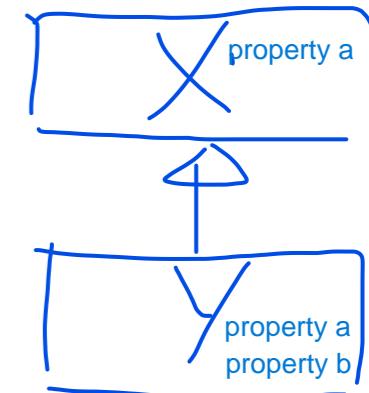
Northeastern
University

Variance: Object-oriented programming in a strict-type context

X >: Y // X is supertype of Y
Y <: X // Y is subtype of X

- Variance

- What is variance and why do we need it?
- Is it some strange feature of functional programming?
- No, it's actually a feature of object-oriented programming, but most O-O languages don't worry about it because they don't require strict typing.



List[Double] // Type Constructor

is List[Double] a sub type of Iterable[Number] // yes (why - concept of Variance?)

can build from is instantiated with types

```
def m (x: X) : Unit = {  
    println(x.a)  
}  
  
m( new Y() ) // works
```



```
def m (y: Y) : Unit = {  
    println(y.b)  
}  
  
m ( new X() ) // doesn't work
```

Type Constructors

- How do we define types?
 - Well, they're defined by type expressions, in a similar way to the way that values are defined by value expressions, e.g. `factorial(10)`.
 - We start with, for example, an *Int* and a *List*. To construct an actual concrete type from these two types, we would need a *type constructor*.
 - There is an entire aspect of Scala that we are not going to get into known as *higher-kinded-types* (HKT).
 - But let's just say that the most common sort of type constructor has one or more parametric types:
 - *List[_]* where the *_* can be made concrete by an actual type such as *Int*.
 - But, normally, we give these parametric types names so, for example, *List[T]*.
 - And, when we want a concrete *List*, we construct that type by writing, for example, *List[Int]*.
 - By analogy, just think of an instance constructor that has one or more value parameters, such as: `new String("Hello World")`

Variance: How do these constructed types relate?

- How do we relate these constructed types?
 - For example, if *String* is a sub-type of *CharSequence* (which it is, by the way), then is *List[String]* a sub-type of *List[CharSequence]*?
 - This is what **variance** is all about?
- And, when we say *X* is a sub-type (or sub-class) of *Y*, what does that really mean?

Liskov Substitution Principle

- The subtype requirement:

- Let $\phi(x)$ be a property provable about objects x of type T . Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T .
- Example:

```
trait Vertebrate extends Animal {  
    def vertebra: Int  
}  
  
trait Mammal extends Vertebrate {  
    def vertebra: Int = 33  
    def sound: String  
}  
  
trait Dog extends Mammal {  
    def sound = woof  
    def woof: String  
}  
  
trait Cat extends Mammal {  
    def sound = miao  
    def miao: String  
}
```

- *sound* is a provable property of an instance of *Mammal*. Therefore, since *Dog* is a subtype of *Mammal*, therefore *sound* must be a provable property of an instance of *Dog* or *Cat*.
- *vertebra* is a provable property of the type *Vertebrate*. Therefore since *Mammal* is a subtype of *Vertebrate*, therefore *vertebra* must be a provable property of a *Mammal*; and so, *vertebra* must be a provable property of a *Dog* or *Cat*.

Expressions and Types

- $\text{val } y: Y = ???$ $X >: Y$
 $X: a$
 - $\text{val } x: X = y$ $Y: a, b$
-
- In the second variable declaration, y is of type Y and, according to the **Liskov substitution principle**, the type Y can be any sub-type of X (or X itself).
 - Why? Because all properties of X (a) must be defined by y , otherwise x would be somehow incomplete. Because Y must be a sub-type of X (or X), it follows that Y supports all properties of X , as well as some that X does not support (b).

Functions and Types

- $\text{def } f(z: Z): Y = z$ $X >: Y >: Z$
 - $\text{val } z: Z = ???$ $X: a$
 - $\text{val } x: X = f(z)$ $Y: a, b$
 $Z: a, b, c$
-
- In the method declaration, z is of type Z and, according to the **Liskov substitution principle**, the type Y can be any super-type of Z .
 - Why? Because the properties promised the caller of f , i.e. a and b , must be supported by z . But the type Z might well have other properties (c) not supported by Y (which are of no interest to the caller).
 - As before, Y can be any sub-type of X .

Variance

- In general, we can pass a parameter which is of type *A* to a method/function expecting type *B* provided that *A* is a subtype of *B*.
- And we can return a result of type *B* when a method is declared to return type *A*, again provided that *A* is a subtype of *B*.
- Let's say we have a method which takes, as a parameter, *List[Any]*.
- What if what we've actually got is a *List[Int]*?
- These are actually two different types!
- But *Int* is a subtype of *Any* so oughtn't *List[Int]* be a sub-type of *List[Any]*?

Invariance

- Suppose we have a type hierarchy where *Dog* is a subtype of *Animal*, and *Chihuahua* is a subtype of *Dog*.
- What about *List[Dog]*? Is this a subtype of *List[Animal]*? That's to say, if a method/function takes a parameter *List[Animal]*, can we pass it a *List[Dog]* and all will be well?
- Well, for our *List*, this is NOT OK, because *List[A]* is **invariant** in A*.
- *ListBuffer[A]* and *Array[A]* are **invariant**. so you cannot pass an instance of *Array[Dog]* where it expects an *Array[Animal]*.

* but don't worry, the real list is not invariant

Parametric List - part two

- In part one, we saw a possible set of operations on *List[A]*. Very much like an *Array*, in fact. Now, let's think about a *covariant* list.

```
package edu.neu.coe.scala
package list

trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A] (head: A, tail: List[A]) extends List[A]

object List {
    def sum(ints: List[Int]): Int = ints match {
        case Nil => 0
        case Cons(x, xs) => x + sum(xs)
    }

    def apply[A](as: A*): List[A] =
        if (as.isEmpty) Nil
        else Cons(as.head, apply(as.tail: _*))
}
```

By preceding the polymorphic type by “+” we say that it is *covariant*.

as before, returns the sum but works only for *Int* (or, possibly, any type *A* that also defines the “+” operator).

Covariance

- Suppose we again have a type hierarchy where *Dog* is a subtype of *Animal*, and *Chihuahua* is a subtype of *Dog*.
- What about *List[Dog]*? Is this a subtype of *List[Animal]*? That's to say, if a method/function takes a parameter *List[Animal]*, we really ought to be able to pass it a *List[Dog]*.
- Well, for *List*, this is OK, because *List[+A]* extends *Seq[A]*. That's to say: *List* is **covariant** on *A*.
- Technically, *List[+A]* is a **type function** that takes a class *A* and creates a list class such that if *A* is a subtype of *B*, then *List[A]* will be a subtype of *List[B]*.

Covariant and Contravariant positions

- Covariant position:
 - The result of a method;
 - Covariant types may only appear as the result of a method.
- Contravariant position:
 - Any parameter of a method;
 - Contravariant types may only appear as a parameter of a method.
- Invariance:
 - An invariant type may appear in any position.

List – actual definition

```
sealed abstract class List[+A] extends AbstractSeq[A]
with LinearSeq[A] with LinearSeqOps[A, List, List[A]]
with StrictOptimizedLinearSeqOps[A, List, List[A]]
with StrictOptimizedSeqOps[A, List, List[A]]
with IterableFactoryDefaults[A, List] with DefaultSerializable
```

```
/** Adds an element at the beginning of this list.
 *  @param x the element to prepend.
 *  @return a list which contains `x` as first element and
 *          which continues with this list.
 *
 *  Example:
 *  {{{1 :: List(2, 3) = List(2, 3).:::(1) = List(1, 2, 3)}}}
 */
def :::[B >: A] (x: B): List[B] = new scala.collection.immutable.:::(x, this)
```



Values of type **A** can only appear in *covariant position*, that's to say as the result of a method (not as a parameter). So, here we must create a new parametric type **B** which is a super-class of **A**. Parameters are in *contravariant position*.

Collections: detail from API

- trait Seq[+A] extends [Iterable\[A\]](#) with [PartialFunction\[Int, A\]](#) with [SeqOps\[A, Seq, Seq\[A\]\]](#) with [IterableFactoryDefaults\[A, Seq\]](#) with [Equals](#)
- trait Map[K, +V] extends [Iterable\[\(K, V\)\]](#) with [MapOps\[K, V, Map, Map\[K, V\]\]](#) with [MapFactoryDefaults\[K, V, Map, Iterable\]](#) with [Equals](#)
- trait Iterable[+A] extends [IterableOnce\[A\]](#) with [IterableOps\[A, Iterable, Iterable\[A\]\]](#) with [IterableFactoryDefaults\[A, Iterable\]](#)
- trait IterableOps[+A, +CC[_], +C] extends [IterableOnce\[A\]](#) with [IterableOnceOps\[A, CC, C\]](#)
- This all looks a bit weird, right? But, unlike in Java, every different behavior (method) is defined in a separate trait.
- In Scala, there is no wildcard (?) as in Java generics, not as such.

Pets

```
trait Base { val name: String }

trait Organelle

trait Organism { def genotype: Seq[Base] }

trait Eukaryote extends Organism { def organelles: Seq[Organelle] }

trait Animal extends Eukaryote { def female: Boolean; def organelles: Seq[Organelle] = Nil }

trait Vertebrate extends Animal { def vertebra: Int; def sound: Sound }

trait Sound { def sound: Seq[Byte] }

trait Voice extends Sound with ((() => String) { def sound: Seq[Byte] = apply().getBytes } )

trait Bear extends Mammal { def sound: Sound = Growl; def growl: String }

case object Woof extends Voice { def apply(): String = "Woof!" }

case object Growl extends Sound { def sound: Seq[Byte] = "growl".getBytes }

trait Mammal extends Vertebrate {
  def vertebra: Int = 33
}

trait Pet extends Animal {
  def name: String
}

trait Dog extends Mammal with Pet {
  def sound: Sound = Woof
  def genotype: Seq[Base] = Nil
}

case class Chihuahua(name: String, female: Boolean, color: String) extends Dog

case class Pets[+X <: Pet with Mammal, -Y <: Sound](xs: Seq[X]) {
  def identify(s: String): X = xs find (_.name == s) get
  def sounders(y: Y): Seq[X] = xs filter (_.sound == y)
}
```

Types and variance in practice

```
object Pets extends App {  
    def create[X <: Pet with Mammal, Y <: Sound](xs: X*): Pets[X, Y] = Pets(xs)  
    // This method takes a Chihuahua and returns it as a Dog which works because Chihuahua is a  
    // subtype of Dog.  
    // All of the required properties of Dog are specified by any instance of Chihuahua  
    def asDog(x: Chihuahua): Dog = x  
    val bentley = Chihuahua("Bentley", female = false, "black")  
    val gingerSnap = Chihuahua("GingerSnap", female = true, "ginger")  
    val ralphie = Chihuahua("Ralphie", female = true, "white")  
    // List[Chihuahua] is a subtype of Seq[Dog] because A is covariant in Seq[A] and because  
    // List is a subtype of Seq  
    val dogs: Seq[Dog] = List(bentley, gingerSnap, ralphie)  
    // Pets[Chihuahua, Sound] is a subtype of Pets[Dog, Voice] because Chihuahua is a subtype of  
    // Dog (and covariant)  
    // while Sound is a supertype of Voice (and contravariant)  
    val pets: Pets[Dog, Voice] = Pets.create[Chihuahua, Sound](bentley, gingerSnap, ralphie)  
    // Dog is a subtype of Mammal: all of the required properties of Mammal are specified by  
    // any instance of Dog  
    val m: Mammal = asDog(bentley)  
    val ps = pets.sounders(Woof)  
}
```

Variance explained

- First, let's define an arbitrary type with both covariant and contravariant parametric types:
 - $X[+S, -T]$
 - For example, we could declare:
 - *type $X[+S, -T] = (T) \Rightarrow S$*
- Now, what's the relationship between any two X types?
 - $X1[S1, T1]$ is a sub-type of $X2[S2, T2]$ provided that:
 - $X1$ is a sub-type of $X2$
 - $S1$ is a sub-type of $S2$
 - $T1$ is a super-type of $T2$

Summarizing variance

- In the following, S is a sub-type of T
- Invariance, e.g. for *Array*:
 - if we expect an $\text{Array}[T]$, we **cannot** give an $\text{Array}[S]$ because $\text{Array}[S]$ is **not** a sub-type of $\text{Array}[T]$
- Covariance, e.g. for *abstract class List[+A]*:
 - if we expect a $\text{List}[T]$, we **can** give a $\text{List}[S]$ because $\text{List}[S]$ is a sub-type of $\text{List}[T]$
- Contravariance, e.g. for *Function1[-T,+R]*:
 - if we expect a $T \Rightarrow \text{Unit}$, we **cannot** give a $S \Rightarrow \text{Unit}$, but...
 - if we expect a $S \Rightarrow \text{Unit}$, we **can** give a $T \Rightarrow \text{Unit}$ because $T \Rightarrow \text{Unit}$ is a sub-type of $S \Rightarrow \text{Unit}$.

Workarounds

- Suppose that you wish to define a method in `List[+A]` such as *reduce*:

```
def reduce(op: (A, A) => A): A
```

- It can't be done. Do this instead:

```
def reduce[B >: A](op: (B, B) => B): B
```

More on variance

```
trait Function1[-T1, +R] extends AnyRef
```

- “-“ defines T_1 to be contra-variant
- “+“ defines R to be co-variant
- That's to say if T_2 is a super-type of T_1 and if S is a sub-type of R then $Function1[T_2, S]$ is a sub-type of $Function1[T_1, R]$
- Let's say you need a function f which takes an object of type T_1 and transforms it into an object of type R . If you can find a function g that takes an object of type T_2 and transforms it into an object of type S , then you can say $f = g$, that's to say g is a sub-type of f (that's required for assignment).
- For example, we have an $x: CharSequence$ and we want to turn it into a different $y: CharSequence$ by writing $val y = f(x)$.
- We have a function g which takes an Any and turns it into a $String$:
 - `val g: Any=>String = _.toString`
- We can use g for f , that's to say $val f = g$ (where g is a sub-type of f). This works because $T_2=Any$ (a supertype of T_1) and $S=String$ (a subtype of R)

Help with variance

- There are some good resources on the internet to help:
 - [Variances](#) (from Tour of Scala at scala-lang.org)
 - [Covariance and contravariance in Scala](#) (blog)
 - [Covariance and contravariance in Scala](#) (another blog—at Atlassian)
 - [Scala by Example](#) (section 8.2—at scala-lang.org)
 - [Scala's pesky contravariant position problem](#) (my blog)

Updated: 2021-03-31

6.1

Parallel Collections

© 2021 Robin Hillyard



Parallel Collections

- In module 4.8 (Futures), we mentioned parallel collections very briefly.
 - But there are times when they can be useful.
 - Here, we will talk about them in more detail.
 - Note that parallel collections are no longer part of the standard Scala library (as they were in 2.12).
 - You need to include in your *build.sbt*:
`"org.scala-lang.modules" %% "scala-parallel-collections" % "1.0.2"`

Parallel Collections: Documentation

- For all the detail on parallel collections, see
<https://github.com/scala/scala-parallel-collections.git>
- And also the overview: <https://docs.scala-lang.org/overviews/parallel-collections/overview.html>
- For the API, you need to look here:
https://javadoc.io/doc/org.scala-lang.modules/scala-parallel-collections_2.13/latest/scala/collection/index.html

What are the parallel types?

- `ParIterable`:
 - `ParMap`:
 - `immutable.ParMap`,
 - `mutable.ParMap`,
 - `immutable.ParHashMap`,
 - `mutable.ParHashMap`,
 - `ParTrieMap`
 - `ParSeq`:
 - `ParRange`
 - `immutable.ParSeq`,
 - `mutable.ParSeq`,
 - `ParArray`:
 - `ParVector`
 - `ParSet`:
 - `mutable.ParSet`
 - `immutable.ParSet`
 - `mutable.ParHashSet`
 - `immutable.ParHashSet`

Creating parallel collections

- The easiest thing to do is to import the collection conversions and invoke the *par* method on a sequential type to get the following parallel types:

| Sequential | Parallel |
|------------------|------------|
| mutable | |
| Array | ParArray |
| HashMap | ParHashMap |
| HashSet | ParHashSet |
| TrieMap | ParTrieMap |
| immutable | |
| Seq | ParSeq |
| Vector | ParVector |
| Range | ParRange |
| HashMap | ParHashMap |
| HashSet | ParHashSet |

An example (in repo)

```
package edu.neu.coe.csye7200.asstwc.par
import scala.collection.parallel.CollectionConverters._
import scala.collection.parallel.immutable
import scala.language.postfixOps

object Parallel extends App {
    val m = 10
    val n = 10000000
    val expected: BigInt = (BigInt(2) * n * n * n + 3L * n * n + n) / 6
    println(s"Benchmark of sum of squares: N = $n with $m repetitions")
    val xs: List[Int] = LazyList from 1 take n toList
    val ys: List[BigInt] = xs map (x => BigInt(x) * x)
    val zs: immutable.ParSeq[BigInt] = ys.par
    val timeN = benchmark("Non-parallel", m, ys.sum)
    val timeP = benchmark("Parallel", m, zs.sum)
    println(s"speed up with parallelization is by a factor of ${((timeN / timeP - 1) * 100).toInt}%")

def benchmark(message: String, m: Int, z: => BigInt) = {
    val (sum, time) = m times z
    if (sum == expected)
        println(s"$message: average time: $time mSecs")
    else {
        println(s"$message: error: $sum, expected: $expected")
    }
    time
}
```

Results

- On my machine, I get results such as the following:

Benchmark of sum of squares: $N = 10000000$ with 10 repetitions

Non-parallel: average time: 433.1002735 mSecs

Parallel: average time: 129.0844142 mSecs

Speed up with parallelization is by a factor of 235%

Updated: 2023-04-04

6.2

Lenses & Enumerated Types

© 2021-23 Robin Hillyard

A large, semi-transparent watermark of the Northeastern University logo is positioned in the lower right quadrant of the slide. The logo consists of a stylized white 'N' and 'U' intertwined, with the text "Northeastern University" centered below them.

Northeastern
University

Inverted map

- We all know that for a functor $F[A]$, the *map* method will be defined thus:
 - $\text{def } \textit{map}[B](f: A=>B): F[B]$
- But, what if we were to have a method m :
 - $\text{def } m[B](f: B=>A): F[B]$
- What kind of a weird method is this m ? Should we call it *unmap*? And how might we describe that function f ? It's kind of backwards, right?

What is a lens?

- A lens is essentially a functional way of accessing individual fields (members) of an object.
- There are two types of lens function:
 - View: gives access to a particular field;
 - Set: creates a copy of an object, but with a particular field's value changed from the original.
- In Scala, case classes have both types of lens function built-in:

```
case Class Employee(name: String, salary: Int)
val robin: Employee = Employee("Robin", 100000)
val robinSalary: Int = robin.salary
val robinUpdated: Employee = robin.copy(salary = 110000)
```

Comparer

- I have an open-source project called *Comparer*:
 - if you did *lab-sorted* with me, you'll be familiar with the idea).
 - There is a method called *snap* in trait *Comparer[T]*:
 - It's like the *unmap* method we saw before.
 - The $U \Rightarrow T$ function f that it takes as a parameter is a lens function: given a U , it will extract a T .

```
val ic = implicitly[Comparer[Int]]  
val comparerY: Comparer[DateJ] = ic.snap(_.year)  
val comparerM: Comparer[DateJ] = ic.snap(_.month)  
val comparerD: Comparer[DateJ] = ic.snap(_.day)  
val comparer: Comparer[DateJ] = comparerYorElse comparerMorElse comparerD
```

Enumerated Types

Enums

- Enums are a little more difficult in Scala 2 than in Java (although Java enums certainly have plenty issues);
- There are essentially two ways to create enums:
 - case objects
 - extending *Enumeration*
- each has some advantages/disadvantages:
 - For detailed information see: [StackOverflow](#)

Cards by enumeration

```
object Rank extends Enumeration {
    type Rank = Value
    val Deuce, Trey, Four, Five, Six, Seven, Eight, Nine, Ten, Knave, Queen, King, Ace = Value
    class RankValue(rank: Value) {
        def isSpot = !isHonor
        def isHonor = rank match {
            case Ace | King | Queen | Knave | Ten => true
            case _ => false
        }
    }
    implicit def value2RankValue(rank: Value) = new RankValue(rank)
}
object Suit extends Enumeration {
    type Suit = Value
    val Clubs, Diamonds, Hearts, Spades = Value
    class SuitValue(suit: Value) {
        def isRed = !isBlack
        def isBlack = suit match {
            case Clubs | Spades => true
            case _                => false
        }
    }
    implicit def value2SuitValue(suit: Value) = new SuitValue(suit)
}
import Rank._
import Suit._
case class Card (rank: Rank, suit: Suit)
```

Cards by case object (1)

```
trait Concept extends Ordered[Concept]{
    val name: String
    val priority: Int
    override def toString = name
    def compare(that: Concept) = priority - that.priority
    def initial: String = name.substring(0,1)
}
sealed trait Rank extends Concept {
    def isHonor = priority > 7
    def isSpot = !isHonor
}
sealed trait Suit extends Concept {
    def isRed = !isBlack
    def isBlack = this match {
        case Spades | Clubs => true
        case _ => false
    }
}
object Concept {
    // This ordering gives the expected rank and suit (in bridge) order, at least for games where A
    // is considered to outrank K.
    implicit def ordering[A <: Concept]: Ordering[A] = Ordering.by(_.priority)
    // This ordering is for the traditional ordering for displaying bridge hands
    def reverseOrdering[A <: Concept]: Ordering[A] = ordering.reverse
}
```

Cards by case object (2)

```
case object Ace extends Rank {val name = "Ace"; val priority = 12 }
case object King extends Rank {val name = "King"; val priority = 11 }
case object Queen extends Rank {val name = "Queen"; val priority = 10 }
case object Knave extends Rank {val name = "Knave"; val priority = 9; override def initial = "J"}
case object Ten extends Rank {val name = "10"; val priority = 8; override def initial = "T"}
case object Nine extends Rank {val name = "9"; val priority = 7}
case object Eight extends Rank {val name = "8"; val priority = 6}
case object Seven extends Rank {val name = "7"; val priority = 5}
case object Six extends Rank {val name = "6"; val priority = 4}
case object Five extends Rank {val name = "5"; val priority = 3}
case object Four extends Rank {val name = "4"; val priority = 2}
case object Trey extends Rank {val name = "3"; val priority = 1}
case object Deuce extends Rank {val name = "2"; val priority = 0}
case object Spades extends Suit { val name = "Spades"; val priority = 3 }
case object Hearts extends Suit { val name = "Hearts"; val priority = 2 }
case object Diamonds extends Suit { val name = "Diamonds"; val priority = 1 }
case object Clubs extends Suit { val name = "Clubs"; val priority = 0 }
case class Card(suit: Suit, rank: Rank) extends Ordered[Card] {
    val bridgeStyle = true // as opposed to poker-style
    private def nameTuple = (suit.initial,rank.initial)
    override def toString = if (bridgeStyle) nameTuple.toString else nameTuple.swap.toString
    def compare(that: Card): Int = implicitly[Ordering[(Suit, Rank)]].compare(Card.unapply(this).get,
Card.unapply(that).get)
}
object Cards extends App {
    println(List(Card(Clubs,Deuce),Card(Clubs,King),Card(Clubs,Ten),Card(Spades,Deuce)).sorted)
}
```

More information

- [The curious incident...](#)
- [Yuri's blog](#)
- [Dotty enums](#)

Updated: 2021-08-31

6.3

Numeric Programming

© 2021 Robin Hillyard



Numbers

- What's the first property of numbers that we should care about?
 - Order. That's why *Numeric[T]* extends *Ordering[T]*
 - *Numeric[T]* supports most of the simple operators, except for *div*
 - *Fractional[T]* supports *div*
 - Our *Rational* type extends *Fractional[Rational]*.
- But, to be honest, the Scala numbers aren't a huge improvement on Java. Most things are still done with *Double*. But *Double* is imprecise. Stuff like 2.9999999999 are just silly, right?
 - An alternative is Spire. <https://github.com/non/spire>
 - Both exact and inexact types. Example of exact type is *Real*. This will always be as precise as it can possibly be.



Mathologer video: this is just as valid a representation of 3

Using Numeric

- *Numeric[T]* is a trait / type-class and extends *Ordering[T]*
 - Methods:
 - *compare(x: T, y: T): Int*
 - *fromInt(x: Int): T*
 - *plus(x: T, y: T): T*
 - *toDouble(x: T): Double*
 - *zero: T = fromInt(0)*
 - etc.
 - The *sum* method in *IterableOnce[A]* is defined:

```
def sum[B >: A](implicit num: Numeric[B]): B = foldLeft(num.zero)(num.plus)
```
- *Numeric* now has a method:
 - *parseString(s: String): Option[T]*

Using *Fractional*

- *Fractional[T]* is a trait / type-class and extends *Numeric[T]*
- Methods:
 - $\text{div}(x: T, y: T): T$
 - etc.
- Standard imports:

```
trait DoubleIsFractional extends DoubleIsConflicted with Fractional[Double] {  
    def div(x: Double, y: Double): Double = x / y  
}
```

Numeric Parsing

- Our previous parser worked fine but it had the following issues:
 - If output doesn't parse correctly, we will get an appropriate message. But, if anything goes wrong in the logic behind the parsers (e.g. division by zero) an exception will be thrown
 - Output type is fixed as *Double* which is not a very good type
 - Let's try to improve it...

Improving our arithmetic expression parser

- How about we return $\text{Try}[T]$ from each parser (generalizing the output type and wrapping it in Try)?
 - originally we had:

```
class Arith extends JavaTokenParsers {  
    trait Expression {  
        def eval: Double  
    }  
    abstract class Factor extends Expression  
    case class Expr(t: Term, ts: List[String~Term]) extends Expression {  
        def term(t: String~Term): Double = t match {case "+"~x => x.eval; case "-"~x => -x.eval}  
        def eval = ts.foldLeft(t.eval)(_ + term(_))  
    }  
}
```

- so now, let's do this (we also change its name to ExpressionParser):

```
abstract class ExpressionParser[T] extends JavaTokenParsers with (String => Try[T]) { self =>  
    def apply(s: String): Try[T]  
    def negate: (T)=>T  
    def plus: (T,T)=>T  
    trait Expression {  
        def value: Try[T]  
    }  
    ...  
}
```

Improving (2)

- so we need:

```
def negate: (T)=>T = ???  
def plus: (T,T)=>T = ???  
  
case class Expr(t: Term, ts: List[String~Term]) extends Expression {  
    def termVal(t: String~Term): Try[T] = t match {case "+"~x => x.value; case "-"~x =>  
negate(x.value)}  
    def value = ts.foldLeft(t.value)((a,x) => plus(a,termVal(x)))  
}
```

- and therefore we need:

```
def lift(t: Try[T])(f: (T) => T): Try[T] = ???  
def map2(t1: Try[T], t2: Try[T])(f: (T,T) => T): Try[T] = ???  
  
case class Expr(t: Term, ts: List[String~Term]) extends Expression {  
    def termVal(t: String~Term): Try[T] = t match {case "+"~x => x.value; case "-"~x =>  
lift(x.value)(negate)}  
    def value = ts.foldLeft(t.value)((a,x) => map2(a,termVal(x))(plus))  
}
```

- how to implement *lift* and *map2*? The usual way!

```
def lift(t: Try[T])(f: (T) => T): Try[T] = t map f  
def map2(t1: Try[T], t2: Try[T])(f: (T,T) => T): Try[T] = for { tt1 <- t1 ; tt2 <- t2 }  
yield f(tt1,tt2)
```

Using Rational

- Here are examples of using *Rational* and its parser:

```
scala> import edu.neu.coe.scala.parse.RationalExpressionParser
import edu.neu.coe.scala.parse.RationalExpressionParser
scala> val parser = RationalExpressionParser
parser: edu.neu.coe.scala.parse.RationalExpressionParser.type = <function1>
scala> parser.parseAll(parser.expr,"2/3")
res2: parser.ParseResult[parser.Expr] = [1.4] parsed:
Expr(Term(FloatingPoint(2),List((/~FloatingPoint(3)))),List())
scala> res2.get.value
res3: scala.util.Try[edu.neu.coe.scala.numerics.Rational] = Success(Rational(2,3))
scala> import edu.neu.coe.scala.numerics.Rational
import edu.neu.coe.scala.numerics.Rational
scala> import edu.neu.coe.scala.numerics.Rational._
import edu.neu.coe.scala.numerics.Rational._
scala> implicit def convert(x: Int): Rational = Rational.apply(x)
warning: there was one feature warning; re-run with -feature for details
convert: (x: Int)edu.neu.coe.scala.numerics.Rational
scala> implicit def convert(s: String): Rational = Rational.apply(s)
warning: there was one feature warning; re-run with -feature for details
convert: (s: String)edu.neu.coe.scala.numerics.Rational
scala> val l = List[Rational]("1", "2/3", "5")
l: List[edu.neu.coe.scala.numerics.Rational] = List(Rational(1,1), Rational(2,3), Rational(5,1))
scala> val (x,y) = (Rational(2,3),Rational(4,5))
x: edu.neu.coe.scala.numerics.Rational = Rational(2,3)
y: edu.neu.coe.scala.numerics.Rational = Rational(4,5)
```

Sorting on Rational

- And now, let's try to create a list of Rationals and sort it.

```
scala> val l = List[Rational]("1", "2/3", "5")
l: List[edu.neu.coe.scala.numerics.Rational] = List(Rational(1,1), Rational(2,3),
Rational(5,1))
scala> l.sorted
res3: List[edu.neu.coe.scala.numerics.Rational] = List(Rational(2,3), Rational(1,1),
Rational(5,1))
```

Some observations on precision

- Scientific/Engineering Observations
 - You observe some natural phenomenon, such as the period of oscillation of a pendulum.
 - Let's say your observations are (in seconds):
 - 6.4, 6.3, 6.5, 6.2, 6.4, 6.3, 6.5, 6.4, 6.2, 6.3
 - These are different! Does that mean that nine of them are wrong and one is right? Of course not!
 - We say that our best estimate of the period is 6.35s with a standard deviation of 0.1. That means we believe of the true period T that:
 - $6.25 < T < 6.45$ with 68% confidence; and
 - $6.15 < T < 6.55$ with 95% confidence.
 - We write this scientifically as $T = 6.35(10)$ seconds.

Precision, continued

- Let's say that what we really want to know is the length of the pendulum:
 - $l = g T^2 / 4 \pi^2$
 - So, we calculate its value as 10.02 m.
 - But wait a moment! Our estimate of T wasn't known to great precision. Come to think of it, we don't know g very precisely either (we do know π pretty well!)
 - So, what's the precision of l ?
 - T^2 is known: $40.32(20)$
 - g is known: $9.805(10)$
 - In general, if $f = f(x,y)$ then $\Delta f = \delta f/\delta x \Delta x + \delta f/\delta y \Delta y$
 - We can simply add the relative errors to get 0.51%
 - Our estimate of l is therefore: $10.020(51)$

Note that we doubled
the standard deviation
(0.5% relative error
bound)

(0.1% relative error
bound)

Fuzzy

```
trait Fuzzy[+T] {
    /**
     * Get method to return an exact value
     * @return the exact value of this Fuzzy, otherwise an exception is thrown
     * @throws FuzzyException
     */
    def get: T
    /**
     * Return true if this value is exact
     * @return true if exact, else false
     */
    def isExact: Boolean
    /**
     * Return a measure of the probability that this fuzzy instance could be equal to u.
     * If U is a continuous type, we return the probability density function at u.
     * If U is a discrete type, we return the actual probability that this equals u.
     * @param u
     * @tparam U
     * @return the probability that this is equal to u.
     */
    def pdf[U >: T : Discrete](u: U): Double
    /**
     * Map this fuzzy instance into an instance of Fuzzy[U].
     * @param f
     * @tparam U
     * @return
     */
    def map[U >: T : Discrete](f: T=>U)(implicit ev1: (U=>Double)): Fuzzy[U] = flatMap(t => unit(f(t)))
    /**
     * FlatMap this fuzzy instance into an instance of Fuzzy[U].
     * @param f
     * @tparam U
     * @return
     */
    def flatMap[U >: T : Discrete](f: T=>Fuzzy[U]): Fuzzy[U]
    def unit[U >: T : Discrete](u: U)(implicit ev1: (U=>Double)): Fuzzy[U]
    def foreach(f: T=>Unit): Unit
}
```

Libraries

- Spire
- Apache Commons math3 (Java library)
- Number: <https://github.com/rchillyard/Number>

6.4

Updated: 2021-08-31

Type Inference

Making the compiler write your code

© 2021 Robin Hillyard



Northeastern
University

Type Inference

- Type Inference...
 - is what allows the left-hand-side or the right-hand-side of an “=” or “=>” determine the type of an identifier;
 - actually, with =>, it’s only the RHS typically that can infer type.
- So, when we write:

val x = 1

- The compiler is able to deduce that the type of *x* is *Int*.
- Similarly, when we write:

val xs: List[Double] = List(1, 2, 3)

- The compiler is able to deduce that the type of *List* required on the right-hand-side is a *List[Double]*, even though the elements appear to be *Ints*.

Extensions of this idea

- Suppose we define a case class:

case class Complex(real: Double, imag: Double)

- And, further let's suppose that we have method:

def process2[P1, P2, T <: Product](f: (P1, P2) => T): Processor[T]

- Where *Processor[T]* is some trait...

- We can create a processor using the *process2* method, even though we may never actually invoke *f*.
- But the compiler *uses* the actual provided value of *f* to determine the underlying type or the resulting *Processor*.
- And the function *f* will almost always come directly from the *apply* method of the case class's companion object.
- Example: *jsonFormat2* in *Poet.scala*

Let's look at an example

```
def comparer2[P0:Comparer, P1:Comparer, T<:Product](f: (P0,P1)=>T): Comparer[T]
  = comparer[T, P0](0) orElse comparer[T, P1](1)
```

- This method *comparer2* builds a *Comparer[T]* where *T* is a case class or tuple (because it is constrained to be a sub-class of *Product*) with **two** parameters, of types *P0* and *P1*.
- The resulting *Comparer[T]* acts by first comparing the 0th (i.e. first) parameters of any two *T* objects, and if they are the same, it will compare their 1st (i.e. second) parameters.
- How does it know how to compare the *P0* values or the *P1* values?
- Because of the context bounds specifying that there must be an implicit parameter of *comparer2* of type *Comparer[P0]* and another of type *Comparer[P1]*.
- In this example, the method *f* is never invoked at all! It is only there for type inference.

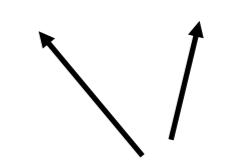
Example of use

```
case class Composite(i: Int, s: String)
object MyComparers extends Comparers {
    val comparer: Comparer[Composite] = comparer2(Composite.apply)
}
import MyComparers._
val c1a = Composite(1, "a")
val c2a = Composite(2, "a")
val c1z = Composite(1, "z")
comparer(c1z)(c1a) shouldBe Less
comparer(c2a)(c1a) shouldBe Less
comparer(c2a)(c1z) shouldBe Less
comparer(c1a)(c1a) shouldBe Same
comparer(c1a)(c2a) shouldBe More
comparer(c1a)(c1z) shouldBe More
```

- In this specification, we need to write `Composite.apply`, not just `Composite` because there is an explicit companion object to `Composite`. Otherwise, we could drop the “`.apply`” part of the parameter.
- There are implicit `Comparer[Int]` and `Comparer[String]` values defined in the companion object of `Comparer`.

An alternative formulation

```
case class Composite(i: Int, s: String)
object Composite {
  implicit val comparer: Comparer[Composite] = Comparer.same[Composite] :| (_.s) :| (_.i)
}
it should "implement Compare" in {
  // NOTE: this uses the implicit val Composite.comparer
  val c1a = Composite(1, "a")
  val c2a = Composite(2, "a")
  val c1z = Composite(1, "z")
  Compare(c1a, c1z) shouldBe Less
  Compare(c1a, c2a) shouldBe Less
  Compare(c1z, c2a) shouldBe More
  Compare(c1a, c1a) shouldBe Same
  Compare(c2a, c1a) shouldBe More
  Compare(c1z, c1a) shouldBe More
}
```



Lens functions

- In the companion object of *Composite*, we define an implicit *Comparer[Composite]* which compares the *s* field first, then the *i* field.
- Its definition is based on the composition of “lens” functions to build a *Comparer* for a *Composite*.

Updated: 2023-04-04

6.5

Scala etc.

© 2021-23 Robin Hillyard

A large, semi-transparent watermark of the Northeastern University logo is positioned in the lower right quadrant of the slide. The logo consists of a stylized white 'N' and 'U' intertwined, with the words "Northeastern University" written in a serif font below them.

Northeastern
University

Scala.js

- The JVM is not the only place you can run Scala!
- You can write browser code in Scala by using [Scala.js](#)
- And you can run it native now too! With LLVM

Cats, Scalaz and Shapeless

- [Cats](#) is a *Lightweight, modular, and extensible library for functional programming*
 - Heavy use of type-classes, monads,
 - (239 SO questions—39 in 2018)
- [Scalaz](#): is an (older) adjunct functional library for Scala.
 - (940 SO questions—12 in 2018)
- [Shapeless](#): ditto
 - (733 SO questions—19 in 2018)

TableParser: use of IO

```
import CrimeParser._  
import cats.effect.unsafe.implicits.global  
implicit val random: Random = new Random()  
val filename = "tmp/Crime.use.Resource.csv"  
val wi: IO[Unit] = for {  
    url <- IO.fromTry(FP.resource[Crime]("2023-01-metropolitan-street-sample.csv"))  
    readResource = Resource.make(IO(Source.fromURL(url)))(src => IO(src.close()))  
    writeResource = Resource.make(IO(new FileWriter(filename)))(fw => IO(fw.close()))  
    ct <- readResource.use(src => Table.parseSource(src))  
    lt <- IO(ct.mapOptional(m => m.brief))  
    st <- IO(lt.filter(FP.sampler(10)))  
    w <- st.toCSV  
    _ <- writeResource.use(fw => IO(fw.write(w)))  
} yield ()  
  
wi.unsafeRunSync()
```

Other stuff

- *Figaro*—Probabilistic programming
- *LaScala*—my random collection of library methods

Updated: 2021-03-31

6.6 Software Engineering

© 2021 Robin Hillyard



Some topics—but we will only touch a few of these

- Teamwork:
 - Project planning
 - Requirements (use cases)
 - Agile development
 - Acceptance Criteria
 - Source-code management
 - Practical dependency management
 - Technical Debt
 - Peer reviews/programming
 - Issue Tracking (JIRA, etc.)
- Eco-systems:
 - Platforms (JDK vs. .Net, etc.)
 - O/S (Windows, Unix, ...)
 - Open source vs. proprietary
 - Licensing
 - Messaging/streaming: pub/sub
 - Persistence
- Complexity
 - Managing dependencies, coupling, etc.
 - Optimization: performance vs. memory
 - Code analysis
 - Privacy, security, cryptography
 - Big data/parallel programming/concurrent programming

Agile Development

The *Agile Manifesto* is based on twelve principles:

1. **Customer satisfaction by early/continuous delivery of valuable software**
2. **Welcome changing requirements, even in late development**
3. Working software is delivered frequently (weeks rather than months)
4. Close, daily cooperation between business people and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. **Working software is the principal measure of progress**
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design
10. Simplicity—the art of maximizing the amount of work not done—is essential
11. **Best architectures, requirements, and designs emerge from self-organizing teams**
12. Regularly, the team reflects on how to become more effective, and adjusts accordingly

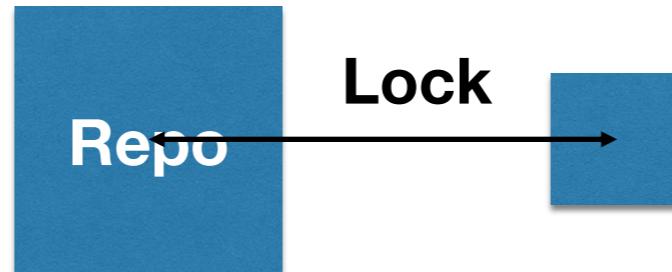
Agile is built on...

- User Stories (c.f. requirements):
 - Who, What, Why?
 - As a *<role>*, I want *<goal>* so that *<benefit>*.
- Iterative Development:
 - Sprints
 - Scrum
 - Velocity
- Continuous integration:
 - User-centric source control (git, Mercurial)
 - **Unit testing** (without unit testing, CI cannot work!!!!)
 - Continuous integration server, e.g. CI, Jenkins

Source Code Management

- “Old” SCM:

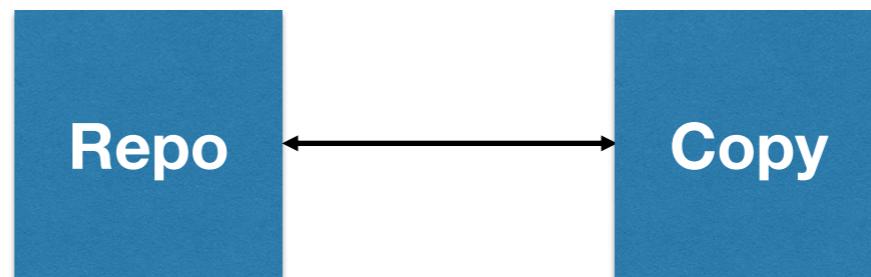
- Subversion
- CVS



User downloads only those files to be edited

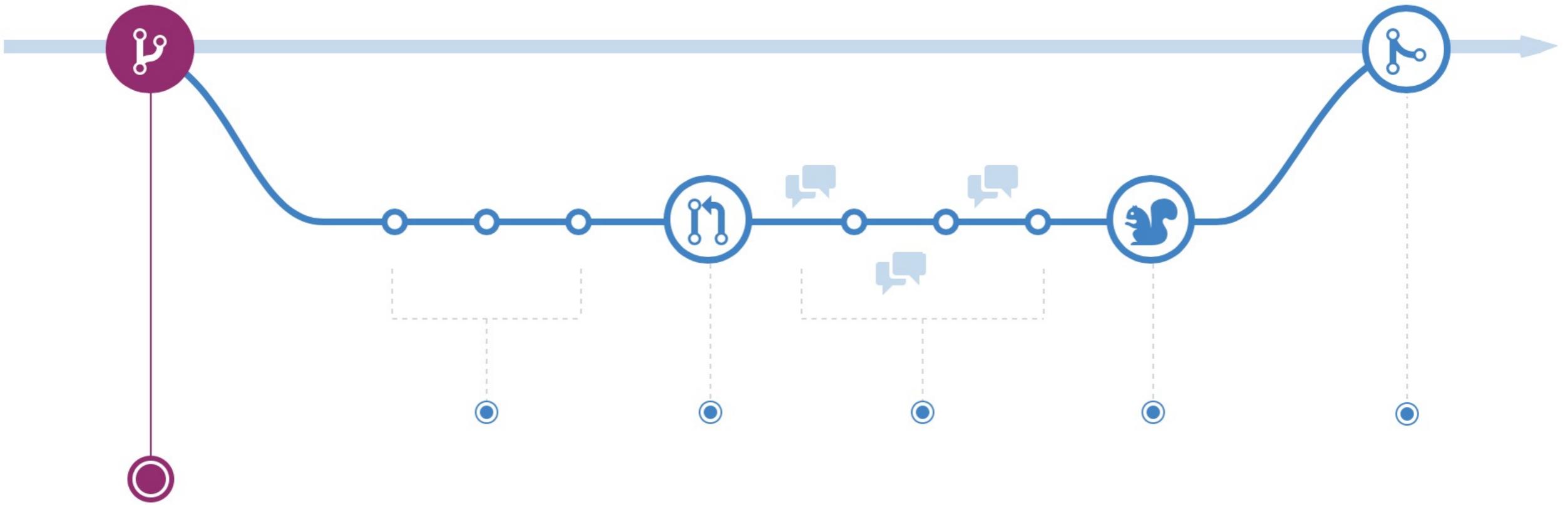
- User-centric SCM:

- Git
- Mercurial



User downloads entire repository

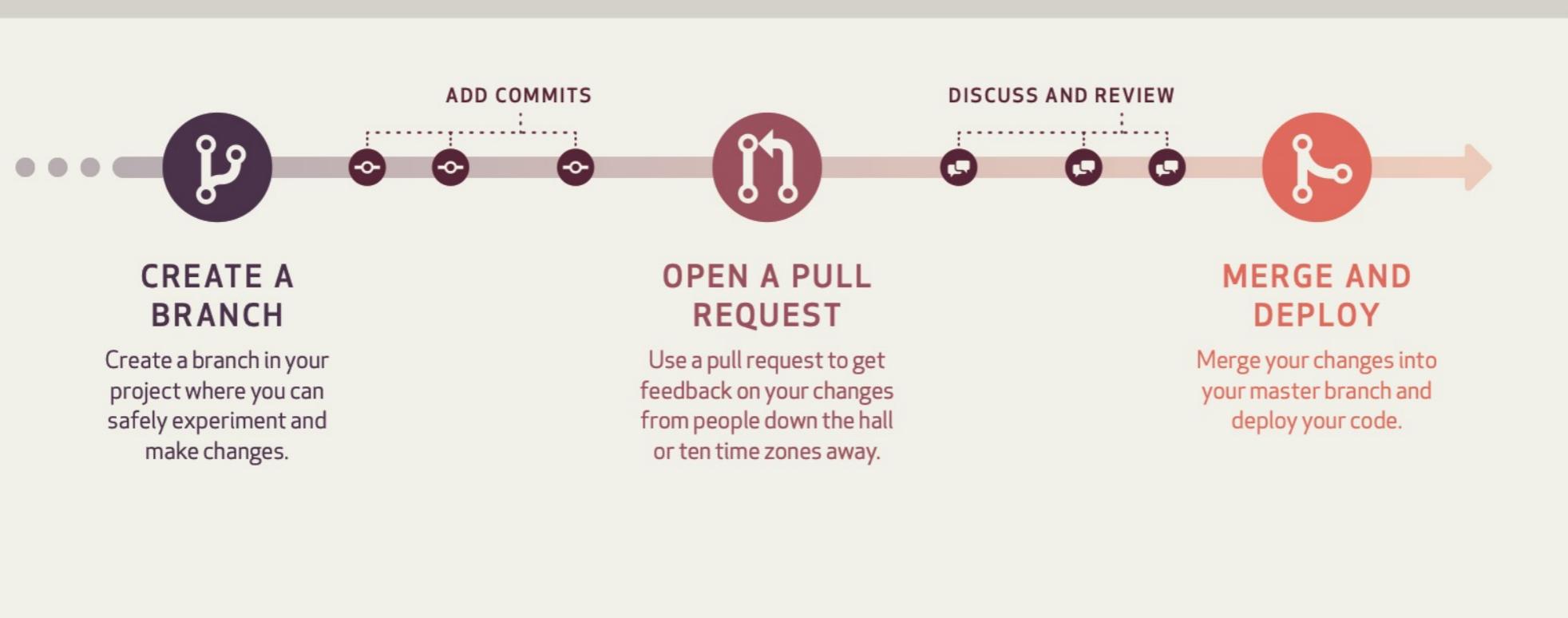
Git workflow



- [GitHub Flow](#)
 - Development branches
 - Pull Requests

WORK FAST WORK SMART **THE GITHUB FLOW**

The GitHub Flow is a lightweight, branch-based workflow that's great for teams and projects with regular deployments. Find this and other guides at <http://guides.github.com/>.



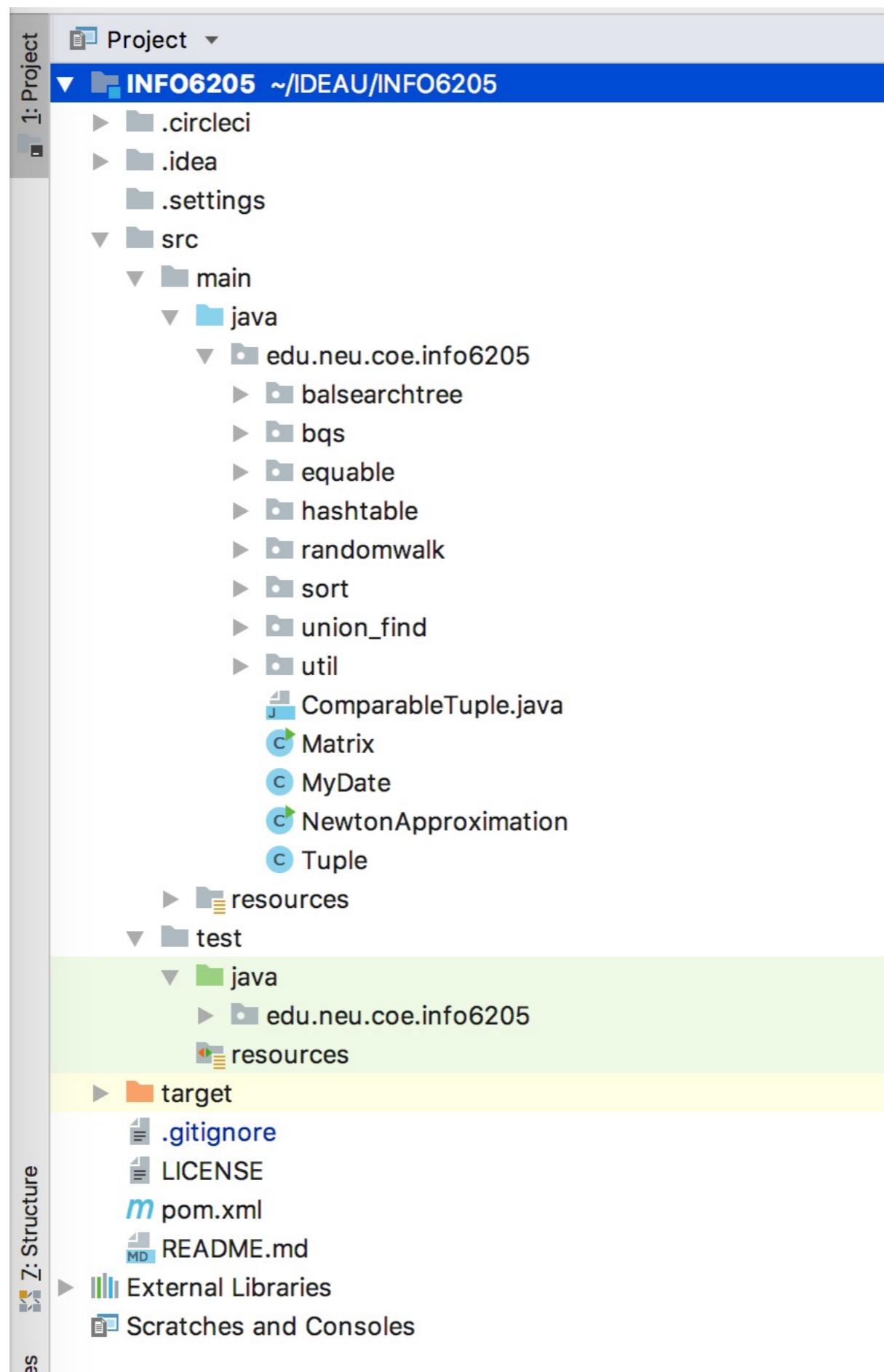
GitHub is the best way to build software together.

GitHub provides tools for easier collaboration and code sharing from any device. Start collaborating with millions of developers today!

Dependencies

- The days of building entire applications with all your own software are long gone
 - (yes, those days did exist!)
- Dependencies must be “managed” by something.
 - *Maven* is both a dependency management system and a build system; q.v. *Ivy*, *Gradle*, etc.
 - *sbt* (for Scala) is a build system which uses *Ivy* underneath
 - Dependencies are retrieved from an artifact repository (“artifactory”) which is either “central” or local (company, department, etc.).
 - The actual dependencies (with group/artifact/version details) are specified in a *POM* file (“Project Object Model”) or its equivalent.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>sf.net.darwin</groupId>
  <artifactId>darwin</artifactId>
  <version>3.0.0-SNAPSHOT</version>
  <name>Darwin API</name>
  <description>Darwin Parent Project</description>
  <scm>
    <developerConnection>scm:svn:http://darwin.svn.sourceforge.net/svnroot/darwin</developerConnection>
    <url>http://darwin.svn.sourceforge.net/svnroot/darwin</url>
  </scm>
  <organization>
    <name>Rubecula Software, LLC</name>
    <url>http://www.rubecula.com/</url>
  </organization>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>buildnumber-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <phase>validate</phase>
            <goals>
              <goal>create</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <doCheck>false</doCheck>
          <doUpdate>false</doUpdate>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>2.2.1</version>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
          <finalName>${project.build.finalName}</finalName>
          <archive>
            <manifest>
              <addClasspath>true</addClasspath>
              <mainClass>net.sf.darwin.platform.main.DarwinPlatform</mainClass>
            </manifest>
          </archive>
        </configuration>
        <executions>
          <execution>
            <id>make-assembly</id> <!-- this is used for inheritance merges -->
            <phase>package</phase> <!-- bind to the packaging phase -->
            <goals>
              <goal>single</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  <packaging>pom</packaging>
</project>
```



Merging

- It happens sooner or later: you and another developer have changed the same module.
- One of you will have to resolve the conflicts. Your IDE or git client will help with this by presenting a three-way merge:
 - *common-ancestor—theirs—yours.*
- Unfortunately, these merge clients are still (after all these years!!!) line-oriented rather than token/block-oriented.
- Nevertheless, you should be able to resolve the conflict unless each of you has essentially changed the same part of the code in different ways.

What goes into source control?

- Put everything that you need—and nothing that you don't need—into the repository.
 - Guidelines for any (source) item:
 - represents IP (intellectual property—requires brainpower to create);
 - cannot be trivially derived from other sources;
 - experience versioning (typically);
 - relatively small (e.g. do not commit your entire data source(s) to git);
 - host-independent (builds should work on any machine);
 - usually in ASCII or Unicode or some other alphanumeric, human-readable encoding;
 - does not pertain to any particular IDE or is removed by clean:
 - files ending in, e.g., `.class`, `.jar`, `.classpath`, `.iml`; `.idea` plus anything under `project` or `target` should be ignored (using `.gitignore` assuming you're using git) and not be placed in the repo.

What other project models are there?

- The primary alternative to agile is “Waterfall”
 - This was the primary project management methodology for many years. It doesn’t work (never did!)
 - Why doesn’t it work?
 - Time lag, hysteresis induced by:
 - the development/testing cycle
 - changing requirements
 - Reality (*The Mythical Man Month*)
 - Why was it used? They didn’t know any better—no unit testing.

Testing: Unit, integration and performance specifications

Our style of development

- What do you notice about the way you are being asked to do the assignments?
- We are essentially using a particular development style. Do you know what it's called?
 - Hint: it has a TLA (three-letter acronym)

Why do we create specifications/unit tests?

- You can prove a *fragment* of code...
 - ...but it's impractical to prove an entire code base...
 - ...so, we aim for code coverage:
 - One of the ways of maximizing coverage is by using random inputs (that's why the random number generator is important)—*property-based testing*.
- Unit tests prevent you from making errors in:
 - bug fixing
 - feature implementation
 - re-factoring
- Unit tests serve as *documentation* (better than comments) on the behavior of software
- Integration tests help find:
 - sloppy APIs, contracts, etc.
 - unacceptable performance

Testing

- Unit Testing is one of the most important developments in software engineering
- Yet, it is still not done properly!!
 - When you go on an interview, ask *them* about their unit testing practices
- Writing unit tests:
 - Test-driven development
 - Automated test writing
 - Mocking
 - Specifications and other styles
 - JUnit vs. Scalatest, etc.
- Coverage
- Functional testing

Property-based Testing

- There is another type of test specification called **property-based testing**.
 - This can save you a lot of time dreaming up suitable input values: a property-based tester will generate random values as well as corner cases.
 - In Scala, we typically use *Scalacheck* for this.

```
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll

class RationalPropertySpec extends Properties("String") {

    import Rational.RationalHelper

    property("RationalFromString") = forAll { (a: Int, b: Int) =>
        val rat = r"$a/$b"
        (rat*b).toInt == a
    }
}
```

Integration Testing

- What's the difference between unit testing and integration testing?
 - A unit test is designed to test *exactly* one thing, no more, no less—an integration test is designed to test an entire module, or even an entire application.
 - A unit test runs instantaneously—an integration test may take some significant time to run.
 - A unit test, ideally, does not use mocking—integration tests use mocking a lot.
 - Unit tests are found in the “test” folder under “src” in a maven/sbt project—integration tests are found somewhere else, typically under “it.”
 - Unit tests are run (or should be) whenever you change code—integration tests are typically run only when you are ready to commit.

Mocking

Then the [Queen](#) left off, quite out of breath, and said to [Alice](#), "Have you seen the Mock Turtle yet?"

"No," said Alice. "*I don't even know what a Mock Turtle is.*"

"It's the thing Mock Turtle Soup is made from", said the [Queen](#).

— *Alice in Wonderland*, chapter 9



What is mocking?

- Let's say for example that you have developed a set of DAO classes—
 - in order to test them, you have to connect to a database;
 - this may be an expensive (slow) operation;
 - the database that you connect to may have changed since you last ran the test;
 - the database may not even be built yet—it may be under development as part of your overall project—or promised by some third-party developer.
- For these reasons, you “mock” the database:
 - The most obvious way is simply to define a mock class in your test module: typically, you will be extending a trait or abstract class and defining your own methods;
 - But if you want to do it “properly” (i.e. with the least inconvenience to yourself), use a mocker:
 - *scalamock*, although you can use Java test runners like *mockito*, etc.

Mock Example

- The following is from [Majabigwaduce](#):

```
class CountWordsSpec extends FlatSpec with Matchers with Futures with ScalaFutures with Inside
with MockFactory {
  "CountWords" should "work for http://www.bbc.com/ http://www.cnn.com/ http://default/" in {
    val wBBC = "http://www.bbc.com/"
    val wCNN = "http://www.cnn.com/"
    val wDef = "http://default/"
    val uBBC = new URI(wBBC)
    val uCNN = new URI(wCNN)
    val uDef = new URI(wDef)
    val hc = mock[HttpClient]
    val rBBC = mock[Resource]
    (rBBC.getServer _).expects().returning(uBBC)
    rBBC.getContent _ expects() returning CountWordsSpec.bbcText
    val rCNN = mock[Resource]
    rCNN.getServer _ expects() returning uCNN
    rCNN.getContent _ expects() returning CountWordsSpec.cnnText
    val rDef = mock[Resource]
    rDef.getServer _ expects() returning uDef
    rDef.getContent _ expects() returning CountWordsSpec.defaultText
    hc.getResource _ expects wBBC returning rBBC
    hc.getResource _ expects wCNN returning rCNN
    hc.getResource _ expects wDef returning rDef
    val nf = CountWords(hc, Array(wBBC, wCNN, wDef))
    whenReady(nf, timeout(Span(6, Seconds)))(i => assert(i == 556))
  }
}
```

Dependencies, versions, coupling, etc.

- Suppose you have a trait $X[T]$ with method $x: T$
 - and a concrete class X_String that extends $X[String]$ that defines a method $x1: String$.
 - Now, you have another class Y where you reference the X_String directly and even its method $x1$.
 - You've introduced *coupling* into your code. Any new version of X_String is liable to require a change to Y .
 - If you are disciplined and access *only* the method x of trait X , you will not be likely to need to change your code very much as traits tend to remain stable (assuming they were well-designed in the first place).

Updated: 2021-03-31

6.7 Advanced Concepts

© 2015 Robin Hillyard



Type Classes

- We looked at these briefly before. Type classes are the way in which Scala programs (Haskell has these too) allow the imposition of behavior on another class
 - In pure O-O, we always have to *extend* traits to add behavior.
 - But that's often impossible, inconvenient, or just plain wrong.

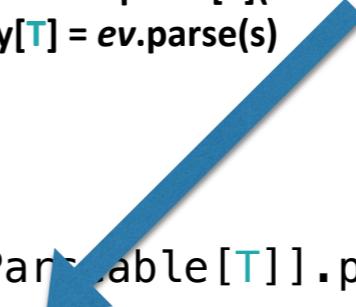
Type classes: review

- Let's say you have in mind a trait but there's nothing appropriate for it to extend:

```
trait Parseable[T] {  
    def parse(s: String): Try[T]  
}  
object Parseable {  
    trait ParseableInt extends Parseable[Int] {  
        def parse(s: String): Try[Int] = Try(s.toInt)  
    }  
    implicit object ParseableInt extends ParseableInt  
}  
    object TestParseable {  
        def parse[T : Parseable](s: String): Try[T] = implicitly[Parseable[T]].parse(s)  
    }
```

This form is called a “context bound”.
But we can also write it as follows:

```
def parse[T](s: String)(implicit ev: Parseable[T]):  
    Try[T] = ev.parse(s)
```



- What we are doing here is adding the behavior of *Parseable* to type *T* without requiring *T* to extend anything.
- Note that you cannot add a context bound to a trait. Why not?

A case in point

- Renderable:
 - In my *LaScala* library, I created a trait *Renderable* which affords a better way to render objects as Strings (better than *toString*).
 - In particular, any type that extends *Renderable* supports the *render* method:
 - Additionally, there are implicit *Renderables* for containers such as:

```
implicit def renderableTraversable(xs: Traversable[_]): Renderable =  
  RenderableTraversable(xs, max = Some(MAX_ELEMENTS))  
  • where:  
  
  case class RenderableTraversable(xs: Traversable[_], bookends: String = "(),", linear:  
    Boolean = false, max: Option[Int] = None) extends Renderable {  
    def render(indent: Int)(implicit tab: (Int) => Prefix): String = {  
      val (p, q, r) =  
        if (linear || xs.size <= 1)  
          ("\" + bookends.head, "\" + bookends.tail.head, "\" + bookends.last)  
        else  
          (bookends.head + nl(indent + 1), bookends.tail.head + nl(indent + 1), nl(indent) +  
            bookends.last)  
      Renderable elemsToString(xs, indent, p, q, r, max)  
    }  
  }
```

This works great but...

- ...there is a bit of a problem:
 - it only works for types which extend *Renderable* or which have been provided for using an implicit;
 - in particular, if you want to write a library class based on a parametric type which is *Renderable*, you have to specify that as a type constraint;
 - instead, you can simply apply a “context bound” to the parametric type requiring “evidence” of a *Renderable*—this is so much more convenient.

Type class *Renderable*

- New *Renderable*:
 - I am in the process of changing *LaScala* to use a type class *Renderable* instead of a polymorphic trait.

```
trait Renderable[A] {  
    /**  
     * Method to render this object in a human-legible manner.  
     *  
     * @param indent the number of "tabs" before output should start (when writing on a new line).  
     * @return a String that, if it has embedded newlines, will follow each newline with (possibly empty) white space,  
     *         which is then followed by some human-legible rendering of *this*.  
     */  
    def render(a: A)(indent: Int = 0): String  
        /**  
         * Method to translate the tab number (i.e. indent) to a String of white space.  
         * Typically (and by default) this will be uniform. But you're free to set up a series of tabs  
         * like on an old typewriter where the spacing is non-uniform.  
         *  
         * @return a String of white space.  
         */  
    def tab(x: Int): Prefix = Prefix(" " * x)  
}
```

- The most obvious differences are that *Renderable* now takes a parametric type *A* and the *render* method itself takes a value of type *A*.

Specification

- Here's part of *RenderableSpec*:

```
class RenderableSpec extends FlatSpec with Matchers with Inside {  
    behavior of "Renderable"  
    import RenderableInstances._  
    it should "render String the hard way" in {  
        import Renderable._  
        render("Hello")() shouldBe "Hello"  
    }  
    it should "render String" in {  
        import RenderableSyntax._  
        "Hello".render shouldBe "Hello"  
    }  
    it should "render Int" in {  
        import RenderableSyntax._  
        1.render shouldBe "1"  
    }  
}
```

- It is slightly annoying to have to specify those imports but there may be a better way.

Specification (part 2)

- Here's another part of *RenderableSpec*:

```
class RenderableSpec extends FlatSpec with Matchers with Inside {  
    behavior of "MockContainer"  
    it should "render correctly" in {  
        val target = MockContainer(Seq(1,2,3))  
        target.toString shouldBe "(\n 1,\n 2,\n 3\n)"  
    }  
}  
  
case class MockContainer[A: Renderable](as: Seq[A]) {  
    import RenderableSyntax._  
    override def toString: String = as.render  
}
```

Using Type classes

- Best Practices:
 - Define the minimum number of methods (often just one!) in your type class trait;
 - Use a type class to add behavior to a parametric type in some other class—don’t use one to add behavior to the other class—be strict about that.
 - If a parametric type needs more than one behavior imposed on it, add additional context bounds:

```
case class MockNumericContainer[A: Renderable: Numeric](as: Seq[A]) {  
    import RenderableSyntax._  
    override def toString: String = as.render  
    def total: A = as.sum  
}
```

Type-class Libraries

- Cats
 - <https://typelevel.org/cats>
 - Cats uses type-classes extensively. There are type-classes for just about everything.

Logging functional style

- Logging styles:

- Libraries such as log4j are ideally suited to a statement-oriented language such as Java...
- ...but Scala is functional, not statement-oriented.
- Why not use a functional style of logging (as in *LaScala*)?

```
trait Spy
object Spy {
  def apply(x: Unit): Spy = new Spy() {}
  lazy private val configuration = ConfigFactory.load()
  var spying: Boolean = configuration.getBoolean("spying")
  def spy[X](message: => String, x: => X, b: Boolean = true)(implicit spyFunc: String => Spy, isEnabledFunc: Spy => Boolean): X = {
    val xy = Try(x) //evaluate x inside Try
    if (b && spying && isEnabledFunc(mySpy)) doSpy(message, xy, b, spyFunc) //if spying is turned on, log an appropriate message
    xy.get //return the X value or throw the appropriate exception
  }
  def log(w: => String, b: Boolean = true)(implicit spyFunc: String => Spy, isEnabledFunc: Spy => Boolean) {spy(w, (), b); ()}
  def noSpy[X](x: => X): X = {
    val safe = spying
    spying = false
    val r = x
    spying = safe
    r
  }
  private val prefix = "spy: "
  val brackets: String = "{}"
  implicit val defaultLogger: Logger = getLogger(getClass)
  implicit def spyFunc(s: String)(implicit logger: Logger): Spy = if (logger != null) Spy(logger.debug(prefix + s)) else Spy()
  implicit def isEnabledFunc(x: Spy)(implicit logger: Logger): Boolean = logger != null && logger.isDebugEnabled
  def getLogger(clazz: Class[_]): Logger = LoggerFactory.getLogger(clazz)
  def getPrintInSpyFunc(ps: PrintStream = System.out): String => Spy = { s => Spy(ps.println(prefix + s)) }
  private def doSpy[X](message: String, xy: => Try[X], b: Boolean, spyFunc: (String) => Spy) = {
    ...
  }
  private def formatMessage[X](x: X, b: Boolean): String = x match {
    case () => "()"
    ...
  }
  // NOTE: If the value to be spied on is Future[_] then we invoke spy on the underlying value when it is completed
  case f: Future[_] =>
    import scala.concurrent.ExecutionContext.Implicits.global
    f.onComplete(spy("Future", _, b))
    "to be provided in the future"
  // NOTE: If the value to be spied on is a common-or-garden object, then we simply form the appropriate string using the toString method
  case _ => if (x != null) x.toString else "<>null<>"
}
private val mySpy = apply()
```

Using Spy

- Here's the Spec file:

```
class SpySpec extends FlatSpec with Matchers {

  behavior of "Spy.spy"

    it should "work with implicit (logger) (with default logger) spy func" in {
      import Spy._

      Spy.spying = true
      (for (i <- 1 to 2) yield Spy.spy("i", i)) shouldBe List(1, 2)
      // you should see log messages written to console (assuming your logging level, i.e. logback-test.xml, is set to DEBUG)
    }
}
```

- Note what it would like look without spying:

```
class SpySpec extends FlatSpec with Matchers {

  behavior of "for comprehension"

    it should "work" in {
      (for (i <- 1 to 2) yield i) shouldBe List(1, 2)
    }
}
```

Another functional logger

```
object Flog {
    implicit class Flogger(message: => String)(implicit logFunc: LogFunction = Flog.loggingFunction) {
        def !![X: Loggable](x: => X): X = Flog.logLoggable(logFunc, message)(x)
        def !|[X](x: => X): X = Flog.logX(logFunc, message)(x)
        def |![X](x: => X): X = x
    }

    var enabled = true
    implicit var loggingFunction: LogFunction = getLogger[Flogger]
    def getLogger[T: ClassTag]: LogFunction = LogFunction(LoggerFactory.getLogger(implicitly[ClassTag[T]].runtimeClass).debug)
    def logLoggable[X: Loggable](logFunc: LogFunction, prefix: => String)(x: => X): X = {
        lazy val xx: X = x
        if (enabled) logFunc(s"log:$prefix:${implicitly[Loggable[X]].toLog(xx)}")
        xx
    }
    def logX[X](logFunc: LogFunction, prefix: => String)(x: => X): X = {
        lazy val xx: X = x
        if (enabled) logFunc(s"log:$prefix:$xx")
        xx
    }
}
```

Updated: 2021-03-31

6.8

Spark Performance Tuning

© 2015-21 Robin Hillyard



Spark Performance Tips

- Know what your Scala code is doing!

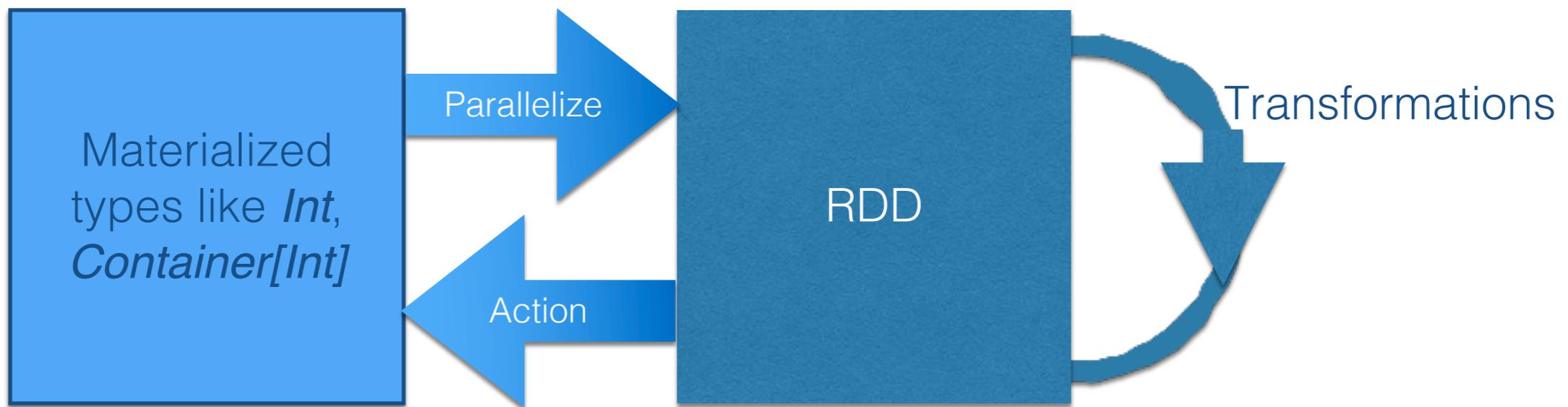
```
def main(args: Array[String]) {  
    if (args.size < 1) {  
        println("Please provide train.csv path as input argument");  
        return;  
    }  
    val data = LoadClaimData.loadData(args(0));  
    val pca = new PCA(60).fit(data.map(_.features))  
    val projected = data.map(p => p.copy(features = pca.transform(p.features)))  
    // Split data into training (60%) and test (40%).  
    val splits = projected.randomSplit(Array(0.6, 0.4), seed = 11L)  
    val training = splits(0).cache()  
    val test = splits(1)  
    // Run training algorithm to build the model  
    val model = new LogisticRegressionWithLBFGS()  
        .setNumClasses(2)  
        .run(training)  
    // Scala class evaluation Get evaluation metrics.  
    val predictionAndLabels2 = test.map {  
        case LabeledPoint(label, features) =>  
            val prediction = model.setThreshold(0.6).predict(features)  
            (prediction, label)  
    }  
    // Precision and Recall using BinaryClassificationMetrics  
    val metrics = new BinaryClassificationMetrics(predictionAndLabels2)  
    println(s"Recall = ${metrics.pr().collect().tail.head._1}")  
    println(s"Precision = ${metrics.pr().collect().tail.head._2}")  
    // Compute raw scores on the test set. clearThreshold() allows for real values in the range of 0 to 1.  
    val predictionAndLabels = test.map {  
        case LabeledPoint(label, features) =>  
            val prediction = model.clearThreshold().predict(features)  
            (prediction, label)  
    }  
    // Calculating logloss error using the real values in the range 0 to 1.  
    val loss = for {  
        pair <- predictionAndLabels  
        error = calculateErrorFactor(pair._1, pair._2)  
    } yield error  
    val logloss = loss.collect().toSeq.reduce(_ + _)  
    println(s"logloss= ${logloss / loss.count * (-1)}")  
}
```

What uses up all the time in Spark?

- Actions!
 - Remember: actions speak louder than transformations—and they use up far more resources
 - Extractions:
 - *collect* (think of this if you like as the opposite of *sc.parallelize*)
 - *foreach*, *saveAsTextFile*, *saveAsObjectFile*.
 - Aggregations—measure an *RDD*:
 - *count*, *aggregate*, *max*, *min*, *reduce*, *treeReduce*, *fold*.
- Transformations (don't take any time: they are lazy!)
 - single *RDDs*:
 - *map*, *flatMap*, *filter*, *distinct*, *sample*, *take*, *drop*, *collect(f)*,
 - single *RDDs* of an appropriate type:
 - *top*, *takeOrdered*, *takeSample*, *countByValue*, *groupByKey*, *sortBy*
 - Combinations—two *RDDs* (of same underlying type):
 - *union*, *intersection*, *subtract*, *cartesian*, *zip*

Working with RDDs (2)

- Because $\text{RDD}[\text{T}]$ is lazy, it's essentially *opaque*



- Once we have parallelized a container as an RDD, we can apply transformations to RDDs, creating new RDDs. Since these of course are lazy, the RDDs are just *decorated*. In order to materialize something from an RDD, we need to apply an “action”.

logging, timing,etc.

- Added a logger, a timer, and type annotations so we can see what we're dealing with:

```
def doClaimsPrediction(filename: String): Unit = {
    val log = Logger.getLogger(getClass.getName)
    val timer1 = Timer.apply
    val data: RDD[LabeledPoint] = LoadClaimData.loadData(filename)
    val pca: PCAModel = new PCA(60).fit(data.map(_.features))
    val projected: RDD[LabeledPoint] = data.map(p => p.copy(features = pca.transform(p.features)))
    // Split data into training (60%) and test (40%).
    val splits: Array[RDD[LabeledPoint]] = projected.randomSplit(Array(0.6, 0.4), seed = 11L)
    val training: RDD[LabeledPoint] = splits(0).cache()
    val test: RDD[LabeledPoint] = splits(1)
    // Run training algorithm to build the model
    val model: LogisticRegressionModel = new LogisticRegressionWithLBFGS()
        .setNumClasses(2)
        .run(training)
    // Scala class evaluation Get evaluation metrics.
    val predictionAndLabels2: RDD[(Double, Double)] = test.map {
        case LabeledPoint(label, features) =>
            val prediction = model.setThreshold(0.6).predict(features)
            (prediction, label)
    }
    val timer2 = timer1.lap("after modeling", log.info(_))
    // Precision and Recall using BinaryClassificationMetrics
    val metrics: BinaryClassificationMetrics = new BinaryClassificationMetrics(predictionAndLabels2)
    log.info(s"Recall = ${metrics.pr().collect().tail.head._1}")
    log.info(s"Precision = ${metrics.pr().collect().tail.head._2}")
    val timer3 = timer2.lap("after metrics", log.info(_))
    // Compute raw scores on the test set. clearThreshold() allows for real values in the range of 0 to 1.
    val predictionAndLabels: RDD[(Double, Double)] = test.map {
        case LabeledPoint(label, features) => (model.clearThreshold().predict(features), label)
    }
    // Calculating logloss error using the real values in the range 0 to 1.
    val loss: RDD[Double] = for (pr <- predictionAndLabels) yield calculateErrorFactor(pr._1, pr._2)
    val logloss = loss.collect().toSeq.reduce(_ + _)
    log.info(s"logloss= ${logloss / loss.count * (-1)}")
    timer3.lap("end", log.info(_))
}
```

Timings:

- lap1: 19.441 (modeling: logistic regression)
- lap2: 2.817 (precision/recall)
- lap3: 5.176 (logloss)

Improving performance

- Do you see any bad code?

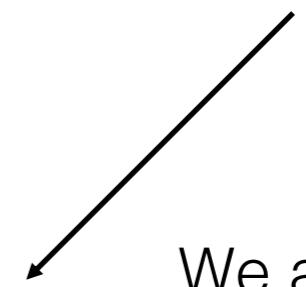
```
def doClaimsPrediction(filename: String): Unit = {
    val log = Logger.getLogger(getClass.getName)
    val timer1 = Timer.apply
    val data: RDD[LabeledPoint] = LoadClaimData.loadData(filename)
    val pca: PCAModel = new PCA(60).fit(data.map(_.features))
    val projected: RDD[LabeledPoint] = data.map(p => p.copy(features = pca.transform(p.features)))
    // Split data into training (60%) and test (40%).
    val splits: Array[RDD[LabeledPoint]] = projected.randomSplit(Array(0.6, 0.4), seed = 11L)
    val training: RDD[LabeledPoint] = splits(0).cache()
    val test: RDD[LabeledPoint] = splits(1)
    // Run training algorithm to build the model
    val model: LogisticRegressionModel = new LogisticRegressionWithLBFGS()
        .setNumClasses(2)
        .run(training)
    // Scala class evaluation Get evaluation metrics.
    val predictionAndLabels2: RDD[(Double, Double)] = test.map {
        case LabeledPoint(label, features) =>
            val prediction = model.setThreshold(0.6).predict(features)
            (prediction, label)
    }
    val timer2 = timer1.lap("after modeling", log.info(_))
    // Precision and Recall using BinaryClassificationMetrics
    val metrics: BinaryClassificationMetrics = new
    BinaryClassificationMetrics(predictionAndLabels2)
    log.info(s"Recall = ${metrics.pr().collect().tail.head._1}")
    log.info(s"Precision = ${metrics.pr().collect().tail.head._2}")
    val timer3 = timer2.lap("after metrics", log.info(_))
    // Compute raw scores on the test set. clearThreshold() allows for real values in the range of 0 to 1.
    val predictionAndLabels: RDD[(Double, Double)] = test.map {
        case LabeledPoint(label, features) => (model.clearThreshold().predict(features), label)
    }
    // Calculating logloss error using the real values in the range 0 to 1.
    val loss: RDD[Double] = for (pr <- predictionAndLabels) yield
    calculateErrorFactor(pr._1, pr._2)
    val logloss = loss.collect().toSeq.reduce(_ + _)
    log.info(s"logloss= ${logloss / loss.count * (-1)}")
    timer3.lap("end", log.info(_))
}
```

Improving performance

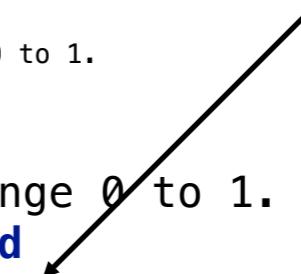
- Do you see any bad code?

```
def doClaimsPrediction(filename: String): Unit = {
    val log = Logger.getLogger(getClass.getName)
    val timer1 = Timer.apply
    val data: RDD[LabeledPoint] = LoadClaimData.loadData(filename)
    val pca: PCAModel = new PCA(60).fit(data.map(_.features))
    val projected: RDD[LabeledPoint] = data.map(p => p.copy(features = pca.transform(p.features)))
    // Split data into training (60%) and test (40%).
    val splits: Array[RDD[LabeledPoint]] = projected.randomSplit(Array(0.6, 0.4), seed = 11L)
    val training: RDD[LabeledPoint] = splits(0).cache()
    val test: RDD[LabeledPoint] = splits(1)
    // Run training algorithm to build the model
    val model: LogisticRegressionModel = new LogisticRegressionWithLBFGS()
        .setNumClasses(2)
        .run(training)
    // Scala class evaluation Get evaluation metrics.
    val predictionAndLabels2: RDD[(Double, Double)] = test.map {
        case LabeledPoint(label, features) =>
            val prediction = model.setThreshold(0.6).predict(features)
            (prediction, label)
    }
    val timer2 = timer1.lap("after modeling", log.info(_))
    // Precision and Recall using BinaryClassificationMetrics
    val metrics: BinaryClassificationMetrics = new
    BinaryClassificationMetrics(predictionAndLabels2)
    log.info(s"Recall = ${metrics.pr().collect().tail.head._1}")
    log.info(s"Precision = ${metrics.pr().collect().tail.head._2}")
    val timer3 = timer2.lap("after metrics", log.info(_))
    // Compute raw scores on the test set. clearThreshold() allows for real values in the range of 0 to 1.
    val predictionAndLabels: RDD[(Double, Double)] = test.map {
        case LabeledPoint(label, features) => (model.clearThreshold().predict(features), label)
    }
    // Calculating logloss error using the real values in the range 0 to 1.
    val loss: RDD[Double] = for (pr <- predictionAndLabels) yield
        calculateErrorFactor(pr._1, pr._2)
    val logloss = loss.collect().toSeq.reduce(_ + _)
    log.info(s"logloss= ${logloss / loss.count * (-1)}")
    timer3.lap("end", log.info(_))
}
```

There are two *collect* operations on the same variable



We are performing the reduce after collect so it can't be done in parallel



Improving performance

- After a little cleanup...

```
def doClaimsPrediction(filename: String): Unit = {
    val log = Logger.getLogger(getClass.getName)
    val timer1 = Timer.apply
    val data: RDD[LabeledPoint] = LoadClaimData.loadData(filename)
    val pca: PCAModel = new PCA(60).fit(data.map(_.features))
    val projected: RDD[LabeledPoint] = data.map(p => p.copy(features = pca.transform(p.features)))
    // Split data into training (60%) and test (40%).
    val splits: Array[RDD[LabeledPoint]] = projected.randomSplit(Array(0.6, 0.4), seed = 11L)
    val training: RDD[LabeledPoint] = splits(0).cache()
    val test: RDD[LabeledPoint] = splits(1)
    // Run training algorithm to build the model
    val model: LogisticRegressionModel = new LogisticRegressionWithLBFGS()
        .setNumClasses(2)
        .run(training)
    // Scala class evaluation Get evaluation metrics.
    val predictionAndLabels2: RDD[(Double, Double)] = test.map {
        case LabeledPoint(label, features) =>
            val prediction = model.setThreshold(0.6).predict(features)
            (prediction, label)
    }
    val timer2 = timer1.lap("after modeling", log.info(_))
    // Precision and Recall using BinaryClassificationMetrics
    val metrics: BinaryClassificationMetrics = new
BinaryClassificationMetrics(predictionAndLabels2)
    val metricsTuple: (Double, Double) = metrics.pr().collect().tail.head
    log.info(s"Recall = ${metricsTuple._1}")
    log.info(s"Precision = ${metricsTuple._2}")
    val timer3 = timer2.lap("after metrics", log.info(_))
    // Compute raw scores on the test set. clearThreshold() allows for real values in the range of 0 to 1.
    val predictionAndLabels: RDD[(Double, Double)] = test.map {
        case LabeledPoint(label, features) => (model.clearThreshold().predict(features), label)
    }
    // Calculating logloss error using the real values in the range 0 to 1. 3300 msecs
    val loss: RDD[Double] = for (pr <- predictionAndLabels) yield
calculateErrorFactor(pr._1, pr._2)
    val logloss: Double = loss.mean()
    log.info(s"logloss= ${logloss / loss.count * (-1) }")
    timer3.lap("end", log.info(_))
}
```

Doing only one *collect*
saved 667 msecs

using *mean*, i.e. instead of
collect then *reduce* saved

3300 msecs

Saving/caching/persisting

- Spark provides several ways that you can save previous work:
 - cache:
 - persist:
 - MLLib: save (model)

Testing

- It doesn't matter how fast your application runs if it gets incorrect results—
 - Actually, the faster an incorrect application runs, the more work you might have to do to mitigate the damage!
 - The students who wrote the example that I've used here actually claimed a much “better” *logloss* value—until I noticed that there was no unit test to check *logloss*. It turned out that they had the wrong formula!
- All software must be unit-tested and functional-tested—
 - Like a verbal contract that “isn’t worth the paper it’s printed on”, a chunk of software that doesn’t have good unit test coverage isn’t worth the space it takes up on the disk!

Unit testing *without* Spark

- Where possible, create and run **unit tests** that don't invoke Spark (they're just going to run a lot faster):

```
package learn
import org.scalatest.{ FlatSpec, Matchers, BeforeAndAfterAll }
class ClaimPredictionSpec extends FlatSpec with BeforeAndAfterAll with Matchers {
  import org.scalactic._
  implicit val efEq = new Equality[Double] {
    def areEqual(a: Double, b: Any): Boolean = math.abs(a-b.asInstanceOf[Double])<5E-3
  }
  val preal = 0.6
  "calculateErrorFactor(preal, 1)" should "match case Math.log(preal)" in {
    ClaimsPrediction.calculateErrorFactor(preal, 1) should === (Math.log(preal))
  }
  it should "match case 0.0" in {
    ClaimsPrediction.calculateErrorFactor(0.0, 1) should === (0.0)
  }
  it should "match case 1.0" in {
    ClaimsPrediction.calculateErrorFactor(1.0, 1) should === (0.0)
  }
  "calculateErrorFactor(preal, 0)" should "match case Math.log(1 - preal)" in {
    ClaimsPrediction.calculateErrorFactor(preal, 0) should === (Math.log(1 - preal))
  }
  it should "match case 0.0" in {
    ClaimsPrediction.calculateErrorFactor(0.0, 0) should === (0.0)
  }
  it should "match case 1.0" in {
    ClaimsPrediction.calculateErrorFactor(1.0, 0) should === (0.0)
  }
}
```

Unit testing *with* Spark

- But don't be afraid to use Spark in your unit tests when necessary [Spark is happy to run on a single machine]

```
package learn
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.SQLContext
import org.scalatest.{BeforeAndAfterAll, FlatSpec, Matchers}
class LogLossSpec extends FlatSpec with BeforeAndAfterAll with Matchers {
    var sc: SparkContext = _
    var sqlContext: SQLContext = _
    override protected def beforeAll(): Unit = {
        super.beforeAll()
        val conf = new SparkConf().setMaster("local[2]")
            .setAppName("Insurance Claim")
            .set("spark.driver.allowMultipleContexts", "true")
        sc = new SparkContext(conf)
        sqlContext = new SQLContext(sc)
    }
    override protected def afterAll(): Unit = {
        try {
            sc.stop()
        } finally {
            super.afterAll()
        }
    }
    "logLoss((0.5, 0.5))" should "equal log one half" in {
        val rdd = sc.parallelize(Array((0.5, 0.5)))
        val loss = for (pr <- rdd) yield ClaimsPrediction.calculateErrorFactor(pr._1, pr._2)
        loss.sum should === (-math.log(2.0) +- 0.005)
    }
}
```



You must set this if you're going to run multiple unit tests

Performance tuning

- There are some good (and obvious) rules that apply to performance tuning—ignore them at your peril:
 - Don't infer results for a full-scale application from a sample—
 - there will typically be some helpful indications but the behavior of a full-scale app might be surprise you.
 - Don't change more than one parameter at a time—
 - you will not be able to deduce which change actually resulted in the increase (or decrease) of performance.
 - Don't guess! Instrument your app and run it—
 - performance tuning can be very *counter-intuitive*;
 - *document* your trials in a notebook (maybe even on paper).
 - Don't waste time on untested code—
 - Ensure that your code is fully functional (unit tests are green!) before performance tuning—and if you have to change the code, go back and run your earlier performance tests to ensure no degradation.

Tweaking: Command Line parameters

- Common params (don't touch these without really understanding what you're doing—Spark will generally use sensible values for these):
 - *—num.executors*
 - *—executor.cores*
 - For example, using HDFS, e.g., it may make sense to have more executors and fewer cores per executor because of the HDFS blocks
 - *—driver-memory*
- Partitions (of an RDD)
 - Ideal number is the total number of cores that you have available.
 - If that number varies, it's best to set the number of partitions to the maximum number of cores that will be in play (the overhead of having too many partitions is small compared to the penalty of having insufficient partitions)

Tweaking: Configuration parameters

- Some of the common configuration parameters...
 - *spark.executor.memory*: memory available to each executor (shared by all cores)
 - *spark.executor.extraJavaOptions*: for tuning each executor's JVM
 - *spark.executor.parallelism*: default number of data partitions
 - *spark.storage.memoryFraction* (0.6 by default) is for persisted RDDs
 - *spark.shuffle.memoryFraction* (0.2 by default) is reserved for the shuffle process

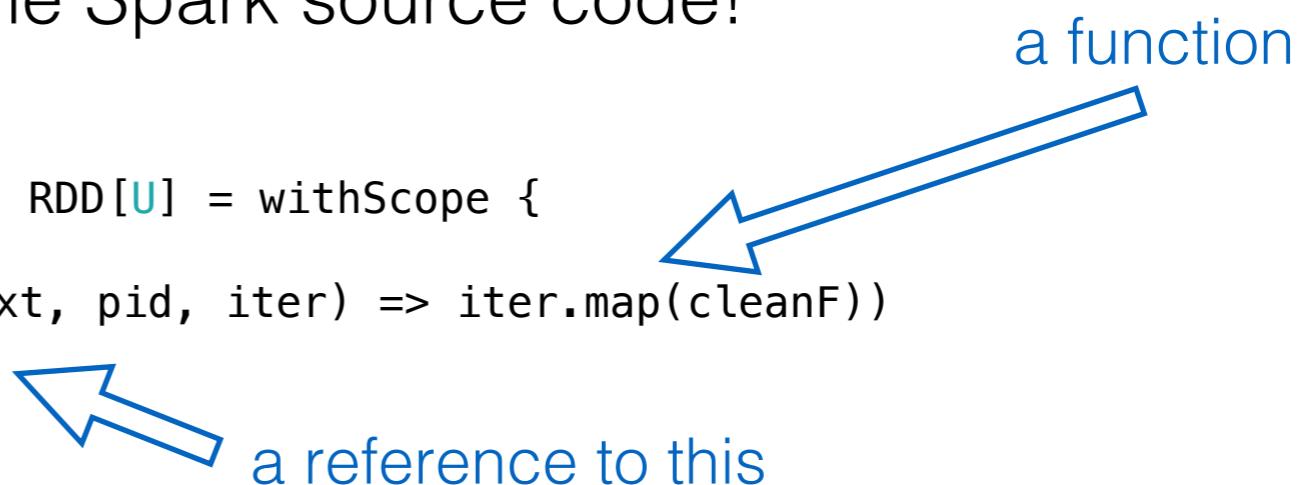
System factors which impact performance

- What sort of files do you work with?
 - Compressed files? Are they splittable into convenient chunks?
 - Text files?
 - Free-form?
 - Structured (CSV, JSON, XML, etc.)
 - In this case, you should read them into a DataFrame (e.g. use the Databricks CSV reader)
 - avoid using XML because such files typically straddle many lines and are therefore not splittable.
 - Sequence files and Avro files
 - Parquet files: excellent for using SparkSQL.

Understanding how it works

- Perhaps the most important thing for writing performant Spark applications:
 - is to understand exactly (or almost) what Spark is doing.
 - that's why it's so important to understand Scala, since Spark is written in Scala.
 - And the most important things about Scala, for understanding Spark, are the functional programming style and the concept of lazy evaluation.
 - Any time you are confused by something, or just want a better understanding, dive into the Spark source code!
 - example: *map*:

```
def map[U: ClassTag](f: T => U): RDD[U] = withScope {  
  val cleanF = sc.clean(f)  
  new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.map(cleanF))  
}
```



Serialization

- Serialization performance is important for any network-distributed application.
- By default, Spark uses Java serialization—
 - but you can configure it to use Kryo (typically much faster):
`conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")`
 - it's a little more complicated if you have your own custom classes, but if you are just using standard Spark classes, you should be all set.
- Functions:
 - Remember that functions, e.g. those you apply in transformations, must be serializable.
 - Most of the time, that means that they should be defined anonymously, or within “objects”. If you pass a class method (i.e. an instance method) as a function, then the class itself must be serializable too.

Compression

- Just like serialization, compression can also be important for the same reasons.
- By default, Spark uses snappy compression—but you can configure other codecs if you really know what you're doing...

Spark UI

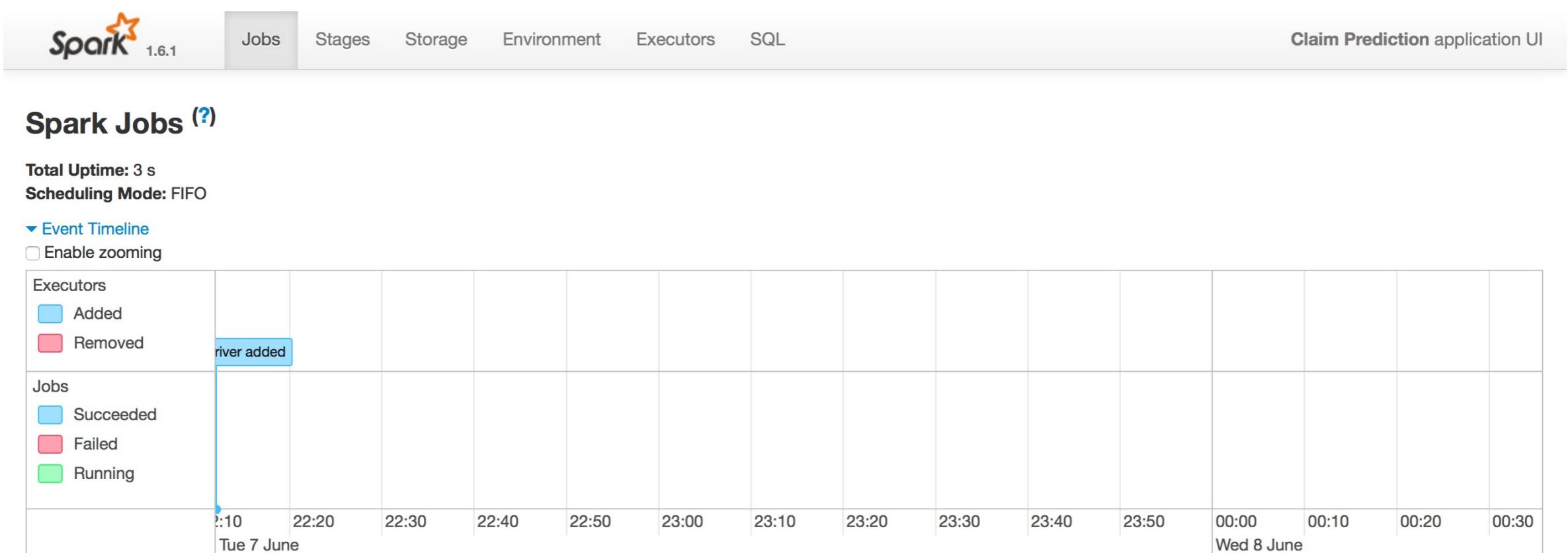
- Don't forget to use the Spark UI (port 4040) to show you what your DAGs look like.
- You can also use accumulators (which show up in the UI) so that you can see if things are actually happening—and how fast.

Spark resources

- Overview: <http://spark.apache.org/docs/latest/index.html>
- Running on AWS EC2: <http://spark.apache.org/docs/latest/ec2-scripts.html>
- Configuration: <http://spark.apache.org/docs/latest/configuration.html>
- Monitoring: <http://spark.apache.org/docs/latest/monitoring.html>
- Tuning: <http://spark.apache.org/docs/latest/tuning.html>
- Spark Streaming: <http://spark.apache.org/docs/latest/streaming-programming-guide.html>
- Data frames/sets, SparkSQL, etc.:
<http://spark.apache.org/docs/latest/sql-programming-guide.html>
- MLlib: <http://spark.apache.org/docs/latest/mllib-guide.html>
- GraphX: <http://spark.apache.org/docs/latest/graphx-programming-guide.html>

Monitoring

- Spark comes with a built-in monitoring UI:
 - <http://driver:4040>



AWS/Spark alternatives

- spark-ec2: kind of a hack that has just stuck around
- Databricks: comprehensive eco-system:
<https://www.gitbook.com/book/databricks/databricks-spark-reference-applications/details>
- Amazon EMR:
<https://aws.amazon.com/elasticmapreduce/details/spark/>
- Flintrock: <https://spark-summit.org/east-2016/events/flintrock-a-faster-better-spark-ec2/>
- Apache BigTop, Ubuntu Juju, Docker, etc.

Miscellaneous topics

- Scala/Java interaction
 - Scala and Java both inhabit the JVM (like Groovy, Clojure, etc.) so they are natural fellows
 - There are some differences in the languages that require you being a little bit careful:
 - The Scala compiler will warn you about these problems (generally speaking, the Scala compiler is very smart about all of these issues);
 - Scala doesn't have primitives (stack objects like *int*, *double*): all objects in Scala live on the heap: *Int*, *Double*, etc. Scala will do the proper thing for you.
 - Collections are similar but not the same!! However, there are implicit conversions which you can use. But bear in mind that these take time.
 - Generics are different (Scala has higher “kinds”): but both languages have type erasure so at run-time, the types aren't known either way.