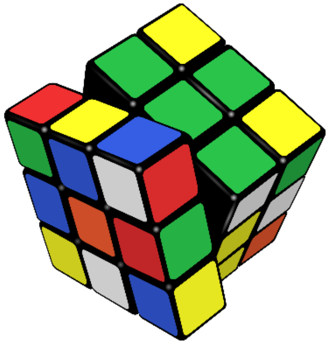


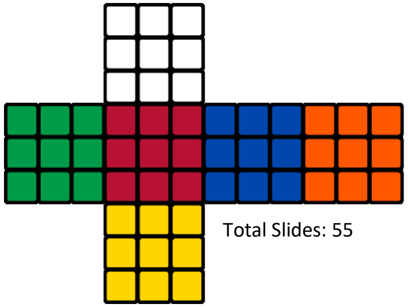
Big Data Engineering For Analytics

Functional Thinking in Scala

Suria R Asai

(suria@nus.edu.sg)

NUS-ISS



Total Slides: 55

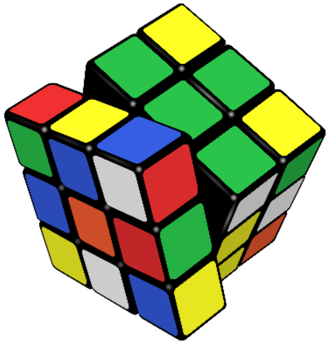
Learning Objectives

Big Data Engineering For Analytics

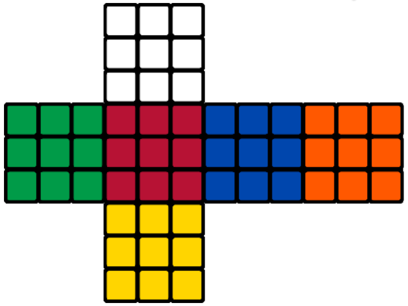
- Understand that **`Scala`** is a multi-paradigm language; it's a mixture of object-oriented and functional programming.
- Understand pure functions, the concept of immutability, and referential transparency
    - Scala Objects
    - Literals and Data Types
    - Operators, Interpolators and Looping
    - Recursion & Conditionals
    - Functional Syntax
    - Hierarchy of Collections
    - Traits
    - Exception Handling

Agenda

- Scala
- Scala Basic Constructs
- Scala Collections
- Scala Functions and Higher Order Functions
- Scala Examples
- Scala IDE
- Summary



Big Data Engineering For Analytics



Scala Class and Object

"The craft of programming begins with empathy, not formatting or languages or tools or algorithms or data structures."

~ Kent Beck

Object and Functions

Big Data Engineering For Analytics

- Scala code has to be in an object or a class

```
// A comment!
/* Another comment */
/** A documentation comment */
object MyFirstModule {
  def abs(n: Int): Int =
    if (n < 0) -n
    else n

  private def formatAbs(x: Int) =  {
   val msg = "The absolute value of %d is %d"
   msg.format(x, abs(x))
  }

  def main(args: Array[String]): Unit =
   println(formatAbs(-42))
}
```

Declares a singleton object, which simultaneously declares a class and its only instance

abs takes an integer and returns an integer

Returns negation if number is less than zero

A string with two place holders marked as %d

Replaces the two %d with x and abs(x)

Main method where execution starts

Scala Application

- Scala provides a trait, scala.App

```
// In file Summer.scala
import Season.calculate
object Summer {
 def main(args: Array[String]) = {
   for (arg <- args)
     println(arg + ": " + calculate(arg))
 }
}
```

```
// In file NewSummer.scala
import Season.calculate

object NewSummer extends App {
    for (season <- List("fall", "winter", "spring"))
         println(season + ": " + calculate(season))
}
```

A Class Example

Big Data Engineering For Analytics

Because of the functional language design, the abstraction remains intact without needing getters and setters.

```
JAVA
public class Product {
    private int id;
    private String category;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getCategory() {
        return category;
    }
    public void setCategory(String category) {
        this.category = category;
    }
}
```

```
JAVA
public class User {
    private String name;
    private List<Order> orders;
    public User() {
        orders = new ArrayList<Order>();
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public List<Order> getOrders() {
        return orders;
    }
    public void setOrders(List<Order> orders) {
        this.orders = orders;
    }
}
```

A Class Example

Big Data Engineering For Analytics

JAVA
```java
public class Order {
    private int id;
    private List<Product> products;

    public Order() {
        products = new ArrayList<Product>();
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public List<Product> getProducts() {
        return products;
    }

    public void setProducts(List<Product> products) {
        this.products = products;
    }
}
```

SCALA
```scala
class User {
    var name: String = _
    var orders: List[Order] = Nil
}

class Order {
    var id: Int = _
    var products: List[Product] = Nil
}

class Product {
    var id: Int = _
    var category: String = _
}
```

SCALA
```scala
case class User(name: String, orders: List[Order])
case class Order(id: Int, products: List[Product])
case class Product(id: Int, category: String)
```

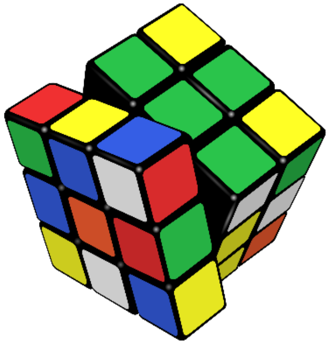
Scala Recursive Functions

Big Data
Engineering
For Analytics

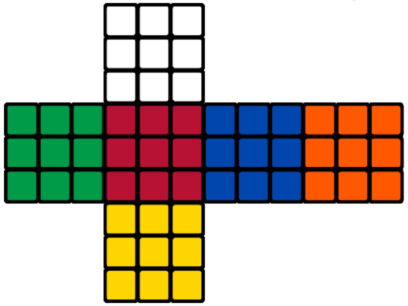
- The way we write loops functionally, without mutating a loop variable, is with a recursive function.

```
// A definition of factorial, using a local, tail recursive function
 def factorial(n: Int): Int = {
   def go(n: Int, acc: Int): Int =
     if (n <= 0) acc
     else go(n-1, n*acc)

   go(n, 1)
 }
```

In Scala, we can define functions inside any block, including within another function definition. Since it's **local**, the **go function** can only be referred to from within the body of the **factorial function**, just like a local variable would. The definition of factorial finally just consists of a call to go with the initial conditions for the loop.



Big Data Engineering For Analytics



Scala Basic Constructs

"When debugging, novices insert corrective code; experts remove defective code."

– Richard Pattis

Vals and vars

Big Data Engineering For Analytics

- When we use a **`val`** keyword to assign a value to any attribute, it becomes a value.
- A **`val`** declaration is used to allow only immutable data binding to an attribute.
- But if an attribute's value is going to change in the course of our program, we can use the **`var`** declaration

```
scala> val a = 10
a: Int = 10
```

```
scala> var b = 10
b: Int = 10
scala> b = 12
b: Int = 12
```

```
scala> val a: String = "I can be inferred.“
a: String = I can be inferred.
```

Literals

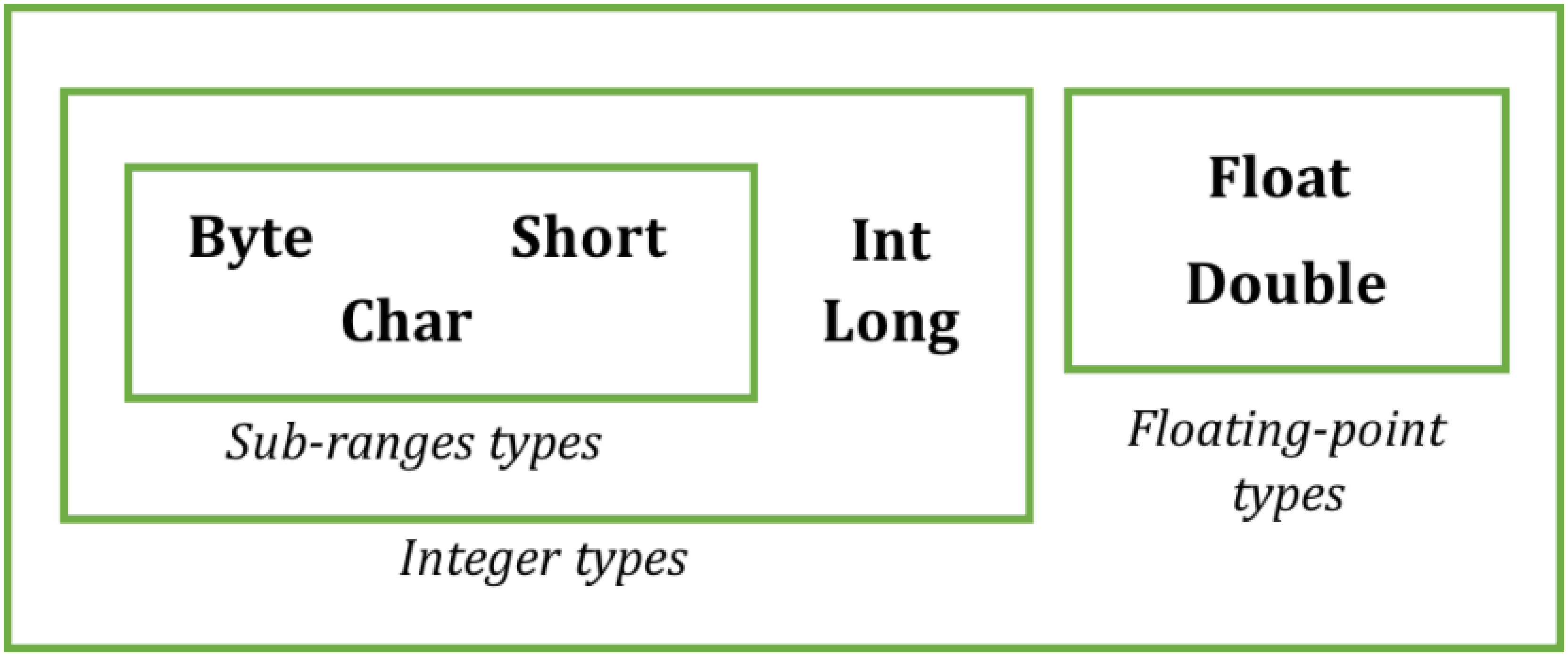
- Integer literals
- Floating point literals
- Boolean literals
- Character literals
- String literals
- Symbol literals
- Tuple literals
- Function literals

```
scala> val aTuple = ("Val1", "Val2", "Val3")
aTuple: (String, String, String) = (Val1,Val2,Val3)
scala> println("Value1 is: " + aTuple._1)
Value1 is: Val1
```

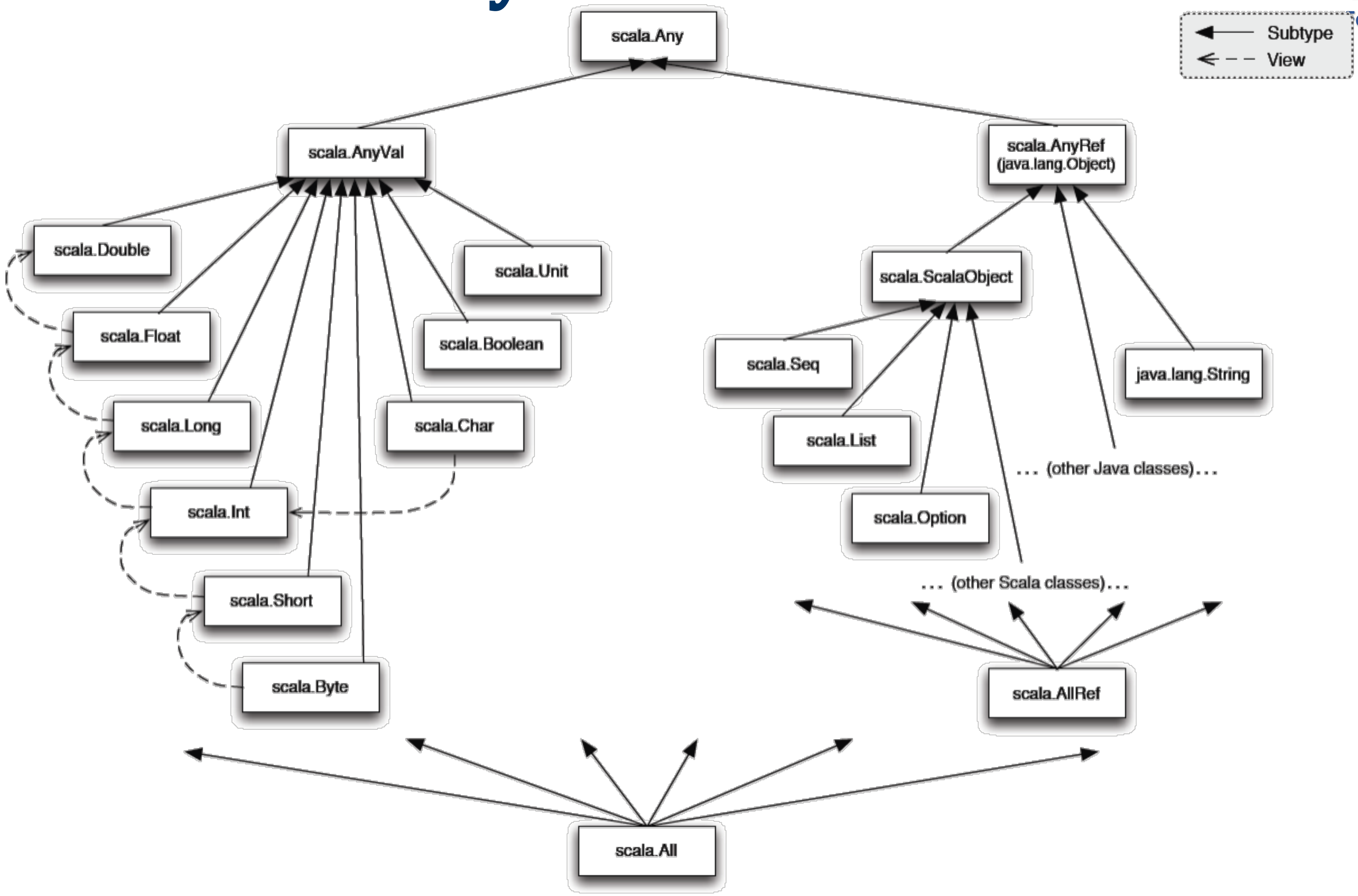
Type	Value	Minimum value	Maximum value
Int		-2^31	2^31 - 1
Long		-2^63	2^63- 1
Short		-2^15	2^15 - 1
Byte		-2^7	2^7 – 1
Float	Needs ‘f’ at end		
Double	Accepts exponential numbers example 3.657e2 = 3.657 * 10^2		
Boolean	Value = true or false		
Symbol	single quote (') followed by alphanumeric identifier		
Tuple	Tuple is a data type in Scala.		
Function	The basic structure of a function is something that can take some parameters and return a response. If we've to represent a function that takes an Int value and respond in String, it will be like this: Int=> String		

Big Data Engineering For Analytics

Data Types


Byte
Short
Char
Int
Long
Sub-ranges types
Integer types
Float
Double
Floating-point types

Class Hierarchy



Null and Nothing

- Null and Nothing are called **Bottom types** in Scala.
- In Scala, Nothing is a subtype of everything, hence the inferred type automatically becomes of type Int.

Visualize Types:

Branch One

[Int] -> … … … … -> AnyVal -> Any

Branch Two

Nothing -> … -> [Int] -> … … … … -> AnyVal -> Any

Operators

- Infix operators
- Prefix operators
- Postfix operators

- Arithmetic operators
  - addition (+), subtraction (-), multiplication (*), division (/), and remainder (%)
- Relational operators
  - ==, !=, >, <, >= and <=
- Logical operators
  - ! (NOT), && (AND), and || (OR)
- Bitwise operators
  - Bitwise AND (&), OR (|), and XOR (^)

Operator Precedence:

! ~

* / %

\+ -

\>> >>> <<

:

= !

\> >= < <=

&

^

|

&&

||

Wrapper Classes

Big Data Engineering For Analytics

Rich Wrapper Classes:	
Base Type	***Wrapper***
Byte	**scala.runtime.RichByte**
Short	**scala.runtime.RichShort**
Char	**scala.runtime.RichChar**
Int	**scala.runtime.RichInt**
Boolean	**scala.runtime.RichBoolean**
Float	**scala.runtime.RichFloat**
Double	**scala.runtime.RichDouble**
String	**scala.collection.immutable.StringOps**

Interpolators

Big Data Engineering For Analytics

- The s interpolator

```
scala> val myAge = s"I completed my $age.
"myAge: String = I completed my 25.
```

- The f interpolator

```
scala> val amount = 100
amount: Int = 100
scala> val firstOrderAmount = f"Your total amount is: $amount%.2f"
firstOrderAmount: String = Your total amount is: 100.00
```

- The raw interpolator

```
scala> val rawString = raw"I have no escape \n
character in the String \n "rawString: String = "I
have no escape \n character in the String \n "
```

Format Specifiers

Format Specifiers	
Specifiers	***Description***
%c	Characters
%d	Decimal Numbers
%e	Exponentials
%f	Floating Point Numbers
%i	Integers
%f	Octal Numbers
%u	Unsigned decimal number
%x	Hexadecimal Number

Loops

```
scala> val stocks = List("APL", "GOOG", "JLR", "TESLA")
stocks: List[String] = List(APL, GOOG, JLR, TESLA)

scala> stocks.foreach(x => println(x))
APL
GOOG
JLR
TESLA
```

```
scala> val stocks = List("APL", "GOOG", "JLR", "TESLA")
stocks: List[String] = List(APL, GOOG, JLR, TESLA)

scala> val iteraatorForStocks = stocks.iterator
iteraatorForStocks: Iterator[String] = non-empty iterator

scala> while(iteraatorForStocks.hasNext) println(iteraatorForStocks.next())
APL
GOOG
JLR
TESLA
```

Loops

```
scala> do println("I'll stop by myself after 1 time!") while(false)
```

```
object ForExpressions extends App {

val person1 = Person("Albert", 21, 'm')
val person2 = Person("Bob", 25, 'm')
val person3 = Person("Cyril", 19, 'f')

val persons = List(person1, person2, person3)

for {
person <- persons
age = person.age
name = person.name
if age > 20 && name.startsWith("A")
} {
println(s"Hey ${name} You've won a free Gift Hamper.")
}

case class Person(name: String, age: Int, gender: Char)
}
```

For Expression

Big Data Engineering For Analytics

For Expressions		
Term	***Ex.***	***Description***
Generator	*person <- persons*	**Generates a new element from sequence**
Definition	*age = person.age*	**Defines a value in scope**
Filter	***if*** *age > 20*	**Filters out a value from scope**

Recursion

```
object RecursionEx extends App {
/*
* 2 to the power n
* only works for positive integers!
*/
def power2toN(n: Int): Int = if(n == 0) 1 else 2 * power2toN(n - 1)
println(power2toN(2))
println(power2toN(4))
println(power2toN(6))
}
```

```
import scala.annotation.tailrec

object TailRecursionEx extends App {

/*
* 2 to the power n
* @tailrec optimization
*/
def power2toNTail(n: Int): Int = {
@tailrec
def helper(n: Int, currentVal: Int): Int = {
if(n == 0) currentVal else helper(n - 1, currentVal * 2)
}
helper(n, 1)
}

println(power2toNTail(2))
println(power2toNTail(4))
println(power2toNTail(6))
}
```

Pattern Matching

Java's switch vs Scala's Pattern Matching

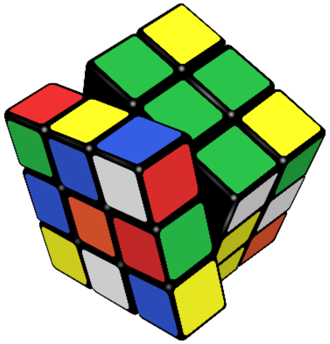
Java's switch statements	*Scala's Pattern Matching*

Java's switch statements

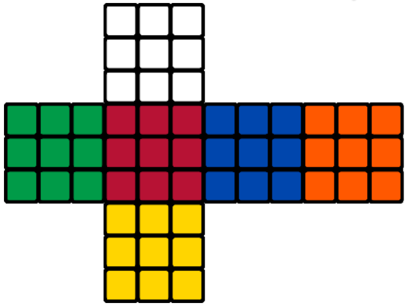
```
switch (expression){
  case value1: //code to be executed;
   break;  //optional

  case value2: //code to be executed;
   break;  //optional

  case value3: //code to be executed;
   break;  //optional

  default:
   //code to be executed if all cases
above are not matched;
}
```

Scala's Pattern Matching

```
value match {
  case value1 => //Code Block to be
executed
  case value2 => //Code Block to be
executed
  case value3 => //Code Block to be
executed
  case _ => //Code Block to be executed
}
```



Big Data Engineering For Analytics



Functional Thinking

"A good programmer is someone who looks both ways before crossing a one-way street."

— Doug Linder

Uniqueness of Scala

Big Data
Engineering
For Analytics

- **Currying:** A single argument can translate multiple arguments in the series of functions.
- **Type Interference:** The Scala programming language is very intelligent and this intelligence reduces the efforts made during the programming.
  - ➢We don’t have to mention the return type of function and data types in a clear and detailed manner these kind of stuffs will be accomplished by the programming itself.
- **Immutability:** In Scala the already declared variables values can’t be modified, this feature is known as Immutability.

Functions have no side effects

Functional languages such as **Standard ML, Scheme** and **Scala** do not restrict side effects, but it is customary for programmers to avoid them. The functional language **Haskell** expresses side effects such as I/O and other stateful computations using monadic actions.

What are side effects? A function has a side effect if it does something other than simply return a result, for example:

- Modifying a variable
- Modifying a data structure in place
- Setting a field on an object
- Throwing an exception or halting with an error
- Printing to the console or reading user input
- Reading from or writing to a file
- Drawing on the screen

Pure Functions

Big Data Engineering For Analytics

1. A pure function depends only on (a) its declared input parameters and (b) its algorithm to produce its result. A pure function has no "back doors," which means:
   - Its result can't depend on *reading* any hidden value outside of the function scope, such as another field in the same class or global variables.
   - It cannot *modify* any hidden fields outside of the function scope, such as other mutable fields in the same class or global variables.
   - It cannot depend on any external I/O. It can't rely on input from files, databases, web services, UIs, etc; it can't produce output, such as writing to a file, database, or web service, writing to a screen, etc.
2. A pure function does not modify its input parameters.

Higher-Order Function (HOF) basically means that
(a) you can treat a function as a value (val) — just like you can treat a String as a value
(b) you can pass that value into other functions.

"**Recursion** is a requirement of functional programming."

```
PF        =   ODI       +   NSE

Pure          Output        No
Function      Depends       Side
              on            Effects
              Input
```

Simple Function Syntax

Big Data Engineering For Analytics

- Generic Syntax

```
def function_name(arg1: arg1_type, arg2: arg2_type,...): return_type = ???
```

- Examples

```
def abs(x: Double) = if (x >= 0) x else -x
```

```
def fac(n: Int): Int = if (n <= 0) 1 else n * fac(n - 1)
```

```
def recursiveSum(args: Int*) : Int = {
  if (args.length == 0) 0
  else args.head + recursiveSum(args.tail : _*)
}
```

```
(x: Double) => 3 * x
```

Anonymous Function

```
val triple = (x: Double) => 3 * x
```

Function as a variable

```
def triple(x: Double) = 3 * x
```

Higher Order Function

```
def mulBy(factor : Double) = (x : Double) => factor * x
val quintuple = mulBy(5)
quintuple(20) // 100
```

Sum Example

```
Scala
// Even in scala one can write like this
def sum( ints: List[ Int]): Int = {
    var sum = 0
    for (i <- ints) {
      sum + = i
    }
    sum
}
```

```
Scala
// more cleaner version would be
def sum( xs: List[ Int]): Int = xs match {
  case Nil = > 0
  case x :: tail = > x + sum( tail)
}
```

- What’s wrong with the imperative approach?
- Who cares if I use a var field in a for loop inside a function? How does that affect anything else?
- Will the **recursive function** blow the stack with large lists?
- Is one approach faster or slower than the other?
- Thinking in the long term, is one approach more maintainable than the other?
- What if I want to write a “parallel” version of a sum algorithm (to take advantage of multiple cores); is one approach better than the other?

Filtering Example

Functional programming is a way of writing software applications using only pure functions and immutable values.

```
JAVA
// Source collection
List<String> employees = new ArrayList<String>();
employees.add("Ann");
employees.add("John");
employees.add("Amos");
employees.add("Jack");
// Those employees with their names starting with 'A'
List<String> result = new ArrayList<String>();
for (String e: employees)
if (e.charAt(0) == 'A') result.add(e);
System.out.println(result);
```

```
SCALA
// Source collection
val employees = List("Ann“, "John", "Amos“, "Jack")
// Those employees with their names starting with 'A'
val result = employees.filter ( e => e(0) == 'A' )
println(result)
```

Nesting of Functions

Big Data Engineering For Analytics

```
object FunctionSyntaxOne extends App {
// Compare Integers
def compareIntegersV4(value1: Int, value2: Int): String = {
 println("Executing V4") val
 result = if (value1 == value2) 0 else if (value1 > value2) 1 else -1
         giveAMeaningFullResult(result, value1, value2)
}
// Private Function for Syntax
 private def giveAMeaningFullResult(result: Int, value1: Int, value2: Int)
     = result match
  { case 0 => "Values are equal" case -1 => s"$value1 is smaller than $value2"
   case 1 => s"$value1 is greater than $value2"
   case _ => "Could not perform the operation" }

// Main code
 println(compareIntegersV4(2,1))
}
```

Function Literals

Big Data Engineering For Analytics

```
object ColorPrinter extends App
{ def printPages(doc: Document, lastIndex: Int, print:
    (Int) => Unit) = if(lastIndex <= doc.numOfPages)
             for(i <- 1 to lastIndex)print(i)

 val colorPrint = (index: Int) => println(s"Printing Color Page $index.")
 val simplePrint = (index: Int) => println(s"Printing Simple Page $index.")
 println("---------Method V1-----------")
 printPages(Document(15, "DOCX"), 5, colorPrint)
 println("---------Method V2-----------")
 printPages(Document(15, "DOCX"), 2, simplePrint) }

case class Document(numOfPages: Int, typeOfDoc: String)
```

For thinking . . .

- Write a function that computes x^n, where n is an integer. Use the following recursive definition:
  - $x^n = y \cdot y$ if n is even and positive, where $y = x^{n/2}$.
  - $x^n = x \cdot x^{n-1}$ if n is odd and positive.
  - $x^0 = 1$.
  - $x^n = 1 / x^{-n}$ if n is negative.

Interoperable …. For Example

Big Data Engineering For Analytics

Scala programs interoperate seamlessly with Java class libraries:

- Method calls
- Field accesses
- Class inheritance
- Interface implementation

all work as in Java.

Scala programs compile to JVM bytecodes.

Scala's syntax resembles Java's, but there are also some differences.

object instead of **static** members

Array[String] instead of String[]

```
object Example1 {
  def main(args: Array[String]) {
    val b = new StringBuilder()
    for (i ← 0 until args.length) {
      if (i > 0) b.append(" ")
      b.append(args(i).toUpperCase)
    }
    Console.println(b.toString)
  }
}
```

Scala's version of the extended **for** loop (use <- as an alias for ←)

Arrays are indexed args(i) instead of args[i]

Functional . … For Example

The last program can also be written in a completely different style:

- Treat arrays as instances of general sequence abstractions.
- Use higher-order functions instead of loops.

```
object Example2 {
  def main(args:     ay[String]) {
    println(args
          .map(_.toUpperCase)
          .mkString(" ")
  }
}
```

map is a method of Array which applies the function on its right to each array element.

Arrays are instances of sequences with map and mkString methods.

mkString is a method of Array which forms a string of all elements with a given separator between them.

A closure which applies the toUpperCase method to its String argument

Precise . . . . For Example

All code on the previous slide used library abstractions, not special syntax.

Advantage: Libraries are extensible and give fine-grained control. Elaborate static type system catches many errors early.

Specify kind of collections: mutable

Specify map implementation: HashMap
Specify map type: String to String

```
import scala.collection.mutable

val capital =
  new HashMap[String, String]
  with SynchronizedMap[String, String] {
      override def default(key: String) =
        "?"
  }

capital += ( "US" → "Washington",
             "France" → "Paris",
             "Japan" → "Tokyo" )

assert( capital("Russia") == "?" )
```

Mixin trait SynchronizedMap to make capital map thread-safe

Provide a default value: "?"

Flatmap Example

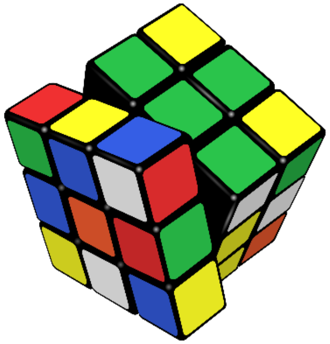
Functional programming is a way of writing software applications using only pure functions and immutable values.

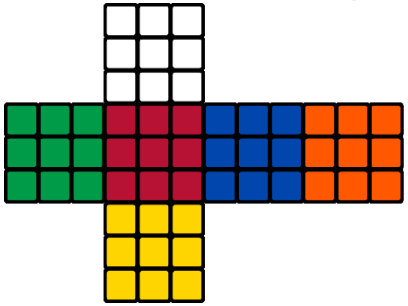
JAVA
```
public List<Product> getProducts() {
    List<Product> products = new ArrayList<Product>();
    for (Order order : orders) {
        products.addAll(order.getProducts());
    }
    return products;
}
```

SCALA
```
def products = orders.flatMap(o => o.products)
```

JAVA
```
public List<Product> getProductsByCategory(String category) {
    List<Product> products = new ArrayList<Product>();
    for (Order order : orders) {
        for (Product product : order.getProducts()) {
            if (category.equals(product.getCategory())) {
                products.add(product);
            }
        }
    }
    return products;
}
```

SCALA
```
def productsByCategory(category: String) =
orders.flatMap(o => o.products).filter(p =>
p.category == category)
```





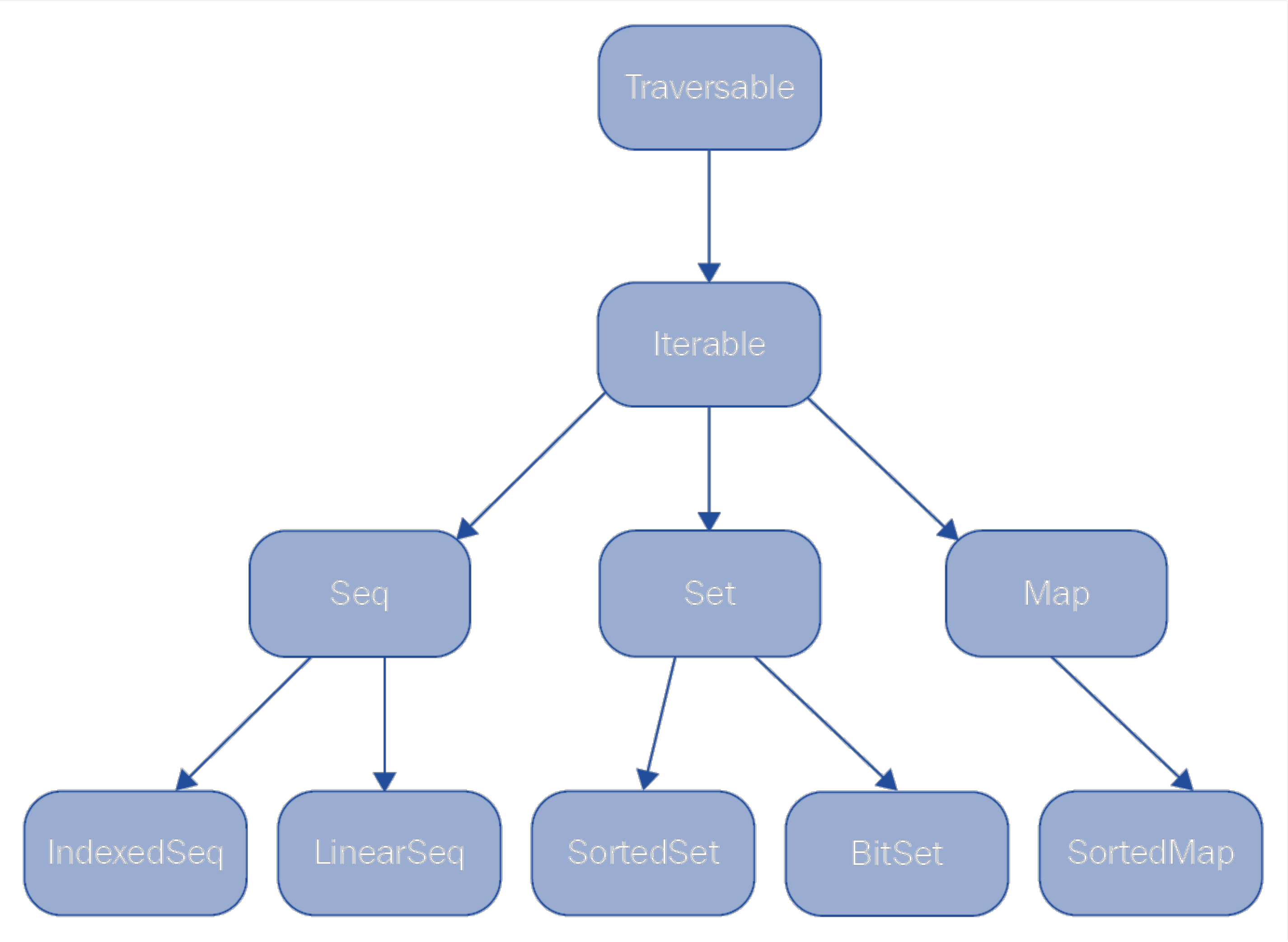
Big Data Engineering For Analytics

Scala Collections & Traits

"Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning."

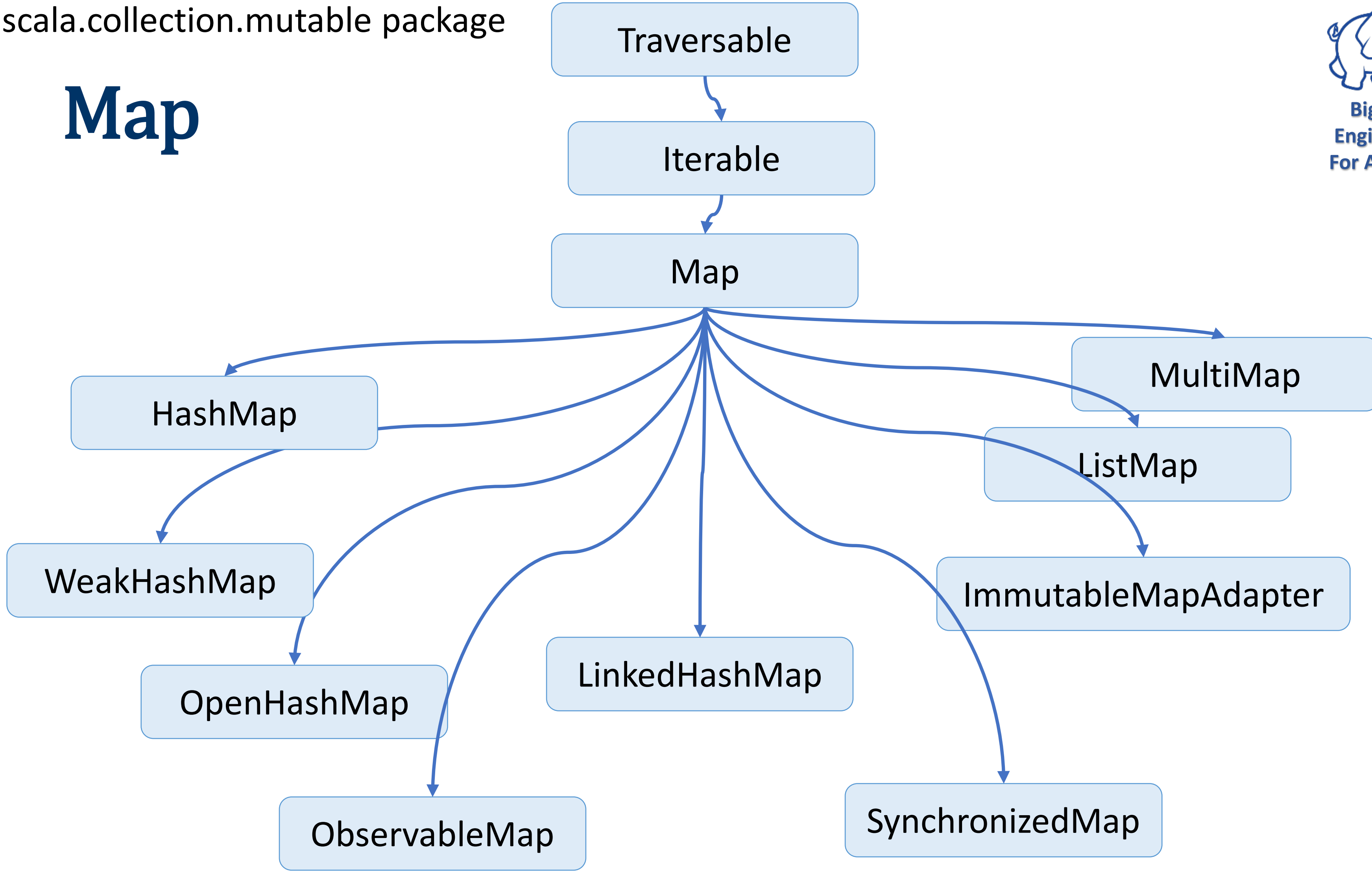
– Rich Cook

root Collection Hierarchy



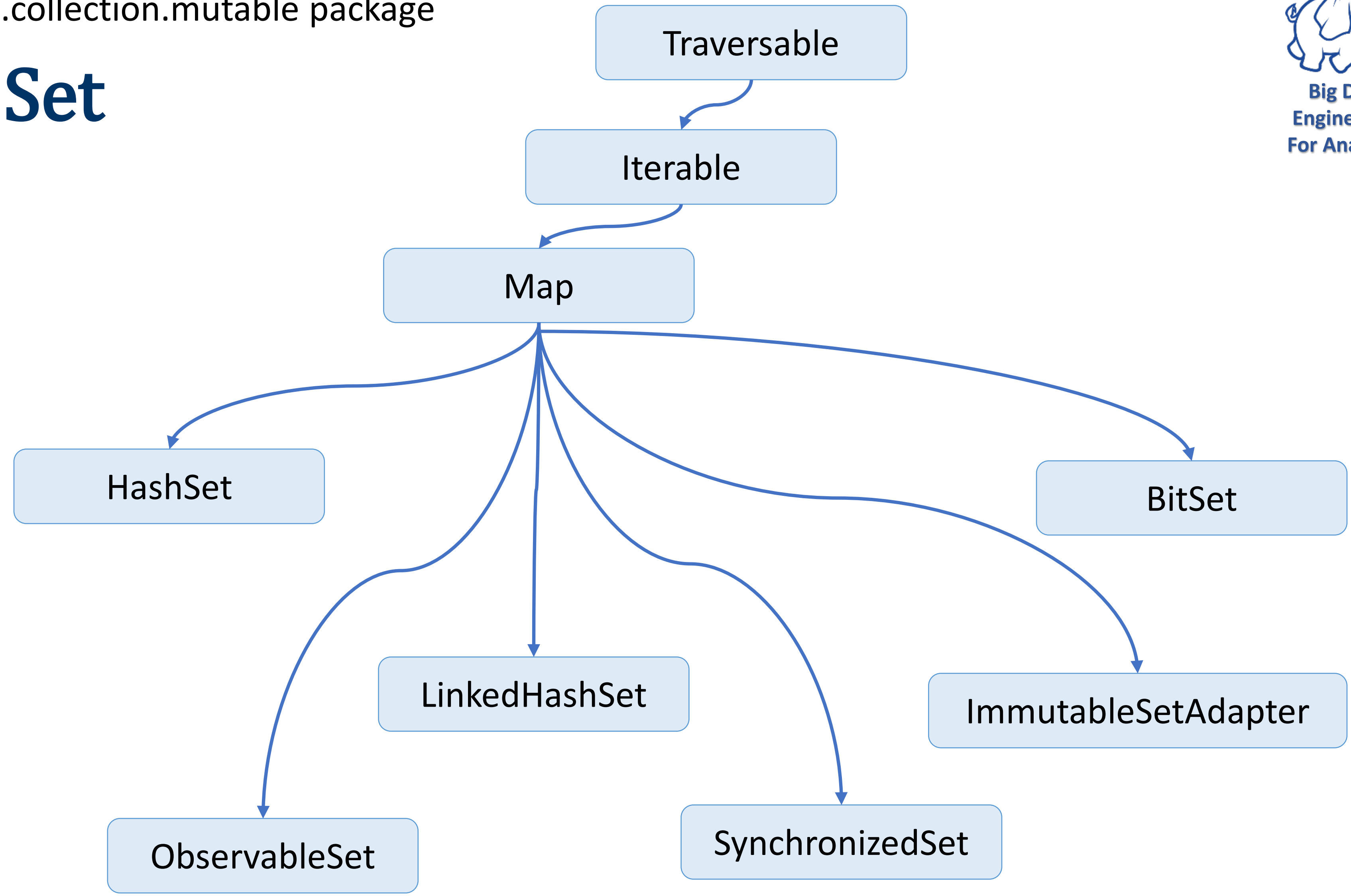
scala.collection.mutable package

Map



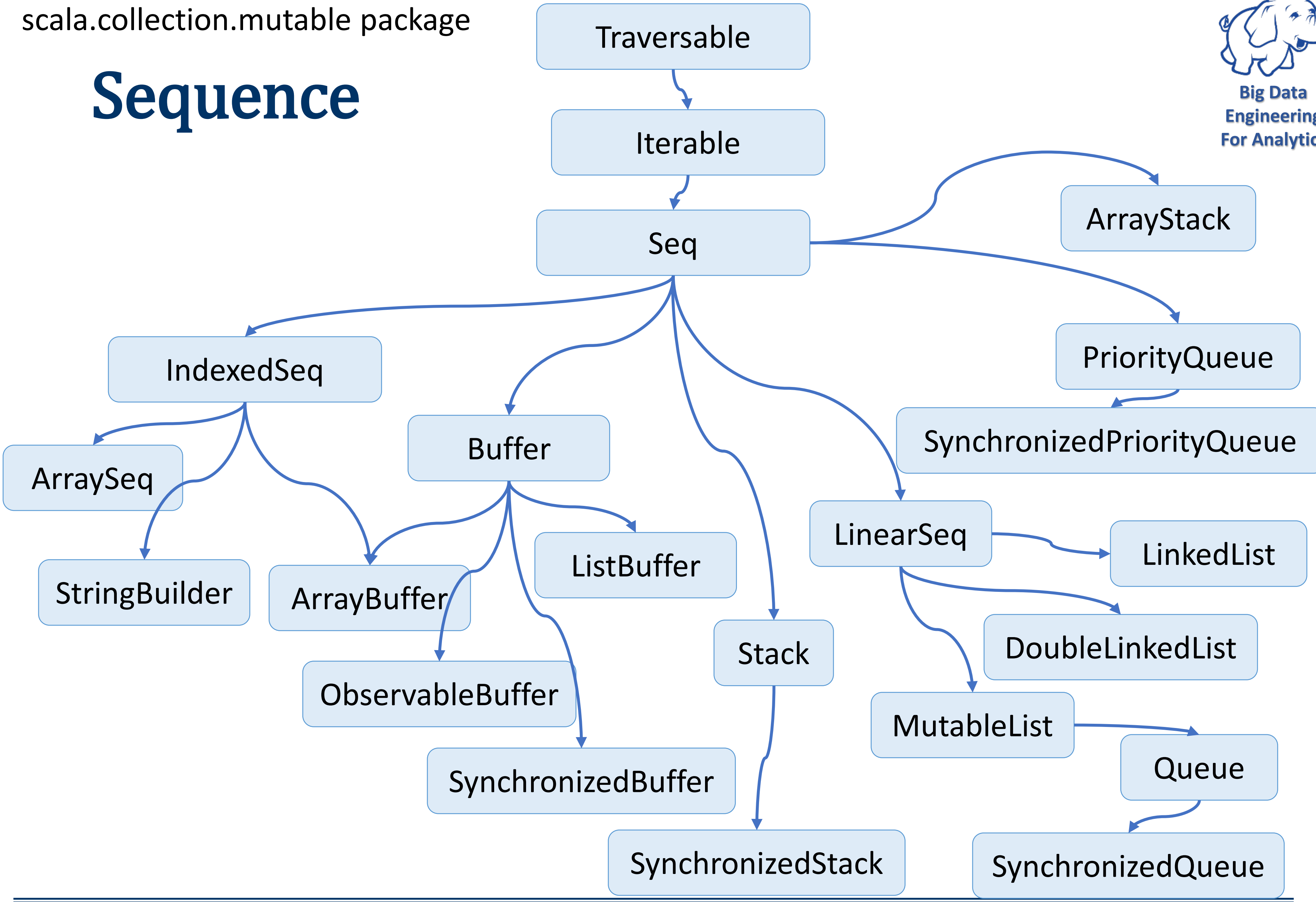
scala.collection.mutable package

Set

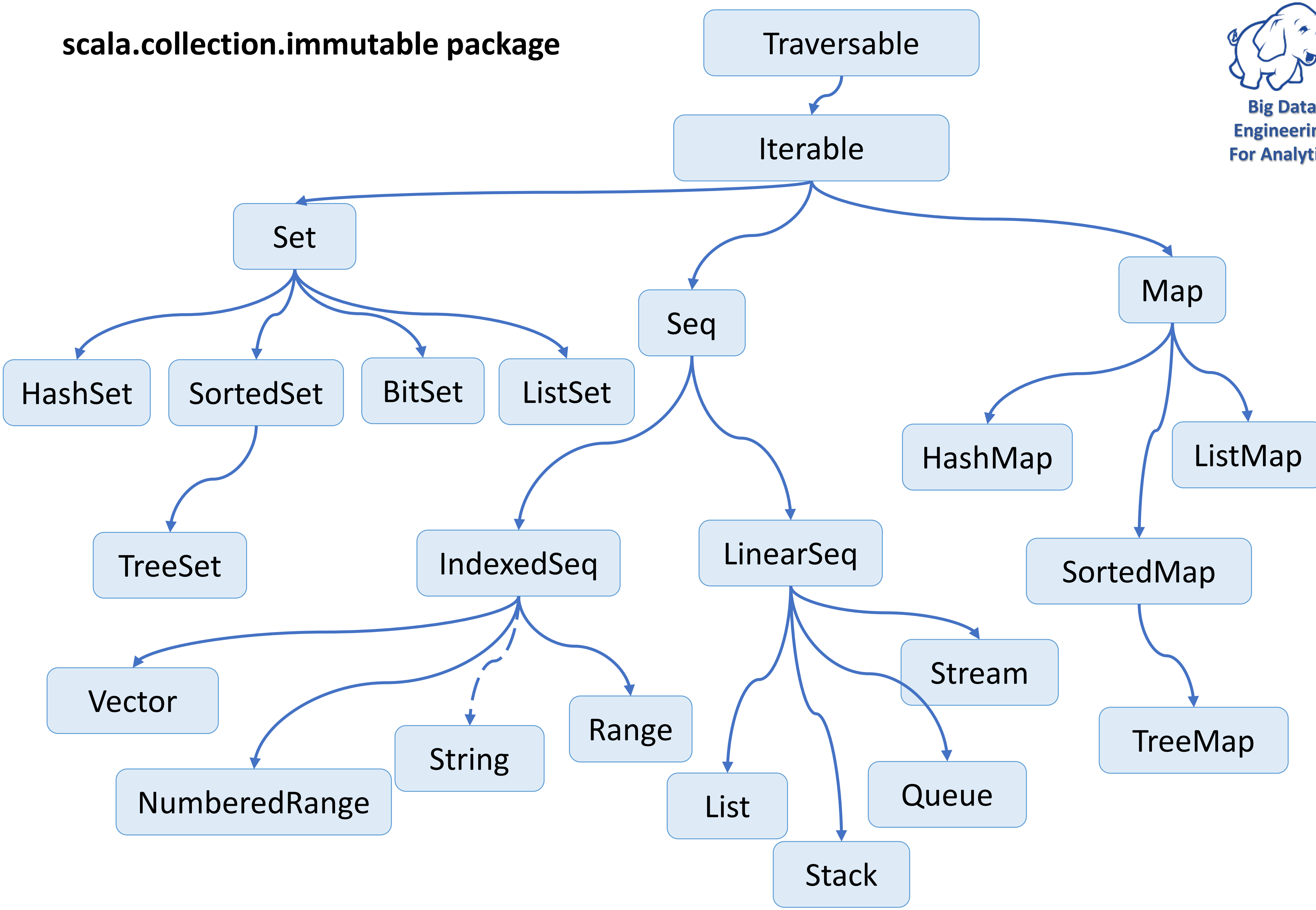


scala.collection.mutable package

Sequence



scala.collection.immutable package



Big Data Engineering For Analytics

Traversable

Big Data Engineering For Analytics

- **Additions**: Methods that append two traversables together. For two traversable collections, such as xs and ys:
  - For example, `xs ++ ys`
- **Transformations**: Methods such as map, flatMap, and collect come in this category:
  - For example, `xs.map(elem => elem.toString + "default")`
- **Conversions**: Methods with a format such as toXXX *or* mkString. These are used to convert one collection to another suitable collection:
  - For example, `xs.toArray`, `xs.mkString`, and `xs.toStream`
- **Copying**: Helper methods that copy elements from a collection to another collection, such as an array or buffer:
  - For example, `xs.copyToBuffer(arr)`
- **Information retrievals**: Methods that retrieve information such as size, or whether the collection has elements or not:
  - For example, `xs.isEmpty`, `xs.isNonEmpty`, and `xs.hasDefiniteSize`
- **Element retrievals**: Methods that retrieve an element from a collection:
  - For example, `xs.head` and `xs.find(elem => elem.toCharArray.length == 4)`
- **Sub collections**: Methods that return a sub-collection, based on ordering, or a predicate:
  - For example, `xs.tail`, `xs.init`, `xs.filter(elem => elem.toCharArray.length == 4)`
- **Folding**: Methods that apply a binary operation on each of the successive elements of a collection. Also, there are some special forms of folding operations:
  - For example, `xs.foldLeft(z)(op)`, and `xs.product`

Iterable

Big Data Engineering For Analytics

- **Sub-iterations**: Methods that return another chunked iterator:
  - For example, xs.grouped(size), and xs.sliding(size)
- **Sub-collections**: Methods that return parts of collections:
  - For example, xs.takeRight(n), and xs.dropRight(n)
- **Zipping**: Methods that return iterable collection elements in pairs:
  - For example, xs.zip(ys), and xs.zipWithIndex
- **Comparisons**: Methods that compare two iterable collections according to the order of elements:
  - For example, xs sameElements ys

Traits

- **Trait** constructs may look similar but are of a different nature to interfaces in Java. The meaning of the word trait is: a distinguishing quality or characteristic, typically one belonging to a person. We mix-in traits rather than extend from them.

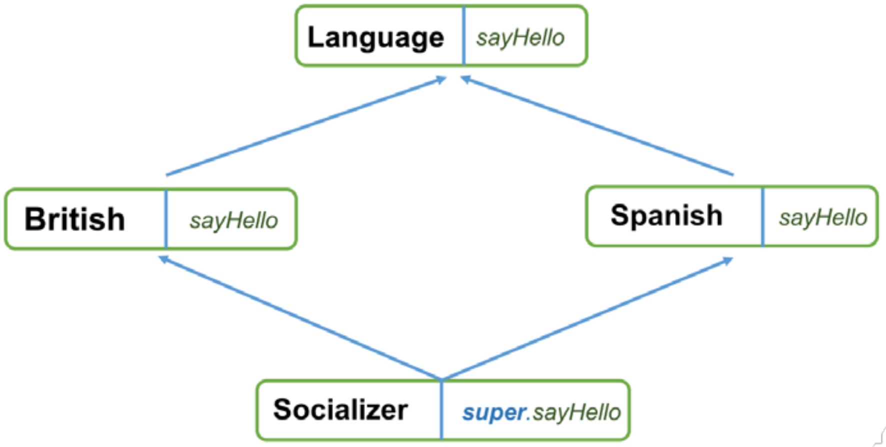
```
trait Socialize {
//people who socialise, greets. def greet(name: String) = "Hello " + name
}
```

```
case class Person(val name: String)
object SocializeApp extends App {
  val person = Person("Victor Mark")
  val employee = new Employee("David Barbara") with Socialize
  println(employee.greet(person.name))
  class Employee(fullName: String) extends Person(fullName)
}
```

Diamond Problem and Linearization

Big Data Engineering For Analytics

```
abstract class Language {
  def sayHello: String
}
 trait British extends Language {
  override def sayHello: String = "Hello"
}
 trait Spanish extends Language {
  override def sayHello: String = "Hola"
}
 class Socializer extends British with Spanish {
  override def sayHello: String = super.sayHello
}
 object Linearization extends App {
   class Person(val name: String)
   val albert = new Person("Alberto")
   val socializer = new Socializer()
   println(s"${socializer.sayHello} ${albert.name}")
}
```



Can Embed DSL (Domain Specific Languages)

Big Data Engineering For Analytics

Scala's flexible syntax makes it easy to define
high-level APIs &
embedded DSLs

Examples:

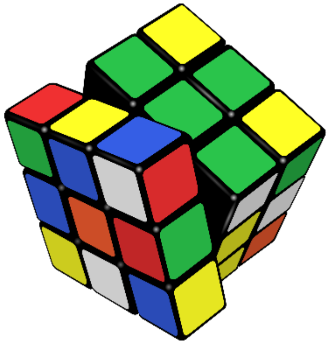
- Scala actors (the core of Twitter's message queues)
- specs, ScalaCheck
- ScalaFX
- ScalaQuery

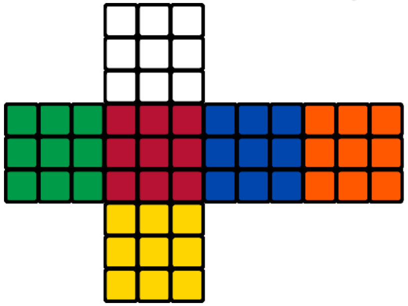
```
// asynchronous message send
actor ! message

// message receive
receive {
  case msgpat₁ => action₁
  …
  case msgpatₙ => actionₙ
}
```

scalac's plugin architecture makes it easy to typecheck DSLs and to enrich their semantics.

In Essence

- Our aim as programmers is to provide a solution to a problem through some logical implementation. Programming languages work as a tool for just that. When we implement a solution to a problem, we must be able to describe the problem (specification) so that a programming language can verify (verification) whether the solution indeed solves the problem.





Summary

"Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program."

– Linus Torvalds (Software engineer and hacker, principal force behind the development of the Linux kernel)

In Essence

- We learnt about
  - ➢the most basic val and var variable constructs
  - ➢how we can write literals, and what data types we have in Scala looping constructs such as for, while, and do while loops
  - ➢the syntax for defining a function
  - ➢the details of Scala classes and object implementation
  - ➢inheritance in Scala and discussed composition and inheritance
  - ➢Collections in Scala
- We started with the basic method and function definitions, investigated the difference between them
- We learnt about Traits
- We are yet to explore Higher Order Functions

References

"Simplicity and elegance are unpopular because they require hard work and discipline to achieve and education to be appreciated."

~Edsger Dijikstra

Books You May Enjoy. . .

- Programming in Scala, 3rd ed, Updated for Scala 2.12, by **Martin Odersky,** Lex Spoon, Bill Benners
- Functional Programming in Scala, by Paul Chiusano, Rúnar Bjarnason, Manning
- Scala for the Impatient, by Cay S. Horstmann, Addison-Wesley
- Programming Scala, Updated for Scala 2.11, by Alex Payne, Dean Wampler, O'Reilly
- Scala in Depth, by Joshua D. Suereth, Manning