Scala for Data Engineers

План

- Введение
- Выражения и типы
- Управляющие конструкции

План

- Коллекции
- ООП: Классы, объекты, трейты
- Implicits
- Полезные инструменты

Введение

Что такое Scala?

- Объектно-ориентированный
- Функциональный
- Статически типизированный
- Работает поверх JVM
- Может переиспользовать Java-код

Scala отлично подходит для

- Анализа данных и ETL (Spark)
- Потоковой обработки (Flink)
- Распределенных приложений (Akka)
- Параллельных и асинхронных вычислений (Monix, ZIO)

Окружение

- Консоль: scala или Ammonite
- Онлайн: Scastie или ScalaFiddle
- Для разработки: IntelliJ IDEA + Scala Plugin
- Альтернативы

Выражения и типы

Выражения

- Базовые выражения - literals (“буквальные”)
- Примеры: `1, "Some words", true`
- Составные выражения:
  `1 + 3, "Some" ++ " words"`

Составные выражения

Не требуется ключевое слово «return» (это необязательно). Последнее выражение возвращает значение.

```
val c = {
  val a = 11
  a + 42
}
```

Если выражение не возвращает значение, то его тип “Unit”

```
def printer(s: String): Unit = println(s)
```

Значения и переменные

- **val** – неизменяемый

```
scala> val xVal: Int = 1
xVal: Int = 1
scala> xVal = 2
<console>:12: error:
reassignment to val
```

- **var** – изменяемый

```
scala> var xVar: Int = 1
xVar: Int = 1
scala> xVar = 2
xVar: Int = 2
```

def, val and lazy val

- **def** - выполняется каждый раз при вызове
- **val** – выполняется когда определен
- **lazy val** – выполняется один раз при первом вызове

Функции могут быть определены как **def** или **val**:

```
def foo() = "foo"
val bar = () => "bar"
foo() + " " + bar()
```

Пример

Методы

- `"Some words".toUpperCase`
- Или в инфиксной нотации: `"Some words" toUpperCase`
- На самом деле операторы тоже являются методами:
  `"Some" concat " words"`
  `"Some" ++ " words"`
  `"Some".++(" words")`
  Вернут один и тот же результат

Функции

- Анонимные функции: `x => x + 1`
- Методы: `def incr(x: Int): Int = x + 1`
- Функции высшего порядка:

```
scala> def add(x: Int) = (y: Int) ⇒ x + y

add: (x: Int)Int => Int

scala> val addOne = add(1)

addOne: Int => Int = <function1>

scala> addOne(2)

res1: Int = 3
```

Типы данных

- String
- Int, Double, ...
- TupleN
- Коллекции: List, Set, Map, etc.
- Изменяемые коллекции
- Option
- Unit (= Void в Java)

Управляющие конструкции

Управляющие структуры: if

Синтаксис похож на другие C-подобные языки

```
if ( x < 20 ) {
   println("This is if statement")
}
```

Также возвращает значение

```
scala> val whichOne = if (false) "Not that one" else "This one"
whichOne: String = This one
```

Управляющие структуры: for

```
for ( a <- 1 to 10) {
   println( "Value of a: " + a )
}
```

Синтаксис очень похож, не правда ли?

Управляющие структуры: for

```
scala> for {
  a <- 1 to 5    // Коллекция 1
  b <- 1 to 5    // Коллекция 2
  if a + b < 6   // Условие
} yield a + b    // Генератор
```

```
res3: Vector(2, 3, 4, 5, 3, 4, 5, 4, 5, 5)
```

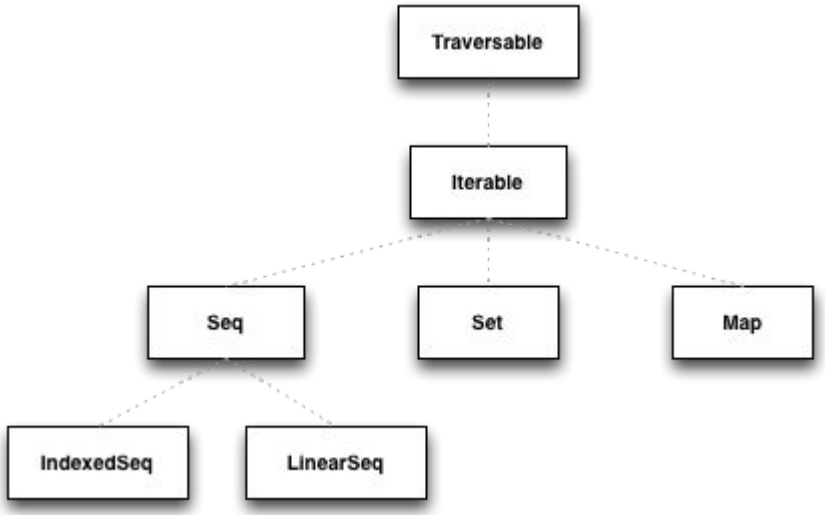
Pattern matching

Это как оператор “switch” на стероидах

```
x match {
   case 1 => "one"
   case "two" => 2
   case y: Int => s"$y is scala.Int"
   case _ => "many"
}
```

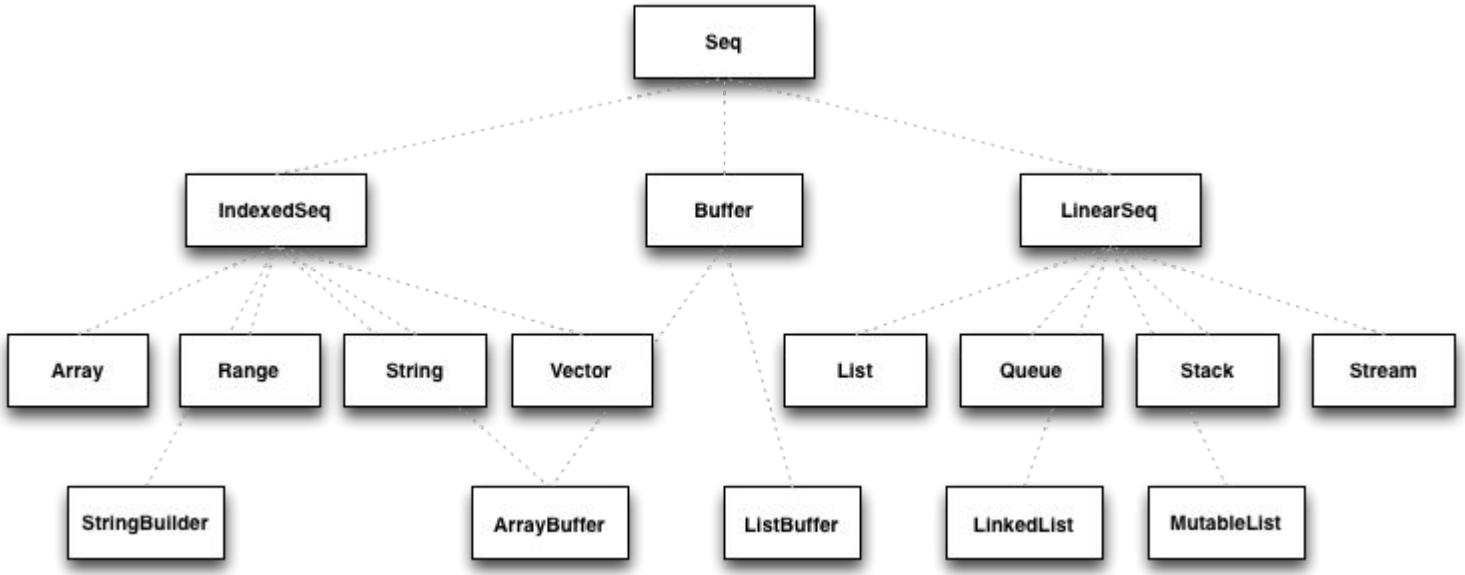
Коллекции

Иерархия коллекций в Scala выглядит так



Коллекции

… с реализацией для различных потребностей



Типичные Коллекции

```
scala> List("apple", "banana", "pear")
res7: List[String] = List(apple, banana, pear)

scala> Map("apple" -> 2, "banana" -> 1, "pear" -> 10)
res8: scala.collection.immutable.Map[String,Int] = Map(apple -> 2, banana ->
1, pear -> 10)

scala> Set("apple", "banana",  "banana",  "banana", "pear")
res9: scala.collection.immutable.Set[String] = Set(apple, banana, pear)
```

Списки

```
val fruit: List[String] = List("apples", "oranges", "pears")
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
val fruit = List.fill(10)("apples")
```

Словари

```
val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF")
println( "Keys in colors : " + colors.keys )
println( "Values in colors : " + colors.values )
println( "Check if colors is empty : " + colors.isEmpty )
```

Кортежи

Коллекция объектов смешанного типа

```
val t = (1, "hello", Console)
```

N-ый элемент может быть вызван командой “x._N”

```
val t = (4,3,2,1)
val sum = t._1 + t._2 + t._3 + t._4
```

Options

Тип Option означает «Может быть пустым» и содержит Some(_) или None.

```
val a: Option[Int] = Some(5)
val b: Option[Int] = None
println("a.getOrElse(0): " + a.getOrElse(0) )
println("b.getOrElse(10): " + b.getOrElse(10) )
scala> a.getOrElse(0): 5
scala> b.getOrElse(10): 10
```

Option может рассматриваться как коллекция с 0 или 1 элементом.

Общие методы

```
scala> List(1, 2, 3).map(x => x + 1)                    // 1 -> 1
res0: List[Int] = List(2, 3, 4)

scala> List(1, 2, 3).foreach(x => print(x))             // 1 -> Unit
123

scala> List(List(1), List(2, 3), List()).flatten
res0: List[Int] = List(1, 2, 3)

scala> List(1, 2, 3).flatMap(x => List(x, x, x))        // 1 -> 0..N
res5: List[Int] = List(1, 1, 1, 2, 2, 2, 3, 3, 3)
```

Фильтры

```
scala> val fruits = List("apple", "banana", "banana", "pear", "orange")
fruits: List[String] = List(apple, banana, banana, pear, orange)

scala> fruits.take(2)
res21: List[String] = List(apple, banana)

scala> fruits.filter(_.endsWith("e"))
res22: List[String] = List(apple, orange)

scala> fruits.exists(_.startsWith("b"))
res23: Boolean = true

scala> fruits.distinct
res24: List[String] = List(apple, banana, pear, orange)
```

Агрегация

```
scala> List(("apple", 1), ("apple", 2), ("apple", 3), ("orange", 2))
       .groupBy(_._1).mapValues(_.size)
res16: scala.collection.immutable.Map[String,Int] = Map(orange -> 1, apple -> 3)

scala> List(1, 7, 2, 9, 3).reduce { (r1, r2) => if (r1 > r2) r1 else r2 }
res17: Int = 9

scala> val accumulator = List(1, 7, 5, 9, 3)
       .foldLeft (0, 0) { (acc, x) => (acc._1 + x, acc._2 + 1) }
accumulator: (Int, Int) = (25,5)
scala> accumulator._1 / accumulator._2
res20: Int = 5   // mean value
```

Словарь

```
scala> val fruitMap = Map("apple" -> 1, "banana" -> 2, "pear" -> 3, "orange" -> 4)
fruitMap: Map[String,Int] = Map(apple -> 1, banana -> 2, pear -> 3, orange -> 4)

scala> fruitMap("banana")
res27: Int = 2

scala> fruitMap.getOrElse("potato", 10)
res28: Int = 10

scala> fruitMap.toList
res29: List[(String, Int)] = List((apple,1), (banana,2), (pear,3), (orange,4))
```

Опция как коллекция

Опция – один из двух: **Some(_)** или **None**

```
scala> Option(3)
res35: Option[Int] = Some(3)

scala> Option.empty[Int]
res36: Option[Int] = None
```

Опция – коллекция из 0 или 1 элементов

```
scala> Some(3).toList
res37: List[Int] = List(3)

scala> None.toList
res38: List[Nothing] = List()
```

Опция как коллекция

И .flatten работает как в других коллекциях:

```
scala> List(Some(1), None, Some(2)).flatten
res33: List[Int] = List(1, 2)
```

Давайте определим toInt (in: String), который возвращает Some [Int] в случае успеха и None в противном случае

```
def toInt(in: String): Option[Int] = {
  try {
    Some(Integer.parseInt(in.trim))
  } catch {
    case e: NumberFormatException => None
  }
}
```

Опция как коллекция

Теперь мы можем делать такие вещи:

```
scala> val iHopeItsNumbers = List("1", "2", "banana", "4")
iHopeItsNumbers: List[String] = List(1, 2, banana, 4)

scala> iHopeItsNumbers.map(x => toInt(x))
res40: List[Option[Int]] = List(Some(1), Some(2), None, Some(4))

scala> iHopeItsNumbers.flatMap(x => toInt(x))
res41: List[Int] = List(1, 2, 4)

scala> iHopeItsNumbers.flatMap(toInt).sum
res42: Int = 7
```

Null, None, Nil, Nothing

null – нет значения

None – пустая Опция

Nil - пустой Список

Nothing – подтип всех типов

One Ring to rule them all...

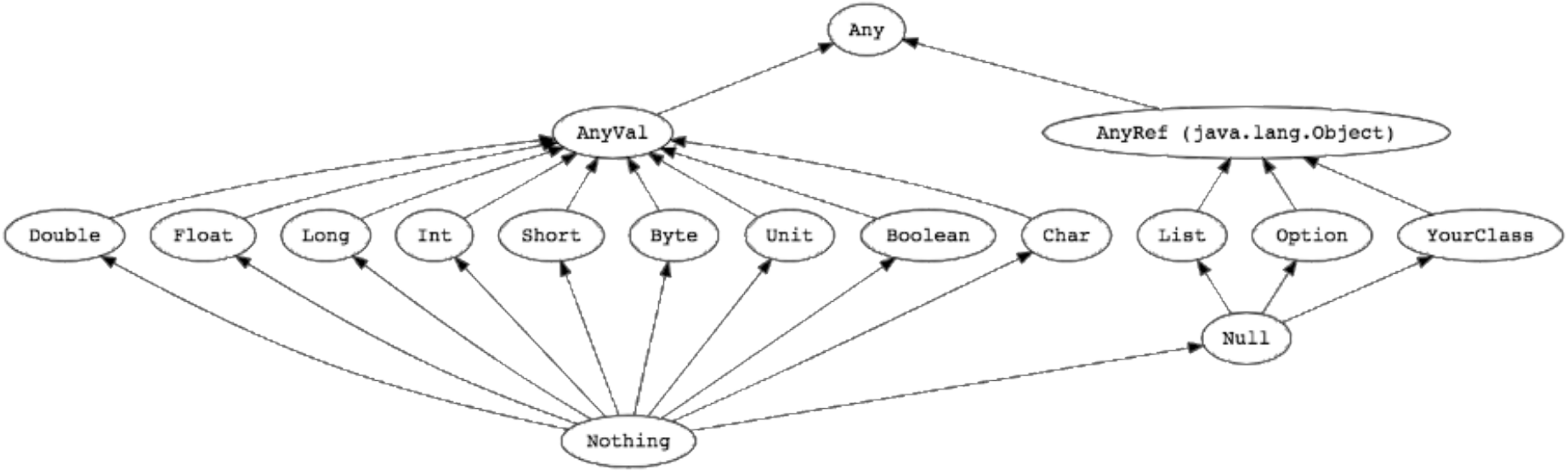
```
dataset.map(x => myCustomTranform(x))
```

Это выражение будет таким же для

- Scala-коллекций
- Параллельных Scala-коллекций
- Spark Datasets
- Flink DataStreams
- Потоков сообщений Akka

ООП:
Классы, объекты, трейты

Иерархия типов Scala



ООП объекты

- Класс (Class)
- Объект (Object)
- Трейт (Trait)

Класс

- По большей части такой же как в других языках
- Использование

```
val pt = new Point(10, 20)
// Move to a new location
pt.move(10, 10)
```

Класс

```
class Point(xc: Int, yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("Point x location : " + x)
      println ("Point y location : " + y)
   }
}
```

Класс

```
class Point(xc: Int, yc: Int) {
    var x: Int = xc
    var y: Int = yc

    def move(dx: Int, dy: Int) {
        x = x + dx
        y = y + dy
        println ("Point x location : " + x)
        println ("Point y location : " + y)
    }
}
```

Конструктор по умолчанию

Атрибуты

Методы

Дополнительные конструкторы

```
class LinkedList() {                                                    // конструктор 0
  var head = null
  var tail = null
  def isEmpty = tail != null
  def this(head: Int) = { this(); this.head = head }                    // конструктор 1
  def this(head: Int, tail: List[Int]) = { this(head); this.tail = tail }  // конструктор 2
}
```

Case классы

- Удобны для моделирования неизменяемых данных
- Сравнивает по структуре, а не по ссылке
- Имеет метод .copy для быстрого копирования неизменяемых объектов
- Не нужно ключевое слово “new” при создании

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

val garry = Person("Garry", 22)

val goodOldGarry = garry.copy(age=60)
```

Case classes

Case class широко используются для сериализации и описания данных

- Библиотеки JSON поддерживают автоматическое кодирование/декодирование CC
- Spark использует CC для описания типизированных наборов данных (Datasets)

Pattern matching + case class

```
val alice = Person("Alice", 25)
val bob = Person("Bob", 32)
val charlie = Person("Charlie", 32)

for (person <- List(alice, bob, charlie)) {
  person match {
    case Person("Alice", _) => println("Hi Alice!")
    case Person("Bob", 32) => println("Hi Bob!")
    case Person(name, age) => println(
      "Age: " + age + " year, name: " + name + "?")
  }
}
```

Объект

- Singleton = класс только с одним экземпляром
- Используется вместо статических методов в классе
- Приложение Scala - это **объект**

```
object Demo {
   def main(args: Array[String]) {
      val point = new Point(10, 20)
      println ("Point x location : " + point.x)
      println ("Point y location : " + point.y)
   }
}
```

Companion Object

- CO – объект с тем же именем что и класс
- Определен в том же исходном файле
- CO может обращаться к методам и полям, являющимся приватными в соответствующем Классе/Трейте

Случаи применения:

- Конструкторы
- Статические методы
- ...

Case класс всегда имеет свой CO (даже если вы его не определили)

Companion Object

```
class MyString(s: String) {
    private var extraData = ""
    override def toString = s + extraData
}
object MyString {
    def apply(base: String, extras: String) = {
        val s = new MyString(base)
        s.extraData = extras
        s
    }
    def apply(base:String) = new MyString(base)
}
println(MyString("hello","  world"))
println(MyString("hello"))
```

Трейт

- Трейт похож на Mixin или Interface в Java
- Использование:

```
class A extends B with TraitC with TraitD { ... }
```

- Основная задача – быть многоразовым для различных классов
- Для примера Seq – трейт

```
trait Seq[+A] extends PartialFunction[Int, A]
                    with Iterable[A]
                    with GenSeq[A]
                    with GenericTraversableTemplate[A, Seq]
                    with SeqLike[A, Seq[A]] { ... }
```

Трейт

Простой пример трейта

```
trait Equal {
   def isEqual(x: Any): Boolean
   def isNotEqual(x: Any): Boolean = !isEqual(x)
}
```

Пример

Абстрактный класс

Способ описать общие свойства и позволить потомкам реализовать их с помощью собственной логики

```
abstract class Animal {
  def name: String
}
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal
```

Наследование

```
abstract class A { val message: String }
class B extends A { val message = "I'm an instance of class B" }
trait C extends A { def loudMessage = message.toUpperCase() }
class D extends B with C

val d = new D
println(d.message)  // I'm an instance of class B
println(d.loudMessage)  // I'M AN INSTANCE OF CLASS B
```

Наследование

Как переопределить некоторую часть родительского класса?

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
  override def toString() = s"$re ${im}i"
}
```

Generic classes

Типы тоже могут быть параметрами

Пример

Это позволяет создавать классы с общей функциональностью для разных типов:

```
class Stack[A] { ... }
val stack = new Stack[Int]
val fruitStack = new Stack[Fruit]
```

Модификаторы доступа

Модификатор	Класс	Компаньон	Подкласс	Пакет	Мир
no modifier	Y	Y	Y	Y	Y
protected	Y	Y	Y	N	N
private	Y	Y	N	N	N

```
class A {
  def publicMethod = println("public")
  private def privateMethod = println("private")
}
```


57

Анонимный класс

Вы можете реализовать или расширить классы на лету

```
abstract class Fruit {
  val name: String
  def printName = println(s"It's $name!")
}
val apple = new Fruit { val name = "apple" }

apple.printName
> It’s apple!
```

«final» и «sealed»

- final классы не могут быть расширены
- sealed классы могут быть расширены только в том же файле

Дополнительная информация здесь:

https://stackoverflow.com/questions/32199989/what-are-the-differences-between-final-class-and-sealed-class-in-scala

Implicits

Implicit class

IC используется для неявного расширения других классов

```
object Helper {
    implicit class StringExtended(str: String) {
        def sayItLouder = println(str.toUpperCase +"!!!")
    }
}

"hi".sayItLouder    // Basic String has method “sayItLouder”
> HI!!!
```

Implicit conversions

Неявные преобразования применяются в двух ситуациях :

- Если выражение e имеет тип S, а S не соответствует выражениям предполагаемого типа T
- При вызове **e.m**, где экземпляр **e** имеет тип **S**, но у типа **S** отсутствует метод **m**

В первом случае ищется преобразование **c**, которое применимо к **e**, и тип результата которого соответствует **T**. Во втором случае ищется преобразование **c**, которое применимо к **e**, и результат которого содержит член с именем **m**.

Неявные преобразования в действии

```
val flag: Boolean = false
val sum: Int = flag + 1
println(sum)
```

```
ScalaFiddle.scala:2: error: type mismatch;
 found   : scala.this.Int(1)
 required: String
  val sum: Int = flag + 1
```

Неявные преобразования в действии

```
val flag: Boolean = false
implicit def bool2int(b: Boolean): Int = if (b) 1 else 0
val sum: Int = flag + 1
println(sum)

> 1
```

Неявные параметры

```
def rub2usd(sum: Double)
           (implicit rub2usd: Double) = sum * rub2usd

implicit val todayRUB2USD: Double = 60.0

println(rub2usd(10.0))
> 600.0
```

Полезные инструменты

Инструменты

Сейчас посмотрим несколько часто используемых инструментов

- SBT + IntelliJ IDEA
- Библиотеки для парсинга JSON
- Typesafe config / Pureconfig
- ScalaTest

Лайфхаки

Если вы хотите разобраться, какой результат генерит тот или иной кусок scala-кода, можно использовать два способа:

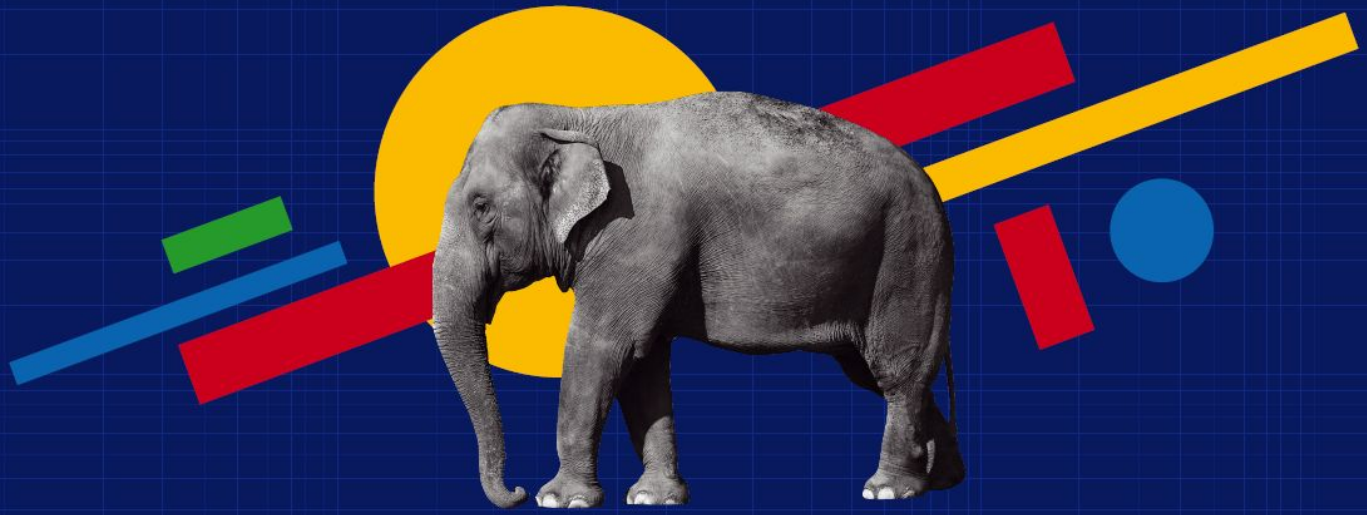
- `scalac -Xprint:all your-code.scala`
  покажет все шаги работы компилятора
- `javap your-code.class`
  покажет дизассемблированный код результата

Дополнительные материалы

Полезные ссылки

- Документация Scala для классов и объектов https://www.scala-lang.org/files/archive/spec/2.11/05-classes-and-objects.html
- О модификаторах доступа http://www.jesperdj.com/2016/01/08/scala-access-modifiers-and-qualifiers-in-detail/
- Некоторые изображения я взял из Alvin Alexander's Blog. Александр автор «Scala Cookbook» и на его сайте куча маленьких полезных рецептов Scala

Егор Матешук
egor@mateshuk.com



Big Data is Love

NEWPROLAB.COM