Введение в Scala

О курсе

Цель

- Научить основам языка
- Теоретическим основам функционального программирования
- Познакомить со стеком технологий
- Развить практические навыки программирования
- Научить работе в команде.

+

О курсе

- План
- Курс лекций. Разбит на 3 основные части
  - введение в Scala
  - углубленное изучение ключевых тем
  - стек технологий
- Практические занятия и самостоятельные работы
- Большое творческое задание

Ресурсы

Скачать

- Текущаяя версия scala
- IntelliJ IDEA
- Java Dev. Kit 1.8
- Клиент GIT + популярный GUI Tortoisegit для Win; Sourcetree для MAC
- SBT
- GITHub школы -

Введение

Классификация

- Реализация.
  - Интерпретируемые
  - Компилируемые (JIT, AOT)
- Требование к типам данных
  - Не типизированные
  - Строго типизированные
  - Строго типизированные с выводом типов
- Представление
  - Native
  - Virtual machine (JVM, LVM)

Введение

Классификация

- Парадигма
  - Императивные
  - ООП
  - Декларативные
  - Функциональные
  - Логические
  - Гибридные

Введение

Scala - язык программирования с множеством парадигм

- JVM Based
- JIT компиляция
- Продвинутый вывод типов (Hindley–Milner)
- Actors
- Императивный, объектно ориентированный
- Декларативный, функциональный

Введение

Примеры

Объектно ориентированный, императивный подход

```
class Executor(msg: String){
  def execute() = print(msg)
}

class ExecutorService {
 def execute(ex: Executor): Unit = {
   ex.execute()
}}

val es = new ExecutorService()
val e = new Executor("hello world")
es.execute(e)
```

Введение

Примеры

Декларативный, функциональный подход

```
def execute(msg: String):() => Unit = () => print(msg)

def executorService(thunk : () => Unit) = thunk()

executorService(execute("hello world"))
```

Введение

Примеры

Развитый вывод типов

```
def printSomething() = " - это 2 плюс 3"

def  calculateSomething() =  1 + 1

def result = calculateSomething + 3 + printSomething

result
```

Часть 1. Основы Scala

Часть 1. Конструкции языка

Пакет

- Задается инструкцией **package**
- Если присутствует, инструкция должна быть первой в файле
- Может быть указана только один раз
- Предназначен для
  - разделения приложения на компоненты
  - контроля за доступом к компонетам
  - уникальной идентификации приложения среди других приложений
- **package object** - альтернативный способ создания пакетов

```
package lectures
class LectureContent {
 def getContent() = {
   "Scala is AAAAWESOME"
 }}
```

Часть 1. Конструкции языка

Импорт

- Задается инструкцией **import**
- Делает возможным использование других компонентов в текущем скоупе
- Может быть указана в произвольном месте
- Инструкция для импорта
  - конкретного класса, объекта или типа и другого пакета

```
import lectures.LectureContent
```

  - списка компонентов

```
import lectures.{LectureContent, LectureContent2}
```

  - или всего содержимого пакета

```
import lectures._
```

  - внутренних компонент из объектов и пакетов

```
import lectures.LectureContent, LectureContent._
```

  - синонима пакета

```
import lectures.{LectureContent2 => LCC2}
```

Часть 1. Конструкции языка

Переменные

```
var variableName: SomeType = value
```

Константы

```
val variableName: SomeType = value
```

Ленивая инициализация

```
lazy val variableName: SomeType = value
```

Часть 1. Конструкции языка

Функции

```
def functionName(inputPrm: SomeType, otherPrm: SomeOtherType): ReturnType =
{
 // FUNCTION BODY
}

// WITH DEFAULT VALUE
def functionName(inputPrm: SomeType = defaultValue): ReturnType = { ...

// OMMIT RETURN TYPE
def functionName(inputPrm: SomeType, otherPrm: SomeOtherType) = { ...

// OMMIT RETURN TYPE AND BODY BRACES
def functionName(inputPrm: Int, otherPrm: Int) =  inputPrm + otherPrm
```

Часть 1. Конструкции языка

Процедуры

```
def procedureName(inputPrm: SomeType, otherPrm: SomeOtherType){
 // PROCEDURE BODY
}
def procedureName(inputPrm: SomeType, otherPrm: SomeOtherType): Unit = ???
```

Переменная длинна аргументов

```
def name(somePrm: Int, variablePrm: String*) = {
 // FUNCTION BODY
}

name(2, "a")

name(2, "a","б")

name(2, Seq("1","2","3","4"): _*)
```

Часть 1. Конструкции языка

Функции могут быть значениями

```
val myFun:(String) => Unit = (msg: String) => print(msg)
// или проще
val myFun = (msg: String) => print(msg)
// тоже, но без синтаксического сахара
val noSugarPlease: Function1[String, Unit] = (msg: String) =>
print(msg)
```

Функции можно передавать и возвращать из других функций, это, так называемые, функции высшего порядка

```
def printer(thunk: () => String): () => Unit =
  () => print(thunk())
```

Все параметры переданные в функции являются константами

Часть 1. Конструкции языка

Каррирование.

Еще один способ выразить в скале понятие функций высшего порядка

```
def curriedFilter(data1: String)(data2: String): Boolean =
 data1 == data2

// аналог curriedFilter(data1: String, data2: String)
val fullyApplied = curriedFilter("data")("data")

// аналог curriedFilter(data1: String): (String)=> Boolean
val partiallyApplied = curriedFilter("data")

val fullyAppliedAgain = partiallyApplied("data")
```

Часть 1. Конструкции языка

Call-by-name параметры или лень в помощь

```
def callByName(x: => Int) = ???
```

Параметры, переданные по имени имеют несколько особенностей

- вычисляются в теле функции только тогда, когда используются
- вычисляются при каждом вызове функций, в которую переданы
- не могу быть var или val

Часть 1. Конструкции языка

Разрешение циклических зависимостей

```
class Application {

 class ServiceA(c: => ServiceC){
   def getC = c
 }
 case class ServiceC(a: ServiceA)

 def a: ServiceA = new ServiceA(c)
 lazy val c: ServiceC = new ServiceC(a)
}

val app = new Application()
val a  = app.a
a.getC
```

Часть 1. Конструкции языка

Повторное вычисление

```
def callByValue(x: Int) = {
 println("x1=" + x)
 println("x2=" + x)
}
def callByName(x: => Int) = {
 println("x1=" + x)
 println("x2=" + x)
}
callByValue(something())
callByName(something())
```

Часть 1. Конструкции языка

Условный оператор

В скале есть только один условный оператор - **IF.** Тернарный оператор, как в JAVA отсутствует

```
val str = "good"
if (str == "bad") {
 print("everything is not so good")
} else if (str == "good") {
 print("much better")
} else {
 print("that's it. Perfect")
}
```

Часть 1. Конструкции языка

Циклы.

В scala 3 основных вида цикла

- **while** - повторяет свое тело пока выполняется условие
- **for** - итерируется по переданной в оператор коллекции или интервалу (Range)
  - в одном операторе можно итерироваться сразу по нескольким коллекциям
  - оператор позволяет фильтровать члены коллекции, по которым итерируется, с помощью встроенного оператора if
  - оператор позволяет определять переменные между вложенными циклами

- **for {} yield {}**. Если перед телом цикла стоит слово **yield**, то цикл становится оператором, возвращающим коллекцию. Тип элементов в итоговой коллекции зависит от типа возвращаемого телом цикла

```
while(condition){
 statement(s);
}
```

Часть 1. Конструкции языка

```
val myArray = Array(
 Array("пельмени","очень","вредная","еда"),
 Array("бетон ","крепче дерева"),
 Array("scala","вообще","не","еда"),
 Array("скорее","бы","в","отпуск")
)

for (anArray: Array[String] <- myArray;
    aString: String <- anArray;
    aStringUC = aString.toUpperCase()
    if aStringUC.indexOf("ЕДА") != -1
) {
 println(aString)
}
```

Часть 1. Конструкции языка

```
val myArray = Array(
 Array("пельмени","очень","вреднаяя","еда"),
 Array("бетон ","крепче дерева"),
 Array("scala","вообще","не","еда"),
 Array("скорее","бы","в","отпуск")
)

val foodArray: Array[String] =
 for (anArray: Array[String] <- myArray;
      aString: String <- anArray;
      aStringUC = aString.toUpperCase()
      if aStringUC.indexOf("ЕДА") != -1
 ) yield {
   aString
 }
```

Часть 1. Конструкции языка

Класс

Это конструкция языка, которая описывает новый тип сущности в приложении.

- способ создания объекта класса описывается в конструкторе
- новый объект класса создаеться с помощью оператора **new**
- членами класса могут методы, переменные, константы, другие классы объекты и трейты
- класс может содержать произвольное количество членов
- класс может быть связан с другими классами объектами и трейтами отношением наследования
- доступ к членам класса определяется модификаторами доступа
    - **private -** член класса доступен только внутри класса
    - **protected -** член класса доступен только внутри класса и его наследниках
    - **public -** уровень доступа по умолчанию, если модификатор не указан. Член класса может быть доступен в любом месте приложения

Часть 1. Конструкции языка

```
class TestClass (val int: Int, var str: String, inner: Long) {

 def publicMethod() {
   print("public method")
 }
 // This constructor inaccessible from outside
 private def privateMethod() {
   print("private method")
 }
}

val testClassInstance = new TestClass(1, "", 0l)

testClassInstance.int
testClassInstance.str
testClassInstance.publicMethod()

// inner is not a member of the class
//testClassInstance.inner

// inaccessible from outside
//testClassInstance.privateMethod()
```

Часть 1. Конструкции языка

Конструктор

- класс должен иметь как минимум один конструктор. Этот конструктор в документации обычно называют главный конструктор или **primary constructor**
- телом главного конструктора является тело самого класса
- любой конструктор может быть primary, public или protected
- тело любого конструктора, кроме главного, должно начинаться с вызова главного конструктора
- члены класса могут быть описаны в сигнатуре главного конструктора, если их описание начинается с val или var
- вторичные конструкторы не могут определять новых членов класса
- все параметры переданные в конструктор без модификатора не являются членами класса, но могут использоваться в имплементации класса

Часть 1. Конструкции языка

Конструктор

```
class TestClass private(val int: Int, var str: String, inner: Long) {

 private var member = 0

 def this(int: Int, str: String) {
   //print("would throw an exception")
   this(int, str, 0)
 }

 // This constructor inaccessible from outside
 def this(int: Int, str: String, inner: Long, member: Int) {
   this(int, str, 0)
   this.member = member
 }
}
```

Часть 1. Конструкции языка

Абстрактный класс

- это класс, у которого один или более членов имеют описание но не имеют определения
- абстрактный класс описывают с помощью ключевого слова **abstract**
- для создания объекта абстрактного класса нужно доопределить все члены класса
- это можно сделать
    - в наследниках класса
    - с помощью сокращенного синтаксиса

```
abstract class TestAbstractClass(val int: Int) {
 def abstractMethod(): Int
}
//  сокращенный синтаксис
new TestAbstractClass(1) {
 override def abstractMethod(): Int = ???
}
```

Часть 1. Конструкции языка

Объекты. Объекты компаньоны

- объекты - это классы с единственным инстансом, созданным компилятором
- членами объекта могут быть константы, переменные, методы и функции. А так же виртуальные типы и другие объекты.
- объекты могут наследоваться от классов и трейтов и объектов
- если объект и класс имеют оно название и определены в одном файле они называются компаньонами

```
trait TestObject{

 val name = "Scala object example"

 class InnerClass

 val innerInstance = new InnerClass

 def printInnerInstance() =  print(innerInstance)
}
```

Часть 1. Конструкции языка

Apply, unapply и немного волшебства

- если объект имеет метод apply, то он может быть использован как функция фабрика
- объект компаньон часто используют для создания класса с помощью метода apply
- метод unapply применяеться в паттерн мэтчинге и для получения членов класса в оперциях присвоения. Подробнее про unapply будет рассказано в разделе про pattern matching

```
object TestWithApply{
 def apply(): TestClassWithApply = new TestClassWithApply()
 def unapply(arg: TestClassWithApply): Option[Unit] = ???
}

class TestClassWithApply(){}

TestWithApply()
```

Часть 1. Конструкции языка

Trait

- это конструкция языка, определяющая новый тип через описание набора своих членов
- может содержать как определенные , так и не определенные члены
- не может иметь самостоятельных инстансов
- не может иметь конструктор
- применяется главным образом для реализации парадигмы множественного наследования.

```
trait Similarity {
 def isSimilar(x: Any): Boolean
 def isNotSimilar(x: Any): Boolean = !isSimilar(x)
}
```

Часть 1. Конструкции языка

Кейс классы

Это классы которые компилятор наделяет дополнительными свойствами. Кейс классы удобны для создания иммутабильных конструкций, сопоставления с образцом и передачи кортежей данных…

Отличия от стандартных классов

- каждый член класса - публичный val
- для кейс классов компилятор переопределяет метод equals
- создается объект компаньон с методами apply и unapply
- от кейс класса нельзя наследоваться
- в кейс классе есть метод copy
- не рекомендуется определять
  - кейс классы без членов
  - несколько конструкторов с разной сигнатурой

Часть 1. Конструкции языка

```
//Good case class
case class ForGreaterGood(someGoody: String)

//COMPILATION ERROR
case class SuperClass(int: Int)
case class SubClass(int: Int) extends SuperClass(int)

//COMPILATION ERROR
case class NoMembers

// Don't do this
case class BadSignature(int: Int) {
 def this(int: Int, long: Long) = {
   this(int)
 }
}
```

Часть 1. Исключительные ситуации

Исключительные ситуации

В scala, по сути, они аналогичны исключительным ситуациям в Java. Подробнее о исключительных ситуациях можно прочитать здесь. Ключевые отличия заключаются в том, что методы в скале не требуют указания checked исключений в своей сигнатуре. Так же отличаются конструкции языка для их обработки.

Если есть необходимость обозначить, что какой-либо метод может бросать исключительную ситуацию, можно использовать аннотацию **@throws**

Для того, что бы вызвать исключительную ситуацию нужно использовать оператор **throw**

Часть 1. Исключительные ситуации

```
class TestClass {

 @throws[Exception]("Because i can")
 def methodWithException(): Int =
   throw new Exception("Exception thrown")

 def methodWithoutException() = {
   print(methodWithException())
 }
}

val t = new TestClass()

// Method would throw an exception
t.methodWithoutException()
```

Часть 1. Исключительные ситуации

Обработка исключений

Существует 2 принципиально разных подхода: императивный и функциональный

Императивный подход с применением конструкции **try { } catch { } finally {}**

- внутри **try** размещается потенциально опасный код
- **catch** - опционален. В нем перечисляются типы исключительных ситуаций и соответствующие обработчики
- **finally**, тоже опционален. Если этот блок присутствует, он будет вызван в любом случае, независимо от того, было ли перехвачено исключение или нет

Часть 1. Исключительные ситуации

```
import java.sql.SQLException

class TestClass {

 @throws[Exception]("Because i can")
 def methodWithException(): Int =
   throw new Exception("Exception thrown")

 def methodWithoutException(): Unit =
   try {
     print(methodWithException())
   } catch {
     case e: SQLException  => print("sql Exception")
     case e: Exception => print(e.getMessage)
     case _ => print("would catch even fatal exceptions")
   } finally {
     println("Ooooh finally")
   }
}

val t = new TestClass()
// Method would throw an exception
t.methodWithoutException()
```

Часть 1. Исключительные ситуации

Обработка исключений

Функциональный подход может быть реализован несколькими способами. Наиболее популярный - с использованием **Try[T]**. В отличии от **try{}, Try[T]** - это объект, а не ключевое слово

- потенциально опасная часть кода размещается в фигурных скобках после **Try[T]**
- в **Try[T],** T - это тип результата, части кода, переданной в **Try[T]**
- **Try[T]** имеет 2- наследников
    - **Success[T]**. Объект этого типа будет создан, если код завершился без ошибок
    - **Failure[Throwable]**. Объект этого типа будет создан, если был выброшен Exception

- **Try[T]** имеет набор методов для обработки полученного результата или выброшенного исключения

Одним из минусов **Try[T]**, является отсутствие среди методов аналога **finally**
В невозможно перехватить фатальные ошибки, такие как OutOfMemoryException

Часть 1. Исключительные ситуации

```
class TestClass {

 @throws[Exception]("Because i can")
 def methodWithException(): Int =
   throw new Exception("Exception thrown")

 def methodWithoutException(): Try[Unit] =
   Try {
     print(methodWithException())
   }. recover {
     case e: SQLException  => print("sql Exception")
     case e: Exception => print(e.getMessage)
     case _ => print("would catch even fatal exceptions")
   }.map{
     case _ => println("Ooooh finally")
   }
}
```

Часть 1. Коллекции

Обзор коллекций

- большинство коллекции в scala находятся в пакете **scala.collection**
- пакет разделяет коллекции на 3 категории
    - в корне пакета **scala.collection** находятся корневые трейты коллекций
    - в пакете **scala.collection.immutable** находятся иммутабльные реализации коллекций
    - в пакете **scala.collection.mutable** находятся мутабильные реализации**.** Т.е. реализации коллекций, которые можно модифицировать не создавая новую копию исходной коллекции

- В корне иерархии коллекций находится трейт Traversable[+A]. Большинство методов, которые нам понадобятся, определены в нем

Часть 1. Коллекции

Методы Traversable

- конкатенация, **++**, объединяет 2 коллекции вместе
- операции **map, flatMap**, и **collect**, создают новую коллекцию, применяя функцию к каждому элементу коллекции.
- методы конвертации **toArray, toList, toIterable, toSeq, toIndexedSeq, toStream, toSet, toMap**
- информация о размере **isEmpty, nonEmpty, size**
- получение членов коллекций **head, last, headOption, lastOption,** и **find**.
- получение субколлекции **tail, init, slice, take, drop, takeWhile, dropWhile, filter, filterNot, withFilter**
- разделение и группировка **splitAt, span, partition, groupBy**
- проверка условия **exists, forall**
- операции свертки **foldLeft, foldRight, reduceLeft, reduceRight**

Часть 1. Коллекции

Часто используемые коллекции.

Для большинства часто используемых коллекций в scala есть короткие синонимы. Чаще всего короткий синоним ведет к иммутабильной версии коллекции

- **Set[A]** - набор уникальныйх элементов типа **A**
- **Map[A, +B]** - ассоциативный массив с ключами типа **A** и значениями типа **B**
- **List[A]** - связный список элементов, типа **A**
- **Array[A]** - массив элементов типа **A**
- **Range -** целочисленный интервал. **1 to N** - создает интервал, включающий N, **1 until N**, не включающий N

Часть 1. Коллекции

```
// размер сета
Set(1,2,3,4).size
Set(1,2,3,4,4).size

// разделить все элемены на 2
List(1,2,3,4,5,6,7,8,9,0).map(_ % 2)
List(1,2,3,4,5,6,7,8,9,11,12,13,14,15,16,17,0).foldLeft(0)((acc, item) => acc + item %
3)

//Инервал
val r =  1 to 100
r.foreach(print(_))

// Map
val letterPosition = Map("a" -> 1, "b" -> 2, "c" -> 3, "d" -> 4)
letterPosition("a")
// throw ann exception
letterPosition("g")
letterPosition.get("g")// == None
```

Часть 1. Коллекции

Option. Some. None.

Option[T] - это тип, который отражает факт неопределенности наличия элемента типа T в этой части приложения. Применение **Option** - очень эффективный метод избавиться от NPE.

Option[T] имеет 2 наследника: Some и None

- **Some[T]** - говорит о наличии элемента
- **None** - об отсутствии
- **Option(String) == Some[String](String)**
- **Option(null) == None**
- **Some(null) == Some[Null](null)**

```
def eliminateNulls(maybeNull: String): Option[String] =
Option(maybeNull)

def returnEven(int: Int): Option[Int] =
 if (int % 2 == 0) Some(int)
 else None
```

Часть 1. Задания

Исправить ошибку

Дан список **List[Int],** в нем необходимо

- все нечетные числа умножить на 2
- подсчитать сумму получившихся членов списка

```
val mapper = (s: String) => if( i%2 != 0) i * 2

List(1, 2, 3, 4, 5, 6, 7, 8, 9).map {
 mapper
}.foldLeft(0) { (acc, v) => acc + v }
```

Часть 1. Задания

Исправить ошибку

Дана заготовка наивной реализации подсчета чисел Фибоначчи.
Необходимо исправить код и вывести 9-ое число Фибоначчи

```
def fibs(num: Int) = {
 if(num == 1) 1
 if(num == 2 ) 2
 fibs(num - 1) + fibs(num - 2)
}

print(fibs(9))
```

Часть 1. Задания

Обработать исключения

Код ниже может породить несколько исключительных ситуаций. Внутри метода **printGreetings** нужно написать обработчик для каждого конкретного типа исключения. Обработчик должен выводить текстовое описание ошибки. Счетчик в методе должен пройти все значения от 0 до 10

```
object PrintGreetings {

 case class Greeting(msg: String)

 private val data = Array(Greeting("Hi"), Greeting("Hello"),
   Greeting("Good morning"), Greeting("Good afternoon"),
   null, null)

 def printGreetings() = {
   for (i <- 0 to 10) {
     println(data(i).msg)
   }
 }
}

PrintGreetings.printGreetings()
```

Часть 1. Задания

Написать сортировку слиянием.

Постарайтесь не использовать мутабильные коллекции и **var**
Подробнее о сортировке можно подсмотреть здесь

```
// merge sort root method
def mergeSort(data: List[Int]): List[Int] = ???
```

Часть 1. Задания

Реализовать простое бинарное дерево поиска.

Это должна быть структура данных, реализуемая кейс классом **ScalaTree**, приведенным ниже.

Со следующими свойствами:

- левая ветка содержит значения, меньшие значения родителя
- правая ветка содержит значения, большие значения родителя
- значения, уже присутствующие в дереве, в него не добавляются
- пустые значения(**null**) не допускаются
- добавлять новые ноды можно только в корень дерева

Для этой структуры нужно реализовать генератор узлов. Генератор не должен использовать переменные или мутабильные структуры.

```
case class ScalaTree(value: Int, left: Option[ScalaTree], right: Option[ScalaTree]) {
 def add(newValue: Int): ScalaTree = ???
}
```

Часть 1. Задания на свободное время

Задача 1. Числа Фибоначчи еще раз

Написать алгоритм подсчета чисел Фибоначчи с использованием аккумулятора подсчитанных значений

Запрещено использовать переменные и мутабильные коллекции

Задача 2. Доработать дерево. Обход

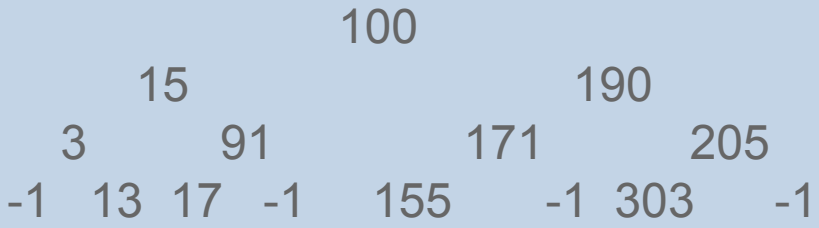
Добавить в дерево обход в ширину и по уровням

Задача 3. Доработать дерево. Метод **toString**

Дерево - сложная структура, поэтому хорошо бы иметь для нее красивое визуальне представление. Для этого нужно переопределить метод **toString.**

Ниде пример распечатанного дерева.

Часть 1. Задания на свободное время



Здесь, **-1** обозначает, что потомок отстствует

Задача 4. Методы map и fold для дерева

Для нашего дерева нужно определить методы обхода со следующими сигнатурами

- **def map(f: (Int) => (Int)): ScalaTree**. Метод должен обойти все узлы дерева, применив к их значениям метод трансформации. На выходе должно быть получено дерево содержащее узлы с трансформированными значениями
- **def fold(aggregator: Int)(f: (Int, Int) =>(Int))**. Метод предназначен агрегирования значений улов дерева. Например, с его помощью можно вычислить сумму значений всех узлов.