



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

- [Текущая 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
```

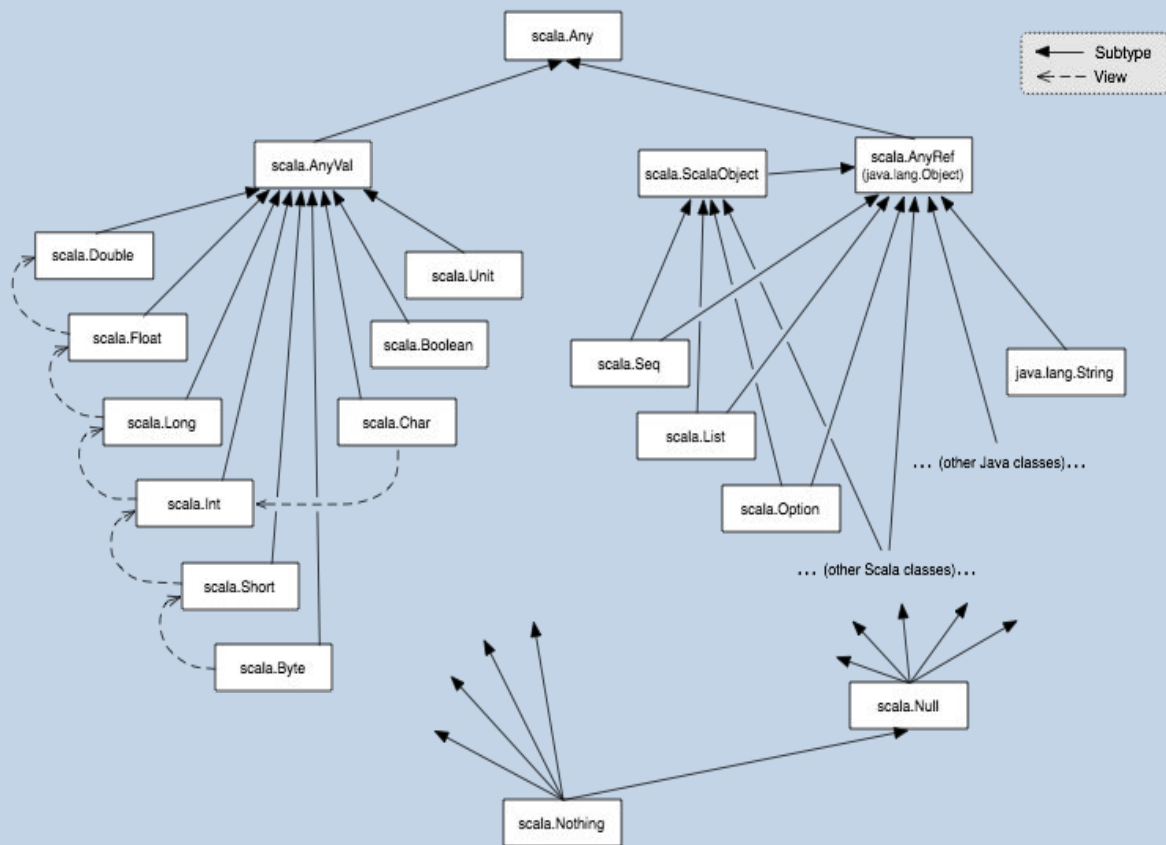
```
def result = calculateSomething + 3 + printSomething
```

```
result
```



Часть 1. Основы Scala

Часть 1. Типы



Часть 1. Типы

```
val set = new scala.collection.mutable.HashSet[Any]
set += "This is a string" // add a string
set += 732 // add a number
set += 'c' // add a character
set += true // add a boolean value
set += printContent _
// add the main function
val iter: Iterator[Any] = set.toIterator

def printContent() {
  for (i <- iter) {
    println(i)
  }
}
printContent()
```

Часть 1. Типы.

Вывод типов

Скала имеет продвинутую систему вывода типов. Это значит, что если выражение строится на основе структур с известными типа, то компилятор сам сможет определить тип возвращаемого результата.

Для членов коллекций, арифметических и др. операций компилятор определит типа, как ближайший общий родитель (см. схему выше)

Разработчик должен воспринимать систему типов, как возможность, воспользовавшись компилятором, доказать правильность, написанного кода.

Часть 1. Типы

Вывод типов

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

```
def calculateSomething() = 1 + 1
```

```
// compiler convert operands into their nearest common ancestor
```

```
// for each operation individually
```

```
// type conversion is left associative
```

```
def result = calculateSomething + 3 + printSomething
```

```
// Compiler use view to convert Int and Long into float
```

```
val numericList: List[Float] = List(1, 1l, 0f)
```

Часть 1. Типы

Вывод типов

Синтаксический сахар, связанный с выводом типов

```
val fullNotion: List[Float] = List[Float](1,2,0f)
```

```
val shortNotion = List(1, 1f, 0f)
```

```
def fullNotionFunction(): List[Float] = {  
  shortNotion  
}
```

```
def shortNotionFunction() = shortNotion
```


Часть 1. Типы

Вывод типов

Когда вывод типов не работает

- когда неизвестен как минимум один из типов участвующий в операции
- когда у рекурсивных функции, не указан явно возвращаемый тип
- для входных атрибутов функций

Часть 1. Типы. Задания

Объяснить вывод типов

lectures.types.TypeInference

Исправить компиляцию

lectures.types.FixCompile

В скале есть выражение - ????. Объясните, что делает метод и почему выражение ниже компилируется.

```
def someFunction(prm1: Int, prm2:String): Option[Int] = ???
```

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 компонентам
 - уникальной идентификации приложения среди других приложений
- **package object** - альтернативный способ создания пакетов

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

Часть 1. Импорт

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. Определения

Переменные

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

Константы

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

Ленивая инициализация

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

Часть 1. Функции

Функции

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

// WITH DEFAULT VALUE

```
def functionName(inputPrm: SomeType = defaultValue): ReturnType = { ...
```

// OMIT RETURN TYPE

```
def functionName(inputPrm: SomeType, otherPrm: SomeOtherType) = { ...
```

// OMIT RETURN TYPE AND BODY BRACES

```
def functionName(inputPrm: Int, otherPrm: Int) = inputPrm + otherPrm
```

Значением функции, является значение последнего в ней выражения

Часть 1. Функции

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", "6")
```

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

Часть 1. Функции

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. Функции

Каррирование.

Еще один способ выразить в скале понятие функций высшего порядка

```
def notCurriedFilter(data1: String): (String)=> Boolean =  
  (data2: String) => data1 == data2
```

// каррированный аналог предыдущей функции

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

```
val fullyApplied = curriedFilter("data1")("data2")
```

```
val partiallyApplied = curriedFilter("data1") _
```

```
val fullyAppliedAgain = partiallyApplied("data2")
```

Часть 1. Функции

Композиция функций одной переменной

Для функции одной переменной определены комбинаторы функций **compose** и **andThen**. Комбинаторы - это функции, позволяющие объединить 2 и более функций в одну. При этом комбинаторы задают последовательность, в которой будут выполняться тела, комбинируемых функций

- **def compose[A](g : scala.Function1[A, T1]) : scala.Function1[A, R]** - принимает функцию, которая будет выполнена перед текущей. Результат переданной функции будет передан на вход текущей
- **def andThen[A](g : scala.Function1[R, A]) : scala.Function1[T1, A]** - аналогична **compose**, но переданная функция будет выполнена после текущей

```
val pow = (int: Int) => int * int
def show(int: Int) = print(s"Square is $int")
//val powAndShow = pow compose show
val powAndShow = pow andThen show
```

```
powAndShow(10)
```

Часть 1. Функции

Композиция функций нескольких переменных

Функции нескольких переменных не имеют комбинаторов, аналогичных функциям одной переменной. Для того, чтобы иметь возможность комбинировать функции нескольких переменных, необходимо свести их к функции одной переменной. Это можно сделать 2-мя способами.

Рассмотрим их на примере функции от 2-х переменных

- **def curried : scala.Function1[T1, scala.Function1[T2, R]]** - каррирует функцию. Т.е. возвращает функцию, которая на вход принимает первый параметр, а на выход возвращает функцию, принимающую второй параметр исходной функции
- **def tupled : scala.Function1[scala.Tuple2[T1, T2], R]** - объединяет все параметры функции в один параметр в виде scala Tuple. Мы рассмотрим этот метод чуть позже, когда будем изучать tuples

Композировать функции удобно, когда есть набор стандартных функций, которые нужно выполнить в определенном порядке. Композиция функций позволяет писать очень выразительный код и часто применяется для написания DSL

Часть 1. Функции

Композиция функций нескольких переменных

Представим, что перед выполнением функции multiply нам надо распечатать входные параметры. Для этого воспользуемся композицией функций

```
val multiply = (i:Int, j: Int) => i * j
val setOperand = multiply.curried
def printOperand[T](a: T) = {println(s"operand is $a "); a}
def printResult[T](a: T) = {println(s"And a result is $a "); a}
def executeWith[T](t: T) = t
def mulitplyWithPrinter(i: Int, j: Int) =
  ((printOperand[Int] _ andThen setOperand)(i) compose
   printOperand[Int] andThen printResult)(j)
// ((printOperand[Int] _ andThen setOperand)(i)
// породит функцию (j : Int) => {
//   println(s"operand is 10" )
//   (j) => 10 * j
// }
mulitplyWithPrinter(11,20)
```

Часть 1. Функции

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. Функции

Разрешение циклических зависимостей

```
class Application {  
  
  class ServiceA(c: => ServiceC){  
    def getC = c  
  }  
  class ServiceC(val 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. Функции

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. Функции. Задания

Подсчитать числа Фибоначчи

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

lectures.functions.Fibonacci

Реализовать более эффективный способ вычисления чисел Фибоначчи

lectures.functions.Fibonacci2

Освоить каррирование и функции высшего порядка

**lectures.functions.Computation, lectures.functions.CurriedComputation,
lectures.functions.FunctionalComputation**

Воспользоваться композицией функций для написания простого DB API

lectures.functions.SQLAPI

Часть 1. Операторы

Условный оператор

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

Еще один важный способ организовать ветвление - это сопоставление с образцом (pattern matching). Мы рассмотрим подробно, отдельно в одной из следующих лекций.

```
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. Операторы

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. Операторы

// ВЫВЕДЕТ ВСЕ ЧИСЛА ВКЛЮЧАЯ 100

```
for(i <- 1 to 100){  
  print(i)  
}
```

// ВЫВЕДЕТ ВСЕ ЧИСЛА ИСКЛЮЧАЯ 100

```
for(i <- 1 until 100){  
  print(i)  
}
```

Часть 1. Операторы

```
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. Операторы

```
val myArray = Array(  
  Array("пельмени", "очень", "вредная", "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. Операторы. Задания

По тренируйтесь в написании циклов и условных операторов

lectures.operators.Competition

Допишите программу из **lectures.operators.EvaluateOptimization**, что бы оценить качество оптимизации из предыдущей задачи

Часть 1. Pattern matching

Сопоставление с образцом(pattern matching) - удобный способ ветвления логики приложения. Чаще всего операция сопоставления выглядит примерно вот так:

```
val x:Int = 10

// By value
val stringValue = x match {
  case 1 => "one"
  case 10 => "ten"
}
```

match, указанный после переменной, указывает на начало операции сопоставления, а ключевые слова **case** определяют образцы, с которыми производится сопоставление

В этом примере будет выбрана ветка “**ten**”

Оператор сопоставления - это полноценное выражение, имеющее возвращаемый тип, определяемый компилятором, как ближайший общий предок для значений всех веток. В данном случае **stringValue** - будет равно “**ten**”

Часть 1. Pattern matching

- Сопоставление идет до первого подошедшего **case**, а не до самого подходящего.
- Pattern matching is exhaustive(исчерпывающий), это значит, что если подходящая ветка обязательно должна быть определена, иначе произойдет исключительная ситуация (Exception).
- Можно указать default case с помощью конструкции **case _ =>**

```
val x:Int = 10
```

```
// "Something" would be chosen despite that '10' is more precise
```

```
x match {  
  case _ => "Something"  
  case 10 => "ten"  
}
```

```
// Compilation error, no matching case
```

```
val stringValue = x match {  
  case 1 => "one"  
  case 11 => "eleven"  
}
```


Часть 1. Pattern matching

Возможности Pattern matching в scala

- сопоставление по значению
- сопоставление по типу
- дополнительные IF внутри case
- объединение нескольких case в один с помощью |
- объявление синонима сопоставленному образцу с помощью @
- сопоставление с regexp
- задание области определения для PartialFunction
- использование функций экстаркторов(unapply)

```
val c: Any = "string"
```

```
c match {  
  case "string" | "otherstring" => "exact match"  
  case c: String if c == "string" || c == "otherstring" => "type match, does the same as  
the previous case"  
  case i: Int => "won't match, because c is a string"  
  case everything @ _ => print(everything)  
}
```

Часть 1. Pattern matching

Pattern matching для кейс классов

```
case class Address(city: String, country: String, street: String, building: Int)
```

```
val kremlin = Address("Russia", "Moscow", "Kremlin", 1)
```

```
val whiteHouse = Address("USA", "Washington DC", "Pennsylvania Avenue",  
1600)
```

```
kremlin match {  
  case inRussian@Address("Russia", _, _, _) => print(inRussian.city)  
  case inUSA@Address("USA", _, _, _) => print(inUSA.city)  
  case somewhereElse@_ => print("Terra incognita!")  
}
```

Часть 1. Pattern matching

Pattern matching для коллекций

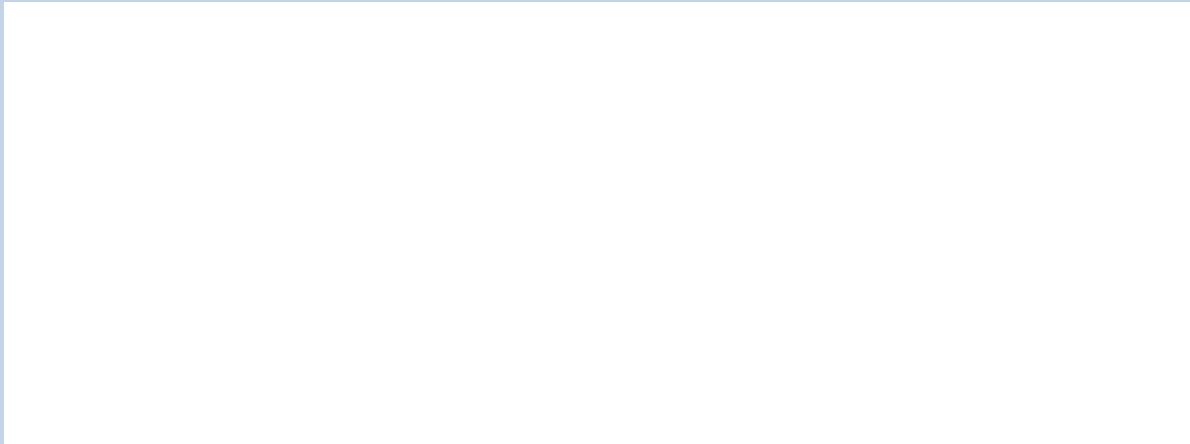
```
// print list in reverse order
val list = List(1, 2, 3, 4, 5, 6, 7, 8, 100500)
def printList(list: List[Int]): Unit = list match {
  case head :: Nil => println(head)
  case head :: tail =>
    printList(tail)
    println(head)
} // TODO fix compilation warning

printList(list)
```

Сопоставление с образцом работает для коллекций и кейс классов благодаря методу `unapply` в объектах компаньонах. Подробнее этот механизм рассмотрен чуть ниже в разделе, посвященном объектам.

Часть 1. Pattern matching

Partial functions



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 создается с помощью оператора **new**
- членами класса могут быть 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

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

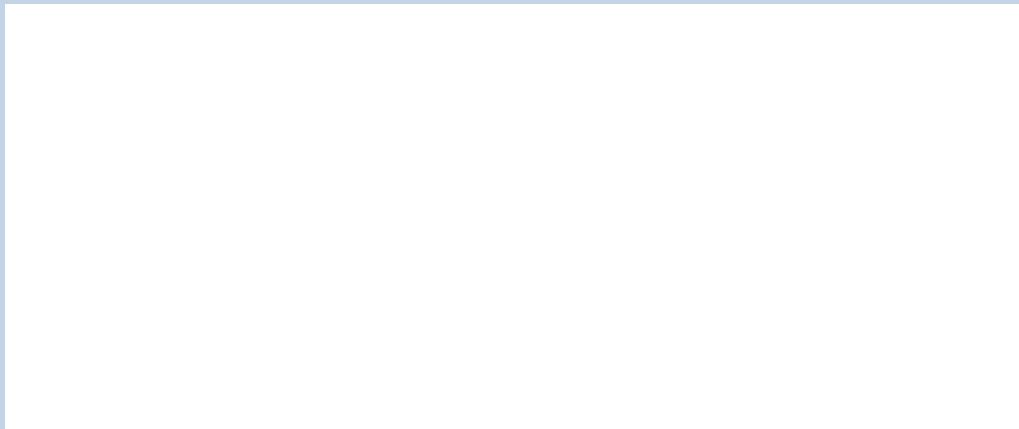
Объекты. Объекты компаньоны

- объекты - это классы с единственным экземпляром, 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 одно название и определены в одном файле они называются компаньонами

```
object TestObject{  
  
  val name = "Scala object example"  
  
  class InnerClass  
  
  val innerInstance = new InnerClass  
  
  def printInnerInstance() = print(innerInstance)  
}
```

Часть 1. Конструкции языка

Чем полезны объекты-компаньоны



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` применяется в паттерн мэтчинге и для получения членов класса в операциях 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

Кейс классы

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

Отличия от стандартных классов

- каждый член класса - публичный **val**
- для кейс классов компилятор переопределяет метод **equals** и **toString**
- 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 иммутабельные реализации коллекций
 - в пакете **scala.collection.mutable** находятся мутабельные 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. Лекция 2. 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**, **toListerable**, **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 иммутабельной версии коллекции

- **Set[A]** - набор уникальных 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 элементы 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)
```

//Интервал

```
val r = 1 to 100
```

```
r.foreach(print(_))
```

// Map

```
val letterPosition = Map("a" -> 1, "b" -> 2, "c" -> 3, "d" -> 4)
```

```
letterPosition("a")
```

// throw an 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) == SomeString**
- **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

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 мутабельные коллекции и **var**
Подробнее о сортировке можно посмотреть [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 мутабельные 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 мутабельные коллекции

Задача 2. Доработать дерево. Обход

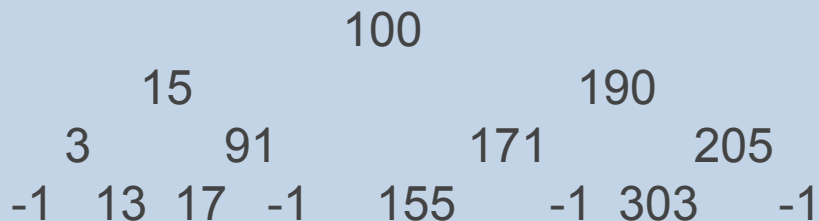
Добавить в дерево обход в ширину и по уровням

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

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

Ниде пример распечатанного дерева.

Часть 1. Задания на свободное время



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

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 узлов дерева. Например, с его помощью можно вычислить сумму значений всех узлов.