Введение в Scala

Контакты

Лекторы

- Попов Сергей; **s.popov2@tinkoff.ru**
- Титаренко Артем; **a.titarenko@tinkoff.ru**

Группа в telegram

- https://t.me/joinchat/EscdIQzviMCQjkUOEZMr6w

Github

- https://github.com/sergeypopov83/Scala-complete-course

О курсе

Цель

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

+

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
- Развитый type polymorphism
- implicit conversion and implicit evidence
- Корекурсия (Streams)

Введение

Примеры

Объектно ориентированный

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

val e = new Executor()
e.apply("hello world")
```

Введение

Примеры

Функциональный подход

```
val func = (str: String) => print(msg)

func("hello world")
```

Введение

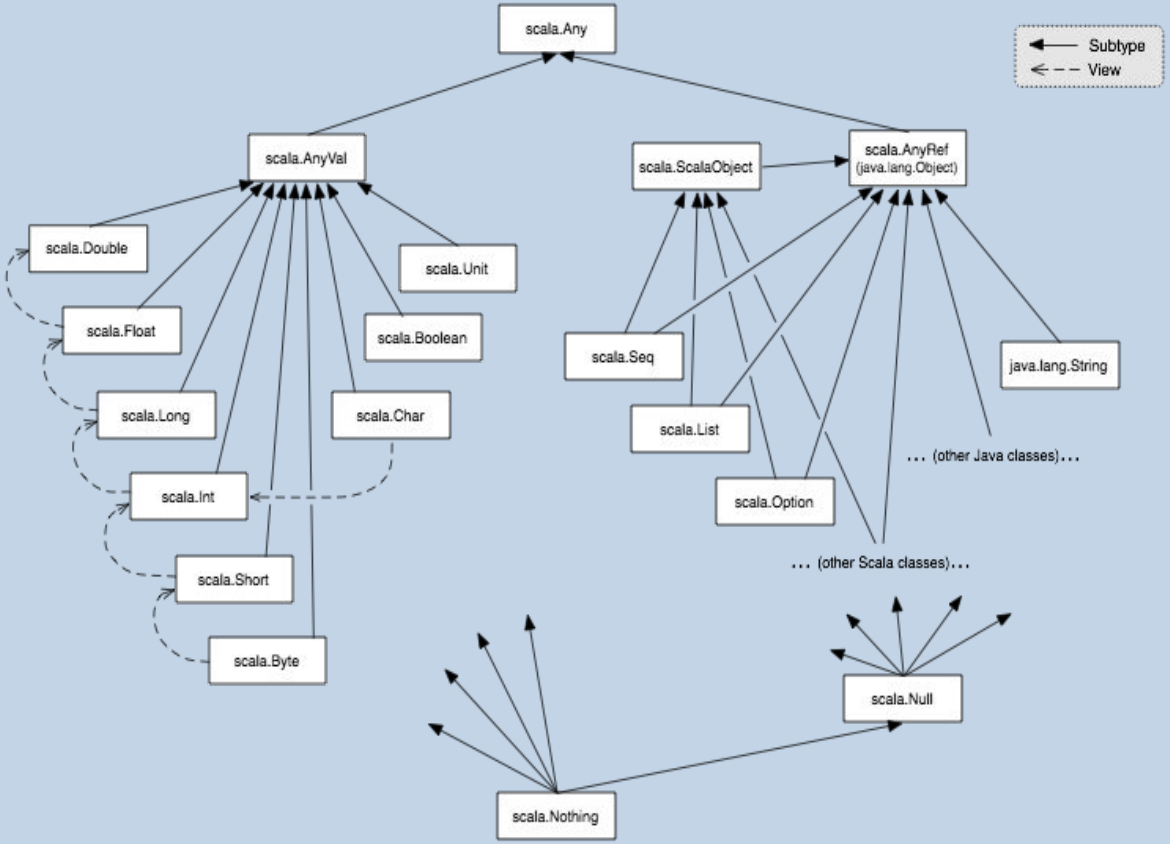
Примеры

Развитый вывод типов

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

def  calculateSomething() =  1 + 1

def result = calculateSomething + 3 + printSomething

result
```

Основы Scala

Часть 1. Типы


scala.Any
Subtype
View
scala.AnyVal
scala.ScalaObject
scala.AnyRef
(java.lang.Object)
scala.Double
scala.Unit
scala.Float
scala.Boolean
scala.Long
scala.Char
scala.Int
scala.Short
scala.Byte
scala.Seq
java.lang.String
scala.List
... (other Java classes)...
scala.Option
... (other Scala classes)...
scala.Null
scala.Nothing

Часть 1. Типы

```
val c = 'd'
val c2: Char = 'd'
val cA: Any = 'd'
// cI - иметь значение - 100
val cI: Int = 'd'
// cI2 - будет иметь значение 'd'
val cI2: Char = 100
val cD: Float = 'd'
// Won't compile
//val cD2: Char = 100.0
// Won't compile
//val cN: Nothing = 'd'
// SET FLOAT
val set  = Set(1, 'c', 20f)
// SET AnyVal
val set2  = Set(1, 'c', 20f, true)
// Set Any
val set3  = Set(1, 'c', 20f, true, "foo")
//Wont't compile
//val set4: AnyVal = Set(1, 'c', 20f, true, "foo")
```

Часть 1. Типы.

Вывод типов

Скала имеет продвинутую систему вывода типов. Это значит, что если выражение строится на основе структур известных типов, то компилятор сможет определить тип возвращаемого результата. Для членов коллекций, арифметических и др. операций компилятор определит тип, как ближайший общий родитель (см. схему выше) операндов, участвующих в выражении.

Вывод типов осуществляется на уровне выражений. Это значит, что если для члена выражения необходимо вывести тип, он будет выведен на основе членов типов этого же выражения. То, как используется этот член, далее в других выражениях, учитываться при выводе не будет.

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] = new List[Float](1,2,0f)
val shortNotion = List(1, 1l, 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. Т.е. вот так, например,

```
// Нельзя, тип X неопределен ( хотя есть языки в которых это сработает)
{ x => x }
// а так можно. Так мы определили функцию, identity для Int
{ x:Int => x}
```

- 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. 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. Определения

Переменные

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

Константы

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

Ленивая инициализация

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

Часть 1. Функции

Функции и методы

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

val funcName2 = (inputPrm: String, otherPrm: String) => {
 inputPrm
}

// 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
```

Значением функции, является значение последнего в ней выражения

Часть 1. Функции

Разница между def и val

Функцией принято называть выражение, определенное с помощью ключевого слова **val**. Такое значение имеет тип **FunctionN** (где 0<=N <= 22 - количество входных параметров). Функция имеет все методы, содержащиеся в типе **FunctionN**, может быть передана как значение, в другие функции. Функции можно композировать.

Методом, называется выражение определенное с помощью ключевого слова **def**, как член объекта или класса . Метод не имеет типа, но может быть генерализован с помощью Type Parameters. Для того, чтобы передать метод как значение, он должен быть преобразован в **val** типа **FunctionN**. Чаще всего, это происходит неявно, но требует дополнительных расходов и может привести к деградации приложения.

Часть 1. Функции

Процедуры

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

Переменная длина 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

Функции с несколькими наборами параметров. Каррирование.

Каррирование - это способ представить функцию от нескольких переменных, как функцию высшего порядка от одной переменной.
Для функций от 2-х и более переменных каррирование доступно через метод **curried**. Для методов оно реализуется через сигнатуры с несколькими наборами параметров.

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

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

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

val partiallyApplied = curriedFilter("data1") _

val fullyAppliedAgain = partiallyApplied("data2")

val f = (data1: String, data2: String) => data1 == data2
//cF: String => (String => Boolean)
val cF = f.curried
```

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 имею 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-я 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 принимет 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 multiplyWithPrinter(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
// }
multiplyWithPrinter(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)

 val a: ServiceA = new ServiceA(c)
 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. Круглые и фигурные скобки

Scala имеет несколько правил относительно круглых () и фигурных {} скобок:

1. Для параметров-функций допускается опускать скобки:
```
list.map( _ * 2 )
list.map({ _ * 2 })
```
2. Допускается опускать скобки при вызове функции, если в списке аргументов есть только один параметр:
```
list.foldLeft(0) ( _ + _ )
list.foldLeft{0} { _ + _ }
list.foldLeft(0)({ _ + _ })
```
3. case превращает метод в PartialFunction и опускание скобок из пункта 1 не работает:
```
list.map(case x => x * 2)     // Won't compile
list.map { case x => x * 2 } // OK
```
4. Остальные случаи использования скобок фиксированы и являются синтаксисом соответствующих конструкций (def, if, while etc.)

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, 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

Потренируйтесь в написании циклов и условных операторов
lectures.operators.Competition

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

Введение в тестирование

План:

- для чего вообще нужно тестирование (кратко)
- TDD (кратко)
- Основные способы:
  - assert
  - ScalaTest: введение, Single Test, Suite
  - ScalaTest FunSuite (assert, assertResult, assertThrows)
  - ScalaTest BDD

Pattern matching

Сопоставление с образцом (pattern matching) - удобный способ ветвления логики приложения.

```
val x:Int  = 10

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

Ключевое слово **match,** после переменной, указывает на начало операции сопоставления, а **case** определяет образцы, с которыми производиться сопоставление

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

Pattern matching

- Сопоставление идет до **первого** подошедшего **case**, а не до наиболее подходящего.
- Pattern matching исчерпывающий (exhaustive), это значит, что подходящая ветка обязательно должна быть определена, иначе будет выброшено исключение **scala.MatchError**.
- Можно указать case по умолчанию с помощью конструкции **case _ =>** или **case someName =>**

```
val x:Int  = 10

// Будет выбрано “Something” несмотря на то, что ‘10’ более подходящий вариант
x match {
 case _ => "Something"
 case 10 => "ten"
}

// Исключение scala.MatchError
val stringValue = x match {
 case 1 => "one"
 case 11 => "eleven"
}
```

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 экстракторов (unapply)

```
val c: Any = "string"

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

Pattern matching

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

Pattern matching для кейс классов

```
case class Address(country: String, city: 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", _, _, _) => println(inRussian.city)
 // тоже что и первый case
 case Address(country, city, _, _) if country == "Russia" => println(city)
 case inUSA@Address("USA", _, _, _) => println(inUSA.city)
 case _ => println("Terra incognita!")
}
```

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)
}

printList(list)
```

Pattern matching

Pattern matching для регулярных выражений

```
// matching без группировки
val rex1 = "[a-z]*[0-9]{3}.*".r
"scala212 is good" match {
 case rex1() =>
   println("matched")
 case _ =>
}
// matching с группировкой
val rex = "([a-z]*)([0-9]{3})(.*)".r
"scala212 is good" match {
case rex(lang, ver) => // не подойдет, т.к. не все группы использованы
   println(s"$lang $ver")
case rex(lang, ver, rest) =>
   println(s"$lang $ver $rest")
 case _ => print("didn't match")
}
```

Pattern matching. Задания

Разберите вещи по коробкам, воспользовавшись pattern matching

lectures.matching.SortingStuff

Partial functions

Partial functions

Частичная функция (partial function) обозначает функцию, для которой область определения задается типами входных параметров и определенным правилом, заданным для них. В scala PF-это функция одного аргумента, имеющая тип **PartialFunction[-A, +B]**

Правило для вычисления области определения задается в виде метода
def isDefinedAt(prm: A): Boolean = ...

```
// from package scala
trait PartialFunction[-A, +B] extends scala.AnyRef with scala.Function1[A, B] ...

val pf = new PartialFunction[Int, String] {
 def apply(d: Int) = "" + 42 / d

 def isDefinedAt(d: Int) = d != 0
}
// despite the fact, that isDefinedAt == false
pf.isDefinedAt(0)
// we still can apply a function to an argument
pf(0)// the same as pf.apply(0)
```

Partial functions

Partial functions

В примере выше

- **def apply(d: Int) -** метод, который будет выполнен при вызове функции
- **def isDefinedAt(d: Int) -** метод, вычисляющий область определения функции

Для partial function есть сокращенная запись. При сокращенной записи тело PF представляет собой набор выражений **case.**

```
// It does the same but using pattern matching
val pf2: PartialFunction[Int, String] = {
 case d: Int if d != 0 => "" + 42 / d
}
pf2.isDefinedAt(0)

// Still error! But another one
pf2(0)
```

Не путайте сокращенную запись PF с pattern Matching. Тело PF не должно содержать исчерпывающие **case** выражения

Partial functions

Partial functions

Метод **lift** превращает **PartialFunction[-A, +B]** в **scala.Function1[A, scala.Option[B]]**

Это избавляет от необходимости проверять isDefined каждый раз, перед вызовом partial function.

```
val liftedPf =  pf2.lift
liftedPf(0)
liftedPf(15)
```

PartialFunction активно применяется в scala.collection. Например метод collect:

```
val list  = List(1, 2, 3, 5, 6, "4", "2", pf, pf2)

// List[Any] -> List[Int]
list.collect {
 case i: Int => i
}
```

Partial functions. Задания

Помогите реализовать авторизацию.

lectures.functions.Authentication

Коллекции

Обзор коллекций

- большинство коллекций 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

Иерархия коллекций в скале имеет более разветвленную структуру, чем в java, это связано с желанием создателей языка повысить переиспользуемость кода и лучше выделить семантические единицы реализации.

Коллекции

Трейты, составляющие основу коллекций в scala

- Traversable[+A]. Этот трейт принято считать корнем иерархии коллекций. Он отражает концепцию контейнера элементов, по которым можно итерироваться. Содержит абстрактный метод foreach. Реализации большинства методов трейта Traversable предоставлены трейтом TraversableLike.
- Iterable[+A]. Вводит в коллекции понятие итератора - специального объекта, имеющего методы next и hasNext, и предназначенного для определения способа итерирования по коллекции.
- *Like. По договоренности трейты, в названии которых присутствует Like, содержат имплементацию методов
- Gen*. Трейты, содержащие в своем названии Gen, по договоренности обозначают коллекции, чьи методы могут быть выполнены как последовательно, так и параллельно
- Seq, IndexedSeq, LinearSeq - трейты, обозначающие последовательность элементов. (Списки, потоки, вектора, очереди...)
- Set - определяет коллекции, не содержащие повторяющиеся элементы.
- Map - корневой трейт для ассоциативных массивов

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**

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 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
- **String -** это сиквенс символов

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)

// затем реализовать то же самое с помощью reduceLeft
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 NoSuchElementException
letterPosition("g")
letterPosition.get("g") // == None
```

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).map(_.toUpperCase)

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

Коллекции. Задания

Реализовать класс MyList

lectures.collections.MyListImpl

Избавиться от NPE

lectures.collections.OptionVsNPE

Написать сортировку слиянием

Постарайтесь не использовать мутабильные коллекции и **var**

Подробнее о сортировке можно подсмотреть здесь.

lectures.collections.MergeSortImpl

Коллекции

For comprehension (FC)

Это синтаксический сахар, предназначенный для повышения читаемости кода, в случаях, когда необходимо проитерироваться по одной или более коллекциям. FC, зависимости от ситуации, может заменить **foreach, map, flatMap, filter** или **withFilter.**

На самом деле, почти все циклы **for** в скале - это трансформированные функции. Если мы пишем цикл по одной или нескольким коллекциям без **yield,** этот цикл превратится в несколько методов **foreach.** Если в цикле присутствует **IF,** то вместо foreach будет использован **withFilter** или **filter,** если **withFilter** недоступен для данной коллекции.

Важно понимать различия между **withFilter** и **filter**. **withFilter** не применяет фильтр сразу, а создает инстанс **WithFilter[T]**, который применяет функции фильтрации по требованию. Это значит, что если в фильтре была использована переменная, которая поменялась в процессе обхода, то результат фильтрации, зависящий от нее, тоже поменяется. В случае метода **filter** это не так, т.к. он будет применен сразу и один раз**.**

Коллекции

For comprehension (FC)

```
val noun = List("филин", "препод")
val adjective = List("глупый", "старый", "глухой")
val verb = List("храпел", "нудел", "заболел")

for(n <- noun; a <- adjective; v <- verb) {
  println(s"$a $n $v")
}
// превратится в
noun.foreach {
 n => adjective.foreach {
   a => verb.foreach {
     v => println(s"$a $n $v") } } }
```

Коллекции

For comprehension (FC)

```
var noTeacher = ""
for(n <- noun if noTeacher != "филин";
   a <- adjective; v <- verb) {
 noTeacher = n
 println(s"$a $n $v")
}
noTeacher = ""
noun.withFilter(_ => noTeacher != "филин").foreach {
 n => adjective.foreach {
   a => verb.foreach {
     v => println(s"$a $n $v") } } }
noTeacher = ""
noun.filter(_ => noTeacher != "филин").foreach {
 n => adjective.foreach {
   a => verb.foreach {
     v => println(s"$a $n $v") } } }
```

Коллекции

For comprehension (FC)

Если цикл должен вернуть какое-либо значение, перед телом цикла ставят ключевое слово **yield**. В этом случае **foreach** нам уже не поможет, т.к. он возвращает тип **Unit.** На помощь приходят методы **map** и **flatMap**

```
val noun = List("филин", "препод")
val adjective = List("глупый", "старый", "глухой")
val verb = List("храпел", "нудел", "заболел")

for (n <- noun if n == "филин"; a <- adjective; v <- verb)
yield {
  s"$a $n $v"
}
// превратится в
noun.withFilter(_ == "филин").flatMap { n =>
 adjective.flatMap { a =>
   verb.map {
     v => s"$a $n $v"
   }}}
```

Коллекции. FC. Задания

For comprehension (FC)

Перепишите код в соответствии с условиями задачи.

lectures.collections.comprehension.Couriers

Коллекции

Tuples

Tuple, кортеж или record - это упорядоченный список элементов. Каждый член списка может иметь свой тип

В scala tuple - это кейс класс типа **Tuple1[T1] - Tuple22[T1,T2… T22]**.

Для создания tuple, начиная с Tuple2, достаточно заключить несколько элементов в круглые скобки, разделив их запятыми. Альтернативный способ создания tuple с 2-я элементами (Tuple2) - с помощью оператора '**->**'

```
val tpl2 = Tuple2("a", 1)
val tpl22 = ("a", 1)
val tpl23 = "a" -> 1
val tpl4 = ("a", 1, List(), () => {})
```

Для доступа к членам tuple автоматически генерируются методы-аксессоры **_n**, где n - это порядковый номер член tuple. Нумерация начинается с 1.

```
val tpl2 = ("a", 1)
println(tpl2._1) // would print 'a'
println(tpl2._2) // would print '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 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

Конструкции языка

Класс. Модификаторы доступа

Модификаторы доступа могут быть дополнительно специфицированы областью действия модификатора. Область действия задается в квадратных скобках после модификатора

- **private[somePackage]** (**protected[this]**) член класса, останется публичным внутри пакета somePackage, для остальных членов приложения он станет приватным
- **private[this]** . Такой скоуп называется object-private. Члены класса, помеченные таким образом, доступны исключительно членам того же инстанса**.**

Конструкции языка

Класс. Модификаторы доступа

```
object Hobbit{
 def  destroyStuff(hobbit:Hobbit) = hobbit.otherStuff
 def  destroyTheRing(hobbit:Hobbit) = hobbit.precious
}
class Hobbit {
 private val otherStuff: String = ""
 private[this] val precious: String = "the Ring"

 private def showSomeStuff() = otherStuff

 private[this] def lookAtPrecious() = {
 }

 def visit(bilbo: Hobbit) = {
   bilbo.showSomeStuff()
   bilbo.lookAtPrecious()
 }
}
```

Конструкции языка

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

 def publicMethod() {
   print("public method")
 }
 // This constructor is 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()
```

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 **private**, **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

Конструкции языка

Конструктор

```
// This constructor is inaccessible from outside
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)
 }

 def this(int: Int, str: String, inner: Long, member: Int) {
  this(int, str, 0)
  this.member = member
 }
}
```

Конструкции языка

VAR под капотом

Любой член класса, помеченный **var**, будет заменен компилятором приватным членом класса того же типа и 2-я методами, аксессором и мутатором

Допустим мы определили в классе **var x: Int = 0,** тогда после компиляции класс будет содержать

- private var x: Int = 0
- def x = x
- def x_=(prm: Int) = {x = prm}

Более того, определяя функции в соответствии с правилами именования, описанными выше, мы можем имитировать наличие нескольких переменных членов класса.

Конструкции языка

VAR под капотом

```
class Thermometer {

    var celsius: Float = _

    def fahrenheit = celsius * 9 / 5 + 32
     def fahrenheit_= (f: Float) {
       celsius = (f - 32) * 5 / 9
     }
     override def toString = fahrenheit +"F/"+ celsius +"C"
   }
val t = new Thermometer

t.celsius = 100
t.fahrenheit

t.fahrenheit = 100f
t.celsius
```

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 = ???
}
```

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
- к одному типу может быть подмешано более одного трейта

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

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, трейтов
- 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)
}
```

Конструкции языка

Чем полезны объекты-компаньоны

- в объекте-компаньоне удобно задавать статические данные, доступные всем инстансам этого типа
- метод apply используют, как фабрику объектов данного типа
- метод unapply используют для декомпозиции объектов в операторе присвоения и pattern matching-е
- имплиситы, определенные в объекте компаньоне, доступны внутри класса
- объекты компаньоны имеют доступ к приватным членам класса

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 - по умолчанию публичный **val**
- для кейс классов компилятор переопределяет метод **hashCode, 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

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)
 }
}
```

Конструкции языка

Подробнее об apply

С **apply** мы уже встречались в **lectures.functions.AuthenticationDomain.scala**. Например, для класса **CardCredentials** нам необходимо генерировать карты со случайными номерами. Вместо того, чтобы повторять этот код везде, где он нужен, мы переносим его в метод **apply.**

Если любой объект(не обязательно объект-компаньон) имеет метод **apply,** этот метод можно вызвать, указав после имени объекта круглые скобки.

```
trait Credentials
object CardCredentials {
 def apply(): CardCredentials =
  CardCredentials((Math.random() * 1000).toInt)
}
case class CardCredentials(cardNumber: Int) extends Credentials
// создаст инстанс CardCredentials со случайными реквизитами,
// это наш apply
CardCredentials()
// будет вызван apply, сгенерированный компилятором
CardCredentials(100)
```

Конструкции языка

Подробнее об apply

Для кейс классов объект компаньон и метод **apply** создаются автоматически. Количество входных параметров, их типы и порядок будут соответствовать членам класса.

```
// Т.е. Для кейс класса с сигнатурой
case class TestClass(t1:T1,  t2: T2)
// будет создан
object TestClass {
 //...
 def apply(xt1: T1, xt2: T2): TestClass = /* generated code */
 //...
}
```

Объект-компаньон можно написать вручную, при этом все методы, созданные автоматически, попадут в него. По этой причине для кейс классов нельзя переопределить метод **apply** с сигнатурой из примера выше.

Конструкции языка

Подробнее об apply

```
trait Credentials

object CardCredentials {
 def apply(): CardCredentials = CardCredentials((Math.random()* 1000).toInt )
}
case class CardCredentials(cardNumber: Int) extends Credentials
// создаст инстанс CardCredentials со случайными реквизитами,
// это наш apply
CardCredentials()
// будет вызван apply, сгенерированный компилятором
CardCredentials(100)
```

Конструкции языка

Подробнее об unapply

unapply обычно совершает действие, противоположное методу **apply**, а именно декомпозирует инстанс на составные части.

Сигнатура метода **unapply** выглядит следующим образом:

```
def unapply(parameter: T1): Option[T2] = ???
```

- **T1** - это тип элемента, разбираемого на части.
- **T2** - тип составной части. Если составных частей много, **T2** будет представлять собой **TupleN[N1, N2… N22]**, где N - количество составных элементов
- Метод **unapply** вернет
  - **Some[T2]**, если разобрать инстанс удалось
  - **None,** если разобрать не удалось

Конструкции языка

Unapply и кейс классы

Для метода **unapply**, созданного для кейс класса, действуют те же правила, что и для метода **apply.**

```
// Т.е. для кейс класса с сигнатурой
case class TestClass(t1:T1, t2: T2)
// будет создан
object TestClass {
 //...
 def unapply(puzzle: TestClass): Option[(T1, T2)] = /* generated code */
 //...
}
```

Конструкции языка

Unapply в операторе присваивания

Метод **unapply** удобно использовать, когда хочется разложить члены класса по переменным.

В примере ниже мы определим класс **ToyPuzzle** и **unapply** для него, возвращающий **Option[String, String, String].** Строки будут содержать значения цветов фигурок, из которых собран **ToyPuzzle.**

В случае, если **unapply** применяется в операторе присвоения и метод по какой-то причине вернул **None**, будет выброшен MatchError.

Конструкции языка

Unapply и pattern matching

Кейс классы и объекты, имеющие определенный метод **unapply**, можно использовать в case части pattern matching. Нужный case будет выбран тогда, когда соответствующий метод **unapply** вернет **Some.**

Пример: **lectures.features.Main**

Задания

Реализовать метод **add** простого бинарного дерева поиска.

Создать генератор дерева. **lectures.oop.BST**

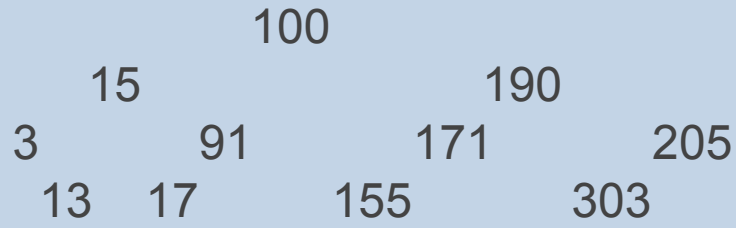
Задача 2. Доработать дерево. Поиск

Добавить в дерево метод find. Для поиска нужного значения метод должен использовать обход 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**.

Задания



Для наглядности можно заменить отсутствующих потомков значением ‘**-1**’

Задача 4. Метод fold для дерева

def fold(aggregator: Int)(f: (Int, Int) =>(Int)). Метод предназначен для агрегирования значений улов дерева. Например, с его помощью можно вычислить сумму значений всех узлов.

Нижнее подчеркивание в Scala

```scala
// Импортировать всё
import scala._

// Импортировать всё, кроме Predef
import scala.{ Predef => _, _ }

// Параметр типа высшего порядка
def f[M[_]] = ???

// Параметр анонимной функции
List(1).fold(0)(_ + _)

// Eta-расширение метода в значение
val m = (a: Int) => a.toString
m _

// Частично примененная функция
m(_)

// Отбрасывание параметра или переменной
List(1).map(_ => 5)
implicit val _ = 5
```

```scala
// Матчинг любых значений
val (a, _) = (1, 2)
List(1).collect { case _ => }
for (_ <- 1 to 10) { }

// Передача массива как списка параметров
def f(ys: Int*) = ys.sum
f(List(1): _*)

// Матчинг всех элементов списка
List(1) match { case List(xs @ _*) => xs.sum }

// Инициализация дефолтным значением
class A { var i: Int = _  }

// Разделитель символов в названиях методов
def abc_<>! = Unit
def foo_=(x: Int) = ???

// Доступ к элементам кортежа
val tuple = (1, 2)
tuple._2
```

Тестирование

Тесты - это приложения, которые проверяют приложения

Классификация тестирования:

- По уровням:
  - unit test - тест небольшой части приложения
  - integration test - тестирование нескольких компонентов
  - system test - тестирование всей системы “в сборе”
- По отношению к пользователям:
  - verification - проверка соответствию спецификации
  - validation - проверка соответствию пользовательским потребностям
- По типам:
  - smoke test - быстрая проверка работоспособности всего приложения
  - regression test - проверка работы основного (старого) функционала после внесения новых изменений
  - functional test - проверка на соответствие пользовательским требованиям
  - destructive testing - проверка реакции на исключительные ситуации
  - performance tests (stress, resilience, scalability) - категория тестов, направленная на проверку “спортивной формы” приложения.

Тестирование

Тесты - это приложения, которые проверяют приложения

Классификация тестирования:

- По наличию информации о приложении:
  - white box - с учетом знания реализации приложения (unit-тестирование).
  - black box - тестирования на основе требований. V&V и smoke
  - grey box - тесты, для которых важно учитывать и техническую информацию о приложении, и функциональные требования. Performance и smoke - чаще всего.

Тестирование

Как тестируем мы.

Перед тем, как попасть на бой, приложение должно пройти несколько ~~кругов ада,~~ этапов тестирования.

- code review - проводят все члены команды
- unit и functional тесты - запускаются при каждом пул реквесте в общую ветку. Наличие тестов - обязательное требование, для успешного прохождения CR.
- V & V на тестовой и закрытой боевой средах. Этим занимается отдел тестирования.
- smoke тесты и стресс тест. Selenium + Gatling
- smoke тест и V & V после релиза

Тестирование

Часто употребляемые термины

- Test Driven Development (TDD) - методология разработки, в которой написание тестов происходит раньше написания основного кода приложения. Wiki
- Behaviour Driven Development (BDD) - это подход при котором тесты представляют собой исполняемую спецификацию приложения. Scala test BDD
- mock, stub, dummy - это модули частично или полностью, подменяющие собой соответствующие модули тестируемого приложения. Интересная статья Мартина Фаулера на тему моков, стабов и подхода к Unit тестированию
- spy - частично примененный mock

Тестирование

ScalaTest

Самый популярный фреймворк для unit и functional тестирования на скале. Домашняя страница - http://www.scalatest.org/

ScalaTest предоставляет программисту на выбор несколько стилей написания тестов. Чтобы было понятнее, сразу перейдем к примерам:

lectures.collections.MergeSortImpFunSuiteTest
lectures.collections.MergeSortImplFlatSpecTest
lectures.collections.MergeSortImplWordSpecTest
lectures.oop.BSTTestWithMocks

Тестирование

ScalaCheck

Это фреймворк, предназначенный для тестирования по свойствам (property testing) Его можно использовать как отдельно, так и в составе ScalaTest.

Property testing - это разновидность автоматизированного grey box тестирования. На вход system under test (SUT) передаются наборы параметров. После обработки параметров в SUT, выходные данные проверяются в соответствии с логикой его работы.

Входные данные для теста могут быть заранее подготовлены разработчиком, в этом случае это тестирование на основе таблиц (table driven). Или данные могут быть сгенерированы автоматически. Последний вид тестирования часто называют generator driven. Подробнее о property testing можно прочитать на соответствующей странице на сайте ScalaTest

Примеры

- table driven: **lectures.check.TableStyleScalaCheckTest**
- generator driven: **lectures.matching.SortingStuffGeneratorBasedTest**

Тестирование. Задания

Завершите реализацию теста для SortingStuff
lectures.matching.SortingStuffGeneratorBasedTest

Напишите тест для Authentication
lectures.functions.AuthenticationTest

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, чтобы вызвать исключительную ситуацию, нужно использовать оператор **throw**

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()
```

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

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()
```

Исключительные ситуации

Обработка исключений

Функциональный подход может быть реализован несколькими способами. Наиболее популярный - с использованием **Try[T]**. В отличие 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 двух наследников
  - **Success[T]**. Объект этого типа будет создан, если код завершился без ошибок
  - **Failure[Throwable]**. Объект этого типа будет создан, если было выброшено исключение

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

Одним из минусов **Try[T]**, является отсутствие среди методов аналога **finally**

В **Try[T]** невозможно перехватить фатальные ошибки, такие как OutOfMemoryException

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")
   }
}
```

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()
```

ООП

Наследование - механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса или интерфейса

Абстракция - механизм языка, позволяющий выделять концептуальные особенности объекта или класса. Обычно абстракция реализуется через интерфейсы и трейты

Инкапсуляция - разграничение доступа членов классов к членам друг друга

Полиморфизм - в общем смысле, это способность функций менять свое поведение. Функции могут менять способ обработки одних и тех же параметров (subtype polymorphism) или менять набор и типы обрабатываемых параметров (ad-hoc, parametric polymorphism)

ООП

SOLID (wiki):

- Single responsibility - у класса или функции должна быть четкая сфера ответственности
- Open/closed principle - элементы приложения должны быть открыты для расширения, но закрыты для модификации
- Liskov substitution - везде, где используется супер класс, может быть использован его класс-наследник
- Interface segregation - много маленьких специфичных интерфейсов лучше чем один большой и “универсальный”
- Dependency inversion - любая реализация должна зависеть от абстракции. Это касается не только отдельных частей приложения, но и всего приложения в целом.

ООП

Наследование в скала

Для того, чтобы сделать класс (объект или трейт) наследником другого класса, нужно использовать ключевое слово **extends.**

Можно наследоваться от:

- трейтов
- классов
- абстрактных классов
- кейс классов.

Нельзя:

- от объектов
- наследовать кейс класс от кейс класса

Ключевое слово **override** говорит о том, что данный член класса переопределяет соответствующий член супер класса.

При переопределении абстрактных членов, **override** указывать не надо.

ООП

Наследование в скала

```
object SuperObject {}
trait SuperTrait {}
class SuperClass {
 val name = "SuperClass"
 protected val secretName = "secret"
}
class SubClass extends SuperClass {
 def printMySecretName = secretName
}
class SubClassWithTrait extends SuperTrait {}
//you can't extend object
class SubClassByObject extends SuperObject{}

class TestApp extends App {
 val sc = new SubClass()
 sc.name
 sc.secretName
 sc.printMySecretName
}
```

ООП

Наследование в скала

Если наследоваться от класса, у которого есть конструктор, все параметры основного конструктора, не имеющие значений по умолчанию, должны быть указаны в скобках после имени класса в выражении extends.

Ключевые слова:

- **super** можно использовать для доступа к членам супер класса, которые не объявлены приватными
- **final** перед определением компонента обозначает, что от этого члена приложения нельзя создать наследника. Абстрактный класс может быть final, но для обычных разработчиков смысла так делать нет, т.к. ни наследника, ни объект такого класса создать нельзя
- **sealed** перед определением компонента обозначает, что наследники этого компонента должны быть определены только в этом классе.

ООП

Наследование в скала

```
class SubClass extends SuperClass("blaa") {

  override val someInfo: String = ""

  def someSuperInfo = super.someInfo

  def printMySecretName = secretName
}

class SubClassWithTrait extends SuperTrait {}

val sc = new SubClass
sc.someInfo
sc.someSuperInfo
```

ООП

Множественное наследование

В скале разрешено множественное наследование. Оно решено в виде так называемого mixin наследования. Суть такого наследования в том, что к классу подмешиваются наборы абстрактных и (или) реальных членов, содержащихся в mixin сущностях (trait в scala). Трейты бывают очень удобны для создания переиспользуемых компонентов и для interface segregation (один из SOLID принципов). Характерным примером использования mixin можно рассматривать библиотеку коллекций в scala.

Трейты можно примешивать с помощью:

- **extends,** если это первый предок в цепочке наследования
- **with,** если это второй и следующие предки. В выражении после **with** могут присутствовать только трейты

ООП

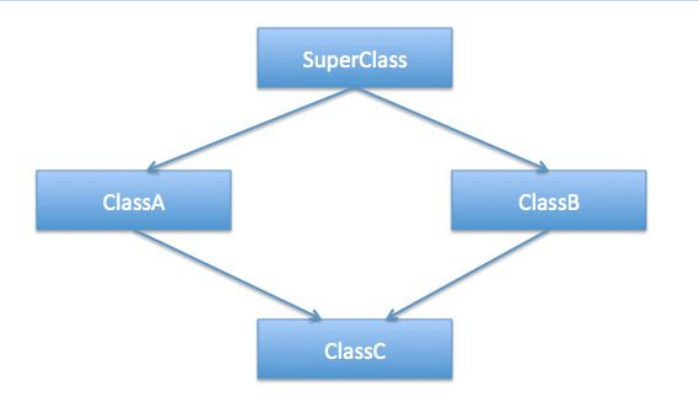
Наследование в скала

```
abstract class AbstractTest {
  val name: String
}

trait NameProvider {
  val name: String = "name provided by trait"
}

trait SomeMarker

class ConcreteClass extends AbstractTest with NameProvider withSomeMarker {
}

object InheritanceTest extends App {
  val k = new ConcreteClass
}
```

ООП

Множественное наследование, diamond problem

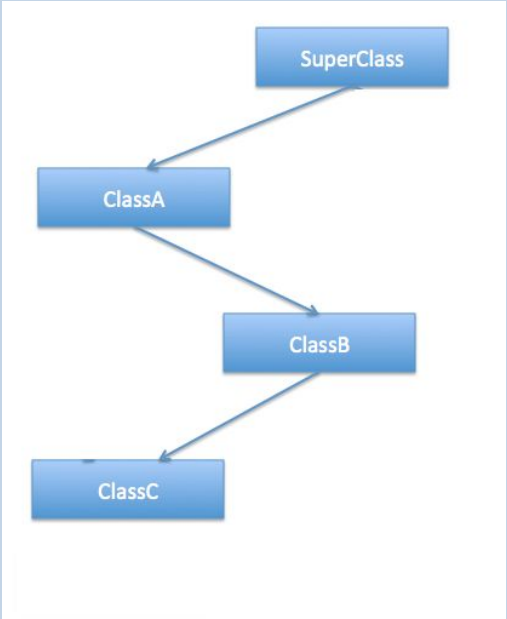
При множественном наследовании часто возникает вопрос: что делать, если несколько родителей предоставляют одинаковые члены класса. Из рисунка ниже понятно, откуда происходит название проблемы.



ООП

Множественное наследование, lineization

В скале применяется метод, называемый линеизацией. Суть в том, что все родители класса выстраиваются “в линию” в соответствии с определенным правилом.



ООП

Множественное наследование, линеизация

Для того, чтобы выстроить зависимости в линию, компилятор идет по всем предкам класса, объявленным после ключевого слова **extends,** и назначает текущий найденный класс или трейт суперклассом всех следующих членов списка предков. Если текущий найденный класс, в свою очередь, имеет предков, к ним так же применяются правила линеизации. Полученная цепочка зависимостей становится в списке, перед текущим найденным предком.

Следствия линеизации:

- Конструкторы классов выполняются в том порядке в котором были расставлены в процессе линеизации. Последним будет выполнен конструктор создаваемого класса
- Доступ к членам супер классов через ключевое слово **super** происходит в обратном порядке. Т.е. **super.memberName** обратится к **memberName** ближайшего суперкласса, полученного в процессе линеизации.

ООП

Множественное наследование, линеизация

Пример линеизации: **lectures.oop.lineization.scala**

ООП

Анонимные классы

Это сокращенная запись создания наследников от практически любой структуры, в том числе от трейтов, абстрактных классов. Эта запись часто применяется, когда нужен синглтон какого-то типа, но сам тип этого синглтона никогда не потребуется. При создании анонимного класса необходимо доопределить все абстрактные члены всех классов и трейтов, которые входят в новый класс.

Анонимные классы могут быть созданы 2-я разными путями

- pre-initialized fields - тело анонимного класса идет перед наследуемыми типами. В этом случае членами тела анонимного класса могут быть только var и val
- post initialized - более привычный способ определения, когда тело класса идет за выражением extends и with

ООП

Анонимные классы

```
trait TestTrait {
 def str: String
 def otherStr  = str
}

abstract class SuperClass(j: Int) {
 val i: Int = 0
}
val postInit = new SuperClass(10) with TestTrait {
 override def str: String = ???
}

val preInit = new {
 val str = "string"
} with TestTrait
```

ООП

Self type annotation

Это механизм дополнительной спецификации типа трейта или класса. Аннотация говорит о том, что все инстансы, в которые входит данный трейт (или класс), так же должны быть наследниками всех типов, перечисленных в аннотации. Благодаря аннотации, внутри аннотированного трейта (или класса) становятся доступны все публичные и protected члены тех типов, которые входят в аннотацию.

Что бы проаннотировать, например, класс, внутри тела класса первым выражением должно стоять выражения вида

self : TypeOne [with Type2 …] => , где

- **self** идентификатор, обозначающий текущий класс
- **TypeOne** - тип, которому должны соответствовать инстансы текущего класса
- **with Type2 -** опциональные, дополнительные типы

ООП

Self type annotation

```
trait RealService {
 protected def doSomething = "done"
}

trait Service {
 self: RealService  =>

 def service() = doSomething
}

class InjectedService extends Service with RealService

val serviceImpl  = new RealService with Service
serviceImpl.service()
```

ООП

Задания.

Помогите рыбаку

lectures.oop.Fisherman.scala

Пример простого DI в скала. Решите задачу и допишите тесты

lectures.oop.Application.scala

lectures.oop.ApplicationTest.scala

Parametric polymorphism

Type parameters (AKA generics)

Type parameters (TP) - это механизм, так же известный как параметрический полиморфизм, где параметром является тип или какое-либо выражение над типом или несколькими типами. Благодаря TP можно:

- создать более строго типизированные приложения
- сконструировать полиморфные типы, чье поведение варьируется в зависимости от TP

В скале для того, чтобы показать, что тот или иной тип принимает TP, после имени типа в квадратных скобках указывают список параметров и (или) выражения над ними.

Полиморфными могут быть не только типы, но также методы и даже переменные и константы.

Передать TP в тип можно несколькими способами:

- на этапе создания наследника типа
- на этапе создания инстанса типа
- передав параметр определенного типа в метод, если TP определен на уровне метода

Т.к. scala имеет полиморфизм 1 ранга, на момент создания инстанса типа все TP должны иметь значения, переданные тем или иным способом

Parametric polymorphism

Type parameters (AKA generics)

```
/**
 * Binder 1 - создает списки того типа,
 * который мы передали во время создания инстанса
 * @tparam T
 */
class Binder[T] {
  def bind(item: T): List[T] = List(item)
}

class StringBinder extends Binder[String]

val staticStringBinder =  new StringBinder()
val instanceStringBinder = new Binder[String]()

staticStringBinder.bind("blaa") == instanceStringBinder.bind("blaa")
```

Parametric polymorphism

Type parameters (AKA generics)

```
/**
 * Развитие событий, мы хотим, чтобы тип создаваемого
*  листа определялся типом переданного параметра
 *
 */
class Binder2 {
  def bind[T](item: T): List[T] = List(item)
}

//class StringBinder extends Binder2[String]
//val instanceStringBinder = new Binder2[String]()
val binder = new Binder2()

binder.bind("blaa")
binder.bind(1)
binder.bind(new Binder2)
```

Parametric polymorphism

Type parameters (AKA generics)

Продолжим развивать наш **Binder**. Теперь мы хотим, чтобы была возможность создавать не только списки, но и вообще любой контейнер элементов.

На выручку нам приходят Existential Types Parameters (ETP)**.** Обозначаются они как нижнее подчеркивание, и могут быть указаны везде, где могут появиться обычные TP (которые иногда называют Universal type parameters).

Для ETP подчеркивание - это сокращенная запись более многословного определения, которое выглядит следующим образом: **(T) forSome { type T }.** Ее можно использовать в тех местах, где компилятор запрещает использовать подчеркивание, например в параметрах методов.

ETP можно воспринимать как placeholder для TP. Он означает, что существует такой тип (или типы), который будет подставлен на место этого плейсхолдера. При этом в текущем определении (метода, класса, трейта и т.д.) значение TP несущественно. Подставить TP на место ETP можно в процессе создания наследника в момент создания инстанса или вызова метода с ETP.

Parametric polymorphism

Type parameters (AKA generics)

Определим трейт, который будет обозначать группу методов для создания инстансов коллекций, при этом нам пока не важно, каких именно. Добавим еще один метод, **fill[T](count: Int, item: T): M[T],** с помощью которого можно будет создавать коллекции нужного размера, заполненные значениями по умолчанию.

Parametric polymorphism

Type parameters (AKA generics)

```
trait Binder3[M[_]] {
  def bind[T](item: T): M[T]
  def fill[T](count: Int, item: T): M[T]
}
class SeqBinder extends Binder3[Seq] {
  override def bind[T](item: T): Seq[T] = Seq(item)
  override def fill[T](count: Int, item: T): Seq[T] = Seq.fill(count)(item)
  def badFill(count: Int, item: (T) forSome {type T}): Seq[_] =
Seq.fill(count)(item)
}
class SetBinder extends Binder3[Set] {
  override def bind[T](item: T): Set[T] = Set(item)
  override def fill[T](count: Int, item: T): Set[T] =
Set(Seq.fill(count)(item): _*)
}
(new SeqBinder).bind(100)
(new SetBinder).bind(100)
(new SeqBinder).fill(10, 100)
(new SeqBinder).badFill(10, 100)
(new SetBinder).fill(10, 100)
//val b = new Binder3[List]() //но вот так мы сделать не можем
```

Parametric polymorphism

Type parameters (AKA generics)

```
trait Binder3[M[_]] {
  def bind[T](item: T): M[T]
  def fill[T](count: Int, item: T): M[T]
}

class ConcreteSetBinder[C] extends Binder3[Set] {
  def bind[T](item: T): Set[T] = Set(item)
  def concreteBind(item: C): Set[C] = Set(item)
  override def fill[T](count: Int, item: T): Set[T] =
Set(Seq.fill(count)(item): _*)
}
val strBinder = new ConcreteSetBinder[String]()
strBinder
```

Parametric polymorphism

Type parameters (AKA generics)

Binder3 - это так называемый higher kinded type (HKT) .

Kind - тип, который порождает другой тип. Можно провести параллель между конструктором класса и HKT. Т.е. конструктор класса порождает конкретный объект, принимая другие объекты в качестве параметров. Kind, в свою очередь, порождает тип, принимая на вход TP.

В случае Binder3 - это kind, который в качестве входного TP ожидает M[_], который в свою очередь должен сам являться (HKT). Именно поэтому в выражении extend Binder3[Set], мы передаем именно Set, а не Set[A], т.к.последнее - это обозначение конкретного типа.

Подробнее о типах и видах - здесь blogs.atlassian.com и wiki

Если добавить еще немного магии имплиситов, мы сможем добиться вот такой записи:

```
import Binder4._
val set = bind[Int, Set](10)
val seq = bind[Int, Seq](10)
```

Разобраться с имплиситами нам еще предстоит, а сейчас разберемся с type bounds

ООП

Type parameters (AKA generics)

Ограничение TP (type parameter bound, TPB) - это способ передать дополнительную информацию о TP. TPB можно также воспринимать, как ограничение на тип, который мы можем передать в качестве TP

TPB бывают 2-х видов:

- upper bound, обозначается с помощью оператора <: например так: [B <: A]. Данное выражение говорит нам о том, что TP B может быть только A или любым наследником A На месте TP A может находится конкретное значение типа, например [B <: Long]
- lower bound, [B >: A] говорит нам о том, что тип B может быть A или любым из его предков. Пример применения lower bound будет дан после введения понятия вариативности

Parametric polymorphism

Type parameters (AKA generics)

```
trait Similar {
  def isSimilar(x: Any): Boolean
}

case class MyInt(x: Int) extends Similar {
  def isSimilar(m: Any): Boolean =
    m.isInstanceOf[MyInt] &&
      m.asInstanceOf[MyInt].x == x
}

object UpperBoundTest extends App {
  def findSimilar[T <: Similar](e: T, xs: List[T]): Boolean =
    if (xs.isEmpty) false
    else if (e.isSimilar(xs.head)) true
    else findSimilar[T](e, xs.tail)

  val list: List[MyInt] = List(MyInt(1), MyInt(2), MyInt(3))
  println(findSimilar[MyInt](MyInt(4), list))
  println(findSimilar[MyInt](MyInt(2), list))
}
```

Parametric polymorphism

Type parameters (AKA generics)

Вариативность (V For Variance). Для типов, принимающих TP, возникает вопрос, как TP влияют на отношение наследования т.е. Если List <: Seq и String <: AnyRef,то будет ли List[String] <: Seq[String] или еще интереснее List[AnyRef] <: Seq[String]. В java этот ответ однозначен и на оба вопроса он - нет. В scala есть механизм, позволяющий более гибко влиять на поведение TP

- covariance. Обозначается символом '**+**' перед TP. Если TP помечен как ковариантный, это значит, что если для типов выполняются условия B <: A и N <: T, то B[N] <: A[T]
- contravariance. Обозначается символом '-', перед TP. Для контравариантных типов выполняются условия, если B <: A и N >: T, B[N] <: A[T].
- invariance. Если перед TP не стоит ни каких символов, такой тип называют инвариантным. Т.е. вне зависимости от отношения наследования между типам, принимающими TP, типы, принявшие разные TP, не будут иметь никакого отношения наследования (по аналогии с Java)

Простое мнемоническое правило для понимания вариативности: обратите внимание на стрелки:

- если они направлены в одну сторону - это ковариантность
- если они направлены в разные стороны - это контравариантность

Parametric polymorphism

Инвариантность

В примере ниже мы не сможем сложить 2 массива, т.к. TP нашего листа инвариантен. Мы не сможем этого сделать, даже если сделаем параметр ковариантным, но уже совершенно по другой причине.

```
case class ListNode[T](h: T, t: ListNode[T]) {
  def head: T = h
  def tail: ListNode[T] = t
  def prepend(elem: T): ListNode[T] =
    ListNode(elem, this)
  def concat(other: ListNode[T]): ListNode[T] = ???
}

val node1: ListNode[AnyRef] = ListNode("blaa", null)
val node2 = ListNode[AnyRef] ("blaa", null)

// NOP ((
node1.concat(node2)
```

Parametric polymorphism

Ковариантность ⇉

Seq[+A] - ковариантен по параметру A. Ответим на вопрос, является ли **List[String]** наследником **Seq[AnyRef]**. Т.к.

- **Seq → List**
- и **AnyRef → String**
- то выполняется **Seq[AnyRef] → List[String]**

Если Seq не был бы ковариантным, мы не смогли бы передать список строк в метода, который принимает AnyRef.

```
def printSeq(someSeq: Seq[AnyRef]): Unit =
 someSeq.foreach(print)

val toPrint = List("a","b","c","d")
printSeq(toPrint)

val wontPrint = List(1,2,3,4,5)
printSeq(wontPrint)
```

ООП

Контравариантность ⇄

Пусть есть классы

- **TheContravariant[-T]**
- **class OtherContravariant[-T] extends TheContravariant[T]**.

Ответим на вопрос, будет ли **OtherContravariant[Any]** наследником **TheContravariant[String]**

Т.к.

- **TheContravariant → OtherContravariant**
- и **String ← Any**
- то выполняется **TheContravariant[String] → OtherContravariant[Any]**

ООП

Контравариантность ⇄

Обещанный пример с lower bound. Дело в том, что все функции в scala имеют контравариантные параметры, а +T ковариантен и не может быть использован в качестве параметра метода.
Довольно доходчивое объяснение, почему параметры функций контравариантны можно найти на [stack overflow]

```
case class ListNode[+T](h: T, t: ListNode[T]) {
 def head: T = h
 def tail: ListNode[T] = t
 def prepend[U >: T](elem: U): ListNode[U] =
   ListNode(elem, this)
 def concat[U >: T](other: ListNode[U]): ListNode[T] = ???
}

val node1: ListNode[AnyRef] = ListNode("blaa", null)
val node2 = ListNode("blaa", null)

// YES
node1.concat(node2)
```

Parametric polymorphism

Абстрактные типы (abstract type, AT)

Абстрактные типы - альтернативный способ создать полиморфные типы. AT определяются в ключевым словом **type** в теле класса,объекта или трейта. К AT применимы все те же правила, что и к TP. AT относятся так к TP, как параметры переданные через конструктор, относятся к унаследованным членам класса.

AT применяются в следующих случаях:

- когда принципиально важно убрать TP из сигнатуры методов или типов.
- если TP имеют крайне сложный вид или их стало очень много и они делают код сложным для восприятия
- что бы подчеркнуть, что AT по смыслу является частью типа в котором описан (отношение is-a).

В остальных случаях предпочтительно использовать TP. Так же как полиморфизм через композицию часто более предпочтителен чем полиморфизм через наследование.

Интервью Одерского на тему AT vs TP на artima

Parametric polymorphism

Абстрактные типы (abstract type, AT)

```
abstract class ListNode {
 type ITEM
 type R <: ListNode
 type U >: ITEM
 val h: ITEM
 def head: ITEM = h
 val t: R
 def tail: R = t
 def prepend(elem: U):  R
 def concat(other: ListNode): ListNode
}

class StringNode(val h: String, val t: StringNode) extends ListNode {
 type R = StringNode
 type ITEM = String
 def prepend(elem: U): StringNode = ???
 def concat(other: ListNode): ListNode = ???
}
val node1: ListNode = new StringNode("blaa", null)
val node2 = new StringNode("blaa", null)
// YES
node1.concat(node2)
```

Parametric polymorphism

Type erasure (TE)

TE - это процедура удаления информации о типах в runtime. Это значит, что значение TP и AT будет заменено на нижнюю границу, определенную для этого TP. Т.е., если мы определили TP таким образом [T], он будет заменен на при компиляции на Object. Если TP имел нижнюю границу, то он буде заменен на это ограничивающее значение. Т.е, например, [T <: List[String]] станет List[Any], а [T <: List[_]] станет тоже List[Any]

Есть способы сохранить информацию о типах используя scala reflection. Его исследования не входит в текущую версию курса и пока остается для самостоятельного изучения.

Parametric polymorphism

Type erasure (TE)

```
def maybePrepend(elem: Any): Unit = elem match {
 case e: T => print("suitable for prepend")
 case notE => print("not suitable")}

def stringListPrepend[TT <: List[String]](other: Any): Unit = other match {
 case e: TT => println("List[Any]")
 case notE => println("else")}

def generalListPrepend[TT <: List[_]](other: Any): Unit = other match {
 case e: TT => println("List[Any]")
 case notE => println("else")
}
val strList = List("")
val valList = List(1,2,3,4)
val str = "bad prm"
maybePrepend(str)
stringListPrepend(strList)
stringListPrepend(valList)
stringListPrepend(str)
generalListPrepend(strList)
generalListPrepend(valList)
generalListPrepend(str)
```

Parametric polymorphism

Задания

Измените lectures.collections.MyList, применив TP. Создайте тест в который поместите выражения, перечисленные ниже. Все выражения должны выполняться без ошибок

```
require(MyList[Int, List[Int]](List(1, 2, 3, 4, 5, 6)).map(p => p * 2).data == List(2, 4, 6, 8, 10, 12))
require(MyList[Long, ListBuffer[Long]](ListBuffer(1, 2, 3, 4, 5, 6)).filter(_ % 2 == 0).data == List(2, 4, 6))
require(MyList[Int, List[Int]](List(1, 2, 3, 4, 5, 6)).foldLeft(0)((tpl) => tpl._1 + tpl._2) == 21)
require(MyList[Float, IndexedSeq[Float]](ArrayBuffer.empty[Float]).foldLeft(0)((tpl) => tpl._1 + tpl._2) == 0)
```

Затем на основе первоначального MyList создайте 2 наследника MyListBuffer и MyIndexedList так, чтобы выполнялись

```
require(MyListBuffer[Long](ListBuffer(1, 2, 3, 4, 5, 6)).filter(_ % 2 == 0).data == List(2, 4, 6))
require(MyIndexedList[Float](ArrayBuffer.empty[Float]).foldLeft(0)((tpl) => tpl._1 + tpl._2) == 0)
```

Implicits

Implicits - это механизм позволяющий внести изменение в работу приложения не делая явных правок в коде, подвергаемом изменению. Этот механизм применим и в compile time и в runtime
Имплиситы имеют 3 основные сферы применения

- неявные параметры (implicit parameters, ImP).
- неявная конвертация (implicit convertions, ImC)
- неявный контекст (implicit context, IC)

В определенном смысле имплиситы можно воспринимать как еще один способ создания полиморфных типов.
Имлиситными могут быть

- переменные и константы. В пример ниже определена имплиситная константа

```
implicit val seqBuilder = new Builder[Seq]{
 override def build[T](item: T): Seq[T] = Seq(item)}
```

- методы

```
implicit def seqBuilder() = new Builder[Seq]{ override def build[T](item: T): Seq[T] = Seq(item) }
```

- классы (начиная с scala 2.10)

```
implicit class IntWithTimes(x: Int) { … }
```

Implicits

Свойства имплиситов

- именование - имплиситы могут иметь любые имена, но для того, чтобы стать имплиситами их определение должно начинаться с ключевого слова **implicit**
- доступность - для того, чтобы имплисит был применен, он должен находиться в скоупе. Поместить имплисит в скоуп можно:
  - определив его внутри класса, в котором он будет использован
  - импортировав с помощью ключевого слова import
  - определив в объекте-компаньоне
- однозначность - в скоупе не должно быть нескольких имплиситов с одной сигнатурой
- сначала явные - если вывод типов не выявил необходимости в применении имплиситов, они применены не будут, даже если доступны в скоупе.
- одноуровневость - компилятор не будет предпринимать попытки применить имплиситы несколько подряд, что бы добиться совпадения типа. Т.е. если в скоупе нет имплисита подходящего типа, имплиситы применяться не будут.

Implicits

Implicit convertions (ImC)

ImC - это способ превратить объект одного типа в объект другого типа, без явной конвертации. Для того, чтобы ImC имело место необходимо 2 условия

- тип объекта, который подлежит конвертации не соответствует ожидаемому, в данном месте приложения. Т.е. объект передают или возвращают из функции, чья сигнатура подразумевает другой тип. Или у объекта вызывают член, которого нет в данном типе
- В текущем скоупе есть доступный способ превратить объект в новый объект подходящего типа

ImC обычно реализуют через implicit функции или implicit классы

Pimp My Library - это шаблон программирования применяемый в основном для расширения сторонних библиотек. Суть его заключается в том, что объекты библиотечных типов неявно конвертируются в объекты расширенных типов. Таким образом внешне код выглядит так, как будто у библиотечных объектов появились новые методы и свойства.

Implicits

Implicit convertions (ImC)

Характерным примером применения ImC является расширение строки в Scala. Благодаря Predef.scala, содержимое доступно без импорта во всех scala файлах.

```
// This is possible due to implicit convertion
//thanks to this line @inline implicit def augmentString(x: String): StringOps = new StringOps(x)
// and Predef.scala in general
val regex = "sdfvsdf".r

//@inline implicit def longWrapper(x: Long)       = new runtime.RichLong(x)
//@inline implicit def floatWrapper(x: Float)     = new runtime.RichFloat(x)
def  strangeSum(arg1: Long, arg2: Float): Double  = arg1 + arg2
```

В примере выше строка неявно конвертирована в тип StringOps, который имеет метод r(), превращающий строку в Regexp

Implicits

Implicit parameters (ImP)

Методы (но не функции) могут иметь один или более параметров, подставляемых неявно. Для этого

- в функции с одним набором параметров, весь набор должен быть помечен implicit

```
class MyImplicit {
 def compare(that: Int): Int = 1
}
implicit val io = new MyImplicit
def doImplicitly()( implicit prm: MyImplicit) ={
 print(prm)
}
doImplicitly
```

- в функциях с несколькими наборами параметров, последний набор должен быть помечен implicit.
- в скоупе в месте вызова такой функции должны быть определены все требуемые имплиситы. В данном случае это могут быть **implicit val** или **implicit object**

Имплиситные параметры часто применяются там, где в при стандартном ООП подходе применяется паттерн стратегия. При этом, полиморфизм из наследственного превращается в параметрический, что дает большую гибкость. Пример : **scala.collection.SeqLike**, метод sorted

Implicits

Implicit context bounds (IC) - это сравнительно новый механизм, пришедший в скалу с версии 2.8. Он может быть применен только в методам или классам. Предназначен он для того, чтобы с помощью TP описать контекст в котором может выполняться данный или создан инстанс данного класса. Под контекстом понимается набор имплиситов доступных в данном скоупе.
Для того, чтобы определить IC после TP, через двоеточие нужно передать тип требуемого имплисита, например так: **[T : Numeric]**. Тип имплисита должен в свою очередь принимать TP

Перед применением IC скала компилятор конвертирует его в имплисит параметр. Например следующие 2 функции практически идентичны

def add[T : Numeric](one: T, other: T) = ???
def add[T](one: T, other: T)(implicit evidence: Numeric[T]) = ???

Разница заключается в том, что в первом случае имплисит параметр недоступен явно в теле функции. Для того, чтобы получить к нему доступ, используется метод **implicitly[T](implicit e: T)**

Implicits

Implicit context bounds (IC)

```
trait Numeric[T] {
 def add(x: T, y: T): T
}

object NumericExperiment extends App{
 implicit val int2Num: Numeric[Int] = new Numeric[Int]() {
   override def add(x: Int, y: Int): Int = x + y
 }
 def doAdd[T: Numeric](one: T, other: T) = {
   val r = implicitly[Numeric[T]]
   println("Added " + r.add(one, other))
 }
 def doAddWithExplicitPrm[T](one: T, other: T)(implicit evidence: Numeric[T]) = {
   println("Added " + evidence.add(one, other))
 }
doAdd(1, 2)
doAddWithExplicitPrm(1, 2)

}
```

Implicits

Type classes (TC) - это паттерн, позволяющий приводить инстансы разных типов к общему классу типов (type class). Реализуется этот паттерн с помощью механизма имплиситов.
Трейт **Numeric[T]** из примера выше представляет собой класс типов, для которых может быть определена операция сложения (add). В примере к этому типу классов приводится тип Int. Таким же образом можно поступить с любым типом для которого может быть определена операция сложения.
Благодаря TC мы можем создавать код, применимый к типам, не имеющим между собой отношения наследования, но которые могут быть сведены к одному классу типов
Хороший пример - тип **scala.math.Numeric[T]**

Implicits

Type classes

```
trait Numeric[T] {
 def add(x: T, y: T): T
}

case class MoneyAmount(money: Double)

implicit val money2Num: Numeric[MoneyAmount] = new Numeric[MoneyAmount]() {
 override def add(x: MoneyAmount, y: MoneyAmount): MoneyAmount = MoneyAmount(x.money +
y.money)
}

def doAdd[T: Numeric](one: T, other: T) = {
 val r = implicitly[Numeric[T]]
 println("Added " + r.add(one, other))
}

doAdd(MoneyAmount(100), MoneyAmount(300))
```

Implicits

Цепочки имплиситов

Как уже упоминалось ранее, компилятор не будет предпринимать попытку произвести несколько преобразований подряд, чтобы добиться совпадения типов. Однако, в случае необходимости, можно добиться поведения практически идентичного цепочке имплиситных преобразований. Этого можно добиться, сделав так, чтобы имплиситное преобразование в свою очередь требовало, имплиситное преобразование. Таким образом, можно выстроить сколь угодно длинную цепочку преобразований.

Implicits

Цепочки имплиситов, пример из scala FAQ

```
class A(val n: Int)

class B(val m: Int, val n: Int)

class C(val m: Int, val n: Int, val o: Int) {
 def total = m + n + o
}

implicit def toA(n: Int): A = new A(n)
implicit def aToB[A1](a: A1)(implicit f: A1 => A): B = new B(a.n, a.n)
implicit def bToC[B1](b: B1)(implicit f: B1 => B): C = new C(b.m, b.n, b.m + b.n)

// works
println(5.total)
println(new A(5).total)
println(new B(5, 5).total)
println(new C(5, 5, 10).total)
```

ООП

Задания
lectures.oop.types.GeneralBST.scala

Нестрогие вычисления

Для начала дадим определение, что такое строгое вычисление (strict evaluation, eager evaluation)

Строгое вычисление - это вычисление, которое происходит в момент, когда оно связывается с переменной.

Нестрогое вычисление - это, соответственно, любое вычисление, которое так или иначе отложено по отношению к связи с переменной.

Нестрогость в скале:

- базовые механизмы:
  - логические операторы **&&**, ||
  - **lazy val -** вычисление по необходимости
  - **prm: => {...} -** передача по имени
- производные механизмы:
  - **Stream** - потенциально бесконечные последовательности значений
  - **View** - коллекция, функции которой вместо строго вычисления значений для элементов коллекции возвращают новый view. Таким образом, эффекты, которые должны быть применены к элементам коллекции, композируются. Они будут применены только в момент получения конкретного значения.

Нестрогие вычисления

View

Идея view состоит в том, чтобы отложить все модификации элементов до момента получения этих элементов из коллекции. Для этого вместо применения какой-либо функции возвращается новый объект view, который содержит в себе информацию о функции, которую нужно вызвать.

View существуют для всех Traversable коллекций.

Для того, чтобы получить view, у объекта достаточно вызвать метод view:

```
// У Traversable view есть
val a = Array("")
val arrView = a.view

val o = Option("")
val optView = o.view

val m = Map()
val v = m.view

// У не Traversable контейнеров view может не быть
val f =  Future{()}
f.view

val t = Try{""}
t.view
```

Нестрогие вычисления

View

Для того, чтобы вычислить все значения во view, можно использовать метод **force,** который вернет преобразованную коллекцию исходного типа.

Рассмотрим особенности реализации конкретного view - Mapped

```
new { val mapping = mapFunction } with AbstractTransformed[B] with Mapped[B]

// in SeqViewLike
trait Mapped[B] extends super.Mapped[B] with Transformed[B] {
  def length = self.length
  def apply(idx: Int): B = mapping(self(idx))
}

//in TraversableViewLike
trait Mapped[B] extends Transformed[B] {
  protected[this] val mapping: A => B
  def foreach[U](f: B => U) {
    for (x <- self)
      f(mapping(x))
  }
  final override protected[this] def viewIdentifier = "M"
}
```

Нестрогие вычисления

Задания

lectures.eval.LazySchedulerView

Нестрогие вычисления

Streams

Stream - это Seq, чьи методы вычисляются лениво. Они предназначены для представления данных с неизвестной размерностью (потенциально бесконечной). В отличие от view, потоки сохраняют вычисленные значения. Поэтому, несмотря на то, что stream-ы представляют неограниченные наборы данных, чаще всего вычислить можно только конечное кол-во значений.

Наиболее востребованные методы

- **#::** - prepend элементов стрима, например: **1 #:: 2 #:: 3 #:: Stream.empty**
- **head** - первый элемент стрима
- **tail** - стрим представляющий собой оставшиеся члены стрима
- **take(n:Int)** - метод предназначенный для ограничения элементов стрима. Этот метод тоже ленив и не приводит к вычислению реальных значений
- **force()** - вычисляет все значения стрима, если вызвать на бесконечном стриме, приведет к зависанию или к RuntimeException
- **append(rest: => ...)** - ленивая реализация конкатинации стрима со значениями, содержащимися в переданном TraversableOnce[A]
- **map, flatMap** - ленивые реализации привычных map и flatMap

Нестрогие вычисления

Streams

```
// Метод вычисляющий степени 2
def powerOf2(): Stream[Int] = Stream[Int](1) append {
  powerOf2().map(_ * 2)
}

powerOf2().take(5).force

// вычисление чисел фибоначчи
def fibFrom(a: Int, b: Int): Stream[Int] = a #:: fibFrom(b, a + b)

val fibs2 = fibFrom(1, 1).take(7)

// вычисление чисел фибоначчи другим способом
val fibs: Stream[BigInt] = BigInt(0) #:: BigInt(1) #:: fibs.zip(fibs.tail).map { n => n._1 + n._2 }
```

Нестрогие вычисления

Structured types (ST)

ST - это способ определить тип, описав сигнатуру его членов. Этот механизм можно использовать для реализации Duck Typing. Описание ST представляет собой заключенные в фигурные скобки описания членов типа и может присутствовать почти везде, где могут быть обычные типы

```
def foo(x: { def get: Int }) = 123 + x.get

def foo2[T<: { def get: Int }] (x: T) = 123  +  x.get

class Foo {
  type T = {def get: Int}
  def foo(x: T)  = 123  +  x.get
}
```

Использование ST может привести к заметной деградации производительности приложения т.к. для реализации ST применяется reflection.

Dependent types

Path dependent types (PDT)

Path dependency - это свойство вложенных типов scala. Допустим, внутри класса **Cnt** объявлен внутренний тип **Inner**. Тогда все использования **Inner**, без спецификации полного “пути” типа, будут path dependent. Это значит, что для каждого объекта внешнего класса будет определен свой внутренний тип, не равный типу в других объектах. Полный путь типа можно описать с помощью символа **#.** Например полный путь типа Inner будет Cnt#Inner. Путь может быть описан для любой глубины вложенности типа.

Dependent types

Path dependent types

```
import scala.reflect.runtime.universe._

object Cnt {
  def tagged[T: TypeTag](t: T) = typeTag[T]
}

class Cnt {
  class Inner
  def getInner = new Inner
  def doPathIndependent(t: Cnt#Inner) = println("Path dependent")
  def doPathDependent(t: Inner) = println("PathDependent")
}
val cnt = new Cnt
val cnt2 = new Cnt
val in = cnt.getInner
val in2 = cnt2.getInner
val inT = Cnt.tagged(in)
val in2T = Cnt.tagged(in2)
cnt.doPathIndependent(in)
cnt.doPathIndependent(in2)
cnt.doPathDependent(in)
//cnt.doPathDependent(in2) won't compile due to  path dependency
assert(!(inT.tpe =:= in2T.tpe))
```

Асинхронность

В Scala применяется та же самая модель памяти, что и в Java. Соответственно, доступны все те же механизмы управления потоками, что и в Java, следовательно, можно:

- синхронизироваться на мониторах объектов: **someInstance.synchronized{}** или даже **this.synchronized{...}**
- создавать **volatile** методы и переменные, с помощью аннотации **@volatile**
- использовать **ReentrantLock**, барьеры, семафоры и другие примитивы
- атомарные конструкции, например **AtomicReference**
- использовать конкурентные коллекции из Java и Scala, например, **TrieMap** из **scala.collection.concurrent** или параллельные коллекции из **scala.collection.parallel**

Несмотря на обширность инструментов работы в многопоточной среде, Scala представляет 2 инструмента, которые можно считать наиболее предпочтительными для написания многопоточного функционального кода. Это **scala.concurrent.Future** и Akka actors. Оба инструмента инкапсулируют в себе особенности работы с памятью и потоками, что позволяет разработчику сфокусировать свое внимание на логике приложения.

Асинхронность

Параллельные коллекции

Начнем рассматривать специальные инструменты Scala для работы в многопоточной среде с более специфичной технологии - параллельных коллекций.

Параллельные коллекции предназначены для выполнения последовательных ассоциативных операций над коллекциями в несколько потоков.

Из любой коллекции можно получить параллельную реализацию с помощью метода **par**

```
val l =  List(1,2,3,4,5,6)

val copied = l.par

val ar = Array(1,2,3,4,5,6,7)

val wrapped = ar.par
```

Чтобы получить однопоточную версию коллекции из параллельной нужно вызвать метода **seq**

Асинхронность

Ограничения параллельных коллекций

- Прирост производительности заметен для коллекций с количеством элементов >> 10k
- Параллельные коллекции сложно контролировать и отлаживать
- Операция должна быть ассоциативной

```
val list = (1 to 1000).toList.par.reduce(_ - _)
list: Int = 221998

val list1 = (1 to 1000).toList.par.reduce(_ - _)
list: Int = 221998

val list2 = (1 to 1000).toList.par.reduce(_ - _)
list2: Int = -238948

val list3 = (1 to 1000).toList.par.reduce(_ - _)
list3: Int = 73500
```

Асинхронность

Параллельные коллекции

В зависимости от базовой коллекции, метод **par** вернет одну из параллельных реализаций

- **immutable.ParVector -** Иммутабильная копия Vector, List или Stream. Создание занимает линейное время
- **immutable.ParHashMap -** Имутабильная версия immutable.Map. Создание занимает линейное время
- **immutable.ParHashSet -** Имутабильная версия immutable.Set. Создание занимает линейное время
- **mutable.ParArray -** Параллельная версия мутабильных коллекций типа ListBuffer и Array. Создание занимает константное время
- **mutable.ParTrieMap -** Параллельная версия concurrent.TrieMap

Полный список конкретных параллельных коллекций и их характеристики можно найти в документации

Асинхронность

Ограничения параллельных коллекций

- Легко столкнуться с race condition, dead lock и т.д.

```
var sum = 0

val list = (1 to 1000).toList.par
list.foreach(sum += _)
sum
res1: Int = 439037

sum = 0
list.foreach(sum += _)
sum
res3: Int = 13964

sum = 0
list.foreach(sum += _)
sum
res5: Int = 456993
```

Асинхронность

Scala Futures (F) and Promises (P)

F - это трейт, который представляет собой контейнер над асинхронной задачей. Он позволяет применять к задачам методы композиции и трансформации, аналогичные методам Traversable. При этом детали реализации, касающиеся работы с потоками и памятью, практически полностью скрыты внутри реализации.

Для того, чтобы создать асинхронную задачу, необходимо

- вызвать **def apply[T](body: =>T).** Пример ниже асинхронно напечатает слово “Hi”
- импортировать в контекст или передать явно **ExecutionContext.** EC отвечает за асинхронное выполнение кода, переданного **Future.apply**.

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

Future{
 print("Hi")
}
```

Асинхронность

Scala Futures (F) and Promises (P)

Каждый раз, когда мы создаем F, “под капотом” создается Task. Task может быть выполнен синхронно в том же потоке, но скорее всего будет выполнен асинхронно. По умолчанию Scala использует ForkJoinPool для хранения и исполнения задач. Задачи могут выполняться не в том порядке, в каком они были созданы, тем не менее, у разработчика есть способ влиять на порядок их выполнения. Это и многое другое можно сделать с помощью функций комбинаторов, описанных чуть ниже.

Исключительные ситуации (кроме фатальных), случившиеся внутри Future, не распространяются на код снаружи. Вместо этого они влияют на тип возвращаемого значения. Таким образом, удобно думать о Future как об асинхронных Try[T]

Часть 1. Асинхронность

Scala Futures (F) and Promises (P)

Статические функции для работы с Future:

- **Future.sequence -** превращает последовательность Future в Future от последовательности результатов.
- **Future.fromTry -** синхронно создаст завершенную Future из Try[]
- **Future.successful -** так же синхронно создаст успешно завершенную Future, содержащую значение, переданное в этот метод
- **Future.traverse -** применит к каждому элементу из TraversableOnce функцию, возвращающую Future, и вернет Future от последовательности результатов

Часть 1. Асинхронность

Scala Futures (F) and Promises (P)

Callback функции

- **future.onComplete, onFailure, onSuccess** - методы, позволяющие передать колбэк, который будет выполнен в зависимости от того, как завершилась фьюча. Методы возвращают Unit и предназначены для выполнения каких-то побочных действий. Может быть задано произвольное количество колбэков. Порядок их определения не влияет на порядок вызов.
- **andThen -** позволяет определить несколько побочных функций. Функции будут выполнены в том порядке, в котором переданы

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util._

val f = Future {5}
f andThen {
 case r => sys.error("runtime exception")
} andThen {
 case Failure(t) => println(t)
 case Success(v) => println(v)
}
```

Часть 1. Асинхронность

Scala Futures (F) and Promises (P)

Методы-комбинаторы **map, flatMap, filter, collect** и т.д. имею тот же смысл, что и функции, описанные в **Traversable**. Используя их, нужно помнить, что каждое применение этих функций порождает новый таск, который кладется в очередь на выполнение. Иногда имеет смысл сэкономить несколько тасков, объединив несколько действий в одну функцию.

Т.к. над Future определены методы **foreach, map, flatMap** и **withFilter**, для композиции фьюч есть возможность применять for comprehension.

Значение Future можно получить несколькими способами. Методы из Await сопряжены с риском возникновения ошибок в приложении и должны быть использованы с осторожностью.

- **Await.result** - блокирует текущий поток и ждет в течение duration результат. Если Future не завершается в отведенное время, будет брошен TimeoutException
- **Await.ready** - блокирует текущий поток и ждет завершения фьючи.
- **future.value** - возвращает **Option[Try[T]]**. Если фьюча еще не завершилась, вернет None

Часть 1. Асинхронность

Scala Futures (F) and Promises (P)

Promise - это трейт, объекты которого могут иметь одно из трех состояний:

- незавершенное
- завершенное
- связанное с другим P

Promise чаще всего используется, для того, чтобы получить объект Future, условие завершения которого находиться за пределами этого Future.

Методы P:

- **Promise.apply** - создает новый инстанс P
- **Promise.failed** - создает P, завершенный неудачно
- **Promise.successful** - создаст P, завершенный удачно
- **p.future** - возвращает объект future, связанный с состоянием текущего P
- **p.complete(Try[T])** - завершает P. Если Try завершится удачно, то P тоже будет завершен удачно, иначе P завершится с ошибкой
- **tryComplete, completeWith, tryCompleteWith, success** и т.д. - способы так или иначе завершить P

Часть 1. Асинхронность

Scala Futures (F) and Promises (P)

Правила написания надежного кода с Future и Promise

- Если метод возвращает Future[T], он никогда не должен кидать ошибок
- Старайтесь избегать методов из Await и вообще старайтесь не смешивать синхронный и асинхронный код
- Если избежать Await или других блокирующих вызовов внутри Future невозможно, создайте для таких фьюч отдельный контекст (или несколько контекстов). Сделайте так, чтобы исчерпание потоков в этом контексте не влияло на функционирование приложения
- Будьте аккуратны с комбинаторами. Помните, что каждый map, filter и т.д. - это новая таска в контексте
- Если нужно создать фьючу от известного значения, используйте **Future.successful**, а не Future {}, т.к. последний создаст ненужный таск.

Часть 1. Асинхронность

Scala Futures (F) and Promises (P)

Пример **lectures.concurrent.PromiseExample**

Часть 1. Асинхронность

Scala Futures (F) and Promises (P)

Задание: **lectures.concurrent.Smooth.scala**

Akka

Akka представляет семейство фреймворков для создания приложений, удовлетворяющих требованиям:

- параллельности вычисления
- устойчивости к ошибкам и падениям
- масштабируемости
- высокой производительности

Официальная документация

Akka: Actor

Задачи актора:

- Получение, обработка и отправка сообщений
- Определение поведения для следующих сообщений
- Управление другими акторами

Модель актора:

- Обработка данных - поведение актора
- Хранение данных - состояние актора
- Ввод-вывод - обмен сообщениями

Akka: Actor

Ключевые особенности:

- Сообщения обрабатываются последовательно
- Все акторы работают одновременно
- Актор - это НЕ поток
- Актор может блокироваться (но нельзя злоупотреблять)
- Нет общего состояния
- Гарантия доставки сообщения: не более одного раза
- Сообщения упорядочены для каждого отправителя

Akka: Actor example

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)

  def receive = {
    case "test" => log.info("received test")
    case _      => log.info("received unknown message")
  }
}

object AkkaExample extends App {
  val system = ActorSystem("mySystem")
  val myActor = system.actorOf(Props[MyActor], "alias")
  myActor ! "test"
  myActor ! "another test"
  system.shutdown()
}
```

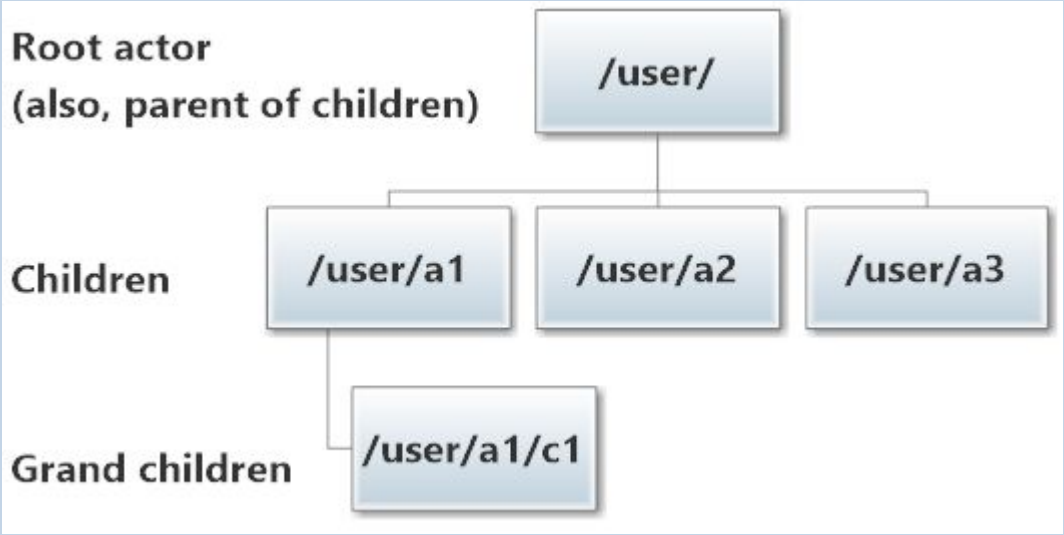
Akka: Messages

- Любой объект (Any)
- Immutable
- Serializable
- Суть - протокол взаимодействия

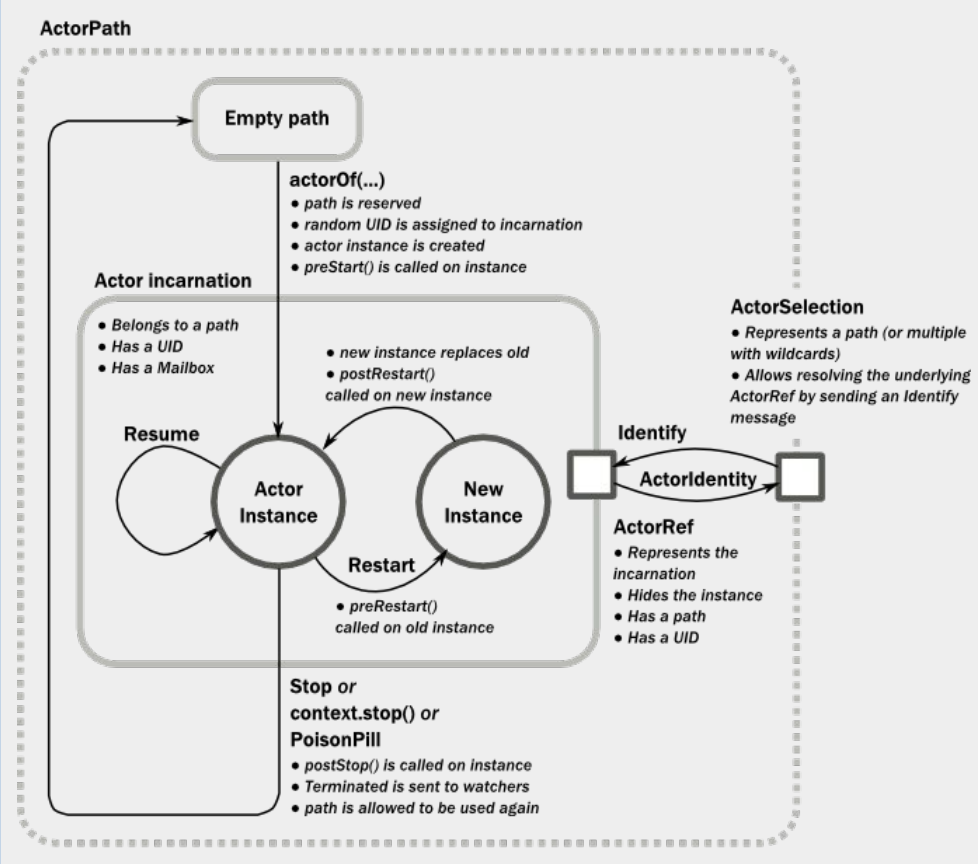
```
object ExampleProtocol {
  trait Response
  case class Success(data: String) extends Response
  case class Failure(error: String) extends Response
}
```

Akka: Nesting actors

- Supervisor strategy
  - Resume, keep state
  - Restart
  - Stop permanently
  - Escalate failure



Akka: Lifecycle



Akka: Features

- Akka remote
  - akka://my-sys/user/service-a/worker1
  - akka.tcp://my-sys@host.example.com:5678/user/service-b
- become / unbecome
- stash / unstash / unstashall
- Routing
  - Strategy: Round-robin, Smallest mailbox
  - Router / Pool / Group
- Dispatching
- Ask pattern (with timeout)

```
def ?(message: Any)(implicit timeout: Timeout): Future[Any]

val resultFuture = myActor ? "test"
```

Akka example

Пример реализации задачи про пинг-понг:

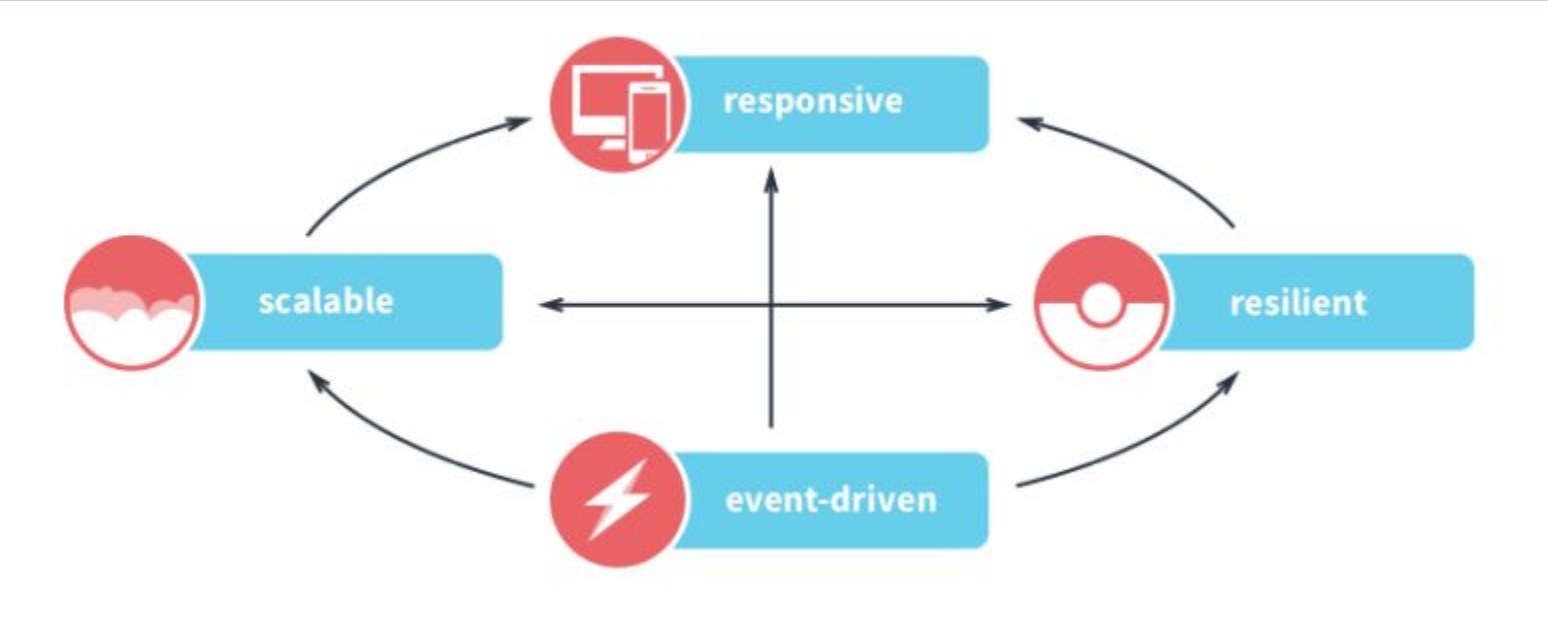
lectures.concurrent.akka.AkkaPinPongExample

Reactive Programming

- Императивное программирование:
  - a := b + c
- Реактивное программирование
  - Автоматическое “проталкивание” изменений
  - Применение:
    - Интерактивные интерфейсы, MVC
    - Отслеживание цепочки событий
  - Виды:
    - Императивное и объектно-ориентированное
    - Функциональное (Push / Pull)

Reactive Manifesto

http://www.reactivemanifesto.org/



Reactive Manifesto

http://www.reactivemanifesto.org/

Responsive

- Ответ за разумное время, есть верхний предел длительности
- Баланс между юзабилити и сложностью
- Проблемы быстро обнаруживаются и эффективно устраняются
- Упрощение обработки ошибок (таймауты, меньше дублей запросов)

Reactive Manifesto

http://www.reactivemanifesto.org/

Resilient

- Работоспособность при падениях (responsive)
- Достигается при помощи:
  - Репликации
  - Изоляции сбоев
  - Делегирования
- Отдельные компоненты падают и восстанавливаются без влияния на систему в целом
- Есть супервизор, следящий за работой компонента и рестартующий его при необходимости
  - Иерархия супервизоров (Akka-like)
  - Выбор супервизора в зависимости от серьезности ошибки
- Клиенты компонента об этом не заботятся

Reactive Manifesto

http://www.reactivemanifesto.org/

Elastic

- Система стабильно реагирует под изменяющейся нагрузкой
- Адаптирование используемых ресурсов под нагрузку
- Достигается за счет:
  - Отсутствия единого узкого места
  - Возможность шардирования/репликации и распределения нагрузки между отдельными частями
  - Мониторинг/предсказание нагрузки
  - Адекватный запас прочности
  - Автоматизация развертывания/свертывания ресурсов на основе текущих данных
  - Эффективное использование большого числа недорого железа/софта

Reactive Manifesto

http://www.reactivemanifesto.org/

Message-driven

- Асинхронная обработка сообщений
- Разграничение компонентов:
  - Слабая связанность
  - Изолирование
  - Location Transparency (один хост, целый кластер, несколько ДЦ)
  - Ошибки - так же сообщения
- Плюшки работы с сообщениями:
  - Единый мониторинг производительности
  - Управление нагрузкой и потоком сообщений
  - Back pressure
- Неблокирующая обработка, экономия ресурсов

Reactive Manifesto

http://www.reactivemanifesto.org/

Это всё понятно, а что же делать-то?

- Reactive programming = Programming with asynchronous data streams
  - Akka Streams (link)
  - ReactiveX (link)
  - Monix (link)