



Введение в Scala

О курсе

Цель

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



+



О курсе

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

Ресурсы

Скачать

- [Текущая версия scala](#)
- [IntelliJ IDEA](#)
- [Java Dev. Kit 1.8](#)
- [Клиент GIT](#) + популярный GUI [Tortoisegit](#) для Win; [Sourcetree](#) для MAC
- [SBT](#)
- GitHub школы -

Введение

Классификация

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

Введение

Классификация

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

Введение

Scala - язык программирования с множеством парадигм

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

Введение

Примеры

Объектно ориентированный, императивный подход

```
class Executor(msg: String){  
  def execute() = print(msg)  
}  
  
class ExecutorService {  
  def execute(ex: Executor): Unit = {  
    ex.execute()  
  }}  
  
val es = new ExecutorService()  
val e = new Executor("hello world")  
es.execute(e)
```


Введение

Примеры

Декларативный, функциональный подход

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

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

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

Введение

Примеры

Развитый вывод типов

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

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

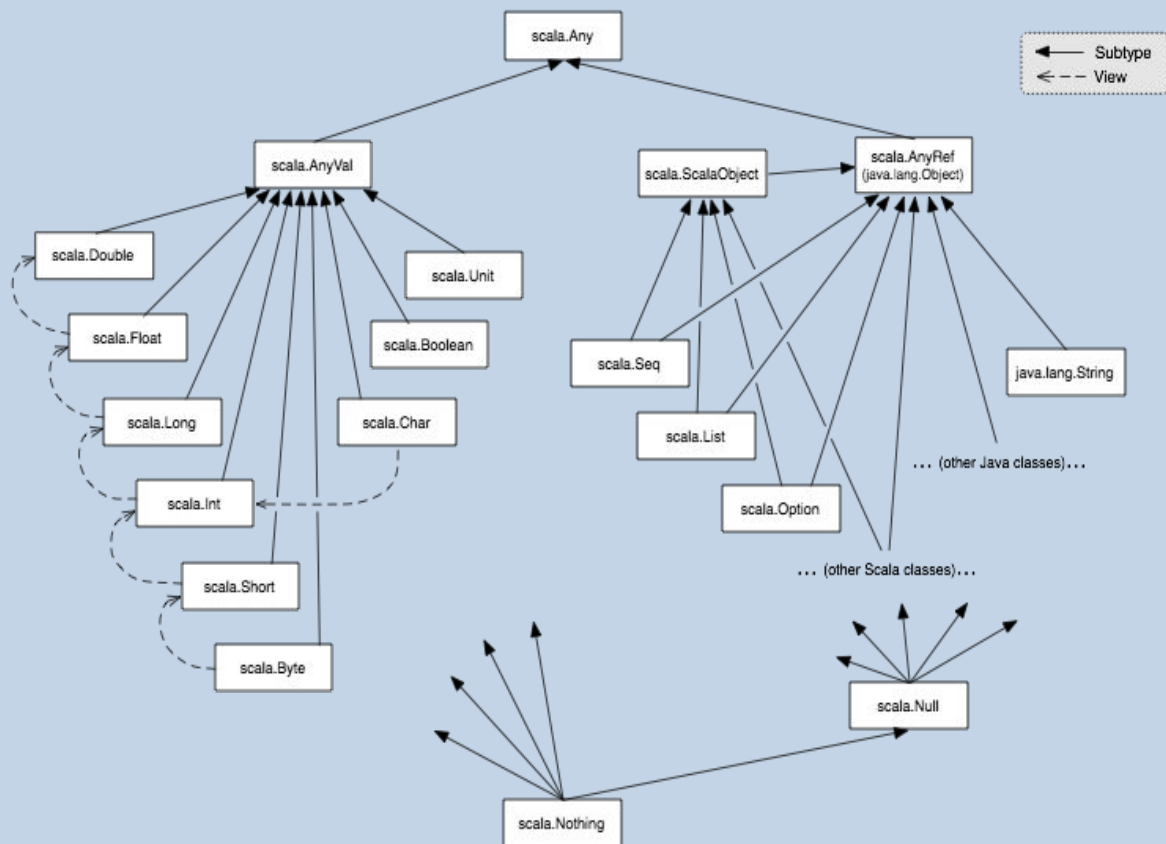
```
def result = calculateSomething + 3 + printSomething
```

```
result
```



Часть 1. Основы Scala

Часть 1. Типы



Часть 1. Типы

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

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

Часть 1. Типы.

Вывод типов

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

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

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

Часть 1. Типы

Вывод типов

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

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

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

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

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

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

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

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

Часть 1. Типы

Вывод типов

Синтаксический сахар, связанный с выводом типов

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

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

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

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


Часть 1. Типы

Вывод типов

Когда вывод типов не работает

- когда неизвестен как минимум один из типов участвующий в операции. 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

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

```
// WITH DEFAULT VALUE
```

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

```
// OMIT RETURN TYPE
```

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

```
// OMIT RETURN TYPE AND BODY BRACES
```

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

Значением функции, является значение последнего в ней выражения

Часть 1. Функции

Процедуры

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

Переменная длинна аргументов

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

```
name(2, "a")
```

```
name(2, "a", "6")
```

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

Часть 1. Функции

Функции могут быть значениями

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

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

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

Все параметры переданные в функции являются константами

Часть 1. Функции

Каррирование.

Еще один способ выразить в скале понятие функций высшего порядка

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

// каррированный аналог предыдущей функции

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

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

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

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

Часть 1. Функции

Композиция функций одной переменной

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

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

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

```
powAndShow(10)
```

Часть 1. Функции

Композиция функций нескольких переменных

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

Рассмотрим их на примере функции от 2-х переменных

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

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

Часть 1. Функции

Композиция функций нескольких переменных

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

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

Часть 1. Функции

Call-by-name параметры или лень в помощь

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

Параметры, переданные по имени имеют несколько особенностей

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

Часть 1. Функции

Разрешение циклических зависимостей

```
class Application {  
  
  class ServiceA(c: => ServiceC){  
    def getC = c  
  }  
  class ServiceC(val a: ServiceA)  
  
  def a: ServiceA = new ServiceA(c)  
  lazy val c: ServiceC = new ServiceC(a)  
}  
  
val app = new Application()  
val a = app.a  
a.getC
```

Часть 1. Функции

Повторное вычисление

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

Часть 1. Функции. Задания

Подсчитать числа Фибоначчи

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

lectures.functions.Fibonacci

Реализовать более эффективный способ вычисления чисел Фибоначчи

lectures.functions.Fibonacci2

Освоить каррирование и функции высшего порядка

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

Воспользоваться композицией функций для написания простого DB API

lectures.functions.SQLAPI

Часть 1. Операторы

Условный оператор

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

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

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

Часть 1. Операторы

Циклы.

В scala 3 основных вида цикла

- **while** - повторяет свое тело пока выполняется условие
- **for** - итерируется по переданной в оператор коллекции или интервалу (Range)
 - в одном операторе можно итерироваться сразу по нескольким коллекциям
 - оператор позволяет фильтровать члены коллекции, по которым итерируется, с помощью встроенного оператора if
 - оператор позволяет определять переменные между вложенными циклами
- **for {} yield {}**. Если перед телом цикла стоит слово **yield**, то цикл становится оператором, возвращающим коллекцию. Тип элементов в итоговой коллекции зависит от типа возвращаемого телом цикла

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

Часть 1. Операторы

// ВЫВЕДЕТ ВСЕ ЧИСЛА ВКЛЮЧАЯ 100

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

// ВЫВЕДЕТ ВСЕ ЧИСЛА ИСКЛЮЧАЯ 100

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

Часть 1. Операторы

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

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

Часть 1. Операторы

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

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

Часть 1. Операторы. Задания

По тренируйтесь в написании циклов и условных операторов

lectures.operators.Competition

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

Часть 1. Pattern matching

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

```
val x:Int = 10

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

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

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

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

Часть 1. Pattern matching

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

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

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

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

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

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


Часть 1. Pattern matching

Возможности Pattern matching в scala

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

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

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

Часть 1. Pattern matching

Pattern matching для кейс классов

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

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

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

```
kremlin match {
```

```
  case inRussian@Address("Russia", _, _, _) => print(inRussian.city)
```

```
  case inUSA@Address("USA", _, _, _) => print(inUSA.city)
```

```
  case somewhereElse => print("Terra incognita!")
```

```
}
```

Часть 1. Pattern matching

Pattern matching для коллекций

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

printList(list)
```

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

Часть 1. Pattern matching. Задания

Разберите вещи по коробкам, воспользовавшись `pattern matching`
`lectures.matching.SortingStuff`

Часть 1. Partial functions

Partial functions

Понятие partial function пришло из математики. Оно обозначает функцию, для которой область определения содержит лишь часть числовой прямой. В scala, partial function обозначает функцию, для которой область определения вычисляется.

PartialFunction - это функция одного аргумента (Function1)

```
// 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)
```

Часть 1. 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

```
// 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 PartialFunction с pattern Matching

Часть 1. 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`.

```
val list = List(1, 2, 3, 5, 6, "4", "2", pf, pf2)  
//list.isDefinedAt(pf _) // no such signature  
list.isDefinedAt(1) // strange method in List  
  
// List[Any] -> List[Int]  
list.collect {  
  case i: Int => i  
}
```

Часть 1. Partial functions. Задания

Помогите реализовать авторизацию.

`lectures.functions.Authentication`

Часть 1. Коллекции

Обзор коллекций

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

Иерархия коллекций в скале имеет более разветвленную структуру, чем в java, это связано с желанием создателей языка разделить интерфейсы на более мелкие части, что бы повысить переиспользуемость кода и лучше выделить семантические единицы реализации.

Часть 1. Коллекции

Трейты, составляющие основу коллекций в scala

- `Traversable[+A]`. Этот трейт принято считать корнем иерархии коллекций. Он отражает концепцию функций, по которым можно итерироваться. Содержит абстрактный метод `foreach` и реализации многих методов, реализуемых через `foreach`. Реализации предоставленные 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 параллельно
- `Seq`, `IndexedSeq`, `LinearSeq` - трейты обозначающие последовательность элементов. (Списки, потоки, вектора, очереди...)
- `Set` - определяет коллекции, не содержащие повторяющиеся элемента.
- `Map` - корневой трейт для ассоциативных массивов

Часть 1. Коллекции

Методы Traversable

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

Часть 1. Коллекции

Часто используемые коллекции.

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

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

Часть 1. Коллекции

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

// разделить все элементы на 2
List(1,2,3,4,5,6,7,8,9,0).map(_ % 2)
// затем реализовать тоже 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 an exception
letterPosition("g")
letterPosition.get("g")// == None
```

Часть 1. Коллекции

Option. Some. None.

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

Option[T] имеет 2 наследника: **Some** и **None**

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

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

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

Часть 1. Коллекции. Задания

Реализовать класс MyList

`lectures.collections.MyListImpl`

Избавиться от NPE

`lectures.collections.OptionVsNPE`

Написать сортировку слиянием.

Постарайтесь не использовать мутабельные коллекции и **var**

Подробнее о сортировке можно посмотреть [здесь](#).

`lectures.collections.MergeSortImpl`

Часть 1. 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** не доступен 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.

Часть 1. 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") } } }
```

Часть 1. 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") } } }
```

Часть 1. Коллекции

For comprehension(FC)

Если в 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"
    }
  }
}
```

Часть 1. Коллекции. FC. Задания

For comprehension(FC)

Перепишите код в соответствии с условиями задачи.

lectures.collections.comprehension.Couriers

Часть 1. Коллекции.

Tuples

Tuple или 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.

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

Другие полезные функции tuple

- `productPrefix` - строка соержжащая имя класса
- `productIterator` - итератор, которым можно пройти по порядку все члены tuple
- `productArity` - размерность
- `productElement(idx: Int): Any` - получает idx-ый член tuple, при этом информация о типе теряется. Если члена с таим индексом нет, мы получим **IndexOutOfBoundsException**

Часть 1. Коллекции.

Tuples

```
val tpl1 = Tuple1(1)
val tpl2 = 1 -> "String" //
val tpl2i2 = (1, "String")
val tpl3 = (3, "Strig", List.empty[Long])
val tpl4 = (3, "Strig", List.empty[Long], (x: Int) => print(x))
```

```
tpl1.productPrefix
tpl4.productPrefix
tpl1.productIterator
tpl1.productArity
```

```
// would throw IndexOutOfBoundsException
//tpl1.productElement(2)
```

Часть 1. Конструкции языка

Класс

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

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

Часть 1. Конструкции языка

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.

Часть 1. 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()  
  }  
}
```

Часть 1. Конструкции языка

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

```
  def publicMethod() {  
    print("public method")  
  }
```

```
  // This constructor inaccessible from outside
```

```
  private def privateMethod() {  
    print("private method")  
  }  
}
```

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

```
testClassInstance.int
```

```
testClassInstance.str
```

```
testClassInstance.publicMethod()
```

```
// inner is not a member of the class
```

```
//testClassInstance.inner
```

```
// inaccessible from outside
```

```
//testClassInstance.privateMethod()
```

Часть 1. Конструкции языка

Конструктор

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

Часть 1. Конструкции языка

Конструктор

```
// This constructor 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
    }
}
```

Часть 1. 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.

Часть 1. 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
```

Часть 1. Конструкции языка

Абстрактный класс

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

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

Часть 1. Конструкции языка

Trait

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

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


Часть 1. Конструкции языка

Объекты. Объекты компаньоны

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

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

Часть 1. Конструкции языка

Чем полезны объекты-компаньоны

- в объекте-компаньоне удобно задавать статические данные, доступные всем экземплярам этого типа
- метод `apply` используют, как фабрику объектов данного типа
- метод `unapply` используют для декомпозиции объектов в операторе присвоения и `pattern matching` -ге
- имплициты, определенные в объекте компаньоне, доступны внутри класса

Часть 1. Конструкции языка

Кейс классы

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

Отличия от стандартных классов

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

Часть 1. Конструкции языка

//Good case class

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

//COMPILATION ERROR

```
case class SuperClass(int: Int)
```

```
case class SubClass(int: Int) extends SuperClass(int)
```

//COMPILATION ERROR

```
case class NoMembers
```

// Don't do this

```
case class BadSignature(int: Int) {
```

```
  def this(int: Int, long: Long) = {
```

```
    this(int)
```

```
  }
```

```
}
```

Часть 1. 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, что бы 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)
```

Часть 1. 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 метода **apply** с сигнатурой из примера выше.

Часть 1. 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)
```

Часть 1. 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

Часть 1. 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 */
  //...
}
```

Часть 1. Конструкции языка

Unapply в операторе присвоения

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**.

Часть 1. 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**

Часть 1. Задания

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

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

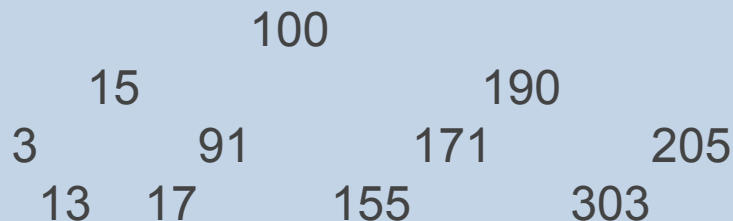
Задача 2. Доработать дерево. Поиск

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**, .

Часть 1. Задания



Для наглядности можно, заменить отсутствующих потомков значением '-1'

Задача 4. Метод fold для дерева

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

Часть 1. Тестирование

Тесты - это приложения, которые проверяют приложения

классификация тестирования:

- unit test - тест небольшой части приложения, функции, реализации какого-либо интерфейса
- functional(system) test - тестирование крупной подсистемы приложения “в сборе”
- validation & verification - тест всего приложения на соответствие требованиям. Очень часто проводится вручную
- smoke test - проверка на соответствие требованиям всего приложения
- performance tests (stress test, resilience test) - категория тестов направленная на проверку “спортивной формы” приложения.

white box - тестирование с учетом знания реализации приложения. Этот подход чаще применяется для unit тестирования.

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.

Часть 1. 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

Часть 1. Тестирование

Часто употребляемые термины

- Test Driven Developments (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

Часть 1. 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. Что бы было понятнее сразу перейдем к примерам

lectures.collections.MergeSortImpFunSuiteTest

lectures.collections.MergeSortImplFlatSpecTest

lectures.collections.MergeSortImplWordSpecTest

lectures.oop.BSTTestWithMocks

Часть 1. 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 соответствующей странице на [сайте ScalaTest](#)

Примеры

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

Часть 1. Тестирование. Задания

Завершите реализацию теста для SortingStuff

lectures.matching.SortingStuffGeneratorBasedTest

Напишите тест для Authentication

lectures.functions.AuthenticationTest

Часть 1. Исключительные ситуации

Исключительные ситуации

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

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

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

Часть 1. Исключительные ситуации

```
class TestClass {  
  
    @throws[Exception]("Because i can")  
    def methodWithException(): Int =  
        throw new Exception("Exception thrown")  
  
    def methodWithoutException() = {  
        print(methodWithException())  
    }  
}  
  
val t = new TestClass()  
// Method would throw an exception  
t.methodWithoutException()
```

Часть 1. Исключительные ситуации

Обработка исключений

Существует 2 принципиально разных подхода: императивный и функциональный

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

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

Часть 1. Исключительные ситуации

```
import java.sql.SQLException

class TestClass {

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

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

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

Часть 1. Исключительные ситуации

Обработка исключений

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

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

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

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

Часть 1. Исключительные ситуации

```
class TestClass {  
  
  @throws[Exception]("Because i can")  
  def methodWithException(): Int =  
    throw new Exception("Exception thrown")  
  
  def methodWithoutException(): Try[Unit] =  
    Try {  
      print(methodWithException())  
    }.recover {  
      case e: SQLException => print("sql Exception")  
      case e: Exception => print(e.getMessage)  
      case _ => print("would catch even fatal exceptions")  
    }.map {  
      case _ => println("Ooooh finally")  
    }  
}
```

Часть 1. Задания

Обработать исключения

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

```
object PrintGreetings {  
  
  case class Greeting(msg: String)  
  
  private val data = Array(Greeting("Hi"), Greeting("Hello"),  
    Greeting("Good morning"), Greeting("Good afternoon"),  
    null, null)  
  
  def printGreetings() = {  
    for (i <- 0 to 10) {  
      println(data(i).msg)  
    }  
  }  
}  
  
PrintGreetings.printGreetings()
```

Часть 1. 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 - это, 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)

Часть 1. ООП

SOLID ([wiki](#))

Single responsibility - у класса или функции должна быть четкая сфера ответственности

Open close principle - элементы приложения должны быть открыты для расширения, но закрыты для модификации

Liskov substitution - везде, где используется супер класс, может быть использован его саб класс

Interface segregation - много маленьких интерфейсов лучше чем один большой

Dependency inversion - любая реализация должна зависеть на абстракцию. Это касается не только отдельных частей приложения, но и всего приложения в целом.

Часть 1. ООП

Наследование в скала

Для того, что бы 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.

Часть 1. 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 extends object
class SubClassByObject extends SuperObject{}

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

Часть 1. 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`.

Ключевое слово

- **super** можно использовать для доступа к членам супер класс, 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.

Часть 1. 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
```


Часть 1. 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 2-ой и следующие предки. В выражении после **with** могут присутствовать только трейты

Часть 1. ООП

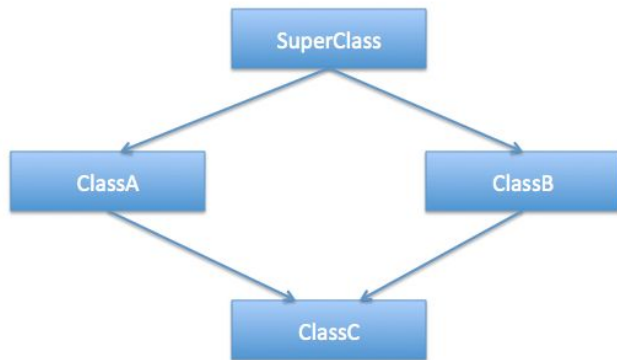
Наследование в скала

```
abstract class AbstractTest {  
  val name: String  
}  
  
trait NameProvider {  
  val name: String = "name provided by trait"  
}  
  
trait SomeMarker  
  
class ConcreteClass extends AbstractTest with NameProvider with  
  SomeMarker {  
}  
  
object InheritanceTest extends App {  
  val k = new ConcreteClass
```

Часть 1. ООП

Множественное наследование, diamond problem.

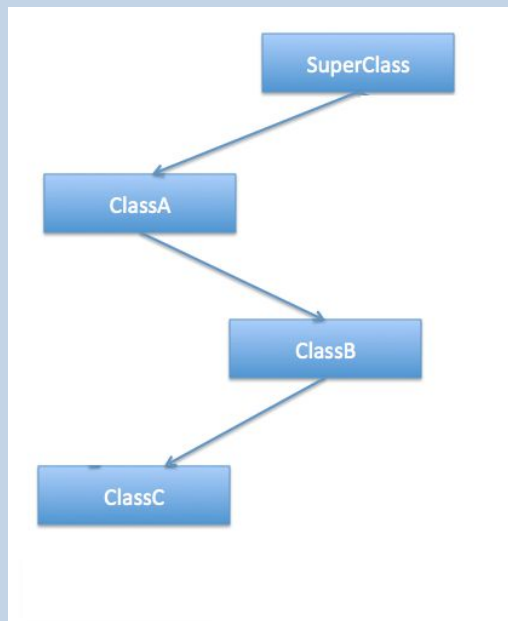
При множественном наследовании часто возникает вопрос: что делать, если несколько родителей предоставляют одинаковые члены класса. Из рисунка ниже понятно откуда происходит название проблемы.



Часть 1. 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.



Часть 1. ООП

Множественное наследование, линеизация.

Суть правила заключается в том, что 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 конструируемого класса
- Доступ к членам супер классов через ключевое слово **super** происходит в обратном порядке. Т.е. **super.memberName** обратиться к **memberName** ближайшего суперкласса, полученного в процессе линеизации.

Часть 1. ООП

Множественное наследование, линеизация

пример линеизации **lectures.oop.lineization.scala**

Часть 1. 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

Часть 1. 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
```


Часть 1. ООП

Self type annotation

Это механизм дополнительной спецификации типа трейта или класса. Аннотация говорит о том, что все экземпляры, 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 про аннотировать, 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 экземпляры текущего класса
- **with Type2** - опциональные, дополнительные, типы

Часть 1. 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()
```

Часть 1. 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

Часть 1. 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, что бы 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 так же 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

Часть 1. 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")
```

Часть 1. Parametric polymorphism

Type parameters (AKA generics)

```
/**
 * Развитие событий, мы хотим, что бы 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)
```

Часть 1. Parametric polymorphism

Type parameters (AKA generics)

Продолжим развивать наш **Binder**. Теперь мы хотим, что бы мы могли 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 (их еще 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 не существенно. Подставить TP на место ETP можно в процессе создания наследника в момент создания инстанса или вызова метода с ETP.

Часть 1. Parametric polymorphism

Type parameters (AKA generics)

Определим трейт, который будет обозначать группу методов для создания экземпляров 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.

Часть 1. 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
```

Часть 1. 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(10) {  
  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
```

Часть 1. Parametric polymorphism

Type parameters (AKA generics)

Binder3 - это, так называемый, higher kinded type (НКТ) .

Kind - тип, который порождает другой тип. Можно провести параллель между конструктором класса и НКТ. 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 (НКТ). 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 имплицитов, мы сможем добиться вот такой записи:

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

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

Часть 1. ООП

Type parameters (AKA generics)

Ограничение TP (type parameter bound, TPB) - это способ передать дополнительную информацию о TP. TPB можно, так же, 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 находиться 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

Часть 1. 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))  
}
```

Часть 1. Parametric polymorphism

Type parameters (AKA generics)

Вариативность (V For Variance). Для типов, принимающих ТР возникает вопрос, как ТР 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 ТР

- covariance. Обозначается символом '+' перед ТР. Если ТР 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 ТР. Для контравариантных типов выполняются условия, если `B <: A` и `N >: T`, `B[] <: A[T]`.
- invariance. Если перед ТР не стоит ни каких символов, такой тип называют инвариантным. Т.е. вне зависимости от отношения наследования между типам, принимающими ТР, типы, принявшие разные ТР 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

Часть 1. Parametric polymorphism

Type parameters (AKA generics)

Хороший пример ковариантности - коллекции в scala. Если Seq не был 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)
```

Часть 1. Parametric polymorphism

Type parameters (AKA generics)

Пример инвариантности. В примере ниже мы не сможем сложить 2 массива T.к. TP нашего листа инвариантен. Мы не сможем этого сделать, даже если поставим '+' перед TP, 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("bla", null)  
val node2 = ListNode[AnyRef] ("bla", null)  
  
// NOP ((  
node1.concat(node2)
```


Часть 1. ООП

Type parameters (AKA generics)

Контравариантность и 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)
```

Часть 1. Parametric polymorphism

Абстрактные типы (abstract type, АТ)

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

АТ применяются в следующих случаях:

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

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

Интервью Одерского на тему АТ vs ТР [на artima](#)

Часть 1. 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)
```

Часть 1. 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.

Часть 1. 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 ln = ListNode(10, null)  
val strList = List("")  
val valList = List(1,2,3,4)  
val str = "bad prm"  
ln.maybePrepend(str)  
ln.stringListPrepend(strList)  
ln.stringListPrepend(valList)  
ln.stringListPrepend(str)  
ln.generalListPrepend(strList)  
ln.generalListPrepend(valList)  
ln.generalListPrepend(str)
```

Часть 1. Parametric polymorphism

Задания

Измените lectures.collections.MyList, применив TP так, что бы следующие выражения require завершались 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, что бы 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)
```

Часть 1. Implicit

Implicit - 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 conversions, ImC)
- неявный контекст (implicit context, IC)

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

Имплиситными 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) { ... }
```

Часть 1. Implicit

Свойства имплицитов

- именованное - имплициты могут иметь любые имена, но для того, чтобы стать имплицитами их определение должно начинаться с ключевого слова **implicit**
- доступность - для того, чтобы имплицит был применен, он должен находиться в скоупе. Что бы поместить имплицит в скоуп можно:
 - определить его внутри класса, в котором он будет использован
 - импортировать с помощью ключевого слова `import`
 - определить в объекте-компаньоне
- однозначность - в скоупе не должно быть нескольких имплицитов с одной сигнатурой
- сначала явные - если вывод типов не выявил необходимости в применении имплицитов, они применены не будут, даже если доступны в скоупе.
- одноуровневость - компилятор не будет предпринимать попытки применить имплициты несколько подряд, чтобы добиться совпадения типа. Т.е. если в скоупе нет имплицита подходящего типа, имплициты применяться не будут.

Часть 1. Implicit

Implicit conversions (ImC)

ImC - это способ превратить объект одного типа в объект другого типа, без явной конвертации. Для того, что бы 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.

Часть 1. Implicits

Implicit conversions (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 conversion  
//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 `Regex`

Часть 1. 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`

Часть 1. 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, что бы 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, что бы 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 не доступен явно в теле функции. Для того, что бы получить к нему доступ используется метод **implicitly[T](implicit e: T)**

Часть 1. 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)  
}
```

Часть 1. Implicits

Type classes (TC) - это паттерн, позволяющий инстансы разных типов к общему классу типов (type class). Реализуется этот паттерн с помощью механизма имплицитов.

Трейт **Numeric[T]** из примера выше, представляет собой класс типов, для которых может быть определена операция сложения(add). В примере, к этому типу классов приводиться 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]**

Часть 1. 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))
```

Часть 1. Implicit

Цепочки имплицитов

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

Часть 1. 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)
```

Часть 1. ООП

Задания

lectures.oop.types.GeneralBST.scala

Часть 1.

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

Часть 1.

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 }
```

Часть 1.

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 }
```

Часть 1.

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.

Часть 1.

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 инстанса внешнего класса будет определен свой внутренний тип, не равный типу в других инстансах. 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.

Часть 1.

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) = print("Path dependent")
  def doPathDependent(t: Inner) = print("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))
```


Часть 1. 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 из 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.

Часть 1. 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, что бы создать асинхронную задачу необходимо

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

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

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

Часть 1. 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 фьючи, не распространяются на код снаружи. Вместо этого они влияют на тип возвращаемого значения. Таким образом удобно бывает думать о фьючах, как об асинхронных Try[T]

Часть 1. Асинхронность

Scala Futures(F) and Promises(P)

Статические функции для работы с Future

- **Future.sequence** - превращает последовательность фьюч во фьючу от последовательности результатов.
- **Future.fromTry** - синхронно создаст завершенную фьючу из Try[]
- **Future.successful** - так же синхронно создаст успешно завершенную фьючу, содержащую значение, переданное в этот метод
- **Future.traverse** - превращает переданный TraversableOnce во фьючу

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. имеют 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 фьючи можно 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 течении `duration` результат. Если фьюча 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

- не завершенное
- завершенное
- связанное с другим P

Promise чаще всего используется, для того, что бы получить объект Future, условие завершения которого находится 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 завершиться удачно, то P тоже будет завершен удачно, иначе P завершиться с ошибкой
- **tryComplete, completeWith, tryCompleteWith, success** и т.д. - способы так или иначе завершить P

Часть 1. Асинхронность

Scala Futures(F) and Promises(P)

Пример `lectures.concurrent.PromiseExample`

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, что бы исчерпание потоков в этом контексте не влияло на функционирование приложение
- Будьте аккуратны с комбинаторами. Помните, что каждый map, filter и т.д. - это новая задача в контексте
- Если нужно создать фьючу от известного значения, используйте **Future.successful**, а не Future{}, т.к. последний создаст ненужный task.

Часть 1. Асинхронность

Scala Futures(F) and Promises(P)

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

Часть 1. Асинхронность

Akka представляет семейство фреймворков для создания распределенных, устойчивых, масштабируемых приложений. В основе технологий Akka лежит идея акторов

Знакомство с этой идеей мы начнем с простого приложения

lectures.concurrent.akka.AkkaPinPongExample

и

страницы [документации](#)))