# 

Материалы курса  
“Введение в Scala”

<https://stepik.org/course/16243>

Транскрибация: Романова Светлана  
<https://stepik.org/users/169050646>

# Содержание

[**Содержание**](#_u078hsgjb4dp) **2**

[**Введение**](#_mteqhzf4cng8) **8**

[**Начало работы**](#_sshbtzse49fd) **10**

[2.1 Установка среды разработки и практика](#_jyn2bz8qbfch) 10

[2.2 Приложение](#_v2d5csix7gjn) 21

[2.3 Переменные и их области видимости](#_6l6gonnxsxn0) 23

[Значение val](#_tdzp2ctzdj70) 23

[Функция без параметров def](#_oufts16p93gm) 24

[Ленивые значения lazy val](#_kckowa3ihit2) 24

[Переменные var](#_yn9e05o2hqhr) 24

[Использование](#_rlj82oe2biea) 25

[Указание типа](#_r1xk6mamgsz) 25

[Области видимости](#_13m42z3r39ix) 27

[2.4 Пространства имен](#_pz804ct7563f) 28

[Пакеты](#_jrjup5z56v17) 28

[Импорт](#_euqy1oxkz6ew) 29

[Идентификаторы](#_a1lj57o391dd) 30

[Множественный импорт](#_kqvyxf4q83lk) 31

[**Типы**](#_s2lvhnhq8l7v) **33**

[3.1 Типы](#_rr45z3yrfpjt) 33

[Подтипы](#_o9sfn89s318f) 33

[Примитивные типы](#_r1i72mdeg8dp) 35

[Как работают подтипы](#_7hcnafa6ddv6) 37

[Практика](#_s5c4c9upzu1c) 38

[3.2 Числа](#_4cmbv58pxjzf) 44

[Числовые операции](#_rj42ptse41s1) 45

[Бинарные операции](#_e5xm9tnot6ga) 45

[Порядок операций](#_svvgxdjn9742) 47

[Большие числа](#_vb8xdlhla0ds) 49

[Практика](#_fourr05lcv0h) 50

[3.3 Булевые значения](#_5csm4omx0w7w) 55

[Операции над булевыми значениями](#_ida5ivo4lkbh) 56

[Равенство](#_nzoi2ep4nr3s) 57

[3.4 Строки](#_w4084c4uinex) 58

[Сложение](#_jc3gv8f62lvk) 59

[Интерполяция](#_puslbtm003wv) 59

[Переносы строк](#_rybrrbv9f5q4) 59

[Операции](#_24ljv2jh68h) 60

[Регулярные выражения](#_jxx7r9q5msjf) 61

[Приведение к строке](#_1nckk4uam1rr) 62

[Практика](#_z639kh5ll2ai) 62

[**Функции**](#_9iddi4ym0r42) **67**

[4.1 Методы](#_7k50ynvnu9vq) 67

[Вложенные методы](#_fzpnl8k064kf) 69

[Повторяемые параметры](#_cobbwsto3hnw) 70

[Значения по умолчанию](#_7lze79tyfq61) 70

[Именованные аргументы](#_42q5s0iz9pxh) 71

[Передача по имени](#_na37u71c0hcr) 71

[Передача блока](#_fsunhg3z7l7e) 72

[Рекурсия](#_dm8kyg4b4es4) 73

[Хвостовая рекурсия](#_ihdmn3awja84) 73

[Практика](#_qnvc59vox8cj) 74

[4.2 Функции](#_o6colxtl5gun) 77

[Лямбда-абстракция](#_4rcdenkw9dpw) 77

[Короткая запись](#_noqvdnnon6lp) 78

[Эта-конверсия](#_793x8luwi0aa) 78

[Функция - значение](#_gftk2w3ijpjn) 79

[Каррирование](#_5wk9uxvmebme) 79

[Композиция](#_64izmkg95fg3) 80

[Практика](#_18c1e5tud351) 81

[4.3 Обобщенные методы](#_qsljtknrnjlz) 85

[Практика](#_6dubrv3jcygp) 87

[**Операторы**](#_ezsaeom6q6ek) **93**

[5.1 Управляющие конструкции](#_sx26o3ik58r5) 93

[Условные выражения if else](#_2bn7vy1xev3i) 93

[While-циклы](#_abp6vibu1ppz) 95

[For-циклы](#_p1zt1ohesl68) 96

[5.2 Pattern matching (Сопоставление с образцом)](#_n8ogayoci8ki) 97

[Case class](#_1cka9ejavuv7) 100

[Коллекции](#_q51svy2ijwkn) 102

[Регулярные выражения](#_i82ub2ur4ho2) 104

[Комбинация](#_17ilmtm6cmxz) 107

[5.3 Частичные функции (Partial functions)](#_ub9h1r23duo8) 108

[Метод .collect](#_8jon8p1h1o6m) 109

[5.4 Кортежи](#_ca5x4k4j4rjj) 110

[Объявление кортежа](#_b3bcfs9s2myo) 110

[Распаковка кортежа](#_ljgqhayk7nyh) 111

[Обращение к элементу](#_1sbths8l37i4) 112

[Пары элементов](#_eqspm2duaix6) 112

[Практика](#_hjxoqed3ww88) 113

[5.5 Опциональные значения](#_bgsenevvcr6h) 116

[Замена пустого значения](#_uei6lhc3ztkg) 118

[Преобразование](#_3sh24ojjfa19) 119

[Фильтрация](#_lbiuolkn2u2x) 119

[Практика](#_nc14jwejqmlb) 120

[5.6 Either[A, B] - исключающее или](#_78ppftm0hsl0) 127

[Обработка ошибок](#_mhj97firg9jo) 129

[Преобразование](#_7yzt22fmgavt) 130

[Практика](#_5evv95eh0m2k) 130

[**Коллекции**](#_f7zlgql9tkxi) **135**

[6.1 Коллекции. Часть 1](#_gwsfazn3e38u) 135

[Виды коллекций](#_ntqmxbthmber) 135

[Массивы](#_sf3dfeuayioj) 136

[Строки](#_fxk1s7qgi5l1) 136

[Изменяемые коллекции](#_f53req8y20gj) 138

[Неизменяемые коллекции](#_rdm0p6reeucc) 139

[Практика](#_643xrqpsl3kb) 141

[6.2 Коллекции. Часть 2](#_qd3c2r3jut0b) 150

[Создание коллекций](#_6f7sfaesj1mo) 150

[Получение элемента](#_9tszmycudj2k) 151

[Информация](#_hwm7mbx2hfid) 152

[Субколлекции](#_h62w8siqhis8) 153

[Условный отбор](#_fupibkpg8szm) 155

[Отображения](#_8vqx472z70uy) 156

[Практика](#_i7uxtdtxp10v) 157

[6.3 For comprehension](#_4zvr9v9t45s0) 162

[Область применения](#_atx0c0ws76jb) 165

[Практика](#_1hl7ochw5q1w) 167

[**Классы**](#_rqnufsgyexn8) **170**

[7.1 Классы](#_xq5b2uv3gpoj) 170

[Определение класса](#_oqyf951jmer8) 170

[Создание экземпляра](#_99mlm5mrr69y) 171

[Параметры](#_vj7xcvcfh5nb) 172

[Инициализация](#_wxhchzw50yy) 173

[7.2 Абстрактные классы](#_euzmegjuyv16) 173

[Trait](#_h0jet1v9mnf3) 173

[Абстрактные классы](#_ja2fw46u2y6e) 174

[Создание экземпляров](#_d5jv1ssd3upi) 175

[Инициализация](#_9m0vl2laeu3x) 175

[Создание экземпляров](#_bkqnoml8gaws) 176

[Практика](#_ks2mj3vpc0fy) 176

[7.3 Объекты](#_3qvkr1avrkou) 182

[Инициализация](#_17sbmxbdpnr4) 182

[Компаньоны](#_rzwfdccvvw3l) 183

[Case object](#_naynmsqaovwg) 183

[7.4 Case-классы](#_b1hhilt6ymfh) 184

[7.5 Наследование](#_790cd7dh58il) 186

[Переопределение](#_te59a4hlnxnl) 188

[Наследование от классов](#_du49sg7cqq8r) 189

[Линеаризация trait](#_7s9nr8chvpqt) 191

[Практика](#_5xsz8upg1als) 192

[7.6 Модификаторы](#_iak69djdtg7h) 198

[Final](#_goxmunpiju19) 198

[Sealed](#_jdanc7mscad2) 199

[Private](#_m0ojvf89ftd2) 200

[Private [this]](#_me58ma3quqp9) 201

[Private [...]](#_qzzi6oadi6kq) 201

[Protected](#_rimxt7ngst7t) 202

[Параметры конструктора](#_2xif77iguvit) 202

[**Параметрический полиморфизм**](#_13jc8nyxqxgo) **204**

[8.1 Обобщенные типы](#_sgm3xchkw6gh) 204

[Ссылки на тип](#_9xgfqo62f2gb) 204

[Абстрактные типы](#_nfnsu3l0wekf) 206

[Несколько параметров](#_dwt56p6wrq7q) 207

[Верхняя граница](#_ww6j476kv1z1) 207

[Нижняя граница](#_rt5st1w8qbjf) 208

[Ссылки](#_nezp0hd1ynrb) 209

[Абстрактные типы](#_3vt7u67n8q2y) 209

[Практика](#_c8a3u46egk7) 210

[Код полностью](#_o9kqc5vfkdvp) 217

[8.2 Вариантность и род](#_j41s2x7j0f5m) 219

[Ковариантность](#_gqmeud8gl2if) 219

[Контрвариантность](#_ze9i6ra6eqma) 220

[Вариантность](#_ayk8ptepmspj) 221

[Род](#_b1xzzfwfx060) 222

[Практика](#_g9bm86xzu32s) 225

[Код полностью](#_qt86abfmx02r) 232

[8.3 Типы: Псевдонимы и компоненты](#_5cu7wovs190q) 234

[Псевдонимы](#_lacx18mut3a0) 234

[Типы-компоненты](#_7r4rsz701mf) 236

[Уточнение](#_ekyhdm8wgmka) 237

[8.4 Неявные параметры](#_uky94zjiz0bl) 239

[Суммируем списки](#_7lbxcvdlpej0) 239

[Выносим за скобки](#_p4vcwq9laegu) 240

[Неявные параметры](#_mszef9q08h6x) 241

[Вывод типов](#_e13auhhzsqx1) 243

[Практика](#_4v7hi53x6i5b) 245

[Код полностью](#_f5bqeafh9qpo) 253

# Введение



Здравствуйте. Я Олег Нижников, программист Тинькофф Банк. Приветствую вас на курсе "Основы программирования на языке Scala".

Мы в Тинькофф очень любим Scala. Много наших программистов постоянно трудятся над мощными распределенными устойчивыми серверными решениями, используя этот язык. И высокое качество наших продуктов объясняется, среди прочего, и нашей любовью к Scala.

Мы выбрали Scala, потому что нашли в ней идеальный баланс красоты, эффективности, продуктивности и прагматизма. Каждое слово многое значит для нас.

Красота важна, потому что красивые вещи мы понимаем и запоминаем лучше всего. Сообщество разработчиков компилятора и библиотек Scala, проделало большую работу, чтобы мы как можно реже загромождали код лишними конструкциями, мешающими воспринимать картину в целом.

Эффективность, обеспечиваемая мощным компилятором, производительностью платформы JVM, современными библиотеками для синхронного и распределенного выполнения, позволяет нам реализовывать сервисы на Scala целиком. Контролировать все нюансы поведения программы не жертвуя масштабируемостью, вычислительными ресурсами и временем пользователя.

Продуктивность следует из выразительности языка и мощной системы типов Scala. Благодаря им, мы можем точно формулировать семантику работы кусочков нашего кода, чтобы быть уверенными, что нигде не ошиблись собирая их вместе. Мы не делаем одну работу дважды, не проверяем в тестах то, что может проверить за нас компилятор. Благодаря этому, наш разум всегда сосредоточен на сути происходящего.

Прагматизм означает, что когда мы ищем способы реализации своей идеи, и нам нужно найти способы взаимодействия с новой базой данных, протоколом, форматом данных, или найти другую библиотеку под наши нужды, часто мы можем найти не одно, а целый набор решений для Scala или Java, и выбрать то, что подходит нам наилучшим образом.

Однако, мы поняли, что ресурсов для изучения Scala на русском языке катастрофически не хватает. И вследствие этого, Scala незаслуженно непопулярна в русскоязычном сообществе. Мы решили создать этот курс, чтобы поделиться нашей любовью к Scala, и таким образом хотим внести лепту в развитие, столь ценной для нас, области.

Когда вы научитесь Scala, перед вами откроется множество возможностей. Вы сможете создавать высоконагруженные распределенные сервисы, обрабатывать петабайты больших данных, с легкостью писать серверную и клиентскую часть веб-приложений, работать с практически каждой известной СУБД, применять технологии на базе блокчейн и многое другое.

Множество компаний захотят вас взять на интересную работу, потому что на Scala делаются обычно самые интересные вещи. И конечно, у вас появится возможность присоединиться к нашей дружной команде.

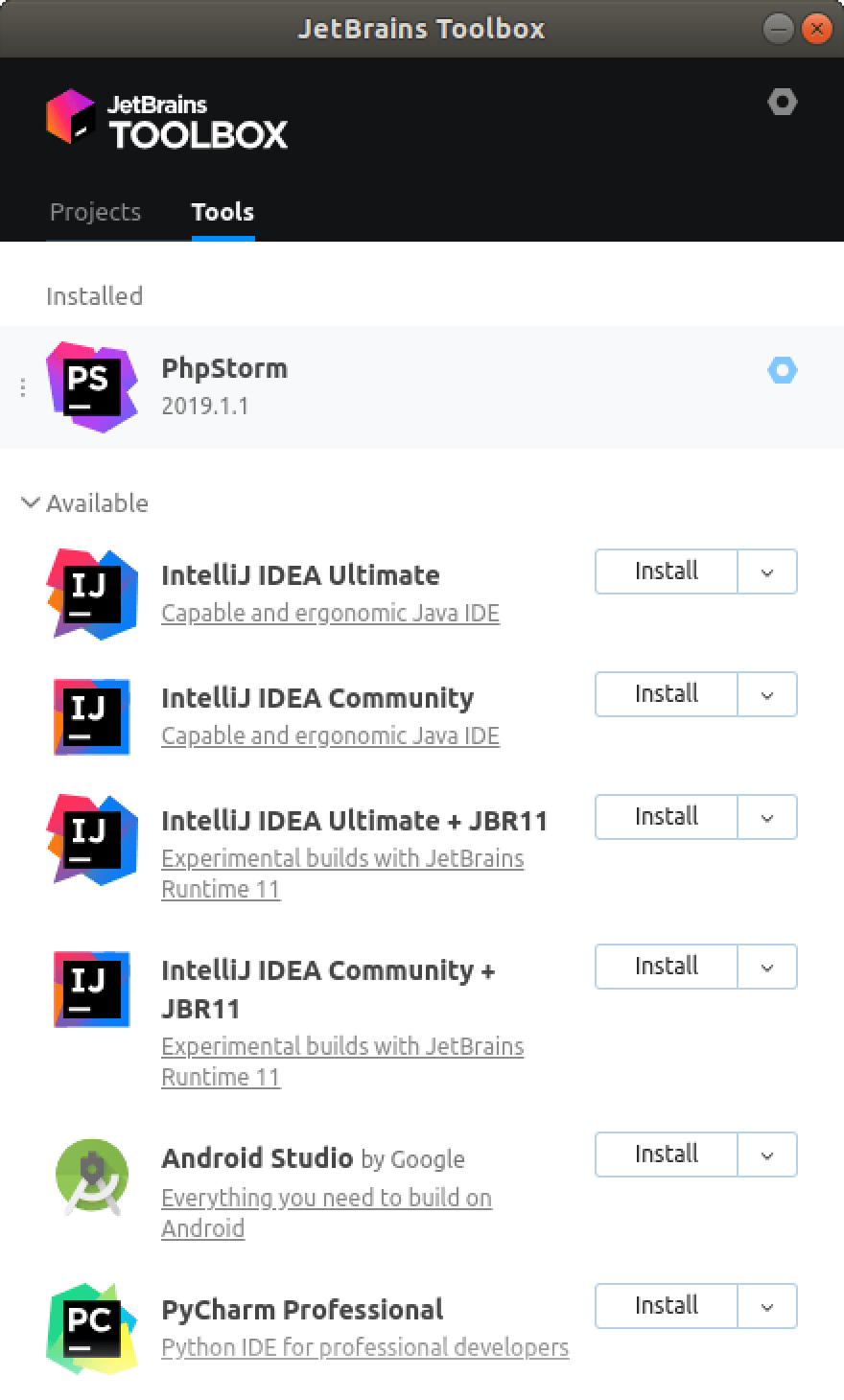
Желаю удачи! Добро пожаловать в мир Scala.

# Начало работы

## **2.1 Установка среды разработки и практика**

Мы с вами готовы написать самую первую программу.

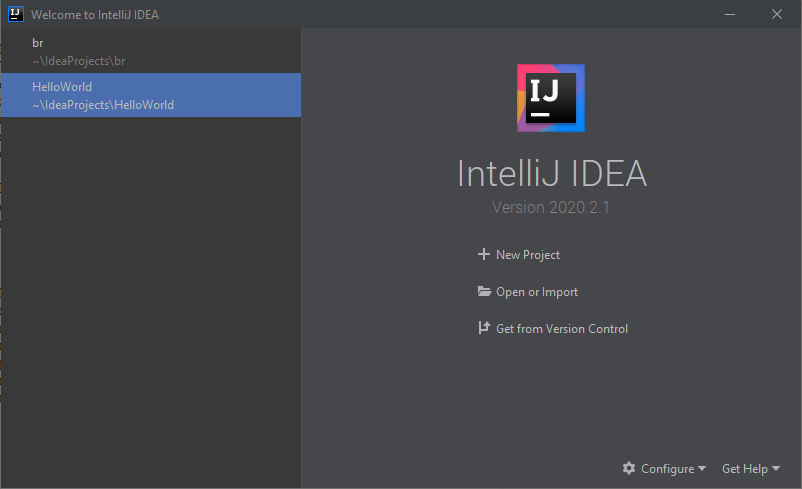
Самый быстрый и практичный способ это сделать, воспользоваться **IDE** (Integrated Development Environment), который будет помогать вам во всем, что связано с написанием программ. Я невольно вынужден стать рекламным агентом компании Jetbrains, которая выпустила, наиболее используемую в мире Scala, IDE под названием **IDEA**.

Для того, чтобы воспользоваться **IDEA**, один из самых удобных способов, не установить ее напрямую, а установить специальное приложение **Jetbrains Toolbox**, которое будет не только устанавливать необходимую версию **IDE**, но и обновлять ее, проверять, когда выходит обновление. Если вы пользуетесь несколькими разными версиями или IDE этой компании, всеми ими вы сможете управлять с помощью приложения Jetbrains Toolbox.

Заходим на сайт [**jetbrains.com**](https://www.jetbrains.com/). Заходим на вкладку **Tools**. Находим пункт **Toolbox App**. Кликаем. Видим большую кнопку **Download**. Нажимаем ее и приложение скачивается. Сохраняем, запускаем, устанавливаем и в нашем трее появляется иконка этого приложения. При нажатии на нее у вас должно выскакивать окошко.

Изначально у вас вверху будет пусто, но вы можете в списке **Available Tools** (доступные инструменты) найти **IntelliJ IDEA Community**. Это совершенно бесплатное приложение, которое я использую каждый день, и выбрать интересующую вас версию. Как правило, я пользуюсь последней версией Early Access Program, то есть версией, в которой я являюсь бета-тестером, сообщающим о проблемах. Именно ее я сейчас запущу.

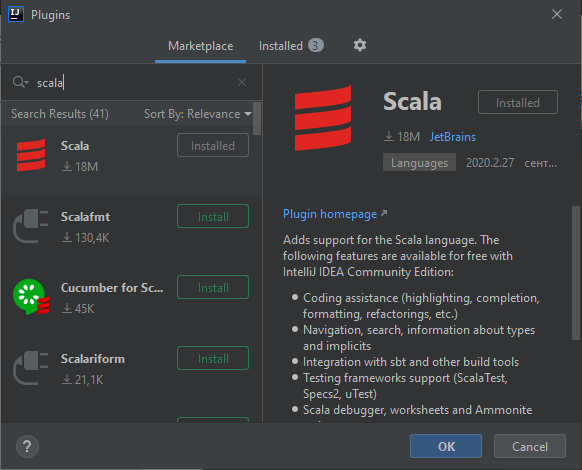
Когда вы запускаете IDEA, откроется ваш предыдущий проект, либо, если еще его нет, откроется стартовое окно. Здесь вы можете выбрать пункт **New Project**.



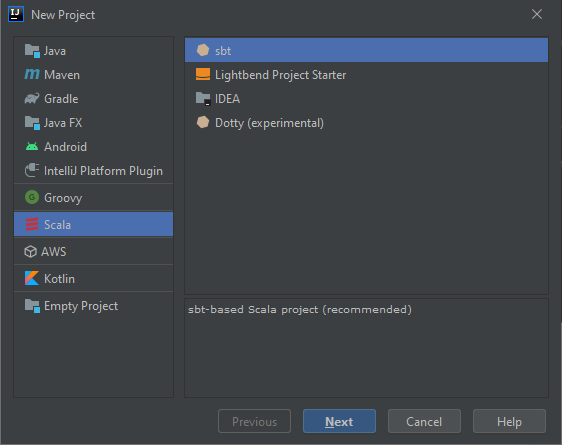
Eсли при установке IDE вы выбрали "установить плагин Scala", вы можете сразу создать новый Scala проект.

Если нет, здесь вы можете нажать **Configure**, перейти в пункт **Plugins**, и установить плагин под названием **Scala**.

Это собственно плагин, который позволит вам разрабатывать на языке Scala и импортировать проекты с использованием Build Tool SBT.



Все это пока не очень важная информация.



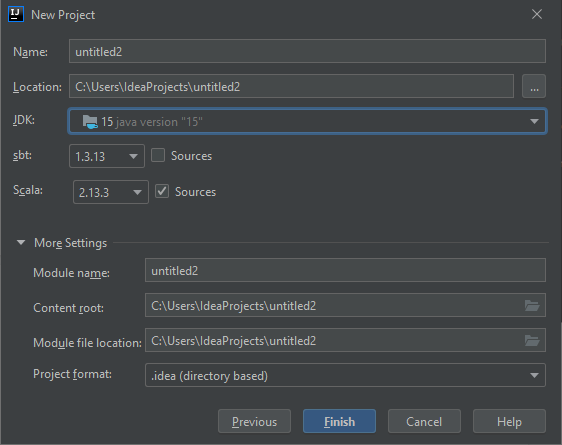
Главное чтобы у вас стоял плагин Scala, и вы могли найти здесь язык Scala, найти здесь SBT, нажать кнопочку **Next**.

Затем IDE предлагает назвать проект и спрашивает, где бы мы хотели, чтобы наш проект стоял.

Вам сейчас нужно будет иметь установленный **JDK** (Java Development Kit), его тоже необходимо скачать. Вы можете воспользоваться [Оpen JDK](https://openjdk.java.net) или скачать с [официального сайта Оracle](https://www.oracle.com/java/technologies/javase-downloads.html).

Мы выбираем Новый (**New**) проект.

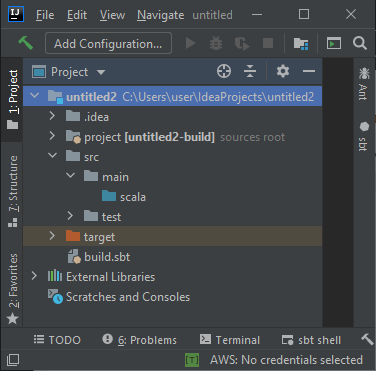
Можем назвать его **Example**. Выбираем версию SBT, выбираем версию Scala. Можно даже ничего не выбирать, все по умолчанию стоит последнее. Нажимаем кнопочку **Finish**.



IDE открывает нам проект. Какое-то время мы должны подождать пока IDEA запустит SBT.

SBT выкачает все необходимые ему файлы, импортирует и мы получим пустой, или точнее, почти пустой проект.

Ждем пока синхронизируется наш стартовый проект. Это не всегда так бывает просто.

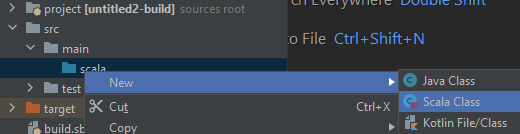


Ну а пока, в исходных файлах, узнаем, где должен лежать код нашей программы.

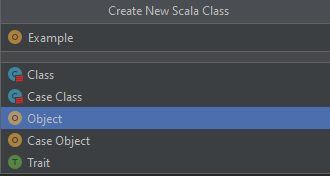
У нас уже есть папка **Example** с нашим проектом. Внутри нее есть папка project, здесь будет лежать очень небольшое количество файлов, связанных с внутренностями вашего проекта. Определение проекта лежит в файле **build.sbt**, если хотите, можете его посмотреть.

Но самое главное будет лежать в папке **src => main => scala**. Это будет вся структура вашего приложения. Каждый файл будет отвечать за набор каких-то определений.

Нам сейчас достаточно создать один простой объект.



Можем назвать его main, можем назвать его example, как угодно. Выбираем здесь тип **Object**.



Видим определение **object Example**

object Example {

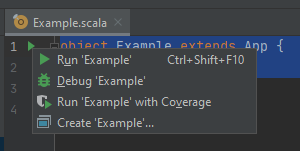
}

Здесь я уже могу начинать работать. Минимум, который мне необходимо сделать, это написать сюда **extends App**.

object Example extends App {

}

Это значить, что автоматически у моего объекта появляется метод **main**, который и будет говорить, что нужно делать, когда программа запускается.



У нас появляется иконка "Запустить" (**Run**). Если я ее сейчас нажму, практически ничего не произойдет, потому что объект **main** сейчас ничего не делает, у **Example** нет никакого содержимого. Можно подождать пока он скомпилируется и увидеть, что ничего не произошло.

Дальше можем написать наш заветный **Hello, World!!!**. Складываем наше сообщение в какую-нибудь переменную и вызываем метод **println()**.

object Example extends App {

val *message* = "Hello, World!!!"

*println*(*message*)

}

Можем нажать на зеленую стрелочку слева, можем нажать **Ctrl+Shift+F10** - это значит запустить текущий объект, и после этого можем его снова запускать - **Ctrl+Shift+F10**.

Можем сделать любые изменения, например, **Hello, Scala!**, убрать несколько восклицательных знаков и нажать **Ctrl+Shift+F10**. Наш код

автоматически перекомпилируется и запускается новый исходный код.

object Example extends App {

val *message* = "Hello, Scala!"

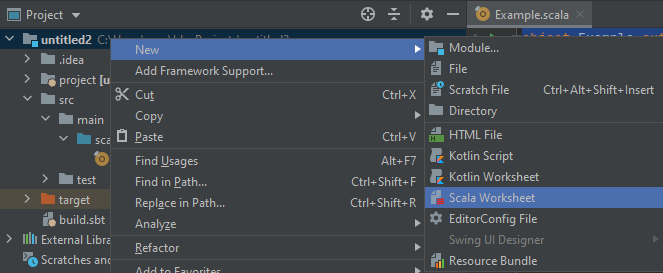
*println*(*message*)

}

Вот, собственно, необходимый минимум для того, чтобы писать код на Scala.

В дереве проекта слева, мы можем создать новый файл, какие-то новые определения, которые нам необходимы.

Можем, в корне проекта, вызвать контекстное меню и выбрать **New => Scala Worksheet** и назвать его **hello**.

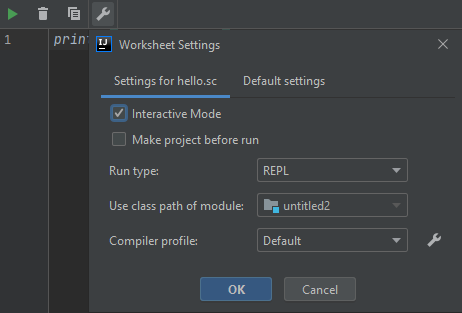


**Worksheet** - это специальный инструмент для упрощения пробных программ в Scala. Этот кусочек кода, каждая его строчка, будет запускаться каждый раз, когда вы меняете исходник. Все его определения будут перезапускаться с самого начала. И о каждом определении, которые вы сделаете внутри этого worksheet, вы узнаете тип этого определения и результат с помощью окна результатов справа.

Я могу написать здесь простой

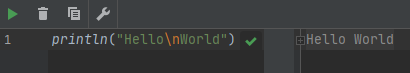
*println*("Hello World")

По умолчанию, для того, чтобы сделать наши **Worksheet** интерактивными, можно нажать просмотр опций и выбрать **Interactive Mode**.

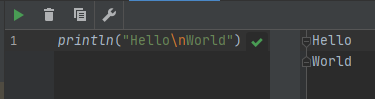


Запускаем. И вот, видим, что мы написали **Hello World**.

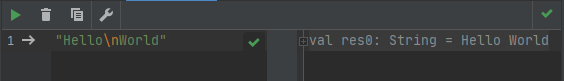
У результатов появляются плюсики, это значит, что здесь есть несколько строчек. Чтобы раскрыть все результаты нужно нажать плюс. Они все относятся к одной строчке.



Если бы мы написали здесь, например, **Hello\nWorld** (**\n** - это специальный символ для разделения строчек), мы бы увидели, что три результата относятся к нашей строчке.



Мало того, мы можем даже убрать **println()**, т.е. оставить только **Hello\nWorld**, и увидеть, что нам даже не нужно ничего выводить на печать, сам результат автоматически отображается в окне справа.



Так мы можем играться, пробовать любые необходимые нам вещи и развиваться в нашем изучении программирования на Scala.

Итак, мы узнали как создавать самую простую первую программу. В следующих уроках мы узнаем про пространство имен.

## **2.2 Приложение**

Вам уже не терпится написать самое простое приложение. Давайте посмотрим, как оно будет выглядеть. Если у вас есть просто консоль или Worksheet, вы можете написать в нем такую простую строчку:

*println*("Hello world!")

Что здесь происходит? Мы создаем строковую константу со строкой 'Hello world!'. Дальше берем метод println и передаем эту строковую константу в этот метод. В результате мы должны увидеть фразу 'Hello world!' в качестве вывода в нашей консоли.

Если вы хотите скомпилировать приложение и запустить его отдельно, вам придется добавить немного дополнительной магии. Вы пишете '**object** Main **extends App**' - это базовый блок, в фигурных скобках, описывайте, строчка за строчкой, все что вам нужно сделать.

object Main extends App {

*println*("Hello world!")

}

В данном случае, вы просто выводите 'Hello world!'. Можете добавить еще одну строчку, сделать несколько действий одновременно.

object Main extends App {

*println*("Hello")

*println*("world!")

}

Вместо того, чтобы писать 'extends App' после 'object Main', вы можете написать такую сложную конструкцию со всеми действиями в этом блоке:

object Main {

def main(args: Array[String]): Unit = {

*println*("Hello World!!!")

}

}

Что это значит, рассмотрим несколькими уроками позже. Сейчас вы должны просто поверить, что это тот самый способ, которым вы можете создать запускаемый код.

По крайней мере, мы знаем, что с помощью функции **println()** мы можем выводить в стандартный вывод.

А как же читать из стандартного вывода?

Для этого, мы можем написать в самом верху одну строчку:

import scala.io.StdIn.readLine

И дальше, у нас волшебным образом появляется функция **readLine()**, которую нужно вызывать без параметров, т.е. просто писать после нее круглые скобки.

Вот пример кода, который мы можем написать с использованием println() и readLine().

import scala.io.StdIn.readLine

object Main {

def main(args: Array[String]): Unit = {

*println*("Hello, enter line")

val line = readLine()

*print*("you entered: ")

*println*(line)

}

}

Сначала вызываем println(), дальше мы говорим readLine(), складываем в строчку, и дальше выводим строчку, которую только что прочитали с консоли. Это базовый набор, который вы можете использовать для того, чтобы строить приложения, которые читают и

пишут из стандартного вывода.

Ну а в следующем разделе, мы поговорим про переменные.

## **2.3 Переменные и их области видимости**

Мы с вами научились писать самые простые программы, но даже простые программы требуют объявления переменных.

### **Значение val**

Переменная - это такое имя, которое обозначает какое-нибудь промежуточное значение. Для того чтобы объявить такое значение, можем воспользоваться ключевым словом **val**.

val x = "value"   
*// Именованное значение, вычисляется сразу*

Пишем: val, имя переменной, равно, значение. На самом деле, когда мы говорим переменная, мы не совсем честны. Если мы объявили переменную с помощью ключевого слова **val**, мы **не можем** потом **менять ее значение**, но мы все равно используем термин переменная.

### **Функция без параметров def**

Альтернативно мы можем воспользоваться ключевым словом **dеf**. Переменные объявленные с помощью **def** будут вычисляться каждый раз, когда вы на них ссылаетесь.

def x = "value"   
*// Метод без параметров, вычисляется каждый раз*

### **Ленивые значения lazy val**

Мы можем выбрать промежуточный вариант написать **lazy val**. Это такая переменная, которая будет **вычислена не больше одного раза**. Она будет вычислена ровно в тот момент, когда вы сошлетесь на нее первый раз.

lazy val x = "value"   
*// Ленивое значение - вычисляется при первом требовании*

### **Переменные var**

Ну и наконец, самая настоящая переменная, вещь которую мы стараемся использовать как можно реже в scala - это ключевое слово **var**. Если мы объявили переменную с помощью var, мы можем в последующем коде присвоить ей новое значение, написав имя переменной равно новое значение.

var x = "value"

x = "new value"

*// Настоящая переменная - может менять значение*

### **Использование**

Переменные при объявлении или при присваивании могут ссылаться на другие переменные. Если мы объявили какие-то переменные с помощью ключевых слов **val** или **var**, мы должны сослаться на них в

последующем коде.

val x = 1

val y = 2

val z = x + y

Если мы воспользовались ключевыми словами **def** или **lazy val**, мы можем сослаться на них даже в коде, который идет перед ними, ну если конечно они находятся в одном месте и переменная верху может увидеть переменные снизу.

### **Указание типа**

Каждое выражение в Scala имеет тип и у переменных тоже есть типы. Мы не указываем их потому, что компилятор Scala может вывести сам эти типы за нас. Но иногда полезно, а иногда даже нужно, явно объявить тип переменной. Для того чтобы сделать это, сразу после имени переменной можем поставить двоеточие и указать тип. О типах мы поговорим позже.

val x: Int = 1

val y: String = "value"

val z: Double = x

*// Можно и часто нужно указывать тип*

### **Области видимости**

У переменных есть области видимости. Это такая часть программы, в течение которой имя переменной видно. Мало того, области видимости могут входить в другие области видимости и даже частично затирать их.

val x = "Outer"

{

val x = "Inner"

*println*(x)

}

Вот, например, мы объявили **переменную x**, внутри мы объявили новую **область видимости** или, иногда говорят по-английски, **scope**, в которой мы объявим ещё одну переменную с тем же самым именем. Функция **println()**, если мы передадим ей **x**, возьмет внутреннюю переменную, потому что она более приоритетная своем scope.

Если мы объявили переменную внутри какого-то scope, внутри какой-то области видимости, за пределами этой области видимости переменная **перестает быть видна**. Поэтому последняя строчка, где мы вызываем **println(x),** выдаст **ошибку** компиляции.

{

val x = "Inner"

*println*(x)

}

*println*(x) *//Ошибка!!!*

Мы с вами узнали много про переменные, давайте попробуем их на практике.

## **2.4 Пространства имен**

На практике мы с вами познакомились с переменными, и будет теперь полезно познакомиться с таким понятием, как пространство имен.

Когда мы написали программу достаточно большого размера, мы скорее всего объявим в ней большое количество переменных, функций, каких-то классов и других типов. Имена всех этих вещей скорее всего начнут пересекаться.

Для того, чтобы избежать этого, и для того, чтобы не писать имена очень очень длинные, в которые мы будем вкладывать всю логическую составляющую того, что представляет у нас данный элемент, мы можем воспользоваться пространствами имен.

### **Пакеты**

Самым простым пространством имен является пакет. Имя пакета пишется в самом верху файла. В самой верхней строчке мы пишем **package** и дальше, разделенное точками, полное **имя пакета**.

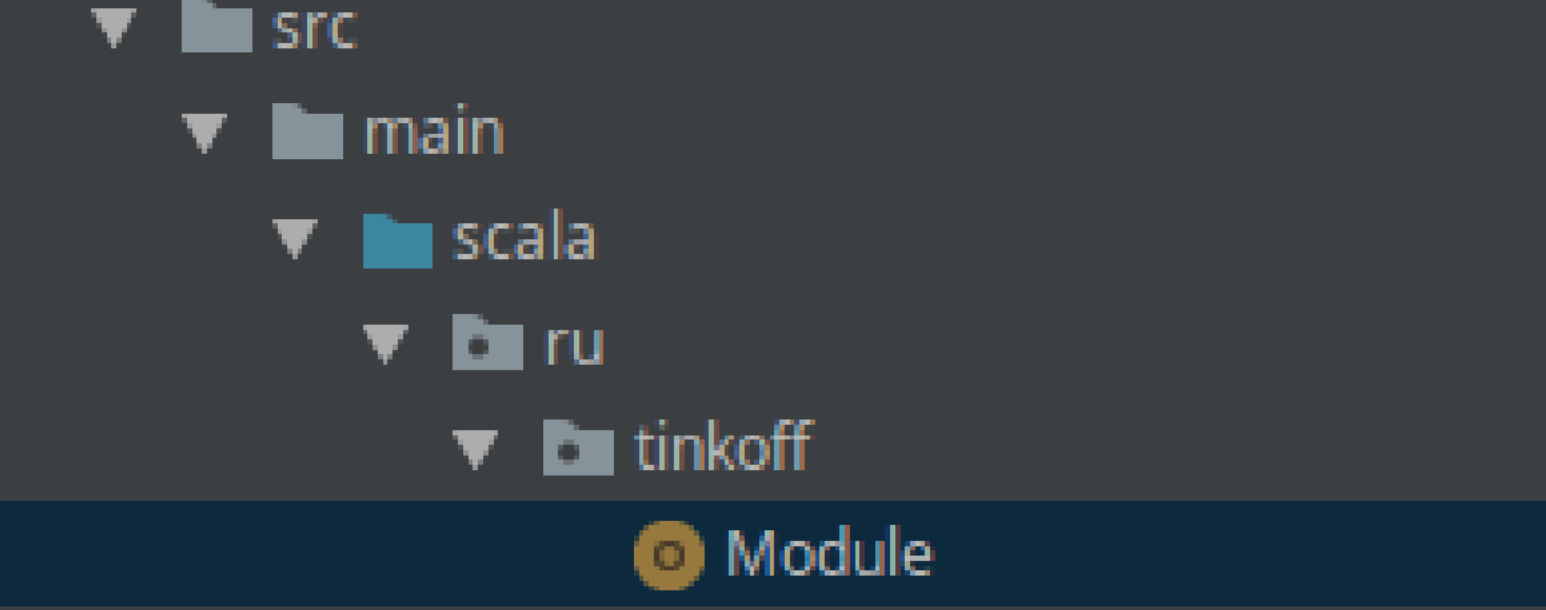
package ru.tinkoff

object Module{

def name = "Oleg"

}

Если мы хотим объявить пакет, компилятор Scala очень попросит нас положить файл, который относится к этому пакету в директорию, соответствующую его имени.



package ru.tinkoff

Т.е. если мы объявили пакет с именем ru.tinkoff, компилятор хочет, чтобы в папке с основными нашими исходниками, в данном случае это **src => main => scala**, была поддиректория **ru**, в ней была поддиректория **tinkoff**, и внутри лежал, собственно, наш файл. Тогда все прочие объявления в этом файле автоматически попадут в этот пакет и дальше мы сможем их использовать.

### **Импорт**

Как мы можем использовать элементы из других пространств имен?

Для этого у нас есть ключевое слово **import**. Мы пишем import и дальше указываем все кусочки, все составные части, весь путь, до нашего пространства имен, которое мы хотим использовать.

import ru.tinkoff.Module

object Main{

val *myName* = Module.name

}

В данном случае, мы импортировали только что объявленный объект import ru.tinkoff.Module, который объявили в предыдущем файле. После этого Module становится именем, которое мы можем использовать в текущем файле, вот так, например, присвоив какой-то переменной.

Но самое интересное, что некоторые вещи, например, **объекты** и другие **переменные**, сами тоже являются пространствами имен, т.е. не только пакеты. Мы можем импортировать имена даже из них.

import ru.tinkoff.Module.name

object Main{

val *myName* = name

}

В данном случае мы можем импортировать переменную **name**, которую мы объявили внутри нашего объекта непосредственно и сразу ей пользоваться.

### **Идентификаторы**

Для того, чтобы импортировать вещи, мы должны понимать какие идентификаторы являются стабильными. Стабильный идентификатор - это что-то, что указывает на значение, которое точно не будет меняться.

**Стабильными идентификаторами** являются: **package**, любой **параметр** вашей функции, **val**, как неизменное значение, **lazy val**, как неизменное значение, но возможно еще не вычисленное, и **object**.

Как пример **нестабильного идентификатора** можем представить **def**, некая вещь, которая будет вычисляться каждый раз при использовании, и возможно, в разных случаях она вернет разные значения.

### **Множественный импорт**

Представим, что мы объявили в нашем модуле несколько имен.

package ru.tinkoff

object Module{

def name = "Oleg"

def name2 = "Kate"

}

Можем ли мы не писать по строчке для каждого из объявленных имен, чтобы импортировать его?

Оказывается можем. Если мы хотим импортировать из одного пространства имен сразу **несколько имен**, мы можем перед именами поставить точку и в фигурных скобках перечислить все имена, которые мы импортируем.

import ru.tinkoff.Module.{name, name2}

*// Можем импортировать несколько имен из пространства*

Мало того, можем даже импортировать **всё содержимое** какого-нибудь пространства имен, поставив после точки **подчёркивание**.

import ru.tinkoff.Module.\_

*// Можем импортировать все имена из пространства*

Если мы хотим **переименовать** какой-то объект при импорте, например, из одного пространства имен мы хотим элемент, из другого элемент с точно таким же именем, и хотим использовать их одновременно, тогда мы можем один из них переименовать. Для этого мы после имени ставим **стрелочку =>** и пишем новое имя в фигурных скобочках.

import ru.tinkoff.Module.{name => olegName}

*// Можем переименовать при импорте*

Если мы хотим сделать обратное, т.е. хотим импортировать из модули всё, но **не импортировать** какие-то из объявленных в нем имен, мы после подчёркивания в фигурных скобочках можем воспользоваться таким вот синтаксисом, пишем имя которое, мы хотим выбросить при импорте, стрелочка => и еще одно подчёркивание, тогда name2 не будет импортировано, его не будет в нашем scope после того как мы написали эту строчку.

import ru.tinkoff.Module.{\_, name2 => \_}

*// Можем импортировать, не импортируя отдельные имена*

Мы с вами полностью познакомились с пространствами имен и в следующем разделе мы познакомимся с типами.

# 

# Типы

## **3.1 Типы**

Мы с вами поговорили про пространства имен и уже неоднократно упоминали, что у наших значений переменных, бывают типы. Давайте узнаем несколько интересных вещей про них.

### **Подтипы**

В Scala есть не только типы, но и отношения между типами, отношении подтипизации.

Мы говорим, что тип A является подтипом B, когда любое значение, в том числе переменную, типа A можно подставить в любое место, где мы хотели бы использовать значение типа B.

| **A <: B**  A является подтипом B, когда все значения типа A  могут использоваться как значения типа B |
| --- |

Мало того, у нас есть некие очень очень общие надтипы, например, **Any** - это надтип всех типов. Иными словами, любой тип является подтипом Any.

Везде, где мы хотим видеть переменную типа Any, мы можем поставить вместо нее переменную типа A, т.е. любого другого типа.

| **A <: Any**  Все типы является подтипом Any — тип всех значений |
| --- |

Таким образом, если вы хотите объявить функцию, которая принимает параметры какие угодно, вы можете написать Any.

У нас есть ссылочные и примитивные типы. Мы разделяем их

с помощью двух базовых надтипов.

**AnyRef** - это базовый надтип всех ссылочных типов, в буквальном смысле типов, значения которых являются ссылками.

| **A <: AnyRef**  Любой ссылочный тип является подтипом AnyRef —  тип всех ссылочных значений |
| --- |

Все значения примитивных типов являются подтипами **AnyVal**. О примитивных типах мы сейчас скажем.

| **A <: AnyVal**  Любой примитивный тип является подтипом AnyVal —  тип всех примитивных значений |
| --- |

И у нас есть особый тип, который магическим образом является подтипом любого типа A. Это довольно странно звучит.

Если мы хотим абсолютно какой-то неизвестный тип A, мы вместо него можем поставить значение типа **Nothing**.

| **Nothing <: A**  Любой тип является надтипом Nothing —  тип, который используется для ошибок |
| --- |

Как так получается? На самом деле у нас нет ни одного экземпляра типа Nothing, вместо этого, практически всё, что генерирует Nothing - это какие-то ошибки, исключения.

Если мы хотим видеть в каком-то месте нормальное значение и подставляем вместо него Nothing, скорее всего вычисление здесь и закончится.

### **Примитивные типы**

Теперь о примитивных типах. У нас есть 9 базовых примитивных типов.

| Целые Числа | Byte, Short, Int, Long |
| --- | --- |
| Дробные числа | Float, Double |
| Символы | Сhar |
| Булевые значения | Boolean |
| Единичный тип | Unit |

Есть 4 типа для **целых чисел**: **Byte** - это 8-битные числа 1-байтные; **Short** - 16-битные 2-байтные; **Int** - 32-битные 4-байтные числа и **Long** - 64-битные 8-байтные.

Диапазон значений у них колеблется, как соответствующий степени двойки. **Byte** колеблется от минус 2х в 7й степени до 2х в 7й степени минус 1.

Как так получилось интересно? В Scala, как и в Java, в этих примитивных типах знак минус обозначается первым битом числа, поэтому получилось, что 0 мы понимаем как всегда положительное число, и таким образом у нас получилось ровно половина всех чисел байт: то есть 2 в 7й степени отрицательные, и ровно половина положительные, соответственно.

Для **Shot** то же самое: диапазон от -2 в 15й до 2 в 15й минус 1.  
Для **Int**: от -2 в 31й до 2 в 31й минус 1.

И для **Long**: от -2 в 63й до 2 в 63й минус 1.

У нас есть специальные **числа с плавающей точкой**: **Float** - 4-байтное число с плавающей точкой и **Double** - 8-байтное число с плавающей точкой. Их точность, соответствующим образом, зависит от их длины.

Если вы хотите достаточно точные числа, скорее всего нужно пользоваться Double, а если вы хотите сократить потребление памяти, или ускорить вашу программу, но вам не так важна точность, вы можете пользоваться Float.

У нас есть специальный 2-байтный, как правило, тип **Char**. Это тип означающий **простые символы**: буквы, цифры, весь практически Юникод.

Есть специальные булевые значения (**Boolean**), которые принимают значения только **true** и **false**.

И есть отдельный специальный тип под названием **Unit**, который вообще принимает ровно одно значение. Как правило, оно обозначается круглыми или фигурными скобками.

Значение типа Unit нам, как правило, не нужно знать. Оно нам не интересно, оно всегда одинаковое, поэтому используем тип Unit для различных мест, где, например, возвращаемый тип какой-то процедуры нам не интересен, она просто что-нибудь делает и **ничего не возвращает**. Тогда мы говорим, что она **возвращает Unit**.

### **Как работают подтипы**

val x: String = "Oleg"

val y: AnyRef = x

val z: Any = y

Можем объявить переменную типа **String**, присвоить какое-нибудь значение. Дальше можем объявить переменную типа и **AnyRef**. Напомним, String является ссылочным типом, поэтому переменная типа **X** у нас подходит, как значение, для переменной типа **Y**. Мы, таким образом, спустили вниз, ослабили наши требования по типам, когда преобразовали X в AnyRef. И дальше можем еще спуститься ниже, еще более ослабить наши требования по типам, и превратить нашу строчку в тип **Any**.

То же самое можем сделать с каким-нибудь примитивным типом. Типом, значение которого, являются просто значениями в байтах, а не ссылками назначения.

val x: Int = 1

val y: AnyVal = x

val z: Any = y

Например, можем объявить переменную типа **Int**. Дальше мы можем обобщить ее до типа **AnyVal**, и дальше, опять же, до типа **Any**.

Вот, наверное, и вся базовая информация, которую мы хотели узнать про подтипы. Давайте попробуем их на практике.

### **Практика**

Мы с вами узнали, что такое типы и хотелось бы попробовать, как они будут выглядеть в реальной программе. Начнем с нашей программы, которую мы с вами написали ранее.

object Example extends App {

val *message* = "Hello, Scala!"

*println*(*message*)

}

Hello, Scala!

У нас есть какая-то переменная и println(message). Давайте попробуем дописать какие-нибудь типы. Например, дописать тип к переменной message. Очевидно по-умолчанию должен быть String. Если запустим нашу программу, получим то же самое.

object Example extends App {

val *message*: **String** = "Hello, Scala!"

*println*(*message*)

}

Hello, Scala!

И как мы уже говорили, мы можем можем упростить этот тип, привести его более высокостоящему в иерархии типов типу, например, AnyRef.

object Example extends App {

val *message*: **AnyRef** = "Hello, Scala!"

*println*(*message*)

}

Hello, Scala!

Запустив нашу программу, видим то же самое, значение переменной не изменилось, изменилось только лишь знание программы, о том, какого эта переменная типа.

Например, если мы здесь напишем String, и ниже напишем еще раз String, равно предыдущей переменной, и будем выводить на печать уже вторую переменную, наша программа все еще будет работать, она будет компилироваться. Для того, чтобы скомпилировать программу без попытки ее запустить, можно нажать в нашей IDE Ctrl+F9.

object Example extends App {

val *message*: **String** = "Hello, Scala!"

val *message2*: **String** = *message*

*println*(*message2*)

}

Hello, Scala!

Для того, чтобы скомпилировать программу без попытки ее запустить, можно нажать в нашей IDE Ctrl+F9.

Теперь попробуем изменить наш тип на какой-нибудь более общий. И так - AnyRef.

object Example extends App {

val *message*: **AnyRef** = "Hello, Scala!"

val *message2*: **String** = message

*println*(*message2*)

}

type mismatch;

found : AnyRef

required: String

val message2: String = message

Мы сразу видим, что наша IDE подсказывает, что переменные неправильного типа, и если мы попробуем скомпилировать, мы видим ту же самую ошибку.

Мы хотели String, а получили AnyRef. Несмотря на то, что на самом деле, в этой переменной лежит строчка, программа уже забыла, что в ней лежит строчка, и программа перестала компилироваться. Поэтому, когда мы храним что-то, зачастую нам выгодно сообщать об

этом много подробностей, но не слишком много. Для того, чтобы пользователь не узнал какие-то слишком глубокие подробности о реализации, нам нужно сообщить максимально точный тип переменной, который позволит пользоваться всем тем, что эта примерная предоставляет.

Можем попробовать какие-нибудь другие типы, например, запустить какую-нибудь функцию. У нас есть объект Math, он встроен в Java. Есть множество встроенных функций в объекте Math. Большинство из них работают с числами с плавающей точкой, например, функция арккосинус. Можем вычислить значение арккосинуса и положить его в переменную message2. Наша программа выдает результат.

object Example extends App {

val *message*: AnyRef = "Hello, Scala!"

val *message2* = **Math.*acos*(1.0)**

*println*(*message2*)

}

0.0

Если мы попробуем вычислить арккосинус от нашей строчки, программа даже не скомпилируется.

object Example extends App {

val *message*: AnyRef = "Hello, Scala!"

val *message2* = **Math.*acos*(message)**

*println*(*message2*)

}

type mismatch;

found : AnyRef

required: Double

val message2 = Math.acos(message)

Таким образом, если мы передаем переменную в функцию, которая ждет переменную другого типа, мы сразу узнаем, что эта вещь сюда не подходит.

Это основная причина зачем мы вообще используем типы в Scala, для того, чтобы мы точно знали, что кусочки нашей программы хорошо подходят друг другу. И мы стараемся как можно больше придумать типов, так чтобы даже очень похожие друг на друга вещи, но которые на самом деле являются разными, никогда не использовались в одном месте.

Например, мы можем только для типизации обернуть строчку внутрь какого-нибудь класса, только для того, чтобы знать, что вот эта строчка не просто строчка, а какой-нибудь идентификатор, или чье-нибудь имя, и не использовать ее в том месте, где мы ждем какую-нибудь другую строчку, которая имеет другую семантику.

Именно поэтому мы хотим, чтобы все наши ошибки, которые потенциально мы могли допустить, выяснялись до того, как мы запустим программу, до того, как мы запустим любой unit-тест. Именно для того, чтобы как можно меньше писать unit-тестов, но в то же время иметь ту же степень корректности нашей программы.

Что мы еще можем сделать с этим кодом?

Мы можем, вспомнить, что наша переменная message это все-таки строчка и приравнять её к типу Any. Как мы помним это работает. Мало того, если мы возьмем какой-нибудь message3 типа Any и приравняем ему арккосинус от единицы и выведем результат на экран.

object Example extends App {

val *message*: String = "Hello, Scala!"

val *message2*: **Any** = *message*

val *message3*: **Any** = Math.*acos*(1.0)

*println*(*message2*)

*println*(*message3*)

}

Hello, Scala!

0.0

Мы видим, что у нас две переменные, которые имеют совершенно разные значения, но имеют один тип. Это то, что мы хотели бы, чтобы происходило как можно реже.

Мало того, есть помимо Any еще один тип, к которому мы можем привести практически все что угодно - это тип Unit.

Unit - это специальный тип, который не хранит никакой информации о результате. Если Any, по крайней мере, означает, что у нас есть результат, то Unit означает, что нам результат совершенно не интересен. И точно так же можем приравнять к типу Unit переменную любого другого типа. Как видите, нет никакой ошибки. Однако, если мы попробуем вывести результат на экран, мы увидим, что мы потеряли результат, потому что, на самом деле, он был выброшен.

object Example extends App {

val *message*: String = "Hello, Scala!"

val *message2*: **Unit** = *message*

val *message3*: **Unit** = Math.*acos*(1.0)

*println*(*message2*)

*println*(*message3*)

}

()

()

У нас есть только один объект типа Unit и обозначается он вот так () круглые скобочки, или то же самое {} фигурные скобочки. Поэтому, например, мы можем приравнять к типу Unit даже результат, например, функции println() и вывести его на экран. Мы опять, по сути, не получим никакой информации, но мы точно теперь знаем, что **Unit - этот тип, который означает любую последовательность действий**.

Мало того, если у нас есть несколько выражений, которые что-то делают, есть блок, который обозначает некое внутреннее именное пространство, в нем объявляем какую-нибудь переменную, значение, **результат всего этого блока, его тип, будет равен типу последнего выражения в этом блоке**. В данном случае, это Double.

Вот основные вещи, которые мы можем делать с типами, основные практические знания о них. А в следующем разделе мы поговорим про числа.

## **3.2 Числа**

Мы с вами познакомились с типами на практике. Давайте поговорим про числа. Числа нам всем знакомы, и как мы уже упоминали в Scala они бывают целыми и числами с плавающей точкой. Для того чтобы просто объявить число, мы можем записать просто несколько цифр подряд, тогда по умолчанию мы объявляем переменную типа **Int**.

val a = 3 *// Int*

Мы можем добавить в самом конце большую букву **L** - это значит, что мы можем написать еще больше цифр, и тогда мы объявим переменную типа **Long**. Ну и конечно можем поставить знак минус перед объявлением переменной, тогда мы объявим отрицательную переменную.

val b = -5L *// Long*

Если мы хотим объявить переменную с плавающей точкой, нам достаточно поставить точку в ее значение, например, вот так 10.3 или даже 10.0, тогда это перемена по умолчанию будет переменной типа **Double**.

val c = 10.3 *// Double*

Если мы хотим объявить переменную типа **Float**, мы можем указать тип переменной, добавив в конце букву **f**.

val d = 10.3f *// Float*

Для значений с плавающей точкой у нас еще существуют специальная, так называемая, **научная форма** записи или форма записи с мантиссой. Для того чтобы воспользоваться ей, мы пишем некую **мантиссу**, пишем буковку **Е**, и дальше пишем **показатель степени десятки** в этой записи числа.

val e = 1.03e1 *// научная форма*

### **Числовые операции**

Над числами можно делать много разных операций. Например, **базовые операции** связанные с числом, как с числом, это: сумма, разность, умножение, деление, остаток по модулю, если мы говорим о целых числах, и унарный минус, то есть нахождение противоположного числа.

val x = -5

val y = 3

x + y *// Сложение*

x - y *// Вычитание*

x \* y *// Умножение*

x / y *// Деление (нацело)*

x % y *// Остаток от деления*

-x *// Противоположное число*

### **Бинарные операции**

Помимо того, что мы можем представлять числа, как числа, мы можем представлять числа, как наборы бит, такие длинные битовые вектора. Для этого нам бывает полезна запись чисел в шестнадцатеричной форме, т.е. состоящий не из цифр от 0 до 9, а из цифр от 0 до 9 и затем от A до F.

Для того, чтобы задать число в шестнадцатеричной форме, мы можем написать **0x** и дальше его **шестнадцатеричные цифры**.

С числами в побитовой форме, у нас есть набор интересных битовых операций.

val x = 0xF

val y = 0xA1

x >> y *// побитовый сдвиг вправо*

x << y *// побитовый сдвиг влево*

x | y *// побитовое ИЛИ*

x & y *// побитовое И*

x ^ y *// побитовое искл. ИЛИ*

~x *// побитовая инверсия*

Например, **побитовый сдвиг вправо**, обозначается двойным больше **>>**. Побитовый сдвиг вправо означает, что мы берем все биты числа, первый бит ставим в ноль, а остальные биты, например, тридцать один бит, если мы говорили про Int, мы сдвигаем ровно на один бит вправо, или на два бита вправо, или на 3 бита вправо, сдвигаем на столько бит вправо, чему у нас равно число Y.

То же самое **побитовый сдвиг влево <<**. Мы берем сколько-то бит числа X и сдвигаем их влево ровно настолько, чему у нас равно число Y.

Дальше есть **побитовое ИЛИ |** - это значит, что мы берем, рассматриваем уже и X и Y, как два битовых вектора, и рассматриваем соответствующие биты. Для каждого бита мы устанавливаем ноль, тогда и только тогда, когда оба бита в этой позиции у X и y Y равны нулю, и единицу в другом случае.

То же самое, есть похожая операция **побитовое И &**, рассматриваем X и Y как битовые виктора, пробегаемся, и для каждой позиции, устанавливаем единицу тогда и только тогда, когда у нас и X, и Y, в данной позиции имеют бит единица.

И **побитовое исключающее ИЛИ ^**, или побитовая сумма с остатком 2, означает, что мы пробегаемся по всем битам, и если эти биты равны, то результирующий бит у нас будет равен нулю, а если не равны то единица.

И операция, похожая на противоположное число - это **волна ~, побитовая инверсия**, пробегаемся по числу, и в каждой позиции, если был 0, ставим единицу, если единица - 0.

Применяются эти операции в некотором коде, который пытается быть очень быстрым. Если вам нужны какие-то очень хитрые ускорения, иногда вы можете достичь их, представляя числа как битовые вектора.

### **Порядок операций**

У всех операций есть некий встроенный порядок. Вы можете явно задавать порядок расставляя скобки в вашем выражении. Это, как и во многих языках, естественная операция, сначала будет выполняться то, что в скобках, а затем то, что без скобок.

Вы также можете предполагать, что некоторые операции имеют больший приоритет, чем другие.

val x = -5

val y = 3

(x \* y) + (2 \* 3)

x \* (y + 2) \* 3

x \* y + 2 \* 3

*// Можем определять порядок выполнения с помощью скобок, изменяя приоритет*

Например, выражение X \* Y + 2 \* 3, так же, как и в естественной речи, мы предполагаем, что произведение будет выполняться с более высоким приоритетом, чем сложение. Т.е. сначала мы умножим X и Y, затем мы умножим 2 и 3, и потом результаты этих двух произведений сложим.

Если быть точным, вот вся таблица порядка всех операций, которые только есть в Scala.

| 1. **символы**  2. **|**  3. **^**  4. **&**  5. **= !**  6. **< >**  7. **:**  8. **+ -**  9. **\* / %**  10. **все остальные знаки** |
| --- |

Самый низкий приоритет имеют символьные операции, т.е. любые операции, методы, чье имя состоит просто из символов. Дальше идет

оператор черта вертикальная |, ну или две черты и вертикальные ||. Дальше идут операторы, которые начинаются с крышечки ^. Дальше, с более высоким приоритетом, т.е. будут выполняться раньше, чем предыдущие, идут операторы, которые начинаются амперсенда &. Затем операторы, которые начинаются с равенства = или с восклицательного знака !. Дальше меньше или больше < >. Затем приоритет у двоеточия :. Затем приоритет у плюса и минуса + -. Затем почти, что самый высокий приоритет в этом списке, это операции умножить \*, разделить / и процент %. И дальше все остальные символы.

Нужно здесь упомянуть, что в Scala, на самом деле, есть не только встроенные операторы, которые мы привыкли с вами видеть. Потом мы с вами поговорим, что вы сами можете у своих типов определить какие-нибудь символьные и числовые операторы, и порядок их будет определяться на основании первого символа.

### **Большие числа**

Помимо встроенных примитивных типов данных, есть пара специальных типов в Scala для того, чтобы обозначать **очень большие числа**, это **ссылочные типы**. Как правило, мы работаем с типами BigInt и BigDecimal.

val x = *BigInt*(10)

val y = *BigDecimal*(10)

val z = *BigInt*("100000000000000000000000000000")

*// Есть ссылочные типы больших чисел*

**BigInt** - это версия Int и Long, только содержащая неограниченное число цифр. **BigDecimal** - это почти то же самое, только она может содержать какое-то количество цифр после десятичной точки, точно так же она может содержать очень большое количество цифр перед точкой и после нее. Можно инициализировать передавая им в качестве параметра простое число или строчку с репрезентацией.

У таких чисел помимо упомянутых операций, есть дополнительные операции.Например, у **BigInt** есть операция **pow** возведение в степень, вот, например написать X pow 100, где X равен BigInt от 10, мы можем найти 10 в сотой степени. Также есть, например, операция вычисления наибольшего общего делителя двух больших целых чисел.

val x = *BigInt*(10)

x pow 100

val y = *BigInt*(10)

x gcd y

*// Операции над ними не дают переполнения*

Мы с вами изучили числа. Давайте попробуем сделать с ними что-нибудь на практике.

### **Практика**

Мы с вами познакомились с числами. Давайте посмотрим, на что они способны. У нас есть целый набор чисел. Например, вот число 123, как видим это целое число. Мы можем автоматически трансформировать его в Long.

123L

val res0: Long = 123

Можем вычислять их суммы.

123 \* 23

val res1: Int = 2829

И если нам потребуется, например, взаимодействие с байтами, мы всегда можем сказать, что какой-нибудь литерал, у нас относится к типу Byte, или к типу Short.

(1: Short)

val res2: Short = 1

Но нужно быть осторожными потому, что если мы будем выполнять любую операцию с такими небольшими целочисленными значениями, они автоматически конвертируются в Int.

(1: Short) + (2: Short)

val res2: Int = 3

И для того, чтобы их вернуть в Shot нам потребуется вызвать метод **toShort**.

((1: Short) + (2: Short)).toShort

val res2: Short = 3

Мало того, у нас есть еще символы, которые состоят из строчек, например, вот символ A. У каждого символа можно взять его численное значение кодировки с помощью метода **toInt**.

'A'.toInt

val res3: Int = 65

И у каждого числового значения можно снова взять его соответствующий символ с помощью метода **toChar**. Вот так мы нашли следующий символ после A, или, например, через десять символов после А.

('A'.toInt + 1).toChar

val res3: Char = B

Можно даже искать разницу между символами, это всегда будет какое-то целое число, которым мы можем оперировать.

'K' - 'A'

val res4: Int = 10

Если числа, с которым мы оперируем слишком большие, у нас неизбежно произойдет **ошибка переполнения**. Добавим цифр побольше, и как видите, внезапно, сумма двух положительных чисел

становится отрицательной.

1231212333 + 1231231231

val res5: Int = -1832523732

Конечно же это неправильное значение. В таком случае, можем превратить их в Long, но даже если мы говорим про Long, даже у него есть свой предел, нужно его аккуратно прощупать, как видите, и с Long мы можем не избежать переполнения.

3231212333999999999L + 7231231231999999999L

val res5: Long = -7984300507709551618

Вот специальная форма чисел, у которой практически нет таких границ - это **большие числа**. Можем сконструировать, как видите, из строчек, какие-то рандомные числа. Можем сконструировать их из массива байт, из Long, из Int, из строчки, и можем задать ей автоматические системы счисления, в которых мы считаем, и собственно для них у нас есть все необходимые операции.

Мало того, как видите, если мы делаем какие-нибудь операции с **BigInt**, и число, которое мы используем в качестве аргумента для этой операции, если одно из чисел уже заведено в BigInt, оно автоматически трансформируется в другой BigInt.

*BigInt*(3231212333999999999L) + 7231231231999999999L

val res5: scala.math.BigInt = 10462443565999999998

BigInt вообще очень полезны, помимо того что они большие, у них есть множество замечательных операций. Например, у них есть операция наибольшего общего делителя.

val a = *BigInt*(3231212333999999999L) + 7231231231999999999L

a.**gcd(123)**

val res5: scala.math.BigInt = 3

Еще, мы можем взять любое число и вывести его в нужной в качестве цифр, используется первые 10 цифр + первые 15 букв латинского алфавита. И максимум, с которым можем так пойти, это 36-ричные системы исчислений, после этого мы снова будем получать просто десятичную запись, такие не валидные радиксы.

val a = *BigInt*(3231212333999999999L) + 7231231231999999999L

a.**toString(32)**

val res6: String = 92cggvml00ivu

Что еще можем делать с числами? Мы можем попробовать сконвертировать любую строчку в число. Введя любую строчку, можем написать **toInt** и получить результат, целочисленное значение.

"121123213".toInt \* 2

val res7: Int = 242246426

Можем здесь где-нибудь ошибиться, например, написать буквку a, и в качестве результата, мы получим вот такую вот ошибку: **NumberFormatException**.

**"a121123213".toInt \* 2**

java.lang.NumberFormatException: For input string: "a121123213"

В дальнейшем, мы, возможно, узнаем о таком типе как Try, поговорим о Try-catch и обработке исключений, но пока что, вы должны знать, что ваши строчки должны быть очень хорошо подобраны, для того чтобы таких ошибок не возникало.

Практически у всех чисел есть все операции в плане деления, вычисления с остатком.

Есть числа с плавающей точкой, их точно также можно складывать. Если вы пытаясь сложить с целым числом, оно автоматически превращается в число с плавающей точкой.

val x = 1.0

x + 3

val res8: Double = 4.0

И если вы попробуете сложить с BigInt, то, результат у вас непредсказуем. Ни в ту, ни в другую сторону, Scala не может определить, что ж ей выбрать: отбросить очень много точной информации о числе, чтобы превратить его в Double, или придумать какую-то информа какие-то цифры о числе Double, чтобы представить, что как будто бы был BigInt. Чтобы не гадать, Scala просто так не делает.

val a = *BigInt*(3231212333999999999L) + 7231231231999999999L

val x = 1.0

a + x

type mismatch;

found : Double

required: scala.math.BigInt

a + x

Вот и все основные вещи, которые вам необходимо знать о работе с числами в Scala. И в следующем разделе мы поговорим про булевые переменные.

## **3.3 Булевые значения**

Мы с вами на практике познакомились с числами и давайте поговорим теперь о булевых значениях. Булевые значения, названные в честь английского математика Джорджа Буля, принимают всего два значения **True** или **False**, и обозначают логическую истину или логическую ложь.

val a = true

val b = false

Когда мы говорили про числа, мы не упомянули, что есть большое количество операций над числами, и кстати не только на числами, которые могут вернуть булевые значения. Это операции сравнения.

Любые два числа мы можем сравнить с помощью операторов, больше **>**, меньше **<**, равно, которое обозначается таким двойным равенством **==**, меньше или равно **<=**, обратите внимание, что меньше и больше в таких неточных равенствах всегда идут слева, не равно обозначается восклицательный знак равно **!=** в Scala.

val a = 1 > 5

val b = 1 < 5

val c = 1 == 5

val d = 1 <= 5

val e = 1 >= 5

val f = 1 != 5

### **Операции над булевыми значениями**

Над самими булевыми значениями тоже есть свои операции, например, имея два булевых значения можно найти логическое И между ними. **Логическое И** обозначается в Scala двойным амперсандом **&&**. Оно, напомню, равно истине тогда и только тогда, когда левая и правая часть, обе истинны.

**Логическое ИЛИ** обозначается двойной вертикальной чертой **||**. Оно будет ложью тогда и только тогда, когда левые и правые аргументы ложь.

И **логическое отрицание** обозначается восклицательным знаком **!**, который идет перед именем переменной. Унарная операция. Она будет равна истине, когда переменная равна лжи и лжи, когда переменная равна истине.

val x = 1 > 5

val y = "Oleg" contains "l"

x && y *// И*

x || y *// ИЛИ*

!x *// Отрицание*

val x: Boolean = false

val y: Boolean = true

val res0: Boolean = false

val res1: Boolean = true

val res2: Boolean = true

### **Равенство**

У любых двух переменных, любых двух значений в scala, есть операция, которая возвращает булевое значение, эта операция **равенство**. Мы любые два значения можем проверить на равенство с помощью оператора равно равно **==**, даже если их типы различаются.

val x = "Kate"

val y = "Oleg"

x == y

Как правило, если их типы точно различаются, и они точно не равны, компилятор скала выведет предупреждение, но это не является ошибкой. Мы можем сравнивать значения разных типов.

Также мы унаследовали одно еще специальные равенство для ссылочных значений, это **сравнение по ссылке**. В Scala оно обозначается специальной операцией **eq**.

Например, представьте, что мы сформировали две строчки, которые абсолютно одинаковы, но они сформированы независимо друг от друга, мы взяли два кусочка "Ka" и "te" и склеили их два раза независимо друг от друга. Мы сконструировали два новых объекта, которые имеют одно и то же содержимое.

Если мы сравним их с помощью оператор равенства, Scala поймет, что это две эквивалентных строчки и выдаст True, но если мы сравним их с помощью оператора eq, мы сравним ссылки, и Scala скажет нам, что две эти ссылки указывают на две разные области в памяти и они не равны.

val x = "Ka"

val y = "te"

val a = x + y

val b = x + y

a == b *// true*

a eq b *// false*

Мы с вами изучили булевые значения. В следующем разделе поговорим про строки.

## **3.4 Строки**

Мы с вами поговорили про булевые переменные и наконец добрались до строк. Самый простой способ определить строчку, воспользоваться самым простым синтаксисом двойными кавычками **""**. Двойные кавычки, дальше а какой-то набор символов закрываете двойные кавычки, можете присвоить какой-нибудь переменной. Строчка - это просто последовательность символов.

val name = "Oleg"

### 

### **Сложение**

Эти строчки можно складывать друг с другом почти также, как вы складывали числа друг с другом. Но в данном случае мы говорим о конкатенации, то есть результирующая строка будет состоять из всех символов самой первой строчки, затем будут идти все символы второй строчки, и затем все символы последней строчки.

val name = "Oleg"

val greet = "Hello " + name + "!!!"

### **Интерполяция**

Иногда вместо конкатенации, мы можем пользоваться таким волшебным свойством, как интерполяция. Для этого мы перед двойными кавычками можем поставить символ s. Если мы поставили символ s, тогда внутри строки, мы можем упоминать любые переменные, которые видны нам данном контексте, и они автоматически ставится в то место, где вы их упомянули.

val name = "Oleg"

val greet = s"Hello **$**name!!!"

### **Переносы строк**

Если вы хотите создать строчку сохранив переносы строк друг между

другом, вы можете пользоваться специальными символами вроде "\n", а можете сделать строчку, в которой есть специальные тройные двойные кавычки, она начинается с тройных двойных кавычек и заканчивается тройными двойными кавычками. Внутри может быть сколько угодно символов разделенных переносами строк, и все эти переносы строк сохраняться в вашей строчке.

val s =

"""

переносы

строк

не

потеряны

"""

Иногда для того, чтобы ваш строковый литерал с переносами строк красиво смотрелся рядом с кодом, вам нужно его подвинуть немного вправо, влево.

Для того, чтобы выбросить все промежуточные символы, мы можем поставить в начале каждой строки там, где нам кажется должна начинаться строка, специальный символ вертикальную палочку **|**, и затем вызвать метод **stripMargin**, тогда все символы до до этой вертикальной палочки в многострочной строке автоматически выбросятся вместе с этой палочкой и останутся только символы справа от нее.

val s =

"""|переносы

|строк

|не

|потеряны

""".stripMargin

### **Операции**

Со строчками есть много операций, которые мы можем делать. Например, булевы операции, которые могут выяснить, начинается ли одна строчку с другой, заканчивается ли одна строчка другой, и содержит ли одна строчка другую где-нибудь в середине. Соответствующие методы **startsWith**, **endsWith** и **contains**.

val s = "aaabbb"

s.startsWith("aa")

s.endsWith("bb")

s.contains("ab")

Иногда мы можем пользоваться дополнительной операцией matches.

Операция matches принимает на вход регулярное выражение, это такая строчка которая содержит кучу специальных дополнительных символов и на самом деле это специальный встроенный язык программирования.

s.matches("a+b+")

### **Регулярные выражения**

Вы можете ввести в поисковой строке Java regular expressions или регулярные выражения в Java или регулярные выражения в Scala и почитать об этом подробнее. Мы конкретно этот синтаксис рассматривать не будем.

Если мы хотим воспользоваться всей мощью регулярных выражений, мы можем превратить строку в специальный объект, который содержит множество специальных операций для регулярных выражений, для этого достаточно вызвать метод **.r** у нашей строчки. О таких объектах мы поговорим позже.

val regex = "a+b+".r

### 

### **Приведение к строке**

Любой объект в Scala, любое значение, можно привести к строчке. Для этого у нас есть метод **toString** практически у любого значения. Мы можем вызвать его явно или воспользоваться оператором конкатенации. Строчку можно конкатенировать не только с другой строчкой, но и с любым другим значением, Scala автоматически вызовет метод toString у них.

val x = 2

x.toString

"x is " + x

Мы с вами изучили строки и давайте попробуем их на практике.

### **Практика**

Мы с вами познакомились со строковыми значениями и давайте что-нибудь попробуем сделать с ними на практике. Самый простой способ определить строковое значение, как мы говорили, это написать кучу кавычек (три двойных кавычки и Еnter). Если мы определяем многострочное значение, как видите, наш IDE позволяет сразу писать их с учетом сдвига. Она сдвигается с помощью вертикальной палочки определения нашей многострочной переменной, где то к ее имени, для того чтобы у нас не было некрасивого кода или некрасивых пробелов справа.

val s =

"""

|

|

|""".stripMargin

Имея такую строковую переменную, мы можем, например, вычислить сколько в ней символов. В ней оказывается 32 символа. Можем попробовать разбить ее на строчки, это получится некий массив, и посчитать сколько элементов в нем. Получили 6 строчек.

val s =

"""

|fdsafsafdfsdf

|d

|

|12312312331

|2

|""".stripMargin

s.length *// символов*

s.split("\n") *// массив*

s.split("\n").length *// строк*

val res0: Int = 32

val res1: Array[String] = Array("", fdsafsafdfsdf, d, "", 12312312331, 2)

val res2: Int = 6

Конкретно, что можем делать с массивами, мы рассмотрим, когда будем говорить о коллекциях, а сейчас вернемся к строчкам.

Представим, что мы сгенерировали более интересную строчку. Например, взяли число 2, мы хотим очень большое число два, и возвели его в 10-тысячную степень. Такая огромная куча цифр.

val x = *BigInt*(2).pow(10000).toString()

val x: String = 1995063116880758384883742162683585083823496831886192454852008949852943883022194663191996168403619459789933112942320912427155649134941378111759378593209632395785573004679379452676524655126605989552055008691819331154250860846061810468550907486608962488809048989483800925394163325785062156830947390255691238806522509664387444104675987162698545322286853816169431577562964076283688076073222853509164147618395638145896946389941084096053626782106462142733339403652556564953060314268023496940033593431665145929777327966577560617258203140799419817960737824568376228003730288548725190083446458145465055792960141483392161573458813925709537976911927780082695773567444412306201875783632550272832378927071037380286639303142813324140162419567169057406141965434232463880124885614730520743199225961...

Мы хотим найти что-нибудь в этой куче цифр, например, у нас какая-нибудь задачка на изобретательность, узнать, есть ли в этом числе семерки. Мы можем превратить эту штуку toString и спросить, содержишь ли ты семерку? Оказывается да, содержит. А две семерки подряд? Тоже содержится. Три семерки подряд? И 3 содержится. Четыре? А вот четыре у нас нет, но три все еще содержит.

x.contains("777")

x.contains("7777")

val res0: Boolean = true

val res1: Boolean = false

Теперь стоит вопрос, а как же нам теперь найти максимальную последовательность цифр одинаковых, которая встречается в этом числе 2 в десятичной степени. С этим нам могут помочь регулярные выражения. Одна из особенностей регулярных выражений, как я уже говорил, с синтаксисом вам лучше ознакомиться самостоятельно, потому, что эта тема не для одной лекции, ну одна из ее особенностей заключается в том, что мы можем помечать некие блоки, в данном случае это блок состоящий из одного любого символа.

В нашей строчки все символы - это цифры. И мы можем снова ссылаться на ту же самую строчку. Например, мы можем сказать: любые два символа подряд. Когда мы сконструировали такое регулярное выражение, можем назвать его reg, можем запустить **метод .r**, который превратит строку reg в регулярное выражение.

val reg = "(.)\\1".r

val reg: scala.util.matching.Regex = (.)\1

У этого объекта есть куча разных интересных методов, которые можем использовать. Нас интересует **findFirstIn**. Ищем первую последовательность из двух цифр одинаковых подряд в нашей строчке X. Нашли некую последовательность 99. Видимо она идет практически со второго знака.

reg.findFirstIn(x)

val res2: Option[String] = Some(99)

Давайте найдем последовательность из трех знаков подряд. Нашли 111. Давайте из четырех знаков подряд. Нашли 3333.

val reg = "(.)\\1\\1\\1".r

reg.findFirstIn(x)

val res2: Option[String] = Some(3333)

Попробуем пять знаков подряд. И вот тут уже мы точно знаем, что в числе 2 в 10 тысячной степени нет 5 знаков подряд.

val reg = "(.)\\1\\1\\1\\1".r

reg.findFirstIn(x)

val res2: Option[String] = None

А что если 2 в стотысячной степени? Ну вот, у нас есть оказывается

5 девяток. Еще больше? И 6-и цифр подряд нет даже в этом огромном числе.

val x = *BigInt*(2).pow(100000).toString()

val reg5 = "(.)\\1\\1\\1\\1".r

val reg6 = "(.)\\1\\1\\1\\1\\1".r

reg5.findFirstIn(x)

reg6.findFirstIn(x)

val res0: Option[String] = Some(99999)

val res1: Option[String] = None

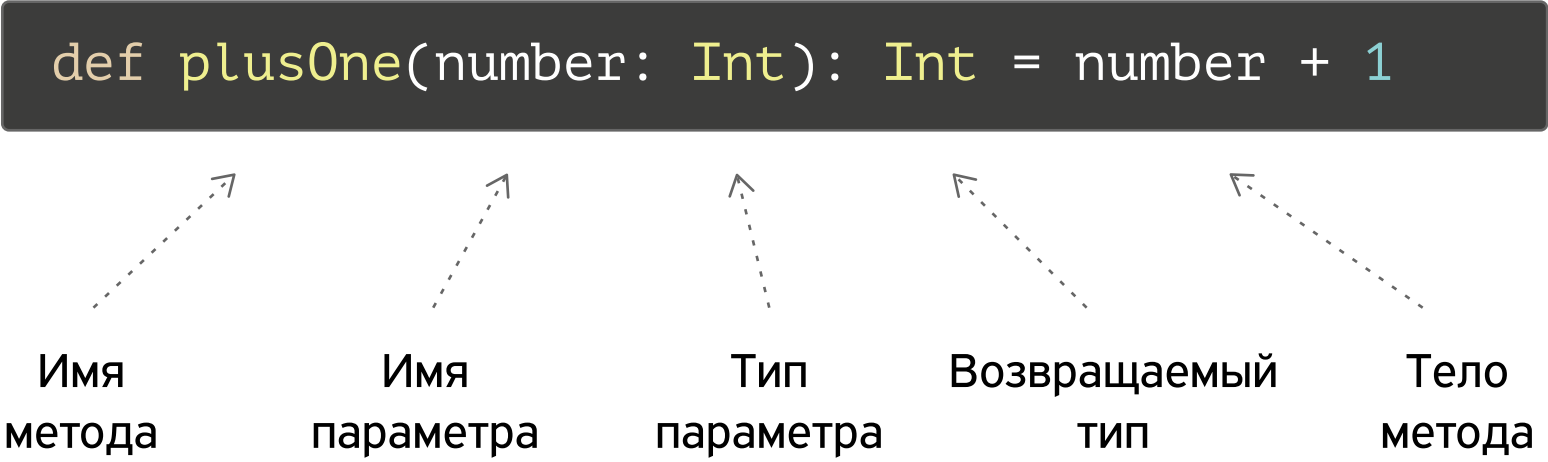
Вот такие интересные задачи на пересечении чисел и строк, вы можете пробовать решать с помощью, например, регулярных выражений.

Ну а в следующем видео мы с вами познакомимся с методами

# Функции

## **4.1 Методы**

Мы с вами познакомились с типами в Scala, настала пора узнать про методы. Методы - это лучший способ определить какой-нибудь переиспользуемый кусочек кода. Когда я хочу определить метод, я пишу ключевое слово **def**, затем пишу имя метода и открываю круглые скобки. В круглых скобках я перечисляю параметры, для каждого записываю имя, двоеточие и тип параметра. Закрываю круглые скобки, ставлю двоеточие и пишу возвращаемый методом тип. Затем ставлю равно и записываю тело метода.



Чтобы вызвать метод, я просто пишу имя метода и в круглых скобках передаю ему все параметры.

plusOne(4) *// 5*

Тип результата можно иногда опускать, если он совершенно очевидно следует из тела метода. Scala попробует вывести тип сама.

def plusOne(number: Int) = number + 1

*// Тип результата иногда можно опускать*

*// Scala попробует вывести тип сама*

У методов может быть несколько параметров, их мы перечисляем через запятую. У методов может быть даже несколько списков параметров. Если так, то каждый список мы ставим в отдельные круглые скобки и то же самое делаем при вызове метода.

def plusOne(x: Int, y: Int, z: Int): Int = x + y + z

*// У метода может быть несколько параметров*

plusOne(1, 2, 3) *// 6*

*plusOne(1, 2)(3) // 6*

*// Параметры можно группировать в список*

У методов может вообще не быть параметров, тогда этот метод можно воспринимать как переменную, которая будет вычисляться каждый раз при ее использовании.

*def sixty: Int = 10 \* 6*

*// У метода может вообще не быть параметров*

*// Это можно воспринимать как переменную, которая будет*

*// вычисляться каждый раз*

*sixty // 60*

Если методы делают что-то более сложное, чем одно выражение, мы можем взять целый блок вычислений в фигурные скобки. Внутри этого блока можем объявлять новые переменные, вызывать какие-то другие методы, и самом конце написать выражение, которое вернет нам результат.

def plusAndPrint(x: Int, y: Int): Int = {

val result = x + y

*println*(s"**$**x + **$**y = **$**result")

result

}

*// Блок вычислений можно взять в фигурные скобки*

plusAndPrint(2, 3) *//prints 2 + 3 = 5 , returns 5*

Если наш блок просто делает какие-то действия и ничего не возвращает, мы ставим у метода возвращаемый тип **Unit**.

def plusAndPrint(x: Int, y: Int): Unit = {

val result = x + y

*println*(s"**$**x + **$**y = **$**result")

}

*// Методы, не возвращающие ничего, должны возвращать*

*// тип Unit*

plusAndPrint(2, 3) *//prints 2 + 3 = 5*

### **Вложенные методы**

Методы можно вызывать внутри любых блоков, в том числе, внутри тел других методов. Если мы объявили метод внутри тела другого метода, он может ссылаться на все параметры внешнего метода.

def plusMul(q: Int, x: Int, y: Int): Int = {

def mul(u: Int) = q \* u

mul(x) + mul(y)

}

*// Внутри любых блоков можно объявлять методы*

*// Внутри тела метода вы можете ссылаться*

*// на его параметры*

plusMul(10, 2, 3) *// 50*

Мы изучили методы в в следующей части рассмотрим их подробнее.

### **Повторяемые параметры**

Мы с вами начали знакомиться с методами - продолжим. Последний параметр в списке параметров может быть объявлен как повторяемый. Для этого мы после его типа ставим звездочку. Тогда, при вызове, вместо одного значения можно передать целый список. В теле метода этот список параметров становится коллекцией, а не одним значением. Что можно делать с коллекциями, мы узнаем позже.

def sumAllTimes(u: Int, nums: Int\*): Int = u \* nums.sum

*// Последний параметр в списке параметров может быть*

*//"повторяемым", тогда в теле метода его можно*

*//использовать как последовательность*

*//О последовательностях и других коллекциях мы узнаем*

*//в следующих уроках*

sumAllTimes(3, 1, 2, 3) *// 3 \* (1 + 2 + 3) = 18*

### **Значения по умолчанию**

Для каждого параметра можем определить значение по умолчанию. Для этого, сразу после того, как мы указали тип параметра, можно написать равно и выражение, которое будет использовано для этого параметра по умолчанию. И тогда при вызове можем опустить передачу некоторых параметров.

def plus3(x: Int, y: Int = 0, z: Int = 0): Int =

100 \* x + 10 \* y + z

plus3(1) *// 100*

plus3(1, 2) *// 120*

plus3(1, 2, 3) *// 123*

### **Именованные аргументы**

Мы можем явно сказать, какое выражение мы передаем для какого параметра. Для этого при вызове можно написать имя равно и значение, которое мы хотим передать именно для этого параметра. Это бывает очень полезно, если у метода много параметров, или, например, есть несколько параметров со значениями по умолчанию, и вы хотите для одного из них передать, а для другого нет.

def plus3(x: Int, y: Int = 0, z: Int = 0): Int =

100 \* x + 10 \* y + z

*// Если параметров много, или несколько параметров*

*// имеют значение по умолчанию, при вызове аргументы*

*// можно упоминать по имени*

plus3(x = 1) *// 100*

plus3(1, z = 2) *// 102*

plus3(x = 1, z = 3, y = 2) *// 123*

### **Передача по имени**

Если у вас есть какой-то параметр с каким-нибудь, например, сложно вычисляемым выражением или выражением, которое содержит side эффекты, и вы хотите чтобы оно вычислялось только в момент, когда оно действительно нужно, вы можете воспользоваться передачей по имени. Для этого перед типом параметра, вы можете указать вот такую стрелку, и тогда, в теле метода, этот параметр будет вычислен только в момент когда он действительно нужен. Но осторожно, если вы сошлетесь на этот параметр несколько раз, он будет вычислен несколько раз.

def replaceNegative(x: Int, z: => Int): Int =

if (x >= 0) x else z

*// Если вы хотите, чтобы значение выражения вычислялось*

*// не всегда, можно использовать передачу "по имени"*

*// Значение параметров, переданных по имени,*

*// вычисляется только в случае необходимости*

replaceNegative(1, 3 \* 3 \* 3) *// 1*

replaceNegative(-1, 3 \* 3 \* 3) *// 27*

### **Передача блока**

Если ваш список параметров состоит ровно из одного параметра, вы можете передать его прямо блоком в фигурных скобках. Это бывает очень полезно когда вы, например, хотите передать какой-то список действий, и тогда вы иногда используете также и передачу по имени.

def replaceNegative(x: Int, z: => Int): Int =

if (x >= 0) x else z

*// Если список параметров содержит ровно один параметр,*

*// его значение можно передать блоком в фигурных скобках*

replaceNegative(1){

*println*("calculated")

3 \* 3 \* 3

} *// 1*

replaceNegative(-1){

*println*("calculated")

3 \* 3 \* 3

} *// prints "calculated"; returns 27*

### 

### **Рекурсия**

Функция может ссылаться сама на себя внутри своего тела с другими параметрами - это называется рекурсией. Если мы определяем рекурсивную функцию, нам нужно обязательно указывать тип, потому что Scala сама по себе не сможет его определить. Таким образом мы можем, например, вычислить сумму чисел каком-нибудь диапазоне. Это будет очень неэффективно, если функция будет сама себя вызывать много много раз, мы можем переполнить стек.

def sumRange(from: Int, to: Int): Int =

if(to < from) 0

else from + sumRange(from + 1, to)

*// Функция в своём теле может снова вызывать себя*

*// Это называется рекурсия*

*// Для рекурсивных функций обязательно указывать тип*

sumRange(1, 10) *// 55*

### **Хвостовая рекурсия**

Для того, чтобы этого не происходило, существует оптимизация под названием хвостовая рекурсия. Функция, которая применяется, эта оптимизация, должна выглядеть очень специальным образом. Грубо говоря, это должно быть большое условное выражение, и каждый результат в этом условном выражении, должен быть либо вызов той же самой функции с другими параметрами, либо какое-то выражение которое не ссылается на эту функцию, тогда вся эта функция превратиться в один большой цикл, и тогда такие рекурсивные вещи можно будет вычислять гораздо быстрее. С другой стороны, мы можем превратить нашу функцию в бесконечный цикл.

def sumRange(from: Int, to: Int, acc: Int = 0): Int =

if (to < from) acc

else sumRange(from + 1, to, acc + from)

*// Если вызовы самой себя происходят только*

*// в "хвостовых точках" вычислений, функция может*

*// быть оптимизирована в "хвостовую рекурсию", которая*

*// внутри себя будет представлять из себя цикл*

sumRange(1, 10) *// 55*

Мы с вами изучили методы, давайте попробуем их на практике.

### **Практика**

Мы с вами познакомились с методами, давайте попробуем применить на практике. Давайте попробуем написать функцию, которая будет вычислять сумму всех четных чисел от единицы до некого N.

Итак сумма целых чисел. Имеем некое целое N и должны вернуть какое-то целое число. Внутри нам нужно определить некую функцию для обхода всех четных чисел. Имея какое-то очередное четное число будем возвращать сумму. Проверяем если наше число стало уже больше n, то нам очевидно нужно закончить обход, т.е. вернуть сумму всех оставшихся чисел равной нулю. Иначе, мы можем просто добавить данное число, предполагая что оно целое, к сумме, и запустить функцию от нашего числа + 2. И начнем наш обход, с самого первого четного целого числа - это 2.

Определили вот такую функцию. Давайте попробуем ее запустить до 100. Получили результат 2550 - похож на правду. Давайте попробуем до 10 тысяч. Получили еще результат 25005000. Давайте попробуем теперь до 100 тысяч. Получили ошибку StackOverflowError.

def sumEven(n: Int): Int = {

*// имея четное число будем возвращать сумму*

def go(i: Int): Int =

*// Если число больше n возвращаем сумму*

*// оставшихся чисел равной нулю*

if(i > n) 0

*// иначе, прибавляем число к сумме*

else i + go(i + 2)

*// начинаем с первого четного числа 2*

go(2)

}

sumEven(100)

*// val res0: Int = 2550*

sumEven(10000)

*// val res1: Int = 25005000*

sumEven(100000)

*// java.lang.StackOverflowError*

Оказалось, что наша функция go, внутри нашей функции, запускала много-много раз себя и виртуальной машине Java это надоело, стек переполнился, и мы больше не можем так вычислять.

Для этого нам нужно оптимизировать наш обход таким образом, чтобы рекурсия была хвостовой, и мы могли бы запускать обход без stackOverflow.

Для этого определяем дополнительный параметр, аккумулятор, тоже типа числа Int, и каждый раз когда рекурсивно запускаем саму функцию go, вместо того чтобы требовать, что мы вернемся из этой функции и добавим к ней что-то, будем просто добавлять какое-то очередное число к аккумулятору. В самом конце мы должны вернуть всю накопленную сумму, т.е. аккумулятор. Ну и начинаем вычисления с аккумулятора равного нулю. Как видите мы легко все вычислили.К несчастью, у нас произошло переполнение самого Int.

def sumEven(n: Int): Int = {

*// определяем дополнительный параметр acc*

def go(i: Int, acc: Int): Int =

*// возвращаем всю накопленную сумму, т.е.*

*//аккумулятор*

if(i > n) acc

*// добавляем число к аккомулятору*

else go(i + 2, acc + i)

*// начинаем вычисления с аккумулятора равного 0*

go(2, acc = 0)

}

sumEven(100)

sumEven(10000)

sumEven(100000)

*// val res2: Int = -1794917296*

Нам нужно переделать возвращаемый тип в Long, то же самое сделаем с аккумулятором и с возвращаемым типом go. Вот теперь получили корректные значения, можно убедиться что мы без особых

проблем можем запустить сумму и до миллиона, и до 10 миллионов и в общем до чего угодно другого.

*// аккумулятор и возвращаемые типы Long*

def sumEven(n: Int): Long = {

def go(i: Int, acc: Long): Long =

if(i > n) acc

else go(i + 2, acc + i)

go(2, acc = 0)

}

sumEven(100000)

*// val res0: Long = 2500050000*

sumEven(1000000)

*// val res1: Long = 250000500000*

sumEven(10000000)

*// val res2: Long = 25000005000000*

## **4.2 Функции**

Мы с вами познакомились на практике с методами и пришло время узнать почему Scala называют функциональным языком программирования.

Итак, функция. Функция - это такое выражение, которое может быть использовано как метод. Тип для этого выражения записывается вот так: пишете тип параметра, стрелочка, тип результата, или типы всех параметров через запятую, в круглых скобках, стрелочка, тип результата. Максимальное количество параметров, которые у Scala может быть у функции - 22.

val x: Int => Int = ...

val y: (Int, Int) => Int = ...

*// Функция — это значение, которое может быть*

*// использовано как метод*

*// Максимальное количество параметров у функции — 22*

### **Лямбда-абстракция**

Задать значение для такой функции, т.е. определить функцию, вы можете, например, с помощью лямбда-выражений. Для того, чтобы определить лямбда-выражение, вы перечисляете имена параметров, которые вы собираетесь использовать: пишете стрелочка и дальше пишите тело выражения, в котором используются параметры с заданными вами именами.

val addOne: Int => Int = x => x + 1

val plus: (Int, Int) => Int = (x, y) => x + y

*// Значения для функций можно задавать*

*// через лямбда-синтаксис*

Вы может также указывать типы параметров прямо рядом c именами параметров, тогда Scala попробует вывести общий тип для вашей функции самостоятельно.

val addOne = (x: Int) => x + 1

val plus = (x: Int, y: Int) => x + y

*// Можно указывать тип параметров прямо в*

*// лямбда-абстракции, тогда scala попробует вывести тип*

*// всего выражения сама*

### **Короткая запись**

Есть еще более короткий способ записать функцию. Если каждый параметр у вас используется ровно один раз, и все параметры используются ровно по порядку в том, котором они у вас определяются в списке параметров, вы можете в выражении вместо каждого параметра просто ставить подчёркивание.

val addOne: Int => Int = \_ + 1

val plus = (\_: Int) + (\_ : Int)

*// Вместо именованых параметров, если они встречаются*

*// один раз в выражении, можно использовать запись*

*// с подчёркиваниями*

### **Эта-конверсия**

Альтернативный способ определить функцию - это взять метод и превратить его в функцию с помощью эта-конверсии. Оператор, который это делает - подчёркивание, которое мы пишем сразу после имени метода.

Мало того, если Scala понимает в каком-то месте, что она хочет видеть здесь функцию, вы можете даже не ставить это подчеркивание, метод автоматически превратится функцию.

def addOne(x: Int) = x + 1

val add1 = addOne \_

def plus(x: Int, y: Int) = x + y

val pl: (Int, Int) => Int = plus

*// Также в функции можно превращать методы. Сделать*

*// это можно, поставив подчёркивание после метода*

*// Если компилятор понимает, что значением должна быть*

*// функция, подчеркивание можно не ставить*

### **Функция - значение**

Функции могут быть использованы везде, где могут быть использованы обычные значения. Вы можете передавать их в качестве параметров в методы и функции, и вы можете возвращать их качество результатов.

def greaterOn(f: Int => Int): (Int, Int) => Boolean =

(x, y) => f(x) > f(y)

*// Функции можно передавать в качестве параметра*

*// и возвращать из методов и функций в качестве значений*

val greaterOnOnes = greaterOn(x => x % 10)

greaterOnOnes(23, 45) *// false*

greaterOnOnes(27, 45) *// true*

### **Каррирование**

Одной из техник, которые используют эту особенность, является каррирование. Представим, что у вас есть функция от нескольких параметров, в данном случае от двух. Вы, вместо функции от двух параметров, представляете функцию от одного параметра, которая возвращает еще одну функцию от одного параметра, который возвращает ещё одну и так далее, и так далее, и в самом конце результат. В данном случае мы сделали это два раза.

def plus: Int => Int => Int = x => y => x + y

*// Вы можете представить функцию многих параметров,*

*// как последовательность функций от одного параметра,*

*// возвращающих функцию*

plus(1)(2) *// 3*

Задавать функции, вы можете, как с помощью обычной лямбды, а можете, например, вызвать метод curried у какой-нибудь простой функции с несколькими параметрами.

val plus3 = (x: Int, y: Int, z: Int) => x + y + z

val plus3c: Int => Int => Int => Int = plus3.curried

*// Вы можете превратить функцию многих параметров*

*// в каррированный вариант с помощью метода curried*

plus3c(1)(2)(3)

### **Композиция**

Если у вас есть две функции, и вы хотите создать новую функцию, которая будет состоять из того, что вы примените одну к результату другой, вы можете использовать готовые методы для композиции.

Например, метод **andThan**, который возьмет результат функции слева и передаст его в качестве аргумента в функцию справа. Или метод **compose**, который сделать ровно наоборот, возьмет результат функции справа и передаст его в качестве аргумента в функцию слева.

val plus1 = (\_: Int) + 1

val mul3 = (\_: Int) \* 3

val plusThenMul = plus1 andThen mul3

val plusBeforeMul = plus1 compose mul3

*// Функции, состоящие из последовательного вызова*

*// других функций, можно задавать композицией*

*// с помощью методов andThen и compose*

plusThenMul(5) *// mul3(plus1(5)) = 18*

plusBeforeMul(5) *// plus1(mul3(5)) = 16*

Мы с вами познакомились с функциями, давайте попробуем их на

практике.

### **Практика**

Мы с вами познакомились с функциями, давайте попробуем сделать что-либо с ними на практике.

Сначала попробуем сделать какую-нибудь функцию. Определим функцию, которая увеличивает число на единицу.

val add = (\_: Int) + 1

Теперь напишем другую функцию, которая у нас будет брать функцию вроде первой, из Int в Int, и запустит ее для значения 42.

val calc42 = (f: Int => Int) => f(v1 = 42)

Мы можем узнать что calc42 от add1 равно 43.

calc42(add1)

val res0: Int = 43

Но, мы так же знаем, что особенность в лямбдах в том, что они могут представлять анонимные функции, то есть функции, которые мы можем определять прямо в месте, где мы их применяем. Например, прямо сюда можно написать "\_ + 1" и получить тот же самый результат.

calc42(\_ + 1)

val res1: Int = 43

Теперь хотим что-нибудь более интересное, например, передать сюда функцию, которая вычисляет сумму всех чисел от единицы до заданного N. Можно написать метод def sumTo. Это некая функция которая будет обозначать Int. Если наше X равно нулю - сумма очевидно равна нулю, в противном случае добавляем X к сумме всех чисел до X минус 1. Можем теперь передать sumTo прямо в качестве параметра в функцию calc42. Получаем результат.

def sumTo(x: Int): Int =

if (x == 0) 0 else x + sumTo(x - 1)  
calc42(sumTo)

val res2: Int = 903

Как нам теперь записать то же самое прямо в теле функции?

Допустим у нас есть какое-то X типа Int и мы проверяем равен ли он нулю, если так возвращаем ноль, иначе X плюс что? Ведь эта функция теперь не имеет никакого имени, нам нужно придумать каким образом сделать из любой произвольной функции ее рекурсивную версию.

Эта задача была давным давно решена, еще на заре развития функционального программирования был придуман, так называемый, (вай Combinator- проверить термин). Вай Combinator, в общем случае, очень прост, находит неподвижную точку или, по-английски, Fix Point, от такой функции.

В данном случае нам нужно искать неподвижную точку из функции, которая принимает на вход функцию и возвращает ее рекурсивную версию. Т.е. некая функция, которая принимает в качестве параметра собственно рекурсивную версию и возвращает ее же, но дополняет какими-то действиями.

Для того, чтобы определить такой Fix Combinator, нам нужно передать в функцию f значение того же самого выражения. Но если мы попробуем вычислить fix от какого-нибудь выражения, например, выражения f, вернем единицу и вернем лямбду, которая для чего угодно возвращает единицу, мы получим StackOverflowError.

def fix(f: (Int => Int) => Int => Int): Int => Int =

f(fix(f))

fix(f => \_ => 1)

java.lang.StackOverflowError

Почему? Потому, что для того чтобы вычислить результат нужно запустить функцию f. Для того чтобы запустить функцию f, она принимает какой-то параметр, нам нужно вычислить ее параметр, для того чтобы вычислить ее параметр нам нужно снова вычислить fix от f.

Чтобы наконец разобраться с этой бесконечной рекурсией, нам нужно сделать что-нибудь более ленивым способом. Для этого мы можем применить, упомянутую технику передачи параметров по имени.

В нашей функции, которая передается в функцию fix, мы можем сказать, что ее параметр, рекурсивная версия, будет ленивой, т.е. будет использована только в тот момент, когда она действительно нужна. Тогда действительно можем вычислить Fix Point практически любой функции.

def fix(f: (**=>** Int => Int) => Int => Int): Int => Int =

f(fix(f))

Например, от функции, которая будет вычислять сумму. Она принимает некую свою рекурсивную форму по имени, дальше принимает очередной параметр "x" и дальше смотрит, если он равен нулю возвращает ноль, иначе рекурсивно запускает себя для "x" минус 1. Результатом будет некая функция из Int в Int.

def fix(f: (=> Int => Int) => Int => Int): Int => Int =

f(fix(f))

fix(rec => x => if (x == 0) 0 else rec(x - 1))

Можем сразу попробовать вычислить, например, от 7, тут получим 0. Ну да, конечно, потому что мы забыли добавить нашу сумму сам "x". И вот получаем результат 28.

def fix(f: (=> Int => Int) => Int => Int): Int => Int =

f(fix(f))

fix(rec => x =>

if (x == 0) 0 else x + rec(x - 1))(v1 = 7)

val res3: Int = 28

Теперь можем написать calc42, прямо сюда передать параметр fix, и передать нашу лямбду, которую только что записали, т.е. переписав рекурсию в анонимной форме. Результат будет 903, тот же, что и прошлый раз.

def fix(f: (=> Int => Int) => Int => Int): Int => Int =

f(fix(f))

calc42(fix(rec => x =>

if (x == 0) 0 else x + rec(x - 1)))

val res4: Int = 903

Вот так мы можем применять функции на практике, а теперь давайте познакомимся с обобщенными методами.

## **4.3 Обобщенные методы**

Мы с вами познакомились с функциями на практике, давайте попробуем разобраться обобщенными методами.

Все методы, которые мы с вами до этого писали, работали с конкретными типами параметров и конкретными типами возвращаемых значений. И могло получиться вот так: мы с вами написали три метода, они практически идентичны, но они работают с разными типами входных параметров и возвращаемых значений.

def ifThenElse**Int**(cond: Boolean)

(t: => **Int**, e: => **Int**): **Int** =

if(cond) t else e

def ifThenElse**String**(cond: Boolean)

(t: => **String**, e: => **String**): **String** =

if(cond) t else e

def ifThenElse**Long**(cond: Boolean)

(t: => **Long**, e: => **Long**): **Long** =

if(cond) t else e

Оказывается, что можем обобщить их в один метод, который будет точно также написан, но будет содержать дополнительную часть - параметр типа.

def ifThenElse[**A**](cond: Boolean, t: => **A**, e: => **A**): **A** =

if(cond) t else e  
*// Методы могут иметь параметры типа*

*// Параметры можно использовать в типах параметров или*

*// возвращаемого значения*

*// Такие методы можно переиспользовать для разных типов*

Такие методы называется обобщенными или Generic методы. Если вы хотите какие-то типы вынести качестве параметров, все эти Type-параметры, их имена вы причисляете в квадратных скобках между имени метода и списком его значений. Тогда при вызове метода, вы можете явно сказать какой из вариантов методов, вы в данный момент вызываете указав конкретный тип в квадратных скобках во время вызовов.

ifThenElse[String](1 > 2, "one", "two")  
*// :String = "two"*

ifThenElse[Int](1 > 2, 1, 2) *// :Int = 2*

Иногда Scala сама может понять, какой из вариантов вашего метода вы сейчас используете, и тогда при вызове вы можете опустить явное указание типа.

ifThenElse(1 > 2, "one", "two") *// :String = "two"*

ifThenElse(1 > 2, 1, 2) *// :Int = 2*

*//Если компилятор может вывести параметры типа*

*//на основе входных параметров, их можно не указывать*

Такие методы очень удобны, когда мы пытаемся определить какие-нибудь комбинаторы для функций или для коллекций.

def combineOn[A, B](comb: (B, B) => B)  
 (f: A => B, g: B => A): (A, A) => A =

(x, y) => g(comb(f(x), f(y)))

*//Их удобно использовать для методов принимающих*

*//функции в качестве параметров*

val sumStrings = combineOn[String, Int](\_ + \_)  
 (\_.toInt, \_.toString)

sumStrings("123", "32") *//: String = "155*

Мы с вами познакомились с обобщенным методами. Давайте попробуем немного разобрать их на практике.

### **Практика**

Мы с вами познакомились с обобщенным методами и пора попробовать их на практике.

Как вы помните, у нас была функция calc42, которая брала любую другую функцию от Int, и возвращала нам Int. Она брала функцию, запускала и вычисляла значение параметра 42. Мы во-первых можем обобщить, для того чтобы обобщить функцию в Scala, нам в любом случае придется переделать ее в метод. К счастью у нас есть безпараметрические методы, которые ведут себя точно так же как функция, и мы можем, например, запустить наш метод от значения от функции, которая, например, будет переводить наше число строчку.

val calc42: (Int => Int) => Int = f => f(v1 = 42)

calc42(\_.toString)

Мы хотели бы так сделать, но к несчастью можем передать сюда функцию только, которую возвращает Int. Нужно обобщить calc42, чтобы он мог работать не только с результатом Int, а с любым другим типом. Теперь можем сделать то, что хотим. Можно даже так написать, произвольное число и строчка "number is переданное число".

def calc42[A]: (Int => A) => A = f => f(v1 = 42)

calc42(i => s"number is **$**i")

val res0: String = number is 42

На прошлой практике мы немного сжульничали, помимо того, что мы использовали функцию из Int в Int, мы взяли 42, которое достаточно маленькое число, поэтому вай-комбинатор, который очевидно запускал простую рекурсию, смог легко справится сумму чисел от 1 до 42 просто потому, что число 42 очень маленькое.

Поэтому давайте возьмем 42 миллиона. Теперь мы хотели бы передать сюда какую-то анонимную функцию и запустить значение от 42 миллионов. Например, мы хотим анонимную функцию, которая вычисляет сумму чисел от 1 до 42 миллионов.

Естественно мы не можем сделать это тем же способом с помощью вай-комбинейтора, с помощью функции fix, но мы можем попробовать определить другой комбинатор, назовем его хвострекурсивный комбинатор.

Так у нас будет какой-то тип, с которым будем работать на каждом шаге и какой-то тип, значение которого будем накапливать, и у нас будет куча функций в качестве параметров. Первая функция будет регулярно превращать А следующий А, вторая функция будет регулярно комбинировать текущее значение В с очередным значением А и генерировать новое значение аккумулятора. И дальше будет некий предикат, то есть функция, которая будет брать очередное значение А и проверять, что не стоит ли нам остановиться, и в момент, когда нам стоит остановиться, мы возвращаем последнее значение аккумулятора. Нам, конечно же, потребуются стартовые значения, для переменной А, и стартовое значение для нашего аккумулятора, и вернем мы финальное значение для аккумулятора.

def calc42M[A]: (Int => A) => A = f => f(v1 = 42000000)

calc42M(i => s"number is **$**i")

def tailRec[A, B](iterac: A => A,

comb: (B, A) => B,

cond: A => Boolean)

(start: A, init: B): B = {}

Как он у нас будет работать? Нам потребуется внутри какая-то рекурсивная функция. Каждый раз у неё будет какое-то очередное значение типа А, назовем его "x", и какое-то очередное значение аккумулятора назовем его "acc". В конце она вернет самое последнее значение аккумулятора, которое нам подойдет. На каждом шаге мы проверяем, выполняется ли все еще условие для "x". Если выполняется, мы запускаем следующую итерацию "go", в качестве "x" мы передаем "iter" от текущего "x", в качестве следующего значения аккумулятора мы передаем "comb" текущее значение аккумулятора и "x". В противном случае, если условие не выполнилось, мы наконец возвращаем текущее значение аккумулятора. Запускаем эту функцию от стартовых значений А и В.

def tailRec[A, B](iterac: A => A,

comb: (B, A) => B,

cond: A => Boolean)

(start: A, init: B): B = {

def go(x: A, acc: B): B =

if (cond(x)) go(iterac(x), comb(acc, x)) else acc

go(start, init)

}

Теперь можем попробовать посчитать сумму чисел с помощью такого комбината. Итак calc42M, на вход мы берем число n, дальше нам нужно пройтись по числу хвострекурсивно.

В качестве очередного элемента для итерации мы будем использовать Int, а в качестве аккумулятора куда мы будем складывать сумму тип Long.

На каждом шаге мы будем вычитать из нашего текущего значения единицу, на каждом комбинирующем шаге мы будем брать текущее значение комбинатора и добавлять в него значение итерирующего элемента, и каждый раз, когда мы хотим проверить условие, мы будем проверять не стало ли наше число меньше 0, потому что в значение меньше 0 мы уже не хотим добавлять сумму.

Стартовым значением для нашего А будет число "n", стартовым значением для нашего аккумулятора будет 0, и конечно же нам нужно делать это пока наше число больше или равно нулю.

calc42M( n =>

tailRec[Int, Long](\_ - 1, \_ + \_, \_ >= 0)(n,init = 0)

)

val res1: Long = 882000021000000

И вот получился такой результат суммы всех чисел от 0 до 42 миллионов. Можете проверить что это корректный результат.

Мы с вами познакомились на практике с обобщенным методами, ну а в следующем разделе попробуем познакомиться с управляющими конструкциями.

# 

# Операторы

## **5.1 Управляющие конструкции**

### **Условные выражения if else**

В прошлых видео мы узнали как нам обращаться с функциями и методами в Scala. Cегодня мы попробуем разобраться, какие управляющие конструкции есть языке Scala.

Конструкции - это такие часть языка, которые позволяют указать, когда и сколько раз выполнить какие части кода.

Самой простой управляющей конструкцией является "if". Вот так он выглядит: у вас есть "if", дальше условие, а дальше действие, которое вам нужно выполнить только в том случае, если условие выполняется.

Конечно вы можете указать больше одного действия после условия, и если вам нужно что-то выполнить в том случае, если условие не выполняется, вы можете воспользоваться дополнительной конструкции "else".

Если у вас есть несколько условий вы можете выстроить их в цепочку: "if" первое условие, "else if" второе условие, "else if" можно и третье, и четвертое, и в самом конце самый последний "else".

Второе условие будет проверяться только если не выполнилось первое, третье если не выполнились первые два, и последний блок выполнится только в том случае если ни одно из условий не выполнились в указанном вами порядке.

if (x == 1)

*println*("one")

else if(x == 2)

*println*("two")

else

*println*("???")

Что еще интересно, в Scala конструкция "if" - это выражение, то есть вы можете вместо того, чтобы указывать явно действие, указать после каждой составной части какое-то значение и, например, приравнять его к переменной, и выполнить ваши действия уже над указаной переменой. Как видите кода стало немного меньше.

val string =

if (x == 1)

"one"

else if(x == 2)

"two"

else

"???"

*println*(string)

Мы даже можем взять и всю сложную конструкцию запихнуть в качестве параметра в какую-нибудь функцию, это тоже вполне корректно.

*println*(

if (x == 1)

"one"

else if(x == 2)

"two"

else

"???"

)

### **While-циклы**

Cледующей формой управляющей конструкции является while-цикл. Это очень простая конструкция, которая позволяет вам выполнить действие несколько возможное бесконечное число раз, пока у вас выполняется какое-то условие.

Выглядит он вот так: while, какое-то условие и дальше какие-то действия. Естественно если вы хотите чтобы ваш while выполнялся не бесконечно, где-то рано или поздно, внутри тела while, вы должны сделать что-то, благодаря чему условие перестанет выполняться. И за этим очень сложно уследить, поэтому while используется достаточно редко в Scala.

var i = 0

while(i <= 10) {

*println*(i)

i += 1

}

Еще одной особенностью является тот факт, что обычные ключевые слова, такие как break, который позволяет нам выйти из цикла в любой момент, или continue, которая позволяет нам перейти к следующей итерации не выполняя оставшуюся часть блока, в Scala просто отсутствует.



Поэтому мы используем while очень редко, только когда нам нужно выполнить какую-то особенную оптимизацию.

### **For-циклы**

Гораздо чаще мы пользуемся for-циклами. Они выглядят вот так: for, дальше идет какое-то новое имя переменной, которой вы хотите присваивать каждый раз новому значению, внутри вашего блока, стрелочка в левую сторону, и дальше идет собственно диапазон значений, по которым должна пробежаться ваша переменная.

for(i <- 1 to 10) {

println(i)

}

Вы можете объединить несколько присваиваний переменных в одном блоке, разделив их точкой с запятой ";" в круглых скобках. И тогда каждую из переменных вы можете использовать в блоке как угодно.

for(i <- 1 to 10; j <- i to 10) {

*println*("$i $j")

}

Если у вас действительно очень много присваиваний, вы можете расположить их вертикально и воспользоваться фигурными скобками вместо круглых. Тогда не нужно ставить точки запятой после каждого присваивания и работает точно также.

for {

i <- 1 to 10

j <- i to 10

} *println*(s"**$**i **$**j")

Мало того, внутри for-циклов, вы можете дополнительно ставить условие на присваиваемые таким образом переменные, например, этот цикл не будет выполняться когда "i" будет меньше или равно "j", и таким образом он полностью эквивалентен предыдущей конструкции.

for {

i <- 1 to 10

j <- 1 to 10 if i > j

} *println*(s"**$**i **$**j")

Мы изучили управляющие конструкции Scala. В следующем разделе попробуем познакомиться со способом ветвлений в Scala pattern matching или сопоставление с образцом.

## **5.2 Pattern matching (Сопоставление с образцом)**

В прошлый раз мы разобрали с вами, как нам обращаться с условиями и циклами, а теперь давайте попробуем разобраться с самым интересным и богатым способом ветвлений в Scala pattern matching или сопоставление с образцом.

Выглядит это так: у вас есть какая-то переменная и вы можете написать ключевое слово match, после чего, в фигурных скобках, один за другим, идет множество конструкций, case и дальше большое выражение matching, стрелочка вправо и выражение которое соответствует этому case, дальше следующий кейс, еще одно выражение, и снова конструкция. В самом простом случае мы просто сопоставляем с возможными значениями.

x match {

case 1 => "one"

case 2 => "two"

}

Давайте определим у такую функцию: у нас есть какой-то параметр и мы пытаемся понять какое значение он принимает. Допустим он принимает только единицы и двойки.

def numberName(x: Int) = x match {

case 1 => "one"

case 2 => "two"

}

Что самое интересное, выводимый тип этого выражения, будет самый общий тип, который объединяет все возможные варианты справа, в данном случае это будет строка.

def numberName(x: Int): String = x match {

case 1 => "one"

case 2 => "two"

}

Если параметр равен единице - мы получаем строку "one", если параметр равен двойке - мы получаем строку "two", и если нас параметр равен чему-то другому, наша функция будет генерировать исключение.

numberName(1) == "one"

numberName(2) == "two"

numberName(3) == throw MatchError **!!!!!**

Это проблема, которую вы можете решить добавив в pattern matching дополнительный "case \_" c подчеркиванием. И тогда любое другое значение параметра заматчиться с вашей строчкой.

def numberName(x: Int): String = x match {

case 1 => "one"

case 2 => "two"

case \_ => "unknown"

}

numberName(1) == "one"

numberName(2) == "two"

numberName(3) == "unknown"

Так же, как и в случае с "if", порядок этих "case" очень важен. Например, если вы поменяете последнюю строчку и вторую местами, получится такой вот интересный случай, что вроде бы вы можете заматчить значение параметра "2", но у вас гораздо раньше срабатывает та ветка, которая у вас подходит для любого значения параметра. Поэтому очень внимательно следите за тем в каком порядке вы пишете свои "case".

def numberName(x: Int): String = x match {

case 1 => "one"

case \_ => "unknown"

case 2 => "two"

}

numberName(1) == "one"

numberName(2) == "unknown" **!!!!!**

numberName(3) == "unknown"

Мало того, у нас есть гораздо больше интересных синтаксических конструкций. Например, вы можете замочить сразу несколько значений в одном "case", разделив их вертикальной чертой.

def numberName(x: Int): String = x match

{

case 1 => "one"

case 2 => "two"

case 3 | 4 => "three or four"

case \_ => "unknown"

}

numberName(1) == "one"

numberName(2) == "two"

numberName(3) == "three of four"

numberName(4) == "three of four"

Вы можете дописать какие-нибудь условия, например, здесь мы пытаемся разделить четные и нечетные числа с помощью дополнительных условий, которые мы дописываем в наш "case".

def numberName(x: Int): String = x match {

case 1 => "one"

case 2 => "two"

case x if x % 2 == 0 => "unknown even"

case \_ => "unknown odd"

}

numberName(1) == "one"

numberName(2) == "two"

numberName(3) == "unknown odd"

numberName(4) == "unknown even"

### **Case class**

Гораздо более интересную и часто используемую форму pattern matching представляет из себя поттер matching с case-классами. Case-классы - это такие структуры данных, описываются они вот так: case class, имя структуры и набор полей.

Если у нас есть такая структура, мы можем прямо внутри pattern matching распаковать эти поля и тут же сравнить с какими-то конкретными значениями.

Вот видите, здесь мы в первой строчке сравниваем все возможные адреса, у которых имя страны "Россия", во второй строчке все возможно адреса у которых имя страны "Япония".

**case class Address(country: String, city: String)**

def addressInfo(address: Address): String =

address match{

case *Address*("Russia", \_) => "russian"

case *Address*("Japan", \_) => "japanese"

case \_ => "no info"

}

Мы можем сравнить не только одно поле, но и оба поля, например, такой более богатый pattern matching. Как вы помните подчеркивание означает любое значение, которое вам не интересно.

def addressInfo(address: Address): String =

address match{

case *Address*("Russia", "Moscow") => "russian capital"

case *Address*("Russia", \_) => "russian"

case *Address*("Japan", "Tokio") => "japanese capital"

case *Address*("Japan", \_) => "japanese"

case \_ => "no info"

}

И собственно, если значение вам интересно, вы внутри, в соответствующем месте, где вы собираете за матчить какое-то значение, можете написать с маленькой буквы и имя какой-нибудь переменной, и тогда справа в значении, которое возвращаете, вы можете эту переменную использовать.

def addressInfo(address: Address): String =

address match{

case *Address*("Russia", "Moscow") => "russian capital"

case *Address*("Russia", \_) => "russian"

case *Address*("Japan", "Tokio") => "japanese capital"

case *Address*("Japan", \_) => "japanese"

case \_ => "no info"

}

Поэтому pattern matching служит не только для того, чтобы понять по какому направлению нам двигаться, но также и распаковать, вытащить какие-то поля из нашей структуры и тут же их использовать.

Мы узнали самые базовые вещи сопоставления с образцом, а более интересные и сложные вещи мы узнаем далее.

### **Коллекции**

В предыдущем уроке мы с вами узнали о простых формах сопоставления с образцом, сейчас рассмотрим варианты поинтересней.

Например, мы хотим с помощью сопоставления с образцом посчитать сумму элементов в списке.

Одним из вариантов было бы написать функцию вот таким образом: в верхней строчке мы сопоставляем наш список с не пустым списком, состоящем из головы и хвоста, во втором мы сопоставляем с пустым списком и таким образом можем пробежаться по всем элементам коллекции просуммировав их.

def sum(xs: List[Int], start: Int = 0 ): Int = xs match {

case x *::* rest => sum(rest, start + x)

case *Nil* => start

}

Но к несчастью, такая форма будет работать по сути только для наших связанных списков.

Если мы хотим делать pattern matching для всех коллекций, для любых коллекций одним и тем же способом, мы можем воспользоваться такой интересной формой: сам по себе объект List оказывается можно использовать в pattern matching вот таким способом.

Например, сверху мы будем матчить все пустые списки, ниже все спички состоящие из одного элемента, еще ниже списки состоящие из двух элементов, еще а ниже списки из трех элементов, вот так наша функция сможет просуммировать как минимум списки из трех элементов.

def sum(xs: List[Int], start: Int = 0 ) = xs match {

case List() => start

case List(x) => start + x

case List(x, y) => start + x + y

case List(x, y, z) => start + x + y + z

case \_ => throw new Exception("too many elements")

}

Но если мы хотим все-таки больше, мы можем воспользоваться вот такой конструкцией, она практически эквивалентна предыдущей.

Если у нас есть пустой список - мы возвращаем накопленный аккумулятор, если у нас не пустой список мы берем первый элемент и видите все остальные элементы, вот этот синтаксис "@\_\*" означает все остальные элементы справа от уже указанных. И с помощью вот такого pattern matching мы реально можем просуммировать всю нашу коллекцию.

def sum(xs: List[Int], start: Int = 0 ): Int = xs match {

case List() => start

case List(x, rest@\_\*) => sum(rest.toList, start + x)

}

И с помощью вот такого pattern matching мы реально можем просуммировать всю нашу коллекцию. И что самое интересное, мы ее можем использовать не только для списка, но и для чего угодно, даже для такого общего подтипа коллекций как Seq.

def sum(xs: Seq[Int], start: Int = 0 ): Int = xs match {

case Seq() => start

case Seq(x, rest@\_\*) => sum(rest, start + x)

}

### **Регулярные выражения**

Следующим интересным вариантом являются регулярные выражения. Регулярные выражения достаточно сложная тема, но мы все-таки попробуем их зацепить.

Вот сверху у нас регулярное выражение описано, которое пытается заматчить два слова, состоящих из каких-то букв, разделенные запятой и пробелом. Ну, например, похоже на город улица или город страна. Напоминаем, что для того, чтобы в регулярных выражениях ставить такие вот слеши, чтобы в строчке они у вас распознались нужно вставить 2 слеша подряд. Превратить строчку в объект регулярного выражения, вы можете вызвав метод ".r" у него.

val address = "\\w+, \\w+".r

И что интересно, что сам по себе объект, который соответствует этому регулярному выражению, может использоваться с левой стороны в pattern matching.

Например, если мы хотим сопоставить любую строчку, которая соответствует нашему регулярному выражению, можем просто написать регулярное выражение, круглые скобочки, и окажется наша функция, которая выясняет адрес это или не адрес, возвращает true для всех строчек, которые представляют из себя последовательность символов, запятая, пробел и еще одна последовательность символов. И для всех других строчек, например, у нас строчка без запятой, возвращает false.

val address = "\\w+, \\w+".r

def isAddress(string: String): Boolean = string match {

case address() => true

case \_ => false

}

isAddress("Volgograd, Russia") == true

isAddress("Soviet Union") == false

Регулярные выражения, точно так же как и case-классы и коллекции, могут вытаскивать элементы. Для этого нам нужно объявить группы в регулярном выражении. Это делается с помощью таких круглых скобочек.

val address = "(\\w+), (\\w+)".r

Итак, если мы имеем в нашем регулярном выражении несколько круглых скобочек, мы можем написать case, регулярное выражение и дальше идут имена переменных, в которых вытащаться значения, заматченные соответствующими группами.

Например, можем объявить такую функцию, которая будет не просто проверять является наша строчка адресом или нет, а которая будет формировать опционально экземпляр структуры, адрес со страной и городом. То есть мы буквально вытащим нашим регулярным выражением из строчки пару кусочков и склеим их вместе с помощью такой вот структуры адреса, который мы объявили.

val address = "(\\w+), (\\w+)".r

case class Address(country: String, city: String)

def readAddress(string: String): List[String] =

string match {

case address(city, country) =>

*Some*(Address(country, city))

case \_ => None

}

readAddress("Volgograd, Russia") ==

*Some*(*Address*("Russia","Volgograd"))

readAddress("Soviet Union") == None

Это очень полезно, и мало того, регулярные выражения, точно так же как и коллекции, могут матчиться таким вот образом. Если вы точно не хотите перечислять все группы, которые вы встретили, вы можете оставшуюся незаматчиную часть заматчить точно так же с помощью синтаксиса "@\_\*", и таким образом получить список, в данном случае, из двух подстрочек, которые представляют адрес.

val address = "(\\w+), (\\w+)".r

def readAddress(string: String): List[String] =

string match {

case address(parts@\_\*) => parts.toList

case \_ => *List*()

}

readAddress("Volgograd, Russia") ==

*List*(Volgograd, Russia)

readAddress("Soviet Union") == *List*()

### **Комбинация**

И самая мощная и самая классная часть по поводу регулярных выражений в том, что все эти способы можно комбинировать один внутри другого. Например, наше регулярное выражение, которое выясняет является ли страна какой-то известной нам, допустим только известны Россия и Япония.

Таким образом мы одновременно распаковываем нашу страну в строчку и проверяем эту строчку на соответствие регулярному выражению.

val regex = "(Russia|Japan)".r

case class Address(country: String, city: String)

def getInfo(address: Address): String =

address match {

case *Address*(**regex(country)**, city) =>

s"**$**city, **$**country"

case *Address*(\_, \_) => "unknown country"

}

Точно так же можем использовать группы, и если нам нужно какую-то часть внутри такого сложного pattern matching приравнять какой-нибудь новой переменной и использовать ее справа, мы можем всегда поставить "@" - специальный символ означающий, что эта переменная, которую я указал слева с маленькой буквы, я ее сопоставляю с этим всем субвыражением внутри case, которую собираюсь использовать справа от стрелочки.

val regex = "(Russia|Japan)".r

case class Address(country: String, city: String)

def getInfo(address: Address): String =

address match {

case *Address*(**country@regex(\_)**, city) =>

s"**$**city, **$**country"

case *Address*(\_, \_) => "unknown country"

}

Вот такие сложные интересные мощные вещи можем делать с помощью сопоставления с образцом pattern matching, а в следующем разделе изучим частичные функции.

## **5.3 Частичные функции (Partial functions)**

Мы с вами на практике познакомились с сопоставлением с образцом и сейчас мы узнаем, что такое частичная функция.

Частичная функция или Partial function - это как просто функция, у нее есть тип входного параметра и тип возвращаемого, но есть один дополнительный метод isDifinedAt, который для любого входного параметра, которые эта функция может обработать, вернет нам true, а для всех остальных вернет false.

PartialFunction[-A, +B]

def isDefinedAt(input: A): Boolean

Что самое интересное, частичные функции это воплощение pattern matching в типы, т.е. можно задать любой pattern matching и с помощью него задать частичную функцию.

Например, частичную функцию, которая берет число 10, и пытается разделить его на наш параметр без остатка.

У нас есть всего четыре числа, для которых она может это сделать и мы убеждаемся, что если мы например придадим ей двойку, на которую 10 делится без остатка, мы получаем результат true, вызывая метод isDefinedAt, а если передадим что-нибудь еще, получаем false.

Если мы вызовем нашу частичную функцию, как простую функцию с параметром, который ей подходит, мы получим результат, если с чем-нибудь еще, мы получим какое-то исключение.

Но самое главное, что мы можем узнать заранее, сгенерирует ли наша функция исключение для данного входного параметра.

val divide10: PartialFunction[Int, Int] = {

case 1 => 10

case 2 => 5

case 5 => 2

case 10 => 1

}

divide10.isDefinedAt(2) == true

divide10.isDefinedAt(3) == false

divide10(2) == 5

divide10(3) == throw new MatchError ***//!!!!!***

### **Метод .collect**

Больше всего мы любим применять эти частичные функции в разных методах, которые их принимают.

Например, метод .collect у всех коллекций. Он возьмет вашу частичную функцию, пробежится по всем вашим элементом коллекции, и для каждого элемента проверит, подходит ли он для частичной функции, и если есть, получит результат, и вернет коллекцию из этих результататов.

Например, можем взять все числа от одного до десяти и получить коллекцию всех результатов деления 10 на эти числа без остатка.

*List*.range(1, 10) *// List(1, 2,..., 10)*

*List*.range(1, 11).collect(divide10)

*//List(10, 5, 2, 1)*

Мы узнали, что такое частичные функции, а в следующих разделах узнаем о разных полезных типах данных в Scala, которые мы постоянно используем.

## **5.4 Кортежи**

Всем привет! Мы с вами познакомились с частичными функциями и пришло время узнать про кортежи.

Кортежи - это предопределенные типы представляющий собой последовательность и значение фиксированной длины.

### **Объявление кортежа**

Если мы хотим упаковать несколько значений в такую последовательность, мы просто перечисляемых внутри скобок разделенными запятой. Тип для такого кортежа будет выглядеть точно также - это несколько типов разделенных запятой в круглых скобках.

val tuple = (value1, value2, ..., valueN)

type TupleType = (Type1, Type2, ..., TypeN)

Кортежи очень полезны для того чтобы возвращать, из функций, например, несколько значений сразу. К примеру, мы хотим вернуть частное и остаток от деления разом, можем определить такую функцию. Или, например, можем определить функцию, которая одновременно возвращает первый, последний символ строчки и ее длину. Как видите, типы разных элементов кортежа не должны обязательно совпадать.

def divMod(x: Int, y: Int): (Int, Int) =

(x / y, x % y)

def firstLastAndCount(line: String): (Char, Char, Int)

(line(0), line.last, line.length)

### **Распаковка кортежа**

Если мы хотим вытащить элементы обратно из кортежа, можем сделать это точно также, перечислив результирующее значение в скобки.

val (val1 : Type1, val2: Type2, ..., valN) = tupl

Например, мы можем объявить сразу несколько переменных, которые мы можем сразу присвоить из какого-нибудь выражения типа кортежа. Или мы можем использовать наши кортежи в pattern matching: точно так же после case пишем круглые скобки и перечисляем переменные, в которые мы хотели бы сложить отдельные члены последовательности кортежа, так же, как, например, с case-классами.

val (div, mod) = divMod(17, 5)

val (first, last, count) = firstLastAndCount("Scala")

def showDiv(x: Int, y: Int) =

divMod(x, y) match {

case (d, r) => s"**$**x is **$**d \* **$**y + **$**r"

}

### **Обращение к элементу**

Если мы не хотим распаковывать, можем воспользоваться специальными полями, которые определены в каждом кортеже: "\_1" - это первое поле, "\_2" - второе поле и так далее.

tuple.\_1

tuple.\_2

...

tuple.\_N

val dm = divMod(17, 5)

val div = dm.\_1

val mod = dm.\_2

### **Пары элементов**

Самым частым используем типом кортежа, является кортеж из двух элементов - пара. Для него даже есть специальный синтаксис, а точнее метод расширения, такая вот тонкая стрелочка "->".

val nameAndPopulation = "Moscow" -> 12e6

val pairs = *List*(1 -> "one", 2 -> "two", 3 -> "three"

Для него у нас есть специальный метод "swap", который может поменять элементы местами. Как видите, типы тоже меняются.

val intAndString: (Int, String) = 1 -> "one"

val stringAndInt: (String, Int) = intAndString.swap

В общем-то и все что мы хотели рассказать о кортежах, попробуем с вами на практике.

### **Практика**

Мы с вами познакомились с кортежами, давайте попробуем применить их на практике. Попробуем реализовать алгоритм Евклида. Напомню что это такое. Представим, что у нас есть пара натуральных чисел "a" и "b". Нам нужно найти такие числа "x" и "y", такие что "а \* x + b \* y" равно какому-то числу "d", где "d" - это наибольший общий делитель "a" и "b". Например, это взаимно простые числа, справа будет единица.

a \* x + b \* y = d

Давайте попробуем написать функцию, которая находит эти "x" и "y". Для начала мы должны понять, что наша функция должна возвращать пару. В качестве аргументов у нее будут два целых числа: "а" целое, "b" целое, а в качестве типа возвращаемого значения тоже пара целых чисел, это можем выразить в виде такого кортежа.

def euclid(a: Int, b: Int): (Int, Int) =

Итак, несколько специальных случаев: мы бы хотели в алгоритме Евклида, чтобы первое число было всегда больше второго, поэтому если так получится, что "b" больше "а", мы выполним нашу функцию для чисел в обратном порядке и поменяем местами коэффициенты.

def euclid(a: Int, b: Int): (Int, Int) =

if (b > a) euclid(b, a).swap

Дальше, что если мы уже все нашли, это значит, что одно из наших чисел равно нулю. Наибольший общий делитель любого числа с нулем - это само это число. Если вдруг так получилось что одно из наших чисел равно нулю, а так рано или поздно должно получиться, и это должно быть число "b", так как оно у нас из двух наименьшее, мы должны вернуть очевидно пару коэффициентов. Наибольшим общим делителем у нас будет теперь число "а", поэтому его мы должны умножить на единицу, а число "b" мы можем умножить на что угодно, например, на ноль.

if (b > a) euclid(b, a).swap

else if (b == 0) (1, 0)

И наконец, если ни одно из этих условий не выполнилось, и у нас есть просто пара натуральных чисел, мы рассматриваем собственно основной рекурсивный случай. Первое, что нам нужно вычислить - это частное и остаток от деления "а" на "b" согласно алгоритма Евклида. Частное равно разделить на "b", остаток от деления равно "а % b".

if (b > a) euclid(b, a).swap

else if (b == 0) (1, 0)

else {

val d = a / b

val r = a % b

}

Теперь мы можем запустить рекурсивный алгоритм Евклида для этих двух чисел. Вызываем функцию "euclid" от пары чисел "b" и "r". Имея теперь такие "b" и "r" нам необходимо воспользоваться результатами запуска этой функции. Функция возвращает пару значений, поэтому нам необходимо их распаковать. Итак, какой-то "x, y", полученый из этих функций, у нас будут равны собственно коэффициентам, которые нам необходимо добавить к "b" и "r", так чтобы сумма этих чисел с этими коэффициентами равнялась наибольшему общему делителю.

else {

val d = a / b

val r = a % b

val (x, y) = euclid(b, r)

}

Следующим шагом мы собственно можем уже вернуть результат, коэффициентом для "а" у нас в этом случае будет просто "y", коэффициентом для "b" у нас будет "x - d \* y", вот такая простая формула.

def euclid(a: Int, b: Int): (Int, Int) =

if (b > a) euclid(b, a).swap

else if (b == 0) (1, 0)

else {

val d = a / b

val r = a % b

val (x, y) = euclid(b, r)

(y, x - d \* y)

}

Попробуем выполнить эту функцию. Допустим для чисел 7 и 4, она вернула -1 и 2. И действительно -1 \* 7 + 2 \* 4 будет равняться единице, то есть наибольшему общему делителю этих двух чисел.

euclid(7, 4)

-1 \* 7 + 2 \* 4

val res0: (Int, Int) = (-1,2)

val res1: Int = 1

Давайте что-нибудь посложнее, например, 234 и 168, будут коэффициенты -5 и 7. Давайте попробуем вычислить так ли это, берем число 234 умножаем его на первый элемент нашей пары и добавляем число 168 умноженное на второй элемент пары. Результат число 6. И 6 - это наибольший общий делитель чисел 234 и 168.

val pair = euclid(234, 168)

234 \* pair.\_1 + 168 \* pair.\_2

val pair: (Int, Int) = (-5,7)

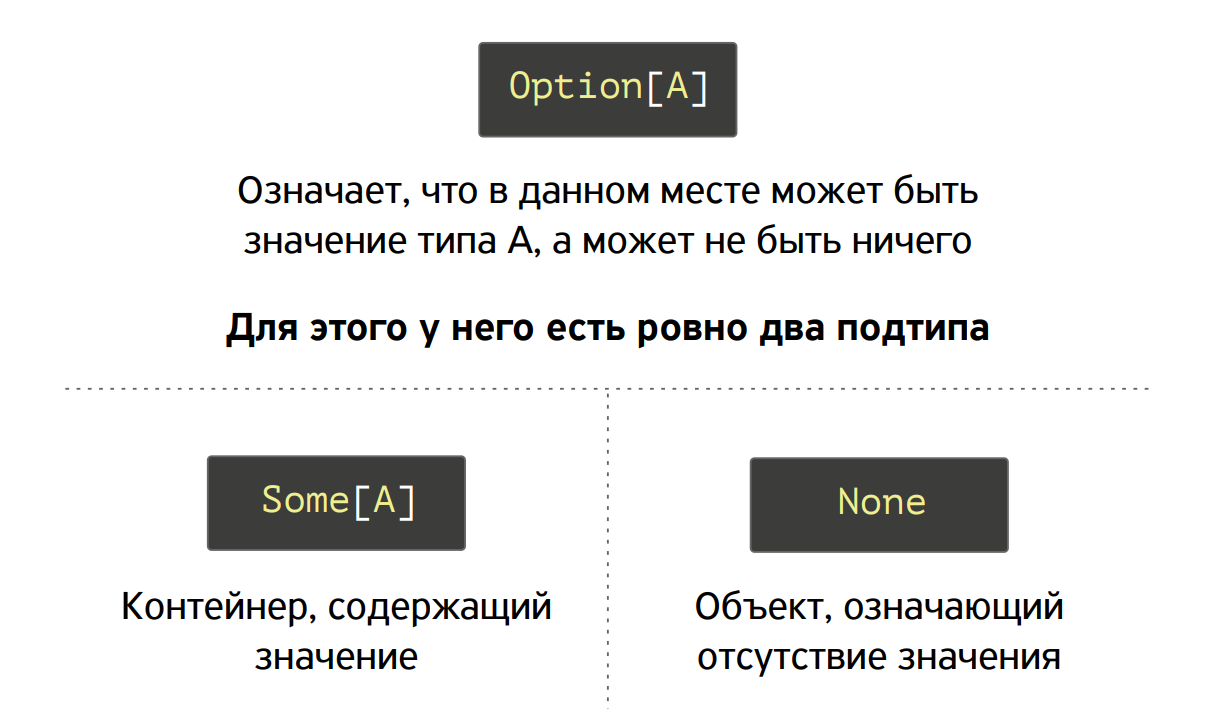
val res2: Int = 6

Вот так мы можем применять как минимум кортежи из двух чисел - пары, в следующем разделе мы поговорим про опциональные значения.

## **5.5 Опциональные значения**

На практике мы с вами познакомились с кортежами, теперь мы узнаем про опциональные значения.

Опциональные значение в Scala обозначаются таким типом Option[A] в квадратных скобках от какого-то другого типа. Это означает, что значение такого типа может содержать, а может не содержать значение типа "A". Для этого у него есть ровно два подтипа тип Some[A] - это просто контейнер, который гарантированно содержит значение типа "A", и тип None, то есть некий подтип который не содержит ничего.



Зачем вообще нужны Option[A]?

Во многих других языках есть пустые ссылки. Они очень часто используются для того, чтобы в каком-то методе, который не всегда может вернуть значение, возвратить что-то, когда метод не может вернуть значение. Например, если у вас есть какой-нибудь словарь и в нем не нашлось вашего ключа, вы возвращаете пустую ссылку.

Пустые ссылки очень проблемны, потому что глядя на метод, вы не можете определить гарантированно он вам вернет не пустую ссылку или все-таки вам нужно проверять пустая она или нет. Чтобы решить эту проблему в Scala существует специальный тип Option[A], который используется везде где вы подразумеваете, что у вас может значения и не быть.

* Option[A] используется для решения проблем **null reference**
* Всюду, где для обозначения **возможного отсутствия значения** может использоваться пустая ссылка, в scala используется Option[A]
* В отличие от пустой ссылки, Option[A] заставляет потребителя **явно** задуматься, что делать, если значения нет

Самый классический пример - деление на ноль. Есть 2 числа, мы хотим написать функцию, которая делит на 0, но если знаменатель равен нулю, естественно она не может вернуть никакого значения, поэтому мы ее определяем как функцию опционально возвращающую целое число.

def divide(x: Int, y: Int): Option[Int] =

if(y == 0) None else *Some*(x / y)

Мы можем проверять наши варианты прямо в pattern matching, если есть вариант мы распаковываем его в какую-нибудь новую переменную, которую можем использовать справа от выражения, а если нет, нам нужно подумать, что делать в случае если нет.

def showDivide(x: Int, y: Int): String =

divide(x, y) match {

case Some(d) => s"**$**x = **$**d \* **$**y"

case None => "null division"

}

### **Замена пустого значения**

Для того, чтобы не пользоваться pattern matching, у типа Option есть множество встроенных методов. Например, метод getOrElse, куда вы передаете какое-то значение, и оно у вас вычисляется только в том случае, если значения нет.

Или, например, метод orElse, похожий по сути смысл, но в нему передаете другое опциональное значение, если первого нет, то вместо него он пробует использует второе. Естественно, если оба значения пустые, результат тоже будет пустым NaN.

*// на значение по умолчанию: getOrElse*

divide(7, 0).getOrElse(1)

*// на другое опциональное значение: orElse*

divide(7, 0).orElse(divide(7, 2)

### **Преобразование**

Вы можете преобразовывать значение внутри таких Option не распаковывая их, не используя pattern matching каждый раз.

Для такого преобразования нас есть функция "map", если в вашем Option есть значение, она его преобразует, если нет - оставит пустым.

И функция "flatMap" точно так же, только она использует функцию, которая возвращает опциональное значение. Если значение есть, она произведет вычисление, запустит вашу функцию, которую вы ей передали, и если эта функция тоже вернула не пустое значение, результат в общем случае будет не пустым.

*// обычной функцией: map*

divide(7, 4).map(x => x + 6)

*// функцией, возвращающей опциональное значение:*

*// flatMap*

divide(17, 3).flatMap(x => divide(x, 3))

### **Фильтрация**

И наконец, фильтрация - самая простая функция "filter", где если у вас значение есть, вы можете проверить соответствует она в вашем условию, если она не соответствует значение становится пустым.

И такая комбинация функции "map" и "filter", функция "collect", мы уже и и вспоминали, когда говорили о частичных функциях, и здесь частичные функции очень классно заходят, можем написать "collect", все значения, которые нам подходят, и во что их собственно преобразить.

*// предикатом: filter*

divide(7, 4).filter(x => x > 2)

*// частичной функцией: collect*

divide(17, 3).collect{

case x if x > 4 => x + 4

}

С помощью этих методов, мы можем переписать тот пример с patter matching, который у нас генерировал строчку из возможно пустого значения. Вот с помощью этих методов, сначала использовали "map", а потом, если ничего не получилось, взяли какое-то значение по умолчанию.

def divide(x: Int, y: Int): Option[Int] =

if(y == 0) None else *Some*(x / y)

def showDivide(x: Int, y: Int): String =

divide(x, y)

.map(d => s"**$**x = **$**d \* **$**y")

.getOrElse("null division")

Мы познакомились с опциональными значениями, попробуем применить их на практике.

### **Практика**

Мы с вами познакомились с опциональными значениями и давайте попробуем применить их на практике.

Попробуем написать такую интересную функцию. Представим, что у нас есть строчка, например, такая "scala + [stepik] = love", и представим что в этой строчке какую-то последовательность символов мы взяли в квадратные скобки.

val string = "scala + [stepik] = love"

Мы могли бы воспользоваться встроенной функцией "indexOf" у строчки. Например, найти какую-нибудь из скобок. Как видите, она у нас возвращает индекс начиная с нуля того самого символа, который мы ищем.

Однако, если мы передадим ей какой-нибудь символ, которого там нет, функция возвращает число "-1", что означает "я не смогла найти".

string.indexOf("[")

string.indexOf("7")

val res0: Int = 8

val res1: Int = -1

Давайте сделаем эту функцию чуть более вменяемой, и объявим функцию, которая будет работать точно также, но в случае отсутствующего символа, возвращает нам None, то есть явно говорит о том, что она могла не найти символ.

Итак у нас есть некая строчка и некий шаблон, который мы в ней ищем, и возвращать функция будет опционально число, то есть индекс элемента. Мы запускаем "indexOf" и как видим наш тип не совпадает, потому что возвращает Int, а нам нужно вернуть Option[Int].

def indexOf(s: String, pattern: String): Option[Int] =

s.indexOf(pattern)

type mismatch;

found : Int

required: Option[Int]

s.indexOf(pattern)

Для того, чтобы чтобы превратить Int в Option[Int], мы можем передать его в качестве аргумента объекту Option. И что интересно этот кейс также будет обрабатывать значение, если вы случайно передали аргумент "null", результат будет Option.

def indexOf(s: String, pattern: String): Option[Int] =

*Option*(s.indexOf(pattern))

Давайте попробуем нашу функцию, например, indexOf, string, “[” у нас равнo Some(8), а indexOf, какое-то число цифра 7 у нас получился Some(-1).

indexOf(string, "[")

indexOf(string, "7")

val res2: Option[Int] = Some(8)

val res3: Option[Int] = Some(-1)

Нам, естественно, нужно обработать этот случай, каким-то образом исключить значение s "-1" из рассмотрения. Для этого мы вызываем функцию "filter", в которой убеждаемся, что полученное число больше или равно нулю. И вот мы получили None в качестве результата на indexOf несуществующей символ.

def indexOf(s: String, pattern: String): Option[Int] =

*Option*(s.indexOf(pattern)).filter(\_ >= 0)

indexOf(string, "[")

indexOf(string, "7")

val res2: Option[Int] = Some(8)

val res3: Option[Int] = None

Давайте теперь напишем функцию, которая нам будет возвращать пару индексов, открывающей скобки "[" и закрывающей квадратной скобки "]". Естественно пара индексов у нас выражается типом "tuple". Функция будет называться "brackets", в ней передаем какую-то строку и возвращаем опционально пару индексов.

def brackets(s: String): Option[(Int, Int)] =

Сначала мы получаем первый индекс запускаем функцию indexOf, в нашей строчке мы ищем открывающую квадратную скобку "[". Мы получили Option от одного Int.

def brackets(s: String): Option[(Int, Int)] =

**indexOf(s, pattern = "[")**

Теперь мы можем воспользоваться методом "flatMap" и предположить, что мы нашли-таки индекс открывающей скобки, назовем его "opening".

def brackets(s: String): Option[(Int, Int)] =

indexOf(s, pattern = "[")**.flatMap{ opening =>**

}

Запускаем функцию indexOf еще раз. Теперь ищем закрывающую квадратную скобку.

def brackets(s: String): Option[(Int, Int)] =

indexOf(s, pattern = "[").flatMap{ opening =>

**indexOf(s, pattern = "]")**

}

Когда мы получили и открывающую и закрывающую, мы можем в качестве результата, "map" от результата, назовем его "closing", превратить наконец в пару, открывающий и закрывающий.

def brackets(s: String): Option[(Int, Int)] =

indexOf(s, pattern = "[").flatMap{ opening =>

indexOf(s, pattern = "]")

**.map(closing => (opening, closing))**

}

Давайте попробуем запустить нашу функцию. Итак, brackets от нашей исходной строчки, вот она, наша пара.

brackets(string)

val res4: Option[(Int, Int)] = Some((8,15))

А если мы найдем, например, в какой-то более странной строчке, например, у нас идет какой-то набор символов, дальше закрывающая квадратная скобка, дальше опять набор символов, открывающая и дальше снова. Окажется, что на этот случай наша функция тоже сработает, она вернет пару индексов, где первый индекс больше второго.

brackets("asljfa ] jfkaldjk [ kdjkfla")

val res5: Option[(Int, Int)] = Some((18,7))

Мы будем рассматривать подстрочку с 18 по 7 символ. Нас это не очень устраивает, поэтому мы можем точно также добавить ограничивающее условие, что этот "closing" должен быть обязательно больше, чем индекс открывающей.

def brackets(s: String): Option[(Int, Int)] =

indexOf(s, pattern = "[").flatMap{ opening =>

indexOf(s, pattern = "]")

**.filter(\_ > opening)**

.map(closing => (opening, closing))

}

Тогда во втором случае мы получили None.

brackets("asljfa ] jfkaldjk [ kdjkfla")

val res5: Option[(Int, Int)] = None

Теперь можем реализовать функцию "cutBrackets", которая будет искать строчку, заключенную в квадратные скобки и возвращать ее. Она будет возвращать естественно Option от String.

def cutBrackets(s: String): Option[String] =

И будет работать так. Представим что мы нашли пару чисел.

def cutBrackets(s: String): Option[String] =

**brackets(s)**

Найдя пару чисел, мы можем извлечь эту пару чисел. Для того чтобы извлечь, прямо внутри "map", мы можем воспользоваться pattern matching, зная что мы здесь получаем пару из целых чисел, мы просто называем их "opening" и "closing", то есть автоматически сразу распаковываем.

def cutBrackets(s: String): Option[String] =

brackets(s)**.map{ case (opening, closing) =>**

**}**

Распаковав их, берем просто подстрочку нашей исходной строчки, начиная со следующего символа после открывающей квадратной скобки и заканчивая, так как собственно не включающийся, нашим "closing" символом.

def cutBrackets(s: String): Option[String] =

brackets(s).map{ case (opening, closing) =>

**s.substring(opening + 1, closing)**

}

Попробуем запустить нашу функцию от "string", и получаем то слово, которое мы искали.

cutBrackets(string)

val res6: Option[String] = Some(stepik)

Если мы уберем отсюда, например, открывающую квадратную скобку, видим, что мы получаем None.

val string = "scala + stepik] = love"

val res6: Option[String] = None

Возвращаем на другое место, мы снова получаем нашу строчку. Убираем закрывающую - снова получаем None. Переносим на другое место - снова получаем исходят из искомую строчку.

val string = "[scala + stepik] = love"

val string = "[scala + stepik = love"

val string = "[scala] + stepik = love"

val res6: Option[String] = Some(scala + stepik)  
val res6: Option[String] = None

val res6: Option[String] = Some(scala)

Вот так замечательно можно работать с опциональными значениями, а теперь давайте попробуем разобраться с интересным типом под названием "исключающее или" либо "Either".

## **5.6 Either[A, B] - исключающее или**

На практике мы с вами познакомились с опциональными значениями и сейчас мы узнаем про специальный тип, означающий "исключающее или", по английски это будет "Either".

Если у вас есть "Either" с тайп параметрами "A" и "B" - это означает что у вас есть либо "A", либо "B". Оно имеет два варианта, два подтипа, "Left" - означающий, что у вас есть значение типа "A", и "Right" - означающий, что у вас есть значение типа "B".



Сгенерировать значение типа, например, Either[Double, String], то есть Double, либо строка, вы можете имея и Double и строку. Если у вас есть значение типа Double - вы используете конструктор "Left", если у вас есть значение строка - вы используете конструктор "Right".

val numOrStr1: Either[Double, String] = *Left*(2.12)

val numOrStr2: Either[Double, String] = *Right*("Scala")

Выяснить какое у вас сейчас конкретно значение, вы можете с помощью pattern matching и конечно же Scala догадается, что если речь идет про Either от Double и String, то если вы встречаете Left, то речь идет именно о Double, а если возвращаете Right, то речь идет именно о строке.

def info(numOrStr: Either[Double, String]): String =

numOrStr match {

case *Left*(num) => s"number **$**num"

case *Right*(str) => s"string **$**str"

}

info(numOrStr1) *// number 2.12*

info(numOrStr2) *// string Scala*

### **Обработка ошибок**

Мы очень любим использовать значения типа Either для того чтобы обрабатывать ошибки, не используя конструкции throw catch. Это очень похоже на то как мы используем Option, но только в отличие от Option, если у вас нет значения - вместо него какое-то конкретное описание, почему у вас этого значения нет.

def sqrt(x: Double): Either[String, Double] =

if (x < 0) *Left*("negative number")

else *Right*(Math.*sqrt*(x))

И точно также как в случае с Option, мы можем обрабатывать эти значения не делая pattern matching.

Например, если у вас есть Either и вы хотели бы в случае ошибки, то есть в случае Left заменить его на какое-то более приемлемое значение, вы можете использовать getOrElse.

Или если вы хотите, как в случае с Option отфильтровать, заменить Right на Left в случае, если значение внутри Right не подходит под ваши условия, вам нужно сопроводить его дополнительным значением для ошибки в этом случае, это делается с помощью функции filterOrElse.

*// Замена на значение по умолчанию: getOrElse*

sqrt(7).getOrElse(0)

*// Фильтрация предикатом: filterOrElse*

sqrt(3).filterOrElse(\_ > 2, "too small")

### 

### **Преобразование**

И точно так же можем преобразовывать значение типа Either. Если у вас есть обычная функция вы можете воспользоваться методом "map", который заменит Right одного типа на Right другого типа.

Если ваша функция возвращает точно также возможно ошибочное значение, то есть еще один Either чей тип ошибки, левый тип точно такой же, либо возможно надтип, можете воспользоваться методом "flatMap", который вернет Right только в том случае, если у вас изначально был Right и ваша функция тоже вернула Right.

*// обычной функцией: map*

sqrt(7).map(\_.toString)

*// функцией, возвращающей Either с таким же*

*// "левым" типом: flatMap*

sqrt(7).flatMap(x => sqrt(x))

Мы познакомились с Either, попробуем применить его на практике.

### **Практика**

Мы с вами познакомились с Either или "исключающее или" и давайте попробуем применить его на практике. Представим что мы хотим вычислить какую-нибудь хитрую арифметическую функцию. Например, мы хотим взять два числа с плавающей точкой и хотим вычислить вроде того: сумма квадратных корней "x" и "y" деленная на корень квадратный из суммы. Например для четырех и пяти это будет что то такое.

import java.lang.Math.*sqrt*

def fun(x: Double, y: Double): Double =

(*sqrt*(x) + *sqrt*(y)) / *sqrt*(x + y)

fun(4, 5)

val res0: Double = 1.4120226591665965

Какие здесь могут возникнуть проблемы? У нас как минимум две функции, которые работают не всегда. Например, если одно из чисел отрицательное, мы квадратный корень из него не можем вычислить. С другой стороны, если результирующая сумма равна нулю, то можем вычислить из нее квадратный корень, но результат получается бесконечность. И в том, и другом случае, как видите, мы получаем некое число NaN (Not a Number) - это число вроде такого опционального типа Doble, но мы на самом деле не знаем, где и какая ошибка у нас произошла.

*fun(4, -5)*

*fun(0, 0)*

val res1: Double = NaN

val res2: Double = NaN

Поэтому нам бы хотелось иметь более подробное описание, что за ошибка у нас в каком месте произошла. Для этого можем воспользоваться типом Either. Для начала, нам нужно добавить несколько функций, которые будут возвращать Either для базовых функций.

Итак, самая первая функция sqrtE, который возможно возвращает ошибку. Мы передаем ей Double, возвращаем Either[String, Double]. И здесь проверяем, если наш "x" меньше 0 - мы возвращаем ошибку "x < 0 !", если все хорошо - возвращаем квадратный корень.

def sqrtE(x: Double): Either[String, Double] =

if (x < 0) *Left*(s"$x < 0 !") else *Right*(*sqrt*(x))

Cледом нам нужно определить операцию деления точно так же с ошибкой. Итак "divE" от двух чисел с плавающей точкой, она у нас будет определена вот так. Она тоже будет возвращать Either[String, Double] и будет проверять если одно число, знаменатель, равен нулю тогда мы возвращаем ошибку "zero division", иначе будет спокойно возвращать результат деления "x" на "y".

def divE(x: Double, y: Double):

Either[String, Double] =

if (y == 0) *Left*(s"zero division !") else *Right*(x / y)

Имея две таких функции, мы можем наконец объявить нашу супер функцию, которая будет использовать весь механизм обработки ошибок через Either. Итак, у нас есть одно число с плавающей точкой, второе число с плавающей точкой, и мы должны вернуть возможно ошибку, а возможно число с плавающей точкой. Сначала нам нужно вычислить квадратный корень от "x", результат мы назовем "sx".

def funE(x: Double, y: Double):

Either[String, Double] =

**sqrtE(x).flatMap{sx =>**

}

Затем мы вычисляем квадратный корень из "y", результат мы назовем "sy".

sqrtE(x).flatMap { sx =>

**sqrtE(y).flatMap { sy =>**

}

}

После того, как мы вычислили два таких квадратных корня, нам нужно вычислить квадратный корень из "x + y", назовем этот результат "sxy".

sqrtE(x).flatMap { sx =>

sqrtE(y).flatMap { sy =>

**sqrtE(x + y).flatMap { sxy =>**

}

}

}

Когда у нас есть 2 таких переменные, мы наконец можем попробовать разделить одно на другое, и вернуть деление "sx + sy" на "sxy". Вот такая интересная функция.

def funE(x: Double, y: Double):

Either[String, Double] =

sqrtE(x).flatMap { sx =>

sqrtE(y).flatMap { sy =>

sqrtE(x + y).flatMap { sxy =>

**divE(sx + sy, sxy)**

}

}

}

Теперь, если мы попробуем запустить ее для 4 и 5, мы получим тот же самый результат, как если мы запустили обычную функцию.

funE(4, 5)

val res3: Either[String,Double] =

Right(1.4120226591665965)

Если мы одно из чисел будет отрицательным, например, -2 и 5, мы говорим что -2 меньше нуля. Если мы одно из чисел будет отрицательным, например, -2 и 5, мы говорим что -2 меньше нуля. Если, например, второе число будет меньше нуля, говорит, что -4 меньше нуля.

funE(-2, 5)

funE(3, -4)

val res4: Either[String,Double] = Left(-2.0 < 0 !)  
val res5: Either[String,Double] = Left(-4.0 < 0 !)

Сумма чисел у нас вряд ли может быть отрицательной, если оба числа положительные, но зато они оба могут быть равны нулю. И если вы перед этим сюда два нуля, мы получим ошибку "zero division!".

funE(0, 0)

val res6: Either[String,Double] = Left(zero division !)

Вот таким образом мы можем сделать механизм альтернативный обычной обработки исключений с throw catch, но только с заранее известным, возможно даже специально изготовленным для этих вычислений типом ошибки.

Таким образом мы можем работать с Either, но в следующем разделе, мы узнаем про коллекции в Scala.

# Коллекции

## **6.1 Коллекции. Часть 1**

На практике мы с вами познакомились с Either и наконец добрались до коллекций. В стандарте библиотеки Scala множество их разновидностей.

### **Виды коллекций**

Есть много неизменяемых коллекций, которые лежат в пакете scala.collection.immutable.

import scala.collection.immutable.\_

List[+A], Vector[+A], Stream[+A], Set[+A], Map[K, +V]

Есть набор изменяемых коллекций, которые лежат в пакете scala.collection.mutable.

import scala.collection.mutable.\_

Buffer[A], Set[A], Map[A, B], Builder[-E, +С]

Есть их общие надтипы, которые в пакете scala.collection.

import scala.collection.\_

Seq[+A], Set[+A], Map[K, +V], Iterator[+A]

И есть несколько специализированных коллекций, таких как массив и строка.

Array[A], String

### **Массивы**

Массив - это очень эффективный, но довольно низкоуровневый набор элементов фиксированной длины. Он эффективный, в том числе, и потому, что у него есть и специализированные версии для примитивных типов, таких как Int, Long, Double и Boolean.

Array[A]

*// Очень эффективный, но низкоуровневый*

*// Фиксированного размера*

*// Специальные версии для примитивов*

*// (Int, Long, Double, Boolean)*

Когда мы хотим работать с массивом, мы просто выделяем новый массив какой-то фиксированной длины, заранее известной, и дальше можно читать его, можем складывать него новые элементы.

val ints = *Array*(1, 2, 3, 5)

ints(2) *// 3*

ints(2) = 6

ints(2) *// 6*

ints(4) = 6 *// error !!!*

### **Строки**

Строки это тоже массив символов, но он неизменяемый. Каждый раз, когда мы хотим сгенерировать новую строку, мы заново выделяем для нее место и заполняем ее новыми символами.

String

*// Массивы символов*

*// Неизменяемые*

*// Любое изменение выделяет новую строку*

Например, у нас есть какие-то уже строки и мы можем конкатенировать их с помощью операции +.

val language = "Scala"

val platform = "Stepik"

val course = language + " " + platform

*// val course: String = Scala Stepik*

Также предполагая, что операция + может быть не всегда эффективной в том плане, что если у вас есть много конкатенаций подряд, каждый раз для каждого промежуточного действия у вас будет выделяться новая строка. Поэтому мы используем специальный синтаксис интерполяции, перед строчкой можно поставить букву "s" и тогда внутри строчки, каждый раз, когда мы ссылаемся на переменную мы ставим перед именем "$", и значения этих переменных вставятся в нужные места внутри строчки. Это будет гораздо эффективнее, чем конкатенировать их вручную.

val cource1 = s"**$**language **$**platform"

*// val course1: String = Scala Stepik*

B так же можем взять какой-нибудь символ просто по индексу, вызвав его через скобочки.

*val char: Char = course(3) // val char: Char = l*

### 

### **Изменяемые коллекции**

Изменяемые коллекции - это довольно привычная, для людей пришедших из других языков, но слегка опасная форма коллекций. Их состояние может меняться со временем и они более эффективны, чем неизменяемые коллекции, если у вас идет большое количество операций подряд. К несчастью копирование их уже не очень эффективно, для того чтобы вам скопировать изменяемую коллекцию, вам нужно выделить новое место для всех элементов, и скопировать все их в новую коллекцию.

import scala.collection.mutable.\_

*// Состояние может меняться со временем*

*// Эффективны для большого количества операций*

*// Копирование неэффективно*

У нас есть такие коллекции, как: Buffer - это такой массив, в который можно добрасывать элементы с конца. Set - это коллекция уникальных элементов, если мы попробуем положить в mutable.Set элемент, в котором он уже есть Set не изменится. Map - ассоциативный массив, который сопоставляет какие-то ключи каким-то значением, если мы попробуем положить туда значение с ключом который уже есть в нашем ассоциативном массиве, значение для этого ключа изменится. Builder - это специальный промежуточный тип, который нужен для того чтобы производить коллекции новых типов.

| **Buffer[A]** | **Саморастущий массив** |
| --- | --- |
| ***Set*[A]** | **Набор уникальных элементов** |
| ***Map*[K, V]** | **Ассоциативный массив**  **"ключ-значение"** |
| **Builder[E, Coll]** | **Промежуточный накопитель**  **для построения коллекции** |

Работаем мы с изменяемыми коллекциями приблизительно так: создаем какую-то коллекцию и дальше вызываем одну за другой несколько операций мутации, а потом как-нибудь собираем наши элементы, вычисляем какую-то информацию.

*import scala.collection.mutable.Buffer*

*val strings = Buffer[String]()*

*strings += "scala"*

*strings += "+"*

*strings += "stepik"*

*strings += "="*

*strings += "love"*

*strings.mkString(" ")*

### **Неизменяемые коллекции**

Неизменяемые коллекции - это наша любимая форма коллекций. Они гораздо удобнее во многих отношениях. Их состояние неизменно. Когда вы генерирует новую коллекцию, вы всегда копируете, но не всегда копируете выделяя пространство для всех элементов. Неизменяемые коллекции могут переиспользовать пространство, переиспользовать кусочки коллекций, из которых вы создаете новый элемент. Так как они неизменяемые, вы можете вычислить от них хэш, таким образом их можно использовать в качестве ключа, например, в ассоциативном массиве, или можно складывать их в Set.

import scala.collection.immutable.\_

*// Состояние неизменно*

*// Наиболее удобны*

*// Достаточно эффективное создание копий при изменении*

*// Hashable, могут храниться в Set,*

*// выступать ключами в Map*

*// Коварианты*

Самые частые виды неизменяемых коллекции это: Односвязный список List - это такой список, который вы всегда можете изменить, добавив один элемент в начало, тогда ваш имеющийся список станет хвостом нового списка. Это Stream - очень похож на односвязный список, но его хвост вычисляется лениво, то есть только в момент, когда вам он действительно нужен. Именно потому, что он вычисляется момент, когда он действительно нужен, он может одновременно не держать в памяти все элементы и может потенциально даже быть бесконечным. Это специальный тип Vector - это такой индексированный список, в отличие от коллекции List и Stream, вы можете достаточно эффективно добавить элемент даже в конец этого вектора, и достаточно быстро получить элемент по любому индексу. Поэтому векторы часто используют когда вам нужна неизменяемая версия массива. Это неизменяемый Set - коллекция уникальных элементов, и это неизменяемый Map - ассоциативный массив "ключ-значение".

| ***List*[A]** | **Связный конечный список**  Легко добавить элемент в начало |
| --- | --- |
| ***Stream*[A]**  ***LazyList[A]*** | **Ленивый связный список, возможно бесконечный**  Легко добавить элемент в начало |
| ***Vector*[A]** | **Индексированный список**  Легко получить элемент по индексу, добавить элемент в начало или конец |
| ***Set*[A]** | **Набор уникальных элементов** |
| ***Map[K, V]*** | **Ассоциативный массив "ключ-значение"** |

Работаем с неизменяемыми коллекциями мы приблизительно так: Мы имеем какую-нибудь исходную коллекцию, и дальше мы начинаем генерировать коллекции на ее основе. Например, во второй строчке, мы добавляем в наш вектор элементы сначала, а в третьей строчке добавляем элементы с конца, и потом с нашим результатом мы можем что-нибудь сделать, собрать все элементы вместе или вычислить что-нибудь на их основе.

val initial = *Vector*[String]("stepik")

val mid = "scala" +: "+" +: initial

val strings = mid :+ "=" :+ "love"

strings.mkString(" ")

Мы с вами изучили коллекции и в следующей части мы узнаем о различных операциях, которые позволят нам делать разные интересные вещи над ними.

### **Практика**

Всем привет! Мы с вами познакомились с коллекциями, давайте попробуем сделать что-либо с ними.

Когда мы думаем, что можно сделать с коллекцией в первую очередь, сразу приходит на ум классическое упражнение сортировки коллекции, то есть упорядочить элементы коллекции так, чтобы элемент слева всегда был меньше элемента справа. Представим, что у нас есть какой-то набор элементов. Создадим новую коллекцию типа List и засунем в нее первое, что придет в голову.

val list = *List*(2, 5, 7, 1, 4)

Самый простой способ отсортировать такие элементы вызвать метод "sorted". Этот метод уже делает все, что нам нужно, но нас это конечно не устраивает.

list.sorted

val res0: List[Int] = List(1, 2, 4, 5, 7)

Нас не устраивает два момента, во-первых наверное какая-то произвольная выдуманная нами коллекция может не всегда найти все проблемы в нашей реализации. Давайте поэтому давайте попробуем сгенерировать какой-нибудь список случайных элементов.

Для этого можем воспользоваться объектом Random из пакета scala.util. Для того чтобы создать какую-то коллекцию кого-то произвольного размера можем воспользоваться методом List.fill.

Первым параметром у него идет количество элементов. У него несколько реализаций, можно передавать одно целое число туда и у нас будет просто список, можно передать пару целых чисел и у нас будет список списков, ну и так далее. В данном случае, мы хотим сгенерировать какую-то коллекцию из произвольного числа элементов не больше ста. Для этого у нас в объекте Random, есть метод nextInt, куда можем передать, собственно то число, которое отраничит количество элементов.

Следующим параметром метода fill должно идти выражение, которым мы будем заполнять ячейку за ячейкой из нашего какого-то набора элементов. В данном случае, этот параметр является передаваемым по имени, то есть он будет вычисляться каждый раз, когда нужно будет сгенерировать произвольный элемент. Например, мы можем взять и нагенерить элементов не больше 1000.

Получаем вот такой список, как видите, все значения различны, то есть Scala не стала запоминать значение первого попавшегося случайного числа, которое мы передали сюда, а вычисляет значение каждый раз.

val randomList =

*List*.fill(Random.nextInt(100))(Random.nextInt(1000))

val randomList: List[Int] =

List(5, 318, 27, 720, 229, 572, 867, 945, 824, 770,

317, 661, 423, 99, 504, 191, 223, 855, 275, 126, 747,

146, 921, 17, 736)

Теперь давайте подумаем, как мы можем отсортировать наш список. Один из классических способов сортировки, достаточно эффективный и очень хорошо подходящий именно для односвязанных списков, является сортировка слиянием. Итак, давайте определим такой метод слияния.

Что такое сортировка слиянием? Сортировка слиянием - это сортировка, которая раз за разом выполняет операцию слияния двух списков. Если у вас есть два уже отсортированных списка или любых коллекций, вы можете сконструировать новую коллекцию, элементы в которой, будут тоже отсортированы.

Вы это можете сделать пройдясь один раз по всем элементам из первого и второго списка. Давайте попробуем это реализовать. Итак, у нас есть элементы "as" - первый список Int, элемент "sb" - второй список Int, и наша функция должна возвращать список, опять же, целых 32 битных чисел.

def merge(as: List[Int], bs: List[Int]): List[Int]

Как мы можем это сделать? У нас есть много вариантов, но мы хотим придумать хорошее эффективное решение и скорее всего нам для этого потребуется хвостовая рекурсия.

В принципе мы можем добавить аккумулятор и сделать нашу функцию хвострекурсивной. Тогда нам бы неплохо задать некое значение по умолчанию для нашего аккумулятора, будем начинать с пустого списка.

def merge(as: List[Int], bs: List[Int],

acc: List[Int] = *Nil*): List[Int]

Теперь нам необходимо проверить, допустим список "а", пустой он или нет? Будем делать pattern matching. Если мы нашли пустой список, можем, например, заматчить вот так "head :: tail".

def merge(as: List[Int], bs: List[Int],

acc: List[Int] = *Nil*): List[Int] =

as match {

case head *::* tail

}

Или лучше, так как нам потребуются обе головы и оба хвоста, возможные от "a", и от "b", мы можем назвать это "a" и "restA".

as match {

case a *::* restA =>

}

Дополнительно мы можем понять, что этот pattern matching с двойным двоеточием в принципе подходит для списков, но если позже мы хотим обобщить наш метод на произвольные сиквенсы, потребуется другой матчер, он выглядит так "+ :". Если двойное двоеточие подходит только для списков, то "+:" подходит для всех сиквенсов. Это очень важно помнить потому, что иногда и достаточно часто, случается ошибка, когда мы работаем с секвенсами, подразумеваем в голове списки, матчим их с помощью двойного двоеточия и объекта Nil, затем подставляем наш метод какую-нибудь другую коллекцию и он перестает работать.

as match {

case a *+:* restA =>

}

Что же делать если наша коллекция пустая и что же делать она не пустая и так если она пустая? Если она пустая, у нас есть какой-то набор накопленных элементов, мы потом подумаем, что с ним сделать. Если она не пустая, можно точно также заматчить теперь коллекцию "b". Если она пустая, мы подумаем что с этим сделать. Если она не пустая, опять же разрезаем ее на голову и хвост и думаем, что делать.

as match {

case List() => *???*

case a *+:* restA => bs match {

case List() => *???*

case b *+:* restB =>

}

}

У нас есть две головы. Мы можем взять голову одного списка, голову другого списка и сравнить. Если оказалось что левая голова меньше чем правая голова, как мы помним оба списка у нас заранее отсортированы, таким образом если левая голова у нас меньше, она меньше чем все элементы в массиве. Нам нужно добавить ее в аккумулятор в качестве очередного элемента.

Запускаем нашу рекурсию, продолжаем с оставшимися элементами из списка "a", продолжаем со всеми элементами из списка "b", потому что мы не знаем ничего, может быть в списке "a" у нас есть еще дополнительные элементы, которые все еще меньше, чем голова списка "b". И в наш аккумулятор добавляем новый элемент "a".

В противном случае запускаем очень похожую рекурсию. Добавляем сюда все элементы списка "a", оставшиеся элементы списка "b", и добавляем "b" в качестве очередного элемента.

as match {

case List() => *???*

case a *+:* restA => bs match {

case List() => *???*

case b *+:* restB =>

**if (a < b) merge(restA, bs, a :: acc)**

**else merge(as, restB, b :: acc)**

}

}

В получившейся коллекции мы будем добавлять элемент за элементом. Элементы, которые потенциально все больше и больше, сначала мы добавим самый маленький, потом чуть больше, потом чуть больше. Самый маленький элемент мы добавим в самом начале, затем в качестве очередной головы добавим побольше, затем в качестве очередной побольше, и так далее. В итоге, получится, что самый большой из рассматриваемых нами элементов будет в самом начале, а самый маленький в конце.

Поэтому, для того, чтобы нам получить снова отсортированную коллекцию элементов, нам нужно список перевернуть. Поэтому, в какой-нибудь из элементов, говорим, что мы хотим перевернуть, вызвать метод reverse у нашего аккумулятора, затем, в качестве хвоста, добавляем оставшиеся элементы из списка "b". Делаем тоже самое, но с элементами из списка "a".

def merge(as: List[Int], bs: List[Int],

acc: List[Int] = *Nil*): List[Int] =

as match {

case List() => **acc.reverse ++ bs**

case a *+:* restA => bs match {

case List() => **acc.reverse ++ as**

case b *+:* restB =>

if (a < b) merge(restA, bs, a :: acc)

else merge(as, restB, b :: acc)

}

}

Попробуем как работает наш метод. Запускаем берем две какие-нибудь отсортированные заранее, это важно, коллекции и смотрим что наш результат действительно производит отсортированное значение.

merge(*List*(2, 5, 6), *List*(1, 4, 9))

val res1: List[Int] = List(1, 2, 4, 5, 6, 9)

Теперь как же нам сгенерировать из этого алгоритм сортировки? Мы можем вызывать вот такое слияние много раз. Передаем список Int, хотим получить список Int.

Что мы с ним делаем? Проверяем, если наша коллекция, которую мы передали пустая, пустая коллекция всегда отсортирована, мы можем ее сразу вернуть. Можно писать такие pattern matching. Итак, если пустая, возвращаем пустой список. Если не пустая, и у нас есть ровно один элемент, опять же можем вернуть ту же самую коллекцию.

def mergeSort(as: List[Int]): List[Int] = as match {

case *Nil* => *Nil*

case a *:: Nil* => as

}

Вот эти две ветки pattern matching мы можем объединить в одну. Так как нам совершенно неважен здесь первый элемент, мы можем заменить его на прочерк и написать так: если коллекция пустая или состоит из одного элемента, возвращаем саму эту коллекцию.

def mergeSort(as: List[Int]): List[Int] = as match {

**case *Nil* | (\_ *:: Nil*) => as**

}

Что будет, если коллекция содержит как минимум два элемента? Тогда мы можем ее разрезать. Берем коллекцию, вызываем метод splitAt. splitAt разрежет ее на каком-то элементе, возьмет сколько-то элементов и вернет в качестве первой коллекции, а оставшиеся в качестве второй.

Для того чтобы наиболее равномерно разрезать наш список, и для того, чтобы наш алгоритм наиболее эффективно сходился, нам необходимо разрезать эту коллекцию ровно пополам. Поэтому мы можем взять длину, поделить пополам, и разрезать, собственно, в этом кусочке. Присваиваем это двум новым подколлекциям "left" и "right".

case *Nil* | (\_ *:: Nil*) => as

case \_ =>

val (left, right) = as.splitAt(as.length / 2)

Разрезали, присвоили. Теперь нам нужно отсортировать обе эти коллекции, убедиться в том, что элементы их будут идти в правильном порядке. Можем сконструировать новую коллекцию val leftSorted, для которой запускаем сортировку слиянием для левой коллекции, тоже делаем rightSorted, запускаем для правой коллекции. Получили 2 сортированные коллекции.

case \_ =>

val (left, right) = as.splitAt(as.length / 2)

**val leftSorted = mergeSort(left)**

**val rightSorted = mergeSort(right)**

Теперь мы можем их прямиком подать в качестве входного значения нашей функции "merge". Возвращаем merge(leftSorted, rightSorted).

def mergeSort(as: List[Int]): List[Int] = as match {

case *Nil* | (\_ *:: Nil*) => as

case \_ =>

val (left, right) = as.splitAt(as.length / 2)

val leftSorted = mergeSort(left)

val rightSorted = mergeSort(right)

**merge(leftSorted, rightSorted)**

}

Получили такую функцию. Давайте попробуем ее запустить. Передадим ей ну сначала наш простой списочек. Получили какой-то сортированный список можно убедиться, что он совпадает с результатом стандартного встроенного list.sorted. Получили true как видите.

mergeSort(list) == list.sorted

val res1: Boolean = true

Теперь можем взять нашу рандомную коллекцию, нашу коллекцию случайных элементов. Она у нас гораздо больше, элементы в ней гораздо разнообразней и проверим то же самое для неё. mergeSort(randomList) == randomList.sorted. Получили снова true. 100 элементов мы неплохо сортируем, а что если у нас будет 10 тысяч элементов? Как видите очень быстро получили true. В общем-то по крайней мере, убедились, что наш алгоритм имеет не квадратичную сложность, этого нам пока достаточно.

val randomList =

*List*.fill(Random.nextInt(10000))(Random.nextInt(1000))

mergeSort(randomList) == randomList.sorted

val res2: Boolean = true

Вот мы и попробовали на практике коллекции и в следующем разделе мы поговорим про коллекции немножко подробнее.

## **6.2 Коллекции. Часть 2**

В прошлом разделе мы с вами узнали, какие коллекции есть в библиотеке Scala, а сейчас узнаем, что с ними можно делать.

### **Создание коллекций**

Создавать коллекции очень просто, пишем имя коллекции и дальше перечисляем все элементы, из которых она должна состоять. Обратите внимание, что если мы говорим про Set, то, например, элемент "2" у нас будет входить в результирующей Set только один раз, а если мы говорим про Map, т.е. ассоциативный массив, то нашими элементами должны быть пары "ключ-значение".

val list = *List*(1, 2, 3, 2)

val vector = *Vector*(1, 2, 3, 2)

val stream = *~~Stream~~*(1, 2, 3, 2)

val lazyList = *LazyList*(1, 2, 3, 2)

val set = *Set*(1, 2, 3, 2)

val map = *Map*("Москва" -> 12e6, "Питер" -> 5e6)

Для всех последовательностей, то есть, например, Vector, LazyList или List, у нас есть такие операции как: добавить слева - это операция "+:", добавить справа - это операция ":+". Обратите внимание, что только для вектора из этих трех операция добавить в конец ":+" будет достаточно эффективной. И если у нас есть две коллекции, мы можем склеить их вместе, сконкатенировать с помощью операции двойной плюс "++".

val phrase1 = *Vector*("+")

val phrase2 = phrase1 +: "Stepik"

val phrase3 = "Scala" :+ phrase2

val phrase4 = *LazyList*.*empty*[String]

val phrase5 = phrase4 :+ "=" :+ "love"

val phrase = phrase3 ++ phrase5

phrase.mkString(" ")

Если у нас есть какая-то неупорядоченная коллекция, как как Map или Set, у нас нет понятия начало и конец, но можем просто добавить элемент с помощью оператора плюс "+", и естественно, в случае Map, ассоциативный массив, мы должны добавлять пару "ключ-значение". И точно так же, если у нас есть целая коллекция, например, список пар, который мы хотим добавить, мы можем добавить их все вместе с помощью операции "++".

val cities1 = *Map*("Питер" -> 5e6, ("Москва", 12e6))

val cities2 = cities1 + ("Волгоград" -> 1e6)

val cities3 = *List*("Ростов-на-Дону" -> 1e6)

val cities4 = cities2 ++ cities3

### **Получение элемента**

Если мы хотим получить какой-нибудь элемент из нашей коллекции, у нас в случае, например, со всеми последовательными, есть метод "apply", то есть можем просто взять и передать индекс элемента в качестве аргумента в нашу переменную, в которой хранится последовательность.

В случае с ассоциативным массивом мы должны передать ключ и тогда он вернет соответствующее значение. В случае с Set метод "apply" возвращает true или false, в зависимости от того входит этот элемент в Set или не входит. Также для последовательностей, есть методы "head" и "last", означающее первый и последний элементы последовательности.

В случае с Map, нам иногда нужно возвращать не только элемент по ключу, но возможно нужно вернуть опциональное значение, которое говорит нам есть этот элемент в нашем ассоциативном массиве или нет, для этого мы используем метод "get".

val cities = *Vector*("Москва", "Волгоград", "Питер")

cities(1) *// Волгоград*

cities.head *// Москва*

cities.last *//Питер*

val cityMap = *Map*("Москва" -> 12e6, "Питер" -> 5e6)

cityMap("Питер") *// 5000000.0*

cityMap.get("Москва") *// Some(1.2E7)*

cityMap.get("Петушки") *//None*

val citySet = *Set*("Москва", "Волгоград", "Питер")

citySet("Волгоград") *// true*

citySet("Петушки") *// false*

### **Информация**

Для большинства коллекций у нас есть метод "size", который говорит сколько и сейчас элементов есть нашей коллекции.

Есть метод "contains", который для последовательностей возвращает есть ли этот элемент среди указанных элементов, и для Set он работает точно так же. А вот для Map мы передаем туда не пару "ключ-значение" как обычно, а просто ключ и метод говорит нам, есть ли этот ключ среди ключей нашего ассоциативного массива.

Если мы хотим узнать список индексов валидных для данной коллекции, можем использовать метод "indices", который вернет нам диапазон всех индексов, по которым мы можем получить элементы из нашей последовательности. Для Map у нас есть похожий метод "keySet", который вернет Set, то есть набор всех ключей нашего ассоциативного массива.

val cities = *Vector*("Москва", "Волгоград", "Питер")

cities.size *// 3*

cities.contains("Москва") *// true*

cities.indices *//0 until 3*

val cityMap = *Map*("Москва" -> 12e6, "Питер" -> 5e6)

cityMap.size *// 2*

cityMap.contains("Питер") *// true*

cityMap.keySet *// Set("Москва", "Питер")*

val citySet = *Set*("Москва", "Волгоград", "Питер")

citySet.size *// true*

citySet.contains("Петушки") *// false*

### **Субколлекции**

Есть различные методы, как мы можем получить субколлекции из наших последовательностей. Метод "slice" возьмет все элементы с одного индекса по другой где второй индекс не включается.

Есть метод "tail", который вернет все элементы кроме первого. Метод "init", который вернет все элементы кроме последнего.

Метод "take", который вернет нам ровно столько элементов сначала, сколько мы захотим. Или метод "drop", который вернет нам все, кроме этих первых элементов.

Есть метод "takeRight", который точно также вернет сколько-то элементов с конца и метод "dropRight", который вернет нам какой-то префикс коллекции, помимо того количества элементов, которые мы справа выбрасываем.

val nums = *Vector*.range(1, 11)

nums.slice(3, 7)

nums.tail

nums.init

nums.take(3)

nums.drop(3)

nums.takeRight(3)

nums.dropRight(3)

Субколлекции для неупорядоченных последовательностей выглядят немножко по-другому. Мы, как правило, можем удалить элемент, в случае с Map удалить можем ключ, в случае Set можем удалить непосредственно именно с помощью операции минус "-" и удалить сразу набор элементов или ключей с помощью операций двойной минус "--".

val cityMap = *Map*(

"Москва" -> 12e6,

"Питер" -> 5e6,

"Волгоград" -> 1e6

)

cityMap - "Питер"

cityMap -- *List*("Москва", "Волгоград")

val citySet = *Set*("Москва", "Волгоград", "Питер")

citySet - "Питер" *// true*

citySet -- *List*("Москва", "Волгоград")

### **Условный отбор**

Также в качестве такого варианта вычисления субколлекции, мы можем отбирать элементы условно, то есть передать некую функцию, которая нам скажет, какие элементы нам подходят, а какие нет.

Метод "filter" вернет нам все элементы коллекции, которые подходят под условия. Метод "filterNot" вернет нам все элементы коллекции, которые не подходят по условию. А "patrition" вернет нам пару из первого и второго варианта.

Точно также метод "takeWhile" вернет нам максимальный префикс удовлетворяющий какому-то условию. Метод "dropWhile" вернет все элементы, кроме этого префикса. А метод "span" вернет нам пару образованную тем и другим методом.

val nums = *Vector*.range(1, 21)

val odds = nums.filter(\_ % 2 == 1)

val evens = nums.filterNot(\_ % 2 == 1)

val (odds1, evens1) = nums.partition(\_ % 2 == 1)

val small = nums.takeWhile(\_ < 10)

val big = nums.dropWhile(\_ < 10)

val (small1, big1) = nums.span(\_ < 10)

### **Отображения**

Мы можем обрабатывать коллекции меняя элементы, пробежавшись по всем элементам коллекции с помощью какой-то функции и вычислив результирующую значению развить общую коллекцию, как набор всех результатов функции. Для этого есть метод "map". Например, мы можем взять какой-нибудь список из индексов и вычислить коллекцию состоящую из символов алфавита с соответствующим индексом.

val nums: List[Int] = *List*.range(0, 10)

val alpha = 'A' to 'Z'

val nums2 = nums.map(i => alpha(i))

val nums3 = nums.map(alpha)

В качестве часто упоминаемого отображения, мы можем использовать метод "collect", некая вещь, которая может одновременно сделать и фильтрацию и мэппинг с помощью какой-то частичной функции. И в качестве частичной функции, мы как правило используем вот такие выражения pattern matching.

val charLists: List[Char] =

nums.collect{

case i if i % 2 == 0 => alpha(i / 2 \* 3)

case 3 => '\_'

case 5 | 7 => '!'

}

У нас есть также способы отображения в случае, если возвращаемое функцией значение, это снова коллекция. Представьте, например, все наша функция возвращает список. Если мы пробежимся по нашему списку с помощью такой функции, мы получим список списков.

Схлопнуть, то есть сконкатенировать все полученные списки, мы можем с помощью методов "flatten", либо вместо того чтобы подряд вызывать сначала “map”, а потом "flatten", мы можем сразу вызвать функцию "flatMap".

val charLists: List[List[Char]] =

nums.map(i => *List*(alpha(i), alpha(i + 3)))

val charList: List[Char] = charLists.flatten

val chars: List[Char] =

nums.flatMap(i => *List*(alpha(i), alpha(i + 3)))

Вот основные операции, их конечно это гораздо больше, которые мы можем использовать, когда мы работаем с коллекциями стандартной библиотеки. Давайте попробуем применить их на практике.

### **Практика**

Мы с вами узнали какие операции бывают у коллекций, как их можно модифицировать, создавая новые неизменяемые коллекции и давайте попробуем применить это на практике.

Одна из хороших областей, где можно применять разные вещи связанные с коллекциями, является не сложная теория чисел. Например, мы можем вычислить является ли наше число простым. Самый очевидный способ вычислить простоту числа - это проверить делимость его на все числа не больше самого числа.

Наша функция будет брать целое число, давайте даже лучше Long и возвращать булевое значение. Воспользуемся коллекцией Stream (аналог после версии Scala 2.13 - LazyList).

Stream - это особая версия списка, особенность которой, заключается в том, что ее хвост ленивый (LazyList - ленивая полностью). Это значит, что он будет вычисляться только момент, когда он нужен. Она позволяет делать очень много замечательных вещей, в том числе позволяет строить бесконечные списки, которые будут ровно настолько вычислены насколько вам нужно. Вы можете такой бесконечный стрим, начинающийся с единицы, и брать пока его элементы не меньше “x”.

def isPrime(x: Long): Boolean =

*Stream*.from(1).takeWhile(p => p < x)

Мы можем даже еще сильнее ограничить здесь. Нам не нужно проверять делители квадрат которых больше чем “x” потому, что для каждого такого делитель должен найтись делитель квадрат которого меньше "x". Поэтому можем брать только делители квадрат которых не превышает "x", и проверить, что для всех этих чисел. Пользуемся методом “forall”, у нас выполняется, что наше исходное число при делении с остатком на это число, остаток отделение не равен нулю.

def isPrime(x: Long): Boolean =

*Stream*.*from*(1).takeWhile(p => p \* p >= x).

forall(x % \_ != 0)

Можем проверить какое-нибудь число, например, 7 на простоту. Оно у нас оказалось непростым, как же так вышло? Оказывается, что наш Stream мы запустили с единицы и конечно же любое число делится на единицу без остатка.

isPrime(7)

val res0: Boolean = false

Если запустим с двойки получается простое. 13 тоже простое, 12 непростое. Например, 101 простое, 102 непростое.

def isPrime(x: Long): Boolean =

*Stream*.*from*(2).takeWhile(p => p \* p >= x).

forall(x % \_ != 0)

isPrime(7)

val res0: Boolean = true

Теперь попробуем оптимизировать наш замечательный алгоритм. Давайте попробуем построить ленивый список, его мы сложим в lazy val, всех простых чисел, ну по крайней мере тех, которые влезают в Long.

Cоздадим такой вот Stream от Long. Для того, чтобы эффективно его использовать, нам обязательно нужно задать хотя бы один элемент нём - это будет двойка. Дальше используем такой вот оператор ленивой катинации "#::" нашей головы с оставшимся хвостом списка. И дальше говорим, что мы возьмём некий Stream, который начнется с тройки и будет шагать через один.

И для каждого такого элемента, который начинается с тройки и будет шагать через один, мы проверим, что это число является простым.

lazy val primes: Stream[Long] = 2 #::

*Stream*.*from*(3, 2).filter(isPrime)

Ну и конечно же мы забыли, что список должен обязательно возвращать Long. У нас, к несчастью, здесь только Int. Для этого можем воспользоваться вместо метода Stream.from, методом Stream.iterate. Начнем с элемента 3, и на каждом шаге будем добавлять 2. Отберем из всех этих элементов только те, которые являются простыми "filter(isPrime)".

lazy val primes: Stream[Long] = 2L #::

*Stream*.*iterate*(3L)(\_ + 2L).filter(isPrime)

Возьмем, с помощью метода "take", например, 50 первых простых чисел. Мы опять забыли сгенерировать наш Stream из чисел и получили некий Stream, как видите, он изначально у нас не вычислен целиком.

primes.take(50)

val res2: Stream[Long] = Stream(2, <not computed>)

Мы можем его вычислить, сконвертировав его в какую-нибудь другую строгую коллекцию, например, "toList" или вызвав метод "force", который означает, вычисли нам все элементы этого Stream, потому что мы знаем что этот Stream у нас не бесконечный. Вот получили что-то очень похожее на список простых чисел.

primes.take(50).toList

primes.take(50).force

val res1: List[Long] = List(2, 5, 7, 9, 11, 13, 15…

val res1: scala.collection.immutable.Stream[Long] =  
 Stream(2, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25…

Мало того, имея такой список простых чисел, мы можем сделать нашу функцию isPrime более эффективной, потому что теперь нам не нужно перебирать все числа от единицы до корня квадратного из "x". Нам достаточно перебрать все простые числа от единицы до корня квадратного из "x". Поэтому мы можем воспользоваться числами которые уже лежат в нашей коллекции primes.

def isPrime(x: Long): Boolean =

primes.takeWhile(p => p \* p >= x).forall(x % \_ != 0)

Как видите наш список не изменился ни капли. Можем взять побольше чисел и убедиться, что мы их тоже достаточно быстро можем вычислить. Вот вычислили огромное количество чисел. Можем посчитать, например, их сумму, если хотите. Сумма первой тысячи простых чисел.

primes.take(1000).sum

val res1: Long = 1001999

Заметьте, что мы получили такое очень сильно рекурсивное определение. У нас есть ленивая коллекция, которая в своем определении, ссылается на функцию. А функция в своем определении ссылается на эту же самую ленивую коллекцию. Вот такие интересные вещи позволяют выполнять ленивые коллекции и разные методы, которые мы узнали.

Стоит также заметить, что коллекция Stream в следующих версиях Scala изменит свое название, она будет называться LazyList. Ну и у всех коллекций в общем-то начиная с версии 2.13.0 немножко изменится API. Вам это нужно будет учитывать, если вы будете использовать Scala, чья версия больше 2.12.

Так мы познакомились на практике с нашими методами коллекций и давайте поговорим про for comprehension.

## **6.3 For comprehension**

Мы с вами познакомились с коллекциями на практике и пришло время узнать про магию for comprehension. Это такой специальный синтаксический сахар, состоящий из двух ключевых слов for и yield. Он трансформируется в цепочку вызовов методов map, flatMap и withFilter. withFilter - это такая специальная версия метода filter.

* **for ... yield ...** — это синтаксический сахар, позволяющий записать цепочку вызовов методов .map, .flatMap, withFilter в выражение в императивном стиле
* **Метод withFilter** — аналог filter, используемый в for-yield выражениях для производительности

Представим, что нам нужно обойти какой-нибудь список, превратив каждый элемент его во что-нибудь другое. Мы можем воспользоваться методом map, а можем воспользоваться for comprehension. Он выглядит точно так же как for-цикл, который мы недавно проходили, отличается ключевым словом yield. После yield у нас идет то самое выражение, результирующий элемент, который попадет в нашу коллекцию.

val nums = *List*(2, 5, 1, 7, 4)

val nums2 =

for(x <- nums)

yield x \* 2

val nums = *List*(2, 5, 1, 7, 4)

val nums2 =

nums.map(x => x \* 2)

Если нам нужно выполнить несколько обходов разных коллекций внутри нашего выражения, можем точно также, как и в случае с циклами, разделить их точкой с запятой. В этом случае подобный обход трансформируется в вызов метода flatMap.

val nums = *List*(2, 5, 1, 7, 4)

val nums2 =

for(x <- nums; y <- 1 to x)

yield y \* 2

val nums = *List*(2, 5, 1, 7, 4)

val nums2 =

nums.flatMap(x => 1 to x)

.map(y => y \* 2)

Мы также можем, вместо точек с запятой, воспользоваться в фигурными скобками и записать каждое выражение на своей строчке. Точно также, как и в циклах, мы можем дописывать условные выражения в наш for comprehension. Элементы, которые не удовлетворяют нашем условном выражением, будут исключены из обхода. Как видите трансформируется в вызов метода filter.

val nums = *List*(2, 5, 1, 7, 4)

val nums2 = for {

x <- nums

y <- 1 to x if y > 3

} yield y \* 2

val nums = *List*(2, 5, 1, 7, 4)

val nums2 =

nums.flatMap(x => 1 to x)

.filter(y => y > 3)

.map(y => y \* 2)

Ну, а если быть точным, то во что это трансформируется выглядит даже немножко сложнее. Вот видите, у нас filter заменился на withFilter. Это такая специальная версия, которая позволяет не создавать промежуточную коллекцию. И вообще весь синтаксис for comprehension, рассчитан на то, чтобы создавать меньше промежуточных коллекций, либо коллекции меньшего размера.

val nums = *List*(2, 5, 1, 7, 4)

val nums2 = for {

x <- nums

y <- 1 to x if y > 3

} yield y \* 2

val nums = *List*(2, 5, 1, 7, 4)

val nums2 =

nums.flatMap(x =>

(1 to x)

.withFilter(y => y > 3)

.map(y => y \* 2))

Если мы хотим добавить переменную, которая не будет обходить никакую коллекцию, а просто зависит от уже объявленных переменных, вот как бы мы добавили val в обычную функцию, мы можем внутри for comprehension написать просто вот такое вот равно. Обрати внимание в какое страшное выражение это превратится, если бы мы ни пользовались синтаксическим сахаром.

val nums = *List*(2, 5, 1, 7, 4)

val nums2 = for {

x <- nums

y <- 1 to x if y > 3

y2 = y \* 2

z <- nums if z < y2

} yield z + y2 - y

val nums = *List*(2, 5, 1, 7, 4)

val nums2 =

nums.flatMap(x =>

(1 to x)

.withFilter(y => y > 3)

.map(y => (y, y \* 2))

.flatMap { case (y, y2) =>

nums

.withFilter(z => z < y)

.map(z => z + y2 - y)

})

### **Область применения**

For comprehension - это очень удобный способ работать с большим количеством вещей. Конечно все эти вещи должны иметь как минимум методы map и flatMap. Ну и хорошо, если они будут иметь метод withFilter, для того чтобы могли в наших for comprehension использовать условие. Как минимум можем работать вот так, чтобы генерировать коллекции из уже существующих.

Мы можем работать с большим количеством опциональных значений, не используя специальные методы map и flatMap, не распаковывая каждый раз значение с помощью pattern matching. Точно также можем работать с типами Either, где мы предполагаем, что левый тип - это тип ошибки.

Дополнительно к этому нас существует целая область асинхронных вычислений. Есть специальные типы Future, Task, IO в разных библиотеках, которые описывают не блокирующие значения, не блокирующие вычисления, которые точно также можем комбинировать с помощью for comprehension.

Есть точно также параллельные асинхронные вычисления с помощью разнообразных асинхронных коллекций или реактивных стримов, которые также обычно поддерживают синтаксис for comprehension.

И любители функционального подхода, могут использовать for comprehension, для того чтобы работать со специальными типами данных, благодаря которым, мы можем избавиться от нашей привязанности к императивному подходу, к мутабельным переменным и другим. Для всего этого есть специальные типы, которые мы легко можем комбинировать, связывать и писать наш код в функциональном стиле с помощью этого замечательного синтаксического сахара.

* Быстрый выход с Option или Either
* Обход сложных коллекций из scala.collection.immutable
* Неблокирующие вычисления с Future, Task, IO
* Комбинаторы асинхронных коллекций Source, Observable, Stream
* Запись вычислений при функциональном подходе

Мы изучили for comprehension и попробуем его на практике.

### **Практика**

Мы с вами узнали про for comprehension и теперь хотелось бы показать его на практике. Конечно, если бы мы хотели показать про for comprehension, все что мы можем рассказать и все способы применения нам нужно было записать очень много видео. Но, мы на самом деле, можем сказать минимум. Начнем с кусочка кода, который мы записывали некоторое время назад. Как помните, этот кусочек кода у нас вычислял какие-то значения от чисел Double и вычислял их скажем так очень явно сообщая об ошибках. То есть если мы случайно пытались вычислить корень квадратный из отрицательного числа, он нам говорил ошибка "деление на 0" и т.д. И дальше мы пытались написать функцию, которая пользуется этим функционалом ошибок с помощью вот такой большой-большой цепочки flatMap. Большая цепочка flatMap у нас очень активно используется и в коллекциях. Мы обязательно воспользуемся for comprehension для коллекций, но сейчас хотелось бы взять этот кусок кода и немного адаптировать. Мы оставим для того чтобы вы помнили в эту функцию, она делает в точности то, что написано здесь, но теперь мы попробуем написать ее версию через for comprehension чтобы сравнить.

import java.lang.Math.*sqrt*

def fun(x: Double, y: Double): Double =

(*sqrt*(x) + *sqrt*(y)) / *sqrt*(x + y)

def sqrtE(x: Double): Either[String, Double] =

if (x < 0) *Left*(s"**$**x < 0 !") else *Right*(*sqrt*(x))

def divE(x: Double, y: Double):

Either[String, Double] =

if (y == 0) *Left*(s"zero division !") else *Right*(x / y)

def funE(x: Double, y: Double):

Either[String, Double] =

sqrtE(x).flatMap { sx =>

sqrtE(y).flatMap { sy =>

sqrtE(x + y).flatMap { sxy =>

divE(sx + sy, sxy)

}

}

}

funE(4, 5)

funE(-2, 5)

funE(3, -4)

funE(0, 0)

Итак, называем ее funEf, на вход нас тут два Double, как и исходной, и возвращать у нас будет Either от String, Double. Функция делает, как вы помните то же самое, берет корни квадратные из двух чисел и делит их на корень квадратный из суммы чисел. Теперь напишем вот то же самое выражение и сразу пишем for c фигурными скобками, предполагая что у нас будет очень много байндинг выражений внутри него.

Начнем с первого. Вычисляем корень квадратный от "x", результат кладем в переменную "sx". Пишем "sx" вытащить значение из корня квадратного с ошибкой от "x". Можем сразу здесь написать "yield", чтобы знали, что это именно та разновидность форм. Дальше тоже самое, вычисляем корень квадратный из "y", результат складываем в переменную "sy". Дальше вычисляем корень квадратный из суммы "x" и "y", результат складываем в переменную "sxy". И, наконец, делим наш "sx + sy" на "sxy", результат складываем в какую-нибудь переменую, например, result, и этот результат возвращаем.

def funE(x: Double, y: Double):

Either[String, Double] =

sqrtE(x).flatMap { sx =>

sqrtE(y).flatMap { sy =>

sqrtE(x + y).flatMap { sxy =>

divE(sx + sy, sxy)

}

}

}

def funEf(x: Double, y: Double):

Either[String, Double] =

for {

sx <- sqrtE(x)

sy <- sqrtE(y)

sxy <- sqrtE(x + y)

result <- divE(sx + sy, sxy)

} yield result

Теперь давайте проверим, что для каждого нашего значения мы получили, то же самое. Берем funE и заменяем на funEf. Получили то же самое. Как видим, семантика нашей функции совершенно не поменялась. Ну, а запись очень сильно изменилась, вместо вот такой вот лесенки flatMap, у нас красивый вертикальный список, такого байндинга или привязывание каких-то переменных к результатам наших методов возможно с ошибкой. То же самое можем сделать, например с коллекциями или с чем угодно, flatMap и большая их цепочка всегда заменяется на аккуратный for comprehension со стрелочками.

Вот, наверное, основная вещь, которую мы хотели чтобы вы знали о коллекциях и о for comprehension. В следующих уроках мы с вами поговорим о началах объектно-ориентированного программирования и в частности о классах.

# Классы

## **7.1 Классы**

Мы с вами познакомились с коллекциями и пришло время познакомиться с объектно-ориентированным программированием в Scala. Мы будем изучать классы.

### **Определение класса**

Класс - это шаблон для создания объектов. Для того чтобы определить класс, мы пишем ключевое слово class, дальше имя класса и после этого фигурные скобочках тело класса.

class Dog{

}

*// Класс — это шаблон для создания объектов*

В теле класса мы перечисляем члены класса один за другим, это разные определения, какие-то значения, методы, может быть даже с вложенные классы или абстрактные типы.

class Dog{

val *name* = "Gray"

def woof() = *println*("гав")

}

*// Его тело может содержать методы, значения*

*// и определения типов*

Эти значения могут ссылаться друг на друга. Они определены в одном пространстве имен.

class Dog{

val *name* = "Gray"

def woof() = *println*(s"**$***name* говорит: гав")

}

*// Они могут ссылаться друг на друга*

Если у нас совсем совсем нет никаких членов класса, мы можем просто выбросить тело класса и не ставит даже фигурные скобочки после имени.

class Dog  
*// Тело класса можно оставить пустым*

### **Создание экземпляра**

Для того чтобы воспользоваться таким классом, нам потребуется экземпляр. Чтобы создать экземпляр, существует ключевое слово "new". В месте, где вы хотите получить новый экземпляр, пишем new, имя класса. После этого, в переменной, которой его присвоили вы можете вызывать методы, можете читать поля объявленные в классе.

class Dog{

val *name* = "Gray"

def woof() = *println*(s"**$***name* говорит: гав")

}

val dog = new Dog

*println*(dog.*name*)

dog.woof()

### **Параметры**

Помимо этого, класс может содержать набор параметров. Конечно, каждый из параметров должен иметь свой тип. Это параметры, так называемого основного конструктора класса. Для того чтобы создать новый экземпляр класса после new и имени класса, мы должны в круглых скобочках перечислить значения для каждого из параметров, как будто бы мы вызываем метод.

class Dog(name: String){

def woof() = *println*(s"**$**name говорит: гав")

}

val dog = new Dog("Gray")

dog.woof()

*// Вы можете определять параметры.*

*// Все параметры должны*

*// быть переданы при создании объекта*

Если мы хотим, чтобы какой-то из параметров автоматически превратился в поле класса, нам достаточно перед его именем добавить ключевое слово "val".

class Dog(val name: String){

def woof() = *println*("$name говорит: гав")

}

val dog = new Dog("Gray")

*println*(dog.name)

dog.woof()

*// Параметры могут быть определены как значения*

*// Тогда на них можно явно ссылаться*

*// как на члены объекта*

### 

### **Инициализация**

Помимо этого, классы во время создания могут делать какую-то работу по собственной инициализации, какие-то действия, возможно с сайд эффектами, изменением внутренних переменных. Если вы хотите такие действия , вы можете прямо в теле класса, между своими определениями, перед ними или после них, писать эти действия, прямо вызывать какие-то методы записывать любые блоки. Они автоматически вызовутся при любом вызове конструктора класса.

class Dog(name: String){

*println*("$name родился!!!")

def woof() = *println*(s"**$**name говорит: гав")

}

val dog = new Dog("Gray") *// Gray родился!!!*

dog.woof()

*// Классы могут содержать блоки инициализации.*

*// Они будут выполняться каждый раз,*

*// когда создаётся экземпляр*

Мы изучили классы, следующем разделе изучим абстрактные типы.

## **7.2 Абстрактные классы**

Мы с вами познакомились с классами и пришла пора значит такое абстрактные типы.

### **Trait**

Первой разновидностью абстрактного типа является trait. Trait - это почти то же самое, что класс, но у него нет списка параметров, хотя это только в текущей версии Scala, в дальнейшем их возможному добавят. И некоторые его члены могут быть не определены. Например, здесь у нас есть функция "name", у нас есть ее тип, но нет определения. Мы предполагаем, что дальше мы каким-то образом ее определим.

trait Animal{

def name: String

val *greeting*: String = s"Hi, I'm **$**name"

}

*// Трейты — это абстрактные типы,*

*// некоторые члены могут быть не определены*

*// В отличие от класса, у Trait не может*

*// быть параметров на момент scala 2.12*

*// В следующих версиях scala это может измениться*

### **Абстрактные классы**

Классы могут быть абстрактными. Для этого мы добавляем ключевое слово "abstract" к определению класса и тогда внутри, точно так же мы можем некоторые члены не определять, оставить их абстрактными.

abstract class Animal(name: String){

val *greeting*: String = s"Hi, I'm **$**name"

}

*// Абстрактные классы — это классы,*

*// у которых некоторые члены могут быть не определены*

*// У них также могут быть определены параметры*

У абстрактных классов и трейтов, точно так же как и у простых классов, мы можем не писать фигурных скобок после имени и оставлять тело пустым.

abstract class Animal(name: String)

trait Mammal

*// Тело класса или трейта*

*// также можно оставить пустым*

### **Создание экземпляров**

Создать экземпляр абстрактного типа не так просто, как экземпляр конкретного типа потому, что у него есть какие-то методы или значения, которые нужно да определить. Для этого после того как мы написали new, имя типа и возможно передали параметры, мы открываем фигурные скобки и фигурных скобках перечисляем определения для всех, еще не заданных абстрактных методов, переменных, значений. Фактически в данный момент мы создаем новый конкретный, но анонимный тип, который является подтипом нашего абстрактного класса.

trait Animal{

def name: String

val *greeting*: String = s"Hi, I'm **$**name"

}

val animal = new Animal{

def name = "Bobik"

}

*// Создать экземпляр абстрактного типа напрямую нельзя*

*// Но можно создать экземпляр "анонимного класса",*

*// доопределяющего абстрактный*

### **Инициализация**

Точно так же как и простые классы, абстрактные классы могут содержать какие-то блоки инициализации.

trait Animal{

def name: String

val *greeting*: String = s"Hi, I'm **$**name"

*println*("$name is created")

}

val animal = new Animal{

def name = "Bobik"

}

*// Абстрактные типы также могут*

*// содержать блоки инициализации*

### **Создание экземпляров**

И точно так же, как и в случае простого класса, если мы создаем экземпляр абстрактного класса, мы должны передать все определенные для него параметры.

abstract class Animal(name: String){

val *greeting*: String = s"Hi, I'm **$**name"

}

val animal = new Animal("Bobik"){ }

*// Если мы создаём экземпляр абстрактного класса,*

*// мы должны передать все параметры*

*// Обратите внимание, что даже если все*

*// абстрактные члены определены, нам всё равно*

*// нужно поставить фигурные скобки*

Мы изучили абстрактные классы, давайте попробуем их на практике.

### **Практика**

Мы с вами познакомились с абстрактными типами, давайте попробуем сделать мне что-нибудь на практике.

Представим, что у нас есть некий абстрактный тип User (abstract class User). У него будет какое-то имя-строчка и будет какой-нибудь абстрактный метод "список друзей", который тоже будет списком юзеров.

Попробуем создать какой-нибудь экземпляр от такого абстрактного типа. val oleg = new User, передаём свои имя и определяем абстрактный метод friends, пусть пока он будет пустым списком.

Мы получили значение типа User, видите оно доопределено какой-то новой информацией возможно, так как мы получили на самом деле не совсем экземпляр того же самого типа, но некого его анонимного подтипа.

abstract class User(name: String) {

def friends: List[User]

}

val oleg = new User("Oleg") {

def friends = *List*()

}

class User

val oleg: User{def friends: List[Nothing]} =  
 $anon$1@2eda26e0

Но вот к несчастью, к строчке у нас не очень красиво приводится это значение, поэтому мы реализуем красивый метод toString, который у нас будет просто возвращать имя. Ну что ж, теперь нас просто Oleg.

abstract class User(name: String) {

def friends: List[User]

override def toString = name

}

Давайте определим еще пару юзеров. У нас будет например катя юзер по имени Катя, у нас будет маша юзер по имени Маша и последний пользователь антон некий пользователь с именем Антон. Эти поиски будут ссылаться друг на друга, в какой-то такой перекрестной форме, поэтому, как правило, для таких ссылок нам лучше определять их как lazy val (ленивые значения).

lazy val oleg = new User("Oleg") {

def friends = *List*()

}

lazy val katya = new User("Katya") {

def friends = *List*()

}

lazy val masha = new User("Masha") {

def friends = *List*()

}

lazy val anton = new User("Anton") {

def friends = *List*()

}

Теперь мы можем сказать, что список друзей олега это, например, катя и маша. И в любом другом коде можно взять переменную олег и можем взять список друзей его, это будет катя и маша. Можем написать явно тип, чтобы мы не получали страшных автоматически выведенных Скалой подтипов для каждого из методов friends.

lazy val oleg = new User("Oleg") {

def friends: List[User] = *List*(katya, masha)

}

oleg.friends

Теперь давайте попробуем решить интересную задачу, найдем список всех друзей друзей - friendsOfFiends. У нас есть уже список юзеров, которые являются друзьями данного пользователя, и у них тоже есть списки юзеров, которые являются друзьями их. Соответственно нам нужно построить какой-то список друзей через одно рукопожатие. Пробежимся по всем друзьям текущего пользователя, пробежимся по всем друзьям полученного пользователя, и вернем этого друга.

abstract class User(name: String) {

def friends: List[User]

def friendsOfFriends =

for {

friend <- friends

friend2 <- friend.friends

} yield friend2

override def toString = name

}

Для этого набросим сюда еще друзей. Друзьями кати, например, будут олег и антон, друзьями маши будут катя и антон, друзьями антона будут катя и маша. Получилось не вполне симметричное отношение, но ничего страшного.

Итак, friendsOfFriends. Нам сразу пишут, что для таких сложно ссылающихся друг на друга значений нам не хватит lazy val, нам хорошо бы еще и расставить типы, чтобы Скала не заблудилась в их поисках. Добавим у каждого значения тип и получаем некий результат: олег, антон, катя, антон.

lazy val oleg: User = new User("Oleg") {

def friends: List[User] = *List*(katya, masha)

}

lazy val katya: User = new User("Katya") {

def friends: List[User] = *List*(oleg, anton)

}

lazy val masha: User = new User("Masha") {

def friends: List[User] = *List*(katya, anton)

}

lazy val anton: User = new User("Anton") {

def friends: List[User] = *List*(katya, masha)

}

oleg.friendsOfFriends

res0: List[User] =

List("Oleg", "Anton", "Katya", "Anton")

Мы видим две проблемы. Во-первых, Олег сам встречается среди своих друзей друзей, наверное, мы бы это не очень хотели можем исключить его прямо на этом этапе. Напишем if friend2 не равно и дальше ключевое слово this, указывающее на ссылку на текущий объект. Как видите сам Олег исключился из списка.

abstract class User(name: String) {

def friends: List[User]

def friendsOfFriends =

for {

friend <- friends

friend2 <- friend.friends if friend2 != this

} yield friend2

override def toString = name

}

res0: List[User] = List("Anton", "Katya", "Anton")

Теперь нам бы хотелось исключить повторение. Для этого мы у результирующего списка, как нам показывает IDE, это будет список юзеров, мы можем вызвать метод distinct. Замечательно проблема решена.

abstract class User(name: String) {

def friends: List[User]

def friendsOfFriends =

(for {

friend <- friends

friend2 <- friend.friends if friend2 != this

} yield friend2).distinct

override def toString = name

}

res0: List[User] = List("Anton", "Katya")

Мы с вами познакомились с абстрактными типами и в следующем уроке поговорим про объекты.

## **7.3 Объекты**

На практике мы с вами познакомились с абстрактными типами и пришло время узнать такое объекты. Иногда у нас есть какой-то набор определений, методов, значений, переменных, для которых мы не хотим создавать специальный класс. Нам они нужны только один раз. Мы можем сгруппировать их в объект.

object Pegasus{

val *name*: String = "Pegasus"

def introduce = "I am the winged stallion"

}

*// Объекты — это экземпляры уникального типа*

*// Вы можете рассматривать их как модули — набор определений*

Объект, это такой модуль, который просто держит какой-то набор членов. Каждый объект экземпляр своего собственного уникального типа.

object Pegasus{

val *name*: String = "Pegasus"

def introduce = "I am the winged stallion"

}

val pegasus: Pegasus.type = Pegasus

*// На них можно ссылаться просто по имени*

*// Тип этого объекта можно получить через `object.type`*

### **Инициализация**

Объекты могут содержать блоки инициализации. Эти блоки вызываются в первый раз, когда вы упоминаете объект.

object Pegasus{

val *name*: String = "Pegasus"

def introduce = "I am the winged stallion"

*println*("Pegasus is initialized")

}

val pegasus: Pegasus.type = Pegasus

*// Они могут содержать блоки инициализации*

*// Эти блоки исполнятся при первом упоминании объекта*

### **Компаньоны**

Иногда бывает полезно к классу или абстрактному типу определить набор статические каких-то методов или значений. Для этого можем использовать объекты-компаньоны. Объект-компаньон - это объект с тем же именем, что основной класс или трейт. Объект-компаньон должен быть объявлен в том же самом исходном файле, что и основной тип.

class Cat(val name: String, val age: Int)

object Cat{

def apply(name: String, age: Int): Cat =

new Cat(name, age)

}

*// Объект - компаньон, имеет то же имя, что и тип.*

*// Должен быть определён в том же исходном файле*

*// Его можно рассматривать как набор статических*

*// определений для класса или трейта*

### **Case object**

Также иногда нам полезно дописать к объекту ключевое слово **case**. По сути он просто сгенерирует красивый метод toString для этого объекта.

case object Unikorn

*// К объекту можно добавить ключевое слово case тогда*

*// у него будет красивый toString*

Мы с вами познакомились с объектами, следующем разделе рассмотрим кейс классы.

## **7.4 Case-классы**

Мы с вами узнали что такое объекты, давайте познакомимся с кейс классами. **Case class** - это разновидность простого класса, но мы ее используем, как такую специальную форму неизменяемой или мутабельной структуры. Как правило, все что нам нужно знать об инстансе (экземпляре) какого-то кейс класса - это значение его полей.

case class Cat(name: String, age: Int)

*// Сase class — это специальные формы классов, можно*

*// понимать их как структуры, представляющие из себя*

*// просто набор полей*

Также, как и у простого класса у кейс класса может быть тело, содержащие различные определения.

case class Cat(name: String, age: Int){

def greet =

*println*("Hi! I'm $name, I'm $age years old")

}

*// Они также могут содержать тело с различными*

*// определениями*

Помимо этого у кейс класса генерируется автоматически компаньон, который содержит несколько методов. Например, метод **apply**, и мы можем создавать экземпляры объектов не используя ключевое слово new.

case class Cat(name: String, age: Int)

val cat = *Cat*("Norbert", 3)

*// Их можно создавать через метод apply в компаньоне*

Каждый параметр основного конструктора автоматически становится полем кейс класса. Нам не нужно записывать ключевое слово val.

case class Cat(name: String, age: Int)

val cat = *Cat*("Norbert", 3)

*println*(cat.name)

*println*(cat.age)

*// Все их поля — автоматически значения `val`*

Кейс класс автоматически генерирует красивый метод toString, а также специальные методы equals и хэш-код, благодаря которым 2 кейс класса считаются одинаковыми и имеют один хэш-код, тогда, когда у них одинаковые значения полей.

case class Cat(name: String, age: Int)

val cat = *Cat*("Norbert", 3)

*println*(cat)

cat == *Cat*("Norbert", 3) *// true*

*// У них автоматически сгенерированы*

*// toString*

*// equals*

*// hashCode`*

Для них автоматически генерируется метод copy. Этот метод copy, будет иметь тот же набор параметров, что и основной конструктор кейс класса. Мало того, для каждого параметра будет определено значение по умолчанию, равное текущему значению этого поля в кейс классе. Зачем же это нужно? Мы можем создать любой экземпляр кейс класса, вызвать у него метод copy и передать по имени только те поля, которые нам нужно изменить. И мы получим копию, где сохраним большую часть полей, но изменим какую-то ее часть.

case class Cat(name: String, age: Int)

val cat = *Cat*("Norbert", 3)

*println*(cat)

cat == *Cat*("Norbert", 3) *// true*

*// У них автоматически сгенерированы*

*// toString*

*// equals*

*// hashCode`*

Мы с вами познакомились с кейс классами, в следующем видео мы узнаем про наследование.

## **7.5 Наследование**

Мы с вами познакомились с классами, давайте поговорим про наследование. Каждый раз когда мы определяем класс, трейт, объект или кейс класс, мы можем определить и него набор базовых типов. Для этого мы пишем ключевое слово **extends**, пишем наш базовый тип, и если нас есть еще базовый тип мы пишем **with** следующий базовый тип, with следующий базовый тип и так далее.

class A extends P1 with P2 with P3 with ...

trait B extends P1 with P2 with P3 with ...

object C extends P1 with P2 with P3 with ...

case class D() extends P1 with P2 with ...

*// Любые определения типов могут*

*// содержать список базовых типов*

Каждое определение из базового типа будет доступно в нашем наследнике и в его экземплярах.

trait Animal{

def name: String

}

trait Woofing extends Animal{

def woof() = *println*(s"**$**name говорит: гав")

}

*// Определения из базовых типов автоматически*

*// становятся**доступными в их наследниках ...*

trait Animal{

def name: String

}

trait Woofing extends Animal{

def woof() = *println*(s"**$**name говорит: гав")

}

val woofer = new Woofing{

def name = "Барбос"

}

woofer.woof

*println*(woofer.name)

*// ... и их экземплярах*

### 

### **Переопределение**

Если базовый тип содержал какие-то абстрактные определения, то есть не содержащие реализации, мы можем определить их в наследниках. Мало того, если у нас был **def** без параметров, мы можем заменить его на **val** или **lazy val**.

trait Animal{

def name: String

def greeting = s"Привет, я - **$**name"

}

object Pegasus extends Animal {

val *name* = "Pegasus"

}

*println*(Pegasus.greeting)

*// Абстрактные определения могут быть заменены на*

*// конкретные при наследовании*

*// Def может быть переопределён как val или lazy val*

Если для кого-то определения у нас была реализация, можно все равно заменить ее, добавив ключевое слово **override** перед определением.

trait Animal{

def name: String

def greeting = s"Привет, я - **$**name"

}

object Pegasus extends Animal {

val *name* = "Pegasus"

override def greeting =

s"Я - **$***name*, крылатый скакун"

}

*println*(Pegasus.*greeting*)

*// Конкретные определения могут быть заменены*

*// на другие с помощью ключевого слова override*

Если в каком-то переопределении, мы хотим сослаться на базовый метод или значение в базовом классе, которую возможно определенны в нашем, мы можем использовать ключевое слово super. **Super** - это как бы экземпляр базового класса с именно его реализациями всех методов или полей.

case class Cat(name: String, age: Int) trait Animal{

def name: String

def greeting = s"Привет, я - **$**name"

}

object Pegasus extends Animal {

val *name* = "Pegasus"

override def greeting =

s"{**$**super.greeting}, крылатый скакун"

}

*println*(Pegasus.*greeting*)

*// При переопределении можно сослаться*

*// на метод предка через super*

### **Наследование от классов**

Если среди наших базовых типов есть класс, мы обязательно должны его указать самым первым. То есть после ключевого слова extends в списке.

abstract class Animal

trait Greeter

object Pegasus extends Animal with Greeter

*// bad*

*// object Pegasus extends Greeter with Animal*

*// Если в списке наследования есть класс, он*

*// должен идти первым в списке наследования*

Мы не можем наследоваться сразу от двух классов одновременно, только от одного.

abstract class Animal

abstract class Plant

trait Greeter

object Pegasus extends Animal with Greeter

*// bad*

*// object Pegasus extends Animal*

*// with Plant*

*// with Greeter*

*// Можно указывать только один базовый класс*

Если базовый класс содержал какой-то набор параметров, когда мы наследуем от него какой-нибудь объект или другой класс, мы должны задать значение для всех параметров базового класса, возможно определенные через значения основного класса.

abstract class Animal(name: String)

trait Greeter

object Pegasus extends Animal("Pegasus")

with Greeter

*// Если у базового класса есть параметры, вы должны*

*// передать их при определении объекта или класса*

Это не касается trait. Trait не должен перечислять значение параметров для базового класса. Однако если потом вы будете наследоваться от этого трейта, вы обязательно должны сделать наследника того класса, который был упомянут среди базовых типов трейта.

abstract class Animal(name: String)

trait Greeter extends Animal

Pegasus extends Animal("Pegasus")

with Greeter

*// Но если вы определяете trait параметры,*

*// передавать не нужно*

*// Но при определении конкретного наследника*

*// этого trait, вам нужно будет обязательно*

*// унаследовать этот базовый класс*

### **Линеаризация trait**

Для наследования трейтов существует тоже своя специфика, о ней много говорят она не очень явная. Но суть в том, что если у вас есть несколько трейтов, которые являются потомками одного базового трейта, то при создании их общего наследника, будет проходить так называемая линеаризация. Об этом мы лучше поговорим на практике.

trait Animal

trait Greeter extends Animal

trait Mammal extends Animal

Pegasus extends Animal with Greeter with Mammal

*// Цепочки из trait, унаследованных от*

*// одного базового trait, упорядочивают*

*// свои реализации особенным образом*

Мы с вами изучили наследование, давайте попробуем его на практике.

### 

### **Практика**

Мы с вами узнали очень много про разное ООП, в том числе изучили наследование и давайте попробуем все это на практике.

Представим что мы хотим сконструировать структуру похожую на бинарное дерево. Определим какой-нибудь абстрактный тип ее узла Node. Например, это будет бинарное дерево Int.

У него будут два подтипа, две разновидности. Это будет Branch, который будет задирать ссылку на левый узел и на правый узел. И это будет некий объект, который будет указывать конец дерева, Leaf. В узлах, в листьях, нам наверное не хватит объекта, мы хотели бы держать кто-нибудь информацию, например, мы хотим хранить там какое-то значение типа String.

Для того, чтобы два этих типа действительно стали разновидностями типа Node, мы говорим что оба они являются наследниками, пишем extends Node.

Теперь мы хотели бы ограничить в том, чтобы у Node нельзя было создать наследников каком-нибудь другом файле. Возможно нам потребуется прямо здесь знание о том, что каждый узел это либо разветвление, либо лист. Мы можем выразить это ключевым словом sealed.

sealed trait Node

case class Branch(left: Node, right: Node) extends Node

case class Leaf(value: String) extends Node

То, что мы сконструировали классически называется алгебраическим типом данных. Ну почему алгебраическим? Потому, что каждый кейс класс, который состоит у нас из набора полей, можно рассматривать как некое произведение типов его полей. А каждый такой sealed trait, можно рассматривать как некую логическую сумму всех типов, которые являются наследниками. Это нас сейчас не очень интересует, но на всякий случай если где-нибудь вы столкнетесь с термином алгебраический тип данных или ADT, как правило речь идет вот о подобных семействах объявлений.

Давайте попробуем что-нибудь с этим деревом сделать. Например, имея вот такое дерево, нам хотелось бы возможно добавить к нему какое-нибудь значение слева. Ну или давайте сначала получим просто список значений, которые есть этом дереве слева направо. Так def values = List[String].

sealed trait Node {

def values: List[String]

}

Как нам создать такой метод values = List[String]? Ну мы можем реализовать этот абстрактный тип в двух наших наследниках. Branch сам по себе не хранит значение, поэтому его values будет очень простым. Это будут значение из левого подузла склееные со значениями из правого поддерева. Дальше, список значений у листа очень прост, он состоит ровно из одного значения.

case class Branch(left: Node, right: Node) extends Node {

def values: List[String] = left.values ++ right.values

}

case class Leaf(value: String) extends Node {

def values = *List*(value)

}

Давайте создадим какое-нибудь простое дерево. Оно у нас будет состоять, например, из трех элементов. Вот у нас лист 1, лист 2 и вот здесь в самом конце лист 3. Вот такое интересное дерево. Посмотрим на список его значений. Получили список: one, two, three.

val tree = *Branch*(*Branch*(*Leaf*("one"), *Leaf*("two")),

*Leaf*("three"))

tree.values

val res0: List[String] = List(one, two, three)

Теперь давайте реализуем какие-нибудь интересные операции над ним. Например, добавление элемента слева. Мы можем назвать ее prepend, а можем воспользоваться классическим оператором "+:". Добавляем какое-нибудь значение типа String конечно же. Результатом должно быть новое неизменяемое дерево типа Node.

sealed trait Node {

def values: List[String]

**def +:(value: String): Node**

}

Вот видите, наши подтипы снова получили такое вот красное подчеркивание, означающие, что у нас появились свежие не имплементированые методы. Давайте их имплементировать.

Что означает что мы добавляем в какую-то развилку элемент слева? На самом деле это значит что мы создаем развилку, где к левому поддереву мы добавляем элемент слева. Правое поддерево остается неизменным.

case class Branch(left: Node, right: Node) extends Node {

def values: List[String] = left.values ++ right.values

**def +:(value: String) = *Branch*(value +: left, right)**

}

Что означает что мы добавляем элемент слева к листу? На самом деле это значит, что мы просто создаем какое-то новое дерево, разветвление, где листом слева будет новый лист с новым значением, а листом справа будет текущий элемент.

case class Leaf(value: String) extends Node {

def values = *List*(value)

**def +:(value: String) = *Branch*(*Leaf*(value), this)**

}

Ну хорошо, возьмем наше дерево, назовем его tree2 и попробуем прибавить к нему слева элемент zero. Итак, tree2.values. Ну вот получили, что в новом дереве список значений у нас вот такой.

val tree2 = "zero" +: tree

tree2.values

val tree2: Branch = Branch(Branch(Branch(Leaf(zero),

Leaf(one)),Leaf(two)),Leaf(three))

val res1: List[String] = List(zero, one, two, three)

Похожим образом можно реализовать добавление элемента справа, ну или например конкатенацию деревьев. Что значит, что мы добавим сюда еще одно дерево. Нам не обязательно заходить в каждый потомок и реализовывать его там. Иногда, мы можем, зная о том, что у нас есть конечное количество вариаций кейс классов, определить прямо здесь. Итак, мы пишем Node. Мы можем просто дать новую ветвь, состоящую из данного дерева и нового дерева.

sealed trait Node {

def values: List[String]

def +:(value: String): Node

**def ++(node: Node): Node = *Branch*(this, node)**

}

Тогда мы можем взять, например, наше уже имеющиеся дерево и конкатенировать с новым вторым деревом и посмотреть на его список значений. Результатом будут все значения левого дерева, затем все значения того же дерева только с нулем посередине.

(tree ++ tree2).values

val res2: List[String] = List(one, two, three, zero, one, two, three)

Давайте теперь попробуем реализовать все-таки функцию добавления с конца. Мы можем реализовать ее точно также, как функцию добавления с начала, но можем реализовать ее альтернативным способом, используя паттерн матчинг. Это тоже вполне легальный способ и для того чтобы вам посмотреть имплементацию, вам не нужно будет заходить отдельно в каждый класс и искать реализацию этого метода в каждом известном подтипе. Вы глядя на одну реализацию сразу сможете понять, что этот метод делает во всех известных вариантах. Иногда это бывает очень полезно.

Мы находимся сейчас в неком классе Node. Мы точно не знаем, что это у нас разветвление или лист, но это точно должно быть что-то одно из двух. Поэтому мы попробуем использовать паттерн матчинг. Если мы набрели на лист с каким-то значением, которое должно быть слева от нашего значения очевидно, мы можем создать новое разветвление, где этот лист будет лежать слева, а наше новое значение будет лежать в листе справа. Ну и как видите, на самом деле нам не обязательно распаковывать лист, достаточно знаеть, что это просто лист. Мы можем использовать, например, вот такой синтаксис, для того чтобы точно понять это лист. И тогда здесь можем сослаться на уже имеющиеся экземпляр лист.

sealed trait Node {

def values: List[String]

def +:(value: String): Node

def :+(value: String): Node =

this match {

**case leaf@*Leaf*(\_) =>**

***Branch*(leaf, *Leaf*(value))**

}

def ++(node: Node): Node = *Branch*(this, node)

}

Если мы встретили разветвление, оно состоит из левого и правого поддеревьев, мы создаем новое разветвление. Левое поддерево оставляем без изменений, а в правое добавляем справа наше значение. Вот такая вот реализация. Видите, если у вас будет очень-очень много строчек разделяющие эти типы, вы сразу можете понять, что происходит глядя только на вот этот кусочек кода.

sealed trait Node {

def values: List[String]

def +:(value: String): Node

def :+(value: String): Node =

this match {

case leaf@*Leaf*(\_) =>

*Branch*(leaf, *Leaf*(value))

**case *Branch*(left, right) =>**

***Branch*(left, right :+ value)**

}

def ++(node: Node): Node = *Branch*(this, node)

}

И можем попробовать наконец наш метод, добавив например, сюда значение "four". И наш список значений, как видите в конце появляется слово "четыре".

(tree ++ tree2 :+ "four").values

val res2: List[String] = List(one, two, three, zero, one, two, three, four)

Вот так мы можем пользоваться нашим ООП. Совмещать его с элементами функционального программирования через pattern matching например. И в следующем разделе мы попробуем разобраться с хитростями связанными с линеаризацией трейтов.

## **7.6 Модификаторы**

На практике мы с вами познакомились с наследованием. Давайте поговорим про модификаторы. Это такие специальные ключевые слова, которые в той или иной степени связаны с наследованием, и каким-то образом ограничивают какие-то ваши действия или доступ каким-то полям.

final

sealed

private

protected

*// С наследованием и доступом связано несколько*

*// особенностей, управляемых ключевыми словами —*

*// модификатор*

### **Final**

Ключевое слово final, когда мы пишем его при определении класса, означает, что мы не можем создавать наследников этого класса.

final class Human

class SuperHuman extends Human *// ошибка!!*

*// Классы могут быть помечены final.*

*// Такие классы нельзя наследовать*

Также ключевое слово final мы можем дописать к любому члену класса или трейта, тогда мы не сможем переопределять конкретно вот этот член класса.

class Human{

final def powerLevel: Int = 100

}

class SuperHuman extends Human{

override def powerLevel: Int = 9001 *// ошибка!*

}

*// Члены классов тоже могут быть помечены как final.*

*// Такие члены нельзя переопределять*

### **Sealed**

Также можем дописать к любому классу или трейту ключевое слово sealed. Sealed - это более мягкая форма final, оно означает, что вы все-таки можете создавать наследников данного типа, но только в том же самом исходном файле. Это означает что по сути у вас есть заранее известный набор наследников и вы можете это использовать, ну например, при паттерн матчинге.

sealed trait Weekday

case object Monday extends Weekday

case object Tuesday extends Weekday

case object Wednesday extends Weekday

case object Thursday extends Weekday

case object Freeday extends Weekday

case object Saturday extends Weekday

case object Sunday extends Weekday

*// Классы и trait могут быть помечены как sealed.*

*// Это значит, что унаследовать их можно только*

*// в том же файле исходного кода*

*// Вы заранее знаете все альтернативы данного типа.*

*// Очень полезно для pattern matching*

### **Private**

Ключевое слово private, когда вы применяете его члену какого-то класса, означает, что вот этот конкретно член может быть использован только в данном классе или в компаньоне. Ни в каком внешнем коде или в другом классе.

class Dog {

private val *kind*: String = "dog"

}

object Dog{

def kindOf(dog: Dog): String = dog.*kind*

}

(new Dog).kind *// ошибка!!!*

*// Члены классы могут быть помечены private.*

*// Это значит доступ к ним возможен только из*

*// экземпляров того же класса или компаньона*

### **Private [this]**

Дополнительно вы можете написать после ключевого слова private [this], вот таких вот квадратных скобках. Это будет означать, что этот данный член класса может быть использован только в данном объекте, его нельзя использовать ни в компаньоне, ни даже в других экземплярах того же самого класса.

class Dog {

private [this] val *kind*: String = "dog"

def kindOf(dog: Dog): String = dog.kind *// Ошибка!!!*

}

*// Пометка private[this] означает, что член класса*

*// может быть доступен только из того же самого*

*// экземпляра*

### **Private [...]**

Любые классы у вас часто могут быть определены в каком-то пакете или могут быть вложены в какой-то класс. Если вы хотите разрешить доступ к кому-то члену вашего класса для всех определений каком-то пакете или в каком-то вышестоящим классе, вы можете дописать имя этого класса или пакета после ключевого слова private в квадратных скобках.

package tinkoff.course.classes

abstract class Dog {

private [Dog] val age: Int

private [classes] val kind: String

private [course] val name: String

}

*// Private[...] означает, что член класса*

*// виден только из какого-то пакета или класса*

*// (полезно для вложенных классов)*

### **Protected**

Protected - это специальное слово, связанное непосредственно с наследованием. Оно означает, что доступ к этому члену класса есть только у наследников этого класса. Не в каком не внешним коде. Также как и у private, к нему можно дописывать различные модификаторы в квадратных скобках.

trait Animal {

protected val kind: String

}

class Dog extends Animal{

protected val *kind* = "dog"

def kindOf(dog: Dog): String = dog.*kind*

}

(new Dog).kind *// ошибка!*

*// Член класса, помеченный как protected, виден*

*// в том числе и всем потомкам этого класса*

### **Параметры конструктора**

Если вы их определяете как val, вы тоже можете дописывать к ним все эти модификаторы, ключевое слово override.

trait Animal {

protected val *kind*: String = "animal"

}

class Mammal

(override protected val kind: String)

extends Animal{

def printKind = *println*(kind)

}

(new Mammal("dog")).printKind

(new Mammal("dog")).kind *// ошибка !!!*

Мы с вами изучили модификаторы и таким образом закончили изучать основы объектно-ориентированного программирования в Scala.

# Параметрический полиморфизм

## **8.1 Обобщенные типы**

В прошлый раз мы с вами говорили про модификаторы доступа, давайте теперь коснемся самых интересных особенностей системы типов Scala. Одна из них обобщенные типы.

Представим, что мы с вами написали два таких вот класса. Они очень похожи друг на друга, как две капли воды, отличаются только типом параметров, даже имена параметры совпадают.

final case class NamedInt(name: String, value: Int)

final case class NamedDouble(name: String,

value: Double)

Мы можем обобщить их, приведя к общему типу Named. Для этого, после имени нашего класса или трейта, в квадратных скобках пишем имя нового параметра типа. Тогда в аргументах конструктора или в теле класса, везде, где мы хотим сослаться на какой-то тип, мы можем сослаться на этот вот параметр типа.

final case class Named[A](name: String, value: A)

*Named*[Int]

*Named*[Double]

### **Ссылки на тип**

Параметр типа не обязательно передавать явно, когда мы создаем новый экземпляр нашего обобщенного типа. Если компилятор Scala может вычислить его из типов передаваемых аргументов.

final case class Named[A](name: String, value: A){

def withName(newName: String): Named[A] =

*Named*(newName, value)

}

*// После этого мы можем конструировать значения Named,*

*// указывая или не указывая тип, параметр типа может*

*// выводиться автоматически*

Когда мы ссылаемся на наш тип, мы можем в том числе передавать его в другие обобщенные типы в теле. В данном случае мы передаем его в качестве аргументов в обобщенный тип Map, который эти все коллекции, о которых мы с вами говорили, также является обобщенным.

final case class Named[A](name: String, value: A){

def toMap: Map[String, A] = *Map*(name -> value)

}

*// Можем ссылаться на другие обобщенные типы*

Если наш обобщенный тип внутри себя имеет обобщенные методы, тогда параметры этого метода, возвращаемый результат и его тело, могут ссылаться как на параметры типа самого метода, так и параметры типа самого обобщенного класса, в котором мы определили этот метод.

*final case class Named[A](name: String, value: A){*

*def mapValue[B](f: A => B): Named[B] =*

*Named(name, f(value))*

*}*

*// Класс также может иметь обобщённые методы, тогда*

*// в теле и параметрах мы можем ссылаться на параметры*

*// типа как класса, так и метода*

### **Абстрактные типы**

Абстрактные типы, как например трейты или абстрактные классы, могут также быть, естественно, обобщенными, также иметь параметры типа. Тогда во всех неопределенных абстрактных компонентах этого класса, мы также можем ссылаться на наш параметр типа как на простой тип.

trait Named[A]{

def name: String

def value: A

def modify(f: A => A): Named[A]

}

Когда мы создаем новый экземпляр какого-то абстрактного типа, компилятор Scala обязательно проверит, что те выражения, которыми вы определяете его абстрактные члены, имеют тип совместимый с вашим параметрам типа, который вы передали в конкретную реализацию.

trait Named[T]{

def name: String

def value: T

def modify(f: A => A): Named[A]

}

def namedInt(n: String, v: Int) =

new Named[Int]{

def name = n

def value = v

def modify(f: A => A) = namedInt(f(v))

}

*// При реализации типы будут проверены*

Если вы наследуетесь от абстрактного типа, вы также должны передать туда в качестве параметра типа какой-нибудь известный тип, возможно определенный тоже на базе вашего параметра типа.

trait Named[A]{

def name: String

def value: A

}

case class NamedList[A](name: String, value: List[A])

extends Named[List[A]]

*// При наследовании вы можете можете передавать*

*// параметры типов*

### **Несколько параметров**

Любой класс может иметь несколько параметров типа и на них распространяется те же самые правила.

final case class Dict[K, V](items: List[(K, V)])

*// Класс может иметь несколько параметров типов*

### **Верхняя граница**

Еще одной особенностью системы типов Scala, является то, что параметры типов могут иметь границы. Вот например верхняя граница означает, что параметр типа обязательно должен быть подтипом какого-то известного типа.

В данном случае мы определяем класс Dict. У него есть один параметр типа и мы говорим что он обязательно должен быть подтипом трейта Item, который мы определили в том же исходном файле. Тогда везде где мы ссылаемся на наш неизвестный абстрактный тип "**I**", мы можем помнить, что он является подтипом "**Item**", а значит имеет все те же самые методы, которые определены в трейте Item. В данном случае мы определяем новый метод, которым используем метод трейта Item.

final case class Dict[I <: Item](items: List[I])

def dict[I <: Item](items: I\*): Dict[I] =

*Dict*(items.toList)

trait Item {

def key: String

def value: String

}

*// Параметры типов могут иметь ограничение "сверху"*

*// Означает, что тип должен быть подтипом указанного*

*// типа*

### **Нижняя граница**

Там также иногда полезна и нижняя граница. В нижней границе мы говорим, что наш параметр типа должен быть надтипом какого-то заранее известного типа или тайп-параметра. В данном случае мы воспользовались этим когда определяли абстрактный метод в классе. В таком случае наш результирующий тип может быть несколько более общим, чем тот, который был из исходно, когда мы добавляем новую строчку в нашу матрицу.

И мы можем использовать оба подхода в данном случае мы добавляем наш класс Dict, в наш словарь, новый элемент, и он должен быть одновременно надтипом заранее известного тайп-параметра "I" и под типом трейта "Item", который определен здесь же.

final case class Dict[I <: Item](items: List[I]){

def +:[J >: I](item: J): Dict[J] =

*Dict*(item :: items)

}

trait Item {

def key: String

def value: String

}

*// Параметры типов могут иметь ограничение "снизу"*

*// Означает, что тип должен быть надтипом указанного*

*// типа*

### **Ссылки**

Мало того, тайп-параметры, когда мы определяем для них границы, могут ссылаться на другие тайп-параметры в том же определении. В данном случае тайп-параметр I ссылается на тайп-параметры К и V, когда мы задаем для него верхнюю границу.

case class Dict[K, V, I <: Item[K, V]](items: List[I])

trait Item[K, V] {

def key: K

def value: V

}

*// Параметры типов могут ссылаться друг на друга*

*// в ограничениях*

### **Абстрактные типы**

И даже, каждый из параметров типа может ссылаться сам на себя. Это называется рекурсивно ограниченная квантификация.

trait Comparable[A <: Comparable[A]]{

def compare(x: A): Int

}

*// Параметры типов могут ссылаться друг на друга*

*// в ограничениях*

Мы с вами прошли обобщенные типы, давайте попробуем их на практике.

### **Практика**

Мы с вами познакомились с обобщенными типами, давайте попробуем их на практике. Итак, у нас есть некий тип Map. Он очень удобен и полезен во многих случаях, имеет одну полезную, иногда не полезную особенность. Если мы вкладываем туда элемент с ключом, который уже есть в нашем Map, мы теряем предыдущий элемент.

Давайте попробуем создать новый тип и исправить эту особенность. Создадим новый класс под названием MultiMap. MultiMap будет состоять из вот такого словаря items, который будет представлять собой Map. У него будет набор ключей, и для каждого ключа у нас будет не один, а целый список значений. Как видите, мы здесь использовали два необъявленных типа Key и Value. Для того, чтобы определить обобщенный тип, ставим здесь квадратные скобки и определяем два эти типа. Теперь мы явно ссылаемся на ключ-значение.

final case class MultiMap[Key, Value](items: Map[Key,

List[Value]]) {

}

Теперь как же такой MultiMap заполнять? Как видно мы можем просто сконструировать его из набора значений, либо мы можем сделать ему объект компаньон, в котором определить еще один метод apply. Один у него уже есть благодаря тому, что это кейс класс, и еще один он также будет принимать два тайп-параметра, ключ и значение нашего MultiMap в котором мы сейчас строим, и собственно наши items, которые будут представлять собой вот такое значение (Key, Value)\*. Звездочка значит что мы хотим передать много таких значений. И в результате, мы предполагаем создать некий MultiMap[Key, Value].

object MultiMap {

def apply[Key, Value](items: (Key,

Value)\*): MultiMap[Key, Value]

}

Как же мы будем создавать такой MultiMap? Ну нам бы хотелось бы взять этот список, каким-то образом сгруппировать все эти вещи по ключу и элементы попавшие в одну группу, скомбинировать список, выбрать только значения, и полученное собственно обернуть в MultiMap.

Так мы и делаем. Пишем обертка MultiMap, дальше пишем items.groupBy, первый элемент, который у нас собственно ключ. Дальше получен уже будет тип Map, но в качестве результата у него видите такой сиквенс пар ключ-значение. Нам нужно взять этот сиквенс, mapValues в значениях и превратить его, сначала отобрав только значения, затем превратить его в список. И полученная вещь у нас собственно будет чем-то вроде MultiMap.

object MultiMap {

def apply[Key, Value](items: (Key,

Value)\*): MultiMap[Key, Value] =

**MultiMap(items.groupBy(\_.\_1).**

**mapValues(\_.map(\_.\_2).toList))**

}

Давайте попробуем. Итак MultiMap, например, яблоко это фрукт. Занимательные факты о яблоке. Яблоко это действительно фрукт. Что еще? Груша тоже фрукт, а яблоко еще и вкусное. Давайте сразу поставим несколько строчек, для того чтобы это было легче читать. Что еще нам известно? Нам известно, что морковь это овощ, а еще морковь, у нас какого-то морковного цвета, не знаю желтого или оранжевого, давайте назовем его желтый. Получили вот такой вот MultiMap.

*MultiMap*(

"apple" -> "fruit",

"pear" -> "fruit",

"carrot" -> "yellow",

"apple" -> "delicious",

"carrot" -> "vegetable"

)

Итак, у груши у нас есть список, скажем так, тегов, из который мы понимаем что это фрукт, у яблока список тегов, из которых мы понимаем, что он фрукт и вкусный, у моркови, что она желтая и овощ. Превосходно.

res0: MultiMap[String, String] =

MultiMap(*Map*(pear -> *List*(fruit),

apple -> *List*(fruit, delicious),

carrot -> *List*(yellow, vegetable)))

Что можно сделать? Ну во первых, нам не очень нравится этот странный toString, поэтому мы можем переопределить у этого метода, toString, дописав ключевое слово override.

Как нам обратить это в строчку? Ну, мы возьмем, например, напишем MultiMap, дальше переберем все items, замапим каждое значение в такую строчку, возьмем ключ и список значений, и для каждого такого ключа и списка значений мы произведем строчку ключ стрелочка и значения разделенные (items.map) запятой. Закроем нашу круглую скобку, а вот эти все items, которые мы сложили, мы их тоже будем разделять чем-нибудь, давайте разделим их точкой с запятой.

final case class MultiMap[Key, Value](items: Map[Key,

List[Value]]) {

override def toString: String =

s"Multimap(**$**{

items.map { case (key, values) =>

s"**$**key -> **$**{values.mkString(", ")}"

}.mkString("; ")})"

Получили вот такую строчку: груша это фрукт, яблоко это фрукт делишес. Вот такой замечательный MultiMap.

val res0: MultiMap[String,String] =

Multimap(pear -> fruit; apple -> fruit, delicious;  
 carrot -> yellow, vegetable)

Можно добавить еще пару методов. Например, метод добавления значений в наш MultiMap. Мы можем здесь реализовать его двумя способами. Первое, мы будем требовать, чтобы ключ у нас был обязательно ключ, а значение был обязательно значение. Получим в результате такой же MultiMap.

Если у нас это какой-то список значений уже существующий, мы добавляем в наш список значений в начало, потому что то просто дешевле, новое значение, и все это мы добавляем виде нового значения по ключу key в наш Map. Итак items + key стрелочка новое значение.

Если у нас нет еще значения с таким ключом, мы добавляем в items список из одного значения. Вот такой код, для того чтобы он еще работал, нам нужно обернуть это в MultiMap.

final case class MultiMap[Key, Value](items: Map[Key,

List[Value]]) {

override def toString: String = ...

def add(key: Key, value: Value): MultiMap[Key, Value]  
 =

*MultiMap*(items.get(key) match {

case *Some*(values) =>

items + (key -> (value :: values))

case None => items + (key -> *List*(value))

})

}

Как видим, у нас есть некая общая часть: items + key стрелочка. Поэтому мы, на самом деле, можем вынести ее за скобки, написать прям вот так items + key стрелочка, здесь закрыть скобочку от стрелочки и собственно проверить, какой же список значений мы можем добавлять. Если у нас есть уже какие-то значения - мы добавляем, если у нас нет никаких значений, мы создаем список ровно из одного значения. Хорошо имеем теперь такой код.

def add(key: Key, value: Value): MultiMap[Key, Value] =

*MultiMap*(items + (key -> (items.get(key) match {

case *Some*(values) => value :: values

case None => *List*(value)

})))

И добавим в наш словарь пару значений. Для начала мы добавим туда какой-нибудь рандомный факт о груше, например, она тоже вкусная, затем добавим туда кто-нибудь еще один факт, ну например, о дыни, о том, что например, тоже желтая.

val dict = *MultiMap*(

"apple" -> "fruit",

"pear" -> "fruit",

"carrot" -> "yellow",

"apple" -> "delicious",

"carrot" -> "vegetable"

)

dict.add("pear", "delicious").add("melon", "yellow")

Получили вот такую реализацию. Похоже что все наши факты сохранились. И имеющиеся значения мы смогли добавить в его список и добавить новое значение.

val res0: MultiMap[String,String] =   
 Multimap(pear -> delicious, fruit;   
 apple -> fruit, delicious;   
 carrot -> yellow, vegetable; melon -> yellow)

Давайте добавим метод map, который сам по себе будет обобщенный, и он будет принимать какую-нибудь функцию из нашего уже имеющиегося типа Value в этот новый тип.

Что мы должны получить? Мы должны получить MultiMap, в котором ключ сохранится, а Value станет B. Что мы должны сделать? Мы должны сконструировать новый MultiMap, пробежаться по всем items и выполнить функцию mapValues, то есть пробежаться по всем значениям. Каждое из значений нашего items - это список. У него мы тоже вызываем этот map и передаем нашу функцию. Вот такой вот map, который на самом деле представляет из себя композицию из двух вложенных map.

final case class MultiMap[Key, Value](items: Map[Key, List[Value]]) {

override def toString: String = ...

def add(key: Key, value: Value): ...

def map[B](f: Value => B): MultiMap[Key, B] = {

*MultiMap*(items.view.mapValues(\_.map(f)).toMap)

}

}

Ну и давайте возьмем наш словарь, и для каждого тега, для каждого факта, ну сначала приведем эту строчку в uppercase и посмотрим, что в результате мы получили теги, которые у нас превратились в верхний регистр.

dict.map(\_.toUpperCase())

val res1: MultiMap[String,String] =   
 Multimap(pear -> FRUIT; apple -> FRUIT, DELICIOUS;  
 carrot -> YELLOW, VEGETABLE)

Дальше мы можем запустить какую-нибудь другую функцию, например, посчитать у каждого тега его длину. И получаем MultiMap из Strings в Int, где говорим, что у груши и какой-то тег длинной 5, у apple два тега длиной 5 и 9, у моркови 6 и 9.

dict.map(\_.length)

val res2: MultiMap[String,Int] =   
 Multimap(pear -> 5; apple -> 5, 9; carrot -> 6, 9)

#### **Код полностью**

**Scala 2.12**

final case class MultiMap[Key, Value](items: Map[Key, List[Value]]) {

override def toString: String =

s"Multimap(**$**{

items.map { case (key, values) =>

s"**$**key -> **$**{values.mkString(", ")}"

}.mkString("; ")})"

def add(key: Key, value: Value): MultiMap[Key, Value] =

*MultiMap*(items + (key -> (items.get(key) match {

case *Some*(values) => value :: values

case None => *List*(value)

})))

def map[B](f: Value => B): MultiMap[Key, B] =

MultiMap(items.mapValues(\_.map(f)))

}

object MultiMap {

def apply[Key, Value](items: (Key, Value)\*): MultiMap[Key, Value] =

MultiMap(items.groupBy(\_.\_1).mapValues(\_.map(\_.\_2).toList))

}

val dict = *MultiMap*(

"apple" -> "fruit",

"pear" -> "fruit",

"carrot" -> "yellow",

"apple" -> "delicious",

"carrot" -> "vegetable"

)

dict.add("pear", "delicious").add("melon", "yellow")

dict.map(\_.toUpperCase())

dict.map(\_.length)

**Scala 2.13**

final case class MultiMap[Key, Value](items: Map[Key, List[Value]]) {

override def toString: String =

s"Multimap(**$**{

items.map { case (key, values) =>

s"**$**key -> **$**{values.mkString(", ")}"

}.mkString("; ")})"

def add(key: Key, value: Value): MultiMap[Key, Value] =

*MultiMap*(items + (key -> (items.get(key) match {

case *Some*(values) => value :: values

case None => *List*(value)

})))

def map[B](f: Value => B): MultiMap[Key, B] = {

*MultiMap*(items.view.mapValues(\_.map(f)).toMap)

}

}

object MultiMap {

def apply[Key, Value](items: (Key, Value)\*): MultiMap[Key, Value] =

{

*MultiMap*(items.groupBy(\_.\_1).

view.mapValues(\_.map(\_.\_2).toList).toMap)

}

}

val dict = *MultiMap*(

"apple" -> "fruit",

"pear" -> "fruit",

"carrot" -> "yellow",

"apple" -> "delicious",

"carrot" -> "vegetable"

)

dict.add("pear", "delicious").add("melon", "yellow")

dict.map(\_.toUpperCase())

dict.map(\_.length)

Вот такие основные вещи, которые мы можем делать используя обобщенные типы и обобщенные методы внутри обобщенных типов. В следующем разделе мы поговорим про вариантность.

## **8.2 Вариантность и род**

На практике мы с вами познакомились с обобщенными типами и давайте продолжим знакомиться с особенностями системы типов Scala.

### **Ковариантность**

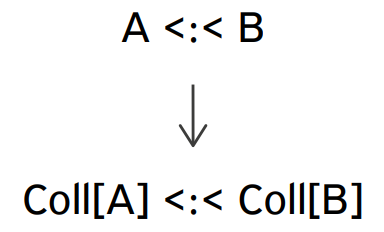
Представим, что у нас есть вот такой вот обобщенный тип Coll, у него есть один аргумент типа А, и мы знаем об этом типе, что он содержит или производит элементы типа А, тогда мы можем объявить его ковариантным, поставив плюсик перед параметрам типа.

trait Coll[+A]{

def apply(i: Int): A

}

Это автоматически будет означать, что мы будем проецировать взаимоотношения между аргументами типа, на взаимоотношения между нашими коллекциями. Иными словами, если тип А является подтипом типа B, тогда Coll от А, является подтипом Coll от B, со всеми вытекающими последствиями.



Когда мы говорим, что какой-то из наших параметров типа к вариантный, мы должны гарантировать, что в его определении он встречается только в ковариантных позициях, для того чтобы наше определение было корректным и мы нигде не получили противоречий. Для этого мы должны убедиться, что тип A, например, встречается в типе результата, или, например, в другом ковариантном типе, в типе результата.

Например, в данном случае нас есть ковариантый тип Option в методе headOption, и сам по себе тип Coll[+A] является ковариантным, поэтому мы тоже можем его поставить в качестве результата нашего метода.

trait Coll[+A]{

def apply(i: Int): A

def headOption: Option[A]

def tail: Coll[A]

}

### **Контрвариантность**

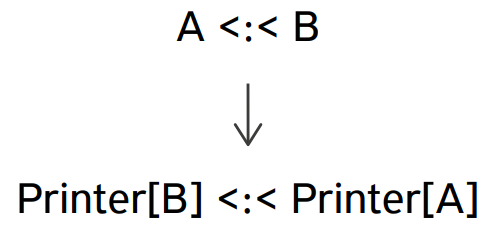
Обратной особенностью является определение контрвариантности. Если ковариантные типы у нас производили или содержали элементы типа A, то контрвариантные у нас как бы поглощают или используют элементы типа A. В данном случае у нас есть некий Printer, который берет один элемент типа A и превращает его в одну строку. Для таких типов мы можем определить параметры типа как контрвариантные поставив минус перед именем параметра.

trait Printer[-A]{

def print(a: A): String

}

И это будет означать ровно наоборот с ковариантностью, что если тип A является подтипом типа B, тогда наш Printer от B, который стоял справа верхнем определении, теперь стоит слева, является подтипом типа Printer от A.



И в данном случае, мы также должны гарантировать, что наш параметр типа встречается в наших определениях только в контрвариантных позициях. Самой обычной позицией, эта позиция аргумента, вот как в методе print. Или, например, он может встречаться в качестве аргумента для другого ковариантного типа, который сам по себе стоит в аргументной позиции, как, например, методе printList. Или он может передаваться в качестве аргумента типа для контрвариантного типа, которые у нас уже в какой-то ковариантной позиции, например, в позиции аргумента, как, например, в методе prefixed.

trait Printer[-A]{

def print(a: A): String

def printList(as: List[A]): String

def prefixed(s: String): Printer[A]

}

### **Вариантность**

Самым классическим примером типа, который объединяет себе контрвариантность и ковариантность, является функция. Функция потребляет элементы типа **A** и производит элементы типа **B**, поэтому она является контрвариантной по первому параметру и ковариантной по второму.

A => B

trait Function1[-A, +R]

*// Функции контрвариантны по первому параметру*

*// и ковариантны по второму*

Мы можем использовать, когда определяем другие ко- и контрвариантные типы. Например, в ковариантном типе **Coll**, мы можем передать функцию, посмотрите как интересно, в качестве аргумента. Функцию, которая принимает на себя аргумент типа **A**, мы опять же ставим в виде аргумента в метод map. Таким образом, **A** у нас присутствует в контрвариантном типе, который сам по себе внутри аргумента, и **A** получается у нас в контрвариантной позиции. Суммарно получаем ковариантную позицию для аргумента типа **A** и компилятор здесь ругаться не будет.

trait Coll[+A]{

def apply(i: Int): A

def map[B](f: A => B): Coll[B]

}

И точно так же когда мы определяем и какой-нибудь контрвариантный тип, мы можем взять функцию, которая возвращает у нас элемент типа **A** нашего аргумента, который мы объявили контрвариантным, и поставить его, например, в качестве аргумента в эту функцию.

trait Printer[-A]{

def print(a: A): String

def contramap[B](f: B => A): Printer[B]

}

### **Род**

Еще одной интересной особенностью системы типов Scala, является наличие родов типов. Род - это что-то вроде типа типов.

Int, String, *List*[Int]: T

*Map*[Int, String], Int => String : T

*// Род — это тип типов или форма типов*

Все типы, о которых мы с вами до этого говорили относится к самому простому роду типов, просто тип. Но когда мы начали говорить с вами про обобщенные типы, то есть типы, которые принимают в себя аргументы, если мы не передадим никого аргумента, сам по себе **List** без уточненного аргумента или **Vector** без уточненного аргумента, являются элементами рода **T**, вот такого, с каким-то одним аргументом.

Int, String, *List*[Int]: T

*Map*[Int, String], Int => String : T

*List*, *Vector* : T[\_]

Также, если мы будем говорить, например, о **Map** и о **Function**, это обобщенные типы, которые имеют два аргумента типа, и они относятся к соответствующему роду.

Int, String, *List*[Int]: T

*Map*[Int, String], Int => String : T

*List*, *Vector* : T[\_]

*Map*, Function : T[\_, \_]

И наши роды могут быть более сложными, например, вот тип **StateT** относится к роду типов с тремя аргументами типов, где второй и третий простые, а вот первый сам по себе принимает качестве аргумента еще один аргумент.

Int, String, *List*[Int]: T

*Map*[Int, String], Int => String : T

*List*, *Vector* : T[\_]

*Map*, Function : T[\_, \_]

StateT: T[\_[\_], \_, \_]

Hoist: T[\_[\_[\_], \_]]

Мы, как правило, используем типы высших родов, то есть всех родов, кроме самого простого, когда говорим о параметрах типов для других обобщенных типов или методов. Например, в данном случае, мы определяем класс **IntContainer**, который содержит какой-то набор элементов типа Int. Мы заранее не знаем что за набор, например, в качестве **F** можем передать **List**, и тогда это будет список целых чисел, или **Vector**, тогда это будет вектор целых чисел.

case class IntContainer[F[\_]](value: F[Int])

*// Параметры типа могут иметь высший род*

Точно также элементы, которые относятся к высшему роду, на них можем накладывать какие-нибудь ограничения. Мало того, если мы накладываем на них ограничения, баунды, о которых мы с вами говорили уже, мы можем использовать некие промежуточные обозначения типов, когда ограничиваем. Например, в данном случае, мы говорим что вот некий наш параметр типа **T**, который относится к более высокому роду, содержащему ровно один аргумент, должен быть всегда подтипом сиквенсов с тем же самым аргументом.

case class IntContainer[F[\_]](value: F[Int])

case class Dict[K, V, T[X] <: Seq[X]](items: T[(K, V)])

Итак, мы с вами изучили вариантность и роды типов и давайте попробуем сделать с ними что-нибудь на практике.

### **Практика**

На прошлом занятии мы с вами познакомились с вариантностью, давайте узнаем на практике что это такое.

[Воспользуемся примером, который делали на прошлой практике](#_o9kqc5vfkdvp), то есть, мы воспользуемся снова типом **MultiMap**. Напомню, это такой словарь, который сопоставляет каждый ключ не одному, а целому списку значений, благодаря чему мы можем добавлять их один за другим, не теряя старые.

Мы познакомились с тем фактом, что типы могут быть ко- и контрвариантными. Например, наш **MultiMap**, он содержит какой-то набор значений типа **Value** и хотелось бы сделать его ковариантым. Для того, чтобы нас была мотивация сделать, давайте в нашем случае разделим наши теги на факты и какие-то личные суждения.

Итак, мы введём новый **трейт Tag**. Он будет sealed, означает, что его можно будет создать подтип только внутри данного исходника. И объявляем 2 его частных проявления. Итак первый **final case class Fact**, который будет просто содержать какое-то описание фактов, и он будет являться членом нашей иерархии **Tag**. Следующий **final case class Personal** - частное суждение.

sealed trait Tag

final case class Fact(name: String) extends Tag

final case class Personal(name: String) extends Tag

Теперь давайте обернем наши свойства в какие-то непосредственные их проявления. Так, факт, что яблоко это фрукт, факт, что груша это фрукт, совершенно факт что морковь бывает как минимум желтой. Ну и давайте словарь на этом и оставим. Добавим в наш словарь два элемента, первое суждение о том, что яблоко это вкусно, и это будет личное суждение, ну а второй о том, что морковь это овощ - это совершеннейший факт. Что мы здесь увидели? Давайте пока уберем все остальное.

val dict = *MultiMap*(

"apple" -> *Fact*("fruit"),

"pear" -> *Fact*("fruit"),

"carrot" -> *Fact*("yellow"))

dict

.add("apple", *Personal*("delicious"))

.add("carrot", *Fact*("vegetable"))

*/\**

*dict.add("pear", "delicious").add("melon", "yellow")*

*dict.map(\_.toUpperCase())*

*dict.map(\_.length)*

*\*/*

Что мы здесь видим? Мы видим две проблемы. Итак, мы пытаемся добавить собственно некий подтип. До этого наш **MultiMap** состоял только из фактов. Это был MultiMap из фактов. Мы пытаемся добавить новые частные суждения. Для того, чтобы попробовать это сделать давайте сделаем наш MultiMap ковариантным пo **Value**. И сразу сталкиваемся с серьезной проблемой.

final case class MultiMap[Key, **+Value**]...

Мы не можем использовать ковариантный тип вот в этой позиции. Действительно, если мы передаем **Value** в качестве аргумента, если мы ожидаем, что это будет наш **Value**, допустим у нас **MultiMap**, где в качестве **Value** выступает факт, и мы передаем факт в качестве аргумента, то когда мы сделаем апкаст, когда мы предположим, что наш ковариантный **MultiMap** от фактов это лишь частный случай от ковариантного **MultiMap** от вообще всех тегов, мы получим, что этот метод будет несовместим, потому что он здесь ожидает конкретно факт и ничто другое. В этом коде внутри мы могли полагаться на тот факт, что это факт, что **Value** будет именно того самого более частного типа.

def add(key: Key, value: **Value**): MultiMap[Key, Value] =

*MultiMap*(items + (key -> (items.get(key) match {

case *Some*(values) => value :: values

case None => *List*(value)

})))

Как же нам можно переделать этот код, для того, чтобы он работал с нашими коллекциями, которые могут быть ковариантны. Мы можем сделать такую стандартную операцию, во все методы, которые у нас складывают какие-то коллекции или добавляют новые элементы в коллекции, мы можем сделать их тоже обобщенными. Можно добавить новый тип **B**. Это будет наше новое значение и потребовать чтобы **Value** было нового типа **B**. Мало того, и сказать, что мы получим **MultiMap** из **B**.

Теперь мы не имеем никакой гарантии, что добавляя теперь в **items** новый элемент с типом **B**, мы действительно можем положить, если мы ждем **MultiMap** типа **B**, но, к сожалению, наши уже имеющиеся типы items никак с вот этим параметром типа **B** не связаны. Мы ничего не сказали о взаимоотношении нашего нового типа **B** и нашего **Value**.

def add[**B**](key: Key, value: **B**): MultiMap[Key, **B**] =

*MultiMap*(**items** + (key -> (**items**.get(key) match {

case Some(values) => value :: values

case None => *List*(value)

})))

Для того, чтобы решить эту проблему, мы можем потребовать, чтобы новый тип **B** был надтипом нашей **Value**.

def add**[B >: Value]**(key: Key, value: B)...

Что это значит? Что автоматически, **Map**, который точно также ковариантен по второму параметру, что и список внутри него, тоже ковариантен по своему единственному параметру. Это автоматически означает, что эта комбинация типов ковариантна по **Value**, находится в ковариантной позиции в ковариантном типе. В результате наша вся комбинация, **Map от Key от List от Value**, ковариантна.

final case class MultiMap[Key,

+Value](items: **Map[Key, List[Value]]**)

И это значит, что наш **items**, даже если он был какого-то типа **Value**, автоматически может быть **Map от Key от List** любого типа, который является надтипом нашего **Value**, в данном случае это некий тип **B**.

def add[B >: Value](key: Key,

value: B): **MultiMap[Key, B]**

Что это значит? Это значит, что когда мы написали здесь вот такой вот метод **add** и добавили метод **Personal**, Scala подобрала наилучший подходящий сюда надтип текущего типа **Fact**, для которого **Personal** тоже подойдет в качестве значения. И это оказался собственно вот такой тип **Product with Serializable with Tag**.

val res0: MultiMap[String,Product with Serializable

with Tag] = Multimap(apple -> Personal(delicious),

Fact(fruit); pear -> Fact(fruit);

carrot -> Fact(vegetable), Fact(yellow))

Tag оказался на последнем месте и неспроста, дело в том, что так как мы реализовали здесь кейс классы они автоматически реализует **Product** и **Serializable**. Этот факт мы на самом деле можем прописать вот здесь, чтобы не раздражаться от дополнительных типов, которые у нас генерируются при выводе, и получим просто **MultiMap** из **строчки в Tag**.

Теперь у нас наш **MultiMap** содержит, как частные суждения, так и строчки. И новый словарь, которым можем добавлять множество частных суждений и множество новых фактов, получит соответствующий правильный тип.

import java.io.Serializable

sealed trait Tag extends Product with Serializable

val res0: MultiMap[String,Tag] = Multimap...

Если мы попробуем в качестве значения положить еще какое-нибудь еще более общее значение, например, мы попробуем сюда положить cherry и какое-нибудь свойство 2, предполагая что 2 ягодки свисают с каждого соцветия, мы получим вот такой странный **MultiMap[String, Any]**, потому, что наибольший общий тип, то есть самый конкретный общий тип у тегов и у целого числа - это **Any**, то есть это даже и не **AnyRef**, поэтому делать такое конечно же не рекомендуется, но вы всегда можете себя проверить.

dict

.add("apple", *Personal*("delicious"))

.add("carrot", *Fact*("vegetable"))

.add("pear", *Personal*("delicious"))

.add("melon", *Fact*("yellow"))

.add("cherry", 2)

val res0: MultiMap[String,Any] = Multimap...

Что было бы, если бы мы попробовали сделать тип **Key** тоже ковариантным?

final case class MultiMap[**+Key**, +Value]...

Здесь мы сразу натыкаемся на одну большую проблему, Ковариантный тип у нас встречается сразу двух позициях. Мы могли бы, в данном случае, исправить точно также, как и в случае с **Value**, принимаем **K** возвращаем **MultiMap от K**. Этот код похоже справляется, но не справляется другой факт.

def add[**K >: Key**,

B >: Value](key: **K**, value: B): MultiMap[**K**, B]...

Map сам по себе инвариантный относительно первого параметра, это очень важный факт, который стоит вспомнить, что в кешированных мэпах мы не можем ослаблять требования на тип, который стоит в ключе. Поэтому, в данном случае никаким образом сделать его ковариантным по вот этому параметру, продолжая использовать в качестве нашей нижестоящей коллекции **Map**, мы не можем, поэтому возвращаем все назад.

def add[**K >: Key**,

B >: Value](key: **K**, value: B): MultiMap[**K**, B]...

Поэтому, в данном случае никаким образом сделать его ковариантным по вот этому параметру, продолжая использовать в качестве нашей нижестоящей коллекции **Map**, мы не можем, поэтому возвращаем все назад.

final case class MultiMap[Key, +Value]...

def add[B >: Value](key: Key,

value: B): MultiMap[Key, B]...

И таким образом, мы познакомились с вами, в простом случае с ковариантностью с ограничениями на тип. Мало того здесь стоит сделать замечание, вот например, что ковариантный тип **Value**, опять же в типе **Map**, как и в лекции встречается в такой интересной позиции, с одной стороны он внутри аргумента, то есть это контрвариантная позиция, с другой стороны три аргумента внутри фанкшн от **Value**, то есть он внутри контрвариантного типа в контрвариантной позиции, результат будет ковариантным.

Вот и все, что мы хотели рассказать вам о контрвариантности, но в следующем разделе мы с вами познакомимся с типами компонентами и типами псевдонимами.

#### 

#### Код полностью

Scala 2.12

import java.io.Serializable

final case class MultiMap[Key, +Value](items: Map[Key, List[Value]]) {

override def toString: String =

s"Multimap(**$**{

items.map { case (key, values) =>

s"**$**key -> **$**{values.mkString(", ")}"

}.mkString("; ")})"

def add[B >: Value](key: Key,

value: B): MultiMap[Key, B] =

*MultiMap*(items + (key -> (items.get(key) match {

case *Some*(values) => value :: values

case None => *List*(value)

})))

def map[B](f: Value => B): MultiMap[Key, B] =

MultiMap(items.mapValues(\_.map(f)))

}

object MultiMap {

def apply[Key, Value](items: (Key, Value)\*): MultiMap[Key, Value] =

MultiMap(items.groupBy(\_.\_1).mapValues(\_.map(\_.\_2).toList))

}

sealed trait Tag extends Product with Serializable

final case class Fact(name: String) extends Tag

final case class Personal(name: String) extends Tag

val dict = *MultiMap*(

"apple" -> *Fact*("fruit"),

"pear" -> *Fact*("fruit"),

"carrot" -> *Fact*("yellow"))

dict

.add("apple", *Personal*("delicious"))

.add("carrot", *Fact*("vegetable"))

.add("pear", *Personal*("delicious"))

.add("melon", *Fact*("yellow"))

Scala 2.13

import java.io.Serializable

final case class MultiMap[Key, +Value](items: Map[Key, List[Value]]) {

override def toString: String =

s"Multimap(**$**{

items.map { case (key, values) =>

s"**$**key -> **$**{values.mkString(", ")}"

}.mkString("; ")})"

def add[B >: Value](key: Key, value: B): MultiMap[Key, B] =

*MultiMap*(items + (key -> (items.get(key) match {

case *Some*(values) => value :: values

case None => *List*(value)

})))

def map[B](f: Value => B): MultiMap[Key, B] = {

*MultiMap*(items.view.mapValues(\_.map(f)).toMap)

}

}

object MultiMap {

def apply[Key, Value](items: (Key, Value)\*): MultiMap[Key, Value] =

{

*MultiMap*(items.groupBy(\_.\_1).

view.mapValues(\_.map(\_.\_2).toList).toMap)

}

}

sealed trait Tag extends Product with Serializable

final case class Fact(name: String) extends Tag

final case class Personal(name: String) extends Tag

val dict = *MultiMap*(

"apple" -> *Fact*("fruit"),

"pear" -> *Fact*("fruit"),

"carrot" -> *Fact*("yellow"))

dict

.add("apple", *Personal*("delicious"))

.add("carrot", *Fact*("vegetable"))

.add("pear", *Personal*("delicious"))

.add("melon", *Fact*("yellow"))

.add("cherry", 2)

## **8.3 Типы: Псевдонимы и компоненты**

В прошлый раз мы с вами познакомились с вариантностью на практике, давайте теперь поговорим про типы псевдонимы и типы компоненты.

### **Псевдонимы**

Псевдоним мы определяем с помощью ключевого слова **type**. После **type** мы пишем имя нового псевдонима равно и его определение с помощью заранее известных нам типов.

type IntList = List[Int]

*// Для типов можно объявлять псевдонимы*

Также псевдонимы могут иметь аргументы типа, также как и любые другие типы. Эти параметры можем использовать как простые типы в его определении.

type DenseMatrix[A] = Vector[Vector[A]]

*// Они также могут иметь параметры*

Также псевдонимы могут иметь аргументы типа, также как и любые другие типы. Эти параметры можем использовать как простые типы в его определении.

Эти аргументы могут быть объявлены ко- или контрвариантными, и компилятор убедиться, что вы ссылаетесь на них в правильных местах, что в определении этот тип стоит на ко- или контрвариантной позиции.

type SparseMatrix[+A] = Map[(Int, Int), A]

*// И передавать вариантность*

Псевдонимы полностью прозрачны для компилятора. Если вы ссылаетесь на **IntList** компилятор помнит, что это **List от Int**, вы можете пользоваться всеми методами списка, и можете неявно конвертировать значение **List от Int** в **IntList**.

type IntList = List[Int]

def evens(xs: IntList): IntList =

xs.filter(\_ % 2 == 0)

*// Псевдонимы полностью прозрачны, их использование*

*// эквивалентно использованию оригинального типа*

Псевдонимы также могут накладывать ограничения на свои параметры типов, также как и простые типы.

type Matrix[F[X] <: Iterable[X], A] = F[F[A]]

*// Они могут накладывать ограничения на свои параметры*

Мало того, вы можете использовать псевдонимы для того чтобы сконструировать тип правильной вариантности и передать его в качестве аргумента в какой-нибудь другой параметризованый обобщенный тип.

type Matrix[F[X] <: Iterable[X], A] = F[F[A]]

type IntMap[A] = Map[Int, A]

type DenseMatrix[A] = Matrix[IntMap, A]

*// С помощью них можно создавать типы желаемого рода*

### 

### **Типы-компоненты**

Если вы определяете какой-нибудь трейт или класс и внутри него, в его определении, определяете какие-нибудь псевдонимы, эти псевдоним будут называться типами-компонентами. Вы можете на них ссылаться так же, как если бы ссылались на аргументы типа в обобщенном типе. Мало того, если сам этот исходный тип абстрактный, как данном случае трейт **Item**, тогда его алиасы, то есть псевдонимы могут быть также не определены, также быть абстрактными, вы все еще можете на них ссылаться в определении.

trait Item {

type Key

type Value

def key: Key

def value: Value

}

*// Типы-псевдонимы без конкретного определения*

*// называются типами-компонентами, они могут*

*// различаться у различных экземпляров родительского*

*// типа*

Когда мы берем какой-нибудь абстрактный тип и у него есть абстрактные типы компонентов, компилятор при ссылке на абстрактный тип забудет какие типы компонентов конкретно были реализованы в конкретном месте. Но это не значит, что мы не можем использовать вот этот наш тип компоненту для того чтобы убедиться, что все остальные определения в теле нашего трейта хорошо подходят друг другу.

trait Сoyoneda[A] { self =>

type Pivot

val start: Pivot

val run: Pivot => A

def value: A = run(start)

}

*// Мы можем «забыть» конкретный тип-компонент у*

*// объекта, но удостовериться, что типы сходятся в*

*// реализациях других компонентов*

### **Уточнение**

С другой стороны компилятор будет стараться сохранить как можно больше информации о типах внутри вашего экземпляра, когда вы выводите из него значение. Например, вот здесь мы явно не указали тип для переменной **ints**. Но когда мы создавали новый экземпляр, мы сказали, что это некий контейнер у которого тип **Item** равен **Int**. Компилятор запомнит эту информацию, и когда вы будете ссылаться на **values**, он поймет, что она должна быть типа **List от Int**.

Чтобы воспользоваться этим механизмом, мы можем использовать так называемые структурные уточнения типов. После любого абстрактного типа, который содержит какие-то типы компонентов, мы можем в фигурных скобках явно уточнить какой тип мы используем в качестве какого типа компонентов. Например, в данном случае, мы говорим что **ints** - это не просто контейнер, а контейнер у которого **Item** - это целое число. И опять же вызываем метод **values** у него, компилятор поймёт что результат должен быть типа списка целых чисел.

trait Container {

type Item

def values: List[Item]

}

val ints = new Container{

type Item = Int

val *values* = *List*(1, 2, 3)

}

ints.*values // List[Int]*

*// Компилятор будет стараться запомнить как можно*

*// больше информации о типах*

Мало того, мы можем взять в это уточнение и повесить на него псевдоним, некий параметризованый алиас, как правило называется **Aux**, и таким образом построить мостик между типами, содержащие абстрактные типы компоненты и параметризоваными типами. Такой вот мостик, такой прием, когда мы превращаем наши абстрактные типы компонентов в параметры какого-нибудь алиаса, их как правило называются **Aux pattern**. **Aux** - это сокращение от **Auxiliary**, значит, что мы добавили несколько типов в определении нашего исходного типа.

trait Container {

type Item

def values: List[Item]

}

object Container{

type Aux[I] = Container{type Item = I}

}

*// Мы `можем поставить в соответствие уточнённому типу*

*// параметрический псевдоним. Этот приём называется*

*// Aux-паттерн*

Вот и все, что мы хотели рассказать про типы-псевдонимы и типы-компоненты, давайте перейдём к не менее интересной вещи в Scala - неявные параметры.

## **8.4 Неявные параметры**

Мы с вами только, что познакомились с псевдонимами типов, и пришла пора поговорить об одной из самых интригующих частей в Scala - неявные параметры.

### **Суммируем списки**

Итак, представим, что у нас есть три таких очень похожих друг на друга функции. Все три принимают на вход какой-нибудь список, все три запускают метод **foldLeft**, начиная с какого-то нейтрального элемента и комбинируют с помощью какой-то бинарной операции.

def concatAll(list: List[String]): String =

list.foldLeft("")((x, y) => x + y)

def sumAll(list: List[Int]): Int =

list.foldLeft(0)((x, y) => x + y)

def forAll(list: List[Boolean]): Boolean =

list.foldLeft(true)((x, y) => x && y)

Мы можем удостовериться, что каждый из этих методов работает по одиночке, но нам бы хотелось каким-то образом обобщить их все в одно определение.

scala> sumAll(*List*(1,2,3))

res0: Int = 6

scala> concatAll(*List*("hello", ", ", "world", "!"))

res1: String = hello, world!

### **Выносим за скобки**

Мы можем вспомнить, что вот такая пара некой бинарной операции и нейтрального элемента, относительно этой бинарной операции, в математике называется **моноид**. Мы создаем такой вот абстрактный тип, описывающий моноиды и комбинируем все наши три функции в одну общую функцию, которая делает по сути тоже самое.

trait Monoid[A]{

def empty: A

def combine(x: A, y: A): A

}

def combineAll[A](list: List[A])(monoid:

Monoid[A]): A =

list.foldLeft(monoid.empty)(monoid.combine)

Мы определяем реализацию нашего моноида для целых чисел, где наши бинарной операцией является **сложение**, для строчек, где нашей бинарной операцией является **конкатенация** и для булевых значений, где нашей бинарной операцией является **логическое и**.

object IntMonoid extends Monoid[Int]{

override def empty = 0

override def combine(x: Int, y: Int) = x + y

}

object StringMonoid extends Monoid[String]{

override def empty = ""

override def combine(x: String, y: String) = x + y

}

object BooleanMonoid extends Monoid[Boolean]{

override def empty = true

override def combine(x: Boolean, y: Boolean) = x && y

}

Убеждаемся, что все работает точно также, как и раньше. Потому, что, скорее всего, для одного типа наша бинарная операция будет одной и той же на протяжении всей программы. Нам не хотелось бы каждый раз явно передавать вот эти значения IntMonoid, StringMonoid в качестве параметра.

scala> combineAll(List(1,2,3))(IntMonoid)

res3: Int = 6

### **Неявные параметры**

Мы можем воспользоваться мощным механизмом под названием неявные параметры. Для этого мы берем определение **combineAll**, которое мы уже написали и вставляем слова **implicit**, прямо вот сюда, перед именем параметра, который хотим, чтобы компилятор определял за нас.

def combineAll[A](list:

List[A])(**implicit** monoid: Monoid[A]): A =

list.foldLeft(monoid.empty)(monoid.combine)

И перед каждым из определений реализации моноида для наших отдельных типов, мы также добавляем ключевое слово **implicit**. Это может быть **implicit** перед **object**, перед **val** или перед **def**, и мало того, этот **def** также может сам по себе ссылаться на другие неявные параметры.

implicit object IntMonoid extends Monoid[Int]{

override def empty = 0

override def combine(x: Int, y: Int) = x + y

}

implicit object StringMonoid extends Monoid[String]{

override def empty = ""

override def combine(x: String, y: String) = x + y

}

implicit object BooleanMonoid extends Monoid[Boolean]{

override def empty = true

override def combine(x: Boolean, y: Boolean) = x && y

}

Сами по себе эти параметры мы не можем определить top level, то есть мы не можем создать просто в каком-то пакете, какой-то им **implicit object**. Нам нужно их куда-то поместить. Поместить их туда, где компилятору их будет легче всего найти, когда он будет искать, собственно реализацию нашего моноида для различных типов. Одним из лучших таких мест является объект-компаньон нашего абстрактного типа **Monoid**. Копируем все реализации туда вместе с ключевыми словами **implicit**.

object Monoid {

implicit object IntMonoid extends Monoid[Int]{

override def empty = 0

override def combine(x: Int, y: Int) = x + y

}

implicit object StringMonoid extends Monoid[String]{

override def empty = ""

override def combine(x: String, y: String) = x + y

}

implicit object BooleanMonoid extends Monoid[Boolean]{

override def empty = true

override def combine(x: Boolean, y: Boolean) =

x && y

}

}

И теперь берем нашу функцию, передаем туда список целых чисел, не передаем больше ничего в качестве второго параметра, и волшебным образом получаем наш целочисленный результат.

scala> combineAll(*List*(1, 2, 3))

res4: Int = 6

### **Вывод типов**

Каким же образом компилятор понял какой именно из трех неявных моноидов можно подставить в качестве значения для неявного аргумента? Оказывается компилятор взял наше определение функции, зная что она хочет какой-нибудь список, и посмотрел список каких элементов на самом деле передан, понял, что передается список целых чисел, а значит параметрам типа для данного обобщенного метода является целое число, а значит нам нужно искать моноид именно для целых чисел. Таким образом компилятор явно понял, что из трех конкурирующих друг с другом определений моноидов нам нужен именно тот, который определен для целых чисел.

combineAll[A](*List*(1,2,3) :

List[A])(monoid: Monoid[A])

↓

combineAll[A](*List*(1,2,3) :

List[Int])(monoid: Monoid[A])

↓

combineAll[Int](*List*(1,2,3) :

List[Int])(monoid: Monoid[A])

↓

combineAll[Int](*List*(1,2,3) :

List[Int])(monoid: Monoid[Int])

Неявные параметры открывает перед нами множество возможностей, но с другой стороны имеют свои плюсы и минусы. В сообществе Scala ведется постоянная дискуссия о том можно ли их применять и в каком количестве.

Среди минусов мы можем назвать тот факт, что они действительно неявные, вы не всегда можете понять, что конкретно передается у вас в качестве аргумента в каком месте. Они могут значительно замедлить в компиляцию кого-то определенного кода, потому что компилятору придется искать или даже конструировать неявные значения вместо вас. Мало того, они могут сделать некоторые части кода неоднозначными, по-разному работающие в зависимости от того в каком контексте вы этот самый кусочек кода используйте, и от того какие неявные параметры он подставит там или там.

С другой стороны, размер вашего исходного кода может значительно сократиться при использовании неявных параметров, так как часть работы по написанию кода вы перекладываете на компилятор. Многие вещи, которые выглядели бы сложно или перегружено будут выглядеть гораздо более просто или читаемо. Вы можете создавать мощные синтаксисы, DSL с использованием неявных параметров или конверсий. А также неявные параметры открывают дорогу в обширную область функционального программирования и реализации более мощных и гибких концептов с помощью них.

Решать использовать **implicit** или не использовать вам, а мы попробуем их на практике.

### **Практика**

Мы с вами наконец познакомились с неявными параметрами и давайте узнаем немного о этом волшебном слове **implicit**.

Итак, представим, что вы хотите создать какой-нибудь свой новый интересный тип, **final case class**, например, это будет тип рациональных дробей. У него будет числитель, это будет большое целое число, и знаменатель, большое целое число.

final case class Ratio (num: BigInt, den: BigInt)

Чтобы создать значение такого типа, вы можете естественно запустить такой вот конструктор, и передать ему два значения.

*Ratio*(1, 4)

val res0: Ratio = Ratio(1,4)

Но возникает проблема естественно не любые два длинных числа могут здесь выступать в качестве числителя и знаменателя. Вам бы хотелось, например, сократить дробь перед тем как сделать с ней что-нибудь. Например, если вы запускаете создание дроби две четвертых, вам возможно хотелось бы, чтобы автоматически она сократилась.

Для этого можете сделать специальный конструктор, так, числитель, знаменатель. Ну и для того чтобы пользоваться только таким конструктором, вы сделали ваш основной конструктор **private**, то есть им не может пользоваться никто кроме самого класса и его компаньона.

final case class Ratio **private[Ratio]**(num: BigInt,

den: BigInt)

**object Ratio{**

**def make(num: BigInt, den: BigInt): Ratio =**

**}**

*Ratio*(2, 4)

Итак, он будет возвращать некую рациональную дробь. Как он будет ее вычислять, ну если наш знаменатель будет равен нулю, мы надеемся такого не случится, ну а в противоположном случае мы попробуем вычислить сначала наибольший общий делитель двух этих значений, слава богу, у нас есть такой метод прямо в объекте BigInt, и затем создать дробь, которая сразу будет поделена на это наибольшие общее частное.

object Ratio{

def make(num: BigInt, den: BigInt): Ratio = {

**val gcd = num.gcd(den)**

***Ratio*(num / gcd, den / gcd)**

}

}

Несмотря на то, что мы объявили его **private**, определенные правила воркшитов не мешают нам все равно его использовать, поэтому нам нужно запустить явно **make**. Как видим дробь автоматически сокращается.

Ratio.***make***(2, 4)

val res0: Ratio = Ratio(1,2)

Но такой способ при делении чисел наверное не очень красиво, поэтому мы хотели бы придумать какой-нибудь красивый синтаксис, для этого в Scala есть неявные преобразования. О неявных преобразованиях вы можете принять отдельно, но заключается в том чтобы придумать некую функцию, которая автоматически будет подхватываться компилятором, и превращаясь в какой-нибудь тип-обёртку над вашим реальным значением, который будет иметь методы, которых нет у вашего оригинального типа. Таким образом вы можете как бы расширить ваш тип новыми методами.

Для того чтобы упростить создание таких оберток, в Scala придумали вот такой синтаксический сахар - **implicit class**. Это может быть **implicit class RatioOps**. **Ops** - значит operations. В качестве значения мы будем передавать наш num, то есть числитель, и в качестве метода мы воспользуемся, допустим, вот таким двойным бэкслешем **\\**, который будет получать еще один **BigInt**, в качестве знаменателя и возвращать рациональное число, и собственно в качестве нашей реализации мы просто вызываем метод **make**.

implicit class RatioOps(val num: BigInt){

def \\(den: BigInt): Ratio = Ratio.*make*(num, den)

}

Прибавим хитрости для того, чтобы сделать эту вещь еще более универсальной, мы можем даже потребовать сделать нашу обёртку для любого типа **T**. Сделать наш знаменатель, потребовать от него, собственно, быть большим числом.

implicit class RatioOps**[T]**(val num: **T**){...

Понять, что, когда вот здесь вот в обертке пишется **4,** на самом деле, здесь изначально было целое число, но у нас есть специальное неявное преобразование, можем нажать комбинацию **Ctrl+Shift+Q** и посмотреть какая, она называется **int2bigInt** - специальная неявная функция, которая преобразует любое целое число в большое число. Поэтому видите наше IDE подчеркнуло нам циферку, говоря, что здесь вроде был **Int**, но на самом деле мы неявно преобразовали его в **BigInt**, чтобы правильно передать в параметр.

*BigInt*(2) \\ 4

Мы можем тоже самую штуку провернуть и здесь. Сказать, что мы возьмем любой тип **T**, который содержит некую функцию для преобразования из **T** в **BigInt**. И вот здесь как видите, автоматически компилятор сразу подхватил это неявное преобразование, и вместо **BigInt от 2** до **4**, мы просто взять вот так **2 \\ 4**, и получили наше рациональное число, которое состоит из двух больших целых чисел.

implicit class RatioOps[T](val num: T){

def \\(den: BigInt)**(implicit f: T => BigInt)**: Ratio =

Ratio.*make*(num, den)

}

**2 \\ 4**

Следующее, что мы делаем, это собственно попробуем найти какие-нибудь еще специальные неявные значения, которые передаются в качестве неявных параметров. На самом деле, в стандартной библиотеке Scala, есть некий набор таких типов, один из них это тип, который используется для сравнения - он называется **Ordering**. Для того, чтобы определить какое-нибудь неявное значение этого типа, мы в компаньоне нашего нового типа можем реализовать экземпляр вот этого **Ordering**. Пишем **Ordering от Ratio**.

object Ratio{

def make(num: BigInt, den: BigInt): Ratio = {...}

implicit val *ordering*: Ordering[Ratio] =

new Ordering[Ratio]

}

**Ordering** это специальный такой тип, который содержит буквально один метод **compare**, он сравнивает два числа типа **Ratio** и обязан вернуть **Int** в качестве результата, это некое целое число, если оно отрицательное значит мы говорим, что **x** у нас получилось меньше, чем **y**, если оно положительное, значит **y** меньше, чем **x**, если оно равно нулю значит эти числа равны.

implicit val *ordering*: Ordering[Ratio] =

new Ordering[Ratio] {

override def **compare**(x: Ratio, y: Ratio): Int = *???*

}

В качестве такой простой реализации, мы можем выбрать вот такую интересную вещь, **x.num \* y.den - x.den \* y.num**, вот все это будет на самом деле неким **BigInt**, явно в **Int** мы его может быть не всегда можем преобразовать, но как минимум, мы можем взять **Ordering**, который у нас уже определена для **BigInt**. И у метода **Ordering** вот здесь вот есть специальный метод **apply**, в котором в качестве параметра передаем любой тип, и он ищет для этого типа реализацию **Ordering** для этого типа. Таким образом, мы призвали **Ordering для BigInt**, после этого можем вызвать у него метод **compare** и сравнить вот эти два значения.

override def compare(x: Ratio, y: Ratio): Int =

Ordering[BigInt].compare(x.num \* y.den, x.den \* y.num)

Что мы теперь можем делать с этим **Ordering**, ну например, мы можем воспользоваться опять же синтаксическими расширениями, которые похожи вот на такой-то **implicit class**, как мы здесь только что написали, только определенные в стандартной библиотеке Scala. Например, можем вызвать **import Ordering.Implicits.\_**. **Implicits**, то есть какие-то неявные штуки, определенные для **Ordering**.

import *Ordering*.Implicits.\_

И в частности, у него есть специальный **implicits**, который автоматически насыщает наши типы какими-то специальными операциями. Например, можем сравнить две четвертых и три пятых. Как видите, наша IDE подчеркнула метод **<** (меньше), говоря о том, что он тоже произведен из некой неявной конвертации нашего **Ratio** в какую-то специальную обертку, которая содержит метод **<** (меньше). Видите, явно метод **<** (меньше) мы нигде не определяли, а он автоматически определился благодаря тому, что у нас есть неявное значение типа **Ordering**, и мы можем узнать меньше это или больше.

(2 \\ 4) < (3 \\ 5)

val res0: Boolean = true

И есть набор стандартных методов, которые требуют **Оrdering** неявно. Например, можем определить список каких-нибудь чисел, из которых нам не совсем понятно кто из них маленький, кто из них большой. Давайте напишем **list**. Результатом будет такой вот список рациональных дробей.

val list =

*List*(1 \\ 3, 2 \\ 5, 3 \\ 7, 2 \\ 7, 4 \\ 19)

val list: List[Ratio] = List(Ratio(1,3), Ratio(2,5),

Ratio(3,7), Ratio(2,7), Ratio(4,19))

Иногда для того, чтобы явно использовать неявные преобразования, мы можем импортировать вот такой флаг компилятор, либо прописать сразу в обзор компиляторы, и тогда мы избежим некоторых варнингов.

import language.*implicitConversions*

После чего, мы можем в этом списке воспользоваться некоторыми методами. Например, метод минимум. Как видите метод минимум (**min**), требует точно также неявный параметр **Оrdering**, то есть сравнение, некий объект, который умеет сравнивать элементы этого списка.

def min[B >: A](implicit ord: Ordering[B]): A

И получили, что минимальное значение, которое есть в списке - это четыре девятнадцатых. Можем уменьшить до семи девятнадцатых и получить, что минимальное значение равно две седьмые.

list.min

val res1: Ratio = Ratio(4,19)

Дальше можно найти максимальный элемент в этом списке, это у нас три седьмых. Можем немножко вот так интересно, после две седьмых минимальная, три седьмых максимальная, можем сделать здесь десять девятнадцатых и он сразу становится максимальным.

list.max

val res2: Ratio = Ratio(3,7)

И мало того, мы можем даже взять пару списков, **list2**, где вот здесь, например, будет три восьмых, и попробовать сравнить их друг с другом. Интересно, но Scala смогла это сделать, она смогла определить какой из типов больше, и точно так же, для того, чтобы найти вот такое неявное сравнение для списков, она воспользовалась специальным методом внутри **Ordering**, который умеет сравнивать списки.

val list =

*List*(1 \\ 3, 2 \\ 5, 3 \\ 7, 2 \\ 7, 10 \\ 19)

val list2 =

*List*(1 \\ 3, 2 \\ 5, 3 \\ 8, 2 \\ 7, 10 \\ 19)

list < list2

val res3: Boolean = false

Если мы посмотрим исходный код, вот здесь мы найдем вот такой вот метод **implicit def seqDerivedOrdering**, который говорит, что если у нас есть какая-то коллекция, которая как минимум последовательность элементов, и мы знаем как сравнивать элементы друг с другом, мы знаем как с сравнивать друг с другом коллекции. И сравнивая коллекции, мы сравниваем их лексикографически, таким образом три восьмых меньше чем три седьмых из-за того что его знаменатель больше, а значит первый список суммарно больше чем второй, потому что состоит из больших элементов. Если мы здесь уменьшим второй элемент, результат сразу станет **false**.

implicit def seqDerivedOrdering[CC[X] <:  
 collection.Seq[X], T](implicit ord: Ordering[T]):  
 Ordering[CC[T]]

Вот такой краткий обзор неявных значений, неявных преобразований, неявных параметров. Более подробно вы сможете изучить на других ресурсов или у нас в финтех школе в Тинькофф, если вы сможете туда попасть или на каких-нибудь курсах по функциональному программированию, в том числе на языке Scala эта информация тоже часто рассказывается, потому что именно через неявные значения, через неявные параметры в Scala реализуется такой паттерн, как type class, то есть некие ограничения, которые накладываются на целый тип, а не какое-то отдельное значение. Спасибо за внимание, это был последний раздел. Удачи с практическими заданиями.

#### Код полностью

import *Ordering*.Implicits.\_

import language.*implicitConversions*

final case class Ratio private[Ratio](num: BigInt, den: BigInt)

object Ratio{

def make(num: BigInt, den: BigInt): Ratio = {

val gcd = num.gcd(den)

*Ratio*(num / gcd, den / gcd)

}

implicit val *ordering*: Ordering[Ratio] = new Ordering[Ratio] {

override def compare(x: Ratio, y: Ratio): Int =

*Ordering*[BigInt].compare(x.num \* y.den, x.den \* y.num)

}

}

implicit class RatioOps[T](val num: T){

def \\(den: BigInt)(implicit f: T => BigInt): Ratio =

Ratio.*make*(num, den)

}

(2 \\ 4) < (3 \\ 5)

val list = *List*(1 \\ 3, 2 \\ 5, 3 \\ 7, 2 \\ 7, 10 \\ 19)

val list2 = *List*(1 \\ 3, 2 \\ 5, 3 \\ 8, 2 \\ 7, 10 \\ 19)

list.min

list.max

list < list2