

# 14. «Функциональность»

или

лямбды, lips, python, map-reduce,  
Stream API

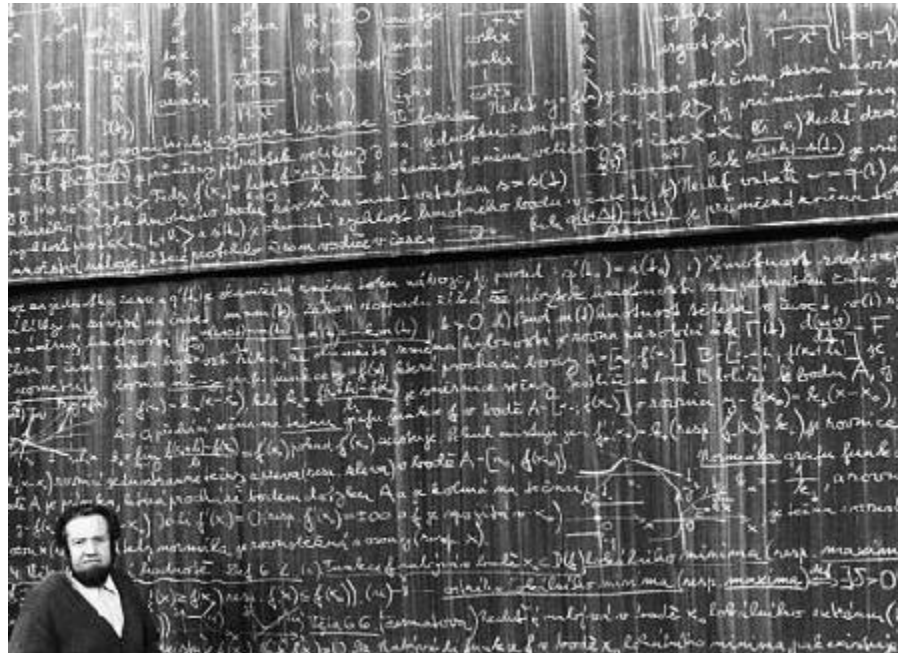
Информатика, ИТИС, 1 курс 2 семестр

М.М.Абрамский

2017

# ALERT

- MUCH MATH IN PRESENTATION!



**ЛИХИЕ 1930<sub>Е</sub>**

**В МАТЕМАТИЧЕСКОМ МИРЕ**

# 1930-е - Поворотные годы в области Theory of Algorithms:

- **Гёдель** доказывает теорему о неполноте – в формальной логической системе есть формула, которую невозможно вывести.
  - Аналог парадокса лжеца: «Я лгу» - истина или ложь?
- **Тьюринг** вводит понятие абстрактной вычислительной машины.

# 1930-е - Поворотные годы в области Theory of Algorithms:

- **Чёрч** вводит формальную систему для описания функций, в которых он формулирует существование алгоритмически неразрешимых задач.
- **Тьюринг** показывает существование невычислимых на машине Тьюринга задач – *проблема остановки*.

# 1930-е - Поворотные годы в области Theory of Algorithms:

- **Гёдель** обобщает модель рекурсивных функций, и доказывает, что частично рекурсивные функции совпадают с функциями, которые можно вычислить на машине Тьюринга.
- **Чёрч и Тьюринг** независимо формулируют **тезис**, об эквивалентности всех формальных моделей функций и интуитивным понятием вычислимости.

# 1930-е - Поворотные годы в области Theory of Algorithms:

- **Гёдель** обобщает модель рекурсивных функций, и доказывает, что частично рекурсивные функции совпадают с функциями, которые можно вычислить на машине Тьюринга.
- **Чёрч и Тьюринг** независимо формулируют **тезис**, об эквивалентности всех формальных моделей функций и интуитивным понятием вычислимости.
- *Вроде все... хотя нет! Мы кое-что забыли!*

# Чёрч (Church)

**в 1932** вводит формальную систему

*Возможно, у этой системы найдутся приложения не только в роли логического исчисления.*

Алонзо Чёрч, 1932 год



# НАШЛИСЬ!

## нашлись, не то слово!

*Программисты всего мира через много лет вспоминают его с благодарностью, ибо он привел в программирование подход, выражаемый одно греческой буквой.*

λ

Да и не только

# Напоминаю

- В начале XX века - кризис математики
  - Парадоксы (и потом Гёдель) напугали всех, поэтому строились новые формальные системы описания математических объектов
    - » В т.ч. и функций.

# *$\lambda$ -исчисление*

В отличие от машины Тьюринга, которая формализует понятие «вычисления», здесь формализуется само понятие «вычислимая функция».

В  *$\lambda$ -исчислении* 2 операции:

- ❖ **Аппликация** – применение функции к аргументу
- ❖ **Абстракция** – построение новых функций

# Аппликация

*$f a$*

«функция  $f$  применяется к значению  $a$ »

В классической математике:  $f(x)$

Но здесь она трактуется как «алгоритм  $f$ ,  
вычисляющий результат по значению  $a$ ».

# Абстракция

$\lambda x.t[x]$

«Новая функция с параметром  $x$  и телом  $t[x]$ »

- Да-да, это сейчас нам очевидно, что такое параметр и тело функции.
- А представьте, как бы вы это объяснили математику, не знакомому с программированием?

**НА ЧТО ПОХОЖИ АБСТРАКЦИЯ И  
АППЛИКАЦИЯ?**

# Ну конечно!

- Абстракция – объявление функции
- Аппликация – ВЫЗОВ

*Повторюсь: сейчас мы все умные, а 80 лет назад не было программирования (не говоря о понятиях «объявление-вызов»)*

# Идем дальше

- Комбинируя аппликацию и абстракцию мы можем получить более интересные выражения:

*$f g x$*

- Применяем  $f$  к  $g$ , а результат применяем к  $x$   
– *wait, what?*



# Да-да, вы не ошиблись

Результатом работы аппликации может быть новая функция, которая применяется к следующему значению

- $f g$  – новая функция, которая применяется к  $x$ .
  - Аналог в обычной математике?

# Сложная функция

$f\ g\ x$

в  $\lambda$ -выражениях

$f(g(x))$

в обычной математике.

# Разница

$$f \ g \ x$$
$$f(g(x))$$

Но в обычной математике сначала мы получим число  $g(x)$ , которое передадим в функцию  $f$ .

А тут мы сразу применяем  $f$  к  $g$ , и “ $f \ g$ ” – значение применения – является функцией!

- Еще раз другими словами - значением функции является **функция!**
- А если приглядеться, то можно увидеть, что и аргумент  $g$  – тоже **функция!**

# На секундочку в наш мир

Если функции можно передавать как параметры и возвращать как значения – значит они могут храниться в переменных.

- Примитивного или ссылочного типа?

# А почему бы и нет?

```
public class Function {  
    // имя функции  
    private String name;  
    // имена параметров  
    private List<String> paramNames;  
    // имена типов параметров  
    private List<String> paramTypeNames;  
    // имя возвращаемого типа  
    private String returnType;  
    // модификаторы метода: static, final, public...  
    private List<String> modifiers;  
}
```

# А почему бы и нет?

```
public class Function {  
    // имя функции  
    private String name;  
    // имена параметров  
    private List<String> paramNames;  
    // имена типов параметров  
    private List<String> paramTypeNames;  
    // имя возвращаемого типа  
    private String returnType;  
    // модификаторы метода: static, final, public...  
    private List<String> modifiers;  
}
```

Презентацию посетил призрак рефлексии. Посетил и ушел ждать вас на 2 курсе.

# Функция как объект

Когда	Где	Что
1980-е – 2000-е	C++	указатели на функции
	JavaScript	объекты класса Function
	Python	«все – объект!»
	Java	рефлексия
	C#	делегаты

# Функция как объект

Когда	Где	Что
1980-е – 2000-е	C++	указатели на функции
	JavaScript	объекты класса Function
	Python	«все – объект!»
	Java	рефлексия
	C#	делегаты
1932	<b>λ-исчисление Чёрча</b> <ul style="list-style-type: none"><li>• до появления первых компьютеров более <b>10</b> лет,</li><li>• до появления первого языка высоко уровня Fortran – <b>22</b> года</li><li>• до появления ООП – <b>35</b> лет</li></ul>	



# Интуитивно понятно

- Ок, да, функции можно не только объявлять и вызывать, но и передавать в другие функции как параметры.
  - Поговорим об этом чуть позже.
- Вернемся ненадолго вновь к  *$\lambda$ -исчислению*.

# ***$\lambda$ -выражения***

- Последовательности операций аппликации и абстракции:
  - Вызов функций, объявленных на функциях, которые вызывают другие функции, объявленные на функциях и т.д.

Двуместная функция на  $x, y$  с телом  $t$



**$\lambda x. \lambda y. t \quad v \quad w$**

*Опять не все так просто.*

# В чем дело

- Аппликация – одноместный оператор
  - Одна функция применяется к одному значению (всегда к одному!)
- Выражение  $\lambda x. \lambda y. t \ v \ w$   
воспринимается как  $(\lambda x. \lambda y. t \ v) \ w$ 
  - Это как вообще так?

# Каррирование (Currying)

Оператор, названный в честь Хаскелла Карри

- Да-да, язык Haskell тоже в его честь.

**Суть:** т.к. функции могут возвращать другие функции как результат, можно применить *многоместную* функцию к *одному* аргументу, считая, что в итоге получается новая функция, применяемая к следующему аргументу — и так далее...

# Наш пример

$(\lambda x. \lambda y. t \ v)$   $w$



Новая функция, которая получается подстановкой  $v$  в тело  $t$  вместо  $x$ .

Обозначается:  $\lambda y. [x \rightarrow v]t$

Полученную функцию (она одноместна) мы применяем к  $w$

# Реакция на происходящее в аудитории

1. *«Что происходит?»*
2. *«Это мы что, только что тратили время на определение того, что такое функция от двух параметров?»*

Вам всем поможет следующий слайд.

# ~~Это ж очевидно все~~

Нет. Не очевидно. Это у вас есть 10 лет школьной + 1 год университетской математики = к математическим абстракциям вы привыкли.

Формальные системы нужно строить **по непротиворечивым правилам**, которые не должны быть сложными.

- Если здравый смысл работает, то тогда дайте решение парадокса лжеца.
- Хотите другой эксперимент?

# Не забыли 1 класс школы?



Определение сложения.  
+ работает на двух числах.



# Тест

Чему равно «сумма(1, 4, 10, 20, 7)»?

# Тест

**Чему равно «сумма(1, 4, 10, 20, 7)»?**

*Мда? 42? И как вы узнали? Там же не написано «+», и слагаемых не 2. Запятые какие-то, скобки.*

# **+ и сумма**

Если показать детям, кто-то ответит, а кто-то нет. Размышление того, кто ответит:


- вспомнить, что такое «сумма»
- вспомнить, что сумма связана с оператором «+»
- Понять, что можно применить + к первым двум числам, а потом к результату суммы прибавить третье число, и т.д.

# Мышление

- И ведь мы говорим про ребенка человека, который обладает способностью думать и обучаться.
- А компьютер? Умеет думать? Абстрагировать?
  - А я напоминаю, что это 1930е,  $\lambda$ -исчисление, где под «функцией» подразумевается «вычислимая функция», «алгоритм».

# Каррирование наглядно без лямбд

$$\underline{\text{sum}}\ a\ b = \underline{(\text{sum}\ a)}\ b$$



Это двуместная функция  
обычной суммы

Это одноместная функция  
«прибавь a к числу»

# Переберемся из 1930х в 1950е

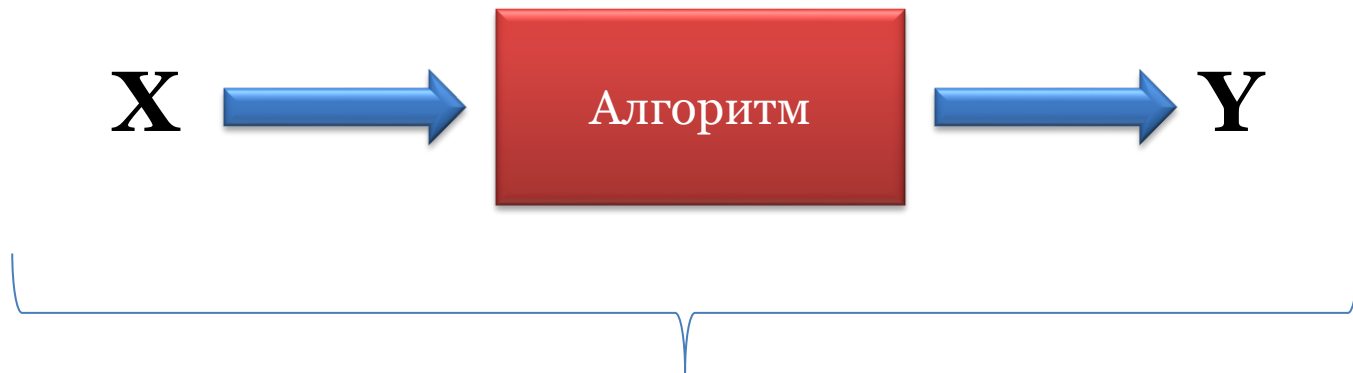
- Компьютеры уже есть.
- Развиваются языки программирования.
- Наибольшую популярность получают *процедурные императивные* языки программирования.

# Императивный. Процедурный. Твой.

- Программа – набор шагов, явно описывающих то, что нужно сделать и, главное – **как** (императивность)
- Шагом программы может быть использование «подпрограмм» – **процедур и функций** – **именованных и параметризованных** блоков кода, которые можно использовать несколько раз
  - Для этого их и придумывали.

# Но если в целом

- Любой алгоритм – преобразование входные данных в выходные.



На чье поведение это похоже?



# Функция

- Не функция в процедурном смысле (именованный параметризованный блок кода)
- Но функция в математическом смысле.
  - **В современном** математическом смысле

# Функциональный подход

- Декларативный
  - Не описывается последовательность шагов.
- Программа – суперпозиция функций.
  - При этом функции вычислимы
  - При этом функции могут быть как параметрами, так и результатами.
    - Что-то напоминает, да?

# Функциональное программирование

Языки: LISP, Haskell, Erlang, Clojure, Scheme, Scala, F#.

Процедурные языки к настоящему времени почти все (даже C++!) сдались и включают в себя элементы функционального программирования.

А его ключевой элемент — это ...

# LAMBDA

- Мы говорим «функциональное программирование» - подразумеваем «лямбда-выражения»!
- Мы говорим «лямбда-выражения» - подразумеваем «функциональное программирование»!

Лямбда-выражение – программная конструкция для объявления функциональных объектов (функций как объектов)

- По смыслу опирается на  $\lambda$ -исчисление.
- Хотя вы уже поняли, что  $\lambda$ -исчисление и на обычные функции в процедурном подходе повлиял.

# Не хватает одной полезной вещи

- **sum** – двуместная функция
- Выражение «sum x y z» – не будет тем, ЧТО МЫ ХОТИМ:
  - Мы хотим:  $x + y + z$ . А будет  $x + y$
  - $(\text{sum } x \ y)$  – это  $x + y$ . Число. Оно применяется как функция к  $z$ . Константа на любом аргументе равна себе самой, т.е.  $C(z) = C$ .

# list a b

- list a b – создает объект «список из a, b»
- Легко обобщается на n элементов:
  - (list a b) c = (list a b c)
- sumlist (list a b c) = sum a (sumlist (list b c))

Так и появился функциональный язык...

# LISP

- *LISt Processing language* – язык обработки списков.
- 1958 Дж.Маккарти
- По факту, имеет немного императивности, может даже ООП, но все равно в основе своей – функциональный язык программирования.

# Примеры кода на LISP

```
(* (+ 2 2) (- 11 1))
```

*Вычисление арифметического выражения*

```
(list 1 2 3)
```

*Возвращает список из (1, 2, 3)*

```
(defun inc(x) (+ x 1);
```

*Создаем функцию, увеличивающую число x на 1.*

```
inc 10
```

*Используем ее.*



# Примеры кода на LISP

```
(car (1 2 3))
```

*Возвращает голову списка - 1*

```
(cdr (1 2 3))
```

*Возвращает хвост списка – (2, 3)*

```
((lambda (x y) (+ x y)) 1 1)
```

*Лямбда-выражение – что делает здесь?*

# И т.д.

- Вообще функциональный подход тесно связан со списками
  - Это мы заметим в будущем.
- А еще интереснее, когда в языке есть и процедурный подход, и функциональный.
  - Посмотрим классический пример такого языка.

# Python (ну очень коротко)

- Интерпретатор
- Операторный скобки = отступы
- Утиная типизация
  - чему присвоил, такой и тип у переменной

*# это - комментарий*

*x = 5 # я - целое число*

*x = "s" # нет, я передумал, я - строка*

*x = 0.5 # нет, я все же число, но вещественное*

## 2 структуры данных

- Список - пронумерованный набор данных разного типа (аналог в Java - List<Object>)

```
lst = ['Airat', 'Nastya', 'Mark']  
print(len(lst)) # 3  
print(lst[0]) # Airat  
print(lst[-1]) # Mark
```

- Строка – аналог String.

# УСЛОВИЯ

- Классический if

*Обязательные 4  
пробела отступа –  
без них не  
запустится.*

```
if y > 0:  
    print(1)  
elif y < 0:    # это else if  
    print(-1)  
else:  
    print(0)
```

- Тернарный оператор:

`x = a if a > b else b`

Аналог Java:

`x = a > b ? a : b;`

# Циклы

- Классический **while**

- Обязательные 4 пробела  
отступа – без них не запустится.

```
while i < 10:  
    i += 1  
    print(i)
```

- **for** (в питоне for each)

- только for each

```
lst = ['Airat', 'Nastya', 'Mark']  
for name in lst:  
    print(name)
```

# Python - функции

Объявление обычной функции

```
def mul5(x):  
    return x * 5
```

Утиная типизация дает нам **«суперполиморфизм»**

- *Это не термин, это так, эмоции.*

```
print(mul5(10)) # 50  
print(mul5("ITIS")) # ITISITISITISITISITIS  
print(mul5(['a', 'z']))  
# ['a', 'z', 'a', 'z', 'a', 'z', 'a', 'z', 'a', 'z']
```

# Функция - объект

```
def mul5(x):  
    return x * 5
```

```
def inc(x):  
    return x + 1
```

```
f = inc  
print f(10)
```

Объект inc присвоен  
переменной f, значит – f –  
функция, ее можно вызвать.

```
def apply(f, x):  
    return f(x)
```

```
print(apply(inc, 10))
```

```
print(apply(len, [1, 2, 3]))
```

apply принимает функцию в  
качестве аргумента

Можно также и возвращать  
функцию-объект



# Lambda в Python

- Тривиально:

```
lambda x, y: x + y
```

- **Анонимная функция**
- Эти объекты – полноценные функции – их тоже можно присваивать и передавать.

# Пример

```
def s (x, y) :  
    return x + y
```

```
s = lambda x, y: x + y
```

Два идентичных объявления функции

Присвоил анонимную функцию переменной s – она получила название.

# Пример 2

```
def apply(f, x):  
    return f(x)
```

```
print(apply(lambda x: x * x, 10))
```

Что делает код?

# Пример 3 - замыкание

```
y = 100
g = lambda x: x * y
print(g(4)) # выведется 400
```

**g – замыкание** – использует не только параметры, но и свободные переменные из того же контекста, где она была объявлена.

# И последнее - генераторы списков

- `range(a,b,c)` – генерирует числа от `a` до `b` (не включительно с шагом `c`)
  - `range(a,b)` с шагом 1
  - `range(a)` – от 0 до `a` с шагом 1
- Генератор общего вида:

```
[input() for i in range(10)] # input() - УТИНЫЙ ВВОД
```

Что делается здесь?

# Ну вы понимаете

- Иные задачи в Python можно решать в одну строчку.
- *Вводится список из 10 чисел, вывести список, где все числа уменьшены на 1:*

```
print([input() - 1 for i in range(10)])
```

# Но хочется еще больше

- Мы хотим обрабатывать списки в функциональном стиле, а не в процедурном:
  - **Процедурный стиль** – перебрать в цикле все элементы, применить к каждому элементу операцию, построить новый список.
  - **Функциональный стиль** – применить операцию сразу ко всему списку, получив новый список
    - Т.е. аппликация к list

# Простая задача

Вводится  $n$ , затем  $n$  чисел, нужно построить список, где хранятся квадраты исходных чисел.

## *Java*

```
int n = scanner.nextInt();
List<Integer> lst = new ArrayList<>();
for (int i = 0; i < n; i++) {
    int x = scanner.nextInt();
    lst.add(x);
}
System.out.println(lst);
```

## *Python*

```
n = input()
array = []
for i in range(n):
    x = input()
    array.append(x * x)

print(array)
```



# Дисклеймер

- Дальше я буду считать, что мне нужно, чтобы список вводился полностью и сохранялся в программе в виде списка.
- Вы можете сказать (справедливо), что для этой задачи не нужно хранить набор исходных чисел.
- Я с вами согласен, но сейчас мы решаем не задачу «оптимальной обработки», а задачу использования конкретных решений — а они используются чаще всего при обработке коллекций данных, которые уже **введены**.

# Код

## *Java*

```
int n = scanner.nextInt();
List<Integer> lst = new ArrayList<>();
for (int i = 0; i < n; i++) {
    int x = scanner.nextInt();
    lst.add(x);
}
System.out.println(lst);
```

## *Python*

```
n = input()
array = []
for i in range(n):
    x = input()
    array.append(x * x)

print(array)
```

А хочется что-нибудь типа:

**СПИСОК = ВЫЧИСЛИТЬ КВАДРАТЫ (ИСХОДНЫЙ НАБОР ЧИСЕЛ)**

**(ОБРАБОТКА ДАННЫХ В ДЕКЛАРАТИВНОМ СТИЛЕ)**

мы ведь понимаем, как это устроено (см пример с sum)

# «Льзя!»

- Специальные функции, которые нам помогут решать такие задачи компактно: `map`, `reduce`, `filter`
- **Функции высшего порядка**

# map

- Применяет функцию к списку, получаем новый список, где каждый элемент – значение функции от исходного элемента на той же позиции
- Он нам и нужен:

**map(квадрат\_числа, [1, 3, 5]) = [1, 9, 25]**


# В Python

```
result_list = map(fuction, original_list)
```

**Для нашего примера:**

```
n = input()
original_list = []
for i in range(n):
    original_list.append(input())
```

*Это все ввод исходного списка  
[1, 2, 3, 4]*



```
result_list = map(lambda x: x * x, original_list)
```

```
print(result_list)
```

*← Это уже вывод - [1, 4, 9, 16]*

# reduce

- Функция применяет к списку двуместную функцию, получая значение функции от всего списка
  - Примеры `sum(list)`, `max(list)`, `min(list)`
- Вот как раз давайте найдем максимум
  - Список уже будет введенным

# Пример

```
original_list = [100, 10, 50, 4, 200, 7, 146]  
maximum = lambda a, b: a if a > b else b  
result = reduce(maximum, original_list)  
print(result)
```

***В одну строчку:***

```
print(reduce(lambda a, b: a if a > b else b, original_list))
```

# filter

- Отбирает из списка элементы, удовлетворяющие определенному условию

```
result_list = filter(boolean_predicate, original_list)
```



# Пример

- Взять только четные числа

```
result_list = filter(lambda x: x % 2 == 0, original_list)
```

# Обобщим

Что делает такую обработку коллекций клевой?

- Функции как объекты + Лямбды
- Функции высших порядков: map, reduce, filter.

Это все очень заманчиво, и с 2014 года в Java 8 появились **лямбда-выражения** и **Stream API**, позволявший работать с коллекциями используя функции высших порядков.

# Но с Java не все легко

- У нас нет функций
- Методы – не объекты
- Коллекции не предназначены для функций высших порядков

# Суть

- Лямбда-выражения – это анонимные функции (по аргументам мы что-то делаем)
- У нас нет анонимных функций, но есть анонимные классы.
- В анонимных классах неанонимные методы
- Но если метод в таком классе – один, то он может играть роль анонимного
  - Потому что раз он один, то справшивая «метод класса» у нас только один вариант ответа.

# T.e.

```
interface I1 {  
    void f1 ();  
    void f2 ();  
}
```

- Видишь метод из I1?
- Какой именно?

```
interface I2 {  
    void f1 ();  
}
```

- Видишь метод из I2?
- Вижу!



Такие интерфейсы называют функциональными.

# Функциональный интерфейс

- Интерфейс с единственным абстрактным методом
  - default не считается.
  - если метод интерфейса совпадает по сигнатуре с методом из Object, он тоже не считается.

# Lambda в Java

- Лямбда выражения в Java – это объект анонимного класса с единственным методом, реализацию которого мы и пишем (это тело lambda-функции)
- Синтаксис: (параметры) -> {тело}

# Пример

- С прошлой пары – Comparator<T>
  - Интерфейс с единственным абстрактным методом compare
- Сравнивать будем опять студентов

```
public class Student {  
    private int year;  
    private String name;  
    // + все необходимые get-set и конструкторы  
}
```



# Слайд с прошлой пары

```
ArrayList<Student> students = new ArrayList<>();  
students.add(new Student("11-602", "Gabdreeva", 96));  
students.add(new Student("11-601", "Mingachev", 94));  
students.add(new Student("11-604", "Romanov", 94));  
students.add(new Student("11-605", "Bagautdinov", 94));  
students.add(new Student("11-603", "Nurgatina", 96));
```

```
Collections.sort(students, new Comparator<Student>() {  
    @Override  
    public int compare(Student o1, Student o2) {  
        return o1.getGroup().compareTo(o2.getGroup());  
    }  
});  
System.out.println(students);
```

# А теперь

```
ArrayList<Student> students = new ArrayList<>();  
students.add(new Student("11-602", "Gabdreeva", 96));  
students.add(new Student("11-604", "Romanov", 94));  
students.add(new Student("11-601", "Mingachev", 94));  
students.add(new Student("11-605", "Bagautdinov", 94));  
students.add(new Student("11-603", "Nugratina", 96));  
  
Collections.sort(students, (s1, s2)  
    -> {  
        return s1.getGroup().compareTo(s2.getGroup());  
    });  
System.out.println(students);
```

# Не совсем ФП

- Тело = тело метода – там можно реализовывать последовательность действий.
- Но нам нужно не чистое ФП, а ВОЗМОЖНОСТИ.

# Stream API

- Декларативная обработка коллекций с помощью функций высших порядков.
- Итак, с Java 8 все коллекции оснастили методом `stream()`, возвращающим объект `Stream`, который может быть аргументом функций высших порядков.
  - Стримы также можно создавать по файлам, по массивам, по набору значений и т.д.

# Методы работы со Stream

- Конвейерные – возвращают новый Stream
  - **map**, **filter**,
  - **distinct** (убирает дубликаты),
  - **sorted** (сортировка по Comparable/Comparator), **mapToInt**, **mapToDouble** (возвращает числовой Stream)
  - и др.
- Терминальные – возвращают значение.
  - **findFirst** (возвращает первый),
  - **collect** (конвертирует Stream в нужный тип данных),
  - **forEach** (применяет функцию к каждому объекту Stream),
  - **reduce**
  - и др.

# Optional<T>

- Особое значение, возвращаемое некоторыми терминальными методами (напр., `findAny`)
- Тип, необходимый для избегания `NullPointerException`

# Использование

```
String s = ...; // либо null, либо нет.  
Optional<String> value = Optional.ofNullable(s);
```

Далее я работаю с `value`, давая возможность задать default значение, возвращаемое, если будет `null`, а также защищая от `NullPointerException`

# Использование

```
String s = null;  
Optional<String> value = Optional.ofNullable(s);  
System.out.println("Hi, " + value); // выведется Optional[Empty]  
System.out.println("Hi, " + value.get()); // Exception!!  
  
String st = "HAHA";  
System.out.println("Hi, " + value.orElseGet(() -> st));  
System.out.println(value.map(x -> "Oh," + x).orElse("Hi, anon"));
```

***Hi, Optional.empty***

***Hi, HAHA***

***Hi, anon***



# Использование

```
String s = "Roman"; // либо null, либо нет.  
Optional<String> value = Optional.ofNullable(s);  
System.out.println("Hi, " + value);  
System.out.println("Hi, " + value.get());  
String st = "HAHA";  
System.out.println("Hi, " + value.orElseGet(() -> st));  
System.out.println(value.map(x -> "Oh," + x).orElse("Hi, anon"));
```

***Hi, Optional[Roman]***

***Hi, Roman***

***Hi, Roman***

***Oh, Roman***

# Примеры StreamAPI

```
final Map<Integer, List<Student>> map = students.stream()  
    .collect(Collectors.groupingBy(Student::getAverageScore));  
  
System.out.println(map);
```

```
{96=[Gabdreeva{group='11-602', score=96}, Nugratina{group='11-603', score=96}],  
94=[Romanov{group='11-604', score=94}, Mingachev{group='11-601', score=94},  
Bagautdinov{group='11-605', score=94}]}
```

# Примеры StreamAPI

```
String [] s = new String[] { "Sema", "Sanya", "Sonya" };  
String res = Arrays.stream(s)  
    .reduce((x, y) -> x + ", " + y).orElse("");  
System.out.println(res);
```

***Sema, Sanya, Sonya***

# Примеры StreamAPI

```
double averageScore = students
    .stream()
    .mapToInt(x -> x.getAverageScore())
    .average()
    .orElse(0);
System.out.println(averageScore);
```

**94.8**

# Ключевые слова

*Optional, Stream API, лямбда выражения, функциональное программирование*

# Почитать

Лямбда-исчисление:

<https://habrahabr.ru/post/215807/>

Stream API

<https://habrahabr.ru/company/luxoft/blog/270383/>