НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

JAVA 8: Новое λ-выражения и Класс Stream

Евгений Беркунский, НУК

eugeny.berkunsky@gmail.com

http://berkut.mk.ua

Лямбда-выражения

Кому из вас приходилось писать такой код?

```
myButton.addActionListener(
    new ActionListener() {
        public void actionPerformed(ActionEvent e){
            doSomething(e);
        }
    }
);
```

Лямбда-выражения

А как бы его написать короче?

```
myButton.addActionListener(e -> doSomething(e));
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Лямбда-выражения

`@FunctionalInterface`

Интерфейс с одним абстрактным методом:

- Comparable
- Comparator
- Runnable
- Callable
- ActionListener
- …

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Лямбда-выражения

`@FunctionalInterface`

Виды FunctionalInterface

- Predicate<T>
- Function<T,R>
- BiFunction<T,U,R>
- Producer<T>
- Consumer<T>
- DoubleUnaryOperator
- DoubleBinaryOperator
- IntUnaryOperator
- IntBinaryOperator
- …

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Streams

1. filter(Predicate)
2. allMatch(Predicate)
3. anyMatch(Predicate)
4. min(Comparator)
5. max(Comparator)
7. reduce(T, BiFunction)
8. forEach(Consumer)
9. sorted(Comparator)
10. collect(Collector)
    - 10.1 Reduction to List or Set
    - 10.2 Statistics
    - 10.3 Reduction to Map
        - 10.3.1 groupingBy(Function) and partitioningBy(predicate)
        - 10.3.2 GroupingBy(Function, Collector)
        - 10.3.3 GroupingBy(Function, Supplier, Collector)

Введение

- Релиз Java 8 представляет огромные изменения по сравнению с предыдущими версиями.
- Эта лекция преследует цель сделать обзор главной функциональности нового типа Stream.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Введение

Для рассмотрения примеров будем использовать тип Flight, интерфейс которого приведен ниже:

```
public interface Flight extends Comparable<Flight>{
    String getCode();
    String getDestination();
    LocalDate getDate();
    Integer getNumPassengers();
    Integer getNumSeats();
    Duration getDuration();
    Double getPrice();
    Double getOccupation();
    void setPrice(Double p);
    void setDuration(Duration d);
    void setNumPassengers(Integer np);
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Введение

- В наших примерах будет реализован тип Airport, который будет манипулировать коллекцией объектов типа Flight (мы будем называть их «рейсами»).
- Эти объекты требуют поддержки определенных операций, рассматриваемых в каждой из частей презентации.
- Сложностью может быть то, что здесь будут использованы типы **Predicate**, **Function**, **Supplier** и т.п., которые не рассматривались ранее.

Класс Stream

- В этой презентации мы сфокусируемся на структуре данных Stream, которая представляет собой виртуальную последовательность объектов.
- Метод stream(), примененный к коллекциям, возвращает объект Stream из объектов коллекции.
- Именно поэтому фрагменты кода на следующих слайдах всегда будут начинаться с flights.stream().
- Далее рассмотрим главные операции класса Stream.

1. filter(Predicate)

- Используется для получения другого **Stream** состоящего только из элементов исходного **Stream**, которые удовлетворяют условию, которое определяется с помощью **Predicate**.
- Этот метод обычно сочетается с такими методами, как count(), для подсчета количества элементов коллекции, удовлетворяющих условию.

1. filter(Predicate)

- Например, сколько есть рейсов в указанную дату:

```
public Long getNumFlightsDay(LocalDate f) {
    return flights.stream()
        .filter(x->x.getDate().equals(f))
        .count();
}
```

- Или, сколько есть полностью загруженных рейсов:

```
public Long getNumFullFlights() {
    return flights.stream()
        .filter(x -> x.getNumPassengers()
        .equals(x.getNumSeats()))
        .count();
}
```

1. filter(Predicate)

- Если объект типа Predicate планируется использовать несколько раз, его стоит сначала определить, а затем вызывать.
- Например, условие что рейс выполняется в указанную дату, будет определено так:

```
Predicate equalDate(LocalDate f){
    return x -> x.getDate().equals(f);
}
```

- А потом его можно использовать так:

```
public Long getNumFlightsDate(LocalDate f) {
   return flights.stream()
         .filter(equalDate(f))
         .count();
}
```

1. filter(Predicate)

*Также метод **filter(Predicate)** может сочетаться с методами типа **findFirst**, которые возвращают первый объект потока **stream**, или **limit(long)**, который возвращает поток **stream**, ограниченный указанным количеством элементов*

2. allMatch(Predicate)

- Возвращает **true**, если все элементы потока stream удовлетворяют указанное условие.
- Например, мы хотим узнать все ли рейсы в указанную дату заполнены:

```
public Boolean allFull(LocalDate f){
    return flights.stream()
         .filter(equalDate(f)).
         .allMatch(x -> x.getNumPassengers()
         .equals(x.getNumSeats()));
}
```

2. allMatch(Predicate)

- Если нужно, Predicate из фильтра метода allMatch может быть объединен с другим (другими) с помощью метода “and” интерфейса Predicate.

```
Predicate flightFull =
        x -> x.getNumPassengers().equals(x.getNumSeats());

public Boolean allFull2(LocalDate f){
    return flights.stream()
                .allMatch(equalDate(f).and(flightFull()));
}
```

Примечание. *Метод allFull2 работает не так, как предыдущий. Различие в том, что в первом случае, если allFull выполняет фильтрацию и ее результат – пустой поток, allMatch вернет **true**. В случае второго метода, если ни один рейс не удовлетворяет условию, результат будет **false**.*

3. anyMatch(Predicate)

- Возвращает true, если хотя бы один элемент потока stream удовлетворяет заданному Predicate.
- Например, узнать есть ли рейс в указанный пункт назначения в требуемую дату:

```
public Boolean existsFlightDestinationDate(
        LocalDate f, String d){
    return flights.stream()
        .anyMatch(x->x.getDate().equals(f) &&
            x.getDestination().equals(d));
}
```

3. anyMatch(Predicate)

- Проверить, есть ли незаполненный рейс на указанную дату, если уже есть определенные предикаты equalDate и flightFull:

```
public Boolean existsFlightDateNotFull(LocalDate f){
        return flights.stream()
                .anyMatch(equalDate(f)
                .and(flightFull().negate()));
}
```

4. min(Comparator)

- Возвращает минимальный элемент коллекции согласно порядку, устанавливаемого с помощью объекта Comparator.
- Для определения класса, реализующего Comparator классов по любым из его атрибутов удобно использовать метод **comparing** из интерфейса Comparator.
- Например, самый дешевый рейс в указанном направлении вычисляемый сравнивая два объекта типа Flight по их цене (Price) используя функцию (Function) Flight::getPrice:

```
public Flight getCheaperFlightDestination(String d){
       return flights.stream()
              .filter(x->x.getDestination().equals(d))
              .min(Comparator.comparing(Flight::getPrice))
              .get();
}
```

4. min (Comparator)

- Интерфейс Comparator позволяет указывать тип сравниваемых объектов.
- Таким образом, в предыдущем примере должен быть использован метод comparingDouble, так как метод getPrice возвращает значение типа double.
- Кроме того, в интерфейсе Comparator есть методы comparingInt и comparintLong. Их предназначение понятно из названия.
- Мы могли бы также использовать естественный порядок сортировки (реализованный методом CompareTo) с вызовом:

```
min(Comparator.naturalOrder());
```

4. min (Comparator)

- Метод min возвращает объект нового в Java 8 типа Optional, потому что результат этого метода может и не существовать: рассмотрим случай расчета минимального элемента пустой коллекции, например, если нет ни одного рейса в пункт назначения.
- Метод get вызывается на последней стадии. Он возвращает элемент, если он существует, и бросает NoSuchElementException в противном случае.
- Эта возможность может использоваться с другими доступными методами после вызова min, как, например, метод ifPresent, который возвращает true, если есть значение в объекте Optional, или методом orElse, что позволяет создать и вернуть объект.

5. max (Comparator)

- Как и в предыдущем случае, но с максимумом вместо минимума. Например, если мы хотим определить наиболее загруженный рейс для указанной даты, мы можем реализовать метод:

```
public Flight getFlightHigherOccupation(LocalDate f) {
    return flights.stream()
        .filter(x->x.getDate().equals(f))
        .max(Comparator.comparingDouble(
              x -> x.hetNumPassengers() / x.getNumSeats()))
        .get();
}
```

5. max (Comparator)

- На предыдущем слайде FunctionDouble передавалась в качестве параметра методу comparingDouble.
- Другое решение было бы возможно, если бы рейс содержал метод, возвращающий его загруженность. В этом случае код будет выглядеть так:

```
public Flight getFlightHigherOccupation(LocalDate f) {
    return flights.stream()
        .filter(x->x.getDate()
        .equals(f))
        .max(Comparator.comparingDouble(
             Flight::getOccupation))
        .get();
}
```

5. max (Comparator)

- Логично, если тип Flight не содержит такого метода, мы можем определить объект Function для него:

```
Function getOccupation =
        x -> 1.*x.getNumPassengers()/x.getNumSeats();
```

- Или объект ToDoubleFunction:

```
ToDoubleFunction getOccupation =
        x -> 1.*x.getNumPassengers()/x.getNumSeats();
```

- Использовать этот объект можно так:

```
public Flight getFlightHigherOccupation(LocalDate f) {
  return flights.stream()
        .filter(x->x.getDate().equals(f))
        .max(Comparator.comparingDouble(getOccupation))
        .get();     // ToDoubleFunction   ^^^
}
```

6. map(Function)

- Это на самом деле семейство методов, с адаптациями mapToInt (ToIntFunction), mapToLong (ToLongFunction) и mapToDouble (ToDoubleFunction).
- Все они возвращают поток объектов другого типа, полученного из базового типа применением функции. Если возвращаемый тип – это Integer, Long или Double, для этого есть методы с соответствующими функциями в качестве аргумента.

6. map(Function)

- Таким образом, если бы в предыдущем случае нас интересовал бы не рейс с наибольшей загрузкой, а непосредственно сама наибольшая загрузка рейса, можно было бы написать так:

```
public Double getHigherOccupation(LocalDate f) {
    return flights.stream()
        .filter(x->x.getDate().equals(f))
        .mapToDouble(x ->
                    x.getNumPassengers()/x.getNumSeats())
        .max()
        .getAsDouble();
}
```

6. map(Function)

- Или, с использованием ранее описанной ToDoubleFunction:

```
public Double getHigherOccupation(LocalDate f) {
    return flights.stream()
        .filter(x->x.getDate().equals(f))
        .mapToDouble(getOccupation())
        .max().getAsDouble();
}
```

- Методы map() могут также упростить ряд операций над возвращаемыми значениями. Например, если нам нужно знать общую выручку с рейсов для определенного пункта назначения, мы можем написать:

```
public Double sumBillingDestination(String d) {
   return flights.stream()
        .filter(x->x.getDestination().equals(d))
        .mapToDouble(x->x.getNumPassengers()*x.getPrice())
        .sum();
}
```

6. map(Function)

- Вместо получения суммы, мы можем найти среднее значение с помощью метода, который называется average. Например, определим функцию ToDoubleFunction:

```
ToDoubleFunction getBilling = x->
                        x.getNumPassengers()/x.getPrice();
```

- Средняя выручка для заданной даты будет такой:

```
public Double averageBillingDate(LocalDate f) {
    return flights.stream()
            .filter(equalDate(f))
            .mapToDouble(getBilling)
            .average()
            .getAsDouble;
}
// average returns an OptionalDouble
// just in case there are no elements
```

7. reduce(T, BiFunction)

- Reduce производит «сведение» потока в соответствии с операцией, описанной в BiFunction, и используя первый аргумент как нейтральный (нулевой). Сумма и среднее значение являются частными случаями reduce из числовых типов.
- Для других типов, созданных программистом, например, Duration, reduce может быть использована для нахождения длительности полетов в определенный день:

Класс Duration имеет метод суммы, способный возвращать сумму двух Duration. Он называется plus()

7. reduce(T, BiFunction)

Метод reduce вызывается с двумя аргументами: первый является нейтральным элементом, который инициализирует аккумулятор, а второй является операцией, которая накапливается:

```
public Duration getDurationFlightsDate(LocalDate f){
    return flights.stream().
            filter(x->x.getDate().equals(f)).
            map(Flight::getDuration).
            reduce(Duration.ZERO, Duration::plus);
}
```

7. reduce(T, BiFunction)

Еще одним способом сделать то же самое является определение BiFunction в коде. В этом случае, поскольку три значения, используемые в определении типа Duration, это частный случай BinaryOperator, которые могут быть определены как:

```
BinaryOperator<Duration> sumdur = (x,y)-> x.plus(y);
```

Даже если Duration не реализует функциональность суммирования, это может быть сделано в коде:

```
BinaryOperator<Duration> sumdur = (x,y)-> {
    return x.plus(y);
};
```

В этом случае, вызов метода reduce будет таким:

```
reduce(Duration.ZERO, sumdur);
```

8. forEach(Consumer)

- Метод forEach используется для выполнения действий, определенных в объекте потребителе, переданном в качестве аргумента по всем элементам потока.
- Например, чтобы изменить длительность полетов к определенному пункту назначения на определенное количество минут, метод будет использоваться так:

```
public void incrementDurationDestination(String d, Integer min){
    flights.stream()
            .filter(x->x.getDestination().equals(d))
            .forEach(x->x.setDuration(x.getDuration()
                  .plus(Duration.ofMinutes(min))));
}
```

8. forEach(Consumer)

- Или для увеличения цены на рейсы позднее определенного дня на 10%:

```
Consumer<Flight> incrementPrice10p =
        x -> x.setPrice(x.getPrice()*1.1);

public void incrementPrices10pfromDate(LocalDate f) {
       flights.stream().
       filter(x->x.getDate().compareTo(f)>0).
       forEach(incrementPrice10p);
}
```

8. forEach(Consumer)

- Этот метод также может быть использован, например, для реализации статического метода, который будет записывать в текстовый файл элементы потока, по одному для каждой строки:

```
public static <T> void writeFile(Stream<T> it, String filename){
   File file = new File(filename);
   try {
       PrintWriter ps = new PrintWriter(file);
       it.forEach(x->ps.println(x));
       ps.close();
  } catch (FileNotFoundException e) {
       System.out.println("File not found "+filename);
  }
}
```

9. sorted(Comparator)

- Метод sorted возвращает поток упорядоченный в соответствии с компаратором, переданным в качестве параметра.
- Если порядок не указан - используется естественный.
- Например, повторно использовав предыдущего статического метода для записи потока в файл, мы могли бы реализовать следующий метод, который записывает поток упорядоченный по дате и продолжительности.

```
public void writeFlightsOrderedDateDuration (String fileName){
       Util.writeFile(flights.stream().
             sorted(Comparator.comparing(Flight::getDate).
             thenComparing(Flight::getDuration)), fileName);
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

10. collect(Collector)

- Функциональность collect - очень мощный инструмент для преобразования потока в коллекцию, отображение или данные.
- Некоторые из этих операций может быть осуществлено с помощью ранее рассмотренных методов, таких как reduce.
- Поскольку существует большое количество возможностей, мы разделим их в соответствии с их целью.

10.1 Reduction to List or Set

- Метод collect обеспечивает очень мощный инструмент для преобразования потока в другое хранилище данных, например, в список или множество.
- Например, если мы хотим получить список с длительностью рейсов в определенный пункт назначения, мы должны написать:

```
public List<Duration> getDurationsDestination(String d){
    return flights.stream().
       filter(x -> x.getDestination().equals(d)).
       map(x -> x.getDuration()).
       collect(Collectors.toList());
}
```

10.1 Reduction to List or Set

- Класс Collectors содержит методы toList и toSet, что означает, что поток длительностей, полученных операцией map может быть преобразован в список или множество объектов соответственно. Например, множество возможных направлений из аэропорта предоставляется таким методом:

```
public Set<String> getDestinations(){
    return flights.stream().
       map(x -> x.getDestination()).
       collect(Collectors.toSet());
}
```

toList и toSet методы являются частными случаями метода toCollection, который принимает Producer, например, вызов конструктора типа. Таким образом, мы можем вернуть SortedSet возможных направлений в определенный день:

10.1 Reduction to List or Set

- toList и toSet методы являются частными случаями метода toCollection, который принимает Producer, например, вызов конструктора типа.
- Таким образом, мы можем вернуть SortedSet возможных направлений в определенный день:

```
public SortedSet<String> getDestinationsInDate(LocalDate f){
   return flights.stream().
       filter(x -> x.getDate().equals(f)).
       map(x -> x.getDestination()).
       collect(Collectors.toCollection(TreeSet::new));
}
```

10.1 Reduction to List or Set

- Мы также можем преобразовывать объекты перед сбором их.
- Например, если мы хотим, чтобы получить список длительностей рейсов в определенный пункт назначения увеличенных на м минут, следующий метод подойдет:

```
public List<Duration>
        getIncrementedDurationsDestination(String d, Integer m) {
   return flights.stream().
       filter( x-> x.getDestination().equals(d)).
       map(x -> x.getDuration().plus(Duration.ofMinutes(m))).
       collect(Collectors.toList());
}
```

10.2 Statistics

- Если нам нужно только узнать количество элементов потока мы можем использовать Collectors.counting().
- Кроме того, если поток содержит числовые данные, коллекторы Collectors.summingDouble (summingInt или summingLong) могут быть использованы, чтобы вернуть сумму потока с соответствующим типом, и Collectors.averagingDouble (averagingInt или averagingLong), которые возвращают среднее арифметическое, всегда в типе Double.

10.2 Statistics

- Также существует тип DoubleSummaryStatistics, он может быть использован для получения значения суммы, максимального, минимального и среднего ряда числовых значений.
- Такой объект возвращается методом Collectors.summarizingDouble, который получает ToDoubleFunction (summarizingInt и summarizinLong также доступны).
- Например, чтобы получить среднее значение цен на рейсы в определенный пункт назначения, мы должны написать:

```
public Double getAveragePricesDestination(String d){
   return flights.stream().
       filter(x->x.getDestination().equals(d)).
       collect(Collectors.summarizingDouble(Flight::getPrice)).
       getAverage();
}
```

10.2 Statistics

- Когда обычные арифметические операции среднего и суммы не доступны, потому что базовый тип не является числовым типом данных (Double, Integer и т.д.), нужно использовать механизм reduce чтобы внедрить новые операции над этим базовым типом.
- Например, если мы хотим, получить сумму длительностей всех рейсов в определенный пункт назначения увеличенных на м:

```
public Duration incrementAndAddDestinationDuration(
                                                String d, Integer m){
   return flights.stream()
        .filter(x->x.getDestination().equals(d))
        .map(x->x.getDuration().plus(Duration.ofMinutes(m)))
        .collect(Collectors.reducing(
              Duration.ZERO, Duration::plus));
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

10.3 Reduction to Map

- Класс Collections предоставляет методы partitioningBy и groupingBy преобразования информации потока в Map.
- Существует большое количество возможностей, так как groupingBy можно использовать различные параметры.

10.3.1 groupingBy (Функция) и partitioningBy (предикат)

- В первом случае, функциональность groupingBy получает функцию, в то время как partitioningBy является частным случаем, в котором функция заменяется на предикат.
- В этом случае, итоговым результатом является Список с объектами потока.
- Рассмотрим различные способы использования этого:
  - Например, чтобы построить Map<Boolean, List <Flight>> разделить рейсы аэропорта в зависимости от того, являются ли они заполненными или нет, мы должны написать:

```
public Map<Boolean, List<Flight>> getMapFullFlight(){
   return flights.stream().
        collect(Collectors.partitioningBy(flightFull()));
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

10.3.1 groupingBy (Функция) и partitioningBy (предикат)

- Более сложный случай будет, если нужно разделить рейсы на Map<Boolean,List<Flight>> в соответствии с тем, прошел ли рейс заданный порог загруженности или нет:

```
public Map<Boolean, List<Flight>> getMapFlightsPercentage(Double p){
    return flights.stream().
       collect(Collectors.partitioningBy(
       x->x.getOccupation().compareTo(p/100)>=0));
}
```

Чтобы получить Map с более чем двумя значениями на оригинальном Map необходимо использовать groupingBy. Таким образом, чтобы получить Map, который организует полеты на сегодняшний день мы должны написать:

```
public SortedMap<Date,List<Flight>> getMapFlightsByDate(){
    return flights.stream().
       collect(Collectors.groupingBy(Flight::getDate));
}
```

10.3.2 GroupingBy(Function, Collector)

Конечно, окончательный набор элементов для Map не обязательно должен быть списком объектов потока, но он также может быть множеством, если, помимо функции, мы даем метод groupingBy класса Collector для окончательного множества:

```
public Map<String, Set<Flight>> getMapFlightSetByDestination(){
   return flights.stream().
       collect(Collectors.groupingBy(
               Flight::getDestination,
              Collectors.toSet()));
}
```

10.3.2 GroupingBy(Function, Collector)

Если информация в итоговом наборе данных является редукцией или операцией над элементами потока, необходимо использовать другие объекты типа коллектора в качестве параметра для groupingBy.
Таким образом, с помощью методов класса Collectors: counting, averaging или summing мы можем получить редукцию типа конечного набора или отображения.
Например, чтобы получить число рейсов в пункт назначения, требуется метод:

```
public Map<String, Long> getMapNumFlightsPerDestination(){
   return flights.stream().
       collect(Collectors.groupingBy(
             Flight::getDestination,
             Collectors.counting()));
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

10.3.2 GroupingBy(Function, Collector)

Кроме того, если мы хотим получить отображение со средней ценой для каждого места назначения:

```
public Map<String, Double> getMapAveragePricePerDestination(){
    return flights.stream().
       collect(Collectors.groupingBy(
             Flight::getDestination,
             Collectors.averagingDouble(
                          Flight::getPrice)));
}
```

Или отображение счетов для каждой даты:

```
public Map<Date, Double> getMapBillingPerDate(){
   return flights.stream().
       collect(Collectors.groupingBy(
             Flight::getDate,
             Collectors.summingDouble(
             (Flight x)->x.getNumPassengers()*x.getPrice())));
}
```

10.3.3 GroupingBy(Function, Supplier, Collector)

Использование объекта Supplier позволяет нам возвращать SortedMap как результат операции groupingBy. Таким образом, чтобы получить SortedMap со списком полетов по дате, мы могли бы написать:

```
public SortedMap<Date,List<Flight>>
        getSortedMapFlightsByDate(){
   return flights.stream().
        collect(Collectors.groupingBy(
                Flight::getDate,
                TreeMap::new,
                Collectors.toList()));
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

10.3.3 GroupingBy(Function, Supplier, Collector)

Более сложная задача – «инвертировать» Map. Это означает, что если у нас есть Map<K,V>, мы хотим получить Map<V,List<K>> или Map<V,Set<K>> где элементы начального и конечного множеств исходного Map меняются ролями в инвертированном отображении.

Логически, т.к. может быть более одного ключа на одно значение, необходимо присвоить множество элементов финального множества каждому элементу финального множества инвертированного Map.

Например, мы ранее видели Map<String, Long> с количеством полетов по заданному назначению. Если нам нужно узнать назначение, в которое летает наибольшее число рейсов, решением может быть инвертирование отображения SortedMap<Long,List<String>> при этом наибольший key дает список назначений с наибольшим количеством полетов.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

10.3.3 GroupingBy(Function, Supplier, Collector)

В Java 7 этот метод мог быть записан так:

```
public static <T,K> SortedMap<T, List<K>> invertMap(Map<K, T> m)
{
    SortedMap<T, List<K>> res = new TreeMap<T,List<K>>();
    Set<K> sp = m.keySet();
    for(K elem: sp){
            T val =m.get(elem);
            if (res.containsKey(val)) {
                res.get(val).add(elem);
            }
            else {
                List<K> list = new LinkedList<K>();
                lista.add(elem);
                res.put(val, list);
            }
    }
    return res;
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

10.3.3 GroupingBy(Function, Supplier, Collector)

В Java 8 код будет более компактным. Например, преобразование отображения в поток с помощью метода entrySet, который возвращает множество пар ключ-значение. После того, как мы преобразовали отображение в поток пар, мы вызываем collect grouping пары по их значениям (что превращает значения в ключи нового отображения) и заставляя окончательное множество, иметь списки оригинальных ключей, используя необходимое отображение. Конструктор TreeMap позволяет получить на выходе SortedMap:

```
public <K,T> SortedMap<K, List<T>> invertMapToList(Map<T, K> m) {
   return m.entrySet().stream().
       collect(Collectors.groupingBy(
             Map.Entry<T,K>::getValue,
             TreeMap::new, Collectors.mapping(
                   Map.Entry<T,K>::getKey,
                   Collectors.toList())));
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Вопросы

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

JAVA 8: Класс Stream

Евгений Беркунский, НУК

eugeny.berkunsky@gmail.com

http://berkut.mk.ua