

Лекция 14.b

Stream API

- 1.1 Въведение в **Stream** API
- 1.2 Основни понятия
- 1.3 Видове streams
- 1.4 Преобразуване на **Stream**
- 1.5 Видове операции в **IntStream**
- 1.6 Ред на изпълнение на операциите
- 1.7 Повторно използване на **Stream**

Задачи

Литература:
Java 8 Lambdas, Richard Warburton, O'Reilly 2015

1 Въведение

Наименованието **Stream** (стрийм) API е сходно с **InputStream** и **OutputStream** от Java I/O. Същевременно Java 8 **Stream** API има съвсем друг смисъл и приложение. **Streams** са т.нар. „монади“ във Функционалното програмиране. Това са *структури, представящи пресмятания във вид на последователност от отделни стъпки*. **Операциите**, дефинирани на отделните стъпки **се изпълняват последователно и задават една верига от операции върху избрана структура от данни**. За целта съществено се използват Ламбда изрази и функционални интерфейси.

2 Основни понятия

Всеки **Stream** се представя с **последователност от елементи**, посредством които се задават операции за пресмятания върху избран източник (масив или структура от данни)

```
public static void main(String[] args) {  
    List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");  
    myList  
        .stream()  
        .filter(s -> s.startsWith("c"))  
        .map(String::toUpperCase)  
        .sorted()  
        .forEach(System.out::println);  
}  
  
run:  
C1  
C2
```

2 Основни понятия

Различаваме следните два вида **Stream** операции:

- Междинни (Intermediate)
- Завършваща (Terminal)

Междинните операции **връщат обект** от тип **Stream** и това позволява така получените междинен резултат да се подаде за обработка на следващата междинна операция във веригата от операции на stream-а без да се прекъсва изпълнението му. *Завършващите* операции могат да са **void** или да връщат различен от **Stream** резултат.

2 Основни понятия

В дадения пример, операциите `filter`, `map` и `sorted` са *междинни* операции, докато `forEach` е *завършваща* операция. Пълен списък на наличните `stream` операции е наличен в документацията на [Stream API](#).

Дефиниция: *Верига от последователно изпълнявани без прекъсване операции, подобни на `stream` операциите, се нарича каскада (pipeline).*

2 Основни понятия

Повечето **stream** операции приемат Ламбда израз като аргумент в съответствие с типа на определен функционален интерфейс, с което се **конкретизира изпълнението** на дадената операция. Най-често се спазва правилото тези операции да са *„непроменими“* и *„детерминистични“*.

Една операция ще наричаме *„непроменима“*, ако *изпълнението ѝ не променя източника на данни, върху който оперира*. В дадения пример, изпълнението на Ламбда израза не променя **myList** с добавянето или изтриването на елементи от тази колекция данни.

2 Основни понятия

Една операция ще наричаме „**детерминистична**“, ако *изпълнението ѝ не зависи от променливи или състояния извън stream израза, които биха довели до промени в нейното изпълнение.*

2 Основни понятия

Един **stream** може да се създаде по няколко различни начина.

1. С използване на **Stream.of(val1, val2, val3...)**

```
public class StreamBuilders {  
    public static void main(String[] args){  
        Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);  
        stream.forEach(p -> System.out.println(p));  
    }  
}
```

2. С използване на **Stream.of(arrayOfElements)**

```
public class StreamBuilders {  
    public static void main(String[] args){  
        Stream<Integer> stream = Stream.of( new Integer[]{1,2,3,4,5,6,7,8,9}  
        stream.forEach(p -> System.out.println(p));  
    }  
}
```

2 ОСНОВНИ ПОНЯТИЯ

3. С използване на `someList.stream()`

```
public class StreamBuilders {
    public static void main(String[] args){
        List<Integer> list = new ArrayList<Integer>();
        for(int i = 1; i< 10; i++){
            list.add(i);
        }
        Stream<Integer> stream = list.stream();
        stream.forEach(p -> System.out.println(p));
    }
}
```

4. С използване на `Arrays.stream()`

```
Integer[] values = {2, 9, 5, 0, 3, 7, 1, 4, 8, 6};
Arrays.stream(values)
    .filter(value -> value > 4)
    .forEach(System.out::println);
```

5. С използване на **междинни операции**

```
Stream<String> stream = Stream.of("d2", "a2", "b1", "b3", "c3")
    .filter(str->str.startsWith("b"));
```

3 Видове streams

Stream-ове могат да се създадат от различни източници на данни и по-специално, колекции от данни. **List** и **Set** поддържат новите методи **stream()** и **parallelStream()** за създаване на **stream** за **последователно** или **паралелно** (многонишково) изпълнение.

Паралелните **stream**-ове могат да се изпълняват на различни нишки и така да се възползват от предимствата на компютърни системи и устройства с многоядрени или множество от процесори. В тази лекция ще разгледаме само последователно изпълнение на **stream**.

3 Видове streams

```
Arrays.asList("a1", "a2", "a3")  
    .stream()  
    .findFirst()  
    .ifPresent(System.out::println);  
  
run:  
a1
```

Извикването на метода `stream()` от списък с обекти връща обикновен обект от тип `Stream`. Не е нужно, обаче, да създаваме колекции от данни, за да работим със `stream`, както например:

```
Stream.of("a1", "a2", "a3")  
    .findFirst()  
    .ifPresent(System.out::println);  
  
run:  
a1
```


3 Видове streams

За да се създаде **stream** от **набор референции** към обекти, е нужно само да се използва метода **Stream.of()**.

Освен обикновени обекти от тип **stream**, Java 8 предоставя **специализирани stream обекти** за **работа с примитивни типове данни** **int**, **long** и **double**.

Наименованията на тези **stream обекти** са съответно **IntStream**, **LongStream** и **DoubleStream**.

В общия случай се ползва интерфейса **Stream<T>**,

Примерно,

Stream<Employee>, **Stream<String>**

3.1 Примери на streams

Всички тези специализирани **stream** типове се изпълняват, както и обикновените **stream** обекти със следните разлики. **Специализираните stream обекти използват съответни специализирани типове** Ламбда изрази т.е. **IntFunction** се използва вместо **Function** или **IntPredicate** вместо **Predicate**. Допълнително, специализираните **stream**-ове **поддържат завършващи операции за обобщаване на резултати** като операцията **average()**.

Тези операции връщат **Optional<T>** тип. Например, при намиране на средно аритметично на цели числа се създава **OptionalDouble** обект.

3.1 Примери на streams

В следния пример отново използваме операцията `ifPresent(System.out::println)`, за да избегнем грешка при евентуално делене на нула при пресмятане на `average()`

```
Arrays.stream(new int[]{1, 2, 3})  
    .map(n -> 2 * n + 1)  
    .average()  
    .ifPresent(System.out::println);
```

run:

5.0

Така, при празен стрийм не се хвърля изключение. Ако вместо `ifPresent()` ползвахме `getAsDouble()`, при празен стрийм се хвърля изключение.

3.1 Примери на streams

Трансформацията `map()` е междинна операция, която съпоставя нова стойностна всеки елемент от текущия `stream` и създава `stream`, съдържащ елементите с новите им стойности (вероятно също от нов тип). Методът `map()` взима единствен аргумент от **функционален интерфейс** `UnaryOperator`.

Понякога се налага **трансформация на обикновен обект stream в специализиран обект stream** или обратно. За целта се използват операциите `mapToInt()`, `mapToLong()` и `mapToDouble()`

3.1 Примери на streams

```
Stream.of("a1", "a2", "a3")  
    .map(s -> s.substring(1))  
    .mapToInt(Integer::parseInt)  
    .max()  
    .ifPresent(System.out::println);
```

run:
3

Първата операция `map()`, преобразува набора от стрингове "a1", "a2", "a3" до "1", "2", "3".

Втората операция `mapToInt()` преобразува елементите на така получения `stream` от предходната междинна операция до `IntStream` от 1, 2, 3. Накрая, завършващата операция `max()`, налична в `IntStream` да намери най- голямото измежду тези числа и то се отпечатва, ако е налично.

3.1 Примери на streams

Специализираните stream обекти могат да се **преобразуват до обикновени stream обекти**, посредством **mapToObj()**:

```
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);
```

run:

a1

a2

a3

3.1 Примери на streams

Да разгледаме и друг комбиниран пример на `mapToObj()`:

```
Stream.of(1.0, 2.0, 3.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);
```

run:

a1

a2

a3

4 Видове операции в `IntStream`

Да разгледаме на `IntStream`.

Статичният метод `of()` на `IntStream` взима за аргумент масив от цели числа и връща `IntStream` за обработка на елементите на този масив.

Завършващата операция `forEach` взима за аргумент **обект** който имплементира функционалния интерфейс `IntConsumer` (package `java.util.function`). Методът `accept` на този интерфейс взима един `int` аргумент и изпълнява **зададеното действие с него**.

4 Видове операции в IntStream

```
public class IntStreamOperations
{
    public static void main(String[] args)
    {
        int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};

        // display original values
        System.out.print("Original values: ");
        IntStream.of(values)
            .forEach(value -> System.out.printf("%d ", value));
        System.out.println();
    }
}
```

4 Видове операции в `IntStream`

Едно типично приложение на `IntStream` позволява да замести често използван `for`-цикъл за инициализация с използване на `IntStream.range()`:

```
IntStream.range(1, 4)
    .forEach(System.out::println);
```

```
run:
```

```
1
2
3
```

4 Видове операции в IntStream

Методите `range` и `rangeClosed` на `IntStream` „произвеждат“ **сортирана последователност** от `int` стойности.

- Двата метода **приемат** `int` аргументи задаващи интервал от стойности.
- Методът `range` „произвежда“ последователност от стойности в отворен от дясно интервал, зададен от двата аргумента на метода.
- Методът `rangeClosed` „произвежда“ последователност от стойности в затворен интервал, зададен от двата аргумента на метода.

4 Видове операции в IntStream

```
// sum range of integers from 1 to 10, exclusive  
System.out.printf("%nSum of integers from 1 to 9: %d%n",  
    IntStream.range(1, 10).sum());
```

```
// sum range of integers from 1 to 10, inclusive  
System.out.printf("Sum of integers from 1 to 10: %d%n",  
    IntStream.rangeClosed(1, 10).sum());
```

```
run:
```

```
45
```

```
55
```

Тези примери показват как да се сумира последователност от стойности в даден интервал от цели числа.

4 Видове операции в `IntStream`

Клас `IntStream` предоставя завършващи операции, които редуцират `IntStream` of цяло число в няколко частни и най- често използвани случая

- `count` връща броя на елементите в `IntStream`
- `min` връща най- малкото цяло число в `IntStream`
- `max` връща най- голямото цяло число в `IntStream`
- `sum` връща сумата на всички елементи в `IntStream`
- `average` връща средната стойност на всички елементи в `IntStream` като `OptionalDouble` (package `java.util`)

Методът `getAsDouble` на `OptionalDouble` връща `double` или хвърля `NoSuchElementException`.

- За **избягване на `NoSuchElementException`**, добра практика е да се изпълни `orElse`, което връща стойността на `OptionalDouble` когато тя е налична, или стойността, подадена като аргумент на `orElse`.
- **Друга възможност е да се изпълни `ifPresent()`**

4 Видове операции в IntStream

```
// count, min, max, sum and average of the values
System.out.printf("%nCount: %d%n", IntStream.of(values).count());
System.out.printf("Min: %d%n",
    IntStream.of(values).min().getAsInt());
System.out.printf("Max: %d%n",
    IntStream.of(values).max().getAsInt());
System.out.printf("Sum: %d%n", IntStream.of(values).sum());
System.out.printf("Average: %.2f%n",
    IntStream.of(values).average().getAsDouble());
```

4 Видове операции в IntStream

```
int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};
// Exception is thrown in case of empty stream
System.out.printf("Average: %.2f%n",
                  IntStream.of(values).average().getAsDouble());
// no exception
System.out.printf("Average: %.2f%n",
                  IntStream.of(values).average().orElseGet(() -> -1));
// no exception
IntStream.of(values)
          .average()
          .ifPresent(p->System.out.format("Average: %.2f%n", p));
// no exception
System.out.printf("Average: %.2f%n",
                  IntStream.of(values).average().orElse(-1));
run:
Average: 5.50
Average: 5.50
Average: 5.50
Average: 5.50
```

4 Видове операции в IntStream

Методът `summaryStatistics` на `IntStream` изпълнява наведнъж операциите `count`, `min`, `max`, `sum` и `average` operations и връща `IntSummaryStatistics` обект (package `java.util`).

```
IntSummaryStatistics stats = IntStream.of(values)
                                   .summaryStatistics();

System.out.println(stats.getAverage()); // average
System.out.println(stats.getCount());   // count
System.out.println(stats.getMax());     // max
System.out.println(stats.getMin());     // min
System.out.println(stats.getSum());     // sum
```

4 Видове операции в `IntStream`

Завършващата операция `reduce()` позволява да се изпълнят потребителски дефинирани редукции на `IntStream` елементи до един резултат от конкретно желан тип. Тази операция взима два аргумента:

- **Първият аргумент (`identity`)** е стойността, с която започва редукцията (**начална стойност**)
 - **Вторият аргумент (`accumulator/combiner`)** е обект от функционален интерфейс `IntBinaryOperator`
- (двата входни параметъра на `IntBinaryOperator` са от един и същи тип, `Integer`)

4 Видове операции в IntStream

Операцията **reduce** прилага началната стойност към комбинираната последователност от изпълнението на **IntBinaryOperator** към всеки отделен елемент на **IntStream**.

```
// sum of squares of values with reduce method
int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};
System.out.printf("Sum of squares via reduce method: %d\n",
    IntStream.of(values)
        .reduce(0, (x, y) -> x + y * y));
```

run:

```
Sum of squares via reduce method: 385
```

4 Видове операции в IntStream

Други примери на операцията **reduce**

```
// sum of values with reduce method
System.out.printf("%nSum via reduce method: %d%n",
    IntStream.of(values)
               .reduce(0, (x, y) -> x + y));
// sum of squares of values with reduce method
System.out.printf("Sum of squares via reduce method: %d%n",
    IntStream.of(values)
               .reduce(0, (x, y) -> x + y * y));
// product of values with reduce method
System.out.printf("Product via reduce method: %d%n",
    IntStream.of(values)
               .reduce(1, (x, y) -> x * y));
```


4 Видове операции в IntStream

Операцията **reduce()** на **IntStream** може да се използва и с един аргумент от тип **IntBinaryOperator**

```
int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};
System.out.printf("Sum of squares via reduce method: %d%n",
    IntStream.of(values)
        .reduce((x, y) -> Integer.sum(x, y * y)).getAsInt());
// is equivalent to
IntStream i = IntStream.of(3, 10, 6, 1, 4, 8, 2, 5, 9, 7);
IntBinaryOperator o = (x, y) -> x + y * y;
i.reduce(o)
    .ifPresent(x->System.out.printf("Sum of squares via reduce method: %d%n", x));
```

където **identity** аргументът е първият елемент на колекцията данни, а **combiner** е **Integer.sum(int a, int b)**

4 Видове операции в IntStream

Междинната операция **filter()** на **IntStream** създава **stream** междинни резултати, които удовлетворяват аргумента на операцията **filter()** от тип **Predicate**. (package **java.util.function**).

Междинната операция **IntStream method sorted()** на **IntStream** сортира елементите на **IntStream** във възходящ ред.

- Всички предходни междинни операции на **stream**-а трябва да са приключили, за да е ясно какво се сортира със **sorted**.
- За сортиране на **IntStream** в обратен ред **sorted()** взима параметър **Comparator.reverseOrder()** или Ламбда израз от тип **Comparator**

4 Видове операции в IntStream

```
// even values displayed in sorted order
System.out.printf("%nEven values displayed in sorted order: ");
IntStream.of(values)
    .filter(value -> value % 2 == 0)
    .sorted()
    .forEach(value -> System.out.printf("%d ", value));
System.out.println();
```

```
// odd values multiplied by 10 and displayed in sorted order
System.out.printf(
    "Odd values multiplied by 10 displayed in sorted order: ");
IntStream.of(values)
    .filter(value -> value % 2 != 0)
    .map(value -> value * 10)
    .sorted()
    .forEach(value -> System.out.printf("%d ", value));
System.out.println();
```

run:

Even values displayed in sorted order: 2 4 6 8 10

Odd values multiplied by 10 displayed in sorted order: 10 30 50 70 90

4.1 Комбиниране на предикати

Много често при филтрирането се налага съставяне на логически изрази от предикати. За целта се използват подразбиращите се методи на интерфейс **Predicate**

✓ `negate()`

✓ `and()`

✓ `or()`

4.1 Комбиниране на предикати

//using AND method

```
IntPredicate intPredGreater11 = (x) -> x > 11;  
IntPredicate intPredLesser    = (x) -> x < 20;  
IntPredicate andExpr= intPredGreater11.and(intPredLesser);  
System.out.println(andExpr.test(17)); // Outputs true  
System.out.println(andExpr.test(25)); // Outputs false
```

//using NEGATE method

```
IntPredicate intPredGreater10 = (x) -> x > 10;  
System.out.println(intPredGreater10.test(11)); //Outputs true  
System.out.println(intPredGreater10.test(9));  //Outputs false
```

```
IntPredicate negationExpr = intPredGreater10.negate();  
System.out.println(negationExpr.test(11)); //Outputs false  
System.out.println(negationExpr.test(9));  //Outputs true
```

4.1 Комбиниране на предикати

```
//using OR method
IntPredicate intPred1 = (x) -> x > 10 && x < 20;
IntPredicate intPred2 = (x) -> x > 40 && x < 50;

IntPredicate orExpr = intPred1.or(intPred2);
//Outputs true. Number is between 10 and 20
System.out.println(orExpr.test(15));
//Outputs true. Number is between 40 and 50
System.out.println(orExpr.test(47));
//Outputs false. Number < 10 and Number < 40
System.out.println(orExpr.test(7));
//Outputs false. Number > 20 and Number < 40
System.out.println(orExpr.test(32));
```

4.1 Комбиниране на предикати

```

class Person {
    private String name;
    private int age;

    // constructor, getter and setter methods
}

// Predicate test
List<Person> list = new ArrayList<>();
list.add(new Person("Aristotel", 11));
list.add(new Person("Anton", 22));
list.add(new Person("Walter", 32));
list.add(new Person("Magda", 34));
list.add(new Person("Radost", 44));
list.add(new Person("Stefan", 21));

Predicate<Person> predicate1 = (p) -> p.getName().startsWith("A");
Predicate<Person> predicate2 = (p) -> p.getName().startsWith("W");
Predicate<Person> predicate1or2 = predicate1.or(predicate2);

list.stream().filter(predicate1or2).forEach(System.out::println);

run:
[name=Aristotel, age=11]
[name=Anton, age=22]
[name=Walter, age=32]
BUILD SUCCESSFUL (total time: 0 seconds)

```

5 Преобразуване на Stream

Stream се **преобразуват в колекции** от данни и масиви посредством операцията **collect()** и клас **Collectors**

```
List<Integer> list = new ArrayList<Integer>();  
for(int i = 1; i < 10; i++){  
    list.add(i);  
}  
Stream<Integer> stream = list.stream();  
List<Integer> evenNumbersList = stream.filter(i -> i%2 == 0).collect(Collectors.toList());  
System.out.println(evenNumbersList);
```

run:

[2, 4, 6, 8]

```
List<Integer> list = new ArrayList<Integer>();  
for(int i = 1; i < 10; i++){  
    list.add(i);  
}  
Stream<Integer> stream = list.stream();  
Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::new);  
System.out.println(Arrays.toString(evenNumbersArr));
```

run:

[2, 4, 6, 8]

5 Преобразуване на Stream

Stream, преобразуван в колекция от данни или масив от данни може отново да се използва за създаване на **Stream** за обработване на данните съхранени в **колекцията** от данни или масива от данни.

5.1 Stream основни операции

Stream интерфейсът декларира множество операции. Ще разгледаме някои основни в следните примери.

```
List<String> memberNames = new ArrayList<>();  
memberNames.add("Антон");  
memberNames.add("Симона");  
memberNames.add("Ангел");  
memberNames.add("Радос");  
memberNames.add("Стефи");  
memberNames.add("Даниела");  
memberNames.add("Здравко");  
memberNames.add("Ива");
```

5.1.1 Stream междинни операции

Междинните операции връщат **Stream** и се изпълняват верижно.

filter()

filter взима за аргумент **Predicate** и филтрира всички елементи на един stream. Това позволява след тази операция да се изпълни друга върху филтрираните данни (например **forEach**) и да се получи нов

```
memberNames.stream().filter((s) -> s.startsWith("A"))  
                .forEach(System.out::println);
```

run:

Антон

Ангел

5.1.1 Stream междинни операции

Междинните операции връщат **Stream** и се изпълняват верижно.

filter()

filter взима за аргумент **Predicate** и филтрира всички елементи на един stream. Това позволява след тази операция да се изпълни друга върху филтрираните данни (например **forEach**) и да се получи нов

```
memberNames.stream().filter((s) -> s.startsWith("A"))  
                .forEach(System.out::println);
```

run:

Антон

Ангел

5.1.1 Stream междинни операции

map()

map е междинна операция, която преобразува всеки елемент в друг обект посредством зададена функция. Следният пример преобразува всеки **String** в String с главни букви.

```
memberNames.stream().filter((s) -> s.startsWith("C"))  
    .map(String::toUpperCase)  
    .forEach(System.out::println);
```

run:

СИМОНА

СТЕФИ

5.1.1 Stream междинни операции

sorted()

sorted е междинна операция, връща сортиран изглед на **stream**-а. Елементите на **stream**-а се сортират в естествен ред, ако не се подаде за аргумент обект от тип потребителски дефиниран **Comparator**.

```
memberNames.stream().sorted()  
    .map(String::toUpperCase)  
    .forEach(System.out::println);
```

```
run:  
АНГЕЛ  
АНТОН  
ДАНИЕЛА  
ЗДРАВКО  
ИВА  
РАДО  
СИМОНА  
СТЕФИ
```

```
memberNames.stream().sorted(Collections.reverseOrder())  
    .map(String::toUpperCase)  
    .forEach(System.out::println);
```

```
run:  
СТЕФИ  
СИМОНА  
РАДО  
ИВА  
ЗДРАВКО  
ДАНИЕЛА  
АНТОН  
АНГЕЛ
```

5.1.1 Stream междинни операции

sorted()

За сортиране по повече от едно свойство се извършва групиране на `Comparator`-и с

`Comparator.thenComparing(Comparator)`

`Comparator.reversed()`

```
List<Employee> employees;
public void multiple_sort()
{
    Comparator<Employee> byFirstName = (e1, e2) -> e1.getEmployeeFirstName()
                                           .compareTo(e2.getEmployeeFirstName());
    Comparator<Employee> byLastName = (e1, e2) -> e1.getEmployeeLastName()
                                                  .compareTo(e2.getEmployeeLastName());

    employees.stream().sorted(byFirstName.thenComparing(byLastName))
              .forEach(e -> System.out.println(e));
}
employees.stream()
          .sorted(byFirstName.reversed()
                  .thenComparing(byLastName).reversed())
          .forEach(e -> System.out.println(e));
```

5.1.2 Stream завършващи операции

`forEach()`

forEach е завършваща операция, която позволява да се обходят елементите на **stream** и да се изпълни една операция със всеки един от тях. Операцията се задава като Ламбда израз (**Consumer**).

```
memberNames.forEach(System.out::println);
```

```
run:
```

```
Антон
```

```
Симона
```

```
Ангел
```

```
Радю
```

```
Стефи
```

```
Даниела
```

```
Здравко
```

```
Ива
```


5.1.2 Stream завършващи операции

`collect()`

`collect` е завършваща операция, която преобразува **`stream`** в колекция от данни в съответствие с метода на `Collectors` класа, подаден като аргумент на **`collect()`**

```
List<String> memNamesInUppercase = memberNames.stream().sorted()  
    .map(String::toUpperCase)  
    .collect(Collectors.toList());  
System.out.print(memNamesInUppercase);
```

run:

```
[АНГЕЛ, АНТОН, ДАНИЕЛА, ЗДРАВКО, ИВА, РАДО, СИМОНА, СТЕФИ]
```

5.1.2 Stream завършващи операции

count()

count е завършваща операция, връща броя на елементите в **stream** като резултат от тип **long**.

```
long totalMatched = memberNames.stream()  
    .filter((s) -> s.startsWith("Д"))  
    .count();
```

```
System.out.println(totalMatched);
```

```
run:
```

```
1
```

5.1.2 Stream завършващи операции

reduce()

reduce е завършваща операция, извършваща редукция в съответствие с функцията, зададена като Ламбда израз от тип **BinaryOperator**. Резултатът е от тип **Optional** и съдържа редуцираната стойност.

```
Optional<String> reduced = memberNames.stream()  
    .reduce((s1, s2) -> s1 + "#" + s2);  
reduced.ifPresent(System.out::println);
```

run:

Антон#Симона#Ангел#Радос#Стефи#Даниела#Здравко#Ива

5.1.2 Stream завършващи операции

match()

match е завършваща операция, позволяваща да се определи дали елементите на **stream**-а удовлетворяват даден **Predicate**. Това може да се **различни операции**, но всички те са завършващи и връщат **boolean**

```
boolean matchedResult = memberNames.stream()
    .anyMatch((s) -> s.startsWith("Д"));
System.out.println(matchedResult);
matchedResult = memberNames.stream()
    .allMatch((s) -> s.startsWith("A"));
System.out.println(matchedResult);
matchedResult = memberNames.stream()
    .noneMatch((s) -> s.startsWith("P"));
System.out.println(matchedResult);
```

run:

true

false

true

5.1.3 Операции за прекъсване

В някои случаи е нужно да се прекъсне изпълнението на операцията при установяване на съвпадение с елемент от stream-а при изпълнение на текущата итерация на обработка.

Аналогична е ситуацията при изпълнение на цикъл и изпълнение на **break** команда. При **Stream** се използват следните операции за тази цел:

anyMatch()

Тази завършваща операция **връща true веднага щом, подаденият ѝ като аргумент Predicate бъде удовлетворен**. При това спира обработката на операцията.

```
boolean matched = memberNames.stream()  
    .anyMatch((s) -> s.startsWith("A"));  
System.out.println(matched);
```

run:
true



5.1.3 Операции за прекъсване

`findFirst()`

Тази завършваща операция връща първият елемент от **stream** и с това се прекратява неговото изпълнение .

```
memberNames.stream()  
    .filter((s) -> s.startsWith("И"))  
    .findFirst()  
    .ifPresent(x->System.out.println(x));
```

run:

Ива

6 Ред на изпълнение на операциите

Да разгледаме **в какъв ред се изпълняват** **stream** операциите. Важна характеристика на всички междинни операции липсата на „*активност*“ (**laziness**). Нека да видим какво се получава при отсъствието на завършваща операция в **stream-a**.

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    });
```

Оказва се, че при липса на завършваща операция **stream-a** не се изпълнява.

6 Ред на изпълнение на операциите

При добавяне на завършваща операция **stream**-а се ИЗПЪЛНЯВА.

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

run:

```
filter: d2
forEach: d2
filter: a2
forEach: a2
filter: b1
forEach: b1
filter: b3
forEach: b3
filter: c
forEach: c
```


6 Ред на изпълнение на операциите

Редът на изпълнение може би е изненадващ, защото наивно може да очакваме операциите да се изпълняват изцяло една след друга в хоризонтален ред.

Вместо това **всеки елемент от stream-а се обработва последователно** от операциите в **stream-а във вертикален порядък**. Първият стринг "d2", удовлетворява **filter** и след това се изпълнява **forEach**, чак след това вторият стринг се тества на **filter** и после се изпълнява **forEach** и т.н. с останалите елементи от данните в **stream-а**.

6 Ред на изпълнение на операциите

Това позволява да се съкрати изпълнението на ненужни операции, както се вижда от следващия пример.

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
        return s.startsWith("A");
    });
```

```
run:
map: d2
anyMatch: D2
map: a2
anyMatch: A2
```

Изпълнението на операциите в `stream`-а продължава, докато `anyMatch()` не върне `true` т.е. при първия елемент от започващ с “A”. Операцията `map()` се изпълнява само **два** пъти.

7 Повторно използване на Stream

Веднъж **създаден**, един **Stream** не може да се използва повторно. **Stream**-ът **се затваря** веднага след изпълнение на завършващата операция.

```
Stream<String> stream
    = Stream.of("d2", "a2", "b1", "b3", "c")
        .filter(s -> s.startsWith("a"));
stream.anyMatch(s -> true);    // ok
stream.noneMatch(s -> true);  // exception
```

```
run:
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
    at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
    at java.util.stream.ReferencePipeline.noneMatch(ReferencePipeline.java:459)
    at test.Test.main(Test.java:181)
```

Проблемът се състои в това, че при повторното изпълнение се опитваме да изпълним завършваща операция към затворен **Stream**

7 Повторно използване на Stream

Това ограничение се преодолява като се **създава нова верига** от **Stream** операции **за всяка отделна завършваща операция**. За целта се използва конструкцията за доставка на **Stream** с готовите междинни операции, съхранени в него.

```
Supplier<Stream<String>> streamSupplier  
    = () -> Stream.of("d2", "a2", "b1", "b3", "c")  
        .filter(s -> s.startsWith("a"));
```

```
streamSupplier.get().anyMatch(s -> true);    // ok  
streamSupplier.get().noneMatch(s -> true);   // ok
```

При всяко изпълнение на **get()** се създава нов **Stream**, към който може да приложим **завършваща операция**

8 Обработка на `Stream<Employee>`

Следващите примери демонстрират възможностите на `lambda` и `stream` при работа със `Stream<Employee>`.

`Class Employee` използван в тези примери представя Служител с `first name`, `last name`, `salary` и `department` заедно със `setter`, `getter`, конструктори и `toString()` методи .

8 Обработка на Stream<Employee>

```
public class Employee {  
    private String firstName;  
    private String lastName;  
    private double salary;  
    private String department;  
  
    // [setter, getter, constructors and toString() ...]  
}
```

8. 1 Създаване и извеждане на `List<Employee>`

При подаване на **референцията към инстанционния метод** `System.out::println` като параметър на **Stream метода** `forEach`, този метод се преобразува от компилатора в обект на клас, имплементиращ функционалния интерфейс **`Consumer`**.

- Методът на **асерт** на интерфейс **`Consumer`** **взима един параметър и връща `void`**. В разглеждания случай, методът `асерт` предава този параметър на инстанционния метод `println` на обекта `System.out`.

Следващият слайд показва създаване на масив от `Employees`, съхранява масива в `List` и **извежда на стандартен изход този `List`**.

8. 1 Създаване и извеждане на List<Employee>

```

1 // initialize array of Employees
2 Employee[] employees = {
3     new Employee("Jason", "Red", 5000, "IT"),
4     new Employee("Ashley", "Green", 7600, "IT"),
5     new Employee("Matthew", "Indigo", 3587.5, "Sales"),
6     new Employee("James", "Indigo", 4700.77, "Marketing"),
7     new Employee("Luke", "Indigo", 6200, "IT"),
8     new Employee("Jason", "Blue", 3200, "Sales"),
9     new Employee("Wendy", "Brown", 4236.4, "Marketing")};
10
11 // get List view of the Employees
12 List<Employee> list = Arrays.asList(employees);
13
14 // display all Employees
15 System.out.println("Complete Employee list:");
16 list.stream().forEach(System.out::println);

```

```

Complete Employee list:
Jason      Red      5000.00    IT
Ashley     Green    7600.00    IT
Matthew    Indigo   3587.50    Sales
James      Indigo   4700.77    Marketing
Luke       Indigo   6200.00    IT
Jason      Blue     3200.00    Sales
Wendy      Brown    4236.40    Marketing

```



8. 2 Филтриране на служители по заплати

На следващия слайд е показано филтриране на `Employees` с обект, имплементиращ `Predicate<Employee>`, дефиниран с Ламбда израз

За повторно използване на Ламбда израз, изразът се присвоява на Функционален интерфейс от съответстващ тип.

Интерфейсът `Comparator` има `static` метод `comparing`. Този метод приема за параметър метод за извличане на данна от потока, на основата на която да се прави сравнение. Методът връща обект от тип `Comparator`.



8. 2 Филтриране на служители по заплати

```

1 // Predicate that returns true for salaries in the range $4000-$6000
2 Predicate<Employee> fourToSixThousand =
3     e -> (e.getSalary() >= 4000 && e.getSalary() <= 6000);
4 // Display Employees with salaries in the range $4000-$6000
5 // sorted into ascending order by salary
6 System.out.printf(
7     "%nEmployees earning $4000-$6000 per month sorted by salary:%n");
8 list.stream()
9     .filter(fourToSixThousand)
10    .sorted(Comparator.comparing(Employee::getSalary))
11    .forEach(System.out::println);
12 // Display first Employee with salary in the range $4000-$6000
13 System.out.printf("%nFirst employee who earns $4000-$6000:%n%s%n",
14    list.stream()
15        .filter(fourToSixThousand)
16        .findFirst()
17        .get()); // returns Employee ifPresent, else- NoSuchElementException

```

Employees earning \$4000-\$6000 per month sorted by salary:

Wendy	Brown	4236.40	Marketing
James	Indigo	4700.77	Marketing
Jason	Red	5000.00	IT

First employee who earns \$4000-\$6000:

Jason	Red	5000.00	IT
-------	-----	---------	----

Повече информация за метод `Comparator.comparing()` ще намерите в примерния сорс код към лекцията в клас `ComparatorTest`

8. 2 Филтриране на служители по заплати

```
static <T, U extends Comparable<? super U>>
Comparator<T> comparing( Function<? super T, ? extends U>
                        keyExtractor)
```

```
public class ComparatorTest {
    static Employee[] employees = new Employee[] {
        new Employee( "John", 25, 3000.0, "9922001"),
        new Employee( "Ace", 22, 2000.0, "5924001"),
        new Employee( "Keith", 35, 4000.0, "3924401")
    };
    public static void main(String[] args) {
        System.out.println( Arrays.toString(employees));
        whenComparingThenSortedByName();
        System.out.println( Arrays.toString(employees));
    }
    public static void whenComparingThenSortedByName() {
        Comparator<Employee> employeeNameComparator
            = Comparator.comparing(Employee::getName);
        Arrays.sort(employees, employeeNameComparator);
    }
}
```

x -> x.getName()

Function consumes T x,
produces U propertyOf type T

8. 2 Филтриране на служители по заплати

```
static <T, U extends Comparable<? super U>>
    Comparator<T> comparing( Function<? super T, ? extends U>
                               keyExtractor)
```

```
public class ComparatorTest {
    static Employee[] employees = new Employee[] {
        new Employee( "John", 25, 3000.0, "9922001"),
        new Employee( "Ace", 22, 2000.0, "5924001"),
        new Employee( "Keith", 35, 4000.0, "3924401")
    };

    public static void main(String[] args) {
        System.out.println( Arrays.toString(employees));
        whenComparingThenSortedByAge (); // comparingLong(), comparingDouble()
        System.out.println( Arrays.toString(employees));
    }

    public void whenComparingIntThenSortedByAge() {
        Comparator<Employee> employeeAgeComparator
            = Comparator.comparingInt(Employee::getAge);
        Arrays.sort(employees, employeeAgeComparator);
    }
}
```



8. 2 Филтриране на служители по заплати

```
// static <T,U> Comparator<T> comparing( Function<? super T,? extends U> keyExtractor,
//                                     Comparator<? super U> keyComparator)

public class ComparatorTest {
    static Employee[] employees = new Employee[] {
        new Employee( "John", 25, 3000.0, "9922001"),
        new Employee( "Ace", 22, 2000.0, "5924001"),
        new Employee( "Keith", 35, 4000.0, "3924401")
    };

    public static void main(String[] args) {
        System.out.println( Arrays.toString(employees));
        whenComparingThenSortedByNameDesc ();
        System.out.println( Arrays.toString(employees));
    }

    public void whenComparingThenSortedByNameDesc() {
        Comparator<Employee> employeeNameComparator
            = Comparator.comparing( Employee::getName,
                                   (s1, s2) -> s2.compareTo(s1)); // is like .reversed()
        Arrays.sort(employees, employeeNameComparator);
    }
}
```



8. 2 Филтриране на служители по заплати

```
//static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
```

```
public class Employee implements Comparable<Employee>{  
    // ...
```

```
    @Override
```

```
    public int compareTo(Employee argEmployee) {  
        return name.compareTo(argEmployee.getName());  
    }  
}
```

```
public void whenNaturalOrder_thenSortedByName() {  
    Comparator<Employee> employeeNameComparator  
        = Comparator.<Employee> naturalOrder();  
  
    Arrays.sort(employees, employeeNameComparator);  
}
```

8. 2 Филтриране на служители по заплати

Важно предимство на Stream обработката е възможността за **прекъсване на обработката на каскадата от операции веднага щом се получи желания резултат.**

Методът `findFirst` на интерфейса `Stream` прекъсва обработката веднага щом открие първия обект в потока от данни, който удовлетворява зададеното условие за филтриране.

`findFirst` **метод връща** `Optional` обект.

Методът `get()` **връща този обект или хвърля изключение, когато този обект е** `null`

8. 2 Филтриране на служители по заплати

Важно предимство на Stream обработката е възможността за **прекъсване на обработката на каскадата от операции веднага щом се получи желания резултат.**

Методът `findFirst` на интерфейса `Stream` прекъсва обработката веднага щом открие първия обект в потока от данни, който удовлетворява зададеното условие за филтриране.

`findFirst` **метод връща** `Optional` обект.

Методът `get()` **връща този обект или хвърля изключение, когато този обект е** `null`

8. 2a Сумиране на заплати на служители

`Stream.reduce` се използва в общия случай при операции за редуциране на структура от данни до конкретна стойност .

Важно е да се гарантира, че двата параметъра на accumulator-а са от един и същ тип, в случая- double. Identity трябва да е от същия тип. Това става с прилагане на `map()` преди `reduce()`.

```
double sumSalary = employeeList
    .stream()
    .map(Employee::getSalary)
    .reduce(
        0d,
        ( partialAgeResult,  employeeSalary) -> partialAgeResult + employeeSalary);
```

8. 2a Сумиране на заплати на служители

В случая същия резултат може да се постигне с преобразуване на `Stream<Employee>` към `DoubleStream` и прилагане на операцията `sum()`

```
sumSalary = employeeList
```

```
    .stream()
```

```
    .mapToDouble(Employee::getSalary)
```

```
    .sum();
```

Операцията `reduce()` всеки път връща нова стойност, същевременно `accumulator`-а също така връща нова стойност при обработката на поредния елемент от `Stream`-а. Поради това не се препоръчва `reduce()` да се използва за създаване на колекция, тъй като `accumulator`-а ще създава нова колекция всеки път, когато добавя елемент към колекцията. За целта е по-добре да се използва редактиране на елементи на съществуваща колекция, създадена с метода `collect()`.



8. 3 Сортиране на служители по повече от едно поле данни

За сортиране на обекти по повече от едно поле данни **е нужен Comparator** който използва **два метода за извличане на тези данни**.

Първо се извиква метода **comparing** на **Comparator** за създавана не **Comparator** с метод за извличане на първото поле данни.

Върху така получения **Comparator**, прилагаме метода **thenComparing** с параметър метод за извличане на второто поле данни.

Така създадения **Comparator** сравнява обектите като използва първото поле данни, а когато има съвпадение на обекти по това поле, сравнението се извършва по отношение на второто поле данни

8. 3 Сортиране на служители по повече от едно поле

```

1 // Functions for getting first and last names from an Employee
2 Function<Employee, String> byFirstName = Employee::getFirstName;
3 Function<Employee, String> byLastName = Employee::getLastName;
4 // Comparator for comparing Employees by first name then last name
5 Comparator<Employee> lastThenFirst =
6     Comparator.comparing(byLastName).thenComparing(byFirstName);
7 // sort employees by last name, then first name
8 System.out.printf(
9     "%nEmployees in ascending order by last name then first:%n");
10 list.stream()
11     .sorted(lastThenFirst)
12     .forEach(System.out::println);
13 // sort employees in descending order by last name, then first name
14 System.out.printf(
15     "%nEmployees in descending order by last name then first:%n");
16 list.stream()
17     .sorted(lastThenFirst.reversed())
18     .forEach(System.out::println);

```

Employees in ascending order by last name then first:

Jason	Blue	3200.00	Sales
Wendy	Brown	4236.40	Marketing
Ashley	Green	7600.00	IT
James	Indigo	4700.77	Marketing
Luke	Indigo	6200.00	IT
Matthew	Indigo	3587.50	Sales
Jason	Red	5000.00	IT

Employees in descending order by last name then first:

Jason	Red	5000.00	IT
Matthew	Indigo	3587.50	Sales
Luke	Indigo	6200.00	IT
James	Indigo	4700.77	Marketing
Ashley	Green	7600.00	IT
Wendy	Brown	4236.40	Marketing
Jason	Blue	3200.00	Sales

8.4 Съпоставка на обекти от различен тип

Този пример демонстрира как да съпоставим обекти от един тип (`Employee`) на обект от друг тип (`String`). В случая ще съпоставим на всеки служител неговото име.

В общия случай е възможно да се изобразят обекти в даден `stream` на обекти от друг тип и така да се създаде друг `stream` със същия брой обекти, както в зададения `stream` .

Методът `distinct()` на `Stream` елиминира дублиращите се обекти в `stream`-а .

8.4 Съпоставка на обекти от различен тип

```
1 // display unique employee last names sorted
2 System.out.printf("%nUnique employee last names:%n");
3 list.stream()
4     .map(Employee::getLastName)
5     .distinct()
6     .sorted()
7     .forEach(System.out::println);
8
9 // display only first and last names
10 System.out.printf(
11     "%nEmployee names in order by last name then first name:%n");
12 list.stream()
13     .sorted(lastThenFirst)
14     .map(Employee::getName)
15     .forEach(System.out::println);
```

Unique employee last names:

Blue
Brown
Green
Indigo
Red

Employee names in order by last name then first name:

Jason Blue
Wendy Brown
Ashley Green
James Indigo
Luke Indigo
Matthew Indigo
Jason Red



8.5 Групиране на Employees

Групиране в `Stream` се извършва с метода `collect()`. Тук ще **покажем как се групират** `Employees` по свойството им `department`.

За целта се ползва `static` метода `groupingBy` на клас `Collectors` като параметър на метода `collect()`

Методът `collect()` има **версия с един параметър** и в този случай параметърът е метод, който групира обектите в `Stream`-а по отношение на стойността, връщана от този метод. В случая това е `department`.

Стойностите върнати от този метод са ключове (`key`) в колекция `Map`. Тази колекция е съставена от уникални ключове и съответните им стойности (`key-value pair`) и представлява резултата от групирането.

Стойностите съответстващи на тези ключове по подразбиране са `Lists` съставени от `stream` елементите на дадена група.

Методът `forEach` на `Map` колекцията обхожда последователно всяка група `key-value pair`. Този метод приема обект от тип `interface BiConsumer`. `BiConsumer`'s има `accept()` метод с два параметъра – първият е ключът (`key`) в случая `department`, а вторият (`value`) е съответната му стойност (в случая `List<Employee>`).



8.5 Групиране на Employees

```

1 // group Employees by department
2 System.out.printf("%nEmployees by department:%n");
3 Map<String, List<Employee>> groupedByDepartment =
4     list.stream()
5         .collect(Collectors.groupingBy(Employee::getDepartment));
6 groupedByDepartment.forEach(
7     (department, employeesInDepartment) ->
8     {
9         System.out.println(department); // printout the key
10        employeesInDepartment.forEach(// printout the values- elements of a List<Employee>
11            employee -> System.out.printf("    %s%n", employee));
12    }
13 );

```

Employees by department:

Sales

Matthew	Indigo	3587.50	Sales
Jason	Blue	3200.00	Sales

IT

Jason	Red	5000.00	IT
Ashley	Green	7600.00	IT
Luke	Indigo	6200.00	IT

Marketing

James	Indigo	4700.77	Marketing
Wendy	Brown	4236.40	Marketing

Извеждане на имена на групи и елементи на групи при групиране (резултатът от групирането е обект от тип `Map<T, List<E>>`)

8.5 Групиране на Employees

В следващия пример да разгледаме приложението на `Stream` метода `collect()` и на `static` метода `groupingBy()` на `Collectors` за групиране на `Employees` по броя на `Employees` във всеки `department`.

В този случай използваме **версия на `static` метода `groupingBy()`** с два параметъра. Както преди, **първият** параметър е метод, който връща стойност, по която се извършва групирането. Вторият параметър е друг `Collector`, наричан вторичен `Collector`.

Вторичният `collector` създаваме със статичния метод `counting()` на клас `Collectors`. Този метод **генерира броя на обектите във всяка отделна група** вместо да добавя тези обекти в `List`.



8.5 Групиране на Employees

```
1 // count number of Employees in each department
2 System.out.printf("%nCount of Employees by department:%n");
3 Map<String, Long> employeeCountByDepartment =
4     list.stream()
5         .collect(Collectors.groupingBy(Employee::getDepartment,
6                                         Collectors.counting()));
7 employeeCountByDepartment.forEach(/* the values are the counters
8     (department, count) -> System.out.printf(
9         "%s has %d employee(s)%n", department, count));
```

Count of Employees by department:
IT has 3 employee(s)
Marketing has 2 employee(s)
Sales has 2 employee(s)

Извеждане на ключ и стойност на
обект от тип Map

8.5.1 Групиране със сортиране

Резултатът от групиране на `Stream` е `Map<K,V>`, където `K` е типът на стойността по отношение на която се групира, а `V` е типът на стойностите съответстващи на ключа. В предходния пример `List<Employee>`, а в последния пример `V` беше `Long`.

По подразбиране `Map<K,V>` не е сортиран. Ако искаме групите `Map<K,V>` да са сортирани по ключовете(групата), то може да ползваме версия на `Collectors.groupingBy()` с три аргумента, където вторият аргумент е от тип `Supplier<Map<K,V>>`. Този аргумент служи за създаване на потребителски дефиниран тип `Map<K,V>` за съхраняване на резултата от групирането. В случая ползваме `TreeMap<String, Long>::new` и това сортира групите по ключа `department`



8.5.1 Групиране със сортиране

1	<code>// count number of Employees in each department</code>
2	<code>System.out.printf("%nCount of Employees by department:%n");</code>
3	<code>Map<String, Long> employeeCountByDepartment =</code>
4	<code>list.stream()</code>
5	<code>.collect(Collectors.groupingBy(Employee::getDepartment,</code>
6	<code>TreeMap<String, Long>::new, Collectors.counting()));</code>
7	<code>employeeCountByDepartment.forEach(</code>
8	<code>(department, count) -> System.out.printf(</code>
9	<code>"%s has %d employee(s)%n", department, count));</code>

```
Count of Employees by department:
IT has 3 employee(s)
Marketing has 2 employee(s)
Sales has 2 employee(s)
```

8.6 Обобщаване на суми и средни стойности

Следващият пример демонстрира приложението на метода **mapToDouble()** от **Stream**, при което обектите **Employees** се проектират в стойности от тип **double** и в резултат се връща **DoubleStream**.

Този метод взима за параметър обект, който имплементира функционалния интерфейс **interface ToDouble method** (package **java.util.method**). Този интерфейс има метод **applyAsDouble()** и е **съвместим с методи**, които имат параметър от тип **Object** и връщат **double** стойност.

8.6 Обобщаване на суми и средни стойности

```

1 // calculate sum of Employee salaries with DoubleStream sum method
2 System.out.printf(
3     "%nSum of Employees' salaries (via sum method): %.2f%n",
4     list.stream()
5         .mapToDouble(Employee::getSalary)
6         .sum());
7
8 // calculate sum of Employee salaries with Stream reduce method
9 System.out.printf(
10    "Sum of Employees' salaries (via reduce method): %.2f%n",
11    list.stream()
12        .mapToDouble(Employee::getSalary)
13        .reduce(0, (value1, value2) -> value1 + value2));
14
15 // calculate average of Employee salaries with DoubleStream average method
16 System.out.printf("Average of Employees' salaries: %.2f%n",
17    list.stream()
18        .mapToDouble(Employee::getSalary)
19        .average()
20        .getAsDouble());

```

```

Sum of Employees' salaries (via sum method): 34524.67
Sum of Employees' salaries (via reduce method): 34525.67
Average of Employees' salaries: 4932.10

```



9 Допълнителни примери за сортиране и групиране

Следващите примери демонстрират допълнителни средства за групиране и сортиране.

За онагледяване на резултатите ще използваме следния клас

```
class Item {  
  
    private String name;  
    private int qty;  
    private double price;  
  
    //constructors, getter/setters toString() method  
}
```

9.1 Допълнителни примери за сортиране и групиране

В този пример се извършва **групиране по свойството name** на **Item** и **с всяка група се извежда общата стойност на друго свойство на Item, а именно qty.**

```

1 List<Item> items = Arrays.asList(
2     new Item("apple", 10, Double.parseDouble("9.99")),
3     new Item("banana", 20, Double.parseDouble("19.99")),
4     new Item("orange", 10, Double.parseDouble("29.99")),
5     new Item("watermelon", 10, Double.parseDouble("29.99")),
6     new Item("papaya", 20, Double.parseDouble("9.99")),
7     new Item("apple", 10, Double.parseDouble("9.99")),
8     new Item("banana", 10, Double.parseDouble("19.99")),
9     new Item("apple", 20, Double.parseDouble("9.99"))
10 );
11 Map<String, Integer> sum = items.stream().collect(
12     Collectors.groupingBy(Item::getName,
13         Collectors.summingInt(Item::getQty));
14 System.out.println(sum);

```

```

run:
{papaya=20, orange=10, banana=30, apple=40, watermelon=10}
BUILD SUCCESSFUL (total time: 0 seconds)

```



9.2 Допълнителни примери за сортиране и групиране

В този пример ще сравним **два** **способа за групиране**.

При **първия** всяка група се състои от списък с обекти. Това е най-обикновеният способ за групиране. Така в този случай получаваме **Map<Double, List<Item>>**

При **втория** способ преобразуваме обектите **Item** от списъците във всяка група в множество от уникални елементи, които се получават с проектиране на свойство (**name**) на обектите в списъците.

Така в този случай получаваме **Map<Double, Set<String>>**

9.2 Допълнителни примери за сортиране и групиране

```

1 List<Item> items = Arrays.asList(
2     new Item("apple", 10, Double.parseDouble("9.99")),
3     new Item("banana", 20, Double.parseDouble("19.99")),
4     new Item("orange", 10, Double.parseDouble("29.99")),
5     new Item("watermelon", 10, Double.parseDouble("29.99")),
6     new Item("papaya", 20, Double.parseDouble("9.99")),
7     new Item("apple", 10, Double.parseDouble("9.99")),
8     new Item("banana", 10, Double.parseDouble("19.99")),
9     new Item("apple", 20, Double.parseDouble("9.99"))
10 );
11 //groups by price, creates a List<Item> per group
12 Map<Double, List<Item>> groupByPriceMap
13     = items.stream().collect(Collectors.groupingBy(Item::getPrice));
14
15 System.out.println(groupByPriceMap);
16 // groups by price, uses 'mapping' to convert List<Item> to Set<String>
17 // groups by price, creates a Set<String> of names per group
18 Map<Double, Set<String>> result
19     = items.stream().collect(Collectors.groupingBy(Item::getPrice,
20         Collectors.mapping(Item::getName, Collectors.toSet())
21     ));
22
23 System.out.println(result);

```

run:

```

{29.99=[Item{name=orange, qty=10, price=29.99}, Item{name=watermelon, qty=10, price=29.99}],
 9.99=[Item{name=apple, qty=10, price=9.99 }, Item{name=papaya, qty=20, price=9.99 },
      Item {name=apple, qty=10, price=9.99 }, Item{name=apple, qty=20, price=9.99 }],
19.99=[Item{name=banana, qty=20, price=19.99}, Item{name=banana, qty=10, price=19.99}]
}

```

```
{29.99=[orange, watermelon], 9.99=[papaya, apple], 19.99=[banana]}
```



В този пример ще демонстрираме приложението на **Function.Identity** при групиране на множество от дублирани обекти

```
1 List<String> items = Arrays.asList("apple", "apple", "banana", "apple",
2                                   "orange", "banana", "papaya");
3 Map<String, Long> result
4     = items.stream()
5         .collect(Collectors.groupingBy(Function.identity(),
6                                         Collectors.counting()))
7     );
8 System.out.println(result);
```

```
{papaya=1, orange=1, banana=2, apple=3}
BUILD SUCCESSFUL (total time: 0 seconds)
```

9.5 Допълнителни примери за сортиране и групиране

Нека сега да сортираме резултатите от предходното групиране

а) В намаляващ ред на стойностите (**броят на всеки плод в списъка** `items`)

б) В нарастващ ред на ключовите стойности (**имената на плодовете**)

Резултатът, който ще получим е следният

```
run:
```

```
[apple=3, banana=2, papaya=1, orange=1]
```

```
[apple=3, banana=2, orange=1, papaya=1]
```

```
BUILD SUCCESSFUL (total time: 0 seconds)
```



9.5 Допълнителни примери за сортиране и групиране

```

1 List<String> items = Arrays.asList("apple", "apple", "banana", "apple",
2                                   "orange", "banana", "papaya");
3 Map<String, Long> result
4   = items.stream()
5         .collect(Collectors.groupingBy(Function.identity(),
6                                         Collectors.counting() )
7   );
8 // Sort groups by Map<String, Long> in reverse order of the values
9 List<Map.Entry<String, Long>> list = result.entrySet().stream()
10    .sorted(Map.Entry.<String, Long>comparingByValue().reversed())
11    .collect(Collectors.toList());
12 System.out.println(list);
13 // Sort groups by Map<String, Long> keys
14 list = result.entrySet().stream()
15    .sorted(Map.Entry.<String, Long>comparingByKey())
16    .collect(Collectors.toList());
17 System.out.println(list);

```

Обърнете внимание как се преобразува типа, връщан от `comparingByValue()` и `comparingByKey()`

`Map<String, Long>` се преобразува в списък от `Map.Entry<String, Long>` обекти, които се сортират по свойството `value` в намаляващ ред.

9.6 Допълнителни примери за сортиране и групиране

За целта ползваме методи на `Map.Entry`

- `comparingByKey()`
- `comparingByKey(Comparator<? super K> cmp)`
- `comparingByValue()`
- `comparingByValue(Comparator<? super K> cmp)`

В предходния пример

```
Map.Entry.<String, Long>comparingByValue().reversed()
```

е задължително да се укажат параметрите за тип на `Comparator`

По- подробно може да напишем

```
Comparator<Map.Entry<String, Long>> cmp =  
    Map.Entry.comparingByValue();  
cmp.reversed();
```



9.6 Допълнителни примери за сортиране и групиране

Нека сега да сортираме групираните данни и резултатът да съхраним в друга колекция от данни.

В предходния пример получаваме групираните данни в `Map<String, Long>`. След сортирането им ще ги съхраним в друг `Map<String, Long>`

run:

```
[apple=3, banana=2, papaya=1, orange=1]
```

```
[apple=3, banana=2, orange=1, papaya=1]
```

```
BUILD SUCCESSFUL (total time: 0 seconds)
```



9.6 Допълнителни примери за сортиране и групиране

```

1 List<String> items = Arrays.asList("apple", "apple", "banana", "apple",
2                                   "orange", "banana", "papaya");
3 Map<String, Long> result
4   = items.stream()
5     .collect(Collectors.groupingBy(Function.identity(),
6                                     Collectors.counting()))
7   );
8 Map<String, Long> finalMap = new LinkedHashMap<>();
9 result.entrySet().stream()
10  .sorted(Map.Entry.<String, Long>comparingByValue().reversed())
11  .forEachOrdered(e -> finalMap.put(e.getKey(), e.getValue()));
12 System.out.println(finalMap);

```

run:

```
{apple=3, banana=2, papaya=1, orange=1}
```

```
BUILD SUCCESSFUL (total time: 0 seconds)
```



Задачи

1. Групирайте по името `name` и обща стойност на цената `price` за всяко име

1	<code>List<Item> items = Arrays.asList(</code>
2	<code>new Item("apple", 10, Double.parseDouble("9.99")),</code>
3	<code>new Item("banana", 20, Double.parseDouble("19.99")),</code>
4	<code>new Item("orange", 10, Double.parseDouble("29.99")),</code>
5	<code>new Item("watermelon", 10, Double.parseDouble("29.99")),</code>
6	<code>new Item("papaya", 20, Double.parseDouble("9.99")),</code>
7	<code>new Item("apple", 10, Double.parseDouble("9.99")),</code>
8	<code>new Item("banana", 10, Double.parseDouble("19.99")),</code>
9	<code>new Item("apple", 20, Double.parseDouble("9.99"))</code>
10	<code>);</code>

```
class Item {
```

```
    private String name;
    private int qty;
    private double price;
```

```
    //constructors, getter/setters toString() method
```

```
}
```



Задачи

2. Групирайте имената **name** по количество **qty** (имената в отделните групи да не се дублират)

1	List<Item> items = Arrays.asList(
2	new Item("apple", 10, Double.parseDouble("9.99")),
3	new Item("banana", 20, Double.parseDouble("19.99")),
4	new Item("orange", 10, Double.parseDouble("29.99")),
5	new Item("watermelon", 10, Double.parseDouble("29.99")),
6	new Item("papaya", 20, Double.parseDouble("9.99")),
7	new Item("apple", 10, Double.parseDouble("9.99")),
8	new Item("banana", 10, Double.parseDouble("19.99")),
9	new Item("apple", 20, Double.parseDouble("9.99"))
10);

```
class Item {

    private String name;
    private int qty;
    private double price;

    //constructors, getter/setters toString() method
}

// очакван резултат
{20=[banana, papaya, apple], 10=[orange, banana, apple, watermelon]}
```

