

# JAVA SE

**Базовый курс. Лекция 14**

# Хорошо ли меня слышно? Видно ли презентацию?

Напишите “+” в чат, если  
все ок.

Сообщите в чате,  
если есть  
проблемы =)



# План лекции

1. От итерации к потоковым операциям
2. Создание потока данных
3. Методы `filter()`, `map()` и `flatMap()`
4. Извлечение и соединение потоков данных
5. Другие потоковые преобразования
6. Простые методы сведения
7. Тип `Optional`



# Потоки данных (Stream API)

Потоки данных обеспечивают представление данных, позволяющее выполнять вычисления на более высоком концептуальном уровне, чем коллекции.

С помощью потока данных можно указать, **что** и **как именно** требуется сделать с данными.

Средствами Stream API можно оптимизировать вычисления, используя, допустим, несколько потоков исполнения для расчета сумм, подсчета и объединения результатов.

# От итерации к потоковым операциям

Для обработки коллекции обычно требуется перебрать ее элементы и выполнить над ними некоторую операцию.

Допустим, что требуется подсчитать все длинные слова в книге.



**Демонстрация кода (Demo1.java, Demo2.java, Demo3.java)**



# От итерации к потоковым операциям

Потоки данных действуют по принципу “что, а не как делать”.

Поток данных похож на коллекцию, так как он позволяет преобразовывать и извлекать данные. Но у потока данных есть следующие отличия:

- 1) Поток данных не сохраняет свои элементы. Они могут храниться в основной коллекции или формироваться по требованию.
- 2) Потоковые операции не изменяют источник данных.
- 3) Потоковые операции выполняются по требованию, когда это возможно.

# От итерации к потоковым операциям

Конвейер операций организуется в следующие три стадии:

- 1) Создание потока данных
- 2) Указание промежуточных операция для преобразования исходного потока данных в другие потоки, возможно, в несколько этапов.
- 3) Выполнение **терминальной** операции для получения результата.





# Создание потока данных

- 1) Пустой стрим: `Stream.empty()`
- 2) Стрим из List: `list.stream()`
- 3) Стрим из Map: `map.entrySet().stream()`
- 4) Стрим из массива: `Arrays.stream(array)`
- 5) Стрим из указанных элементов: `Stream.of("a", "b", "c")`



## Демонстрация кода (Demo4.java)



# Интересно знать (!)

В прикладном программном интерфейсе Java API есть ряд методов, возвращающих потоки данных. В классе **Pattern** есть метод **splitAsStream()**, разделяющий последовательность символов типа **CharSequence** по регулярному выражению.

Пример разделения символьной строки на отдельные слова:

```
public static void main(String[] args) throws IOException {  
    //Считываем данные из файла  
    byte[] bytes = Files.readAllBytes(Paths.get("data.txt"));  
    String contents = new String(bytes, StandardCharsets.UTF_8);  
  
    Stream<String> words = Pattern.compile("\\PL+").splitAsStream(contents);  
}
```

# Интересно знать (!)

Статический метод **Files.lines()** возвращает поток данных типа **Stream**, содержащий все строки из файла:

```
public static void main(String[] args) throws IOException {  
    try(Stream<String> lines = Files.lines(Paths.get("demo.txt"))){  
        //обработать строки  
    }  
}
```

# Методы filter(), map() и flatMap()

В результате преобразования потока данных получается другой поток данных, элементы которого являются производными от элементов исходного потока.

Поток символьных строк преобразуется в другой поток, содержащий только длинные слова. В качестве аргумента метода **filter()** указывается объект типа **Predicate<T>**, т.е. функция, преобразующая тип **T** в логический тип **boolean**.

```
public static void main(String[] args) throws IOException {  
  
    //Считываем данные из файла  
    byte[] bytes = Files.readAllBytes(Paths.get( first: "data.txt"));  
  
    //Преобразуем текст из файла в символьную строку  
    String contents = new String(bytes, StandardCharsets.UTF_8);  
  
    //Выделяем из всего файла - список слов  
    List<String> words = Arrays.asList(contents.split( regex: "\\PL+"));  
  
    //Осуществляем преобразование  
    final Stream<String> longWords = words  
        .stream()  
        .filter(w -> w.length() > 12);  
  
}
```

# Методы filter(), map() и flatMap()

Часто значения в потоке данных требуется преобразовать каким-то образом.

Для этой цели можно воспользоваться методом **map()**, передав ему функцию, которая выполняет нужное преобразование.

Например, буквы во всех словах можно сделать строчными следующим образом:

```
public static void main(String[] args) throws IOException {  
  
    //Считываем данные из файла  
    byte[] bytes = Files.readAllBytes(Paths.get("data.txt"));  
  
    //Преобразуем текст из файла в символьную строку  
    String contents = new String(bytes, StandardCharsets.UTF_8);  
  
    //Выделяем из всего файла - список слов  
    List<String> words = Arrays.asList(contents.split("\\PL+"));  
  
    //Осуществляем преобразование  
    final Stream<String> lowerWords = words  
        .stream()  
        .map(String::toLowerCase); //здесь передается ссылка на метод  
  
}
```

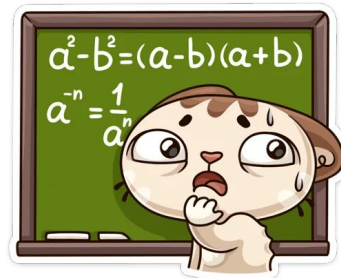
# Методы filter(), map() и flatMap()

```
public static void main(String[] args) throws IOException {  
  
    //Считываем данные из файла  
    byte[] bytes = Files.readAllBytes(Paths.get("data.txt"));  
  
    //Преобразуем текст из файла в символьную строку  
    String contents = new String(bytes, StandardCharsets.UTF_8);  
  
    //Выделяем из всего файла - список слов  
    List<String> words = Arrays.asList(contents.split("\\PL+"));  
  
    //Полученный поток содержит первую букву каждого слова  
    final Stream<String> firstLetters = words  
        .stream()  
        .map(s -> s.substring(0, 1)); //здесь передаем лямбду  
  
}
```

Можем получить новый поток данных, содержащий только первые буквы от каждого слова, передав в map() лямбду

# Методы `filter()`, `map()` и `flatMap()`

При вызове метода `map()` передаваемая ему функция применяется к **каждому элементу** потока данных, в результате чего образуется новый поток данных с полученными результатами





# Демонстрация кода (Demo10.java)

```
/**
```

```
* Допустим у нас есть метод, возвращающий не одно значение, а поток значений.
```

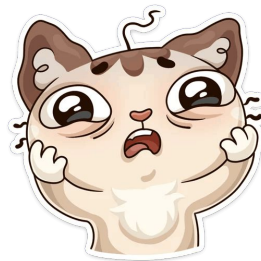
```
* Например, в результате вызова letters("boat")
```

```
* образуется поток данных ["b", "o", "a", "t"]
```

```
*/
```



## Интересно знать (!)



Аналогичный метод `flatMap()` можно обнаружить и в других классах (не только для работы с потоками данных `Stream API`). Он отражает общий принцип вычислительной техники.

Допустим, что имеется обобщенный тип `G` (например, `Stream`) и функции `f()` и `g()`, преобразующие некоторый тип `T` в тип `G<U>`, а `U` в тип `G<V>` соответственно.

В этом случае функции можно составить вместе, используя метод `flatMap()`, т.е. применить сначала функцию `f()`, а затем функцию `g()`.

(В этом состоит главная идея монад, но использовать `flatMap()` можно и без знания о монадах)

# Извлечение и соединение потоков данных

В результате вызова `поток.limit(n)` возвращается поток данных, заканчивающийся после `n` элементов или по завершении исходного потока данных, если тот короче.

Поток данных, состоящий из 100 произвольных чисел:

```
public static void main(String[] args) {  
    final Stream<Double> limit = Stream.generate(Math::random).limit(100);  
}
```

# Извлечение и соединение потоков данных

В результате вызова `поток.skip(n)` происходит совершенно противоположное: отбрасываются `n` элементов.

Избавиться от нежелательной первой пустой строки можно так:

```
public static void main(String[] args) throws IOException {  
    //Считываем данные из файла  
    byte[] bytes = Files.readAllBytes(Paths.get(PATH_TO_DEMO_TXT_FILE));  
  
    //Преобразуем текст из файла в символьную строку  
    String contents = new String(bytes, StandardCharsets.UTF_8);  
  
    //Выделяем из всего файла - список слов и пропуск первой пустой строки  
    final Stream<String> skip = Stream.of(contents.split( regex: "\\PL+" )).skip(1);  
}
```

# Извлечение и соединение потоков данных

Два потока данных можно соединить вместе с помощью статического метода `concat()` из интерфейса `Stream()`. Первый из этих потоков не должен быть бесконечным, иначе второй поток не сможет соединиться с ним:

```
public static void main(String[] args) {  
    final Stream<String> combined = Stream.concat(letters( s: "Hello"), letters( s: "World"));  
    //получается следующий поток данных ["H", "e", "l", "l", "o", "W", "o", "r", "l", "d"]  
}  
  
public static Stream<String> letters(String s) {  
    List<String> result = new ArrayList<>();  
    for (int i = 0; i < s.length(); i++) {  
        result.add(s.substring(i, i + 1));  
    }  
    return result.stream();  
}
```

## Другие потоковые преобразования

Метод **distinct()** возвращает поток данных, получающий свои элементы из исходного потока данных в том же самом порядке, за исключением того, что дубликаты в нем подавляются.

Пример:

```
public static void main(String[] args) {  
    final Stream<String> distinct = Stream.of("Hello", "Hello", "Hello", "world").distinct();  
    System.out.println(distinct.count()); // 2  
}
```

# Другие потоковые преобразования

Для сортировки потоков данных имеется несколько вариантов метода `sorted()`.

Один из них служит для обработки потоков данных состоящих из элементов типа `Comparable`, а другой принимает в качестве параметра компаратор типа `Comparator`.

Пример сортировки строк таким образом, чтобы первой в потоке данных следовала самая длинная строка:

```
public static void main(String[] args) throws IOException {  
    //Считываем данные из файла  
    byte[] bytes = Files.readAllBytes(Paths.get(PATH_TO_DEMO_TXT_FILE));  
  
    //Преобразуем текст из файла в символьную строку  
    String contents = new String(bytes, StandardCharsets.UTF_8);  
  
    //Выделяем из всего файла - список слов  
    List<String> words = Arrays.asList(contents.split(regex: "\\PL+"));  
  
    //метод sorted() выдает новый поток данных, элементы которого поражаются  
    // из исходного потока и располагаются в отсортированном порядке  
    final Stream<String> sorted = words  
        .stream()  
        .sorted(Comparator.comparing(String::length).reversed());  
}
```

## Другие потоковые преобразования

Метод `peek()` выдает другой поток данных с теми же самыми элементами, что и у исходного потока, но передаваемая ему функция вызывается всякий раз, когда извлекается элемент. Это удобно для отладки:

```
public static void main(String[] args) throws IOException {  
    final Object[] powers = Stream.iterate( seed: 1.0, p -> p * 2)  
        .peek(e -> System.out.println("Fetching " + e))  
        .limit(20)  
        .toArray();  
}
```



# Демонстрация кода (Demo17.java)



# Простые методы сведения (терминальные методы)

Методы сведения выполняют терминальные операции, сводя поток данных к не потоковому значению, которое можно далее использовать в программе.

Метод `count()` является терминальным.

Методы `max()`, `min()` также терминальные. Но они на самом деле возвращают значение типа **`Optional<T>`**, которое включает в себе ответ на запрос данных из потока или обозначает, что запрашиваемые данные отсутствуют, так как поток мог быть пустым.

# Простые методы сведения (терминальные методы)

Пример получения максимального значения из потока данных:

```
public static void main(String[] args) throws IOException {  
    //Считываем данные из файла  
    byte[] bytes = Files.readAllBytes(Paths.get(PATH_TO_DEMO_TXT_FILE));  
  
    //Преобразуем текст из файла в символьную строку  
    String contents = new String(bytes, StandardCharsets.UTF_8);  
  
    //Выделяем из всего файла - список слов  
    List<String> words = Arrays.asList(contents.split( regex: "\\PL+"));  
  
    final Optional<String> largest = words  
        .stream()  
        .max(String::compareToIgnoreCase);  
    System.out.println("largest: " + largest.orElse( other: "ничего не нашлось"));  
}
```

# Простые методы сведения (терминальные методы)

Метод **findFirst()**  
возвращает первое значение  
из непустой коллекции.  
Часто он применяется  
вместе с методом **filter()**.

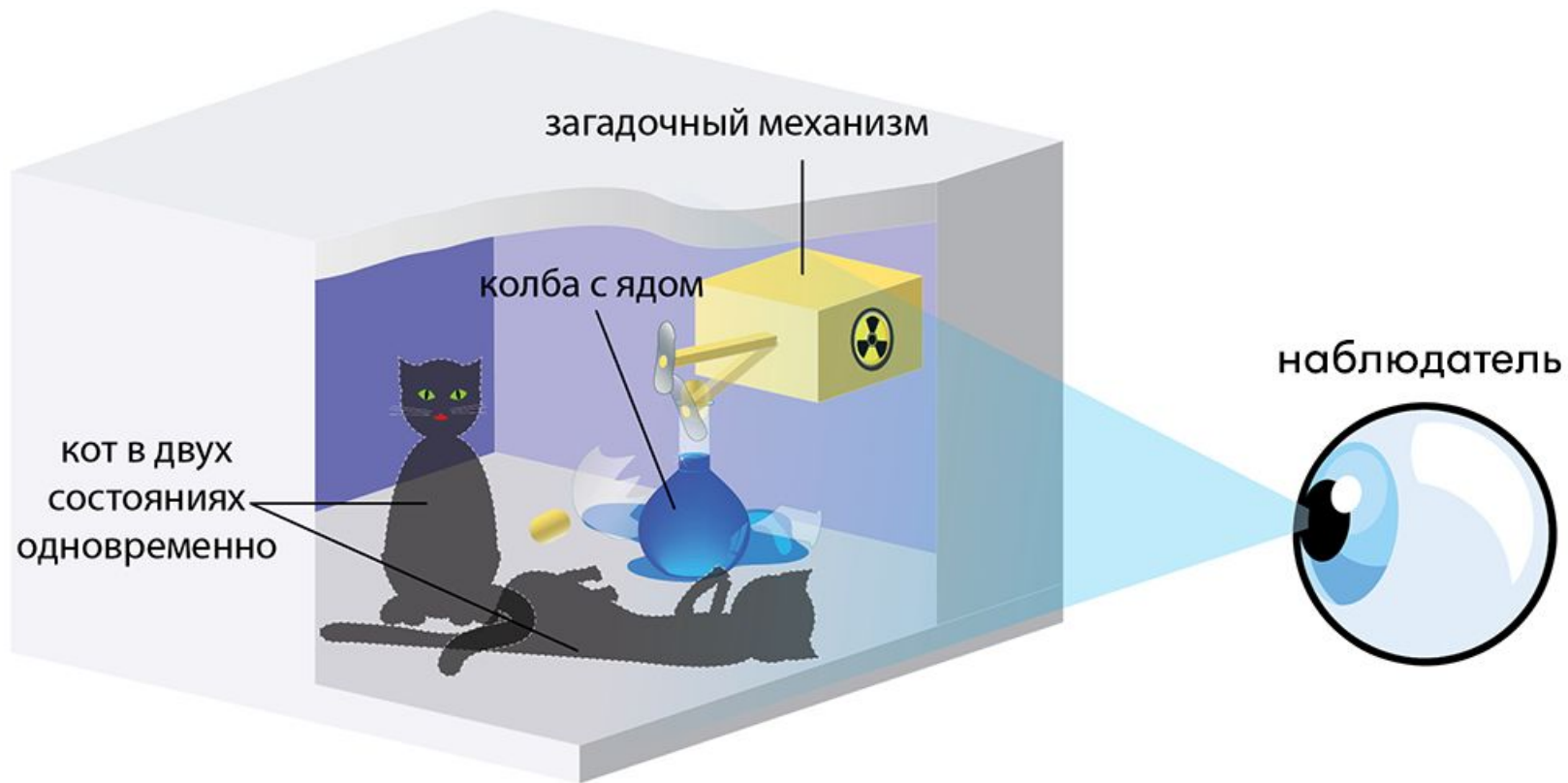
Поиск первого слова,  
начинающегося на букву  
"Q":

```
public static void main(String[] args) throws IOException {  
    //Считываем данные из файла  
    byte[] bytes = Files.readAllBytes(Paths.get(PATH_TO_DEMO_TXT_FILE));  
  
    //Преобразуем текст из файла в символьную строку  
    String contents = new String(bytes, StandardCharsets.UTF_8);  
  
    //Выделяем из всего файла - список слов  
    List<String> words = Arrays.asList(contents.split( regex: "\\PL+" ));  
  
    //Поиск первого слова, начинающегося с буквы "Q"  
    final Optional<String> startWithQ = words  
        .stream()  
        .filter(s -> s.startsWith("Q"))  
        .findFirst();  
}
```

# Демонстрация кода (Demo20.java, Demo21.java)



# Тип Optional



# Тип Optional

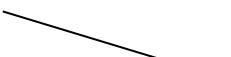
Объект типа `Optional<T>` служит оболочкой для объекта обобщенного типа `T`.

Для эффективного применения типа `Optional` самое главное - выбрать метод, который возвращает альтернативный вариант, если значение отсутствует, или использует его, если оно присутствует.

# Как работать с необязательными значениями

Часто имеется значение, возможно, пустая строка "", которое требуется использовать по умолчанию в отсутствии совпадения:

```
public static void main(String[] args) {  
    final Optional<String> optionalString = Optional.of("Data"); //допустим строку получаем по сети  
    //получаем либо строку либо ""  
    final String s = optionalString.orElse( other: "");  
    System.out.println(s); //Data  
}
```




```
public static void main(String[] args) {  
    String data = null; //допустим строку получаем по сети  
    final Optional<String> optionalString = Optional.ofNullable(data);  
    //получаем либо строку либо ""  
    final String s = optionalString.orElse( other: "");  
    System.out.println(s); //"" пустая строка  
}
```



# Как работать с необязательными значениями

Также можно вызвать функцию, для вычисления значения, в случае, если `Optional` содержит пустоту:

```
public static void main(String[] args) {  
    String data = null;  
    final Optional<String> optionalString = Optional.ofNullable(data);  
  
    final String result = optionalString.orElseGet(() -> System.getProperty("user.dir"));  
    System.out.println(result);  
}
```



---

```
/Users/elenaoshkina/.sdkman/candidates/java/8.0.265-zulu/zulu-8.jdk/Contents/Home/bin/java ...
```


```
Результат=/Users/elenaoshkina/IdeaProjects/lesson-14
```

```
Process finished with exit code 0
```

# Как работать с необязательными значениями

Если значение отсутствует можно сгенерировать исключение:

```
public static void main(String[] args) {  
    String data = null;  
    final Optional<String> optionalString = Optional.ofNullable(data);  
  
    final String result = optionalString.orElseThrow(IllegalStateException::new);  
    System.out.println("Результат=" + result);  
}
```



```
/Users/elenaoshkina/.sdkman/candidates/java/8.0.265-zulu/zulu-8.jdk/Contents/Home/bin/java ...
```

```
Exception in thread "main" java.lang.IllegalStateException Create breakpoint
```

```
at java.util.Optional.orElseThrow(Optional.java:290)
```

```
at ru.oshkina.demo7.Demo24.main(Demo24.java:11)
```

# Как работать с необязательными значениями

Другая методика обращения с необязательными значениями заключается в том, чтобы воспользоваться значением только если оно присутствует:

Метод `ifPresent()` принимает функцию в качестве аргумента. если необязательное значение существует, оно передается данной функции, иначе ничего не происходит:

```
optionalValue.ifPresent(v -> Обработать v);
```

Если во множество стоит добавить значение:

```
optionalValue.ifPresent(v -> results.add(v));
```

или просто:

```
optionalValue.ifPresent(results::add);
```

# Демонстрация кода (Demo25.java)



# Как НЕ следует работать с необязательными значениями

Метод `get()` получает заключенный в оболочку `Optional` элемент значения типа `Optional`, если это значение существует, а иначе - генерирует исключение типа `NoSuchElementException`.

```
Optional<T> optionalValue = ...;
```

```
optionalValue.get().method(); //(!) достаем без проверки, а вдруг там null
```

**не надежнее, чем:**

```
T value = ....;
```

```
value.someMethod();
```

# Как НЕ следует работать с необязательными значениями

Метод `isPresent()` извещает, содержит ли значение объект типа `Optional<T>`. А вот выражение:

```
if(optionalValue.isPresent()) {  
    optionalValue.get().someMethod();  
}
```

не проще, чем:

```
if (value != null) {  
    value.someMethod();  
}
```

# Резюме

- 1) Итераторы подразумевают конкретную методику обхода и препятствуют организации эффективного параллельного выполнения
- 2) Потоки данных можно создавать из коллекций, массивов, генераторов или итераторов.
- 3) Для отбора элементов из потока служит метод `filter()`, а для преобразования - метод `map()`.
- 4) Другие операции преобразования потоков данных реализуются методами `limit()`, `distinct()`, `sorted()`
- 5) Для получения результата из потока данных служат операции сведения, реализуемые в частности методами `count()`, `max()`, `min()`, `findFirst()`, `findAny()`. Некоторые из этих методов возвращают необязательное значение `Optional`.
- 6) Тип `Optional` специально предназначен в качестве надежной альтернативы обработке пустых значений `null`. Для надежного применения этого типа служат методы `ifPresent()` и `orElse()`