HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

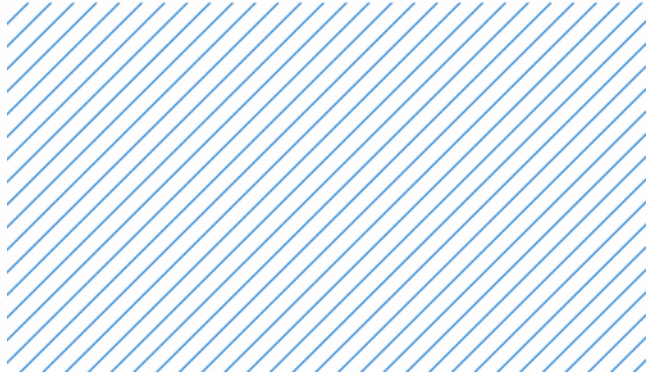
Лекция 14

Стримы

Программирование на языке Java

Роман Гуров

ВШЭ БИ 2021



Streams API

Стрим (англ. **stream** – поток) – последовательность (возможно бесконечная) некоторых объектов, которая позволяет применить к себе последовательность преобразований

Стримы позволяют описывать эти преобразования без использования привычных нам циклов и условных операторов – одной линейной цепочкой

Стримы вместе с функциональными интерфейсами являются ещё одним нововведением `Java 8`, которое приближает язык в сторону *функционального программирования*

Интерфейс Stream

```
package java.util.stream;

public interface Stream<T> extends BaseStream<T, Stream<T>> {
    // ОЧЕНЬ много методов
}
```

Дженерик-параметр `T` задаёт тип элементов в стриме

Также, есть версии `IntStream`, `LongStream` и `DoubleStream` для примитивных типов

В отличие от итератора, который просто позволяет по одному последовательно получить элементы, стрим предоставляет огромное количество методов

Стрим даёт средства полноценного описания алгоритма *обработки и преобразования последовательности*

Стримы vs коллекции

Также, стрим отличается и от коллекций

Коллекция хранит в себе все элементы, а значит конечна
Стриму это не требуется, он может быть бесконечным

Коллекция часто позволяет явно обратиться к элементу, например, по индексу или ключу
Стрим такой возможности не даёт

Коллекция позволяет изменять свой набор элементов
Преобразования стрима никак не изменяют оригинальный источник из которого берутся элементы

Использование стрима

Работа со стримом проходит в три этапа:

1. Создание стрима
2. Промежуточные операции преобразования
3. Терминальная операция

Создание стрима

Рассмотрим некоторые способы создать стрим

Стрим можно получить из любой коллекции её методом `stream()`:

```
Set<String> vocabulary = /* ... */;
Stream<String> stream1 = vocabulary.stream();
```

`BufferedReader` (о нём узнаем на другой лекции) умеет создать стрим по строчкам текста:

```
BufferedReader reader = /* ... */;
Stream<String> stream2 = reader.lines();
```

Также, есть способ итерироваться по файлам (их путям) в директории (но о файлах тоже не сегодня):

```
Path path = /* ... */;
Stream<Path> stream3 = Files.list(path);
Stream<Path> stream4 = Files.walk(path);
```

В отличие от `list()`, `walk()` заходит и во вложенные директории, производя рекурсивный обход всего дерева

Создание стрима

Рассмотрим некоторые способы создать стрим

Можно получить стрим символов из строки методом `chars()`

```
IntStream chars = "hello".chars();
```

Так как не существует `CharStream`, этот метод возвращет стрим интов, что не очень удобно

Есть динамические способы генерации стримов
Например, генерация с помощью `Supplier`'а, который будет вызываться для получения очередного элемента

```
DoubleStream randomNumber = DoubleStream.generate(Math::random);
```

Создание стрима

Рассмотрим некоторые способы создать стрим

Методом `iterate()` стрим можно создать как бесконечную последовательность $N_{i+1} = f(N_i)$ N_0 задаётся первым аргументом метода

```
IntStream integers = IntStream.iterate(0, n -> n + 3);
```

=> $0, 3, 9, 12, 15, \ldots$

Стрим можно задать по целочисленному промежутку:

```
IntStream rangeIntegers = IntStream.range(0, 100);

IntStream rangeIntegers2 = IntStream.rangeClosed(0, 100);
```

В первом случае будет полуинтервал $[0, 100)$
Во втором – отрезок $[0, 100]$

Создание стрима

Рассмотрим некоторые способы создать стрим

Два стрима можно последовательно склеить в один:

```
IntStream combinedStream = IntStream.concat(rangeIntegers, integers);
```

Создание заведомо пустого стрима:

```
IntStream empty = IntStream.empty();
```

Создание стрима из обычного массива:

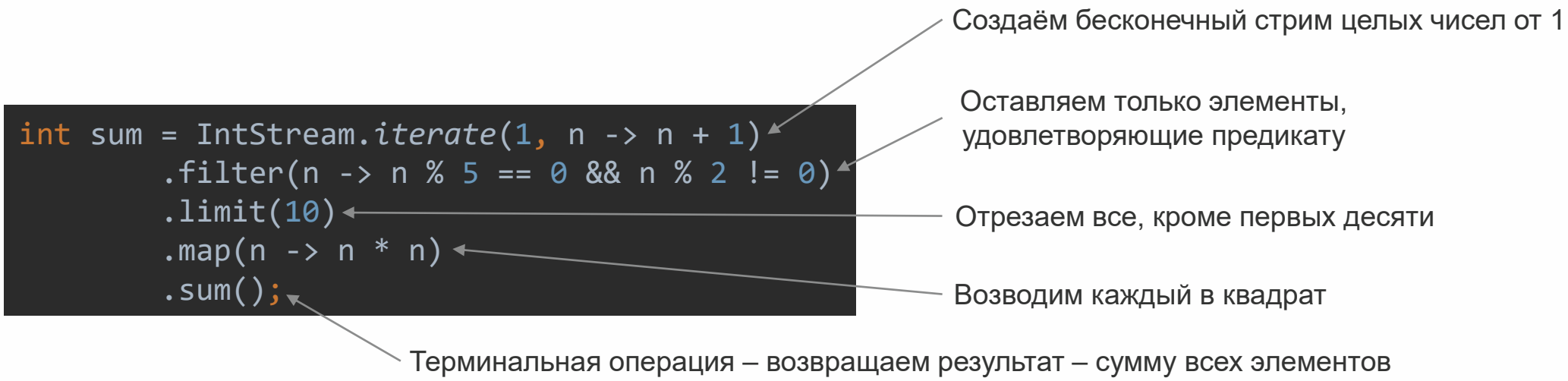
```
double[] array = /* ... */;
DoubleStream streamFromArray = Arrays.stream(array);
```

Создание стрима перечислением элементов:

```
IntStream streamOfElements = IntStream.of(1, 1, 2, 3, 5, 8);
```

Пример использования

Рассмотрим пример жизнного цикла стрима



Важно, что абсолютно никакие действия не выполняются до тех пор, пока не вызовется терминальная операция

Закрытие стрима

Объект стрима также обладает методом `close()`, который закрывает ресурс, над которым брался стрим

Для многих источников типа коллекций или генерирующих функций закрытие не требуется

Но для некоторых видов стрмиов это необходимо, иначе произойдёт утечка ресурсов:

```
BufferedReader reader = /* ... */;
Stream<String> stream2 = reader.lines();

// use stream and get result...
stream2.close();
```

```
Path path = /* ... */;
Stream<Path> stream3 = Files.list(path);
Stream<Path> stream4 = Files.walk(path);
// их тоже нужно закрывать
```

К счастью, интерфейс `Stream` наследует интерфейс `AutoClosable`,
а значит работает в конструкции `try-with-resources`:

```
BufferedReader reader = /* ... */ new BufferedReader(new StringReader(""));
try (Stream<String> stream2 = reader.lines()) {
    // use stream and get result
}
```

Промежуточные операции

Посмотрим, какие бывают промежуточные операции

```
Stream<T> filter(Predicate<? super T> pred);
```

`filter()` фильтрует стрим: через него проходят только значения, удовлетворяющие предикату

```
Stream<T> limit(long count);
```

`limit()` ограничивает размер стрима: проходят только первые `count` значений

```
Stream<T> skip(long count);
```

`skip()` пропускает первые `count` значений с начала стрима: проходят только следующие за ними

Промежуточные операции

```
<R> Stream<R> map(Function<? super T, ? extends R> func);
```

map() применяет к каждому элементу стрима функцию f :

$$x_1, x_2, x_3, \ldots \rightarrow f(x_1), f(x_2), f(x_3), \ldots$$

При этом тип элементов в стриме может измениться

Для изменения типа элементов на примитивный существуют версии:

```
IntStream mapToInt(ToIntFunction<? super T> var1);
LongStream mapToLong(ToLongFunction<? super T> var1);
DoubleStream mapToDouble(ToDoubleFunction<? super T> var1);
```

У стримов из примитивных элементов наоборот – есть версия для превращения в `Object`
К примеру, посмотрим на методы `IntStream`:

```
IntStream map(IntUnaryOperator var1);
<U> Stream<U> mapToObj(IntFunction<? extends U> var1);
LongStream mapToLong(IntToLongFunction var1);
DoubleStream mapToDouble(IntToDoubleFunction var1);
```

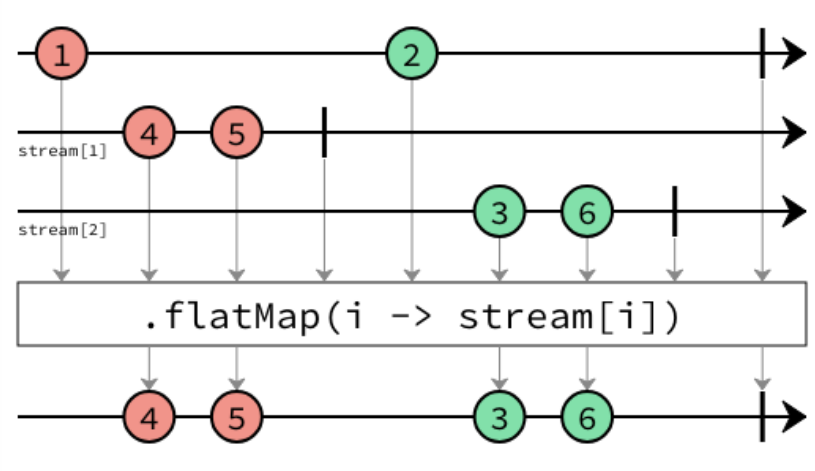
Промежуточные операции

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> func);
```

`flatMap()` похож на обычный `map()`, но функция должна возвращать целые стримы
В результате получается не стрим стримов, а их конкатенация (как будто все сшили в один)

Слово `flat` по сути означает *сплющить*, то есть раскрыть вложенные стримы наружу

Таким образом можно каждый элемент превратить в несколько (или ни во сколько)



Сплющивание выглядит так:

```
[[a, b, c], [d, e, f], [g, h, i]]
  -> [a, b, c, d, e, f, g, h, i]
```

Есть вариации, возращающие примитивные стримы:

```
IntStream flatMapToInt(Function<? super T, ? extends IntStream> var1);
LongStream flatMapToLong(Function<? super T, ? extends LongStream> var1);
DoubleStream flatMapToDouble(Function<? super T, ? extends DoubleStream> var1);
```

Промежуточные операции

```
Stream<T> distinct();
```

`distinct()` убирает из стрима дубликаты (в плане `equals`)

```
Stream<T> sorted();
Stream<T> sorted(Comparator<? super T> comp);
```

`sorted()` сортирует элементы стрима в порядке возрастания
Если элементы не `Comparable` или нужен другой порядок, то можно передать свой компаратор

```
Stream<T> peek(Consumer<? super T> var1);
```

Метод `peek()` особенный – он позволяет подсмотреть состояние стрима до вызова терминальной операции:

```
int sum = IntStream.iterate(1, n -> n + 1)
        .filter(n -> n % 5 == 0 && n % 2 != 0)
        .limit(10)
        .peek(x -> System.out.print(x + " "))
        .map(n -> n * n)
        .sum();
```

```
5 15 25 35 45 55 65 75 85 95
```

Терминальные операции

Терминальные операции запускают выполнение всей цепочки стрима
и позволяют наконец-то получить из него какой-то результат

После терминальной операции, объект стрима становится непригодным для дальнейшего использования

```
void forEach(Consumer<? super T> var1);
```

`forEach()` просто применяет к каждому элементу некоторую функцию:

```
IntStream stream = /* ... */;
stream.forEach(System.out::println);
```

```
Optional<T> findFirst();
```

`findFirst()` возвращает первый элемент стрима, если стрим непустой

```
OptionalInt result = stream.findFirst();
```

Если стрим пустой, то вернётся пустой `Optional`

Также есть и `findAny()`, возвращающий произвольный элемент, как удобно стриму:

```
Optional<T> findAny();
```

Терминальные операции

```
boolean allMatch(Predicate<? super T> pred);
```

`allMatch()` возвращает true тогда и только тогда, когда каждый элемент удовлетворяет предикату:

```
Stream<String> stream = /* ... */;
boolean allStringsAreAtLeast10Chars =
        stream.allMatch(s -> s.length() >= 10);
```

Есть ещё две версии:

```
boolean anyMatch(Predicate<? super T> pred);
boolean noneMatch(Predicate<? super T> pred);
```

Они проверяют, что хотя бы один удовлетворяет и что ни один не удовлетворяет предикату, соответственно

Терминальные операции

```
Optional<T> min(Comparator<? super T> var1);
Optional<T> max(Comparator<? super T> var1);
```

`min()` и `max()` возвращают минимальное и максимальное значение в стриме, соответственно

Например, можно получить самую короткую строку в стриме:

```
Stream<String> stream = /* ... */;
Optional<String> minString =
        stream.min(Comparator.comparingInt(String::length));
```

```
long count();
```

`count()` просто возвращает количесвто элементов, оставшихся в стриме после всех преобразований

`IntStream ->`

```
int sum();
```

У каждого примитивного стрима есть метод `sum()`, возвращающий сумму всех его элементов

```
DoubleStream stream = /* ... */;
double sum = stream.sum();
```

Терминальные операции

```
T reduce(T var1, BinaryOperator<T> var2);
Optional<T> reduce(BinaryOperator<T> var1);
```

`reduce()` позволяет посчитать свёртку элементов:
Оператор будет применяться попарно ко всем элементам, сворачивая последовательность в один элемент

Результат эквивалентен такому коду:

```
T result = identity;
for (T element : this_stream)
    result = accumulator.apply(result, element);
return result;
```

Порядок вызова не обязательно такой, поэтому оператор обязан быть ассоциативным

Можно посчитать сумму `BigInt`'ов:

```
Stream<BigInteger> bigInts = /* ... */;
BigInteger sum = bigInts.reduce(
        BigInteger.ZERO, BigInteger::add);
```

```
Object[] toArray();
<A> A[] toArray(IntFunction<A[]> generator);
```

`toArray()` преобразует стрим в массив, при этом нужно передать функцию, которая массив создаёт:

```
Person men = people.stream()
        .filter(p -> p.getGender() == MALE)
        .toArray(Person[]::new);
```

Терминальные операции

И последняя терминальная операция на сегодня: `collect()`

```
<R, A> R collect(Collector<? super T, A, R> var1);
```

`collect()` собирает элементы стрима в некоторое новое хранилище при помощи `Collector`'а

Например, можно конвертировать стрим в список:

```
Stream<String> stream = /* ... */;
List<String> list = stream.collect(Collectors.toList());
```

По сути, `collect()` производит свёртку, подобно `reduce`'у

Но как устроен `Collector`?

Интерфейс Collector

```
<R, A> R collect(Collector<? super T, A, R> var1);
```

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();

    Set<Collector.Characteristics> characteristics();
}
```

T – исходный тип данных (в стриме)
A – скрытый промежуточный тип используемый для свёртки
R – результирующий тип всего процесса свёртки

Коллектор задаётся четырьмя функциями, которые вместе аккумулируют элементы
в некотором изменяемом объекте типа A, с опциональной возможностью преобразовать финальный объект:

- создание объекта-аккумулятора (`supplier()`)
- добавление элемента к объекту-аккумулятору (`accumulator()`)
- объединение двух объектов-аккумуляторов в один (`combiner()`)
- финальное преобразование из типа A в результат типа R (`finisher()`)

Для нормальной работы коллектора требуется выполнение пары свойств:

- объединение объекта-аккумулятора `first` со свежесозданным `second` даёт результат, эквивалентный `first`
- ассоциативность: результат не изменяется при разбиении стрима на части,
которые аккумулируются по отдельности, а потом комбайнятся воедино

Интерфейс Collector

Пусть, хотим заколлектить для элемента `t1` и `t2`

```
A a1 = supplier.get();
accumulator.accept(a1, t1);
accumulator.accept(a1, t2);
R r1 = finisher.apply(a1);  // результат без разбиения
```

```
A a2 = supplier.get();
accumulator.accept(a2, t1);
A a3 = supplier.get();
accumulator.accept(a3, t2);
R r2 = finisher.apply(combiner.apply(a2, a3));  // результат с разбиением
```

Зачем вообще такая сложная структура, с поддержкой разбиваемости?

Ответ – возможность параллельных многопоточных вычислений
Каждую отдельную часть разбиения можно аккумулировать параллельно с остальными,
что может позволить ускорить процесс свёртки в разы

Ещё нужно реализовать метод `characteristics()`, который должен возвращать `Set` свойств коллектора
Например:

- `Collector.Characteristics.IDENTITY_FINISH` – у коллектора тривиальный `finisher()`, который можно не вызывать; при этом тип `A` должен быть преобразуем к `R`
- `Collector.Characteristics.UNORDERED` – коллектору плевать на порядок, элементы разрешается аккумулировать и комбайнить в любом порядке

Создание Collector’a

Коллектор можно удобно создать методом `Collector.of()`:

```
static <T, R> Collector<T, R, R> of(
    Supplier<R> supplier,
    BiConsumer<R, T> accumulator,
    BinaryOperator<R> combiner ,
    Function<A, R> finisher,
    Collector.Characteristics... characteristics)
```

При этом есть версия без `finisher`’а, которая использует тривиальный

Пример: `Collectors.toList()`

```
public static <T> Collector<T, ?, List<T>> toList() {
    return Collectors.of(ArrayList::new, List::add, (left, right) -> {
        left.addAll(right);
        return left;
    });
}
```

Создание Collector’a

Пример: `Collectors.joining()` – склеивает строки в одну

Тут используется finisher для превращения `StringBuilder` в `String`

```
public static Collector<CharSequence, ?, String> joining() {
    return Collectors.of(StringBuilder::new, StringBuilder::append, (r1, r2) -> {
        r1.append(r2);
        return r1;
    }, StringBuilder::toString);
}
```

Пример: `Collectors.summingInt()`

mapper преобразует объекты в инты, все инты суммируются, как при `reduce`

```
public static <T> Collector<T, ?, Integer> summingInt(ToIntFunction<? super T> mapper) {
    return Collectors.of(() -> {
        return new int[1];
    }, (a, t) -> {
        a[0] += mapper.applyAsInt(t);
    }, (a, b) -> {
        a[0] += b[0];
        return a;
    }, (a) -> {
        return a[0];
    });
}
```

Чтобы использовать `int` как дженерик-тип, его обернули в массив

Если вам чего-то не хватает, не забываем поискать в документации: `Stream`, `Collectors`

Пример стримов: факториал

Посчитаем $n!$ с помощью стримов

```
public static BigInteger factorial(int n) {
    return IntStream.rangeClosed(1, n)
        .mapToObj(BigInteger::valueOf)
        .reduce(BigInteger.ONE, BigInteger::multiply);
}
```

Пример стримов: палиндромность текста

Проверим, является ли текст палиндромом, используя стрим для фильрации букв и цифр, и приведения к нижнему регистру

Не забываем, что `chars()` возвращает `IntStream`

```
public static boolean isPalindrome(String s) {
    StringBuilder leftToRight = new StringBuilder();

    s.chars().filter(Character::isLetterOrDigit)
        .map(Character::toLowerCase)
        .forEach(leftToRight::appendCodePoint);

    StringBuilder rightToLeft = new StringBuilder(leftToRight).reverse();

    return leftToRight.toString().equals(rightToLeft.toString());
}
```