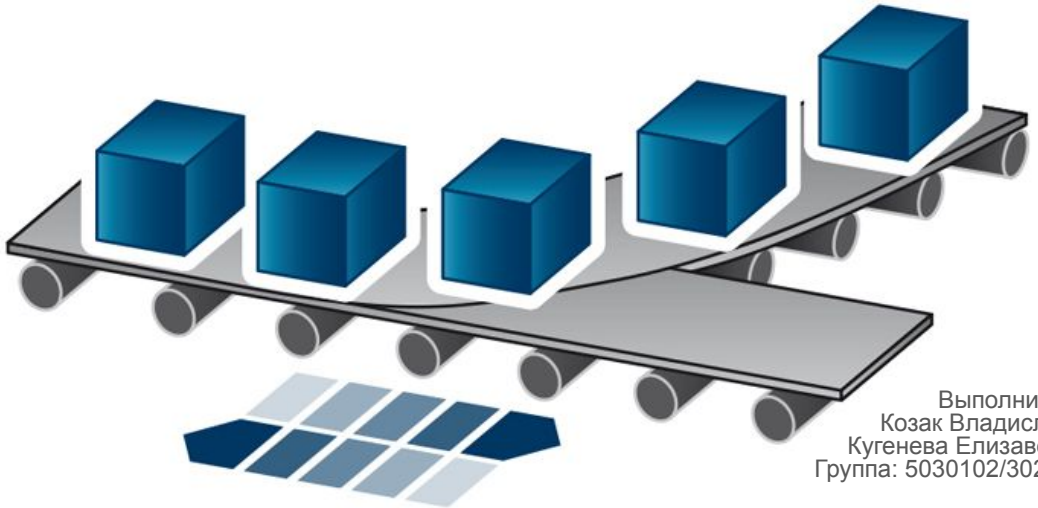
Stream API в Java:

Функциональный подход к обработке данных



Выполнили:
Козак Владислав
Кугенева Елизавета
Группа: 5030102/30202

Что такое Stream API?

Stream API — это инструмент для работы с коллекциями данных в Java, введенный в восьмой версии Java. Он позволяет выполнять различные операции над данными, такие как фильтрация, маппинг, сортировка и агрегация, используя функциональный подход с лямбда-выражениями.

С потоком:

```
IntStream.of(50,60,70,80,90,100,110,120)
        .filter(x -> x < 90)
        .map(x -> x + 10)
        .limit(3)
        .forEach(System.out::print);
```

Без потока:

```
int[] arr = {50,60,70,80,90,100,110,120};
int count = 0;
for (int x : arr){
    if (x >= 90) continue;
    x += 10;
    count++;
    if (count > 3) break;
    System.out.print(x);
```

Недостатки императивного подхода

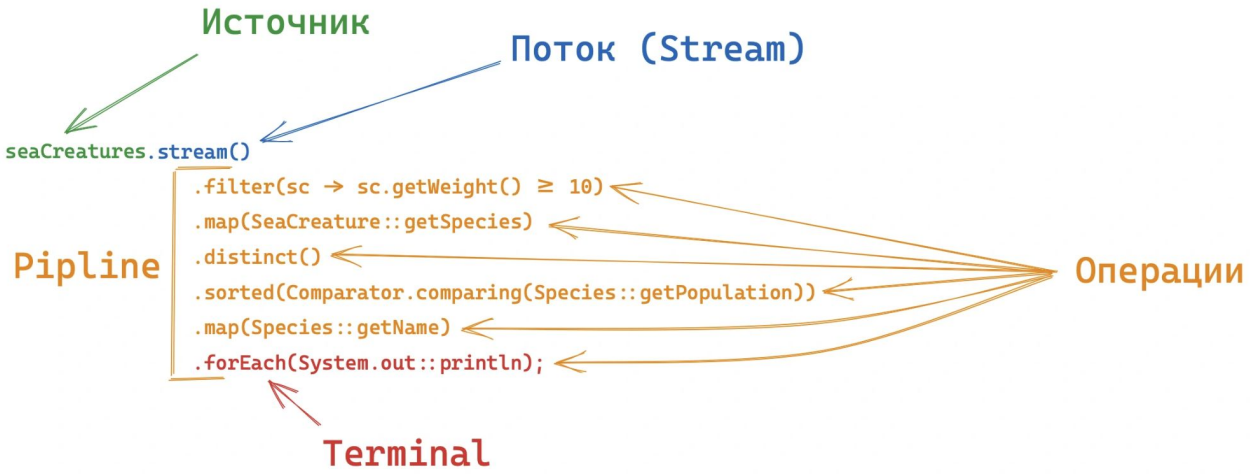
- Избыточность и низкая выразительность кода
- Сложность композиции
- Побочные эффекты и изменяемое состояние
- Отсутствие ленивых вычислений
- Сложность рефакторинга и модификации

Преимущества Stream API

- Декларативный стиль
- Выразительность кода
- Ленивые вычисления
- Легкость рефакторинга
- Естественная композиция операций
- Изоляция и неизменяемость

Основные компоненты Stream API

- Источники данных: колекции, массивы, файлы, строки, пустой поток…
- Промежуточные операции: фильтрация, преобразование, сортировка, отладка.
- Терминальные операции: сокращение, сбор, поиск(`findFirst()`, `findAny()`), проверка (`anyMatch()`, `noneMatch()` - `boolean`), итерация(`forEach()`, `forEachOrder()` - `void`)



Промежуточные операции (Intermediate)

Фильтрация:

- filter(Predicate<T>) - фильтрация элементов
- distinct() - удаление дубликатов
- limit(n) - ограничение количества
- skip(n) - пропуск первых n элементов

Сбор:

- collect(n) - преобразование в коллекцию

Преобразование:

- map(Function<T,R>) - преобразование каждого элемента
- flatMap(Function<T,Stream<R>>) - "разворачивание" вложенных структур

Сортировка:

- sorted() - естественная сортировка
- sorted(Comparator<T>) - сортировка с компаратором

Терминальные операции (Terminal)

Сокращение:

- `count()` - количество элементов
- `min()`, `max()` - минимальный/максимальный элемент
- `reduce()` - сворачивание элементов

Проверка:

- `anyMatch()`, `allMatch()`, `noneMatch()` - проверка условий
- `findFirst()`, `findAny()` - поиск элементов

Итерация:

- `forEach()` - выполнение действия для каждого элемента

Пример 1. Сравнение (1|3)

Условие задачи: имеется список студентов со следующей информацией: имя, средний балл, группа.

Требуется выполнить следующее:

- Для каждой группы найти топ-2 студента с наивысшим средним баллом
- Имена выбранных студентов преобразовать в верхний регистр
- Полученные имена отсортировать по алфавиту
- Результат представить в виде отображения "группа → список имен"

```
static class Student {
    String name;
    double grade;
    String group;

    Student(String name, double
grade, String group) {
        this.name = name;
        this.grade = grade;
        this.group = group;
    }
}
```

Пример 1. Сравнение (2|3)

```
Map<String, List<Student>> grouped = new HashMap<>();
   for (Student s : students) {
       grouped.computeIfAbsent(s.group, k -> new ArrayList<>()).add(s);
   }

   Map<String, List<String>> result2 = new HashMap<>();
   for (Map.Entry<String, List<Student>> entry : grouped.entrySet()) {
       List<Student> groupStudents = entry.getValue();
       groupStudents.sort((s1, s2) -> Double.compare(s2.grade, s1.grade));

       List<String> topNames = new ArrayList<>();
       for (int i = 0; i < Math.min(2, groupStudents.size()); i++) {
           topNames.add(groupStudents.get(i).name.toUpperCase());
       }

       Collections.sort(topNames);
       result2.put(entry.getKey(), topNames);
   }

   result2.forEach((group, names) ->
           System.out.println("Группа " + group + ": " + names));
}
```

Пример 1. Сравнение (3|3)

```
Map<String, List<String>> result = students.stream()
        .sorted((s1, s2) -> Double.compare(s2.grade, s1.grade))
        .collect(Collectors.groupingBy(
                s -> s.group,
                Collectors.collectingAndThen(
                        Collectors.toList(),
                        list -> list.stream()
                                .limit(2)
                                .map(s -> s.name.toUpperCase())
                                .sorted()
                                .collect(Collectors.toList())
                )
        ));

result.forEach((group, names) ->
        System.out.println("Группа " + group + ": " + names));
```

Параллельные потоки

Параллельные потоки — это расширение Stream API, которое автоматически распределяет обработку элементов потока по нескольким потокам выполнения.

```
// Просто заменить stream() на parallelStream()
long count = largeList.parallelStream()
    .filter(item -> item.isValid())
    .count();

// Или явно вызвать parallel()
long count = largeList.stream()
    .parallel()
    .filter(item -> item.isValid())
    .count();
```

Когда использовать:

- Большие объемы данных
- "Тяжелые" операции
- Нет зависимости между элементами

Ограничения:

- Stateful операции (`sorted()`, `distinct()`)
- Порядок обработки не гарантируется

Best Practices

✅ Делать:

- Использовать ссылки на методы (`String::toUpperCase`)
- Разбивать длинные цепочки на логические блоки
- Использовать специализированные стримы для примитивов (`IntStream`, `LongStream`)
- Фильтровать в начале цепочки

❌ Не делать:

- Использовать стримы для простых циклов
- Изменять внешние переменные внутри операций

Ограничения

- Одноразовость - нельзя повторно использовать
- Нет доступа к индексу элемента
- Сложная отладка (стектрейсы)
- Производительность - для маленьких коллекций обычный цикл может быть быстрее

Заключение

- Stream API позволяет писать чистый, декларативный и легко читаемый код.
- Stream API повышает эффективность для работы с большими наборами данных за счет ленивых вычислений и естественной композиции.
- Однако Stream API не всегда является оптимальным выбором. Для простых задач или небольших коллекций императивный подход может быть быстрее и понятнее.
- Правильное использование Stream API требует понимания его возможностей и ограничений, включая одноразовость потоков и сложность отладки.