Stream API

Stream API — это **новый способ** работать со **структурами данных в функциональном стиле**. Его **задача** - **упростить работу** с наборами данных, в частности, упростить операции **фильтрации**, **сортировки** и другие манипуляции с данными.

Основные характеристики Stream API:

1. **Декларативный стиль**: Позволяет описывать, что нужно сделать с данными, а не как это сделать. Это делает код более читаемым и понятным.
2. **Ленивая оценка**: Операции над потоками выполняются только тогда, когда это необходимо. Это позволяет оптимизировать производительность, так как не все данные обрабатываются сразу.
3. **Параллельная обработка**: Stream API поддерживает параллельную обработку данных, что позволяет использовать многоядерные процессоры для ускорения выполнения операций.
4. **Поддержка различных источников данных**: Потоки могут быть созданы из различных источников, таких как коллекции, массивы, файлы и даже генераторы.

Промежуточные и терминальные операции:

Создание потока - Поток можно создать из коллекции, массива или других источников:

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

Промежуточные операции: Эти операции возвращают новый поток и могут быть объединены:

1. **filter**: Фильтрует элементы потока, оставляя только те, которые соответствуют заданному предикату.
   stream.filter(element -> element > 10);
2. **map**: Преобразует элементы потока, применяя функцию к каждому элементу:
   stream.map(String::toUpperCase);

3. **flatMap**: Преобразует элементы потока в потоки и объединяет их в один поток.
   stream.flatMap(element -> Stream.of(element.split(",")));
4. **distinct**: Удаляет дубликаты из потока.
   stream.distinct();
5. **sorted**: Сортирует элементы потока. Можно использовать как по умолчанию, так и с компаратором.
   stream.sorted();
   stream.sorted(Comparator.reverseOrder());
6. **peek**: Позволяет выполнять действие для каждого элемента потока, не изменяя сам поток. Обычно используется для отладки.
   stream.peek(element -> System.out.println(element));
7. **limit**: Ограничивает количество элементов в потоке.
   stream.limit(5);
8. **skip**: Пропускает заданное количество элементов в потоке.
   stream.skip(3);

Терминальные операции — это операции, которые инициируют процесс обработки данных и возвращают результат, завершая работу потока. После вызова терминальной операции поток больше не может быть использован.

1. **forEach**: Применяет заданное действие к каждому элементу потока.
   stream.forEach(element -> System.out.println(element));
2. **collect**: Преобразует элементы потока в коллекцию или другую структуру данных. Это одна из самых часто используемых терминальных операций.
   List list = stream.collect(Collectors.toList());
   Set set = stream.collect(Collectors.toSet());
   Map<Integer, String> map = stream.collect(Collectors.toMap(String::length, Function.identity()));
3. **reduce** Выполняет редукцию элементов потока, используя бинарную операцию. Позволяет **агрегировать** значения.
   Optional sum = stream.reduce((a, b) -> a + b);
4. **count**: Возвращает количество элементов в потоке.
   long count = stream.count();
5. **anyMatch**: Проверяет, соответствует ли хотя бы один элемент потока заданному предикату.
   boolean hasEven = stream.anyMatch(n -> n % 2 == 0);
6. **allMatch**: Проверяет, соответствуют ли все элементы потока заданному предикату.
   boolean allPositive = stream.allMatch(n -> n > 0);
7. **noneMatch**: Проверяет, не соответствует ли ни один элемент потока заданному предикату.
   boolean noneNegative = stream.noneMatch(n -> n < 0);

8. **findFirst**: Возвращает первый элемент потока, если он существует.
   Optional first = stream.findFirst();
9. **findAny**: Возвращает любой элемент потока, если он существует. Может быть полезно в параллельных потоках.
   Optional any = stream.findAny();

3. **max**: Возвращает максимальный элемент потока по заданному компаратору.
   Optional max = stream.max(Comparator.naturalOrder());
4. **min**: Возвращает минимальный элемент потока по заданному компаратору.
   Optional min = stream.min(Comparator.naturalOrder());
5. **toArray**: Преобразует элементы потока в массив.
   String[] array = stream.toArray(String[]::new);

Эти терминальные операции позволяют завершить обработку данных в потоке и получить результаты в различных форматах, таких как коллекции, массивы или простые значения. Пример использования терминальных операций.