ЛАБОРАТОРНА РОБОТА 6

Тема: Використання Stream API

Задача програмного інтерфейсу Stream API є спростити роботу з наборами даних(операції фільтрації, сортування та ін). Основна функціональність даного API забезпечено пакетом java.util.stream. Ключовим поняттям у Stream API є потік даних - канал передавання даних із джерела даних, у якості якого можуть виступати як файли, так і масиви або колекції. Нижче наведено простий приклад, в якому в масиві цілих чисел підраховується кількість чисел, що більше 0:

```
	import java.util.stream.IntStream;

public class Example {

	public static void main(String[] args) {
		long count = IntStream.of(-5,-4,-3,-2,-1,0,1,2,3,4,5).filter(w-> w > 0).count();
		System.out.println(count);
	}
}
```

Зазвичай таку задачу можна вирішити через застосування циклу і перевірки умови задачі, у разі виконання якої певна змінна збільшує своє значення. З використанням Stream API формалізується потік даних, до якого застосовується метод filter() з аргументом умови задачі у вигляді лямбда виразу.

Усі операції з потоками бувають або термінальними (terminal), або проміжними (intermediate). Проміжні операції повертають трансформований потік. У наведеному прикладі метод filter приймає потік чисел та повертає перетворений потік, в якому тільки числа, що більше 0. До потоку, що повертається також можна застосувати декілька проміжних операцій.

Кінцеві або термінальні операції повертають конкретний результат. У наведеному вище прикладі метод count() виконує термінальну операцію і повертає число. Після термінальної операції проміжні операції застосовувати не можна. Усі потоки виконують обчислення тільки тоді, коли до них застосовується термінальна операція. Це стосується і проміжних операцій. Таким чином застосовується відкладене виконання.

В основі Stream API лежить інтерфейс BaseStream. Його повне визначення:

interface BaseStream<T , S extends BaseStream<T , S>>

Параметр T означає тип даних у потоці, а S - тип потоку, який наслідується від інтерфейсу BaseStream. BaseStream визначає базовий функціонал для роботи з потоками, які реалізуються через його методи:

void close() : закриває потік
boolean isParallel(): повертає true, якщо потік є паралельним
Iterator<T> iterator(): повертає посилання на ітератор потоку
Spliterator<T> spliterator(): повертає посилання на сплітератор потоку
S parallel(): повертає паралельний потік (паралельні потоки можуть використовувати декілька ядер процесора у багатоядерних архітектурах)
S sequential(): повертає послідовний потік
S unordered(): повертає неупорядкований потік

Від інтерфейсу BaseStream наслідується ряд інтерфейсів, які призначено для створення конкретних потоків:

Stream<T>: використовується для потоків даних, що представляють тип посилання

IntStream: використовується для потоків з типом даних int
DoubleStream: використовується для потоків з типом даних double
LongStream: використовується для потоків з типом даних long

При роботі з потоками, які представляють певний примітивний тип - double, int, long слід використовувати саме інтерфейси DoubleStream, IntStream, LongStream. При роботі з більш складними даними використовується інтерфейс Stream<T>.

Методами цього інтерфейсу є:

boolean allMatch(Predicate<? super T> predicate): повертає true, якщо усі елементи потоку задовольняють умові предикату predicate. Термінальна операція.

boolean anyMatch(Predicate<? super T> predicate): повертає true, якщо хоча б один елемент потоку задовольняє умові предикату predicate. Термінальна операція.

<R,A> R collect(Collector<? super T,A,R> collector): додає елементи до незмінного контейнеру з типом R. T представляє тип даних з потоку, що викликається, а A - тип даних у контейнері. Термінальна операція.

long count(): повертає кількість елементів у потоці. Термінальна операція.

Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b): об'єднує два потоки. Проміжна операція

Stream<T> distinct(): повертає потік, у якому є тільки унікальні дані з типом T. Проміжна операція.

Stream<T> dropWhile(Predicate<? super T> predicate): пропускає елементи, які відповідають умові в predicate, до тих пір, поки не зустрічається елемент, що не відповідає умові. Вибрані елементи повертаються у вигляді потоку. Проміжна операція.

Stream<T> filter(Predicate<? super T> predicate): фільтрує елементи у відповідності до умови в predicate. Проміжна операція.

Optional<T> findFirst(): повертає перший елемент із потоку. Термінальна операція.

Optional<T> findAny(): повертає якийсь елемент із потоку. Термінальна операція.

void forEach(Consumer<? super T> action): для кожного елементу виконується дія action. Термінальна операція.

Stream<T> limit(long maxSize): залишає у потоці тільки maxSize елементів. Проміжна операція.

Optional<T> max(Comparator<? super T> comparator): повертає максимальний елемент із потоку. Для порівняння елементів застосовується компаратор comparator. Термінальна операція.

Optional<T> min(Comparator<? super T> comparator): повертає мінімальний елемент із потоку. Для порівняння елементів застосовується компаратор comparator. Термінальна операція.

<R> Stream<R> map(Function<? super T,? extends R> mapper): перетворює елементи типу T в елементи типу R і повертає потік з елементами R. Проміжна операція.

<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper): дозволяє перетворювати елемент типу T у декілька елементів типу R і повертає потік з елементами R. Проміжна операція.

boolean noneMatch(Predicate<? super T> predicate): повертає true, якщо ні один із елементів у потоці не задовольняє умові предикату. Термінальна операція.

Stream<T> skip(long n): повертає потік, в якому відсутні перші n елементів. Проміжна операція.

Stream<T> sorted(): повертає відсортований потік. Проміжна операція.

Stream<T> sorted(Comparator<? super T> comparator): повертає відсортований потік, відповідно компаратору. Проміжна операція.

Stream<T> takeWhile(Predicate<? super T> predicate): обирає з потоку елементи, поки вони відповідають умові predicate. Вибрані елементи повертаються у вигляді потоку. Проміжна операція.

Object[] toArray(): повертає масив із елементів потоку. Термінальна операція.

Усі ці операції дозволяють взаємодіяти з потоком як з деяким набором даних подібно до колекцій, але є наступні відмінності:

- Потоки не зберігають елементи. Елементи, що використовуються у потоках, можуть зберігатись в колекції, або при необхідності можуть бути прямо згенеровані.
- Операції з потоками не змінюють джерела даних. Операції з потоками тільки повертають новий потік з результатами цих операцій.
- Для потоків характерно відкладене виконання — виконання усіх операцій з потоком відбувається тільки тоді, коли виконується термінальна операція і повертається конкретний результат, а не новий потік.

Для створення потоку даних можна застосовувати різні методи. В якості джерела потоку можна використовувати колекції. В інтерфейсі Collection є такі методи для роботи з потоками:

default Stream<E> stream: повертає потік даних із колекції

default Stream<E> parallelStream: повертає паралельний потік даних із колекції

Приклад з використанням ArrayList

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.stream.IntStream;

public class JavaApplication2 {

    public static void main(String[] args) {
    ArrayList<String> cities = new ArrayList<String>();
    Collections.addAll(cities, "Париж", "Лондон", "Мадрид");
    cities.stream().filter(s->s.length()==6).forEach(s->System.out.println(s));
  }
}
```

Результат:

```
Лондон
Мадрид
```

За допомогою виклику cities.stream() отримуємо потік, який використовує дані зі списку cities. За допомогою кожної проміжної операції, що застосовується до потоку, також можна отримати потік з врахуванням модифікацій. Наприклад, можна змінити попередній приклад наступним чином:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.stream.IntStream;
import java.util.stream.Stream;
```

```
public class JavaApplication2 {

    public static void main(String[] args) {

    ArrayList<String> cities = new ArrayList<String>();
    Collections.addAll(cities, "Париж", "Лондон", "Мадрид");
    Stream<String> citiesStream = cities.stream(); // отримуємо потік
    citiesStream = citiesStream.filter(s->s.length()==6); // застосовуємо фільтрацію
    citiesStream.forEach(s->System.out.println(s)); // виводимо результат
    }
}
```

Після використання термінальних операцій інші термінальні або проміжні операції до цього потоку вже не можуть застосовуватись тому, що потік вже використано. Наприклад, у наступному прикладі буде помилка:

```
citiesStream.forEach(s->System.out.println(s));
long number = citiesStream.count(); // тут буде помилка - потік вже використано
System.out.println(number);
citiesStream = citiesStream.filter(s->s.length()>5); // та сама помилка
```

Фактично життєвий цикл потоку проходить наступні три стадії:

- Створення потоку
- Застосування до потоку декількох проміжних операцій
- Застосування до потоку термінальної операції і отримання результату

Існують і інші способи для створення потоків даних. Один з таких способів представляє метод Arrays.stream(T[] array), який створює потік даних із масиву:

```
Stream<String> citiesStream2 = Arrays.stream(new String[]{"Париж", "Лондон", "Мадрид"}) ;
citiesStream2.forEach(s->System.out.println(s)); // виведення елементів масиву
```

Для створення потоків IntStream, DoubleStream, LongStream можна використовувати відповідні перевантажені версії цього методу:

```
IntStream intStream = Arrays.stream(new int[]{1,2,4,5,7});
intStream.forEach(i->System.out.println(i));

LongStream longStream = Arrays.stream(new long[]{100,250,400,5843787,237});
longStream.forEach(l->System.out.println(l));

DoubleStream doubleStream = Arrays.stream(new double[] {3.4, 6.7, 9.5, 8.2345, 121});
doubleStream.forEach(d->System.out.println(d));
```

Ще одним способом створення потоку є статичний метод of(T..values) класу Stream:

```
Stream<String> citiesStream =Stream.of("Париж", "Лондон", "Мадрид");
citiesStream.forEach(s->System.out.println(s));
```

Також можна передати масив:

```
String[] cities = {"Париж", "Лондон", "Мадрид"};
Stream<String> citiesStream2 =Stream.of(cities);

IntStream intStream = IntStream.of(1,2,4,5,7);
intStream.forEach(i->System.out.println(i));

LongStream longStream = LongStream.of(100,250,400,5843787,237);
longStream.forEach(l->System.out.println(l));

DoubleStream doubleStream = DoubleStream.of(3.4, 6.7, 9.5, 8.2345, 121);
doubleStream.forEach(d->System.out.println(d));
```

Для перебору елементів потоку застосовується метод forEach(), який представляє термінальну операцію. В якості параметру він приймає об’єкт Consumer<? super String>, який представляє дію, що виконується для кожного елементу набору.

Наприклад:

```
Stream<String> citiesStream = Stream.of("Париж", "Лондон", "Мадрид","Берлин", "Брюссель");
citiesStream.forEach(s->System.out.println(s));
```

Фактично це буде аналогічно перебору усіх елементів у циклі for і виконанню з ними дії, а саме виведення на консоль.

Для фільтрації елементів у потоці застосовується метод filter(), який представляє проміжну операцію. Він приймає в якості параметру деяку умову у вигляді об’єкту Predicate<T> і повертає новий потік із елементів, які задовольняють цій умові:

```
Stream<String> citiesStream = Stream.of("Париж", "Лондон", "Мадрид","Берлин", "Брюссель");
citiesStream.filter(s->s.length()==6).forEach(s->System.out.println(s));
```

Умова `s.length()==6` повертає true для тих елементів, довжина яких дорівнює 6 символів.

Інший приклад фільтрації. Створено наступний клас Phone:

```
class Phone{

    private String name;
    private int price;

    public Phone(String name, int price){
        this.name=name;
        this.price=price;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getPrice() {
        return price;
```

```
    }

    public void setPrice(int price) {
        this.price = price;
    }
}
```

Фільтруємо набір телефонів за ціною:

```
    Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", 5400), new
Phone("Lumia 950", 4500), new Phone("Samsung Galaxy S 6", 4000));

phoneStream.filter(p->p.getPrice()<5000).forEach(p->
                                            System.out.println(p.getName()));
```

Відображення або маппінг дозволяє задати функцію перетворення одного об’єкту в інший, тобто отримати із елементу одного типу елемент іншого типу. Для відображення використовується метод `map`, який має наступне визначення:

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

Функція, що передається у метод map, задає перетворення від об’єктів типу T до типу R. І у результаті повертається новий потік з перетвореними об’єктами. Для прикладу використаємо вище визначений клас телефонів і виконаємо перетворення від типу Phone до типу String:

```
    Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", 5400), new
Phone("Lumia 950", 4500), new Phone("Samsung Galaxy S 6", 4000));
phoneStream.map(p-> p.getName()).forEach(s->System.out.println(s));
```

Операція `map(p-> p.getName())` передає до нового потоку тільки назви телефонів, тому на консолі будуть тільки їх назви. Перетворення може мати і інший вигляд:

```
    phoneStream.map(p-> "назва: " + p.getName()
            + " ціна: " + p.getPrice()).forEach(s->System.out.println(s));
```

У цьому випадку потік результату має у рядках назви та ціни. Для перетворення об’єктів у типи Integer, Long, Double визначено спеціальні методи mapToInt(), mapToLong() та mapToDouble().

Плоске відображення виконується при необхідності отримати із одного елемента декілька. Таку операцію виконує метод flatMap:

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

Наприклад, як для прикладу вище виводиться назва телефону та його ціна, але необхідно встановити для кожного телефону акційні ціни та звичайні ціни. Таким чином із одного об’єкту Phone необхідно отримати два об’єкти з інформацією у вигляді рядку. Для цього застосовується flatMap:

```
    Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", 5400), new
Phone("Lumia 950", 4500), new Phone("Samsung Galaxy S 6", 4000));
```

```
phoneStream
    .flatMap(p->Stream.of(
String.format("назва: %s акційна ціна: %d", p.getName(), p.getPrice()),
String.format("назва: %s ціна без акції: %d", p.getName(), p.getPrice() -
(int)(p.getPrice()*0.1))
        ))
    .forEach(s->System.out.println(s));
```

Колекції, на основі яких часто створюються потоки, вже мають спеціальні методи для сортування вмісту. Але клас Stream також має можливість сортування. Таке сортування можна задіяти, коли іде набір проміжних операцій з потоком, які створюють нові набори даних і є необхідність зробити сортування цих наборів. Для простого сортування за зростанням використовується метод sorted():

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Program {

    public static void main(String[] args) {

        List<String> phones = new ArrayList<String>();
        Collections.addAll(phones, "iPhone X", "Nokia 9", "Huawei Nexus 6P",
                "Samsung Galaxy S8", "LG G6", "Xiaomi MI6",
                "ASUS Zenfone 3", "Sony Xperia Z5", "Meizu Pro 6",
                "Pixel 2");

        phones.stream()
                .filter(p->p.length()<12)
                .sorted() // сортування за зростанням
                .forEach(s->System.out.println(s));
    }
}
```

ПРАКТИЧНЕ ЗАВДАННЯ

1. Вивчить теоретичну частину лабораторної роботи та виконайте наведені приклади.

2. Для наданого файлу:
   - розробіть клас відповідно структурі даних в ньому;
   - напишіть читання даних з файлу та створення потоку з них;
   - наведіть приклади коду реалізації фільтрації за декількома різними типами для цього потоку;
   - наведіть приклади коду реалізації плоского відображення для цього потоку;
   - наведіть приклади коду реалізації сортування за декількома різними типами для цього потоку.
3. Підготуйте звіт.