Глава 10. Stream API

Начиная с Java 8, в состав библиотеки языка входит пакет java.util.stream, который позволяет работать с потоками данных. Причем обработка данных может быть произведена либо последовательно (по умолчанию), либо в параллельных потоках, что позволяет увеличить скорость обработки. Схема работы с потоками данных выглядит следующим образом:

Источник -> Операция 1 -> ... -> Операция N -> Терминальная операция

В качестве источника данных могут выступать списки, множества и другие элементы коллекции, а также массивы, файлы и др. Над данными производятся различные промежуточные операции — например, фильтрация данных. Любая промежуточная операция возвращает поток, с которым можно выполнять другие промежуточные операции. Промежуточные операции обычно заканчиваются терминальной операцией. При выполнении терминальной операции работа с потоком прекращается. Следует учитывать, что промежуточные операции являются «ленивыми», т. е. все промежуточные операции выполняются только при выполнении терминальной операции.

10.1. Введение в Stream API

Начиная с JDK 8 в Java появился новый API - Stream API. Его задача - упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Вся основная функциональность данного API сосредоточена в пакете java.util.stream.

Ключевым понятием в Stream API является поток данных. Вообще сам термин "поток" довольно перегружен в программировании в целом и в Java в частности. В одной из предыдущих глав рассматривалась работа с символьными и байтовыми потоками при чтении-записи файлов. Применительно к Stream API поток представляет канал передачи данных из источника данных. Причем в качестве источника могут выступать как файлы, так и массивы, и коллекции.

Одной из отличительных черт Stream API является применение лямбда-выражений, которые позволяют значительно сократить запись выполняемых действий.

При ближайшем рассмотрении мы можем найти в других технологиях программирования аналоги подобного API. В частности, в языке C# некоторым аналогом Stream API будет технология LINQ.

Рассмотрим простейший пример. Допустим, у нас есть задача: найти в массиве количество всех чисел, которые больше 0. До JDK 8 мы бы могли написать что-то наподобие следующего:

```
1  int[] numbers = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
2  int count=0;
3  for(int i:numbers){ if(i > 0) count++; }
4  System.out.println(count);
```

Теперь применим Stream API:

```
1  import java.util.stream.*;
2  //.....................
3  long count = IntStream.of(-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5).filter(w -> w > 0).count();
4  System.out.println(count);
```

Теперь вместо цикла и условных конструкций, которые использовали до JDK 8, мы можем записать цепочку методов, которые будут выполнять те же действия.

При работе со Stream API важно понимать, что все операции с потоками бывают либо терминальными (terminal), либо промежуточными (intermediate). Промежуточные операции возвращают трансформированный поток. Например, выше в примере метод filter принимал поток чисел и возвращал уже преобразованный поток, в котором только числа больше 0. К возвращенному потоку также можно применить ряд промежуточных операций.

Терминальные (конечные) операции возвращают конкретный результат. Например, в примере выше метод count() представляет терминальную операцию и возвращает число. После этого никаких промежуточных операций естественно применять нельзя.

Все потоки производят вычисления, в том числе в промежуточных операциях, только тогда, когда к ним применяется терминальная операция. То есть в данном случае применяется отложенное выполнение.

В основе Stream API лежит интерфейс BaseStream. Его полное определение:

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

Здесь параметр T означает тип данных в потоке, а S - тип потока, который наследуется от интерфейса BaseStream.

BaseStream определяет базовый функционал для работы с потоками, который реализуется через его методы:

- void close(): закрывает поток
- boolean isParallel(): возвращает true, если поток является параллельным
- Iterator<T> iterator(): возвращает ссылку на итератор потока
- Spliterator<T> spliterator(): возвращает ссылку на сплитератор потока
- S parallel(): возвращает параллельный поток (параллельные потоки могут задействовать несколько ядер процессора в многоядерных архитектурах)

- S sequential(): преобразует параллельный поток в последовательный
- S unordered(): делает поток неупорядоченным. Упорядоченный поток требует дополнительных затрат ресурсов для поддержания порядка следования элементов. Если сделать поток неупорядоченным, то при использовании параллельных потоков можно на некоторых операциях получить прирост производительности

От интерфейса BaseStream наследуется ряд интерфейсов, предназначенных для создания конкретных потоков:

- Stream<T>: используется для потоков данных, представляющих любой ссылочный тип
- IntStream: используется для потоков с типом данных int
- DoubleStream: используется для потоков с типом данных double
- LongStream: используется для потоков с типом данных long

При работе с потоками, которые представляют определенный примитивный тип - double, int, long проще использовать интерфейсы DoubleStream, IntStream, LongStream. Но в большинстве случаев, как правило, работа происходит с более сложными данными, для которых предназначен интерфейс Stream<T>. Рассмотрим некоторые его методы:

- boolean allMatch(Predicate<? super T> predicate): возвращает true, если все элементы потока удовлетворяют условию в предикате. Терминальная операция.
- boolean anyMatch(Predicate<? super T> predicate): возвращает true, если хоть один элемент потока удовлетворяют условию в предикате. Терминальная операция.
- <R,A> R collect(Collector<? super T,A,R> collector): добавляет элементы в неизменяемый контейнер с типом R. T представляет тип данных из вызывающего потока, а A - тип данных в контейнере. Терминальная операция.
- long count(): возвращает количество элементов в потоке. Терминальная операция.
- Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b): объединяет два потока. Промежуточная операция.
- Stream<T> distinct(): возвращает поток, в котором имеются только уникальные данные с типом T. Промежуточная операция.
- Stream<T> dropWhile(Predicate<? super T> predicate): пропускает элементы, которые соответствуют условию в predicate, пока не попадется элемент, который не соответствует условию. Выбранные элементы возвращаются в виде потока. Промежуточная операция.
- Stream<T> filter(Predicate<? super T> predicate): фильтрует элементы в соответствии с условием в предикате. Промежуточная операция.
- Optional<T> findFirst(): возвращает первый элемент из потока. Терминальная операция.
- Optional<T> findAny(): возвращает первый попавшийся элемент из потока. Терминальная операция.
- void forEach(Consumer<? super T> action): для каждого элемента выполняется действие action. Терминальная операция.
- Stream<T> limit(long maxSize): оставляет в потоке только maxSize элементов. Промежуточная операция.
- Optional<T> max(Comparator<? super T> comparator): возвращает максимальный элемент из потока. Для сравнения элементов применяется компаратор comparator. Терминальная операция.
- Optional<T> min(Comparator<? super T> comparator): возвращает минимальный элемент из потока. Для сравнения элементов применяется компаратор comparator. Терминальная операция.
- <R> Stream<R> map(Function<? super T,? extends R> mapper): преобразует элементы типа T в элементы типа R и возвращает поток с элементами R. Промежуточная операция.
- <R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper): позволяет преобразовать элемент типа T в несколько элементов типа R и возвращает поток с элементами R. Промежуточная операция.
- boolean noneMatch(Predicate<? super T> predicate): возвращает true, если ни один из элементов в потоке не удовлетворяет условию в предикате. Терминальная операция.
- Stream<T> skip(long n): возвращает поток, в котором отсутствуют первые n элементов. Промежуточная операция.
- Stream<T> sorted(): возвращает отсортированный поток. Промежуточная операция.
- Stream<T> sorted(Comparator<? super T> comparator): возвращает отсортированный в соответствии с компаратором поток. Промежуточная операция.
- Stream<T> takeWhile(Predicate<? super T> predicate): выбирает из потока элементы, пока они соответствуют условию в predicate. Выбранные элементы возвращаются в виде потока. Промежуточная операция.
- Object[] toArray(): возвращает массив из элементов потока. Терминальная операция.

Несмотря на то, что все эти операции позволяют взаимодействовать с потоком как неким набором данных наподобие коллекции, важно понимать отличие коллекций от потоков:

- Потоки не хранят элементы. Элементы, используемые в потоках, могут храниться в коллекции, либо при необходимости могут быть напрямую сгенерированы.
- Операции с потоками не изменяют источника данных. Операции с потоками лишь возвращают новый поток с результатами этих операций.
- Для потоков характерно отложенное выполнение. То есть выполнение всех операций с потоком происходит лишь тогда, когда выполняется терминальная операция и возвращается конкретный результат, а не новый поток.

10.2. Создание потока данных

Для создания потока данных можно применять различные методы. В качестве источника потока мы можем использовать коллекции. В частности, в JDK 8 в интерфейс Collection, который реализуется всеми классами коллекций, были добавлены два метода для работы с потоками:

- default Stream<E> stream: возвращается поток данных из коллекции
- default Stream<E> parallelStream: возвращается параллельный поток данных из коллекции

Рассмотрим пример с ArrayList:

```
1   import java.util.stream.Stream;
2   import java.util.*;
3   public class Program {
4
5       public static void main(String[] args) {
6
7           ArrayList<String> cities = new ArrayList<String>();
8           Collections.addAll(cities, "Париж", "Лондон", "Мадрид");
9           cities.stream() // получаем поток
10              .filter(s->s.length()==6) // применяем фильтрацию по длине строки
11              .forEach(s->System.out.println(s)); // выводим отфильтрованные строки на консоль
12      }
13  }
```

Здесь с помощью вызова cities.stream() получаем поток, который использует данные из списка cities. С помощью каждой промежуточной операции, которая применяется к потоку, мы получаем поток с учетом модификаций. Например, мы можем изменить предыдущий пример следующим образом:

```
1   ArrayList<String> cities = new ArrayList<String>();
2   Collections.addAll(cities, "Париж", "Лондон", "Мадрид");
3
4   Stream<String> citiesStream = cities.stream(); // получаем поток
5   citiesStream = citiesStream.filter(s->s.length()==6); // применяем фильтрацию по длине строки
6   citiesStream.forEach(s->System.out.println(s)); // выводим отфильтрованные строки на консоль
```

Важно, что после использования терминальных операций другие терминальные или промежуточные операции к этому же потоку не могут быть применены, поток уже употреблен. Например, в следующем случае мы получим ошибку:

```
1   citiesStream.forEach(s->System.out.println(s)); // терминальная операция употребляет поток
2   long number = citiesStream.count(); // здесь ошибка, так как поток уже употреблен
3   System.out.println(number);
4   citiesStream = citiesStream.filter(s->s.length()>5); // тоже нельзя, так как поток уже употреблен
```

Фактически жизненный цикл потока проходит следующие три стадии:

1-й Создание потока
2-й Применение к потоку ряда промежуточных операций
3-й Применение к потоку терминальной операции и получение результата

Кроме вышерассмотренных методов мы можем использовать еще ряд способов для создания потока данных. Один из таких способов представляет метод Arrays.stream(T[] array), который создает поток данных из массива:

```
1   Stream<String> citiesStream = Arrays.stream(new String[]{"Париж", "Лондон", "Мадрид"}) ;
2   citiesStream.forEach(s->System.out.println(s)); // выводим все элементы массива
```

Для создания потоков IntStream, DoubleStream, LongStream можно использовать соответствующие перегруженные версии этого метода:

```
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));
```

И еще один способ создания потока представляет статический метод of(T..values) класса Stream:

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

// можно передать массив
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));
```

10.3. Фильтрация, перебор элементов и отображение

10.3.1. Перебор элементов. Метод forEach

Для перебора элементов потока применяется метод forEach(), который представляет терминальную операцию. В качестве параметра он принимает объект Consumer<? super String>, который представляет действие, выполняемое для каждого элемента набора. Например:

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

Фактически это будет аналогично перебору всех элементов в цикле for и выполнению с ними действия, а именно вывод на консоль. В итоге консоль выведет:

```
Париж
Лондон
Мадрид
Берлин
Брюссель
```

Кстати мы можем сократить в данном случае применение метода forEach следующим образом:

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

Фактически здесь передается ссылка на статический метод, который выводит строку на консоль.

10.3.2. Фильтрация. Метод filter

Для фильтрации элементов в потоке применяется метод filter(), который представляет промежуточную операцию. Он принимает в качестве параметра некоторое условие в виде объекта Predicate<T> и возвращает новый поток из элементов, которые удовлетворяют этому условию:

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

Здесь условие s.length()==6 возвращает true для тех элементов, длина которых равна 6 символам. То есть в итоге программа выведет:

```
Лондон
Мадрид
Берлин
```

Рассмотрим еще один пример фильтрации с более сложными данными. Допустим, у нас есть следующий класс Phone:

```
class Phone{
    private String name;
    private int price;
```

```
4
5       public Phone(String name, int price){
6           this.name=name;
7           this.price=price;
8       }
9        public String getName() { return name; }
10      public void setName(String name) { this.name = name; }
11      public int getPrice() { return price; }
12      public void setPrice(int price) { this.price = price; }
13  }
```

Отфильтруем набор телефонов по цене:

```
1   Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", 54000), new Phone("Lumia 950", 45000),
2               new Phone("Samsung Galaxy S 6", 40000));
3
4   phoneStream.filter(p->p.getPrice()<50000).forEach(p->System.out.println(p.getName()));
```

10.3.3. Отображение. Метод map

Отображение или маппинг позволяет задать функцию преобразования одного объекта в другой, то есть получить из элемента одного типа элемент другого типа. Для отображения используется метод map, который имеет следующее определение:

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

Передаваемая в метод map функция задает преобразование от объектов типа T к типу R. И в результате возвращается новый поток с преобразованными объектами.

Возьмем вышеопределенный класс телефонов и выполним преобразование от типа Phone к типу String:

```
1   Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", 54000), new Phone("Lumia 950", 45000),
2               new Phone("Samsung Galaxy S 6", 40000));
3
4   phoneStream
5       .map(p-> p.getName()) // помещаем в поток только названия телефонов
6       .forEach(s->System.out.println(s));
```

Операция map(p-> p.getName()) помещает в новый поток только названия телефонов. В итоге на консоли будут только названия:

```
iPhone 6 S
Lumia 950
Samsung Galaxy S 6
```

Еще проведем преобразования:

```
1   phoneStream
2       .map(p-> "название: " + p.getName() + " цена: " + p.getPrice())
3       .forEach(s->System.out.println(s));
```

Здесь также результирующий поток содержит строки, только теперь названия соединяются с ценами.

Для преобразования объектов в типы Integer, Long, Double определены специальные методы mapToInt(), mapToLong() и mapToDouble() соответственно.

10.3.4. Плоское отображение. Метод flatMap

Плоское отображение выполняется тогда, когда из одного элемента нужно получить несколько. Данную операцию выполняет метод flatMap:

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

Например, в примере выше мы выводим название телефона и его цену. Но что, если мы хотим установить для каждого телефона цену со скидкой и цену без скидки. То есть из одного объекта Phone нам надо получить два объекта с информацией, например, в виде строки. Для этого применим flatMap:

```
1   Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", 54000), new Phone("Lumia 950", 45000),
2               new Phone("Samsung Galaxy S 6", 40000));
3
4   phoneStream
5       .flatMap(p->Stream.of(
6               String.format("название: %s цена без скидки: %d", p.getName(), p.getPrice()),
7               String.format("название: %s цена со скидкой: %d", p.getName(), p.getPrice() - (int)(p.getPrice()*0.1))
8       ))
9       .forEach(s->System.out.println(s));
```

Результат работы программы:

```
название: iPhone 6 S цена без скидки: 54000
название: iPhone 6 S цена со скидкой: 48600
название: Lumia 950 цена без скидки: 45000
название: Lumia 950 цена со скидкой: 40500
название: Samsung Galaxy S 6 цена без скидки: 40000
название: Samsung Galaxy S 6 цена со скидкой: 36000
```

10.4. Сортировка

Коллекции, на основе которых нередко создаются потоки, уже имеют специальные методы для сортировки содержимого. Класс Stream также включает возможность сортировки. Такую сортировку мы можем задействовать, когда у нас идет набор промежуточных операций с потоком, которые создают новые наборы данных, и нам надо эти наборы отсортировать.

Для простой сортировки по возрастанию применяется метод sorted():

```
1   import java.util.ArrayList;
2   import java.util.Collections;
3   import java.util.List;
4
5   public class Program {
6
7       public static void main(String[] args) {
8
9           List<String> phones = new ArrayList<String>();
10          Collections.addAll(phones, "iPhone X", "Nokia 9", "Huawei Nexus 6P",
11                  "Samsung Galaxy S8", "LG G6", "Xiaomi MI6",
12                  "ASUS Zenfone 3", "Sony Xperia Z5", "Meizu Pro 6",
13                  "Pixel 2");
14
15          phones.stream()
16                  .filter(p->p.length()<12)
17                  .sorted() // сортировка по возрастанию
18                  .forEach(s->System.out.println(s));
19      }
20  }
```

Консольный вывод после сортировки объектов:

```
LG G6
Meizu Pro 6
Nokia 9
Pixel 2
Xiaomi MI6
iPhone X
```

Однако данный метод не всегда подходит. Уже по консольному выводу мы видим, что метод сортирует объекты по возрастанию, но при этом заглавные и строчные буквы рассматриваются отдельно.

Кроме того, данный метод подходит только для сортировки тех объектов, которые реализуют интерфейс Comparable.

Если же у нас классы объектов не реализуют этот интерфейс или мы хотим создать какую-то свою логику сортировки, то мы можем использовать другую версию метода sorted(), которая в качестве параметра принимает компаратор.

Например, пусть у нас есть следующий класс Phone:

```
1   class Phone{
2       private String name;
3       private String company;
4       private int price;
5
6       public Phone(String name, String comp, int price){
7           this.name=name;
8           this.company=comp;
9           this.price = price;
10      }
11
12      public String getName() { return name; }
13      public int getPrice() { return price; }
14      public String getCompany() { return company; }
15  }
```

Отсортируем поток объектов Phone:

```
import java.util.Comparator;
import java.util.stream.Stream;

public class Program {
    public static void main(String[] args) {

        Stream<Phone> phoneStream = Stream.of(new Phone("iPhone X", "Apple", 600),
            new Phone("Pixel 2", "Google", 500),
            new Phone("iPhone 8", "Apple",450),
            new Phone("Nokia 9", "HMD Global",150),
            new Phone("Galaxy S9", "Samsung", 300));

        phoneStream.sorted(new PhoneComparator())
                .forEach(p->System.out.printf("%s (%s) - %d \n", p.getName(), p.getCompany(), p.getPrice()));
    }
}
class PhoneComparator implements Comparator<Phone>{

    public int compare(Phone a, Phone b){
        return a.getName().toUpperCase().compareTo(b.getName().toUpperCase());
    }
}
```

Здесь определен класс компаратора PhoneComparator, который сортирует объекты по полю name. В итоге мы получим следующий вывод:

```
Galaxy S9 (Samsung) - 300
iPhone 8 (Apple) - 450
iPhone X (Apple) - 600
Nokia 9 (HMD Global) - 150
Pixel 2 (Google) - 500
```

10.5. Получение подпотока и объединение потоков

Ряд методов Stream API возвращают подпотоки или объединенные потоки на основе уже имеющихся потоков.

10.5.1. takeWhile

Метод takeWhile() выбирает из потока элементы, пока они соответствуют условию. Если попадается элемент, который не соответствует условию, то метод завершает свою работу. Выбранные элементы возвращаются в виде потока.

```
import java.util.stream.Stream;

public class Program {

    public static void main(String[] args) {

        Stream<Integer> numbers = Stream.of(-3, -2, -1, 0, 1, 2, 3, -4, -5);
        numbers.takeWhile(n -> n < 0).forEach(n -> System.out.println(n));
    }
}
```

В данном случае программа выбирает из потока числа, пока они меньше нуля. Консольный вывод программы:

```
-3
-2
-1
```

Несмотря на то, что в потоке еще имеются отрицательные числа, метод завершает работу, как только обнаружит первое число, которое не соответствует условию. В этом и состоит отличие, например, от метода filter().

Чтобы в данном случае охватить все элементы, которые меньше нуля, поток следует предварительно отсортировать:

```
Stream<Integer> numbers = Stream.of(-3, -2, -1, 0, 1, 2, 3, -4, -5);
numbers.sorted().takeWhile(n -> n < 0).forEach(n -> System.out.println(n));
```

Консольный вывод программы:

```
-5
-4
-3
-2
-1
```

10.5.2. dropWhile

Метод dropWhile() выполняет обратную задачу - он пропускает элементы потока, которые соответствуют условию до тех пор, пока не встретит элемент, который НЕ соответствует условию:

```
1  Stream<Integer> numbers = Stream.of(-3, -2, -1, 0, 1, 2, 3, -4);
2  numbers.dropWhile(n -> n < 0).forEach(n -> System.out.println(n));
```

Консольный вывод программы:

```
0
1
2
3
-4
```

10.5.3. concat

Статический метод concat() объединяет элементы двух потоков, возвращая объединенный поток:

```
1  import java.util.stream.Stream;
2
3  public class Program {
4
5      public static void main(String[] args) {
6
7          Stream<String> people1 = Stream.of("Tom", "Bob", "Sam");
8          Stream<String> people2 = Stream.of("Alice", "Kate", "Sam");
9
10         Stream.concat(people1, people2).forEach(n -> System.out.println(n));
11     }
12 }
```

Консольный вывод:

```
Tom
Bob
Sam
Alice
Kate
Sam
```

10.5.4. distinct

Метод distinct() возвращает только уникальные элементы в виде потока:

```
1  Stream<String> people = Stream.of("Tom", "Bob", "Sam", "Tom", "Alice", "Kate", "Sam");
2  people.distinct().forEach(p -> System.out.println(p));
```

Консольный вывод:

```
Tom
Bob
Sam
Alice
Kate
```

10.6. Методы skip и limit

Метод skip(long n) используется для пропуска n элементов. Этот метод возвращает новый поток, в котором пропущены первые n элементов.

Метод limit(long n) применяется для выборки первых n элементов потоков. Этот метод также возвращает модифицированный поток, в котором не более n элементов.

Зачастую эта пара методов используется вместе для создания эффекта постраничной навигации. Рассмотрим, как их применять:

```
1  Stream<String> phoneStream = Stream.of("iPhone 6 S", "Lumia 950", "Samsung Galaxy S 6", "LG G 4", "Nexus 7");
2
3  phoneStream.skip(1)
4      .limit(2)
5      .forEach(s->System.out.println(s));
```

В данном случае метод skip пропускает один первый элемент, а метод limit выбирает два следующих элемента. В итоге мы получим следующий консольный вывод:

```
Lumia 950
Samsung Galaxy S 6
```

Вполне может быть, что метод skip может принимать в качестве параметра число большее, чем количество элементов в потоке. В этом случае будут пропущены все элементы, а в результирующем потоке будет 0 элементов.

И если в метод limit передается число, большее, чем количество элементов, то просто выбираются все элементы потока. Теперь рассмотрим, как создать постраничную навигацию:

```
1   import java.util.ArrayList;
2   import java.util.Arrays;
3   import java.util.List;
4   import java.util.stream.*;
5   import java.util.Scanner;
6
7   public class Program {
8
9       public static void main(String[] args) {
10
11          List<String> phones = new ArrayList<String>();
12          phones.addAll(Arrays.asList(new String[]
13                  {"iPhone 6 S", "Lumia 950", "Huawei Nexus 6P",
14                  "Samsung Galaxy S 6", "LG G 4", "Xiaomi MI 5",
15                  "ASUS Zenfone 2", "Sony Xperia Z5", "Meizu Pro 5",
16                  "Lenovo S 850"}));
17
18          int pageSize = 3; // количество элементов на страницу
19          Scanner scanner = new Scanner(System.in);
20          while(true){
21              System.out.println("Введите номер страницы: ");
22              int page = scanner.nextInt();
23
24              if(page<1) break; // если число меньше 1, выходим из цикла
25
26              phones.stream().skip((page-1) * pageSize)
27                  .limit(pageSize)
28                  .forEach(s->System.out.println(s));
29          }
30      }
31  }
```

В данном случае у нас набор из 10 элементов. С помощью переменной pageSize определяем количество элементов на странице - 3. То есть у нас получится 4 страницы (на последней будет только один элемент).

В бесконечном цикле получаем номер страницы и выбираем только те элементы, которые находятся на указанной странице.

Теперь введем какие-нибудь номера страниц, например, 4 и 2:

```
Введите номер страницы:
4
Lenovo S 850
Введите номер страницы:
2
Samsung Galaxy S 6
LG G 4
Xiaomi MI 5
```

10.7. Операции сведения

Операции сведения представляют терминальные операции, которые возвращают некоторое значение - результат операции. В Stream API есть ряд операций сведения.

10.7.1. count

Метод count() возвращает количество элементов в потоке данных:

```
1   import java.util.stream.Stream;
2   import java.util.Optional;
3   import java.util.*;
4   public class Program {
5
6       public static void main(String[] args) {
7
8           ArrayList<String> names = new ArrayList<String>();
9           names.addAll(Arrays.asList(new String[]{"Tom", "Sam", "Bob", "Alice"}));
```

```
        System.out.println(names.stream().count()); // 4

        // количество элементов с длиной не больше 3 символов
        System.out.println(names.stream().filter(n->n.length()<=3).count()); // 3
    }
}
```

10.7.2. findFirst и findAny

Метод findFirst() извлекает из потока первый элемент, а findAny() извлекает случайный объект из потока (нередко так же первый):

```
ArrayList<String> names = new ArrayList<String>();
names.addAll(Arrays.asList(new String[]{"Tom", "Sam", "Bob", "Alice"}));

Optional<String> first = names.stream().findFirst();
System.out.println(first.get());   // Tom

Optional<String> any = names.stream().findAny();
System.out.println(first.get());   // Tom
```

10.7.3. allMatch, anyMatch, noneMatch

Еще одна группа операций сведения возвращает логическое значение true или false:

- boolean allMatch(Predicate<? super T> predicate): возвращает true, если все элементы потока удовлетворяют условию в предикате
- boolean anyMatch(Predicate<? super T> predicate): возвращает true, если хоть один элемент потока удовлетворяют условию в предикате
- boolean noneMatch(Predicate<? super T> predicate): возвращает true, если ни один из элементов в потоке не удовлетворяет условию в предикате

Пример использования функций:

```
import java.util.stream.Stream;
import java.util.Optional;
import java.util.ArrayList;
import java.util.Arrays;
public class Program {

    public static void main(String[] args) {

        ArrayList<String> names = new ArrayList<String>();
        names.addAll(Arrays.asList(new String[]{"Tom", "Sam", "Bob", "Alice"}));

        // есть ли в потоке строка, длина которой больше 3
        boolean any = names.stream().anyMatch(s->s.length()>3);
        System.out.println(any);   // true

        // все ли строки имеют длину в 3 символа
        boolean all = names.stream().allMatch(s->s.length()==3);
        System.out.println(all);   // false

        // НЕТ ЛИ в потоке строки "Bill". Если нет, то true, если есть, то false
        boolean none = names.stream().noneMatch(s-> s.equals("Bill"));
        System.out.println(none);  // true
    }
}
```

10.7.4. min и max

Методы min() и max() возвращают соответственно минимальное и максимальное значение. Поскольку данные в потоке могут представлять различные типы, в том числе сложные классы, то в качестве параметра в эти методы передается объект интерфейса Comparator, который указывает, как сравнивать объекты:

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

Оба метода возвращают элемент потока (минимальный или максимальный), обернутый в объект Optional.
Например, найдем минимальное и максимальное число в числовом потоке:

```
1   import java.util.stream.Stream;
2   import java.util.Optional;
3   import java.util.ArrayList;
4   import java.util.Arrays;
5   public class Program {
6
7       public static void main(String[] args) {
8
9           ArrayList<Integer> numbers = new ArrayList<Integer>();
10          numbers.addAll(Arrays.asList(new Integer[]{1,2,3,4,5,6,7,8,9}));
11
12          Optional<Integer> min = numbers.stream().min(Integer::compare);
13          Optional<Integer> max = numbers.stream().max((a, b) -> a.compareTo(b));
14          System.out.println(min.get()); // 1
15          System.out.println(max.get()); // 9
16      }
17  }
```

Интерфейс Comparator - это функциональный интерфейс, который определяет один метод compare, принимающий два сравниваемых объекта и возвращающий число (если первый объект больше, возвращается положительное число, иначе возвращается отрицательное число). Поэтому вместо конкретной реализации компаратора мы можем передать лямбда-выражение или метод, который соответствует методу compare интерфейса Comparator. Поскольку сравниваются числа, то в метод передается в качестве компаратора статический метод Integer.compare().

При этом методы min и max возвращают именно Optional, и чтобы получить непосредственно результат операции из Optional, необходимо вызвать метод get().

Рассмотрим более сложный случай, когда нам надо сравнивать более сложные объекты:

```
1   import java.util.stream.Stream;
2   import java.util.Optional;
3   import java.util.ArrayList;
4   import java.util.Arrays;
5
6   public class Program {
7       public static void main(String[] args) {
8
9           ArrayList<Phone> phones = new ArrayList<Phone>();
10          phones.addAll(Arrays.asList(new Phone[]{
11              new Phone("iPhone 8", 52000),  new Phone("Nokia 9", 35000),
12              new Phone("Samsung Galaxy S9", 48000),  new Phone("HTC U12", 36000)
13          }));
14
15          Phone min = phones.stream().min(Phone::compare).get();
16          Phone max = phones.stream().max(Phone::compare).get();
17          System.out.printf("MIN Name: %s Price: %d \n", min.getName(), min.getPrice());
18          System.out.printf("MAX Name: %s Price: %d \n", max.getName(), max.getPrice());
19      }
20  }
21  class Phone{
22      private String name;
23      private int price;
24
25      public Phone(String name, int price){
26          this.name=name;
27          this.price=price;
28      }
29      public static int compare (Phone p1, Phone p2){
30          if(p1.getPrice() > p2.getPrice())
31              return 1;
32          return -1;
33      }
34      public String getName() { return name; }
35      public int getPrice() { return price;}
36  }
```

В данном случае мы находим минимальный и максимальный объект Phone: фактически объекты с максимальной и мини-

мальной ценой. Для определения функциональности сравнения в классе Phone реализован статический метод compare, который соответствует сигнатуре метода compare интерфейса Comparator. И в методах min и max применяем этот статический метод для сравнения объектов.

Консольный вывод:

```
MIN Name: Nokia 9 Price: 35000
MAX Name: iPhone 8 Price: 52000
```