Потоки данных

Итерирование коллекции

Типичная операция – обойти коллекцию, применяя итерацию к каждому элементу

```
List <Student> allStudents;
. . .
int count=0;
for (Student  s:allStudents){
  if (s.isFrom(“Sfedu”))
          count++;
}
```

```
List <Student> allStudents;
. . .
int count=0;
Iterator<Student> it =
    allStudents.iterator();
while(it.hasNext()){
Student  s = it.next();
  if (s.isFrom(“Sfedu”) )
          count++;
}
```

Проблемы

- Много стереотипного кода
- Невозможно распараллелить
- При вложенности циклов с большим количеством кода затемняется смысл

Внутреннее итерирование

```
long    count = allStudents.stream()
                           .filter(st ->st.isFrom(“Sfedu”))
                           .count();
```

Для параллельного выполнения

```
long    count = allStudents.parallelStream()
                           .filter(st ->st.isFrom(“Sfedu”))
                           .count();
```

Поток данных

- Средство конструирования сложных операций над коллекциями (массивами, генераторами или итераторами) с применением функционального подхода
- Не сохраняет свои элементы, они или хранятся в основной коллекции или формируются по требованию
- Потоковые операции не изменят их источник, а формируют новые потоки или выдают результат

Создание потока

- Интерфейс Stream<T>
- Для любой коллекции
  - методом stream() или parallelStream()
- Для массива
  - метод Sream.of(array);
  - метод Arrays.stream(array,from,to)
- Пустой поток
  - Stream<String> s=Stream.empty();

Создание потока

- Бесконечный поток
  - Статические методы интерфейса Stream
  - Используя объект функционального интерфейса Supplier<T>

```
Stream<String> echo =Stream.generate(
                        () -> “Echo”);
Stream<Double> randoms =Stream.generate(
                        Math::random);
```

Создание потока

- Бесконечный поток
  - Статические методы интерфейса Stream
  - Используя объект функционального интерфейса UnaryOperator <T>

```
Stream <BigInteger> integers = Stream.iterate(
    BigInteger.ZERO, n -> n.add(BigInteger.ONE));
```

Создание потока

- Ограничить поток
  - Используя предикатную функцию

```
BigInteger limit = new BigInteger(“10000000000”);
Stream <BigInteger> integers = Stream.iterate(
    BigInteger.ZERO,
    n ->n.compareTo(limit),
    n -> n.add(BigInteger.ONE));
```

Создание потока

- Статические методы других классов

```
String str;
String regex = . . .
. . .
Stream <String> words =Pattern.compile(regex)
                                        . splitAsStream(str);

try( Stream<String> lines =  Files.lines(path)){
 . . .
}
```

Методы потоков

- Преобразуют поток в другой поток
  - filter() формирует поток с данными, удовлетворяющими условию.
  - Условие – объект типа Predicate<T>

```
List <String> words =…;
Stream<String> longW = words.stream()
                                    . filter(w->w.length()>12);
```

Методы потоков

- Преобразуют поток в другой поток
  - map() формирует поток с преобразованием

```
List <String> words =…;
Stream<String>
    lowercaseW= words.stream()
                        . map(String::toLowerCase);
Stream<String>
    firstLetters =  words.stream()
                        . map(s -> s.substring(0,1));
```

Методы потоков

- Ограничение потока
  - limit(n)
  - skip (n)
  - takeWhile(предикат)
  - dropWhile(предикат)

```
Stream <Double> randoms = Stream.generate(Math::random)
                                                .limit(100);
```

Методы потоков

- Соединение двух потоков
  - concat(поток1, поток 2)
- Другие преобразования
  - reversed() обратный порядок
  - distinct() подавление дубликатов в потоке
  - sorted(компаратор) сортировка

```
Stream<String>  lengthSort   =
        words.stream()
                .sorted(Comparator.comparing(String::length));
```

Методы потоков

- Выполнение для каждого элемента потока
  - peek(функция)

Методы сведения

- Выполнят операции, сводя поток к непотоковому значению
  - count()
  - anyMatch()
  - allMatch()
  - noneMatch()
  - возвращающие значение Optional<T>
    - max()
    - min()
    - findFirst()
    - findAny()

```
Optional<String> startWithQ  =
        words.parallel()
                .filter(s -> s.startWith(“Q”)).
                .findAny();

boolean aWordStartWithQ  =
        words.parallel()
                .anyMatch (s -> s.startWith(“Q”));
```

Накопление результатов

- просмотр результатов

  stream.forEach(System.out::println);

  srteam.forEachOrdered(System.out::println);

- сохранение в структуре данных

  String [ ] result = stream.toArray(String[ ]::new);

  List <String> result = stream.collect (
  Collectors.toList());

  Set <String> result = stream.collect (
  Collectors.toSet());

Дополнительно

Как уйти от обобщенного типа в интерфейсе

```
public class Letters extends AbstractCollection<Character>{
//implements Collection<Character>{
 // @Override
    public boolean contains(Character o) {
        return collect.indexOf(o)>=0;
        //throw new UnsupportedOperationException("Not supported yet.");
      //To change body of generated methods, choose Tools | Templates.
    }
```