@techwithvishalraj

Java

Stream API

Follow

Share

@techwithvishalraj

Stream API

- Java provides a new additional package in Java 8 called java.util.stream.
- This package consists of classes, interfaces and enum to allows functional-style operations on the elements.
- A stream is a sequence of elements that supports various operations to perform computations upon those elements. Streams do not store elements; they are functional-style constructs that allow for processing elements on-demand.
- The elements of a stream are only visited once during the life of a stream.

- **Intermediate Operations:** Operations that can be chained together to form a pipeline for processing elements in a stream. Examples include filter, map, flatMap, distinct, sorted, limit, skip, etc.

- **Terminal Operations:** Operations that terminate a stream pipeline and produce a result. Examples include forEach, collect, reduce, count, min, max, anyMatch, toList, etc.

- **Lazy Evaluation:** intermediate operations are only executed when a terminal operation is invoked.

- **Parallel Streams:** Streams can be processed in parallel to take advantage of multi-core processors. This can be achieved by invoking the parallel() method on a stream.

@techwithvishalraj

```
List<Integer> list = Arrays.asList(1, 4, 5, 3, 2, 8, 6, 3, 0);
```

filter()

syntax: Stream<T> filter(Predicate<? super T> predicate)

It returns a stream consisting of the elements of this stream that match the given predicate.

```
List<Integer> evenList = list.stream().filter(x->x%2==0).toList();
Sop(evenList);                    -------->  [4, 2, 8, 6, 0]
```

map()

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

It returns a stream consisting of the results of applying the given function to the elements of this stream.

```
List<Integer> evenSquaresList = list.stream().map(x-> x*x).toList();
Sop(evenSquaresList);              ----->  [1, 16, 25, 9, 4, 64, 36, 9, 0]
```

sorted()

```
List<Integer> sortedlist = list.stream().sorted().toList();
Sop(sortedlist);                        ----->  [0, 1, 2, 3, 3, 4, 5, 6, 8]
```

@techwithvishalraj

```
List<Integer> list = Arrays.asList(1, 1, 3, 3, 2, 2, 6, 3, 0);
```

distinct()

It eliminates duplicate elements, ensuring that each element in the resulting stream is unique.

```
List<Integer> distinctList = list.stream().distinct().toList();
System.out.println(distinctList);          -------> [1, 3, 2, 6, 0]
```

skip(long n)

skips the first n elements of the stream and returns a new stream consisting of the remaining elements. If the stream contains fewer than n elements then empty stream is returned.

```
List<Integer> skippedList = list.stream().skip(3).toList();
System.out.println(skippedList);          ----> [3, 2, 2, 6, 3, 0]
```

limit(lomg maxSize)

returns a stream consisting of the first maxSize elements of the original stream. If the original stream contains fewer than maxSize elements, then all elements of the original stream is returned.

```
List<Integer> limitedList = list.stream().limit(2).toList();
System.out.println(limitedList);                ----> [1, 1]
```

@techwithvishalraj

flatmap()

- flatMap() method is an intermediate operation that is used to flatten nested streams into a single stream.
- It is particularly useful when dealing with nested collections or when you want to transform each element in a stream into multiple elements and then flatten those elements into a single stream by concatenating them.

```
List<List<Integer>> nestedList = Arrays.asList(Arrays.asList(5, 6, 7),
                                               Arrays.asList(2, 1, 3),
                                               Arrays.asList(3, 5, 0));

List<Integer> flattenedList = nestedList.stream()
                                        .flatMap(List::stream)
                                        .toList();
System.out.println("flattenedList: "+flattenedList);

List<Integer> sortedFlattenedList = nestedList.stream()
                                              .flatMap(List::stream)
                                              .sorted()
                                              .toList();
System.out.println("sortedFlattenedList: "+sortedFlattenedList);
```

```
output: flattenedList: [5, 6, 7, 2, 1, 3, 3, 5, 0]
        sortedFlattenedList: [0, 1, 2, 3, 3, 5, 5, 6, 7]
```

@techwithvishalraj

flatmap()

- **flatMap() is used is when you have a collection of objects, and each object contains a collection of elements.**

```
class Person{
   String name;
   List<Long> phoneNo;

   Person (String name, Long... phoneNo){
      this.name=name;
      this.phoneNo= Arrays.asList(phoneNo);
   }
}
public class Test2 {
   public static void main(String[] args) {
      List<Person> persons = Arrays.asList(
            new Person("ram", 9087897899L, 9089076540L),
            new Person("shyam", 8978900890L, 7897897890L),
            new Person("baburao", 8888888888L, 9999887890L));
      List<Long> phoneNos= persons.stream()
                              .flatMap(person -> person.phoneNo.stream())
                              .toList();
      System.out.println(phoneNos);
   }
}
```

output: [9087897899, 9089076540, 8978900890, 7897897890, 8888888888, 9999887890]

@techwithvishalraj

forEach()

```
List<Integer> list = Arrays.asList(1, 1, 3, 3, 2, 2, 6, 3, 0);

list.stream().filter(x-> x%2==0).forEach(x->System.out.print(x+" "));
                                                    ----->   2 2 6 0
list.stream().filter(x-> x%2==0).forEach(System.out::print);  ---> 2260
```

count(), max(), min()

```
List<Integer> list = Arrays.asList(1, 1, 3, 3, 2, 2, 6, 3, 0);

long count = list.stream().count();                          ----------> 9

Optional<Integer> max = list.stream().max(Integer::compareTo);
System.out.println(max.orElse(null));                        ----------> 6

Optional<Integer> min = list.stream().min(Integer::compareTo);
System.out.println(min.orElse(null));                        ----------> 0
```

anyMatch()

Syntax: boolean anyMatch(Predicate<? super T> predicate)
It check whether any element of the stream matches a given predicate.

```
boolean result = list.stream().anyMatch(x-> x>5);          -----> true
```

@techwithvishalraj

reduce()

Syntax: T reduce(T identity, BinaryOperator<T> accumulator)

Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value. This is equivalent to:

```
T result = identity;
for (T element : this stream)
        result = accumulator.apply(result, element)
return result;
```

```
List<Integer> list = Arrays.asList(1, 1, 3, 3, 2, 2, 6, 3, 0);

int sum = list.stream().reduce(0, (a, b)->a+b);                    ----> 21
int sum = list.stream().reduce(0, Integer::sum);                   ----> 21
int sum = list.stream().reduce(2, (a, b)->a+b);                    ----> 23


int max = list.stream().reduce(Integer.MIN_VALUE,
                                        (a, b)->Math.max(a, b));   -----> 6
int min = list.stream().reduce(Integer.MAX_VALUE,
                                        (a, b)->Math.min(a, b));   -----> 0
```

```
List<String> names = Arrays.asList("ram", "shyam", "baburao");
String result = names.stream().reduce("", (s1, s2)-> s1+s2);
                                          ----------> ramshyambaburao
```

@techwithvishalraj

collect()

- It is used to accumulate the elements of a stream into a collection or another container.
- It's a versatile operation that allows you to collect stream elements into various data structures such as lists, sets, maps, and even custom collectors.

```
List<Integer> list = Arrays.asList(1, 1, 3, 3, 2, 2, 6, 3, 0);

List<Integer> collectedList = list.stream().collect(Collectors.toList());
System.out.println(collectedList);          -----> [1, 1, 3, 3, 2, 2, 6, 3, 0]

Set<Integer> set = list.stream().collect(Collectors.toSet());
System.out.println(set);                          ------> [0, 1, 2, 3, 6]
```

```
List<String> names = Arrays.asList("ram", null, "shyam", "baburao");

String result = names.stream().filter(Objects::nonNull)
                    .collect(Collectors.joining("-"));
System.out.println(result);                 ---> ram-shyam-baburao
```

@techwithvishalraj

collect(Collectors.toMap())

```
class Employee{
    private String name;
    private Integer age;
    private String city;

    public Employee(String name, Integer age, String city) {
        this.name = name;
        this.age = age;
        this.city = city;
    }
  // getters and setters
}

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

        List<Employee> employees = Arrays.asList(
                                        new Employee("ram", 27, "culcutta"),
                                        new Employee("shyam", 28, "delhi"),
                                        new Employee("baburao", 45, "mumbai"));

        Map<String, Integer> map = employees.stream()
                        .collect(Collectors.toMap(Employee::getName,
                                                  Employee::getAge));

        System.out.println(map); ----> {shyam=28, baburao=45, ram=27}
    }
}
```

@techwithvishalraj

Sorting more than once

```
class Employee{
   private String name;
   private Integer age;
   private String city;
   private Integer salary;
 // constructor, getters and setters
}
public class Test {
  public static void main(String[] args) {
      List<Employee> employees = Arrays.asList(
                    new Employee("baburao", 45, "mumbai", 4500),
                    new Employee("ram", 27, "kalkatta", 0),
                    new Employee("shyam", 25, "jaipur", 1000),
                    new Employee("sam", 25, "delhi", 1000));

      employees.stream().sorted(Comparator
                                .comparing(Employee::getSalary)
                                .thenComparing(Employee::getAge)
                                .thenComparing(Employee::getCity))
                               .forEach(x-> System.out.println(x.getName()+
                                        " : "+ x.getAge()+" : "+ x.getCity()+
                                        " : "+x.getSalary()));
      }
}
```

```
output: ram : 27 : kalkatta : 0
        sam : 25 : delhi : 1000
        shyam : 25 : jaipur : 1000
        baburao : 45 : mumbai : 4500
```

@techwithvishalraj

HashMap Sorting

```
public class Test {
  public static void main(String[] args) {
    Map<String, Integer> map = new HashMap<>();
    map.put("baburao", 45);
    map.put("ram", 27);
    map.put("shyam", 25);
    map.put("sam", 25);

Map<String, Integer> sortedMap = new LinkedHashMap<>();
map.entrySet().stream().sorted(Map.Entry.comparingByValue())
          .forEachOrdered(x-> sortedMap.put(x.getKey(), x.getValue()));

 Sop(sortedMap);      ----> {shyam=25, sam=25, ram=27, baburao=45}

 Map<String, Integer> sortedMap1 = new LinkedHashMap<>();
 map.entrySet().stream().sorted(Map.Entry.comparingByKey())
          .forEachOrdered(x->sortedMap1.put(x.getKey(), x.getValue()));

 Sop(sortedMap1);    -----> {baburao=45, ram=27, sam=25, shyam=25}

Map<String, Integer> sortedMap2 = new LinkedHashMap<>();
map.entrySet().stream()
                .sorted(Map.Entry.<String, Integer>comparingByValue()
                .thenComparing(Map.Entry.comparingByKey()))
        .forEachOrdered(x->sortedMap2.put(x.getKey(), x.getValue()));

Sop(sortedMap2);      ----> {sam=25, shyam=25, ram=27, baburao=45}
```

@techwithvishalraj

Thank you!

vishal-bramhankar

techwithvishalraj

Vishall0317