Lecture 4 & Lab 4

Streams API(Since java8)

0. **Reference(Java SE 17 & JDK 17)**

   > Stream (Java SE 17 & JDK 17) (oracle.com)

1. Aim:

   Used to process collections of objects

2. Create a Stream

   1. Approach I: from a collection

      examples:

      ```
      List<String> list = new ArrayList<String>();
      Stream<String> s = list.stream();
      ```

   2. Approach II: use supplier

      Could generate "infinite" streams

      ```
      static Stream<T> generate(Supplier<T> s)
      ```

      example:

      ```
      Stream<Double> s = Stream.generate(Math::random);

      class NatualSupplier implements Supplier<Integer> {
          int n = 0;
          public Integer get() {
              n++;
              return n;
          }
      }
      Stream<Integer> natual = Stream.generate(new NatualSupplier());
      natual.limit(20).forEach(System.out::println);
      ```

   3. Approach III: use Stream.of()

      take any number of arguments

      ```
      static <T> Stream<T> of(T ... values);
      ```

      examples

```java
Stream<String> sentence = Stream.of("This","is","Java","2");

Stream<int[]> s = Stream.of(new int[]{1, 2, 3});
// Only one element in the Stream, that is, the int array.
```

4. Primitive Type Streams

REFERENCE(Java SE 17 & JDK 17):

> IntStream (Java SE 17 & JDK 17) (oracle.com)

1. How to deal with `int[]`?

```java
// unrealizable
int [] a = {1, 2, 3};
List<Integer> l = Arrays.asList(a); //compilation error
```

solutions: using primitive type streams

2. The stream library has specialized types `IntStream`, `LongStream`, and `DoubleStream` that store primitive values directly, without using wrappers (e.g., Integer).

3. create an IntStream

Arrays.stream

```java
// convert an array to IntStream
// In "Arrays"
public static IntStream stream(int[] array);
public static IntStream stream(int[] array,
                               int startInclusive,
                               int endExclusive);
/*
Parameters:
array - the array, assumed to be unmodified during use
startInclusive - the first index to cover, inclusive
endExclusive - index immediately past the last index to cover
*/

//example
IntStream stream0 = Arrays.stream(new int[]{1,2,3});
```

IntStream.of

```java
// generate by several ints
static IntStream of(int... values);

//example
int [] ints = {1, 2, 3};
IntStream stream1 = IntStream.of(1,2,3,5,8);
IntStream stream1 = IntStream.of(ints); // ok
```

IntStream.range

```
// generate by a range [5, 10)
static IntStream range(int startInclusive,
                       int endExclusive);

//example
IntStream stream2 = IntStream.range(5,10);
```

s.mapToInt

```
// transform an stream to IntStream
IntStream mapToInt(ToIntFunction<? super T> mapper);

//example
Stream<String> sentences = Stream.of("This","is","Java","2");
IntStream stream3 = sentences.mapToInt(String::length);
```

4. boxed to Integers

```
Stream<Integer> boxed();

// example
Stream <Integer> Is = IntStream.Of(1, 2, 3).boxed();
```

Operations of Stream<T>

1. Immediate Operations
    1. Intermediate (non-terminal) operations **transform** or **filter** the elements in the stream
    2. We **will** get a **new stream** back as the result when adding an intermediate operation to a stream

       (pay attention to the return value, it's a new Stream)
    3. machanism: **Lazy evaluation**

       All intermediate operations do not get executed until a **terminal operation** is invoked (discussed later)
    4. `filter()`

       Returns a stream consisting of the elements of this stream that **match** the given predicate.

```
//syntax
Stream<T> filter(Predicate<? super T> predicate);
//example
List<Integer> list = Arrays.asList(10,20,33,43,54,68);
list.stream()
    .filter(element -> (element % 2==0))
    .forEach(element -> System.out.print(element+ " "));
```

5. `map()`

   Returns a stream consisting of the results of **applying/mapping the given function** to the elements of this stream.

```
// syntax
// transform a Stream<T> to Stream<R> by a funcion
<R> Stream<R> map(Function<? super T,? extends R> mapper);
// example
List<String> strList = new ArrayList<String>();
strList.add("123");
strList.add("456");
strList.stream()
    .map(Integer::parseInt)
    .forEach(System.out::println);
```

6. `distinct()`

   Returns a stream consisting of the distinct elements (**removing duplicates** and keeping only one of them)

```
// syntax
Stream<T> distinct();
// example
List<String> strList = new ArrayList<String>();
strList.add("apple");
strList.add("orange");
strList.add("banana");
strList.add("apple");
List<String> result = strList.stream()
    .distinct()
    .collect(Collectors.toList());
```

7. sorted

   `sorted()`: sort the elements by natural order

```
// syntax
Stream<T> sorted();
// example
list.stream().sorted().forEach(System.out::println);
```

   `sorted(Comparator<? super T> comparator)`: sort the elements according to the given Comparator

```
// syntax
Stream<T> sorted(Comparator<? super T> comparator);
// example
class Point
{
    Integer x, y;
```

```
    Point(Integer x, Integer y) {
        this.x = x;
        this.y = y;
    }
}
aList.stream()
    .sorted((p1, p2)->p1.x.compareTo(p2.x))
    .forEach(System.out::println);
```

8. `peek()`

   Returns a stream consisting of the elements of this stream(**never delete**), **additionally** performing the provided action on each element **when** elements "pass through" the `peek()` method.

   Mainly used for **debugging**. By `skip()`, we can inspect every elemtn in the Stream.

```
// syntax
Stream<T> peek(Consumer<? super T> action);
// example
Stream.of("one", "two", "three", "four")
        .filter(e -> e.length() > 3)
        .peek(e -> System.out.println("Filtered value: " + e))
        .forEach(System.out::println);
/*
Filtered value: three
three
Filtered value: four
four
*/
```

9. `limit()`

   Returns a stream consisting of the elements of this stream, **truncated to be no longer than** `maxSize` in length.

```
// syntax
Stream<T> limit(long maxSize);
// example
class NatualSupplier implements Supplier<Integer> {
    int n = 0;
    public Integer get() {
        n++;
        return n;
    }
}
Stream<Integer> s = Stream.generate(new NatualSupplier());
s.limit(20).forEach(System.out::println);
```

10. `skip()`

Returns a stream consisting of the **remaining elements** of this stream after **discarding the first `n` elements** of the stream.

If this stream contains **fewer than `n` elements** then an **empty stream** will be returned.

```
// syntax
Stream<T> skip(long n);
// example
class NatualSupplier implements Supplier<Integer> {
    int n = 0;
    public Integer get() {
        n++;
        return n;
    }
}
Stream<Integer> s = Stream.generate(new NatualSupplier());
s.skip(10).limit(20).forEach(System.out::println);
```

2. Terminal operation
    1. A terminal operation marks the **end** of the stream and is always the **last operation** in the stream pipeline
    2. A terminal operation returns a **non -stream** type of result
    3. machanism: Eager execution

        Terminal operations are executed immediately
    4. `anyMatch()`

        Returns whether **any elements** of this stream **match** the provided predicate (check whether any element in list satisfies a given condition)

        ```
        // syntax
        boolean anyMatch(Predicate<? super T> predicate);
        // example
        boolean x = sList.stream().anyMatch(e -> e.startsWith("Java"));
        ```

    5. `findFirst()`

        Returns an **Optional** describing the first element of this stream, or an **empty Optional** if the stream is empty

        ```
        // syntax
        Optional<T> findFirst();
        // example
        List<String> stringList = new ArrayList<String>();
        stringList.add("one");
        stringList.add("two");
        stringList.add("three");
        Stream<String> stream = stringList.stream();
        Optional<String> result = stream.findFirst();
        System.out.println(result.orElse("unknown"));
        ```

6. Collecting Results

REFERENCE:

> Collectors (Java SE 17 & JDK 17) (oracle.com)

When you are done with a stream, you often want to **collect the result in a data structure**

1. Transforming to arrays:

```
// syntax
// Represents a function that accepts an int-valued argument and
produces a result.
public interface IntFunction<R>{
    R apply(int value);
}
<A> A[] toArray(IntFunction<A[]> generator);
// example
String[] result = stream.toArray(String[]::new);
```

2. Transforming to Collections or Maps

   collectors class

   Implementations of `Collector` that implement various useful reduction operations, such as **accumulating elements into collections**, **summarizing elements** according to various **criteria**, etc.

```
// example
Stream<String> stream = Stream.of("a", "bb", "cc", "ddd");
```

   1. syntax

```
// collector specifies how elements are collected
<R,A> R collect(Collector<? super T,A,R> collector);
```

   2. transforming to a list

```
public static <T> Collector<T,?,List<T>> toList();
// example
List<String> result = stream.collect(Collectors.toList());
```

   3. transforming to a Set

```
public static <T> Collector<T,?,Set<T>> toSet();
// example
Set<String> result = stream.collect(Collectors.toSet());
```

   4. transforming to other collections

```
public static <T,C extends Collection<T>> Collector<T,?,C>
toCollection(Supplier<C> collectionFactory);
/*
Parameters:
collectionFactory - a supplier providing a new empty Collection
into which the results will be inserted
*/
TreeSet<String> result =
stream.collect(Collectors.toCollection(TreeSet::new));
```

5. transforming to maps

```
public static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<?
super T,? extends K> keyMapper, Function<? super T,? extends U>
valueMapper);
/*
keyMapper - a mapping function to produce keys from elements in
the Stream
valueMapper - a mapping function to produce values from elements
in the Stream
*/
static <T> Function<T,T> identity()
// Returns a function that always returns its input argument.
Map<String, Integer> map =
stream.collect(Collectors.toMap(Function.identity(),
String::length));
```

6. transforming to a string

```
public static Collector<CharSequence,?,String>
joining(CharSequence delimiter);
// example
String joined = stream.collect(Collectors.joining("$"));
```

3. **grouping**

We use `Collectors.groupingBy` for grouping objects by some property and storing results in a Map instance.

```
// syntax
public static <T,K,A,D> Collector<T,?,Map<K,D>>
groupingBy(Function<? super T,? extends K> classifier, Collector<?
super T,A,D> downstream);
/*
Parameters:
classifier - a classifier function mapping input elements to keys
downstream - a Collector implementing the downstream reduction in a
group that mapping to the same key.
*/
T = city;
k = String;

//example
Stream<String> stream = Stream.of("a", "bb", "cc", "ddd", "a",
"bb");
Map<String, Long> group =
stream.collect(Collectors.groupingBy(Function.identity(),
Collectors.counting()));
```

7. Reduction: `reduce`
    1. A reduction is a terminal operation that aggregates a stream into a type.
    2. Performs a reduction **on the elements of this stream**, using an **associative** accumulation function.

       Returns an `Optional`:

       Stream has nothing => nothing

       Otherwise => return reduced value

```
// syntax
Optional<T> reduce(BinaryOperator<T> accumulator);
// with an initial value identity, we can certainly return a T
T reduce(T identity, BinaryOperator<T> accumulator)
// example
List<Integer> l = new ArrayList<>(Arrays.asList(1, 2, 3));
int product = l.stream().reduce(1, (a, b)->a*b);
// product = 1 if al.size() == 0
// else return the product of the list
List<Integer> l = new ArrayList<>(Arrays.asList(1, 2, 3));
int product = l.stream().reduce((a, b)->a*b).orElse(0);
// product = 0 if al.size() == 0
// else return the product of the list
```

3. Stream features
    1. When encountering a terminal operation, the Stream will execute all the operations.
    2. If a immediate operation can operate the element one by one, then the this operation will operate next element until the current element is **terminated** in the stream or **thrown** in some steps.

```
Stream.of("209", "CS", "303", "A", "B")
        .sorted((s1, s2) -> {
            System.out.printf("sort: %s; %s\n", s1, s2);
            return s1.compareTo(s2);
        })
        .filter(s -> {
            System.out.println("filter: " + s);
            return s.startsWith("C") || s.startsWith("A");
        })
        .map(s -> {
            System.out.println("map: " + s);
            return s.toLowerCase();
        })
        .forEach(s -> System.out.println("forEach: " + s));

// answer
/*
sort: CS; 209
sort: 303; CS
sort: 303; CS
sort: 303; 209
sort: A; 303
sort: A; CS
sort: B; A
sort: B; CS
filter: 209
filter: 303
filter: A
map: A
forEach: a
filter: B
filter: CS
map: CS
forEach: cs
*/
```

3. Under the machnism of 2, a Stream will automatically **terminate** when the terminal operation **figure out the answer.**

```
Stream.of("CS", "209", "A").map(s -> {
    System.out.println("map: " + s);
    return s.toLowerCase();
})
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
        return s.startsWith("c");
    });
// answer
/*
map: CS
anyMatch: cs
```

```
13 */
14 // when dealing with the cs, Stream knows the answer is false
```

4. Reuse the Stream

   Actually we cannot reuse the Stream after the terminal operation

```
1 Stream<String> stream =
2                 Stream.of("CS", "209", "A")
3                         .filter(s -> s.startsWith("C"));
4 stream.anyMatch(s -> true);    // ok
5 stream.forEach(System.out::println);   // exception
```

   But we can generate the same Stream repitively with a **supplier**.

```
1 Supplier<Stream<String>> myStreams =
2     () -> IntStream.range(0, 50).boxed().filter(x -> x>10);
3 streamSupplier.get().anyMatch(s -> s < 40);   // ok
4 streamSupplier.get().forEach(System.out::println);  // ok
```

Optional<T>

1. Purpose: prevent **NPE** elegantly

```
1 if(obj1 != null){
2     ... // handling
3 }
```

   it's a type-level solution for representing **optional values** instead of null references

2. What is `optional<T>`?

   A container object which may or may not contain a non-null value (safe alternative for "object or null")

3. creating `optional<T>` values

```
1  // empty
2  public static <T> Optional<T> empty();
3  // Returns an empty Optional instance. No value is present for this
   Optional.
4
5  // of
6  public static <T> Optional<T> of(T value);
7  // Returns an Optional describing the given non-null value.
8  // throw NPE if value is null
9  // If you want to let the problem arise as soon as possible, of is more
   recommended then ofNullable
10 /* For example,
11 if you're sure this value in your project will never be null, just use of.
   When the exeception occur, you should check your code
```

```
*/

// ofNullable
public static <T> Optional<T> ofNullable(T value)
// Returns an Optional describing the given value, if non-null, otherwise
returns an empty Optional.
```

4. some methods in Optional<T>

```
// get
public T get();
// If a value is present, returns the value, otherwise throws
NoSuchElementException.

// judge whether have value
public boolean isPresent();
// If a value is present, returns true, otherwise false.
public boolean isEmpty();
// If a value is not present, returns true, otherwise false.

// deal with the empty case
public T orElse(T other);
// If a value is present, returns the value, otherwise returns other.
public T orElseThrow();
// If a value is present, returns the value, otherwise throws
NoSuchElementException.
```

5. `flatMap` vs `map`

   if `T` can easily get `U` and want `U` to be wrapped automatically, use `map`.

   if `T` can easily get `option<U>` , use `flatMap`.

```
// map
public <U> Optional<U> map(Function<? super T,? extends U> mapper);
// If a value is present, returns an Optional describing (as if by
ofNullable(T)) the result of applying the given mapping function to the
value, otherwise returns an empty Optional.

// flatMap
public <U> Optional<U> flatMap(Function<? super T,? extends Optional<?
extends U>> mapper);
// If a value is present, returns the result of applying the given Optional-
bearing mapping function to the value, otherwise returns an empty Optional.
```