Lab 4. Using Functional Interfaces and Streams

Authors: Yao Zhao, Yida Tao

1. Definitions and Usages of Common Functional Interfaces

`Predicate<T>`

The `java.util.function.Predicate<T>` interface defines an abstract method named `test` that accepts an object of generic type `T` and returns a boolean. You might want to use this interface when you need to represent a boolean expression that uses an object of type `T`.

```
@FunctionalInterface
public interface Predicate<T>{
    boolean test(T t);
}
```

For example, you can define a lambda that accepts String objects, and only returns the nonempty strings.

```
public static <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> results = new ArrayList<>();
    for (T s : list) {
        if (p.test(s)) {
            results.add(s);
        }
    }
    return results;
}

public static void main(String[] args) {
    List<String> listOfStrings = new ArrayList<>();
    listOfStrings.add("");
    listOfStrings.add("abc");
    listOfStrings.add("\n");
    listOfStrings.add("e");
    Predicate<String> nonEmptyStringPredicate = (s) -> !s.isEmpty();
    List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
    System.out.println(nonEmpty);
}
```

`Consumer<T>`

The `java.util.function.Consumer<T>` interface defines an abstract method named `accept` that takes an object of generic type `T` and returns no result (`void`). You might use this interface when you need to access an object of type `T` and perform some operations on it.

```
@FunctionalInterface
public interface Consumer<T>{
    void accept(T t);
}
```

For example, you can use it to create a method `forEach`, which takes a list of `Integers` and applies an operation on each element of that list. In the following listing you use this `forEach` method combined with a lambda to print all the elements of the list.

```
public static <T> void forEach(List<T> list, Consumer<T> c) {
    for (T s : list) {
        c.accept(s);
    }
}

......

forEach(nonEmpty, System.out::print);
```

`Function<T, R>`

The `java.util.function.Function<T, R>` interface defines an abstract method named `apply` that takes an object of generic type `T` as input and returns an object of generic type `R`. You might use this interface when you need to define a lambda that maps information from an input object to an output (for example, extracting the weight of an apple or mapping a string to its length).

```
public interface Function<T, R> {
    R apply(T t);
}
```

In the listing that follows, we show how you can use it to create a method `map` to transform a list of Strings into a list of Integers containing the length of each String.

```
public static <T, R> List<R> map(List<T> list, Function<T,R> f){
    List<R> results = new ArrayList<>();
    for(T s : list){
        results.add(f.apply(s));
    }
    return results;
}

public static void main(String[] args) {
    ......
    List<Integer> listStrLen2 = map(nonEmpty, String::length);
```

```
    forEach(listStrLen2, (s)-> System.out.print(s + ","));
}
```

You may learn `UnaryOperator<T>` and `BinaryOperator<T>` by yourselves. Refer to `FunctionalInterfaceExample.java` for the sample code.

2. Using Streams

2.1. Creating streams

Streams can be created from various data sources, especially collections. `StreamCreation.java` shows a lot of examples how Streams be created.

2.2. How stream operations are processed

An important characteristic of intermediate operations is laziness. Look at this sample where a terminal operation is missing:

```
Stream.of("CS", "209", "A").filter(s -> {
    System.out.println("filter: " + s);
    return true;
});
```

When executing this code snippet, nothing is printed to the console. That is because intermediate operations will only be executed when a terminal operation is present. For more examples, please refer to `StreamProcessingOrder.java`.

2.3. Reusing Streams

Streams cannot be reused. As soon as you call any terminal operation the stream is closed:

```
Stream<String> stream = Stream.of("CS", "209", "A").filter(s ->
s.startsWith("C"));

stream.anyMatch(s -> true); // ok
stream.noneMatch(s -> true); // exception
```

Calling `noneMatch` after `anyMatch` on the same stream results in the following exception:

```
Exception in thread "main" java.lang.IllegalStateException: stream has already
been operated upon or closed
  at java.util.stream.AbstractPipeline.evaluate (AbstractPipeline.java:229)
  at java.util.stream.ReferencePipeline.noneMatch(ReferencePipeline.java:459)
  at StreamProcessingOrder.main(StreamProcessingOrder.java:95)
```

To overcome this limitation, we have to to create a new stream chain for every terminal operation we want to execute, e.g., we could create a stream supplier to construct a new stream with all intermediate operations already set up:

```
Supplier<Stream<String>> streamSupplier =
        () -> Stream.of("CS", "209", "A")
                .filter(s -> s.startsWith("C"));

streamSupplier.get().anyMatch(s -> true);   // ok
streamSupplier.get().noneMatch(s -> true);  // ok
```

Check `StreamReuse.java` for sample code.

2.4. Stream Reduction

The JDK contains many terminal operations (such as average, sum, min, max, and count) that return one value by combining the contents of a stream. These operations are called *reduction* operations.

The JDK also contains reduction operations that return a collection instead of a single value. Many reduction operations perform a specific task, such as finding the average of values or grouping elements into categories.

The JDK provides you with the general-purpose reduction operations `reduce` and `collect`. Please refer to this official guide for examples of these two operations.

We also provide a `StreamReduction.java` to demonstrate common usages for stream reduction.