5 - Java Stream API

< Previous | Next >

- `public static <T> Collector<T,?,Long> counting()` Returns a Collector accepting elements of type T that counts the number of input elements collected as a long. If no elements are present, the result is 0.
- Stream.count() returns long.
- `Arrays.asList("a", "b").parallelStream().reduce("_", (String::concat));` Since we are creating a parallel stream, it is possible for both the elements of the stream to be processed by two different threads. In this case, the identity argument will be used to reduce both the elements. Thus, it will print `_a_b`. It is also possible that the result of the first reduction (`_a`) is reduced further using the second element (`b`). In this case, it will print _ab. Even though the elements may be processed out of order individualy in different threads, the final output will be produced by joining the individual reduction results in the same order. Thus, the output can never have b before a.
- The range method includes the starting number but not the ending number. The rangeClosed method includes the ending number also.
- The average method of all numeric streams (i.e. IntStream, LongStream, and DoubleStream) returns an OptionalDouble and not a double. It never returns a null. (If there are no elements in the stream, it returns OptionalDouble.empty but not 0). Note that this is unlike the sum method which always returns a primitive value of the same type as the type of the stream (i.e. int, long, or double).
- OptionalDouble's toString method returns a String of the form `OptionalDouble[<double value>]`.
- findAny should return the first element. However, findAny is deliberately designed to be non-deterministic. Its API specifically says that it may return any element from the stream. If you want to select the first element, you should use findFirst.
- `findAny()` returns Optional object.
- A reduction operation (also called a fold) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation, such as finding the sum or maximum of a set of numbers, or accumulating elements into a list. The streams classes have multiple forms of general reduction operations, called reduce() and collect(), as well as multiple specialized reduction forms such as sum(), max(), or count().
- min and max are valid reduction operations. The Stream version of these methods take a Comparator as an argument, while the versions in specialized streams such as IntStream and DoubleStream do not take any argument.
- getAsDouble will throw a java.util.NoSuchElementException if you call it on a OptionalDouble containing OptionalDouble.empty. To avoid this problem, instead of getAsDouble, you should use orElse(0.0). That way, if the OptionalDouble is empty, it will return 0.0.
- `IntStream.of()` does not take a List as argument. It takes either an int or a varargs parameter of type int.
- `Comparator<Book> c1 = (b1, b2)->b1.getGenre().compareTo(b2.getGenre());` is correct! Comparator is a functional interface and the lambda expression captures it correctly.
- Manipulating a stream doesn't manipulate the backing source of the stream. Here, when you chain the sorted method to a stream, it returns a reference to a Stream that appears sorted. The original List which was used to create the stream will remain as it is. If you want to sort a List permanently, you should use one of the Collections.sort methods.
- The Collectors.toMap method uses two functions to get two values from each element of the stream. The value returned by the first function is used as a key and the value returned by the second function is used as a value to build the resulting Map.
- The Collector created by Collectors.toMap throws java.lang.IllegalStateException if an attempt is made to store a key that already exists in the Map. If you want to collect items in a Map and if you expect duplicate entries in the source, you should use Collectors.toMap(Function, Function, BinaryOperator) method. The third parameter is used to merge the duplicate entries to produce one entry.

`Collectors.toMap(b->b.getTitle(), b->b.getPrice(), (v1, v2)->v1+v2`

- The forEach method of a Map requires a BiConsumer. This function is invoked for each entry, that is each key-value pair, in the map. The first argument of this function is the key and the second is the value.
- The Collectors.partitioningBy method takes a Predicate and returns Collector that distributes the elements of the stream into two groups - one containing elements for which the Predicate returns true, and another containing elements for which the Predicate returns false. The return type is a Map containing two keys - true and false and the values are Lists of the elements.
- Only final or effective final local variables can be used in a lambda expression. Effectively final means that even though it is not declared as final, it is not assigned any value anywhere else after the first assignment. So if a variable is assigned a new value INSIDE a lambda it is NOT effectively final.
- Following method is part of the Map interface:
`public V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction` - If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is null. This method may be of use when combining multiple mapped values for a key. For example, to either create or append a String msg to a value mapping: `map.merge(key, msg, String::concat)` If the function returns null the mapping is removed. If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.
- forEach expects a Consumer object as argument. Not a UnaryOperator.
- The replaceAll method replaces each element of this list with the result of applying the operator to that element. str->str.toUpperCase() is a valid lambda expression that captures UnaryOperator function.
- `Collectors.joining()` returns a Collector that operates on a Stream containing CharSequences (String extends CharSequence) and joins all the elements into one big String. There is no Collectors.joining method that takes any functional interface as argument. There is a Collectors.joining method that takes a String as argument, used as delimiter.
- `letters.forEach(letter->letter.toUpperCase());` The first statement does convert each element to upper case. However, the new upper case value does not get back in the the list. It is lost.
- flatMap is used when each element of a given stream can itself generate a Stream of objects. The purpose of this method is to extract the elements of each of those individual streams and return a stream that contains all those elements.
- Remember that Stream has only two overloaded collect methods - one that takes a Collector as an argument and another one that takes a Supplier, BiConsumer as accumulator, and BiConsumer as combiner.
- `Collectors.mapping()` method requires two arguments - the first argument must be a Function that maps one element type into another, and the second argument must be an appropriate Collector in which you can hold the result.
- Only non-terminal operations such as filter, peek, map, distinct, sorted, and skip are lazy. All terminal operations such as forEach, reduce, collect, count, min, max, allMatch, findAny are eager.
- Streams support several aggregate operations such as forEach, count, average, and sum.

 # java.util.Optional:

1. Optional has a static method named of(T t) that returns an Optional object containing the value passed as argument. It will throw NullPointerException if you pass null. If you want to avoid NullPointerException, you should use Optional.ofNullable(T t) method. This will return Optional.empty if you pass null.
2. You cannot change the contents of Optional object after creation. Optional does not have a set method. Therefore, grade.of, although technically correct, will not actually change the Optional object referred to by grade. It will return a new Optional object containing the passed argument.
3. The orElse method returns the actual object contained inside the Optional or the argument passed to this method if the Optional is empty. It does not return an Optional object. Therefore, `print(grade1.orElse("UNKNOWN"))` will print `UNKNOWN` and not `Optional[UNKNOWN]`.
4. isPresent() returns true if the Optional contains a value, false otherwise.

5. ifPresent(Consumer) executes the Consumer object with the value if the Optional contains a value. Note that it is the value contained in the Optional that is passed to the Consumer and not the Optional itself.

- Optional.of method throws NullPointerException if you try to create an Optional with a null value. If you expect the argument to be null, you should use Optional.ofNullable method, which returns an empty Optional if the argument is null.
- Optional.of takes only one argument!
- Optional's orElseGet method takes a java.util.function.Supplier function as an argument and invokes that function to get a value if the Optional itself is empty. Just like the orElse method, this method does not throw any exception even if the Supplier returns null. It does, however, throw a NullPointerException if the Optional is empty and the supplier function itself is null.

 # `Collectors.groupingBy()` :

```
    List<Course> s1 = Arrays.asList(
            new Course("OCAJP", "Java"),
            new Course("OCPJP", "Java"),
            new Course("C#", "C#"),
            new Course("OCEJPA", "Java"));

    s1.stream()
            .collect(Collectors.groupingBy(c->c.getCategory()))
            .forEach((m, n)->System.out.println(n));

    // output:
    // [C# C#]
    // [OCAJP Java, OCPJP Java, OCEJPA Java]
```

1. `Collectors.groupingBy(Function<? super T,? extends K> classifier)` returns a Collector that groups elements of a Stream into multiple groups. Elements are grouped by the value returned by applying a classifier function on an element.
2. It is important to understand that the return type of the collect method depends on the Collector that is passed as an argument. In this case, the return type would be Map> because that is the type specified in the Collector returned by the groupingBy method.
3. Java 8 has added a default forEach method in Map interface. **This method takes a BiConsumer function** object and applies this function to each key-value pair of the Map. In this case, m is the key and n is the value.
4. The given code provides a trivial lambda expression for BiConsumer that just prints the second parameter, which happens to be the value part of of the key-value pair of the Map.
5. The value is actually an object of type List, which is printed in the output. Since there are two groups, two lists are printed. First list has only one Course element and the second list has three.

< Previous | Next >