CHAPTER 13

■ ■ ■

Streams

QUESTIONS AND EXERCISES

1. What are streams and aggregate operations on streams?

   Answer:

   A stream is a sequence of data elements supporting sequential and parallel aggregate operations. An *aggregate operation* computes a single value from a collection of values.

2. How do streams differ from collections?

   Answer:

   Collections in Java focus on data storage and access to the data whereas streams focus on computations on data. Streams do not have storage whereas collet ion do.

3. Fill in the blanks:

   a. Collections have storage whereas streams have ____ storage.

   b. Collections support external iteration whereas streams support _____ iteration.

   c. Collections support imperative programming whereas streams support _____ programming.

   d. Collections support a finite number of elements whereas streams support an _____ number of elements.

   e. Streams support sequential and _____ processing of its elements.

f. A stream does not start pulling elements from its data source until a _____ operation is called on the stream.

g. Once a terminal operation is called on a stream, the stream _____ be reused.

Answer:

a. Collections have storage whereas streams have no storage.

b. Collections support external iteration whereas streams support internal iteration.

c. Collections support imperative programming whereas streams support functional programming.

d. Collections support a finite number of elements whereas streams support an infinite number of elements.

e. Streams support sequential and parallel processing of its elements.

f. A stream does not start pulling elements from its data source until a terminal operation is called on the stream.

g. Once a terminal operation is called on a stream, the stream cannot be reused.

4. Describe the difference between intermediate and terminal operations on streams.

Answer:

Each intermediate operation takes elements from an input stream and transforms the elements to produce an output stream. The terminal operation takes inputs from a stream and produces the result. An intermediate operation on a stream does not process the elements of the stream until a terminal operation is performed on the stream.

5. Create a `Stream<Integer>` of all integers from 10 to 30 and compute the sum of all integers in the list.

Answer:

```
// Solution #1
```

```
 Stream<Integer> numbers = Stream.iterate(10, n -> n <= 30, n -> n + 1);
 int sum = numbers.reduce(0, Integer::sum);
 System.out.println(sum);

// Solution #2
Stream<Integer> numbers = IntStream.rangeClosed(10, 30).boxed();
int sum = numbers.reduce(0, Integer::sum);
System.out.println(sum);
```

6. Complete the following snippet of code, which computes the sum of characters in a list of names using a stream.

```
List<String> names = List.of("Mo", "Jeff", "Li", "Dola");
int sum = names.stream()
               ./* your code goes here*/;
System.out.println("Total characters: " + sum);
```

   The expected output is as follows:

```
Total characters: 12
```

   Solution:

```
List<String> names = List.of("Mo", "Jeff", "Li", "Dola");
int sum = names.stream()
              .collect(Collectors.summingInt(String::length));
System.out.println("Total characters: " + sum);
```

7. Complete the following snippet of code, which creates two empty `Stream<String>`s. You are supposed to use different methods of the `Stream` interface to complete the code.

```
Stream<String> noNames1 = Stream./* Your code goes here*/;
Stream<String> noNames2 = Stream./* Your code goes here*/;
```

   Solution:

```
Stream<String> noNames1 = Stream.empty();
Stream<String> noNames2 = Stream.of();
```

8. What method of the `Stream` interface is used to limit the number of elements in a stream to a specified size?

   Answer:

The `limit(long maxSize)` method.

9. What method of the `Stream` interface is used to skip a specified number of elements in a stream?

Answer:

The `skip(long n)` method.

10. Describe the characteristics of the stream produced by the following snippet of code:

```
Stream<Integer> stream = Stream.generate(() -> 1969);
```

Answer:

The code generates an infinite stream of number 1969.

11. What is the use of the instances of the `Optional<T>` class?

Answer:

An `Optional<T>` is a wrapper for a non-null value that may or may not contain a non-null value. Methods that may return nothing should return an `Optional` instead of `null`, to handle `NullPointerException` gracefully.

12. Complete the following snippet of code, which is supposed to print the names of people along with the number of characters in the names in the non-empty `Optional`s in the list:

```
List<Optional<String>> names = List.of(Optional.of("Ken"),
                                       Optional.empty(),
                                       Optional.of("Li"),
                                       Optional.empty(),
                                       Optional.of("Toto"));

names.stream()
```

```
        .flatMap(/* Your code goes here */)
        .forEach(/* Your code goes here */);
```

The expected output is as follows:

```
Ken: 3
Li: 2
Toto: 4
```

Solution:

```
List<Optional<String>> names = List.of(Optional.of("Ken"),
        Optional.empty(),
        Optional.of("Li"),
        Optional.empty(),
        Optional.of("Toto"));

names.stream()
     .flatMap(Optional::stream)
     .forEach(e -> System.out.println(e + ": " + e.length()));
```

13. What is the use of the `peek()` method in the `Stream` interface?

Answer:

The `peek()` method can be used to look at the elements of the stream as they pass through the pipeline (for debugging purposes).

14. What is the use of the `map()` and `flatMap()` methods in the `Stream` interface?

Answer:

The `map()` method returns a stream consisting of the results of applying the specified function to the elements in this stream. The `flatMap()` method returns a stream consisting of the results of applying the specified function to the elements in this stream; the function produces a stream for each input element and the output streams are flattened.

15. Compare the filter and map operations on a stream with respect to the type of elements and number of elements in the input and output streams of these operations.

Answer:

Both are intermediate operations.

The `filter()` method of the `Stream<T>` interface takes a `Predicate<T>` as argument and returns a `Stream<T>` with elements of the original stream for which the specified `Predicate` returns true. The type of output elements is the same as the input elements. The number of output elements may be the same or fewer than the number of input elements.

The `map()` method of the `Stream<T>` interface takes a `Function` as argument. Each element in the stream is passed to the `Function` and a new stream is generated containing the returned values from the `Function`. The type of output elements is typically different than that of the input elements; however, it may be the same. The number of output elements is the same as the number of input elements.

16. What is a reduction operation on a stream? Name three commonly used reduction operations on streams.

Answer:

The reduce or reduction operation combines all elements of a stream to produce a single value by applying a combining function repeatedly. Commonly used reduce operations are `sum()`, `count()`, and max().

17. Write the logic to compute the sum of all integers in the following array using a parallel stream and the `reduce()` method of the `Stream` interface.

```
int[] nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Answer:

```
int[] nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int sum = Arrays.stream(nums).boxed()
                .parallel()
                .reduce(0, Integer::sum);

System.out.println(sum);
```

In this snippet of code, `Arrays.stream(nums).boxed()` produces a `Stream<Integer>`. Note that the question asks you to use the reduce() method of the Stream interface and this code accomplished this.

The following snippet of code uses `IntStream` and its `sum()` method to achieve the same:

```
int[] nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int sum = Arrays.stream(nums)
                .parallel()
                .sum();

System.out.println(sum);
```

The following snippet of code achieves the same result using an `IntStream` and its `reduce()` method:

```
int[] nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int sum = Arrays.stream(nums)
                .parallel()
                .reduce(0, Integer::sum);
System.out.println(sum);
```

18. Complete the following snippet of code to print the unique non-null values in a map:

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(2, "One");
map.put(3, null);
map.put(4, "Two");

map.entrySet()
   .stream()
   .flatMap(/* Your code goes here */)
   ./* Your code goes here */
   .forEach(System.out::println);
```

The expected output is as follows:

```
One
Two
```

Solution:

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(2, "One");
map.put(3, null);
map.put(4, "Two");

map.values()
```

```
        .stream()
        .map(Optional::ofNullable)
        .flatMap(Optional::stream)
        .distinct()
        .forEach(System.out::println);
```

19. Complete the missing pieces in the following snippet of code, which is supposed to count the number of even and odd integers in a list of integers:

```
List<Integer> list = List.of(10, 19, 20, 40, 45, 50);
Map<String,Long> oddEvenCounts = list.stream()
        .map(/* Your code goes here */)
        .collect(/* Your code goes here */);

System.out.println(oddEvenCounts);
```

The expected output is as follows:

```
{Even=4, Odd=2}
```

Solution:

```
List<Integer> list = List.of(10, 19, 20, 40, 45, 50);
Map<String,Long> oddEvenCounts = list.stream()
        .map(t -> t % 2 == 0 ? entry("Even", 1) : entry("Odd", 1))
        .collect(Collectors.groupingBy(Entry::getKey, Collectors.counting()));

System.out.println(oddEvenCounts);
```

20. The following snippet of code is supposed to print a sorted list of odd integers in the list, which are separated by colons. Complete the missing pieces of the code.

```
List<Integer> list = List.of(5, 1, 2, 7, 3, 4, 8);
String str = list.stream()
                 ./* Multiple method calls go here */;

System.out.println(str);
```

The expected output is as follows:

```
1:3:5:7
```

Solution:

```
List<Integer> list = List.of(5, 1, 2, 7, 3, 4, 8);
String str = list.stream()
        .map(t -> t % 2 == 1 ? Optional.of(t) : Optional.empty())
        .flatMap(Optional::stream)
        .sorted()
        .map(String::valueOf)
        .collect(Collectors.joining(":"));

System.out.println(str);
```
