Correction Java Stream API - 0

B - true

stream => {1, 2, 3, 4, 5, … }. It is an infinite stream.

Predicate ‘i -> i > 1’ returns true for any Integer greater than 1.

As 2 > 1, so true is printed and operation is terminated. Code doesn’t run infinitely.

NOTE: ‘stream.allMatch(i -> i > 1)’ returns false as 1st element of the stream (1) returns false for the predicate and ‘stream.noneMatch(i -> i > 1)’ returns false as 2nd element of the stream (2) returns true for the predicate.

Correction Java Stream API - 1

C - 1.1.

findFirst() is terminal operation.

list.stream() => [-80, 100, -40, 25, 200].

filter(predicate) is executed for each element until just one element passes the test. Because findFirst() will terminate the operation on finding first matching element.

NOTE: a new instance of Predicate is used, hence every time ctr will be initialize to 1.

For -80, Output is ‘1.’ but predicate returns false, hence findFirst() doesn’t terminate the operation.

For 100, ‘1.’ is appended to previous output, so on console you will see ‘1.1.’ and predicate returns true, hence findFirst() finds an element and terminates the operation.

Final output is: ‘1.1.’

Correction Java Stream API - 2

C - An instance of MyException is thrown at runtime

orElseThrow throws the instance of provided Exception if optional is empty.

In this case optional is an empty OptionalDouble, hence an instance of MyException is thrown at runtime.

Correction Java Stream API - 3

B - System.out.println(stream.reduce(1, (i, j) -> i* j));

C - System.out.println(stream.reduce(res, (i, j) -> i * j));

Integer class doesn't have 'multiply' method, hence options containing 'Integer::multiply' will cause compilation failure.

To understand, 'stream.reduce(1, (i, j) -> i * j)' can be written as:

```
int result = 1;
for (int element : stream) {
    result = op.applyAsInt(result, element);
}
return result;
```

Above code is just for understanding purpose, you can't iterate a stream using given loop.

Note: 'op' in above code is of IntBinaryOperator and target type of given lambda expression.

Check IntPipeline class which implements IntStream for the details of reduce method.

If 1st argument of reduce is 0, then overall result will be zero.

'stream.reduce(1, (i, j) -> i * j)' and 'stream.reduce(res, (i, j) -> i * j)' are correct options.

Correction Java Stream API - 4

B - 4

Set doesn’t allow duplicates, which means generated set will have 4 elements [“java”, “python”, “c”, “c++] and therefore set.size() will return 4.

Correction Java Stream API - 5

A -

```
Point(0, 0)
Point(−1, −1)
```

‘Point::filter’ is an example of “Reference to an Instance Method of an Arbitrary Object of a Particular Type”. Equivalent lambda expression is: ‘(Point p) -> p.filter()’.

As filter(…) method accepts Predicate<? super Point> instance as an argument, hence given method reference syntax is the correct implementation of the Predicate. Line n1 compiles successfully.

Result of filtration is the Stream instance containing Point objects whose x == y. Therefore, this stream contains Point(0, 0) and Point(-1, -1).

forEach(System.out::println) prints Point(0, 0) and Point(-1, -1) on to the console.

Correction Java Stream API - 6

A -

```
false
true
true
```

Method signatures:

boolean anyMatch(Predicate<? super T>) : Returns true if any of the stream element matches the given Predicate. If stream is empty, it returns false and predicate is not evaluated.

boolean allMatch(Predicate<? super T>) : Returns true if all the stream elements match the given Predicate. If stream is empty, it returns true and predicate is not evaluated.

boolean noneMatch(Predicate<? super T>) : Returns true if none of the stream element matches the given Predicate. If stream is empty, it returns true and predicate is not evaluated.

In this case, as stream is empty anyMatch returns false, whereas allMatch and noneMatch both returns true.

Correction Java Stream API - 7

A - 98

IntStream.rangeClosed(int start, int end) => Returns a sequential stream from start to end, both inclusive and with a step of 1.

IntStream.map(IntUnaryOperator) => Returns a stream consisting of the results of applying the given function to the elements of this stream.

IntStream.rangeClosed(1,3) => [1,2,3].

map(i -> i * i) => [1,4,9].

map(i -> i * i) => [1,16,81].

sum() => 1+16+81 = 98.

Correction Java Stream API - 8

B -

```
Kathy Sierra
Udayan Khattry
```

books –> [{Head First Java,Kathy Sierra,24.5}, {OCP,Udayan Khattry,20.99}, {OCA,Udayan Khattry,14.99}]. Ordered by insertion order.

books.stream() returns Stream type: [{Head First Java,Kathy Sierra,24.5}, {OCP,Udayan Khattry,20.99}, {OCA,Udayan Khattry,14.99}].

books.stream().collect(Collectors.groupingBy(Book::getAuthor)) returns a Map<String, List> type, key is the author name and value is the List of book objects.

map –> [{Kathy Sierra, {Head First Java,Kathy Sierra,24.5}}, {Udayan Khattry, {{OCP,Udayan Khattry,20.99}, {OCA,Udayan Khattry,14.99}}}].

forEach method accepts a BiConsumer<String, List> , so first parameter of accept method is key and 2nd parameter is value. So in the given lambda expression ‘a’ is key and ‘b’ is value.

System.out.println(a) prints keys(author names) to the console.

Correction Java Stream API - 9

B - Optional.empty

ofNullable method creates an empty Optional object if passed argument is null.

Optional.empty is printed on to the console for empty Optional.

Correction Java Stream API - 10

C - Compilation error

There is no stream() method available in Map interface and hence map.stream() causes compilation error.

Though you can first get either entrySet or keySet or values and then invoke stream() method.

For example, below code prints all the key value pairs available in the map:

map.entrySet().stream().forEach(x -> System.out.println(x.getKey() + “:” + x.getValue()));

Correction Java Stream API - 11

D - Optional[2nd3rd]

filter method filters all the strings ending with “d”.

‘stream.reduce((s1, s2) -> s1 + s2)’ returns ‘Optional’ type, whereas ‘stream.reduce(““, (s1, s2) -> s1 + s2)’ returns ‘String’.

Correction Java Stream API - 12

A -

```
Rope [red, 5]
Rope [Red, 10]
Rope [RED, 7]
```

‘new Rope.RedRopeFilter()::filter’ is an example of “Reference to an Instance Method of a Particular Object”.

Equivalent lambda expression is: ‘(Rope r) -> new Rope.RedRopeFilter().filter(r)’.

As filter(…) method accepts Predicate<? super Point> instance as an argument, hence given method reference syntax is the correct implementation of the Predicate. Line n1 compiles successfully.

Result of filtration is the Stream instance containing Rope objects of ‘red’ color. Please note string ‘red’ can be in any case(upper, lower or mixed) Therefore, this stream contains Rope [red, 5], Rope [Red, 10] and Rope [RED, 7].

forEach(System.out::println) prints Rope [red, 5], Rope [Red, 10] and Rope [RED, 7]) on to the console.

If ‘filter(Rope)’ is declared as static, then to achieve same output, you will have to change the method reference syntax to: ‘filter(Rope.RedRopeFilter::filter)’.

Correction Java Stream API - 13

C - Compilation error

Generic Stream interface has following methods:

Optional min(Comparator<? super T> comparator);

Optional max(Comparator<? super T> comparator);

Primitive Stream interfaces (IntStream, LongStream & DoubleStream) has methods min(), max(), sum(), average() and summaryStatistics().

In this case, as stream is a generic interface, hence stream.sum() causes compilation error.

Correction Java Stream API - 14

C - [Ready to stop, Slow Down]

Though it looks like very complex code, but it is simple.

TrafficLight class stores the enum Color and text message to be displayed with the color.

tl1 –> {GREEN, “Go”}.

tl2 –> {GREEN, “Go Now!”}.

tl3 –> {YELLOW, “Ready to stop”}.

tl4 –> {YELLOW, “Slow Down”}.

tl5 –> {RED, “Stop”}.

list.stream() –> [{GREEN, “Go”}, {GREEN, “Go Now!”}, {YELLOW, “Ready to stop”}, {YELLOW, “Slow Down”}, {RED, “Stop”}].

Collectors.groupingBy(TrafficLight::getColor, Collectors.mapping(TrafficLight::getMsg, Collectors.toList())); => Group above stream on the basis of Color (key is enum constant Color: RED, GREEN, YELLOW).

So, intermediate object returned by ‘Collectors.groupingBy(TrafficLight::getColor)’ is of Map<Color, List> type. But the 2nd argument, ‘Collectors.mapping(TrafficLight::getMsg, Collectors.toList())’ passed to groupingBy(…) method converts List to List and returns ‘Map<Color, List>’

map –> {GREEN=[Go, Go Now!], YELLOW=[Ready to stop, Slow Down], RED=[Stop]}.

System.out.println(map.get(Color.YELLOW)); prints [Ready to stop, Slow Down] on to the console.

Correction Java Stream API - 15

C - Compilation error

MyException is a checked exception, so ‘handle or declare’ rule must be followed.

‘orElseThrow(MyException::new)’ can throw checked exception at runtime, so it must be surrounded by a try-catch block or main method should declare proper throws clause.

Correction Java Stream API - 16

E - 24

IntStream.rangeClosed(1, 4); => [1, 2, 3, 4]

To understand, ‘stream.reduce(res++, (i, j) -> i * j)’ can be somewhat written as:

```
int result = res++;
for (int element : stream) {
    result = accumulator.applyAsInt(result, element);
}
return result;
```

Above code is just for understanding purpose, you can’t iterate a stream using given loop.

result will be initialized to 1 and after that res will be incremented to 2. But value of ‘result’ is used and not ‘res’.

Hence output will be result of ‘1 * 1 * 2 * 3 * 4’, which is 24.

Correction Java Stream API - 17

E - Nothing is printed on to the console

Streams are lazily evaluated and as sorted() is an intermediate operation, hence stream is not evaluated and you don’t get any output on to the console.

Correction Java Stream API - 18

C - []

Rest of the code is very simple, let us concentrate on partitioning code.

Collectors.partitioningBy(s -> s.equals(“OCA”)) => s in this lambda expression is of Certification type and not String type.

This means predicate ‘s -> s.equals(“OCA”)’ will return false for “OCA”. None of the certification object will return true and hence no element will be stored against ‘true’.

[] will be printed in the output.

Correct predicate will be: ‘s -> s.getTest().equals(“OCA”)’.

For above predicate, output for ‘System.out.println(map.get(true));’ will be:

[{S001, OCA, 87}, {S002, OCA, 82}, {S003, OCA, 60}, {S004, OCA, 88}]

Correction Java Stream API - 19

D - Above code terminates successfully after printing text other than “YES” on to the console

MyString class doesn’t override toString() method, hence when instance of MyString class passed to System.out.print(…) method, it prints @.

list.stream() returns an object of Stream and this stream object has 3 instances of MyString class.

map(s -> s) returns the Stream object containing exactly same 3 instances of MyString class.

forEach(System.out::print) prints text in the format @ for all the 3 instances.

Text in above format is printed for the 3 elements of the stream and program terminates successfully.

Hence correct option is: ‘Above code terminates successfully after printing text other than “YES” on to the console’

To print “YES” on to the console, change the last statement to:
list.stream().map(s -> s.str).forEach(System.out::print);

‘s’ represents the instance of MyString class and as MyString class is defined within the same package, hence instance variable ‘str’ can be accessed using dot operator.

Correction Java Stream API - 20

D - -1

Stream.of() creates an empty stream.

stream.findFirst(); => returns an empty Optional.

Hence, orElse method is executed and prints -1 on to the console.

Correction Java Stream API - 21

B - Line 1 and Line 2 print same output

In Comparator.comparing(a -> a), keyExtractor is not doing anything special, it just implements Comparator to sort integers in ascending order.

Integer::compareTo is a method reference syntax for the Comparator to sort integers in ascending order.

NOTE: Comparator implementations must return following:

-1 (if 1st argument is less than 2nd argument),

0 (if both arguments are equal) and

1 (if 1st argument is greater than 2nd argument).

Integer::max accepts 2 arguments and returns int value but in this case as all the 3 elements are positive, so value will always be positive.

Line 3 will print different output as it will not sort the list properly.

Correction Java Stream API - 22

B - It can print any number from the stream.

D - It will never print -1 on to the console.

findAny() may return any element from the stream and as stream is not parallel, it will most likely return first element from the sorted stream, which is -10. But this is not the guaranteed result.

As this stream has 3 elements, hence -1 will never get printed on to the console.

Correction Java Stream API - 23

A - Program executes successfully but nothing is printed on to the console

Method signature for anyMatch method:

boolean anyMatch(Predicate<? super T>) : Returns true if any of the stream element matches the given Predicate.

If stream is empty, it returns false and predicate is not evaluated.

As given stream is empty, hence predicate is not evaluated and nothing is printed on to the console.

Correction Java Stream API - 24

D - Line 8 throws NullPointerException

Optional.of(null); throws NullPointerException if null arguments is passes.

You can use ‘Optional.ofNullable(null);’ to create an empty optional.

Correction Java Stream API - 25

C - OptionalDouble

Only 3 primitive variants available: OptionalDouble, OptionalInt and OptionalLong.

Correction Java Stream API - 26

B - [c, c++, java, python]

‘TreeSet::new’ is same as ‘() -> new TreeSet()’

TreeSet contains unique elements and in sorted order, in this case natural order.

Output will always be: [c, c++, java, python]

Correction Java Stream API - 27

D - flatMapToInt

s -> s.chars() is of IntStream type as chars() method returns instance of IntStream type.

All 4 are valid method names but each specify different parameters.

Only faltMapToInt can accept argument of IntStream type.

Correction Java Stream API - 28

A - list.stream().sorted(Comparator.comparing(f -> f.countryOfOrigin, Fruit::comp)).forEach(System.out::println);

B - list.stream().sorted(new Fruit().reversed()).forEach(System.out::println);

As Comparable and Comparator are interfaces, so Fruit class can implement both the interfaces.

Given code compiles successfully.

compareTo(Fruit) method of Comparable interface will help to sort in ascending order of Fruit's name and compare(Fruit, Fruit) method of Comparator interface will help to sort in ascending order of Fruit's country of origin.

By looking at the expected output it is clear that output is arranged in descending order of Fruit's country of origin. This means Comparator needs to be used in reversed order. Out of the given options 'list.stream().sorted(new Fruit().reversed()).forEach(System.out::println);' will display the expected output.

'list.stream().sorted()' sorts Fruit instances on the basis of ascending order of their names, not the correct option.

'list.stream().sorted(new Fruit())' sorts Fruit instances on the basis of ascending order of their country of origin, not the correct option.

'list.stream().sorted(Comparator.comparing(f -> f.countryOfOrigin, Fruit::comp))': keyExtractor is accepting Fruit object and returning countryOfOrigin, which is of String type. The 2nd argument passed is Fruit::comp, which is keyComparator and it sorts the Fruit objects in descending order of the key (courntryOfOrigin) in this case. Hence this is also a correct option.

Correction Java Stream API - 29

A - Replace stream.sorted() with stream.sorted((s1,s2) -> s1.length() - s2.length())

Given code sorts the stream in natural order (a appears before b, b appears before c and so on).

To get the expected output, stream should be sorted in ascending order of length of the string.

Replacing ‘stream.sorted()’ with ‘stream.sorted((s1,s2) -> s1.length() - s2.length())’ will do the trick.

Correction Java Stream API - 30

D - Good Morning!

optional –> [null]

optional.isPresent() returns false, hence Message object is returned.

‘msg’ filed of this object refers to “Good Morning!”, which is returned by toString() method.

Correction Java Stream API - 31

B - Compilation error

Method signature for anyMatch method:

boolean anyMatch(Predicate<? super T>) : Returns true if any of the stream element matches the given Predicate. If stream is empty, it returns false and predicate is not evaluated.

ref is a local variable and it is used within lambda expression.

++ref causes compilation failure as variable ref should be effectively final.

Correction Java Stream API - 32

D - 1.1.1.1.1.

count() is terminal operation and it needs all the elements of the stream to return the result. So, all the stream elements will be processed.

list.stream() => [-80, 100, -40, 25, 200].

NOTE: a new instance of Predicate is used, hence every time ctr will be initialize to 1.

For -80, Output is ‘1.’ but predicate returns false, hence element is not included.

For 100, ‘1.’ is appended to previous output, so on console you will see ‘1.1.’ and predicate returns true, hence element is included.

You don’t have to count the filtered elements as result of count() is not printed.

As 5 elements in the stream so final output will be: ‘1.1.1.1.1.’

Correction Java Stream API - 33

B - Compilation error

stream.mapToInt(i -> i) => returns an instance of IntStream.

average() method of all the 3 primitive streams (IntStream, LongStream & DoubleStream) return an instance of OptionalDouble.

OptionalDouble has getAsDouble() method and not getAsInt() method.

Correction Java Stream API - 34

C - nullOneTwoThree

Given reduce method concatenates null + “One” + “Two” + “Three” and hence the output is: ‘nullOneTwoThree’.

To concatenate just the stream contents, use below code:

stream.reduce(““, (s1, s2) -> s1 + s2)

OR

stream.reduce((s1, s2) -> s1 + s2).get()

Correction Java Stream API - 35

D - false

Constructor of Boolean class accepts String argument.

If passed argument is null, then Boolean object for ‘false’ is created.

If passed argument is non-null and equals to “true” in case-insensitive manner, then Boolean object for ‘true’ is created otherwise Boolean object for ‘false’ is created.

So list contains 4 elements and all are Boolean objects for false.

As, BinaryOperator extends BiFunction<T,T,T> so in this case signature of apply method will be: Boolean apply(Boolean, Boolean). Given lambda expression correctly implements the apply method.

To understand, ‘stream.reduce(false, operator)’ can be written as:

```
Boolean result = false;
for (Boolean element : stream) {
    result = operator.apply(result, element);
}
return result;
```

Above code is just for understanding purpose, you can’t iterate a stream using given loop.

As 1st argument of reduce method (also known as identity) is set to false and all the 4 stream elements are also false, hence list.stream().reduce(false, operator) will return false.

So, in this case false will be printed on to the console.

If you change identity to true, then statement ‘System.out.println(list.stream().reduce(true, operator));’ will print true.

Correction Java Stream API - 36

A - not

stream => [“and”, “Or”, “not”, “Equals”, “unary”, “binary”].

Test::isFirstCharVowel is the predicate, to invoke negate() method, it needs to be type-casted to ‘Predicate’.

stream.filter(((Predicate)Test::isFirstCharVowel).negate()) => [“not”, “binary”].

findFirst() => Optional object containing “not”.

optional.get() => “not”.

Correction Java Stream API - 37

A - [c, c++, java, python]

stream.collect(Collectors.toList()) returns an instance of ArrayList and hence output will always be in ascending order as stream was sorted using sorted() method before converting to list.

Correction Java Stream API - 38

B - Both 1 & 2

Variable id has package scope and as class Test is in the same package hence p.id doesn't cause any compilation error.

'Collectors.toMap(p -> p.id, Function.identity())' and 'Collectors.toMap(p -> p.id, p -> p)' are exactly same, as 'Function.identity()' is same as lambda expression 'p -> p'.

Collectors.toCollection(TreeMap::new) causes compilation error as TreeMap doesn't extend from Collection interface.

Correction Java Stream API - 39

D - Program executes successfully but nothing is printed on to the console

Stream.of() returns blank stream. As Type of stream is specified, stream is of ‘Stream’, each element of the stream is considered to be of ‘StringBuilder’ type.

map method in this case accepts ‘Function<? super StringBuilder, ? extends StringBuilder>’.

In Lambda expression ‘s -> s.reverse()’, s is of StringBuilder type and hence no compilation error.

As stream is blank, hence map and forEach methods are not executed even once. Program executes fine but nothing is printed on to the console.

Correction Java Stream API - 40

A - System.out.println(stream.reduce(0.0, (d1, d2) -> d1 + d2));

E - System.out.println(stream.reduce(0.0, Double::sum));

‘stream.reduce(0.0, (d1, d2) -> d1 + d2)’ and ‘stream.reduce(0.0, Double::sum)’ are exactly same and adds all the stream contents.

stream.sum() causes compilation error as sum() method is declared only in primitive streams (IntStream, LongStream and DoubleStream) but not in generic stream, Stream.

reduce method parameters are (Double, BinaryOperator).

0 (int literal) cannot be converted to Double and hence compilation error for ‘stream.reduce(0, (d1, d2) -> d1 + d2)’ and ‘stream.reduce(0, Double::sum)’.

You can easily verify this by writing below code:

```
public class Test {
    public static void main(String[] args) {
        print(0); //Compilation error as int can't be converted to
         Double
    }

    private static void print(Double d) {
        System.out.println(d);
    }
}
```

Correction Java Stream API - 41

C - X

findFirst() will never return empty Optional if stream is not empty. So no exception for get() method.

Also list and stream are not connected, which means operations done on stream doesn't affect the source, in this case list.

list.get(2) will print 'X' on to the console.

Correction Java Stream API - 42

B - ifPresent(System.out::println)

It is very simple as you don’t have to worry about return type of the code snippet.

stream is of IntStream type. Even method filter returns instance of IntStream type.

findFirst() returns an OptionalInt as it is called on IntStream.

Of all the given options, OptionalInt has ‘ifPresent’ method only. Hence correct answer is: ‘ifPresent(System.out::println)’.

Correction Java Stream API - 43

B - Replace stream.sorted(lengthComp) with stream.sorted(lengthComp.thenComparing(String::compareTo))

Current code displays below output:

```
d
a
mm
bb
zzz
www
```

if string's length is same, then insertion order is preserved.

Requirement is to sort the stream in ascending order of length of the string and if length is same, then sort on natural order.

lengthComp is for sorting the string on the basis of length, thenComparing default method of Comparator interface allows to pass 2nd level of Comparator.

Hence replacing 'stream.sorted()' with 'stream.sorted(lengthComp.thenComparing(String::compareTo))' will do the trick.

stream.sorted(lengthComp.reversed()) will simply reversed the order, which means longest string will be printed first, but this is not expected.

Correction Java Stream API - 44

D - On execution sum, average, max, min and count data will be printed on to the console

There are 3 summary statistics methods available in JDK 8: IntSummaryStatistics, LongSummaryStatistics & DoubleSummaryStatistics.

summaryStatistics() method in IntStream class returns an instance of IntSummaryStatistics.

summaryStatistics() method in LongStream class returns an instance of LongSummaryStatistics.

summaryStatistics() method in DoubleStream class returns an instance of DoubleSummaryStatistics.

The 3 summary statistics classes override toString() method to print the data about count, sum, min, average and max.

All the 3 summary statistics classes have methods to extract specific stat as well: getCount(), getSum(), getMin(), getMax() and getAverage().

Summary Statistics are really useful if you want multiple stats, say for example you want to find both min and max. As min and max are terminal operation for finite stream so after using one operation stream gets closed and not possible to use the same stream for other terminal operations.

Correction Java Stream API - 45

D - blue

Stream.of(“red”, “green”, “blue”, “yellow”) => [“red”, “green”, “blue”, “yellow”].

sorted() => [“blue”, “green”, “red”, “yellow”].

findFirst() => [“blue”]. findFirst returns Optional object.

Correction Java Stream API - 46

B - true

Method signatures:

boolean anyMatch(Predicate<? super T>) : Returns true if any of the stream element matches the given Predicate. If stream is empty, it returns false and predicate is not evaluated.

boolean allMatch(Predicate<? super T>) : Returns true if all the stream elements match the given Predicate. If stream is empty, it returns true and predicate is not evaluated.

boolean noneMatch(Predicate<? super T>) : Returns true if none of the stream element matches the given Predicate. If stream is empty, it returns true and predicate is not evaluated.

In the given code,

Stream.generate(() -> new Double(“1.0”)).limit(10); => returns a Stream containing 10 elements and each element is 1.0.

stream.filter(d -> d > 2) => returns an empty stream as given predicate is not true for even 1 element.

allMatch method, when invoked on empty stream, returns true.

Correction Java Stream API - 47

B - Compilation error

Stream.of() returns blank stream. As Type of stream is not specified, stream is of ‘Stream’, each element of the stream is considered to be of ‘Object’ type.

map method in this case accepts ‘Function<? super Object, ? extends R>’.

There is no ‘reverse()’ method in Object class and hence lambda expression causes compilation failure.

Correction Java Stream API - 48

A - Optional[10]

Optional is a final class and overrides toString() method:

```
public String toString() {
    return value != null
        ? String.format("Optional[%s]", value)
        : "Optional.empty";
}
```

In the question, Optional is of Integer type and Integer class overrides toString() method, so output is: Optional[10]

Correction Java Stream API - 49

D - -9 8 23 42 55

‘(i1, i2) -> i2.compareTo(i1)’ helps to sort in descending order. Code is: ‘i2.compareTo(i1)’ and not ‘i1.compareTo(i2)’.

comp.reversed() returns a Comparator for sorting in ascending order. Hence, the output is: ‘-9 8 23 42 55’.

Correction Java Stream API - 50

A - [{S001, OCP, 79}, {S002, OCP, 89}]

Collectors.groupingBy(Certification::getTest) => groups on the basis of test which is String type.

Hence return type is: Map<String, List>.

There are 4 records for OCA exam and 2 records for OCP exam, hence map.get(“OCP”) returns the list containing OCP records.

Correction Java Stream API - 51

B -

```
5
6
```

As variable seed is of long type, Hence Stream.iterate(seed, i -> i + 2) returns an infinite stream of Stream type. limit(2) returns the Stream object containing 2 elements 10 and 12. There is no issue with line n1.

Though the lambda expression with 2 arrows seems confusing but it is correct syntax. To understand, Line n2 can be re-written as:

```
LongFunction<LongUnaryOperator> func = (m) -> {
    return (n) -> {
        return n / m;
    };
};
```

And corresponding anonymous class syntax is:

```
LongFunction<LongUnaryOperator> func = new
        LongFunction<LongUnaryOperator>() {
    @Override
    public LongUnaryOperator apply(long m) {
        LongUnaryOperator operator = new LongUnaryOperator() {
            @Override
            public long applyAsLong(long n) {
                return n / m;
            }
        };
        return operator;
    }
};
```

So, there is no issue with Line n2. Let's check Line n3.

stream.mapToLong(i -> i) returns an instance of LongStream and LongStream has map(LongUnaryOperator) method.

'func.apply(2)' returns an instance of LongUnaryOperator, in which applyAsLong(long) method has below implementation:

```
LongUnaryOperator operator = new LongUnaryOperator() {
    @Override
    public long applyAsLong(long n) {
        return n / 2;
    }
};
```

As stream has elements 10 and 12, so map(func.apply(2)) returns an instance of LongStream after dividing each element by 2, so resultant stream contains elements 5 and 6.

forEach(System.out::println) prints 5 and 6 on to the console.

Correction Java Stream API - 52

A - Replace stream.map(s -> s.length()) with stream.mapToInt(s -> s.length())

text.split(" ") => {"I", "am", "going", "to", "pass", "OCP", "exam", "in", "first", "attempt"}.

Arrays.stream(text.split(" ")); => ["I", "am", "going", "to", "pass", "OCP", "exam", "in", "first", "attempt"]. Stream instance is returned.

stream.map(s -> s.length()) => [1, 2, 5, 2, 4, 3, 4, 2, 5, 7]. Stream is returned.

summaryStatistics() method is declared in IntStream, LongStream and DoubleStream interfaces but not declared in Stream interface and hence 'stream.map(s -> s.length()).summaryStatistics();' causes compilation failure.

Out of the given options, replacing 'stream.map(s -> s.length())' with 'stream.mapToInt(s -> s.length())' will correctly return an instance of IntStream and hence summaryStatistics() method can easily be invoked.

As you had to select only one option, so you can stop here. No need to validate other options. I am explaining other options just for knowledge purpose.

stat.getCount() will return 10 so not a correct option.

text.split(" ") delimits the text on the basis of single space.

text.split(",") will delimit it on the basis of comma but as no comma is present in the given text, hence whole text will be returned and stat.getMax() will print 44.

Correction Java Stream API - 53

A -

```
TRUE
FALSE
TRUE
```

count() is a terminal method for finite stream, hence
peek(System.out::println) is executed for all the 3 elements of the stream.

count() method returns long value but it is not used.

Correction Java Stream API - 54

D - 7

new Random().ints(start, end) => start is inclusive and end is exclusive.

So this code generates random integers between 1 and 6. All the 6 integers from 1 to 6 are possible.

Above code will never generate 7.

Correction Java Stream API - 55

B - true : 2018-01-01

stream => [{2018-1-1}, {2018-1-1}].

stream.distinct() => [{2018-1-1}].

findAny() => Optional[{2018-1-1}].

optional.isPresent() => true. isPresent method returns true if optional is not empty otherwise false.

optional.get() => Returns LocalDate object {2018-1-1}, toString() method of LocalDate class pads 0 to single digit month and day.

'true : 2018-01-01' is printed on to the console.

NOTE: In real world projects, it is advisable to check using isPresent() method before using the get() method.

```
if(optional.isPresent()) {
    System.out.println(optional.get());
}
```

Correction Java Stream API - 56

B - Compilation error

stream is of Stream type, which is a generic stream and not primitive stream.

There is no min() method available in generic stream interface, Stream and hence, ‘stream.min()’ causes compilation error.

Generic Stream interface has following methods:

Optional min(Comparator<? super T> comparator);

Optional max(Comparator<? super T> comparator);

To calculate min for generic Stream, pass the Comparator as argument: stream.min(Double::compareTo), but note it will return an instance of Optional type.

Correction Java Stream API - 57

B -

```
bonita
John
Peter
```

In this example, Stream is used. sorted method accepts Comparator<? super String> type.

compareToIgnoreCase is defined in String class and it compares the text by in case-insensitive manner.

Even though ‘b’ is in lower case it is printed first, followed by ‘J’ and ‘P’.

Correction Java Stream API - 58

D - Optional[7000.0]

In real exam, don't predict the output by just looking at the method name.

It is expected that highestSalary(…) method will print 'Optional[12000.0]' on to the console but if you closely check the definition of Employee.salaryCompare(…) method you will note that it helps to sort the salary in descending order and not ascending order.

Rest of the logic is pretty simple.

emp => [{"Jack", 10000.0}, {"Lucy", 12000.0}, {"Tom", 7000.0}].

emp.map(e -> e.getSalary()) => [10000.0, 12000.0, 7000.0].

max(Employee::salaryCompare) => Optional[7000].

NOTE: There are 3 methods in Stream interface, which returns Optional type:

1. Optional max(Comparator<? super T> comparator);
2. Optional min(Comparator<? super T> comparator);
3. Optional reduce(BinaryOperator accumulator);

Correction Java Stream API - 59

A - Bumrah Dhoni Pandya Rohit Sami Shikhar Virat

stream is not of Stream type rather it is of Stream<String[]> type.

flatMap method combines all the non-empty streams and returns an instance of Stream containing the individual elements from non-empty stream.

stream => [{“Virat”, “Rohit”, “Shikhar”, “Dhoni”}, {“Bumrah”, “Pandya”, “Sami”}, {}].

stream.flatMap(s -> Arrays.stream(s)) => [“Virat”, “Rohit”, “Shikhar”, “Dhoni”, “Bumrah”, “Pandya”, “Sami”].

sorted() => [“Bumrah”, “Dhoni”, “Pandya”, “Rohit”, “Sami”, “Shikhar”, “Virat”].

Correction Java Stream API - 60

A - UnaryOperator operator = c -> (char) c (c.charValue() + 1);

E - Function<Character, Character> operator = x -> (char)(x + 1);

As ‘vowels’ refers to List, hence vowels.stream() returns Stream type. So, map method of Stream type has signature: Stream map(Function<? super Character, ? extends R> mapper);

Since forEach(System.out::print) is printing BFJPV, hence result of map(x -> operator.apply(x)) should be Stream and not Stream.

This means correct reference type of ‘operator’ variable is Function<Character, Character>. Now as UnaryOperator extends Function<T, T>, so UnaryOpeartor is also correct reference type of ‘operator’ variable.

Out of 6, we are left with 4 options. Let’s check the options one by one:

‘UnaryOperator operator = c -> c + 1;’: ‘c + 1’ results in int and int can be converted to Integer but not Character, so this causes compilation failure.

‘UnaryOperator operator = c -> c.charValue() + 1;’: ‘c.charValue() + 1’ results in int and int can be converted to Integer but not Character, so this causes compilation failure.

‘UnaryOperator operator = c -> (char)(c.charValue() + 1);’: This expression adds 1 to the current char value (primitive char is compatible with primitive int) and resultant int value is type-casted to char, which is converted to Character by auto-boxing. Hence, this is correct option.

‘Function<Character, Character> operator = x -> (char)(x + 1);’: This is also correct option. x + 1 results in int, which is type-casted to char and finally converted to Character by auto-boxing.

Correction Java Stream API - 61

B - Nothing is printed and program runs infinitely

Stream.generate(() -> new Double(“1.0”)); generates an infinite stream of Double, whose elements are 1.0.

stream.sorted() is an intermediate operation and needs all the elements to be available for sorting.

As all the elements of infinite stream are never available, hence sorted() method never completes.

So among all the available option, correct option is: ‘Nothing is printed and program runs infinitely.’

Correction Java Stream API - 62

C - OptionalDouble.empty

average() method in IntStream, LongStream and DoubleStream returns OptionalDouble.

As stream is an empty stream, hence ‘stream.average()’ returns an empty optional.

OptionalDouble.empty is printed on to the console for empty Optional.

Correction Java Stream API - 63

C - Runtime Exception

In this case, value variable inside Optional instance is null.

optional.isPresent() => false. isPresent method returns true if optional is not empty otherwise false.

If value variable inside Optional instance is null (empty optional), then NoSuchElementException is thrown at runtime.

In real world projects, it is advisable to to check using isPresent() method before using the get() method.

```
if(optional.isPresent()) {
    System.out.println(optional.getAsLong());
}
```

NOTE: There are 3 primitive equivalents of Optional interface available. Remember their similarity with Optional class.

Optional:

Optional empty(),

T get(),

boolean isPresent(),

Optional of(T),

void ifPresent(Consumer<? super T>),

T orElse(T),

T orElseGet(Supplier<? extends T>),

T orElseThrow(Supplier<? extends X>),

Optional filter(Predicate<? super T>),

Optional map(Function<? super T, ? extends U>),

Optional flatMap(Function<? super T, Optional>).

OptionalInt:

OptionalInt empty(),

int getAsInt(),

boolean isPresent(),

OptionalInt of(int),

void ifPresent(IntConsumer),

int orElse(int),

orElseGet(IntSupplier),

int orElseThrow(Supplier).

[filter, map and faltMap methods are not available in primitive type].

OptionalLong:

OptionalLong empty(),

long getAsLong(),

boolean isPresent(),

OptionalLong of(long),

void ifPresent(LongConsumer),

long orElse(long),

long orElseGet(LongSupplier),

long orElseThrow(Supplier).

[filter, map and faltMap methods are not available in primitive type].

OptionalDouble:

OptionalDouble empty(),

double getAsDouble(),

boolean isPresent(),

OptionalDouble of(double),

void ifPresent(DoubleConsumer),

double orElse(double),

double orElseGet(DoubleSupplier),

double orElseThrow(Supplier).

[filter, map and faltMap methods are not available in primitive type].

Correction Java Stream API - 64

B - NullPointerException is thrown at runtime

If null argument is passed to of method, then NullPointerException is thrown at runtime.

Correction Java Stream API - 65

A - Program executes successfully but nothing is printed on to the console

Streams are lazily evaluated, which means for finite streams, if terminal operations such as: forEach, count, toArray, reduce, collect, findFirst, findAny, anyMatch, allMatch, sum, min, max, average etc. are not present, the given stream pipeline is not evaluated and hence peek() method doesn’t print anything on to the console.

Correction Java Stream API - 66

B - Optional[7.5]

‘stream.reduce((d1, d2) -> d1 + d2)’ returns ‘Optional’ type whereas ‘stream.reduce(0.0, (d1, d2) -> d1 + d2)’ returns ‘Double’.

Correction Java Stream API - 67

B - Compilation error

Method isPresent() returns boolean whereas method ifPresent accepts a Consumer parameter.

‘first.ifPresent()’ causes compilation failure.