**Advanced Streams Labs**

1. Stream a list of *int* primitives between the range of 0 (inclusive) and 5 (exclusive). Calculate and output the average. (QID 2.2023)
2. Given the *Item* class (in the zip file), declare a *List* typed for *Item* with the following *Item*’s:
   1. id=1 name=”Screw”
   2. id=2 name=”Nail”
   3. id=3 name=”Bolt”

Stream the list and sort it so that it outputs “BoltNailScrew” i.e. alphabetic *name* order. Use *Stream*’s *forEach* method to output the names (use the method reference version for the required *Consumer* lambda). (QID Q2.1762)

1. Given the *Student* class (in the zip file), declare a *List* typed for *Student* with the following *Student*’s:
   1. name=”S1” grade=A (note: grade is a *static* *enum* in *Student*)
   2. name=”S2” grade=A
   3. name=”S3” grade=C

Stream the list and collect the information from the stream into a *Map* so that it appears as follows: {C=[S3], A=[S1,S2]} (QID Q2.1804)

1. Generate a *Stream<List<String>>* using the *Stream.of(Arrays.asList(“a”, “b”), Arrays.asList(“a”, “c”))* method call. Filter the stream so that only list’s that contain “c” make it through the filter. Flatten the *Stream<List<String>>* to a *Stream<String>* using the *flatMap()* operation. Use *forEach()* to kick off the whole operation and output the stream. (QID 2.1787)
2. Which of the following method(s) of the java.util.stream.Stream interface is/are used for reduction?  
    - filter, reduce, sum, max, add (select 2) (QID 2.1738)
   1. is *sum()* defined in the Stream interface?
   2. what does *mapToInt()* do?
   3. which version of max() requires a *Comparator* – *Stream* or *IntStream*? Why?
   4. why does the reduce(identity, BinaryOperator) not return an Optional whereas reduce(BinaryOperator) does?
      1. Note: BinaryOperator is a special case of BiFunction where the 2 inputs and the output are all the same type T.
3. Code a method *public static Optional<String> getGrade(int marks)* (QID 2.1826)
   1. in the method *getGrade*:
      1. declare an empty optional, typed for *String* called *grade*
      2. insert the following code:
         1. *if (marks > 50) {grade = Optional.of(“PASS”);} else {grade.of(“FAIL”);}*
   2. in *main():*
      1. declare an *Optional*, typed for *String* named *grade1* which is initialised to the return value of calling *getGrade(50)*
      2. declare an *Optional*, typed for *String* named *grade2* which is initialised to the return value of calling *getGrade(55)*
      3. using *orElse()* on *grade1*, output the value of *grade1* or “UNKNOWN”
      4. *if(grade2.isPresent())* is true: use *ifPresent(Consumer)* to output the contents of *grade2*; if false, use *orElse()* to output the contents of *grade2* or “Empty”
      5. Notes:
         1. *Optional*’s are immutable.
         2. *Optional.of(null);* // NullPointerException
         3. *Optional.ofNullable(null);* // Optional.empty returned
4. Given the *Book* class (in the zip file), declare a *List* typed for *Book* with the following *Book*’s:
   1. title=”Thinking in Java”, price=30.0
   2. title=”Java in 24 hrs”, price=20.0
   3. title=”Java Recipes”, price=10.0

Stream the books and calculate the average price of the books whose price is > 10.   
Change the filter to books whose price is > 90. Ensure you do not get an exception. (QID 2.1809)

1. Given the *Book* class (in the zip file), declare a *List* typed for *Book* with the following *Book*’s:
   1. title=”Atlas Shrugged”, price=10.0
   2. title=”Freedom at Midnight”, price=5.0
   3. title=”Gone with the wind”, price=5.0

Stream the books and instantiate a *Map* named ‘*bookMap*’ that maps the book *title* to its *price*. To do this use the *collect(Collectors.toMap(Function fnToGetKey, Function fnToGetValue))*. Iterate through ‘*bookMap’* (using the *Map* *forEach(BiConsumer)* method). The *BiConsumer* only outputs prices where the title begins with “A”. (QID 2.1846)

1. Given the *Book* class (in the zip file), declare a *List* typed for *Book* with the following *Book*’s:
   1. title=”Gone with the wind”, price=5.0
   2. title=”Gone with the wind”, price=10.0
   3. title=”Atlas shrugged”, price=15.0

In a pipeline which has no return type: stream the books, generate a *Map* that maps the book *title* to its *price*, and output the title and price of each entry in the map. What happened and why? Fix this by using the *Collectors.toMap(Function, Function, BinaryOperator)* method. (QID 2.1847)

1. Given the *Person* class (in the zip file), declare a *List* typed for *Person* with the following *Person*’s:
   1. name=”Bob”, age=31
   2. name=”Paul”, age=32
   3. name=”John”, age=33

Pipeline the following where the return type is *double*: stream the people, filter the stream so that only *Person*’s whose age is >= 30 remain (i.e. filter out if age < 30), map to *int* primitives and calculate the average. This should generate a NoSuchElementException. Fix the pipeline so that no exception is generated i.e. 0.0 is returned. (QID 2.1810)

1. Declare an *Optional*, typed for *Double*, named ‘*price’* using the *Optional.ofNullable(20.0).* Output the *Optional* value for *‘price’* 3 times: using *ifPresent(Consumer), orElse(T)* and *orElseGet(Supplier).* (QID 2.1849)
   1. declare a new *Optional*, typed for *Double,* named ‘*price2*’*.* Use *Optional.ofNullable* again but this time, pass in *null*.
      1. Output ‘*price2’* in a normal *System.out.println().*
      2. check to see if *price2* *isEmpty()* and if so output “empty”.
      3. do (ii) again except this time use the more functional “*ifPresent(Consumer)*” method.
      4. initialise a *Double x* to the return of “*price2.orElse(44.0)*”. Output and observe the value of *x*.
   2. declare a new *Optional*, typed for *Double,* named ‘*price3*’*.* Use *Optional.ofNullable* passing in *null*.
      1. initialise a *Double z* to the return of “*price3.orElseThrow(() -> new RuntimeException(“Bad Code”)*”. Output and observe the value of z.
2. Given the *AnotherBook* class (in the zip file), declare a *List* typed for *AnotherBook* namely ‘*books*’ with the following *AnotherBook*’s:
   1. title=”Gone with the wind”, genre=”Fiction”
   2. title=”Bourne Ultimatum”, genre=”Thriller”
   3. title=”The Client”, genre=”Thriller”

Declare the following: *List<String> genreList = new ArrayList<>();*

Stream *books* so that *genreList* refers to a *List* containing the genres of the books in the *books List*. (QID 2.1858)

1. Generate a *DoubleStream* using the *of()* method consisting of the numbers 0, 2 and 4. Note that this stream is a stream of primitives and not a stream of types. Filter the stream to filter in odd numbers only. Sum the remaining stream. You should get 0. Why does the *sum()* operation return *double* but *average()* return *OptionalDouble*? (QID 2.2024)
2. Declare the following *List<Integer> ls = Arrays.asList(11, 11, 22, 33, 33, 55, 66);*
   1. stream the *List* (note that this is possible because *List* is a *Collection* and *Collection* defines a *stream()* method); ensure only distinct (unique) numbers are streams; check if “any match” 11. You should get *true* for this.
   2. stream the *List* again (this is necessary because once a stream is closed by a previous terminal operation, you must re-create the stream); check to see if “none match” the expression x%11>0. Note that the terminal operation *noneMatch(Predicate)* needs to return *false* for every element in the stream for *noneMatch()* to return *true*. In other words, “none of them match this….that’s correct, none of them do; return true”. You should get *true* here as well.

(QID 2.1840)

1. Insert the following code. Note that an AtomicInteger is a version of Integer that is safe to use in concurrent (multi-threaded) environments. The method incrementAndGet() is similar to *++ai*  
    a) Why is the value of *ai* at the end 0 (and not 4)?

AtomicInteger ai = new AtomicInteger(); // initial value of 0

Stream.of(11, 11, 22, 33)

.parallel()

.filter(n -> {

ai.incrementAndGet();

return n % 2 == 0;

});

System.out.println(ai);

1. Insert the following code. Why does it generate an *IllegalStateException* exception?

AtomicInteger ai = new AtomicInteger(); // initial value of 0

Stream<Integer> stream = Stream.of(11, 11, 22, 33).parallel();

stream.filter( e->{

ai.incrementAndGet();

return e%2==0; });

stream.forEach(System.out::println);// IllegalStateException

System.out.println(ai);