Streams Assignment

These exercises are based on question id's (QID's) from Enthuware, the excellent Java Certification training tool. Enthuware is an MCQ tool and this is why some answers in the video refer to "options". However, I have left these discussions in the solutions video as it enables me to discuss extra material, which aids in the learning experience. Note: the QID's are from Java 8 OCP question bank.

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:
   a. id=1 name="Screw"
   b. id=2 name="Nail"
   c. 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)

3. 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. Note that *flapMap()* states in the API "*Each mapped stream is closed after its contents have been placed into this* [new] *stream.*". Use *forEach()* to output the new stream. (QID 2.1787)

4. There are several parts to this: (QID 2.1738)

   a. Using 1, 2 and 3 create a *List* of *Integer*s.
      i. Stream the list and calculate the sum, using the *sum()* method from *IntStream*.
      ii. Stream the list again and calculate the maximum value, using the *max()* method from *IntStream*.
   b. Given the *Person* class (in the zip file), declare a *List* typed for *Person* with the following *Person*'s:
      i. "Alan", "Burke", 22
      ii. "Zoe", "Peters", 20
      iii. "Peter", "Castle", 29

      Using the *max(Comparator)* from *Stream*, calculate the oldest person in the list.
   c. Using 10, 47, 33 and 23 create a *List* of *Integer*s. Stream the list and using the following versions of *reduce()*, calculate the maximum value:
      i. *Optional<T> reduce(BinaryOperator<T> accumulator)*
      ii. *T reduce(T identity, BinaryOperator<T> accumulator)*

5. Code a method *public static Optional<String> getGrade(int marks)* (QID 2.1826)
    a. in the method *getGrade*:
        i. declare an empty optional, typed for *String* called *grade*
        ii. insert the following code:
            if (marks > 50) {grade = Optional.of("PASS");} else {grade.of("FAIL");}
    b. in *main():*
        i. declare an *Optional*, typed for *String* named *grade1* which is initialised to the return value of calling *getGrade(50)*
        ii. declare an *Optional*, typed for *String* named *grade2* which is initialised to the return value of calling *getGrade(55)*
        iii. using *orElse()* on *grade1*, output the value of *grade1* or "UNKNOWN"
        iv. *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"
        v. Notes:
            1. *Optional*'s are immutable.
            2. *Optional.of(null);* // NullPointerException
            3. *Optional.ofNullable(null);* // Optional.empty returned

6. Given the *Book* class (in the zip file), declare a *List* typed for *Book* with the following *Book*'s:
    a. title="Thinking in Java", price=30.0
    b. title="Java in 24 hrs", price=20.0
    c. 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)

7. Given the *Book* class (in the zip file), declare a *List* typed for *Book* with the following *Book*'s:
    a. title="Atlas Shrugged", price=10.0
    b. title="Freedom at Midnight", price=5.0
    c. 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)

8. Given the *Book* class (in the zip file), declare a *List* typed for *Book* with the following *Book*'s:
    a. title="Gone with the wind", price=5.0
    b. title="Gone with the wind", price=10.0
    c. title="Atlas shrugged", price=15.0

    In a pipeline which has no return type: (QID 2.1847)
    - stream the books
    - using the *collect()* method, generate a *Map* that maps the book *title* to its *price*
    - using *forEach()*, 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.

9. Given the *Person* class (in the zip file), declare a *List* typed for *Person* with the following *Person*'s:
    a. name="Bob", age=31
    b. name="Paul", age=32
    c. name="John", age=33

    Pipeline the following where the return type is *double*: (QID 2.1810)
    - stream the people
    - filter the stream for *Person*'s whose *age* is < 30
    - map to *int* primitives
    - calculate the average *age*.

    This should generate a *NoSuchElementException*. Using *orElse()*, fix the pipeline (not the filter) so that 0.0 is returned instead of an exception being generated.

10. A question about *Optional*. Let us look at this in parts:
    a. 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)
    b. declare a new *Optional*, typed for *Double,* named '*price2*' (or comment out (a) and re-use '*price*'). Use *Optional.ofNullable* again but this time, pass in *null*.
        i. Output '*price2'* in a normal *System.out.println().*
        ii. check to see if *price2 isEmpty()* and if so output "empty".
        iii. do (ii) again except this time use the more functional "*ifPresent(Consumer)*" method.
        iv. initialise a *Double x* to the return of "*price2.orElse(44.0)*". Output and observe the value of *x*.
    c. declare a new *Optional*, typed for *Double,* named '*price3*' (or comment out (b) and re-use '*price*'). Use *Optional.ofNullable* passing in *null*.
        i. initialise a *Double z* to the return of "*price3.orElseThrow(() -> new RuntimeException("Bad Code")*. Output and observe the value of z.

11. Given the *AnotherBook* class (in the zip file), declare a *List* typed for *AnotherBook* namely '*books*' with the following *AnotherBook*'s:
    a. title="Gone with the wind", genre="Fiction" (QID 2.1858)
    b. title="Bourne Ultimatum", genre="Thriller"
    c. 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*.

12. There are two parts:
    a. 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 in odd numbers only and sum the remaining stream. You should get 0. (QID 2.2024)
    b. Using 1.0 and 3.0, generate a stream of *Double*'s. Map them to primitive *double*'s. Filter in even numbers only and calculate the average. Output the result without running the risk of generating an exception.

13. This question demonstrates lazy evaluation. Declare the following *List<Integer> ls = Arrays.asList(11, 11, 22, 33, 33, 55, 66);*
    a. 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 streamed; check if "any match" 11. You should get *true* for this.
    b. 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)

14. Examine 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)? (QID 2.1841)

```
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);
```

    b) The following code generates an *IllegalStateException*. Fix the code.

```
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);
```