1. Consider the two following functions:

```
public static boolean compute(final int number) {
      return number > 5 ? true : false;
}
```

And

```
public static boolean process(final int number) {
      return number % 3 == 0 ? true : false;
}
```

These two are used as follows:

```
public static void main(String[] args) {
      final int number = 4;
      if (compute(number) && process(number)) {
            System.out.println("TRUE");
      } else {
            System.out.println("FALSE");
      }
}
```

Convert the above `main(String[])` method so that it uses in-line lambda expressions instead of calling the external functions `compute(int)` and `process(int)` . Recall that the syntax of a lambda expression in java is simply `parameter -> expression body`.

2. Write a method called `addAll(Collection<Integer>)` . This method should be a proper function (i.e., it should not have any "side effects"), and must be implemented using the reduce() method from java.util.stream.Stream . Also think about the following:

   1. What should be the return type of this function?
   2. Do you think this method is better off as an instance method or a static method? Why?

   Clarification: This method is asking you to add all the integers in the given collection (passed as the parameter to the `addAll` method) to obtain their total. It is NOT about adding multiple integers into a collection.

3. Write a method called shortestString(List<String>) , which takes a list of strings as its input parameter and returns the string from that list with the fewest characters. If the input list is empty, this method should return the empty string. Your implementation must use the reduce() method again.

4. With the Stream library, it is easy to write long sequences of actions as a single method chain in Java. This is possible because once you convert a collection of items to a stream, you can perform operations whose output is also a stream. These are called intermediate operations . The final method in such a chain transforms its input to a non-stream data type. This is called a terminal operation . Functions like filter() , map() , limit() , etc. are intermediate operations. Others, like reduce() , findAny() , collect() , forEach() , etc. are terminal operations. Spend some time going through the Stream functions here:
   https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html.

   Note the functions whose input parameter type and return type are both Stream<T> . These are the intermediate operations. Armed with this knowledge, use higher-order functions such as map() , filter() , etc. from Java's Stream library to do the following: given a set of positive integers, retain only those that are of exponents of 2 (e.g., 2, 4, 8, 16), and return them as a list. You are free to mix other stream operations, of course, but try to implement the whole code as a single method chain.

5. When working with streaming data, we often encounter situations where we deal with multiple streams, and we may want to apply the same method to all of them. Consider a scenario where you are reading a text file, and you simply want to collect all the words in the file. Read the documentation for the flatMap() function:

   https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#flatMap-java.util.function.Function, and write a functional piece of code that takes the path to such a file as its input parameter, and returns a set of all the words in that file. For simplicity, assume that all words are separated by a single space, and they are all lowercase. You also don't have to worry about punctuation and other symbols for this question.

6. How would you generalize your solution to question 5, if you are also asked to maintain the counts of the words? Clearly, returning the words as a set will not work. So, think about the data type you want to return, and how you can use the Stream library functions to accomplish your new goal.