Java Final Test 2023 – feedback on Section A

Here is some general feedback on solutions to Section A of the final test.

Question 1 – populating the Note class

This question was pretty trivial in terms of getting the functionality to work. Half the marks for the question were thus devoted to design issues:

- Use of encapsulation (all fields should be private, except for – optionally – named constants), and there should be no non-private methods except those asked for in the spec
- Use of immutability-enforcing constructs (all fields should be final, and the class should be final)
- Use of named constants instead of magic numbers: fairly harsh deductions were made for the explicit use of the magic numbers 12, 64 and 200 – instead, to get full marks, you needed to introduce named constants – static final fields named things like NUM_NOTES_IN_OCTAVES
- A lot of (most!) students lost quite a few marks on the magic numbers front, either for neglecting to use named constants at all, or for declaring them but not using static and final correctly
- To map notes to names, the best solution was to use a switch statement in toString. It was also OK to declare a hash map or array, mapping each note within an octave to its name. However, with such an approach it was important to make the hash map / array static, otherwise every note that ever exists gets its own mapping, and the mappings are all identical

Question 2 – equality on notes

This was done very well. Most students appeared to have used the IDE to auto-generate equals and hashCode, which is fine.

Some students lost a style mark due to the IDE generating an equals method with a parameter named "o", which violated Google Style (public method parameters cannot be single letter). When you use an IDE to help in a project, do make sure that you adapt its help to fit the style of the project.

Some students forgot to override hashCode. This was (deliberately) not asked for explicitly in the spec, as I wanted you to remember that when you override equals you should always override hashCode, but a test had been designed to fail if it was not done.

Question 3 – additional Note methods

This was an easy 5 marks, and pretty much everyone got them.

Question 4 – the Tune interface and a simple implementation

There were a few minor issues here:

- Some students used "tune" as the name for the list of notes associated with a StandardTune. This is a bad name because this field is a *component* of a StandardTune, which "is a" Tune, so it is strange to have a component of an object with a name that reflects the whole object. A better name would be "notes"

- Very minor, but “notesList” - while an OK name – is not as good as “notes”, because the type of the field already documents that it is a list
- Some students forgot to make the field(s) of StandardTune private
- The getNotes method was required to return a mutable list of the notes in the tune that – when modified – would not change the original tune. Most students correctly realised they needed to return a new list with the same contents as the old list. Few students got that this could be done easily, by: “return new ArrayList<>(notes)”

Question 5 – transposing tunes

Solutions to this question fell into three broad categories:

- Functionally correct solutions that used a helper method to extract common code for shifting notes, so that the code is not duplicated between getNotes and addNote. This required observing that pitch shifting needs to be reversed for addNote.
- Functionally correct solutions that did not use such helper methods.
- Solutions that followed the required design (having a TransposedTune interact with a target tune, so that a TransposedTune does not actually keep track of its own notes), but with functional problems.
- Solutions that missed the point of the question, so that a TransposedTune would have its own list of notes that it would initially copy from the target tune, but after that not interact with the target tune. The problem with this approach is that changes to the target tune do not get reflected in the transposed tune, and vice versa.

Question 6 – calculating the duration of any tune

Here, I was looking for getTotalDuration to be added to the Tune interface as a default method. This way, it can be invoked on *any* Tune object that *ever* exists – not just StandardTunes and TransposedTunes, but other implementations of the Tune interface.

Solutions that implemented getTotalDuration separately in StandardTune and TransposedTune received very low marks here. Similarly, inserting an abstract class between Tune and Standard/TransposedTune was not the right approach as this would not help with Tunes that implement the interface directly.

The most elegant solution was to stream the result of getNotes(), map Note::getDuration over this stream, and then reduce with identity 0 using Integer::sum.