Imperial College London

Department of Computing

"Kotlin Tunes"

THREE HOURS
(including 10 minutes of suggested planning time)

- The maximum total is **100 marks**.
- Credit is awarded throughout for conciseness, clarity, *useful* commenting, and the appropriate use of the various language features.
- **Important:** Marks are deducted from solutions that do not compile in the test environment, which will be the same as the lab machines. Comment out any code that does not compile before you submit.
- After you have finished reading the spec, you can start coding by running the IDEA shortcut on your Desktop, which will open our provided .iml file. **Please do not attempt to open your project in any other way**, and please do not delete any of our files.
- The extracted files should be in your Home folder, under the "`test`" subdirectory. **Do not move any files**, or the test engine will fail, resulting in a compilation penalty.
- The examples and test cases here are not guaranteed to exercise all aspects of your code. You are therefore advised to define your own tests to complement the ones provided.
- When you are finished, simply **save everything and log out**. **Do not shut down your machine**. We will fetch your files from the local storage.
- If your IDE fails to build your code, you can still compile via the terminal using the provided `./compile-all` perl script. You can then test your code via `./test-all`.

Problem Description

Your task is to write a number of classes and interfaces to represent musical tunes, and a collection of tunes. You are not expected to have any prior knowledge of musical notation to complete this task.

A tune is made up from notes, where each note comprises a pitch and a duration. You will write a class that represents a note, an interface capturing the services provided by a tune, and four implementations of this interface:

- a *standard* tune, comprising a list of notes;
- a *transposed* tune, which provides a view of another tune in which the pitch of every note is shifted by a specified amount;
- a *stretched* tune, which provides a view of another tune in which the duration of every note is scaled by a given factor;
- a *thread-safe* tune, which uses locks to allow a given tune to be manipulated concurrently by multiple threads.

You will also write a class that implements a collection of songs, where a song comprises a name and a tune. A song collection will be arranged as a binary search tree, ordered by song name, and your job will be to implement certain binary search tree operations.

You will write all of this code in Kotlin, with the exception of the "stretched" implementation of the tune interface, which you will implement in Java.

Getting Started

The skeleton files are located under the `src` package.

Under `src` there are `main` and `test` sub-directories. The code you write to implement functionality should go under `main`, while any test code should go under `test`.

Under `main` there are `kotlin` and `java` sub-directories, and code for the respective language should be located under the corresponding sub-directory. Within each of the `kotlin` and `java` sub-directories there is a directory called `tunes`. All functional code you write for this exercise will belong to the `tunes` package, and thus should reside in one of these `tunes` directories.

Under `test` there is only a `kotlin` sub-directory (because you are not expected to write tests in Java and all the provided tests are in Kotlin). Under this sub-directory is a `tunes` sub-directory. All provided tests plus any new tests that you create should be located here, in the `tunes` package.

You may feel free to add additional methods and classes, beyond those specified in the instructions, as you see fit. For example, this may be in order to follow good object-oriented principles, or for testing. Any new files should be placed in the `tunes` package for the appropriate programming language.

Testing

There is a test class, `Question`*i*`Tests`, for each question i. These contain initially commented-out tests to help you gauge your progress. **As you progress through the exercise you should un-comment the test class associated with each question in order to test your work.** In some cases you will be required to add additional tests to these classes as part

of your task. Furthermore, you are welcome to add tests to these classes to help you debug your solution.

These tests are not exhaustive and are merely intended to guide you. Your solution should pass all of the given tests. However, passing all of the given tests does not guarantee that your solution is fully correct, and says nothing about your coding style and the appropriateness of your use of Kotlin and Java features. You should thus think carefully about whether your solution is complete, and pay attention to coding style and practices, even if you pass all of the given tests.

What to do

1. **Populating the `Note` class.**

   Adapt and flesh out the provided skeleton class `Note` to meet the following requirements:

   - A note should be represented by an integer `pitch` and a floating-point `duration` (of type `Double`). These should be provided on construction. The pitch should be non-negative and no larger than 200. The duration should be positive and no larger than 64.0. The constructor of `Note` should throw an `IllegalArgumentException` if either of these requirements is not satisfied.
   - Two notes should be considered equal if, and only if, their pitches are the same and their durations are the same.
   - The `Note` class should be immutable, and it should not be possible to create sub-classes of `Note`.
   - Clients of `Note` should be provided with read access to the `pitch` and `duration` of the note.
   - The `toString()` method of `Any` should be overridden to turn a note into a string comprising the *name* of the note, the *octave* of the note, and the *duration* of the note in parentheses. The *octave* of a note is an integer and is obtained by dividing the pitch of the note by 12 (the number of notes in an octave), and rounding down if this is not a whole number. The *name* of the note is then derived from the value of the note within its octave, which is the pitch of the note remainder 12. The mapping from in-octave values to names is as follows:

     `0` $\mapsto$ `C` `1` $\mapsto$ `C#` `2` $\mapsto$ `D` `3` $\mapsto$ `D#` `4` $\mapsto$ `E` `5` $\mapsto$ `F`
     `6` $\mapsto$ `F#` `7` $\mapsto$ `G` `8` $\mapsto$ `G#` `9` $\mapsto$ `A` `10` $\mapsto$ `A#` `11` $\mapsto$ `B`

     Here are some examples of the `toString()` representations of various notes:

Pitch	**Duration**	`toString()`
0	4.0	`C0(4.0)`
1	3.0	`C#0(3.0)`
12	7.0	`C1(7.0)`
13	5.0	`C#1(5.0)`
23	11.0	`B1(11.0)`
53	12.0	`F4(12.0)`

   Add the following public methods to the `Note` class, implemented to meet the given specifications:

- `fun hasNoteAbove(): Boolean`

  Returns true if and only if the pitch of the note is not the maximum allowed pitch—i.e., there exists a note with a pitch above the pitch of this note.

- `fun hasNoteBelow(): Boolean`

  Returns true if and only if the pitch of the note is not the minimum allowed pitch—i.e., there exists a note with a pitch below the pitch of this note.

- `fun noteAbove(): Note`

  Returns a note with the same duration as this note, but with a pitch value one higher. This method can assume that `hasNoteAbove()` holds, without needing to check it.

- `fun noteBelow(): Note`

  Returns a note with the same duration as this note, but with a pitch value one lower. This method can assume that `hasNoteBelow()` holds, without needing to check it.

Test your solution using (at least) the tests in `Question1Tests`. You can also use the tests in `GoodPracticesTestsNote` to check whether you are following certain good coding practices related to visibility and mutability.

[20 marks]

2. **The `Tune` interface and a simple implementation.**

   For this question, you will first need to create an interface, `Tune`, specifying the following service:

   - Read access to an abstract property `notes` of type `List<Note>`, representing the list of notes that comprise the tune.
   - Read access to a default property `totalDuration` of type `Double`, which yields the sum of the durations of all notes in the tune.
   - An abstract method `addNote` that takes a `Note` parameter. In implementing classes, this method should lead to the given note being added to the list of notes that comprise the tune.
   - A default method that allows the notes in a tune to be iterated over using Kotlin's '`in`' syntax for looping over the elements of a collection.

   Next, create a class, `StandardTune`, that implements this interface.

   `StandardTune` should represent a tune as a mutable list of notes, but its `notes` property should only provide clients with an immutable view of this list.

   The `addNote` method in `StandardTune` should add the given note to the underlying mutable list.

   Test your solution using (at least) the tests in `Question2Tests`. You can also use the tests in `GoodPracticesTestsTune` and `GoodPracticesTestsStandardTune` to check whether you are following certain good coding practices related to visibility and mutability.

   [20 marks]

Note: Questions 3, 4, 5 and 6 are independent from one another and can be attempted in any order. You are encouraged to read each question briefly before attempting to solve any of them, so as to decide on the best order in which to approach them.

3. **Transposing tunes.**

   Transposing a tune involves shifting the pitch of each note in the tune up or down by a specified amount, leaving the duration of each note unchanged.

   Write a class, `TransposedTune`, that implements the `Tune` interface. A `TransposedTune` should be constructed from a `Tune`, the *target tune*, and an integer *pitch offset*, which may be positive, negative or zero.

   Instead of recording its own list of notes, the notes of a transposed tune should be derived from the notes of the target tune. That is, `notes` should yield the notes of the target tune, but with the pitch of each note shifted according to the pitch offset.

   Read access by a client to the notes of a `TransposedTune` should leave the notes of the target tune *unchanged*.

   If shifting the pitch of a note would lead to an out-of-range pitch value, the pitch value of the transposed note should be set to the minimum or maximum allowed pitch value, depending on whether the pitch is being shifted down or up.

   Calling `addNote(n)` on a `TransposedTune` should result in a note `m` being added to the *target* tune, such that when `m` is shifted by the pitch offset of the transposed tune, the resulting note is `n`. There are two exceptions to this: if the pitch of the required note `m` would be too high, a note with the highest allowed pitch and the same duration as `n` should be added to the target tune; similarly if the pitch of the required note `m` would be too low, a note with the lowest allowed pitch should be added.

   Important: when manipulating notes, the methods of `TransposedTune` should use only the public properties and methods of `Note` that you implemented in Question 1. You should not make any changes to `Note` in order to implement `TransposedTune`.

   Test your solution using (at least) the tests in `Question3Tests`. You can also use the tests in `GoodPracticesTestsTransposedTune` to check whether you are following certain good coding practices related to visibility and mutability.

   [20 marks]

4. **Stretching tunes.**

   Your code for Question 4 should be written in Java. If you wish, you may first write a working Kotlin solution and then port this to Java. However, no credit will be given for Kotlin code for this question.

   Stretching a tune involves scaling the duration of each note in the tune by a given positive factor, leaving the pitches of notes unchanged. This has the effect of slowing down the tune (if the scaling factor is larger than 1.0), or speeding up the tune (if the scaling factor is less than 1.0).

   Complete the provided skeleton Java class, `StretchedTune`, so that it implements the `Tune` interface. The `StretchedTune` class should be visible to code anywhere in your project, and it should not be possible to create sub-classes of `StretchedTune`. A `StretchedTune` should be constructed from a `Tune`, the *target tune*, and a *stretch factor* of type `double`. You may assume, but do not need to check, that the stretch factor is always positive.

   Instead of recording its own list of notes, the notes of a stretched tune should be derived from the notes of the target tune. That is, `notes` should yield the notes of the target tune, but with the duration of each note scaled according to the stretch factor.

Read access by a client to the notes of a `StretchedTune` should leave the notes of the target tune *unchanged*.

If scaling the duration of a note would lead to the duration exceeding the maximum allowed duration, the duration of the stretched note should be set to the maximum allowed duration.

Calling `addNote(n)` on a `StretchedTune` should result in a note `m` being added to the *target* tune, such that when `m` is scaled by the stretch factor the stretched tune, the resulting note is `n`. There is one exception to this: if the duration of the required note `m` would be too high, a note with the maximum allowed duration and the same pitch as `n` should be added to the target tune. You need not be concerned with issues related to floating-point round-off during this part of the exercise. This means you can ignore the fact that for certain floating-point numbers, round-off means that multiplying by a stretch factor and then multiplying by the inverse of that stretch factor will not yield the original floating-point number.

Because Java does not have properties, you will need to implement the properties of the `Tune` interface and any extra properties using suitable fields and methods.

You may find that your Java class does not inherit default methods from the `Tune` interface, and that calling these default methods using `super` does not work. In this case, you should implement functionally-equivalent versions of such methods in your Java class.

Important: when manipulating notes, the methods of `StretchedTune` should use only the public properties and methods of `Note` that you implemented in Question 1. You should not make any changes to `Note` in order to implement `StretchedTune`.

Test your solution using (at least) the tests in `Question4Tests`.

[10 marks]

5. **Thread-safe tunes.**

   This question involves writing a thread-safe tune: a `Tune` implementation that provides access to another tune, but protects all accesses to methods and properties of that tune using a lock. You will also write some test code that uses a thread-safe tune in a multi-threaded environment.

   Write a Kotlin class, `ThreadSafeTune`, that implements the `Tune` interface. A `ThreadSafeTune` should be constructed with a *target tune*: a reference to another `Tune` instance. Its other property should be a *lock*. Every method and property of the `Tune` interface should be implemented by acquiring the lock, delegating to the corresponding method or property of the target tune, and then releasing the lock. You should use the facilities that Kotlin provides to avoid explicit *lock* and *unlock* operations.

   Next, write a Kotlin class called `Composer` that can be executed by a Java `Thread` instance. A `Composer` should be constructed from a list of notes (of type `List<Note>`) and a target tune (of type `Tune`). The target tune can be any tune, and may or may not already contain some notes. A `Composer`'s job is to add the given list of notes to the target tune. Thus, when a `Composer` runs, it should iterate through the given notes, adding each note to the target tune.

   Finally, for this question, you should write some code to test whether your `ThreadSafeTune` class provides safe concurrent access by multiple `Composer`s.

   Un-comment the the `concurrencyTest` method in `Question5Tests`. This is an incomplete test that is designed to run 20 times. On each run, it constructs eight lists of notes.

Following the `TODO` comments in the incomplete test, flesh this test out so that:

- A `ThreadSafeTune` is constructed, providing thread-safe access to a `StandardTune`;
- Eight `Composer`s are created, one from each of the lists of notes;
- Eight `Thread`s are created, one from each `Composer`;
- All eight `Threads` are started;
- All eight `Threads` are joined.

The assertion at the end of each test run checks that, regardless of the order in which the composers added their notes to the thread-safe tune, the set of notes that end up in the tune (when order is ignored) should be the same.

Thus, test your solution using (at least) your fleshed out version of the test in `Question5Tests`. You can also use the tests in `GoodPracticesTestsThreadSafeTune` to check whether you are following certain good coding practices related to visibility and mutability.

[10 marks]

6. **A collection of songs.**

   Study the code in the Kotlin class `SongCollection`. This is a currently-incomplete class for representing a collection of `Song`s, where `Song` is a pair comprising a name and a tune.

   The songs in a `SongCollection` will be organised as a binary search tree, sorted by song name. The class includes a nested class, `TreeNode`, representing a node of the tree—the song at the tree node, and possibly-null left and right `TreeNode` children.

   The root of the tree is represented by an initially-null `root` property of type `TreeNode?`.

   Your task is to implement 3 methods to complete the implementation of a song collection:

   - `addSong`, which takes a name and a tune, and inserts a song with this name and tune into the tree. This method should throw an `UnsupportedOperationException` if it finds that a song with the given name is already present in the tree. Otherwise, a new node should be added to the tree at a suitable position. Remember that in a binary search tree, all elements in a node's left sub-tree should be smaller than the element at the node (according to the ordering used by the tree), while all elements in a node's right sub-tree should be larger than the element at the node.
   - `getTune`, which takes a name and searches for a song in the tree with the given name, returning the `Tune` associated with this song. If the tree does not contain any song with the given name, a `NoSuchElementException` should be thrown.
   - `getSongNames`, which returns a list of strings comprising the names of all songs in the tree, in sorted order. This should be achieved by performing an in-order traversal of the tree. An in-order traversal visits a node's left sub-tree, then visits the node, and then visits the node's right sub-tree. This kind of traversal guarantees that the elements in the tree are considered in ascending order (according to the ordering used by the tree). Thus, by construction, `getSongNames` will yield a sorted list of song names, if implemented correctly.

   Test your solution using (at least) the tests in `Question6Tests`. You can also use the tests in `GoodPracticesTestsSongCollection` to check whether you are following certain good coding practices related to visibility and mutability.

   [20 marks]

Total: 100 marks