Java Final Test

Imperial College London

Department of Computing

Java Final Test

Tuesday 02 May 2023
14:00 to 17:00
THREE HOURS
(including 10 minutes planning time)

- There are **TWO** parts: Section A and Section B, each worth 50 marks.
- The maximum total is **100 marks**.
- Credit will be awarded throughout for code that successfully compiles, which is clear, concise, *usefully* commented, and has any pre-conditions expressed with appropriate assertions.
- **Important:** It is critical that your solution compiles for automated testing to be effective. Up to **SIX MARKS** will be deducted from solutions that do not compile (-5 marks per Section). Comment out any code that does not compile before you log out.
- In each Section, the tasks are in increasing order of difficulty. **Manage your time so that you attempt both Sections**. You may wish to solve the easier tasks in both Sections before moving on to the harder tasks.
- 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. No additional marks are awarded for extra tests.
- You can use the terminal or an IDE like IDEA to compile and run your code. **You are advised to create a separate project for each Section**. Do not ask for help on how to use an IDE.
- The files are in your Home folder, under the "`javafinaltestparta`" and "`javafinaltestpartb`" subdirectories. **Do not move any files**, or the test engine will fail resulting in a compilation penalty.
- 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.

Useful commands

```
cd ~/javafinaltestparta/

javac
-g
-d out
-cp "./lib/*"
-sourcepath src:test
src/**/*.java test/**/*.java

java
-ea
-cp "./lib/*:out"
org.junit.runner.JUnitCore tunes.Question#Tests

java
-ea
-cp "./lib/*:out"
tunes.TestSuiteRunner



cd ~/javafinaltestpartb/

javac
-g
-d out
-cp "./lib/*"
-sourcepath src:test
src/**/*.java test/**/*.java

java
-ea
-cp "./lib/*:out"
org.junit.runner.JUnitCore
datastructures.AuxiliaryCollectionImplTest
datastructures.MemoryImplTest
datastructures.ChaosMonkeyTest

java
-ea
-cp "./lib/*:out"
datastructures.TestSuiteRunner
```

Section A

Problem Description

Your task is to write a number of classes and interfaces to represent musical 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 two implementations of this interface: a *standard* tune, comprising a list of notes, and a *transposed* tune, in which the pitch of every note is shifted by a specified amount.

Getting Started

The skeleton files are located in the `tunes` package. This is located in your Lexis home directory at:

- `~/javafinaltestparta/src/tunes`

During the test, you will populate the `Note.java` file, and add various other Java files.

You may feel free to add additional methods and classes, beyond those specified in the instructions, as you see fit, e.g. in order to follow good object-oriented principles, and for testing purposes. Any new Java files should be placed in the `tunes` package.

Testing

You are provided with a set of test classes under:

- `~/javafinaltestparta/test/tunes`

There is a test class `Question`*i*`Tests.java` for each question *i*. These contain initially commented-out JUnit tests to help you gauge your progress during Section A. **As you progress through the exercise you should un-comment the test class associated with each question in order to test your work.**

These tests are not exhaustive and are merely intended to guide you: while it is good if your solution passes these tests, your work will be assessed with respect to a larger, more thorough test suite. You should thus think carefully about whether your solution is complete, even if you pass all of the given tests.

What to do

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

   Flesh out the `Note` class to meet the following requirements:

   - The `Note` class should be immutable, and it should not be possible to create subclasses of `Note`.
   - A note should be represented by an integer pitch and an integer duration, which 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. The constructor of `Note` should throw an `IllegalArgumentException` if these requirements are not satisfied. Remember that this is an unchecked exception.

- The `toString()` method of `Object` 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 obtained by dividing its pitch by 12 (the number of notes in an octave). 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	`C0(4)`
1	3	`C#0(3)`
12	7	`C1(7)`
13	5	`C#1(5)`
23	11	`B1(11)`
53	12	`F4(12)`

Test your solution using (at least) the tests in `Question1Tests`.

[10 marks]

2. **Equality on notes.**

   Override the `equals` method of `Object` in the `Note` class so that two notes are deemed equal if and only if their pitches are the same and their durations are the same.

   Test your solution using (at least) the tests in `Question2Tests`.

   [6 marks]

3. **Additional `Note` methods.**

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

   - `boolean hasNoteAbove()`: 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.
   - `boolean hasNoteBelow()`: 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.
   - `Note noteAbove()`: returns a note with the same duration as this note, but with a pitch value one higher. The is method can assume (but need not check) that `hasNoteAbove()` holds.
   - `Note noteBelow()`: returns a note with the same duration as this note, but with a pitch value one lower. The is method can assume (but need not check) that `hasNoteBelow()` holds.
   - `int getDuration()`: returns the duration of this note.

   Test your solution using (at least) the tests in `Question3Tests`.

   [5 marks]

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

   Create an interface, `Tune`, specifying the following method signatures:

   - `List<Note> getNotes()`: in implementing classes, this method should return the list of all notes in the tune.
   - `void addNote(Note note)`: in implementing classes, this method should cause the given note to be added to the end of the tune.

   Create a class, `StandardTune`, that implements this interface.

   `StandardTune` should represent a tune as a list of notes.

   The `addNote()` method should add the given note to the underlying list.

   The `getNotes()` method should return a mutable list comprising all of the notes currently in the tune. However, if further notes are added to the list that `getNotes()` returns, this should *not* affect the notes of the tune.

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

   [8 marks]

5. **Transposing tunes.**

   Transposing a tune involves shifting the pitch of each note in the tune up or down by a specified amount, leaving the durations of notes 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, `getNotes()` should yield the notes of the target tune, but with the pitch of each note shifted according to pitch offset.

   A call to `getNotes()` on 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 clamped 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 methods of `Note` that you implemented in Questions 1–3. In particular, `TransposedTune` should not have access to the fields of `Note`, and you should not add any additional public methods to `Note`.

   Test your solution using (at least) the tests in `Question5Tests`.

   [15 marks]

6. **Calculating the duration of any tune.**

   Finally, add a method `int getTotalDuration()` that can be called on *any* `Tune` instance (i.e., both `StandardTune`, `TransposedTune`, and any future implementation of the `Tune` interface). This method should return the sum of the durations of all the notes in the tune.

   You should implement this method without making any changes to the `StandardTune` and `TransposedTune` classes.

   For full credit, your implementation of `getTotalDuration` should work by performing a reduction on a stream.

   Test your solution using (at least) the tests in `Question6Tests`.

   [6 marks]

Total for Section A: 50 marks

Section B

Problem Description

In several computing applications, we leverage a technique called *caching* to enable fast access of frequently used items. Using caching, when an item is read or written we store it in a small and expensive memory called **cache**. Caches are in-memory data structures which allow significantly faster *read* and *write* operations compared to reading and writing from the secondary storage (e.g. a harddisk or SSD). However, computer memory is expensive and as such memories tend to have restricted capacity. A side-effect of restricted capacity, is that we can quickly run out of space. To deal with this several *cache eviction* or *cache replacement* policies have been suggested which dictate how the cache should manage its space.

In *Section B* you are asked to complete the implementation of a cache data structure called *MemoryImpl* which enforces a **least recently used** (LRU) cache eviction policy. Such a data structure is also known as an LRU Cache. According to the LRU policy, to make space in a cache which is full, the item which was used (read or written) the least recently is evicted. This is known as the least recently used item.

To implement an LRU cache we will use the data structure shown in Figure 1. Items are stored in a *Collection* implemented using a doubly linked list. In a doubly-linked list we use nodes which have a reference of both their previous and their next node in the *Collection*. The `head` of the *Collection* always points to the *least recently used* (read or written) item. The `tail` of the *Collection* always points to the *most recently used* item. Reading an item which does not exist in memory is not allowed. Writing a new value for an item which already exists in memory is allowed. Figure 1 shows a sequence of `read` and `write` operations and their effect on our data structure.

To make our `read` and `write` operations more efficient, our `Memory` data structure uses a `HashMap` which helps us find nodes in the Collection in constant time. You are not required to implement a `HashMap`. Instead we will use the build-in `HashMap` from `java.util`. A `HashMap` allows storing key-value pairs. As shown in Figure 1 our Memory uses it to map the key of an item with its reference in the Collection.

The following are useful `HashMap` methods you can use:

- `map.remove(Object key)` Removes the mapping for the specified key from this map if present. Returns the previous value associated with key, or `null` if there was no mapping for key. (A `null` return can also indicate that the map previously associated `null` with key.)
- `map.put(K key, V value)` Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced. Returns the previous value associated with key, or `null` if there was no mapping for key. (A `null` return can also indicate that the map previously associated `null` with key.)

- `map.size()` Returns the number of key-value mappings in this map.
- `map.get(Object key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
- `map.containsKey(Object key)` Returns true if this map contains a mapping for the specified key.

Implementation Details. We will use a `DoublyLinkedNode` node to represent an item. The `AuxiliaryCollectionImpl` is a LinkedList-based collection of items currently in the cache. The `MemoryImpl` is the cache implementation which maintains a `HashMap` where the key is the `DoublyLinkedNode` key and the value is the node itself.

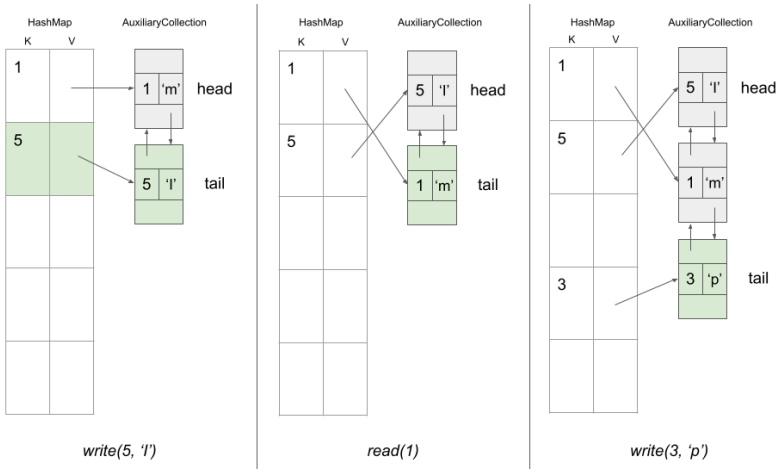

Figure 1: Memory Data Structure

Getting Started

The files used in Section B can be found in the SectionB directory in your Lexis home directory: `~/javafinalpartb/src/`

During Section B, you will be working in the following files:

- `AuxiliaryCollectionImpl.java`: class that implements the `AuxiliaryCollectionI` interface. It uses an implementation with doubly-linked Nodes. **You will need to complete two methods in this class.** This class has auxiliary methods: `find()` for finding if a node is in the *Collection*; `size()` which returns the value of the `size` class member variable which keeps track of the *Collection*'s size; *getHead()* which returns the *head* of the *Collection*; and `allKeysInOrder()` which traverses the *Collection* to find and store the keys of all *DoublyLinkedNode* nodes currently in the *Collection*.

- `MemoryImpl.java`: class that implements the `MemoryI` interface. **You will need to complete two methods in this class**. This class has auxiliary methods: `size()` for returning the current size, `getCapacity()` for returning the maximum capacity of the *Memory*, and `allMemKeysInOrder()` which returns the list of keys of all items currently in the underlying *Collection*.
- ***NOT PROVIDED*** `MemoryImplCoarseGrainedSync`: class that you need to create for a coarse-grained thread-safe implementation of the *Memory* (see *Task 3)*.

You may also need to make use of (though you should **NOT** need to edit) the following classes:

- `DoublyLinkedNode.java`: class that implements a Node. We will use these Nodes to represent the items our LRU cache stores. The *key* of the Node will represent the *id* of the item. The *key* of the Node should be unique in the *Memory* data structure. The *value* of the Node will represent the most recent value written for that item.
- `AuxiliaryCollectionI.java`: interface which represents a collection of methods to be used by a LinkedList implementation of the interface, such as the one in the *AuxiliaryCollectionImpl* class.
- `MemoryI.java`: interface which represents a collection of methods to be used by a Memory class, such as the `MemoryImpl` class.

Testing

You have been provided with a variety of tests to help you compile and test your code, located under the directory `test`. `TestSuiteRunner` currently runs a few sample tests from the `AuxiliaryCollectionImplTest` and the `MemoryImplTest` classes.

- When you finish *Task 1* you can test the `AuxiliaryCollectionImpl` *Collection* by uncommenting the tests in the `AuxiliaryCollectionImplTest` class.
- When you finish *Task 2* you can test the `MemoryImpl` *Collection* by uncommenting the tests in the `MemoryImplTest` class.
- When you finish *Task 3* you can test your concurrent version of the *Memory* data structure by uncommenting the tests in the `ChaosMonkeySyncTest` class and uncommenting the line in the `TestSuiteRunner` class.

The tests are a good starting point but you are encouraged to add your own tests.

What to do

Your must complete the provided implementation. You must also carefully read the commented source files.

Task 1. Complete `AuxiliaryCollectionImpl`

Look for the `TODO` statements in the `AuxiliaryCollectionImpl` class to locate the two following methods to complete:

1. `boolean append(DoublyLinkedNode toAdd)`. This should append a Node to the Collection.

   [5 marks]

2. `boolean remove(DoublyLinkedNode toRemove)`. This should remove the given Node from the Collection.

   [10 marks]

Task 2. Complete `MemoryImpl`

Look for the `TODO` statements in the `MemoryImpl` class to locate the two following methods to complete:

1. `Optional<V> read(K key)`. Reads an item with the given key (`K`) from the LRU memory.

   [10 marks]

2. `boolean write(K key, V value)`. Writes a value (`V`) to a memory location with key `K`.

   [10 marks]

Task 3. Thread-Safe Memory

Your last task is to make the *Memory* data structure you just created thread-safe.

1. **Coarse-grained:** Create at least one new class in its own file under the `datastructures` package called *MemoryImplCoarseGrainedSync.java*. This should contain the *MemoryImplCoarseGrainedSync* class, which should provide a coarse-grained thread-safe implementation of the `MemoryI` interface. The class *MemoryImplCoarseGrainedSync* is called in your helper test class `ChaosMonkeySyncTest` and by the autotesters to test your concurrent version of the *Memory* data structure.

   [10 marks]

2. **Fine-grained:** At the bottom of your *MemoryImplCoarseGrainedSync* file, write a comment explaining how you would go about implementing a fine-grained implementation of the `MemoryI` interface. Write 2-3 paragraphs, mentioning details such as which classes need to be changed, any new methods or member variables, your synchronization strategies, and how strictly your implementation respects the LRU cache policy. You need only discuss the methods mentioned in Tasks 1 and 2.

   [**5 marks**]

Total for Section B: 50 marks

Good Luck!