Kotlin Interim Test

Imperial College London

Department of Computing

"Text Files"

Friday 15 March 2024
14:00 to 17:00
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 "`kit`" 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 that represent *text files*. For simplicity, the exercise focuses on in-memory representations of text files; we shall not be concerned with reading or writing files to or from the hard drive.

A high level overview of your task is as follows, and the requirements are explained in more detail below. All the code you write should be Kotlin, except where the use of Java is explicitly required.

1. Write a simple implementation of text files, where the contents of the file is represented via a `StringBuilder` object **(10 marks)**.
2. Complete a more efficient implementation of text files, where a file is represented via multiple `StringBuilder` objects **(20 marks)**.
3. In Java, write a *lazy* implementation of text files, where a series of updates to a text file are queued, and only actually applied when their effects are needed **(15 marks)**.
4. Write a *thread-safe* implementation of text files, that uses locking to allow safe concurrent access by multiple threads, together with some code that demonstrates the use of a thread-safe text file in action **(20 marks)**.
5. Add the ability for text files to be *compared* according to their contents **(5 marks)**.
6. Write a *file map* class that uses a hashmap data structure to maintain a mapping from string file names to text files. **(30 marks)**

Getting Started

The skeleton files are located in `~/kit/src`, inside the Home folder.

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 `textfiles`. All functional code you write for this exercise will belong to the `textfiles` package, and thus should reside in one of these `textfiles` directories. A number of files are already present in these directories. Their purpose is explained in the detailed instructions below.

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 `textfiles` sub-directory. All provided tests plus any new tests that you create should be located here, in the `textfiles` 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 `textfiles` 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 code to these test 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

The test is split into six questions. It is suggested that you start with question 1 as it is the easiest, and most of the other questions depend on it.

After question 1, you may attempt the remaining questions in any order as they do not depend on one another, except that the tests for question 5 rely on a solution to question 2 being in place.

All questions depend on a `TextFile` interface, with which you are provided. Study this interface, reading the comments associated with each method and property carefully. Comments in the `TextFile` interface refer to a custom exception class, `FileIndexOutOfBoundsException`, which is also provided.

1. **Single string text files**

   To start with you will write an implementation of `TextFile` that represents a text file using a single `StringBuilder` object. If you cannot remember how to use `StringBuilder` then you may use `String` instead, but you will not get full credit.

   - Create a new class, `SingleStringTextFile`, implementing the `TextFile` interface.
   - `SingleStringTextFile` should be visible anywhere in your project and it should not be possible to create subclasses of `SingleStringTextFile`.
   - The class should use a `StringBuilder` property to represent the contents of the text file.
   - A `SingleStringTextFile` should be constructed from a `String` that provides the initial contents of the text file.
   - Implement the `length` property and the `insertText` and `deleteText` methods according to the comments in the `TextFile` interface.
   - The string representation of a `SingleStringTextFile` is simply the contents of the file, i.e. the contents of the `StringBuilder` property.

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

   [10 marks]

2. **Multi string text files**

   A problem with `SingleStringTextFile` is that it is inefficient. Inserting or deleting text requires shifting all subsequent content. `StringBuilder` takes care of the details of this memory copying, but for large files the performance overhead could be problematic.

The provided `MultiStringTextFile` class is an incomplete implementation of `TextFile` that has the potential to be more efficient. Instead of using a *single* `StringBuilder` to represent a text file, this class uses a *list* of `StringBuilder`s. In this way, the file is split into a sequence of blocks of text such that each `StringBuilder` represents a block.

An insertion operation thus only affects the block of text associated with the insertion point, and a deletion operation only affects the blocks that represent the range of characters to be deleted.

The `BLOCK_SIZE` constant (declared as a top level property) indicates the preferred number of characters in each block of text. For simplicity we use a small block size in this exercise; a larger block size would likely be more efficient in practice.

After a series of insertion and deletion operations, the sizes of the blocks that underlie a `MultiStringTextFile` may end up deviating considerably from `BLOCK_SIZE`. A currently un-implemented method, `rebalance`, is intended to address this by reorganising blocks so that the content of a `MultiStringTextFile` is represented by blocks of size exactly `BLOCK_SIZE` (except that the final block may have a smaller size if the length of the file is not a multiple of `BLOCK_SIZE`). Clients of `MultiStringTextFile` can call `rebalance` periodically if they have efficiency concerns. Notice that the constructor of `MultiStringTextFile` initialises the `blocks` property so that the file is represented by a *single* block. It then immediately calls `rebalance` so that this single block is split into multiple blocks if needed.

- Read the comment associated with `rebalance` carefully and then implement this method.
- A `MultiStringTextFile` should be turned into a string by concatenating the string representations of the blocks that underlie the file, in order.
- Implement the `length` property so that the length of a `MultiStringTextFile` is computed as the sum of the lengths of the underlying blocks. You should achieve this *without* concatenating blocks (i.e. it is not acceptable to return the length of the string representation of the `MultiStringTextFile`).
- Implement `insertText` so that it meets the specification described by the comment for this method in the `TextFile` interface. When `insertText` is called to insert some text at position `offset`, the text should be inserted into the block that currently contains the character at position `offset`. This may cause a block to end up having a size different from `BLOCK_SIZE`; this is OK (i.e. your implementation of `insertText` should *not* call `rebalance`). If `offset` is exactly the length of the file then a new block should be added containing the inserted text.

  For example, suppose a `MultiStringTextFile` currently uses three blocks to represent the content `"Alphanum P4rty"`:

  ```
  [['A', 'l', 'p', 'h', 'a', 'n', 'u', 'm'], [' ', 'P', '4', 'r'], ['t', 'y']]
  ```

  Then inserting `"OK"` at offset 2 should lead to:

  ```
  [['A', 'l', 'O', 'K', 'p', 'h', 'a', 'n', 'u', 'm'], [' ', 'P', '4', 'r'], ['t', 'y']]
  ```

  Instead, inserting `"OK"` at offset 8 should lead to:

  ```
  [['A', 'l', 'p', 'h', 'a', 'n', 'u', 'm'], ['O', 'K', ' ', 'P', '4', 'r'], ['t', 'y']]
  ```

  Or, instead, inserting `"OK"` at offset 14 should lead to a new block being added:

  ```
  [['A', 'l', 'p', 'h', 'a', 'n', 'u', 'm'], [' ', 'P', '4', 'r'], ['t', 'y'], ['O', 'K']]
  ```

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

[20 marks]

3. **Lazy text files**

   You should implement this part of the exercise using Java. If you wish, you may choose to get your implementation working using Kotlin first and then port the Kotlin code to Java, but credit will only be given for Java code.

   Suppose multiple pieces of text are inserted into a text file at the same location. Rather than repeatedly updating the text file at this location, it might be more efficient to combine the insertions into a *single*, larger insertion and perform it only when needed. Your task now is to create a *lazy* text file class that provides this optimisation as a wrapper around an existing text file. You should write this class in Java.

   - You are provided with an empty Java class, `LazyTextFile`. Edit this class so that it implements the `TextFile` interface. Make the class visible across the entire project, and make it impossible to create subclasses of `LazyTextFile`.
   - A `LazyTextFile` should hold a reference to a *target* `TextFile`, which should be provided on construction.
   - Additionally, a `LazyTextFile` should have fields to capture (a) whether an insertion into the target text file is pending, and (b) if so, details of the pending insertion. Initially, no insertion should be pending.
   - The `getLength` method (which corresponds to the `length` property in the Kotlin `TextFile` interface) and the `deleteText` method should first flush any pending insertion by applying it to the target text file, and then call the corresponding method of the target text file. Similarly, when the string representation of a `LazyTextFile` is requested, any pending insertion should be flushed, and the string representation of the target text file should be returned.
   - When `insertText` is called with arguments `offset` and `toInsert`, if there is already a pending insertion whose offset is equal to `offset` then this pending insertion should be updated to incorporate the additional text from the `toInsert` parameter. The pending insertion should be updated so that its effect, if applied, would be the same as that of flushing the existing pending insertion and then applying the new insertion. Otherwise, any existing pending insertion should be flushed, and the `LazyTextFile` should be updated so that an insertion of `toInsert` at `offset` is pending.

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

   [15 marks]

4. **Thread-safe text files**

   You should use Kotlin to implement the remaining questions in the test.

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

   Write a Kotlin class, `ThreadSafeTextFile`, that implements the `TextFile` interface. A `ThreadSafeTextFile` should be constructed with a *target* `TextFile`: a reference to another `TextFile` instance. Its other property should be a *lock*. Every method and property of the `TextFile` interface, as well as the method that provides a string representation of

a text file, should be implemented by acquiring the lock, delegating to the corresponding method or property of the target text file, 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 `Author` that can be executed by a Java `Thread` instance. An `Author` should be constructed from:

- a list of strings (of type `List<String>`);
- a target text file (of type `TextFile`);
- a random number generator (of type `java.util.Random`).

The target text file can be any text file, and may or may not already contain some text. An `Author`'s job is to insert the given strings one by one at random valid offsets in the target text file. The `java.util.Random` class provides a `nextInt` method that takes an integer argument n and returns a random integer in the range $[0, n)$.

Finally, for this question, you should write some code to test whether your `ThreadSafeTextFile` class provides safe concurrent access by multiple `Author`s. To do this, un-comment the `concurrencyTest` method in `Question4Tests`. This is an incomplete test that is designed to run 20 times. On each run, it constructs eight lists of strings. Following the comments in the incomplete test, flesh this test out so that:

- A `ThreadSafeTextFile` named `threadSafeTextFile` is constructed, providing thread-safe access to the existing `SingleStringTextFile` named `singleStringTextFile`;
- Eight `Author`s are created, one from each of the lists of strings, each with access to the text file and each with its own instance of a `java.util.Random` object;
- Eight `Thread`s are created, one from each `Author`;
- 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 authors added their strings to the thread-safe text file, the same characters should ultimately occur in the text file.

Test your solution using (at least) this fleshed out version of the test in `Question4Tests`.

[20 marks]

5. **Implementing `Comparable`**

The generic `Comparable<T>` interface allows instances of a class to be compared with objects of type `T` according to a programmer-defined ordering. If a class implements `Comparable<T>` then it must provide a method with the following signature:

```
override fun compareTo(other:  T): Int
```

This method should return a negative integer, zero, or a positive integer according to whether the receiving object is regarded as less than, equal to, or greater than `other`.

You will now add support for comparing text files, so that two text files are ordered according to their string representations.

- Adjust the `TextFile` interface so that it extends the `Comparable<T>` interface with respect to a suitable concrete type in place of `T`.

- Make it so that `TextFile`s are ordered based on their string representations, where strings are ordered using the `compareTo` method of the `String` class.
- In achieving this, make as few changes as possible to files other than `TextFile`. In particular, you should not have to change any other Kotlin files, though (due to limitations of Kotlin/Java interoperability) you may need to change the Java class of Question 3.

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

[5 marks]

6. **A file map**

Your task in this question is to implement a hashmap data structure that provides a mapping between file names (of type `String`) and text files (of type `TextFile`). In this part of the exercise you may make use of Kotlin's list interfaces and classes. However, you must not make use of any Kotlin classes or interfaces related to maps or sets.

Write a class, `FileMap`, that uses a hashmap data structure (details of which are described below) to provide the following service to its clients:

- Read access to a `size` property that indicates how many entries are present in the map. Retrieving the `size` of the map should not involve traversing the contents of the map.
- A function `get` that takes a `String` argument, representing a file name. The function should return null if the map does not contain an entry that maps this file name to a `TextFile`. Otherwise, the function should return the `TextFile` associated with the file name.
- A function `set` that takes a `String` and a `TextFile` as arguments, representing a file name and a text file, respectively. If the map already contains an entry associated with the given file name, this entry should be replaced with an entry that associates the file name with the given text file. Otherwise, a new entry should be added to the map associating the given file name with the given text file.

You should use operator overloading so that array indexing notation (square brackets) can be used in place of `get` and `set`.

Regarding how you should implement the hashmap data structure on which your `FileMap` class should be based: `FileMap` should maintain a list of *buckets*, where each bucket is a list of (file name, text file) pairs.

Initially, a `FileMap` should contain a list of four empty buckets.

The bucket in which a (file name, text file) pair should be stored, and thus also searched for, should be determined by computing a *bucket index*: the hash code of the file name, modulo the number of buckets. The relevant bucket is then the bucket at this index of the list of buckets.

When a `set` operation adds a brand new entry to the map (rather than merely replacing an existing entry), the size of the `FileMap` should be increased. If the size is found to be larger than 0.75 times the number of buckets, the hashmap should be *resized*. This involves:

- Recording all entries that are currently in the `FileMap` in some temporary storage, e.g. in a list;

- Replacing the current list of N buckets with a list of $2 \cdot N$ empty buckets;
- Re-hashing all of the recorded entries so that each is placed in an appropriate bucket with respect to the larger list of buckets.

Finally, you should add support for *iterating* over `FileMap`s. To do this, add an `iterator` method to `FileMap`. This method should return an `Iterator` object that gives access to the (file name, text file) pairs contained in the `FileMap`.

For partial credit you may implement `iterator` by constructing a list of all entries in the `FileMap` and returning an iterator for this list.

However, for full credit you should instead implement an "on demand" iterator that traverses the hashmap lazily, yielding entries one by one, each time the `next` method is called. This is rather tricky to implement, and is intended to be a stretch goal for which only a small number of marks will be reserved, so you are advised to only attempt implementing this kind of iterator if you have successfully completed the rest of the test.

Regardless of which approach you take, use appropriate Kotlin features such that it is possible to iterate over the entries of a `FileMap` using Kotlin's "`in`" keyword.

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

[30 marks]

Total: 100 marks