Imperial College London

Department of Computing

"Kotlin Collections"

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 "`kotlin-lexis-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

This test is about the implementation, testing and fixing of various collections classes. It will involve:

- Writing additional unit tests for a linked list implementation.
- Writing a resizing array list implementation from scratch.
- Writing an extension method on a list interface that allows a specific use of subtyping.
- Converting a linked list implementation to Java.
- Identifying and fixing various problems with a hashmap implementation.
- Writing a thread-safe "striped" hashmap implementation.

Getting Started

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

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 `collections`. All functional code you write for this exercise will belong to the `collections` package, and thus should reside in one of these `collections` 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 `collections` sub-directory. All provided tests plus any new tests that you create should be located here, in the `collections` 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 `collections` package for the appropriate programming language.

Testing

You are provided with various test classes containing test cases to help you gauge your progress. Some of these tests are commented out because they rely on code that you will implement during the exercise. **As you progress through the exercise you should use these test classes to 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. **Writing high-coverage unit tests for a linked list class.**

   Look at the `ImperialMutableList` interface. This is a simplified version of the interface of the same name that was studied in the lectures. Look at the `SinglyLinkedList` class. This is an implementation of `ImperialMutableList` similar to the singly-linked list implementation that was studied in the lectures, but a bit simpler: the implementation used here does not feature a property representing the tail of the list.

   The test class `SinglyLinkedListTests` features a set of unit tests for `SinglyLinkedList`. However, these unit tests provide no coverage of the `removeAt` and `remove` methods.

   Your first task is to write a series of additional unit tests in the `SinglyLinkedListExtraTests` file that thoroughly test the `removeAt` and `remove` methods of `SinglyLinkedList`. The `SinglyLinkedListExtraTests` file contains a placeholder test; you should remove this and replace it with a number of new tests, according to the following guidelines:

   - Use your own judgement to decide how many tests to write, up to a maximum of 10 tests in total.
   - The tests you write for `removeAt` should confirm that the method behaves correctly when invoked with both in-bounds and out-of-bounds indices.
   - The tests you write for `remove` should confirm that the method behaves correctly when invoked with an element that *does* appear in the list, and when invoked with an element that *does not* appear in the list.
   - Each test should be designed with a specific purpose, and the name of the test should reflect this purpose.
   - Think about "edge case" inputs to these functions when deciding on test inputs.

   As you write new tests it is a good idea to make temporary changes to the `removeAt` and `remove` methods of `SinglyLinkedList` to confirm that introducing errors into the implementations of these methods would lead to some of your new tests failing.

   However, you should not make any permanent changes to `SinglyLinkedList`, so it is a good idea to create a backup of this file that you can restore after you have finished writing your new tests.

   [10 marks]

2. **Writing a resizing array-based list class.**

   Your next task is to write an alternative list implementation: a resizing array-based list, as studied extensively during the course. To this end, complete the skeleton class `ResizingArrayList` so that it implements the `ImperialMutableList` interface to provide this kind of list implementation. Follow these guidelines when designing your implementation:

   - It should be possible to construct a `ResizingArrayList` with a given initial capacity. If a negative value is provided for the initial capacity then an `IllegalArgumentException` should be thrown.
   - It should be possible to construct a `ResizingArrayList` without specifying an initial capacity. In this case, a default initial capacity of 16 should be used.

- The string representation of a `ResizingArrayList` should comprise the elements of the list, each separated by a comma and a space, enclosed in square braces.
- Remember that the Kotlin function `arrayOfNulls` cannot be used to create an array with element type `T?`, but that it is acceptable to use this function to create an array with element type `Any?` and then downcast this to an array with element type `T?`.
- When an element is added to the list, if the current list capacity is too small to accommodate the new element then the array that backs the list should be resized. The resized array should be at least large enough to store the elements of the enlarged list, and at least double the size of the array that previously backed the list.
- Your `iterator` implementation should provide access to elements of the list in an "on demand" fashion. That is: you should *not* simply copy the list elements into a Kotlin collection and return an iterator for that collection.

Test your solution using the tests in `ResizingArrayListTests`, which you will need to uncomment.

To test your solution more thoroughly it is a good idea to create versions of the tests you created in `SinglyLinkedListExtraTests`, modified to work with `ResizingArrayLists` instead of `SinglyLinkedLists`. If you do this, add the resulting tests to `ResizingArrayListTests`. You will not receive marks for including these modified tests, but they might help you to catch bugs in your implementations of `removeAt` and `remove` in `ResizingArrayList`, and you will *lose* marks if your solution exhibits such bugs.

[25 marks]

Note on question ordering: At this point it is up to you in which order you attempt:
- Question 3
- Question 4
- Questions 5 and 6 (Question 6 depends on Question 5)

3. **A `removeAll` extension method.**

In the `ImperialMutableListUtilities.kt` file, write an extension method called `removeAll` on `ImperialMutableList<T>`. This method should take an `ImperialMutableList` containing elements of type `T` as a parameter, and should remove all elements of this list from the receiving list.

Use the Kotlin type system to design `removeAll` so that it can accept a parameter of type `ImperialMutableList<S>` for any subtype `S` of `T` when the receiver has type `ImperialMutableList<T>`. For example, if `list1` has type `ImperialMutableList<Any>` and `list2` has type `ImperialMutableList<String>` it should be possible to write:

```
list1.removeAll(list2)
```

to remove each element of `list2` from `list1`.

Only a small amount of credit will be given for solutions that do not meet this subtyping requirement.

Test your solution using the tests in `ImperialMutableListUtilitiesTests`, which you will need to uncomment.

[5 marks]

4. **Writing a Java version of the `SinglyLinkedList` class.**

   The given file `SinglyLinkedListJava.java`, located under `src/main/java/collections`, contains an incomplete Java class, `SinglyLinkedListJava`. This class contains the beginnings of a Java version of the Kotlin `SinglyLinkedList` class. Your task in this question is to complete this Java implementation.

   Because the Java standard library does not feature a *pair* data type, you are provided with a simple pair class, `ImperialPair`, which you should briefly study.

   The `SinglyLinkedListJava` class is already equipped with:

   - A static nested class `Node`, which is equivalent to the `Node` nested class in `SinglyLinkedList`.
   - Two fields, `size` and `head`, and method, `getSize`. These correspond to the `size` and `head` properties of `SinglyLinkedList`.
   - A complete implementation of `toString`.
   - Java versions of the `checkIndexInBounds`, `traverseTo` and `unlink` helper methods, where `traverseTo` uses the `ImperialPair` helper class.
   - A partial implementation of `iterator` that returns an instance of an anonymous class that implements the `Iterator` interface. This is analogous to the anonymous object returned by `iterator` in `SinglyLinkedList`. However, the anonymous class currently has no fields, and its methods are stubs.

   Your job is to:

   - Implement all of the stub methods of `SinglyLinkedListJava`, each of which currently throws a `RuntimeException` with a "TODO" message.
   - Complete the anonymous class used in the implementation of `iterator` by equipping it with suitable fields and filling out the stub methods.

   Test your solution using the tests in `SinglyLinkedListJavaTests`.

   To test your solution more thoroughly it is a good idea to create versions of the tests you created in `SinglyLinkedListExtraTests`, modified to work with `SinglyLinkedListJava` objects instead of `SinglyLinkedList`s. If you do this, add the resulting tests to `SinglyLinkedListJavaTests`. You will not receive marks for including these modified tests, but they might help you to catch bugs in your implementations of `removeAt` and `remove` in `SinglyLinkedListJava`, and you will *lose* marks if your solution exhibits such bugs.

   [20 marks]

5. **Fixing problems in a hashmap implementation.**

   Look at the `ImperialMutableMap` interface. This is similar to the map interface that you studied during the lab exercises for the course. Look at the `Hashmap` class. This is an attempt at implementing `ImperialMutableMap` to provide a hashmap implementation. It makes use of a file, `HashmapConstants.kt`, which contains constants and type aliases on which the `Hashmap` class depends. The buckets of a `Hashmap` are `ImperialMutableList` objects, and the `bucketFactory` function property of `Hashmap` is used to create new buckets when they are required.

   Briefly look at the test class `ImperialMutableMapTestsParent` and its `HashmapTests` subclass. The abstract parent class provides a suite of tests for checking the correctness of an `ImperialMutableMap` implementation. The child class puts these tests into practice to

allow testing of the `Hashmap` class, by implementing two `emptyCustomMutableMap...` methods so that they return `Hashmap` objects.

The given `Hashmap` class suffers from several problems:

- It has *correctness* problems. For example, the '`test entries after some putting, removing and setting`' test case fails.
- It has *performance* problems. For example, the '`performance test 1`' and '`performance test 2`' tests do not complete within reasonable time.
- It does not follow the good software engineering practices that you were taught during the course.

By carefully studying the `Hashmap` class, and by carefully understanding why provided test cases fail or under-perform, identify and fix as many problems with this class as you can find.

For each problem you identify, as well as fixing the problem in the code of the class, use the comment block at the top of the class to write a brief summary of the problem and a brief summary of how you solved the problem.

If you have successfully fixed the correctness and performance problems, you should find that the tests in `HashmapTests` (including the tests from `ImperialMutableMapTestsParent` that this class inherits) run efficiently and all pass. However, the problems that relate to good software engineering practices do not cause test failures, so while adapting the class so that all tests pass is good, it is not enough to get full credit for this question.

[20 marks]

6. **Writing a thread-safe "striped" hashmap.**

   If possible, make sure that you have adapted the `Hashmap` class of Question 5 so that it is functionally correct and efficient before embarking on this question: ideally the tests in `HashmapTests` (including the inherited tests from `ImperialMutableMapTestsParent`) should run efficiently and all pass.

   However, if you are having real trouble identifying and fixing the problems with the `Hashmap`, feel free to move on to this question despite the remaining `Hashmap` problems.

   Your task in this question is to create a thread-safe hashmap implementation that uses the "striped" hashmap design that you studied during the lab sessions for the course. An extra challenge in this question that was not part of the related lab exercise is that the `ImperialMutableMap` interface exposes the size of the hashmap to its clients via the `size` property.

   See Appendix A for a reminder of relevant background on striped hashmaps, including details of how buckets should be protected by locks and how resizing should work.

   Write your striped hashmap implementation in the `StripedHashmap.kt` file, which initially contains a placeholder class. Your `StripedHashmap` class should implement the `ImperialMutableMap` interface, and construction of a `StripedHashmap` should work similarly to construction of a `Hashmap`.

   When filling out the properties and methods of `StripedHashmap`, start by following the approach provided in `Hashmap`, but corrected to avoid the problems you identified in Question 5. Adapt the approach as needed to make suitable use of locks in order to ensure thread safety. A number of imports that you may find useful are commented out at the top of the `StripedHashmap.kt` file.

Like `Hashmap`, your `StripedHashmap` implementation should make use of the helper file `HashmapConstants.kt`.

It is expected that there will be quite a bit of duplication of code between `Hashmap` and `StripedHashmap`. This is OK, and for simplicity you should **not** attempt to reduce this duplication (for example, do **not** make `StripedHashmap` a subclass of `Hashmap`).

Because it will be possible for multiple threads to add and remove elements from the hashmap concurrently, you need to be careful about data races on the property that records the size of the map. The size of the map should be exposed to clients as an `Int` as required by the `ImperialMutableMap` interface. However, instead of using a regular `Int` to represent the size of the map internally, use the `AtomicInteger` class from `java.util.concurrent.atomic`. This provides access to an integer value that can be incremented or decremented using special methods `incrementAndGet()` and `decrementAndGet()`. These have the effect of incrementing/decrementing the counter in an indivisible, atomic way, avoiding the possibility of increments being lost due to data races between threads. The methods return the value of the counter after the increment/decrement operation has completed. The `AtomicInteger` class also has a `get()` method to read the value of the integer without modifying it.

Test your code using the `StripedHashmapTests` class, which you will need to uncomment. This is a subclass of `ThreadSafeImperialMutableMapTestsParent`, which is in turn a subclass of `ImperialMutableMapTestsParent`. This means that `StripedHashmapTests` inherits all of the unit tests for standard maps (from `ImperialMutableMapTestsParent`), plus unit tests for thread-safe maps (from `ThreadSafeImperialMutableMapTestsParent`). You will see that the tests for thread-safe maps feature multiple repeat runs, to help guard against the problem where test failures are intermittent due to nondeterministic concurrency bugs.

[20 marks]

Total: 100 marks

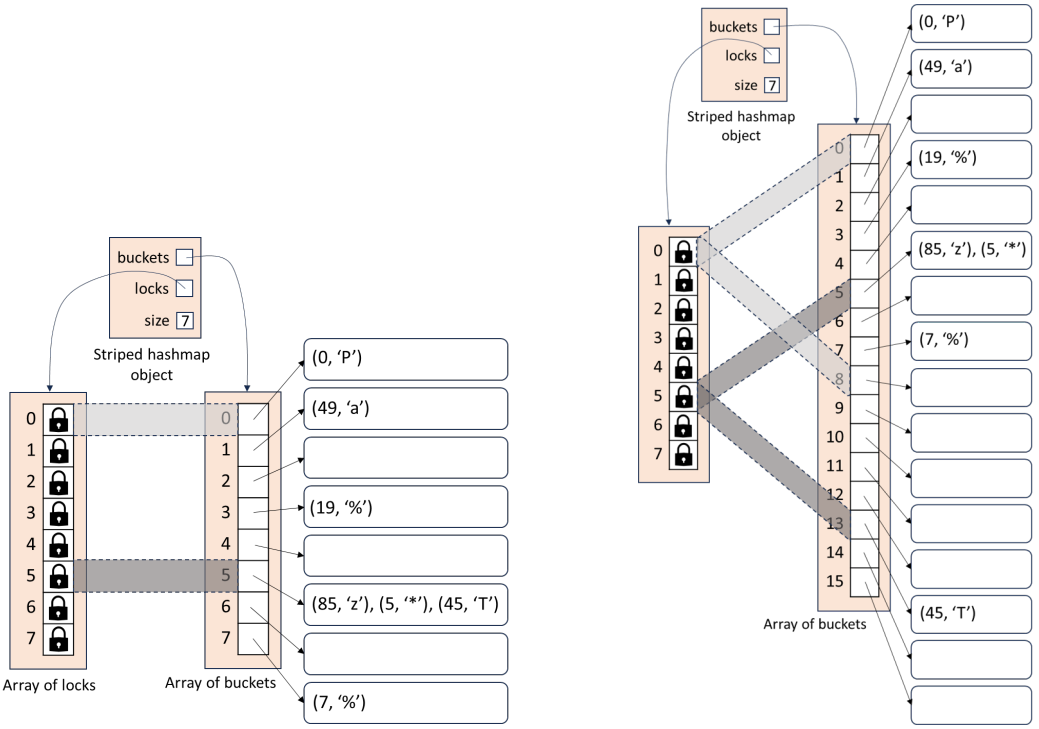
A Background on striped hashmaps

A striped hashmap maintains an array of buckets (as per a standard hashmap) and an array of locks. Initially, each bucket is protected by a separate lock, so that multiple threads can access distinct buckets concurrently. When the hashmap is resized, the number buckets doubles (as per a standard hashmap) but the number of locks remains fixed, and each lock is responsible for protecting twice the number of buckets that it previously protected: after the first resize each lock protects two buckets, after the second resize each lock protects four buckets, etc.

This is illustrated by the diagrams of Figure 1.

Figure 1(a) shows a striped hashmap that was initialised with 8 buckets and 8 locks, and the ith bucket is protected by the ith lock. The figure uses two stripes to illustrate the relationship between locks and buckets: the light stripe indicates that lock 0 protects bucket 0, and the darker stripe indicates that lock 5 protects bucket 5.

Figure 1(b) depicts the hashmap after a resize operation. The contents of the map are the same, but the number of buckets has doubled and the entries have been re-hashed over this larger array of buckets. There are still 8 locks, but now each lock protects 2 buckets: the ith and $(i + 8)$th buckets are protected by the ith lock. The figure uses four stripes to illustrate this: the light stripes indicate that lock 0 now protects buckets 0 and 8, and the darker stripes indicate that lock 5 now protects buckets 5 and 13.



(a) A striped hashmap with 8 locks protecting 8 buckets. The hashmap is ready to be resized.

(b) After resizing, each lock now protects twice the number of buckets than it previously did.

Figure 1: Illustration of striped hashmaps, adapted from Figure 13.6 of "The Art of Multiprocessor Programming" by Herlihy and Shavit

In more detail, a striped hashmap works as follows:

- In addition to an array of buckets, the hashmap stores an array of locks, one for each bucket initially present in the hashmap. So, if the hashmap initially has 32 buckets it should be equipped with an array of 32 locks. In what follows, call the original number of hashmap buckets N.
- Initially, the lock at position i in the array protects bucket number i: any operation that will access bucket i should do so by first locking lock i, and on completion should release lock i.
- When a thread determines that the hashmap needs to be resized, it should acquire all N bucket locks in order. This is because resizing involves accessing *all* buckets of the hashmap, and no bucket should be tampered with until exclusive access to all buckets has been ensured. Once all locks have been acquired, the thread should check whether the hashmap *still* needs to be resized: it could be that in the meantime another thread has already performed a resize operation. Once the thread has finished resizing (or confirmed that no resize is needed after all) it should release all of the locks.
- Resizing should double the number of buckets but should *not* double the number of locks. That is, even though the number of buckets will grow from N to $2N$ to $4N$, etc., the number of bucket locks will always be N.
- After each resize, every lock should protect twice the number of buckets that it previously protected: after the first resize, the lock at position i should protect bucket i and bucket $i+N$. After the second resize, this lock should protect bucket i, $i+N$, $i+2N$ and $i+3N$, etc., i.e. lock k protects bucket i if and only if $k = i \bmod N$.