Week 3

3.1

Section 2

The thread safe class has no concurrent execution of the of method calls or field access (read/write) which can result in race conditions. Our class uses Semaphores:
-Binary Semaphores (set to 1)
-capacity Semaphore (set to desired capacity of our data structure in our case is linked list)
-Semaphore with available items
The threads go to the insert method where we check is they can:

- Add element to the list (we try to run request to acquire from capacity Semaphore)
- Access the add function by acquiring the binary Semaphore
    - If they can acquire the element is added to the list

If everything went fine the thread release the binary Semaphore and release item from available items Semaphore

The threads go to the take method where we check is they can:

- Remove element form the available items Semaphore
- Access the remove function by acquiring the binary Semaphore
    - If they can acquire the element is removed from the list

If everything went fine the thread release the binary Semaphore and release item from capacity Semaphore

The thread safe class analysis includes:
- **class state**: no concurrent access to shared state (state are the fields defined in the class which could be shared between many threads) In case of our exercise the shared field is LinkedList.
- **escaping:** it is important not to expose the shared state (in our case the LinkedList which is made private, we don't return reference to the list object from any of the methods)
- **safe publication:** We initialize the LinkedList in the constructor before making the object reference accessible
-**object initialization and visibility:** the thread which executes the constructor will not have problems with seeing the value of the fields but other threads with a reference to the fields might not see the instantiated values. In order to avoid it we used the final keyword on the fields of the class. This approach works because they implement happens-before relation (it reads after the constructor finishes, on JVM level final can not be cached or reordered during initialization but only if the constructor doesn't leak the reference to the object).
-**immutability:** In case of our exercise we have a final LinkedListand field called list whose reference is immutable (we change only the content of the list). The make sure that the class is immutable we need to make sure that: (we should think about it)

- fields can not be modified after publication,
- objects are safely published
- access to inner mutable object do not escape

-(if class is state mutable) mutual exclusion: only one thread at the same time can change the state. In order to enforce it the shared state has to be protected by the locks.

Section 3

The barriers can not be used because they await a certain number of threads. If for example we would have a barrier that awaits 5 threads and we insert onl;y four times the barrier wouldn't allow the rest of the threads to proceed and they would starve.

3.2

2. Constructors are thread safe because we are using synchronized keywords. The threads will get the current value of the id and increment it.
4. We didn't find the error but it may be the case of not testing the implementation with enough race conditions.

Week 4

4.1.

1. Implement a functional correctness test that finds concurrency errors in the `add(Integer element)` method in `ConcurrentIntegerSetBuggy`. Describe the interleaving that your test finds.

ANSWER:

In the implementation of the tests it was decided not to use the size() method from the ConcurretnIntegerSetBuggy class for two reasons:

1. The first one is that while testing a specific function of a class there should not be used another function of the same class because the second function could have bugs too.
2. And the second reason is because even if the size function was correctly implemented it still could not check if the add function added more elements than it should.

The test fails because the HashSet util is not thread safe. The reason is that if thread A tries to add an object(a) to the Set the add() function will check if the object(a) exists as key to the hashMap (hashset are hashMaps that use the value as a key) then the add function will return true (which means that an object is add to the set). But if at the same time thread B tries to add the object(a) and this action happens before thread A finishes then thread B will add object(a) successfully. Although if we print the set we will probably see the set correctly (meaning that we will have only one instance of object(a)) the return value of the add() function will be wrong.

2. Implement a functional correctness test that finds concurrency errors in the remove(Integer element) method in ConcurrentIntegerSetBuggy. Describe the interleaving that your test finds.

ANSWER:

similarly , to the previous answer but here the problem will be the remove() function of the set that could return true when it should not.(for more details look above)

3. In the class ConcurrentIntegerSetSync, implement fixes to the errors you found in the previous exercises. Run the tests again to increase your confidence that your updates fixed the problems. In addition, explain why your solution fixes the problems discovered by your tests.
The problem in the previous exercises was with reading outdated content of the set. After applying synchronized keywords to add, size and remove functions we make sure that value will not be removed or added many times and that the size of the set is accurate.

4. Run your tests on the ConcurrentIntegerSetLibrary. Discuss the results. Ideally, you should find no errors—otherwise please submit a bug report to the maintainers of Java concurrency package :)

All tests passed 14000/14000

5. Do a failure on your tests above prove that the tested collection is not thread-safe? Explain your answer.
ANSWER:
If it fails once, it proves to us that it is not thread safe. This means that one of the threads did not did not access the shared variable correctly.

6. Does passing your tests above prove that the tested collection is thread-safe (when only using add() and remove())? Explain your answer.
ANSWER:
No, we cannot prove that the tested collection is thread-safe, since we cannot reproduce the appropriate environment to prove it is indeed thread-safe.