Exercise 10.2

Question 1

We started by creating the constructor, which simply creates an array of `TxnInteger` and use the factory method `newTxnInteger` to individually creating each of them. This is not done within an `atomic` block, since the constructor is not accessed concurrently.

The `increment` and `getCount` simply wrap the code in an `atomic` block from the Multiverse API. As the value in `getSpan` is final, we simply return this value.

Question 2

Yes, we got the correct solution. We double checked the values with our earlier implementation of `Histogram` from Exercise 3.

Question 3

For `getBins` we create a `final` array with the same span as the histogram. We then atomically retrieve each bin count individually in a loop. This gives us a lot of short transactions, instead of getting all the values in one long transaction, thus making retries less costly.

We have used this method in a new `dump2` method, to ensure that it works as intended. The method is used at the end of `countPrimeFactorsWithStmHistogram`.

Question 4

The `getAndClear` method simply caches the previous value before setting it to zero and returning the cached value. This is all wrapped in an `atomic` block.

It seems the TxnInteger supports this atomic action already, but we found the documentation to be too insufficient and even conflicting to be sure that we got the correct results.

Question 5

The `transferBins` method works on each bin seperately as in `getBins`. We use the `getAndClear` method to get and reset the bin value, and atomically increment the bin value in the current histogram.

Question 6

We introduced a new histogram `total`, which is updated every 30 ms. using `transferBins`. We use the `getNumberWaiting` method on the `stopBarrier` to see when all the threads are done, so we can stop the loop. At the end we dump both histograms to ensure `total` contains all counts and that `histogram` is empty.

Question 7

This shouldn't have any effect other than probably a slight decrease in performance. Each value will be atomically read and cleared, and then the retrieved value will be added to the current value, thereby restoring the difference. We therefore do not lose any values.

This has no effect on our implementation.

Exercise 10.3

Question 1

We have chosen to implement `get` by using two calls to `atomicWeakGet` on `buckets` and `bs[bucket]`. In this way we don't have to enclose everything in the method in an `atomic` block.

Question 2

We have implemented `forEach` similarly to `get` by using `atomicWeakGet` to get a reference to the array of `TxnRef`s. We use this reference when we retrieve each bucket. Again, we use `atomicWeakGet` to retrieve the first node before even calling the `consumer`. We iterate the immutable chain and call the `consumer` on each node. This ensures that `consumer` is only called once for each node.

Question 3

We were not able to rely only on the predefined methods, such as `setAndGet`, on `TxnRef` when implementing `put`, `putIfAbsent` and `remove`. This was because retrieve the node and either delete it or insert a new node in the same atomic operation. We need the return value to know what to put back in the bucket, so an atomic operation would not be possible. Instead we wrap the critical part in an `atomic` block, and do the necessary manipulations there.

We currently do not expect the value of `buckets` to ever change, as we do not reallocate it anymore, so we don't wrap the retrieval of it's value in any atomic blocks.

Question 4

We instantiated a variable `size` as a `TxnInteger` containing the total number of entries. This is called in all of the methods created in question 3 where we decrement or incement it respectively. Using this `TnxInteger` could cause a bottleneck as it is frequently called.

Question 5

We have drafted some code showing how we think a blocking mechanism could work. The code is found in a section labeled `CONCEPT CODE FOR 10.3.5` in the source file `TestStmMap.java`.

The idea is to keep a boolean transaction variable called `isReallocating` which is set to true only when some thread is reallocating the `buckets` array.

Methods that need to block while the reallocation is happening, must start out by checking if any reallocation is currently happening. An example can be seen in the `blocksWhileReallocating` method. We make an atomic block and start out by checking the value of `isReallocating`. If the value is true, we call `retry()` in order to block and wait for the value to change again, so we can do our work. This call is followed by the method's actual code. Should it be the case that reallocation has started when reaching the end of the transaction, the transaction will be aborted and restarted and the `retry` method will be called (provided that the value still hasn't changed) thereby explicitly blocking until the reallocation is done.

The `reallocateBuckets` method, which is responsible for resizing the buckets array, has to block all other methods that try to write until it is done. It starts out by checking if `isReallocating` is true. If this is the case, another thread is already resizing, therefore we simply return to avoid doing the same work twice. Otherwise, it sets the variable to true and exits the atomic block to ensure the value is visible to the other threads immediately. This will cause writing methods to block from now on. We then start to do the reallocation and overwrite the `buckets` variable atomically with the new array using the `atomicSet`. Finally we atomically set the value of `isReallocating` to false, so the blocking threads can continue their work.