3.1)
3.1.1 ) So i created a class that implements the required interface and I overran the given methods so that we have a “put” like method and a “take” like method. I wrapped the main action of these methods in if conditions so that they can only occur if a specific condition is met, in this case, having enough items to take out from and the amount of elements not being bigger than the capacity of the buffer. The main reason I did this was so I could throw an error on the other side of these conditions, in case they were not met. For the synchronization, I used two semaphores. One semaphor is a mutex semaphor that only allows one thread to operate on the shared data (the bounded buffer) at a time. The other one is a count semaphor, so that one of the operations can only acquire that and the other can only release it. This way, the number of operations has to be equal: a take cannot occur until a previous put is in place.

3.1.2)
I tested my code with more threads and it does not throw any of my errors, but errors only show the presence of a bug not the absence of it. My code is safe as it runs around thread safe classes. Mainly and most importantly, our bounded buffer class. Ill break it down why the bounded buffer is thread safe.

1) Such class is thread safe as its fields suffer from no data racing when run in a concurrent scenario. The methods that alter the fields are all safe as they need a mutex lock to be ran and another lock, the count lock, to make sure that the invariants of the class are maintained. The fields are therefore safely published, any thread that has to access them can do so without stale data or having a specific interleaving of actions that could compromise the running of the program.
2) Not only the fields are accessed by methods that are concurrent safe, the fields cannot be accessed in any other way, as they are private. There is no possibility of escaping.
3) Everything that can be immutable, is immutable so that there is less state to think about when in a multi thread scenario. In this case, the capacity of the buffer is a final field.

On top of the BoundedBuffer being thread safe, its also a correct program. It maintains its main characteristic in any state situation. It always has to put out as much as puts in, the total value produced equals the value consumed. This is guaranteed by the count semaphore. One of the threads performs the method that adds one to the lock and the other thread performs the method that removes one from the lock.There must be something in the buffer so that something can be removed, therefore, and the consumers are forced to wait for the producers.

3.1.3) I believe this program could have been made only using barriers but it would lead to more cumbersome code. This is due to the nature of the problem where there must be some sort of control that depends on the alternation of threads, which I believe would be harder to replicate using only barriers. We could use a countdown latch to wait for the producer before the consumer does anything. This would work but it will likely have more specific conditions that wouldnt be necessary using a semaphore. If you have a producer thread that always puts the same or more in the buffer than the consumer takes away, then this would work fine. But more complex scenarios where the consumers have to wait to one of the producers

to create enough locks so they can use the locks and put more stuff in the buffer could be a little harder.

3.2)
I created a Person class and added a static field to track how many people were created. I made the possible fields immutable to easy the transition of the class to a thread safe class.