SWE 619 Assignment 8

// TODO

Goal: Generics

Consider the BoundedQueue example from in-class exercise #7C.

Complete the generic part of the exercise: The result should be fully generic, and there should not be any compiler warnings. You should adopt Bloch's advice about lists vs. arrays; doing so will eliminate the need for many of the instance variables.

Keep the same methods, but update the behavior (and document with contracts!) to include exception handling for all cases not on the happy path.

Include the constructor in your considerations. In particular, consider whether you think a zero-sized buffer is a reasonable possibility. Document your reasoning.

Implement repOk() and toString(). (Note that adopting Bloch's advice has the beneficial side effect of greatly simplifying the rep-invariant.)

Add putAll() and getAll(). Define the method signatures carefully. Use exception-handling consistent with that for get() and put(). Use bounded wildcards as appropriate.

As before, turn in a clear, concise *story* demonstrating completion of the assignment.

Note: There are quite a few decisions in this assignment. I expect to see robust Piazza discussions.

We need to create a new class **BoundedQueue** and write JUnit tests to test the corresponding class. Hence, I first created a **new gradle application** as follows:

- Open command prompt
- `cd` to the directory where you want to initialise your project
- Type `gradle init`
- Follow the onscreen instructions and Gradle should create a workspace for you

```
> Task :init
Get more help with your project: https://docs.gradle.org/5.6.2/userguide/tutorial_java_projects.html

BUILD SUCCESSFUL in 1m 17s
2 actionable tasks: 2 executed
C:\Users\haram\OneDrive\Desktop\SWE 619 Assignment 5>
```

Now, we have our project ready to roll!

LET'S GET STARTED!!!

1. Creating the class BoundedQueue:

The BoundedQueue class is already outlined for us. We need to simply generify the class and implement additional functionalities along with the various methods it contains. Bag uses Map data structure.

2. Adding Generics to the BoundedQueue class:

We added generics to the BoundedQueue class. We followed Bloch's advice and replaced arrays with generic lists for better performance. Since we were using Lists, we could end up dropping some of the instance variables too.

```
public class BoundedQueue <E>{
private List<E> rep;
private int size;

public BoundedQueue(int length)
{
    this.size = length;
    this.rep = new ArrayList<E>(length);
}

public int count()
{
    int k =0;
    for(E i:rep)
    {
```

```
            if (i != null) {
                k++;
            }
        }
        return k;
    }

    public boolean isFull()
    {
        if(count() == size)
            return true;
        else
            return false;
    }

    public boolean isEmpty()
    {
        if(count()==0)
            return true;
        else
            return false;
    }

    public void put(E e)
    {
        if(e != null && !isFull() )
            rep.add(e);
        else
            throw new IllegalArgumentException("No element");
    }

    public int size()
    {
        return rep.size();
    }

    public E get() {
```

```
        if(!isEmpty())
        {
            E result = null;
            result = rep.get(0);
            rep.remove(0);
            return result;
        }
        else
            throw new IllegalArgumentException("No element");
    }
}
```

3. Implementing the `putAll()` and `getAll()` methods

```
    public void putAll(List<E> al) {
        if (al.size() >= getSize())
            rep.addAll(al);
        else
            throw new IllegalArgumentException("No element to put");
    }

    public List<E> getAll() {
        List<E> ll = new ArrayList<E>();
        if (!isEmpty()) {
            ll.addAll(rep);
            rep.clear();
        } else {
            throw new IllegalArgumentException("No elements to
get");
        }
        return ll;
    }
```

4. Implementing the `repOk()` method

```
public boolean repOk() {
    for (E elem: rep) {
        if (elem == null) {
            return false;
        }
    }
    if (getCount() > getSize()) {
        return false;
    }
    return true;
}
```

5. Implementing the `toString()` method

```
public String toString() {
    return rep.toString();
}
```

TESTING:

1. Test `put()`

```
@Test
public void testPut() {
    BoundedQueue<String> bqueue = new BoundedQueue<>(5);
    bqueue.put("Haramrit");
    assertEquals(bqueue.get(), "Haramrit");
}
```

2. Test `get()`

```
    @Test
    public void testGet() {
        BoundedQueue<String> bqueue = new BoundedQueue<>(5);
        bqueue.put("Neel");
        assertEquals(bqueue.get(), "Neel");
    }
```

3. Test `putAll()`

```
    @Test
    public void testPutAll() {
        BoundedQueue<String> bqueue = new BoundedQueue<>(3);
        List<String> original = new ArrayList<>();
        original.add("Tejasvi");
        original.add("Rashi");
        original.add("Shruthi");
        bqueue.putAll(original);
        assertEquals(original, bqueue.getAll());
    }
```

4. Test `getAll()`

```
    @Test
    public void testGetAll() {
        BoundedQueue<String> bqueue = new BoundedQueue<>(5);
        List<String> expected = Arrays.asList("Neel", "Nabeel",
"Ajit");
        bqueue.put("Neel");
        bqueue.put("Nabeel");
        bqueue.put("Ajit");
        assertEquals(expected, bqueue.getAll());
    }
```

Conclusion:

Thus we implemented a generified version of BoundedQueue class. We wrote method implementations to perform additional operations such as getAll, putAll, repOk, etc. We also followed Bloch's advice while generifying which made things a lot easier, especially by reducing the amount of instance variables to be defined.