SWE - 619 Final Exam

Tuljasree Bonam
G01179672

1A.

A-1.1-

`enQueue()`

Partial Contract

- Precondition
  The requirement would be such that **e** should not be null.
- Postcondition
  e will be added to the list.

```
 public void enQueue (E e) {
      elements.add(e);
      size++;
      }
```

Total Contract

- Precondition
  No precondition as this is total contract
- Postcondition
  If e is null, we throw IllegalArgumentException
  If not, we add e to the list.

```
 public void enQueue (E e) {
      if(e == null)
      throw new IllegalArgumentException();
      elements.add(e);
      size++;
      }
```

A-1.2-

Rep Invariant according to the contract would be to not have the arraylist be null and also to maintain the size in the range of 0 to the size of the list .

```
      elements != null && (size >= 0 && size <= elements.size())
```

A-1.3-

We can implement this similarly as we did in Stack class. Explanation is in the comments-

```
@Override public String toString() {
      String result = "size = " + size; //to print size as the
following manner- e.g.,size = 4
      result += "; elements = ["; //result representation
```

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

```
for (int i = 0; i < elements.size(); i++) { //iterating the list to
append each value with a “,” at the end
         if (i < elements.size()-1)
            result = result + elements.get(i) + ", ";
         else
            result = result + elements.get(i);
      }
      return result + "]"; //we return the final string with “]” at
the end of the output string
   }
```

A-1.4-
Explanation is in the comments

```
public E deQueueAll () {
     //if (size == 0) throw new
IllegalStateException("Queue.deQueue"); -not needed since we assume
that size > 0 and deQueue() has already checked this.
  while(size > 0){
      deQueue();
      }
   }
```

Contract
Precondition
- Size should be greater than 0.

Postcondition
- List will be emptied/dequeued completely.

A-1.5-
Immutable version of deQueue(). Here, we are not disturbing the original Queue. Instead, creating a copy of it (creating a new list and storing all the values of elements into it) and making necessary changes to it.

```
 public List<E> deQueue(List<E> elements) {
     if (isEmpty()) throw new IllegalArgumentException();
      List<E> copyQueue = new ArrayList<>(elements); //clone
     E result = copyQueue.get(0);
     copyQueue.remove(0);
     return new ArrayList<E>(copyQueue);
   }
```

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

A-1.6-
Explanation is in the comments

```
public List<E> clone(List<E> elements){
 ArrayList cloneQueue = new ArrayList();
 cloneQueue = (ArrayList)elements.clone();
//creating a new ArrayList and cloning the value of elements (input)
into the cloneQueue at the initialization stage
return cloneQueue; //returning the cloned queue
}
```

2A.

```
import java.util.*;
import java.util.concurrent.*;

// Bloch's final version
public class GenericChooser<T> {
   private final List<T> choiceList;

   public GenericChooser (Collection<T> choices) {
      choiceList = new ArrayList<>(choices);
   }

   public T choose() {
      Random rnd = ThreadLocalRandom.current();
      return choiceList.get(rnd.nextInt(choiceList.size()));
   }
}
```

According to the discussions in class,

CONTRACT- PRECONDITIONS AND POSTCONDITIONS (for all the below questions)

(i) **GenericChoose**r

- Preconditions-
  - This doesn't need to specify any requirements.
- Postconditions
  - If choices is null, throw IAException
  - If `choices` is empty, throw IAException.
  - If choices.contains(null), throw exception NPE
  - Create choiceList with choices

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

(ii) **Choose**

- Precondition
    - This doesn't need to specify any requirements
- Postcondition
    - Returns random choice in List<T> choiceList

A-2.1-
According to the preconditions and postconditions assumed, the rep invariant would be-

```
choiceList != null && size(choices) > 0
```

This makes sure the choiceList is not null and is not empty.

A-2.2-
We can recode and add the following because with the given postconditions assumed, we would have to change the GenericChooser in this way-

```
 public GenericChooser (Collection<T> choices) {
      if(choices == null) throw new IllegalArgumentException();
//ADD
      if(choices.size() == 0) throw new
IllegalArgumentException(); //ADD
      if(choices.contains(null)) throw new NullPointerException();
 choiceList = new ArrayList<>(choices);
      }
```

And for the choose, if our preconditions and postconditions are assumed differently and our rep invariant would have resulted in such way- `choiceList != null && size(choices) >= 0`
We would have to modify the chooser in this way-

```
   public T choose() {
      if(choiceList.size() == 0) throw new
IllegalArgumentException();//ADD
       Random rnd = ThreadLocalRandom.current();
       return choiceList.get(rnd.nextInt(choiceList.size()));
   }
```

But, since our rep invariant is assumed to be correct, we don't have to change anything for the choose() method.

A-2.3-

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

According to the rep invariant and the postconditions I assumed for the constructor, which already checks if the input list is empty or null or even contains a null entry in the input, we can be assured that the choiceList will never be empty or null or even contain null entries. Therefore, the choose() method can be left as it is without any modifications further. Therefore, this all depends on how we assume our postconditions. Had my postconditions or rep invariant been assumed any differently, like mentioned in the above question i.e., (**`choiceList != null && size(choices) >= 0)`**, we would be required to add another piece of code which checks if the choiceList is empty in this way-

```
   public T choose() {
      if(choiceList.size() == 0) throw new
IllegalArgumentException();//ADD
       Random rnd = ThreadLocalRandom.current();
       return choiceList.get(rnd.nextInt(choiceList.size()));
   }
```

But since we already checked that, it can be noticed that we don't have to make any additional changes to the choose() method.

3A.

```
import java.util.*;

public class StackInClass {
   private Object[] elements; private int size = 0;

   public StackInClass() { this.elements = new Object[0]; }

   public void push (Object e) {
     if (e == null) throw new NullPointerException("Stack.push");
     ensureCapacity(); elements[size++] = e;
   }

   public void pushAll (Object[] collection) { for (Object obj:
collection) { push(obj); } }

   public Object pop () {
     if (size == 0) throw new IllegalStateException("Stack.pop");
     Object result = elements[--size];
     // elements[size] = null;
     return result;
   }
```

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

```
   @Override public String toString() {
      String result = "size = " + size;
      result += "; elements = [";
      for (int i = 0; i < elements.length; i++) {
         if (i < elements.length-1)
            result = result + elements[i] + ", ";
         else
            result = result + elements[i];
      }
      return result + "]";
   }
  private void ensureCapacity() {
      if (elements.length == size) {
         Object oldElements[] = elements;
         elements = new Object[2*size + 1];
         System.arraycopy(oldElements, 0, elements, 0, size);
      }
   }
}
```

A-3.1-

In stack, as the array keeps updating with push() and pop() methods, we need to make sure that we are keeping a track of the top. That can be done by making use of a size variable. But in the toString() method, we are iterating the entire array instead of till the top. So, we can change it to size instead of elements.length in the following piece of code-

```
  @Override public String toString() {
      String result = "size = " + size;
      result += "; elements = [";
      for (int i = 0; i < size; i++) {
         if (i < size-1)
            result = result + elements[i] + ", ";
         else
            result = result + elements[i];
      }
      return result + "]";
   }
```

A-3.2-

Since global variables are almost always prone to violating the encapsulation concepts and since the pushAll() method makes use of the push() method, there are high chances of overriding the original push() method. Therefore, if we made use of the **protected/final** keyword for the push() method, we can be assured that we wouldn't alter the original push() method.

Contract
Precondition- None
Postcondition- If an element or all the elements in the array is null, throw NullPointerException. Will push all the elements we receive in the input.

A-3.3-
We can maintain another place holder stack that stores all the elements from our stack (array) and would be returning this new stack instead of the object in the end when the pop() method is called. However, we can have a separate method to return the popped element. Also, we have the size stored in a new variable. By doing so, we are not disturbing the original stack and thus preserving the immutability property.

```
  public Object[] pop (Object[] elements) {
     if (size == 0) throw new IllegalStateException("Stack.pop");
      Object[] newArray = elements; // creating the placeholder stack
      Int newArrSize = size - 1; //updating the size
      newArray[newArrsize] = null; //nullify the end value as we are
popping it off
     return newArray;
   }
```

A-3.4-
Usage of lists over stacks would definitely be advantageous since lists are dynamic in nature and shrinking/expanding them would be a lot easier comparatively. Also, in lists, we can directly make use of list's inbuilt/default .equals() method instead of implementing a user defined equals() like in the case of arrays. In arrays, a null value at the end wouldn't be counted when we calculate it's size which would result in something like the following-
arrayNumber1[1,2,3] and arrayNumber2[1,2,3,null] would have the same size and the equals() method would be violated.

4A.
A-4.1-
- Yes, it satisfies the given specification.
- Let's begin with the precondition i.e., y>=1 and initially start off with y = 1 and x = 0 as given.
- After first entering the while loops- (0<1), x becomes 2.
- Now this satisfies the postcondition i.e.,x becomes greater than y
- Therefore, it's proved that the specification is satisfied by the program

A-4.2-

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

Loop Invariants-

- x>=0
- y>=1
- True
-

According to the Hoare's condition, we know that,
WP(while[I] B do S , {Q}) =

1. i and
2. (i & B) => WP(S,I)
3. (i & !B) => Q

A-4.3-

```
 WP(x:=0; while[x >= 0] x < y do x= x + 2, {x> =y}) =

  1. x >= 0

  2. (x >= 0 & x<y)    => WP(x+=2, {x >= 0})
       By simplifying it further, we get
       x<y =>  x+2 >= 0
       x<y => (True)

3.  x >= 0 & !(x<y) => x>=y
       We can rewrite this as
        x>=y & x>=y => x>=y
        x=y => x=y (True)
Now,
x >= 0 && True && True
we get x>=0

b)    WP(x:=0; {x >= 0})=
By substituting x we get
      0 >= 0

c) Verification
(P => wp(..)) P => WP(x := 0; while[x >= 0] x<y do x= x + 2,{x>=y})=
P => x >= 0 =( True) Holds.
```

Therefore, it's significantly strong.

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

A-4.4-

```
WP(x:=0; while[y>=1] x < y do x= x + 2, {x> =y}) =
  1. y>=1

  2. (y>=1 & x<y)   => WP(x= x + 2, {y>=1})
      (y>=1 & x<y) =>  y>=1 (False)
      We Cannot simplify this further.
       Which is false

  3. y>=1 & !(x<y) => x>=y
      We can rewrite this as
       y>=1 & x>=y => x>=y (False)


(i)Since the WP of the while loop is False, WP of the entire program
is also False.

WP(x:=0; WP{False}) = WP{False}

(ii)Verification
(P => wp(..))
P => False
y>=1 => False
False
```

As we cannot determine the validity of the loop invariant using Hoare triple, this is insufficiently strong.

5A.

A-5.1-

According to the lecture class, for example, in the below piece of code-

```
{n >=0} //Condition for N
i := 0;

while(i < n){
i := n
}
Condition- i == n
```

Loop Invariant for the above-

- I <= n
- I >= 0

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

- n >= 0
- True

According to the Hoare's condition, we know that,
WP(while[I] B do S , {Q}) =
1. i and
2. (i & B) => WP(S,I)
3. (i & !B) => Q

Now from the above,

```
WP(while[i <= n]  i < n do i := n, {i == n})
1.    i <= n
2.    (i <= n & i < n) => WP(i := n, {i <= n})
i < n => n <= n
i < n => True
True
3.    i <= n & !(i < n) => i == n
i == n => i == n
True
```

As it holds True for i <= n, we can assert that the program is correct.

(ii) **`{n >=0} //Condition for N`**
```
i := 0;

while(i < n){
i := i * 2
}
Condition- i == n
```

According to the Hoare's condition, we know that,
WP(while[I] B do S , {Q}) =
1. i and
2. (i & B) => WP(S,I)
3. (i & !B) => Q

Now from the above,

```
WP(while[i <= n]  i < n do i = i * 2, {i == n})
1.    i <= n
```

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

```
2.    (i <= n & i < n) => WP(i := i * 2, {i <= n})
i < n => i * 2 <= n
False
3.    i <= n & !(i < n) => i == n
False
```

A-5.2-

Loop Invariant-

A statement that must be true when entering a loop and after every iteration of the loop.
Example-

```
{n >=0} //Condition for N
i := 0;

while(i < n){
i := n
}

Condition- i == n
```

Loop Invariant for the above-

- I <= n
- I >= 0
- n >= 0
- True

Rep Invariant-

A Rep Invariant (RI) is one that maps rep values to booleans:

RI : R → boolean

For a rep value r, RI(r) is true if and only if r is mapped by Abstract Function (AF). In other words, RI tells us whether a given rep value is well-formed. Alternatively, you can think of RI as a set: it's the subset of rep values on which AF is defined.

Both the rep invariant and the abstraction function should be documented in the code, right next to the declaration of the rep itself-

Example-

For the Queue class, the rep invariant would be,

```
(i)elements != null
(ii)size >= 0 && size <= elements.size()
```

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

Contract-
The contract of a class or interface refers to the publicly exposed methods (or functions) and properties (or fields or attributes) of that class interface along with any comments or documentation that apply to those public methods and properties.

For example,
```
import java.util.*;
import java.util.concurrent.*;

// Bloch's final version
public class GenericChooser<T> {
   private final List<T> choiceList;

   public GenericChooser (Collection<T> choices) {
      choiceList = new ArrayList<>(choices);
   }

   public T choose() {
      Random rnd = ThreadLocalRandom.current();

      return choiceList.get(rnd.nextInt(choiceList.size()));
   }
}
```

Contract-

(i) **GenericChoose**r
- Preconditions-
  - This doesn't need to specify any requirements.
- Postconditions
  - If choices is null, throw IAException
  - If `choices` is empty, throw IAException.
  - If choices.contains(null), throw exception NPE
  - Create choiceList with choices

(ii) **Choose**
- Precondition
  - This doesn't need to specify any requirements
- Postcondition
  - Returns random choice in List<T> choiceList

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

A-5.3-

From what I understand, with Parameterized tests (JUnit Tests) you can supply a series of static inputs to a test case. JUnit Theories are similar but different in concept. The idea behind them is to create test cases that test on assumptions rather than static values. So if the supplied test data is true according to some assumptions, the resulting assertion is always deterministic. One of the driving ideas behind this is that you would be able to supply an infinite number of test data and your test case would still be true. Also, often you need to test a universe of possibilities within a test input data, for example, negative numbers. If you test that statically, that is, supply a few negative numbers, it is not guaranteed that your component will work against all negative numbers, even if it is highly probable to do so. According to my understanding, JUnit frameworks try to apply theories' concepts by creating all possible combinations of your supplied test data. Both should be used when approaching a scenario in a data-driven scenario (i.e only inputs change, but the test is always doing the same assertions over and over). But, since theories seem experimental, I would use them only if I needed to test a series of combinations in my input data. For all the other cases I'd use Parameterized tests.

For example,
JUnit Theory
```
@RunWith(Theories.class)
 public class UserTest {

      @DataPoint
       public static String GOOD_USERNAME = "optimus";
       @DataPoint
       public static String USERNAME_WITH_SLASH = "optimus/prime";

       @Theory
       public void filenameIncludesUsername(String username) {
           assumeThat(username, not(containsString("/")));
           assertThat(new User(username).configFileName(),
containsString(username));
       }
 }
```

Standard JUnit Tests-
```
package demo.tests;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
public class JUnitProgram {
```

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

```
    @Test
    public void test_JUnit() {
        System.out.println("This is the testcase in this class");
        String str1="This is the testcase in this class";
        assertEquals("This is the testcase in this class", str1);
    }
}
```

`Also,`

On a input with 4 values and 2 test methods

`@RunWith(Theories.class)` - will generate 2 JUnit tests

`@RunWith(Parameterized.class)` - will generate 8 (4 inputs x 2 methods) JUnit test

A-5.4-
Testing

- Observable properties
- Verify program for one execution
- Manual development with automated regression
- Most practical approach now

Proving

- Any program property
- Verifies program for all executions
- Manual development with automated proof checkers
- May be practical for small programs after a long period
- Although proofs aren't practical, they tell us how to think about program correctness, important for development, inspection, foundation for static analysis tools

The goal of Hoare logic is to provide a compositional method for proving the validity of Hoare triples. That is, the structure of a program's correctness proof should mirror the structure of the program itself. We deduce the formal reasoning about program's correctness using pre- and postconditions
• Syntax: {P} S {Q}
• P and Q are predicates
• S is a program
• If we start in a state where P is true and execute S, then S will terminate in a state where Q is true. If we cannot prove using Hoare Logic, the program does nothing.

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

A-5.5-
According to Liskov, Substitutability is a principle in object-oriented programming stating that an object and a sub-object must be interchangeable without breaking the program.
For example, in mathematics, a Square is a Rectangle. Indeed it is a specialization of a rectangle. The "is a" makes you want to model this with inheritance. However if in code you made Square derive from Rectangle, then a Square should be usable anywhere you expect a Rectangle. This makes for some strange behavior. Imagine you had SetWidth and SetHeight methods on your Rectangle base class; this seems perfectly logical. However if your Rectangle reference pointed to a Square, then SetWidth and SetHeight doesn't make sense because setting one would change the other to match it. In this case Square fails the Liskov Substitution Test with Rectangle and the abstraction of having Square inherit from Rectangle is a bad one.
Few other examples-
Bad example-

```
public class Bird{
 public void fly(){}
}
public class Duck extends Bird{}
```

The duck can fly because it is a bird, but what about this-

```
public class Ostrich extends Bird{}
```

Ostrich is a bird, but it can't fly. Ostrich class is a subtype of class Bird, but it shouldn't be able to use the fly method, that means we are breaking the LSP principle.

Good example-

```
public class Bird{}
public class FlyingBirds extends Bird{
 public void fly(){}
 }
public class Duck extends FlyingBirds{}
 public class Ostrich extends Bird{}
```

6.
Group Members-
Tuljasree Bonam- Regular participant; contributed reliably
Rachana Thota- Regular participant; contributed reliably
Srikar Reddy Karemma- Regular participant; contributed reliably

SWE - 619 Final Exam

Tuljasree Bonam
G01179672

Srujan Reddy Tekula- Regular participant; contributed reliably

7.

a) Everything was perfect. Professor has managed to make all the students engage and collaborate with each other despite online classes.
b) The fact that I enjoyed working in teams and got to collaborate every week has been the best part the professor managed to ace.
c) To be honest, nothing.