Advait Gogte
G01301827

SWE619: Final Exam

Question 1

Queue.java

1.
   a. Partial contract:
      Precondition – input should not be NULL
      Postcondition – input element is appended to the list.
      (No changes in the code)
   b. Total contract:
      Precondition – True
      Postcondition – if input element is NULL, throw IllegalArgumentException, else, input element is appended to the list.
      Here, I changed the code so that the precondition allows all inputs, but to maintain the integrity of the code, it checks for the NULL input. This made the postcondition stronger.

```java
public void enQueue(E e) {
  if (e == NULL) {
    throw IllegalArgumentException("Input cannot be null")
  } else {
    elements.add(e);
    size++;
  }
}
```

2. The rep-invariants are:
   a. e!=NULL
      The element ‘e’ that needs to be added should not be NULL
   b. 0<=size<=element.length
      We also need to make sure that the current size of the queue (given by the variable size) should not exceed the length of the ArrayList provisioned for the queue.

3. toString() code

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

I borrowed this code from in-class exercises of this course and made some modifications.

In this toString() implementation, first we'll print out the size of the queue using the 'size' variable. Then we will iterate through the elements of the queue. If the index of the element is smaller than the size of the queue, we will append it to the result. In the end, we will return the string 'result'.
To make the output easy to read, I have included some formatting options such as square brackets and commas.

4. dequeueAll()
    Precondition: queue should not be empty
    Postcondition: queue is empty, and all elements are added to a collection.

```java
public void deQueueAll(Collection<? super E> dQelements)
{
    while(size>0)
    {
        dQelements.add(deQueue());
    }
}
```

For this code, I referred Bloch's Item 31.
Here, the deQueueAll() method dequeues all elements one by one and adds them to the collection called 'dQelements'. The type of input parameter to deQueueAll() should not be some 'collection of E' but 'collection of some supertype of E'. I took inspiration from Bloch's item 31.

5.
Precondition: True
Postcondition: If size == 0, throw IllegalStateException, else create a temporary list, add elements from queue to this list, remove the first element (element 0), return this list as a generic ArrayList.
Here, this method is immutable, as we do not modify the original queue, instead we make a new List each time and copy the contents and modify this new List and return it.

```java
public ArrayList<E> deQueue(List<E> elements)
{
    if(size==0)
    {
        throw new IllegalStateException("Queue is Empty!!");
    }
    ArrayList<E> temp= new ArrayList<E>();
    temp.addAll(elements);
    temp.remove(0);
    return temp;
}
```

6.

```java
@Override public Queue<E> clone() throws CloneNotSupportedException
{
    try
    {
        Queue<E> q = (Queue<E>) super.clone();
        q.elements = this.clone();
        return q;
    }
    catch (CloneNotSupportedException e)
    {
    throw new AssertionError();
    }
}
```

For this question, I referred Bloch's Item 13.

A class implementing Cloneable is expected to provide a properly functioning public clone method. The clone method on Queue must duplicate the queue's internals to operate properly. The simplest method to achieve this is to call clone on the ArrayList recursively.

If the clone function just returns super.clone(), the new Queue instance will have the right size field value, but the elements field will correspond to the same array as the original Queue instance. Modifying the original will break the clone's invariants, and vice versa. We may get illogical results or a NullPointerException.

A try-catch block contains the call to super.clone(). This is because Object specifies its clone function to raise a checked exception called CloneNotSupportedException.

7. Loop invariants - A loop invariant is some predicate (condition) that holds for every iteration of the loop. It should satisfy the three conditions:

- before the loop starts
- before every iteration of the loop
- after the loop ends

In the toString () method written by me above, the loop invariant would be:

- result !=ϕ	//result string will never be empty
- size>= i	//stronger loop invariant
- True	//weakest and trivial loop invariant

Rep-invariants – used to capture principles that are crucial in deciding whether or if a data structure is proper. Rep-invariants for the enQueue() method in Queue.java program would be that e!=NULL && size(e)>0.

Contract/specifications – These include precondition and postcondition. A precondition is something that must be true at the start of a function for it to work correctly. A postcondition is something that the function guarantees are true when it finishes.

Contract for the deQueue() in Queue.java program is:

<u>Precondition</u>: True // (weak precondition that accepts all inputs)

Postcondition: If queue is empty, throw IllegalStateException, else return the first element and remove it from the queue. //(strong postcondition that takes care of input)

Question 2

1. The rep-invariants can be –
    a. choices != NULL (choices cannot be NULL)
    b. choices != ϕ (choices cannot be empty)

2.
    a. Constructor GenericChooser(Collection<T> choices)
    Precondition: True
    Postcondition: if input is either empty or NULL, throw IllegalArguentException, else initiate a new ArrayList with provided input.
    Not checking for these conditions will decrease the goodness of the code and may cause problems in the choose() method.
    Changes made in code:

```
public GenericChooser (Collection<T> choices)
{
    if(choices.size() == 0 || choices == NULL){
        throw IllegalArgumentException("Invalid argument");
    }
    else{choiceList = new ArrayList<>(choices);}
}
```

    b. choose()
    Precondition: True
    Postcondition: return a random element from the collection of choices.
    No need to make changes in code.

3. I did not update the choose() method as all the cases that could cause problems were taken care of in the constructor definition. The constructor itself works as a gatekeeper for the correct input, so the input received by choice() method is error-free and we need not check twice.

Question 3

1. The current toString() prints out all the elements that were ever pushed on the stack. This includes the elements that have been popped off/removed. A simple fix is that instead of iterating over the entire array (elements.length), we iterate over the current size of the array (use the 'size' variable)
Modified 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 + "]";
}
```

2. Encapsulation describes the idea of bundling data and methods that work on that data within one unit, like a class in Java. This concept is also often used to hide the internal representation, or state of an object from the outside.
   More specifically, in Java, if a method is public, anyone can access that method and can override it in further implementation.
   In our case, the pushAll() method calls push() method which is public. There may be a case where in the push() method is inherited and modified/overridden. This will cause violation on encapsulation principle.
   To solve this issue, we can either make this method 'private' or declare it as 'final'. Doing so will ensure that the method cannot be modified.
   ```
   public final void push (Object e){...}
   ```

   Precondition: Input should not be NULL and should not contain NULL elements.
   Postcondition: The input elements are pushed onto the stack.

3. To make an immutable version of pop(), we create a new instance of Object array and copy all the elements from original object array to this new one, and then remove the top element. Furthermore, I created a new Stack object, pushed all the elements to this new stack and returned it. This way, we do not change the original input object array.

```
public Object[] pop (Object[] obj)
{
    if (size == 0) throw new IllegalStateException("Stack empty");
    Object[] temp = obj;
    temp[size--] = NULL;
    Stack s = new Stack();
    s.pushAll(obj);
    return s;
}
```

   Also, created a peek() method that just returns the last variable on the stack.

```
public Object peek(){
    return elements[size];
}
```

4. Bloch suggests us to prefer using Lists over Arrays. If, we add an element of different datatype to Array, we get a runtime error, whereas in Lists, we get a compile time error. Compile time errors are always better than runtime errors. A list is dynamic, and

java.util.ArrayList package also provides a method to compare ArrayLists. We do not need to implement equals() method from scratch and can simply use the '.equals()'. For example, ArrayList1.equals(ArrayList2);

Question 4

```
{y ≥ 1} // precondition
x := 0;
while(x < y)
        x += 2;
{x ≥ y} // post condition
```

1. The precondition and the first statement in the code makes it sure that the condition of while loop will always be satisfied i.e., the program will always enter the loop when initiated (when initiated, y will always be greater than x).
   Since the code in the while loop only modifies the variable x (increments by 2 with each iteration), we can say with a guarantee that when the control exits the loop, the value of x will be incremented and it will be greater than or equal to the value of y, which is the postcondition.
   There exists no example, which violates the postcondition, given the precondition is satisfied, so we can say that code satisfies the given contract, and the program is correct.

2. Here our goal is to demonstrate the correctness of a program 'S' in terms of a given precondition 'P' and postcondition 'Q'.
   To prove this, we follow the following steps:
   - Step 1: Calculate program S's weakest precondition with regard to postcondition Q, WP (S,Q)
   - Step 2: If S contains a loop, we must give a loop invariant 'I'.
   - Step 3: Then we must formulate the verification condition which is: P=>WP(S,Q)
     Where P is precondition, S is the program, and Q is the postcondition.
     If this verification condition is true, the Hoare triple is accurate, which means S is correct with relation to P and Q. Otherwise, the Hoare triple is invalid, but we can't say that S is erroneous in relation to P and Q. (it could be that we picked a weak loop invariant)

3. The three loop invariants that can be used for the same code are:
   a. x>=0 (passes the 3 conditions mentioned below)
   b. x<=y+1 (passes the 3 conditions mentioned below)
   c. True (passes the 3 conditions mentioned below)
   d. y>=0 (passes the 3 conditions mentioned below)

   The loop invariant is a condition that must be true:
   - before the loop starts
   - before every iteration of the loop
   - after the loop ends

4. WP(while[I] b do S,Q)= I and (I and b=>WP(S,I) and (I and !b) =>Q)

- Step 1: I = x<=y+1 (sufficiently strong loop invariant)

- Step 2:
  (x<=y+1 && x<y) => WP(x=x+2, x<=y+1) //(I and B) => WP(S,I)
  (x<=y+1 && x<y) => x+2<=y+1
  (x<=y+1 && x<y) => x+1<=y
  x<y => x<y //simplified
  True

- Step3:
  (x<=y+1 && !(x<y)) => x>=y //(I and !B) => Q
  (x<=y+1 && x>=y) => x>=y
  x=y => x=y //plotting these inequalities on graph, we get this solution)
  True

  (I and b=>WP(S,I) and (I and !b) =>Q) = x<=y+1 and True and True = x<=y+1
  WP(x=0, x<=y+1)
  = 0<=y+1

  Verification condition:
  P => WP([x = 0; while[x<=y+1] x<y do x=x+2], {x>=y})
  y>=1 => 0<=y+1 //from above
  y>=1 => y>=-1
  True.

  Here the precondition y>=1 implied the loop invariant x<=y+1, and so we can say that the program is correct.

5. WP(while[I] b do S,Q)= I and (I and b=>WP(S,I) and (I and !b) =>Q)
   - Step 1: I = y>=0 //weak invariant, I
   - Step 2:
     (y>=0 && x<y) => WP(x=x+2, y>=0) //(I and B) => WP(S,I)
     (y>=0 && x<y) => WP(x=x+2, y>=0)
     (x<y) => WP(x=x+2, y>=0)
     we can't simplify more, so just leave as it is.
   - Step 3:
     (y>=0 && !(x<y)) => x>=y //(I and !B) => Q
     (y>=0 && x>=y) => x>=y
     x>=y => x>=y
     True

   WP(x=0; { y>=0 && (y>=0 && x<y) => WP(x=x+2, y>=0) && True})
   = y>=0 && (y>=0 && 0<y) => WP(x=x+2, y>=0) && True
   = y>=0 && {(True && True) => WP(x=x+2, y>=0) } && True
   = y>=0 => WP(x=x+2, y>=0)

// As we notice, we cannot solve this further

Verification condition:
P => WP
y>=1 => y>=0 => WP(x=x+2, y>=0)
//cannot be solved further

Thus, using this loop invariant is not sufficiently strong as we cannot use it prove the validity of the Hoare triple. It does not mean that program is incorrect, it just means that we need to choose a stronger loop invariant.

6.

a. I have written the following JUnit theory test for the provided code:

```
import org.junit.experimental.theories.DataPoints;
import org.junit.experimental.theories.Theories;
import org.junit.experimental.theories.Theory;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertTrue;
@RunWith(Theories.class)
public class JUnitTheories {
    @DataPoints
    public static int[] datapoints() {
        return new int[]{
            1,15,60,1501,Integer.MAX_VALUE
        };
    }
    @Theory
    public void Q4Theories(int y){
        if(y>=1)//checking the precondition
        {
            int x = new Obj...//create an object of the program in Q4
                            // and save the value of x to the new int x declared here.
            System.out.println("Testing with: X = "+ x +" and Y = "y);
            assertTrue(x >= y);
        }
    }
}
```

a. The test data pieces in a JUnit test are statically specified, and you, as the programmer, are responsible for determining what data is required for a specific set of tests. You'll probably want to make tests broader at times. For example, rather of testing for specific numbers, you might need to test for a broader range of acceptable input values. We favor JUnit theories in this circumstance

Meanwhile, in Hoare logic we tap into the more mathematical style of proving the correctness of the program. We take into account much more details like the loop invariant and the contract. Moreover, we also need to provide loop invariants for each loop, very often, we might not choose the perfect loop invariant which may result in the Hoare logic not being satisfied. This condition still doesn’t say anything about the correctness of the program. The Hoare logic condition cannot be satisfied but the program can still be correct if we choose a weak loop invariant.

Question 5

1. Group members – Advait Gogte, Mary George, Hsien Tien Shen, Murudeshwar Barole
2. Occasionally attended, but didn't contribute reliably – Murudeshwar Barole
   Regular participant; contributed reliably – Advait Gogte, Hsien Tien Shen, Mary George

Question 6

1. My favorite thing in the class was the in-class exercises. It's always more effective for me to discuss and learn in a group than the traditional teaching style. Favorite topic was the contracts/specifications. The concept is so fundamental, but it was something I did not learn about in my previous classes.
2. The reading reflections was a very good concept. It made us keep reading the textbook every week, so we were well versed with what we've learned so far. If there were no reflections, we wouldn't have read the textbook completely.
3. Nothing, this class was very good.