Final Exam

Utkrist P. Thapa

G01322365

Answers:

Question 1:

1. Partial Contract:

   Precondition: e != null

   Modifies: this

   Postcondition: e is added to the queue

   Total Contract:

   Modifies: this

   Postcondition: if e is null throws NullPointerException, else e is added to the queue

   For the partial contract, there is no need for changing the code. For the total contract, since it takes all possible inputs including null, we need to change the implementation of enqueue such that the method throws a NullPointerException if e is null.

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

2. The rep invariants for this class are as follows:
   c.elements != null &&
   for all integers 0 <= i < c.size => c.elements[i] != null

   The first rep invariant says that elements list cannot be null. The second rep invariant says that no item inside elements list can be null.

3. The toString() implementation is as follows:

```
public String toString() {
        String r = "Queue: ";
```

```
        if (this.size == 0) {
                r += "[]";
        }
        else {
                r += "[ ";
                for (E e : this.elements) {
                        r += e.toString();
                        r += " ";
                }
                r += "]";
        return r;
  }
```

Here, I return empty brackets "[]" when the queue size is 0. Otherwise, I iterate through all the items in elements list, retrieve their String representation using their toString() method, and enclose all queue items within square brackets. The String variable r contains our string representation and is returned at the end.

4. The contract and implementation of dequeueAll are as follows:

```
// precondition: -
// modifies: this
// postcondition: if size == 0 throws IllegalStateException, otherwise dequeues all existing items in queue (elements) and returns a list of all dequeued items
public List<? extends E> dequeueAll () {
  if (size == 0) throw new IllegalStateException("Queue.deQueue");
  List<? extends E> result = new ArrayList<? extends E>();
  for (int i = 0; i < size; i++) {
        result.add(elements.get(0));
        elements.remove(0);
        size -= 1;
  }
  return result;
}
```

There is no precondition here since the implementation defines empty queue cases with IllegalStateException. I have initialized a new ArrayList<E> result that stores the dequeued items from elements. Since assigning elements to result would cause result variable to be a reference to the same arraylist as elements. I have used for loops to add dequeued items to result and remove items from elements. I also update the size attribute. Finally, I return result.

Bloch advises use of bounded wildcards for increasing API flexibility. I have followed this advice here: the dequeueAll method can run successfully even if elements contains subtypes of E.

5. Rewritten dequeue method for immutable version of this class:

```
  public Queue<E> deQueue () {
     if (size == 0) throw new IllegalStateException("Queue.deQueue");
     Queue<E> result = new Queue<E>();
     for (int i = 1; i < size; i++) {
           result.enQueue(elements.get(i));
```

```
    }
    return result;
  }
```

Here, the dequeue method returns a new dequeued queue called result by assigning all the items in elements except the first item into Queue<E> result. The internal representation of this class is not affected/modified by this new implementation.

6. A reasonable implementation of clone method is as follows:

```
public Queue<E> clone() {
  Queue<E> result = new Queue<E>();
  for (int i = 0; i < size; i++) {
            result.enQueue(elements.get(i));
  }
  return result;
}
```

The implementation of the clone method here is almost identical to the implementation of the immutable deQueue method in question 5. The only difference here is, I have set the loop index i to start from 0 instead of 1 in deQueue. I create a new queue object, enqueue all the items in the elements list, and then return this new object.

Question 2:

1. Good rep invariants for this class are as follows:
c.choiceList != null &&
c.choiceList.size > 0 &&
for all integers 0 <= i < c.size => c.choiceList[i] != null

The first rep invariant says that choiceList cannot be null. The second rep invariant says that the choiceList size must be greater than zero. This is necessary because the GenericChooser class needs to be able to randomly choose at least one item from the list. Finally, the third rep invariant says that no null items are allowed within choiceList.

2. //precondition: choices != null
// postcondition: initializes new GenericChooser object by assigning input collection to the internal representation choiceList

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

//precondition: -
//postcondition: returns a randomly chosen item from choiceList

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

Since the rep invariants don't allow choiceList to be null, the precondition for the constructor may or may not be necessary. The contract for choose() method certainly relies on the rep invariants: it must return a non-null value. It would not matter if the postconditions hold in the methods if the class rep invariant is violated. Hence, I thought it would be appropriate to rely on the rep invariant when writing the contract for the method.

3. In order for a program to be correct, it must satisfy its contract, i.e., given that the precondition holds, the postcondition must hold after the successful execution and termination of the program body. Here, the method choose() does not have preconditions. Since the method does not modify the internal state of the class object, we also do not have to worry about the rep invariant being violated (if we assume that the rep invariant held true before the program is called).
The method generates a random integer within the range of 0-choiceList.size() and uses it as index to retrieve a random item from choiceList. If we assume that the rep invariant holds at the beginning, it is guaranteed that the postcondition holds true. Hence, this program is correct.

Question 3:

1. The toString() method given does not change the object to its string representation. Furthermore, the for loop goes from 0 to elements.length when it should be the case that the loop should go from 0 to size. I have included the corrected code here:

```
@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].toString() + ", ";
       else
           result = result + elements[i].toString();
   }
    return result + "]";
}
```

2. The method pushAll() violates encapsulation because a subtype that extends StackInClass can call the super type method push() even if it overrides this method. This can happen if the subtype calls pushAll() method, which in turn calls the push() method of the supertype.

   Here is a contract for pushAll:
   // precondition: -
   // modifies: this
   // postcondition: if collection == null throws NullPointerException, otherwise pushes all the items in collection to the stack

3. I have rewritten the pop() method for an immutable version of the Stack class.

```
public IStack pop () {
  if (size == 0) throw new IllegalStateException("Stack.pop");
  IStack result = new IStack();
  for (int i = 0; i < size – 1; i++) {
         result.push(elements[i]); }
  return result;
}
```

Here, instead of retrieving the item to be popped and removing it from elements, the immutable version of pop creates a new immutable stack, pushes all items in elements except for the last, and returns this new immutable stack. This version of pop does not affect the internal representation of the immutable stack class.

4. Implementing the equals() method would be easier if the array was replaced by a list because when working with Object[] array, the compiler cannot possibly guarantee type safety during program execution, which is crucial for implementing the equals() method. If the array is replaced by a list, we can employ generics which make it much more convenient to ensure type safety.

Question 4

1. The precondition requires y >= 1. Given that this precondition holds, we assign x to be 0 and enter the while loop since x = 0 < y (because y >= 1). The body of the while loop increments x by 2, and this body is repeatedly executed until x < y is no longer true. This means after the execution of the while loop body, x is either equal to y or greater than y. Hence, the postcondition x >= y holds given that the precondition holds.

2. The three loop invariants are as follows:
a. y >= 1 // since the loop never modifies y this is an invariant
b. x >= 0 // since the loop never decrements x and x starts as 0, this is an invariant
c. x > -1 // since x starts at 0 and the loop only increments x, this is an invariant

3. We use x >= 0 as a sufficiently strong loop invariant. To generate WP, we have:
   a. I : x >= 0
   b. B : x < y
   c. S : x += 2

1. I
   x >= 0

2. I & B → WP(S, I)
   x >= 0 & x < y → WP(x = x+2; {x >= 0})
   x >= 0 & x < y → x + 2 >= 0
   TRUE (x >= 0 implies x+2 >= 0)

3. I & !B → Q
   x >= 0 & !(x < y) → x >= y
   x >= 0 & x >= y → x >= y
   TRUE (x >= y implies x >= y)

   Hence, the weakest precondition WP : x >= 0. Now, the verification condition is given by:
   P → WP
   y >= 1 → x >= 0

4. We use y >= 1 as an insufficiently strong loop invariant. To generate WP, we have:
   a. I : y >= 1
   b. B : x < y
   c. S : x += 2

1. I
   y >= 1

2. I & B → WP(S, I)
   y >= 1 & x < y → WP(x = x + 2; {y >= 1})
   x < 1 → WP(x = x + 2; {y >= 1}) [Cannot solve further]

   Since we cannot further simplify here, this invariant is not sufficiently informative.

Question 5

1. A correct program is one which satisfies the user specifications. This means, given that specified preconditions hold, the postconditions of a correct program must also hold after execution and termination of the program body.

Correct Example:

//precondition: -
//postcondition: returns a randomly chosen item from choiceList

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

Incorrect Example:

//precondition: -
//postcondition: returns a randomly chosen item from choiceList

```
public T choose() {
      Random rnd = ThreadLocalRandom.current();
      return choiceList.get(rnd.nextInt(10000));
}
```

This is an incorrect method because this method does not guarantee that the postcondition holds given that the precondition holds. The large window for random integer might generate IndexOutOfBoundsException.

2. Rep invariants are invariant properties within the concrete representation of a data abstraction that all legitimate objects should satisfy in order for the abstraction to work as expected. For instance, the list that represents the Queue class should never be null. This is a rep invariant that every object of Queue class must satisfy.
Loop invariants are properties that hold before the execution of the while loop, and at the end of the body of the while loop. For instance, in question 4, y >= 1 is an invariant since it holds true for each iteration of the while loop.
Pre/post conditions are contract specifications. These are conditions that define the requirement and behavior of a method. For instance, in question 4, y >= 1 is a precondition because the behavior of the program is unspecified if this condition is not fulfilled.

3. **Junit Theories** allow us to create various different situations/environments for testing that Junit normally does not allow. It grants us more flexibility and provides more convenience.

4. The difference between proving and testing is that proofs are conclusive: a proven program will work on all possible input cases in all possible conditions. Testing is a dynamic process: we generate test inputs, and try it by running the program on them. It is possible to miss corner cases in testing, but this is not true for proving. Proving is slower and more complicated however.
   If you cannot prove using Hoare logic, it just means that there isn't sufficient evidence to prove the correctness of the program. This does not imply that the program is incorrect.

5. Liskov Substitution Principle states that any object that is an instantiation of a subclass must be able to replace all instances of superclass objects without causing errors in a program. For instance, if we have an instance bruno = new GermanShepherd("Bruno") and another instance dog = new Dog("dog"), and GermanShepherd extends Dog, Bruno must be able to replace dog since Bruno inherits all of dog's methods and attributes.