Gustavo Paz
CS 619
Final Exam
December 13, 2021

Question 1

1. For enQueue, write (i) a partial contract and (ii) a total contract. For each part, if you need to change the code for the contract, do so and explain what you did.
    a. Partial Contract
        i. Precondition:
        ii. Postcondition: E is added to the queue
        iii. No changes required for this contract.
    b. Total Contract
        i. Precondition:
        ii. Postcondition: E is added to the list.
        iii. No changes required for this contract.
2. Write the rep invs for this class. Explain what they are.
    a. elements != null
        i. The ArrayList should never be null.
    b. size == elements.size()
        i. The count of size should always be equal to the size of elements.
        ii. This implies that size will always be greater than or equal to zero because the size of ArrayList can never be negative.
3. Write a reasonable toString() implementation. Explain what you did
    a.

```
public String toString() {
   return "Queue: elements="+elements.toString();
}
```

    b. Returning the state of the Queue without exposing too much of the inner workings of the object. Even though I feel like adding the ‘elements’ identifier to the String is exposing a bit, it does indicate what the list being returned refers to, the elements in the Queue.

4. Consider a new method, deQueueAll(), which does exactly what the name suggests. Write a reasonable contract for this method and then implement it. Be sure to follow Bloch's advice with respect to generics. Explain what you did.
   a. Contract
      i. Precondition: None
      ii. Postcondition: Returns a list of elements in the same order as they are dequeued from this. If the number of elements in this is equal to 0, an IllegalStateException is thrown.
   b. Implementation
      i.
```
public List<E> deQueueAll () {
   if (size == 0) throw new IllegalStateException("Queue.deQueueAll");
   List<E> dequeued = new ArrayList<E>(this.elements);
   for(int i = 0; i < this.elements.size(); i++){
      elements.remove(i);
   }
   size = 0;
   return dequeued;
}
```
      ii. To adhere to Bloch's advice about preparing for inheritance, I've implemented the same removal of elements from the list to avoid possibly having subclasses changing the implementation of deQueue(), which would inherently modify the implementation of deQueueAll().
      I've also made deQueueAll() throw an IllegalStateException if size is equal to 0, which is similar to the implementation of deQueue(). I could've also just called deQueue() and made deQueue() final, but this implementation is more portable.
5. Rewrite the deQueue() method for an immutable version of this class. Explain what you did.
   a.
```
public Queue<E> deQueueImmutable () {
   if (size == 0) throw new IllegalStateException("Queue.deQueueImmutable");
   Queue<E> newQueue = new Queue<E>();
   for(int i = 1; i < this.elements.size(); i++){
      newQueue.enQueue(this.elements(i));
   }
   return newQueue;
}

public E peek () {
   if (size == 0) throw new IllegalStateException("Queue.deQueue");
   return elements.get(0);
}
```
   b. Similar to how deQueue() throws an exception when there no elements, the immutable version does the same. Then we iterate through the list of elements in this starting from index 1, essentially dequeuing the element at index 0, until the end of the list of elements and we are enqueuing the elements in order into the new object to be returned since this is an immutable version of the method. We finally return the new Queue object without the first element. We've also implemented peek() which returns the first element in the Queue.

6. Write a reasonable implementation of clone(). Explain what you did.
    a.
    ```
    public Queue<E> clone () {
       Queue<E> clone = super.clone();
       clone.elements = this.elements.clone();
       return clone;
    }
    ```
    b. The implementation calls the super's clone, which should call Object's clone method which will copy the primitives fine, but it will still reference this' elements. So we assign a clone of the list of elements to prevent modifications happening in one object to also affect the other object.

Question 2

1. What would be good rep invariants for this class? Explain each.
    a. choiceList != null
        i. This is to prevent NullPointerExceptions. But because the list is assigned in the constructor, this object should never be null.
    b. choiceList.size() > 0
        i. This is to avoid causing a IndexOutOfBounds in choose() which could be caused if the number of elements in the list is 0.
2. Supply suitable contracts for the constructor and the choose() method and recode if necessary. The contracts should be consistent with your answer to the previous question. Explain exactly what you are doing and why.
    a. Constructor
        i. Precondition: none
        ii. Postcondition: If choices is null, a NullPointerException is thrown. If choices is empty, ie: choices.size() == 0, an IllegalArgumentException is thrown, else this is populated with the elements in choices.
        iii.
```
public GenericChooser (Collection<T> choices) {
  if (choices == null) {
    throw new NullPointerException("GenericChooser.constructor");
  } else if (choices.size() == 0) {
    throw new IllegalArgumentException("GenericChooser.constructor");
  }

  choiceList = new ArrayList<>(choices);
}
```
        iv. I'm throwing exceptions in the constructor to prevent the rep-invariant defined in question 1 from being broken. The NullPointerException is explicitly thrown to avoid relying on ArrayList's implementation to throw it.
    b. Choose
        i. Precondition: none
        ii. Postcondition: Returns an element in this. The element returned is selected at random.
        iii. No modifications.
        iv. There are no modifications because the implementation does exactly as it's supposed to given that the rep-invariant is not violated.
3. Argue that the choose() method, as documented and possibly updated in your previous answers, is correct. You don't have to be especially formal, but you do have to ask (and answer) the right questions.
    a. A method is correct it adheres to the method's contract and if it does not violate the rep-invariant. The choose() method is correct because it adheres to the method's contract, it does return an element randomly using the Random class to select an index in the list to return, which is upper-bounded by the size of the list to prevent any exceptions given that the rep-invariant has not been violated prior to calling the method; and the method itself does not violate the rep-invariant after executing.

Question 3

1. What is wrong with toString()? Fix it.
    a. It exposes to much of the inner workings of the object by also printing size, it does not have the object type in the string returned, and it prints the indices with null objects which is passed the index of size.
    b. The fix:
        i.
```
@Override
public String toString() {
   String result = "StackInClass: elements = [";
   for (int i = 0; i < size; i++) {
      if (i < elements.length-1)
         result = result + elements[i] + ", ";
      else
         result = result + elements[i];
   }
   return result + "]";
}
```
2. As written, pushAll() requires documentation that violates encapsulation. Explain why and then write a contract for pushAll().
    a. It violates encapsulation because it uses an overridable method to push the elements. Along with this it could also leave the object in an inconsistent state if the collection does have a null object.
    b. Contract – with the implementation from the website
        i. Precondition: Collection is not null.
        ii. Postcondition: Pushes all the Objects in collection to this, if the Objects in collection are null, a NullPointerException is thrown and it leaves this with elements returned in the order given by the iterator up to the null object.
    c. The fix – does not match the contract from b – this is to show how it should be implemented to not have an inconsistent state.
        i.
```
public void pushAll (Object[] collection) {
   for (Object e: collection) {
      if (e == null) throw new NullPointerException("StackInClass.pushAll");
   }

   for (Object e: collection) {
      ensureCapacity();
      elements[size++] = e;
   }
}
```

3. Rewrite the pop() method for an immutable version of the Stack class. Keep the same instance variables. Rewrite what you did.
    a.
    ```
    public StackInClass pop () {
       if (size == 0) throw new IllegalStateException("StackInClass.pop");
       StackInClass stack = new StackInClass();
       for(int i = 0; i < elements.size() - 1; i++) {
          stack.ensureCapacity();
          stack.elements[i] = this.elements[i];
          stack.size++;
       }
       return stack;
    }

    public Object peek () {
       if (size == 0) throw new IllegalStateException("StackInClass. peek ");
       return this.elements[size – 1];
    }
    ```
    b. I made the pop method return a new instance of StackInClass without the last element in the array of elements and updated the size field to match the number of elements added. I also added a peek method to still give the ability to get the next element in the stack, but it also throws the exception to still follow the implementation from pop().
4. Implementing the equals() method for this class is a messy exercise, but would be much easier if the array was replaced by a list. Explain why. Note: You are not required to provide a implementation in your answer, but if you find it helpful to do so, that's fine.
    a. Currently the implementation of equals would require not only checking the object type passed in, but it would verify they have the same elements up to the size index in the respective StackInClass instances. Along with this you'll also have to iterate through all the Objects in elements of each instance are equal in for each index.
    If the implementation would be a list, we would not have to worry about iterating up to size, because when elements are removed from Lists, the size fields are automatically update, so their iterators only go up to the size maintained by the list object. So when the equals method is called to compare the two instances, per documentation, the check we would have to do with an array, the List.equals() method does for us.

Question 4

1. Informally argue that this program satisfies the given specification (pre/post conditions).
    a. If the precondition is satisfied, then y is greater than 1. Now because x is assigned 0, the while loop is guaranteed to execute, because x = 0 which implies x < 1 <= y. Given it enters, it guarantees x will be incremented, and will continue to increment x until x < y is no longer true, which implies x >= y is true, which is the postcondition.
2. Give 3 loop invariants for the while loop in this program. For each loop invariant, informally argue why it is a loop invariant.
    a. x >= 0
        i. Because x is assigned 0 before entering the loop it is guaranteed to start at 0, and because it increments at least once because it enters the loop given the precondition, it will be greater than 0.
    b. x < y || y <= x
        i. Because x starts at 0, the loop is guaranteed to execute at least once and therefore x is less than y when it starts. Though after executing it could be that x is greater than y since we're
    c. TRUE
        i. Because this does not depend on any variables in the loop, this holds before and after executing the body trivially.
3. Sufficiently strong loop invariants: Use a sufficiently strong loop invariant to formally prove that the program is correct with respect to given specification. This loop invariant can be one of those you computed in the previous question or something new.

    • Note: show all works for this step (e.g., obtain weakest preconditions, verification condition, and analyze the verification condition).

    • Recall that if the loop invariant is strong enough, then you will be able to do the proof. In contrast, if it is not strong enough, then you cannot do the proof
    a. Loop invariant: x >= 0
        i. WP(while [x >= 0] (x < y) do x := x + 2, {x >= y})
        ii. I
            1. x >= 0
        iii. I && b => WP (S, I)
            1. (x >= 0 & x < y) => wp(x := x + 2, { x >= 0 })
            2. (x >= 0 & x < y) => (x + 2 >= 0)
            3. x < 0 || x > y || x + 2 >= 0
            4. x < 0 is always false so we remove it, now we have x > y || x + 2 >= 0
            5. x > y || x + 2 >= 0, can be true when you enter the loop because x is already greater or equal to 0, so adding two will be true after adding 2. x > y may not be true when you enter, but it can be true when you exit the loop, so what matters in this step is x + 2 >= 0.
        iv. I && !b => Q
            1. (x >= 0 & x >= y) => x >= y
            2. x < 0 || x < y || x >= y

            3. x < 0 is always false so we remove it, now we have
               x < y || x >= y which is x < y ||! (x < y), which is a tautology, making this always TRUE.
        v. So now we have x >= 0 & x + 2 >= 0 & TRUE which is equivalent to x >= 0 & x + 2 >= 0. Which is true when you enter the loop because x is assigned 0 and still greater than 0 when the body executes and adding 2 to x is also true when you enter the loop since its assigned 2 and when you execute the body because there is no decrementing, just adding 2 so this holds.

4. Insufficiently strong loop invariants: Use another loop invariant (could be one of those you computed previously) and show that you cannot use it to prove the program.

   • Note: show all work as the previous question.

   a. Loop invariant: x != y
      i. WP(while [x != y] (x < y) do x := x + 2, {x >= y})
      ii. I
          1. x != y
      iii. I && b => WP (S, I)
          1. (x != y & x < y) => wp(x := x + 2, { x != y })
          2. (x != y & x < y) => x + 2 != y
          3. The left hand side could be true and the right hand side can be false when x = 0 and y = 2, making this FALSE.
      iv. I && !b => Q
          1. (x != y & x >= y) => x >= y
      v. 3 shows that this invariant does not hold.

Question 5

1. What does it mean that a program (or a method) is correct? Give (i) an example showing a program (or method) is correct, an (ii) an example showing a program (or method) is incorrect.
   a. A method is correct if it follows the method's contract and the class rep-invariant.
   b. Correct version
      i.
```
public class EndlessStack {
   private List<Object> elements;
   // rep-invariant: elements != null and elements from 0 to elements.size() are not null
   public EndlessStack () { this.elements = new ArrayList<>(); }

   // precondition: none
   // postcondition: if e is null, NPE is thrown else e is added to this
   // modifies: this
   public void push (Object e) {
      if (e == null) throw new NullPointerException("Stack.push");
      elements.add(e);
}
```
   c. Incorrect version
      i.
```
public class EndlessStackError {
   private List<Object> elements;
   // rep-invariant: elements != null and elements from 0 to elements.size() are not null
   public EndlessStack () { this.elements = new ArrayList<>(); }

   // precondition: none
   // postcondition: e is added to this
   // modifies: this
   public void push (Object e) {
      elements.add(e);
}
```
2. Explain the difference between rep invariants, loop invariants, and contract/specifications (i.e., pre/post conds). Use concrete examples to demonstrate the difference.
   a. Rep-invariants are things that should be true for an instance to function correctly.
      i.
```
public class EndlessStack {
   private List<Object> elements;
   // rep-invariant: elements != null and elements from 0 to elements.size() are not null
   public EndlessStack () { this.elements = new ArrayList<>(); }

   // precondition: none
   // postcondition: if e is null, NPE is thrown else e is added to this
   // modifies: this
   public void push (Object e) {
      if (e == null) throw new NullPointerException("Stack.push");
      elements.add(e);
}
```

      The rep invariant defined is the line right above the constructor.

b. Loop invariants are things that should hold true before a loop and after the execution of the body.
    i. Given this code with a loop and precondition:

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

    A loop invariant could be y >= 1 because it hold true before and after execution the loop's body.

c. Contracts are specific to class methods in that they define the intended behavior of a method. The contract will define things that the client should do before (preconditions) calling the method, things the method should do in the event of some parameter (postcondition), as in either returning value or throwing errors, or indicating the objects that are modified (modifies clause).
    i.
    ```
    public class EndlessStack {
       private List<Object> elements;
       // rep-invariant: elements != null and elements from 0 to elements.size() are not null
       public EndlessStack () { this.elements = new ArrayList<>(); }

       // precondition: none
       // postcondition: if e is null, NPE is thrown else e is added to this
       // modifies: this
       public void push (Object e) {
          if (e == null) throw new NullPointerException("Stack.push");
          elements.add(e);
    }
    ```

    A contract for the method push indicates the things a client should expect from the method call.

3. What are the benefits of using JUnit Theories comparing to standard JUnit tests. Use examples to demonstrate your understanding.
   a. Junit theories allows datapoints to be fed into method to allow a combination of cases to be tested rather than manually writing each test case expected.
   b. Example:
      i.

```
@DataPoints
public static final Integer[] points ={1, -1, -2, -3, 10};

@Theory
public void testWorks(Integer e)
   assertFalse(e.equals(0));
}

@Test
public void testWorksSingle(){
   assertFalse(new Integer(1).equals(0));
   assertFalse(new Integer(-1).equals(0));
   assertFalse(new Integer(-2).equals(0));
   assertFalse(new Integer(-3).equals(0));
   assertFalse(new Integer(10).equals(0));
}
```

4. Explain the differences between proving and testing. In addition, if you cannot prove (e.g., using Hoare logic), then what does that mean about the program (e.g., is it wrong)?
   a. Proving is providing evidence that code will execute as given a set of preconditions and postconditions ensuring it works given all possible inputs. Testing is ensuring the code does not have bugs or issues with a finite number of inputs.
   b. If you can't prove your code, then it is likely your defined preconditions or postconditions are not strong or weak enough.
5. Explain the Liskov Substitution Principle (LSP). Use a concrete example to demonstrate LSP. Note: use a different example than the one given in Liskov.
   a. LSP states that any important property of a class should also be held for any subtypes, working the same on subtypes.
   b. This would prevent any subclasses from using the method push() as if it were an instanceof EndlessStack.

```
public class EndlessStack {
   private List<Object> elements;

   public EndlessStack () { this.elements = new ArrayList<>(); }

   public void push (Object e) {
      if (e == null) throw new NullPointerException("Stack.push");
      if (this.getClass() != EndlessStack.class) return;
      elements.add(e);
   }
}
```

6 Question 6

This question helps me determine the grade for group functioning. It does not affect the grade of this final.

1. Who are your group members?

2. For each group member, rate their participation in the group on the following scale:

(a) Completely absent
(b) Occasionally attended, but didn't contribute reliably

(c) Regular participant; contributed reliably

Steven Story– C for participation

Eden Knudson– C for participation

Ukrist Thapa - B for participation

7 Question 7

There is no right or wrong answer for the below questions, but they can help me improve the class. I might present your text verbatim (but anonymously) to next year's students when they are considering taking the course (e.g., in the first week of class) and also add your advice to the project description pages.

1. What were your favorite and least aspects of this class?

    Class interaction, because no one liked to participate.

Favorite topics?

    Going over theories and diferent kinds of testing.

2. Favorite things the professor did or didn't do?

    Did: Always tried to engage conversations with students.

3. What would you change for next time?

    Nothing.