Eden Knudson
Final Exam

Question 1

Consider Queue.java.

```
public class Queue <E> {

   private List<E> elements;
   private int size;

   public Queue() {
      this.elements = new ArrayList<E>();
      this.size = 0;
   }

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

   public E deQueue () {
      if (size == 0) throw new IllegalStateException("Queue.deQueue");
      E result = elements.get(0);
      elements.remove(0);
      size--;
      return result;
   }

   public boolean isEmpty() {
      return size == 0;
   }
}
```

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.

Partial Contract (contract with preconditions)
// Requires: e (new element) != null
// Effects: e is added to the end of the queue
// Modifies: this

Partial Contract (contract without preconditions)

// Effects: throw NPE if e is a null element, else add e to the end of the queue
Modifies: this

2. Write the rep invs for this class. Explain what they are.

- elements != null
- elements list does not contain null values
- size == elements.size()
- size >= 0

The rep invariants of the Queue class deal with the two fields (elements and size). The elements list cannot be null, nor contain null values. The size field should be equal to the size of the elements list and be greater than or equal to zero.

3. Write a reasonable toString() implementation. Explain what you did.

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

The toString method includes the type (Queue) and the current state of the object (size and elements).

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.

```
// Requires:
// Effects: throw IllegalStateException if the queue is empty, else remove all of the elements in the queue and return them in a list.
// Modifies: this
public List<E> deQueueAll()
{
      if (size == 0) throw new IllegalStateException("Queue.deQueueAll");
      List<E> deQueueList = new ArrayList<>();
      int listSize = size;
      for (int i = 0; i < listSize; i++)
         deQueueList.add(deQueue());
      return deQueueList;
}
```

Bloch teaches that using generics are safer. Generics are type safe and the compiler will find errors at compilation time rather than run time. Therefore I used a generics list (List<E>).

5. Rewrite the deQueue() method for an immutable version of this class. Explain what you did.

```
// Requires:
// Effects: throw IllegalStateException if the queue is empty, else return an empty Queue and pass the elements stored in the queue.
public final Queue deQueueAll(final Collection<? super E> dst)
{
	if (size == 0) throw new IllegalStateException("Queue.deQueueAll");
	dst.addAll(new ArrayList<>(elements)); // make a defensive copy
	return new Queue();
 }
```

To make the class immutable I made the class final (to prevent the method from being overridden) and made it return a new Queue object that is empty (with size 0). The program can also pass in a collection to collect all the elements being deQueued.

6. Write a reasonable implementation of clone(). Explain what you did.

```
@Override
   public Queue clone()
   {
      Queue q = new Queue();
      for (int i = 0; i < size; i++)
         q.enQueue(elements.get(i));
      return q;
   }
```

I created a new Queue object with a new ArrayList. This creates a defensive copy that prevents the program from editing the original list when using the cloned list.

Question 2

Consider Bloch's final version of his Chooser example, namely GenericChooser.java.

Suppose you want to write a Chooser class with a constructor that takes a collection, and a single method that returns an element of the collection chosen at random.

```
public class Chooser<T> {
   private final List<T> choiceList;

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

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

1. What would be good rep invariants for this class? Explain each.

- choiceList != null
- choiceList does not contain null values
- choiceList.size() > 0

Null pointers were called the "billion dollar mistake" after they were invented. Neither the object, nor any elements stored in the object should be null. I also added that the choiceList size should be greater than zero, because the whole point of the Chooser class is to choose an element of a list. If there are no elements, the class is purposeless.

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.

```
   // Effects: throws NPE if choices is null or contains null values,
   // else throws IllegalStateException if choices is empty
   // else creates a new Chooser object with a list of choices inputted
  // Modifies: this
   public Chooser(Collection<T> choices) {
      if (choices == null || choices.contains(null))
         throw new NullPointerException("Chooser cannot be null or contain null values");
      if (choices.isEmpty())
         throw new IllegalStateException("Chooser must not be empty");
```

```
        choiceList = new ArrayList<>(choices);
    }

    // Effects: returns an element of the collection chosen at random
    public T choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceList.get(rnd.nextInt(choiceList.size()));
    }
```

The constructor is the method that modifies the state of the object, thus it makes sure the rep invariant is good (the choiceList is not null, does not contain nulls, and is not empty) before a new Chooser object is created. Exceptions are thrown if the input is not valid to the rep invariant.

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.

The choose method does not modify the state of the object and accepts no inputs, so no checks need to be performed. The method already assumes the rep invariant is true, so it will just return a random element in the list.

Question 3

Consider StackInClass.java. Note of the push() method is a variation on Bloch's code.

```
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;
  }

  @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);
     }
  }
}
```

1. What is wrong with toString()? Fix it.

Currently it will print all of the elements in the elements array, even if the elements have been popped. Instead it should only print up to the size.

```
@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. As written, pushAll() requires documentation that violates encapsulation. Explain why and then write a contract for pushAll().

The pushAll method violates encapsulation because pushAll uses the push method. The push method can be overridden (using inheritance). Changing push will then change pushAll. Instead, Bloch suggests using Composition or making push final.

Contract for current code:
// Requires:
// Effects: throws NPE if collection contains a null,
// else adds all elements in collection to this (stack)

3. Rewrite the pop() method for an immutable version of the Stack class. Keep the same instance variables. Rewrite what you did.

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

  public final Object peek()
```

```
{
    if (size == 0) throw new IllegalStateException("Stack.pop");
    return elements[size - 1];
}
```

Pop needs two methods to be immutable. Pop() returns a new StackInClass method with one less value in the elements array. Peek() returns the top of the stack. Both methods are final and will not change the state of this.

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 an implementation in your answer, but if you find it helpful to do so, that's fine.

Bloch teaches that lists are preferable over arrays.
If we use a list over an array for StackInClass then we don't need to keep (and print in toString) a separate variable "size" to keep track of Stack size. Then we can just call the toString method on the elements list, instead of iterating over the array.

Question 4

Consider the program below (y is the input).

```
1 {y ≥ 1} // precondition
2
3 x := 0;
4 while(x < y)
5 x += 2;
6
7 {x ≥ y} // post condition
```

1. Informally argue that this program satisfies the given specification (pre/post conditions).

The program begins with y >= 1 and x = 0. That means the while loop will be entered at least once. If x is less than y then the while loop is entered and x is incremented by 2. If the while loop is not entered again that means x is greater than or equal to y, 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.

The loop invariant is a property that holds when the loop is entered and is preserved after the loop body is executed.

- x > 0
    - x starts at 0, and will enter the loop at least one time because y is greater than or equal to 1.
- x >= 2
    - x starts at 0, and will enter the loop at least one time because y is greater than or equal to 1, this will set x to 2.
- y >= 1
    - This is the precondition

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.

WP(while[I] B do S, {Q})
1. I and
2. (I & B) => WP (S,I) // loop invariant holds and enters body of loop
3. (I & !B) => Q // loop invariant holds and does not enter body of the loop - postcondition will hold

Loop invariant: x>=2

WP(while [x>=2] x < y do x+=2, {x>=y})
1. x>=2
True

2. (x>=2 & x<y) => WP(x = x + 2, {x>=y})
x >= 2 & x < y => x + 2 >= y
True
// examples: x = 2, y = 3; x = 4, y = 5

3. x>=2 & !(x<y) => x >= y
x >= 2 & x >= y => x >= y
True

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.**

WP(while[I] B do S, {Q})
1. I and
2. (I & B) => WP (S,I) // loop invariant holds and enters body of loop
3. (I & !B) => Q // loop invariant holds and does not enter body of the loop - postcondition will hold

Loop invariant: y>=1

WP(while [y>=1] x < y do x+=2, {x>=y})
1. y>=1
True

2. (y>=1 & x<y) => WP(x = x + 2, {x>=y})
y >= 1 & x < y => x + 2 >= y
False
// counter example x = 1, y = 100

3. y>=1 & !(x<y) => x >= y
y >= 1 & x >= y => x >= y

True

Question 5
Note: you can reuse your answers/examples in previous questions to help you answer the following questions.

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 program is correct if the program terminates and precondition and postcondition are satisfied in respect to P and Q.

i) Correct example:

```
// precondition: True
// postcondition: x = 1
Public void correct()
{
        int x = 1;
}
```

ii) Incorrect example

```
// precondition: True
// postcondition: x > 1
Public void incorrect()
{
        int x = 1;
}
```

2. Explain the difference between rep invariants, loop invariants, and contract/specifications (i.e., pre/post conds). Use concrete examples to demonstrate the difference.

The rep invariant is a property that holds when you enter and when you exit a method. For example, the Stack class from earlier verified that none of the values in the stack are null. At the beginning of the method we assume the rep invariant is true. If the method modifies the state and adds elements to the stack, the method will check that the elements being added are not null, ensuring that the rep invariant stays true when the method exits.

The loop invariant is a property that holds when the loop is entered and is preserved after the loop body is executed. For example, a loop invariant for loop in the Stack pushAll method is size >= startSize. As the loop is entered and executed the size will only grow. This shows that the loop invariant is valid.

The precondition is an invariant that holds when entering a method and the postcondition is an invariant that holds when exiting a method. For example, the postcondition of the Stack pop

method is that the size of the elements array is - 1. This will only be true at the method exit if the method is completed.

3. What are the benefits of using JUnit Theories comparing to standard JUnit tests. Use examples to demonstrate your understanding.

JUnit Theories allows for tests to be performed on a subset of data points. This allows for testing more input values, while using less code than standard JUnit tests.

```
@DataPoints
        Public static int[] dataPoints {
                Return new int[]  { 11, 12, 32, -1 };
        }

        @Theory
        Public void maxTheory(Integer a, Integer b) {
                System.out.println("Running with Data points - " + a + ", " + b);

                int max = Math.max(a, b);
                if (a >= b)
                        assertEquals(max, a);
                else
                        assertEquals(max, b);
}
```

This theory will test on all possible pairs of dataPoints (16 total tests).

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)?

Testing is dynamic analysis. It involves running a program on inputs. Its strengths are that it is rast and does not need to analyze complex code. However, it could miss corner cases.

Proving is static analysis which analyzes the source code without running it. It doesn't run on any inputs, but instead it tries to reason the program on all possible inputs. However, this is slower than testing and much more difficult.

If you cannot prove a program (using Hoare logic) that does not mean that the program is wrong. It just means that you don't have a sufficiently strong invariant that you are testing.

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.

The Liskov Substitution Principle (LSP) is used to design class hierarchy. The following three properties should be supported:

- Signature Rule. The subtype objects must have all the methods of the supertype, and the signatures of the subtype methods must be compatible with the signatures of the corresponding supertype methods.
- Methods Rule. Calls of these subtype methods must “behave like” calls to the corresponding supertype methods.
    - If B is a subtype of A, B can always be substituted for A
    - B should be more precise than A, strengthen properties of A
        - Overridden methods in B should WEAKEN the precondition of A or STRENGTHEN the postcondition of A
- Properties Rule. The subtype must preserve all properties that can be proved about supertype objects.

Example: Teenager extends Person. Teenager can be used in any place where Person is used and uses all of the same properties. The getAge method behaves the same in Teenager (subclass), but strengthens the postcondition of Person.

```
public class Person
{
	private String name;
	private int age;

	// Effects: returns an age between 0 and 120
	public int getAge()
	{
		return age;
	}

}

public class Teenager extends Person
{
	// Effects: returns an age between 11 and 19
	public int getAge()
	{
		return age;
	}
}
```

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?

Steven Story - c
Utkrist Thapa - c
Gustavo Paz - c

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

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? Favorite topics?

I enjoyed learning about pre and post conditions. I feel like this was a really valuable insight to programming and system architecture.

The Hoare logic was difficult to grasp, but I believe I finally got it.

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

I appreciated that the professor regularly reviewed the homework and in-class exercises.

3. What would you change for next time?

I would do less group work. It allowed for some students to do less work overall. If they saw that another student in their group completed an assignment they wouldn't attempt it themselves.