Final – Ramachandra Rao Seethiraju

==========

Answers for 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**
   i. **Partial Contract:** Contract that has preconditions is called a partial contract.
      - Here I am assuming that we don't want to add null values to the Queue.
      - The current implementation could have the following partial contract:
        - REQUIRES: *e to be non-NULL*
        - EFFECTS: *Adds e to the end of the Queue*
        - MODIFES: Queue to have new element

   ii. **Total Contract:** Contract that has NO preconditions is called a total contract
      - To have a total contract we will need to change the code:
        - Changed the code to handle NULL check before adding element to the end of the Queue.
        - Throwing IllegalArgumentException when NULL element is passed
      - A simple implementation could be written as follows to avoid the partial contract:

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

      - For the above implementation, following will be the total contract:
        - REQUIRES: *None*
        - EFFECTS: *Throws IllegalArgumentException if element is NULL, Else, adds the element to the end of the queue*
        - MODIFIES: *Queue to add new element to the end of the queue*

2. **Write the rep invs for this class. Explain what they are.**
   For this class as-is (without any code modifications), follows can be observed as rep-invariants:
   i. *elements != null*
   ii. *size >= 0*

3. **Write a reasonable toString() implementation. Explain what you did**
   i. Below is one possible implementation of the *toString()* method for the Queue class
      - The toString() prints the class name followed every element in the Queue.
      - Basically prints the contents of the Queue.
      - It doesn't print the "size" attribute, because it is an implementation detail.

   ii. *toString()* Implementation:

```
/**
 * EFFECTS: Prints the contents of the Queue
 */
public String toString() {
    return "Queue{" +
            "elements=" + elements +
            '}';
}
```

   iii. Sample output of *toString()* method:
      - *Queue{elements=[item1, item2, item3]}*

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**
   i. Implemented *deQueueAll()* using Bloch's recommendation using generics
   ii. Following is one possible implementation of *deQueueAll()*

```
public List<? extends E> deQueueAll() throws IllegalStateException {
    /**
     * EFFECTS: Throws IllegalStateException if there are no elements in queue
     *          else, removes (from original queue) and returns all elements
     * MODIFIES: Queue by removing all elements
     */
    if (elements.isEmpty()) {
        throw new IllegalStateException();
    }
    ArrayList<E> newList = new ArrayList<E>();
    Iterator<E> iterator = elements.iterator();
    while(iterator.hasNext()) {
        newList.add(deQueue());
    }
    return newList;
}
```

<u>For the questions 5 and 6, I implemented *ImmutableQueue.java*. Please consider this full implementation for my answers:</u>

```
/**
 *  What I did to make it ImmutableQueue:
 * 1.   Made the class final and made the elements final
 * 2.   Wrote new Rep-Invariants
 * 3.  Wrote new Constructor that makes copy of elements to create new ImmutableQueue
 * 4.  Modified enQueue to make it immutable - added contracts
 * 5.  Modified deQueue to make it immutable - added contracts
 * 6.   Wrote new method head() to get dequeued element before dequeuing
 * 7.  Modified isEmpty to use elements.size()
 * 8.  Wrote clone() that calls new parameterized constructor
 */
import java.util.*;
public final class ImmutableQueue <E> {
    /**
     * REP-INVARIANTS:
     * 1) elements != null
     * 2) each element within the ImmutableQueue is not null
     * NOTE:
     * - We can get rid of the size attribute in new code
     * - Because elements.size can be used
     * - No need to mention size in rep-inv because elements
     *   is always initialzed to 0 (when not null)
     */
    private final ArrayList<E> elements;

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

    private ImmutableQueue(ArrayList<E> initialElements) throws
IllegalArgumentException {
        /**
         * EFFECTS: Throws IAE if initialElements is NULL
         *          Else, returns new ImmutableQueue with the elements
         */
```

```
        if(initialElements == null) {
            throw new IllegalArgumentException();
        }
        this.elements = new ArrayList<>(initialElements);
    }

    public ImmutableQueue<E> enQueue (E e) {
        /**
         * EFFECTS: Throws IAE if e is NULL
         *          Else, returns ImmutableQueue with the elements
         * MODIFIES: Queue to add new Element to the end
         */
        if (e == null) {
            throw new IllegalArgumentException("Cannot add null element to the Queue");
        }
        elements.add(e);
        return new ImmutableQueue<>(elements);
    }

    public ImmutableQueue<E> deQueue() throws IllegalStateException {
        if (elements.size() == 0) {
            throw new IllegalStateException("Queue.deQueue");
        }
        elements.remove(0);
        return new ImmutableQueue<>(elements);
    }

    public E head() throws IllegalStateException {
        /**
         * EFFECTS: Throws ISE if null or zero elements
         *          Else, returns the first element from the Queue
         */
        if(elements == null || elements.size() == 0) {
            throw new IllegalStateException();
        }
        return elements.get(0);
    }

    public boolean isEmpty() {
        /**
         * EFFECTS: Returns true if empty, else false
         */
        return elements.isEmpty();
    }

    public ImmutableQueue<E> clone() {
        /**
         * EFFECTS: Returns a new Queue with new copies of same elements
         */
        return new ImmutableQueue<E>(elements);
    }

    /**
     * EFFECTS: Prints the contents of the Queue
     */
    public String toString() {
        return "ImmutableQueue{" +
                "elements=" + elements +
                '}';
    }
}
```

5. **Rewrite the deQueue() method for an immutable version of this class. Explain what you did:**
   i. What I did:
      - Based on the full ImmutableQueue implementation I provided above, to make deQueue() immutable, we will have to return the resultant Queue after dequeuing instead of the element we removed. In this process, we will have to create a new method called head() for clients to first consume the head elements from the Queue before they can actually dequeue from the Queue. Also in this process, I created a new constructor that makes a new copy of the Queue with the desired elements, that can be used to return.
   ii. Code changes:
      - I am putting my code snippet for immutable *deQueue()*:

```
public ImmutableQueue<E> deQueue() throws IllegalStateException {
    if (elements.size() == 0) {
        throw new IllegalStateException("Queue.deQueue");
    }
    elements.remove(0);
    return new ImmutableQueue<>(elements);
}
```

      - Following supporting code is required to make the above code work:
        - *head()* method to get the element to be dequeued before dequeuing:

```
public E head() throws IllegalStateException {
    /**
     * EFFECTS: Throws ISE if null or zero elements
     *      Else, returns the first element from the Queue
     */
    if(elements == null || elements.size() == 0) {
        throw new IllegalStateException();
    }
    return elements.get(0);
}
```

        - New parameterized constructor that makes copies of elements and returns new Queue

```
private ImmutableQueue(ArrayList<E> initialElements) throws
IllegalArgumentException {
    /**
     * EFFECTS: Throws IAE if initialElements is NULL
     *          Else, returns new ImmutableQueue
     */
    if(initialElements == null) {
        throw new IllegalArgumentException();
    }
    this.elements = new ArrayList<>(initialElements);
}
```

6. **Write a reasonable implementation of clone(). Explain what you did.**
   i. What I did:
      - I basically used the newly implemented parameterized constructor to make a clone of Queue
      - The new constructor already kind of clones a new ArrayList for the new Queue object
      - So just calling the new constructor with current elements should suffice the clone() implementation.
   ii. Code changes:
      - Based on the above full implementation, I am putting my code snippet for *clone()*:

```
public ImmutableQueue<E> clone() {
    /**
     * EFFECTS: Returns new Queue with copies of same elements
     */
    return new ImmutableQueue<E>(elements);
}
```

      - New parameterized constructor that makes copies of elements and returns new ImmutableQueue is used

==========

Answers for Question 2

Following is the modified GenericChooser class. Please consider this full implementation for my answers:

```
public class GenericChooser<T> {
    /**
     * REP-INVARIANTS
     * 1. choiceList != null
     * 2. Each choice is non-NULL (Not mandatory but makes sense to have this)
     */
    private final List<T> choiceList;

    public GenericChooser (Collection<T> choices) throws IllegalArgumentException {
        /**
         * EFFECTS: Throws IAE if choices is null,
         *          Else if, choices have any null elements throws IAE
         *          Else, initialize the choiceList with choices passed-in
         */
        if(choices == null) {
            throw new IllegalArgumentException("Choices cannot be null");
        }
        else if(choices.size() > 0) {
            for (T t: choices) {
                if(t == null) {
                    throw new IllegalArgumentException("Each choice must be non-null");
                }
            }
        }
        choiceList = new ArrayList<>(choices);
    }

    public T choose() throws IllegalStateException {
        /**
         * EFFECTS: Throws IllegalStateException when no choices in the choiceList
         *          Else, returns one choice randomly
         */
        if(choiceList.size() == 0) {
            throw new IllegalStateException();
        }
        Random rnd = ThreadLocalRandom.current();
        return choiceList.get(rnd.nextInt(choiceList.size()));
    }
}
```

1. **What would be good rep invariants for this class? Explain each.**
   - Ideally we would want the Rep Invariant to be the following:
     - choiceList != null
       - Throughout the class we want choiceList to be non-NULL
       - Otherwise this can cause null pointer exceptions
     - Each choice is non-NULL (Not mandatory but makes sense to have this)
       - Having null choices may lead to client errors when the result of choose() method is useds by the client.
       - We could avoid this rep-inv – just have partial contract (precoditions) to the constructor to pass only non-null choices. But total contract (no precondition) is better.
     - Note:
       - choiceList may be empty list. The code should be able to handle this case.
       - This case needs special handling in the choose() method modified implementation

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.**
   - What I did:
     - I defined the rep-invariants and wrote contracts for both the Constructor and the choose() method
     - I modified the code for both constructor and the choose() method to satisfy the rep-invariant and also their contracts
     - Key things I considered while modifying the code are:
       - Ensure the code handles edge cases, ensuring no preconditions (total contract)
       - Handled cases for constructor being called with null or empty list and also list with null values in them
       - Handled cases for choose() method when called when empty choiceList is present
   - Code changes:
     - Modified choose() method with contracts

```
public T choose() throws IllegalStateException {
    /**
     * EFFECTS: Throws IllegalStateException when no choices in the choiceList
     *          Else, returns one choice randomly
     */
    if(choiceList.size() == 0) {
        throw new IllegalStateException();
    }
    Random rnd = ThreadLocalRandom.current();
    return choiceList.get(rnd.nextInt(choiceList.size()));
}
```

     - Modified Constructor code with contracts

```
public GenericChooser (Collection<T> choices) throws IllegalArgumentException {
    /**
     * EFFECTS: Throws IAE if choices is null,
     *          Else if, choices have any null elements throws IAE
     *          Else, initialize the choiceList with choices passed-in
     */
    if(choices == null) {
        throw new IllegalArgumentException("Choices cannot be null");
    }
    else if(choices.size() > 0) {
        for (T t: choices) {
            if(t == null) {
                throw new IllegalArgumentException("Each choice must be non-
null");
            }
        }
    }
    choiceList = new ArrayList<>(choices);
```

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.**
   Following are the reasons why I think the code I have is correct.
   - The modified code I have including the constructor and the choose() method satisfy all the rep-invariants I have defined and the contracts (pre-conditions and post conditions).
   - Following are the possible scenarios, expected behavior and how my code satisfies the correctness
     - When constructor is called with null choices:
       - In this scenario there is an `IllegalArgumentException` thrown, which satisfies the contract of the constructor and hence satisfies the rep-invariant by not allowing null choiceList to be created
     - When constructor is called with empty list
       - In this scenario, there is no exception. New instance of GenericChooser is created with empty choices

- When constructor is called with NON-empty list, but with some NULL choices in it
  - In this scenario, `IllegalArgumentException` is thrown. As per rep-invariant, no elements must be null

- When choose() is called when empty choice list is there
  - In this scenario, `IllegalStateException` is thrown. Satisfies method contract and also still satisfies rep-inv (just a getter method, no modification of the underlying list)
- When choose() is called when non-EMPTY choice list is there
  - In this scenario, no exception is thrown. Satisfies method contract and also still satisfies rep-inv (just a getter method, no modification of the underlying list)

==========

Answers for Question 3

==========

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

- The problem with the existing implementation:
  - The loop ends at **elements.length**, which prints the values that are obsolete values or obsolete referencs.
  - The absolete references/values could be:
    - Values in array that were already popped out AND/OR
    - The NULL values which are placeholders for next push values. Because the underlying datastructure is arrays and the capacity is dynamically increased by some factor everytime the capacity is reached by putting some NULL values
- Fixed code:
  - A simple change of replacing ***elements.length*** with ***size*** should fix this issue.
  - Following is the fixed code:

```
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 problem with the existing implementation:
  - The code will fail in the middle if pushAll gets a collection as an input with at least one element in between that is a null
  - In this case, the code will push elements until it finds a null, throws a NullPointerException (in push() method) and exits the call, leaving out remaining elements in the collection.
  - This causes consistency issues
  - Ideally we want either everything thing to be pushed or nothing to be pushed to the underlying data structure
  - The best way to fix this would be to ensure all elements are valid before we start pushing elements one by one
- Fixed code including the contract:

```
public void pushAll(Object[] collection) {
    /**
     * EFFECTS: Throws IAE when collection is NULL,
     *          Throws IAE if at least one element in the collection is NULL,
     *          Else, pushes all elements into the Stack
     */
```

```
        if (collection == null) {
            throw new IllegalArgumentException();
        }
        for (Object obj : collection) {
            if (obj == null) {
                throw new IllegalArgumentException();
            }
            push(obj);
        }
        for (Object obj : collection) {
            push(obj);
        }
    }
```

3. **Rewrite the pop() method for an immutable version of the Stack class. Keep the same instance variables. Rewrite what you did.**
   - What I did:
     - Although it would make sense change the variable to ArrayList, if we have to keep the same instance variables, we have to do the following to make pop() immutable:
       - Modify the return type of pop method to StackInClass instead of the Object
       - Make code changes to return StackInClass after the dereferencing of top element
       - Write a new top() method to get the top of the Stack before dereferencing from Stack
       - Write a new method or new parameterized constructor that kind of clones the Stack in the current state (after dereferencing) with elements and size variables
       - Add relevant contracts to all the methods
   - Code changes:
     - Changes to pop() method

```
    public StackInClass pop() {
        /**
         * EFFECTS: Throws IllegalStateException when size = 0,
         *          Else return new Stack by removing top element from Stack
         * MODIFIES: The contents of the Stack by removing the top element
         */
        if (size == 0) throw new IllegalStateException("Stack.pop");
        size = size - 1;
        // elements[size] = null;
        return new StackInClass(elements, size);
    }
```

     - Supporting code changes
       - A new method top() to get the top of the Stack before dereferencing from Stack

```
    public Object top() {
        /**
         * EFFECTS: Throws IllegalStateException when size = 0,
         *          Else returns the top element from the Stack
         */
        if (size == 0) throw new IllegalStateException("Stack.pop");
        int topIndex = size-1;
        Object result = elements[topIndex];
        return result;
    }
```

       - New parameterized constructor that kind of clones with elements and size variables

```
    public StackInClass(Object[] newElements, int newSize) {
        /**
         * EFFECTS: Returns a new instance of Stack with copied
    values of elements and size
         */
        this.elements = newElements.clone();
        this.size = newSize;
    }
```

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.**
    - Yes. Implementing a List instead of an array would make equals() method and also the overall code much more simpler.
    - Reasons:
        - Comparison of List with primitive objects is very easy. We could simply do list1.equals(list2) because we don't have worry about size vs length issues.
        - With arrays, especially with dynamically growing array with a possibility of existence of elements that are obsolete/dereferenced in the array, we would have to manually check every value by iterating through both the Stacks to compare and determine their equality.
        - Example: **Stack{size=3;["cat", "dog", "bat"]}** and **Stack{size=3;["cat", "dog", "bat", null]}** are technically equal but equals operator would results in a false (although lengths are different)
        - Whereas, if we implement using Lists, the push and pop would take care of the sizes vs length issues and we could simply do a equals() comparision

==========

Answers for Question 4

==========

Given:

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

Hence:

P = {y ≥ 1}

Q = {x ≥ y}

1. **Informally argue that this program satisfies the given specification (pre/post conditions).**
    - Informally reason that this program is correct with the given P and Q.
        - For any input satisfying the pre-condition P, when the program executes successfully and upon termination, if post condition Q also holds true, then we can say a program satisfies specification.
        - The given P and Q conditions will hold true for the program (while loop) and they satisfy the Hoarse Triple. Because for any value of **y** that satisfies the precondition **y >= 1**, the while loop increments value of **x by 2** in each iteration until the value of **x** < **y**, which satisfies the post condition **x >= y**.
        - For example, lets take y = 3 which satisfies the precondition. The program runs fully and by the time it terminates, the value of x will be 4, which satisfies the post condition also.

2. **Give 3 loop invariants for the while loop in this program. For each loop invariant, informally argue why it is a loop invariant.**
    - Three possible loop invariants.
        - **x <= (y+1)**
            - This can be considered as a loop invariant because, for any value of y that satisfies the precondition, throughout all the iterations the value if x is always less than or equal to y+1
        - **y >= 1**
            - This can be considered as a loop invariant because, for any value of y that satisfies the precondition, throughout all the iterations the value if y is always greater than or equal to 1
        - **x >= 0**
            - This can be considered as a loop invariant because, for any value of y that satisfies the precondition, throughout all the iterations the value if x is always less than or equal to y+

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**

I am considering **x <= (y+1)** as the sufficiently strong loop invariant:
Let's consider the following Hoare triple logic to prove the validity of this chosen loop invariant:

I && (I && B → WP(S, I)) && (I && !B → Q)

In our case:

I is ***x <= (y+1)***
B is ***x < y***
S is ***x = x+2***
Q is ***x >= y***

1. **x <= (y+1)**
2. ***I && B → WP(S, I)***
   In our case
   (x <= (y+1) && x < y) → *WP((x = x+2), {x <= y+1})*
   Let's simplify both sides
   (x <= (y+1) && x < y) → *(x+2 <= y+1)*
   (x – 1 <= y && x < y) → *(x+1 <= y)*
   (x <= y) → *(x+1 <= y)*
   (x <= y) → *(x <= y)*
   TRUE - Because both left and right sides were simplified to same values
3. ***I && !B → Q***
   In our case
   (x <= y+1) && !(x < y) → (x >= y)
   (x <= y+1) && (x >= y) → (x >= y)
   (x >= y) → (x >= y)
   TRUE - Because both left and right sides were simplified to same values

Hence,

- 1 && 2 && 3 => (x <= (y+1)) && TRUE && TRUE
- Will result in **x <= (y+1)**

The Verification Condition VC in this case is:
VC ((x <= (y+1)) → (x <= (y+1)))

- **TRUE because both left and right sides are same**

Result:
Hence, we can prove that the validity of Hoarse Triple because the VC is TRUE for loop invariant x <= y+1

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**

I am considering **y >= 1** as the sufficiently strong loop invariant:
Let's consider the following Hoare triple logic to prove the validity of this chosen loop invariant:

I && (I && B → WP(S, I)) && (I && !B → Q)

In our case:

I is ***y >= 1***
B is ***x < y***
S is ***x = x+2***
Q is ***x >= y***

1. **y >= 1**
2. ***I && B → WP(S, I)***
   In our case
   (y >= 1 && x < y) → *WP((x = x+2), {x <= y+1})*
   Let's simplify both sides
   (y >= 1 && y >= x) → *(x+2 <= y+1)*
   (y >= 1 && y >= x) → *(x+1 <= y)*
   (y >= 1 && y >= x) → *(x+1 <= y)*
   FALSE- Because this cannot be further simplified
3. ***I && !B → Q***
   In our case
   (y >= 1) && !(x < y) → (x >= y)
   (y >= 1) && (x >= y) → (x >= y)
   (x >= y) → (x >= y)
   TRUE - Because both left and right sides were simplified to same values

Hence,
- 1 && 2 && 3 => (x <= (y+1)) && FALSE && TRUE
- Will result in FALSE

The Verification Condition VC in this case will fail:
VC ((x <= (y+1)) → FALSE)
- **FAILS**

Result:
Hence, we cannot prove that the validity of Hoarse Triple because the VC is TRUE for loop invariant y >= 1

==========

Answers for 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.**
   - Lets consider: **P** (precondition) **{S}** **Q** (postcondition)
   - Hoare Tripple says, a program is called correct, given a precondition P, we execute a program {S}, assuming the program terminates and if the postconditions Q satisfies, then that means {S} is correct with respect to the contract.
   - For example the following program and precondition and postconditions:
     {y ≥ 1} // precondition
     x := 0;
     while(x < y)
         x += 2;
     {x ≥ y} // post condition
     For all inputs that satisfy the preconditions y ≥ 1, if the while loop program successfully terminates and at the end if it also satisfies the postcondition x ≥ y, then we can call that the program/while loop is correct

2. **Explain the difference between rep invariants, loop invariants, and contract/specifications (i.e., pre/post conds). Use concrete examples to demonstrate the difference.**
   Lets consider the following example program to explain the differences:

```
public class PersonBankAccount {
    /**
     * == REP-INVARIANT ==
     * 1.   name != null && name is not empty
     * 2.   salary must be between min_sal and max_sal
     * 3.   sumOfTransactionsToday must cap at max_sum_per_day
     */
```

```
    private String name;
    private double salary;
    private ArrayList<Double> transactionsToday = new ArrayList<>();
    private static final double min_sal = 0.0;
    private static final double max_sal = 100000.0;
    private static final double max_sum_per_day = 50000.0;

    public PersonBankAccount(String nameIn, double salaryIn) throws
IllegalArgumentException{
        if(nameIn == null || nameIn.isEmpty()) {
            throw new IllegalArgumentException();
        }
        if(salaryIn < min_sal || salaryIn > max_sal) {
            throw new IllegalArgumentException();
        }
        this.name = nameIn;
        this.salary = salaryIn;
    }

    public double sumOfTransactionsToday() {
        double totalSum = 0.0;
        /**
         * == PRE-CONDITION ==
         * totalSum >= 0
         */
        Iterator<Double> iter = transactionsToday.iterator();
        while (iter.hasNext()) {
            /**
             * == LOOP INVARIANT ==
             * totalSum >= 0
             */
            totalSum = totalSum + iter.next();
        }

        /**
         * == POST-CONDITION  ==
         * totalSum <= 50000
         */
        return totalSum;
    }

    public void modifySalary(double changePct) {
        /** == CONTRACTS ==
         *  REQUIRES: None
         *  MODIFIES: Person's salary based on changePct.
         *            If resultant salary goes below min_sal, sets salary to min_sal,
         *            If resultant salary goes above max_sal, sets salary to max_sal,
         */
        double result = this.salary + (this.salary/100)*changePct;
        if (result < min_sal) {
            this.salary = 0.0;
        }
        else if(result > max_sal) {
            this.salary = max_sal;
        }
    }
}
```

- **Rep-Invariants**
  - Rep-Invariants are at the object state level. It can help determine the corrects of the state of the object at any point of time.

- From the example program, here are the rep-invariants

```
/**
 * == REP-INVARIANT ==
 * 1.   name != null && name is not empty
 * 2.   salary must be between min_sal and max_sal
 * 3.   sumOfTransactionsToday must cap at max_sum_per_day
 */
```

- Notice that these are irrespective of the implementation of the class itself

- **Loop-Invariants**
  - Loop invariants are defined for a code block of any loop. A valid loop invariant is valid for every iteration of the loop at the beginning and at the end of the loop, for a given input that satisfies the precondition and also satisfies the postcondition after exiting the loop.
  - From the example program, here is the loop-invariant:

```
public double sumOfTransactionsToday() {
        double totalSum = 0.0;
        /**
         * == PRE-CONDITION ==
         * totalSum >= 0
         */
        Iterator<Double> iter = transactionsToday.iterator();
        while (iter.hasNext()) {
            /**
             * == LOOP INVARIANT ==
             * totalSum >= 0
             */
            totalSum = totalSum + iter.next();
        }
        /**
         * == POST-CONDITION  ==
         * totalSum <= 50000
         */
        return totalSum;
    }
```

  - Notice that here in the example, totalSum >= 0 will always be valid at the beginning and at the end of the while loop.

- **Contracts**
  - Contracts are preconditions and postconditions for a method that are defined between client/customer and the code/service.
  - From the example program, here is the contract

```
public void modifySalary(double changePct) {
        /** == CONTRACTS ==
         *  REQUIRES: None
         *  MODIFIES: Person's salary based on changePct.
         *            If resultant salary goes below min_sal, salary=min_sal,
         *            If resultant salary goes above max_sal, salary=max_sal
         */
        double result = this.salary + (this.salary/100)*changePct;
        if (result < min_sal) {
            this.salary = 0.0;
        }
        else if(result > max_sal) {
            this.salary = max_sal;
        }
    }
```

  - Notice that contracts are irrespective of the implementation

3. **What are the benefits of using JUnit Theories comparing to standard JUnit tests. Use examples to demonstrate your understanding.**
   i. The concept of JUnit theories allows a code/functionality to be tested an infinite set of data points. The test case is split into two parts - DataPoints and Theory, where the DataPoints are the set of inputs that can be used to test against a functionality and the Theory is a functionality/concept we desire to test.
   ii. This concept helps tremendously in modularizing the test cases. Having the DataPoints independent of the functionality testing technically provides infinite options to test.
   iii. Example:
      a. Lets say we want to test ***object1.equals(object2)*** with exhaustive list of inputs.
      b. @DataPoints public static String[] inputs = {"cat", "cat", "dog"};
      c. @Theory tests whether ***assertTrue(input1.equals(inpu2) && input2.equals(input1))***
      d. This test case requires two parameters
      e. With this setup the assertTrue statement is executed with various inputs with various combinations of inputs taken from
      f. A total of 9 combinations of inputs are passed to the Theory from the DataPoints
         i. cat and cat
         ii. cat and cat
         iii. cat and dog
         iv. cat and cat
         v. cat and cat
         vi. cat and dog
         vii. dog and cat
         viii. dog and cat
         ix. dog and dog
      g. Actual code for the above example is follows:

```
@DataPoints
public static String[] inputs = {"cat", "cat", "dog"};

@Theory
public void equalsTest(String input1, String input2) {
       assertTrue(input1.equals(inpu2) && input2.equals(input1))
}
```

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)?**
   - **Proving:** Verifying methodically whether the implementation is correct with respect to specification/contract. This basically involves using mathematical definitions and proofs.
   - **Testing:** Testing is performed on a program based on a finite set of inputs. Ideally there can be infinite set of inputs but it is not possible to test with all possible values.
   - **If we cannot prove:** If we cannot prove if an implementation is correct based on any methodology like Hoare logic, it still does NOT mean that the program is wrong. All it means is that we are unable to PROVE that the program is correct with respect to the specification.

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.**
   i. Liskov states that "Type hierarchy requires the members of the type family to have related behavior. In particular, the supertype's behavior must be supported by the subtypes: subtype objects can be substituted for supertype objects without affecting the behavior of the using code. This property is referred to as the substitution principle. It allows using code to be written in terms of the supertype specification, yet work correctly when using objects of the subtype"
   ii. Liskov has defined following rules for valid subtypes:
      a. Signature Rule
         i. Subtypes must implement all methods from SuperType and method signatures and specification should match

b. Methods Rule
    i. The behavior of the methods of Subtype should match with SuperType's
    ii. Also, we should check for:
        1. Precondition of supertype should be stronger than subtype
        2. Postcondition of supertype should be weaker than subtype
        3. Invariant must be preserved

c. Properties Rule
    i. Subtypes must ensure all attributes of SuperType are preserved.

iii. For example:

```
class Database {
    public String selectQuery(String sql) {
        /**
         * REQUIRES: sql != null
         * EFFECTS: returns a selectQuery based on sql
         */
        // SOME CODE
        return "selectQuery";
    }
}
class MySql extends Database {
    public String selectQuery(String sql) {
        /**
         * EFFECTS: when sql is null uses dummy query to make sql,
         *          else returns a selectQuery based on sql
         */
        if (sql == null) {
            sql = "dummysql";
        }
        // SOME CODE
        return "selectQuery";
    }
}
```

a. Precondition check:
    i. SuperType (Database) has stronger precondition compared to SubType (Mysql)
    ii. Because Database class requires the input to be non-null, but not the MySql class
    iii. Hence, this is valid

b. Postcondition check:
    i. No change (some implementation detail change)
    ii. Hence this is valid

c. Hence we can say that this Hierarchy is a valid one according to Liskov's Substitution Principle.

==========

Answers for Question 6

==========

Who are your group members? We are a group of 4

- Ram (me)
- Mikaela
- Oza
- Mathias

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

- Ram (me) - Regular participant; contributed reliably
- Mikaela - Regular participant; contributed reliably
- Oza - Occasionally attended, but didn't contribute reliably
- Mathias - Regular participant; contributed reliably

==========

Answers for Question 7

==========

What were your favorite and least aspects of this class? Favorite topics?
Substitution principles were my favorite. And the least favorite was the Hoare logic.

Favorite things the professor did or didn't do?
Very clear explanation of concepts. Overall it was great learning experience.

What would you change for next time?

- Wish the inclass exercises are shorter
- Wish the quizzes are discussed in detail in next class