SWE 619 Final Examination   Srikar Reddy Karemma.   G01238290

Question 1 Consider Queue.java.

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

Ans:

Partial Contract :

Precondition: The object e should not be equals to null, if null throw IllegalArgumentException.

Postcondition: The object e shall be added to elements list.

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

Total Contract :

Precondition: None (Since total contracts shall not have a pre-condition)
Post-condition: Checks if the element e is equal to null. If so, throws IllegalArgumentException. If not null, the element e gets added to the elements list.

```
public void enQueue (E e) {

if(e == null){
   throw new IllegalArgumentException();
}
    elements.add(e);
    size++;
  }
```

---- ------   ------   -------   -------   ------   -------   --------

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

Ans:

Representation invariants : **elements != null && size must always lie between 0 inclusive and elements.size() inclusive.**

Reason : elements is the list of objects and it should not be equal to NULL. The program should maintain the state of the size such that it is >= zero and <= elements.size().

---- ------   ------   -------   -------   ------   -------   --------

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

Ans:

```
@Override public String toString() {
    String result = "The current size of the Queue = " + size;
    result += ";
    The elements of current Queue from Front to Back = Head → ";
    for (int i = 0; i < size; i++) {
      if (i < size-1){
        result = result + elements.get(i) + ", ";
       }
      Else {
        result = result + elements.get(i);
      }
    }
    return result + " ← Tail";
  }
```

The above code traverses the queue from front to back. So, the zeroth element shall be the head of queue and the size-1th element shall be the tail of the queue. So, starting from 0 to size-1, in the incremental fashion, we shall be traversing through the entire queue. The else condition executed for the last element to avoid the "," at the last. Suppose the elements in the queue are 1,2,3,4,5. Where 1 is head and 5 is the tail.

The **toString()** would print something like :

```
The current size of the Queue = 5
The elements of the current Queue from Front to Back = Head → 1,2,3,4,5 ← Tail
```

---- ------ ------ ------- ------- ------ ------- --------

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

Ans:

Preconditions :

1. The size should be greater than 0.

Postconditions :

2. The size shall be equal to 0 and the elements.size() shall be equal to 0.

```
public void deQueueAll(){
  int cur_size = size;

for(int i=0;i<cur_size;i++){
   deQueue();
}
}
```

Explanation : Here I have initialized the cur-size variable to be equal with the size variable. So, we should basically call the deQueue() method cur_size number of times. So, inside for loop I have made sure to do the same. Storing size inside of cur_size makes sure to trigger the deQueue size number of times till it becomes empty.

---- ------ ------ ------- ------- ------ ------- --------

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

Ans:

```
Public List<E> deQueue(List<E> elements) {
  if(isEmpty()){ throw IllegalStateException(); }
  List<E> new_elements = new ArrayList<E>();
  new_elements.addAll(elements);
  new_elements.remove(0);
  return new new_elements;

  }
```

Explanation: So, in the immutable version of the deQueue, the underlying elements list should not be tampered with. Hence, we should be copying the existing elements list into a new list called new_elements. Now, as we know that the head points to the first element of the Queue, new_elements.remove(0) removes the head of the newer queue. Now, we shall return the new_elements without the head. The client side code will update it's older list of elements(queue) with the newer ones and use it for the further operations.

Client Side Code :
List<E> new_queue = deQueue(older_elements);

---- ------ ------ ------- ------- ------ ------- --------

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

Ans:

```
public List<E> clone(List<E> original)
  {
     List<E> copy = new ArrayList<>();
     copy = (ArrayList)original.clone();
     return copy;
  }
```

This method of cloning is called as the field-by-field copy or simply the field copy. So, basically if the field value is a reference to an object, it basically copies the reference and in the case of primitive data types, it just copies the values of the primitive types. In reference, this phenomenon is also referred to as shallow copy.

---- ------ ------ ------- ------- ------ ------- --------
---- ------ ------ ------- ------- ------ ------- --------

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

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

Answer : The rep invariants for this class are:

choiceList != null && size(choices) > 0

If the choiceList is null….

---- ------ ------ ------- ------- ------ ------- --------

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.

The code after modifications:

```

import java.util.*;
import java.util.concurrent.*;

// Bloch's final version
public class GenericChooser<T> {
   private final List<T> choiceList;

   public GenericChooser (Collection<T> choices) {

        if(size(choices) == 0){
         throw new IllegalArgumentException();

}

      if(choices == null){
        throw new IllegalArgumentException();

     }


if(choices.contains(null){
       throw new IllegalArgumentException();

     }

     choiceList = new ArrayList<>(choices);
   }

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

Ans:
Pre Conditions :
None
Post Conditions :

1. choices must not contain null, if it contains null, we should throw IllegalArgumentException.
2. If the entire choices is null, should throw IllegalArgumentException
3. If size(choices) == 0, should throw IllegalStateException.
4. Load the choices into the choiceList (creating the chooser with 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.

Ans :
For choose():
Pre Conditions :
None (Based on the previous cases)
Post Conditions: We should return a random choice from the list of choiceList

I feel the choose() method need not be changed further. The changes made to the previous code makes sure to return a random element from the existing list. We are already making sure that our choiceList cannot contain null values, choiceList is not null and also choiceList is size is greater than zero by checking all these conditions in the choices which is passed as an argument to the GenericChooser constructor.

---- ------ ------ ------- ------- ------ ------- --------
---- ------ ------ ------- ------- ------ ------- --------

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

1. What is wrong with toString()? Fix it.
Ans : Here, the push() method adds an element to stack and increases the size by 1. In the sameway, pop() method removes an element and decrements the value of size by 1. So, at any given state, the valid length of the stack is given to us by the size variable but not the size of elements array. Because, the elements array would also contain some junk elements which we would not like to print. We would want to print elements only upto the size variable. Also, to be more precise, there can be some sensitive information in the elements array and the current toString() method would print them all using the toString() method which is very inappropriate.

The Fix :

Replace **elements.length** with **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().

Ans: Here, the pushAll() method relies upon the push() method which functions in such a way that after making sure that the object **e** is not null, it calls the ensureCapacity() and pushes the new element into elements[] and also increments the size. Yet, the problem is that push method is public and can be accessed anywhere and this opens up a scope for overriding of the push method. This violates the principle of Encapsulation since the whole point is that data should be accessed by only designated entities. By leaving the push as public, we are hiding the actual/internal mechanisms of the function. In such scenario, the functionality of pushAll() method shall be tampered as well. Hence, making push() as final would be one good way to fix these issues.

The Newer Contract for pushAll():

Pre-condition: The collection shall not contain any null values. If so, raise a NullPointerException.
Post-condition: All the elements of collection would be pushed into the elements array.

---- ------ ------ ------- ------- ------ ------- --------

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

Ans : So, in an immutable version of Stack, the main idea is to make sure the underlying stack architecture (Objects [] in our case) stays immutable. So, the better way would be to return a new Stack (Objects []) in immutable version of the Stack. So, the new Objects[] would not contain the popped element from the previous Objects[]. So, our client code would take this updated Objects[] method and assign it to the existing stack. In this way, the immutability would be achieved by not changing the existing Objects[] as well as the size variable.

Immutable Pop():

```
public Object[] pop (Object[] elements) {
    if (size == 0) throw new IllegalStateException("Stack.pop");
    Object[] New_Stack = elements;
    int New_Size = size - 1;
    New_Stack[New_Size] = null;
    // elements[size] = null;
    return New_Stack;
  }
```

Client Code :
```
New_Stack = pop(Existing_Stack);
```

---- ------ ------ ------- ------- ------ ------- --------

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.

Ans: Implementing equals method in this case is indeed quite messy. The main reason is that, since the status of current stack is given to us by making use of the size variable, it disables us to make use of the equals method due to factors such as the current size variable as well as the null values present in either of the two arrays under the comparison. So, when we are using arrays, we need to check for equality index-wise. It becomes much more complicated when either of two arrays are not equal in size. All this process can be avoided if we use lists in the place of arrays. The usage of size variable could also be avoided. The push operation would just be as simple as list.add(new_element). The pop() operation would be simplified to list.remove(list.size()-1). We can iterate from top to bottom of the stack by print(list.size()-1), print(list.size()-

1)..print(list.size()-list.size()). Moreover, the implementation of equals() would be as simple as return list.equals(list1).

---- ------ ------ ------- ------- ------ ------- --------
---- ------ ------ ------- ------- ------ ------- --------

4 Question:

Consider the program below (y is the input).

```
{y ≥ 1} // precondition

x := 0;
while(x < y)
        x += 2;

{x ≥ y} // post condition
```

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

Ans : This program satisfies the given specification due to the following points. Starting off lets assume the value of y and start off our verification process. Let's say y value is 2 (since y >=1). Now, since x = 0 and x is less then y which is 2, the control enters the while loop. Now, the value of x will be incremented to 2. Now, since x is no more less than y, the loop terminates with x and y values to be both equal to 2. Now, the post-condition also satisfies since x >=y since 2 >=2. Hence by these deductions, the program indeed satisfies the given specifications.

---- ------ ------ ------- ------- ------ ------- --------

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 Following are the loop invariants :

1. Y >= 1, this loop in-variant holds because y greater than or equal to 1 is the precondition and any where in the loop, this condition always holds since we are not tampering the value of Y inside the loop.
2. X >=0, even this loop in-variant always holds since zero is the initial value of X and it cannot go any lower since we are only performing the addition operation with positive term 2 upon x but not division/subtration.
3. TRUE is also a valid loop-invariant.

---- ------ ------ ------- ------- ------ ------- --------

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.

Ans : Now let us use hoare logic and take one of the loop invariants inorder to prove our program to be correct :

Let us take our condition Q = {x>=y}

Capturing Loop Invariant: WP(while[I] B do S, {Q}) =

- I and
- (I & B) => WP(S , I)
- (I & !B) => Q

Now, let us use X>= 0 to prove our program to be correct:
Now B is the condition which is X<Y. S is the body which is X=X+2. We know I which is X>=0. Now let us substitute all these values in the above logic.

WP(while[X>=0] X<Y do X+=2, {X>=Y}) =

- X>=0 and

- (X>=0 & X<Y)   => WP(X=X+2, {X>=0})
  X<Y => X+2>0
  X<Y => (True)
  According to DeMorgan's it is **True**.

- X>=0 & !(X<Y) => X>=Y
  Hence we can write this as:
  X>=0 & X>=Y

This is True

So finally, since X>=0 && **True** && **True**, we can say that X>=0

WP(X:=0; {X>=0})=
By substituting X we get
0 >=0 => 0>=0
True by DeMorgans Law.

Verification:

vc (P => wp(..)) P => WP(x := 0; while[X>=0] X<X do X =X +2, {X>=Y}) =
P => X>=0 => X >= 0, which holds to be **True. This is sufficiently string loop invariant.**

---- ------ ------ ------- ------- ------ ------- --------

B. 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.

Capturing Loop Invariant: WP(while[I] B do S, {Q}) =

- I and
- (I & B) => WP(S , I)
- (I & !B) => Q

Now, let us use Y>= 1 to prove our program to be imcorrect:
Now B is the condition which is X<Y. S is the body which is X=X+2. We know I which is Y>=1. Now let us substitute all these values in the above logic.

Ans :

WP(x = 0; WP(while[Y>=1] X < Y do X=X+2, {X> =Y}) =

1. Y>=1

2. (Y>=1 & X<Y) => WP(X+=2, {Y>=1})
   (Y>=1 & X<Y) => Y>=1 **Which is False**

3. Y>=1 & !(X<Y) => X>=Y
   We can change this to:
   Y>=1 & X>=Y => X>=Y **Which is False**

Verification condition:
vc (P => wp(..)) P => **False**
y>=1 => **False**
Hence, this is not strong enough to prove as hoare triplet.

---- ------ ------ ------- ------- ------ ------- --------
---- ------ ------ ------- ------- ------ ------- --------

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?
    i. Give (i) an example showing a program (or method) is correct, an (ii) an example showing a program (or method) is incorrect.

Let us assume this short code snipped to be our program:

```
{N >=0} //Condition for N
I := 0;

While(I<N){
I := N
}

{I == N}
```

For the above program, the following Loop Invariants seems reasonable :
Loop Invariant: N >= 0
Loop Invariant: I <=N
Loop Invariant: I >= 0
Loop Invariant: TRUE

Now let us use hoare logic and take one of the loop invariants inorder to prove our program to be correct :

Let us take our condition Q = {I == N}

Capturing Loop Invariant: WP(while[I] B do S, {Q}) =

- I and
- (I & B) => WP(S , I)
- (I & !B) => Q

Now, let us use I <= N to prove our program to be correct:
Now B is the condition which is I < N. S is the body which is I := N. We know I which is I <=N. Now let us substitute all these values in the above logic.

WP(while[I<=N] I<N do I:=N, {I == N})=

- I <= N

- (I <=N & I < N) => WP(I=N, {I<=N})

  I<N => N<=N

  I < N => True

  According to the demorgans law:
  True

- I <= N & !(I<N) => I == N
  I == N => I ==N
  True

Hence using the given loop invariant, I <= N, we have proved our program to be correct.

---- ------ ------ ------- ------- ------ ------- --------

(ii) an example showing a program (or method) is incorrect.

Let us assume this short code snipped to be our program:

```
{N >=0} //Condition for N
I := 0;

While(I<N){
I := I + 10
}

{I == N}
```

For the above program, the following Loop Invariants seems reasonable :
Loop Invariant: N >= 0
Loop Invariant: I >= 0
Loop Invariant: TRUE

Now let us use hoare logic and take one of the loop invariants inorder to prove our program to be incorrect :

Let us take our condition Q = {I == N}

Capturing Loop Invariant: WP(while[I] B do S, {Q}) =

- I and
- (I & B) => WP(S , I)
- (I & !B) => Q

Now, let us use I <= N to prove our program to be correct:
Now B is the condition which is I < N. S is the body which is I := I + 10. We know I which is I <=N. Now let us substitute all these values in the above logic.

WP(while[I<=N] I<N do I:=I+10, {I == N})=

- I <= N

- (I <=N & I < N) => WP(I=I+10, {I<=N})

  I<N => N+10<=N

  I < N => **False**

  According to the demorgans law:
  False

- I <= N & !(I<N) => I == N
  False

Hence using the given loop invariant, I <= N, we have proved our program to be incorrect.

---- ------ ------ ------- ------- ------ ------- ---------- ------- ------- ------ ------- ----------

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

Ans :

A loop invariant is a well-defined statement which should be true when entering a loop and also after every iteration of the loop which is checked before the test of that particular loop. Following the crucial points which differentiates the loop invariant from the remaining invariants:

- loop initialization in most cases establishes the loop invariant.

- As soon as we enter the loop body the invariant in most cases is assumed.
- Again in the end of the loop, the invariant shall be typically be re-established

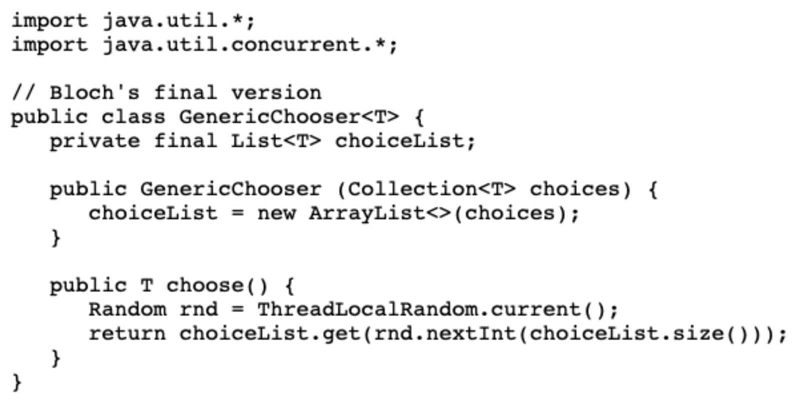
Let us assume this short code snipped to be our program where above points are also validated:

```
{N >=0} //Condition for N
I := 0;

While(I<N){
I := N
}
```

For the above program, the following Loop Invariants seems reasonable :

Loop Invariant: N >= 0
Loop Invariant: I <=N
Loop Invariant: I >= 0
Loop Invariant: TRUE

Representation Invariant : Representation invariants are mainly meant to grab the rules that are true of correct DataStructures. Their main role can be regarded as capturing the rules that are important to reasoning about the data structure and also their validity throughout the program execution. One of the crucial difference between loop invariant and the representation invariant is that, loop invariant mainly deals with the correct programmatic execution of a loop where as representation invariant extends it's scope through out the program and focuses mainly on the global data structural use cases and also where the state of the logic may lose it's correctness.

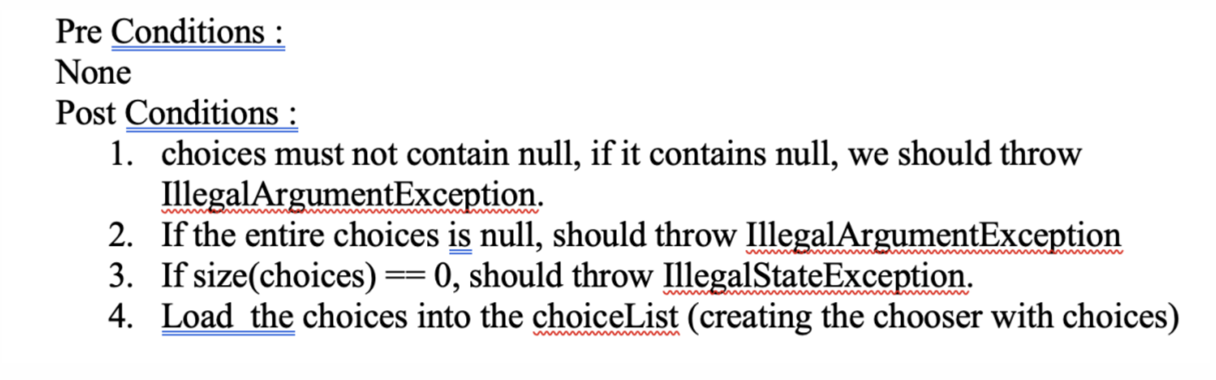


```
import java.util.*;
import java.util.concurrent.*;

// Bloch's final version
public class GenericChooser<T> {
   private final List<T> choiceList;

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

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

The representative invariant for the above program would be **choiceList != null && size(choices) > 0.** This makes sure that the choiceList which is a List of having objects of type T is not null and also the number of choices passed as an argument to constructor has a size of greater than zero.

Pre and Post-conditions :

The pre-condition is usually what must be done initially in order the program to make sure that it shall behave in correspondence with the developer who has written the code. For example, the pre-condition, for a function which shall delete a value out of the ArrayList, the precondition is such that the size of the linkedlist is atleast equal to Where as the post-condition is something the program makes sure if the pre-condition is valid. For example, in the case of deletion of element from the ArrayList, the post condition after this function's execution shall be ArrayList's size to be one less than the previous case (before calling the remove() function).



Pre Conditions :
None
Post Conditions :

1. choices must not contain null, if it contains null, we should throw IllegalArgumentException.
2. If the entire choices is null, should throw IllegalArgumentException
3. If size(choices) == 0, should throw IllegalStateException.
4. Load the choices into the choiceList (creating the chooser with choices)

For the same above example of GenericChooser, the pre and post conditions would be as above.

---- ------   ------   -------   -------   ------   -------   --------

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

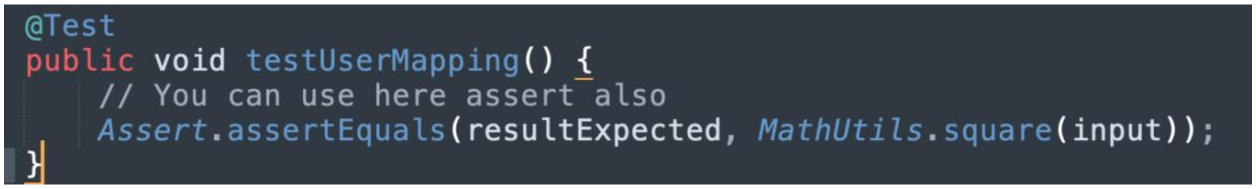
Solution:
The following are the key differences I have observed :

JUNIT Tests vs Theories

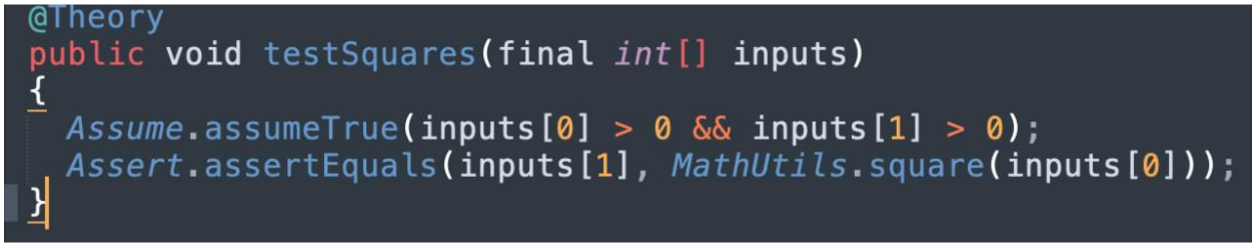
- Test method which is annotated with **@Test** and does not take arguments for JUNIT Tests and the Test method is annotated with **@Theory** and can take arguments in the case of Theories.
- In JUNIT, the. inputs are passed to the constructor (mandatory) and used by the test method. In case Theories, the inputs are directly passed to the test method.

- Parameterized.class tests parametrize tests with a single variable. Where as the Theories.class parametrize with all combinations of many variables.

Examples of both the implementations :



```java
@Test
public void testUserMapping() {
    // You can use here assert also
    Assert.assertEquals(resultExpected, MathUtils.square(input));
}
```



```java
@Theory
public void testSquares(final int[] inputs)
{
  Assume.assumeTrue(inputs[0] > 0 && inputs[1] > 0);
  Assert.assertEquals(inputs[1], MathUtils.square(inputs[0]));
}
```

---- ------ ------ ------- ------- ------ ------- --------

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)?
Ans :

Testing vs Proving

1. Testing asserts the facts where as the proof just tells how to approach the program's correctness.

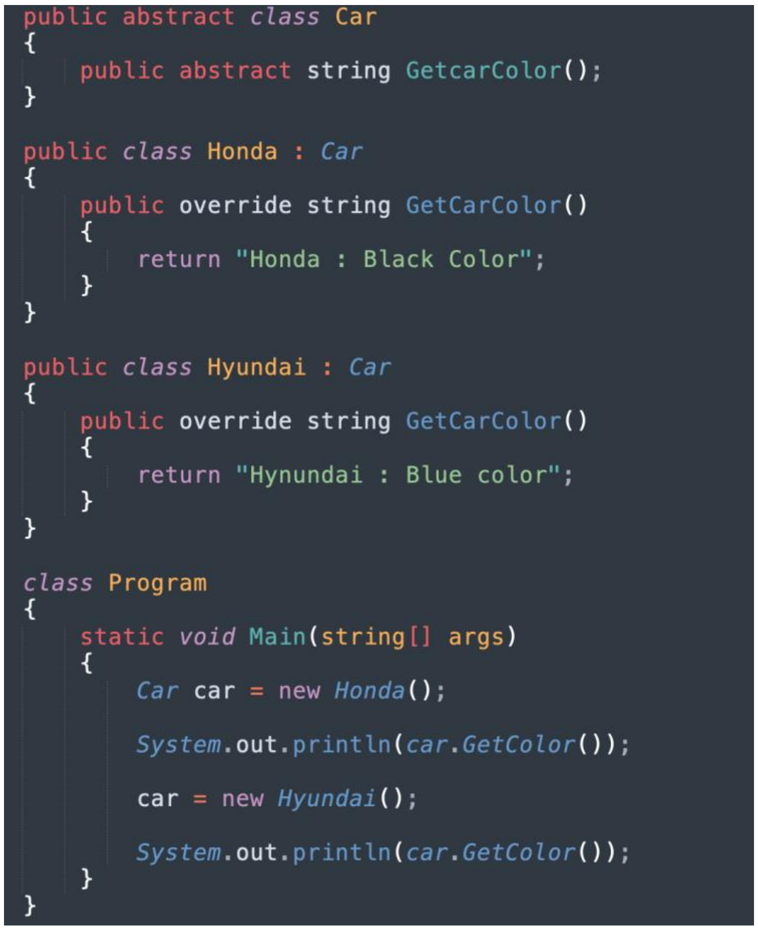
2. The testing mainly focuses upon the practical approach in verifying the correctness where as proving in most cases is not as practical as testing. Proving may be practical for relatively smaller programs.
3. The testing mainly focuses on verifying the program for a single run of execution. Where as in this case, the proving takes a holistic approach of trying to verify the program in all the possible execution cases.

Hoare Logic often considers the following inorder to prove our programs to be correct :
{P} S {Q}. So, if correctness of a program cannot be proved by hoare logic, then the program cannot be assumed to be completely wrong as the implementation of a function is partially correct with respect to its specification if, assuming the precondition is correct just before the function executes, then if the function terminates, the postcondition is true.

---- ------ ------ ------- ------- ------ ------- --------

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.

Ans : In simple words, if class *A* is a subtype of class *B*, then we should be able to replace objects of *B* with objects of *A* without changing the behavior of our program. So, what that statement means is that an object of a superclass should be replaceable by objects of its subclasses without causing any problems in the program. This also says that all the subclasses must, therefore, operate in the same manner as their base classes. The main intention behind following the Liskovs Substitution Principle is toavoid the overuse/misuse of inheritance.
Here is the code which applies LSP:



```
public abstract class Car
{
    public abstract string GetcarColor();
}

public class Honda : Car
{
    public override string GetCarColor()
    {
        return "Honda : Black Color";
    }
}

public class Hyundai : Car
{
    public override string GetCarColor()
    {
        return "Hynundai : Blue color";
    }
}

class Program
{
    static void Main(string[] args)
    {
        Car car = new Honda();

        System.out.println(car.GetColor());

        car = new Hyundai();

        System.out.println(car.GetColor());
    }
}
```

The above code prints Honda : Black Color and Hynundai : Blue Color. So here the above program validates the LSV principle of "Derived classes should be substitutable for their base classes and "Methods that use references to base classes have to be able to use methods of the derived classes without knowing about it or knowing the details."

---- ------ ------ ------- ------- ------ ------- --------

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?

Srikar Reddy Karemma (Regular participant; contributed reliably)
Tuljasree Bonam (Regular participant; contributed reliably)
Rachana Thota (Regular participant; contributed reliably)
Srujan Reddy Tekula (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? 2. Favorite things the professor did or didn't do? 3. What would you change for next time?

Ans : I have enjoyed the class thoroughly. Everything was went well. Learned a lots of stuff. I really liked the in-class exercised which were favorite part of my class. Weekly quizzes were also really informative and learned a lots of stuff. Everything was picture perfect. A great class overall.