Test Case Exercises :

https://github.com/cph-cs241/TEST_GetStartedExercise_Triangles/tree/master/Assignment_03

Equivalence Classes

1. Make equivalences classes for the input variable for this method:
   `public boolean isEven(int n)`

Equivalence Classes	**Test Case (n)**
(n % 2) = 0	even: 2, 4, 6, … (true)
(n % 2) = 1	odd: 1, 3, 5, … (false)

   If the number (n) divided by 2 has no remainder/modulo, the result is true (even numbers).
   If the number (n) divided by 2 has a remainder/modulo 1, the result is false (odd numbers).

2. Make equivalences classes for an input variable that represents a mortgage applicant's salary. The valid range is $1000 pr. month to $75,000 pr. month

Equivalence Classes	**Test Case (salary)**
Salary < 1000	Invalid: out of range
Salary > 75000	Invalid: out of range
999 < Salary < 75001	Valid

   For further understanding, see the boundary analysis on the next set of the exercise.

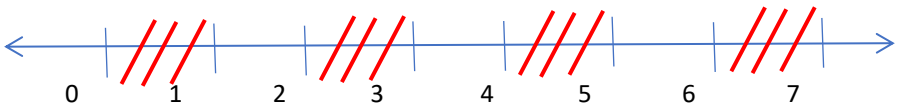
3. Make equivalences classes for the input variables for this method:
   `public static int getNumDaysinMonth(int month, int year)`

Equivalence Classes	**Test Case (month, year)**
1 <= month <= 12	Valid: month
1582 <= year	Valid: year
year < 1582	Invalid: year
month < 1	Invalid: month
month > 12	Invalid: month

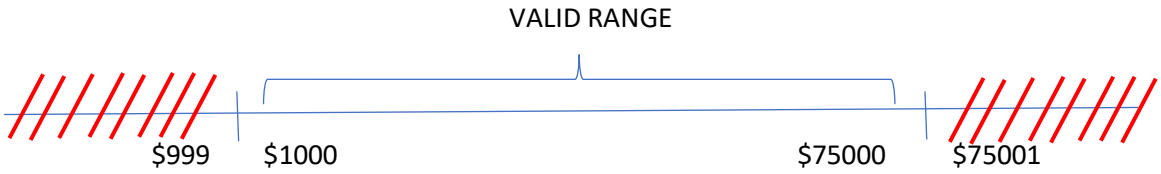
Year based on Gregorian Calendar has been introduced, in a range to the future. Month is based on the range from 1-12 representing the months in a year.

Boundary Analysis

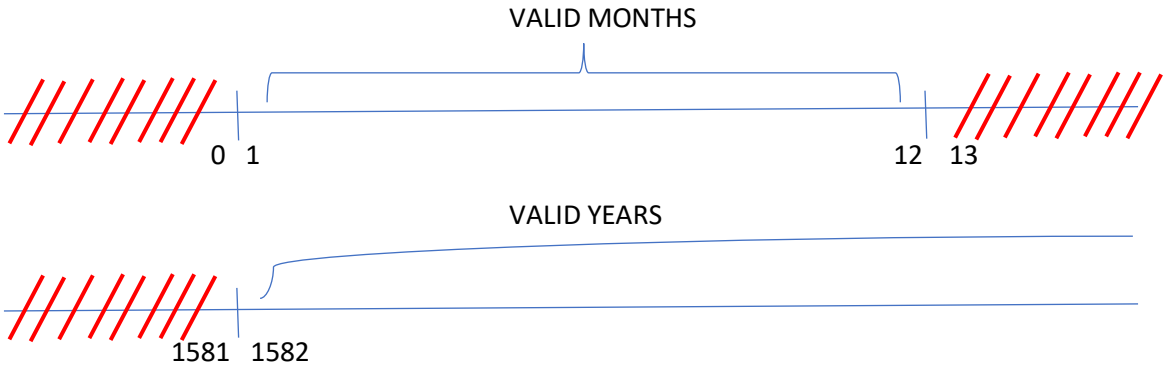
1. BVA - isEven(int n)



2. BVA - Mortgage Applicant's Salary [$999, $1000, $75000, $75001]



3. BVA - getNumDaysinMonth (int month, int year) {month: [0,1,12,13], year: [1581, 1582]}



Decision Tables

1. Make a decision table for the following business case: No charges are reimbursed (DK: refunderet) to a patient until the deductible (DK: selvrisiko) has been met. After the deductible has been met, reimburse 50% for Doctor's Office visits or 80% for Hospital visits.

Conditions				
Hospital visits	T	T	F	F
Doctor's Office visit	T	F	T	F
Actions				
0%				Y
50%			Y	
80%	Y	Y		

2. Make a decision table for leap years. Leap year: Most years that are evenly divisible by 4 are leap years. An exception to this rule is that years that are evenly divisible by 100 are not leap years, unless they are also evenly divisible by 400, in which case they are leap years.

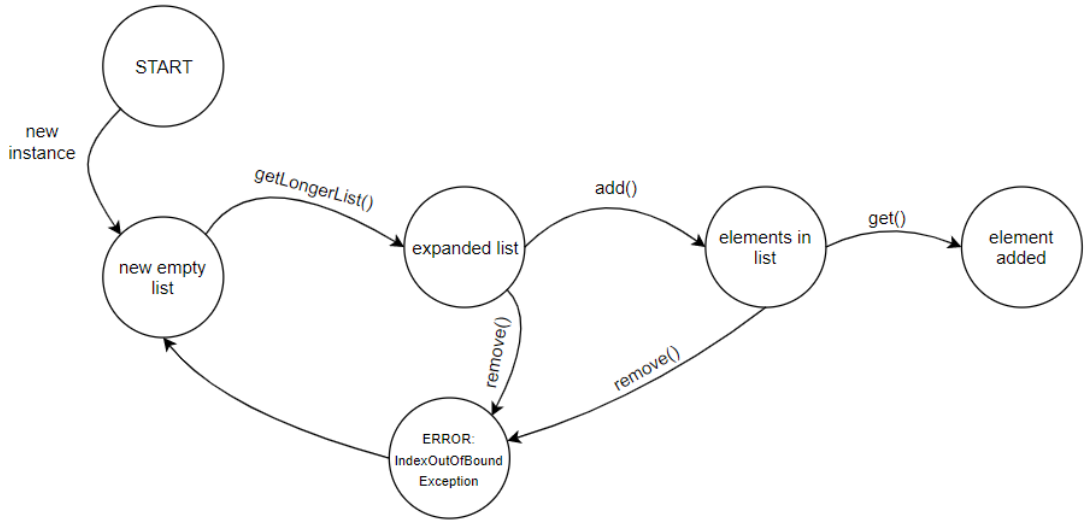
Conditions								
4	T	T	T	T	F	F	F	F
100	T	T	F	F	T	T	F	F
400	T	F	T	F	T	F	T	F
Actions								
Leap Year	Y		Y		Y		Y	

State Transition

State transition testing is another black box test design technique where test cases are designed to execute valid and invalid transitions.

Use this technique to test the class `MyArrayListWithBugs.java`(find code on last page). It is a list class implementation (with defects) with the following methods:

1. Make a state diagram that depicts the states of `MyArrayListWithBugs.java` and shows the events that cause a change from one state to another (i.e. a transition).
   STATE DIAGRAM in adding new element:



STATE TABLE based on the diagram:

	New Instance	getLongerList()	add()	get()	remove()
S1) Start State	S2	-	-	-	-
S2) New Empty List	-	S3	-	-	-
S3) Expanded List	-	-	S4	-	S6
S4) Elements in List	-	-	-	S5	S6
S5) Element Added	-	-	-	?	-
S6) ERROR Exception	S1	-	-	-	-

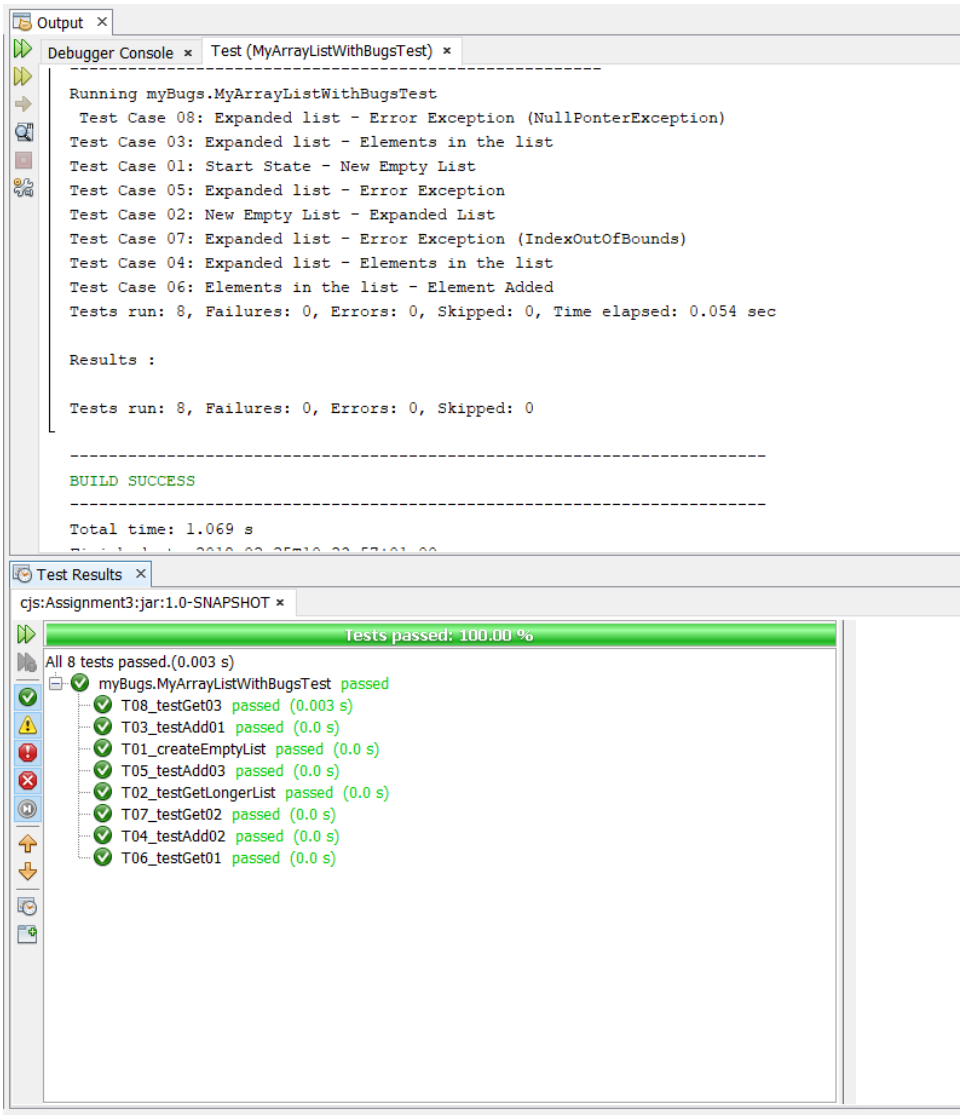
2. Derive test cases from the state diagram.

TEST CASE	STATES #	NEXT STATE #	INPUTS	OUTPUTS
01	S1) Start State	S2) New Empty List	new MyArrayListWithBugs()	size = 0
02	S2) New Empty List	S3) Expanded List	getlongerList()	list (doubled capacity)
03	S3) Expanded List	S4) Elements in the list	add(element)	size = 1
04	S3) Expanded List	S4) Elements in the list	add(0, element)	list size + 1
05	S3) Expanded List	S6) ERROR Exception	add(-1, element)	IndexOutOfBoundsException
06	S4) Elements in the list	S5) Element Added	get(size-1)	element

07	S4) Elements in the list	S6) ERROR Exception	get(-1)	IndexOutOfBoundsException
08	S4) Elements in the list	S6) ERROR Exception	get(1)	NullPointerException

3. Implement automated unit tests using the test cases above.

   See here:
   https://github.com/cph-cs241/TEST_GetStartedExercise_Triangles/blob/master/Assignment_03/src/test/java/myBugs/MyArrayListWithBugsTest.java



4. Detect, locate (and document) and fix as many errors as possible in the class.

   Errors found:

   - *Zero is a valid index. It should be :* `if (index < 0 || nextFree < index)`

```
if (index <= 0 || nextFree < index) {
    throw new IndexOutOfBoundsException("Error (get): Invalid index"
            + index);
```

- *The capacity of the list should be higher than the nextFree to make sure that there is a space available. It should be:* `if (list.length >= nextFree)`

```
// Inserts object at the end of list
public void add(Object o) {
    // check capacity
    if (list.length <= nextFree) {
        list = getLongerList();
    }
```

- *It should update the nextFree as well here.*

```
    public void add(int index, Object o) {
        if (index < 0 || nextFree < index) {
            throw new IndexOutOfBoundsException("Error (add): In
                    + index);
        }

        // check capacity
        if (list.length <= nextFree) {
            list = getLongerList();
        }

        // Shift elements upwards to make position index free
        // Start with last element and move backwards
        for (int i = nextFree - 1; i > index; i--) {
            list[i] = list[i - 1];
        }

        list[index] = o;
        nextFree++;
    }
```

- *Nothing handles if the index is valid but no element in that location. I added this:*

```
if(list[index] == null){
    throw new NullPointerException("Error (get): No element found"
            + index);
}
```

5. Consider whether a state table is more useful design technique. Comment on that.

I think state table is more useful design technique since it can be used to determine invalid systems transitions. In the state table, it shows how going to the next state could possibly go wrong by having invalid inputs. It helps to have a visual representation of system behavior which will help the tester define tests efficiently. In my opinion in the given exercise, invalid inputs would lead to the state of getting an exception or else it will just go smoothly between states until it reaches the defined action which is the element added in the list.

6. Make a conclusion where you specify the level of test coverage and argue for your chosen level:

• Percentage of states visited – *100%, all states would possibly be visited depending on the validity of the input. Only invalid input would hit the S6 (Error Exception)*

• Percentage of transitions exercised – *all the possibilities in changing states has been tested.*