Test Case Exercises (10 Study points - mandatory)

Equivalence classes

1. Make equivalences classes for the input variable for this method:

public boolean isEven(int n)

Equivalence classes	Test Case
n modolus 2 != 0 (n % 2 != 0)	Odd (False): 3
n modolus 2 == 0 (n % 2 == 0)	Even (True): 6

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
amount < 1000	Invalid : 500
1000 <= ammount <= 75000	Valid: 25000
75000 < ammount	Invalid: 10000000

3. Make equivalences classes for the input variables for this method:

public static int getNumDaysinMonth(int month, int year)

Equivalance classes	Test Case
0 < month < 13	Valid: 2
0 > month or month > 13	Invalid: -2 or 13
-2147483648 <= year	Valid : -1000
2147483647 > year	Invalid: 2147483648

My comment: I googled and the min/max size of the int could determine the equivalences classes for the year because Java Calendar should support such a low number. Therefore I don’t think it need to be restricted any further.

Boundary Analysis

1. Do boundary value analysis for input values exercise 1

Odd	Even	Odd	Even	Odd	Even	etc.
1	2 - 2	3 - 3	4 - 4	5 - 5	6 - 6	...

2. Do boundary value analysis for input values exercise 2

Invalid	Valid	Invalid
-∞ to 999	1000 to 75000	75001 to ∞

3. Do boundary value analysis for input values exercise 3

Month:

Invalid	Valid	Invalid
-∞ to 0	1 to 12	13 to ∞

Year:

Invalid	Valid	Invalid
-∞ to -2147483648	1 to ∞	2147483647 to ∞

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:	Rule 1	Rule 2	Rule 3	Rule 4
Doctors Office	T	T	F	F
Dedutible meet	T	F	T	F
Actions/Outcomes:				
50% reimbursed	Y	-	-	-
80% reimbursed	-	-	Y	-
0% reimbursed	-	Y	-	Y

2. Make a decision table for leap years.

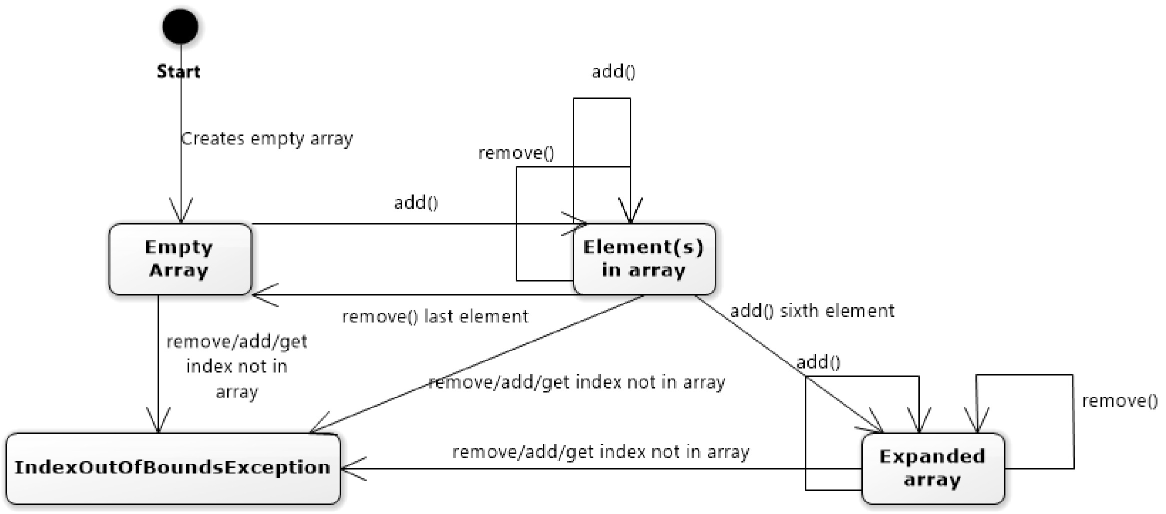
Leap year: Most years that are evenly divisible by 4 are leap years. An exception to this rule is that years that are evenly divisible by 100 are not leap years, unless they are also evenly divisible by 400, in which case they are leap years.

Conditions:	Rule 1	Rule 2	Rule 3	Rule 4
Year Divisible by 4	T	T	T	T
Year Divisble by 100	T	T	F	F
Year Divisble by 400	T	F	T	F
Actions/Outcomes:				
Leap year	Y	-	Y	Y
0% reimbursed	-	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).



“Not all events have an effect in all states. Where an event does not gave an effect on a given state, it is usually omitted”

2. Derive test cases from the state diagram.

derived from add()				
Test Case No.	**INPUT**	**OUTPUT**	**From State**	**To State**
1	add(Object o)	size() == 1	S2. Empty array	S3. Elements in array
2	5 x add(Object o)	size() == 5	S3. Elements in array	S3. Elements in array
3	7 x add(Object o)	size() == 7	S5. Expanded array	S5. Expanded array
derived from get				
Test Case No.	**INPUT**	**OUTPUT**	**From State**	**To State**
4	get(0)	IndexOutOfBounds	S2. Empty array	S4. IndexOutOfBounds
5	5 x add(Object oN)get(4)	o5	S3. Elements in array	S3. Elements in array
6	8 x add(Object oN)get(7)	o7	S5. Expanded array	S5. Expanded array
derived from remove				
Test Case No.	**INPUT**	**OUTPUT**	**From State**	**To State**
7	remove(0)	IndexOutOfBounds	S2. Empty array	S4. IndexOutOfBounds
8	5 x add(Object oN) remove(4)	o1 size == 5	S3. Elements in array	S3. Elements in array
9	8 x add(Object oN) remove(7)	o1 size == 6	S5. Expanded array	S5. Expanded array
10	5 x add(Object o) remove(0-4)	size == 0	S3. Elements in array	S2. Empty array
e/add/get index not i				
Test Case No.	**INPUT**	**OUTPUT**	**From State**	**To State**
11	5 x add(Object o) add(5)	IndexOutOfBounds	S3. Elements in array	S4. IndexOutOfBounds
12	5 x add(Object o) get(5)	IndexOutOfBounds	S3. Elements in array	S4. IndexOutOfBounds

13	6 x add(Object o) add(6)	IndexOutOfBounds	S5. Expanded array	S4. IndexOutOfBounds
14	6 x add(Object o) get(6)	IndexOutOfBounds	S5. Expanded array	S4. IndexOutOfBounds

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

See https://github.com/Games-of-Threads/TestEX3-Emmely for implementation.

4. *Detect, locate (and document) and fix as many errors as possible in the class. a. Define (more) relevant test cases applying black box and white box techniques b. Use xUnit to implement and run the same tests cases again after fixing c. Study the implementation (code) d. Use debugger to locate errors*

Bug 1:

I changed index <= 0 to index < 0 - There was a logical error so that we could not access element in array where index == 0 with the get function. In the jUnit test teste case 5 failed because getting the first element in the array with get(0) throw an IndexOutOfBounds exception.

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

    return list[index];
}
```

Bug 2:

In use case testCaseFiften the test fails when we add new Object with at index 1. The Object is inserted correctly but the size of the array doesn't change. The fix for this logical error is to make sure that the array expand before we shift the elements. Without this implementations this add method works like an update function.

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

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

"As is obvious, the State Transition diagram only shows the valid transitions between the states. (all transitions not shown are considered invalid). And this is a limitation of deriving test cases from a State Transition Diagram.
In order to see the total number of combinations of states and transitions, both valid and invalid, a State Table can be used."
-http://www.getsoftwareservice.com/state-transition-testing/

I first made the state diagram and then tried to run some unit test but I couldn't find any bugs in this way. I personally found it hard to derive test cases from the diagram.
I therefore decided to make a state table and derive test cases from this. I found the state table to be a good tool for this.
The state table gave me a better overview of what I was testing and test coverage than the digram did.
My test strategy was to implement all the test cases that could be derived from the state table and just stick to those. Then within each test case test from boundary values and perform testing on what could be weaknesses.

	add	get	remove	add sixth element	remove last element	remove last in expanded array	remove/ add/get index not in array	
S1. Start state								
S2. Empty array	S3	S4	S4				S4	
S3. Elements in array	S3	S3	S3	S5	S2		S4	
S4. IndexOutOfBounds								
S5. Expanded array	S5	S5	S5			S3	S4	

"We generate test cases by stepping through the ST. Each row/column intersection is a test case." - http://www.getsoftwareservice.com/state-transition-testing/

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

Percentage of states visited Percentage of transitions exercised

I have 15 test cases derived from the state table with 27 states where are 11 invalid[1] states. So the test cases covers around 75% of the states.

[1] Emmely Lundberg cph-el69

Within each test case we should also cover Boundary values which I have cover just about with my unit tests.

S