Міністерство освіти і науки України
Національний технічний університет України „КПІ”
Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки
інформації та управління

ЗВІТ

до лабораторної роботи № 7
з предмету:

„ОСНОВИ ТЕХНОЛОГІЙ ПРОГРАМУВАННЯ”

Виконав студент *ІП-61 Кушка Михайло Олександрович, 2-й курс, ІП-6116*

(№ групи, прізвище, ім’я, по батькові, курс, номер залікової книжки)

Прийняв *Подрубайло О.О.*

(посада, прізвище, ім’я, по батькові)

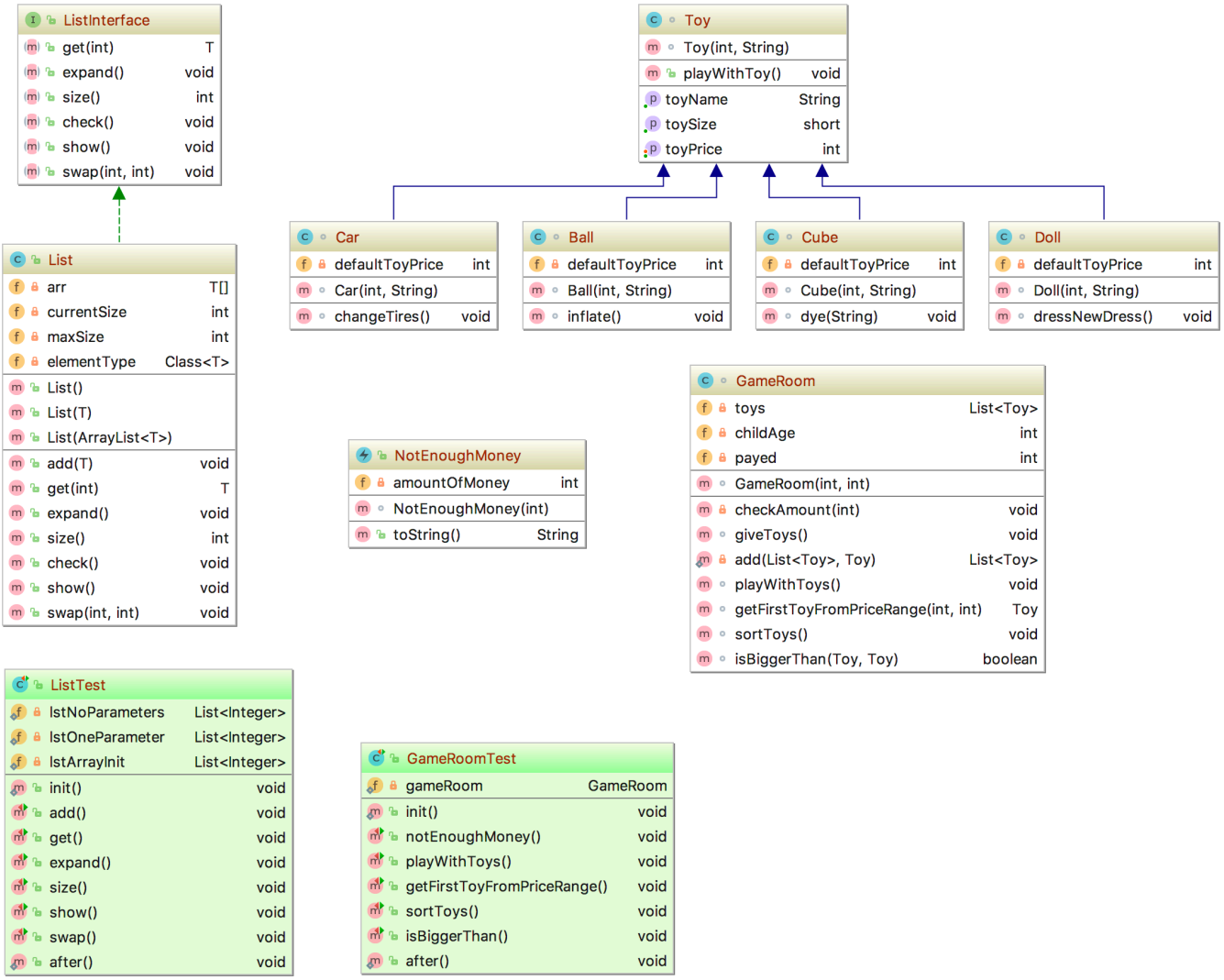
Київ 2018

ЗМІСТ

1. ПОСТАНОВКА ЗАДАЧІ

1. Модифікувати класи з попередніх лабораторних робіт (лабораторні роботи №5 та №6) таким чином, щоб обробка виключних ситуацій відбувалась за допомогою стандартних засобів мови програмування Java. Створити власний клас обробник виключних ситуацій.
2. Написати JUnit-тести для перевірки працездатності усіх методів та виключних ситуацій.
3. Всі початкові дані задаються у виконавчому методі. Код повинен відповідати стандартам JCC та бути детально задокументований.

2. ДІАГРАМА КЛАСІВ



3. ВИСНОВОК

Були складнощі з настроюванням junit тестів у Inteliij Idea в зв’язку з існуванням 5-ї версії junit, що на даний момент є бета-версією. Після того, як вдалося запустити перший тест, створення інших тестів не викликало жодних проблем.

4. КОД ПРОГРАМИ

```
import java.util.ArrayList;

/**
 * Java labs – Lab7
 * @version 1.0 2018-04-07
 * @author Misha Kushka
 */

public class Main {

    public static void main(String[] args) {

        // Init game room's object to play with it.
        GameRoom gameRoom = new GameRoom(0, 12);

        gameRoom.playWithToys();

        System.out.println("\n-- SORTED ------------------------------");

        gameRoom.sortToys();
        gameRoom.playWithToys();

        System.out.println();

        int min = 5;
        int max = 13;

        Toy firstToyFromRange = gameRoom.getFirstToyFromPriceRange(min, max);

        if (firstToyFromRange != null) {
            System.out.println("Toy from " + min + " to " + max + " is " +
firstToyFromRange.getToyName() + ".");
        } else {
            System.out.println("There is no toys from range (" + min + ", " + max + ")");
        }

    }

}



/**
 * Ball toy for girls & boys of different ages.
 */
class Ball extends Toy {

    private int defaultToyPrice = 1; // price of the toy, not considering child age

    /**
     * Call's the constructor of the parent's Toy class,
     * sets toy's price, which isn't depends on the age
     * of the child.
     *
     * @param childAge Age of the child.
     * @param newToyName Name of the toy.
     */
    Ball(int childAge, String newToyName) {
        super(childAge, newToyName);
        setToyPrice(defaultToyPrice);
    }

    /**
```

```
     * Some another method for this class.
     */
    void inflate() {
        System.out.println("Inflate the ball.");
    }

}


/**
 * Car toy for boys of different ages.
 */
class Car extends Toy {

    private int defaultToyPrice = 3; // price of the toy, not considering child age

    /**
     * Call's the constructor of the parent's Toy class,
     * sets toy's price depends of the default toy price
     * and age of the child.
     *
     * @param childAge Age of the child.
     * @param newToyName Name of the toy.
     */
    Car(int childAge, String newToyName) {
        super(childAge, newToyName);
        setToyPrice(defaultToyPrice * getToySize());
    }

    /**
     * Some another method for this class.
     */
    void changeTires() {
        System.out.println("Now your car is equiped with the new tires.");
    }

}


/**
 * Cube toy for girls & boys of different ages.
 */
class Cube extends Toy {

    private int defaultToyPrice = 4; // price of the toy, not considering child age

    /**
     * Call's the constructor of the parent's Toy class,
     * sets toy's price depends of the default toy price
     * and age of the child.
     *
     * @param childAge Age of the child.
     * @param newToyName Name of the toy.
     */
    Cube(int childAge, String newToyName) {
        super(childAge, newToyName);
        setToyPrice(defaultToyPrice * getToySize());
    }

    /**
     * Some another method for this class.
     */
    void dye(String color) {
        System.out.println("Now color of the your cube is " + color + ".");
```

```
    }

}


/**
 * Doll toy for girls of different ages.
 */
class Doll extends Toy {

    private int defaultToyPrice = 5; // price of the toy, not considering child age

    /**
     * Call's the constructor of the parent's Toy class,
     * sets toy's price depends of the default toy price
     * and age of the child.
     *
     * @param childAge Age of the child.
     * @param newToyName Name of the toy.
     */
    Doll(int childAge, String newToyName) {
        super(childAge, newToyName);
        setToyPrice(defaultToyPrice * getToySize());
    }

    /**
     * Some another method for this class.
     */
    void dressNewDress() {
        System.out.println("Now your doll wears in the new dress.");
    }

}


/**
 * Implementation of the gaming room for children
 * of different ages.
 */
class GameRoom {

    private List<Toy> toys = new List<>(); // array of toys in the game room
    private int childAge; // age of the child
    private int payed; // how much was payed for the room

    /**
     * Check is there are enough money to visit the game room.
     */
    private void checkAmount(int amount) throws NotEnoughMoney {
        if (amount < 1) {
            throw new NotEnoughMoney(amount);
        }
    }

    /**
     * Allow to pay for playing in the game room.
     * Depending on the amount of money child can
     * play with different number of toys.
     *
     * @param amount Amount of money to pay for playing.
     * @param age Age of the child in the room.
     */
    GameRoom(int amount, int age) {
        // Too low payment checker.
```

```
        try {
            checkAmount(amount);
        } catch (NotEnoughMoney e) {
            System.out.println("Error: " + e);
        }
        childAge = age;
        payed = amount;

        // Fill toys array with toys.
        giveToys();

        // Show how many toys are available depends of the payed amount.
        try {
            System.out.println("Now you can play with " + toys.size() + " toys.");
        } catch (NullPointerException e) {
            System.err.println("Add elements to the toys array first.");
            System.exit(2);
        }
    }

    /**
     * Fill the toys array with different toys object's
     * depends of the payed amount for the room.
     */
    void giveToys() {

        // Toys, which are in the room.
        List<Toy> defaultToys = new List<>();

        defaultToys.add(new Car(childAge, "super car"));
        defaultToys.add(new Doll(childAge, "cool doll"));
        defaultToys.add(new Ball(childAge, "amazing ball"));
        defaultToys.add(new Cube(childAge, "crazy cube"));

        int[] defaultToyPrices = new int[defaultToys.size()]; // prices of toys in the room

        // Set prices for all toys in the room depending on the child age.
        for (int i = 0; i < defaultToyPrices.length; i++) {
            defaultToyPrices[i] = defaultToys.get(i).getToyPrice();
        }

        int totalPrice = 0; // total price of all toys for the current child
        int iteration = 0; // number of iterations of adding toys

        // A little bit randomly choose toys for the particular child
        // depending on the child age and payed amount.
        while (totalPrice < payed) {
            switch (iteration%4) {
                case 0:
                    if (totalPrice + defaultToyPrices[0] <= payed) {
                        toys = add(toys, defaultToys.get(0));
                        totalPrice += defaultToyPrices[0];
                    }
                    break;
                case 1:
                    if (totalPrice + defaultToyPrices[1] <= payed) {
                        toys = add(toys, defaultToys.get(1));
                        totalPrice += defaultToyPrices[1];
                    }
                    break;
                case 2:
                    if (totalPrice + defaultToyPrices[2] <= payed) {
                        toys = add(toys, defaultToys.get(2));
                        totalPrice += defaultToyPrices[2];
                    }
```

```
                    break;
                default:
                    if (totalPrice + defaultToyPrices[3] <= payed) {
                        toys = add(toys, defaultToys.get(3));
                        totalPrice += defaultToyPrices[3];
                    }
                    break;
            }
            iteration++;
        }

        System.out.println("Total price: $" + totalPrice);

    }

    /**
     * Add element to the Toy's array.
     *
     * @param originalArray Array to put element into.
     * @param newItem Element to put.
     * @return New array with added element.
     */
    private static List<Toy> add(List<Toy> originalArray, Toy newItem) {
        int currentSize = originalArray.size();
        List<Toy> tempArray = new List<>();
        for (int i = 0; i < currentSize; i++) {
            tempArray.add(originalArray.get(i));
        }
        tempArray.add(newItem);
        return tempArray;
    }

    /**
     * Execute method of playing with all toys
     * of the particular child.
     */
    void playWithToys() {
        for (int i = 0; i < toys.size(); i++) {
            toys.get(i).playWithToy();
        }
    }

    /**
     * Get first toy from the setted range by toy's price.
     *
     * @param min Minimum price of the toy to find.
     * @param max Maximum price of the toy to find.
     * @return First toy with price from range if found,
     * or null otherwise.
     */
    Toy getFirstToyFromPriceRange(int min, int max) {

        // Check is min < max.
        if (min > max) {
            System.out.println("Attention! min value is bigger, than max value!");
        }

        // Iteratively find toy from the given range.
        for (int i = 0; i < toys.size(); i++) {
            if (toys.get(i).getToyPrice() >= min && toys.get(i).getToyPrice() <= max) {
                return toys.get(i);
            }
        }

        return null;
```

```
    }

    /**
     * Sort toys by the name of their classes alphabetically.
     */
    void sortToys() {
        int i, j; // iterators
        int n = toys.size(); // length of the toys array
        Toy temp; // temporary Toy object to swap elements

        // Bubble sort for the array of toys.
        for (i = 0; i < n-1; i++) {
            for (j = 0; j < n - i - 1; j++) {
                if (isBiggerThan(toys.get(j), toys.get(j+1))) {
                    toys.swap(j, j+1);
                }
            }
        }
    }

    boolean isBiggerThan(Toy first, Toy second) {
        int firstIndex, secondIndex;

        // Align class names with indexes.
        // First object.
        switch (first.getClass().getName()) {
            case ("Ball"):
                firstIndex = 0;
                break;
            case ("Car"):
                firstIndex = 1;
                break;
            case ("Cube"):
                firstIndex = 2;
                break;
            default:
                firstIndex = 3;
        }

        // Second object.
        switch (second.getClass().getName()) {
            case ("Ball"):
                secondIndex = 0;
                break;
            case ("Car"):
                secondIndex = 1;
                break;
            case ("Cube"):
                secondIndex = 2;
                break;
            default:
                secondIndex = 3;
        }

        if (firstIndex > secondIndex)
            return true;

        return false;
    }

}
```

```java
/**
 * List interface with it's main methods.
 * @param <T> Generic parameter.
 */
public interface ListInterface<T> {

    /**
     * Get array's element by index.
     * @param index Index of searching element.
     * @return Element from array by index.
     */
    public T get(int index);

    /**
     * Expand the array, if it's too small.
     */
    public void expand();

    /**
     * Get current size of the array.
     * @return Size of the array.
     */
    public int size();

    /**
     * Check if the list is empty.
     */
    public void check() throws EmptyList;

    /**
     * Show the array on the screen.
     */
    public void show();

    /**
     * Swap elements in the list.
     * @param i Index of the first element.
     * @param j Index of the second element.
     */
    public void swap(int i, int j);

}


import java.lang.reflect.Array;
import java.util.ArrayList;

/**
 * List implementation with it's main methods.
 * @param <T> Generic parameter.
 */
public class List<T> implements ListInterface<T> {

    private T[] arr; // array to store items in the list
    private int currentSize; // current length of the array
    private int maxSize; // current maximum size of the array
    private Class<T> elementType; // type of elements in the array

    /**
     * List class constructor with one parameter: type of elements.
     */
    public List() {
        maxSize = 15;
        currentSize = 0;
```

```
}

/**
 * List constructor with 2 parameters: type of elements and one element.
 * @param element Element to add to the list.
 */
public List(T element) {
    maxSize = 15;
    currentSize = 1;
    elementType = (Class<T>) element.getClass().getSuperclass();
    arr = (T[]) Array.newInstance(elementType, maxSize);
    arr[0] = element;
}

/**
 * List constructor with 2 parameters: type of elements and array of elements
 * to put in this structure.
 * @param newArr Array to put to the list.
 */
public List(ArrayList<T> newArr) {
    maxSize = 15;
    currentSize = newArr.size();

    // Array is not empty
    if (currentSize != 0) {
        elementType = (Class<T>) newArr.get(0).getClass().getSuperclass();

        while (newArr.size() > maxSize) {
            expand();
        }

        arr = (T[]) Array.newInstance(elementType, maxSize);

        for (int i = 0; i < newArr.size(); i++) {
            arr[i] = newArr.get(i);
        }
    }
}

/**
 * Add element to the end of the array.
 * @param element Element to push.
 */
public void add(T element) {
    // Expand array if it's too small
    if (currentSize >= maxSize) {
        expand();
    }

    if (elementType == null) {
        elementType = (Class<T>) element.getClass().getSuperclass();
        arr = (T[]) Array.newInstance(elementType, maxSize);
    }

    arr[currentSize] = element;
    currentSize++;
}

/**
 * Get array's element by index.
 * @param index Index of searching element.
 * @return Element from array by index.
 */
public T get(int index) {
    try {
```

```
            return arr[index];
        } catch (NullPointerException e) {
            System.err.println("Index is out of range");
        }

        return (T) null;
    }

    /**
     * Expand the array, if it's too small.
     */
    public void expand() {
        try {
            int newSize = maxSize + (int) (maxSize * 0.3);
            T[] newArr = (T[]) Array.newInstance(elementType, newSize);

            System.arraycopy(arr, 0, newArr, 0, maxSize);

            maxSize = newSize;
            arr = newArr;
        } catch (NullPointerException e) {
            System.err.println("You can't expand an empty array.");
        }
    }

    /**
     * Get current size of the array.
     * @return Size of the array.
     */
    public int size() {
        return currentSize;
    }
    /**
     * Check if the list is empty.
     */
    public void check() throws EmptyList {
        if (currentSize < 1) {
            throw new EmptyList(currentSize);
        }
    }

    /**
     * Show the array on the screen.
     */
    public void show() {
        try {
            check();
            for (int i = 0; i < currentSize; i++) {
                System.out.println(arr[i]);
            }
        } catch (EmptyList e) {
            System.out.println("Caution: " + e);
        }
    }

    /**
     * Swap elements in the list.
     * @param i Index of the first element.
     * @param j Index of the second element.
     */
    public void swap(int i, int j) {
        try {
            T temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
```

```
        } catch (NullPointerException e) {
            System.err.println("One or both of indexes are out of range.");
        }
    }

}



/**
 * Implementation of the real-world toy with it's
 * properties such as size, type, color and such things
 * to do with it as playing with toy.
 */
class Toy {

    private short toySize; // size of the toy
    private int toyPrice; // price of the toy
    public final String toyName; // name of the toy

    /**
     * Toy's constructor, which sets toy's name
     * depending on the child age.
     *
     * @param childAge Age of the child.
     * @param newToyName Part of the toy name without
     * appendix of it's size.
     */
    Toy(int childAge, String newToyName) {
        if (childAge <= 5) {
            toySize = 1;
            newToyName = "small " + newToyName;
        } else if (childAge > 5 && childAge <= 10) {
            toySize = 2;
            newToyName = "medium " + newToyName;
        } else {
            toySize = 3;
            newToyName = "big " + newToyName;
        }

        toyName = newToyName;
    }

    /**
     * Immitates process of playing with toy.
     */
    public void playWithToy() {
        System.out.println("Now child is playing with the " + toyName + ".");
    }

    /**
     * Setter for the toy price.
     * @param newPrice New price of the toy to set.
     */
    public void setToyPrice(int newPrice) {
        toyPrice = newPrice;
    }

    /**
     * Getter for the toy size.
     * @return Size of the toy.
     */
    public short getToySize() {
        return toySize;
    }
```

```
    /**
     * Getter for the toy prise.
     * @return Price of the toy.
     */
    public int getToyPrice() {
        return toyPrice;
    }

    /**
     * Getter for the toy name.
     * @return Toy's name.
     */
    public String getToyName() {
        return toyName;
    }

}



/**
 * Exception to indicate when there are not enough money to visit the game room.
 */
public class NotEnoughMoney extends Exception {

    private int amountOfMoney; // amount of money for the room

    /**
     * Sets amount of money for the game room.
     * @param amount Amount of money.
     */
    NotEnoughMoney(int amount) {
        amountOfMoney = amount;
    }

    /**
     * Caution, which tells why exception runs.
     * @return Caution about too small amount of money.
     */
    public String toString() {
        return "Sorry, but $" + amountOfMoney + " is not enough to visit the game room.";
    }
}



/**
 * Exception to indicate when list is empty.
 */
public class EmptyList extends Exception {
    private int arraySize; // size of the array

    /**
     * Set size of the array.
     * @param size Size of the array.
     */
    EmptyList(int size) {
        arraySize = size;
    }

    /**
     * Message which says that can't show the array.
     * @return Caution about empty array.
     */
    public String toString() {
```

```
            return "The list is empty, can't show it.";
        }
    }


import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;

import static org.junit.Assert.*;

/**
 * Tests for the GameRoom class.
 */
public class GameRoomTest {

    private static GameRoom gameRoom;

    @BeforeClass
    public static void init() {
        gameRoom = new GameRoom(28, 12);
    }

    @Test
    public void notEnoughMoney() {
        GameRoom gr = new GameRoom(-5, 5);
    }

    @Test
    public void playWithToys() {
        gameRoom.playWithToys();
    }

    @Test
    public void getFirstToyFromPriceRange() {
        Toy firstToyFromRange = gameRoom.getFirstToyFromPriceRange(5, 13);
        assertEquals("big super car", firstToyFromRange.getToyName());

        Toy firstToyFromRange2 = gameRoom.getFirstToyFromPriceRange(10, 13);
        assertEquals(null, firstToyFromRange2);
    }

    @Test
    public void sortToys() {
        gameRoom.sortToys();
    }

    @Test
    public void isBiggerThan() {
        List<Toy> toys = new List<>();

        toys.add(new Car(5, "super car"));
        toys.add(new Doll(5, "cool doll"));
        toys.add(new Ball(5, "amazing ball"));
        toys.add(new Cube(5, "crazy cube"));

        assertEquals(true, gameRoom.isBiggerThan(toys.get(1), toys.get(0)));
        assertEquals(true, gameRoom.isBiggerThan(toys.get(1), toys.get(2)));
        assertEquals(true, gameRoom.isBiggerThan(toys.get(3), toys.get(2)));
    }

    @AfterClass
    public static void after() {
        gameRoom = null;
```

```
    }

}


import org.junit.AfterClass;
import org.junit.BeforeClass;

import java.util.ArrayList;

import static org.junit.jupiter.api.Assertions.assertEquals;

/**
 * Tests for the List class
 */
public class ListTest {

    private static List<Integer> lstNoParameters;
    private static List<Integer> lstOneParameter;
    private static List<Integer> lstArrayInit;

    @BeforeClass
    public static void init() {
        lstNoParameters = new List<>();
        lstOneParameter = new List<>(5);

        ArrayList<Integer> arr = new ArrayList<>();

        arr.add(1);
        arr.add(-9999);
        arr.add(1034);

        lstArrayInit = new List<>(arr);
    }

    @org.junit.Test
    public void add() {
        lstNoParameters.add(15);
        lstOneParameter.add(15);
        lstArrayInit.add(15);

        lstNoParameters.add(-12);
        lstOneParameter.add(-12);
        lstArrayInit.add(-12);
    }

    @org.junit.Test
    public void get() {
        assertEquals((long)15, (long)lstNoParameters.get(0));
        assertEquals((long)5, (long)lstOneParameter.get(0));
        assertEquals((long)1, (long)lstArrayInit.get(0));

        lstOneParameter.get(15);
        lstOneParameter.get(15);
        lstArrayInit.get(15);
    }

    @org.junit.Test
    public void expand() {
        lstNoParameters.expand();
        lstOneParameter.expand();
        lstArrayInit.expand();
    }
```

```
    @org.junit.Test
    public void size() {
        assertEquals(0+2, lstNoParameters.size());
        assertEquals(1+2, lstOneParameter.size());
        assertEquals(3+2, lstArrayInit.size());
    }

    @org.junit.Test
    public void show() {
        lstNoParameters.show();
        lstOneParameter.show();
        lstArrayInit.show();

        List<Integer> lst = new List<>();
        lst.show();
    }

    @org.junit.Test
    public void swap() {
        lstNoParameters.swap(0, 1);
        lstOneParameter.swap(0, 1);

        lstArrayInit.swap(0, 1);
        assertEquals((long)-9999, (long)lstArrayInit.get(0));
    }

    @AfterClass
    public static void after() {
        lstNoParameters = null;
        lstOneParameter = null;
        lstArrayInit = null;
    }
}
```