МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ФАКУЛЬТЕТ ІНФОРМАТИКИ І ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

НАЦІОНАЛЬНОГО ТЕХНІЧНОГО УНІВЕРСИТЕТУ УКРАЇНИ

"КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ"

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

ЛАБОРАТОРНА РОБОТА **№8**

З КУРСУ "**ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ**"

Виконав: **студент 1 курсу**

**гр. ІО-35**

**Головенько В.Д.**

Номер залікової книжки: **3504**

Прийняв: **Невдащенко М.В.**

Київ - 2014

**Тема:** Обробка виключних ситуацій та основи тестування в мові програмування Java.

**Мета:** Здобуття навичок у використанні механізму обробки виключних ситуацій та написанні тестів для перевірки працездатності методів в мові програмування Java.

Модифікувати класи з попередніх лабораторних робіт таким чином, щоб обробка виключних ситуацій відбувалась за допомогою стандартних засобів мови програмування Java. Створити власний клас-обробник виключних ситуацій.

Написати JUnit-тести для перевірки працездатності усіх методів та виключних ситуацій.

Всі початкові дані задаються у виконавчому методі. Код повинен відповідати стандартам JCC та бути детально задокументований.

**Код програми (модифіковані методи):**

**Основний клас з виконавчим методом:**

/\*\*Класс, выполняющий задание шестой лабораторной работы.

\* **@author** Golowenko Slava

\*/

**public** **class** Bouquet {

/\*\*

\* Метод, возвращающий результаты работы программы.

\*/

**public** **static** String printFlowerParametrs(Flower f){

**return** "Flower \"" + f.getNameOfFlower() + "\" cost " + f.getPrice() + " UAH" + ", her length is " + f.getLength() + " cm" + ", her freshness is " + f.getFreshness() + " days.";

}

/\*\*

\* Метод, выводящий результы работы программы на консоль.

\*/

**public** **static** **void** outputArray(MyCollection bouquet){

**for** (**int** i = 0; i < bouquet.size(); i++){

System.*out*.println(*printFlowerParametrs*(bouquet.get(i)));

}

}

/\*\*

\* Метод поиска цветка по длинне.

\*/

**public** **static** Flower searchByFlowerLength(MyCollection bouquet, **double** length1, **double** length2){

Flower result = bouquet.get(0);

**for** (**int** i = 0; i < bouquet.size(); i++){

**if** (bouquet.get(i).getLength() >= length1 && bouquet.get(i).getLength() <= length2)

result = bouquet.get(i);

}

**return** result;

}

/\*\*

\* Метод, который считает сумму за букет цветов.

\*/

**public** **static** **int** totalBouquetPrice(MyCollection bouquet) {

**int** totalPrice = 0;

**for** (**int** i = 0; i < bouquet.size(); i++) {

totalPrice += bouquet.get(i).getPrice();

}

System.*out*.println("Total bouquet price: " + totalPrice + " UAH");

**return** totalPrice;

}

/\*\*

\* Метод сортировки цветов по свежести.

\*/

**public** **static** **void** sortByFreshness(MyCollection bouquet, **int** k) {

System.*out*.println("Bouquet is sorted by decrision of freshness:");

Flower buf;

**for** (**int** i = 0; i <= k - 1; i++) {

**for** (**int** j = 0; j <= k - 1; j++) {

**if** (bouquet.get(i).getFreshness() > bouquet.get(j).getFreshness()) {

buf = bouquet.get(i);

bouquet.set(i,bouquet.get(j));

bouquet.set(j,buf);

}

}

}

}

/\*\*

\* Основной класс.

\*/

**public** **static** **void** main(String[] args) {

**final** **int** count = 7;

/\*\*

\* Коллекция цветов.

\*/

MyCollection bouquet=**new** MyCollection();

bouquet.add (**new** Rose(3,50.5,80));

bouquet.add (**new** Rose(4,49.6,80));

bouquet.add (**new** Rose(3,48.0,80));

bouquet.add (**new** Orchid(2,28.3,120));

bouquet.add (**new** Orchid(2,27.9,110));

bouquet.add (**new** Lily(1,15.4,50));

bouquet.add (**new** Rose());

/\*\*

\* Вывод коллекции цветов на консоль.

\*/

System.*out*.println("Bouqeot consist of:");

*outputArray*(bouquet);

/\*\*

\* Вывод отсортированной по свежести коллекции цветов.

\*/

System.*out*.println();

*sortByFreshness*(bouquet, count);

*outputArray*(bouquet);

/\*\*

\* Вывод на консоль результата из поиска цветка по длинне.

\*/

System.*out*.println();

System.*out*.println("Serched Flower by length:");

System.*out*.println(*printFlowerParametrs*(*searchByFlowerLength*(bouquet, 27.0, 28.0)));

System.*out*.println();

*totalBouquetPrice*(bouquet);

}

}

**Клас квітка:**

/\*\*Класс является родительским классом цветов.

\* **@author** Golowenko Slava

\*/

**public** **class** Flower {

/\*\*

\* Свежесть цветка, указывается в количестве дней хранения.

\*/

**private** **int** freshness;

/\*\*

\* Длинна цветка, указывается в сантиметрах.

\*/

**private** **double** length;

/\*\*

\* Цена цветка, указывается в гривнах.

\*/

**private** **int** price;

/\*\*

\* Название цветка.

\*/

**private** String nameOfFlower;

/\*\*

\* Максимальный срок хранения (свежесть) цветка.

\*/

**private** **int** MAX\_FRESHNESS = 5;

/\*\*

\* Максимальная длинна цветка.

\*/

**private** **double** MAX\_LENGTH = 100;

/\*\*

\* Максимальная цена цветка.

\*/

**private** **int** MAX\_PRICE = 120;

/\*\*

\* Гэтер возвращает свежесть (срок хранения) цветка.

\* **@return** сколько дней хранится цветок.

\*/

**public** **int** getFreshness(){

**return** freshness;

}

/\*\*

\* Гэтер возвращает длинну цветка.

\* **@return** длинна цветка.

\*/

**public** **double** getLength(){

**return** length;

}

/\*\*

\* Гэтер возвращает цену цветка.

\* **@return** цена цветка.

\*/

**public** **int** getPrice(){

**return** price;

}

/\*\*

\* Гэтер возвращает название цветка.

\* **@return** название цветка.

\*/

**public** String getNameOfFlower(){

**return** nameOfFlower;

}

/\*\*

\* Сэтер устанавливает свежесть цветка.

\* **@param** f устанавливает свежесть цветка.

\*/

**public** **void** setFreshness(**int** f){

**if** (f <= MAX\_FRESHNESS){

freshness = f;

}

}

/\*\*

\* Сэтер устанавливает длинну цветка.

\* **@param** l устанавливает длинну цветка.

\*/

**public** **void** setLength(**double** l){

**if** (l <= MAX\_LENGTH){

length = l;

}

}

/\*\*

\* Сэтер устанавливает цену цветка.

\* **@param** p устанавливает цену цветка.

\*/

**public** **void** setPrice(**int** p){

**if** (p <= MAX\_PRICE){

price = p;

}

}

/\*\*

\* Сэтер устанавливает название цветка.

\* **@param** fl название цветка.

\*/

**public** **void** setNameOfFlower(String fl){

nameOfFlower = fl;

}

/\*\*

\* Конструктор без параметров, устаналивает значения полей класса, ввёденных в конструкторе.

\*/

**public** Flower(){

freshness = 5;

length = 25.5;

price = 100;

nameOfFlower = "Hrizantema";

}

/\*\*

\* Конструктор с параметрами, устанавливает значения полей класса, ввёденных как параметры конструктора при его вызове.

\* **@param** f Свежесть цветка.

\* **@param** l Длинна цветка.

\* **@param** p Цена цветка.

\*/

**public** Flower(**int** f, **double** l, **int** p){

**if** (f <= MAX\_FRESHNESS) freshness = f;

**if** (l <= MAX\_LENGTH) length = l;

**if** (p <= MAX\_PRICE) price = p;

nameOfFlower = "Hrizantema";

}

/\*\*

\* Метод возвращает цену букета.

\* **@return** возвращает цену букета.

\*/

**public** **int** bouquetTotalPrice(){

**int** totalPrice = getPrice();

**return** totalPrice;

}

}

**Клас колекція:**

**import** java.util.Arrays;

**import** java.util.Collection;

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.ListIterator;

/\*\*

\* Класс, реализирующий коллекцию.

\* **@author** Golowenko Slava

\*/

**public** **class** MyCollection **implements** List<Flower> {

**private** Object[] elements; // Массив из объектов.

**private** **int** size; // Количество добавленных эллементов в массив.

**private** Flower[] a = **new** Flower[15] ; // Объект для итератора.

**private** **int** count = 0; // Значение счётчика для итератора.

/\*\*

\* Пустой конструктор.

\*/

**public** MyCollection() {

elements = **new** Flower[15];

size = 0;

}

/\*\*

\* Конструктор, в который передаётся 1 объект обобщённого класса.

\* **@param** o объект обобщённого класса.

\*/

**public** MyCollection(MyCollection o) {

**this**();

size++;

elements[0] = o;

}

/\*\*

\* Конструктор, в который передаётся стандартная коллекции объектов.

\* **@param** c стандартная коллекции объектов.

\*/

**public** MyCollection(Collection<Flower> c) {

elements = c.toArray();

size = elements.length;

checkCapacity(15);

}

/\*\*

\* Добавляет элемент в конец массива и в случае удачного добавления элемента возвращает true, в противном случае false.

\*/

@Override

**public** **boolean** add(Flower o) {

**if** (o != **null** && contains(o)) {

checkCapacity(size + 1);

elements[size] = o;

size++;

**return** **true**;

} **else** {

**return** **false**;

}

}

/\*\*

\* Добавляет элемент в массив по указанному индексу, сдвигая идущие после него элементы.

\*/

@Override

**public** **void** add(**int** index, Flower o) {

checkCapacity(index + 1);

System.*arraycopy*(elements, index, elements, index + 1, size - index);

elements[index] = o;

size++;

}

/\*\*

\* Добавляет все элементы в указанной коллекции в этой коллекции.

\* **@return** true, если все элементы в указанной коллекции добавлены.

\*/

@Override

**public** **boolean** addAll(Collection<? **extends** Flower> c) {

Object[] a = c.toArray();

checkCapacity(size + a.length);

System.*arraycopy*(a, 0, elements, size, a.length);

**return** a.length != 0;

}

/\*\*

\* Добавляет все элементы в указанный массив в этой коллекции начиная с index.

\* **@return** true, если все элементы в указанном массиве добавлены.

\*/

@Override

**public** **boolean** addAll(**int** index, Collection<? **extends** Flower> c) {

**int** newSize = size()+c.size();

Object[] tmpArray = c.toArray();

**for**(**int** i=0; i<tmpArray.length; i++){

add(index, (Flower) tmpArray[i]);

index++;

}

**return** newSize==size();

}

/\*\*

\* Полностью очищает массив.

\*/

@Override

**public** **void** clear() {

**for** (**int** i = 0; i < size; i++)

elements[i] = **null**;

size = 0;

}

/\*\*

\* Возвращает true, если этот массив содержит указанный элемент.

\*/

@Override

**public** **boolean** contains(Object o) {

**int** i = 0;

**while** ((i < size) && (elements[i] != o))

i++;

**if** (i == size)

**return** **true**;

**return** **false**;

}

/\*\*

\* Возвращает true, если этот массив содержит все элементы указанной коллекции.

\*/

@Override

**public** **boolean** containsAll(Collection<?> c) {

Object[] a = c.toArray();

**for** (**int** i = 0; i < a.length; i++)

**if** (contains(a[i]) == **false**)

**return** **false**;

**return** **true**;

}

/\*\*

\* Возвращает элемент в указанной позиции в этом массиве.

\*/

@Override

**public** Flower get(**int** index) {

**return** (Flower) elements[index];

}

/\*\*

\* Возвращает индекс первого вхождения заданного элемента в этом массиве, или -1, если этот массив не содержит элементов.

\*/

@Override

**public** **int** indexOf(Object o) {

**for** (**int** i = 0; i < size; i++) {

**if** (o.equals(elements[i]))

**return** i;

}

**return** -1;

}

/\*\*

\* Возвращает true, если этот массив пустой и false если в массиве есть хотя бы один элемент.

\*/

@Override

**public** **boolean** isEmpty() {

**return** size() == 0;

}

/\*\*

\* Возвращает индекс последнего вхождения заданного элемента в этом массиве, или -1, если этот массив не содержит элементов.

\*/

@Override

**public** **int** lastIndexOf(Object o) {

**int** checkIndex = -1;

**if** (o != **null**) {

**for** (**int** i = 0; i < size(); i++) {

**if** (o.equals(elements[i])) {

checkIndex = i;

}

elements[i] = elements[i+1];

}

**return** checkIndex;

}

**return** -1;

}

/\*\*

\* Удаляет первое вхождение указанного элемента из этого массива, если он присутствует, в случае

\* **@return** true, если элемент массива был удален.

\*/

@Override

**public** **boolean** remove(Object o) {

**int** i = 0;

**while** (o != elements[i])

i++;

**if** (i == 0)

**return** **false**;

**int** numMoved = size - i - 1;

**if** (numMoved > 0)

System.*arraycopy*(elements, i + 1, elements, i, numMoved);

elements[--size] = **null**;

**return** **true**;

}

/\*\*

\* Удаляет элемент массива по указаному индексу.

\* **@return** элемент в ранее указанной позиции.

\*/

@Override

**public** Flower remove (**int** index) {

**int** numMoved = size - index - 1;

**if** (numMoved > 0)

System.*arraycopy*(elements, index + 1, elements, index, numMoved);

elements[--size] = **null**;

**return** **null**;

}

/\*\*

\* Удаляет все элементы, которые содержатся в указанной коллекции.

\* **@return** true, если элементы удалены.

\*/

@Override

**public** **boolean** removeAll(Collection<?> c) {

Object[] a = c.toArray();

**boolean** check = **false**;

**for** (**int** i = 0; i < a.length; i++) {

remove(a[i]);

**if** (remove(a[i]))

check = **true**;

}

**return** check;

}

/\*\*

\* Сохраняет только те элементы, которые содержатся в указанной коллекции.

\* **@return** true, если этот массив изменился в результате вызова.

\*/

@Override

**public** **boolean** retainAll(Collection<?> c) {

Object[] a = c.toArray();

**boolean** check = **false**;

**for** (**int** i = 0; i < a.length; i++)

**if** (contains(a[i]) == **false**) {

remove(a[i]);

**if** (remove(a[i]))

check = **true**;

}

**return** check;

}

/\*\*

\* Изменяет элемент массива по указаном индексу.

\* **@return** элемент ранее в указанной позиции.

\*/

@Override

**public** Flower set (**int** index, Flower o) {

**if** (checkCapacity(index) && elements[index] == **null**)

size++;

elements[index] = o;

**return** o;

}

/\*\*

\* Возвращает размер массива.

\*/

@Override

**public** **int** size() {

**return** size;

}

/\*\*

\* Метод проверяет достаточно ли места для записи нового элемента. И если не

\* достаточно, то увеличивает размер массива на 30%.

\*/

**private** **boolean** checkCapacity(**int** takenCapacity) {

**int** oldCapacity = elements.length;

**if** (takenCapacity > oldCapacity) {

**int** newCapacity = oldCapacity + (oldCapacity \* (**int**)0.3);

**if** (newCapacity < takenCapacity)

newCapacity = takenCapacity;

elements = Arrays.*copyOf*(elements, newCapacity);

**return** **true**;

}

**return** **false**;

}

/\*\*

\* Возвращает видимую часть этого массива между указанными fromIndex и toIndex.

\*/

@Override

**public** List<Flower> subList(**int** fromIndex, **int** toIndex) {

MyCollection deposit = **new** MyCollection();

List<Flower> ar = deposit.subList(1,4);

System.*out*.println("Мини-коллекция" + ar);

**return** deposit;

}

/\*\*

\* Возвращает массив, содержащий все элементы в этой коллекции в правильной последовательности.

\*/

@Override

**public** Object[] toArray() {

**return** Arrays.*copyOf*(elements, size);

}

/\*\*

\* Возвращает массив, содержащий все элементы в этой коллекции в правильной последовательности

\* возвращаемым типом массива является то, что переданно в качестве параметра.

\*/

@SuppressWarnings("unchecked")

@Override

**public** <T> T[] toArray(T[] a) {

**if** (a.length < size)

**return** (T[]) Arrays.*copyOf*(elements, size, a.getClass());

System.*arraycopy*(elements, 0, a, 0, size);

**if** (a.length > size)

a[size] = **null**;

**return** a;

}

/\*\*

\* Метод инициализирует итератор.

\*/

@Override

**public** Iterator<Flower> iterator() {

**return** **new** MyIterator(a,count);

}

/\*\*

\* **@return** возвращает итератор, установленый на начало списка.

\*/

@Override

**public** ListIterator<Flower> listIterator() {

**return** listIterator();

}

/\*\*

\* **@param** g индекс, на который нужно установить итератор.

\* **@return** возвращает итератор, установленый на задней позиции.

\*/

@Override

**public** ListIterator<Flower> listIterator(**int** g) {

**return** listIterator(g);

}

}

**Клас обробки виключної ситуації:**

/\*\*

\* **@author** Golowenko Slava

\* Класс обработки исключительной ситуации.

\*/

**public** **class** FlowerException **extends** Exception {

**private** **int** p;

**public** FlowerException(**int** a){

p=a;

}

**public** String toString(){

**return**("Цена не может быть отрицательной!");

}

}

**Клас тестування:**

**import** **static** org.junit.Assert.\*;

**import** junit.framework.Assert;

**import** org.junit.AfterClass;

**import** org.junit.Before;

**import** org.junit.BeforeClass;

**import** org.junit.Test;

Клас

/\*\*

\* Класс тестировщик для проверки работоспособности методов.

\* **@author** Golowenko Slava \*

\*/

**public** **class** TestMyList {

**private** **static** MyCollection *mycol1ection*;

@BeforeClass

/\*\*

\* Метод выполняющийся перед классом - создает экземпляр коллекции депозитов.

\*/

**public** **static** **void** beforeClass(){

*mycol1ection* = **new** MyCollection();

}

@Before

/\*\*

\* Метод выполняющийся перед тестом наполняет коллекцию.

\*/

**public** **void** initialize() {

**try** {

*mycol1ection*.add(**new** Flower(5, 20, 2));

*mycol1ection*.add(**new** Flower(10, 10000, 7));

*mycol1ection*.add(**new** Flower(3,25000,3));

*mycol1ection*.add(**new** Flower());

} **catch** (Exception e) {

}

}

@Test

/\*\*

\* Тест-метод для проверки правильности работы метода проверки коллекции на пустоту.

\*/

**public** **void** test\_isEmpty\_mycol1\_equals\_true(){

**if**(*mycol1ection*.size()>0)

**for**(**int** i = *mycol1ection*.size(); i > 0; i--)

*mycol1ection*.remove(i);

Assert.assertEquals("Fail", **true**,*mycol1ection*.isEmpty());

}

@Test

/\*\*

\* Тест для проверки на работоспособность метода добавления при попытке добавить в коллекцию null.

\*/

**public** **void** test\_add\_null() {

Assert.assertEquals("Fail", **false**, *mycol1ection*.add(**null**));

}

@Test

/\*\*

\* Тест для проверки метода установки значения элемента коллекции в определённой позиции.

\*/

**public** **void** set\_true(){

Flower o = **new** Flower();

*mycol1ection*.set(3, o);

Assert.assertEquals("Fail", o, *mycol1ection*.get(3));

}

@Test

/\*\*

\* Тест для проверки на работоспособность метода нахождения индекса вхождения тарифа в коллекцию.

\*/

**public** **void** indexOf\_3(){

Flower o = **new** Flower();

*mycol1ection*.set(3, o);

Assert.assertEquals("Fail", 3, *mycol1ection*.indexOf(o));

}

@Test

/\*\*

\* Метод-тест проверяющий правильность работы удаления из коллекции элемента с заданным индексом.

\*/

**public** **void** remove\_4(){

Flower i = **new** Flower();

*mycol1ection*.set(4, i);

Assert.assertEquals("Fail", i, *mycol1ection*.remove(4));

}

@Test

/\*\*

\* Тест для проверки работоспособности метода увеличения размера коллекции.

\*/

**public** **void** checkCapacity(){

Assert.assertEquals("Fail", **true**, *mycol1ection*.checkCapacity((*mycol1ection*.size()\*3)));

}

@Test

/\*\*

\* Тест для проверки на работоспособность метода нахождения последнего вхождения депозита в коллекцию.

\*/

**public** **void** lastIndexOf(){

Flower i = **new** Flower();

*mycol1ection*.set(0, i);

*mycol1ection*.set(2, i);

*mycol1ection*.set(*mycol1ection*.size(), i);

Assert.assertEquals("Fail", *mycol1ection*.size(), *mycol1ection*.lastIndexOf(i));

}

@Test

/\*\*

\* Тестовый метод для проверки работоспособности метода осуществляющего очистку коллекции.

\*/

**public** **void** clear(){

*mycol1ection*.clear();

**for**(**int** i = 0; i < *mycol1ection*.size(); i++)

Assert.assertEquals("Fail", **null**, *mycol1ection*.get(i));

}

@Test

/\*\*

\* Тест для метода, который проверяет депозит на его наличие в

коллекции.

\*/

**public** **void** contains(){

Flower a = **new** Flower();

*mycol1ection*.add(a);

Assert.assertEquals("Fail", **true**, *mycol1ection*.contains(a));

}

@AfterClass

/\*\*

\* Метод выполняющийся после класса(после всех тестов), который выводит на экран сообщение с именем автора.

\*/

**public** **static** **void** authorName(){

System.*out*.println("Golowenko Slava");

}  
}

**Опис роботи програми:**

Обробка виключних ситуацій відбувається за допомогою стандартних засобів мови програмування Java та створених власноруч класів обробників виключних ситуацій. Також створений клас тестувальник для перевірки працездатності методів.

**Висновок:**

В даній лабораторній роботі ми ознайомились з обробкою виключних ситуацій в мові програмування Java. Виконавши дану роботу я навчився створювати власні та використовувати стандартні методи обробки виключних ситуацій, а також тестувати методи в мові програмування Java.