Національний Технічний Університет України

«Київський Політехнічний Інститут»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Лабораторна робота №7

Робота з колекціями в мові програмування Java.

|  |  |
| --- | --- |
| Прийняв  Ст.в. Невдащенко М.В.  «\_\_»\_\_\_\_\_\_\_\_ 2014 р. | Виконала Студентка 1-ого курсу ФІОТ  групи ІО-32  Шапран К.О. |

**Варіант:**

№ залікової книжки – 3229

С2 = 1 – інтерфейс Set

C3 = 1 – однозв’язний список

**Завдання:**

Створити клас, що описує типізовану колекцію (типом колекції є узагальнений клас з лабораторної роботи №6), що реалізує заданий варіантом інтерфейс (п.2) та має задану внутрішню структуру (п.3). Реалізувати всі методи інтерфейсу, а також створити не менше ніж 3 конструктори (1 – порожній, 2 – в який передається 1 об’єкт узагальненого класу, 3 – в який передається стандартна колекція об’єктів). Всі початкові дані задаються у виконавчому методі. Код повинен відповідати стандартам JCC та бути детально задокументований.

**Лістинг:**

Клас Main

/\*\*

\* @(#) Main.java 1.0 03/06/14

\*

\* Copyright (c) 2014 Karina Shapran

\*/

/\*\* Main class, which has executive method main

\*

\* **@version** 1.0 03 June 2014

\* **@author** Karina Shapran

\* **@since** 1.0

\*/

**public** **class** Main {

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

/\*

\* Create new collection

\*/

StoneCollection<Stones> necklace = **new** StoneCollection<Stones>();

Stones s1 = **new** Stones("Rubin", 3.95, 600, 2.1);

Stones s2 = **new** Stones("Agate", 5.89, 200, 1.7);

Stones s3 = **new** Stones("Quartz ", 10.2, 573, 5.3);

Stones s4 = **new** Stones("Diamond", 2.72, 11830, 15.1);

Stones s5 = **new** Stones("Hyacinth", 3.91, 167, 2.56);

Stones s6 = **new** Stones("Onyx", 8.34, 720, 4.21);

/\*Filling the collection of stones\*/

necklace.add(s1);

necklace.add(s2);

necklace.add(s3);

necklace.add(s1);

necklace.remove(s2);

StoneCollection<Stones> necklace1 = **new** StoneCollection<Stones>();

necklace1.add(s4);

necklace1.add(s5);

necklace1.add(s6);

necklace.addAll(necklace1);

/\*Adding collection to array \*/

Object[] mas = necklace.toArray();

**for** (Object object : mas) {

Stones s = (Stones)object;

System.*out*.println(s.getName());

}

}

}

Клас ListElement

/\*\*

\* @(#) ListElement.java 1.0 03/06/14

\*

\* Copyright (c) 2014 Karina Shapran

\*/

/\*\* Class ListElement, which have link list of the collection

\*

\* **@version** 1.0 03 June 2014

\* **@author** Karina Shapran

\* **@since** 1.0

\*/

**class** ListElement {

/\*\*

\* **@param** next, a pointer to the next element

\* **@param** data, an element of Stones

\*/

ListElement next;

Stones data;

}

Клас List

/\*\*

\* @(#) List.java 1.0 03/06/14

\*

\* Copyright (c) 2014 Karina Shapran

\*/

/\*\*

\* Class List, which realize singly-linked list

\* **@version** 1.0 03 June 2014

\* **@author** Karina Shapran

\* **@since** 1.0

\*/

**class** List {

/\*\*

\* **@param** head, pointer of the first element

\* **@param** tail, pointer of the first element

\*/

**private** ListElement head;

**private** ListElement tail;

/\*\*

\* Method, which get the first element of the list

\* **@return** head, the first element

\*/

**public** ListElement getHead() {

**return** head;

}

/\*\*

\* Method, which adds new elements to the beginning of a list

\* **@param** data

\*/

**void** addFront(Stones data) {

ListElement a = **new** ListElement();

a.data = data;

**if** (head == **null**) {

head = a;

tail = a;

} **else** {

a.next = head;

head = a;

}

}

/\*\*

\* Method, which adds new element to the end of a list

\* **@param** data

\*/

**void** addBack(Stones data) {

ListElement a = **new** ListElement();

a.data = (Stones) data;

**if** (tail == **null**) {

head = a;

tail = a;

} **else** {

tail.next = a;

tail = a;

}

}

/\*\*

\* Method, which deletes an element

\*/

**void** delEl(Stones data) {

**if** (head == **null**)

**return**;

**if** (head == tail) {

head = **null**;

tail = **null**;

**return**;

}

**if** (head.data == data) {

head = head.next;

**return**;

}

ListElement t = head;

**while** (t.next != **null**) {

**if** (t.next.data == data) {

**if** (tail == t.next) {

tail = t;

}

t.next = t.next.next;

**return**;

}

t = t.next;

}

}

}

Клас StoneCollection

/\*\*

\* @(#) StoneCollection.java 1.0 03/06/14

\*

\* Copyright (c) 2014 Karina Shapran

\*/

**import** java.lang.reflect.Array;

**import** java.util.Collection;

**import** java.util.Iterator;

**import** java.util.Set;

/\*\* Class StoneCollection, which has methods

\* and constructors for collection of the stones

\*

\* **@version** 1.0 03 June 2014

\* **@author** Karina Shapran

\* **@since** 1.0

\*/

**public** **class** StoneCollection <T> **implements** Set<Stones> {

/\*\*

\* **@param** s, new list

\* **@param** size, size of the list

\*/

**private** List s = **new** List();

**private** **int** size = 0;

/\*\*

\* Constructs an empty list

\*/

**public** StoneCollection (){

}

/\*\*

\* Passing an object to the list

\* **@param** stn, an element of the Stones

\*/

**public** StoneCollection (Stones stn){

add(stn);

}

/\*\*

\* Constructs a list containing

\* the elements of the specified collection

\* **@param** stn, the collection whose elements

\* are to be placed into this list

\*/

**public** StoneCollection(Collection<? **extends** Stones> stn){

addAll(stn);

}

/\*\*

\* Method, which adds an element to the beginning of a list

\* **@param** e, an element of the Stones

\* **@param** flag

\* **@return** flag

\*/

@Override

**public** **boolean** add(Stones stn) {

**boolean** flag = **true**;

ListElement t = s.getHead();

**while** (t != **null**) {

**if** (t.data==stn) {

flag = **false**;

**break**;

}

t = t.next;

}

**if** (flag) {

s.addBack(stn);

size++;

}

**return** flag;

}

/\*\*

\* Method, which adds all elements of the collection

\* **@param** c, the collection whose elements

\* are to be placed into this list

\* **@param** flag

\* **@return** flag

\*/

@Override

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

Iterator<? **extends** Stones> iterator = c.iterator();

**boolean** flag = **false**;

**while** (iterator.hasNext()) {

flag = add((Stones) iterator.next());

}

**return** flag;

}

/\*\*

\* Method, which cleans the list

\*/

@Override

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

s = **null**;

s = **new** List();

}

/\*\*

\* Method, which checks for the existence of the object

\* **@param** flag

\* **@return** flag

\*/

@Override

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

**boolean** flag = add((Stones) o);

**if** (flag) {

s.delEl((Stones) o);

**return** flag==**false**;

}

}

/\*\*

\* Method, which checks for the existence of the collection

\* **@param** flag

\* **@return** flag

\*/

@Override

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

**boolean** flag = **true**;

Iterator<Stones> iterator = (Iterator<Stones>) c.iterator();

**while** (iterator.hasNext()) {

flag = contains((Stones) iterator.next());

**if** (flag==**false**){

**break**;

}

}

**return** flag;

}

/\*\*

\* Method, which checks the emptiness of the list

\*/

@Override

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

**return** s.getHead()==**null**;

}

/\*\*

\* Method, which return iterator

\*

\*/

@Override

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

**return** **new** StoneIterator();

}

/\*\*

\* Method, which removes an element of the list

\* **@param** o, object of the collection

\*/

@Override

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

**if** (contains(o)) {

s.delEl((Stones) o);

size--;

**return** **true**;

} **else** {

**return** **false**;

}

}

/\*\*

\* Method, which removes all elements of the collection

\* **@param** flag;

\* **@return** flag;

\*/

@Override

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

**boolean** flag = **false**;

**for** (Object object : c) {

flag = remove(object);

}

**return** flag;

}

/\*\*

\* Method, which implements the set intersection

\* **@param** c, the collection whose elements

\* are to be placed into this list

\* **@param** flag

\* **@return** flag

\*/

@Override

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

**boolean** flag = **true**;

Iterator<Stones> iterator = iterator();

**while** (iterator.hasNext()) {

Stones stones = (Stones) iterator.next();

flag = c.contains(stones);

**if** (flag == **false**)

remove(stones);

}

**return** flag == **false**;

}

/\*\*

\* Method, which have size of the collection

\* **@return** size;

\*/

@Override

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

**return** size;

}

/\*\*

\* Method, which turns objects of collection to array

\* **@param** i

\* **@return** mas

\*/

@Override

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

**int** i = 0;

Object[] mas = **new** Object[size];

Iterator<Stones> iterator = iterator();

**while** (iterator.hasNext()) {

mas[i] = iterator.next();

i++;

}

**return** mas;

}

/\*\*

\* Method, which records elements of the collection

\* in the array until it has a place

\* **@param** array;

\* **@param** size;

\* **@param** i;

\* **@return** array;

\*/

@Override

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

**int** size = size();

**if** (array.length < size) { array = (T[]) Array.*newInstance*(array.getClass().getComponentType(), size);

} **else** **if** (array.length > size) {

array[size] = **null**;

}

**int** i = 0;

**for** (Stones e: **this**) {

array[i] = (T) e;

i++;

}

**return** array;

}

/\*\*

\* Interal class StoneIterator, which implements interface of Iterator

\***@version** 1.0 03 June 2014

\* **@author** Karina Shapran

\* **@since** 1.0

\*

\*/

**class** StoneIterator **implements** Iterator {

/\*\*

\* **@param** thisElement

\*/

**private** ListElement thisElement = s.getHead();

/\*\*

\* Method, which (checks if the next element exists)

\*/

@Override

**public** **boolean** hasNext() {

**return** thisElement != **null**;

}

/\*\*

\* Method,which have next element of the collection

\* **@param** s, an element of the Stones

\* **@return** s;

\*/

@Override

**public** Stones next() {

Stones s = thisElement.data;

thisElement = thisElement.next;

**return** s;

}

/\*\*

\* Method, which remove elements of the collection

\*/

@Override

**public** **void** remove() {

s.delEl(thisElement.data);

}

}

}

Клас Stones

/\*\*

\* @(#) Stones.java 1.0 03/06/14

\*

\* Copyright (c) 2014 Karina Shapran

\*/

/\*\* Class Stones, which describes the stone

\*

\* **@version** 1.0 03 June 2014

\* **@author** Karina Shapran

\* **@since** 1.0

\*/

**public** **class** Stones {

/\*\*

\* **@param** name

\* **@param** weight

\* **@param** price

\* **@param** clarity

\*/

**private** **double** weight, clarity;

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

**private** String name;

/\*\*Constructor of Stones\*/

Stones(String name, **double** weight, **int** price, **double** clarity){

**this**.name = name;

**this**.weight = weight;

**this**.price = price;

**this**.clarity = clarity;

}

/\*\*

\* **@return** name

\*/

**public** String getName(){

**return** name;

}

/\*\*

\* **@return** weight

\*/

**public** **double** getWeight(){

**return** weight;

}

/\*\*

\* **@return** price

\*/

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

**return** price;

}

/\*\*

\* **@return** clarity

\*/

**public** **double** getClarity(){

**return** clarity;

}

}

**Опис програми:**

StoneCollection – клас, що описує колекцію із внутрішньою структурою однозв’язний список, однозв’язний список описується в класі List. Клас StoneCollection реалізує всі методи інтерфейса Set. Клас Stones описує об’єкти, з яких складається колекція, клас ListElement описує один елемент колекції, що містить об’єкт та посилання на наступний елемент.

**Висновок:**

У даній лабораторній роботі ми ознайомились із колекціями у мові програмування Java і здобули навичок у створенні власних та використанні стандартних колекцій в мові програмування Java, створивши власну типізовану колекцію із внутрішньою структурою однозв’язний список.