Лабораторна робота №10.

Обробка параметризованих контейнерів

**Мета роботи:**

* Розширення функціональності параметризованих класів.

**1. Вимоги до лабораторної роботи**

* 1. **Розробник:**

*студент Литвин Ігнатій Ігоревич; КІТ-26А; Варіант №7*

* 1. **Загальне завдання**

Використовуючи програму рішення завдання лабораторної роботи №9:

1. Розробити параметризовані методи (Generic Methods) для обробки колекцій об’єктів згідно прикладної задачі.
2. Продемонструвати розроблену функціональність (створення, управління та обробку власних контейнерів) в діалоговому та автоматичному режимах.

* Автоматичний режим виконання програми задається параметром командного рядка -auto. Наприклад, java ClassName -auto.
* В автоматичному режимі діалог з користувачем відсутній, необхідні данні генеруються, або зчитуються з файлу.

1. Забороняється використання алгоритмів з Java Collections Framework.
   1. **Прикладна задача**

Бюро знайомств. Сортування за датою реєстрації, за кількістю властивостей в розділі “відомості про себе”, за кількістю властивостей в розділі “вимоги до партнера”.

**2. Опис програми**

Програма реалізована у вигляді інтерактивного консольного вікна з діалоговим режимом роботи з користувачем.

Основне призначення: демонстрація управління масивом domain-об’єктів. Реалізовано додавання та генерування нових об’єктів, видалення, показ інформації.

**2.1 Ієрархія та структура класів**

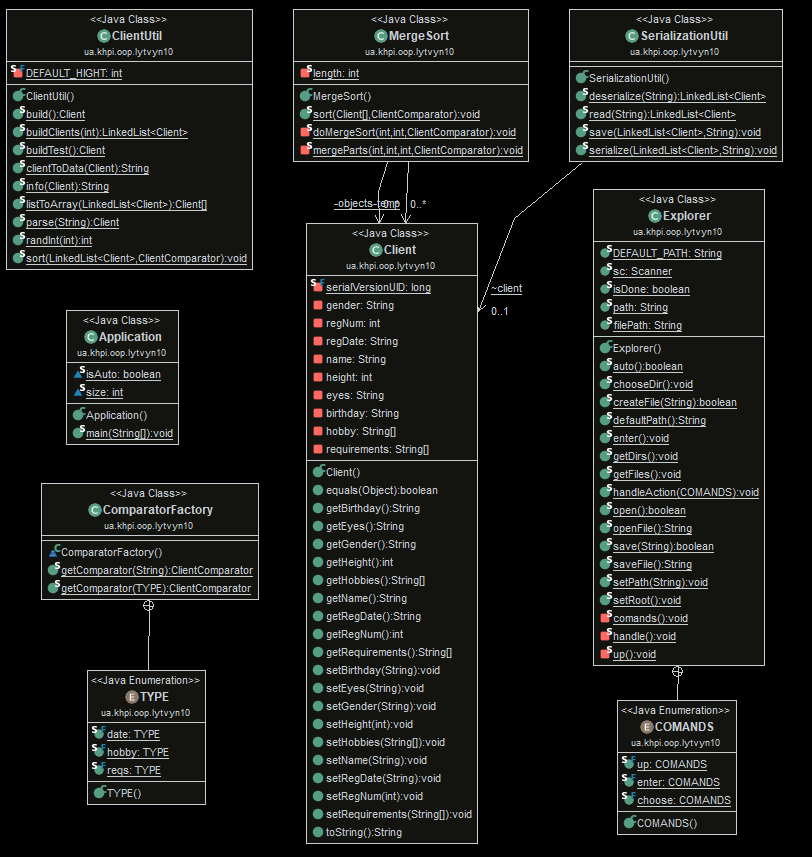
****

Рисунок 1 « Ієрархія та структура класів»

Рисунок 1 « Ієрархія та структура класів»

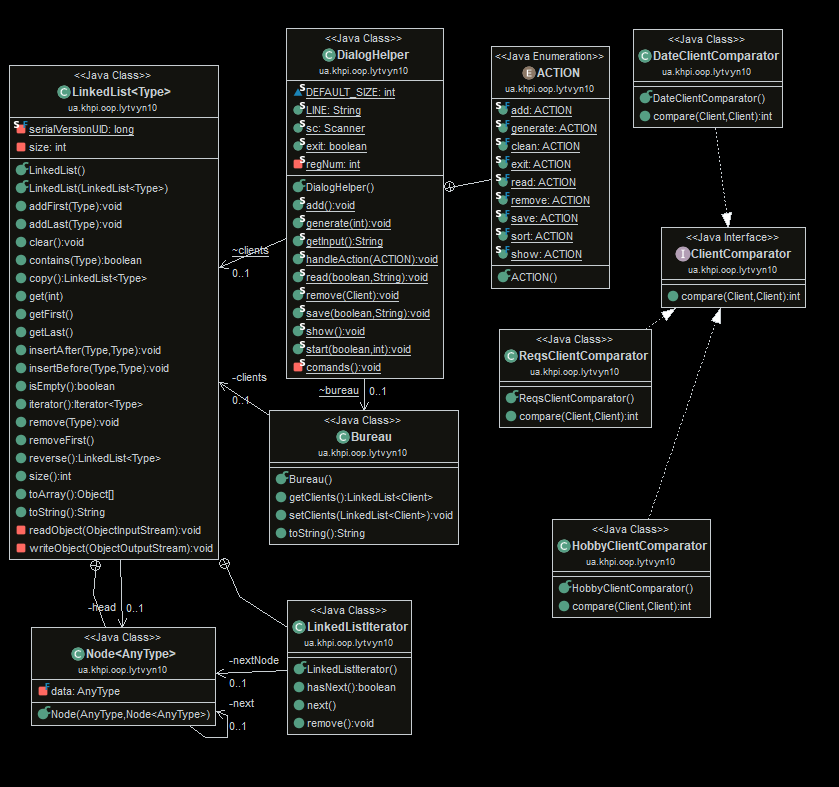


Рисунок 2 « Ієрархія та структура класів»

**2.2 Важливі фрагменти програми**

package ua.khpi.oop.lytvyn10;

import java.io.Serializable;

import java.util.Iterator;

import java.util.NoSuchElementException;

/\*\*

\* Список

\*

\* @author student Lytvyn I.I. KIT-26A

\* @param <Type>

\* тип зберігаємих елементів

\*/

public class LinkedList<Type> implements Iterable<Type>, Serializable {

/\*\*

\* Ітератор списку

\*

\* @author student Lytvyn I.I. KIT-26A

\*/

private class LinkedListIterator implements Iterator<Type> {

private Node<Type> nextNode;

public LinkedListIterator() {

nextNode = head;

}

@Override

public boolean hasNext() {

return nextNode != null;

}

@Override

public Type next() {

if (!hasNext()) {

throw new NoSuchElementException();

}

final Type res = nextNode.data;

nextNode = nextNode.next;

return res;

}

@Override

public void remove() {

throw new UnsupportedOperationException();

}

}

/\*\*

\* Уособлює ланку списку

\*

\* @param <AnyType>

\* тип елементу

\*

\* @author student Lytvyn I.I. KIT-26A

\*/

private static class Node<AnyType> {

private final AnyType data;

private Node<AnyType> next;

public Node(AnyType data, Node<AnyType> next) {

this.data = data;

this.next = next;

}

}

/\*\*

\* Унікальний ідентифікатор версії класу

\*/

private static final long serialVersionUID = 3842190203164935628L;

/\*\*

\* Початок списку

\*/

transient private Node<Type> head;

/\*\*

\* Розмір списку

\*/

transient private int size;

/\*\*

\* Constructs an empty list

\*/

public LinkedList() {

head = null;

size = 0;

}

/\*\*

\* Створює список з іншого списку

\*

\* @param list

\* список

\*/

@SuppressWarnings("unchecked")

public LinkedList(LinkedList<Type> list) {

head = null;

size = 0;

for (final Object x : list) {

addLast((Type) x);

}

}

/\*\*

\* Додавання елементу на початок списку

\*

\* @param item

\* елемент

\*/

public void addFirst(Type item) {

head = new Node<>(item, head);

size++;

}

/\*\*

\* Додавання елементу в кінець списку

\*

\* @param item

\* елемент

\*/

public void addLast(Type item) {

if (head == null) {

size--;

addFirst(item);

} else {

Node<Type> tmp = head;

while (tmp.next != null) {

tmp = tmp.next;

}

tmp.next = new Node<>(item, null);

}

size++;

}

/\*\*

\* Очищення списку

\*/

public void clear() {

head = null;

size = 0;

System.gc();

}

/\*\*

\* Перевіряє чи присутній елемент у списку

\*

\* @param x

\* елемент для перевірки

\* @return чи присутній елемент у списку

\*

\*/

public boolean contains(Type x) {

for (final Type tmp : this) {

if (tmp.equals(x)) {

return true;

}

}

return false;

}

/\*\*

\* Повертає копію списку

\*

\* @return копія списку

\*/

public LinkedList<Type> copy() {

final LinkedList<Type> twin = new LinkedList<>();

Node<Type> tmp = head;

if (head == null) {

return null;

}

while (tmp != null) {

twin.addFirst(tmp.data);

tmp = tmp.next;

}

return twin.reverse();

}

/\*\*

\* Повертає елемент, що знаходиться на заданій позиції

\*

\* @param pos

\* позиція елементу

\* @return елемент

\*/

public Type get(int pos) {

if (head == null) {

throw new IndexOutOfBoundsException();

}

Node<Type> tmp = head;

for (int k = 0; k < pos; k++) {

tmp = tmp.next;

}

if (tmp == null) {

throw new IndexOutOfBoundsException();

}

return tmp.data;

}

/\*\*

\* Повертає перший елемент у списку

\*

\* @return перший елемент списку

\*/

public Type getFirst() {

if (head == null) {

throw new NoSuchElementException();

}

return head.data;

}

/\*\*

\* Повертає останній елемент у списку

\*

\* @return останній елемент списку

\*/

public Type getLast() {

if (head == null) {

throw new NoSuchElementException();

}

Node<Type> tmp = head;

while (tmp.next != null) {

tmp = tmp.next;

}

return tmp.data;

}

/\*\*

\* Вставляє елемент після заданого

\*

\* @param key

\* елемент після якого необхідно вставити

\* @param toInsert

\* елемент для вставляння

\*/

public void insertAfter(Type key, Type toInsert) {

Node<Type> tmp = head;

while (tmp != null && !tmp.data.equals(key)) {

tmp = tmp.next;

}

if (tmp != null) {

tmp.next = new Node<>(toInsert, tmp.next);

}

size++;

}

/\*\*

\* Вставляє елемент перед заданим

\*

\* @param key

\* елемент перед яким необхідно вставити

\* @param toInsert

\* елемент для вставляння

\*/

public void insertBefore(Type key, Type toInsert) {

if (head == null) {

return;

}

if (head.data.equals(key)) {

addFirst(toInsert);

return;

}

Node<Type> prev = null;

Node<Type> cur = head;

while (cur != null && !cur.data.equals(key)) {

prev = cur;

cur = cur.next;

}

if (cur != null) {

prev.next = new Node<>(toInsert, cur);

}

size++;

}

/\*\*

\* @return чи порожній список

\*/

public boolean isEmpty() {

return head == null && size == 0;

}

/\*\*

\* Ітератор

\*/

@Override

public Iterator<Type> iterator() {

return new LinkedListIterator();

}

/\*\*

\* Видаляє перший знайдений елемент

\*

\* @param key

\* ключ для пошуку

\*/

public void remove(Type key) {

if (head == null) {

throw new RuntimeException("cannot delete");

}

if (head.data.equals(key)) {

head = head.next;

return;

}

Node<Type> cur = head;

Node<Type> prev = null;

while (cur != null && !cur.data.equals(key)) {

prev = cur;

cur = cur.next;

}

if (cur == null) {

throw new RuntimeException("cannot delete");

}

prev.next = cur.next;

size--;

}

/\*\*

\* Видалення перншого елементу

\*

\* @return видалений елемент

\*/

public Type removeFirst() {

final Type tmp = getFirst();

head = head.next;

return tmp;

}

/\*\*

\* Змінює порядок розташування елементів у списку на обернений

\*

\* @return обернений список

\*/

public LinkedList<Type> reverse() {

final LinkedList<Type> list = new LinkedList<>();

Node<Type> tmp = head;

while (tmp != null) {

list.addFirst(tmp.data);

tmp = tmp.next;

}

return list;

}

/\*\*

\* @return розмір списку

\*/

public int size() {

return this.size;

}

/\*\*

\* Конвертує список у масив обєктів

\*

\* @return масив обєктів

\*/

public Object[] toArray() {

final Object[] copy = new Object[this.size];

int i = 0;

for (final Object object : this) {

copy[i++] = object;

}

return copy;

}

/\*\*

\* Returns a string representation

\*

\*/

@Override

public String toString() {

final StringBuffer result = new StringBuffer();

for (final Object x : this) {

result.append(x + " ");

}

return result.toString();

}

/\*\*

\* Відновлює екземпляр <tt>LinkedList</tt> з потоку (тобто десеріалізує

\* його).

\*

\* @param inStream

\* потік, з якого відновлюється екземпляр <tt>LinkedList</tt>

\* @throws java.io.IOException

\* виключна ситуація при введені/виведені

\* @throws ClassNotFoundException

\* виключна ситуація при відсутності необхідного класу

\*/

@SuppressWarnings("unchecked")

private void readObject(java.io.ObjectInputStream inStream)

throws java.io.IOException, ClassNotFoundException {

/\* Зчитування розміру та інших прихованих речей \*/

inStream.defaultReadObject();

/\* Зчитування довжинb масиву \*/

final int size = inStream.readInt();

/\* Initialize header \*/

head = new Node<>(null, null);

/\* Зчитування всіх елементів у належному порядку \*/

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

addLast((Type) inStream.readObject());

}

}

/\*\*

\* Зберігає екземпляр <tt>LinkedList</tt> в потоці (тобто серіалізує його).

\*

\* @param outStream

\* потік, в який записується екземпляр <tt>LinkedList</tt>

\* @throws java.io.IOException

\* виключна ситуація при введені/виведені

\*/

private void writeObject(java.io.ObjectOutputStream outStream)

throws java.io.IOException {

/\* Запис кількості елементів та інших прихованих речей \*/

outStream.defaultWriteObject();

/\* Запис довжини масиву \*/

outStream.writeInt(size);

/\* Запис всіх елементів у належному порядку. \*/

Node<Type> tmp = head;

while (tmp != null) {

outStream.writeObject(tmp.data);

tmp = tmp.next;

}

}

}

1. **РЕЗУЛЬТАТ РОБОТИ**

Для налагодження роботи програми було успішно проведено її тестування.

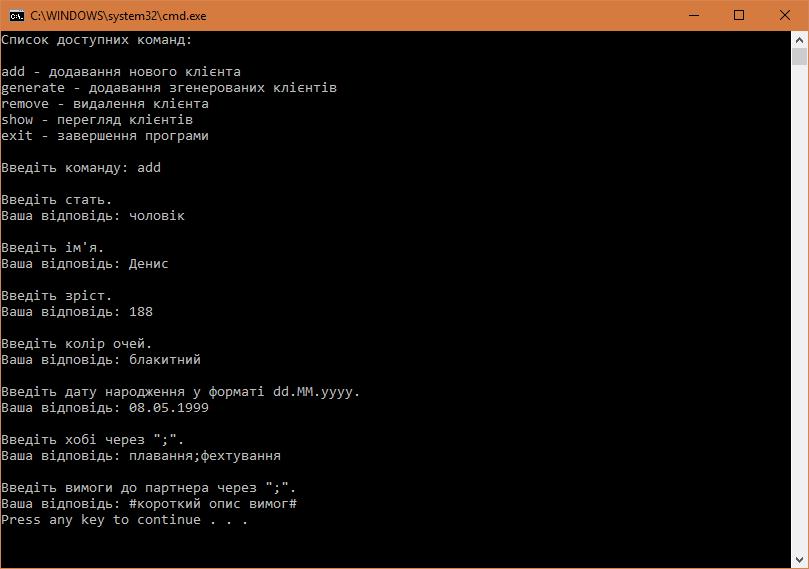
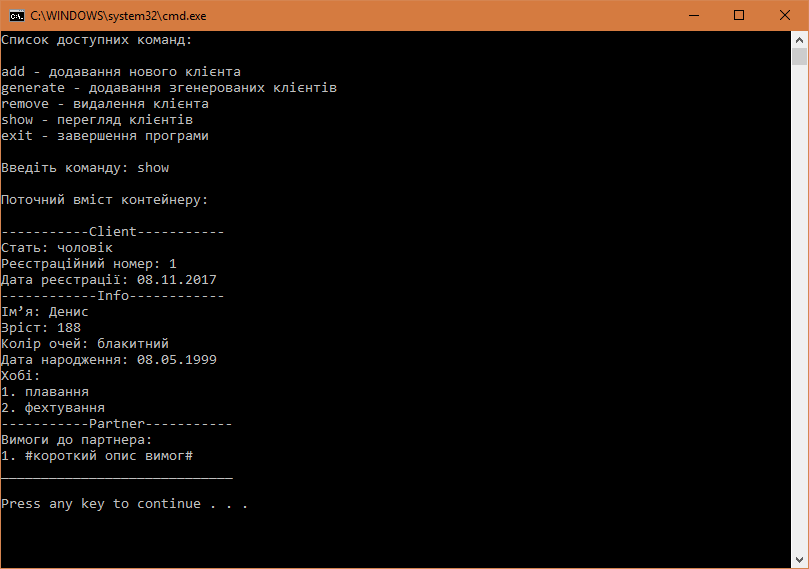
**

Рисунок 3 "Результати"

Рисунок 4 "Результати"

****ВИСНОВКИ**

*Створено і налагоджено програму, що повністю виконую поставлене індивідуальне завдання та відповідає вимогам.*

*Було отримано і вдосконалено навички у використанні об’єктно-орієнтованого підходу для розробки об’єкта предметної (прикладної) галузі.*