|  |
| --- |
|  |
| МИНОБРНАУКИ РОССИИ |
| *Федеральное государственное бюджетное образовательное учреждение высшего образования* ***«МИРЭА – Российский технологический университет»***  **РТУ МИРЭА** |

**Институт информационных технологий (ИТ)**

**Кафедра инструментального и прикладного программного обеспечения (ИиППО)**

**Дисциплина «Программирование на языке Джава»**

**ОТЧЕТ**

**ПО ПРАКТИЧЕСКИМ ЗАНЯТИЯМ №7, №8**

Выполнил студент группы ИНБО-02-20 Колмаков Е.Ю.

Принял Степанов П.В.

Практические работы выполнены «\_\_\_»\_\_\_\_\_\_\_2021г.

«\_\_\_\_\_\_\_\_\_\_\_\_\_\_» «\_\_\_»\_\_\_\_\_\_\_2021г.

Отметка о выполнении

**Москва 2021 г.**

**Практическое занятие №7**

**Цель работы:** изучение на практике приемов работы со стандартными контейнерными классами Java Collection Framework.

**ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Java Collections Framework — это набор связанных классов и интерфейсов, реализующих широко используемые структуры данных — коллекции. На вершине иерархии в Java Collection Framework располагаются 2 интерфейса: Collection и Map. Эти интерфейсы разделяют все коллекции, входящие в фреймворк на две части по типу хранения данных: простые последовательные наборы элементов и наборы пар «ключ — значение» (словари).

Рисунок 7.1

Vector — реализация динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента. Vector появился в JDK версии Java 1.0, но, как и Hashtable, эту коллекцию не рекомендуется использовать, если не требуется достижения потокобезопасности. Потому как в Vector, в отличии от других реализаций List, все операции с данными являются синхронизированными. В качестве альтернативы часто применяется аналог — ArrayList.

Stack — данная коллекция является расширением коллекции Vector. Была добавлена в Java 1.0 как реализация стека LIFO (last-in-first-out). Является частично синхронизированной коллекцией (кроме метода добавления push()). После добавления в Java 1.6 интерфейса Deque, рекомендуется использовать именно реализации этого интерфейса, например ArrayDeque.

ArrayList — как и Vector является реализацией динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента. Как можно догадаться из названия, его реализация основана на обычном массиве. Данную реализацию следует применять, если в процессе работы с коллекцией предполагается частое обращение к элементам по индексу. Из-за особенностей реализации обращение к элементам по индексу, которое выполняется за константное время O(1). Использование данной коллекции рекомендуется избегать, если требуется частое удаление/добавление элементов в середине коллекции.

LinkedList — вид реализации List. Позволяет хранить любые данные, включая null. Данная коллекция реализована на основе двунаправленного связного списка (каждый элемент списка имеет ссылки на предыдущий и следующий). Добавление и удаление элемента из середины, доступ по индексу, значению происходит за линейное время O(n), а из начала и конца за константное время O(1). Ввиду реализации, данную коллекцию можно использовать как абстрактный тип данных — стек или очередь. Для этого в ней реализованы соответствующие методы.

Интерфейс Set.

Представляет собой неупорядоченную коллекцию, которая не может содержать одинаковые элементы и является программной моделью математического понятия «множество».

Интерфейс Queue.

Этот интерфейс описывает коллекции с предопределённым способом вставки и извлечения элементов, а именно — очереди FIFO (first-in-first-out). Помимо методов, определённых в интерфейсе Collection, определяет дополнительные методы для извлечения и добавления элементов в очередь.

Рисунок 7.2

PriorityQueue — является единственной прямой реализацией интерфейса Queue (была добавлена, как и интерфейс Queue, в Java 1.5), не считая класса LinkedList, который так же реализует этот интерфейс, но был реализован намного раньше. Особенностью данной очереди является возможность управления порядком элементов. По умолчанию, элементы сортируются с использованием «natural ordering», но это поведение может быть переопределено при помощи объекта Comparator, который задаётся при создании очереди. Данная коллекция не поддерживает null в качестве элементов.

ArrayDeque — реализация интерфейса Deque, который расширяет интерфейс Queue методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out). Интерфейс Deque и реализация ArrayDeque были добавлены в Java 1.6. Эта коллекция представляет собой реализацию с использованием массивов, подобно ArrayList, но не позволяет обращаться к элементам по индексу и хранение null.

Как заявлено в документации, эта коллекция работает быстрее чем Stack, если используется как LIFO коллекция, а также быстрее чем LinkedList, если используется как FIFO коллекция.

**ЗАДАНИЯ.**

Напишите программу в виде консольного приложения, которая моделирует карточную игру «пьяница» и определяет, кто выигрывает. В игре участвует 10 карт, имеющих значения от 0 до 9, большая карта побеждает меньшую; карта «0» побеждает карту «9».

Карточная игра “ В пьяницу”. В этой игре карточная колода раздается поровну двум игрокам. Далее они открывают по одной верхней карте, и тот, чья карта старше, забирает себе обе открытые карты, которые кладутся под низ его колоды. Тот, кто остается без карт, - проигрывает.

Для простоты будем считать, что все карты различны по номиналу и что самая младшая карта побеждает самую старшую карту (“шестерка берет туз”).

Игрок, который забирает себе карты, сначала кладет под низ своей колоды карту первого игрока, затем карту второго игрока (то есть карта второго игрока оказывается внизу колоды).

**Входные данные.**

Программа получает на вход две строки: первая строка содержит 5 карт первого игрока, вторая - 5 карт второго игрока. Карты перечислены сверху вниз, то есть каждая строка начинается с той карты, которая будет открыта первой.

**Выходные данные.**

Программа должна определить, кто выигрывает при данной раздаче, и вывести слово first или second, после чего вывести количество ходов, сделанных до выигрыша. Если на протяжении 106 ходов игра не заканчивается, программа должна вывести слово botva.

**Пример ввода.**

13579 24680

second 5

**Упражнение 1.**

Используйте для организации хранения структуру данных Stack.

**Упражнение 2.**

Используйте для организации хранения структуру данных Queue.

**Упражнение 3.**

Используйте для организации хранения структуру данных Dequeue.

**Упражнение 3.**

Используйте для организации хранения структуру данных DoubleList.

**Упражнение 4\*.**

Реализуйте более усложненный вариант решения задачи из упражнения 1. Реализация должна иметь интерактивный интерфейс для взаимодействия с пользователем.

**Реализация:**

**Класс TestLab7ex1:**

package Lab7;  
  
import java.util.Stack;  
  
public class TestLab7ex1 {  
 Stack<Integer> FirstPlayer, SecondPlayer;  
  
 public TestLab7ex1(String fp, String sp) {  
 this.FirstPlayer = new Stack<>();  
 this.SecondPlayer = new Stack<>();  
 for(int i = 4; i >= 0; i --){  
 this.FirstPlayer.push(Integer.*parseInt*(fp.substring(i, i+1)));  
 this.SecondPlayer.push(Integer.*parseInt*(sp.substring(i, i+1)));  
 }  
 }  
  
 private void pushBack(Stack<Integer> s, Integer item){  
 Stack<Integer> abc = new Stack<>();  
 while (!s.isEmpty())  
 abc.push(s.pop());  
 s.push(item);  
 while (!abc.isEmpty())  
 s.push(abc.pop());  
 }  
  
 public String play(){  
 int count = 0;  
 while(!FirstPlayer.isEmpty() && !SecondPlayer.isEmpty() && count <= 106){  
 if(FirstPlayer.peek() > SecondPlayer.peek()  
 && SecondPlayer.peek() != 0){  
 pushBack(FirstPlayer, FirstPlayer.pop());  
 pushBack(FirstPlayer, SecondPlayer.pop());  
 }else{  
 pushBack(SecondPlayer, FirstPlayer.pop());  
 pushBack(SecondPlayer, SecondPlayer.pop());  
 }  
 count++;  
 }  
 String res = "";  
 if(FirstPlayer.isEmpty()) res += "second ";  
 else if (SecondPlayer.isEmpty()) res += "first ";  
 res += count;  
 if(count >= 106) res += " botva";  
 return res;  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println(new TestLab7ex1("23569", "14780").play());  
 System.*out*.println(new TestLab7ex1("72345", "61890").play());  
 System.*out*.println(new TestLab7ex1("12345", "67890").play());  
 }  
  
}

**Класс TestLab7ex2:**

package Lab7;  
  
import java.util.ArrayDeque;  
import java.util.Queue;  
  
public class TestLab7ex2 {  
 Queue<Integer> FirstPlayer, SecondPlayer;  
  
 public TestLab7ex2(String fp, String sp) {  
 this.FirstPlayer = new ArrayDeque<>();  
 this.SecondPlayer = new ArrayDeque<>();  
 for(int i = 0; i < 5; i ++){  
 this.FirstPlayer.add(Integer.*parseInt*(fp.substring(i, i+1)));  
 this.SecondPlayer.add(Integer.*parseInt*(sp.substring(i, i+1)));  
 }  
 }  
  
 public String play(){  
 int count = 0;  
 while(!FirstPlayer.isEmpty() && !SecondPlayer.isEmpty() && count < 106){  
 if(FirstPlayer.peek() > SecondPlayer.peek()  
 && SecondPlayer.peek() != 0){  
 FirstPlayer.add(FirstPlayer.remove());  
 FirstPlayer.add(SecondPlayer.remove());  
  
 }else{  
 SecondPlayer.add(FirstPlayer.remove());  
 SecondPlayer.add(SecondPlayer.remove());  
 }  
 count++;  
 }  
 String res = "";  
 if(FirstPlayer.isEmpty()) res += "second ";  
 else if (SecondPlayer.isEmpty()) res += "first ";  
 res += count;  
 if(count >= 106) res += " botva";  
 return res;  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println(new TestLab7ex2("23569", "14780").play());  
 System.*out*.println(new TestLab7ex2("72345", "61890").play());  
 System.*out*.println(new TestLab7ex2("12345", "67890").play());  
 }  
  
}

**Класс TestLab7ex3:**

package Lab7;  
  
import java.util.ArrayDeque;  
import java.util.Deque;  
  
public class TestLab7ex3 {  
 Deque<Integer> FirstPlayer, SecondPlayer;  
  
 public TestLab7ex3(String fp, String sp) {  
 this.FirstPlayer = new ArrayDeque<>();  
 this.SecondPlayer = new ArrayDeque<>();  
 for(int i = 0; i < 5; i ++){  
 this.FirstPlayer.add(Integer.*parseInt*(fp.substring(i, i+1)));  
 this.SecondPlayer.add(Integer.*parseInt*(sp.substring(i, i+1)));  
 }  
 }  
  
 public String play(){  
 int count = 0;  
 while(!FirstPlayer.isEmpty() && !SecondPlayer.isEmpty() && count < 106){  
 if(FirstPlayer.peek() > SecondPlayer.peek()  
 && SecondPlayer.peek() != 0){  
 FirstPlayer.add(FirstPlayer.remove());  
 FirstPlayer.add(SecondPlayer.remove());  
  
 }else{  
 SecondPlayer.add(FirstPlayer.remove());  
 SecondPlayer.add(SecondPlayer.remove());  
 }  
 count++;  
 }  
 String res = "";  
 if(FirstPlayer.isEmpty()) res += "second ";  
 else if (SecondPlayer.isEmpty()) res += "first ";  
 res += count;  
 if(count >= 106) res += " botva";  
 return res;  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println(new TestLab7ex3("83469", "17520").play());  
 System.*out*.println(new TestLab7ex3("72345", "61890").play());  
 System.*out*.println(new TestLab7ex3("12345", "67890").play());  
 }  
  
}

**Класс TestLab7ex4:**

package Lab7;  
  
import java.util.LinkedList;  
  
public class TestLab7ex4 {  
 LinkedList<Integer> FirstPlayer, SecondPlayer;  
  
 public TestLab7ex4(String fp, String sp) {  
 this.FirstPlayer = new LinkedList<>();  
 this.SecondPlayer = new LinkedList<>();  
 for (int i = 0; i < 5; i++) {  
 this.FirstPlayer.add(Integer.*parseInt*(fp.substring(i, i + 1)));  
 this.SecondPlayer.add(Integer.*parseInt*(sp.substring(i, i + 1)));  
 }  
 }  
  
  
 public String play() {  
 int count = 0;  
 while (!FirstPlayer.isEmpty() && !SecondPlayer.isEmpty() && count <= 106) {  
 if (FirstPlayer.peek() <= SecondPlayer.peek()  
 || SecondPlayer.peek() == 0) {  
 SecondPlayer.add(FirstPlayer.remove());  
 SecondPlayer.add(SecondPlayer.remove());  
 } else {  
 FirstPlayer.add(FirstPlayer.remove());  
 FirstPlayer.add(SecondPlayer.remove());  
  
 }  
 count++;  
 }  
 String res = "";  
 if(FirstPlayer.isEmpty()) res += "second ";  
 else if (SecondPlayer.isEmpty()) res += "first ";  
 res += count;  
 if(count >= 106) res += " botva";  
 return res;  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println(new TestLab7ex4("83469", "17520").play());  
 System.*out*.println(new TestLab7ex4("72345", "61890").play());  
 System.*out*.println(new TestLab7ex4("12345", "67890").play());  
  
 }  
}

**Вывод:** В ходе выполнения данной работы мы изучили на практике приемы работы со стандартными контейнерными классами Java Collection Framework.

**Практическое занятие №8**

**ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

**Различные виды списков ожидания**

Вначале, рассмотрите класс, представленный на диаграмме - общий класс список ожидания. Допустим мы решили, что нам понадобиться еще два вида списков ожидания:

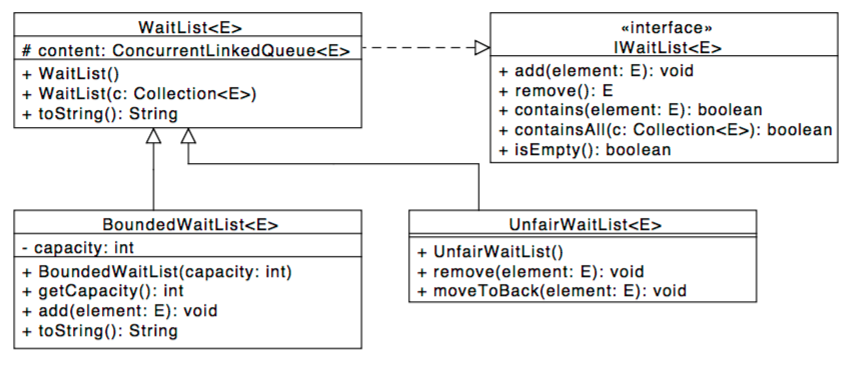
• BoundedWaitList: Этот список ожидания имеет ограниченную емкость, указываемую в момент создания. Он не принимает более элементов, чем заранее задано (возможное количество потенциальных элементов в списке ожидания).

• UnfairWaitList: В этом списке ожидания, можно удалить элемент, который не является первым в очереди - и помните он не может вернуться обратно! (Возможны различные реализации, но в вашей реализации необходимо удалить первое вхождение данного элемента.) Также возможно, чтобы например, первый элемент будет отправлен обратно в конец списка.

После описания всей задачи в целом, мы сможем решить, что мы нам нужен интерфейс IWaitList, и затем нужно создать три разных класса для трех списков ожидания. Также предполагается, что один из списков ожидания должен быть супер классом для двух других списков ожидания.

Задание

1. Исследуйте UML диаграмму классов на рисунке 1 и понаблюдайте, как она выражает то, что мы говорили выше в словах. Убедитесь, что вы понимаете все аспекты диаграммы.
2. Расширить и модифицировать исходный код WaitList, как необходимо, чтобы полностью реализовать всю схему UML. Включить комментарии Javadoc. Обратите внимание на переключение ролей после реализации каждого интерфейса / класса!
3. Изучение работу метода main(), которая использует ваши новые классы и интерфейс.



# Реализация:

# Класс WaitList:

package Lab8;  
  
import java.util.Collection;  
import java.util.concurrent.ConcurrentLinkedQueue;  
  
public class WaitList<E> implements IWaitList<E> {  
 protected ConcurrentLinkedQueue<E> content;  
  
 public WaitList() {  
 content = new ConcurrentLinkedQueue<>();  
 }  
  
 public WaitList(ConcurrentLinkedQueue<E> content) {  
 this.content = content;  
 }  
  
 @Override  
 public String toString() {  
 return "WaitList{" +  
 "content=" + content +  
 '}';  
 }  
  
 @Override  
 public void add(E element) {  
 content.add(element);  
 }  
  
 @Override  
 public E remove() {  
 return content.remove();  
 }  
  
 @Override  
 public boolean contains(E element) {  
 return content.contains(element);  
 }  
  
 @Override  
 public boolean containsAll(Collection<E> c) {  
 return content.containsAll(c);  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return content.isEmpty();  
 }  
}

# Класс UnfairWaitList:

package Lab8;  
  
public class UnfairWaitList<E> extends WaitList<E> {  
 public UnfairWaitList() {  
 }  
  
 void remove(E element) {  
 content.remove(element);  
 }  
  
 void moveToBack(E element) {  
 content.remove(element);  
 content.add(element);  
 }  
}

# Класс BoundedWaitList:

package Lab8;  
  
public class WBoundedWaitList<E> extends WaitList<E> {  
 public int capacity;  
  
 public BoundedWaitList(int capacity) {  
 this.capacity = capacity;  
 }  
  
 public int getCapacity() {  
 return capacity;  
 }  
  
 public void add(E element) {  
 if (content.size() < capacity) {  
 content.add(element);  
 }  
 }  
}

# Класс IWaitList:

package Lab8;  
  
import java.util.Collection;  
  
public interface IWaitList<E> {  
 void add(E element);  
 E remove();  
 boolean contains(E element);  
 boolean containsAll(Collection<E> c);  
 boolean isEmpty();  
}

# Класс TestLab8:

package Lab8;  
  
public class TestLab8 {  
 public static void main(String[] args) {  
 WaitList<Integer> list1 = new WaitList<>();  
 for (int i = 0; i <= 5; i++) {  
 list1.add(i);  
 }  
 System.*out*.println(list1);  
 System.*out*.println(list1.remove());  
 System.*out*.println(list1.remove());  
 int i = list1.remove();  
 UnfairWaitList<Integer> list2 = new UnfairWaitList<Integer>();  
 list2.add(1);  
 list2.add(2);  
 list2.add(3);  
 System.*out*.println(list2);  
 list2.moveToBack(2);  
 System.*out*.println(list2);  
  
  
 }  
}

# Вывод: В ходе выполнения данной работы мы научились использовать списки ожидания.

# Git: https://github.com/efim1111/KolmakovJavaINBO-02-20