250522JFC P2

<<Iterator>>

next()

hasNext()

этот IF позволил реализовать forEach с 5.0 JDK

Level 1

задает общее поведение для всех коллекций

IF<<Collections>>

size()

isEmpty()

clear()

add()

remove()

…

Стоят в стороне от других контейнеров, тк характеризируются принципиально другим способом хранения данных и 🡪иной реализацией управления элементами (введена доп сущность entry с полями key-value)

IF<<ListIterator>>

Level 2

Cпециализации поведения Collections

IF<<Map>> хранят эл-ты , как ключ- знач-е, ключ-уникален, любого типа данных

IF<<Queue>>

Используется в многопоточности, алгоритм обработки данных(FIFO – first input-first output)

IF<<Set>> уникальность объектов

IF<<List>> инд.доступ

get(index)

set(index, E)

… 90% использов-я

Set реализованы на базе map, но используются только ключи

IF<<SortedMap>>

<<Deque>>

LIFO – last input-first output

IF<<SortedSet>>

Упорядоченные уникальные объекты

Существуют устаревшие варианты этого контейнера hashtag и его разновидность dictionary (словарь)все ключи -уникальны

Level 3

Абстрактные классы

Level 4

1. Реализации 20% (Implementations)

**ArrayList (85%)==Vector<-Stack (динамические массивы,см. DynamicBasket)**

**LinkedList (15%)Двухсвязный список**

**HashSet** ассоциативный массив (с мат т зр – хэш таблица)

**LinkedHashSet,** реализованный на основе односвязного списка

**TreeSet,** реализован на основе TreeMap,реализованного в свою очередь как бинарное (черно-красное дерево) по деф - нативная сортировка

\*Стоит отметить, что дефолтная реализация контенеров типа Hash подходит только для примитивных типов, для пользовательских классов требуется обязательное переопределение логики сравнения и добавления объектов

Level 5

В соответствии c Java Bean сущность должна иметь:

-инкапсуляцию полей🡪setters&getters

-конструкторы: как минимум деф и с инициализацией все состояние

-переопределенные методы hashCode, equals,toString

В абстракции нашего проекта (Product) остались не реализованы hashCode и equals, переопределять их нужно совместно тк hashCode использует equals, в связи с тем, что хэшкод ограничен значением инта и может повторяться

\*смотри статьи на hub как грамотно преопределить hashCode и equals

Чтобы мы могли корректно добавлять наши продукты в контейнеры использующие хэш, нам необходимо правильно переопределить данные методы, не только у базового класса, но и у каждого наследника hashCode: для формирования гарантированно уникального значения объекта

Применяется формула

Int prime = 31; реком-й вар-т, тк \*31 = побитовуму сдвигу на 5, но можно и любое простое число

Int хэш = prime + Object.hash(поле1)

Хэш = хэш\* prime + Object.hash(поле2)

Хэш = хэш\* prime + Object.hash(поле3)

Return хэш

Equals(о)

Если (о==this) 🡪true проверка объекта на соответствие самосу себе (по ссылке)

Если(о==null)🡪false если на текущем объекте вызван данный метод,значит он существует🡪!=null

Если(getClass()==o.getClass())🡪false соответствие классов текущего объекта и переданного🡪можем привести тип переданного объекта к текущему и сравнивать поля

Obj obj = (Obj)o;

Если (поле1 != о.поле1)🡪false

Если (поле2 != о.поле2)🡪false

Иначе true

Для добавления наших сущностей в Treeset необходимо еще имплементить интерфейс comparable для того чб объект был сопоставим и смог быть частью последовательности, реализуем такую возможность в базовом классе указав при имплементации конкретный тип нашего продукта

public class Product implements Comparable<Product>{

для имплементации метода compareTo(Product o) используется следующий алгоритм:

compareTo(Product o)

если (поле1>o.поле1) 🡪 положительное число (любое)

иначе если (поле1<о.поле1)🡪 отрицательное число (любое)

иначе 🡪0

@Override  
public int compareTo(Product o) {  
 if (price > o.price) {  
 return +1;  
 } else if (price < o.price) {  
 return -1;  
 } else {  
 return 0;  
 }  
}

В производных классах

Проверяем равны ли типы производных если нет, делегируем базовому классу или возвращаем число !=0

Если равны приводим к типу данного объекта и сравниваем специализированные характерисктики поочередно

@Override  
public int compareTo(Product o) {  
 if (getClass() != o.getClass()) return super.compareTo(o);  
 Orange orange = (Orange) o;  
 if (diameter > orange.diameter) {  
 return 1;  
 } else if (diameter < orange.diameter) {  
 return -1;  
 } else return 0;  
  
}

Если поле одно, для числовых значений можно просто вернуть разницу между полем текущего объекта и передаваемого

@Override  
public int compareTo(Product o) {  
 if (getClass() != o.getClass()) return super.compareTo(o);  
 Orange orange = (Orange) o;  
 return diameter - orange.diameter;  
}

Для дабл

return Double.*compare*(o.price,price);

Для строки можно воспользоваться готовой реализацией

@Override  
public int compareTo(Product o) {  
 if (getClass() != o.getClass()) return super.compareTo(o);  
 Orange orange = (Orange) o;  
 return name.compareTo(orange.name);  
}

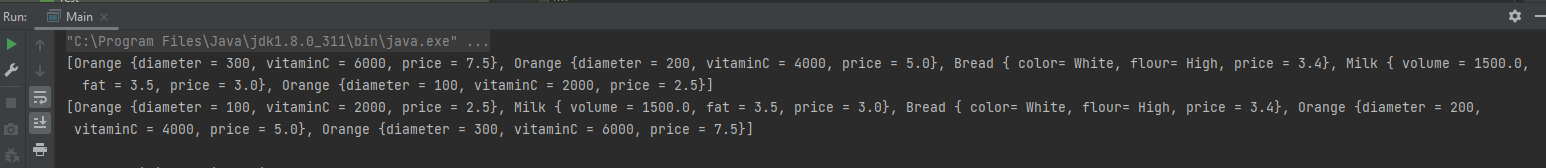
Для возможности расширения возможностей сортировки по другим параметрам в соответствии OCP возвращаемся к паттерну стратегия, реализованную в IF Comparator, уже заложенному в свойства Treeset и требующему только реализации самого класса, поддерживающую уникальность и сортировку.

В абстрактном классе оставлем реализацию сранения для сортировки по возрастанию, как дефолтную, а по убыванию реализуем в отдельном классе в директории comparator на уровне entity

package by.itStep.yandr.megaShopProject.model.entity.comparator;  
  
import by.itStep.yandr.megaShopProject.model.entity.abstracts.Product;  
  
import java.util.Comparator;  
  
public class SortByPriceDescending implements Comparator<Product> {  
  
 @Override  
 public int compare(Product o1, Product o2) {  
 return Double.*compare*(o2.getPrice(), o1.getPrice());  
 }  
}

Тестируем:

public static void main(String[] args) {  
 Orange orange1 = new Orange(100,2000,2.5);  
 Orange orange2 = new Orange(200,4000,5);  
 Orange orange3 = new Orange(300,6000,7.5);  
 Milk milk1 = new Milk(1000,3.5,2.5);  
 Milk milk2 = new Milk(1500,3.5,3);  
 Milk milk3 = new Milk(2000,3.5,4.2);  
 Milk milk4 = new Milk(4000,2.5,5);  
 Bread bread =new Bread("White","High",3.4);  
 Bread bread1 =new Bread("Black","Middle",2);

  
 Product[] products= {orange1, orange2,orange3,milk2,bread};  
 Set<Product> set = new TreeSet<>(new SortByPriceDescending());  
 Collections.*addAll*(set,products);  
 System.*out*.println(set);  
  
 set = new TreeSet<>();  
 Collections.*addAll*(set,products);  
 System.*out*.println(set);

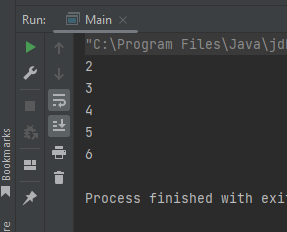
Реализации Queue:

Стандартная очередь, как правило реализовывается на основе LinkedList

Queue<Integer> queue= new LinkedList<>();

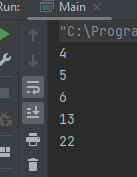
Большинство очередей строится на базе списков, тк способы управления идентичны, отличие в том что элементы удаляются после обработки, часть методов генерирует исключения, а часть null, основные методы :

Offer добавление элементов

Poll обращение, с последующим удалением

Queue<Integer> queue= new LinkedList<>();  
queue.offer(2);  
queue.offer(3);  
queue.offer(4);  
queue.offer(5);  
queue.offer(6);  
while (!queue.isEmpty()){  
 System.*out*.println(queue.poll());  
}

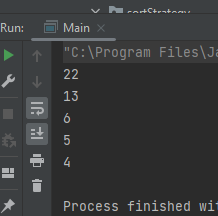
Существует понятие приоритета определяющее порядок очереди, реализуемый обычной сортировкой🡪на основе Comparator – очередь типа PriorityQueue, реализована на базе статичного массива

Queue<Integer> queue= new PriorityQueue<>();  
queue.offer(22);  
queue.offer(13);  
queue.offer(4);   
queue.offer(5);  
queue.offer(6);  
while (!queue.isEmpty()){  
 System.*out*.println(queue.poll());  
}

С реализацией по умолчанию, порядок очереди-нативный, для введения своего порядка, создаем свой компаратор и передаем в качестве параметра, при создании объекта

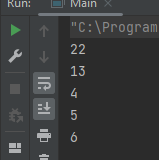
package by.itStep.yandr.megaShopProject.model.entity.comparator;  
import java.util.Comparator;  
public class SortByValueDesc implements Comparator<Integer> {  
 @Override  
 public int compare(Integer o1, Integer o2) {  
 return -Integer.*compare*(o1, o2);  
 }  
}

добавляем новый класс при создании очереди с приоритетом

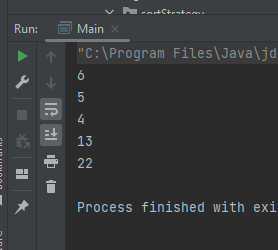
Queue<Integer> queue= new PriorityQueue<>(new SortByValueDesc());  
queue.offer(22);  
queue.offer(13);  
queue.offer(4);  
queue.offer(5);  
queue.offer(6);  
while (!queue.isEmpty()){  
 System.*out*.println(queue.poll());  
}

\*В пакете java.utilесть директория concurrent – небольшая библиотека для автоматизации работы с потоками, например пакет atomic содержит классы потокобезопасных оберток AtomicInteger, AtomicBoolean и пр…тж в пакете concurrent доступен интерфейс BlockingQueue, которая сама реализует блокировку(состоит из ограниченного количества объектов и если очередь заполнена, а поток пытается добавить новый объект, то поток блокируется до момента освобождения места в очереди, тж самое при попытке обратится к пустому объекту).

Тж существует реализация очереди, как ArrayDeque, в которой порядок сортировки определяется моментом поступления объекта

Queue<Integer> queue= new ArrayDeque<>();  
queue.offer(22);  
queue.offer(13);  
queue.offer(4);  
queue.offer(5);  
queue.offer(6);  
while (!queue.isEmpty()){  
 System.*out*.println(queue.poll());  
}

В объектах типа Deque, поменять порядок очереди можно просто используя соответствующие методы

Queue<Integer> queue= new ArrayDeque<>();  
Deque<Integer> deque= (Deque<Integer>)queue;  
deque.push(22);  
deque.push(13);  
deque.push(4);  
deque.push(5);  
deque.push(6);  
while (!queue.isEmpty()){  
 System.*out*.println(deque.pop());  
}