Вся библиотека коллекций делится на 3 части:

1. Интерфейсная часть 70% (Interfaces)

главный иф коллекции, предоставляющий способ перебора элементов

Iterable<E> (<<Iterable>>)

iterator()

<<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**

Имеет 3 конструктора:

- создание без указания размера, использует default capacity =10

Vector и Stack устаревший вариант коллекций, который сильно проигрывает по производительности, но является потокобезопасным (тк предоставляет доступ только одному потоку к общему ресурсу – kw - synchronized)

- с указанием размера

- прием готовых значений

Создание элементов:

- т.к AL дженерализирован, как и др коллекции, то при создании объекта без указания типа, создается список элементов типа Object, тж называемый “сырым” ,

ArrayList list = new ArrayList();

что приносит неудобства, тк требует ручной реализации типобезопасности

- по этой же причине создание коллекций объектов примитивного типа невозможно, тк ни один из них не наследуется от класса Object, для хранения примитивных значений используются классы-обертки

Назначение классов-оберток:

1. Доп функц-л, для работы с примитивными типами данных
2. Автоупаковка и автораспаковка
3. Создание коллекций, эмулирующих примитивные данные

ArrayList <Integer> list = new ArrayList();

- добавление объектов :

1. add() принимает значение или индекс и значение)

ArrayList <Integer> list = new ArrayList();  
list.add(1,2);

2. addAll() принимает список значений

List<Integer>temp = null;  
list.addAll(temp);

- Используя функционал класса Arrays

ArrayList<Integer> list = new ArrayList(Arrays.*asList*(2, 3, 4));

Или

ArrayList<Integer> list = new ArrayList();  
list.addAll(Arrays.*asList*(2, 3, 4));

Или

Integer[]ar ={2, 3, 4};  
list.addAll(Arrays.*asList*(ar));

*\*Стоит помнить что при передаче значений через переменную, создается ее поверхностная копия, а следовательно изменения будут отражатся и в исходной переменной*

- Используя функционал класса Collections:

ArrayList<Integer> list = new ArrayList();  
Collections.*addAll*(list,2, 3, 4);

**Generic methods - шаблонные методы:**

Методы обычного класса, принимающие значение любого типа, плейсхолд которого указывается в даймонд операторе перед возвращаемым типом

public <T> String sum(T a, T b){

return a+"+\n"+b;}

типов мб несколько, тж можно шаблонизировать тип возвращаемого значения

public <T, K> T sum(T a, K b){return a;}

Основное ограничение, что такие методы могут использовать только унифицированные для объектов всех типов операции. Расширение функционала реализовывается спецификацией типов

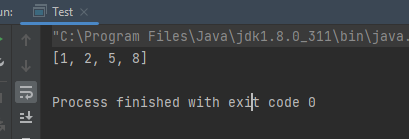
public <T extends Number> int sum(T a,T b){  
 return a.intValue()+b.intValue();}

Любая коллекция мб преобразована в массив

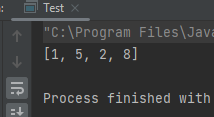
|  |  |
| --- | --- |
| **Hash** | **Elements** |
| **10258** | **2** |
| **326589** | **32 5** |
| **11857** | **85** |

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

Элементы добавляются по hash, поддерживая одновременно принципы и эквивалентности и уникальности, мы можем добавить под одним хэшем несколько разных объектов, или одинаковые объекты под разные хэши. По умолчанию, Хэш принимает значение рандомной подстановкой числа, которое рекомендуется переопределять исходя из логики проекта. Итерирование объектов производится тж по hash.

Добавление элементов будет происходить в условном порядке, (+устраняются эквив-е**)**

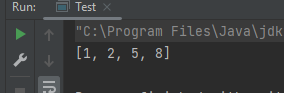
Set<Integer> set = new HashSet<>();  
Integer[] ar = {1, 5, 2, 8, 2, 1};  
Collections.*addAll*(set,ar);   
System.*out*.println(set);

Для сохранения исходного порядка исп-ся:

**LinkedHashSet,** реализованный на основе односвязного списка

Set<Integer> set = new LinkedHashSet<>();  
Integer[] ar = {1, 5, 2, 8, 2, 1};  
Collections.*addAll*(set,ar);  
System.*out*.println(set);

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

Set<Integer> set = new TreeSet<>();  
Integer[] ar = {1, 5, 2, 8, 2, 1};  
Collections.*addAll*(set,ar);  
System.*out*.println(set);

**1 ,12 ,4 ,7 ,23 ,4 ,5 ,46 ,7 , 8, 2**

**Первый элемент добавляется по умолчанию, а каждый следующий сравнивается с предидущими, если >-вправо, <-влево, =-игнорируется**

**1, 12>1->**

**4>1но<12**

7>4но<12

23>12….. итд

Обходы:

Левосторонний возрастание

Правосторонний убывание

В памяти реализуется как список

1

null

23

12

🡪поиск и добавление элементов logN

\*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