[java.util.Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html" \t "_blank)

## Интерфейс java.util.Collection имеет абстрактный класс AbstractCollection, то есть некоторая "абстрактная коллекция", которая представляет собой скелет для остальных реализаций (о чём написано в JavaDoc над классом java.util.AbstractCollection).

Списки (List)

списки реализуют интерфейс [java.util.List](https://docs.oracle.com/javase/8/docs/api/java/util/List.html).

List знает про индекс элемента. Это позволяет получить (get) элемент по индексу или задать значением для определённого индекса (set). Методы коллекций add, addAll, remove позволяют указать индекс, с которого необходимо их выполнять. Кроме того, у List есть своя версия итератора, которая называется ListIterator. Этот итератор знает про индекс элемента, поэтому он умеет итерироваться не только вперёд, но и назад. Его даже можно создать от определённого места в коллекции.

Две самые часто используемые: ArrayList и LinkedList. Во-первых, ArrayList — это список (List) на основе массива (Array). Это позволяет добиться "Произвольного доступа" ([Random Access](https://en.wikipedia.org/wiki/Random_access" \t "_blank)) к элементам. Произвольный доступ — это возможность сразу достать элемент по индексу, а не перебирать все элементы, пока не найдём элемент с нужным индексом. Именно массив как основа позволяет этого достичь.

LinkedList — это связанный (Linked) список (List). Каждая запись в связанном списке представлена в виде Entry, которая хранит сами данные, а так же ссылку на следующую (next) и предыдущую (previous) Entry. Таким образом LinkedList реализует "Последовательный доступ" ([Sequential Access](https://en.wikipedia.org/wiki/Sequential_access" \t "_blank)). Понятно, что чтобы найти 5-тый элемент нам придётся пройти от первого элемента до последнего.

Когда заканчивается место в массиве, то он увеличивается в 1.5 раза. Вот код увеличения: int newCapacity = oldCapacity + (oldCapacity >> 1); Другим отличием в работе является любое смещение элементов. Например, при добавлении в середину или удалении элементов. Чтобы удалить из LinkedList элемент достаточно убрать ссылки на этот элемент. В случае же с ArrayList мы вынуждены сдвигать каждый раз элементы при помощи метода System.arraycopy. Таким образом, чем больше элементов, тем больше действий придётся совершить.

Стэк — это интересная структура, которая является LIFO структурой last-in-first-out (последним пришёл, первым ушёл). Стэк в переводе с английского — стопка (как стопка книг, например). Стэк реализует дополнительные методы: peek (взглянуть, посмотреть), pop (вытолкнуть), push (затолкать).

Метод peek Данный метод позволяет посмотреть "на вершину" стэка, т.е. получить последний элемент не снимая (т.е. не удаляя) его из стэка

Метод push заталкивает (добавляет) в стэк новый элемент и возвращает его же, а метод pop элемент выталкивает (удаляет) и возвращает удалённый. Во всех трёх случаях (т.е. peek, pop и push) мы работаем только с последним элементом (т.е. с "вершиной стэка"). В этом основная особенность структуры стэк.

ArrayList

ArrayList<Object> arr\_list = new ArrayList<>();

ArrayList arr\_list = new ArrayLis(); //1 и 2 варианты одинаковые, но лучше 1 вариант использовать

ArrayList<String > arr\_list = new ArrayList<>(); // определенный тип

List< String > list = new ArrayList<>();// можно так, list это интерфейс, методы в list одинаковые, но в ArrayList and LinkedList methods could be created differently, but anyway they exist.

## Common ArrayList methods

stringList.add("Item");

indexOf("Item")

set(2, s2);- replacement

stringList.size()

isEmpty()

contains()

linkedList.set(2, s2); (вставка элемента с заменой)

Метод remove (int index) и метод remove(Object o)

Поскольку ArrayList допускает дублирование, стоит отметить, что remove(Object o) удаляет первое вхождение указанного элемента из этого списка, если он присутствует

copyOfStringList.addAll(stringList); Код добавит все элементы stringList во вновь созданный copyOfStringList

stingList.clear() удалить все элементы stingList

String[] itemArray = new String[stringList.size()];

String[] returnedArray = stringList.toArray(itemArray); toArray превращает ArrayList в массив Array

List list = Collections.synchronizedList(new ArrayList(...)); Чтобы синхронизировать ArrayList в Java, чтобы сделать его доступным для нескольких потоков, для этого вы можете использовать служебный класс Collections, как показано ниже.

ArrayList stringList = Arrays.asList(new String[]{"One", "Two", "Three"); //this is not read only List you can still update value of existing elements

hasNext() — если в итерируемом объекте (пока что это Collection) остались еще значение — метод вернет true

Вы можете использовать Iterator или ListIterator для обхода ArrayList . ListIterator позволит вам перемещаться в обоих направлениях, в то время как Iterator и ListIterator позволят вам удалять элементы из ArrayList в Java при обходе .

Iterator itr = stringList.iterator(); или ListIterator listItr = stringList.listIterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

Collections.sort(stringList) ; Метод Collections.sort(List list) для сортировки ArrayList Java в естественном порядке, определенном интерфейсом Comparable и может использовать метод Collections.sort(List list, Comparator c) для сортировки ArrayList Java на основе пользовательского Comparator

LinkedList

LinkedList has links to the data, next element of link and previous element

Вся работа с LinkedList сводится к изменению ссылок.

## Common methods of LinkedList

add(str1)- adding to the end of a list

add(1, str1)- adding an element and making it 1(1 means second element)

linkedList.clear()- delete all elements

remove(1) – delete element

indexOf()

contains()

size()

set() (вставка элемента с заменой)

addFirst()

addLast()

peekFirst(), peekLast()- returns first, last element

pollFirst(), pollLast() - returns first, last element and deletes it

toArray() –transfers list to an array, but it needs toconverse (String[])

String[] strarr = (String[]) linkedList.toArray(new String[3]);

## Стэк(Stack)

## import java.util.Stack;

## LIFO — "last in — first out"

## Stack<Card> deck = new Stack<>();

push() — добавляет элемент на верх стека. Когда мы отправляем карту в сброс, она ложится поверх сброшенных ранее карт;

pop() — удаляет верхний элемент из стека и возвращает его. Этот метод идеально подходит для реализации механики “игрок берет карту”

peek() — возвращает верхний элемент стека, но не удаляет его из стека

empty() – true if object is empty

int search(Object element) Ищет элемент в стеке. Если найден, возвращается его смещение от вершины стека. В противном случае возвращается 1.

## Очередь (Queue)

## Очередь (Queue) — это структура, знакомая нам из жизни. Очереди в магазины, к врачам. Кто первее пришёл (First In), тот первее и выйдет из очереди (First Out). В Java очередь представлена интерфейсом java.util.Queue.

element() - возвращает элемент из головы очереди. Элемент не удаляется. Если очередь пуста, инициируется исключение NoSuchElementException.

remove() - удаляет элемент из головы очереди, возвращая его. Инициирует исключение NoSuchElementException, если очередь пуста.

peek() - возвращает элемент из головы очереди. Возвращает null, если очередь пуста. Элемент не удаляется.

роll() - возвращает элемент из головы очереди и удаляет его. Возвращает null, если очередь пуста.

boolean offer(Е оbj) - пытается добавить оbj в очередь. Возвращает true, если оbj добавлен, и false в противном случае.

## Двусторонняя Очередь (dequeue)

## В документации сказано, что "Deques can also be used as LIFO (Last-In-First-Out) stacks. This interface should be used in preference to the legacy Stack class.", то есть вместо Stack рекомендуется использовать реализации Deque.

void addFirst(Е obj) - добавляет obj в голову двунаправленной очереди. Возбуждает исключение IllegalStateException, если в очереди ограниченной емкости нет места.

void addLast(Е obj) - добавляет obj в хвост двунаправленной очереди. Возбуждает исключение IllegalStateException, если в очереди ограниченной емкости нет места.

Е getFirst() - возвращает первый элемент двунаправленной очереди. Объект из очереди не удаляется. В случае пустой двунаправленной очереди возбуждает исключение NoSuchElementException.

Е getLast() - возвращает последний элемент двунаправленной очереди. Объект из очереди не удаляется. В случае пустой двунаправленной очереди возбуждает исключения NoSuchElementException.

boolean offerFirst(Е obj) - пытается добавить obj в голову двунаправленной очереди. Возвращает true, если obj добавлен, и false в противном случае. Таким образом, этот метод возвращает false при попытке добавить obj в полную двунаправленную очередь ограниченной емкости.

boolean offerLast(E obj) - пытается добавить obj в хвост двунаправленной очереди. Возвращает true, если obj добавлен, и false в против ном случае.

Е рор() - возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возбуждает исключение NoSuchElementException, если очередь пуста.

void push(Е obj) - добавляет элемент в голову двунаправленной очереди. Если в очереди фиксированного объема нет места, возбуждает исключение IllegalStateException.

Е peekFirst() - возвращает элемент, находящийся в голове двунаправленной очереди. Возвращает null, если очередь пуста. Объект из очереди не удаляется.

Е peekLast() - возвращает элемент, находящийся в хвосте двунаправленной очереди. Возвращает null, если очередь пуста. Объект из очереди не удаляется.

Е pollFirst() - возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возвращает null, если очередь пуста.

Е pollLast() - возвращает элемент, находящийся в хвосте двунаправленной очереди, одновременно удаляя его из очереди. Возвращает null, если очередь пуста.

Е removeLast() - возвращает элемент, находящийся в конце двунаправленной очереди, удаляя его в процессе. Возбуждает исключение NoSuchElementException, если очередь пуста.

Е removeFirst() - возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возбуждает исключение NoSuchElementException, если очередь пуста.

boolean removeLastOccurrence(Object obj) - удаляет последнее вхождение obj из двунаправленной очереди. Возвращает true в случае успеха и false если очередь не содержала obj.

boolean removeFirstOccurrence(Object obj) - удаляет первое вхождение obj из двунаправленной очереди. Возвращает true в случае успеха и false, если очередь не содержала obj.

Давайте посмотрим, какие есть реализации. И увидим интересный факт — в стан очередей "затесался" LinkedList ) То есть LinkedList реализует интерфейс как List, так и Deque. Но есть и "только очереди", например PriorityQueue. Про неё не часто вспоминают, а зря. Во-первых, в этой очереди нельзя использовать "non-comparable objects", т.е. должен быть или Comparator указан или все объекты должны быть comparable. Во-вторых, "this implementation provides O(log(n)) time for the enqueuing and dequeuing methods". Логарифмическая сложность тут не просто так

## Set

В множествах Set каждый элемент хранится только в одном экземпляре, а разные реализации Set используют разный порядок хранения элементов. [java.util.Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html).

Set — это "collection that contains no duplicate elements". Интересно, что сам интерфейс Set не добавляет новых методов к интерфейсу Collection, а лишь уточняет требования (про то, что не должно содержать дубликатов). Кроме того, из прошлого описания следует, что просто так из Set нельзя получить элемент. Для получения элементов используется Iterator.

Set имеет ещё несколько связанных с собой интерфейсов. Первый это SortedSet. Как и следует из названия, SortedSet указывает на то, что такой набор отсортирован, а следовательно элементы реализуют интерфейс Comparable или указан Comparator.

методы first (самый маленький по значению элемент) и last (самый большой по значению элемент).

У SortedSet есть наследник — NavigableSet. Цель этого интерфейса — описать методы навигации, которые нужны для более точного определения подходящих элементов. Из интересного — NavigableSet добавляет к привычному iterator (который идёт от меньшего к большему) итератор для обратного порядка — descendingIterator. Кроме того, NavigableSet позволяет при помощи метода descendingSet получить вид на себя (View), в котором элементы идут в обратном порядке. Это называется View, потому что через полученный элемент можно изменять элементы изначального Set. То есть по сути это представление изначальных данных другим способом, а не их копия. Интересно, что NavigableSet, подобно Queue, умеет pollFirst (минимальный) и pollLast (максимальный) элементы. То есть получает этот элемент и убирает из набора. Какие же есть реализации? Во-первых, самая известная реализация — на основе хэш-кода — [HashSet](https://javarush.ru/groups/posts/2147-hashset-v-java). Другая не менее известная реализация — на основе красно-чёрного дерева — [TreeSet](https://www.baeldung.com/java-tree-set).

## HashSet

## Стоит отметить, что порядок добавления стран во множество будет непредсказуемым. HashSet использует хэширование для ускорения выборки. Если вам нужно, чтобы результат был отсортирован, то пользуйтесь TreeSet.

public Iterator iterator()

public int size()

public boolean isEmpty()

public boolean contains(Object o)

public boolean add(Object o)

public boolean addAll(Collection c)

public Object[] toArray()

public boolean remove(Object o)

public boolean removeAll(Collection c)

public boolean retainAll(Collection c) - (retain — сохранить). Выполняет операцию "пересечение множеств".

public void clear()

public Object clone()

Методы аналогичны методам ArrayList за исключением того, что метод add(Object o) добавляет объект в множество только в том случае, если его там нет. Возвращаемое методом значение — true, если объект добавлен, и false, если нет.

## TreeSet

Класс TreeSet создаёт коллекцию, которая для хранения элементов применяет дерево. Объекты сохраняются в отсортированном порядке по возрастанию.

void add(Object o) Добавляет указанный элемент к этому набору, если он еще не присутствует.

boolean addAll(Collection c) Добавляет все элементы в указанной коллекции к этому набору.

void clear() Удаляет все элементы из этого набора.

Object clone() Возвращает мелкую копию этого экземпляра TreeSet.

Comparator comparator() Возвращает компаратор, используемый для порядка этого отсортированного набора, или null (нуль), если этот набор деревьев использует свои элементы естественного упорядочения.

boolean contains(Object o) Возвращает true, если этот набор содержит указанный элемент.

Object first() Возвращает первый (самый низкий) элемент, находящийся в этом отсортированном наборе.

SortedSet headSet(Object toElement) Возвращает представление (вид) части этого набора, элементы которого строго меньше, чем toElement.

boolean isEmpty() Возвращает true, если этот набор не содержит элементов.

Iterator iterator() Возвращает итератор над элементами этого набора.

Object last() Возвращает последний (самый высокий) элемент, находящийся в этом отсортированном наборе.

boolean remove(Object o) Удаляет указанный элемент из этого набора, если он присутствует.

int size() Возвращает количество элементов в этом наборе (его мощность).

SortedSet subSet(Object fromElement, Object toElement) Возвращает представление (вид) части этого набора, элементы которого варьируются от fromElement, включительно, до toElement, исключительно.

SortedSet tailSet(Object fromElement) Возвращает представление (вид) части этого набора, элементы которого больше или равны fromElement.

## SortedSet

E first(): возвращает первый элемент набора

E last(): возвращает последний элемент набора

SortedSet<E> headSet(E end): возвращает объект SortedSet, который содержит все элементы первичного набора до элемента end

SortedSet<E> subSet(E start, E end): возвращает объект SortedSet, который содержит все элементы первичного набора между элементами start и end

SortedSet<E> tailSet(E start): возвращает объект SortedSet, который содержит все элементы первичного набора, начиная с элемента start

## Карты (Map)

java.util.Map.

Карты — это такая структура данных, в которой данные хранятся по ключу. Например, ключом может служить ID или код города. И именно по этому ключу будут искаться данные.

Маппинг "ключ - значение" не является коллекцией. И карты можно скорей представить как коллекция ключей, коллекция значений, коллекция пар "ключ - значение". Т.к. карты не являются коллекциями (не наследуются от Collections), то они не содержат метод contains. И это ведь логично. Карта состоит из ключей и значений. Что из этого должен проверять метод contains и как не запутаться? Поэтому, интерфейс Map имеет две разные версии: containsKey (содержит ли ключ) и containsValue (содержит ли значение).

При помощи keySet позволяет получить набор ключей (тот самый Set). А при помощи метода values можем получить коллекцию значений в карте. Ключи в карте уникальны, что подчёркивается структурой данных Set. Значения же могут повторяться, что подчёркивает стуктура данных Collection. Кроме того, при помощи метода entrySet можем получить набор пар "ключ - значение".

Хотелось бы ещё увидеть, что HashMap очень похож на HashSet, а TreeMap на TreeSet. У них даже схожие интерфейсы: NavigableSet и NavigableMap, SortedSet и SortedMap.

Закончить можно занимательным фактом, что коллекция Set внутри себя использует Map, где добавляемые значения — это ключи, а значение везде одинаковое. Занимательно это потому, что Map не является коллекцией и возвращает Set, который является коллекцией, но по факту реализован как Map. Немного сюр, но вот так вот вышло )

