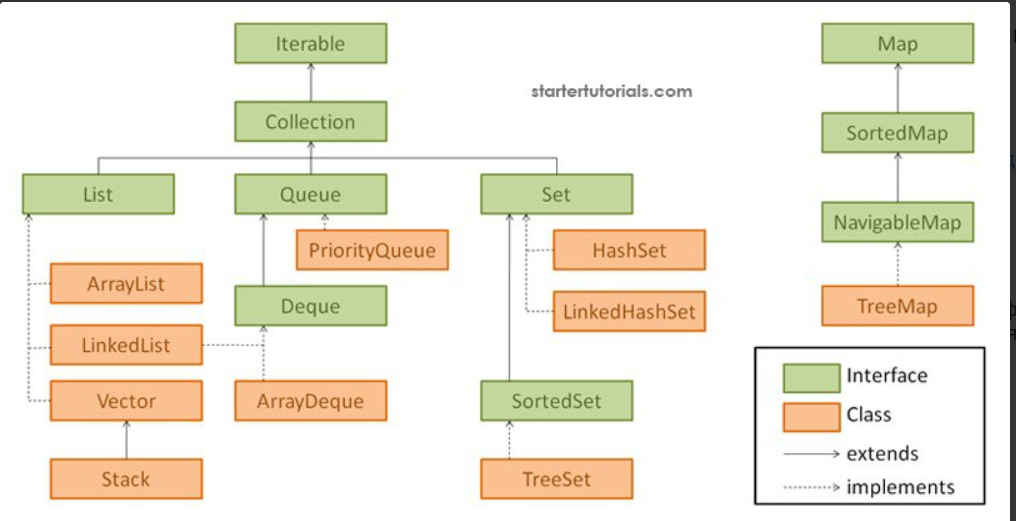
|  |  |  |
| --- | --- | --- |
| **Интерфейс** | **Класс/Реализация** | **Описание** |
| List | ArrayList | Список |
| LinkedList | Список |
| Vector | Вектор |
| Stack | Стек |
| Set | HashSet, LinkedHashSet | Множество |
| TreeSet | Множество |
| SortedSet interface | Отсортированное множество |
| Map | HashMap, LinkedHashMap | Карта/Словарь |
| TreeMap | Карта/Словарь |
| SortedMap (interface) | Отсортированный словарь |
| Hashtable | Хеш-таблица |

<https://metanit.com/java/tutorial/5.1.php>



Интерфейс Collection является базовым для всех коллекций, определяя основной функционал:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface Collection<E> extends Iterable<E>{        // определения методов  } |

Интерфейс Collection является обобщенным и расширяет интерфейс Iterable, поэтому все объекты коллекций можно перебирать в цикле по типу for-each.

Обобщенный интерфейс Queue<E> расширяет базовый интерфейс Collection и определяет поведение класса в качестве однонаправленной очереди. Она добавляет в конец очереди и может возвратить начальный элемент очереди.метод offer для добавления в очередь, метод poll для извлечения элемента из головы очереди

Интерфейс Deque расширяет вышеописанный интерфейс Queue и определяет поведение двунаправленной очереди, которая работает как обычная однонаправленная очередь, либо как стек, действующий по принципу LIFO (последний вошел - первый вышел).Она добавляет в конец/начало очереди и может возвратить начальный/конечный элемент очереди

класс **ArrayDeque<E>**. Этот класс представляют обобщенную двунаправленную очередь, наследуя функционал от класса AbstractCollection и применяя интерфейс Deque.

Обобщенный класс LinkedList<E> представляет структуру данных в виде связанного списка. реализует интерфейсы List, Dequeue и Queue

У каждого элемента помимо тех данных, которые он хранит, имеется ****ссылка на предыдущий и следующий элемент****. По этим ссылкам можно переходить от одного элемента к другому. ****Вся работа с** LinkedList **сводится к изменению ссылок.****Вставка и удаление в середину LinkedList устроены гораздо проще, чем в ArrayList. Мы просто переопределяем ссылки соседних элементов, а ненужный элемент “выпадает” из цепочки ссылок. В то время как в ArrayList мы:

* проверяем, хватает ли места (при вставке)
* если не хватает — создаем новый массив и копируем туда данные (при вставке)
* удаляем/вставляем элемент, и сдвигаем все остальные элементы вправо/влево (в зависимости от типа операции). Причем сложность этого процесса сильно зависит от размера списка. Одно дело — скопировать/сдвинуть 10 элементов, и совсем другое — сделать то же самое с миллионом элементов.

LinkedList требует больше памяти для хранения такого же количества элементов, потому что кроме самого элемента хранятся еще указатели на следующий и предыдущий элементы списка, тогда как в ArrayList элементы просто идут по порядку

с операциями произвольного доступа LinkedList работает менее эффективно.

То есть, если в твоей программе чаще происходят операции вставки/удаления с серединой списка, LinkedList должен быть быстрее, чем ArrayList.

****доступ к элементу осуществляется в** ArrayList **за фиксированное время.****

## **Операция поиска**

Операция поиска в ArrayList довольно быстро по сравнению с операцией поиска LinkedList. Метод ArrayList get (int index) дает производительность O (1), в то время как производительность LinkedList равна O (n). Это связано с тем, что ArrayList позволяет произвольный доступ к элементам в списке, поскольку он работает с основанной на индексе структурой данных, в то время как LinkedList не разрешает произвольный доступ, поскольку он не имеет индексов для непосредственного доступа к элементам, он должен пересекать список для извлечения или получить доступ к элементу из списка.

## **Манипуляции**

Манипуляция с ArrayList медленная, поскольку она внутренне использует массив. Если нам нужно вставить или удалить элемент в ArrayList, он может принимать O (n), поскольку он внутренне использует массив, и нам может потребоваться сдвинуть элементы в случае вставки или удаления. С другой стороны, манипуляции с LinkedList быстрее, чем ArrayList, потому что он использует двусвязный список, поэтому в памяти не требуется сдвиг бит. Если нам нужно вставить или удалить элемент в LinkedList, он будет принимать O (1), поскольку он внутренне использует дважды.

## **Накладные расходы памяти**

ArrayList поддерживает индексы и данные элементов, в то время как LinkedList поддерживает данные элемента и два указателя для соседних узлов, следовательно, потребление памяти в LinkedList сравнительно велико.

Интерфейс **Set** расширяет интерфейс Collection и представляет набор уникальных элементов.

Интерфейс **SortedSet** предназначен для создания коллекций, который хранят элементы в отсортированном виде (сортировка по возрастанию).

Интерфейс **NavigableSet** расширяет интерфейс **SortedSet** и позволяет извлекать элементы на основании их значений.

Обобщенный класс TreeSet<E> представляет структуру данных в виде дерева, в котором все объекты хранятся в отсортированном виде по возрастанию.И поскольку при вставке объекты сразу же сортируются по возрастанию, то при выводе в цикле for мы получим отсортированный набор

Для того, чтобы объекты можно было сравнить и сортировать (компилятор не знает как нужно сортировать объекты), они должны применять интерфейс **Comparable<E>**. При применении интерфейса он типизируется текущим классом.

Интерфейс Comparable содержит один единственный метод int compareTo(E item), который сравнивает текущий объект с объектом, переданным в качестве параметра. Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвратит ноль, значит, оба объекта равны. Также можно задать и свою реализацию.

Однако перед нами может возникнуть проблема, что если разработчик не реализовал в своем классе, который мы хотим использовать, интерфейс Comparable, либо реализовал, но нас не устраивает его функциональность, и мы хотим ее переопределить? На этот случай есть еще более гибкий способ, предполагающий применение интерфейса Comparator<E>(если нужны другие сравнения кроме **Comparable**).

 Для применения интерфейса нам вначале надо создать класс компаратора, который реализует этот интерфейс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class PersonComparator implements Comparator<Person>{        public int compare(Person a, Person b){            return a.getName().compareTo(b.getName());      }  } |

 Теперь используем класс компаратора для создания объекта TreeSet:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | PersonComparator pcomp = new PersonComparator();  TreeSet<Person> people = new TreeSet<Person>(pcomp);  people.add(new Person("Tom"));  people.add(new Person("Nick"));  people.add(new Person("Alice"));  people.add(new Person("Bill"));  for(Person  p : people){ |

Начиная с JDK 8 в механизм работы компараторов были внесены некоторые дополнения. В частности, теперь мы можем применять сразу несколько компараторов по принципу приоритета. Для этого создаем несколько компараторов.

Comparator<Person> pcomp = new PersonNameComparator().thenComparing(new PersonAgeComparator());

В данном случае сначала применяется сортировка по имени, а потом по возрасту.

Интерфейс **Map<K, V>** представляет отображение или иначе говоря словарь, где каждый элемент представляет пару "ключ-значение". При этом все ключи уникальные в рамках объекта Map.

**HashMap**

Интерфейс **SortedMap** расширяет Map и создает отображение, в котором все элементы отсортированы в порядке возрастания их ключей.

Интерфейс **NavigableMap** расширяет интерфейс SortedMap и обеспечивает возможность получения элементов отображения относительно других элементов.

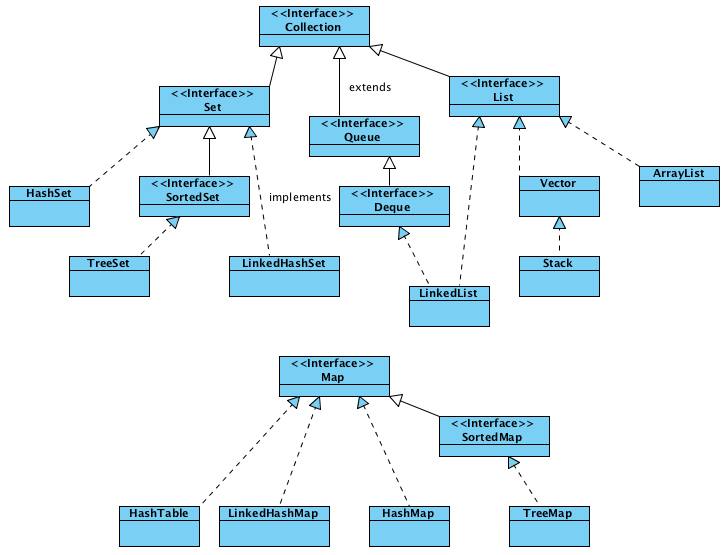
Класс TreeMap<K, V> представляет отображение в виде дерева. все объекты автоматически сортируются по возрастанию их ключей.

Одним из ключевых методов интерфейса Collection является метод Iterator<E> iterator(). Он возвращает итератор - то есть объект, реализующий интерфейс **Iterator**.

 E next();

 boolean hasNext();

 void remove();



#### **21. Какую функциональность представляет класс Collections**

Некоторые из методов

|  |  |
| --- | --- |
| Сигнатура метода | Описание |
| Collections.sort(List myList) | Сортирует список в естественном порядке. |
| Collections.sort(List, Comparator c) | Сортировка с использованием компаратора. |
| Collections.shuffle(List myList) | Перемешивает коллекцию в случайном порядке. |
| Collections.reverse(List myList) | Переворачивает коллекцию в обратном порядке. |
| Collections.binarySearch(List mlist, T key) | поиск в коллекции по ключу с использованием бинарного поиска. |
| Collections.copy(List dest, List src) | Копирует коллекцию источник src в dest. |
| Collections.frequency(Collection c, Object o) | Возвращает число вхождений объекта в коллекции. |
| Collections.synchronizedCollection(Collection c) | Возвращает синхронизированную (потокобезопасную) коллекцию. |
| * Collections.unmodifiableList(list); |  |

#### **25. Как получить коллекцию только для чтения?**

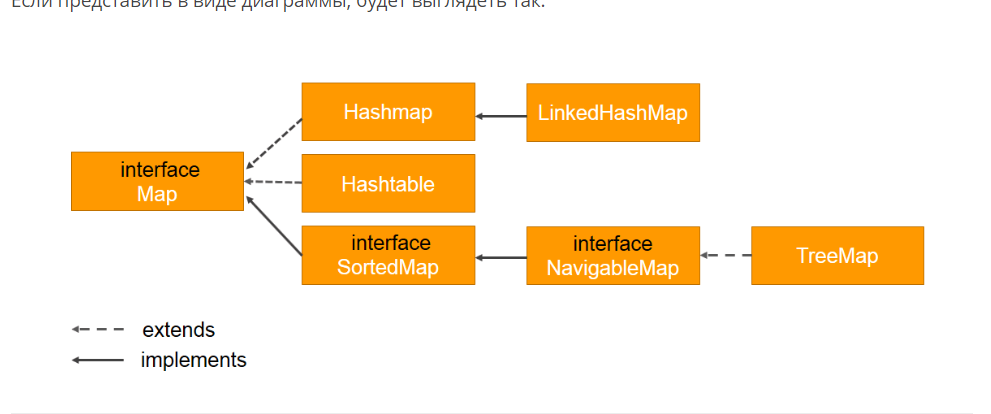
Используйте следующие методы:

* Collections.unmodifiableList(list);
* Collections.unmodifiableSet(set);
* Collections.unmodifiableMap(map);

Все они принимают коллекцию в качестве параметра, и возвращают коллекцию только для чтения с теми же элементами внутри.

Map

это структура данных, в которой объекты хранятся не по одному, как во всех остальных, а в паре "ключ - значение".



* ****HashMap****- хранит значения в произвольном порядке, но позволяет быстро искать элементы карты. Позволяет задавать ключ или значение ключевым словом ****null****.
* ****LinkedHashMap****- хранит значения в порядке добавления.
* ****TreeMap****- сама сортирует значения по заданному критерию. TreeMap используется либо с Comparable элементами, либо в связке с Comparator. Смотрите статью ["Интерфейсы Comparable и Comparator"](https://vertex-academy.com/tutorials/ru/interfejsy-comparable-comparator-java/).
* ****Hashtable****- как ****HashMap****, только не позволяет хранить ****null****и синхронизирован с точки зрения многопоточности - это значит, что много потоков могут работать безопасно с Hashtable. Но данная реализация старая и медленная, поэтому сейчас уже не используется в новых проектах.

## ****Операции с Map****

Ниже мы рассмотрим 6 операций с Map:

1. ****put(K key, V value)**** - добавляет элемент в карту;

2. ****get(Object key)****- ищет значение по его ключу;

3. **remove**(Object key)****- удаляет значение по его ключу;

4. ****containsKey(Object key)****- спрашивает, есть ли в карте заданный ключ;

5. ****containsValue(Object value)****- спрашивает есть ли в карте заданное значение;

6. ****size()**** - возвращает размер карты (количество пар "ключ-значение").

for(Map.Entry<String, String> entry: myHashMap.entrySet())

            System.out.println(entry.getKey() + " - " + entry.getValue());

Примеры работы  
<https://vertex-academy.com/tutorials/ru/map-v-java-hashmap/>

iterable vs iterator java отличие - оба интерфейсы  
<https://overcoder.net/q/28602/%D0%B2-%D1%87%D0%B5%D0%BC-%D1%80%D0%B0%D0%B7%D0%BD%D0%B8%D1%86%D0%B0-%D0%BC%D0%B5%D0%B6%D0%B4%D1%83-%D0%B8%D1%82%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%BE%D0%BC-%D0%B8-%D0%B8%D1%82%D0%B5%D1%80%D0%B8%D1%80%D1%83%D0%B5%D0%BC%D1%8B%D0%BC-%D0%B8-%D0%BA%D0%B0%D0%BA-%D0%B8%D1%85-%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D1%8C>

Классы, реализующие интерфейс Iterable, могут применяться в конструкции for-each, которая использует Iterator.