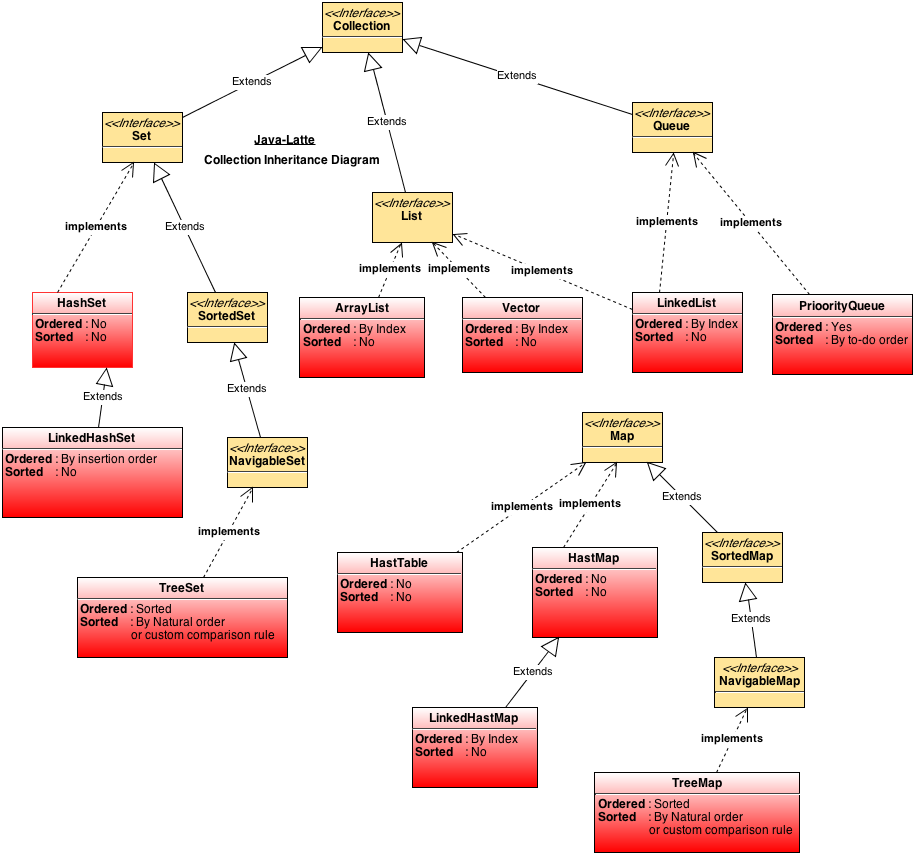
**Про Java Collections Framework: почти все и сразу**

[**Java Collections Framework**](http://docs.oracle.com/javase/tutorial/collections/intro/index.html) это коллекция интерфейсов и классов, которые используются для сохранения и обработки данных.



Для хранения наборов данных в Java предназначены массивы. Однако их не всегда удобно использовать, прежде всего потому, что они имеют фиксированную длину. Эту проблему в Java решают коллекции. Однако суть не только в гибких по размеру наборах объектов, но в и том, что классы коллекций реализуют различные алгоритмы и структуры данных, например, такие как стек, очередь, дерево и ряд других.

Классы коллекций располагаются в пакете java.util, поэтому перед применением коллекций следует подключить данный пакет.

Хотя в Java существует множество коллекций, но все они образуют стройную и логичную систему. Во-первых, в основе всех коллекций лежит применение того или иного интерфейса, который определяет базовый функционал. Среди этих интерфейсов можно выделить следующие:

* **Collection**: базовый интерфейс для всех коллекций и других интерфейсов коллекций
* **Queue**: наследует интерфейс Collection и представляет функционал для структур данных в виде очереди
* **Deque**: наследует интерфейс Queue и представляет функционал для двунаправленных очередей
* **List**: наследует интерфейс Collection и представляет функциональность простых списков
* **Set**: также наследует интерфейс Collection и используется для хранения множеств уникальных объектов
* **SortedSet**: расширяет интерфейс Set для создания сортированных коллекций
* **NavigableSet**: расширяет интерфейс SortedSet для создания коллекций, в которых можно осуществлять поиск по соответствию
* **Map**: предназначен для созданий структур данных в виде словаря, где каждый элемент имеет определенный ключ и значение. В отличие от других интерфейсов коллекций не наследуется от интерфейса Collection

Эти интерфейсы частично реализуются абстрактными классами:

* **AbstractCollection**: базовый абстрактный класс для других коллекций, который применяет интерфейс Collection
* **AbstractList**: расширяет класс AbstractCollection и применяет интерфейс List, предназначен для создания коллекций в виде списков
* **AbstractSet**: расширяет класс AbstractCollection и применяет интерфейс Set для создания коллекций в виде множеств
* **AbstractQueue**: расширяет класс AbstractCollection и применяет интерфейс Queue, предназначен для создания коллекций в виде очередей и стеков
* **AbstractSequentialList**: также расширяет класс AbstractList и реализует интерфейс List. Используется для создания связанных списков
* **AbstractMap**: применяет интерфейс Map, предназначен для создания наборов по типу словаря с объектами в виде пары "ключ-значение"

С помощью применения вышеописанных интерфейсов и абстрактных классов в Java реализуется широкая палитра классов коллекций - списки, множества, очереди, отображения и другие, среди которых можно выделить следующие:

* **ArrayList**: простой список объектов
* **LinkedList**: представляет связанный список
* **ArrayDeque**: класс двунаправленной очереди, в которой мы можем произвести вставку и удаление как в начале коллекции, так и в ее конце
* **HashSet**: набор объектов или хеш-множество, где каждый элемент имеет ключ - уникальный хеш-код, сортировка по хеш-коду
* **TreeSet**: набор отсортированных объектов в виде дерева
* **LinkedHashSet**: связанное хеш-множество, в порядке котором элементы были вставлены
* **PriorityQueue**: очередь приоритетов
* **ArrayDeque** - базируется на массиве и используется при реализации стека
* **HashMap**: структура данных в виде словаря, в котором каждый объект имеет уникальный ключ и некоторое значение
* **TreeMap**: структура данных в виде дерева, где каждый элемент имеет уникальный ключ и некоторое значение

Схематично всю систему коллекций вкратце можно представить следующим образом:

**Интерфейс Collection**

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.

Среди методов интерфейса Collection можно выделить следующие:

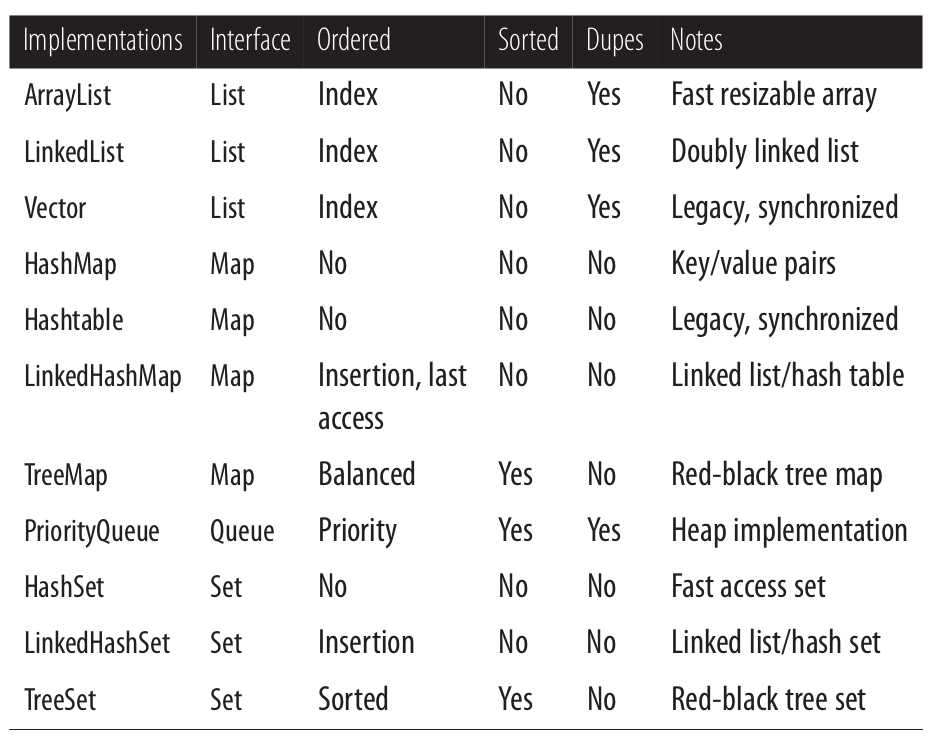
* boolean **add** (E item): добавляет в коллекцию объект item. При удачном добавлении возвращает true, при неудачном - false
* boolean **addAll** (Collection<? extends E> col): добавляет в коллекцию все элементы из коллекции col. При удачном добавлении возвращает true, при неудачном - false
* void **clear** (): удаляет все элементы из коллекции
* boolean **contains** (Object item): возвращает true, если объект item содержится в коллекции, иначе возвращает false
* boolean **isEmpty** (): возвращает true, если коллекция пуста, иначе возвращает false
* Iterator<E> **iterator** (): возвращает объект Iterator для обхода элементов коллекции
* boolean **remove** (Object item): возвращает true, если объект item удачно удален из коллекции, иначе возвращается false
* boolean **removeAll** (Collection<?> col): удаляет все объекты коллекции col из текущей коллекции. Если текущая коллекция изменилась, возвращает true, иначе возвращается false
* boolean **retainAll** (Collection<?> col): удаляет все объекты из текущей коллекции, кроме тех, которые содержатся в коллекции col. Если текущая коллекция после удаления изменилась, возвращает true, иначе возвращается false
* int **size** (): возвращает число элементов в коллекции
* Object[] **toArray** (): возвращает массив, содержащий все элементы коллекции

Все эти и остальные методы, которые имеются в интерфейсе Collection, реализуются всеми коллекциями, поэтому в целом общие принципы работы с коллекциями будут одни и те же. Единообразный интерфейс упрощает понимание и работу с различными типами коллекций. Так, добавление элемента будет производиться с помощью метода add, который принимает добавляемый элемент в качестве параметра. Для удаления вызывается метод remove(). Метод clear будет очищать коллекцию, а метод size возвращать количество элементов в коллекции.

Java Collections Framework состоит из следующих компонентов:

* List это упорядоченный список объектов (иногда еще называют последовательностью объектов). Элементы списка могут вставляться или извлекаться по их индексу в списке (индексация начинается с 0). В эту группу входят: ArrayList, LinkedList, (старые синхронизированные Vector, Stack).
* Set это не упорядоченная коллекция. Главная особенность множеств - уникальность элементов, то есть один и тот же элемент не может содержаться в множестве дважды. Есть такие имплементации Set интерфейса: HashSet, TreeSet, LinkedHashSet, EnumSet. HashSet хранит элементы в хэш-таблице, что предполагает эффективную реализацию, но не гарантирует порядок итерации элементов. TreeSet хранит элементы в [**красно-чёрном дереве**](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D1%91%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE) и упорядочивает элементы по их значениям, эта коллекция существено медленее чем HashSet. LinkedHashSet реализована как хэш-таблица и хранит элементы в связаном списке, в порядке котором они были вставленны.
* Map (иногда можно встретить названия: отображения, словарь, ассоциативный массив, карты) отображает ключи в значения и не может иметь дубликаты ключей. Есть три основных имплементации Map интерфейса TreeMap, HashMap и LinkedHashMap.  TreeMap хранит элементы в [**красно-чёрном дереве**](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D1%91%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE) и упорядычивает элементы по их ключам и медленее чем HashMap. HashMap не гарантирует воспроизводимость порядка вставки элементов. LinkedHashMap упорядычивает элементы в порядке их вставки.
* Iterator/ListIterator используются для итерации элементов коллекции. Основное отличие между Iterator и ListIterator в том, что первый позволяет итерировать только в одном направлении, а второй в обоих направлениях.

Выше сказаное можно резюмировать в такой таблице



Производительность методов каждого класса можно записать через *O*-нотацию

| **Метод** | **Тип** | **Время** |
| --- | --- | --- |
| get, set | ArrayList | O(1) |
| add, remove | ArrayList | O(n) |
| contains, indexOf | ArrayList | O(n) |
| get, put, remove, containsKey | HashMap | O(1) |
| add, remove, contains | HashSet | O(1) |
| add, remove, contains | LinkedHashSet | O(1) |
| get, set, add, remove (с любого конца) | LinkedList | O(1) |
| get, set, add, remove (по индексу) | LinkedList | O(n) |
| contains, indexOf | LinkedList | O(n) |
| peek | PriorityQueue | O(1) |
| add, remove | PriorityQueue | O(log n) |
| remove, get, put, containsKey | TreeMap | O(log n) |
| add, remove, contains | TreeSet | O(log n) |
|  |  |  |

**List**

**ArrayList**

ArrayList это массив с изменяемым количесвтом элементов (стандартный array имеет фиксированый размер), его элементы могут быть доступны непосредственно по индексу. Он реализует все операции списка и позволят вставлять все объекты, включая null. Контейнер ArrayList, оптимизированный для произвольного доступа к элементам, но с относительно медленными операциями вставки/удаления элементов в середине списка.

В случае переполнения массива появляется необходимость в новом, имеющем больше места. Размещение и перемещение всех элементов будет занимать O(n) времени. Также, необходимо добавление и удаление элементов для передвижения существующих элементов в массиве. Это, возможно, самое большое неудобство в использовании ArrayList.

Поиск в ArrayList проходит за O(1). Удаление первого элемента (худший случай) происходит за O(n), для последнего элемента (лучший случай) за O(1). Вставка происходит за O(n).

Пример

ArrayList<String> lst = **new** ArrayList<>();

*// добавление элементов*

lst.add("Mercury");

lst.add("Venus");

lst.add("Earth");

lst.add("Saturn");

lst.add("Neptune");

*// добавлене несколько элементов сразу*

lst.addAll(Arrays.asList("PlanetX", "PlanetY", "PlanetZ"));

*// отображение массива*

System.out.println(lst);

*// добавление элемента в определеную позицию*

lst.add(0, "Jupiter");

lst.add(1, "Uranus");

*// получение True если значение есть в массиве, иначе False*

boolean isExist = lst.contains("Earth")

*// получение True если все значения есть в массиве, иначе False*

ArrayList<String> favorite = **new** ArrayList<>();

favorite.add("Earth");

favorite.add("Saturn");

boolean isContains = planets.containsAll(favorite);

*// удаление элемента по значению*

lst.remove("Saturn");

lst.remove("Neptune");

*// оставить в списке только указаные элементы*

ArrayList<String> favorite = **new** ArrayList<>();

favorite.add("Earth");

favorite.add("Saturn");

planets.retainAll(favorite);

*// отображение массива*

System.out.println(lst);

*// удаление элемента по индексу*

lst.remove(1);

*// удалить все элементы в списке*

ArrayList<String> favorite = **new** ArrayList<>();

favorite.add("Earth");

favorite.add("Saturn");

planets.removeAll(favorite);

*// удалить элементы списка, которые соответсвуют условию Predicate*

**class** **SamplePredicate**<T> **implements** **Predicate**<T> {

T varc1;

**public** boolean **test**(T varc) {

**if**(varc1.equals(varc)) {

**return** true;

}

**return** false;

}

}

SamplePredicate<String> filter = **new** SamplePredicate<>();

filter.varc1 = "Saturn";

planets.removeIf(filter);

*// обновление значения по индекску*

lst.set(1, "Neptune");

*// получить индекс элемента по значению, если не найденно то -1*

**int** pos = lst.indexOf("Jupiter");

*// получение элемента по индексу*

String planet = lst.get(3)

*// размер массива*

**int** amount = lst.size()

*// получить часть массива*

ArrayList<String> lst2 = **new** ArrayList<>(lst.subList(1, 3));

*// обэдинение массивов*

lst.addAll(lst2)

*// удалить все элементы*

lst.clear()

*// отсортировать список в обратном порядке*

List reversedList = Collections.reverse(planets);

*// заменить значение каждого элемента списка на результат оператор*

*// оператор*

**class** **MyOperator**<T> **implements** **UnaryOperator**<T>{

T varc1;

**public** T **apply**(T varc){

**return** varc1;

}

}

MyOperator<String> **operator** = **new** MyOperator<>();

**operator**.varc1 = "Earth";

planets.replaceAll(**operator**);

System.out.println(planets);

*// [Earth, Earth, Earth, Earth]*

*// выполнить действие над каждым элементом списка*

*// действие*

**class** **MyConsumer**<T> **implements** **Consumer**<T>{

**public** **void** **accept**(T planet){

System.out.println("We are flying to " + planet);

}

}

MyConsumer<String> action = **new** MyConsumer<>();

planets.forEach(action);

**LinkedList**

LinkedList — это связанный список ссылок на элементы. Таким образом, для доступа к элементу в центре, приходится производить поиск с самого начала и до конца листа. С другой стороны, добавление и удаление элемента в LinkedList быстрее (чем в ArrayList) по той причине, что эти операции лишь изменяют ссылки.

Контейнер LinkedList, оптимизированный для последовательного доступа, с быстрыми операциями вставки/удаления в середине списка. Произвольный доступ к элементам LinkedList выполняется относительно медленно, т.к. требует полного перебора элементов. Класс представляет структуру данных связного списка и реализует интерфейсы List, Dequeue, Queue.

Поиск в LinkedList проходит за O(n). Удаление происходит за O(1). Вставка происходит за O(1).

Пример

LinkedList<String> lst = **new** LinkedList<>();

*// добавление элементов*

lst.add("Mercury");

lst.add("Earth");

lst.addLast("Saturn");

lst.addFirst("Neptune");

lst.add(1, "Venus");

*// вставка в начало*

lst.push("Kopernik")

*// отображение списка*

System.out.println(lst);

*// добавлене несколько элементов сразу*

lst.addAll(Arrays.asList("PlanetX", "PlanetY", "PlanetZ"));

*// изменить элемент списка по индексу*

String planet = lst.set(1, "PlanetA")

*// получение True если значение есть в списке, иначе False*

boolean isExist = lst.contains("Earth")

*// получение элемента по индексу*

String planet = lst.get(3)

*// получение первого элемента*

String planet = lst.getFirst()

*// получение последнего элемента*

String planet = lst.getLast()

*// получить все элементы списка в виде массива*

String[] planets = lst.toArray()

*// удалить все элементы*

lst.clear()

*// получить индекс первого элемента с указаным значением, если не найденно то -1*

int pos = lst.indexOf("Jupiter");

*// получить индекс последнего элемента с указаным значением, если не найденно то -1*

int pos = lst.lastIndexOf("Jupiter");

*// удаление первого элемента со значением*

lst.remove("PlanetX");

*// удалить первый элемент из списка и вернуть значение*

String planet = lst.removeFirst();

*// удалить первый элемент из списка и вернуть значение*

String planet = lst.poll();

*// удалить первый элемент из списка и вернуть значение*

String planet = lst.pop();

*// удалить последний элемент из списка и вернуть значение*

String planet = lst.removeLast();

*// размер списка*

int amount = lst.size()

*// перебор элементов списка*

**for**(String str: lst) {

System.out.println(str);

}

*// быстрый способ добавить/объединить списки*

String[] morePlanets = {"Planet1", "Planet2", "Planet3"};

Collections.addAll(lst, moreInts);

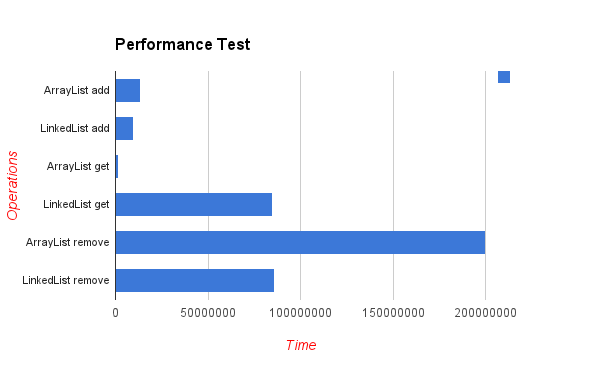
*// конвертирование LinkedList в ArrayList*

List<String> arr = **new** ArrayList<>(lst);

Сравнение LinkedList и ArrayList по скорости операций:

| **Метод** | **Arraylist** | **LinkedList** |
| --- | --- | --- |
| get(index) | O(1) | O(n) |
| add(E) | O(n) | O(1) |
| add(E, index) | O(n) | O(n) |
| remove(index) | O(n) | O(n) |
| Iterator.remove() | O(n) | O(1) |
| Iterator.add(E) | O(n) | O(1) |

Сравнеие производительности Arraylist и LinkedList

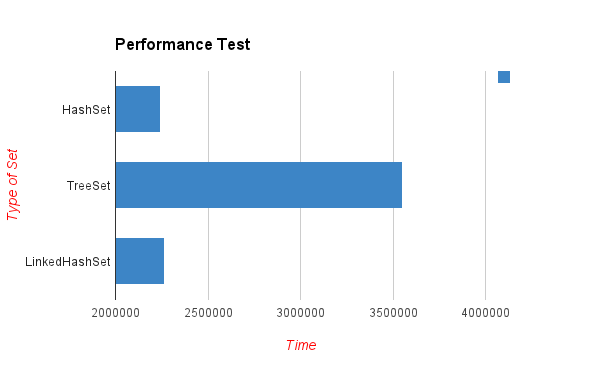
**[](http://www.programcreek.com/2013/03/arraylist-vs-linkedlist-vs-vector/)**

**Set**

HashSet, TreeSet и LinkedHashSet относятся к семейству Set. В множествах Set каждый элемент хранится только в одном экземпляре, а разные реализации Set используют разный порядок хранения элементов. В HashSet порядок элементов определяется по сложному алгоритму и не гарантирует порядок в котором элементы вставлялись. Если порядок хранения для вас важен, используйте контейнер TreeSet, в котором объекты хранятся отсортированными по возрастанию в порядке сравнения или LinkedHashSet с хранением элементов в порядке добавления.

Множества часто используются для проверки принадлежности, чтобы вы могли легко проверить, принадлежить ли объект заданному множеству, поэтому на практике обычно выбирается реализация HashSet, оптимизированная для быстрого поиска.

Сравним времени на операцию add() для HashSet, TreeSet, LinkedHashSet.

**[](http://www.programcreek.com/2013/03/hashset-vs-treeset-vs-linkedhashset/)**

**HashSet**

HashSet расширяет AbstractSet и реализует интерфейс Set. HashSet создает коллекцию, которая использует хеш-таблицу для сохранения своих значений. Класс Object и его наследники имеют метод hashCode(), который используется классом HashSet для эффективного размещения объектов, заносимых в коллекцию. Ключ используется для определения уникальности элемента и называется хеш-кодом.

Особености HasSet:

1. HashSet не поддерживает порядок своих элементов, а это значит, что элементы будут возвращены в любом порядже.
2. HashSet не разрешает хранить дубликаты. Если вы добавите существующий элемент, то старое значение будет переписано.
3. HashSet разрешает добавить в колекцию null значение, но только одно значение.
4. HashSet не синхронизировано.

Пример

HashSet<String> planets = **new** HashSet<>();

*// добавление элементов*

planets.**add**("Mercury");

planets.**add**("Earth");

planets.**add**("Saturn");

planets.**add**("Neptune");

planets.**add**("Venus");

*// отображение множества, обратите внимание на порядок вывода*

System.**out**.println(planets);

*// получение True если значение есть в множестве, иначе False*

boolean isExist = planets.contains("Earth")

*// пустое множество?*

boolean isEmpty = planets.isEmpty()

*// удаление элемента по значению*

planets.**remove**("Saturn");

*// удалить все элементы*

planets.clear()

*// размер множества*

**int** amount = planets.size()

*// перебор элементов множества*

**for**(String str: planets) {

System.**out**.println(str);

}

*// конвертирование HashSet в ArrayList*

List<String> arr = **new** ArrayList<>(planets);

*// конвертирование List в Set*

Set<Integer> **set** = **new** HashSet<>(list);

*// конвертирование List в Set, если надо учитывать сравнение элементов*

Set<Integer> **set** = **new** TreeSet<>(aComparator);

**set**.addAll(list);

**LinkedHashSet**

Класс LinkedHashSet расширяет класс HashSet, не добавляя никаких новых методов. Класс поддерживает связный список элементов множества в том порядке, в котором они вставлялись.

Пример

LinkedHashSet<String> planets = **new** LinkedHashSet<>();

*// добавление элементов*

planets.**add**("Mercury");

planets.**add**("Earth");

planets.**add**("Saturn");

planets.**add**("Neptune");

*// отображение множества, обратите внимание на порядок вывода*

System.**out**.println(planets);

**TreeSet**

TreeSet реализует интерфейс SortedSet.

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

Время доступа к элементам небольшое, что делает TreeSet отличным выбором для сохранения большого количкства отсортироыанных данных и к которым должен быть обеспечен быстрый доступ.

TreeSet не поддерживает хранение элементов типа null.

Пример

TreeSet<String> planets = **new** TreeSet<>();

*// добавление элементов*

planets.**add**("Mercury");

planets.**add**("Earth");

planets.**add**("Saturn");

planets.**add**("Neptune");

planets.**add**("Venus");

*// добавлене несколько элементов сразу*

planets.addAll(Arrays.asList("PlanetX", "PlanetY", "PlanetZ"));

*// отображение множества, обратите внимание на порядок вывода*

System.**out**.println(planets);

*// получение True если значение есть в множестве, иначе False*

boolean isExist = planets.contains("Earth")

*// пустое множество?*

boolean isEmpty = planets.isEmpty()

*// получить первый элемент*

String planet = planets.first();

*// получить последний элемент*

String planet = planets.last();

*// получить все элементы перед указаным*

Set planetsBefore = planets.headSet("Neptune");

*// получить все элементы после указаного*

Set planetsAfter = planets.tailSet("Neptune");

*// получить все элементы между двумя указаными*

Set planetsSub = planets.subSet("Mercury", "Venus");

*// удаление элемента по значению*

planets.**remove**("Saturn");

*// удалить все элементы*

planets.clear()

*// размер множества*

**int** amount = planets.size()

*// перебор элементов множества*

**for**(String str: planets) {

System.**out**.println(str);

}

*// получить индекс элемента*

**int** pos = planets.headSet("Neptune").size()

*// определим произволный Comparator для TreeSet*

Set planets2 = **new** TreeSet<>(**new** Comparator<String>() {

@Override

**public** **int** **compare**(String o1, String o2) {

*// define comparing logic here*

**return** Integer.compare(o1.length() , o2.length());

}

});

planets2.**add**("Earth");

planets2.**add**("Saturn");

planets2.**add**("Neptune");

System.**out**.println(planets2);

**NavigableSet**

Интерфейс NavigableSet наследуется от SortedSet и предоставляет возможность навигации по множеству в обоих направлениях.

Пример

NavigableSet<String> planets = **new** TreeSet<>();

*// добавление элементов*

planets.**add**("Mercury");

planets.**add**("Earth");

planets.**add**("Saturn");

planets.**add**("Neptune");

planets.**add**("Venus");

*// печать множества*

System.**out**.println(planets);

*// обратный порядок*

NavigableSet<String> planetsReverse = planets.descendingSet();

System.**out**.println(planetsReverse);

*// получить два последних элемента*

NavigableSet<String> twoLast = planets.tailSet("Saturn", true);

System.**out**.println(twoLast);

*// получить элемент, который перед указанным*

String lower = planets.lower("Saturn");

System.**out**.println(lower);

*// получить элемент, который после указаного*

String higher = planets.higher("Saturn");

System.**out**.println(higher);

*// получить первый элемент и удалить его из множества*

String first = planets.pollFirst();

*// получить последний элемент и удалить его из множества*

String last = planets.pollLast();

**Queue**

Очереди предствляют собой упорядоченый список элементов. Элементы вставляются в конец очереди, а извлекаются из начала очереди. Обычно, но не обязательно, очереди работают по принципу **FIFO** — первым пришел, первым ушел.

Используйте очередь если вы хотите обрабатывать поток элементов в том же порядке в котором они поступают. Хорошо для списка заданий и обработки запросов.

**Queue**

В Java Collections API интерфейс Queue реализован в

* LinkedList - реализует принцип FIFO.
* PriorityQueue - хранит элементы либо в естественном порядке (natural order) либо в соответсвии с Comparator, переданым в конструктор.

Пример

Queue<String> planets = **new** LinkedList<>();

*// добавить элемент, в случаи проблемы - выбросить exception*

planets.**add**("Neptune");

*// добавить элемент, в случаи проблемы - вернуть false*

planets.offer("Venus");

*// получить последний элемент очереди, без удаления*

String top = planets.element();

System.**out**.println(top);

*// получить последний элемент очереди, с удалением*

String saturn = planets.**remove**();

System.**out**.println(saturn);

*// итерация элементов*

**for**(String planet : planets) {

System.**out**.println(planet);

}

**Deque**

Deque (Double Ended Queues) это двухсторонняя очередь, которая позволяет вставлять/удалять элементы как с конца очереди, так и с начала. Deque интерфейс расширяет Queue интерфейс.

В Java Collections API интерфейс Deque реализован в

* ArrayDeque - базируется на массиве и используется при реализации стэка (принцип **LIFO** – последним пришел первым ушел).
* LinkedList - базируется на двух-связном списке и используется при реализации очереди (принцип FIFO).

Пример

Deque<String> planets = **new** LinkedList<>();

planets.addLast("Mercury");

planets.offerLast("Earth");

planets.offerLast("Saturn");

planets.offerLast("Venus");

System.**out**.println(planets);

*// пустая очередь?*

boolean isEmpty = planets.isEmpty()

*// итерация по очереди и извлечение по одному элементу*

**while** (planets.peekFirst() != null) {

System.**out**.println("Первый элемент: " + planets.peekFirst());

planets.removeFirst();

System.**out**.println("Очередь: " + planets);

}

*// получить первый элемент очереди, без удаления*

String top = planets.peekFirst();

System.**out**.println(top);

*// получить первый элемент очереди, с удалением*

String saturn = planets.pollFirst();

System.**out**.println(saturn);

**Map**

В контейнерах Map (отображения, словарь, ассоциативный массив, карты) хранятся два объекта: ключ и связанное с ним значение. Map позволяет искать объекты по ключу.

И ключи, и значения являются объектами. Ключи могут быть уникальными, а значения могут дублироваться. Некоторые отображения допускают пустые ключи и пустые значения.

Интерфейс Map соотносит уникальные ключи со значениями. Ключ - это объект, который вы используете для последующего извлечения данных. Задавая ключ и значение, вы можете помещать значения в объект отображения. После того как это значение сохранено, вы можете получить его по ключу.

Основные методы - get() и put(), чтобы получить или поместить значения в отображение.

Интерфейс Map предоставляет три представления (view) хранящихся данных:

* множество всех ключей
* множество всех значений
* множество объектов **Entry**, содержащих в себе и ключ и значение

Отображения не поддерживают реализацию интерфейса Iterable, поэтому нельзя перебрать карту через цикл for в форме for-each. Перебрать можно отдельно **или** ключи **или** значения **или** объекти **Entry**.

- Интерфейс SortedMap расширяет интерфейс Map и гарантирует, что элементы размещаются в возрастающем порядке значений ключей.

- Интерфейс NavigableMap расширяет интерфейс SortedMap, которому дополнительно добавили методы для поиска ближайшего значения к заданному значению поиска.

- Интерфейс Map.**Entry** позволяет работать с элементом отображения, в частности используется при переборе элементов. Основные методы интерфейса Map.**Entry**

* equals(Object o) сравнение на равенство двух элементов
* getKey() получить ключ элемента
* getValue() получить значение элемента
* hashCode() получить хэш-код элемента
* setValue(V value) замена значения элемента на value

**HashMap**

HashMap использует хэш-таблицу для реализации  Map  интерфейса, что позволяет операциям get() и put() выполнятся за константное время даже для больших наборов. HashMap обеспечивает максимальную скорость выборки, а порядок хранения его элементов не очевиден.

HashMap подобен Hashtable с нескольками исключениямия:

* HashTable потокобезопасна, а HashMap нет
* HashTable не может содержать элементы null, тогда как HashMap может содержать один ключ null и любое количество значений null
* Итератор у HashMap, в отличие от перечислителя  HashTable, работает по принципу *fail-fast*(выдает исключение при любой несогласованности данных)

Пример

HashMap<String, Double> planets = **new** HashMap<>();

*// добавление элементов*

planets.put("Mercury", **new** Double(2439.7));

planets.put("Earth", **new** Double(6371));

planets.put("Saturn", **new** Double(58232));

planets.put("Neptune", **new** Double(24622));

planets.put("Venus", **new** Double(6051.8));

*// отображение множества, обратите внимание на порядок вывода*

System.**out**.println(planets);

*// получить значение по ключу*

Double radius = planets.**get**("Earth");

*// получение True если ключ есть в отображении, иначе False*

boolean isExist = planets.containsKey("Earth")

*// получение True если значение есть в отображении, иначе False*

boolean isExist = planets.containsValue("Earth")

*// пустое отображение?*

boolean isEmpty = planets.isEmpty()

*// удаление элемента по ключу*

planets.**remove**("Saturn");

*// список ключей*

Set keys = planets.keySet();

*// список значений*

Collection<Double> values = planets.values();

ArrayList<Double> arr = **new** ArrayList<>(planets.values());

*// множество элементов ввиде объектов интерфейса Map.Entry*

Set entries = planets.entrySet();

*// перебор элементов Entry отображения с помощью for*

**for** (Map.Entry me : planets.entrySet()) {

System.**out**.println("Key: "+me.getKey() + " & Value: " + me.getValue());

}

*// перебор элементов отображения с помощью while*

Set entries = planets.entrySet();

Iterator i = entries.iterator();

**while**(i.hasNext()) {

Map.Entry me = (Map.Entry)i.next();

System.**out**.print(me.getKey() + ": ");

System.**out**.println(me.getValue());

}

*// размер отображения*

**int** amount = planets.size()

*// удалить все элементы*

planets.clear()

*// конвертирование Map в ArrayList: список ключей*

List keyList = **new** ArrayList(planets.keySet());

*// конвертирование Map в ArrayList: список значений*

List valueList = **new** ArrayList(planets.valueSet());

*// конвертирование Map в ArrayList: список ключ-значения*

List entryList = **new** ArrayList(planets.entrySet());

*// упорядочивание Map по значениям*

List list = **new** ArrayList(planets.entrySet());

Collections.sort(list, **new** Comparator() {

@Override

**public** **int** **compare**(Entry e1, Entry e2) {

**return** e1.getValue().compareTo(e2.getValue());

}

});

**TreeMap**

TreeMap реализует Map интерфейс с помощью структуры  *красно-чёрного дерева*  и хранит ключи отсортированными по возрастанию (естественный порядок, natural order). Переопределить сортировку можно предоставив экземпляр класса Comparator, метод compare которого и будет использован для сортировки ключей. Обратите внимание, что все ключи добавленные в словарь должны реализовывать интерфейс Comparable (это необходимо для сортировки).

Для вставки, удаления и поиска элементов в Map лучше использовать HashMap. Если же нужно последовательно обходить карты в неком порядке то лучше использовать TreeMap. Иногда, в зависимости от размера коллекции, лучше добавить элементы в HashMap, а потом сконвертировать в TreeMap для упорядоченого обхода элементов.

Пример

TreeMap<String, Double> planets = **new** TreeMap<>();

*// добавление элементов*

planets.put("Mercury", **new** Double(2439.7));

planets.put("Earth", **new** Double(6371));

planets.put("Saturn", **new** Double(58232));

planets.put("Neptune", **new** Double(24622));

planets.put("Venus", **new** Double(6051.8));

*// отображение множества, обратите внимание на порядок вывода*

System.**out**.println(planets);

*// получить значение по ключу*

Double radius = planets.**get**("Earth");

*// получение True если ключ есть в отображении, иначе False*

boolean isExist = planets.containsKey("Earth")

*// получение True если значение есть в отображении, иначе False*

boolean isExist = planets.containsValue("Earth")

*// пустое отображение?*

boolean isEmpty = planets.isEmpty()

*// удаление элемента по ключу*

planets.**remove**("Saturn");

*// список ключей*

Set keys = planets.keySet();

*// список значений*

Collection<Double> values = planets.values();

ArrayList<Double> arr = **new** ArrayList<>(planets.values());

*// множество элементов в виде объектов интерфейса Map.Entry*

Set entries = planets.entrySet();

*// перебор элементов отображения с помощью for*

**for** (Map.Entry me : planets.entrySet()) {

System.**out**.println("Key: "+me.getKey() + " & Value: " + me.getValue());

}

*// перебор элементов отображения с помощью while*

Set entries = planets.entrySet();

Iterator i = entries.iterator();

**while**(i.hasNext()) {

Map.Entry me = (Map.Entry)i.next();

System.**out**.print(me.getKey() + ": ");

System.**out**.println(me.getValue());

}

*// размер отображения*

**int** amount = planets.size()

*// удалить все элементы*

planets.clear()

*// сортировка элементов по значению*

*// компаратор*

**public** **static** <K, V extends Comparable<V>> Map<K, V> **sortByValues**(final Map<K, V> map) {

Comparator<K> valueComparator = **new** Comparator<K>() {

**public** **int** **compare**(K k1, K k2) {

**int** compare = map.**get**(k1).compareTo(map.**get**(k2));

**if** (compare == 0)

**return** 1;

**else**

**return** compare;

}

};

Map<K, V> sortedByValues = **new** TreeMap<K, V>(valueComparator);

sortedByValues.putAll(map);

**return** sortedByValues;

}

Map sortedMap = sortByValues(planets);

**for** (Map.Entry me : sortedMap.entrySet()) {

System.**out**.println("Key: "+me.getKey() + " & Value: " + me.getValue());

}

**LinkedHashMap**

LinkedHashMap реализует связанный список элементов отображения и хранит ключи в порядке вставки. Также позволяет сортировать элементы в порядке последнего доступа. LinkedHashMap не обеспечивает скорость поиска как HashMap.

Пример

LinkedHashMap<String, Double> planets = **new** LinkedHashMap<>();

*// добавление элементов*

planets.put("Mercury", **new** Double(2439.7));

planets.put("Earth", **new** Double(6371));

planets.put("Saturn", **new** Double(58232));

planets.put("Neptune", **new** Double(24622));

planets.put("Venus", **new** Double(6051.8));

*// отображение множества, обратите внимание на порядок вывода*

System.**out**.println(planets);

*// получить значение по ключу*

Double radius = planets.**get**("Earth");

*// получение True если ключ есть в отображении, иначе False*

boolean isExist = planets.containsKey("Earth")

*// получение True если значение есть в отображении, иначе False*

boolean isExist = planets.containsValue("Earth")

*// перебор элементов отображения с помощью for*

**for** (Map.Entry me : planets.entrySet()) {

System.**out**.println("Key: "+me.getKey() + " & Value: " + me.getValue());

}

*// еще один вариант перебора элементов отображения с помощью for*

**for** (Map.Entry me : planets.entrySet()) {

String key = me.getKey();

Double **value** = me.getValue();

System.**out**.println("Key: " + key + " & Value: " + **value**);

}