Коллекции

Хорошая информация по коллекциям : <http://www.quizful.net/post/Java-Collections>

Информация по потокам ввода и вывода : <https://javarush.ru/quests/lectures/questcore.level08.lecture04>  
 <https://metanit.com/java/tutorial/6.1.php>

1)Основополагающим для классов коллекций в библиотеке Java является интерфейс Collection. В его состав входят два основных метода:

public interface Collection<E> {

boolean add(E element);

Iterator <E> iterator();

…

}

В дополнение к ним имеется еще несколько методов, обсуждаемых далее в этой главе. Метод add () добавляет элемент в коллекцию. Он возвращает логическое значение true, если добавление элемента в действительности изменило коллекцию, а если коллекция осталась без изменения — логическое значение false. Так, если попытаться добавить объект в коллекцию, где такой объект уже имеется, вызов метода add () не даст желаемого результата, поскольку коллекция не допускает дублирование объектов. А метод iterator () возвращает объект класса, реализующего интерфейс Iterator. Объект итератора можно выбрать для обращения ко всем элементам коллекции по очереди. Подробнее итераторы обсуждаются в следующем разделе.

2)Начиная с версии Java SE 8, для перебора элементов коллекции можно даже не организовывать цикл. Для этого достаточно вызвать метод forEachRemaining () с лямбда-выражением, где употребляется элемент коллекции. Это лямбда-выражение вызывается с каждым из элементов до тех пор, пока их больше не останется в коллекции:

iterator.forEachRemaining(element -> сделать что-нибудь с элементом element) ;

3) Порядок, в котором перебираются элементы, зависит от типа коллекции. Так, если осуществляется перебор элементов коллекции типа ArrayList, итератор начинает его с нулевого индекса, увеличивая последнее значение на каждом шаге итерации. Но если осуществляется перебор элементов коллекции типа Hash Set, то они получаются в совершенно случайном порядке. Можно быть уверенным лишь в том, что за время итерации будут перебраны все элементы коллекции, хотя нельзя сделать никаких предположений относительно порядка их следования. Обычно это не особенно важно, потому что порядок не имеет значения при таких вычислениях, как, например, подсчет суммы или количества совпадений.

4) В состав интерфейса Iterator входят три метода:

public interface Iterator<E> {

Е next();

boolean hasNext();

void remove();

}

Многократно вызывая метод next (), можно обратиться к каждому элементу коллекции по очереди. Но если будет достигнут конец коллекции, то метод next () сгенерирует исключение типа NoSuchElementException. Поэтому перед вызовом метода next () следует вызывать метод hasNext (). Этот метод возвращает логическое значение true, если у объекта итератора все еще имеются объекты, к которым можно обратиться.Метод remove () из интерфейса Iterator удаляет элемент, который был возвращен при последнем вызове метода next (). Во многих случаях это имеет смысл, поскольку нужно проанализировать элемент, прежде чем решаться на его удаление.

5) Но важнее то, что между вызовами методов next () и remove () существует определенная связь. В частности, вызывать метод remove () не разрешается, если перед ним не был вызван метод next (). Если же попытаться сделать это, будет сгенерировано исключение типа IllegalStateException.

6)Интерфейсы Collection и Iterator являются обобщенными, а следовательно, можно написать служебные методы для обращения к разнотипным коллекциям.

7) Разработчики библиотеки Java решили, что некоторые из этих служебных методов настолько полезны, что их следует сделать доступными из самой библиотеки. Таким образом, пользователи библиотеки избавлены от необходимости заново изобретать колесо. По существу, в интерфейсе Collection объявляется немало полезных методов, которые должны использоваться во всех реализующих его классах. К числу служебных методов относятся 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

8) В табл перечислены коллекции из библиотеки Java вместе с кратким описанием назначения каждого из их классов.

•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 расширяет 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

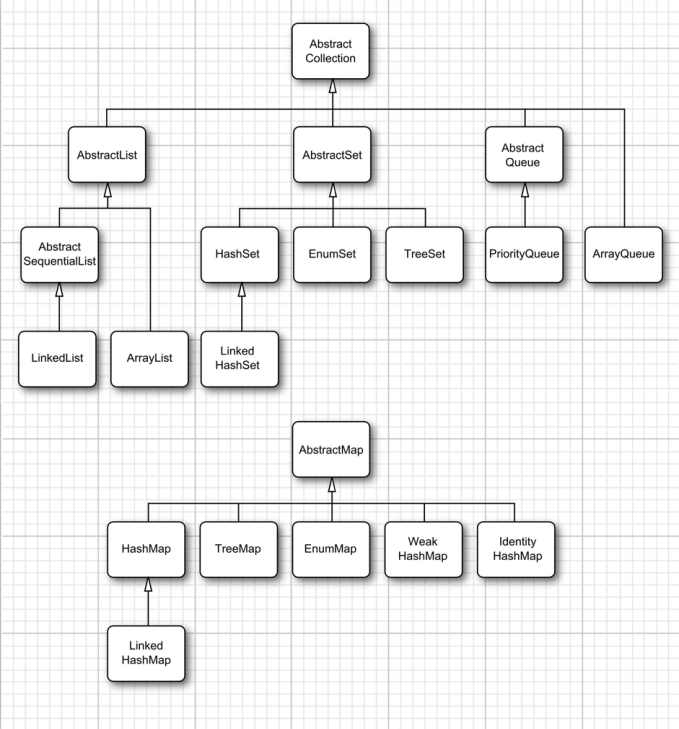
•TreeSet: набор отсортированных объектов в виде дерева

•LinkedHashSet: связанное хеш-множество

•PriorityQueue: очередь приоритетов

•HashMap: структура данных в виде словаря, в котором каждый объект имеет уникальный ключ и некоторое значение

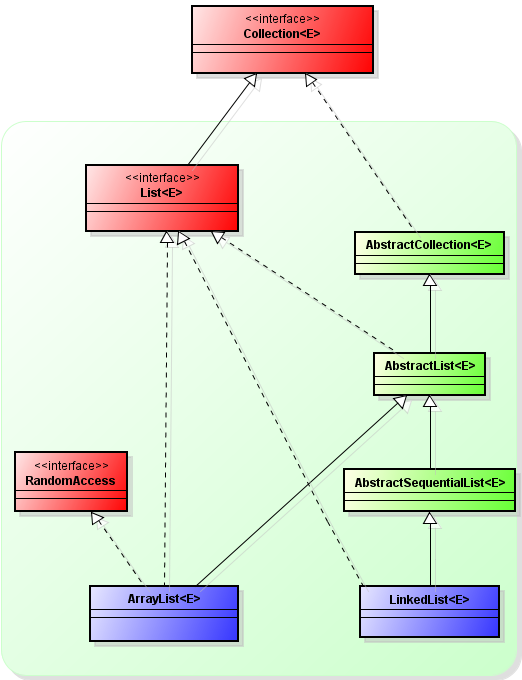
•TreeMap: структура данных в виде дерева, где каждый элемент имеет уникальный ключ и некоторое значение



9)**ArrayList**- пожалуй самая часто используемая коллекция. ArrayList инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов.  
Так как ArrayList использует массив, то  время доступа к элементу по индексу минимально (В отличии от LinkedList). При удалении произвольного элемента из списка, все элементы находящиеся «правее» смещаются на одну ячейку влево, при этом реальный размер массива (его емкость, capacity) не изменяется. Если при добавлении элемента, оказывается, что массив полностью заполнен, будет создан новый массив размером (n \* 3) / 2 + 1, в него будут помещены все элементы из старого массива + новый, добавляемый элемент.

**LinkedList**- Двусвязный список. Это структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и две ссылки на следующий и предыдущий узел списка. Доступ к произвольному элементу осуществляется за линейное время (но доступ к первому и последнему элементу списка всегда осуществляется за константное время — ссылки постоянно хранятся на первый и последний, так что добавление элемента в конец списка вовсе не значит, что придется перебирать весь список в поисках последнего элемента). В целом же, LinkedList в абсолютных величинах проигрывает ArrayList и по потребляемой памяти и по скорости выполнения операций.

Существуют два способа для обхода элементов: посредством итератора и произвольного доступа с помощью методов get() и set(). Второй способ не совсем подходит для связных списков, но применение методов get() и set() совершенно оправдано для массивов.

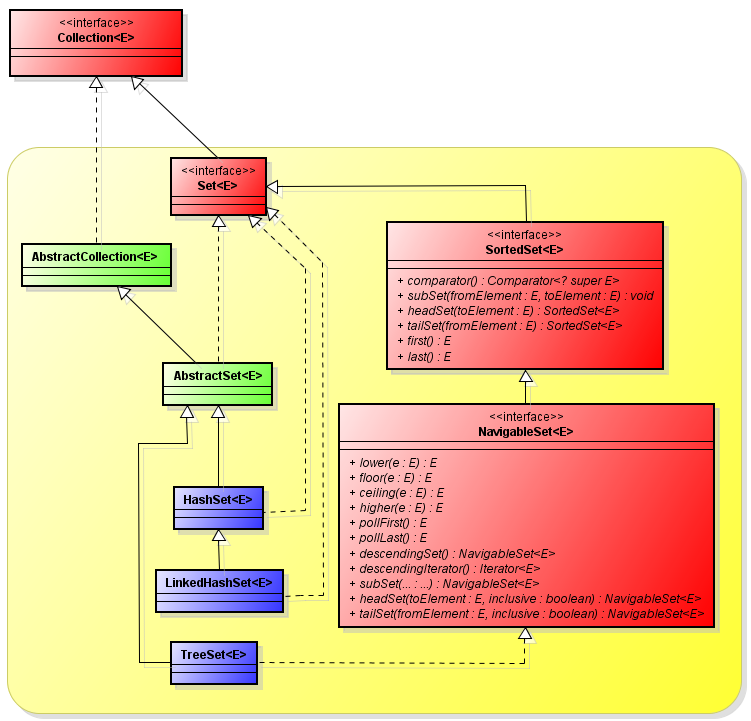


10) **HashSet**- коллекция, не позволяющая хранить одинаковые объекты(как и любой Set).  **HashSet**инкапсулирует в себе объект **HashMap**(то-есть использует для хранения хэш-таблицу).  
Как большинство читателей, вероятно, знают, хеш-таблица хранит информацию, используя так называемый механизм хеширования, в котором содержимое ключа используется для определения уникального значения, называемого хеш-кодом. Этот хеш-код затем применяется в качестве индекса, с которым ассоциируются данные, доступные по этому ключу. Преобразование ключа в хеш-код выполняется автоматически — вы никогда не увидите самого хеш-кода. Также ваш код не может напрямую индексировать хеш-таблицу. Выгода от хеширования состоит в том, что оно обеспечивает константное время выполнения методов add(), contains(), **remove()** и **size()** , даже для больших наборов.

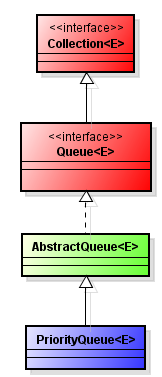
Если Вы хотите использовать **HashSet** для хранения объектов СВОИХ классов, то вы ДОЛЖНЫ переопределить методы hashCode() и equals(), иначе два логически-одинаковых объекта будут считаться разными, так как при добавлении элемента в коллекцию будет вызываться метод hashCode() класса Object (который скорее-всего вернет разный хэш-код для ваших объектов).  
Важно отметить, что класс HashSet не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не порождает сортированных наборов. Если вам нужны сортированные наборы, то лучшим выбором может быть другой тип коллекций, такой как класс TreeSet.

**LinkedHashSet**-  поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор. То есть, когда идет перебор объекта класса LinkedHashSet с применением итератора, элементы извлекаются в том порядке, в каком они были добавлены

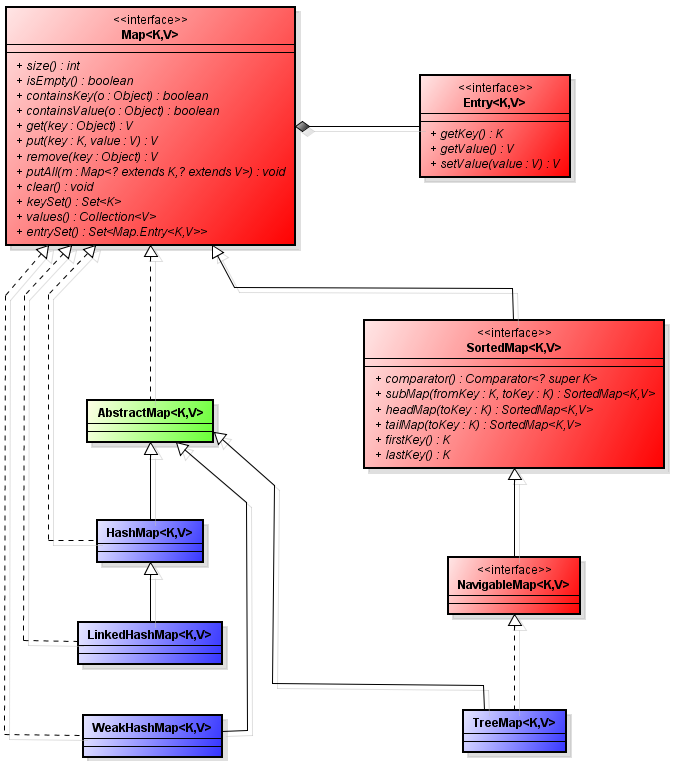
**TreeSet**- коллекция, которая хранит свои элементы в виде упорядоченного по значениям дерева. TreeSet инкапсулирует в себе TreeMap, который в свою очередь использует сбалансированное бинарное красно-черное дерево для хранения элементов. TreeSet хорош тем, что для операций add, remove и contains потребуется гарантированное время log(n).



12) **PriorityQueue**- единственная прямая реализация интерфейса **Queue** (не считая **LinkedList**, который больше является списком, чем очередью).  
Эта очередь упорядочивает элементы либо по их натуральному порядку (используя интерфейс **Comparable**), либо с помощью интерфейса **Comparator**, полученному в конструкторе.



13) 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 карты. После того как это значение сохранено, вы можете получить его по ключу.



**HashMap**— основан на хэш-таблицах, реализует интерфейс Map (что подразумевает хранение данных в виде пар ключ/значение). Ключи и значения могут быть любых типов, в том числе и null. Данная реализация не дает гарантий относительно порядка элементов с течением времени.

**TreeMap**- расширяет класс **AbstractMap**и реализует интерфейс **NavigatebleMap**. Он создает коллекцию, которая для хранения элементов применяет дерево. Объекты сохраняются в отсортированном порядке по возрастанию. Время доступа и извлечения элементов достаточно мало, что делает класс TreeMap блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена.

**LinkedHashMap**-  расширяет класс **HashMap**. Он создает связный список элементов в карте, расположенных в том порядке, в котором они вставлялись. Это позволяет организовать перебор карты в порядке вставки. То есть, когда происходит итерация по коллекционному представлению объекта класса LinkedHashMap, элементы будут возвращаться в том порядке, в котором они вставлялись. Вы также можете создать объект класса LinkedHashMap, возвращающий свои элементы в том порядке, в котором к ним в последний раз осуществлялся доступ.

**WeakHashMap**- коллекция, использующая слабые ссылки для ключей (а не значений). Слабая ссылка (англ. weak reference) — специфический вид ссылок на динамически создаваемые объекты в системах со сборкой мусора. Отличается от обычных ссылок тем, что не учитывается сборщиком мусора при выявлении объектов, подлежащих удалению. Ссылки, не являющиеся слабыми, также иногда именуют «сильными».

14) Если обращение к коллекции происходит из нескольких потоков исполнения, то нужно каким-то образом исключить ее непреднамеренное повреждение. Было бы, например, губительно, если бы в одном потоке исполнения была предпринята попытка ввести элемент в хеш-таблицу в тот момент, когда в другом потоке производилось бы повторное хеширование ее элементов. Вместо реализации потокобезопасных классов разработчики библиотеки коллекций воспользовались механизмом представлений, чтобы сделать потокобезопасными обычные коллекции. Например, статический метод synchronizedMap () из класса Collections может превратить любое отображение в объект типа Мар с синхронизированными методами доступа следующим образом:

Map <String,Employee> map = Collections.synchronizedMap(new HashMap<String,Employee>());