## Коллекции в Java 1.5

## 

Перейдем к коллекциям, которые появились в Java 1.5. Все коллекции унаследованы от одного интерфейса Collection из пакета java.util. Интерфейс **Collection**, в свою очередь, реализует интерфейс **Iterable**. Это означает, что все классы, унаследованные от Collection могут использоваться в цикле foreach. Например так:

List<String> stringList = Arrays.asList("1", "2", "4", "3");

for( String entry : stringList ){

System.out.println( entry );

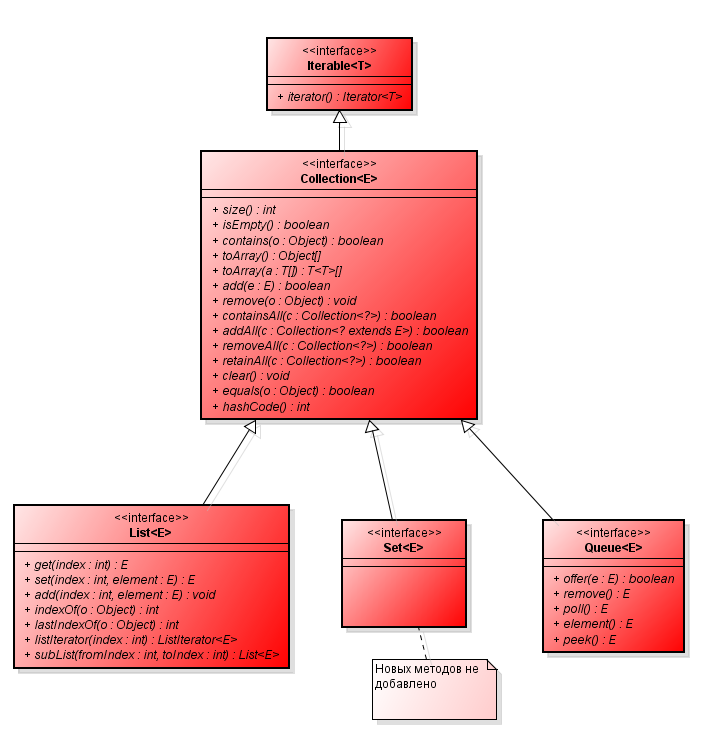
}

Учтите, что удаление элементов из коллекции в цикле foreach небезопасно. Для этих целей используйте итератор, например так:

for (Iterator<?> it = c.iterator(); it.hasNext(); )

if (!cond(it.next()))

it.remove();

Иерархия интерфейсов, унаследованных от **Collection** представлена на Рис.1. Как можно заметить, отсутствует интерфейс **Map** - он не унаследован от **Collection** и о нём разговор будет отдельный.

Стоит заметить, что в Java нет ни одного класса, который напрямую реализует интерфейс Collection. Это происходит через промежуточные интерфейсы List, Set и Queue.

## Общепринятые правила использования Collection

1. Всегда корректно описывайте методы ***equals()*** и ***hashCode()*** в объектах, которые будут добавлены в коллекцию. Например, для того, чтобы метод ***contains(Object o)*** вернул корректное значение, необходимо чтобы выполнялось следующее условие: (o==null ? e==null : o.equals(e)). Метод ***hashCode()*** гарантирует, что два объекта с разными значениями hashCode() не могут быть равны. Также метод ***hashCode()*** может использоваться коллекцией для наиболее эффективного размещения объектов в памяти. Это правило не касается стандартных типов вроде String, Integer, Float и т.д. так как в этих классах уже реализованы оба метода.

2. Каждый класс, реализующий интерфейс **Collection**, имеет как минимум два конструктора: конструктор по умолчанию и конструктор, принимающий коллекцию (объект Collection). Это правило ложиться на плечи разработчика так как у интерфейсов не может быть конструкторов. Пример:

List<String> stringList = Arrays.asList( "2", "1", "2", "4", "3");

Set<String> stringSet = new HashSet<String>( stringList );

for( String entry : stringSet ){

System.out.print( entry );

}

// Output: 3214

3. Класс реализует интерфейс Collection и **если метод не поддерживается коллекцией, то необходимо генерировать исключение UnsupportedOperationException**. Это является необязательным правилом. Например, если коллекция не может быть модифицирована, то при вызове метода add() может произойти исключение UnsupportedOperationException, а может и не произойти если это не критично.

## Основные преимущества использования коллекций

* **Снижает количество написанного кода.** Предоставляя полезную структуру данных и алгоритмов, коллекции освобождают от низкоуровневого программирования и позволяют сосредоточится на других частях программы.
* **Повышает быстродействие и качество программы.** Коллекции обеспечивают высокую производительность, высокое качество реализации полезных структур данных и алгоритмов. Различные реализации каждого интерфейса являются взаимозаменяемыми, так что программы могут быть легко настроены на переключение разных реализаций коллекций.
* **Позволяет взаимодействовать не связанным между собой API.** Коллекции общеупотребляемые, поэтому могут легко использоваться в качестве аргументов функций и их возвращаемых значений.
* **Уменьшает количество усилий, приложенных для изучения новых API.** Многие API получают коллекции на вход и отдают другие коллекции на выходе. В недавнем прошлом каждое API имело свою собственную реализацию своих коллекций. Из-за этого приходилось изучать структуры данных, свойственные каждому API.
* **Уменьшает количество усилий, приложенных для разработки новых API.** Это обратная сторона предыдущего преимущества. Разработчикам нет необходимости создавать собственные коллекции, достаточно использовать имеющиеся или реализовать свои на основе существующих интерфейсов.
* **Способствует повторному использованию программного обеспечения.** Новые структуры данных, унаследованные от существующих реализаций коллекций или интерфейсов коллекций, легко могут быть использованы где-угодно.

## Базовые интерфейсы

В библиотеке коллекций Java существует два базовых интерфейса, реализации которых и представляют совокупность всех классов коллекций:

1. **Collection**- коллекция содержит набор объектов (элементов). Здесь определены основные методы для манипуляции с данными, такие как вставка (add, addAll), удаление (*remove, removeAll, clear*), поиск (*contains*)  
2. **Map**-  описывает коллекцию, состоящую из пар "ключ — значение". У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения (тар). Такую коллекцию часто называют еще словарем (dictionary) или ассоциативным массивом (associative array). Никак НЕ относится к интерфейсу Collection и является самостоятельным.

Хотя фреймворк называется **Java Collections Framework**, интерфейс **map**и его реализации **входят**в фреймворк тоже !  
Интерфейсы Collection и Map являются базовыми, но они не есть единственными. Их расширяют другие интерфейсы, добавляющие дополнительный функционал. О них мы ещё поговорим.

Как видно с диаграммы, интерфейс **Collection**не является базовым (какая интрига :D). Интерфейс **Collection**расширяет интерфейс **Iterable**, у которого есть только один метод iterator(). Это значит что любая коллекция, которая есть наследником **Iterable**должна возвращать итератор.

**Итератор**(<http://ru.wikipedia.org/wiki/%D0%98%D1%82%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80>) -   объект, абстрагирующийся за единым интерфейсом доступ к элементам коллекции. Итератор это паттерн позволяющий получить доступ к элементам любой коллекции без вникания в суть ее реализации.

Идем дальше. Как видим с рисунка, интерфейс **Collection**расширяют интерфейсы **List, Set**и **Queue**. Давайте рассмотрим, зачем нужен каждый.  
    1. **List**- Представляет собой упорядоченную коллекцию, в которой допустимы дублирующие значения. Иногда их называют последовательностями (sequence ). Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.   
    2. **Set**- описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Это соответствует математическому понятию множества (set).  
    3. **Queue**- очередь. Сразу запоминаем как правильно произносится: Queue - КЬЮ. Это коллекция, предназначенная для хранения элементов в порядке, нужном для их обработки. В дополнение к базовым операциям интерфейса Collection, очередь предоставляет дополнительные операции вставки, получения и контроля.

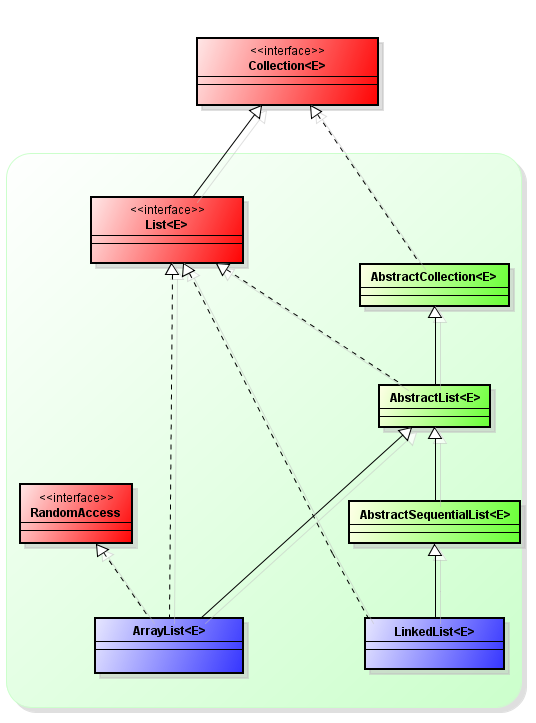
В этой статье речь пойдет о, пожалуй, наиболее часто используемых коллекциях List, а именно о таких классах как AbstractList, ArrayList, LinkedList.

List - упорядоченная коллекция иногда называемая списком или последовательностью. Список может содержать повторяющиеся элементы. Разработчик имеет абсолютный контроль над тем, в каком месте списка будет вставлен каждый элемент. Также пользователь может получить доступ к элементам списка по индексу. В дополнение к стандартным операциям интерфейса Collection, List содержит следующие:

* **Доступ по позиции**;
* **Поиск**;
* **Специальный итератор ListIterator**;
* **Диапазон элементов (sublist)**.

## Реализации интерфейса List

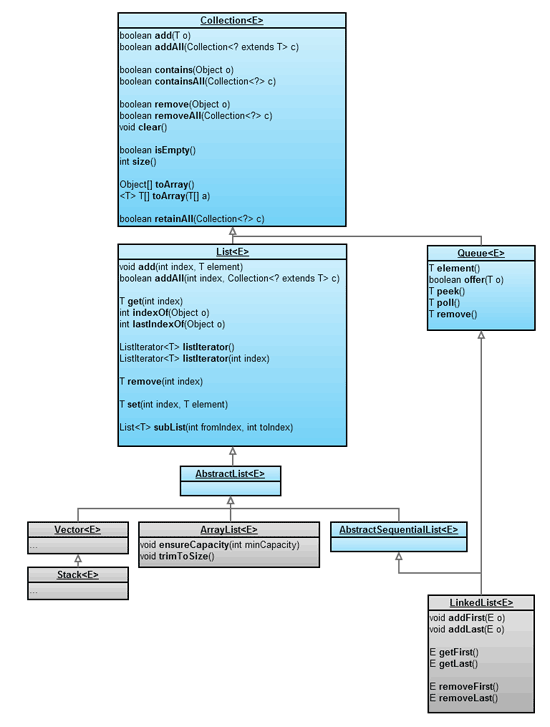
Сразу смотрим на иерархию классов.



*Красным здесь выделены интерфейсы, зеленым - абстрактные классы, а синим готовые реализации. Сразу хочу заметить что здесь не вся иерархия, а только основная её часть.*

**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 («связки») 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.

[](http://www.seostella.com/images/post/java/collections/collection-list.png)  
  
Рис 1. Иерархия List во фреймворке Collections

Интерфейсу List также принадлежит два устаревших класса: Vector и Stack.

В отличие от Set, List может содержать повторяющиеся элементы. Также может содержать null-элементы.

Некоторые реализации списков имеют ограничения на элементы, которые они могут содержать. Например, в некоторых реализациях запрещено использовать null в качестве элемента для вставки (на ArrayList и LinkedList это правило не распространяется).

Два объекта типа List одинаковы (т.е., метод equals() возвращает true) только в том случае, когда они содержат одинаковые элементы в одном и том же порядке.

ArrayList - наиболее широко используемая реализация List. ArrayList обладает наибольшей производительностью в плане доступа к случайному элементу в массиве.

LinkedList - еще одна реализация интерфейса List. В отличие от ArrayList, LinkedList обладает большей скоростью вставки элемента в произвольное место в списке, однако доступ к элементу прямо пропорциональный его позиции относительно начала последовательности.

### **Базовые интерфейсы**

В официальной документации они все перечислены, но я не буду пока приводить его полностью, напишу пока самые важные (на мой взгляд конечно). Основная идея при рассмотрении этих интерфейсов должна быть такая — весьма умные люди разработали список методов, которые крайне важны для определенных типов коллекций — списков, множеств, очередей и прочая. Список имеет свои особенности, множество — свои, очередь — свои. Набор методов для списка и для множества будет различаться, т.к. эти типы коллекций (список и множество) имеют некоторые важные отличия. Рассматривайте их как специализированные инструменты — например, для закручивания шурупов нужен шуруповоерт, для бетонных стен — перформатор, для сверления лунки — ледобур. Заметьте, что они все имеют «одну природу», но каждый имеет некоторую специализацию:

**java.util.Collection** — основной интерфейс, который описывает базовые методы, которыми должна обладать любая коллекция. Т.е. если какой-то класс претендует на звание КОЛЛЕКЦИЯ — он должен реализовать те методы, которые описаны в этом интерфейсе. Проводя аналогию с нашим набором сверлильных инструментов — интерфейс **java.util.Collection** их общий родитель — у него есть возможнсть сверлить. Советую зайти на сайт с документацией и честно просмотреть все его методы. Возможно, что Java версии 8 (и выше) покажется вам сложноватой, поэтому для начала советую зайти на документацию по Java версии 7. [java.util.Collection](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html). Большая часть методов говорит сама за себя, так что почитайте.

**java.util.List** — интерфейс для операций с коллекцией, которая является списком. Список обладает следующими важными признаками:

Список может включать одинаковые элементы

Элементы в списке хранятся в том порядке, в котором они туда помещались. Самопроизвольных перемещений элементов в нем не происходит — только с вашего ведома. Например, вы можете добавить элемент на какую-то позицию и тогда произойдет сдвиг других элементов.

Можно получить доступ к любому элементу по его порядковому номеру/индексу внутри списка

Т.е. если вам требуется, чтобы коллекция обладала такими свойствами — выбирайте класс, который реализует интерфейс **java.util.List**

**java.util.Set** — интерфейс для хранения множества. В отличии от **java.util.List** этот интерфейс как раз не может иметь одинаковые элементы (смотрим методы **equals** и **hashCode** в статье [Решения на основе классов](http://java-course.ru/begin/simpleclasses/)) и порядок хранения элементов в множестве может меняться при добавлении/удалении/изменении элемента. Может возникнуть вопрос, зачем такая коллекция нужна — это удобно в случае, когда вы создаете набор уникальных элементов из какой-то группы элементов.

**java.util.SortedSet** — это наследник интерфейса **java.util.Set** и его дополнительным функционалом является автоматическое выстраивание элементов внутри множества по порядку. Как этот порядок настаивается, мы поговорим позже.

**java.util.Queue** — интерфейс предлагает работать с коллекцией как с очередью, т.е. коллекция имеет метод для добавления элементов в один конец и метод для получения элемента с другого конца — т.е. настоящая очередь по принципу FIFO — First In First Out — если первым пришел, то первым и уйдешь. Для широкого круга задач такая конструкция работы с коллекцией бывает достаточно удобной структурой.

**java.util.Map** — очень удобная конструкция, которая хранит данные не в виде списка значений, а в виде пары ключ-значение. Это очень востребованная форма, в которой вы получаете доступ к значению в коллекции по ключу. Например, доступ к данным пользователя на сайте может быть осуществлен по логину (по email например). Самих данных может быть достаточно много, но для поиска можно использовать очень короткую строку-ключ.

И еще раз скажу самое важное — коллекция позволяет вам работать с группой объектов и специализация коллекции определяется требованиями к самим данным и к тем операциям, которые нужно использовать при работе с данными.

### **Простой пример использования коллекций**

Предлагаю посмотреть пример (демонстрацию) использования основных методов интерфейса **java.util.Collection**. Сначала просто напишу код примера и после этого прокомментирую его

package edu.javacourse.collection;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Iterator;

public class ExampleCollection

{

public static void main(String[] args) {

// Создаем коллекции для демонстрации

Collection col1 = createFirstCollection();

Collection col2 = createSecondCollection();

// Демонстрация прохода по коллекции

System.out.println("============= Проход по коллекции");

for(Object o : col1) {

System.out.println("Item:" + o);

}

System.out.println();

// Демонстрация прохода по коллекции через итератор

System.out.println("============= Проход по коллекции через итератор");

for (Iterator it = col1.iterator(); it.hasNext(); ) {

String s = (String)it.next();

System.out.println("Item:" + s);

}

System.out.println();

// Демонстрации групповых операций

System.out.println();

System.out.println("============= Групповые операции");

// Можно проверить сожержаться ли ВСЕ элементы col2 в col

if(col1.containsAll(col2)) {

System.out.println("Коллекция col содержит все от col2");

}

System.out.println("============= Добавление всех элементов в col1 из col2");

// Можно добавить элементы из col2 в col1

col1.addAll(col2);

for(Object o : col1) {

System.out.println("Item:" + o);

}

System.out.println("============= Удаление всех элементов col2, которые есть в col1");

// Можно удалить ВСЕ элементы col2, которые есть в col1

col1.removeAll(col2);

for(Object o : col1) {

System.out.println("Item:" + o);

}

// Пересоздаем коллекции для дальнейшей демонстрации

col1 = createFirstCollection();

col2 = createSecondCollection();

System.out.println("============= Удаление элементов из col1, которых нет в col2");

col1.retainAll(col2);

for(Object o : col1) {

System.out.println("Item:" + o);

}

System.out.println("============= Очистка коллекции - не будет элементов");

col1.clear();

for(Object o : col1) {

System.out.println("Item:" + o);

}

System.out.println();

// Удаление элемента коллекции

// Снова создаем коллекцию для демонстрации

col1 = createFirstCollection();

// Удаляем один элемент

col1.remove("1");

System.out.println("============= Удаляем элемент '1' - его не будет в списке");

for(Object o : col1) {

System.out.println("Item:" + o);

}

// Удаление коллекции через итератор

// Снова создаем коллекцию для демонстрации

col1 = createFirstCollection();

System.out.println("============= Удаление через итератор");

while(!col1.isEmpty()) {

Iterator it = col1.iterator();

Object o = it.next();

System.out.println("Удаляем:" + o);

// Удаляем элемент

it.remove();

}

}

// Первая коллекция для примера

private static Collection createFirstCollection() {

// Создать коллекцию на основе стандартного класса ArrayList

Collection col = new ArrayList();

// Добавление в коллекцию

col.add("1");

col.add("2");

col.add("3");

col.add("4");

col.add("5");

col.add("6");

col.add("7");

return col;

}

// Вторая коллекция для примера

private static Collection createSecondCollection() {

// Создать коллекцию на основе стандартного класса ArrayList

Collection col2 = new ArrayList();

col2.add("1");

col2.add("2");

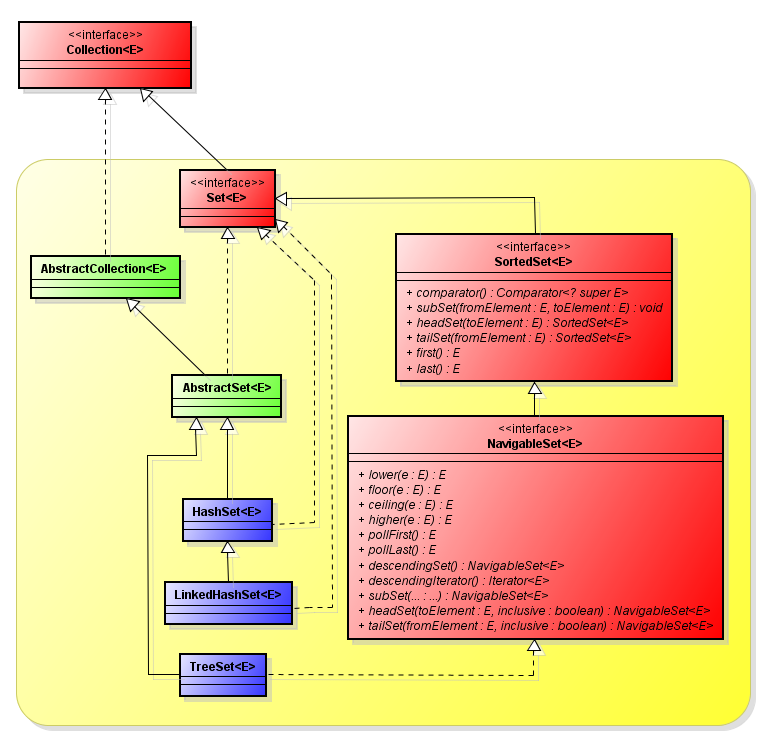
col2.add("3");

return col2;

}

}

Реализации интерфейса Set



**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.

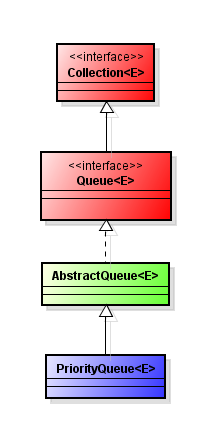
*Если Вы хотите использовать***HashSet***для хранения объектов СВОИХ классов, то вы ДОЛЖНЫ переопределить методы hashCode() и equals()*, иначе два логически-одинаковых объекта будут считаться разными, так как при добавлении элемента в коллекцию будет вызываться метод hashCode() класса Object (который 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).

Реализации интерфейса Queue

Здесь я привел очень упрощенную иерархию.



**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.

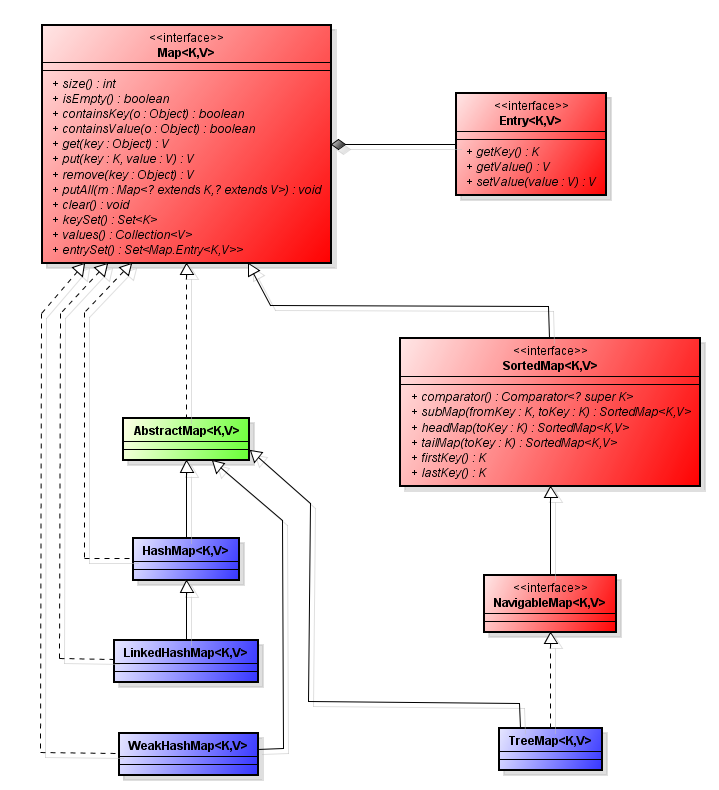
Реализации интерфейса Map

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. Интерфейс **Map**— это обобщенный интерфейс, объявленный так, как показано ниже.

interface Мар<К, V>

Здесь К указывает тип ключей, а V — тип хранимых значений.

Иерархия классов очень похожа на иерархию Set'а:



**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. Хорошая статья <http://habrahabr.ru/post/128017/>

**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.   
Рекомендую так же прочитать <http://habrahabr.ru/post/129037/>

**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.  
Моя статья про TreeMap <http://www.quizful.net/post/Java-TreeMap>

**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».  
<http://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%B0%D0%B1%D0%B0%D1%8F_%D1%81%D1%81%D1%8B%D0%BB%D0%BA%D0%B0>