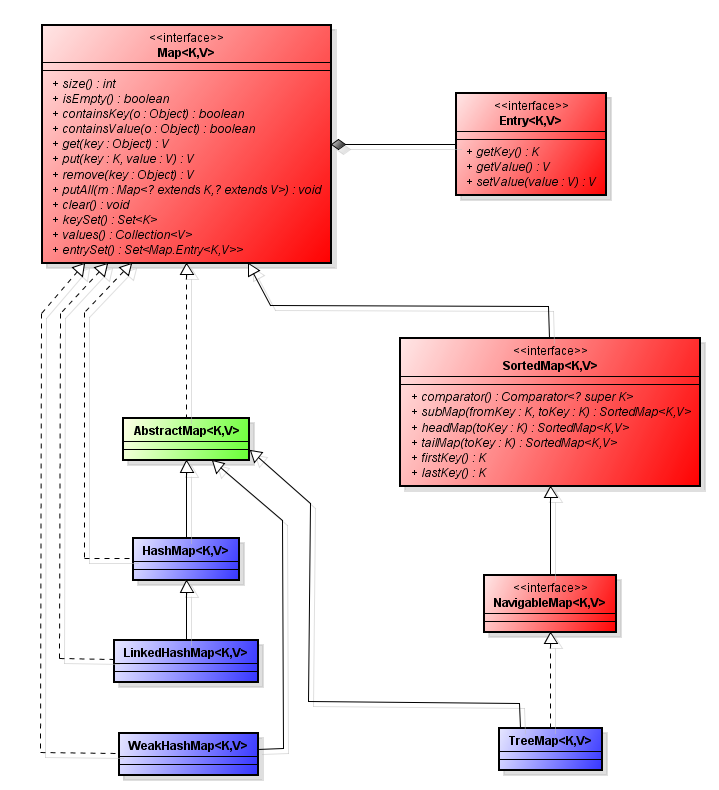
Реализации интерфейса Map

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.

**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.

**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. 

**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.

**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».

|  |  |
| --- | --- |
| **Методы интерфейса Map** | |
| Int size() | Возвращает колличество записей в карте |
| Boolean isEmpty() | Возвращает true если карта пуста |
| V get(Object key) | Вернет значение элемента карты по заданному ключу. Вернёт null если ключ равен null или если значение по ключу равно null |
| Boolean containsKey(Object key) | Вернёт true если карта содержит запись с заданным ключом |
| V put(K key, V value) | Добавит значение по указанному ключу. Если такой ключ уже есть – перезапишет ему значение. Вернет старое значение или null |
| Void putAll(Map<? Extends K, ? extends V>m) | Скопирует все записи из m. Если ключи уже есть, то перезапишет им значения. Если m пуста – выбросит исключение. |
| V remove(Object key) | Удалит запись по ключу если таковой имеется. Вернёт удаляемое значение по ключу |
| Void clear() | Очистит карту |
| Boolean containsValue(Object value) | Вернёт true если есть хоть одна запись с задаваемым значением |
| Set<K> keySet() | Вернёт SET ключей карты |
| Collection<V> values() | Вернёт список значений карты типом Collection |
| Set<Map.Entry<K,V>>entrySet() | Вернет множество записей карты типом SET |
|  |  |
|  |  |

**Создание объекта**

Map<String, String> hashmap = new HashMap<String, String>();

Footprint{Objects=2, References=20, Primitives=[int x 3, float]}  
Object size: 120 bytes  
  
Новоявленный объект hashmap, содержит ряд свойств:

* **table** — Массив типа **Entry[]**, который является хранилищем ссылок на списки (цепочки) значений;
* **loadFactor** — Коэффициент загрузки. Значение по умолчанию 0.75 является хорошим компромиссом между временем доступа и объемом хранимых данных;
* **threshold** — Предельное количество элементов, при достижении которого, размер хэш-таблицы увеличивается вдвое. Рассчитывается по формуле **(capacity \* loadFactor)**;
* **size** — Количество элементов HashMap-а;

В конструкторе, выполняется проверка валидности переданных параметров и установка значений в соответствующие свойства класса. Словом, ничего необычного.

*// Инициализация хранилища в конструкторе*

*// capacity - по умолчанию имеет значение 16*

table = new Entry[capacity];

[[https://habrastorage.org/storage1/f5998744/554eeee7/3d597647/2f404c04.png](http://habrastorage.org/storage1/c7aef3f0/b31342f1/a6194b46/1c4a2384.png)](http://habrastorage.org/storage1/c7aef3f0/b31342f1/a6194b46/1c4a2384.png)  
Вы можете указать свои емкость и коэффициент загрузки, используя конструкторы **HashMap(capacity)** и **HashMap(capacity, loadFactor)**. Максимальная емкость, которую вы сможете установить, равна половине максимального значения **int** (1073741824).

**Добавление элементов**

hashmap.put("0", "zero");

Footprint{Objects=7, References=25, Primitives=[int x 10, char x 5, float]}  
Object size: 232 bytes  
  
При добавлении элемента, последовательность шагов следующая:

1. Сначала ключ проверяется на равенство null. Если это проверка вернула true, будет вызван метод **putForNullKey(value)** (вариант с добавлением null-ключа рассмотрим чуть [позже](https://habr.com/ru/post/128017/#putForNullKey)).
2. Далее генерируется хэш на основе ключа. Для генерации используется метод **hash(hashCode)**, в который передается **key.hashCode()**.
3. static int hash(int h)
4. {
5. h ^= (h >>> 20) ^ (h >>> 12);
6. return h ^ (h >>> 7) ^ (h >>> 4);
7. }

Комментарий из исходников объясняет, каких результатов стоит ожидать — *метод* ***hash(key)*** *гарантирует что полученные хэш-коды, будут иметь только ограниченное количество коллизий (примерно 8, при дефолтном значении коэффициента загрузки).*  
  
В моем случае, для ключа со значением **''0''** метод **hashCode()** вернул значение 48, в итоге:

h ^ (h >>> 20) ^ (h >>> 12) = 48

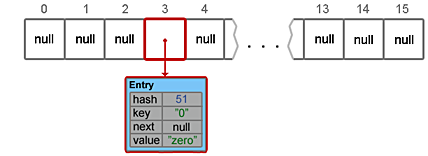
h ^ (h >>> 7) ^ (h >>> 4) = 51

1. С помощью метода **indexFor(hash, tableLength)**, определяется позиция в массиве, куда будет помещен элемент.
2. static int indexFor(int h, int length)
3. {
4. return h & (length - 1);
5. }

При значении хэша **51** и размере таблице **16**, мы получаем индекс в массиве:

h & (length - 1) = 3

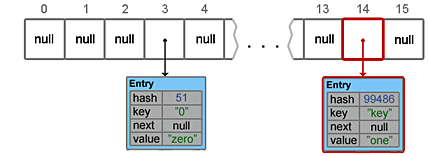
1. Теперь, зная индекс в массиве, мы получаем список (цепочку) элементов, привязанных к этой ячейке. Хэш и ключ нового элемента поочередно сравниваются с хэшами и ключами элементов из списка и, при совпадении этих параметров, значение элемента перезаписывается.
2. if (e.hash == hash && (e.key == key || key.equals(e.key)))
3. {
4. V oldValue = e.value;
5. e.value = value;
7. return oldValue;
8. }
9. Если же предыдущий шаг не выявил совпадений, будет вызван метод **addEntry(hash, key, value, index)** для добавления нового элемента.
10. void addEntry(int hash, K key, V value, int index)
11. {
12. Entry<K, V> e = table[index];
13. table[index] = new Entry<K, V>(hash, key, value, e);
14. ...
15. }

[[](http://habrastorage.org/storage1/517a990b/09fe6798/cf03fb7f/88d6f1dd.png)](http://habrastorage.org/storage1/517a990b/09fe6798/cf03fb7f/88d6f1dd.png)

Для того чтобы продемонстрировать, как заполняется HashMap, добавим еще несколько элементов.

hashmap.put("key", "one");

Footprint{Objects=12, References=30, Primitives=[int x 17, char x 11, float]}  
Object size: 352 bytes

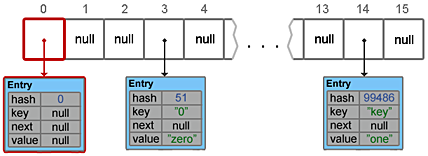
1. Пропускается, ключ не равен null
2. **''key''.hashCode()** = 106079
3. h ^ (h >>> 20) ^ (h >>> 12) = 106054
4. h ^ (h >>> 7) ^ (h >>> 4) = 99486
5. Определение позиции в массиве
6. h & (length - 1) = 14
7. Подобные элементы не найдены
8. Добавление элемента  
     
   [[](http://habrastorage.org/storage1/912edc07/a8ee5362/c52fe95f/1fea7b76.png)](http://habrastorage.org/storage1/912edc07/a8ee5362/c52fe95f/1fea7b76.png)

hashmap.put(null, null);

Footprint{Objects=13, References=33, Primitives=[int x 18, char x 11, float]}  
Object size: 376 bytes  
  
Как было сказано выше, если при добавлении элемента в качестве ключа был передан null, действия будут отличаться. Будет вызван метод **putForNullKey(value)**, внутри которого нет вызова методов **hash()** и **indexFor()** (потому как все элементы с null-ключами всегда помещаются в **table[0]**), но есть такие действия:

1. Все элементы цепочки, привязанные к **table[0]**, поочередно просматриваются в поисках элемента с ключом null. Если такой элемент в цепочке существует, его значение перезаписывается.
2. Если элемент с ключом null не был найден, будет вызван уже знакомый метод **addEntry()**.

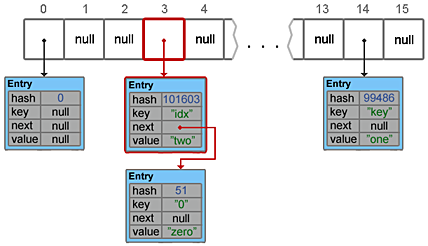
addEntry(0, null, value, 0);

[[](http://habrastorage.org/storage1/a1d8de60/ae91ecf5/b79fe023/33b07c6a.png)](http://habrastorage.org/storage1/a1d8de60/ae91ecf5/b79fe023/33b07c6a.png)

hashmap.put("idx", "two");

Footprint{Objects=18, References=38, Primitives=[int x 25, char x 17, float]}  
Object size: 496 bytes  
  
Теперь рассмотрим случай, когда при добавлении элемента возникает коллизия.

1. Пропускается, ключ не равен null
2. **''idx''.hashCode()** = 104125
3. h ^ (h >>> 20) ^ (h >>> 12) = 104100
4. h ^ (h >>> 7) ^ (h >>> 4) = 101603
5. Определение позиции в массиве
6. h & (length - 1) = 3
7. Подобные элементы не найдены
8. Добавление элемента
9. *// В table[3] уже хранится цепочка состоящая из элемента ["0", "zero"]*
10. Entry<K, V> e = table[index];
11. *// Новый элемент добавляется в начало цепочки*
12. table[index] = new Entry<K, V>(hash, key, value, e);

[[](http://habrastorage.org/storage1/9cd24c82/9dd96244/dc1dd8aa/72923c80.png)](http://habrastorage.org/storage1/9cd24c82/9dd96244/dc1dd8aa/72923c80.png)

**Resize и Transfer**

Когда массив **table[]** заполняется до предельного значения, его размер увеличивается вдвое и происходит перераспределение элементов. Как вы сами можете убедиться, ничего сложного в методах **resize(capacity)** и **transfer(newTable)** нет.

void resize(int newCapacity)

{

if (table.length == MAXIMUM\_CAPACITY)

{

threshold = Integer.MAX\_VALUE;

return;

}

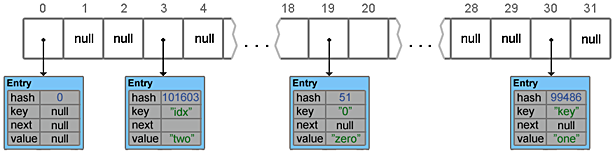
Entry[] newTable = new Entry[newCapacity];

transfer(newTable);

table = newTable;

threshold = (int)(newCapacity \* loadFactor);

}

Метод **transfer()** перебирает все элементы текущего хранилища, пересчитывает их индексы (с учетом нового размера) и перераспределяет элементы по новому массиву.  
  
Если в исходный **hashmap** добавить, скажем, еще 15 элементов, то в результате размер будет увеличен и распределение элементов изменится.  
  
[[](http://habrastorage.org/storage1/e96dd8c1/e5029883/e46934bc/8d1fca4b.png)](http://habrastorage.org/storage1/e96dd8c1/e5029883/e46934bc/8d1fca4b.png)

**Удаление элементов**

У HashMap есть такая же «проблема» как и у ArrayList — при удалении элементов размер массива **table[]** не уменьшается. И если в ArrayList предусмотрен метод **trimToSize()**, то в HashMap таких методов нет (хотя, как сказал один мой коллега — "*А может оно и не надо?*").  
  
Небольшой тест, для демонстрации того что написано выше. Исходный объект занимает 496 байт. Добавим, например, 150 элементов.  
  
Footprint{Objects=768, References=1028, Primitives=[int x 1075, char x 2201, float]}  
Object size: 21064 bytes  
  
Теперь удалим те же 150 элементов, и снова замерим.  
  
Footprint{Objects=18, References=278, Primitives=[int x 25, char x 17, float]}  
Object size: 1456 bytes  
  
Как видно, размер даже близко не вернулся к исходному. Если есть желание/потребность исправить ситуацию, можно, например, воспользоваться конструктором **HashMap(Map)**.

hashmap = new HashMap<String, String>(hashmap);

Footprint{Objects=18, References=38, Primitives=[int x 25, char x 17, float]}  
Object size: 496 bytes

**Итераторы**

HashMap имеет встроенные итераторы, такие, что вы можете получить список всех ключей **keySet()**, всех значений **values()** или же все пары ключ/значение **entrySet()**. Ниже представлены некоторые варианты для перебора элементов:

*// 1.*

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

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

*// 2.*

for (String key: hashmap.keySet())

System.out.println(hashmap.get(key));

*// 3.*

Iterator<Map.Entry<String, String>> itr = hashmap.entrySet().iterator();

while (itr.hasNext())

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

Стоит помнить, что если в ходе работы итератора HashMap был изменен (без использования собственным методов итератора), то результат перебора элементов будет непредсказуемым.

**Итоги**

— Добавление элемента выполняется за время O(1), потому как новые элементы вставляются в начало цепочки;  
— Операции получения и удаления элемента могут выполняться за время O(1), если хэш-функция равномерно распределяет элементы и отсутствуют коллизии. Среднее же время работы будет Θ(1 + α), где α — коэффициент загрузки. В самом худшем случае, время выполнения может составить Θ(n) (все элементы в одной цепочке);  
— Ключи и значения могут быть любых типов, в том числе и null. Для хранения примитивных типов используются соответствующие классы-оберки;  
— Не синхронизирован.

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. Такие коллекции облегчают поиск элемента, если нам известен ключ - уникальный идентификатор объекта.

Следует отметить, что в отличие от других интерфейсов, которые представляют коллекции, интерфейс Map НЕ расширяет интерфейс Collection.

Среди методов интерфейса Map можно выделить следующие:

* void clear(): очищает коллекцию
* boolean containsKey(Object k): возвращает true, если коллекция содержит ключ k
* boolean containsValue(Object v): возвращает true, если коллекция содержит значение v
* Set<Map.Entry<K, V>> entrySet(): возвращает набор элементов коллекции. Все элементы представляют объект Map.Entry
* boolean equals(Object obj): возвращает true, если коллекция идентична коллекции, передаваемой через параметр obj
* boolean isEmpty: возвращает true, если коллекция пуста
* V get(Object k): возвращает значение объекта, ключ которого равен k. Если такого элемента не окажется, то возвращается значение null
* V getOrDefault(Object k, V defaultValue): возвращает значение объекта, ключ которого равен k. Если такого элемента не окажется, то возвращается значение defaultVlue
* V put(K k, V v): помещает в коллекцию новый объект с ключом k и значением v. Если в коллекции уже есть объект с подобным ключом, то он перезаписывается. После добавления возвращает предыдущее значение для ключа k, если он уже был в коллекции. Если же ключа еще не было в коллекции, то возвращается значение null
* V putIfAbsent(K k, V v): помещает в коллекцию новый объект с ключом k и значением v, если в коллекции еще нет элемента с подобным ключом.
* Set<K> keySet(): возвращает набор всех ключей отображения
* Collection<V> values(): возвращает набор всех значений отображения
* void putAll(Map<? extends K, ? extends V> map): добавляет в коллекцию все объекты из отображения map
* V remove(Object k): удаляет объект с ключом k
* int size(): возвращает количество элементов коллекции

Чтобы положить объект в коллекцию, используется метод put, а чтобы получить по ключу - метод get. Реализация интерфейса Map также позволяет получить наборы как ключей, так и значений. А метод entrySet() возвращает набор всех элементов в виде объектов **Map.Entry<K, V>**.

Обобщенный интерфейс Map.Entry<K, V> представляет объект с ключом типа K и значением типа V и определяет следующие методы:

* boolean equals(Object obj): возвращает true, если объект obj, представляющий интерфейс Map.Entry, идентичен текущему
* K getKey(): возвращает ключ объекта отображения
* V getValue(): возвращает значение объекта отображения
* Set<K> keySet(): возвращает набор всех ключей отображения
* V setValue(V v): устанавливает для текущего объекта значение v
* int hashCode(): возвращает хеш-код данного объекта

При переборе объектов отображения мы будем оперировать этими методами для работы с ключами и значениями объектов.

**Классы отображений. HashMap**

Базовым классом для всех отображений является абстрактный класс **AbstractMap**, который реализует большую часть методов интерфейса Map. Наиболее распространенным классом отображений является **HashMap**, который реализует интерфейс Map и наследуется от класса AbstractMap.

Пример использования класса:

import java.util.\*;

public class Program{

    public static void main(String[] args) {

        Map<Integer, String> states = new HashMap<Integer, String>();

       states.put(1, "Germany");

       states.put(2, "Spain");

       states.put(4, "France");

       states.put(3, "Italy");

       // получим объект по ключу 2

       String first = states.get(2);

       System.out.println(first);

       // получим весь набор ключей

       Set<Integer> keys = states.keySet();

       // получить набор всех значений

       Collection<String> values = states.values();

       //заменить элемент

       states.replace(1, "Poland");

       // удаление элемента по ключу 2

       states.remove(2);

       // перебор элементов

       for(Map.Entry<Integer, String> item : states.entrySet()){

           System.out.printf("Key: %d  Value: %s \n", item.getKey(), item.getValue());

       }

       Map<String, Person> people = new HashMap<String, Person>();

       people.put("1240i54", new Person("Tom"));

       people.put("1564i55", new Person("Bill"));

       people.put("4540i56", new Person("Nick"));

       for(Map.Entry<String, Person> item : people.entrySet()){

           System.out.printf("Key: %s  Value: %s \n", item.getKey(), item.getValue().getName());

       }

    }

}

class Person{

    private String name;

    public Person(String value){

        name=value;

    }

    String getName(){return name;}

}

Чтобы добавить или заменить элемент, используется метод put, либо replace, а чтобы получить его значение по ключу - метод get. С помощью других методов интерфейса Map также производятся другие манипуляции над элементами: перебор, получение ключей, значений, удаление.